Universiteit Antwerpen

Parallel Programming with MPI and Fault Tolerance

Proefschrift voorgelegd op 15 September 2009 tot het behalen van de graad van Doctor in de Wetenschappen, bij de faculteit Wetenschappen, aan de Universiteit Antwerpen.

Promotoren:
Prof. Dr. Jan Broeckhove
Prof. Dr. Frans Arickx

David Dewolfs

CoMP Research Group Computational Modelling and Programming

Contents

Preface

Despite the fact that Moore's Law [1] has been steadily upheld for the past few decades [2], the processing needs for solving various scientific problems are continually on the rise [3, 4, 5]. Both in terms of required memory, CPU or other resources, regular single-processor computers are unable to keep pace with the increase in computational demand. As computing problems become more complex, scientists are continually involved in a never ending struggle to expand the available capacity for high-speed, high volume computing.

Over the past decades, many solutions have appeared, only to disappear again. They range from the specialized vector machines that began the current age of supercomputing [6], over the still popular Beowulf Clusters [7], to recent products like IBM's Blue Gene parallel computer [8]. Some scientists even foster visions of world-spanning computing "grids" [9]. Generally, we can state that parallel computing solutions currently occupy the lead position in the "race of supercomputing".

With relatively recent developments, like MPI [11], actually becoming mainstream for tackling large-scale problems, and solutions like Seti@home [10] leading the way towards ever growing amounts of computing power, one could be tempted to believe that a solution to the problem of satisfying scientists' resource hunger is finally on the horizon. Especially the aforementioned MPI, a standardized specification for message-passing based parallel programming, offers promises – both by (1) having many implementations, on many platforms, and (2) by having many libraries available using it. Of course, such conclusions would be too far-reaching.

Mainly, the issue is not just with the "big problems" anymore. Scientists in smaller research groups are facing very real resource shortages. On the one hand, these groups do not have the means to acquire "big iron" hardware – they have to deal with assorted commodity-scale, heterogeneous hardware – potentially spread over multiple locations. On the other hand, they often have to deal with legacy software – the creators of which did not have a parallel computing solution in mind when they originally wrote it. Both of these issues contribute to circumstances which are far from ideal when trying to find a parallel solution. Especially when using MPI. Solving some of the issues thus becomes an interesting research topic.

Part I

Overview

Overview

Message passing is presently one of the most popular approaches to parallel programming, and a perfect fit for many computational problems. Arguably, MPI has become the de-facto standard for writing parallel software using the message-passing paradigm (we elaborate on MPI and its nature over the course of the next two chapters and defer any further discussion of its specifications until then).

Thanks to the MPI standard, parallelizing code in a relatively straightforward manner, without losing portability and choice of platform, proves attainable. This leaves us with the question of hardware platform – and this is where things turn out less idyllic...

Present-day "big-iron" parallel hardware is relatively dependable, and offers great performance – at a premium price. A price which proves unaffordable for research groups of our size. There is, fortunately, an alternative: unused CPUs are available all over campus if one only looks for them.

Older machines that are no longer being used can be turned into clusters. Computers from the student labs can be used for scientific purposes outside academic hours. One could even "swap in/out" such resources based on availability (i.e. even during lab periods, detection mechanisms could be used to sense whether a computer is in use or not; if it isn't, use it for scientific computations; if it is, immediately remove it from the "ad-hoc" parallel platform and make it available to whoever needs it). And cooperations between research groups can open up access to additional resources, albeit probably spread over multiple locations (buildings, campuses, maybe even countries).

All of these solutions offer a more democratically priced entry into parallel computing, but they all come with one disadvantages: unreliability. Older hardware is more prone to failure. Lab computers – subject to student "abuse" as well; certainly if you use the "ad-hoc computing" approach, as resources are dynamically swapped into – and out of – your parallel platform. And geographically spread resources are as vulnerable as the network infrastructure connecting them. And here lies the rub: MPI – previously our seemingly perfect parallel programming approach – does not provide for fault-tolerance.

But there are more problems ahead: integrating legacy software into an object-oriented framework is not as straightforward as it seems. The issues one faces are:

- Many smaller research groups around the world are facing resource shortages. Parallel computing is the only realistic solution to their needs.
- Moreover, if these groups want to exchange code, working with an accepted standard for parallel programming is a must. The viable option here is MPI.
- Due to a limited investment in hardware, all of these groups have a very real need for fault-tolerance. MPI, however, does not provide this.
- Many research groups still possess a lot of Fortran code.
- For various reasons, it is useful (or even necessary) to integrate this code with newer, mostly C++ code (e.g. to introducing object oriented design and behavior for better maintainability, for cooperation with C++ frameworks, or simply because of the higher availability of C++ programmers vs. Fortran).
- Integrating modern Fortran code (especially Fortran 99) into C++ turns out to be less than straightforward

In order to address these issues, we need to investigate the following three questions:

1. With regard to MPI implementations: what are the current options w.r.t. fault-tolerance and MPI, what would be the best approach for the particular case of small research groups working with low-budget hardware and what yet remains to be done in this area to make fault-tolerant use of MPI valid.
2. With regard to MPI software: what – if any – changes are needed to make it fault tolerant, given an MPI implementation that supports it. What would a generic "cookbook" for creating fault-tolerant MPI-based software look like.
3. With regard to Fortan legacy code: what needs to be done to make easy integration of that code with C++ code work in a user-friendly manner.

These are the question that we will try and formulate an answer for over the course of the next chapters.

In the first part of the text, we will investigate the answers to the first of our three questions: what about MPI implementations and fault-tolerance? For clarity, we first offer a general introduction on distributed and parallel computing, afterwards zooming in on MPI and its specific qualities. Eventually we focus on MPI implementations and fault tolerance. We will highlight the different options and our reasons for selecting a particular one. And most importantly present our body of original work to solve the problems involved.

In the second part of our text, we investigate the remaining to questions. We do this using a quantum nuclear scattering problem as a case study. The conclusions are, however, generally applicable to any similar project involving legacy Fortran code that needs to be integrated with C++ and made fault-tolerant.

Specifically, we shall discuss how we integrate legacy code with C++ code. This involves our second major body of original work, which we will show serves to generally integrate any Fortan (90) based code into a C++ framework.

Second, we shall investigate the changes needed to make the code fault-tolerant. From these steps, we shall derive a series of cookbook-like instructions that can help programmers make any piece of MPI-based code fault-tolerant.

Finally, in the third part of this text, we shall offer our conclusions.

Part II

MPI and FT-MPI

CHAPTER 1

Parallel Computing

The majority of modern high performance computing solutions use some degree of parallelization or distribution to maximize performance.

A popular analogy to illustrate the advantages of parallel processing (over traditional "monolithic" solutions), is that of the horse and the cart. If one wants to drive faster, it is easier to span a second horse before your cart than to grow a new breed that is twice as fast and powerful as the current one.

But the task of parallelizing scientific computations is not quite as simple as this popular analogy would suggest. One of the major problems has always been the task of writing elegant and manageable software for parallel platforms [13].

Solutions that make efficient use of the resources and capabilities of parallel hardware are not trivial. One cannot help but notice that there exists a major gap between:

1. the highly automated compiler technologies available for serial computing platforms, and,
2. the coding practices for modern parallel computing platforms.

The former can produce relatively generic and resource-efficient software, as well as offering portability over a great range of different platforms. More often than not, the latter still has to be hand-optimized for a specific parallel architecture computing problem. All of these specific optimizations are highly dependent upon the parallel platform's particular combination of technologies (SMP/shared memory vs. distributed memory approaches, caching architectures, communication bandwidth and latency etc.), as well as upon the specific properties of the computational problem.

Auto-parallelizing compilers exist and are rapidly improving, but they are still not up-to-par with solutions requiring developers to explicitly define parallelism [14]. All sorts of libraries have also been developed [58, 15, 16, 17, 18], mostly by the

same vendors that created the parallel hardware platforms they were written to work with. Due to a lack of standards however, these libraries were mostly mutually incompatible [19]. This leads to all kinds of practical problems when trying to port a program from one platform to the other, or when merging code originally written for two different libraries.

In other words: writing software for parallel platforms stretches beyond the singular act of designing a parallel or distributed algorithm. As parallel programming developed, many attempts were made to provide tools that make parallel programming easier, more generic and more efficient. One of the most popular early libraries was PVM [58], mostly aimed at clusters of commodity hardware. The most popular approach to programming any kind of parallel hardware today is probably MPI[11]. Both were designed for efficiency, functionality and support for heterogeneous parallel architectures.

MPI adds to this the aim to have multiple parallel implementations with full source code portability. This is done in order to support an maximize performance on the broadest possible range of parallel platforms. These attributes make MPI into a very attractive platform for high end computing at any scale, from high end solutions like the IBM Blue Gene to the more down to earth low-end, heterogeneous and geographically distributed cluster approaches.

In order for us to be able to discuss the characteristics of MPI further on in this work, we first need to lay down a basic framework for describing and categorizing the attributes of a parallel platform. We start out by identifying and specifying three basic properties, common to each parallel programming system: **memory model**, **problem decomposition** and **programming model**. Each of these will be discussed in the following subsections. The section will be closed with a general overview of some other basic issues thats must be taken care of when writing a parallel program.

1.1 Memory models

The memory model for a parallel platform specifies how the parallel processors access their data. Any memory model will put certain restrictions on what can be done with regard to implementing a parallel program on top of it. The memory model used in a parallel program will often be closely related to the underlying hardware. A software layer might hide most of the details of the memory model to the average programmer though, as he might have little use for such low-level knowledge. The following paragraphs specify two basic memory models. In the end, almost any memory model in use can be typified as either one – or a combination – of these two.

1.1.1 Shared memory architecture

A shared-memory architecture provides the programmer with an underlying data access model, composed of multiple processors which are connected to a global memory spaces. This space is shared by each of the individual processors, i.e.: each of the

processors will, at the same moment in time, find the same data item at the same address in the common memory space. Careful locking and synchronization mechanisms have to be employed to avoid data corruption (through racing conditions and similar problems). The shared-memory architecture is an example of an approach to parallelism known as *control* parallelism in which "(parallelism) ... is not determined by data independence, but is explicitly specified by the programmer." [27]

Figure 1.1 shows a schematic example of a shared memory architecture with 4 processors reading from and writing to a single memory space.

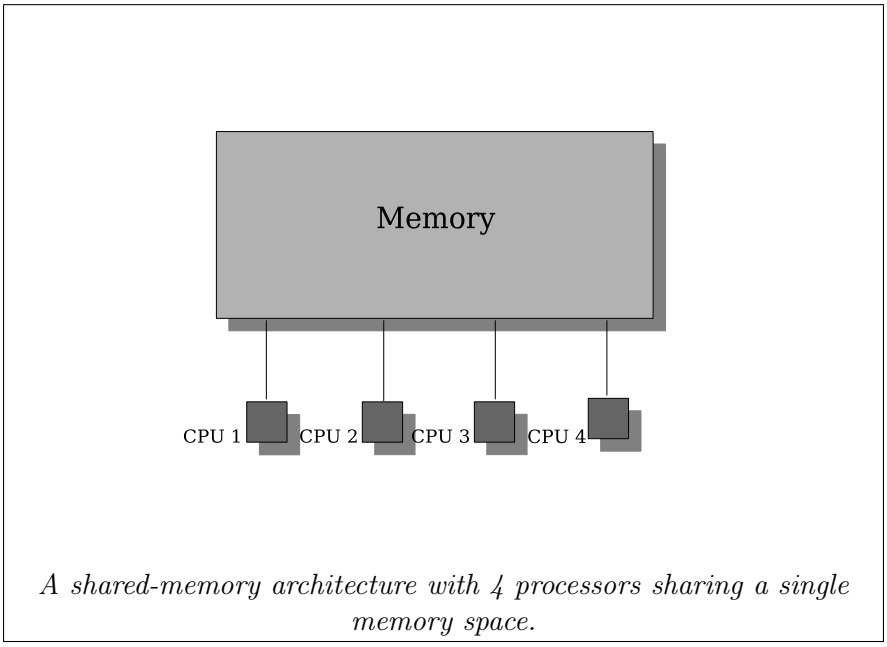


Figure 1.1: Shared-memory architecture

Examples of this memory architecture can be found with the early parallel vector machines, as well as with the modern-day Symmetric Multiprocessor (SMP) architectures. The latter might gather a limited number of processors, from two up to a few tens, sharing a single (logical as well as physical) memory space. However: caching issues create problems with any form of this design, and the severity of these problems rises as the amount of processors increases. A number of processors higher than the suggested few tens produces a range of practical problems that are currently hard to overcome.

1.1.2 Distributed memory architecture

Systems with a distributed memory architecture divide the available processing power into individual nodes, each of which has its own processor and memory space that can only be accessed by the local processor. Data is shared between the nodes by means of a communication network. If a processor needs to access some data in the local memory space of another processor, the data must be copied over from

the source processor to a communication buffer, communicated to the destination processor and written to its memory space of the destination processor.

Figure 1.2 shows a schematic example of a distributed memory architecture with 4 processors, each having its own separated memory. Each processor can only access its own memory space.

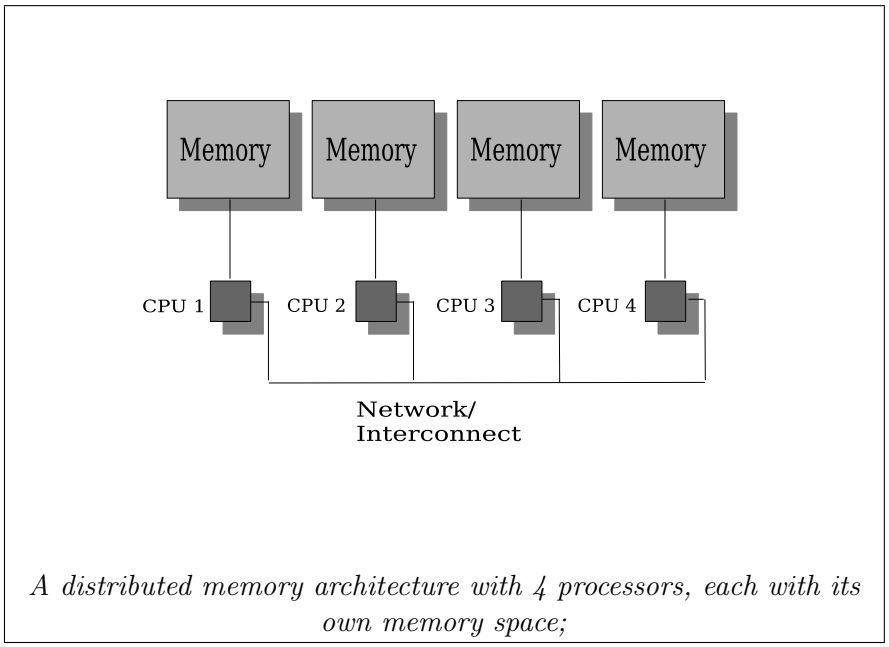


Figure 1.2: Distributed memory architecture

The use of this memory architecture is well exemplified in the popular "Beowulf" clusters [7]. These systems gather multiple "PC-like" machines into a single system for parallel computing. The individual nodes are normally composed of off-the-shelf (and thus relatively cheap) commodity hardware running some variant of the free and open-source Linux operating system. Specialized hardware for these clusters has been developed (e.g. fast interconnection networks like Myrinet) to overcome some of the performance limitations of the regular commodity parts.

Of course, hybrid varieties of these architectures are common, e.g. most recent cluster systems are composed of SMP-based nodes with multiple memory-sharing local processors per node.

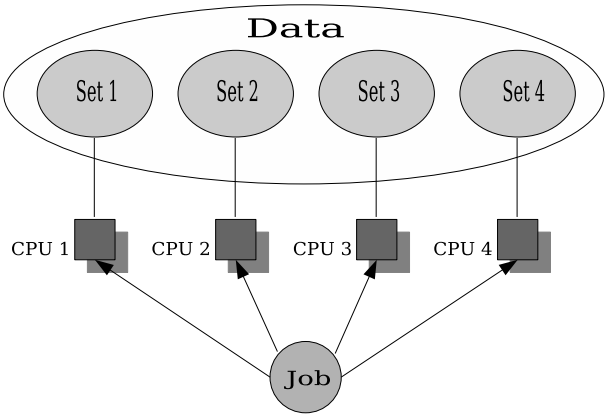
1.2 Problem decomposition

Any solution to a computational problem will eventually be composed of two parts: at one end the program, at the other the data which it must process. If any computational problem is to be parallelized, some part of it will have to be decomposed

into smaller fractions that can be distributed over multiple processors.

1.2.1 Domain decomposition

The decomposition of the data part of the problem is referred to as domain decomposition, or *data parallelism*. Using this approach, the data set to be processed is divided into multiple parts and divided among each of the processors. The program itself looks very much like a sequential program, as the parallelism comes purely from the approach to handling the data. Figure 1.3 shows an example of domain decomposition of a serial computation running on 4 physical processors with 4 data subsets.



A domain decomposition with a single monolithic task running on 4 physical processors, each working on a single data subset.

Figure 1.3: Domain decomposition

This partitioning of the data is the easiest to automate. For example: High Performance Fortran (HPF) [23] and OpenMP [24, 25] provide the programmer with so-called compiler "hints" to help the compiler with automatically parallelizing code. The code otherwise looks just as if it were written for a regular serial processor. However, not all computational problems are equally suited to this kind of treatment, and cannot easily, if at all, be distributed using this scheme [20]. Moreover, this is not necessarily the most *efficient* method, especially in cases where the data assigned to different processors might require greatly differing amounts of time to process. Speedup for the whole of the computation is limited by the slowest data set.

1.2.2 Functional decomposition

The approach of applying decomposition to the program itself is called functional decomposition, or *task parallelism*. In this case, the algorithm is divided into many

smaller subtasks, which might (or might not) be functionally related to – or dependent upon – each other. These tasks are then assigned to processors as they become available. Processors that finish quicker than others are assigned more tasks (as soon as possible, limited by issues like data dependency etc.) to minimize processor idle time. Figure 1.4 shows an example of functional decomposition with 4 subtasks running on 4 physical processors. In this particular case, the data pool remains undivided and can be accessed by all subtasks.

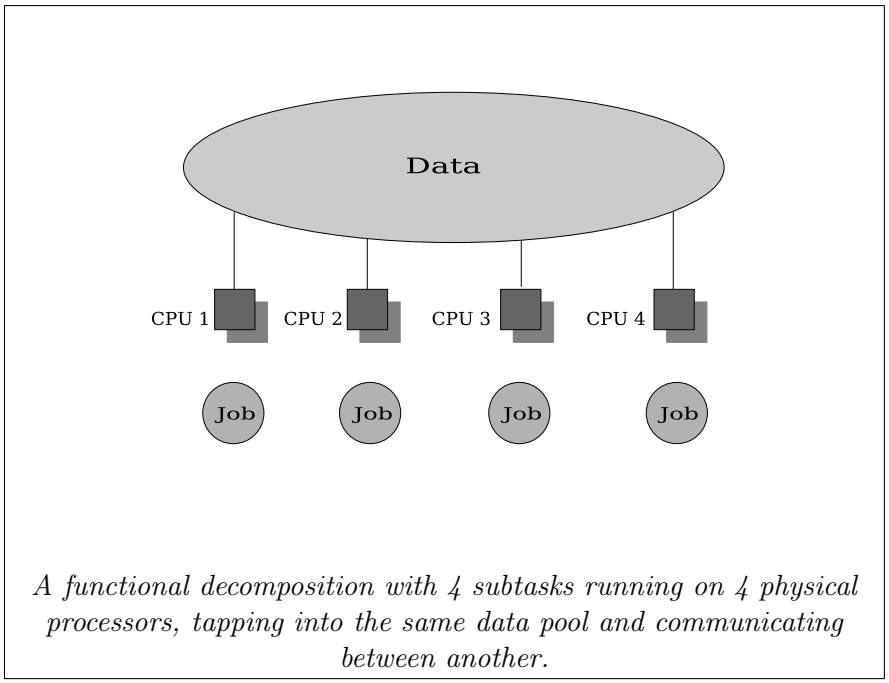


Figure 1.4: Functional decomposition

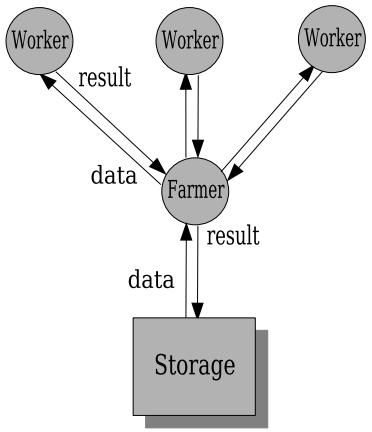
The Farmer/Worker (FW) model is a popular implementation of this approach. Figure 1.5 shows an example of the farmer/worker model with one farmer task reading data from the complete pool and delivering it in batches to 4 worker subtasks.

1.3 Programming models

Traditionally, there have been two main paradigms for writing parallel software: directives-based and message-based.

1.3.1 The directives-based model

Under a directives-based programming model, code is made parallel by adding compiler directives to the program, either as extensions of the programming language or as a special case of commentary code. These directives inform the compiler on how



Farmer/Worker model with 1 farmer and 3 worker subtasks. Data is read by the farmer and delivered to the workers for computation. Results are returned to the farmer for output to a result set.

Figure 1.5: Farmer/Worker model

to parallelize the code: the *low-level* management of parallelization (computation, communication, scheduling, ...) is therefore turned into an automated process.

This paradigm is most popular when combined with shared-memory architectures: these greatly simplify the writing of the specialized compilers [21]. It is also the default approach used when parallelizing problems through domain decomposition, though its possible applications extend beyond this particular approach to problem decomposition. Figure 1.6 shows an example of directives-based parallel code using high-performance Fortran (HPF).

1.3.2 The message-passing model

The message-passing programming model puts a far greater amount of control and responsibility in the hands of the programmer. Under this paradigm, a parallel computation is explicitly treated as the result of an interaction between multiple concurrent processes. Each of these has its own local data space. These processes cannot directly access any of the other's memory space. Data is shared between these processes by means of messages. These are passed between processes by calling send and receive routines at the originator and the recipient processes respectively. One of the defining aspects of message passing is the fact that *each message always requires an explicit call by all processes involved.* It might also be useful to remark here that *a process does not necessarily equate to a processor*, though this will often be the case. Figure shows an example of message passing parallel code using the Parallel Virtual Machine (PVM).

```
PROGRAM jacobi
      REAL  a (100,100),  old_a (100,100)

    !HPF$ PROCESSORS  p (4,4)
    !HPF$ ALIGN a (:,:) WITH old_a (:,:)
    !HPF$ DISTRIBUTE old_a (BLOCK, BLOCK) ONTO p

      ...  set boundary values of A as required
      a (2:99, 2:99) = 0.0      ! initialise interior of A to 0
      old_a = 0.0               !     "      all of OLD_A to 0

      DO WHILE ( ANY (a - old_a > 1.0E-07 * a) )
        old_a = a
        a (2:99, 2:99) = 0.25 * (a (1:98, 2:99) + a (3:100, 2:99)  &
                                 + a (2:99, 1:98) + a (2:99, 3:100))
       ENDDO
     END PROGRAM
```

Example of directives based parallel code using High Performance Fortran (HPF). Lines with !HPF$ are annotations with compiler hints on how to parallelize.

Figure 1.6: The directives-based programming model

The message-passing model offers advantages to the writer of parallel software:

- It is fairly *universal and general*, allowing it to be used on a broad selection of parallel hardware platforms.
- Its *expressiveness* is adequate for implementing solutions to a wide range of computational problems.
- As every process has its own isolated memory space, it gets complete control over memory references. This offers considerable *advantages when debugging* the program – no trivial task when it comes to parallel software.
- The control over data-locality also offers advantages when it comes to the efficiency of local caching, which is *possibly beneficial to the performance* of the parallel program, even on a platform that uses a shared-memory architecture at the hardware level [22].

A popular variant on the message passing model is constituted by the systems that support remote memory operations (also referred to as *active messaging* [26]). This approach transfers data between processes by directly copying it from the memory space of the originating to that of the receiving process. The major difference with message passing schemes here is that only one call is needed to transfer the data. This will be either a put-operation by the source process (to send the data to the memory space of the destination), or conversely, a get-operation by the destination (to retrieve the data from the source).

hello :

```
#include "pvm3.h"

main() {
      int cc, tid, msgtag;
      char buf[100];

      printf("i'm t%x\n", pvm_mytid()); //print my ID

      //Spawn a new process called "hello_other"
      cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

      if (cc == 1) {
            msgtag = 1;
            pvm_recv(tid, msgtag); //redeive data from hello_other
            pvm_upkstr(buf);       //unpack the message
            printf("from t%x: %s\n", tid, buf);
      } else printf("can't start hello_other\n");

   pvm_exit();
}
```

hello_other :

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent(); //get parent ID

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);  //initiate message send to hello
   pvm_pkstr(buf);                //pack message
   pvm_send(ptid, msgtag);        //send message

   pvm_exit();
}
```

Message passing parallel code using the Parallel Virtual Machine (PVM).

Figure 1.7: The message passing programming model

1.4 Further issues

Total execution time of a parallel program is made up of 3 components:

- *Computation time* is the total time spent on actual computations. Ideally, if a computation took T time to run on one processor, and N processors were

available, a computation would be done in T/N time, if all processors spent their time in computation. Which is, of course, not the case.

- *Idle time* is the time spent by a processor waiting, e.g. for data from another processor, IO, or until another process has reached a certain state etc.. No useful work is done during this time, and time spent in this fashion by the processor should thus be minimized.

- *Communication time* is the time it takes for a process to complete the sequences of sending and receiving data. Communication time is measured in terms of latency and bandwidth, *latency* being the time it takes to actually set up the communications channel and get the first bit to its destination, and bandwidth being the measure of how many bits are actually transported per time unit.

Computation time must be divided among the available processors as efficiently as possible, to reduce idle time. This *load balancing,* is a necessity to ensure efficient use of the available processors. This is theoretically easy when all processes in the program are equal or at least very similar. Large variations in duration and unpredictability of processing will greatly complicate this issue.

Minimizing communication is a major goal in any parallel computation. It is a function of both the underlying hardware and the implementation. Fast networks and switches enhance communication times in any cluster-like system, as do efficient data transfer routines in a shared-memory system. A good implementation however is paramount. Strategically overlapping interprocess communication with computation wherever possible is the most plausible approach. This will reduce CPU idle time by a far greater amount than a faster communication. It can be accomplished for example, by using either non-blocking communications, advance data transfers or data-unspecific computations wherever possible.

CHAPTER 2

MPI – The Message Passing Interface

This part of the document presents a concise overview of MPI, its origins, purpose and features. The goal is to familiarize the reader with all of the concepts that are used further on in the text, and to provide the necessary background information to support the research presented in later parts.

2.1 What is MPI?

MPI stands for "Message Passing Interface". In short: **MPI is a *standard library specification* for *message-passing* based parallel software**. First, MPI is a *standard* – meaning that all implementations should be at least compile-and-link-level compatible. Second, MPI is a *library* – it is not a language or language expansion and no adaptations to the compiler or other steps in the build process are needed to use it except inclusion of the library itself in binary form. Third, it is a *specification* – there is no "standard implementation" of MPI. Every vendor provides his own implementation of the library, and no single implementation "one-ups" another with regard to the standard itself. Last, MPI (in its core version) exclusively addresses the "pure" message-passing programming model – this is the case for any implementation, even those for hardware platforms that use shared memory at the core.

The MPI standard was designed by committee, almost exclusively by people deeply rooted in the world of parallel programming rather than the more general area of distributed computing. This is reflected in every aspect of its structure and philosophy.

2.2 A brief history

Sometime around early 1992, it became clear that many people were favorably inclined towards the creation of a parallel programming standard to deal with these issues. It was during the Supercomputing '92 conference in November 1992 that a number of the parties involved decided to start out on a standardization effort. The general goal was to develop a parallel programming library specification based upon the message passing programming model. The library specification would offer precise descriptions of names, calling sequences and return types of the subroutines and functions to be called from either the C or Fortran77 programming languages. Implementation of the specification would be left to individual vendors, though a reference implementation would be made available as soon as possible as a proof of concept for third-party developers.

An organization was set up to coordinate and guide the development of this standard: the MPI Forum. All members basically agreed on the following issues before the start of the standard definition process:

- The standard would be designed by committee and would not be "official" (as in ANSI or similarly approved), rather relying on innate attractiveness to vendors and users alike (through the obvious advantages it offered) to gain widespread following.
- The Forum would operate in a completely open way, allowing anyone to join in the discussions.
- The goal was to deliver a complete standard within the span of a single year.
- The Forum decided to adopt a format similar to that used for the earlier work in the High Performance Fortran Forum, which had proven to be highly successful.

The MPI Forum housed a wide representation of all the parties involved, gathering more than 40 organizations in its fold: these consisted of parallel computer vendors, as well as portable software library writers and parallel application specialists. Design of the standard was done through discussions over e-mail and physical meetings every 6 weeks.

A fully functional core message passing standard was completed by May 1994. A number of issues were deliberately left out of this specification, in order to gain time. These points would be elaborate on in later extensions to the original standard.

Over the course of 1995 to 1997, the forum reconvened to add functionality to the original standard concerning remote memory operations, parallel I/O, bindings for C++ and Fortran90 and other features. Most of these specifications became part of the MPI-2 standard, released in 1997. Most of the discussion in the rest of this thesis will focus on the capabilities offered in the original MPI-1 standard. This is mostly because the capabilities in MPI-2 are not needed for either a basic understanding of MPI, or the work further described in this thesis.

2.3 The goals of MPI

The MPI Forum had a number of very specific goals in mind when designing the standard. In a few words: the MPI standard should guarantee that compliant software offers the best possible combination of *portability, efficiency and functionality.* Ideally, MPI should enable software developers to write software that can take advantage of any specialized hard- and software that might be available on any platform, in a transparent manner. Library writers should be able to use MPI for creating functional and efficient parallel libraries.

- Portability:
  One of the primary goals was source code portability. The standard would have to offer compliant software full ability to compile and run with different implementations and on many hardware platforms without change. This feature was crucial if the standard was ever to find universal recognition. MPI implementations would be made available by parallel computer vendors for each of their available hardware products, possibly side-by-side with existing hardware-specific solutions for each of those platforms.

- Efficiency:
  The standard should allow for optimal implementations on a wide range of architectures with different memory-models, inter-processor communication systems etc.. Whatever the hardware platform, be it a cluster system, an MPP or anything in between: the available resources should be used as efficiently as possible to guarantee maximum performance, within reasonable but minimal boundaries with regard to the trade-off towards portability.

- Functionality:
  Lastly, an MPI-compliant library should provide a software writer with a decent amount of functionality in terms of library calls for parallel-programming purposes. Specifically, the MPI standard must offer, at least, different types of communication. This specifically with regard to collective data-transfer operations. It should also be able to handle different user-defined data types. finally, it should allow the user to easily map a diversity of processor topologies to problem domain topologies.

As a last important issue, the MPI standard should be able to support heterogeneous parallel architectures, a situation commonly found in commodity hardware based clusters, often built out of older machines with differing specifications. This point is of major importance towards grid systems, which could hypothetically span all over the world and connect a wide variety of machines.

Explicitly not included in the standard are issues such as how to launch an MPI program, as these might greatly vary from system to system. Items like process management and parallel I/O were deliberately postponed until the release of the second version of the MPI standard, MPI-2.

2.4 The MPI approach to parallel programming

Before going deeper into the specifics of the MPI standard, it is useful to first offer a broad overview of the general manner in which MPI supports the different aspects of parallel programming. To remain coherent with the preceding chapter, a general description of MPI will be offered in terms of memory architecture, problem decomposition and programming model. A few remarks on other points of interest complete this general description.

2.4.1 Memory model

At the logical (interface) level, MPI provides the programmer with a virtual **distributed memory system**. A job consists of multiple independent (and potentially different) processes, each of which has its own protected memory space. (Note that the MPI specification does not mention *processors* – it is up to the host system and possibly the MPI implementation to determine how many processes can be run, and how to divide them among actually available physical processors). All communication between processes must be done through the purely logical concept of "communicators" which bind processes together within a virtual common communication medium. If one process wants to share data with another, it must contact the receiving process through a shared communicator and transfer the data by means of one of the several communication routines specified in the standard.

To be perfectly clear: *the MPI specification itself puts no limitation on hardware that may run an MPI implementation.* If allowed to do so by the host system and the specific MPI implementation, it is entirely possible to run more processes than you actually have physical processors available. More importantly, it is possible to provide an efficient implementation of MPI on a shared memory hardware system by offering an appropriate implementation for the communication routines. For example: on a cluster system, these routines will involve copying the data to a network buffer, performing a network transfer and copying the new data further to its intended place at the destination process. In contrast, on a shared memory system, the routine might accomplish this same task by simply performing a memory-to-memory copy, thus ensuring the goal of efficiency.

The basic principle is to hide as much of the low-level details of communication as possible from the programmer, allowing him to focus on the actual task of parallelization. This allows for a fully portable set of software subroutines that can be used in any program, while the implementation takes care of optimizing the operation of each of these routines for individual platforms. Sidestepping these routines and hand-coding communication routines specific to a single platform cannot be allowed if the goals of portability and support for heterogeneity are to be observed.

2.4.2 Problem decomposition

Problem decomposition is then obviously done purely through **functional decomposition**, i.e. task parallelism. All parallelism has to be deliberately programmed

by the software developer. This is a necessity if the standard is to achieve the goal of sufficient generality towards a wide range of problems, many of which might not be suited to the other approach.

2.4.3 Programming model

Evident by its name, MPI approaches parallel programming through the **message-passing programming model**. Variations on this approach are strictly not supported in the first version of the standard, again to ensure generality. MPI-2, however, does support limited functionality for remote memory operations (active messaging).

The standardized routines offer different solutions towards problems like load-balancing and communication efficiency. This is accomplished by allowing for multiple types of calls and different levels of synchronization. Processes can block on a call, or remain in a running status as required by the programmer, thereby supporting overlap of communication and execution.

2.5 Characteristics and capabilities of MPI

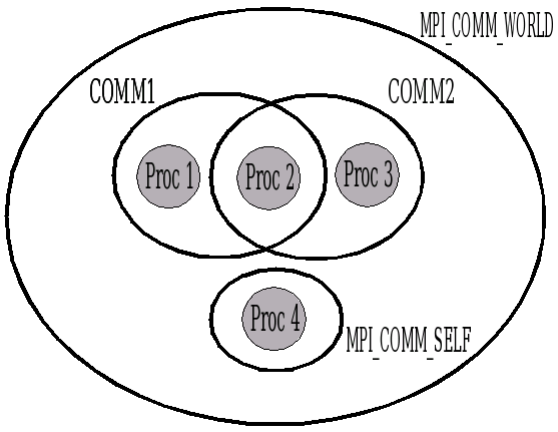
In order to get an adequate insight into the full scope of MPI, a little more detail is required. The following paragraphs present a concise introduction to the characteristics of the MPI standard and the set of library calls it offers. Many of these will be illustrated by code examples. For the sake of brevity and simplicity, these will only be presented in C code. Code examples for both Fortran and C++ (MPI 2) are very similar.

2.5.1 Basic entities

Central to the MPI approach are two concepts: the process and the communicator.

A **process** is a single computational entity with its own dedicated memory space. Processes should only be able to influence each other through appropriate calls defined in the MPI specification if consistency is to be guaranteed. The number of processes is fixed at startup, and there are no provisions in MPI 1 for adding new processes during runtime (though MPI 2 does define an MPI_spawn() call for exactly this purpose).

A **communicator** is an abstract entity representing a virtual communication medium shared between two or more processes. Basically, if a number of processes share a communicator, they can send messages to each other. In practice, processes access a communicator through a reference variable. Each process has a sequence number, referred to as its *rank* in MPI terminology, within any given communicator in which it participates, which is unique for that single communicator. Processes send messages to each other by passing to the messaging call a) the reference specifying the communicator over which to send the message and b) the target process' rank within that communicator.



Processes sharing a number of communicators. MPI_ COMM_ WORLD contains all processes within the same job. A communicator can be split up in new (sub)communicators (COMM1, COMM2). Each process also has a communicator MPI_ COMM_ SELF containing only itself.

Figure 2.1: Processes & communicators

At the start of a program, a single communicator, available through a reference variable called MPI_COMM_WORLD for all processes, is made available which contains all processes to be used for the job. More communicators may be created during the lifetime of a job. Figure 2.1 shows a schematic example of a number processes sharing a number of communicators.

2.5.2 Overview of MPI calls

MPI calls can basically be divided into 4 distinct types:

- "system" calls: **initialization** (e.g. setting up MPI_COMM_WORLD, starting communications, . . .), **management** (e.g. determining the number of processes available within MPI_COMM_WORLD or any other given communicator, setting up a new communicator, ...) and **termination** of an MPI program (closing down communications channels)
- ***point to point communication*** calls, between pairs of processes (e.g. blocking and unblocking send and receive operations, ...)
- ***collective communication*** calls, among groups of processes (e.g. gather-scatter, ...)
- calls for creating and managing user-defined **derived data types**

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      MPI_Init();
      //Do stuff
      MPI_Finalize();

      return 0;
}
```

This piece of example code represents the minimal MPI program (in C): header file inclusion plus a single main routine with initialization and termination calls.

Figure 2.2: Initialization and termination

2.5.3 Initialization, management and termination

Each MPI program requires an initialization and termination call to function properly. We illustrate this with a code example in figure 2.2.

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank;

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      //Do more stuff
      MPI_Finalize();

      return 0;
}
```

C example code to illustrate the use of management call bindings for retrieving the number of processes in a given communicator (MPI_ COMM_ WORLD) and the current process' rank within that communicator.

Figure 2.3: Some management calls

Management calls are all kinds of calls which do not involve messaging or handling data types. Some of the most frequently used of these calls are those to gain information on the number of processes within a given communicator and the rank of a given process within a communicator. Examples of these are given in figure 2.3. Other management calls include means of creating and managing new communicators and virtual topologies. Further discussion of these features goes beyond the

scope of this work though.

2.5.4 Point to point communication calls

Point to point communications are the basic communication calls in MPI, used to send a message from one sender process to a single other receiver process. There are two calls for each point-to-point interaction: a "send" call by the sending process, and a corresponding "receive" call by the destination process. Both an explicit send and receive action are mandatory – no data will be transferred without the participation of both processes. The message consists of an envelope indicating the source and destination process, and a body which contains the actual data to be sent.

The body of the message is written to / read from a buffer in memory, indicated by a pointer. The type of data in the buffer is identified by one of a series of cross-platform data types part of the MPI standard (e.g. INTEGER, FLOAT, ...) or a type defined by the user, and the number of data items to be sent/received is included as a parameter with each call.

Blocking point-to-point communication calls

Regular send and receive calls are *blocking*, meaning that they do not return before relevant completion criteria have been met, e.g. that the data has at least been copied to the send-buffers, and that the original buffer can be safely overwritten etc.

Straightforward blocking point-to-point interaction between two processes would look as shown in figure 2.4.

Non-blocking point-to-point communication calls

A send/receive may also be non-blocking. A non-blocking call returns immediately, freeing the process to do other things while communication goes on in the background. A separate call can be used later on to check whether the operation has actually completed or not.

An average non-blocking point-to-point interaction looks as shown in figure 2.5.

Non-blocking point-to-point interactions allow for greater overlapping of communication time and computing time, potentially reducing CPU idle time in some cases. Care must be taken though because a return on a non-blocking call does not guarantee that relevant completion criteria have been satisfied. Until fully complete, one can not assume that the message has been received or that the variables to be sent may be safely overwritten.

2.5.5 Collective communications

Collective communications allow larger groups of processes to share data in various ways which are more complex than the point-to-point approach, for example one-to-many or many-to-one. Although any of the basic effects of collective calls can also be produced through equivalent structures using point-to-point calls, using the collective calls will always carry a number of advantages.

For one, code using collective communications whenever possible will always be significantly less complex than any equivalent approach using point-to-point calls, simply because a single collective call will generally require multiple point-to-point calls to replace it. Thus, the possibility of errors is reduced and the code becomes

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank, some_data[5];

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      if (my_rank == 0) {
            //sender
            //initialize some_data
            MPI_Send(some_data, 5, MPI_INT,1, 0, MPI_COMM_WORLD);
      }
      else if (my_rank == 1) {
            //receiver
            MPI_Receive(some_data, 5, MPI_INT, 0, 0,
                  MPI_COMM_WORLD);
      }
      MPI_Finalize();

      return 0;
}
```

Example code for a pair of blocking point-to-point communication calls, one call each for sender and receiver processes.

Figure 2.4: blocking point-to-point (point-to-point) communication

easier to debug. On top of this, optimized forms of the collective routines are often faster than the equivalent operation expressed in terms of point-to-point routines.

Examples of collective communications include broadcasts, scatter, gather and reduction operations. The *root process* of a collective communication is the single process from which the data is sent, or on which the data is collected. Illustrated examples of the different collective operations follow.

Broadcast operations

In broadcast operations, the root process sends a copy of some piece of data to all other processes within a single communicator. Figure 2.6 shows a schematic representation of a broadcast operation. In the "before situation", the root process has a piece of data, which needs to be sent to the 4 other processes. The "after" situation shows that all processes within the communicator now have the data available for further processing. Figure 2.7 shows the corresponding code example.

Scatter operations

In scatter operations, a collection of data (e.g. an array) initially located at the root process is distributed over different processes within the same group (communicator). Figure 2.8 again shows a schematic example of a scatter operation, with a before and after situation sketch. A code example is given in figure 2.9.

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank, some_data[5];
      MPI_Request handle;
      MPI_Status status;

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      if (my_rank == 0) {
            //sender
            //initialize some_data
            MPI_Isend(some_data, 5, MPI_INT,1, 0, MPI_COMM_WORLD;
                  &handle);
            MPI_Wait(&handle, &status);
      }
      else if (my_rank == 1) {
            //receiver
            MPI_Ireceive(some_data, 5, MPI_INT, 0, 0,
                  MPI_COMM_WORLD, &handle);
            MPI_Wait(&handle, &status);
      }
      MPI_Finalize();

      return 0;
}
```

Example code for a pair of non-blocking point-to-point communication calls, one call each for sender and receiver processes.

Figure 2.5: non-blocking point-to-point communication

Gather operations

The gather operation is the inverse operation to the scatter operation: it "gathers" different pieces of data scattered over a group of processes into a collection in the memory space of the root. It is schematically illustrated in figure 2.10, with corresponding code given in figure 2.11.

Reduction operations

The reduction operation is sort of the inverse of the broadcast operation, taking data items from all processes within a communicator, and combining it into a single data item in the memory space of the root process. A reduction might be used, for example, to calculate the sum, maximum, minimum or all other kinds of results from a series of elements which are initially distributed over multiple processes within the same communicator. Figures 2.12 and 2.13 give both a schematic and code example.

2.5.6 Derived data types

MPI offers a whole range of standardized data types, the use of which guarantees consistency and compatibility when working across multiple computing platforms,

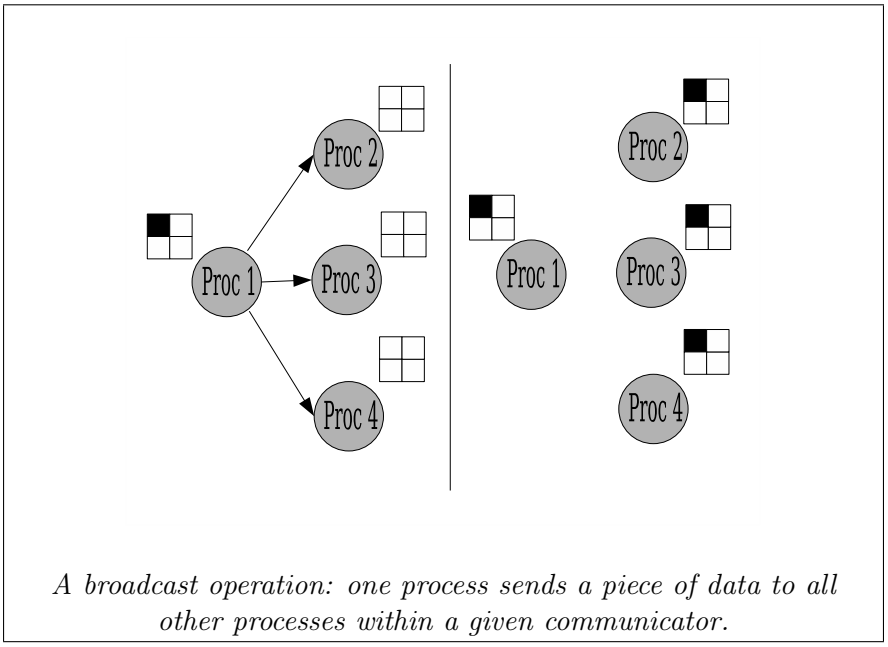


Figure 2.6: A broadcast operation

networks and programming languages. MPI also supports transfers of compound data in two ways:

- by calls for *packing* the data into a generic byte array which can then be transferred by means of a regular message
- by allowing users to define a *derived data type,* which can be reused throughout a program when certain combinations of data are in frequent use.

2.6 MPI in practice

To convey a practical idea of what it actually feels like to use MPI, this section will give an example describing how to run the MPI implementation called MPICH2 on a cluster of Linux-based machines, connected TCP/IP sockets over Ethernet.

MPICH2 is a freely available, portable implementation of MPI developed at Argonne National Laboratory. It is free software, and available for most flavors of UNIX and Microsoft Windows, as well as more specialized parallel hardware like MPPs.

Administrative issues

First, installing MPICH2 will require the system administrator to download and build the MPICH2 source code, and distribute the resulting binaries on the different machines in the cluster. This might be accomplished by uploading the binaries locally to the machines with ftp or like tools, or by putting the binaries on some kind of shared directory system. This process will provide all of the machines in the cluster

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank, some_data[5];

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      if (my_rank == 0) {
            //sender
            //initialize some_data
            MPI_Bcast(some_data, 5, MPI_INT, 0, MPI_COMM_WORLD);
      }
      else {
            //receivers -notice the collective nature of the call!
            MPI_Bcast(some_data, 5, MPI_INT, 0, MPI_COMM_WORLD);
      }
      MPI_Finalize();

      return 0;
}
```

Example code for a broadcast operation as illustrated in figure 2.6: note the identical call at both ends of the transaction (root and receiver process); there is no separate call for broadcast send and broadcast receive. All collective calls are handled in a similar manner.

Figure 2.7: Example code for a broadcast operation

with access to 3 important executable files: *mpd*, *mpicc* and *mpirun*.

MPD is an external process manager, used by MPICH for scalable startup of large MPI jobs. MPD forms a ring of daemons on the machines that will run MPI programs. The administrator can bring up a ring of MPDs by adding the names of all machines that are to run MPICH2 to a file called *mpd.hosts*, and then starting up the daemons by using the command *mpdboot*. The administrator can always inspect the current state of the ring by running the commands *mpdtrace* and *mpdringtest*.

User issues

A user can now write, compile, link and run MPI programs on the cluster. Writing MPI programs is done as presented in the many code examples given earlier. MPI programs can be compiled and linked using a set of scripts that use the same compiler that MPICH2 was built with. These are *mpicc*, *mpicxx*, *mpif77* and *mpif90* for C,++, Fortran77 and Fortran90 programs respectively.

After compiling and linking, the user can run the program from a master node of his own choice by using the command *mpiexec*, adding a flag *-n* to denote the number of nodes that is to be used in the job. The number of processes need not match the number of nodes in the ring – if there are more, they will wrap around.

When the job finishes, it will send output to *stdout* of the chosen master node.

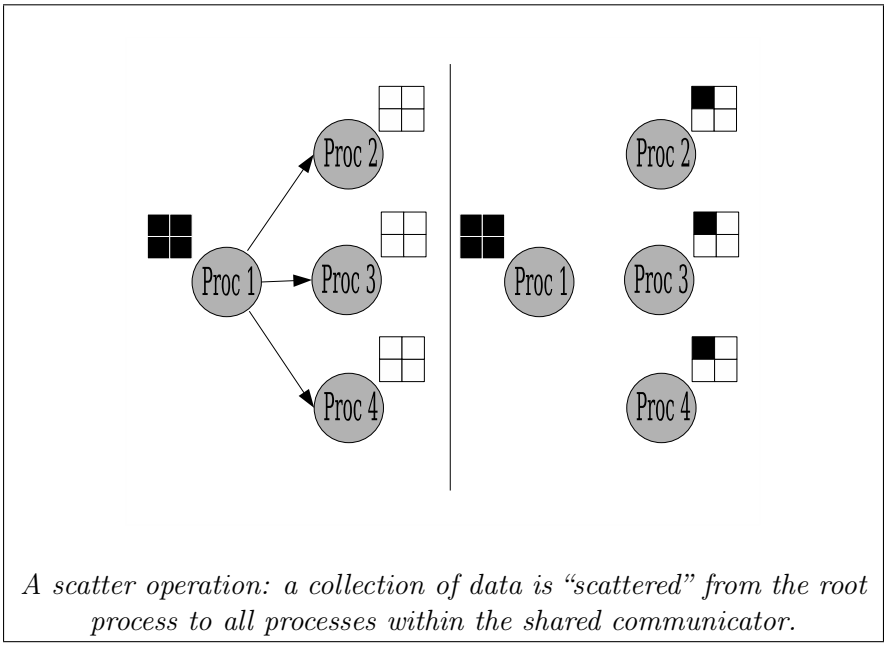


Figure 2.8: A scatter operation

2.7 What does MPI not do?

Despite offering a rather broad selection of possible features within the message passing paradigm, there are some issues which are not covered by the MPI-1 standard. Support for some of these was added with the introduction of MPI-2, notably a number of calls for process startup and management during runtime (which had been available in PVM for a long time). Other topics expanded upon by MPI-2 include thread-safety (MPI-1 allowed, but did not require implementations to be thread-safe, without going deeper into the subject) and an innovative approach to parallel I/O.

MPI-2 even adds (portable and platform independent) calls for safe remote memory operations, thereby stepping outside the boundaries of strict message passing (this was mainly due to requests from the MPP side of the parallel programming community: remote memory operations add possibilities for performance optimization on such parallel computing platforms, and some problems are simply easier to express in terms of one sided remote-memory operations than in terms of a two-sided message exchange).

One problem which is completely disregarded in any of the MPI standards though, is the issue of fault tolerance. The reasons given for this omission are generally quoted as follows [27]:

- standards rarely specify the behavior of erroneous programs or consequences of events beyond the standard (such as power failures)

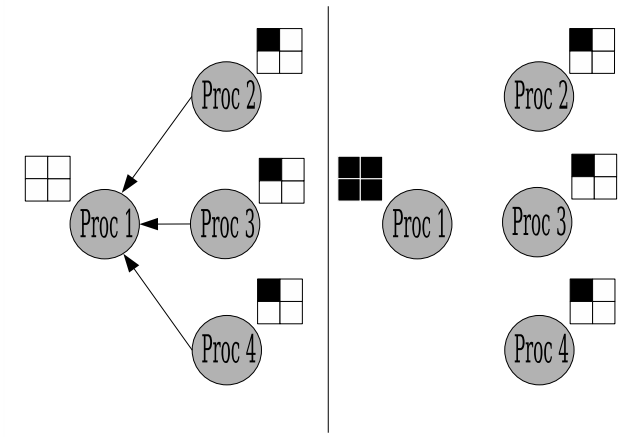
```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank, some_data[5], piece_of_data;

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      if (my_rank == 0) {
            //sender
            //initialize some_data
            MPI_Scatter(some_data, 5, MPI_INT, null,
                  0, MPI_INT, 0, MPI_COMM_WORLD);
      }
      else {
            //receivers -notice the collective nature of the call!
            MPI_Scatter(null, 0, MPI_INT, piece_of_data, 1, MPI_INT,
                  0, MPI_COMM_WORLD);
      }
      MPI_Finalize();

      return 0;
}
```

Example code for the scatter operation illustrated in figure 2.8. Code is given for both an evenly and an unevenly scattered array.

Figure 2.9: Example code for a scatter operation

- failure of one process among a thousand during an MPI collective operation makes it very difficult to recover

When working with systems where communication between processes is guaranteed by the underlying operating environment, this offers maximal potential for implementing scalable and high-performance collective routines. Fault tolerance is an acute issue though when working with more loosely coupled environments, like a network of workstations. As our focus for this thesis lies with work on loosely coupled, heterogeneous, geographically distributed resources, fault tolerance becomes the natural focus for our next chapter.



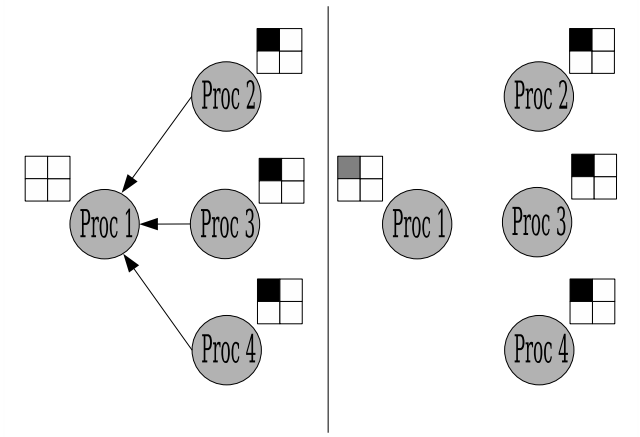
Data is "gathered" from the memory spaces of all processes in the communicator into the memory space of the root process.

Figure 2.10: A gather operation

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank, some_data[5], piece_of_data;

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      if (my_rank == 0) {
            //sender
            //initialize some_data
            MPI_Gather(some_data, 5, MPI_INT, null,
                  0, MPI_INT, 0, MPI_COMM_WORLD);
      }
      else {
            //receivers -notice the collective nature of the call!
            MPI_Gather(null, 0, MPI_INT, piece_of_data, 1, MPI_INT,
                  0, MPI_COMM_WORLD);
      }
      MPI_Finalize();

      return 0;
}
```

Example code for the gather operation illustrated in figure 2.10. Code is given for both an evenly and an unevenly scattered array.

Figure 2.11: Example code for a gather operation



Data items distributed over the memory spaces of different processes within the communicator are combined into one new value on the root process.

Figure 2.12: A reduction operation

```
#include "mpi.h"

int main(int argc, char* argv[]) {
      int numpprocs, my_rank, reduced_data, piece_of_data;

      MPI_Init();
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      if (my_rank == 0) {
            //receiver
            //initialize some_data
            MPI_Reduce(null, &reduced_data, 1, MPI_INT,
                  MPI_MIN, 0, MPI_COMM_WORLD));;
      }
      else {
            //senders -notice the collective nature of the call!
            MPI_Reduce(&piece_of_data, null, 1, MPI_INT,
                  MPI_MIN, 0, MPI_COMM_WORLD));;
      }
      MPI_Finalize();

      return 0;
}
```

Example code for the reduction operation illustrated in figure 2.12.

Figure 2.13: Example code for a reduction operation

CHAPTER 3

Fault Tolerance and MPI

In this chapter, we will already try to answer at least part of our first research question: *with regard to MPI implementations: what are the current options w.r.t. fault-tolerance and MPI, what would be the best approach for the particular case of small research groups working with low-budget hardware.*

Since the introduction of MPI-2, the standard way of dealing with any MPI-related run-time error – not related to I/O – has been to invoke the error handler MPI_ERRORS_ARE_FATAL. (This was the unspecified default behavior of most MPI-1 implementations.) The result of this approach is that any fault which occurs during a non-IO MPI call will cause the running job to stop all computation and abort. (Due to their specific nature, the designers of the MPI-2 standard decided to use the error-handler MPI_ERRORS_RETURN for all MPI-IO related operations, as problems like disk failure might not directly influence the ability of modern hardware platforms to go on computing a job.)

It is possible for the user to override an error-handler, but the standard gives no guarantees whatsoever as to the further usability or stability of the MPI runtime after the fault was detected and the error-handler invoked. The intention for the introduction of error handlers in the MPI-2 standard was mostly to offer the user a way to salvage as much already computed data as possible before abort, rather than to provide a mechanism for fault-tolerance. Even so, the standard offers no certainties as to how much of the already computed data, if any, could potentially be saved.

3.1 General concepts

It might be useful to clarify some frequently used terms before going further into this subject matter.

3.1.1 Faults

A parallel computing system consists of multiple hard- and software components, which run a *job*. We say that a *fault* occurs when one of these components starts to function incorrectly during a job. In general, stating that a component *fails* means that it ceases to function at all, though the two terms are often used interchangeably. Obviously, failure always implies a fault as "ceasing to function" is clearly a form of incorrect behavior. Faults might occur in hardware components (*hardware-level faults*) or in software components (*software-level faults*). A fault in one component might eventually lead to faulty behavior in another, causing a *fault cascade*.

Parallel hardware will generally consist of computing components (*nodes*), connected by linking components (*links*). Thus, hardware faults will generally be classified as *node-level faults* if they occur in a computing component, or *link-level faults* if they occur in a linking component. Depending on the context, a third type of hardware fault might be discerned: *storage-level faults* can occur when the parallel computer uses storage components that function independently from the nodes, e.g. elements within a Storage area Network (SAN). (On an abstract level, independent storage components might be seen as a special type of node, with the connections of the SAN itself being a special type of link). Hardware-level faults might be caused by many circumstances, such as natural wear (e.g. the component is simply too old), failure in supporting hardware (e.g. overheating due to failed air-conditioning) or external actions (e.g. accidental cutting of a network cable).

A job consists of one or more *processes*, which communicate through *messages*. This implies that software faults can generally be divided into *process-level faults* – faults within the regular execution process of a job – and *message level faults*: faults with the regular transmission of messages. Process-level faults can be the result of a fault cascade caused by hardware-level faults (e.g. because of failure in the node running the process), they can be caused by problems with supporting software components (e.g. bugs in the operating system), or they can be the result of bugs in the software component itself (e.g. buffer overflows, memory leaks, ...). Message-level faults will generally be the result of a fault-cascade involving earlier node-level, link-level or process-level faults or of faults in supporting software components (e.g. a bug in the IP stack).

3.1.2 Fault-detection

It is important to note that faults are not automatically detected by the system. This is the responsibility of *fault detection* components, which might be supported both in soft- and hardware. Some types of faults will be very hard to detect, because, even though the failing component produces no – or faulty – output, it will outwardly still behave as if it works correctly (e.g. a faulty network interface, which still receives and transmits but invisibly corrupts data in the process). We refer to these deviously undetectable errors as *Byzantine failures*.

3.1.3 Fault-tolerance

When a fault occurs during a job, this creates an exceptional situation with potential effects on further execution of the job. If this situation remains undetected and unhandled, this will normally lead to an *error,* as regular execution of the job can no longer be expected to consistently produce correct results. Erroneous behavior might lead to the production of incorrect output, and/or the loss of the ability to produce some or all output.

We say that a system is *fault-tolerant* if it is able to consistently experience one or more faults – perhaps only in certain components – without producing errors. From this, it becomes clear that fault-tolerance can come in many degrees, depending on how many faults can be handled, and in which components. If a system supports *node-level fault-tolerance*, it is able to survive one or more node-level faults without producing errors. Likewise, *link-level fault-tolerance* implies that a system is able to survive one or more link-level faults*, process-level fault-tolerance* the ability to survive faults in one or more processes and *message-level fault-tolerance* that the system knows how to deal with one or more faulty messages.

It is obvious that decent process-level fault-tolerance will require at least some degree of node-level fault-tolerance, just as comprehensive message-level fault-tolerance will require some link-level fault-tolerance. This line of reasoning also makes it clear that fault-detection is crucial to the issue of fault-tolerance. A system will never be able to consistently handle faults if it doesn't know that a fault happened to begin with.

3.2 Viability of the standard MPI approach

3.2.1 Monolithic architectures

The current MPI approach is quite reasonable within the context of highly integrated, monolithic MPP like architectures which mostly use shared memory at the hardware level. An MPI-level fault in one of these machines will, excluding bugs within the MPI implementation itself, most probably indicate some kind of grave hardware-level fault. A fault like this will likely endanger the integrity of the machine as a whole. Such acute problems will make it extremely unwise to go on with any running job. Depending on the particular hardware problem, there might even already be a realistic chance of data corruption. In this case the best approach would naturally be to bring the machine down, and any running jobs with it.

However, when dealing with parallel computing platforms which use interconnected, independent computing "nodes" with their own CPU and memory (distributed), as have been becoming more and more popular over the last 10 years [7], things change. Paradoxically, the situation becomes both more complex and potentially less dramatic as a wide array of measures might be taken to make such systems more robust.

3.2.2 Modular architectures

Within the context of modern day distributed memory parallel architectures, non-Byzantine node and link-level faults will generally not result in destabilizing the parallel computer as a whole. Under normal circumstances, computing might go on using any remaining active nodes. For example, this comes natural to cluster systems, as each node in a cluster is an independent computer. Normally, failure of a single node will not directly influence the stability of other nodes in the network. Lately, even the more tightly integrated supercomputers have been sharing in this ability, as research into hardware modularity has led to massive improvements in uptime [8]. To illustrate the efficiency of modular design with regard to node-level fault-tolerance, consider the following example: it is estimated that the 1 petaflop Cray "Baker" system with around 25.000 nodes, scheduled to go into service at ORNL in 2008, will suffer an average failure rate of 10 nodes a day. Still, even under such circumstances, the machine will be able to run without global downtime for an average of days [69]. Note that we are purely discussing fault-tolerance at the hardware and OS layer in this case – node failure will influence any running processes on the failed node. In other words, *they support node-level fault-tolerance at the platform-level (OS), but not at the MPI-level.* This is a fact which we will come back to in the next section.

Depending on the particular system architecture, link-level faults can be dealt with as well by providing backup-links in the networking fabric and/or by re-routing calls to circumvent broken interconnects. Careful design of the interlink structure and communication patterns will be able to overcome even multiple failures at the hardware level before the communication fabric effectively splits. Hypothetically speaking, pathological link failure on such a scale probably points towards technical causes that would likely lead to global machine failure anyhow.

3.2.3 Byzantine failures

Byzantine failures are harder to detect and handle, but even these are not necessarily unrecoverable. Problems with unstable – but still active – memory or network modules might lead to data corruption. A detection mechanism like CRC checking makes it very likely that such failures are discovered, and possibly even corrected, in time for the OS to take appropriate action without taking the system down as a whole [51].

The possibilities for fault-tolerance at the hardware level created by these modular parallel computing architectures force us to take another look at the issue of fault-tolerance in MPI, as it currently doesn't allow for effective use of the hardware-level fault-tolerance features of the new modular parallel computer designs.

3.2.4 The necessity of fault tolerance

Bringing fault-tolerance into MPI is not a straightforward matter. As was mentioned earlier, fault-tolerance at the system-wide scale does not change anything about the loss of data which occurs during node failure, or the loss of messages during a failure at the link level. The MPI standard does currently not include any features to deal with the loss of processes or messages from an active runtime. No matter how long

the parallel computing system as a whole manages to stay up and running at the hardware level, the previously quoted 10 node failures per day will bring any MPI job using those nodes to halt with a very real risk of losing all previously computed data. In other words: the global system might remain stable for days, but any MPI application using all available resources will statistically last no longer than a few hours. This is, of course, unacceptable.

All of this makes the question of how to integrate fault tolerance with MPI into a very active research topic.

3.3 Process-level fault tolerance

3.3.1 Process-level fault-tolerance in MPI

Any discussion on adding process-level fault tolerance to MPI will eventually lead to a divergence into two distinct practical approaches.

"Do it at the MPI-level"

The first philosophy states that actually, the issue of fault-tolerance should not be the problem of parallel software writers, nor should it be something which the MPI standard itself has to worry about. Fault tolerance should be provided at a lower level, invisible to the user, within the parallel computing middleware. In other words: *fault tolerance is not a problem of MPI-based software or even the MPI standard itself – it is a feature that should be transparently provided by an appropriate MPI implementation.* We will refer to the proponents of this perspective as the school of **transparent fault-tolerance**. The idea of transparent fault-tolerance is very attractive for several reasons. For one, a transparently fault-tolerant MPI implementation removes the need for any changes to the current MPI standard. Additionally, any regular MPI-based software which is run on such an MPI implementation will automatically and immediately become fault-tolerant, without any need for changes to the code-base of the parallel software itself.

According to the school of transparent fault-tolerance, the need to actually provide fault-tolerance at the application level would place an unwanted burden on parallel software developers. It would make the task of writing correct, efficient parallel software even harder than it currently already is. As well, the added complexity would naturally come with a price in terms of performance and an increased chance of bugs. The combined weight of all these significant arguments has led most researchers currently working on the issue of bringing fault-tolerance into MPI to focus their attention on this particular approach. Research in this area mostly focuses on two techniques for adding transparent fault-tolerance to MPI implementations: checkpoint-restart and message logging.

"Do it at the application level"

The second philosophy, takes an approach which is pretty much opposite to the one advanced by the school of transparent fault-tolerance. The responsibilities for

fault-tolerance, from this perspective, are mostly put into the hands of the parallel application itself. According to this line of reasoning, MPI should provide only minimal "direct" fault-tolerance of itself. Instead, it should offer application developers a practical toolbox of methods for handling faults within the application itself. Briefly: *fault tolerance is an application-dependent issue – the MPI runtime itself should only provide a basic ability to survive the loss of nodes and/or links, leaving the mechanics for recovery of data or the process of resuming computation after an error up to the specific application.* We will refer to followers of this approach as the school of **application-controlled fault tolerance**.

This approach has a number of ramifications – for one, it will require additions to the current MPI standard. For example, the MPI standard does not specify the required behavior of the MPI runtime in case of node-level failure. Such questions as to what should happen to communicators containing the failed node, or collective communications involving a failed node, require a definite and unambiguous answer within the MPI specification. For another direct implication, requiring programmers to take care of fault-tolerance themselves will naturally increase the complexity of parallel code in comparison to that of current MPI-based software.

Supporters of application-controlled fault-tolerance in MPI, however, bring up a number of points to their defense. First, they reason that different applications will have different needs in terms of fault-tolerance. While the transparent approach is generic, it is also quite complex - more complex than a lot of applications might require. A farmer-worker type of problem for example, would be rather simple to recover in case of failure in one or more of the workers: it would only require the farmer to resend the task data that did not return a result due to the crash, either to a re-spawned version of the failed process, or by re-dividing it among the remaining worker processes. In this case, some kind of checkpointing is only necessary for the farmer. Another example includes a class of algorithms for solving partial differential equations referred to as "naturally fault-tolerant". Based on mesh-less methods and chaotic relaxation, these algorithms will converge correctly even when a marginal number of processes is lost – a marginal number of processes in this context can still be 100 process failures in a 100.000 processor job [57]. Last, but not least, PVM users have actually been doing application controlled fault recovery for years [58]. This is, in fact, one of the main reasons for many research groups to prefer it over MPI. This proves the feasibility of the approach for at least a wide range of problems.

3.3.2 Transparent fault-tolerance

All current MPI implementations which offer transparent fault-tolerance use some form of checkpoint/restart or log-based recovery scheme [63]. We will describe these recovery techniques in some detail in the following two sections. The basic idea here is that the failed process is restarted and then "fast-forwarded" to its last known consistent state. This requires that the MPI run-time collects and retains data on the state of various processes running at any time during execution. This is a process which will naturally require resources in terms of storage and CPU time, while potentially interfering with the regular execution of other processes in a job, i.e. because some of these techniques require that all processes in a job (including

the ones that did not fail) synchronize on a globally consistent state.

Checkpoint/restart-based recovery

Checkpoint-restart systems in general keep track of a process by periodically making an image (or *snapshot*) of the process that is to be protected (checkpointing). This snapshot contains all necessary information on the state of the process in order to restore it to it's exact shape at the time of checkpointing (data in memory, register values, ...). This data is then written to some form of stable storage. In case of process failure, the snapshot is retrieved from stable storage and the process is immediately restored (rolled-back) to its last registered consistent state directly from its snapshot [30]. Many implementations of checkpoint/restart systems are available such as libckpt [33], Condor [32], CRAK (Checkpoint/Restart As a Kernel module) [34] and BLCR (Berkeley Lab's Checkpoint/Restart) [31]. These differ in many ways, including the way in which process state is preserved, how much of it is retained, the way in which it is stored, APIs etc.

As they consist of multiple, partly independent processes, parallel applications will require special checkpoint/restart protocols that make a global snapshot of the application. This is accomplished by taking the union of the individual snapshots, accounting for global state [29]. The two most popular methods used for current fault-tolerant MPI implementations are uncoordinated and coordinated checkpointing.

Uncoordinated checkpointing [30] leaves the decision on when to make a snapshot up to each individual process. The advantage of this approach is that each process can checkpoint whenever it is most convenient, for instance when the amount of state data to be saved is small [64]. Its major disadvantage is its vulnerability to the so-called *domino-effect*.

The domino-effect describes a situation in which restart of the failed process will force other processes to rollback to a previous snapshot as well because of shared dependencies, one after the other until the global state is effectively returned to the starting position [39]. A simple example involves one process checkpointing before a receiving a message from another and crashing after receiving his message but before new snapshot can be taken. Simply restoring the crashed process to the checkpoint will lead to an inconsistence in global state, as the restored process will be unaware of the lost message while the other will actually consider it sent and received. Thus, the other process has to be rolled back as well, creating a further opening for situations of this kind, potentially all the way until the application loses all the work performed before the failure.

Other problems with uncoordinated checkpointing include the fact that checkpoints might be taken which will never be part of a globally consistent state and the issue that multiple checkpoints need to be maintained for the same process, all of these being associated with a cost in storage and CPU resources, complicated further because some form of non-trivial garbage collection will be required.

Coordinated checkpointing [30] attempts to solve these problems by centrally forcing the checkpoints of all processes to form a global consistent state. This means that there will never be useless checkpoints, as well as simplifying recovery by re-

moving the threat of the domino-effect, since every process can simply roll back once to the most recent checkpoint. Another advantage is that only one checkpoint needs to be maintained per process at a time, removing the need for garbage collection. The main disadvantage is that committing output is hampered by the need to globally commit a checkpoint before output in order to maintain guaranteed consistent output.

In its simplest form, a single coordinator tells all processes to block all communications during the snapshot and confirm successful completion of a tentative checkpoint using some kind of two-phase-commit protocol [40]. This approach generates a lot of overhead though, and non-blocking schemes are preferable [41]. Distributed snapshot [29] and checkpoint indices [42] are examples of such non-blocking coordinating protocols. Loosely synchronized clocks can also facilitate checkpoint coordination [65].

In general though, "vanilla" coordinated checkpointing will always require all processes to participate in every checkpoint, with possibly severe consequences in terms of scalability. Some optimization of the number of checkpoints that needs to be taken might be possible, based upon what communication happened since the last global checkpoint [66].

Log-based recovery

Log based recovery protocols model process execution as a sequence of deterministic state intervals, each initiated by the execution of a non-deterministic event [43]. These non-deterministic events have traditionally been identified with message receipt [45, 43, 44], giving rise to the more common descriptor of "message logging protocols" for these recovery schemes. (In this context, it is important to note that *sending* a message is not non-deterministic.) Message logging techniques are based on the premise that a log entry on all non-deterministic events (messages) can be registered onto some kind of stable storage (also called *committing*). In case a process fails, it can simply be restarted from scratch, and restored to its most recent recoverable stable state by replaying all of its log entries. In practice, all message logging protocols employ some kind of checkpointing as well to reduce the impact of log replaying during recovery.

Pessimistic logging protocols assume that failure can occur after any message receive and therefore, that no message should be allowed to influence the system before being logged. Thus, in their most straightforward form, pessimistic message logging protocols will synchronize log commits with all message events. Hence, pessimistic logging is also referred to as *synchronous logging*. These protocols assume that failures will happen in any place, at any time and with any frequency – an assumption which is "pessimistic" since, in reality, failures should be rare. This approach implies that all effect of failure and recovery are confined to the failed process only. As well, the state of a recovered process will always be consistent within the context of global state as it will always be recovered to a state including its last interaction with any other process or with the outside world. These are both highly desirable features [67].

These advantages come at the price of performance. Each write operation to

stable storage will consume more time than regular operations in volatile memory, and latency will be incurred on any individual write operation. Thus, pessimistic message logging will effectively maximize the amount of time lost to committing log entries by not allowing log entries to accumulate before commit. Implementations must therefore use some kind of special technique to reduce the impact of logging on performance. Some protocols use special hardware to accomplish this goal [45, 46]. Others limit the number of failures that they are able to handle simultaneously [49, 48], or relax the requirement of logging atomicity [47, 48].

Optimistic logging protocols do allow log entries to accumulate in volatile memory, only periodically flushing them to stable storage under the "optimistic" assumption that logging will complete before a failure occurs. Thus, as logging is not synchronized with messaging, this approach is also known as *asynchronous logging*. This allows the protocol to minimize time loss due to commit latency, as well as enabling processes to continue computation before a log entry has been written. Log entries can be committed when it will have the least impact on performance, and therefore, applications will incur little overhead from it. Recovery does become more complex though.

Most importantly, optimistic logging allows for the creation of "orphans" - processes which are no longer consistent with the global state of the application after recovery of the failed process. For example, after recovery, a failed process might be unaware of a message receipt which was not committed to log in time, while other (surviving) processes will be aware of sending the message, forcing these processes to roll back as well. Thus, in order to achieve consistency, optimistic logging protocols will have to roll back orphan processes until their state does not depend on messages which have not been logged. This implies that, contrary to pessimistic logging schemes, optimistic logging protocols need to retain multiple checkpoints of the same process at the same time.

Examples of transparently fault-tolerant MPI implementations

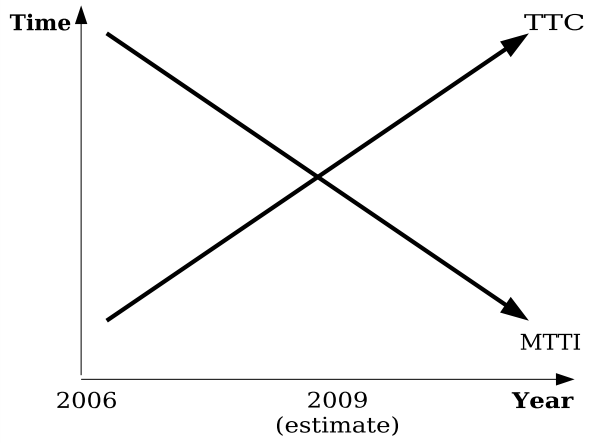
A few MPI libraries have attempted to integrate (a selection of) these techniques in various ways [63]. Starfish [35] provides support for coordinated and uncoordinated checkpointing. MPICH-V [36] combines an uncoordinated checkpoint/restart protocol with message logging to account for process state. CoCheck [37] uses a coordinated checkpoint/restart protocol with the Condor checkpoint/restart system. LAM-MPI [38] distinguishes itself from these previous examples, which are all tightly coupled with one specific checkpoint/restart system, by providing a modularized approach which allows for integrating multiple checkpoint/restart systems into its code base. LAM-MPI however supports only coordinated checkpointing. OpenMPI [68] is a joint research project into combining the efforts from some of these systems into a single new, unified, next-generation MPI implementation. OpenMPI requires the ability to support multiple checkpoint/restart services (BLCR, libckpt, ...) and a variety of checkpoint/restart protocols (coordinated, uncoordinated, ...), including the ability to support checkpointing on heterogeneous systems which combine different checkpoint/restart services.

Issues with transparent fault-tolerance.

Though a lot of problems with transparent fault-tolerance have been solved, the current evolution of parallel hardware might be threatening its fundamental principles. The amount of processing units in a single machine keeps on rising. Practical plans for petaflop-scale computers in the near future are a reality. But parallel computation at this size brings about a number of new problems. A critical issue with machines of this size is the decreasing Mean Time To Interrupt (MTTI) [57].

Going back to the example of the planned 1 petaflop Cray "Baker" system at ORNL: at an average failure rate of 10 nodes a day, a single job running for a week could crash 70 times before delivering a result. In this particular case, the transparent approach is estimated to require checkpointing about every 20 minutes with snapshots of up to 150 TB. Knowing that the system will "only" have about 175 TB of storage in total, and taking into account (a) the time needed for checkpointing and (b) the actual computing time lost due to the failures themselves, one might wonder if there will be any storage space or CPU cycles left for actual computing [69].

Projecting from existing supercomputers, a 100.000 processor supercomputer with all associated support systems could see a failure every few minutes [57, 69]. In fact, IBM warns users of the BlueGene/L supercomputer that "faults are expected to be the norm, rather than the exception" [28]. As the amount of processors goes up, the MTTI decreases while the time to checkpoint will increase. This is illustrated in figure 3.1.



At current rates, some authors [69, 28] estimated MTTI and TTC to cross over around 2009. At such a point, checkpointing an application on the largest machines would effectively take longer than the time between two failures.

Figure 3.1: Crossover of *Mean Time To Interrupt* and *Time To Checkpoint*

The good news is that currently (2009), MTTIs turn out quite a bit better than expected at the time of the cited studies. However, they still remain on the decrease. More importantly, due to the simple realities with regard to systems of ever growing complexity, there is no reason to expect that this trend will turn around. Thus, the concerns raised remain valid – even if the advances in current technology have proven able to temporarily buy us some extra time.

3.3.3 Application-controlled fault-tolerance

In order to enable application developers to take on fault handling themselves, we require an answer to three fundamental questions: (1) how can we detect a failure, (2) how do we notify an MPI application process of this failure, and (3) how can a developer make the process able to recover from the failure? Support for the first problem would require some kind of failure detection within the MPI implementation. The second problem could be resolved by defining an appropriate special error code, or by using the mechanisms for supporting user-defined error handlers. Assuming that this gives us an adequate answer to the first two questions, lets assume the abstraction that a process changes state from "no failures" to "failure recognized" when it is notified of the failure. We are then confronted with another set of three questions: (1) what are the necessary steps and options that an application process must take in order to start the recovery procedure and bring itself back into "no failure", (2) what is the status of MPI objects after recovery and (3) what is the status of ongoing communication and messages during and after recovery? [60]

Research into application controlled fault-tolerance has been sparse up till now. Actually, there is currently only one research project which has seriously investigated these questions: the FT-MPI project of the Innovative Computing Laboratory at the University of Tennessee, Knoxville (UTK) [59].

FT-MPI: overview

The FT-MPI project covers a double effort.

First, it is an extension proposal to the current MPI standard adding elements to cover the issue of fault-tolerance [60]. This part of the MPI project is basically a text which, just like the MPI standard itself, puts as little emphasis as possible on implementation or platform-dependent issues. The extension proposal covers the basic answers to the questions above and introduces a number of "modes" for application recovery, MPI object recovery and message recovery (both point-to-point and collective), all of which define how an application will eventually recover from a fault. It puts an emphasis on adding as little as possible to the original standard to ensure that any software written for regular MPI will also function on FT-MPI, and vice-versa – albeit of course without fault-tolerance.

Second, the UTK provides a reference implementation of MPI-1 and selected parts of MPI-2, incorporating the features in said extension [62]. Packed with the reference implementation come a number of example implementations for a diverse range of problems, demonstrating the ability of FT-MPI to effectively render these algorithms fault tolerant at a minimal cost in terms of performance [61].

The FT-MPI specification

The only assumption which the FT-MPI specification makes is that the run-time environment effectively discovers failures.

Following this, all remaining processes are notified of the failed process through the usage of a special error code. As soon as a process receives this notification, it changes state from "no failures" to "failure recognized". While in this state, the process is restricted in the types of actions it can perform, depending on initial parameters chosen by the administrator of the parallel application at startup. Basic functionality of FT-MPI is covered by 4 distinct operational "modes". The FT-MPI spec does not define how to initialize these modes for an application, as this procedure will be platform-dependent.

The recovery mode covers all of the steps needed to bring the process back to "no failure". FT-MPI provides 3 modes for recovery. First, there is the automatic recovery mode *FTMPI_RECOVERY_MODE_AUTO*, where recovery is started automatically by the MPI run-time as soon as a failure-event has been recognized. After recovery has successfully finished, the error handler of MPI_COMM_WORLD is called (as other communicators will have seized to exist at this point in time – see the paragraph on the communicator mode for further explanation). Second, there is a manual recovery mode called *FT-MPI_RECOVERY_MODE_MANUAL*, where the application itself has to start the recovery procedure through the usage of a special MPI function. In practice, the MPI run-time calls the error handler attached to the current communicator – the user is however not allowed to call any MPI communication function before recovery has been started. Finally, there is a mode *FTMPI_RECOVERY_MODE_IGNORE*, where recovery does not have to be initiated at all, but any communication to a failed process will raise an error (point-to-point communication, collective communication and communicator creations).

The communicator mode describes the way in which FT-MPI recovers MPI objects, more specifically MPI_COMM_WORLD. All communicator modes will (a) recover all objects which only contain local information, and (b) recover no objects which contain global information, with the exception of MPI_COMM_WORLD – such objects are "destroyed" during recovery, and only objects available after MPI_Init (i.e. MPI_COMM_WORLD and MPI_COMM_SELF) are re-instantiated by FT-MPI. The communicator mode thus effectively describes various ways in which to rebuild MPI_COMM_WORLD after failure.

Using the mode *FTMPI_COMM_MODE_REBUILD*, FT-MPI will replace the failed process by a new one (which will all be started from line 1, i.e. no checkpoints). All surviving processes retain their original rank, but no guarantee is given as to which resources the new processes are allocated to. Instead of replacing the failed process, a user can leave the ranks of failed processes blank using *FTMPI_COMM_MODE_BLANK*. The size of MPI_COMM_WORLD will remain unchanged, but the failed processes are "blanked out", i.e. any point-to-point message sent to them will not raise an error – but won't transfer any data either. Any receive posted on any blanked out process will return a null-status and leave the receive buffer unchanged. Collective operations will act as if the process simply didn't exist. Alternatively, the user can also request that FT-MPI compacts MPI_COMM_WORLD

in order for it to remain contiguous using *FTMPI_COMM_MODE_SHRINK*. The size of MPI_COMM_WORLD will be adjusted to the number of surviving processes. This might alter the ranks of surviving processes, but the original sequence is guaranteed to be identical to the situation before recovery. Finally, FT-MPI retains a mode called *FTMPI_COMM_MODE_ABORT* for backward compatibility, which will abort execution gracefully if one or several processes have failed.

The message mode controls the expected behavior of messages before, during and after recovery. It defines the way in which FT-MPI treats messages which are "on the fly" when a failure occurs. In general, all message modes will discard (a) all messages from and to dead processes and (b) all collective operations. Using *FTMPI_MSG_MODE_RESET*, all ongoing and posted messages will be discarded as soon as a recovery operation has been started. In *FTMPI_MSG_MODE_CONT*, any message sent from a surviving process to a surviving process before the failure will be delivered after recovery. The user has to take care that in this case, the communicator is reconstructed in the identical order as used previously. Message originator and destination will be adjusted by FT-MPI when needed to reflect any potential new situation after failure when using FTMPI_COMM_MODE_SHRINK.

The collective mode, finally, determines the guarantees given for collective operations. While it is not very complex to define when a point-to-point operation has failed, it is more difficult to provide a similar definition for collective operations. One has to take into account that the MPI standard does not guarantee that collective operations synchronize all participating processes on receive (which is not desirable in regular, fault-less cases because of the associated cost in performance). So, although all collective communications ongoing during a failure are discarded, it might happen that some processes manage to complete the operation successfully (before failure) while others (after failure) can't. This would lead to inconsistency in the code returned to all participating processes. FT-MPI offers users the choice whether they are willing to risk this possibility or not. Using *FTMPI_COLL_MODE_NONATOMIC*, no guarantees are given - some processes might report that the operation has succeeded while others report an error. Under *FTMPI_COLL_MODE_ATOMIC*, all collective operations are guaranteed to be atomic, i.e. all processes will either will report either success or failure, at the possible cost of memory consumption and higher execution times.

Although the current spec does cover most issues, some items do currently remain open for discussion. One example is the question on how to handle non-blocking receives from a failing source using wildcards while employing message mode FTMPI_MSG_MODE_CONT. After failure, an implementation will not cancel the receive, unaware that it is actually, but implicitly, directed to a source which has failed and will thus never be sending it. This would lead the application to deadlock. Another example directly relates to the fact that error handlers were originally never designed with actual fault-tolerance in mind. Its major drawback is that just one error handler can be registered at a time. This hampers the ability of FT-MPI to be used in writing fault-tolerant libraries: any library, given a sub-communicator at initialization, will normally duplicate it in order to avoid collisions with application messages. If both of these register their own error handlers to deal with failures,

the library will replace the error handler instantiated by the application. FT-MPI suggests possible solutions for these problems without conflicting with the current MPI standard. It is clear however, that – in some cases directly related to MPI's current functionality as with the example of error handlers – resolving some of this issues will require clarifications or adaptions to the current mechanisms.

FT-MPI in practice

Analogous to the short MPICH2 case study at the end of the previous chapter, this section gives an example illustrating the practical side of running FT-MPI. As in the previous example, the reference platform will be a cluster of Linux-based machines, connected TCP/IP sockets over Ethernet.

This section will purely give an overview of how to set up FT-MPI and run programs with it. It will not go very deep into the precise mechanisms that make up the FT-MPI runtime: this topic will be covered in more detail in the next chapter. Likewise, this section will also stay clear of discussions on the programming techniques needed to make use of FT-MPI's fault-tolerance features: these will be described in the chapter 5.

Administrative issues

Installing FT-MPI requires downloading and installing the precompiled binaries available from the web – alternatively, a system administrator can also download and compile FT-MPI directly from the source code. As with MPICH2, the binaries have to be made available to all of the different machines in the cluster, either by uploading or through some sort of shared folders.

For administrative purposes, FT-MPI bundles computing nodes into an abstract entity called a virtual machine (VM) (cfr. PVM). In order to be able to run jobs on the cluster, the administrator will first have to add the nodes of the cluster to a VM.

In practice, this comes down to running a daemon (called the *startup-daemon*) on each of the nodes. Analogous to MPICH2, FT-MPI uses these daemons for scalable startup of large MPI jobs. The administrator can add a machine to the VM by bringing up the necessary daemons manually on each node, or by using the FT-MPI console.

The console offers a PVM-like interactive, textual, menu-based user interface for performing a diverse set of VM management tasks. This includes individual menu entries for adding hosts to the VM, removing them from the VM, getting an overview of jobs running on the VM, killing individual processes within the VM etc.. The console can be started from the command line using the command *console*. This will start up the console with a default VM. Usage of the console is mostly self-explanatory. Once the administrator has set up a VM, the user can use it to run MPI jobs.

User issues

A user can now write, compile, link and run MPI programs on the cluster. Writing regular MPI programs is done as presented in the many code examples given in chapter 2. Regular MPI programs will run seamlessly on FT-MPI. MPI programs can be compiled and linked using two scripts called *ftmpicc* and *ftmpif77* for C and Fortran77 respectively.

After compiling and linking, the user can run the program from a master node of his own choice by using the command *ftmpirun*, adding a flag *-np* (compare with the *-n* flag of MPICH2) to denote the number of processes that will run for the job. As with MPICH2, the number of processes need not match the number of nodes in the ring – if there are more, they will wrap around. Adding to this, the user can pass *-msg_mode* and *-comm_mode* flags which specify the respective FT-MPI modes to be used at startup of the job.

When the job finishes, it will send output to *stdout* of the chosen master node.

3.4 Message-level fault tolerance

Most current research on fault tolerance in MPI has focused on the issue of process-level fault tolerance, intrinsically assuming all communication channels to be reliable. This premise is rather optimistic in the context of very large parallel computing systems. A few examples of research being done in the area of message-level fault tolerance do exist however. MPICH builds its parallel runtime environment on top of an entity referred to as the Multi Purpose Daemon (MPD) [56], which provides scalability and a degree of fault-tolerance for internal communications by using a ring topology for some operations and a tree topology for others. Another example of research in this area, currently under development within the context of FT-MPI, specifies a protocol for broad- and multicast type internal communications of a parallel system merging both tree and ring structures into a single communication fabric to circumvent faulty links [55]. The most prolific research project to investigate message-level fault-tolerance is probably LA-MPI [51], and due to the scope of its approach deserves to be expanded upon a little more further.

3.4.1 LA-MPI

Development of LA-MPI was driven by the rationale that most individual components are typically manufactured for small or desktop systems with a relatively high tolerance for faults. When these components are aggregated in to large HPC systems however, system-wide error rates may be too great to successfully complete a long application run [54]. LA-MPI relies on the availability of hardware-level redundancy and end-to-end data checking to provide fault-tolerant data delivery and guaranteed message integrity to MPI applications.

Redundancy of NICs and links has been a regular feature of modern parallel hardware for quite some time now. This theoretically allows software to fail-over from one network device to another when the first is generating too many errors and to reroute messages around faulty links. Sadly, not too many MPI implementations have been making use of these abilities up till now. LA-MPI internally uses the concept of generic “paths” as a hardware-independent abstraction to deal with these capabilities. This allows both for improved performance as well as for fault-tolerance. When all paths between two processes are available, a message can be striped over multiple paths for increased bandwidth. When a path becomes unavailable because

of failure, recovery can be done by re-routing data over (an) other path(s) [53].

At the software level, a combination of retransmission protocols and CRC checking is used to ensure both reliable message delivery and message integrity. The designers of LA-MPI decided not to use TCP because of performance issues: TCP/IP is a rather heavyweight protocol, designed to handle unreliable, inhomogeneous and oversubscribed high-latency networks. In contrast, LA-MPI aims mostly for the cluster market where networks feature – and applications require – very low latency over a range of different network devices (e.g. Myrinet, Quadrics, Gigabit Ethernet, ...). Therefore, LA-MPI uses a lightweight and highly efficient (though more limited) checksum/retransmission protocol of its own. The LA-MPI reliability layer is implemented in user space, with the ability to disable checksumming at run-time for additional performance at the cost of guaranteed data integrity [52].

3.5 Conclusion

After taking a good look at all the options available, we can already state an answer to some parts of our first research question:

- *Concerning MPI implementations: what are the current options w.r.t. fault-tolerance and MPI?*
  First, a prospective user must investigate the kind of fault-tolerance needed for his particular application. There are two kinds of fault-tolerance that can be supported (both if required): (1) process-level fault-tolerance deals with faults occurring at the process level, i.e. processes becoming unavailable in any way (e.g. crashing); (2) message-level fault-tolerance, which is concerned with handling faults occurring at the communication level. Concerning process-level fault-tolerance, there are two major philosophies: (1) transparent fault-tolerance assumes that fault-tolerance is the responsability of MPI itself, that it should run within basic MPI and that the developer / user should not be aware of the availability / absence of fault-tolerance features; (2) application-controlled fault-tolerance requires some additions to MPI, but allows the developer to provide a parallel application with made-to-order fault- tolerance, with minimal overhead – basically, this is the PVM philosophy as applied to MPI.

- *What would be the best approach for the particular case of small research groups working with low-budget hardware.*
  Given that the type of low-cost clusters we are looking at is running on top of TCP, message-level fault tolerance is already covered by our communication substrate. This leaves us only with the need for process-level fault-tolerance. Transparent process- level fault-tolerance has the major advantage that it requires no intervention on the application level; however, it comes with a major, per-processor overhead which actually grows with the size of a job. Given the limited means that we are looking at, said overhead is unacceptable. Hence,

application-controlled fault tolerance becomes pretty much the only valid approach for our purposes...

CHAPTER 4

Extending FT-MPI

After the advances we were able to make in the previous chapter, we are left with only the last part of our first research question left unanswered: *what yet remains to be done in the area of fault-tolerance to make fault-tolerant use of MPI a valid approach?*

Some time into the first year of our research, after a lot discussion about the means at our disposal, we had come to the conclusion that FT-MPI was pretty much the only platform to offer the features that fit the problem we were investigating. However, while further investigating FT-MPI from a practical angle, we found out that it also lacked a number of capabilities that were necessary for our purposes.

Specifically, we were investigating methods for bundling loose collections of resources into a single, relatively cheap and light-weight computing platform. These resources could be expected to be spread over multiple networks, at multiple separate physical locations and in different administrative domains. This is where FT-MPI falls short: it was specifically designed to function on full cluster systems and MPPs – which are situated single locations and are generally covered under single administrative domains.

The resources that we were looking at weren't easily forced into such a straightforward model. Moreover, we didn't feel comfortable about setting up heavily centralized control structures like campus-wide – or even cross-campus – VPNs. One of the solutions that seemed close to what we wanted done was grid technology. With the release of the OGSA [103], a standardized approach had finally been put down for accessing grid infrastructures, encompassing a great number of disparate resources, spread over multiple locations. However, setting up an OGSA-based grid still requires an amount of centralized coordination that went too far for our tastes.

Another solution that we came across during our "study-round" was the Harness meta-computing framework. Although interesting in its own right, it did not quite fit our requirements either. But it did lead us to discover a follow-up project to Harness that had been in the works at Emory University's Distributed Computing Lab (DCL),

under the auspices of Prof. Sunderam. This project had resulted in the creation of the H2O metacomputing framework. H2O seemed an ideal complement to our needs: a light weight infrastructure for building self-organizing, grid-like computing frameworks.

Our interest in the H2O framework led to mutual contacts, and said contacts eventually led to a research visit in the beginning of the year 2005. Our cooperation on H2O and FT-MPI could actually build on earlier research done at the Emory DCL, and led us to identify other issues with FT-MPI that needed resolving. These issues mostly revolved around FT-MPI's name service component.

Specifically, we discovered that FT-MPI employed a name service for retaining state data, which was prone to bottlenecks and suffered from limited scalability when used on "scattered" resources spanning multiple locations and administrative domains. In other words: exactly the kind of resources we were trying to use. Moreover, the name service constituted a single point of failure for the whole of the FT-MPI distributed virtual machine.

Our mutual work in Atlanta would concentrate on a fix for these issues, using the H2O framework to our advantage. Naturally, straightforward access to name services – both from the FT-MPI runtime and H2O – would be crucial a element in our research. With H2O being developed in Java, it was logical for us to seek among the diverse Java APIs to work out a solution. In particular, Java's JNDI API seemed like an interesting starting point – and indeed, it would prove to be the key to our eventual results – results which were, eventually, put down into two Springer / Lecture Notes on Computer Science publications.

The layout of the rest of this chapter will be as follows: 1) in a first section, we elaborate on the H2O meta-computing framework and its qualities, 2) in the following section, we will provide documentation on the JNDI API, and 3) in the final section, we provide a full explanation of the issues with FT-MPI's name service and offer insight on the way in which we combine both H2O and JNDI with original work to tackle these issues. We conclude the chapter with the full content of the two papers which resulted from our work.

4.1 The H2O meta-computing framework

One of the cornerstone technologies we use for bringing FT-MPI to our specific type of target resources is the H2O meta-computing framework. In short, H2O's design goal revolves all about aggregating heterogeneous disparate resources, potentially spread over multiple geographic locations, into a custom, "made-to-order" computing framework. In the following sections, we will be describing the H2O framework: its origins, its basic tenets and some details on its architecture, as well as the way it fits into our research.

4.1.1 Background

Before elaborating on the design goals that underly the H2O meta-computing framework, it is perhaps appropriate to first define what exactly constitutes a "meta-computing framework".

In short, a *meta-computing framework* is a framework for building computing frameworks. Being a framework, it is more than just a set of libraries: it can be composed of run-time components, code-generation tools and pre-generated functionality. Generally, frameworks take an approach which is somewhat contrary to libraries: library use involves including pre-written routines and data structures into a user-created "skeleton". Frameworks, however, generally work the other way around: they offering clear interfaces to pre-defined skeleton functionality, which is then configured to include user-created objects. Most frameworks consist of pre-written components, setting up a set of basic rules, logic and interfaces that allow a user to specify, add to, change and / or customize the expected behavior of the software.

A meta-computing framework, then, provides for such skeleton functionality, implementing a set of run-time components, tools and functionality for a developer to expand and customize into a full computing framework that perfectly serves his needs. The goal is to allow the developer to use major parts of the framework wholesale, developing new functionality only where needed, and to minimize the need for writing new code, while still covering a broad potential of computing needs.

The H2O framework was built upon experience gained from a previous project: the Harness meta-computing framework [102]. Harness was developed around the intent of exploiting the reusability and adaptability aspects of the framework concept. The idea was for multiple computing systems (e.g. MPI, PVM, OpenMP [104], ...) to be implemented on top of a single basis. This basis would provide the ability to "plug in" components as needed. Some of these components would provide a user-end experience, e.g. the user-side APIs necessary for implementing PVM, MPI, ... Others would provide the underlying infrastructure to make the system work at the framework side: communication substrates, redundancy and persistency mechanisms, state management, specific optimizations etc.

The goals of Harness: 1) to minimize the work on new computing frameworks by maximizing code sharing with other, existing or new computing frameworks, 2) to leverage this shared code to minimize long-term upkeep as well, 3) to make new and improved functionality – originally written for one framework – quickly and easily accessible to others, 4) to easily change underlying mechanisms to effect different kinds of tradeoffs without influencing the user-experience (i.e. add redundancy at the cost of performance or vice-versa, optimize for certain architectures when usefull, ...) and 5) to offer a convenient platform for research into computing framework technology, allowing researchers to experiment with new features without having to write a complete computing framework from scratch.

The H2O project was based around these same principles, specifically the "pluggable" component-based approach. However, a few priorities were added. Specifically, pluggability was to be exploited to assist in the goal of *resource sharing*, very

much in the style of grid systems. H20 aims directly for light-weight, easy resource sharing - across administrative boundaries – of geographically separated heterogeneous resources. The most important difference between H2O and most "regular" grid system is that H2O aims to support "lighter", more "ad hoc"-style shared resource platforms. We illustrate this difference, amongst other characteristics of the H2O meta-computing framework, in the following section.

4.1.2 Characteristics of the H2O

To recapitulate: the primary goal for H2O is to facilitate resource sharing by leveraging pluggability. This signifies the need for two important entities: 1) "some thing" to plug "stuff" into, and 2) "some thing" to be plugged. In H2O, the first entity is referred to as *the H2O kernel*. The functionality to be plugged, then, is encapsulated in components referred to as *pluglets*.

Basically, the H2O kernel acts as a universal container for all kinds of pluglets. A single kernel is to be run on every resource that its owner intends to be accessible for sharing with other parties. Pluglets can be added to a H2O kernel by the resource owner, by an internal or external user, or even by a third-party reseller. The requirement, of course, is that the resource owner has full control over the use of his resources: he decides who gets access to the kernel, when and to which degree, as well as how his kernel can be discovered by outside parties. In a similar manner to kernel owners, pluglet owners in their turn will also have full control over the use of their code, of course within the boundaries allowed by the resource owner. A full set of rules can be set to customize and regulate interaction among pluglets, as well as the way that pluglets can be discovered within a kernel.

Users can access a H2O pluglet in a kernel through a straightforward, functional interface. Access to pluglets can either be direct, or through redirection via e.g. a UDDI repository.

All of the above signifies the major difference between H2O and other resource sharing platforms like OGSA-based grids: these typically bundle the responsability for resource and component management with the resource owner, and thus lack the 3-way transparency of the H2O approach. Contrary to the traditional grid model which typically results in rather heavy, strongly organized systems where large blocks of resources are shared as a whole in a pre-determined manner, H2O aims to ease the creation of an ad-hoc resource sharing infrastructure for smaller projects.

Another set of platforms that might be compared to the H2O meta-computing framework are the world wide, user-level computing networks like Distributed.Net and Seti@Home [10]. The most significant departure from this model by H2O is the addition of dynamic coupling, "hot-swappability" and upgradeability without intervention by the resource owner once the latter has given the necessary authorization to the specific project in question. The available authentication and security infrastructure in H2O allows for simplification as well, as these facets don't have to be rewritten for every new type of project.

All in all, the H2O meta-computing framework focusses on...

- being light-weight (both w.r.t. implementation and organizational aspects, very much in a "you get what you pay for" kind of mentality: the component-oriented

approach allows resource and component owners to get exactly the functionality – and degree of complexity – they want, and nothing more)
- being self-organizing and decentralized (by leaving as much as possible to the different participating parties themselves – every side involved sets out the boundaries of what "goes", and within the common ground set out by the sum of all of these boundaries, things "just work")
- retaining minimal state (using a component-based approach for easy setup and (re)configurability – the framework contains only minimum functionality in and of itself – an infrastructure for strong control over authorization issues; everything that concerns the use of the kernel and components among themselves and the user; all other functionality – the "actual computing infrastructure" – is set up by the individual components)
- facilitating resource sharing via components (instead of linking a single use to a single resource, a resource can be combined with multiple kinds of logical functionality, as needed at any given particular point in time without a large reconfiguration overhead, thereby maximizing usefull application of the available potential)

4.1.3 Technical aspects of the H2O

About the use of Java

At the basic level, the framework consists of a number of components built on top of the powerfull Java programming infrastructure, leveraging aspects of: 1) the language (object orientedness, fluent support for aspects like multi-threading and garbage collection), 2) its associated set of coding practices (uniform component formats and event models), 3) its large base of standard libraries (extensive and cross-platform support for essential things like authentication, encryption, general building blocks for implementing security across the board in projects and resource management) and 4) its virtual machine technology (platform-independentness through the existence of many implementations, both free and commercial, supported by multiple vendors on a large range of platforms).

It is important to note here that the choice of this development platform – for the framework itself – in no way limits either access to the framework, or development of components. Communication with the framework and its components can be based on independent standards like Corba, Soap (and related Web Services-based standards), and there has even been some work to make the framework more accessible through OGSA-based protocols [107] – in other words: clients need not be Java-aware.

RMIX

This flexibility is accomplished by means of a framework called RMIX [105]. RMIX is an inter-process communication framework, written as an extension on Java Remote Method Invocation (RMI). Java RMI, part of the standard Java APIs, is an object oriented Remote Procedure Call (RPC) API which allows Java objects to refer to – and call methods of – other Java objects in separate Java VMs, either on the same or on a remote machine. The problem with standard RMI, however, is that it is

limited as to the type and use of RPC protocol. It only supports two communication protocols: the Java-only Java Remote Method Protocol (JRMP) and Corba/IIOP. The use of these is fixed at compile time.

RMIX tackles this problem by allowing the user to "plug" multiple RPC protocols beneath the RMI API. This means that users can resolve RMI operations through the standard RMI protocols, but also through e.g. SOAP/HTTP, RPC, ARPC (supported out-of-the-box) or any other type of communication medium for which the necessary *Protocol Provider* classes are available. RMIX even makes it possible to switch RPC protocols at run-time, while RMI stubs are fully active.

In practice, this means that apart from communication between Java objects in different VMs, users can now also use RMI to communicate between Java objects using RMI at one end, and e.g. web services using SOAP at the other end – all transparently to either the RMIX caller or the web service. Moreover, two RPC actors can start out communication in one default RPC protocol at startup, and then negotiate another protocol at runtime, e.g. for performance purposes.

Another feature of RMIX is that, aside from RPC protocols, it features pluggability for a lot of other things. By generalizing the communication endpoints, communication fabrics other than IP can be used for communication, e.g. it is possible for two services to communicate through JXTA sockets. Moreover, communication can be manipulated through pluggable modules, allowing for transparent insertion of functionality like security (e.g. SSL encryption), authentication (X.509 certificates) or compression. This functionality can be layered, and hence combined, as dictated by the needs of both communicating parties.

Pluglets

Pluglets, in turn, can be written in any combination of computer languages, and either accessed through the Java Native Interface (JNI), or by directly downloading an executable binary to a so-called "chroot-sandbox" and running it on the target resource (within the technical boundaries of the resource itself of course – platform independentness can only apply in a limited fashion here). A specific upload / external file management pluglet has been provided to upload and manage such pluglets remotely in a transparent manner.

Appropriate to the H2O approach, pluglets are not bound to use any of the H2O functionality except what is needed to perform the necessary authentication and valorisation for required services, i.e.: pluglets can, but are not required to use, RMIX or any of H2Os other internal services, and can communicate with other compatible pluglets or outside parties using their own protocols as well (which is exactly what our FT-MPI pluglet happens to do).

Pluglets can be plugged into a H2O kernel by resource providers / kernel owners, clients with properly authorized clients or authorized by third party resellers. Pluglets can access each other as allowed by the respective pluglet owners, choosing either to implement their own authorization/authentication schemes and security, or to use the available H2O infrastructure as we mentioned earlier.

Pluglets can provide any kind of service: from direct services to end-clients (like our FT-MPI pluglet), over specific services that can be shared by many end-user

pluglets (like the proxy-pluglet we describe in our papers), to basic "plumbing" functionality like the file transfer pluglet, or pluglets implementing lookup or coordinating services. Any of these can be strung together in complex patterns as allowed by their respective owners, or the kernel owner(s), to create elaborate scalable, ad-hoc and self-organizing computing systems.

Authentication, authorization and rights management

Authorization and authentication of pluglets, clients and potential third-party resellers is taken care of by a gatekeeper, based on the JAAS (Java Authentication and Authorization Services) framework. JAAS helps the gatekeeper to retain modularity through its native extension mechanism based on the concept of Pluggable Authentication Modules (PAM). For every attempt at access to the H2O kernel, the gatekeeper keeps track of where it originates, whom it is made by, whom it is digitally signed by and/or the time of the attempted access, as appropriate for the kind of access in question.

Using the RMIX framework, any attempts at access can be subject to encryption, certification or any kind of security requested by either the resource/kernel owner and/or the pluglet owner as appropriate. Negotiation can be performed in one protocol while actual transactions can be performed in another, or multiple protocols can be used in parallel, e.g. for performing computing services in one while periodically maintaining authentication issues in another.

Each H2O kernel maintains such an authentication, authorization and security infrastructure of its own, and interaction among pluglets spread over multiple kernels and between such pluglets and clients is subject to the demands of each, leading to the formation of an ad-hoc distributed rights maintenance system, all without a central coordinating infrastructure. A kernel can put constraints on pluglets based on the type of service it offers, the clients that can plug or access it, access level of pluglets and their clients or priorities of the service w.r.t. access to the resources offered by the kernel.

Conclusion

All in all, H2O ends up providing us with some great features for building the sort of ad-hoc computing resources that we are looking for: a low-cost, "left-over resources"-composed parallel computing resource framework. It allows us to access resources on machines in computer labs, library computer rooms etc., spread over multiple buildings, campuses and departments within multiple administrative domains. The framework provides full control over the access to – and maintenace of – the resources to their respective owners, while providing users with a unified, versatile and platform-independent access mechanism. All without the requirement of an explicit centralized control structure.

Due to its provisions for pluggability, it serves as the perfect mechanism to "load" resources with FT-MPI systems in a unified manner. It also enables us to implement a number of extensions to the original FT-MPI infrastructure, to adapt it to our particular needs with a number of extensions. The flexibility in this approach is what we will demonstrate in the two papers concluding this chapter. These needs particularly concern the deployment of FT-MPI on heterogeneous and geographically separated

resources within multiple administrative domains, subject to network bottlenecks and specific points of failure that do not occur in FT-MPI's default environment (cluster systems and MPPs). There is another reason why H2O makes an interesting solution to our challenge: some work has already been done at Emory DCL, using H2O to help FT-MPI cross multiple administrative domains [99] - exactly the kind of functionality that we were looking for.

4.2 The JNDI API

The JNDI API is another crucial part of the solutions presented in the two papers following this section. JNDI – the Java Naming and Directory Interface – is one of many APIs that are part of the standard library for the Java programming language. Specifically, as the acronym would suggest, JNDI is concerned with the ability for any piece of software, written in the Java programming language, to access naming services and directory services.

JNDI is a major component of high-profile Java-based products like the Enterprise JavaBeans framework. It is, however, also practical for more "mundane" seeming tasks like simply looking up an IP address in DNS (the internet Domain Name Service), querying up and editing data in a Microsoft Active Directory or OpenLDAP server, or even for opening a file handle into a filesystem or retrieving a HTML page from the World Wide Web.

In short: JNDI aspires to be a generic API that enables programmers to interact with a potentially unlimited range of naming systems and directory services. Over the coming sections, we will explain how the API fulfils this rather ambitious goal.

4.2.1 Naming and directory services

Names

Before we elaborate on JNDI itself, it would probably be a good idea to properly define what we exactly mean with expressions like "naming service / server" and "directory service / server".

First, let us define what we mean when we refer to a *name*. A name can be any entity – comprised of alphabetic, numeric or even special characters (dots, slashes, ...) – that can be used to indicate a corresponding object. A name can exist as an entity onto itself, indivisible into any usefull content for referencing to other entities. In this case, we speak about an *atomic* name. A simple example of this case would be "Fred", but a UUID or a product identifier could also be considered atomic names. A name can also be the result of combining one or more atomic names into a new name. In this case, we talk about a *compound* name. A straightforward example of this case would be a first- and a surname, e.g. "Bob Jones", but a Java package name like "java.lang" would qualify just as well.

When a set of names – atomic and / or compound – is created, used and combined (from atomic to compound) in a similar manner, using similar rules within a similar context, we say that the names comprise a *namespace*. For example, internet domain

names are compound, being composed from at least two atomic names. The atomic names are read from right to left, and separated by a dot. The first name must always be present in the root repositories for the DNS system (e.g. “com”, “uk”, ...). Internet domain names are always case-insensitive, and must adhere to a number of construction rules (i.e. specific characters are prohibited, like spaces). In other words, all internet domain names share a similar structure and are built using similar rules yielding similar results (e.g. “slashdot.org”, “www.google.com”, “mail.telenet.be”) – hence, we refer to the namespace of internet domain names.

A name can also be the result of combining multiple names – atomic or compound – from different namespaces. In this case, we refer to it as a *composite* name. Good examples of composite names are most WWW Uniform Resource Locators (URLs) which are normally composed of a protocol specifier from the namespace of internet protocols (e.g. “ftp”, “http”, ...), followed by a separator of the form“://”, after which one finds a name from the internet domain namespace, continued with a “/” separator, and potentially concluded by a name from a filesystem namespace (e.g. http://www.w3c.org/standards/xml/index.html).

Naming services

Now, whenever we have a systematic approach, a set of rules, for unilaterally linking names (atomic, compound or composite) to objects – in the broadest sense of the word – we refer to this set as a *naming system*. Crucial to this definition is that a naming system allows for substituting a name with an object (or, in other words, looking up the object through its name), but not the other way around. Neither does it allow a user to look up an object on any other of its properties – whatever it may have – but its name. Whenever a naming system couples a name with an object, we say that the name is *bound* to the object. Whenever a (name, object) couple is added to the naming system, we say that we added a *binding* to the system.

The practical elements of a name system can be implemented in software: a database is set up to contain (name,object) tuples to represent the bindings. Some kind of indexing is applied to make the names available as a key for searching, as required by the naming service definition. Finally, a protocol is published towards those services who are supposed to access the system and use it to store and retrieve bindings. Any specification describing a system like that is a *name service specification*, and an implementation of said specification is a *name* service, or, when the services are made available over the network, a *name server*.

One popular example of such a naming system specification for software implementations is DNS, the internet Domain Name Service. The DNS specification specifies a naming system, protocol and behavior for retrieving IP addresses (the “objects”) of machines based on compound domain names. The DNS specification is implemented by multiple domain name server (DNS server) projects (e.g. Bind). As we mentioned earlier, the DNS naming system is used as part of composite naming schemes in other naming systems as well, e.g. URLs.

The objects returned by a name service implementation can be more complex than a simple string-like object though: a good example here is the name service implemented by most file systems. Again, a naming system (e.g. in Unix-like systems:

case-sensitive alphanumeric strings with a limited number of special characters like '_', each name starting of with '/' and atomic names being separated by '/' as well, read from left to right) is combined with an access protocol (a C API) which returns a file handle for each successfull use. A file handle potentially represents a whole lot of complex data (inodes) that allow the user access to the contents of a file on disk.

Directory services

A *directory service* allows for all the above functionality, i.e. it allows users to add (name, object) bindings to a database, which is searchable by name. However, it allows for more than a regular name service – specifically, it allows for adding *attributes* to the bindings. These attributes can be retrieved through a search by name, but it is also possible to do a search based on attributes directly, retrieving both the object and its name. As we stressed in the above paragraphs, this kind of functionality is explicitly not available in a regular name service. A user can add attributes to a binding, remove them or change them. It it possible for a directory service administrator to impose certain restrictions on which attributes can be added, removed or changed w.r.t. which kinds of bindings in the service. Such a specification is called a *schema*.

A good example of a directory service specification is LDAP, the Lightweight Directory Access Protocol, which is implemented by all kinds of directory server software like OpenLDAP or Microsoft ActiveDirectory. Like name services, LDAP specifies a naming system (a sequence of name specifiers '=' name value couples for each atomic name composing the compound names bound into the service; e.g. "cn=Rosanna Lee, o=Sun, c=US" specifies a customer name [cn] of the organisation [o] Sun, which is registered in the country [c] US). LDAP also specifies an eponymous protocol for adding and removing bindings to this naming system. What makes it different from regular name services is that it also provides for protocol calls to add attributes to bindings, edit these attributes and – most importantly – search on them. For instance, searchable attributes like gender, age, etc. could be added to the binding types referring to people, to query the directory service for all kinds of statistics. Moreover, a schema could be used to make sure that e.g. the gender attribute is only applied to bindings representing actual people.

4.2.2 JNDI support for naming and directory services

Basics

JNDI provides a universal API for accessing all kinds of naming and directory services. This is accomplished by means of a three step process. First, an application retrieves a so-called *context* related to the kind of service it wants to query. This context object gives the application access to the *naming manager* which manages the operations on a high, abstract level. The naming manager, in its turn, passes the calls from the application on to the *Service Provider Interface* (SPI).

The SPI is a compatibility layer, provided by the service provider, to access the name or directory service through its own native protocol (e.g. DNS to access a Bind server, LDAP to interact with a Microsoft OpenDirectory etc.). SPIs are technically the heart of the JNDI, mapping JNDI's abstractions onto the realities

of the underlying service. The Java SDK comes with three service providers "out of the box": one for LDAP, one for the CORBA Common Objects Services name service and one for the Java Remote Method Invocation (RMI) registry. Vendors or third parties can provide an SPI for just about any naming or directory service on the market, and SPIs are available for services as diverse as DNS, the Network Information Service (NIS) or even file systems.

In the context of our research, we have used the packaged LDAP SPI from Sun, and an SPI for the Harness Distributed Name Server (HDNS) [97] which was developed at Emory University. HDNS is a research-based nameserver, originally developed in C for the Harness project, which was later ported to Java at Emory University, specifically for the H2O metacomputing system.

Accessing objects in a name service

The first thing an application will have to do, if it wants to query a service, is to retrieve an *initial context* through a static method which serves as a front-end for a "pluggable" factory method. This method takes a hash map that must be initialized with the proper data to access the appropriate naming or directory service.

The initial context is only one instance of the generic concept of a context, introduced in the previous section, which allows the application to perform all kinds of operations on a service. Specifically, a context offers the application appropriate methods to perform the following operations on naming services: add a binding to the context, remove a binding from the context, list bindings within the context, look up objects within the context through their names and change the name component of a binding within the context (rename). The SPI maps these operations onto the appropriate protocol for the service in question.

When performing a lookup operation through a context, an object is returned to the calling application. This object can be an actual "query endpoint", i.e. an IP address for DNS, or a telephone number when looking up in a telephone directory. It can also be a new context, referred to as a *subcontext* which allows for further querying, and effective "browsing" of the namespace.

For example, when looking for a file called "/etc/rc.d/network" in a Unix filesystem, initial context retrieval would return a context referring to "/", the root directory. From root, we would be able to look up the name "etc", which would return a subcontext linked to the directory "/etc", and a lookup of the name "rc.d" in this subcontext would subsequently return yet another subcontext - linked to the directory "/etc/rc.d". Finally, a lookup operation for the name "network" would return the "query endpoint", a file handle for the file "/etc/rc.d/network" directly. The context for "/" allows an application to directly look up the fully qualified path including all subcontext names directly, or it allows for browsing of the whole filesystem, with each directory corresponding to an appropriate subcontext and each actual file returning as a file handle for further processing. Names can be entered as strings, or by using the JNDI naming API which presents an appropriate interface for performing complex operations with names.

The "query endpoint" objects can be either direct representations of the objects themselves, aliases referring to actual objects in other parts of the namespace or

"references" – data which allows an application to create the object – or a stub to it – from scratch.

Examples for each option: 1) when looking up a fully qualified name in DNS, the result will be an actual IP address object, 2) when looking up a symbolic link in the filesystem, the result will be an alias object referring to the actual file in another part of the filesystem hierarchy and 3) when looking up a HP JetDirect enabled printer in a directory, the object returned could be a reference containing an IP address and a port number, for creating a printer stub to contact the printer on the network; from this stub, the calling application could then print documents, query the status or the document queue or perform other, more complex operations.

Accessing objects in a directory service

Operations on a directory service are performed through a *dircontext* instead of a regular context. Dircontext allows for the same kind of name-based searching available in regular contexts, and "browsing" the directory can be done in a similar manner as well, by returning *subdircontexts* as a result of query operations. The major addition in dircontexts are the methods for attribute handling.

Dircontext provides functionality for adding attributes to a directory object (which might include a subdircontext), for retrieving and modifying said attributes but most importantly: for searching based on attributes. JNDI provides the necessary means for an application to create *search filters* that can be passed to a search action, returning a set of objects as results.

A dircontext also offers support for enforcing schemas, and for tracking events within directory services by means of callbacks – the latter functionality is of major importance in the implementation work we did for our second LNCS paper.

Lastly, for LDAP-based directories, a special package is provided to support LDAP specific functionality – it being one of the most widely deployed protocols for accessing all kinds of directory services.

4.2.3 Summary

Given the above features, the JNDI API provides us with all the functionality to create a generic solution for interacting with the FT-MPI naming service. It also allows for seemless integration with the H2O framework.

4.3 Name service issues in FT-MPI

In the section following this one, we present two papers written in joint cooperation between UA's CoMP research group and Emory DCL. For proper understanding of the latter however, it would be best that we first elaborate on the nature of FT-MPI's name service and the particular issues we found with it. Most importantly, we describe how we combine H2O and the JNDI API with newly developed functionality to provide a solution to these issues.

In the following pages, we will mainly concentrate on those issues for which we did not have sufficient "page real-estate" to describe them fully in the papers proper.

For the other parts, we refer to the actual papers themselves.

4.3.1 The FT-MPI VM and state

FT-MPI enables users to control its runtime system by means of a console application similar to PVM. This console allows users to add and remove resources to and from an FT-MPI VM, to kill processes in a VM and to query the VM status. The FT-MPI runtime takes resources from the VM in a round-robin fashion as requested at the start of a job (by means of a comman-line parameter to *ftmpirun*). The console allows for scheduling a single resource multiple times, allowing the VM a controlled way to concurrently run multiple processes on it. Of course, it also allows the user to schedule resources that will never be used in a (faultless) job. A single resource may belong to multiple VMs, and may be scheduled for multiple jobs (in the same or different VMs). All of these can be tracked through a single console.

Whenever a resource becomes unavailable to the VM, the FT-MPI runtime will pick the next scheduled resource from the list, once more round-robin wise. The console, thus, allows the user / VM administrator to influence scheduling of resources as redundant, ready to take over whenever a single other resource goes down, or to expressly state that some resources should be used double first if there are tasks which are important enough to justify the loss in performance.

The console does not allow users to specify which resource is to be allocated to which job – resources will be allocated in the order they were added to the VM, but when multiple jobs are started in parallel by independent parties, there is no way to guarantee that one or the other job will receive certain particular resources. Neither is it possible to specify, within a job, which resource should be assigned to which process. Also, there is no way to specify which "backup" resource must be used in case another resource fails within a particular job. Finally, none of the above behavior is specified as such by the FT-MPI specification – it is completely implementation dependent.

All of the above functionality requires that an amount of state be retained by multiple parties within the FT-MPI system. On a global level, we need to keep track of: 1) which resources are available within the FT-MPI system, 2) which VMs have been defined, 3) which jobs are running, 4) which resources within the FT-MPI system have been assigned to which jobs and 5) the status of said resources (OK or FAIL). All of this state information is necessary for the console to be able to add and remove resources from the system, and for the VMs to be able to act upon the (un)availability of certain resources, and its effect upon the VM or the jobs that are running within it. Particularly, constant availability of this state data is – obviously – an absolute necessity for proper execution of the recovery process.

This state information must be made available to the individual FT-MPI runtimes on each resource. Some of this data needs to be permanently available to each of the resources: 1) the VMs which the resource is part of, 2) the jobs which the resource is being used for and 3) the other resources within the jobs running on it, and the necessary contact information to reach them. The latter bit is crucial for the functioning of the job within the VM, and can be considered to remain static (not withstanding failures). The first two points are mutable, but should

not be changing too frequently. Another piece of state maintained by the individual resources concerns the state of communications between the different resources. This status data, in turn, will be very mutable. Some state information must only be made available to resources on very special occasions (i.e. failures), like the status of other resources used within their running jobs.

4.3.2 The FT-MPI name service

In other words: in order for an FT-MPI system to function, there is a lot of state which needs to be made available to the resources on different occasions and at different expected frequencies. This state state information must be retained in some kind of manner which allows for global coordination. This is the role of the FT-MPI name service.

The name service allows resources to look up different elements of an FT-MPI system by "name" – either actual names, or other defining features that allow for unique identification. It can be used to look up VM names to find out which resources belong to it, job IDs to return the resources running it etc. – all the information needed by the FT-MPI system to run jobs.

It is not just a name service however – it contains directory service-like elements at well. Specifically, it makes heavy use of a directory service-like callback mechanism to keep resources up to date with changes to the system that affect their proper operation – including failures. But it also contains functionality which does not conceptually belong in a name service at all: for instance, it retains numerical data on which it can do operations like an atomical read and increment (equivalent to e.g. the "i++" operation in C). In other words, instead of acting as a straight name service, the FT-MPI name service stands in as a general, problem-specific state database.

The name service runs as a single daemon, alongside other daemons used by the FT-MPI system: the startup daemons (1 for each resource) and one or more notifier daemons. The startup daemons is used to start up and monitor processes on each resource, while the notifier daemon serves to serialize access to the name service when this would be necessary. The name service keeps track of state for tracking state in both the FT-MPI runtimes (which run within all individual processes) and the daemons (idle, active, down, ...).

One advantage of having the name service as a single, central daemon is that it is very easy to perform all the compound atomic operations on it: it just takes in each operation, performing each one of them atomically in a round-robin fashion. In other words: the name service daemon contains no parallelism.

Even so, some calls to the name service have to be performed in a certain order, without interruption. In order to force an optimal solution to this requirement, FT-MPI was designed in such a way that only one machine can do specific important writes to the names service. In internal FT-MPI terminology, this machine is referred to as the "leader". The leader is "elected" through a "grab the token"-type of challenge, whereby every machine in the system attempts to set a certain parameter and read the results in an atomic operation. The first machine that manages to set the parameter from the old ID to its own (both passed as parameters to the operation)

becomes the leader.

The leader is responsible, then, for properly ordering messages to the service and other name service maintenance tasks when needed, with the help of a notification daemon when it is running. Other resources receive warning about changes in the names service by means of a call-back mechanism. After leader election, each non-leader resource subscribes to the name service by providing a proper call-back handle.

4.3.3 Issues with the FT-MPI name service

Despite its advantages w.r.t. atomicity issues, there is also a downside to having the name service work as a single process.

The most glaring problem with the name service is one of fault-tolerance. With the other interacting parties within the FT-MPI system, there is no problem: 1) the system was specifically written to survive failure of one or more processes / FT-MPI runtimes, so handling these does not pose any hindrances beyond the regular fault-tolerance mechanic, 2) a failure within one or more startup daemons will result in the loss of its associated process (which would happen most of the time anyhow), and the FT-MPI system will lose the ability to start a new process on the given resource (again, mostly not much of a problem since the resource can be expected to be down anyway) – the latter situation can easily be rectified by re-adding the resource from the console when the resource becomes available again; 3) the notifier daemon, at last, is not a critical component for operations – it just takes some performance burden away from the name service by re-ordering some operations before passing them on – moreover, it is actually possible to run multiple of them for redundancy.

Things are not so easy for the name service however: it contains all state for the FT-MPI system, and is absolutely necessary for all other recovery mechanisms to work. In other words: take the name service daemon down, and everything goes down – irrevocably. A ring-based redundant name service has been in the works for quite some time, but has never been released. And we can expect the name service daemon to run on the same kinds of resources that we want to use for our calculations – i.e. mostly lower-reliability systems. In other words, this poses a problem w.r.t. the kind of work we want to do with FT-MPI.

There is another problem though: the name service daemon's central position within the FT-MPI system turns it into a potential choke-point when communication has to cross multiple networks over lower-bandwidth connections (e.g. the Internet). Whenever communication remains within a single administrative domain, this will typically will not pose a problem as speed within such networks will generally be high. We want to cross buildings and, campusses – the more resources we have on another network from the name service daemon, the higher the latency and bandwidth issues involved. While some latency will remain unresolvable, some steps should be taken to minimize the scalability problems introduced by FT-MPI's current architecture.

Techniques

It turns out that both the fault-tolerance and the scalability problem can be resolved through a similar technique. After some intense correspondence on the issues between the Antwerp and Emory teams, we came to the following conclusion: 1) the only

means of fixing the problem is to replace the current name service daemon with something different - something more fault-tolerant and scalable, 2) developing a new fault-tolerant and scalable name service on our own is too far removed from our research domain, 3) moreover, it is questionable wether such a development has any use: there are already quite a few name services out there that provide at least the necessary fault-tolerance features like checkpointing and redundancy – building yet another nameserver of our own would be double and redundant work.

Which brings us to our solution of choice: instead of replacing the name service daemon with a new system of our own, we would replace it with an existing name service. More importantly: we should be able to make the replacement "pluggable" – fully in the spirit of the H2O meta-computing framework. It is important, however, that we want to change as little about the current FT-MPI implementation as possible.

Our approach: instead of having the FT-MPI system signal and request state directly to / from the name service daemon, we have it contact a proxy service on a H2O kernel. This proxy service, a relatively light-weight process, translates the FT-MPI name service protocol into JNDI calls to the real, central name service. Because we use JNDI, this solution allows operators of an FT-MPI system to choose any name service as a back-end, as long as it comes with a proper SPI.

Scalability

Scalability is enhanced by bundling all calls to the name service into a single connection from the H2O kernel to the actual service: concurrent calls from individual processes remain on the local administrative domain, while calls that need to be resolved through the name service are channeled through one channel per administrative domain.

Although we don't go any further than protocol translations for most subsystems, it is possible to enhance scalability even further by using the proxy services as caching entities, only contacting the central name service when strictly necessary. One example of such an optimization can be found in the work we did on streamlining the leader election process [106].

Atomicity

The most important part of the solution pertains to the issue of atomicity that comes up for some compound operations, like leader election (in that particular case: look up a name – compare a value – if equal set, otherwise return false). Under the original architecture, each call could be serialized at the name service daemon, and there would be no concurrency issues.

This situation changes under the new architecture however. Calls from processes to the different proxies will still be serialized – but calls from the proxies to the real name service are not. All of these calls happen in parallel, and must hence be performed concurrently. This introduces problems with the operations that must remain atomical. Hence, some form of technique must be applied to make sure that communications between the proxies and the central name service become subject to some kind of basic transaction management.

After many discussions, we decided to make use of the single atomic feature

provided to us by the JNDI API: JNDI guarantees atomicity of the *bind* operation. This feature, now, can be used to implement a locking scheme. Within the backend, all resources that require atomicity of compound operations can be associated with a *lock object*. Whenever a proxy wants to perform a compound operation on such a (set of) values, it tries to *bind* this object within the nameservice.

If this operation completes successfully, the lock is gained and operations can proceed without fear of other proxies interfering. Whenever the bind returns a failure, the proxy knows that the name service objects in question are being accessed by another proxy server. If the back-end supports event notification, the proxy then subscribes to the *unbind* event notification from the central name service, and tries to grab the the lock again as soon as it becomes free. Otherwise, a retry scheme with gracefull back-off is used to periodically check for availability of the lock.

One problem with this approach is that, due to the unreliability of the actors involved in the transactions (the processes themselves), as well as the communication medium (networks), the locks themselves will necessarily be *unreliable*. This is why a proper mechanism has to be implemented to clear out *stale locks*. Basically, the acting proxy must set and renew a timestamp for his lock object within a set period of time. Periodically, all waiting proxies will check this timestamp to make sure that the current acting proxy is still active. If it isn't, a protocol is used to resolve the stale lock by forcefully taking it from the previous owner.

Below, we present an overview of the steps taken by the proxies to maintain consistency in the backend. We will provide for algorithms both for systems that do and do not provide for event subscription:

Lock acquisition:

1. attempt bind of lock object – success: goto 2 / fail: goto 9
2. set timestamp tracker
3. "do stuff"
4. timeout: goto 5 / done: goto 7
5. update lock with new timestamp – success: goto 2 / fail: goto 6
6. enter ***lock recovery***
7. unbind lock object
8. ***done***
9. enter ***lock queueing***

For the timestamp tracker during lock "maintenance", we use half the lease time. When a timeout occurs, the connection to the backend server is immediately closed. Given sufficiently short socket times (which Java allows to be set by the user), we can thus prevent "late" updates from occurring.

Lock queueing:

1. subscribe to unbind/rebind event for lock object
2. set timestamp tracker
3. wait
4. receive unbind: goto 5 / receive rebind: goto 2 / timeout: goto 8
5. unsubscribe unbind/rebind

6. re-enter ***lock acquisition***
7. look up lock object – (re)new(ed): goto 2 / removed: re-enter ***lock acquisition*** / stale: goto 8
8. enter ***lock recovery***

We provide an algorithm only for name services that support event reporting (which both of the services used in our field tests – OpenLDAP and HDNS – do). The same effect can be accomplished with name services that do not support event tracing, by means of active polling. This will, of course, incur a serious performance overhead – something which cannot be avoided in any active polling scenario...

Lock recovery:

1. rebind lock object
2. wait for 2 * socket timeout
3. look up lock
4. lock acquired? Y: ***done*** / N: re-enter ***lock queueing***

The above procedure depends purely on socket timeouts, and the principle that a rebind will either properly proceed and completely succeed, or time-out and completely fail. The major advantage here is that lock recovery becomes completely decentralized. The disadvantage is one of performance: the full operation will allways take up to three times the socket timeout. This can be somewhat mitigated, once more, by configuring socket timeouts to be sufficiently short, but the basic nature of the problem dictates hard limits here. This was the original approach we used for our locking algorithm in the first PVM/MPI paper.

Another way to approach the issue is to do lock recovery through a single "master proxy". This could be accomplished as follows:

Alternative lock recovery:

1. On entering the FT-MPI system, each proxy binds a proxy object
2. This proxy object is subject to periodic rebinds and serves as a kind of "heartbeat"
3. The proxy object name incorporates an index number, e.g. proxy_1, proxy_2, ...
4. Each proxy, on entering the FT-MPI system, attempts to bind the lowest possible proxy number, incrementing on failures to bind
5. Whenever a proxy has been unable to properly update its heartbeat, it registers a new proxy object, invalidating the old one
6. The lowest ranked proxy is the "master" – the current master always indicates its status as such as part of its status object
7. Each proxy subscribes to heartbeat events on its "preceding" peer
8. Whenever the preceding peer fails its heartbeat, a proxy starts monitoring the "next" preceding proxy
9. Whenever the failing preceding peer is the master proxy, the next proxy becomes the new master
10. The master looks up all existing locks and subscribes to all locking events

11. Each lock is entered into a timed event queue, and updated on each rebind event
12. Whenever a lock goes stale (i.e. it registers as a timeout-event on the event queue), the lock object is unbound by the master
13. Other proxies can now, once more, compete for the lock

The advantage to this approach is that a stale lock can immediately be removed, without the delays involved with the lookup-timeout-rebind-timeout cycle of the first method. This is the approach we eventually settled upon, but which never made it into a publication before this thesis.

A last bit that needs to be taken care of is that of "dangling updates", i.e. updates that have been left in an unfinished state because the acting entity lost contact with the backend. This could happen, for instance, if an atomic operation requires two updates and the connection went down after updating the first (but not the second) value. This problem can be mitigated by using double indirection in the backend, combined with a shadow paging algorithm. Basically, all changes are written to a new version of the information record (the shadow page), until every part of the compound operation has been successfully performed. At that point, the reference to the old record (double indirection) is changed to the new one.

The only drawback to this approach is that it leaves "garbage" data behind in the back-end. This data will have to be cleaned up by a garbage collecting process.

Improvements

The above approach has been shown to work over a number of experiments. The attentive observer will, however, notice, that the procedure is quite a bit more heavy-weight than what is currently implemented. If all resources in the FT-MPI system constantly have to access and work with the name service in this manner, tone would think that it will put a severe delay on FT-MPI operations.

Although complex operations on the independent backend will effectively take more time to be resolved than under the original system, there are a few things that need to be taken into account. First and foremost, of course, we gain something as well: scalability through the introduction of hierarchical forwarding over the slowest links in the system (WAN), and of course any fault-tolerance capabilities inherited from the chosen backend system.

As to the incurred delays: as long as it doesn't concern recovery, all operations to the backend will happen in parallel with any running computational processes, without influencing their respective execution times. While theoretical recovery times could be significantly longer, there is more to recovery than just the name service operations. Except when using relatively advanced techniques to implement asynchronicity between calculation and MPI communication (we refer to the latter part of the next chapter for details on this approach), all processes have to synchronize on recovery – this means that all "current" computations have to finish before recovery can start. More often than not, this should incur heavier delays on recovery than anything at the end of the name service.

Most importantly however, the situation wherein multiple proxies attempt to change the same data at the backend is one that should almost never happen: as we

previously mentioned, FT-MI itself implements a leader election mechanism exactly to forego this kind of behavior in the original name service as well. Under normal circumstances, this is how a running FT-MPI system is supposed to work with the name service:

1. The FT-MPI system elects a leader
2. All non-leader processes subscribe to callbacks from the name service
3. The leader does the major majority of updates to the name service
4. the others receive notification through the respective callbacks

The above implies that "battle of the locks" kind of situations are not going to occur. An FT-MPI system will resolve the major majority of calls to the name service, including those calls that would regularly incur conflicts, through the single leader process. This puts a hard cap on the delay incurred through the proxied naming operations.

Most importantly however, the proxies also allow for performance improvements through local resolution of certain actions and local caching. An example of this capability is presented in the second PVM/MPI paper, where we demonstrate how to locally resolve most FT-MPI's leader election processes between the different proxies and the processes directly, drastically reducing the need to go to the backend for this particular procedure – from once per process to once per proxy. Given more time, a great number of such localized "tweaks" should absolutely minimize the performance disadvantages of the proxy solution, while emphasizing the scalability gain.

4.4 A Pluggable Name Service FT-MPI

This section is a transcript of the paper "Applicability of Generic Naming Services and Fault-Tolerant Metacomputing with FT-MPI" [100]

System clustering has rapidly grown to become one of the most popular approaches to supercomputing. Traditionally, clusters are built within strongly controlled environments, using homogeneous resources within a single administrative domain (AD). However, there is a growing interest in clustering resources that feature high levels of geographical distribution across multiple ADs to perform large scale collaborative computations.

MPI is arguably the most popular approach to programming parallel applications on cluster systems. Projects like MPICH-G2 [88] give response to a rising demand for adapting MPI to higher levels of heterogeneity and geographical distribution than those available in traditional cluster systems. These initiatives have proven to be quite successful at the user level [88, 96, 89]. However, they create a number of issues at the administrative level as resources in different administrative domains must all use common policies and be properly synchronized [94].

Also, these systems lack fault-tolerance. A number of solutions have been developed to address the latter issue [92, 93, 37, 35, 91]. One such solution is FT-MPI [98].

FT-MPI roughly divides fault recovery into two major phases: MPI level recovery and application level recovery. After a failure, FT-MPI makes sure that the MPI environment is correctly restored to a functioning state (automatic MPI-level recovery). From there on, it is up to the application itself to restore it's own state instead of relying on automated but potentially unscalable solutions like global distributed checkpointing. This makes FT-MPI an interesting solution for highly geographically distributed, heterogeneous resources.

However, FT-MPI is currently confined to single ADs. Also, bottlenecks and potential single points of failure (SPoFs) become an issue when deploying it on multiple ADs, connected by less reliable (WAN) networks than those available within a single AD (LAN, specialized infrastructure). One of the points where such problems might occur is the FT-MPI name service (NS). We address this issue, not by providing "yet another NS", but rather by making the NS "pluggable", allowing deployers of FT-MPI to use an existing and generic, "off-the-shelf" NS of their own choice (e.g. ActiveDirectory, OpenLDAP, HDNS[97], ...). Such products often provide built-in fault-tolerance and performance enhancing features, as well as other capabilities that might be of interest to the operator of an FT-MPI system. The goal is to enable an operator to choose an NS that corresponds best to his specific needs and available resources, instead of being bound to a single, custom-made NS as is currently the case. The design includes a hierarchical message delivery mechanism which allows for setting up FT-MPI with only a single connection to the NS per AD. This greatly reduces the network bottlenecks that might occur when massive amounts of machines spread over multiple ADs all have to connect to the NS concurrently.

To address the interoperability issues, we use features of the H2O [90] metacomputing framework. H2O gives us the capability to set up, run and use an FT-MPI environment spanning multiple ADs without the need for system accounts – as long as they are accessible via the H2O framework. H2O allows operators to do this in a centralized, transparent and generic manner, hiding the heterogeneity of the underlying systems.

4.4.1 Design Overview

Basic FT-MPI architecture

FT-MPI offers the user a virtual machine (VM) built around the interactions of the MPI library with three different types of daemons. Roughly speaking: the *MPI library* takes care of the message-passing and other MPI-related issues. The *startup daemons* start up processes on the individual nodes on which they run, and keep a handle for each of those for the duration of the job. The *notifier daemons* are non-critical processes that marshal and manage failure notification.

For the work described in this paper, we focus on the third: the *naming daemon*. Each VM runs exactly one such daemon. The naming daemon provides a NS, which functions as a repository for contact information of nodes in the VM as well as a general data repository. It also plays a crucial role in the recovery mechanism. It is used for setting and determining the currently active nodes within a job or VM during either VM buildup, job startup or job recovery. Currently, the naming daemon constitutes a potential SPoF, as it is highly state-retaining and critical for the

general functioning of the VM. It is also a possible choke-point when communicating between different ADs, as these are most probably connected by slower, higher latency networks (WAN/internet) than available within a single AD (LAN). Finally, the naming daemon does not support features like replication and load balancing. These features would be desirable to improve scalability at very large VM sizes.

Extensions to the FT-MPI architecture

Extended design. As stated in the introduction, we decided not to address these issues with the NS by providing a completely new one. Rather, we enable an operator of an FT-MPI VM to use an NS of his own choice. There is a wide range of "off the shelf" NS systems available, ranging from commercially available solutions by major vendors like Sun or Microsoft to open source alternatives like OpenLDAP or research project like HDNS[97]. These provide a wide range of fault-tolerance and performance features like load distribution, replication, checkpointing etc.

Our intent is to enable the operator to choose an NS that is most appropriate w.r.t. his performance and fault tolerance requirements. The ultimate goal is to move all state-retaining functionality away from FT-MPI and into this pluggable NS, thus using any fault-tolerance features in the "plugged" NS to remove the potential SPoF. In order to accomplish this goal, we expanded on the basic FT-MPI design as follows:

- instead of directly contacting the NS, components of FT-MPI now contact a proxy server which resides on the gateway between the single AD and the "outside world"; this proxy server acts as a "*front-end*" to the real NS, translating the internal FT-MPI protocol calls to a format that is understood by the real, "*back-end*" name service; the front-end does not retain internal state – thus, failures can be handled through simple measures like a trivial replication scheme or a restart
- all nodes on a single AD retain an open connection to the NS front-end for that AD, and each NS front-end retains a single connection with the NS back-end (hierarchical message forwarding)
- the NS front-end is implemented as a H2O "pluglet" making it fully remotely deployable by operators on any machine that runs an H2O kernel

The front-end services all of the nodes within the single AD – posing as the real NS – while retaining only a single connection with the back-end. This hierarchical message forwarding approach significantly cuts down the number of connections that would have to cross multiple ADs. On top of this, it potentially enables us to resolve certain calls to the NS locally at the proxy through caching, without having to pass on every call to the back-end. Thus, this approach drastically reduces the chance of bottlenecks on in-transit calls between multiple ADs, and offers room for improvements to scalability. This also solves issues with AD-specific configuration schemes like network address translation (NAT) for the NS (these issues were already addressed in [99] for computational nodes). With the front-end situated on the gateways, it can take in messages from the nodes at the internal interface using virtual IP addresses, and transfer them over the "outside" interface using real IP addresses, or even through a completely different communication fabric (JXTA, ...).

More importantly, this enables the NS to be FT-MPI "agnostic". All FT-MPI

related logic is encapsulated within the front-end. Communications between proxy and back-end are done through a generic interface (more on this under "approach"), making the NS back-end fully pluggable. Using proxies also enables us to integrate this design change without a recompile of the FT-MPI sources. Basically, it suffices to start up the proxy and an appropriate back-end instead of the original NS, and then run the rest of FT-MPI without further modifications. Last, but not least, using H2O to implement this design enables us to take away any extra overhead in setting up this scheme from the individual AD administrators. Given that the necessary resources run a H2O kernel, operators of an FT-MPI VM can now deploy this complete setup remotely and in a transparent manner. For remote setup, H2O provides the necessary features to manage the whole process from code staging up to setup and running of the whole mechanism.

Approach. For the implementation of this design we opted to use of Java programming language, libraries and related coding and interface standards. This decision was motivated by a) it being used for H2O, of which we wanted to inherit the useful features described above, b) its own inherent qualities for implementing server-side solutions across heterogeneous resources and c) its powerful API. Specifically, we make use of JNDI, the Java Naming and Directory Interface. JNDI provides uniform access to a diverse range of NS solutions, ranging from LDAP (the Lightweight Directory Access Protocol) to DNS (the Domain Name Service). Any provider can make a NS "JNDI-enabled" by implementing a Service Provider Interface (SPI). All of this works in a manner that is fully transparent to the user. Thus, using JNDI allows us to make access to the backend generic towards different NS's. The only backend-specific code which needs to be written concerns naming conventions. These widely differ between different NS implementations. Impact of this issue is minimized by separating this functionality into a very basic pluggable name resolver. Providing an implementation of this for the specific back-end is enough to make the whole scheme work transparently for the rest of the process. A basic resolver for LDAP has been implemented, and others are currently in the pipeline.

Extending upon JNDI. Though it can solve the problems regarding SPoF, decoupling the NS from FT-MPI introduces new issues concerning concurrency. FT-MPI assumes a centralized NS. This NS is basically single-threaded and queues all incoming requests on receipt. Thus, it never poses a problem w.r.t. multiple requests accessing the same data resource. However, the new design we propose has the front-ends running and accessing the back-end in parallel. This introduces the issue of atomicity.

To enable for genericity, JNDI assumes a base level, lowest common denominator approach to accessing a NS. However, the calls in the protocol which FT-MPI uses to access its NS are quite high-level in nature, requiring compound operations like atomic increment, atomic compare and set etc. to be resolved in a single call. FT-MPI is highly dependent upon atomicity of these calls. This results in a necessity to resolve certain protocol calls to the NS through multiple primitives in JNDI, requiring separate lookups and subsequent binds. This makes potential concurrent access to

shared data resources, and potential resulting problems (e.g. race conditions), an important issue.

We have tackled these issues by building an unreliable named locking scheme on top of JNDI. This locking scheme allows us to acquire named locks, thus giving individual front-ends exclusive access to a given shared resource when needed. Locks are given a limited lease time, and must be retained by the current lock owner. If a lock is not properly retained, it grows stale and can be forcefully taken by another contender for the same resource. By using an abstract interface for the implementation, NS providers are able to supply native locking to the user if it is supported in their product, simply by offering their own extension of the interface. For NS systems that do not support native locking, we offer our own implementation of the interface which provides locks using basic primitives of JNDI (natively implemented locking, of course, potentially yields better performance).

This is accomplished as follows. JNDI guarantees atomicity of bind. Therefore, binding a new named object into the NS will either completely succeed, or completely fail, without leaving the NS in an inconsistent state. On top of this, a regular bind will not succeed if the object to be bound is already present in the NS (using the same name) – you'd have to use a *rebind* operation to do that. We can use these features to bind a "lock" object with a given name into the NS. If the bind succeeds, the lock has been acquired and further operations can proceed. If the bind does not succeed, this means that a lock with that name was already bound by another agent. In this case, a repetitive retry scheme with graceful back-off is used to wait until the lock becomes either available (the owner un-binds it), or stale (the owner does not renew the lease in time). If the lock has grown stale, it is forcefully transferred on the next attempt to acquire it.

However, the locking scheme we use is necessarily unreliable, due to the nature of the intervening medium between the front-ends and the back-end (the network). This could potentially lead to inconsistencies in the backend state when a failure occurs or a lock grows stale, somewhere in the middle of the compound update process.

To overcome this issue, we have adopted an approach to the design of our update algorithms which enables for consistent state up to and until the last bind-operation for a given protocol call. Failure in any phase before the final bind can simply be solved by applying a garbage-collection operation to the back-end. Garbage collection can potentially be piggybacked on other operations for performance gain. On top of this, to make certain that no state-changing operations from a previous lock owner come through after a stale lock was forcefully transferred, the state-changing bind is only done when the remaining lease time on the remote lock is higher than the socket time out. In this way, we can be certain that the "final" bind has either succeeded or failed before we lose the lock, as the socket will have timed out and the bind will have returned an error before expiration of the lease. This does require resetting the socket time-out to more reasonable levels than the default, which is trivial in Java.

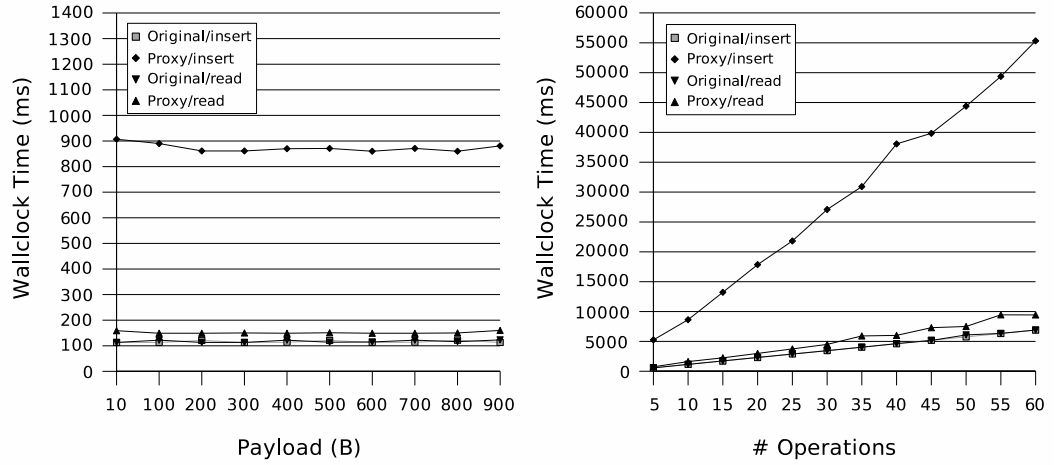


Figure 4.1: Evolution of wall-clock time as the payload per transaction (graph) and the amount of operations increase

4.4.2 Evaluation

To evaluate communication overhead generated by the new design in comparison to the old, we created a proof-of-concept implementation of the front-end. We performed a comparison experiment on two nodes : one in Atlanta (Georgia), USA, the other situated in Antwerp, Belgium. The node in Atlanta is a 2.4 GHz Pentium 4 with 1GB memory running Mandrake Linux 10.0. The node in Antwerp is a 1.90GHz Pentium 4 with 256MB memory running Suse Linux 7. Both nodes are connected to the internet through a broadband internet provider, and communicate through plain, unencrypted TCP sockets. This setup was used in order to simulate the conditions which the design is aimed at: geographically distributed, heterogeneous resources. For the back-end in the new design, we used a standard installation OpenLDAP version 2.1.25 with BDB (Berkeley Data Base) for storage. The node in Atlanta ran the original FT-MPI NS or OpenLDAP depending on the case being tested. The node in Antwerp ran a basic client program in both cases, plus the front-end in the case of the new design.

We ran two experiments: in the first, we first inserted and then read entries with progressively growing payloads (10-900 B, using 100 B steps from 100 to 900 B) and measured wall-clock time for both insertion and read. In the second, we inserted and read batches with a progressively growing amount of equal-sized entries into the NS and again, measured wall-clock time for both cases. These experiments evaluate scalability in terms of transaction size and transaction frequency. They were primarily geared towards testing feasibility, scalability and stability of the new design. Ultimately, we want the new NS to behave as scalable and stable as the original. When it comes to raw performance however, it is to be expected that the original NS will outperform the new design in its current state. This is because a) the original NS was implemented using RAM-based data storage only and b) the original NS uses purely static memory structures as well. Of course, this is not the

case with the OpenLDAP backend. On top of this, OpenLDAP will show a heavy bias towards retrieval operations as it optimizes these over modification/insertion.

The experiment successfully ran to completion leaving the backend in a fully consistent state, proving the feasibility of the new design. The results of our experiment are shown in figure 4.1. As expected, the numbers show a performance advantage for the original NS on insert vs. our new design. Read operations though are almost on an equal level for both cases. The figure also clearly shows that in both cases, the evolution of the measured wall clock time remains linear. This confirms that, despite the need to replace the monolithic features of the original NS by several aggregated operations in the backend (especially on insert), the system remains scalable and delays remain predictable. On top of this, performance remains more than acceptable for regular jobs, as the NS overhead only comes into account at job startup and during fault recovery. Regular job performance will not be hindered by the performance loss on the NS. Even when taking into account the factor 9 performance hit on insert, operation timings remain below 1 second.

4.4.3 Summary

We have discussed issues concerning the deployment of FT-MPI for large scale computations on highly geographically distributed, heterogeneous resources. We have shown that "vanilla" FT-MPI poses some limitations in this area due to the nature of the naming service which is used internally by FT-MPI.

We have worked out a design which address these issues by enabling operators of an FT-MPI setup to a) transparently set up and run an FT-MPI system across multiple administrative domains and b) "plug in" their own name services. This feature is highly desirable as existing off-the-shelf name services often provide numerous features for improved fault tolerance and performance (e.g. distribution, redundancy, replication, checkpointing, journalling, automated management and restart etc.). The proposed design does not require changes to the FT-MPI source. To accomplish all of this, we use the features of the H2O metacomputing framework and leverage the features of JAVA and its component for name service management, JNDI. Also, the staging mechanism employed in the design reduces the nr. of connections to the name service to one per administrative domain, thereby reducing bottlenecks on potentially slower network connections between multiple administrative domains and allowing for local optimization through caching.

4.5 A performance Evaluation of FT-MPI Name Services

This section is a transcript of the paper "FT-MPI, Fault-Tolerant Metacomputing and Generic Name Services: A Case Study" [106]

Over the course of the last ten years, clusters running some implementation of MPI have become some of the most popular supercomputing platforms. Recently, there has been a growing interest in clustering resources that feature extensive geographical distribution across multiple Administrative Domains (ADs). This raises

the issue of fault-tolerance. FT-MPI [98] differs from other solutions to the fault-tolerance problem [93, 37, 91, 92, 35], in that it allows the application itself to restore it's own state, instead of relying on automated – but potentially unscalable – solutions like global distributed checkpointing. This makes it an interesting solution for highly geographically distributed, heterogeneous resources with a need for customized, lightweight recovery mechanisms.

However, FT-MPI is currently confined to single ADs. Also, bottlenecks and potential single points of failure (SPoFs) become an issue when deploying it over slower AD interconnects. One of the critical modules is the FT-MPI name service (NS). We have previously addressed these points [99, 100] by developing a proxy-based solution which allows FT-MPI administrators to use any NS of their own choice (including any fault tolerance features available with it). Further, we use features of the H2O metacomputing framework [90] to span multiple ADs without the need for individual accounts on each system.

Thus we need to focus on improving performance of operations over the proxies. We demonstrate the ability of our approach to transparently and scalably switch between different NSs. We will also present performance test data for the improved algorithms using different backend NSs.

4.5.1 Design Overview

Basic FT-MPI architecture

A running FT-MPI virtual machine (VM) deploys one FT-MPI runtime per node and a number of daemons to assist it in setting up and managing jobs: a *startup-daemon* on each node (semi-critical), one or more *notifier daemons* (non critical), and a single *naming daemon* (figure 1).

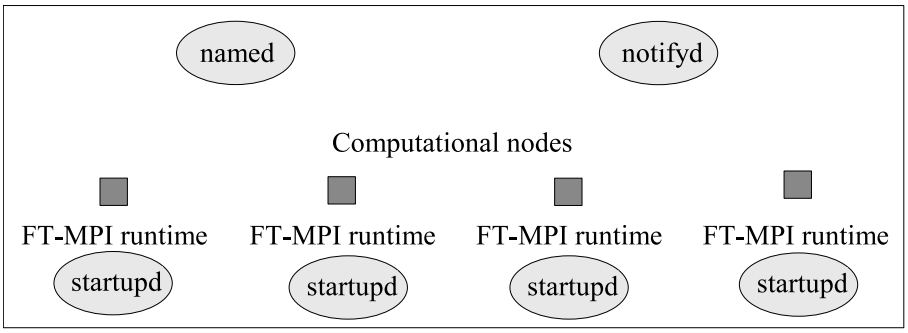


Figure 4.2: a typical running FT-MPI system

Each VM needs exactly one naming daemon (however, a single NS instance can manage multiple VMs). It provides a custom NS and serves a crucial role in VM buildup, job startup and job recovery. Specifically, the FT-MPI runtime uses it to keep records on VM and job membership. To ensure data consistency, editing of records for job and task state in the NS is done by the FT-MPI runtime of single *leader node*. The leader edits these records during the error recovery phase to clean up job and task state. FT-MPI runtimes on other nodes are then notified of the

changes through a system of callbacks. Leaders are *elected* through a custom call in the NS.

We note the following issues with the daemon in the currently available version of FT-MPI: 1) it constitutes a potential SPoF, as it is highly state-retaining and critically important for the general functioning of the VM, 2) it is also a possible choke-point when communicating over slow AD interconnects (this issue was recently addressed [101] and an adapted recovery algorithm should be added to future releases of FT-MPI) and 3) it does not support features like replication and load balancing, which would be desirable to improve scalability at very large VM sizes. We note that many generic name servers currently available do offer these features.

Extensions to the FT-MPI architecture

We use proxies to bridge between the custom FT-MPI NS protocol and any generic NS, enabling an operator of an FT-MPI VM to use a NS of his own choice:

- instead of directly contacting the NS, components of FT-MPI contact a proxy which resides on the gateway between the single AD and the "outside world"; this proxy acts as a "*front-end*" to the real NS, translating FT-MPI protocol calls to a format that is understood by the real, "*back-end*" name service; the front-end does not retain internal state – thus, failures can be handled through simple measures like a trivial replication scheme or a restart
- all nodes on a single AD retain an open connection to the NS front-end for that AD, and each NS front-end retains a single connection with the NS back-end (hierarchical message forwarding)
- the NS front-end is implemented as a H2O "pluglet" making it fully remotely deployable by operators on any machine that runs an H2O kernel

The setup is best illustrated by the example in figure 2.

This approach allows FT-MPI to use one of a wide range of "off the shelf" NSs available. Many of these provide important fault-tolerance and performance features lacking from the current FT-MPI NS (load distribution, replication, checkpointing etc.).

The proxies are implemented in Java. This allowed us to use JNDI, the Java Naming and Directory Interface, which provides uniform access to a diverse set of NSs, ranging from LDAP to DNS. Any provider can make a NS "JNDI-enabled" by implementing a Service Provider Interface (SPI). All interaction with the NS is fully transparent to the user. Thus, using JNDI allows us to make access to the backend generic w.r.t. different NSs.

Concurrency, atomicity and JNDI

FT-MPI assumes a centralized, single-threaded NS which queues all incoming requests on receive. A number of its calls resolve compound operations like increment, compare and set etc. in a single atomic call. However, 1) the new design we propose has front-ends running in parallel and accessing the back-end concurrently and 2) certain single (atomic) calls in the NS have to be resolved through multiple primitives in JNDI, requiring separate lookups and subsequent binds. This introduces the possibility for concurrency problems, e.g. race conditions.

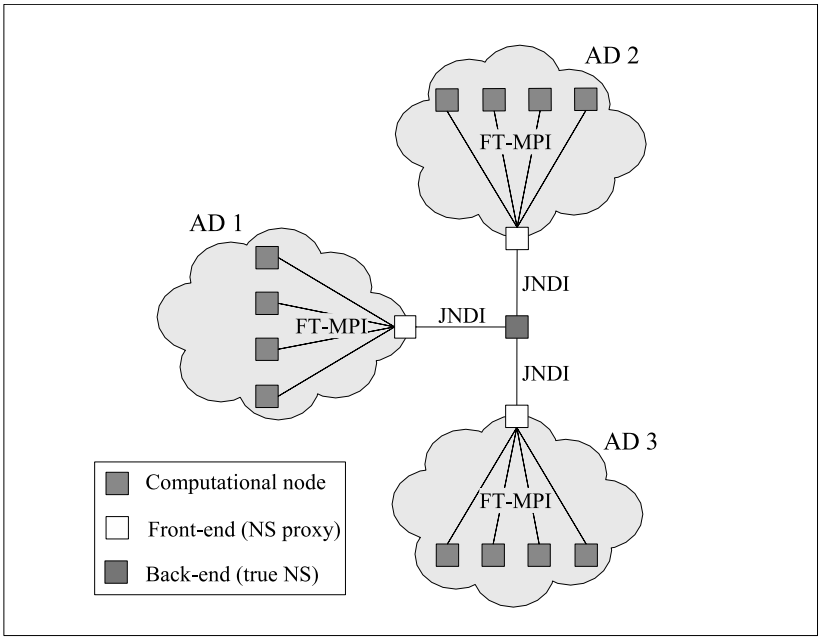


Figure 4.3: An FT-MPI VM using proxies and a generic back-end NS

JNDI primitives We previously discussed a solution through the use of remote unreliable locks, composed from basic JNDI primitives[100]. We will show that it is possible to handle a majority of NS interactions without the use of said locks by exploiting the NSs single-update / multiple-callback architecture. To accomplish our goals, JNDI provides us with the following (relevant) atomic primitives:

- bind(*name,object*): binds *object*, which can contain an arbitrary number of fields to *name*; returns success or failure; appropriate exception is thrown if *name* is already bound
- rebind(*name,object*): replaces the current object bound to *name* by *object*, or acts identical to bind in case *name* hasn't been bound yet; returns success or failure
- lookup(*name*): returns the object bound to *name*; an appropriate exception is thrown in case of problems

JNDI also supports a callback mechanism, enabling us to register and "listen" for updates to the NS, very much like the original FT-MPI NS.

Leader election The most important part of custom functionality to implement is the leader election system. Once a leader gets elected, all editing of records for job and task state is done through him, eliminating the problem of concurrency. The FT-MPI NS implements leader election as a "grab the token" type of contest. The NS provides a custom call of the general form *swap(token,old_leader,contender)* which swaps the ownership of *token* from *old_leader* to *contender* if the current owner of *token* is *old_leader*. In other words: the first contender node to get its message

handled by the NS gets to swap ownership of the token (and become leader) whilst a failure message is returned to the others on all subsequent messages.

An adapted election algorithm Given the primitives available to us, we perform leader election by implementing "grab a token" as "bind a token". For each token which is swapped during the lifetime of the VM, an object is stored in the NS with an *election_count* keeping track of how many swaps have already been performed on it. This token is read during the initialization phase of the proxy and the counter is locally cached for later use. When an election takes place, all contender nodes send the appropriate message to their respective proxies and the following sequence of actions is performed:

1. the proxies each increase the cached leader counter for *token* by one, once – for all contenders who share the same proxy, the contest is resolved locally at the proxy
2. each proxy, for its respective local winner, attempts to bind an object under the name "<*token*>_<counter>" – the node for which the bind succeeds is the *winner node*, all others are *loser nodes*
3. the proxy acting for the winner node rebinds *token* with the new ownership data (triggering a callback) – the winner token becomes the leader and the outcome is relayed back to the new leader node - meanwhile, the proxies handling the calls for the respective loser nodes wait for a callback on a rebind for *token*, eventually relaying the outcome to their respective loser nodes as normal
4. if something goes wrong during the winner' actions in step 3 (non-atomic), this means something is wrong with the proxy, the gateway on which it resides, or its network connection; all of these will get nodes in their respective AD into trouble and register with the FT-MPI runtime as an error – the FT-MPI runtime will then recommence the recovery procedure (including a potential new leader election) as normal

This leaves us only with the problem of compound operations: what if something goes wrong with the leader in the middle of a compound operation? JNDI only allows for atomic operations on a single object at a time. This would lead to inconsistencies in the backend. We deal with this problem by using a single state object which contains pointers to all objects involved in the compound operation. We do not directly write to the objects themselves, but to a copy, keeping the old state intact until all actions in the compound operation have been performed. When ready, a rebind of the index record turns everything over to the new state within a single operation. This may leave spurious objects in the NS, but these can easily be cleaned up by an independent garbage removal process.

Results Advantages of this approach are 1) the ability to drastically reduce dependence on remote locks, enhancing performance by reducing the amount of JNDI calls that would normally be needed, and 2) the ability to do partial local resolution of the leader election process at the proxy, bringing down the amount of effective calls going out to the back end NS. This reduces the potential for choke-points on connections between different ADs and helps spreading load for very large, geographically

dispersed VMs. Also, the number of callbacks is similarly reduced to one per proxy instead of one per node.

4.5.2 Evaluation

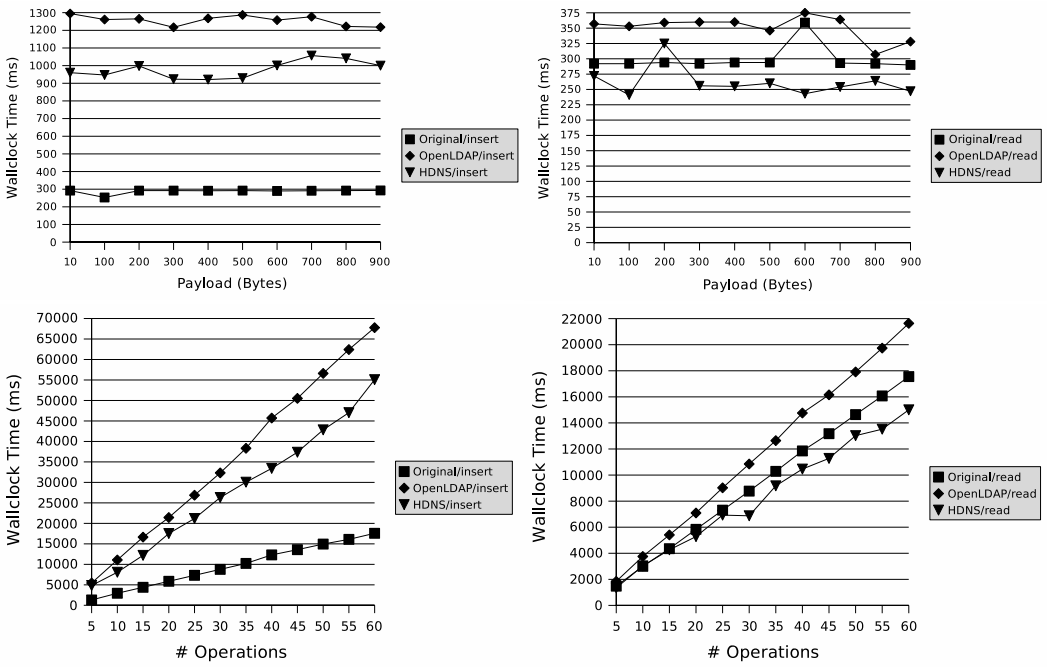


Figure 4.4: Evolution of wall-clock time with increasing payload and # operations(read/write)

Setup and Experiments

To demonstrate the ability of our setup to transparently switch between multiple NS backends,we performed a comparative experiment on two nodes: one in Atlanta (Georgia), USA, the other situated in Antwerp, Belgium. The node in Atlanta is a 4 CPU 2.8 GHz Pentium 4 with 1GB memory running Mandriva Linux 2006. The node in Antwerp is a 1.90GHz Pentium 4 with 256MB memory running Suse Linux 7. This setup was used in order to simulate the conditions which the design is aimed at: geographically distributed, heterogeneous resources. The node in Atlanta ran the original FT-MPI NS, OpenLDAP or HDNS depending on the test case. The node in Antwerp ran a basic client program in both cases, plus the front-end in the case of the new design.

We ran a number of performance tests comparing the original NS with two alternatives: the LDAP-based OpenLDAP using the Berkeley DB, and HDNS [97]. HDNS is a naming service initially developed for the Harness Project[102]. While developing the SPI, a completely new version of HDNS has been designed and implemented. Both of the NSs support distribution and a number of features like fault-tolerance and persistency, which are not available in the original FT-MPI NS.

The following experiments were performed to evaluate scalability in terms of transaction size and frequency: 1) insert and read back entries with progressively growing payloads (10-900 B, using 100 B steps from 100 to 900 B) and 2) insert and read batches with a progressively growing number of equal-sized entries into the NS – measure wall-clock time for both cases. Ultimately, we want the new NS to be as scalable and stable as the original. We tested the performance of insertion and deletion without locks, allowed by the leader election mechanism described above.

Results

We note that all experiments successfully ran to conclusion and left the back-end in a consistent state. From a practical point of view, we noticed that changing between OpenLDAP and HDNS was very easily accomplished. None of these experiments required any kind of code change or recompile of either the original FT-MPI code, or the Java-code for the proxies. A few changes to a configuration file and command-line parameters suffice to change NS back-ends from one experiment to another.

Looking at the figures, we conclude that the ability to do insertion without locking (though still less efficient than the original NS) provides us with a notable performance improvement over previous experiments in which we did use locking [100], the performance gain consistently being around 40%. It should prove interesting to do further research on improving performance of compound insertion operations, bringing figures even closer to those of the original NS. We also note that HDNS performs rather well as a backbone, outperforming OpenLDAP on both insert and read operations in both experiments. On read operations it even succeeds at slightly outperforming the original NS. We are currently investigating possible reasons for this remarkable behavior. Further, both graphs show linear growth on both insert and read for both experiments, proving that our design remains scalable and stable.

4.5.3 Summary

To summarize, we have discussed an algorithm which allows us to implement a leader election system without locking, and note that it is possible to minimize the amount of locking in general. This results in a significant performance gain over previous implementations, both in terms of the amount of JNDI primitives needed and the amount of data transferred over connections between multiple administrative domains. We have presented experimental results which 1) confirm the efficacy of this approach, as well as 2) show the effective ability to transparently change between different back-ends, as demonstrated by our use of both LDAP and HDNS back-ends without significant changes.

4.6 Conclusion

As to the last part of our first research question: *what yet remains to be done in the area of fault-tolerance to make fault-tolerant use of MPI a valid approach?*

In order for MPI to provide a valid platform for the kind of applications we are researching, we need an MPI that provides (1) application-controlled fault-tolerance

which (2) functions and scales across heterogeneous and geographically spread resources, divided among multiple administrative domains.

FT-MPI is currently the only form of MPI that supports application-controlled fault-tolerance. However, (1) it has problems when deployed over multiple administrative domains - luckily, it turns out that we can tackle these problems using the H2O meta-computing framework; and (2) FT-MPI has scalability problems and suffers from single-point-of-failure issues due to the structure of its name service, which serves as a state database.

This issue was eventually tackled through a combination of many technologies, involving (again) the H2O meta-computing framework and Java's JNDI API, and concentrates on enabling the administrator deploying an FT-MPI system with any JNDI-enabled name service (and corresponding fault-tolerance features) of his own choice, enabling for scalability through a hierarchical forwarding mechanism with the potential for caching. Key to – and the meat of – this solution is a distributed coordination mechanism using unreliable locks, based on the limited means available through JNDI and most basic name services.

This opens the way for FT-MPI to become our MPI implementation of choice, and concludes the overview of the work that we did on FT-MPI proper.

Part III

Refactoring for MPI and FT-MPI

Introduction

In this part, we focus on the following research questions:

- With regard to MPI software: what – if any – changes are required to make it fault tolerant, given an MPI implementation that supports it. What does a generic "cookbook" for creating fault-tolerant MPI-based software look like.
- With regard to Fortan legacy code: what needs to be done to integrate the code with C++ code work in a user-friendly manner.

We investigate these questions by applying them to a case study, and tracking its evolution – through a sequence of step-by-step re-factoring stages – from a classical, sequential code-base, to a parallel and fault-tolerant hybrid Fortan 90 / C++ based (FT-)MPI program.

From this development process, we deduce a series of guidelines for tackling each of our two research questions vis-a-vis any other code in a similar situation, i.e. general, non-parallel, non-fault-tolerant Fortran code.

The case study is a quantum nuclear scattering application, developed by F. Arickx and V. S. Vasilevsky at the University of Antwerp [70, 71]. It is a prototype of a smaller-scale research project realized by a small group, with limited funding, yet facing very real resource shortages.

These resource problems could be solved by parallelizing such an application. A number of resources are available for running parallel applications at the University of Antwerp:

- a number of commodity hardware based clusters, composed both from new and "recovered" old hardware
- a large number of machines might be made available "after-hours" - or at a more fine-grained level "between hours" – in student computer labs
- CalcUA, a 2 teraflop parallel supercomputer with 256 (dual processor) nodes, available since march 2005
- preferably, combinations of the above, maximizing computing potential

Both the first, the second and the fourth options involve clustering heterogeneous resources, spread over multiple networks and administrative domains. As mentioned,

some of these resources have served a number of other tasks before being assimilated into a cluster: they are old, potentially prone to breakdown, but also too useful and plenty to ignore.

Being the most prominent tool for parallelization at this moment, MPI is the natural way to go in porting the application. As mentioned in previous chapters however, MPI is currently not fault-tolerant. This is an absolute need when taking into account heterogeneity, older and physically spread resources, which brings FT-MPI into the picture.

We begin the chapter with a short introduction into the case study. Then, we elaborate on some of the preparatory steps in the refactoring process of the code, to lay a clear foundation for the actual research work to follow. Then, we will first focus on finding the answer to our third research question, before concentrating on answering the second one in the latter part of the chapter. It is more natural to first port part of the code to C++ and integrate it with the legacy-code and then refactor the code-base to be fault-tolerant, as greater ease in maintenance and further refactoring is one of our main reasons to start working with C++ in the first place.

CHAPTER 5

A Quantum Nuclear Scattering Application

5.1 Three-cluster Model for ^5H

Let us first, for the sake of greater clarity, offer some background knowledge on the basic functionality of the legacy code itself.

The ^5H nucleus has a large neutron excess and lies beyond the neutron drip line. It has, in the last five years, been the object of quite a few experimental investigations aimed at finding clear evidence of the existence of resonance structures in ^5H.

Different theoretical models and methods have been used to calculate the energy and width of the resonances. We focus on an application of the microscopic three-cluster model [70] formulated in the context of the Modified J-Matrix method described in [71], and model ^5H with the $^3H + n + n$ three-cluster configuration.

The three-cluster wave function can be written as

$$\Psi_{SL;JM} = \widehat{\mathcal{A}} \left\{ \left[\Phi_1 (t) \, \Phi_2 (n) \, \Phi_2 (n) \right]_S \phi_L (\mathbf{q}_1, \mathbf{q}_2) \right\}_{JM} \tag{5.1}$$

where $\Phi_1 (t)$ is an antisymmetric shell-model wave function describing the internal structure of ^3H with three nucleons in the s-shell. The neutron wave function $\Phi_2 (n)$ only includes spin and isospin variables of the neutron. $\widehat{\mathcal{A}}$ stands for the overall antisymmetrization operator. The relative behavior of clusters can be described by two sets of Jacobi coordinates as shown in Figure 5.1.

The inter-cluster wave function $\phi_L (\mathbf{q}_1, \mathbf{q}_2)$ of relative three-cluster motion is to be determined by solving the Schrï¿œinger equation with the appropriate boundary conditions. We therefore expand the wave function $\phi_L (\mathbf{q}_1, \mathbf{q}_2)$ onto a Hyperspherical Harmonic basis:

$$\phi_L (\mathbf{q}_1, \mathbf{q}_2) \quad = \quad \sum_{l_1, l_2} \phi_{l_1, l_2; L} (\rho, \theta) \left\{ Y_{l_1} (\mathbf{q}_1) \, Y_{l_2} (\mathbf{q}_2) \right\}_{LM}$$

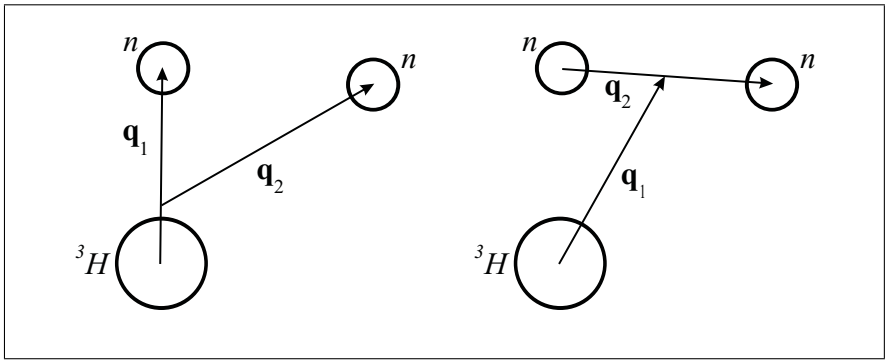


Figure 5.1: Possible choices of Jacobi coordinates*Time To for the* $^3H + n + n$ *configuration in* ^5H.

$$= \sum_{K,l_1,l_2} \phi_{K;l_1,l_2;L}(\rho)\, \chi_K^{(l_1,l_2)}(\theta)\, \{Y_{l_1}(\mathbf{q}_1)\, Y_{l_2}(\mathbf{q}_2)\}_{LM} \qquad (5.2)$$

Hypermomentum K and partial angular momenta l_1 (along $\mathbf{q}_1$) and l_2 (along $\mathbf{q}_2$) define the three-cluster geometry, and characterize the different scattering channels. These three quantum numbers are collectively denoted as $c = \{K; l_1, l_2\}$. The hyperradial wave function is expanded onto the basis of the 6-dimensional radial oscillator:

$$\phi_{c;L}(\rho) = \sum_{n_\rho} C_{n_\rho}^{(c;L)} \Phi_{n_\rho}^{(K)}(\rho) \qquad (5.3)$$

It should be noted here that all spectroscopic calculations – bound state energies, phase shifts, cross sections etc. – are independent of the specific choice of Jacobi coordinate system (see Figure 5.1) if one uses a complete set of Hyperspherical Harmonics for given momentum K (i.e. all possible values of partial angular momenta compatible with this K).

We solve the Schrödinger equation by substituting (5.1) as an ansatz with (5.2), (5.3) and obtain a matrix equation in the expansion coefficients $C_{n_\rho}^{(c;L)}$ (see [70]). Because of the oscillator expansion, we can obtain the solution through the Modified J-Matrix approach [71]. This is an implementation of the Resonating Group Method in function space. As in coordinate space, we distinguish an interaction and an asymptotic region, but now in the space of expansion coefficients. The latter are split in two subsets: the first set represents the three-cluster wave function in the internal region and is determined by solving the Schrï¿œinger matrix equation; the second set is connected with the asymptotic form of the wave functions and has to represent the appropriate boundary conditions. The asymptotic set for a scattering boundary condition is obtained from the solution of reference hamiltonians describing three non-interacting clusters. These hamiltonians are defined in [70]. There will be N_c (the number of channels) independent solutions, with an asymptotic behavior for large hyperradius ρ

$$\Psi_c^{(c_0)} \quad \rightarrow \quad \Phi_1(A_1)\, \Phi_2(A_2)\, \Phi_3(A_3) \sum_c f_c^{(c_0)}(\rho)\, \chi_c(\Omega_5) \qquad (5.4)$$

where c_0 refers to the incoming channel, and f_c denotes the asymptotic expansion coefficients, defined by

$$f_c^{(c_0)}(\rho) \to \delta_{c_0,c}\psi_c^{(-)}(k\rho) - S_{c_0,c}\psi_c^{(+)}(k\rho) \tag{5.5}$$

Here $\psi_c^{(-)}(k\rho)$ ($\psi_c^{(+)}(k\rho)$) is the incoming (outgoing) channel wave function and $S_{c_0,c}$ is the S-matrix describing the transition from the initial channel c_0 to the final channel c.

The resonance parameters can be deduced from the eigenphase shifts, obtained by diagonalizing the S-matrix. In this eigenchannel representation one has (α enumerates the uncoupled eigenchannels)

$$S_\alpha = \exp\{2i\delta_\alpha\}, \qquad \alpha = 1, 2, \ldots N_c$$

The relation between the original $\left\|S_{c,c'}\right\|$ and diagonal $\|S_\alpha\|$ forms of the S- matrix is

$$S_{c,c'} = \sum_\alpha U_\alpha^c S_\alpha U_\alpha^{c'}$$

with $\|U_\alpha^c\|$ an orthogonal matrix. The extraction of resonance position and width is done in the traditional way by

$$\left.\frac{d^2\delta_\alpha}{dE^2}\right|_{E=E_\alpha} = 0, \quad \Gamma = 2\left.\left(\frac{d\delta_\alpha}{dE}\right)^{-1}\right|_{E=E_\alpha} \tag{5.6}$$

The problem to be solved is thus essentially twofold: (1) set up the matrix equation by substituting (5.1), (5.2) and (5.3) for the solution of the schrï¿œinger equation, i.e. calculate the energy matrix, and (2) solve this equation subject to the appropriate boundary conditions. A major advantage of the current approach is that an explicit representation of the wave function is obtained. This allows for a detailed analysis of the resonance wave functions, as well as for the possibility to calculate additional physical quantities at the resonance energies.

In this contribution we will focus on resolving the first problem in a distributed way for an extensive basis.

The energy matrix to be determined can be denoted by ($c = \{K; l_1, l_2\}$)

$$\left\langle c_i; L, n_i \left| \hat{H} \right| c_j; L, n_j \right\rangle = \left\langle K_i, l_i, n_i \left| \hat{H} \right| K_j, l_j, n_j \right\rangle \tag{5.7}$$

where $\hat{H}$ is the Hamiltonian, or energy, operator, and i and j distinguish basis states; the right hand side in (5.7) simplifies the notation, omitting L as an overall constant, and replacing the combination $(l_1 l_2)_i$ by l_i

The theory to obtain (5.7) [70] is well beyond the scope of this thesis, but it can be broken down to

$$\left\langle \left\langle K_i, l_i \left| \hat{H} \right| K_j, l_j \right\rangle \right\rangle \tag{5.8}$$

$$= \sum_t \sum_{l_r} \sum_{l_s} R(K_i, l_i, l_r, t) \left\langle \left\langle K_i, l_r \left| \hat{H} \right| K_j, l_s \right\rangle \right\rangle_t R(K_j, l_s, l_j, t) \tag{5.9}$$

where $\langle\langle\rangle\rangle$ stands for a matrix over all n_i, n_j indices. The R factors are so-called Raynal-Revai coefficients as discussed in [70], and the nature and range of index t depends on the nucleus (^5H) and its cluster decomposition ($^3H + n + n$).

The granularity of the problem is clear from (5.9), and reduces the problem to a fork and join algorithm by calculating all independent $\left\langle\left\langle K_i, l_r \left| \hat{H} \right| K_j, l_s \right\rangle\right\rangle_t$ matrices for fixed K_i, K_j and all allowed combinations l_r, l_s, t (the fork), followed by a summation to obtain (5.9) (the join).

5.2 The legacy application

5.2.1 Introduction

Before we get to the "meat" of our research, it is opportune to illustrate the refactoring process of the legacy code from the very beginning. Even though these steps do not directly contribute to answering our remaining two research questions, illustrating them in proper detail here will make it easier to follow up the evolution of the legacy code in its later stages.

The very first version of the code-base for the ^5H calculations had been written by V. S. Vasilevsky and F. Arickx some time before the current research. This code has been written and compiled for deployment with the Intel Fortran 77 (F77) compiler.

F77 however lacks basic facilities for dynamic memory management and pointers. The lack of pointers implies that F77 compilers do not have to deal with aliasing problems, allowing for some useful low-level optimizations. The downside, however, is that some crutches have to be used to deal with cases where no information on memory usage is available at compile time.

This problem was prominent with the first version of the ^5H code-base as the size of the $\left\langle\left\langle K_i, l_r \left| \hat{H} \right| K_j, l_s \right\rangle\right\rangle_t$ matrices changes with different values for K and L (the higher the values, the more memory is needed).

There are generally two ways of dealing with this problem. One is to use cross-language calls to a C subroutine to do the dynamic allocation (as heap-allocation does not pose any problems in C). Given a number of relatively simple measures, exchanging data between C and F77 code is not too big a problem. We will come back to this issue and its details in section Section 6.2.3 – p. 109. The other solution is to write data to file when available memory is full and to retrieve the data back from file when it becomes necessary for further computation. This was the approach used by the original code-base. Of course, this leads to a huge performance overhead when large amounts of read/write operations occur.

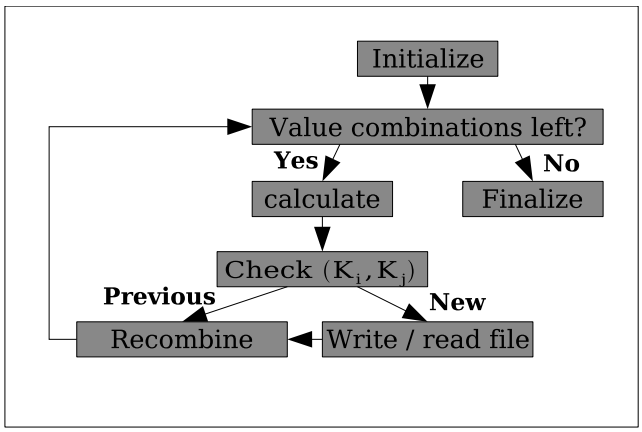
As this performance overhead became unacceptable even for low values of K and L, and cross-calling to C for this kind of purpose is generally seen as a "hack", it was decided to port the code to Fortran 90 (F90). F90 does support dynamic memory allocation and requires the programmer to use a number of compiler hints to minimize the performance overhead from pointer aliasing. At this point, the opportunity was also taken to make minor adaptions to the code in order for it to compile both

on the Intel and the GNU compiler version 4, which since recently supports F90 as well. This new code-base handled memory allocation issues through the use of F90's dynamic arrays feature (we will delve deeper into the topic of F90 dynamic memory management in later parts of this text, specifically in Section 6.2.4.2 – p. 118). Because of the potential size of the individual matrices, it was still decided to retain to-and-from disk swapping for the whole $\left\langle\left\langle K_i, l_r \left| \hat{H} \right| K_j, l_s \right\rangle\right\rangle_t$ matrices when changing between calculations for different K_i, K_j, l_1- l_2 combinations and t values.

5.2.2 Design

This is where the code was handed over for re-factoring in order to add parallelism at the level of individual matrix / matrix element computation. At this point in time, the code still consisted of a single, more or less monolithic block of F90 code with a single function call for computation of the individual matrix elements.

From a high-level point of view, the original software goes through a number of phases as illustrated in figure 5.2.



After initialization, the software goes into a main loop consisting of 1) computation, 2) potential write to (read back from) disk of the results matrix for different K_iK_j values and 3) recombination.

Figure 5.2: High-level overview of the original F90 code-base

Step-by-step:

1. Initialization phase: the program reads all necessary initialization data from a number of files, all necessary memory for result matrices (1 for each combination of K_i and K_j) is allocated through use of the dynamic arrays, all the

results matrix files for different (K_i, K_j) pairs are created, etc.

2. Enter main loop: this is in fact a series of embedded loops, each of which runs through all possible values for the following parameters: K_i, the pair (l_1, l_2) for this value of K_i, K_j, the pair (l_1, l_2) for this value of $K_j$and the range of t-indices.
   (a) Calculation: each combination of the above parameters is used as input for the calculation function; this function returns matrix $\left\langle\left\langle K_i, l_r \left| \hat{H} \right| K_j, l_s \right\rangle\right\rangle$
   (b) Check the current values for the pair (K_i, K_j) – if the current pair (K_i, K_j) differs from the previous iteration:
      i. write the current values for the $\left\langle\left\langle K_i, l_i \left| \hat{H} \right| K_j, l_j \right\rangle\right\rangle$ matrix to the appropriate file for the previous pair (K_i, K_j)
      ii. read the stored values for the $\left\langle\left\langle K_i, l_i \left| \hat{H} \right| K_j, l_j \right\rangle\right\rangle$ matrix for the new (K_i, K_j) pair from file into memory
   (c) Recombine: the data from the matrix $\left\langle\left\langle K_i, l_r \left| \hat{H} \right| K_j, l_s \right\rangle\right\rangle$ is recombined into the $\left\langle\left\langle K_i, l_i \left| \hat{H} \right| K_j, l_j \right\rangle\right\rangle$ matrix.
   (d) Repeat this process until all possible values for the 7 input parameters to the calculate function have been generated (at least the 7 parameters that actually change in value during the runtime of the process - all other input parameters remain at a fixed value as read during the initialization phase).

3. Finalization phase: delete all allocated memory from the initialization phase. Write out the currently open "total" results matrix to file. Write basic calculation data to an output file. Add debug data when required (based upon the actual contents of the combined results files).

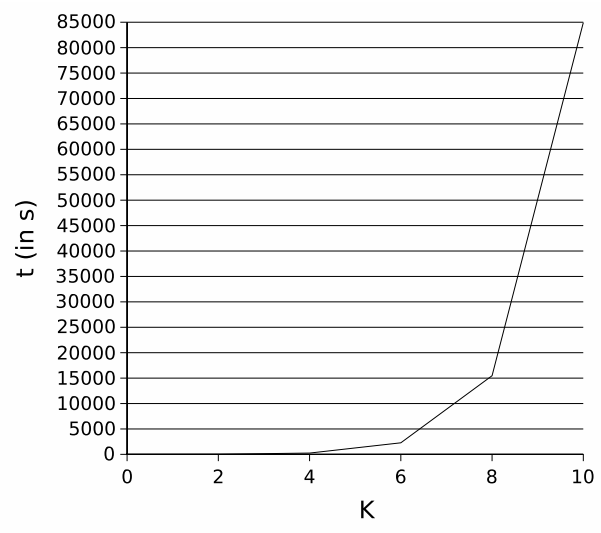
5.2.3 Performance

In order to demonstrate that it is infeasible to hold on to a sequential application of this kind for any reasonably high K number, we discuss results for a calculation with $L = 0$, $K = 0, 2, 4, .., 10$, a range of $l_1 = l_2 = 0, 1, .., K/2$ values, and 45 t values. All of the computational code was compiled with the GNU Fortran 90 compiler version 4.0.2.

All timings were taken on an Intel Pentium 4 machine 1.7 GHz with 512 MB RAM and hyper threading enabled. As can be seen from figure 5.3, computation times for K values lower than 8 are quite reasonable. Computation for $K = 10$ however already takes about 24 hours. Obviously, it becomes unreasonable to attempt computation at any K numbers higher than this.

5.2.4 Conclusion

The need for parallelization in this case (if we want to do calculations for any reasonably high values for K or L) is self-evident. Since we already decided to parallelize



Total execution time t increases rapidly as K rises. At a K value of 12, it is estimated that a single node will require up to 6 days to complete the computation.

Figure 5.3: Evolution of t (in s) *for* $L = 0$ and $K = 0, 2, 4, .., 10$.

using MPI, the next question becomes: how to go about porting the application to MPI.

CHAPTER 6

Re-factoring for MPI

6.1 Re-engineering the legacy code

6.1.1 Basic parallelization

As 1) MPI supports a call interface for F77, and 2) F77 calls remain perfectly compatible with F90, we decided that the fastest and most convenient way to add parallelism in the primary stage was to do this first re-factoring step in pure Fortran 90. We decided that – apart from the changes discussed in section 6.1.2 – we should retain the Fortran code as much as possible in its original state, only adding MPI calls where necessary for the parallel computation.

This leads us to a design in which

- only the computations for the partial results matrices are being distributed.
- a single process takes care of both parameter set generation and partial-to-total result matrix recombination (note that none of these activities are computationally intensive)
- the calculation work is being done by multiple additional processes (which, of course, do require high amounts of computational power).

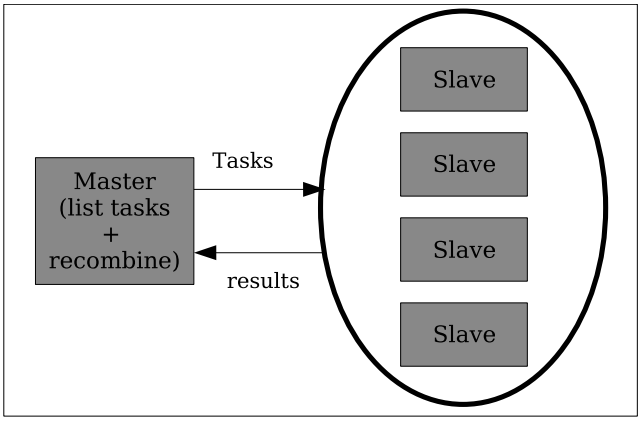
In other words, we get a classical master-slave type of application, as illustrated in figure 6.1.

6.1.2 Other re-factoring choices

During this phase, we also decided to re-engineer the code in two other, more low-level areas, which do not directly relate to the issue of parallelization.

Splitting up into functions

First, in order to add greater modularity, re-usability and readability, we decided to do something about the monolithic nature of the original code-base. Thus, the



The ^5H calculation as a master-slave parallel application. The master generates input data sets for the calculation function which runs in the slaves, passing them on as messages. The partial result matrices generated by the slaves are sent back as a return message, with the master recombining them into their respective total result matrices as they arrive.

Figure 6.1: The ^5H calculation as a parallel process

code was separated into a limited number of distinct functions, each of which is called from the main loop. This change is primarily useful with an eye towards code maintenance and further re-factoring steps, but will turn out to have some positive repercussions for the next re-factoring step as well (see section 6.2).

Practically, we decided to write the split-off functions in a way which 1) strictly enforces pass-by-value semantics for the function input parameters, while 2) passing back all output by means of the function return value. Our primary concerns in using a strictly separated input / output mechanism were mostly:

1. the issue of the parameter list size for the calculation and recombination functions, which were growing unpractically large
2. the fact that, as a result of this, it became exceedingly hard to make it clear to users of the function which parameters were
   - input – data used, but not changed by the function, commonly abbreviated as IN
   - output – new data produced by the function, equivalently referred to as OUT
   - both of the above – data used by the function for initialization, but also altered by it during processing – which we will refer to as INOUT
3. the lack of parameter list checking in Fortran function calls (unlike e.g. C++, Fortran does not use name mangling during the compile/link process for discovering parameter list errors – we will come back to this issue in section 6.2.4)

Strict separation of input and output semantics helps greatly in clearing up these issues. The separation is accomplished in practice by:

1. passing all IN and INOUT parameters as member fields of an input structure which all input values are assigned to before the function call (thus enforcing call-by-value semantics)
2. assigning the values within the input structure to local variables on entering the function itself (making it effectively possible to transfer the original code into the new function *verbatim*)
3. analogously, assigning all OUT and INOUT values to the appropriate fields in a single output structure after the function has completed, and passing it back to the caller by means of the function return value (thus strictly separating input and output in the call semantics)
4. and finally, assigning the values within the fields of the output structure to their respective variables in the calling code, after the function has successfully returned (thus allowing for a clear indication of which variables are OUT, or INOUT)

A code snippet illustrating the function calling mechanic is shown in figure 6.2.

This approach is counter to classical coding practices in Fortran, as parameters in Fortran functions are all passed by reference. The classical approach to writing Fortran functions leads to a coding style which 1) promotes doing both input and output of a function by means of the parameters, while 2) using the return value only for signaling success or failure or 3) simply not providing a return value at all (i.e. using a subroutine). This style of programming used to be popular in C as well, but has been dropped from more recent computer languages like C++, Java and others. (These languages use an exception-based mechanism to handle success / failure apart from the input / output parameter mechanism, and support declaring parameters as constants within the context of the function in order to further allow separating input vs. output parameters).

However, the suggested changes to the function calling style do effectively help out in dealing with the issues mentioned earlier, and as a bonus, they will prove to be very useful over the course of the following re-factoring steps, as we will generally get too see in section 6.2, and specifically in 6.2.3.1.

Eliminating dynamic arrays

A final alteration to the original code-base during this stage, concerns the use of dynamic memory. Basically, F90 offers two ways for dealing with dynamic memory: dynamic arrays and pointers. Dynamic arrays offer basic functionality for dealing with array-like structures of which the size cannot be determined at the compile time, while F90 pointers function analogously to pointers types in other programming languages (the topic of F90 dynamic memory management will be further fleshed out in section 6.2.4.2). As we mentioned earlier, in the introductory part of this chapter, the original F90 code-base used dynamic arrays for its heap memory management.

The problem with dynamic arrays is that their use translates badly to the new function call approach described earlier. Specifically, it is not possible to directly pass a dynamic array as a member of a struct without performing a whole array copy

```
!!!!!!!!!!!!!!!!!!!!!!!!
! DEFINE IN/OUTPUT TYPES
!!!!!!!!!!!!!!!!!!!!!!!!

TYPE SANITY_CHECK_INPUT
     INTEGER K_min_abs, K_max, K_j_max, K_i_min, K_j_min, K_i_max
END TYPE SANITY_CHECK_INPUT

TYPE SANITY_CHECK_OUTPUT
     INTEGER K_j_max, K_i_min, K_j_min, K_i_max
END TYPE SANITY_CHECK_OUTPUT

!!!!!!!!!!!!!!!!!!!!!!!!
! DEFINE LOCAL VARIABLES
!!!!!!!!!!!!!!!!!!!!!!!!

INTEGER K_min_abs, K_max, K_j_max, K_i_min, K_j_min, K_i_max
INTEGER K_j_max, K_i_min, K_j_min, K_i_max

!!!!!!!!!!!!!!!
! CALL FUNCTION
!!!!!!!!!!!!!!!

TYPE(SANITY_CHECK_INPUT) input
TYPE(SANITY_CHECK_OUTPUT) output

!ASSIGN INPUTS

input%K_min_abs = K_min_abs
input%K_max = K_max
input%K_j_max = K_j_max
input%K_i_min = K_i_min
input%K_j_min = K_j_min
input%K_i_max = K_i_max

output = SANITY_CHECK(input)

!ASSIGN OUPUTS

K_j_max = output%K_j_max
K_i_min = output%K_i_min
K_j_min = output%K_j_min
K_i_max = output%K_i_max
```

Adapted function calling in Fortran: instead of doing both input and output through the pass-by-reference function parameters, all input data is passed as set of member fields of the input parameter; likewise, output data is assigned to the fields of a structure which is passed back to the caller as a return value.

Figure 6.2: Code snippet: adapted function call semantics in Fortran

(the only way to pass a dynamic array by reference is directly through a function parameter). What we need in the input and output structs, of course, is a reference

to the array, not a whole copy of it. The problem is further illustrated by the code example in figure 6.3.

```
!!!!!!!!!!!!!
! DEFINE DATA
!!!!!!!!!!!!!

! DYNAMIC ARRAY
INTEGER, DIMENSION(:), ALLOCATABLE, TARGET :: array1
INTEGER, DIMENSION(:), ALLOCATABLE :: array2

! POINTER
INTEGER, DIMENSION(:), POINTER :: ptr1, ptr2

!!!!!!!!!!!!!!!
! EXAMPLE CODE
!!!!!!!!!!!!!!!

ALLOCATE(array1(size)) ! Allocate space on heap for dynamic array
array2 = array1        ! *ERROR*
ALLOCATE(array2(size)  ! Allocate other array
array2 = array1        ! Array copy!!!

ALLOCATE(ptr1(size)    ! Allocate space on heap for ptr to array
ptr2 => ptr1           ! Pointer assign. (No copy!)
ptr1 => array1         ! Also possible because array1 is TARGET.
```

Dynamic arrays vs. pointers in F90. While dynamic arrays are passed by reference when using them as parameters to a function, assigning one to a dynamic array structure member actually results in copying the whole array. Using our adapted approach to function calling in F90, this would actually result in two full array copies, which is certainly not the intention...

Figure 6.3: Code snippet: Dynamic arrays vs. pointers in F90

This issue is easily settled by using Fortran 90 pointers instead of dynamic arrays, as fortunately, they are almost 100% fully compatible with dynamic arrays in practical use.

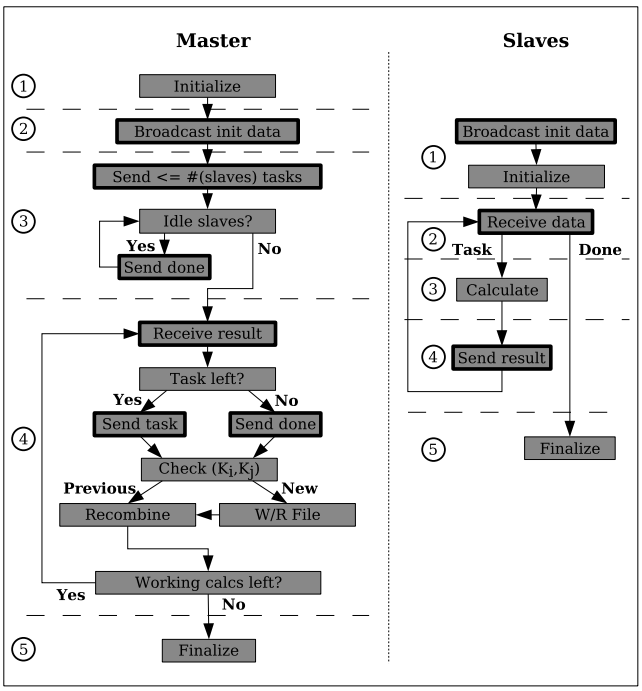
This effectively means that allocating and deallocating memory for pointers and dynamic arrays happens identically, as do other dynamic memory operations – all we need to change is the declaration of the variable itself.

Again, we will discover that the use of pointers instead of dynamic arrays at this stage will provide us with further benefits during later stages of the re-factoring process, as demonstrated in section 6.2.4 and beyond.

6.1.3 The new code layout

The basic layout of the new code-base is illustrated in figure 6.4. A step-by-step overview follows.

The general layout of the parallel program at this stage is as follows :



*Thick-lined boxes represent MPI-operations (broadcast / send / receive). The **master** code 1) initializes, 2) broadcasts initialization data to the slaves and 3) sends initial task data to start off the slaves. It then goes into 4) the main loop. Therein it a) waits for a slave to send back a result, b) sends either a new task or a "done" message and d) recombines the partial result matrices returned into the total result matrices. Finally, it leaves the main loop to 5) finalize.*
*The **slave** code 1) waits for init data from the master, then 2) waits for a task from the master, 3) calculates a partial result matrix for the given task, 4) sends it back to the master and repeats the process from step 2 on until it is told by the master to quit. It then 5) finalizes.*

Figure 6.4: High-level overview of the F90 code-base after the first re-factoring step

- For the master code

  1. Initialize:
     Comparable to the first version, the major difference being that most of the code has been split off into a separate function, and that we now add information about the MPI virtual machine such as the available amount of slave processes. The whole initialization process is now resolved through one function call from the main program as described in 6.1.2.1.

2. Broadcast:
   The information that was previously read and deduced from the configuration files is broadcast from the master to all of the slaves in order to allow them to do their own initialization. This allows us to take on the initialization phase for all processes without having to distribute the config files to all the individual slave machines or having to use a distributed filesystem. The whole broadcast process in this version (which involves multiple MPI interactions) is also resolved through a single function analogously to step 1.
3. Send initial data:
   From now on, we will refer to a single unique collection of input parameters as a *task*. Every slave is sent one task to start out with. If all tasks have been sent while slaves remain idle, send the remaining slaves a quit message. Once all the slaves have received one message, the master enters the main loop.
4. The main loop, as previously, consists of multiple nested loops which traverse all input parameter combinations for the calculations. The main loop remains the core body of the code. During the main loop, the program goes through the following phases:
   (a) Wait for result data:
   As long as slaves are running, the master waits for result data.
      i. Send new data
      When an answer from a slave comes in, send another task if available. Otherwise, send quit message. The slave is "marked" as done.
      ii. Recombine
      After sending the new task, the master recombines the partial result matrix received from the slave into the appropriate total result matrix. When necessary, the current total result matrix is written to – and the proper one recovered from – file. Recombination has now been split off into its own function as well.
5. Master finalize:
   When all tasks have been sent and all result data has been received, the final results are written to file and finalization proceeds as previously. This final part is now performed through a function call as well.

- For the slave code:

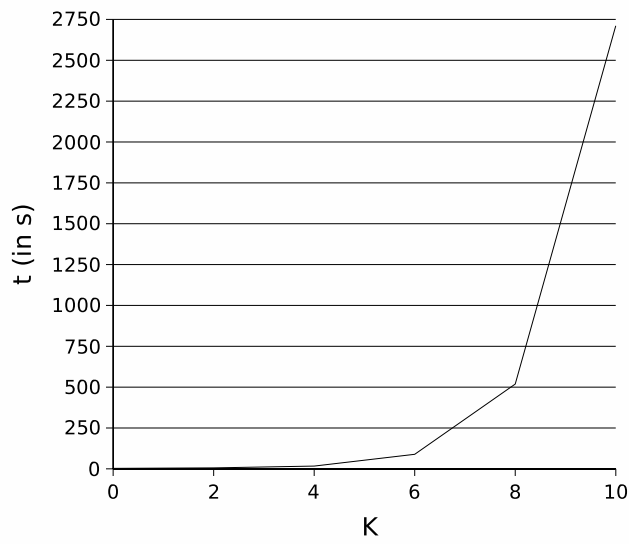
1. Initialize:
   The slave waits for the broadcast initialization data from the master, and uses the received to perform its own initialization phase.
2. Main loop – the slave now goes into a main loop consisting of:
   (a) Wait for task:
   The slave waits until it receives a tasks from the master

(b) Calculate:
The slave calculates results for the task, producing a partial result matrix.

(c) Send results:
The slave sends back the partial result matrix to the master.

3. Finalize:
When the slave receives a null message instead of task data, it exits the main loop and starts its finalization process, mainly consisting of deleting the dynamic memory allocated during initialization. No files are written by the slave.

6.1.4 Performance



Running the program, again for $L = 0$ and $K = 0, 2, 4, .., 10$, *on a 32-node heterogeneous cluster. Two processes are run on each host to take advantage of hyper-threading in some of the hosts and minimize processor idle time in general, for a total of 64 processes including the master.*

Figure 6.5: Speedup through parallel computation

Once more, we calculate for $L = 0$ and $K = 0, 2, 4, .., 10$, a range of $l_1 = l_2 = 0, 1, .., K/2$ values, and 45 t values. This version already delivers a great improvement in total wallclock time when run on our cluster systems as shown in figure 6.5.

As can be seen, the new graph is very comparable to the previous one, though with a significantly reduced total wallclock time for each step except $K = 0$. A run for $K = 10$ which would otherwise take about a whole day can be performed in

slightly more than 45 minutes on the cluster. For values of $K = 6$ and higher, a total speedup of around a factor 30 could consistently be reached. This is sufficiently close to the actual number of available processors to conclude that, at least for the given values for K and L, our straight port is sufficiently successful at making efficient use of the available resources.

6.1.5 Basic profiling

We can roughly translate the sequence of phases which the software goes through into a number of general states which are mostly shared across master and slaves: initialization (*init*, master and slave step 1), broadcast initialization data (*bcast*, master and slave step 2), send task (*send*, master steps 3.1 and 3.2.2, slave step 2.3), receive (*recv*, slave step 2.1, master step 3.2.1), calculate (*calc*, slave step 2.2), recombine (*recomb*, master step 3.2.3) and finalize (*fin*, master step 4 and slave step 3). A tool called the Multi Process Environment (MPE) now allows us to investigate whether the software makes efficient use of the available processing resources by tracking how much time is spent in each of the states, giving us the opportunity to look for detrimental synchronization effects and other issues with the new implementation.

MPE [72] is a library which provides a basic set of procedure calls that can be used for logging events within an MPI program. MPE actually uses MPI calls itself for internal communication, and will thus seamlessly "plug" into any standards-compliant implementation of MPI, i.e. it is implementation-independent. Couples of event calls can be used to characterize states in a program, e.g. we can log an event before sending a piece of data ("*send*") and after the send is done ("*sent*"), the time in between which would comprise a state ("*sending*").

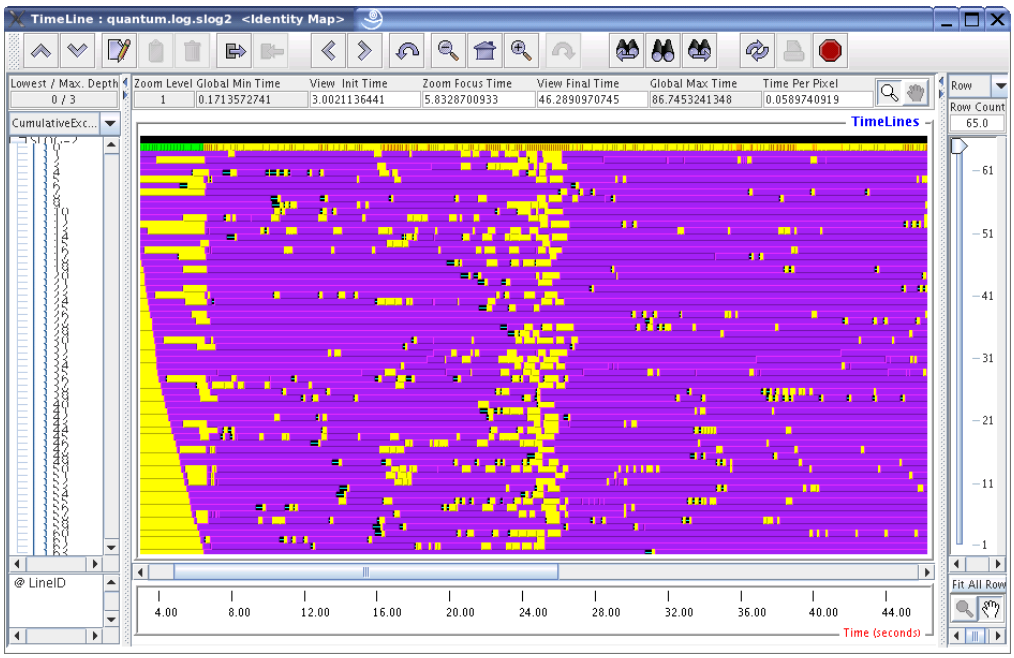
Timestamps for all of the states are logged on the individual processes. All of this localized timing data is then normalized and integrated into a single logfile tracking time spent within – and between – different states. This allows programmers, using a set of about 4 basic different calls, to perform elementary profiling for any MPI-based software. The logged results can be visualized with any type of specialized software capable of reading the employed data file formats.

For our program, we instrumented the code with a number of event calls to indicate the virtual states referred to above (*init, bcast, send, recv, calc, recomb, fin*). No further changes were made to the software, and the code – at this point – contains no further references of any kind to the concept of "state". (Future revisions of the code will make explicit use of state machines as a basic framework for computation – and thus, will map even more easily to MPE 7.3.3.)

Profiling results for $L = 0$

We ran the instrumented program once more for $K = 0, 2, 4, ..., 10$, and gathered and studied the contents of the log files with a visualization program called JumpShot 4 [73]. Results for $K = 6$ with $L = 0$ are shown in figure 6.6.

Each line in the figure corresponds with the state progression of one process in the job as a timeline, with colored bars representing different states. Timelines for all the jobs are lined up and normalized, with events at the same point on different lines effectively happening at the same time. The topmost timeline represents the



State changes for all processes are shown. Changes from the calculating state into communicating states are quite fluid, and little time is lost by the slaves waiting for data from the master, or waiting for the master to handle results.

Figure 6.6: basic profiling of the parallel program for $K = 6$ and $L = 0$

master process, the others represent the slaves. Purple represents calculation, yellow communication, orange recombination.

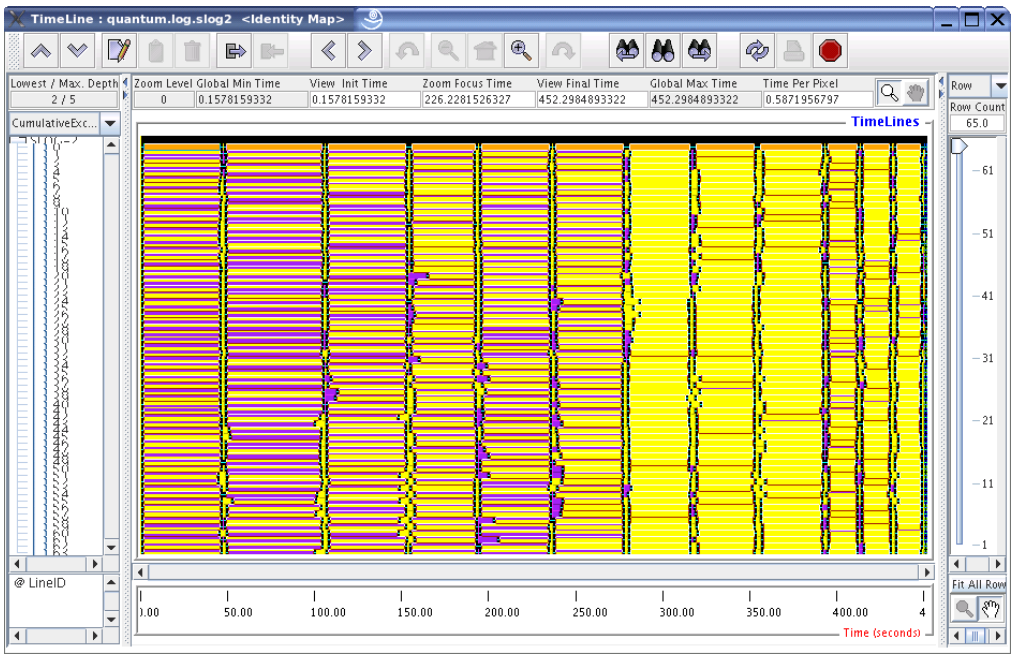
The data shown in this figure concurs with the earlier evaluation we made based on improvements in wallclock time vs. # available processes : for the give K and L values, the program runs relatively efficiently and the slaves are continuously kept at work while the master process loads, recombines and saves cumulative matrices when necessary. Globally synchronized communication intervals occur at two occasions, but they never dominate the calculation sequences and CPU usage is generally near-optimal. General cluster performance reports affirm that all CPUs are kept at 100% activity for the whole duration of the job and resource usage is near-optimal.

Profiling results for $L = 2$

The next phase in analysis is to try on a number of job runs for $L = 2$. This yields the graph shown in figure 6.7.

As is to be expected, computation times for $L = 2$ are generally longer than those for $L = 0$, but they seem to remain within reasonable bounds. A notable difference is visible in the profiling visualization though. While the figure for $L = 0$ showed a more or less continuous sequence of computational states in the slaves, we can now

see large time gaps between them in which the slave is waiting for a new task.



A calculation for $K = 6$, $L = 2$. Notice the difference with the previous figure, notably in the first line (master) vs. those below it (slaves).

Figure 6.7: Profiling the program for $L = 2$

The reason for this is the recombination phase in the master process : for $L = 2$, this phase requires so much time that slaves are starved from new tasks while the matrices from a previous task are being read and recombined. The obvious way around this problem will be to distribute the recombination process as well.

6.1.6 Conclusion

In general, parallelization as a solution for reaching higher values for K has proven to be successful, as was to be expected. MPI turns out to be both simple and effective as a tool for adding parallelization without requiring great overhauls in code organization, and without adding great complexity either.

There are, however, still some problems, due to the nature of the computations themselves, which require a greater reworking of the original code. We specifically note problems with the recombination process, the natural solution for which would be to add parallelization to it as well.

The changes necessary for this re-factoring step, however, are bigger than for the first step. Given that this will not be the last re-factoring step by far, and that an even further growth in complexity is to be expected during further steps, this means

that we will have to look into means of making the code-base easier to re-factor and more maintainable.

6.2 Porting and integrating with C++

In this section, we finally set out to investigate the answer to our third research question: *With regard to Fortan-based legacy code: what needs to be done to make easy integration of said code with modern C++-based code work in a user-friendly manner.*

We will, after some preparatory steps, investigate the issues concerning the integration of Fortran (first Fortran 77, then Fortran 90 code) with C. Then, we will move on to C++, specifically concentrating on the problems that show up when trying to integrate C++ code with F90.

6.2.1 Rationale

This is the point in the refactoring process where we came to the conclusion that porting part of the code to C++ would be a valid option as a next step in re-factoring. A few reasons came up as a justification for this approach.

A first – admittedly minor, yet practical – reason, is, simply, that we have a greater supply of C++ programmers than Fortran programmers - certainly Fortran 90 programmers. More important, however, is that the object-oriented aspects of C++, and the many software engineering solutions and libraries available for it, make it a perfect fit for our goals of modularity, adaptability and re-usability. Given the availability of an MPI API for C++, porting the parallel code itself should, at a first glance, not be a practical problem. Since parallelism is our main concern at this moment, we will leave the issue of porting the calculation code for a later date.

This allows us to retain a conceptual division in the structure of the code. On the one hand, we have the parallel code in C++, which can be written, extended and maintained by computer scientists specialized in the issue of parallelization. On the other hand, the computational F90 code can still be maintained by physicists well-versed with the current Fortran code-base without them having to worry about MPI issues.

This adds one extra step in complexity though: we will have to integrate the two codes, bridging calls between C++ and Fortran.

6.2.2 Preparatory steps

In a first stage, the C++ code will probably be looking very similar to the Fortran program, having an initialization phase, some management of the parallelization process, main loop, communication, calculation, recombination and finalization phase. The parallelization and communication part is easily translated from Fortran to C++ using MPI. The main loop is an elementary construction which is also easily translated into the new language.

We already discussed our decision to keep the computational part of the code in Fortran. We will retain the recombination procedure in Fortran as well for similar reasons. The finalization phase is another issue, especially as it concerns output of the results. The total result matrices are currently written to unformatted Fortran files, the exact structure of which is not easily duplicated in C++. Hence, this part will remain in the original languages as well.

Now, it turns out that splitting these parts of the code off into functions during the previous step was not just practical for the purpose of cleaning up the code and making it easier to maintain - in the coming sections, we will see that the added modularity also helps us in our effort to bridge between our two programming languages.

6.2.3 F77 and C++: issues and approach

Basic bridging between the C programming language and Fortran 77 is a long-established practice. Hence, mixing C and F77 is the first issue we shall investigate before addressing C++/F77 programming. Finally, we shall see that multi-language programming in C++ and Fortran 90 brings about some unexpected issues – enough to merit it its own specific, and rather extensive, section.

C and F77

As mentioned in the previous section, C to F77 cross-calling was classically used to add dynamic memory (heap) management capabilities to F77 code. The other way round, F77 calls were often used in scientific software otherwise completely written in C, especially for certain critical calculations which C compilers are unable to optimize without heavy – and hard to re-use – custom coding.

An example of this are complex calculations involving multiple arrays, which standard F77 compilers are able to heavily optimize through loop elimination techniques and foregoing the use of memory – and CPU – consuming temporary values. Another classical application of C/F77 intercommunication was for using code from older, established but legacy, function libraries which had historically been written and maintained in Fortran.

Calling Fortran code from C software, and vice-versa, is only possible through function calls. There is no standardized way to directly "embed" Fortran code in C (or the other way round). Fortran/C or C/Fortran function calls, however, are a relatively straightforward affair as long as a number of basic rules are taken into account, especially from the C side of the equation.

1. All function calls in C are purely pass-by-value (pass by reference parameters are simulated through the use of pass-by-value pointers, referring to the original parameters), while all Fortran function calls use purely pass-by-reference semantics. This problem is easily solved by passing all C arguments to a Fortran function as pointers, and by only accepting pointers as arguments to C functions called from Fortran. When calling a Fortran function from C, of course, an appropriate header file with function prototypes for the Fortran functions needs to be supplied to the C compiler. Likewise, when calling C functions from Fortran, care has to be take to properly declare the C functions as "ex-

tern" in the case that the C function's return value is not void (subroutines – or C functions returning void – never have to be declared within the context of their use in Fortran).

2. Fortran makes an explicit distinction between subroutines (not providing a return value) and functions (providing a return value). C functions always specify a return value in their declaration, with functions without "actual" return values explicitly declaring their return type as "void". This is simply handled by treating Fortran subroutines as functions returning void in C, and – vice versa – calling C functions returning void as subroutines from Fortran.
3. Most Fortran linkers require functions to have a trailing "_" (underscore) character in their name. Fortran compilers will usually add this underscore silently to the object files while compiling. The GNU compiler can be instructed not to add the trailing underscore to ease cross-calling with C, but explicitly warns that this might lead to object resolution conflicts with Fortran built-in functions (which do not have the trailing underscore by default).
4. Most Fortran compilers will, by default, move all names of compile targets in object files to lowercase as Fortran is case-insensitive. As C is case sensitive, a *SUBROUTINE FOO* in Fortran should be called as function *void foo_ ()* in C.

The use of Fortran functions from C as described above, is illustrated by the code snippets in figure 6.8.

There are a number of things we need to be cautious about though.

One important point which we already mentioned in the previous section, is that Fortran functions are not checked for their parameter list at link time. This means that it is extremely important to make sure that the declarations in the header file match exactly with the actual Fortran definition when calling Fortran from C. It also implies that it is technically impossible to do compile- or link-time checks on the function parameter list at all when calling C from Fortran.

Fortran and C also use entirely different approaches in the way they handle arrays. 1) In C, elements in multi-dimensional arrays are addressed in row-major order (e.g., for a 2-dimensional array, the first index will indicate the row, while the second index will point to the row). Fortran, however addresses elements in column-major order, i.e. the other way round. 2) Elements along a single dimension in a C array are indexed from index nr. 0 to index nr *[nr. of elements -1]*. The default for Fortran is to start counting from 1 up to index *[nr of elements]*.

In fact Fortran allows programmers to address arrays using arbitrary ranges of indices, and even to use arbitrary index steps between individual, consecutive elements. C has no built-in facility to support any of these features. This implies that C code will have to reverse the index order, and decrement indexes by one, for every array it receives from Fortran.

An example of the differences is given by the code snippets and the illustration in figure 6.9.

Yet another point of notice is the difference in the way that C and Fortran treat strings. C strings are basically null-terminated arrays of characters (with 1 character = 1 byte). Hence, a C string of length n will actually take up $n+1$ bytes in memory. The trailing null character is used in order not to have to pass string lengths to

Fortran 77 code

```
!!!!!!!!!!!!!!!!
! functions.f77)
!!!!!!!!!!!!!!!!

SUBROUTINE F77_ROUTINE(input)
    INTEGER input ! Subroutine parameter.
    ! Do stuff...
END SUBROUTINE
```

C code

```
/****************
* F77_functions.h
*****************/

extern "C" {
    void f77_routine_(int* input);
}

/*********
* caller.c
**********/

#include "F77_functions.h"

// ..code...
f77_routine_(value);
// ...code...
```

Makefile

```
functions.o : functions.f77
    mpif77 -c functions.f77           # Compile f77

caller.o : caller.c F77_functions.f77
    gcc -c  caller.c                  # Compile C

program : caller.o functions.o
    gcc -o program caller.o functions.o -L/usr/lib/gcc
.................-lgfortran           # Link with C linker
```

C and F77 code snippets. The F77 code (functions.f77) provides a subroutine, and is written according to regular Fortran coding practices. The programmer need not be aware of the fact that the code will be called from C. In order to call it from C, we provide a header file (F77_functions.h) with the appropriate function prototypes. We use return value "void" for the subroutine, state all parameters as being pointers, and add trailing underscores to all the function names. Compile each unit (caller.c / functions.f77) under its respective compiler and link. When using the GNU compiler, we explicitly need to link in the Fortran lib in order to use the Fortran standard functions.

Figure 6.8: Calling F77 functions from C

function calls, which is, in fact, a requirement when passing other types of array with extents that are unknown at the compile time. In Fortran, strings are not null-terminated, hence a string of length n will actually take up n bytes. This implies that

```
/
*************************************************************
**
*²In C - index starting from '0' in each dimension, i.e. :
*
*   00  01  02  03
*   10  11  12  13
*   20  21  22  23
*
*   Access in row-major order: a sequence of #rows (3) arrays of
*   array of #columns (4) integers, in rising order of rows.
*
*   i.e. { {00,01,02,03}, {10,11,12,13}, {20,21,22,23} }
*************************************************************
****/

// Examples of use
int array[3][4];   // A 3x4 array.
array[0][0];   // Returns first int in sequence.
array[2][3];   // Returns last int in sequence.
Array[1][2];   // Returns seventh int in sequence.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!
!   In Fortran - index starting from '1'in each dimension,
i.e. :
!
!   11  12  13  14
!   21  22  23  24
!   31  32  33  34
!
!   Access in column-major order : a sequence of #columns (4)
!   arrays of array of #rows (3) integers, in rising order of
!   columns.
!
!   i.e. { {11,21,31}, {12,22,32}, {13,23,33}, {14,24,34} }
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!

! Examples of use
INTEGER array(4, 3)    ! A 3x4 array.
array(1,1)         ! Returns first int in sequence.
array(3,4)         ! Returns last int in sequence.
Array(2,3)         ! Retun eighth int in sequence.
```

Array sequencing in C vs. Fortran. The code snippets shows a 2-dimensional array for storing a 3x4 matrix of integers, and the way it is turned into a sequence in either C and Fortran. Notice that the same indices – corrected for indexing from 0 in C – can point to different elements in the sequence.

Figure 6.9: Fortran indexing vs. C indexing

the string length has to be passed as a parameter to any function using it – an act which the compiler will take care of for the programmer by passing the string length at run time by means of a *hidden variable*, which is silently added to the parameter list by the compiler. Problem is: the placement of this hidden variable is *compiler-dependent* (an expression which we will, sadly, be hearing quite a bit more often in the next section). Some compilers will add the hidden variable after the string itself in the call list, others might tack it on at the end of the regular parameter list, etc..

This mechanism is illustrated for different compilers by the code snippets in figure 6.10.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!     In Fortran - function accepts two strings, metadata is hidden.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

SUBROUTINE FORTRAN_ROUTINE(string1, string2)
     CHARACTER * string1, string2 ! Parameters
END SUBROUTINE FORTRAN_ROUTINE

/*****************************************************************
*     IN C  - function call requires explicit meta-data. Function
*     call format differs between different compilers!
*****************************************************************/

// Note that hidden Fortran parameter for string length is passed by
// value!

// GNU : string length follows char*
#ifdef COMPILER_GNU
void fortran_routine_(char* string1, int len1,
     char* string2, int len2);
#endif

// AT&T : string length follows at end of parameter list
#ifdef COMPILER_ATT
void fortran_routine_(char* string1, char* string2,
     int len1, int len2);
#endif
```

C and F77 code snippets illustrating the passing of strings from C to Fortran 77. It illustrates different ways to call an F77 function with two string parameters from C, depending on the specific compiler/linker being used. The GNU compiler tacks the hidden string length variables on to the end of the parameter list, while the AT&T compiler adds a hidden variable after each string parameter itself.

Figure 6.10: C to F77 strings

Luckily, for us, the function call semantics discussed in 6.2.2 actually allow us to short-circuit this compiler-dependent issue: by including the string in a structure, we make sure that 1) the compiler knows about maximum string lengths and 2) knows how to format and write to / read from the data blocks in both C and Fortran without needing any hidden parameters. This saves us a lot of trouble.

There is also the issue of so called "common blocks" – Fortran's mechanism for implementing global variables across multiple compilation units. Despite being against structured programming guidelines, common blocks still occur in quite a lot of code, mostly due to the lack of scoping options in early version of Fortran. Capturing common blocks in C++ is tricky but possible by using a global struct in C. This approach is illustrated by the code snippet in figure 6.11.

```
!!!!!!!!!!!!!!
! In Fortran:
!!!!!!!!!!!!!!

INTEGER lp
DOUBLE PRECISION r0, u, rp


COMMON
 &  /r/r0
 &  /pot/u(4,3), rp(3), lp

/*******
* In C:
********/

extern struct {
    int lp;
} r_;

extern struct {
    double u[3][4]; // Notice the reversal of indices.
    double rp[3];
    int lp;
} pot_;
```

C and F77 code snippets illustrating the use of common blocks across C and Fortran 77. The code illustrates how the common block in F77 is mapped to a structure in C, allowing C code to read and set common block variables without needing changes to the Fortran code-base.

Figure 6.11: C to F77 common blocks

Unfortunately, common blocks are used in the Fortran code-base. But, yet again, by careful re-factoring in step 1, the common blocks have been relegated to the individual functions for init, broadcast, calculation, recombination and finalization. The main loop and communication / distribution parts of the code have no need for – and do not use – common blocks. Hence, we will be able to call the Fortran functions without having to know about the common blocks from within the non-Fortran code.

C++ and F77

From this point on, it becomes relatively easy to extend our approach for doing C-F77 cross calls to doing C++-F77 calls, given a number of extra caveats.

A first important issue to take care of when cross-calling functions between C++ and Fortran is that of name mangling. Contrary to both Fortran and C, C++ actually does support function parameter list checking during the link phase. Practically, this means that trying to link a function call of the type *void foo(std::String)* to an implementation in library file *bar.a* of the type *void foo(int)* will result in an error at the link time because the function arguments do not match up, and cannot be

equated by means of simple automatic conversion. C++ compilers accomplish this by means of a technique called "name mangling".

When applying the name mangling scheme, C++ compilers actually change the name of the C++ function in the object file to match the parameter list. For example, the Fortran compiler will compile the function *INTEGER FOO(INTEGER)* simply into *"_foo_"* (notice the trailing underscore added by the Fortran compiler), while a C++ compiler could take the similar function *void foo(int)* and "mangle" its object file name into "*_foo_ int*" (parameter type added to object file name, no trailing underscore) for the sake of correct error checking during the link phase. This technique is also used by the linker to address the correct version of a function in the case of function overloading, i.e. a call to function *void foo(int)* or *void foo(char)*, both in library file *bar.a* will respectively link to the "mangled" object names *_foo_ int* and *_foo_ char*.

This is an issue we will need to take into account, both when calling C++ functions from Fortran (we should actually be calling the mangled name of the function) and when calling Fortran functions from C++ (the Fortran function should have a name that matches to the mangling scheme used by the C++ compiler). Alas, name mangling is, once more, a compiler-dependent feature (we are starting to see a pattern here).

Luckily, this problem can also be overcome by declaring function prototypes in the C++ header files as "extern C". This ensures that the compiler handles these functions as if they were plain C functions, without applying name mangling. This does mean however, that link-time parameter type checking is lost when calling these prototypes. This can be remedied by addressing the "extern C" Fortran function prototype by means of a C++ stub, which does apply name mangling and thus allows for link-time error checking. We illustrate these approaches with code snippet in figure 6.12.

In this context, it is important to note another important issue when calling Fortran functions by means of a C++ prototype: as soon as we cross the language barrier, all of the extra C++ meta-data in the parameter list is lost. For example, the compiler does not offer any guarantees that a Fortran function with an *extern C* prototype of *void foo(const int*)* will actually leave its int argument constant once the runtime crosses over to the Fortran function.

A few other short issues deserve to be mentioned here. First: sadly, no compiler collections to date make use of the C++ namespace scoping mechanism to address the problem of calling Fortran module members from C++, despite the fact that both approaches conceptually line up nicely. The Chasm[74] cross-language development tool does allow the use of namespaces to accomplish this goal, but it has issues of its own – we will discuss Chasm and its features in more detail in the next section. Second, despite that – given the above guidelines – linking C++ to F77 is not much of a problem with most compiler suites, the same does not always appear to be true for calling C from Fortran. Some compiler documentation actually advises putting the contents of the *PROGRAM* segment of a Fortran program in a subroutine and calling it from a *main()* function in C(++), in order that C(++) to Fortran semantics apply instead of the other way round [79].

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!   In Fortran—- argument types not specified in function
!   signature. Function does not change arguments.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!

SUBROUTINE FORTRAN_ROUTINE(arg1, arg2)
    INTEGER arg1
    DOUBLE PRECISION arg2
!   Do stuff...
END SUBROUTINE FORTRAN_ROUTINE
```

```
/*****************************************************************
*   IN C++  - argument types specified in function signature,
*   both in “extern C” block and regular code. Argument type
*   is *only* checked within regular C++ code!
*****************************************************************/

// Force users to call stub instead of calling Fortran function
// directly.
namespace /*UNNAMED*/ {
    extern “C” {
        // ARGUMENTS NOT TYPE-CHECKED AT FUNCTION CALL!
        // CONST-NESS NOT GUARANTEED BY CALL TO FORTRAN!
        // Changing argument within Fortran routine will lead to
        // semantic error!
        void fortran_routine_(int const *, double const *);
    }
} // namespace UNNAMED

// C++ stub - use C++ function signature conventions; parameter
type
// is checked at function call (name mangling). Argument const-
ness,
// sadly, can still not be guaranteed - we're going to have to
trust
// the Fortran function... at least, constants can be used as
// arguments to this function without compiler complaints.
void fortran_routine(const int& arg1, const double& arg2) {
    fortran_routine_(&arg1, &arg2);
}
```

C++ and F77 code snippets to illustrate the use of stub functions to call Fortran functions from C++ using “extern C” function prototypes without losing compile-time parameter type checking.

Figure 6.12: C++ to F77 – dealing with name mangling

To end this section on a positive note, C++ does support both call by value and call by reference semantics for its function calls. This feature is nicely integrated when cross-calling from C++ to Fortran. This means that when working from C++, one does not have to pass pointers to the Fortran function arguments, but that rather, one can specify pass-by-reference parameters in the function prototype and pass the parameters as is. Have a look at the code snippet in figure 6.13 for an example.

Conclusion

We have illustrated that it is possible to do cross-calling from C++ to Fortran 77 functions using techniques which have been fairly well tested through widespread use, given a few caveats. The issue becomes murkier, however, as soon as we go on to calling Fortran 90 functions. The main issue here concerns the addition of

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!   In Fortran-- no specific acions needed.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!

SUBROUTINE FORTRAN_ROUTINE(arg1)
    INTEGER arg
!   Do stuff...
END SUBROUTINE FORTRAN_ROUTINE
```

```
/*************************************************************
*   IN C++  - we can use references straight in the extern "C"
*   Fortran subroutined prototype...
*************************************************************/

// In header file...
extern "C" {
    void fortran_routine_(int&);
}

// In program...

#include "fortran_header.h"

int x  =  5;
fortran_routine_(x);
```

C++ and F77 code snippets to illustrate the use of pass-by-reference arguments for calling F77 functions from C++. This makes function calls from C++ less of a fuss than for C.

Figure 6.13: C++ to F77 – pass-by-reference calling

pointers to Fortran in version '90. We will discuss these issues in more detail in the following section.

6.2.4 F90 and C++: fundamental issues

Given the ability to communicate with F77, what do we need in order to do so with Fortran 90 procedures as well? For that, we will have to look at the two major additions to Fortran in version 90: modules and the introduction of pointer data-types and dynamic memory management.

C++ and F90 modules

On this subject, we can be short: referring to functions, subroutines and variables that reside within the scope of modules is another compiler dependent problem. The main reason for this is that the mechanisms for function calling from C++ to Fortran were actually designed with C in mind – and C simply doesn't have any mechanic that resembles Fortran modules. In short, there is no compiler independent way to address Fortran module members from C. The GNU compiler doesn't even have a documented way of doing this at all. Calling Fortran module members from Fortran non-module members within C code however does not pose any problems at all. This implies that any modularity in the Fortran code has to be implemented beyond – and decoupled from – the API visible to C.

An introduction to F90 dynamic memory management

We already mentioned in section 6.1.2.2 that F90 offers two major tools for handling dynamic memory: dynamic arrays and pointers. Both of these are "allocatable", i.e. it is possible, using a function call, to provide a pointer or a dynamic array with arbitrary amounts of contiguous memory. A bit further on in the same section, we also made the observation that both of the provided techniques use virtually the same semantics for their dynamic memory management, i.e. the syntax for allocation / deallocation is identical for either F90 pointers and dynamic arrays, and changing between them comes down to simply changing the variable / parameter declaration.

Though no policy is enforced by F90 itself, it is generally assumed that Fortran programmers will use dynamic arrays as the primary way for managing array-like structures on the heap. The use of pointers, then, is supposedly reserved for creating and managing the more intricate types of dynamic data structures like linked lists, trees, hash-maps etc.. This means that our earlier choice to remove dynamic arrays from the code-base, in favor of pointers, is generally going against these assumptions as we are basically still manipulating arrays in memory.

However, going against the implied policy – apart from its usefulness w.r.t. our function calling semantic – proves to be a wise choice when taking our porting process into account. C++, as the successor to C, has supported pointers since its inception (in fact, pointers and dynamic memory management are widely used throughout just about any relevant piece of C++ code). However, C++ has no language-level equivalent of F90 dynamic arrays. Hence, pointers will be the only logical way to go about transferring dynamic memory structures of any nature from one language to the other. Yet another one of our previous re-factorings turns out to be useful beyond its original motivation. The repercussions of attempting to pass pointers to arrays between C++ and F90 is one we will address in the next section, but passing plain arrays between the two languages is, in any case, not a problem, as illustrated by the code snippet in figure 6.14.

A major difference between F90 and C++ pointers which is worth mentioning at this point, is that Fortran explicitly differentiates between pointers to individual data elements of a type, and pointers to array structures of the same type. This differentiation does not occur in C and C++, where a pointer to an array of elements is just a plain pointer to the first element of the array. These do not offer access to any further information as to the size of the array pointed to, or any other kind of meta-data concerning the array. This difference – and especially the way it is practically implemented by modern F90 compilers – has some unexpected repercussions and will require work on the part of the C++ code to overcome. The code snippet in figure 6.15 gives an overview of dynamic array and pointer use in F90 and C++.

Pointer passing between C++ and F90 – a first attempt

Recapitulating from the previous section, we have seen that both F90 and C++ support pointers as a way to handle dynamic memory. And we know that the current code-base confronts us with a need to exchange dynamically created arrays between the C++ part and the F90 parts.

Specifically, we need to 1) exchange dynamically allocated arrays between the

```
/**********
* In C++ :
***********/

//Fortran function taking an array argument
 extern "C" {
      void fortran_routine(int const & extent, int array[]);
}

int const size = 5;               // define array size
int array[size];                  // define array of 5 ints
fortran_routine(size, array);     // pass array + extent to Fortran

!!!!!!!!!!!!!!!
! In Fortran :
!!!!!!!!!!!!!!!

SUBROUTINE FORTRAN_ROUTINE(extent, array)
      ! "array" is an array with extent "extent" - it can be
      ! pointed to by a pointer to array
      INTEGER :: extent
      INTEGER, DIMENSION(extent), TARGET :: array

      i = array(4)          ! retrieve 4th element from "array"
END SUBROUTINE FORTRAN_ROUTINE
```

The code snippet illustrates the declaration and use (allocation, handling, deallocation) of a both dynamic array and pointer in F90, and a similar set of actions in C++. Notice that when it comes to handling (addressing and working with array members), both F90 and C++ allow the programmer to use pointers to array as if they actually were regular, plain arrays.

Figure 6.14: Comparing arrays between F90 and C++

F90 initialization function and the main loop in C++ for both the master and slave code, 2) send arrays from the main C++ loop of the slave to the calculation function written in F90 and back and 3) once more, from the C++ main loop in the master process to the F90 finalization procedure to write out the total result matrices. Deallocation of dynamic memory could conceptually be handled from either of the two languages, whichever suits us best. Generating debug data from the F90 finalization function is done pure internally in F90, and requires no further communication with the C++ part of the code.

In short, the ability to exchange pointers between C++ and F90 should make transitions between the two relatively seamless.

It turns out however, that there is a catch. The practical question we have to ask ourselves is: how does F90 exchange pointers between function calls, and how exactly do we accomplish the same thing from F90 to C++ and vice-versa. From our previous observations, we note 1) that Fortran has a distinct type for pointers to array while C++ uses a regular pointer to the array member type, and 2) that

```
/***********
* In C++ :
***********/

int * ptr;             // define : ptr is a pointer to int
int array[5];          // define : array is an array of 5 ints
int i = array[3];      // retrieve 4th element from array

ptr = array;           // ptr now points to 1st element of array
int j = ptr[2];        // index memory referenced by ptr like array

int size = 5;          // help variable
ptr = new[] int[size]; // allocate space for a new array of 5 ints
int k = ptr[4];        // retrieve last element from new array
delete[] ptr;          // deallocate space for new array

!!!!!!!!!!!!!!!
! In Fortran :
!!!!!!!!!!!!!!!

! define : "array" is an arry of 5 ints
! define : "ptr" is a pointer to a 1-dimensional array
! define : "dyn" is a 1-dimensional dynamic array
INTEGER, DIMENSION(5), TARGET :: array
INTEGER, DIMENSION(:), POINTER :: ptr
INTEGER, DIMENSION(:), ALLOCATABLE :: dyn

! help variables
INTEGER :: i, j, k, l

i = array(4)           ! retrieve 4th element from "array"
ptr => array           ! "ptr" now points to "array"
j = ptr(2)             ! index "ptr" like array
ALLOCATE(ptr(extent))  ! allocate space for new array
ALLOCATE(dyn(extent))  ! allocate space to "dyn"
k = ptr(4)             ! retrieve last element from new array
l = dyn(4)             ! retrieve last element from "dyn"
DEALLOCATE(ptr)        ! deallocate space for new array
DEALLOCATE(dyn)        ! deallocate space for "dyn"
END SUBROUTINE FORTRAN_ROUTINE
```

The code snippet illustrates the declaration and use (allocation, handling, deallocation) of a both dynamic array and pointer in F90, and a similar set of actions in C++. Notice that when it comes to handling (addressing and working with array members), both F90 and C++ allow the programmer to use pointers to array as if they actually were regular, plain arrays.

Figure 6.15: Dynamic arrays and pointers to array in F90 and C++

Fortran handles all variable exchanges through function calls by reference, which we emulate in C++ by using either a pointer or a reference to the exchanged type in the function signature.

If we were to try out a mapping from one to the other, that would logically

lead us to the following the following assumptions when passing a pointer to an array between C++ an F90: 1) a pointer to array type in F90 maps to a pointer to member type in C++, and 2) passing it by reference to a function yields a pointer or reference to the passed type in C++, i.e. either a pointer to a pointer to the member type, or a reference to a pointer to the array member type.

This act of passing a pointer to a pointer data type (otherwise known as *double indirection*), at first sight, might seem strange to some people not used to the C[++] programming style. In fact, structures like this are not exceptional in C[++] code at all, and they are especially common when working with data collections. Hence, once more, handling this case in C++ should not pose any problems.

Let us try this out with some practical code trying to access in C++ an array, allocated in F90, as illustrated by the snippet in figure 6.16. Unfortunately, though, the code doesn't work – that is to say: the program seems to crash unpredictably at seemingly random occasions some time after the F90 call. This happens, even if we don't do anything special with the pointer value... However, whenever we try to access the array for a read – and the runtime makes it that far without crashing – the retrieved value does seem to be correct!

It seems as, instead of simply providing a pointer to a pointer as we would expect, Fortran passes some kind of unknown entity to C++ which 1) actually does directly point to the array in memory, but 2) has some allocation problems of itself when passing from one language to the other... What exactly is going on here?

Under the hood: pointers to array in Fortran 90

It turns out that the core of the problem lies in the demands which the Fortran 90 standard puts upon pointer types.

Specifically the standard puts forth some important requirements to ensure that F90 code is able to use pointers with existing F77 functions that historically use regular data types. The F90 standard specifically states that programmers should be able to exchange allocated "pointers to array" as if they were actually regular arrays, without dereferencing the pointer. As a matter of fact, Fortran 90 doesn't even have a referencing / dereferencing function or operator. The only operator which was added to the language for the sake of pointers (apart from the obvious allocation / deallocation of course) is a pointer assignment operator, which copies the pointer address instead of the item pointed to itself. Otherwise, data items and F90 pointers to the same data elements are as good as perfectly interchangeable.

While smart compilers are able to take on this requirement without too much of a hassle for pointers to regular data types, some additional problems crop up when it comes to pointers to array types.

In Fortran, certain functions are available to poll basic data about regular arrays, like its extent, upperbounds, lower-bounds etc. (remember that arrays in F90 support custom indexing schemes). Given the requirement that we should be able to employ pointers to array at just about any place in the code where we can use regular arrays, these functions should also be able to operate on array-pointers. While for regular arrays, we are able to deduce these parameters at the compile time, this is not possible for dynamically allocated structures.This implies that any F90 compiler will

```
!!!!!!!!!!!!!!!
! In Fortran:
!!!!!!!!!!!!!!!

SUBRROUTINE FORTRAN_ROUTINE(ptr)
      INTEGER, DIM(:), POINTER : ptr
      ! other declarations...

      ! allocate array and fill with ints from 1 ... 20
      ALLOCATE(ptr(20))
      ptr = (/ I, I=1,20 /)
END SUBROUTINE FORTRAN_ROUTINE

/*********
* In C++:
**********/

// define fortran_routine as accepting a pointer to pointer
extern "C" {
      void fortran_routine_(int** ptr);
}

// create and send pointer to Fortran for allocation
int* ptr;
fortran_routine_(&ptr);

// from now on, things get hairy... crashes at unpredicable
// locations in code follow

// attempt to read from e.g. the 10th element of the array
cout << ptr[9]; // correctly prints out 10 if we get this far!
```

An attempt to pass pointers to array from Fortran to C++, by attempting in C++ to accept from the F90 function call a pointer (pass-by-reference semantics) to a pointer to member type (the only way to do array pointers in C++). When we try to access one of the array members, the result seems correct. However, we get unpredictable crashes.

Figure 6.16: An attempt at passing an F90 pointer-to-array to C++.

need to provide a mechanism for creating, keeping and passing runtime meta-data on any pointers to array.

Present-day Fortran 90 compilers accomplish this by instantiating a so-called *dope-vector* whenever a program declares a "pointer to array" type of variable. Whenever allocation or deallocation is performed on the pointer, all of the relevant runtime-data is written into (or removed from) the proper fields of the dope-vector. Likewise, whenever a pointer assignment is done, any relevant data from the dope-vector associated with the source pointer is copied into the dope vector of the destination pointer.

```
// The GNU F90 dope vector described as a C struct

typedef struct dope_vec_GNU_ {
  void*   base_addr;/* base address of the array */
  void*   base;     /* base offset */
  size_t  dtype;    /* elem_size, type (3 bits) and rank (3 bits) */

  struct {
    size_t stride_mult;  /* distance between successive elements */
    size_t lower_bound;  /* first index for a given dimension */
    size_t upper_bound;  /* last index for a given dimension */
  } dim[7];
} dope_vec_GNU;
```

The dope-vector as employed by the GNU F90 compiler. Notice that the actual address of the array allocated on the heap is only one field of the whole structure, which needs to be retrieved in order to access the array itself.

Figure 6.17: Dope-vectors as used by the GNU F90 compiler

The heart of the problem

So what exactly is the nature of the problem we get when passing a pointer from Fortran 90 to C++? The issue is that we are not passing an actual pointer in the C[++] sense of the word, i.e. an address pointing to an item in memory. What we are actually passing is the dope-vector associated with the pointer. As a matter of fact, while C++ compilers refer directly to an array by means of a pointer to the first element (single indirection), F90 compilers use the dope vector as an indirect reference to the array, which needs to be parsed before we are able to obtain the actual address in memory of the pointer (double indirection).

But then, how is it possible that we were able to read from the array in the example in figure 6.16 in the previous section? We were only able to accomplish this because the actual address of the array – coincidentally – also happens to be the first field of the dope-vector, as employed by the GNU compiler. In other words, we were unintentionally reading the first field of the dope-vector in C++, treating it as a C++ pointer to member type, referring to the first element in the array (which, effectively, it is). Hence, we were left with the impression that we were actually dealing with a single-indirection parameter, rather that the effective double-indirection structure that it is.

Now, from a programming point of view, this was pure coincidence: as the observant reader might expect, the internal structure of dope-vectors is generally compiler-dependent – an issue which we will need to take into account when designing our C++ solution in section 6.2.5. (From the compiler writer's point of view, of course, this is design choice is not coincidental at all: it allows for directly accessing pointed-to members without having to use complex redirection algorithms.)

The question that remains is: given that in C++, we were able to access the

pointed-to array correctly through the dope vector – which we were unknowingly accessing – why did the code in figure 6.16 still result in unpredictable crashes during the runtime? This problem relates to the way in which Fortran functions pass around dope-vectors.

As a compiler-generated pseudo-object, dope-vectors effectively happen to be the only variable entities which are technically passed *by value* between Fortran functions (and hence, C++ functions as well). In other words, during a function call, instead of passing a reference to the dope vector (in order to get the double indirection we had been expecting all along), the new function pushes a new dope vector on stack, and the existing dope-vector is copied into it as if it were a hidden local variable. This will happen for every function instance that it happens to pass through. While this approach will inevitably cost more memory, it helps computing performance by minimizing the amount of double lookups that need to be done: instead of first having to look up the dope vector, and then the address of the array, we can immediately access the array address through the local copy of the dope vector. The dope vector is pushed onto stack, and cleaned up together with all other local variables at the end of the function.

So, looking back at the code snippet, this is what actually happened when exchanging the Fortran array pointer with C++ code:

1. C++: call F90 function; the compiler has reserved a large enough amount of memory on the stack for the return-type specified in the C++ function prototype, i.e. a pointer to member
2. F90: enter Fortran function – the compiler makes sure that enough memory for the full dope-vector is available in the newly pushed stackframe; allocate new array on heap; copy appropriate data into dope-vector
3. F90: before returning, copy the full dope-vector into the return value – however, the dope-vector is larger than the space which is allocated for the return value in the stack frame of the calling context; the first field of the dope-vector (the address of the array on heap) is copied correctly, but the rest of the return value effectively overwrites the stack frame of the C++ context!
4. C++: on returning, we can use the pointer to member to correctly access the allocated array, but our runtime is no longer stable as we now have a corrupted stack – chaos ensues...

A basic solution to this problem is easily proposed: if we were able to create a full-fledged dope-vector in C++, we could use it to correctly capture the F90 array pointer as a return value without corrupting the stack, and then we can access the array through the dope vector's first data field as we should. This solution is far from simple, however, as dope vectors are hidden objects – effectively compiler generated artifacts.

Solving the problem

As most interaction between C[++] and Fortran code is traditionally done for the sake of communicating with legacy F77 code, this particular problem is one which has not been coming up a lot in the field. In fact, if you were to look around on the web, you would find that most of the available advice will just warn against any

attempts to call pointer-using F90 code from C++ or vice-versa. This does not help a lot in our particular case of course...

Still, there are a few packages available which might be able to address our problem: both Chasm [74] and Babel [77] claim to be fully functional cross-language development tools, and both the Blitz++ library [81] and the popular Boost [85] set of libraries for C++ claim some value as tools for communicating with Fortran as well. Let's go over them one by one and have at their respective merits...

Chasm

The Chasm tool works by parsing Fortran and C/C++ source files with a tool called XMLGen (using a library called the Program Database Toolkit (PDT)[76]) and generating an XML file. The XML file describes C++ classes, Fortran modules, user-defined types, functions, and function parameters. Expat, an XSL stylesheet transformer and a set of specific XSLT transformations are used to translate the XML file into bridging code that is used to call C++ and Fortran routines.

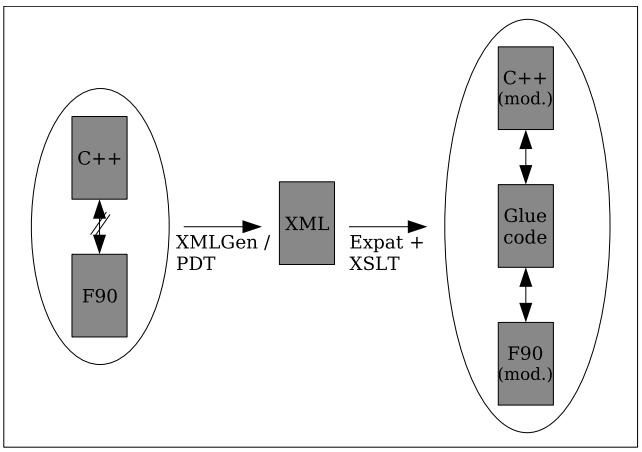
One of the the features of Chasm is that it allows for mapping C++ namespaces to modules – otherwise, elements in ++ modules are unavailable to any C++ code.

More interestingly, Chasm also supplies an array-descriptor library that provides an interface between C and F90 assumed-shape arrays. This allows arrays to be created in one language and then passed to and used by the other (foreign) language [75].

From all the above, it seems that Chasm would make for a good tool to take on our problem. Reality turns out otherwise however. Trying to run our code through Chasm version 1.4RC2 results in generated Fortran and C++ code which, sadly, turns out not to compile... Going back to a previous version of Chasm does not help us either.

When looking at the Chasm 1.4RC2 code, it turns out that the only functionality currently supported by this version is the ability to create dynamic arrays in C++ and pass them 'as is' to Fortran code as function parameters. No ability to support the opposite is provided, and there is no functionality to provide array-pointers through structs either – the class names are provided, but not even the interfaces have been implemented yet. On top of this, all of the code uses *purely* low-level calls – it does not allow programmers to provide differentiation between parts of code that they want to handle through low-level (maximally efficient) code, and others that one would like to handle through a higher-level, more portable solution (in those parts of the code where efficiency matters less).

After a few unsuccesfull attempts to contact the developers of the Chasm project to investigate the issue, it was clear: if we were to provide a solution to our problem in short order, Chasm would not be of help to us. In retrospect, we can probably state that the developers of the Chasm tool have priorities different from ours, which is a pity. However, Chasm might still indirectly turn out to be helpful, as the source code to it contains a great amount of information on the low-level aspects of the Fortran dope-vector. This will turn out to be crucial to the work that we will present in section6.2.5 - hence, our time spent on studying the Chasm project was an interesting one, and will certainly not be wasted.



An overview of the Chasm Fortran/C++ cross-language development tool. The source code for the C++ and Fortran parts is written according to guidelines put forth by the Chasm project. This code is then parsed, and an XML datafile describing the structure of the code is generated. This XML file is then, in its own turn, parsed. By means of XSLT transformations, it is turned in a combined body of extra code in both Fortran and C++, bridging the original two bodies of code.

Figure 6.18: A short overview of Chasm

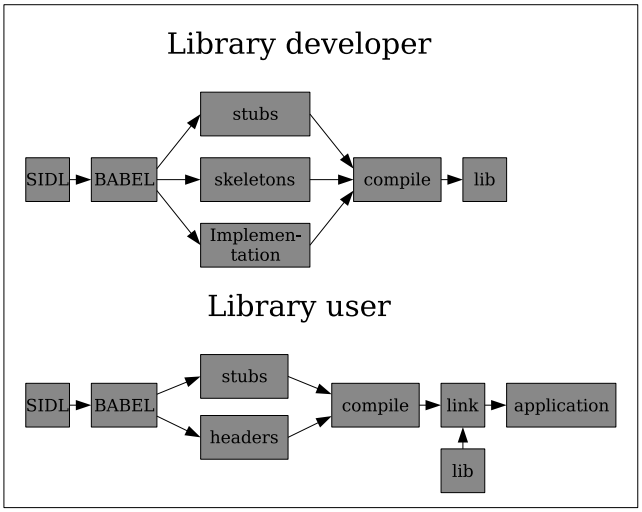
Babel

Babel (pronounced babble) is a java-based tool that attempts to bridge the gap between multiple programming languages, including C, C++, Fortran, Python, Java and others. It addresses the language interoperability problem using Interface Definition Language (IDL) techniques. An IDL describes the calling interface (but not the implementation) of a particular software library. Babel uses this interface description to generate glue code that allows a software library implemented in one supported language to be called from any other supported language. It implements the Scientific Interface Definition Language (SIDL), which attempts to address the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to Java.

The Babel tool suite consists of a number of separate pieces: a SIDL parser, a code generator, and a small run-time support library. The Babel parser reads SIDL interface specifications and generates an intermediate XML representation. The

idea is that a scientist downloading a particular software library from a component repository will receive not only that library but also the required language bindings generated automatically by the Babel tools.

The Babel code generator reads SIDL XML descriptions and automatically generates glue code for the specified software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space and, eventually, across memory spaces for distributed objects. The code generators create four different types of files: stubs, skeletons, Babel internal representation, and implementation prototypes. The Babel internal object representation created by the code generators is similar to that used by COM, CORBA's Portable Object Adaptor, and scientific libraries such as PETSc. The internal object representation is essentially a table of function pointers, one for each method in an object's interface, along with other information such as internal object state data, parent classes and interfaces, and Babel data structures. Stub and skeleton code translates between the calling conventions of a particular language and the internal Babel representation [78].



An overview of the Babel general cross-language development tool. Library developers write SIDL specifications for their lib and use BABEL tools to turn these into skeletons and stubs, adding extra implementation details when necessary. Everything then compiles into a cross-language lib. Users use the SIDL to generate client-side stubs and headers, and link against the babelized lib to eventually create the application.

Figure 6.19: A short overview of Babel

When trying out Babel on a few of the provided examples for C++ and Java, Python etc, it turns out to work fine.

When looking at Fortran 90 support however, we are confronted with an unpleasant surprise. It turns out that, for Fortran 90 to C++ compatibility, the Babel project actually uses Chasm to get the job done... In other words, despite the good ideas on which it is built, Babel will be no more helpful to us than Chasm turned out to be.

Other solutions

Two other major libraries claim to provide some support for Fortran-like structures as well: the Blitz++ template library, and the well-known Boost library for C++.

Boost [85] is an all-purpose library for C++ supporting many types of functionality, like added support for function-object wrappers, reference-counted smart-pointers, and efficient array structures. One of the array structures in Boost allows for different approaches to indexing the array contents, as we require for reading the Fortran arrays in C++. However, Boost does not provide means for exchanging arrays between both programming languages.

Blitz++ [82] on the other hand, is very focused on accomplishing a single goal: using template meta-programming techniques [84] to create a package for efficiently handling complex operations with multiple arbitrary-dimension array-like structures. As this has classically been the strongest point going for the Fortran programming language, reaching Fortran-like execution speeds and emulating Fortran-like array manipulation capabilities has always been a clear main issue with the developers behind Blitz++. Blitz++ has been able to show some rather positive results [83] from its extensive use of template meta-programming techniques in order to optimize many operations at the compile time, while hiding most of the complexity of the implementation behind a clear user-level interface, which will be instantly familiar to many Fortran programmers.

Blitz++ offers more than just performance benefits to the user: it also allows for a number of operations which are not available through the standard library facilities, like array slicing, as well as a multitude of matrix-like operations.

Interesting to us, the library – like Boost – also offers an interface to change the indexing properties of C++ arrays, i.e. column-major indexing instead of row-major indexing, arbitrary indexing ranges and the ability to do index-stepping. Again, these could allow us some limited ability to exchange arrays between C++ and Fortran through memory pointers given that there is some facility to handle dope vectors. Sadly, once more, such is not present... A logical approach for the Blitz++ developers, as their main goal is to provide a C++ alternative to Fortran, and to easy transition in porting, not to promote cross-language development.

So it turns out that either of these two tools might effectively help us – if we were attempting to do a complete port. Which is obviously not what we are looking for. If we want to reach our goal of easy cross-language design, it seems like we will have to provide the necessary functionality ourselves...

6.2.5 Custom solution: F90::Array for C++

A proper solution to the "pointer to array" problem when cross-language programming between C++ and F90 should address the following issues:

1. It should provide a proper type for Fortran pointers (i.e. dope vectors) which is usable in both F90 and C++.
2. We should be able to allocate to a pointer to array in either language.
3. We should be able to transparently pass pointers to array from either language to the other.
4. Ideally, we should be able to deallocate a pointer to array, allocated in either language, from the other as well.
5. We should be able to seamlessly pass arrays as members of structs or as function return types
6. It should support column-major order indexing in C++ as in F90

Some other likeable features would be that...

- it supports different indexing ranges in C++, as in F90
- it supports same-style indexing in C++ and in F90 (i.e. indexing multiple dimensions through a single '()' style operator)
- it supports different levels of portability, i.e. enable a developer to choose between a faster but low-level, compiler dependent implementation or a higher-level, compiler independent implementation as needed
- it supports the use of Fortran array-pointers at similar runtime speeds in C++ as in Fortran itself – ideally, again, with both compiler dependent and compiler independent implementations available
- it is able to accomplish this through a relatively straightforward interface – ideally, including a library in C++ should be enough to get us going

In the truly ideal world, a proper library included in both the C++ and Fortran code would allow us fully symmetric inter-changeability between both languages. Given the nature of F90, however, this would be next to impossible using just library calls. The Fortran 90 language offers little to no tools for the developer to do things like manipulating low-level data structures, or redefining the functionality of operators while effectively hiding a lot of behind-the-screens activity from the developer, without the ability to change the behavior of some of these (like the placement of hidden variables for string lengths, or the operation of dope-vectors).

However, C++ does allow for extensive language customization. Its detailed object-oriented mechanisms for defining and handling new data types should perfectly well allow us to create all types of structures for handling and manipulating dope-vectors. Proper coding should allow us to add a multitude of F90 operations to the new datatype, and have the developer differentiate between multiple implementations as needed.

The ability of C++ to overload operators should allow us to define the proper indexing operations, again – if necessary – supporting multiple implementations under full control of the developer. Specifically, it allows us to seamlessly provide column-major order indexing and customized index ranges for the C++ array-pointer datatypes.

Furthermore, C++ inherits from its predecessor, C, the capability to do precise, low-level memory management. This is exactly what we need for handling the dope-vectors on a basic level – grosso-modo without hidden side-effects (we will see one

notable exception to this principle in section 6.2.5.4).

Finally, the C++ template mechanism should allow us to do all of this for a range of array pointer types with different member types and extents, adapting the code generated at compile time transparently to the user.

While it would be virtually impossible to use a library mechanic for adding extensive cross-language functionality to Fortran, it should present less of a problem to do this for C++.

Indexing techniques

Row-major vs. column-major logical order indexing is one of the major points we need to address. Hence, we will have a more in-depth look at the algorithmical side of this problem before we elaborate on actual code designs.

Basically, when indexing an element in an array, we get an n-tuple of indices (n-dimensional) and need to turn it into an index into linear memory (1-dimensional). Solving this problem comes down to calculating a polynome of the form shown in figure 6.20.

Given:
- n dimensions, characterized by lower and upper bounds $(l_1, u_1)...(l_n, u_n)$
- n indices $i_1...i_n$
- a range of n logical order dependent factors $f_1...f_n$

- the linear index i_L for accessing an array element in a continuous linear memory range, starting to count from 0, can be found by solving the following polynome:

$$i_L = (i_1 - l_1).f_1 + ... + (i_n - l_n).f_n$$

Figure 6.20: Calculating a generic linear index into n-ary arrays

The value of the factors f is dependent upon the logical ordering, as shown in figures 6.21 and 6.22, which show the correct factors for calculating a linear index into row-major-ordered and column-major ordered arrays respectively. For those who were wondering about it's absence in the previous formula : this is where the upper bounds $u_{1...n}$ come into play.

Basic design

Given that we have shown that, in C++, it is possible to address the language interaction issue through pure library mechanics, this seemed like the most straightforward option to go ahead with an implementation. After carefully considering all the design parameters, we came up with a three-tier design for a library called, unimaginatively, "*arrays.h*", illustrated in figure 6.23.

The thee tiers of functionality offer the following options to the user:

1. Tier 1:
   The functionality in this tier allows a developer to access the dope-vector, and the array it points to, from the lowest level. This requires direct access to the

Given that:

- for an n-dimensional array and a given dimension x, the extent e_x (i.e. the number of elements along dimension x) is computed through the following formula:

$$e_x = u_x - l_x + 1$$

- the array uses **row**-major logical ordering

- the correct factors for determining the index in linear memory of a given element of the array are determined as follows:

$$\begin{aligned} f_1 &= e_n.e_{n-1}\left(...\right).e_2 \\ f_2 &= \frac{f_1}{e_2} \\ &\vdots \\ f_n &= \frac{f_{n-1}}{e_n} \end{aligned}$$

Figure 6.21: Calculating factors for determining a linear index into row-major ordered arrays

Given that:

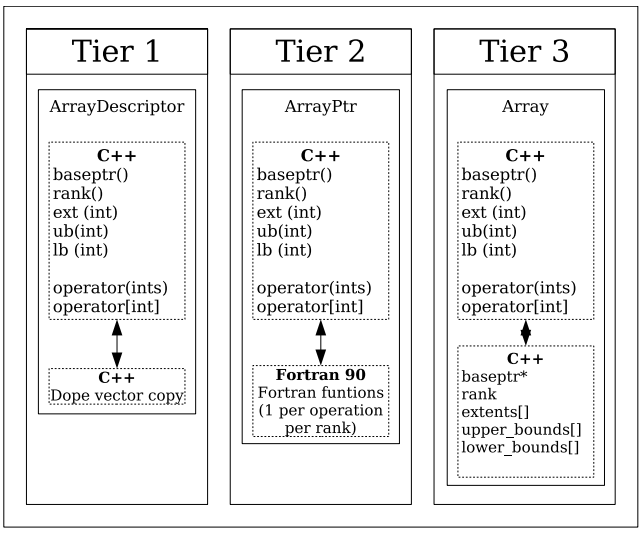
- the same definition of extent e holds as presented in figure 6.21
- the array uses **column**-major logical ordering

- the correct factors for determining the index in linear memory of a given element are determined as follows:

$$\begin{aligned} f_1 &= 1 \\ f_2 &= f_1.e_1 \\ &\vdots \\ f_n &= f_{n-1}.e_{n-1} \end{aligned}$$

Figure 6.22: Calculating factors for determining a linear index into column-major ordered arrays

dope-vector and its fields, the structure of which is fully compiler-dependent. All data needed for activities like member indexing, retrieving array data (extent, upper bounds, lower bounds etc.) is taken directly from the dope-vector. The functionality in this tier is available to the user by means of the



A high-level overview of our cross-language compatibility library for C++ and Fortran 90. Notice the three tiers of functionality available to the user – compiler-dependent, low-level actions through the F90::ArrayDescriptor; similar, high-level and portable but slower (cross-language) actions through F90::ArrayPtr; and another version of the same actions, faster than those available through F90::ArrayPtr, but without the ability to directly pass the array structure to an F90 call.

Figure 6.23: A C++/F90 compatibility library

F90::ArrayDescriptor class, which implements the correct memory structure for the dope-vector, and adds all methods needed to make use of it (element indexing, array meta-data enquiry) without the developer needing to directly access the dope-vector himself. It is possible to directly pass F90::ArrayDescriptors between F90 and C++ functions. In F90, they will map to an F90 pointer to array of the appropriate type.

2. Tier 2:
   This tier allows a developer to pass pointers to array from F90 to C++ and vice-versa, and offers methods for accessing members and array meta-data as in tier 1. The major difference with tier 1 is that here, we require only minimal compiler-dependence: there is no low-level manipulation of the dope vector. Functionality like allocation and meta-data retrieval in C++ is accomplished purely through cross-language procedure calls. This cross-calling does come at a price in terms of performance. The developer can access the functionality in tier 2 through the F90::ArrayPtr class. It is possible to directly pass F90::ArrayPtrs between F90 and C++ functions. Like an F90::ArrayDescriptor, an F90::ArrayPtr casts straight into an F90 pointer to

array without adaptions to the data structure.

3. Tier 3:
   Like tier 2, tier 3 allows developers to access member data and meta-data from F90 pointers to array with an absolute minimum of compiler-dependence. Contrary to the previous tier, the code accomplishes this goal by fetching all the necessary meta-data at instantiation, and caching it in the object's structure. The major advantage of this approach is that performance-wise, it is much more efficient than the tier 2 code, not requiring a cross-language procedure call for every member and meta-data lookup. A lookup in the C++ class structure is equivalent to a lookup in the dope vector in Fortran, and will be about equally fast. The downside to this method is that the dope vector needs to be stripped of the additional meta-data before it can be exchanged with F90. The tier 3 utility class is F90::Array. F90::arrays can be allocated and deallocated in C++ from scratch. F90::Arrays cannot be used as parameters to Fortran functions, members of a struct passed as a parameter and cannot be retrieved as a return value. It is, however, possible to initialize an F90::Array with an F90::ArrayPtr or F90::ArrayDescriptor, or to generate and F90::ArrayPtr or F90::ArrayDescriptor from one.

All of the previously described utility classes set and retrieve their meta-data (e.g. for indexing elements) at runtime. Naturally, this is the only available option if we are receiving arrays from F90 without knowing about bounds or extents at the compile time. (The type and nr. of dimensions of the array are *always* required at the compile time though, either in Fortran or C++.)

However, when doing cross-language development, we might encounter a situation in which we actually *do* know about the array meta-data in C++, while remaining unaware of such information in the F90 part of the code. Imagine, for instance, a multi-purpose array manipulation library written in F90 which can handle arbitrary sized and bounded arrays, by means of pointers. The library cannot possibly be aware of the kind of arrays it will receive. C++ code using the F90 library, however, *is* very likely to be aware of the array size and bounds at compile time.

A perfect solution in this case would be that the C++ code were ...

- able to exchange the array with the F90 code, and ...
- be able to make use of all its features as an F90 array, while ...
- retaining the performance advantages of being able to know all of the meta-data at compile time

Thanks to the relatively recent advances in the field of meta-programming with C++ templates [84], such a solution should now be possible. As F90 does not support generic programming, it is not feasible to do something similar from the F90 of the proverbial fence.

To the extent of our knowledge, no-one has yet attempted to design library supporting the functionality described above (i.e. allowing array pointer interchange with F90 and providing full F90 array functionality in C++ while still allowing for full compile-time optimization of array operations in C++). Even though we do not

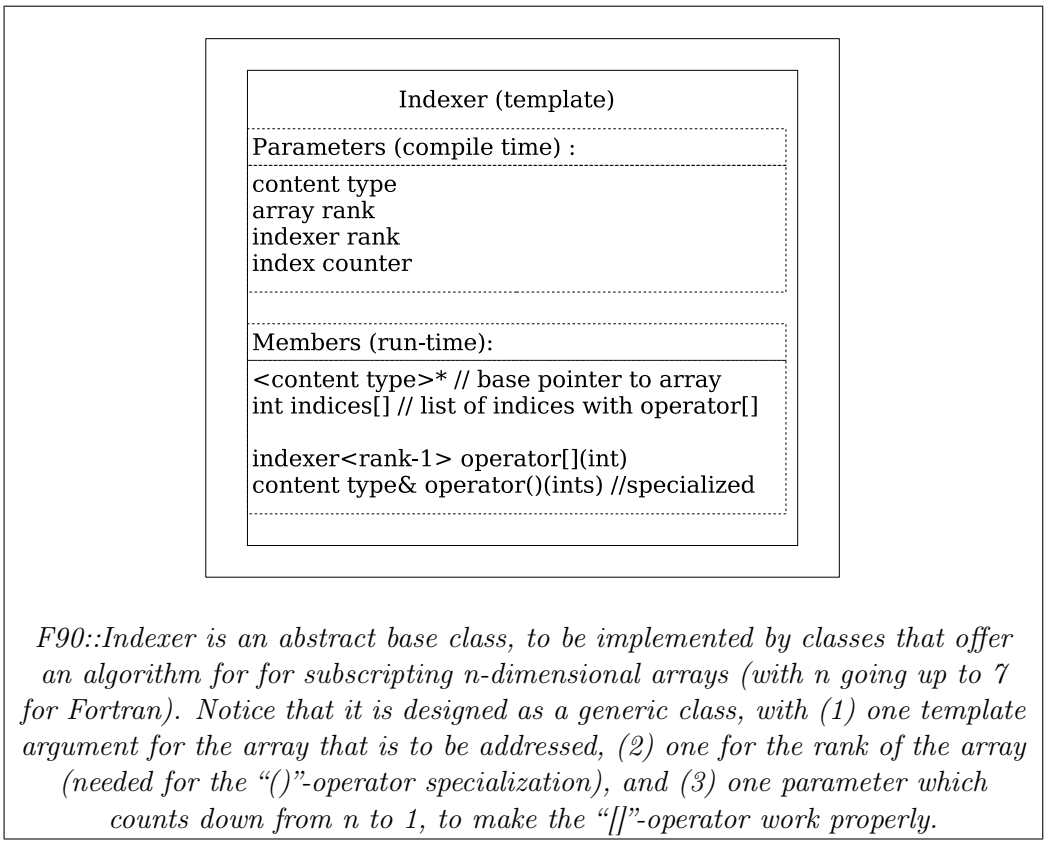
currently require such functionality for our cross-language development library, the possible applications are too tempting to ignore.

Therefore, we have developed the groundwork for an F90::Fast::Array class which uses extensive template meta-programming to maximize compile-time performance gains in C++, while still allowing for full interaction with F90. This 1) gives us an interesting reference result for later benchmarks of our arrays.h library and 2) provides for a solid springboard to do some interesting future work along the lines of the previously stated example...

In the next few sections, we will be looking at the components of the "arrays.h" library in closer detail.

F90::Indexer

One of the cornerstone concepts of the *arrays.h* library is that of the *Indexer*. F90::Indexer is a virtual base class that specifies an interface for addressing elements through indices (i.e. subscripting) in arbitrary-dimensional arrays. Figure 6.24 presents an overview of the class design.



F90::Indexer is an abstract base class, to be implemented by classes that offer an algorithm for for subscripting n-dimensional arrays (with n going up to 7 for Fortran). Notice that it is designed as a generic class, with (1) one template argument for the array that is to be addressed, (2) one for the rank of the array (needed for the "()"-operator specialization), and (3) one parameter which counts down from n to 1, to make the "[]"-operator work properly.

Figure 6.24: F90::Indexer

Each indexer is initialized with a reference to an array, and provides two operators for addressing that array's elements with indices: either 1) through multiple consec-

utive applications of the subscripting operator ("[]", as in C[++]), or 2) by means of an n-ary operator "()" with n = *array rank*, as in Fortran. The "()"-operator has been implemented through template specializations for arrays of 1 - 7 dimensions (the maximum rank for arrays in Fortran) for performance reasons. A less optimized and less compile-time safe, but shorter version could be written using default values. An indexer object is created by any array-like class whenever a user wants to address an array element by means of indices.

Specific indexer implementations can provide any subscripting algorithm desired by a developer. In *arrays.h*, we provide four implementations: 1) F90::FortranIndexer allows users to address any array-like structure in column-major order, as in Fortran, 2) F90::CIndexer – likewise – allows users to address any array in colum-major order, 3) F90::Fast::FortranIndexer addresses arrays in column-major order as well, but does so by casting to the appropriate C array type and using the native C "[]"-operator, which is optimized for fast element access at the compile-time and 4) F90::Fast::CIndexer does the same for row-major order array-traversing. All of the Indexer types in F90::Fast only work for array types in F90::Fast, as they require knowledge of upper and lower bounds for each array rank at compile time.

The indexer type is passed to all of the array types as a template parameter. Once declared, all indexing of elements in an array is done through that particular type of indexer. The indexer template parameter defaults to F90::FortranIndexer for regular array types, and F90::Fast::FortranIndexer for the array types in F90::Fast. It is not possible to switch indexer types at runtime, but this does not pose a problem as multiple objects of the different types in *arrays.h* can point to the same area in memory.

Indexers are created only by array-like objects – never by users themselves. An Indexer is only handled directly by users when returned as an rvalue by applying the "[]"-operator. The implementations in *arrays.h* allocate memory on heap for storing the indices when applying the operator "[]". Control over this memory range is passed from instance to instance, and finally cleaned up in a similar manner to the *<memory>* class *auto_ptr*. In general, the operator "()" will be more efficient, as it requires only a single function call for a full set of indices. The "[]"-operator takes one call per subscript to be resolved.

The code snippet in figure 6.25 illustrates the practical use of indexers.

F90::ArrayDescriptor

F90::ArrayDescriptor instances allow the developer to access array storage, referred to by a dope-vector, from C++ 1) without calling F90 functions, and 2) without storing any data beyond the dope-vector itself. The dope-vector is accessed through a struct which maps directly to the internals of the dope-vector as allocated by the F90 compiler.

This is the only code that actually makes use of parts of the Chasm 1.4.2 codebase – Chasm provides detailed information on the structure of the GNU F90 dope-vector. Of course, all of this code is compiler-dependent. When not using GNU, developers wanting to use F90::ArrayDesc will have to provide code for their particular compiler themselves.

```
// The insides of an arrays.h array-like class template :
//    * INDEXER is a template argument to the class, itself a
//      template, describing how to index the array (e.g. Fortran-
//      style, C++ style)
//    * MyType is a typedef, describing the class itself
//    * RANK is a template argument to the class, giving the nr. of
//      dimensions of the array accessed through an object of the
//      instantiated class template
//    * ReturnType is the return type of the indexer, i.e. An
//      either a value or an indexer with different parameters for
//      resolving the next dimension of the array structure.

// the indexing operator
ReturnType operator[](const int& index) {
      INDEXER<MyType,RANK,RANK,1> indexer(*this);
      return indexer[index]; // forward request to indexer
}
```

An indexer is instantiated by any of the classes in arrays.h whenever an element is addressed through indices using the "[]" or "()" operator. The subscripting operator returns an indexer as rvalue whenever subscripting an array (or an indexer) of any rank higher than 1.

Figure 6.25: Using F90::Indexer

An F90::ArrayDescriptor can be allocated by 1) passing it "blank" to an F90 function, 2) through an *allocate()* method, and, 3) setting it apart from any other class in *array.h*, it is the only type that can be initialized with a pointer to a previously allocated C++ array. It can be passed to any F90 function which takes a pointer to array type as a parameter, it is able to capture a pointer to array return value of an F90 function, and it can be passed as part of a struct to an F90 function. The basic layout of the function is described in figure 6.26.

F90::ArrayDescriptor supports operations for retrieving meta-data about the array (directly from the dope-vector) and for retrieving elements from the indexer by means of an indexer (again, using size and boundary data directly from the dope-vector).

Declaration-wise, this functionality is identically to that provided by all of the other array-like types in *arrays.h*, and hence it was a prime candidate for explicit specification through a pure-virtual base class. However, this approach would introduce polymorphism into the code-base, which leads to hidden changes in the basic in-memory structure of the F90::Arraydescriptor. This could make it impossible to properly pass the descriptor to and from F90 functions and, hence, cannot be considered an option.

The use of F90::ArrayDescriptor is illustrated by the code snippets in figure 6.27.

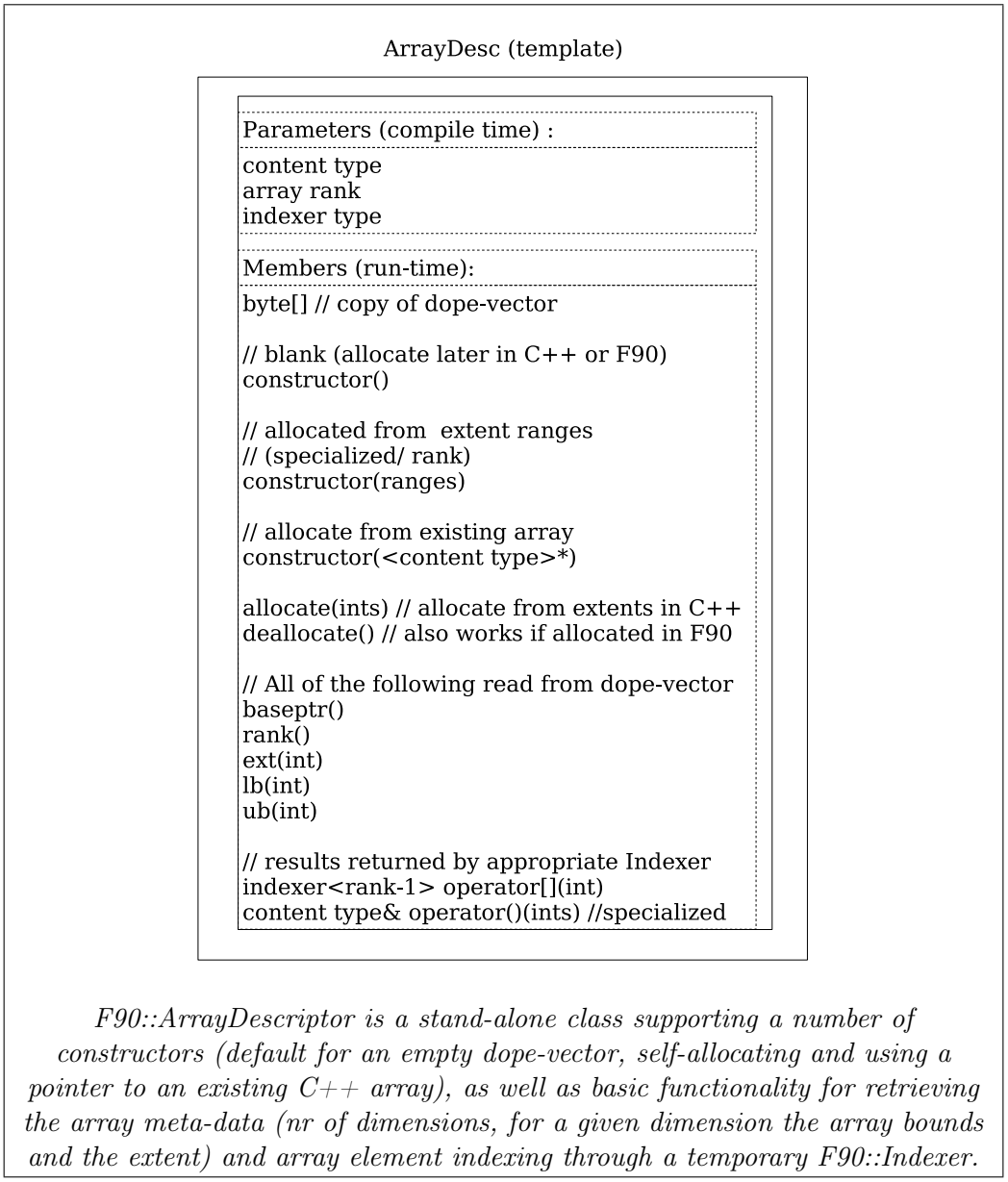


Figure 6.26: F90::Arraydescriptor

F90::ArrayPtr

F90::ArrayPtr objects combine the ability to be passed directly to and from F90 functions – as in F90::ArrayDescriptors – with a minimal dependence on dope-vector internals. Practically speaking, the only compiler-dependant information needed is the size of the dope-vector. All of the data required for the meta-data inquiry methods is obtained by calling an appropriate F90 function. The design of F90::ArrayPtr is superficially very similar to F90::ArrayDescriptor, as we can see in figure 6.28.

```
!!!!!!!!!!
! In F90:
!!!!!!!!!!

SUBROUTINE FORTRAN_ROUTINE(arrayptr)
     INTEGER, DIMENSION(:), POINTER :: ptr
     ALLOCATE(arrayptr(20))
     arrayptr = (/ I, I=1,20 /)
END SUBROUTINE FORTRAN_ROUTINE

/*********
* In C++:
**********/

extern "C" {
     void fortran_routine_(ArrayDescriptor ad);
}

// Construct and initialize array descriptors
ArrayDescriptor<int,1> ad1; // Index like F90, implicit.
fortran_routine_(ad1); // Initialize through F90 call.

ArrayDescriptor<int,1,FortranIndexer> ad2;// Index like F90, explicit
ad2.allocate(20); // Initialize through allocate()-method

int array[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
ArrayDescriptor<int,1,CIndexer> ad3(array); // Initialize with array
// Index above array as in C   must be declared explcitly!

// Demonstrate array access
cout << ad1[5] << endl; // Displays '5'. (Indexed in Fortran fashion.)
cout << ad2(5) << endl; // Displays '5'. (Indexed in Fortran fashion.)
cout << ad3[5] << endl; // Displays '6'! (Indexed in C fashion.)
```

Instantiating and using F90::ArrayDescriptor in practical examples, illustrating the 3 different construction methods and the use of both indexing operators.

Figure 6.27: Using F90::ArrayDescriptor

An F90::ArrayPtr can be initialized by using the allocate()-method, or it can be initialized by an F90 function or with data referred by an F90::ArrayDescriptor. The dope-vector can be retrieved from the pointer as an F90::ArrayDescriptor. F90 function calls are used by the meta-data retrieval methods on every access. F90 functions are called once to retrieve boundary data for indexing operations, which is then passed on to an indexer as in F90::ArrayDescriptor. This implies that every call to a meta-data method *and* every indexing operation will require one or more cross-language calls, which can be expected to accumulate a potentially serious overhead when it is being heavily used (these calls cannot be inlined or optimized through any other means). Hence, the class is most useful for passing arrays between multiple functions with only an occasional call to member methods.

The use of F90::ArrayPtr is illustrated by the code snippets in figure 6.29.

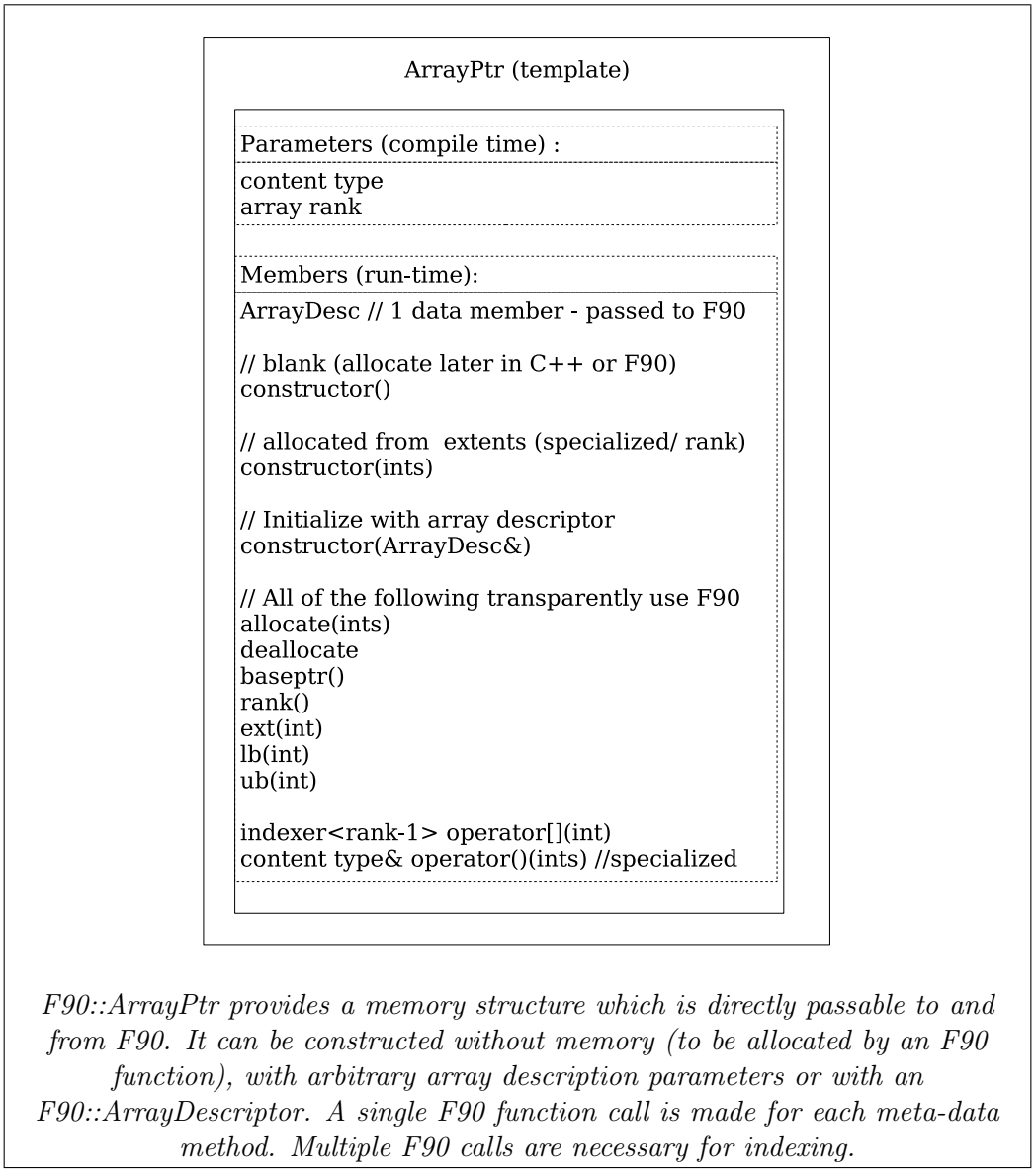


Figure 6.28: F90::ArrayPtr

F90::Array

The last *regular* array-like type provided by *arrays.h* is F90::Array. It provides the performance advantages of F90::ArrayDescriptor, but without the ability to pass it directly into or out of an F90 function. If we want to pass the array to F90, we need to obtain an F90::ArrayPtr to it by means of a method call. Its implementation is illustrated in figure 6.30.

F90::Arrays can be allocated during construction, or can receive memory from an F90::ArrayPtr or an F90::ArrayDescriptor. During construction, a local (private,

```
!!!!!!!!!!
! In F90:
!!!!!!!!!!

SUBROUTINE FORTRAN_ROUTINE1(arrayptr)
      INTEGER, DIMENSION(:), POINTER :: arrayptr
      ALLOCATE(arrayptr(20))
      arrayptr = (/ I, I=1,20 /)
END SUBROUTINE FORTRAN_ROUTINE

SUBROUTINE FORTRAN_ROUTINE2(arraydesc)
      INTEGER, DIMENSION(:), POINTER :: arraydesc
      ALLOCATE(arraydesc(20))
      arraydesc = (/ I, I=1,20 /)
END SUBROUTINE FORTRAN_ROUTINE
```

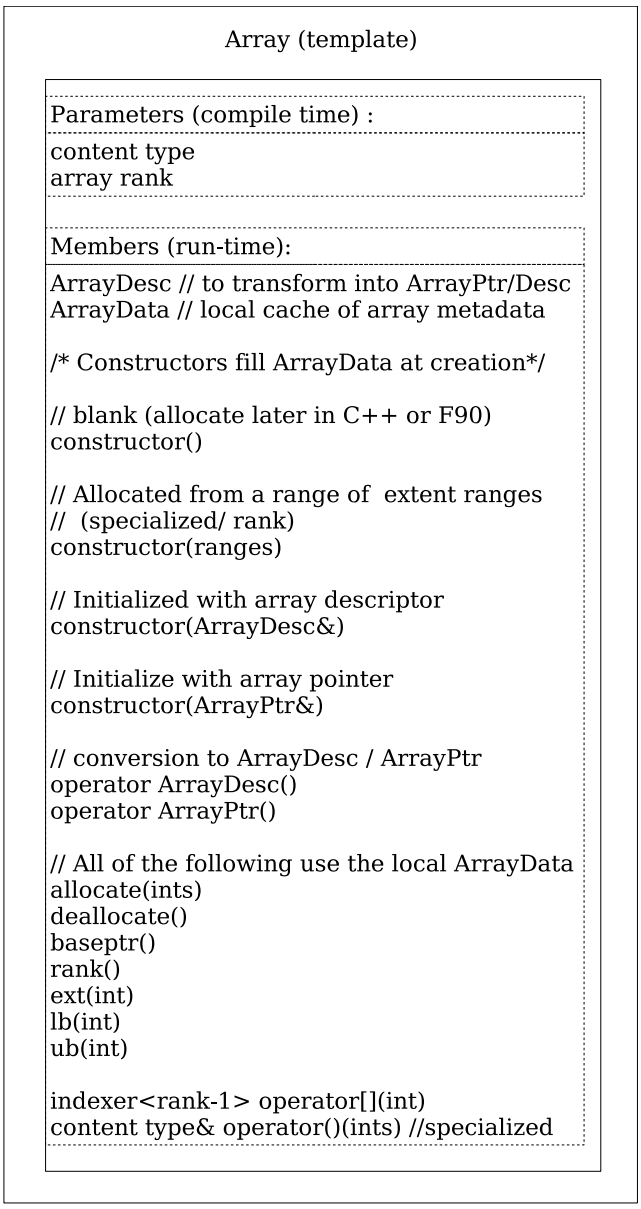
```
/*********
* In C++:
**********/

extern "C" {
      void fortran_routine1_(ArrayPtr&);
      void fortran_routine2_(ArrayDesc&);
}

// Construct and initialize arrayptrs
ArrayPtr<int,1> ap1;// Index like F90, explicit
ap1.allocate(20); // Initialize through allocate()-method

ArrayPtr<int,1,FortranIndexer> ap2; // Index like F90, implicit.
fortran_routine1_(ap2); // Initialize through F90 call.

ArrayDesc<int,1> ad;
fortran_routine2_(ad);
ArrayPtr<int,1,CIndexer> ap3(ad); // Initialize using ArrayDesc.

// Array access as with ArrayDesc
```

Instantiating and using F90::ArrayPtr in practical examples, illustrating the use of constructors and indexing operators.

Figure 6.29: Using F90::ArrayPtr

object-internal) cache is filled with all of the available meta-data by means of F90 function calls. From this point on, all meta-data methods retrieve their data from local cache, and local cache is also used to initialize indexers.

If we want to pass the array to F90, we can pass it to a function taking a F90::ArrayDescriptor or F90::ArrayPtr parameter without adding extra code, as long as the F90 function does not change anything about the pointer-to-array's meta-data properties (e.g. by deleting and / or reallocating). This is accomplished by making



Like F90::ArrayPtr, F90::Array internally uses an F90::ArrayDescriptor to refer to the dope-vector. The major difference here is that all meta-data is retrieved using F90 calls only once during construction. The data is then cached locally within the F90::Array object and used for every meta-data method-call or indexing operation.

Figure 6.30: F90::Array

use of the C++ conversion operator mechanism and C++ implicit casting semantics.

It is extremely important at this point, to stress the importance of the constantness requirement for the array properties. When using implicit casting, the F90::Array is turned into an F90::ArrayPtr(/ArrayDescriptor) *rvalue* which is by nature temporary. This means that any changes to the converted value will *not* have repercussions on the F90::Array itself, thereby potentially rendering it corrupt. It is possible to indicate to C++ that this requirement has been fulfilled, by properly adjusting the F90 function declarations in the C++ headers. Function calls with *const F90::ArrayPtr&* or *F90::ArrayDescriptor&* parameters will allow for the implicit conversion, function calls with just *F90::ArrayPtr* or *F90::ArrayDescriptor* won't. Naturally, the developer should take great care here to make sure that the declaration corresponds with the actual behavior of the F90 function: as we mentioned earlier 6.2.3.1, the validity of the C++ headers cannot be verified by the compiler.

Proper constructors, implementing appropriate conversion semantics, make it similarly possible to make this principle work symmetrically for return values from F90 functions. If we are not able to pass or receive F90::Arrays directly, at least conversion to and from "passable" types can be resolved "mostly" transparently.

The use of F90::Array is demonstrated through the code snippets in figure 6.31.

F90::Fast::Array

Finally, the classes in the F90::Fast namespace provide an alternative implementation of the above functionality, but optimized by means of template meta-programming techniques. The prime requirement for use of these is that we know all meta-data-properties of the F90 array at compile-time, in all C++ compilation units. Hence, they are prime candidates for use in new OO code which uses F90 only for calls to legacy libraries, while doing all of its memory management in C++. This functionality has been encapsulated in the F90::Fast::Array class.

We want this class to accomplish the following goals:

1. In C++, using F90::Fast::Arrays should offer performance comparable to native C++ arrays.
2. In C++, using F90::Fast::Arrays should offer us capabilities comparable to F90 arrays.
3. In F90, an F90::Fast::Array instance should be treated as a regular pointer to array.

The problem with C++ template meta-programs is that, even though they offer very powerful options for advanced compile-time optimization, the techniques used make them 1) harder to read than regular code, and 2) *a lot* harder to describe.

Basically, we are using parameterized C++ types (*class templates*, i.e. a genericized user-defined type) as *meta-objects* – "objects" that are instantiated at compile time and *meta-object classes* (a class that encapsulates a meta-object for passing it to other template constructions). The compile-time attributes of these are being manipulated by template expressions within other class templates that are used as if they were functions – so-called *meta-functions* (class templates that directly generate a compile-time result) and *meta-function classes* (classes that contain a meta-function class – again, encapsulation allows for easy passing of the meta-function to other

```
!!!!!!!!!!
! In F90:
!!!!!!!!!!

SUBROUTINE FORTRAN_ROUTINE1(arrayptr)
    INTEGER, DIMENSION(:), POINTER :: arrayptr
    ALLOCATE(arrayptr(20))
    arrayptr = (/ I, I=1,20 /)
END SUBROUTINE FORTRAN_ROUTINE1

SUBROUTINE FORTRAN_ROUTINE2(arraydesc)
    INTEGER, DIMENSION(:), POINTER :: arraydesc
    ALLOCATE(arraydesc(20))
    arraydesc = (/ I, I=1,20 /)
END SUBROUTINE FORTRAN_ROUTINE2

/*********
* In C++:
**********/

extern "C" {
    void fortran_routine1_(ArrayPtr&);
    void fortran_routine2_(ArrayDesc&);
}

// Construct and initialize arrays
Array<int,1> a1;// Index like F90, explicit
a1.allocate(20); // Initialize through allocate()-method

ArrayPtr<int,1> ap;
fortran_routine1_(ap);
Array<int,1,FortranIndexer> a2(ap); // Initialize using
ArrayPtr.

ArrayDesc<int,1> ad;
fortran_routine2_(ad);
ArrayPtr<int,1,CIndexer> a3(ad); // Initialize using ArrayDesc.

// Cannot pass directly to F90! Convert to ArrayDesc/Ptr first.
a1.deallocate();
a2.deallocate();
fortran_routine1_(a1.getArrayPtr()); // Excplicit conversion.
fortran_routine1_(a2) // Conversion using conversion operator.

// Array access as with ArrayDesc
```

Instantiating and using F90::Array in practical examples – we demonstrate construction with new memory and existing memory behind an F90::ArrayPtr and an F90::ArrayDescriptor. The snippet also demonstrates explicit casting to F90::ArrayPtr& and implicit conversion using a conversion operator.

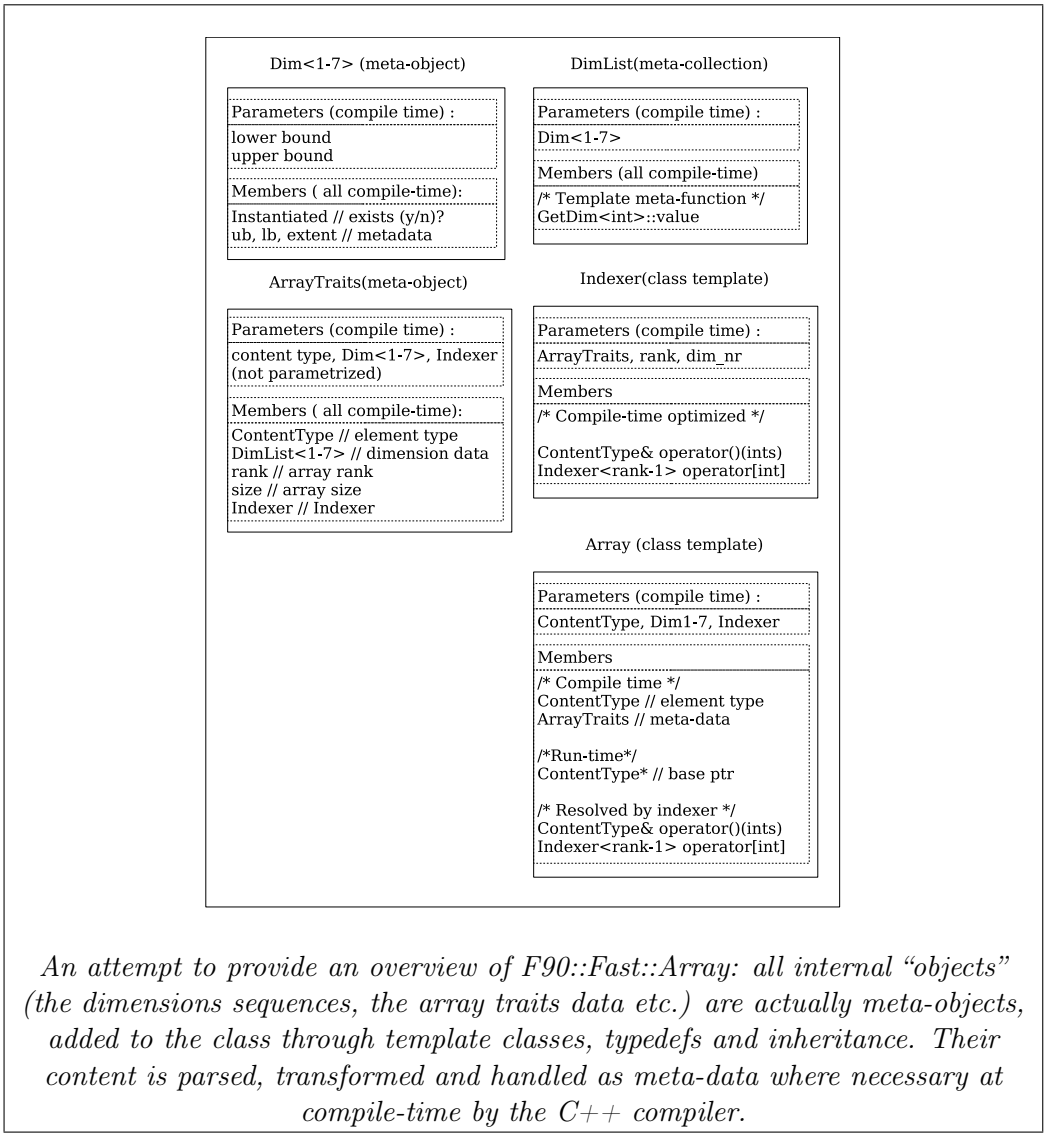
Figure 6.31: Using F90::Array

meta-functions). These "return" by instantiating other meta-objects that can be further used as the "result" of the meta-function. Typedef statement are used in this context as a type of assignment to capture and pass back intermediate and final results, and public inheritance is used for referencing and forwarding meta-data.

Although this is a most fascinating topic, we will not go deeper into the general techniques of C++ template programming here – interested readers can find more

details in other, more in-depth and focused sources like [84].

In other words: the tools of generic programming are being used in a way which – quite radically – differs from their classical application, and regular illustrative techniques, like class diagrams, will not offer a proper or understandable overview of the techniques applied. As a matter of fact, C++ template meta-programming constructs bear greater resemblance to expressions from functional programming languages than regular generic programming in the C++ context. Still, figure 6.32 shows an attempt to sketch out the basic structure of the F90::Fast::Array class.



An attempt to provide an overview of F90::Fast::Array: all internal "objects" (the dimensions sequences, the array traits data etc.) are actually meta-objects, added to the class through template classes, typedefs and inheritance. Their content is parsed, transformed and handled as meta-data where necessary at compile-time by the C++ compiler.

Figure 6.32: F90::Fast::Array

Basically, we create an F90::Fast::Array type by providing the class template with a series (one for each rank of the array) of *Dimension* meta-objects (other

template types), initialized with an upper and a lower bound. These dimension meta-objects are passed as template parameters to the instantiation of the particular F90::Fast::Array, together with a *Type* meta-object parameter which encapsulates the content type of the array, and an optional *Indexer* template. This last template parameter is actually a *meta-function* which creates a proper Indexer type at compile-time, using the other init parameters of the array class to produce the necessary compile-time optimized indexing algorithms. An example of F90::Fast::Array instantiation is shown in figure 6.33.

All of the array meta-data is retained and accessed as part of the type's traits, and no run-time caching is needed – all meta-data inquiry functions return values which are "baked into" the type at instantiation of the Fast::Array template. The dimensional traits of the array are kept in a meta-sequence – a type structure representing a compile-time collection of meta-objects, that can be indexed by means of a constant int. Their functionality is basically comparable to that of some existing run-time collection types, and advanced implementations of such meta-data collections even offer functionality like associative mapping (type-to-type), insertion/deletion and meta-data iterators [86].

Similarly, when instantiating a Fast::Array, a customized indexer type is also generated with all the data required for mapping the indices to array elements hard-wired into the algorithm. Two implementations have been provided for comparison. The first uses essentially the same algorithm as the regular array-like types described over the course of the previous sections, but without the need for retrieving the data from the dope-vector, Fortran or object cache. The second implementation goes a step further, and internally maps the Fortran array type to its transposed native C++ equivalent to virtually address it by column-major order. Thus, the correct element is retrieved by indexing the array in "reverse" order. (C++ requires that all of the extents of a multi-dimensional reference/pointer to array are known at compile-time, except for the last dimension – hence, this technique could not be applied in the previous cases.) A performance comparison of both implementations is presented in section 6.2.5.8.

The meta-programming approach has multiple advantages that allow us to accomplish All of our previously stated goals.

As we no longer need to look up the array meta-data for calculating indices, performance will be vastly better than that of F90::ArrayPtr, and even better that that of F90::Array or F90::ArrayDescriptor. Generally, performance should improve even more as the number of dimensions goes up. Looking ahead at the results of our performance test, we happily observe that the transposed-matrix type algorithm effectively succeeds at producing performance nearly equal to native C++ arrays under similar circumstances – with an interesting twist... again, we refer to section 6.2.5.8 for more details. Hence, our first requirement – near-optimal performance in terms of speed – has been successfully addressed.

The meta-programming approach allows us to apply techniques similar to those used in the earlier referenced Blitz++ [81] library. The Blitz++ array operators (matrix addition, matrix scalar multiplication etc.) have shown comparable performance to similar operations in Fortran [83]. It also transparently offers other Fortran-like

mechanics, not available for regular native or STL C++ array types, like array slicing. Although such support has not been added to F90::Fast::Array at the time of writing, Blitz++ has proven that it is possible, and the necessary techniques are known and well developed. Such an addition to *arrays.h* would provide an interesting avenue for future work. All in all, we can state that, even if the second goal has not been *exactly* reached yet, it should not prove to be a problem to do so.

To our great satisfaction, our third point of interest – the ability to directly pass Fast::Arrays to F90 functions and vice-versa – can also be marked for success, almost as a side-effect. As all meta-data has now been incorporated in F90::Fast::Array's complex type structure, there is no need to "pollute" our runtime in-memory object structure with cached data, hence allowing us to hold on to the pure dope-vector memory without having to resort to compiler-dependent manipulations. As it is for F90::ArrayPtr and F90::Array, the only compiler-dependent information we need to know is the size of the dope-vector.

Examples of use for F90::Fast::Array are shown in figure 6.33.

Performance tests

In order to compare the multiple implementations, we ran a limited benchmark measuring basic performance when read- and write-accessing array data for arrays of multiple dimensions, under two circumstances: consecutive access (optimizable through low-level caching techniques) and random-access (not optimizable). The results of the consecutive access test are shown in figure 6.34. Results for random access are shown in figure 6.35.

Benchmarks are performed for all array types, for integer arrays of 1 to 4 dimensions. For consecutive access, we are working with extents of 11 in each dimension (ranging -5 to +5). For random walks, we use bigger arrays to try and make sure that caching will have as little influence as possible. We try to keep close to 10,000 elements in the whole array (i.e. (10000), (100,100), (10,10,100), (10,10,10,10)).

The run-time allocated array types use the regular F90::FortranIndexer for retrieving elements. The compile-time optimized F90::Fast::Array are compared using 2 different indexers: F90::Fast::FortranIndexer and F90::Fast::NativeFortranIndexer. All code was compiled using the GNU C++ compiler version 4.1.

The results are largely as expected: F90::ArrayDescriptor and F90::Array perform about equally fast, while F90::ArrayPtrs lag behind. Either of the three runtime allocated array classes perform worse than native C++ arrays.

The F90::Fast:Array type shows a different image though. We compare the use of F90::Fast::Arrays with 2 different indexers : F90::Fast::FortranIndexer uses the algorithm presented in section 6.2.5.1, while the indexer F90::Fast::NativeFortranIndexer uses the native C++ array transposition optimization for addressing the array elements described in section 6.2.5.7.

First conclusion: using the optimized NativeFortranIndexer, F90::Fast::Arrays perform about equally fast as native C++ arrays for consecutive accesses. Arrays using the regular Fast::FortranIndexer perform less optimal in this case. When measuring results for random-access however, the results are reversed, on *both* counts: F90::Fast::Arrays using F90::Fast::FortranIndexer actually perform faster than either

```
!!!!!!!!!!
! In F90:
!!!!!!!!!!

SUBROUTINE FORTRAN_LIBRARY_ROUTINE(arrayptr)
      INTEGER, DIMENSION(:), POINTER :: arrayptr

      ! Initialize - size of array pointed to is unknown by F90
      ! It is implicitly looked up in the dope vector by SIZE.
      arrayptr = (/ I, I=1,SIZE(arrayptr) /)
END SUBROUTINE FORTRAN_LIBRARY_ROUTINE

/*********
* In C++:
**********/

extern "C" {
      void fortran_library_routine_(Fast::Array&);
}

// Construct an 1D Fast::Array of 20 integers,Fortran-style
// indexing using the transposing indexer
Fast::Array<Type<int>, Dim1<1,20>, TransposeFortranIdexer> a;
a.allocate();  // No arguments needed - extents known!

// Directly pass a to F90 library function for arrays of generic
// size
fortran_library_routine_(a);

// Array lookup directly resolved with C++ arraylookup
cout << a(3) << endl; // displays "3"
```

Instantiating and using F90::Fast::Array in practical examples, ...

Figure 6.33: Instantiating and using F90::Fast::Array

Fast::Arrays using NativeFortranIndexers *and* native C++ arrays.

In other words, we can choose a proper Indexer method depending on the kind of access to the array that we can expect for a given problem, and – under GNU 4.1 at least – actually gain performance which is better than that for native arrays!

6.2.6 F90Arrays unit testing issues

Unexpectedly, one of the more complex issues in developing the F90::Fast types *arrays.h* turned out to be one only tangential to the given problem: unit testing.

While writing the different array classes, it turned out that numerous small mistakes in the implementation could result in major problems when applying the code to our new MPI-enhanced quantum physics code-base. Hence, decent unit testing is paramount. For the regular, run-time allocated array types, this does not pose a problem: the tests can be performed with meta-data that is generated within a

	Read Speed			
	1D	**2D**	**3D**	**4D**
C	54 ms	70 ms	112 ms	149 ms
Fast:NativeFortranIndexer	54 ms	92 ms	141 ms	149 ms
Fast:FortranIndexer	62 ms	185 ms	282 ms	294 ms
ArrayDesc	230 ms	841 ms	864 ms	894 ms
Array	141 ms	702 ms	794 ms	901 ms
ArrayPtr	334 ms	1145 ms	1346 ms	1276 ms

	Write Speed			
	1D	**2D**	**3D**	**4D**
C	107 ms	91 ms	144 ms	166 ms
Fast:NativeFortranIndexer	117 ms	101 ms	142 ms	170 ms
Fast:FortranIndexer	126 ms	124 ms	170 ms	200 ms
ArrayDesc	139 ms	1063 ms	834 ms	875 ms
Array	129 ms	1061 ms	834 ms	882 ms

Benchmarking arrays.h *for consecutive reads and writes. Notice that the results pretty much follow expected trends.*

Figure 6.34: Consecutive reads and writes from and to *arrays.h*.

	Read Speed			
	1D	**2D**	**3D**	**4D**
C	1065 ms	506 ms	572 Ms	962 ms
Fast:NativeFortranIndexer	1101 ms	998 ms	607 Ms	1440 ms
Fast:FortranIndexer	340ms	392 ms	389 ms	586 ms
ArrayDesc	3261 ms	1073 ms	1128 ms	3601 ms
Array	2238 ms	1064 ms	1191 ms	3901 ms
ArrayPtr	4289 ms	2241 ms	2346 ms	4521 ms

	Write Speed			
	1D	**2D**	**3D**	**4D**
C	10076 Ms	14913 ms	16151 ms	12700 ms
Fast:NativeFortranIndexer	10282 Ms	15318 ms	11410 ms	13132 ms
Fast:FortranIndexer	1076 ms	1099 ms	1157 ms	1249 ms

Benchmarking arrays.h *for "random-walk" reads and writes. The result turns out pretty interesting...*

Figure 6.35: Random reads and writes from and to *arrays.h*

set of loop statements. For the F90::Fast::Arrays, things are different however : as we need the array meta-data at compile-time, we cannot produce it through regular loops. For our test however, it is essential that we try out a range of meta-data values including border cases...

Generic meta-loops

In other words, we need a loop mechanism that produces integer values at compile-time. Basic compile-time loop techniques were the subject of some of the first investigations into C++ template meta-programming [84]. Some of these techniques were

later formalized in libraries like Boost::MPL [86] and Loki [87]. While these libraries offer many solutions for the use of compile-time collections through "meta-iterators", none of them present a truly generic, simple technique for using templates to emulate complex run-time loops. Hence, there was some more template meta-programming over the horizon.

After some coding work, we came up with a solution which allows us to write "meta-loops" as illustrated by the code snippet in figure 6.36.

As can be seen in figure 6.36, we provide a template class called *Loop* which takes four template parameters:

- a meta-object acting as *meta-iterator*, corresponding to the loop variable in a regular for-statement; the meta-object can have any structure that the other parameters to the loop are able to deal with; the example in figure 6.36 provides each of the nested meta-loops with a meta-object of type *Int*, each of which is initialized with a value
- a meta-predicate class: a class containing a meta-function to act as *loop* condition – it "checks" the meta-iterator at the start of each "iteration step"
- another meta-function class acting as *update expression*, analogously to a for-statement
- a function-template class: this class contains a function template which is instantiated with a meta-object for each "iteration" in the loop structure, and corresponds to the *loop body* of a for-statement.

Basically, *Loop* generates an *Iteration* meta-object for each meta-iteration step – this is used as a template parameter to create one unique instance of the *loop body* function template per meta-iteration. The generation of *Iteration* objects is limited by the initialization value passed to the *meta-iterator*, and the result of evaluating each generated *Iteration* meta-object using the *meta-predicate*. Each of the generated *loop body* function templates can access the corresponding *meta-iterator*, and all the compile-time data it contains, through its template parameter. To trigger the proper evaluation of the meta-loop at runtime, the *call()* member function of *Loop* is called, triggering a continuous chain of function calls to each of the function template instances.

When properly implemented, most compilers will inline and concatenate all of the generated functions, similar to what happens when a compiler applies *loop unwinding*, but in this case consistently and automatically.

The *Loop* class itself has been implemented as a function template class, allowing multiple *Loops* to be "nested". In this case, the *body* function template can access the contents of each of the surrounding *Loops* using the *Parent* member of each "internal" loop in proper succession. A helper function template class *TraverseIterators* has been provided as well, which goes through all of the generated meta-iterators for each iteration step, in sequence from the inside out, for an arbitrary number of nested loops.

The main issue with using these meta-loops is one of performance: not runtime performance (which should actually be optimal), but *compile-time* performance. Each meta-loop iteration results in the creation of multiple meta-objects (types), which severely impacts compilation speed. The problem grows when nesting loops,

as the amount of generated meta-objects will effectively grow exponentially. It gets even worse if, within the meta-loop body, we happen to be working with types that use heavy template meta-programming themselves, as the amount of generated code gets still bigger.

In practice, it was not practical to use meta-loops for testing *F90::Fast::Array* code using more than 4 dimensions. Luckily for us, this is the maximum amount needed by the quantum physics code. Code using 5 dimensions or more would result in compilation sequences lasting longer that 24 hours at best – mostly, such attempts would lead to a near-freeze of the machine due to kernel swapping (because of memory exhaustion), or the compilation process would simply be prematurely halted by the Linux memory management daemon.

Even for simple meta-loop bodies, compilation will quickly slow down. While stand-alone and single-nested meta-loops compile swiftly, double- and triple- nested meta-loops already take more than half a minute, and a quadrupally nested meta-loop will take over an hour to compile.

Another limitation when using meta-loops is that many compilers impose a (compiler-dependent) maximum to the number of template instances that a meta-program can recursively create. Most compilers will allow this limit to be extended using command-line parameters though.

Due to all of these issues, meta-loops are probably not suited as a potential general replacement for loop-unwinding compiler optimizations. Still, for the purpose of unit testing the meta-programming code in *F90::Fast::Array*, they are invaluable.

6.2.7 Summarizing C++ and F90 integration

We have been able to show that it is possible to write practical and performant C++-F90 cross-language code. When using the *F90::Array* and *F90::Fast::Array* types provided in the *arrays.h* library, the task can actually be accomplished with a minimum of effort and compiler-dependent code. But the task does require careful consideration of a number of issues concerning compiler-dependence and performance. Generally though, we can consider the problem solved. Developing *arrays.h* has not been a straightforward feat though. And there are still some issues than can benefit from future work.

The main body of extra work remains on the F90 side of the issue. Basically, while templates allow us to generate appropriate code for any type of *F90::Array* in C++, we cannot do the same in F90. For now, the F90 functions used by the *F90::Array* types have all been written by hand for each type of array used in the quantum physics code.

This aspect would greatly benefit from an automation process similar to that used by Babel or Chasm. Both of these tools use a pre-processing step for the whole process though, which – given the functionality in *arrays.h* – seems somewhat overkill (at least for the C++-F90 case). A relatively simple, limited, C++ template-like token substitution pre-processor for F90 should be able to do the job without any additional effort on the C++ side.

Another aspect which needs some further investigation is that of indexation. In the current version, all of the indexer classes in *arrays.h* have been provided with

specialized implementations for each rank supported, up to 7 dimensions (the F90 maximum). (The indexers are still fully member type-agnostic though.) It would be useful to do some study as to the possibility to simplify this implementation, and open it up to an arbitrary number of indices (e.g. by relying on loop unrolling optimizations, meta-loops or other meta-programming techniques, recursive inline function calls with the possible addition of tail recursion optimization, etc...).

6.2.8 Porting to C++

We will now illustrate how the above conclusions apply to our example legacy application.

The basic layout of the hybrid C++-F90 code-base is illustrated in figures 6.37 and 6.38. A step-by-step overview follows.

The general layout of the parallel program at this stage is as follows:

- For the master code:
  1. Initialize:
     Basic information about the MPI virtual machine, such as the available amount of slave processes, is retrieved right after startup in the C++ main body. The rest of the initialization process (opening and reading the config files, creating the unformatted result files, allocating the necessary memory structures on heap, ...) is taken care of by the same F90 function that was used for this purpose in the previous version. All of the allocated memory structures are passed back to C++ as *F90::ArrayPtrs*. For performance reasons, those that get further use in the C++ code are rolled into *F90::Arrays*.
  2. Broadcast:
     The slave initialization phase is resolved by the same F90 function that handled the task in the previous re-factoring iteration.
  3. The main loop works exactly as in the previous version, yet has now been completely translated into C++, but for the recombination process. The program still goes through similar phases:
     (a) Send initial data:
     The master uses messages to send tasks to all slaves using C++ MPI calls. Note that we are effectively mixing F90 and C++ MPI calls in a single program. According to the standard, this should not pose a problem, and indeed, MPICH-2 resolves the mixing without a hitch.
     (b) We then go through the following sequence:
        i. Wait for result data:
        As in the previous re-factoring step, but using C++ MPI calls.
        ii. Send new data
        id.
        iii. Recombine
        The C++ code calls the existing F90 function to resolve this phase. Again, we pass the dynamic structures as *F90::ArrayPtrs*, extracting these from *F90::Arrays* where necessary.

4. Master finalize slaves:
   When all tasks have been distributed, slave processes are sent a message to shut them down.
5. Master finalize:
   When all tasks have been sent and all result data has been received, all data structures allocated in C++ are cleaned up, before calling the existing F90 code to write out the final results (which must be done in F90 for convenient access to the unformatted data files) and clean up any structures allocated internally in F90.

- For the slave code:
  1. Initialize:
     The slave waits for the broadcast initialization data from the master, and uses the received data to perform its own initialization phase, as previously. This part consists of hybrid C++ and F90 code, reusing part of the existing code for allocating some of the dynamic structures.
  2. Main loop:
     (a) Wait for task:
     As in the previous version, but translated into C++.
     (b) Calculate:
     The slave calculates results for the task, using an F90 function call. The partial result matrices are returned as *F90::ArrayPtrs*.
     (c) Send results:
     The slave sends back the partial result matrix to the master.
  3. Finalize:
     When the slave receives a null message instead of task data, it exits the main loop and starts the finalization process, again a hybrid C++-F90 process.

6.2.9 Performance

Once more, we make a calculation for $L = 0$ and $K = 0, 2, 4, .., 10$, a range of $l_1 = l_2 = 0, 1, .., K/2$ values, and 45 t values. As we can gather from the resulting graph, the results are mostly identical to those presented in figure 6.5.

As we can see, the new graph is very comparable to the previous one, proving that we have been able to accomplish our goal: to create a C++-F90 cross-language solution for our problem, which combines the object oriented qualities of C++ with the legacy code-base in Fortran without incurring any serious performance overhead. Due to the fact that we haven't yet made any drastic changes to the basic call structure of the code, profiling with MPE produces similar results as well.

6.2.10 Conclusions on Fortan and C++ integration

During the previous re-factoring step, runtime profiling revealed scalability problems with the existing code structure. We came to the conclusion that, to fix the problem, we would need to parallelize the recombination process – this implied some major

changes to the code-base. On top of this, it was foreseeable that – due to the nature of the changes needed to make the software fault-tolerant in later stages of the refactoring process – even greater changes would have to be made to the software structure.

While considering the amount of change necessary, we made the observation that strategic use of object oriented (OO) language features could be useful to help us further in our endeavor. However, since F90 lacks OO features, this would entail porting to another programming language. The modularity added in the previous step could help us ease the problem, if we were only able to a) retain those pieces of code which would not see much change in F90 (actual calculation and recombination of partial result matrices, creating and writing unformatted result files), while b) only porting those parts which would be seeing a lot of rewrites in the coming steps (structural code for controlling which calculations were performed when, and on which hosts in which particular order, further messaging code, ...).

Due to the known compatibility between F90's predecessor F77 and the C programming language, C' s object oriented successor C++ was chosen as the logical target language for the port.

This naturally leads us to answer our next general research question: *With regard to Fortan-based legacy code: what needs to be done to make easy integration of said code with modern C++-based code work in a user-friendly manner.*

It turns out, that, even though the techniques for bridging between C and F90 have been well documented by others, cross-language development using C++ and F90 is not as simple as it seems. This is mostly due to the "obscured background activity", performed by the computer to support the F90 dynamic memory features.

Software solutions exist that claim to ease the bridging process (Chasm, Babel, elements of Boost and Blitz++), but it turns out that either a) they do not support all the necessary features (both versions of Chasm, Babel), or that b) they don't even allow for cross-language development at all, but rather aim to provide Fortran-like tools for development in pure C++ (Boost, Blitz++). Moreover, the only truly applicable existing solution – the Chasm tool – takes a rather heavy-weight approach to the matter on both the C++ and the F90 ends of the issue, while most of the work – at least on the C++ side of things – could probably be done simpler through C++'s built in generic/generative programming features (templates). Another shortcoming of the Chasm tool is that it doesn't allow the developer to choose between a reasonable customizable range of compiler dependent / compiler independent features.

From this, we concluded that the best approach to solving our problem would probably be to write a new cross-development tool from scratch. We wrote this tool with the following goals in mind: 1) it should enable us to use pointers and dynamic memory across C++ and F90 in a transparent and interchangeable manner, 2) it should enable developers to use the matrix-like structures that are stored within these dynamic memory ranges in a similar manner in both C++ and F90, 3) it supports different levels of portability, carefully balancing compiler-independence vs. performance in a manner which is customizable to some degree by the developer and 4) ideally, it should be able to accomplish this purely by means of a library mechanism as much as possible.

By creating the *Arrays.h* library, we have shown that all of these requirements are attainable, delivering a balanced set of options to the developer through careful design and extensive use of the C++ features wrt. generic-, generative- and meta-programming. Minor elements are left as future work, but *Arrays.h* currently provides for all our needs to complete the porting process, and more. While *Arrays.h* provides similar functionality to Chasm in some areas, it is more extensive, comprehensive, user-customizable, and to the point for the C++-F90 case than the latter. As well, it provides for a simpler, more straightforward development and build process. It accomplishes these goals by applying both a design and a set of techniques which significantly differ from those used in the Chasm tool, firmly putting apart *Arrays.h* as a solid, unique creative effort.

But, as we mentioned, the attained results go beyond the needs of the quantum physics project: specifically, we created the *F90::Fast::Array* library, with which were capable of producing some very interesting performance figures for cases in which array meta-data is known at compile time in C++, but not in F90. These results were accomplished by thoroughly applying template meta-programming techniques in C++. Although these techniques aren't directly applicable to the problem at hand, we have illustrated cases in which the given functionality definitely would be useful to developers. To the extent of our knowledge, no-one has yet studied or produced a tool for C++-F90 cross-language development with the given qualities.

After re-factoring the F90 code-base, it was split up into 1) a "permanent" F90 part which re-uses the existing code-base, and 2) a "mutable" C++ part, which will be seeing some heavy modification over the course of the coming re-factoring steps. Performance testing produced solid evidence that we were able to reach our software engineering goals of adding the OO features without sacrificing runtime performance, as required. Hence, all of our stated goals for this re-factoring step have been accomplished, and we are now ready to move on to the next.

6.3 Re-factoring for performance

Before we set out on answering our final research question, we first return to the point in our preliminary refactoring process where we "side-tracked" for a moment to investigate the Fortran-C++ integration issues. Specifically, before we start thinking about the fault-tolerance issue, we first need to fix the performance problems that we noted for values of L greater than 2.

6.3.1 Rationale

Now that we have paved the way for making significant changes to the code-base, beyond just adding parallelism to the calculation step, let us go back to our conclusion from re-factoring step 1. During the profiling phase of that step, the data for calculations with $L = 0$ was most promising. However, visual inspection of the gathered data for $L = 2$ showed an problem with the calculators. Specifically, it turned out that all of the slaves were going through large periods of idle time while

waiting to send their result data back to the master.

When looking at the master process, we could see that these idle periods lined up with recombination activity in the master process. Remember : the current process does both task distribution and recombination in the master process. Practically: every time that a calculator sends back a result, the master receives it as soon as it can, sends back a new task and then recombines the received results. This will immediately provide the sending calculator with new data. However, other calculators that want to send back results will have to wait until the master is done with recombination. And as long as the calculators are unable to let go of their results, they do not receive new tasks - and hence remain idle.

The slowness of the recombination process for higher values of L can be attributed to two major components. The first is straight mathematical complexity: recombination times, in terms of CPU cycles, will significantly rise for higher values of L – this is inevitable. The second issue lies with file operations: in the current implementation, the master does recombination for all the result matrices. As the result matrices will typically be too large to all fit in memory at the same time - most certainly for large values of the input parameters – results have to be switched from and to file whenever a partial matrix comes in for a result matrix that differs from the currently active one. These operations on disks are slow, and since the files are too large to properly fit into operating system cache, they can not be sped up.

In a naive solution scenario, the file operations could be omitted if we were to calculate separate result matrices sequentially (while still calculating the partial results in parallel). This, however, would not solve the complexity issue – at one point, the length of the calculations will once more lead the master to starve the calculators.

The only option which allows for maximum scalability at the master side is to completely decouple recombination from task distribution, i.e. to parallelize the recombination process. Again we could take this on in two ways. In both scenarios, the current master process would only be doing task distribution – initializing processes with data at the beginning and waiting for calculators to send it a new task request, answering the request with a new task. A calculator, once it is done with a task, would send the partial result matrix to a recombinator process, and then request a new task from the master.

As to the recombinators, once more, the naive option is to "sequentialize" the mechanism: provide one recombinator process, and calculate the different result matrices sequentially – again, while still calculating the partial result matrices in parallel. As complexity grows however, we will bump into similar problems as before, where calculators will be waiting for the recombinator – instead of the master – to receive their results while the latter is busy recombining previous partial results. While it does short-circuit the file "swapping" process, we would be no better of than in our previous naive setup.

This leads us to the conclusion that an optimal solution would be to provide one recombinator process per result matrix to be computed, with different partial result matrices for different recombinator processes to be calculated in parallel. When carefully interleaving tasks for different result matrices, this setup should maximally

distribute the workload among all of the recombinators, minimizing any bottlenecks (and hence idle time) towards calculator processes.

Buffering could probably provide even extra protection against recombination bottlenecks in any of these scenarios, but we will not be experimenting with the buffered modes of MPI at this moment.

6.3.2 Design basics

The basic structure laid out in the introduction leads us to the following practical design, as illustrated in figure 6.40.

1. The master still generates input data sets for the calculation function which runs in the calculators, but dispatching tasks is its only function now; it servers all calculators with tasks at startup and waits for new task requests coming in from calculators.
2. The calculators calculate tasks distributed to them by the master process, sending on the result to an appropriate recombinator. As soon as the send operation returns, the calculator requests a new task from the master. The master, not being burdened by recombination can now promptly send the new task.
3. The recombinators wait for partial result matrices from the calculators. As soon as they receive a result, it is recombined into the result matrix. By interleaving tasks for different recombinators, all recombinators can be kept active and bottlenecks on behalf of any single recombinator can be minimized.

6.3.3 The new code layout

Probably the most interesting aspect of the new design is that the ^5H calculation software, at this point in the re-factoring process, has definitively ceased to be a straightforward master-slave type of application. In the coming re-factoring steps (when adding fault-tolerance), we will see that the application will be moving even further away from its simple roots in step 1.

A more detailed overview of the new work-flow is presented in figure 6.41:

- For the master code
    1. Initialize and broadcast:
       As in previous versions, read init data from file and broadcast. The master has to resolve three broadcasts now, as the recombinators require initialization data that the calculators don't an vice-versa. The broadcasts are resolved in 3 steps: first through the global communicator for the common initialization data, and through 2 specific (new) communicators for the calculators and the recombinators respectively. These new communicators are used throughout the whole process now to logically separate communication between master<->calculators and master<->recombinators respectively.
    2. The main loop still works as previously, though we don't have a recombination phase now. Hence, the main loop now goes through the following phases :

   (a) Send initial data:
   The master uses messages to send tasks to all slaves using MPI point-to-point calls.
   (b) We then go through the following sequence:
      i. Wait for a task request from a calculator:
      Once all the slaves have received one task, the master waits for requests for new tasks from any slave.
      ii. Send new task
      When a new task request from a slave comes in, that slave is sent another task, or "done" when there are no further tasks. If all calculators have been sent "done", exit the main loop.

3. Finalize:
   The master waits for the result data from the recombinators. When a result has been received, it is written to an UNF file. When all recombinators have returned a result, finalization proceeds as previously, except for those parts that were split off into the recombinator.

- For the calculator code:

1. Initialize:
   The calculator waits for the broadcast initialization data from the master, both over the global and the calculator-specific communicator, and uses the received data to perform its own initialization phase as previously.
2. Main loop – the calculator now goes into a main loop consisting of:
   (a) Wait for task:
   The calculator waits until it receives a tasks from the master
   (b) Calculate:
   The calculator calculates results for the task, producing a partial result matrix.
   (c) Send results:
   The calculator sends the partial result matrix to the appropriate recombinator
   (d) Request a new task
   The calculator requests a new task from the master.
3. Finalize:
   When the calculator receives a null message instead of task data, it exits the main loop and starts its finalization process as previously.

- For the recombinator code:

1. Initialize:
   The recombinator waits for the broadcast initialization data from the master, both over the global and the recombinator-specific communicator.
2. Main loop – the recombinator now goes into a main loop consisting of:
   (a) Wait for result:
   The recombinator waits until it receives a result from a calculator.

(b) Recombine:
The recombinator recombines the result with the matrix.

3. Send result matrix
When all partial result matrices have been received, the recombinator sends the total result matrix to the master.

4. Finalize
Part of the cleanup that was originally part of the master is now done in the recombinator.

6.3.4 Profiling

Once more, we will try on a number of job runs for $L = 2$. This yields the graph shown in figure 6.42.

A we can see in the figure, re-factoring has brought about a drastic change in the performance behavior of the application. The large "idle gaps" which characterized the application in its previous version have disappeared, and the graph now shows a clean sequence of computational states in both the calculators and the recombinators. No harmful synchronization of any kind is happening now, and all processing resources are being used to their fullest, except perhaps for the master process. This will not be a problem, as it implies that we should be able to run the master in parallel with another process (calculator or recombinator) without negatively influencing resource efficiency.

6.3.5 Conclusion and future work

After remodeling the code-base for easier re-factoring in the previous step, we have addressed the inefficiencies in the recombination process. After considering a number of alternatives, we decided to solve the problem through further parallelization. Specifically, we decided to farm out recombination to a number of dedicated processes, one per result matrix. By interleaving tasks for different result matrices, we should be able to recombine in parallel and minimize bottlenecks. In addition to this, we now permanently keep the result matrix in memory, which means no more costly swap-to-disk type operations. This results in an added improvement to CPU efficiency. After another round of tests, we can positively claim that the problems related to recombination-bottlenecks have been successfully solved.

We will not be investigating the scalability of the current recombinator implementation for higher values of the input parameters. However, if more bottlenecks would show up, there are still some possibilities for improvement left. We leave a closer study of this potential up to future work.

One approach would be to add buffering for the calculator->recombinator send operation. The most obvious way to add user-controlled buffering would be through MPI's buffered send API. It mostly limits user control to buffer allocation and management only, but for most applications this will be enough. If this were not enough, other approaches are possible though they are all tricky.

Explicit user-managed buffering could be done by adding a dedicated buffering process between the calculator and the recombinator, but a) this adds another send

operation to the work-flow and b) requires careful planning of which machines the buffering processes will run on, because – for high values of the input parameters – they could require sizeable amounts of memory. This approach has the advantage of applying parallelism not just to CPU, but also for memory resources. However, careful study would be necessary to judge the performance loss associated with the extra send vs. simple write-and-read-back operations to disk. (Memory problems could be alleviated – of course – by using OS-controlled swapping combined with something like a solid state disk-like device – however, these devices are prohibitively expensive and would lead us away from our explicit focus on lower-cost devices.)

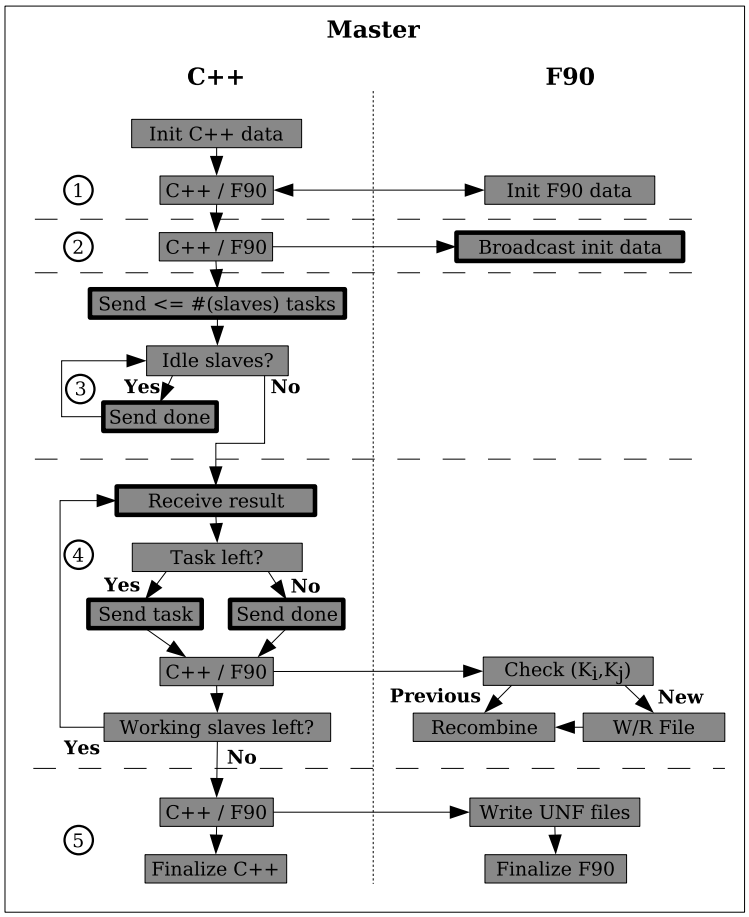
Another way to do it is to receive and buffer the result from the calculator in a separate thread within the recombinator itself. Care should be taken with this approach though, as the MPI standard explicitly does not enforce thread safety on any implementation. As long as all of the MPI-based operations (initialization, receive from calculator, send to master) are kept in one thread, while the rest (actual recombination) is done in another, this should not prove to be a problem. Of course, buffering would be strictly limited to the memory resources that are available on the machine running the recombinator.

A last method we thought of, not directly involving buffering, would be to parallelize the recombination process even further. Practically, this would involve utilizing multiple recombinator processes for a single result matrix (instead of the 1 result matrix - 1 recombinator approach we use now), adding a single (or staged series of -) "meta-recombination" step(s) at the end to produce a single result matrix from the multiple "sub"-recombinators. This is, of course, only possible as long as the recombination process is commutative. This approach, however, will also require more CPU resources – either to be added, or taken from calculator resources.

```
/* A simple loop : loop over a const range and display results */

// Function template object, to be executed by meta-loop.
// Display the current value of the iterator of the surrounding
// meta-loop.
class LoopEcho {
public:
      template<class LOOP>
      static void call() {
            cout << LOOP::Iter::Value << " "1;
      }
};

// The loop: initialize a meta-iterator of type Int<> with value
// "0"; call LoopEcho::call() and increment the meta-iterator as
// long as it remains smaller than "3". Notice the similarity in
// structure with a regular loop.
Loop< Int<0>, LessThanInt<3>, IncrementInt<1>,
      LoopEcho
>::call();

cout << endl; // Displays "0 1 2".

/* A nested loop */

// Display the sum of the current values of the two surrounding
// meta-loops (LOOP::Iter and LOOP::Parent::Iter)
class LoopSum {
public:
      template<class LOOP>
      static void call() {
            cout << LOOP::Iter::Value + LOOP::Parent::Iter::Value
<< " ";
      }
};

// The loop : again, initialize two meta-iterators of type Int<>
// and call LoopSum::call(), incrementing the iterators as in a
// regular loop until the meta-predicates return false.
Loop< Int<0>, LessThanInt<3>, IncrementInt<1>,
      Loop< Int<1>, LessThanInt<3>, IncrementInt<1>,
            LoopSum
      >
>::call();

cout << endl; // Displays "1 2 2 3 3 4".
```

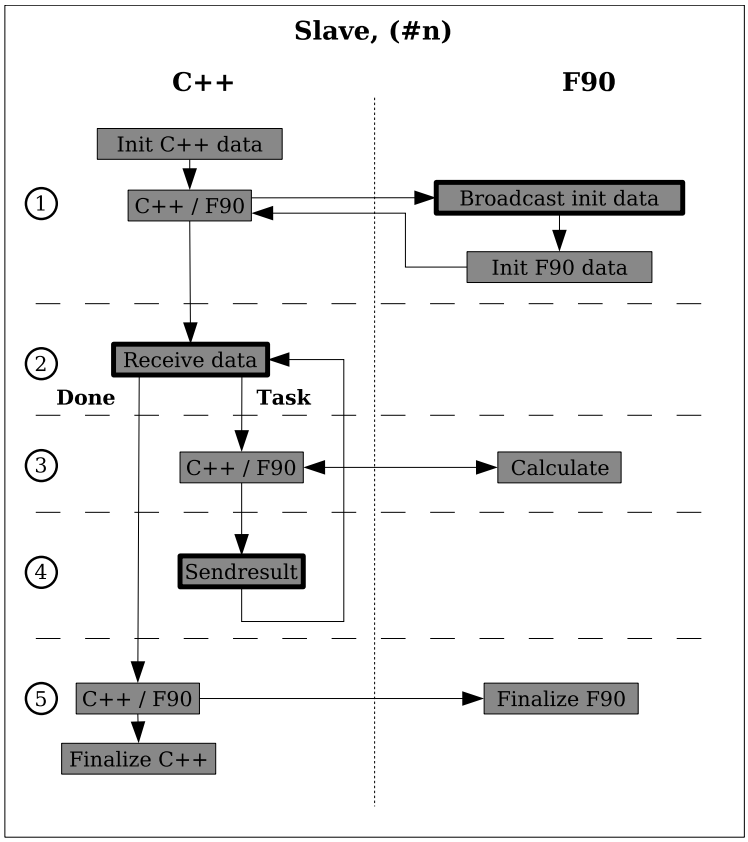
C++ meta-loops in action. Note that the meta-loop takes parameters in a fashion analogous to a regular for-loop. Note also that meta-loops can be nested in order to produce complex sequences of compile-time data.

Figure 6.36: Meta-loops



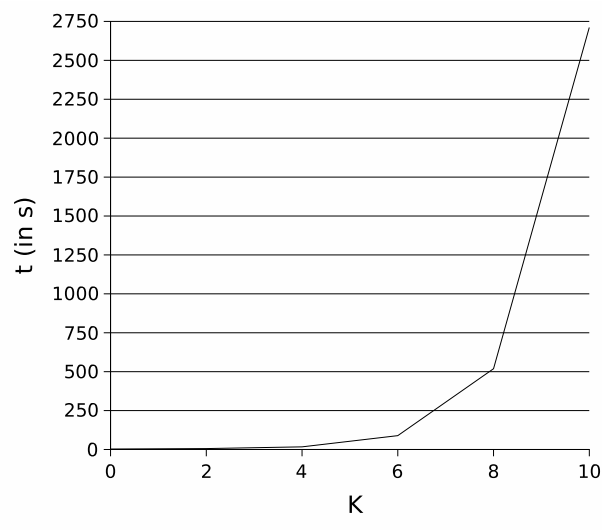
*We now have two bodies of code: one part in C++ and one in F90. The **master** code 1) initializes [C++ and F90], 2) performs an initialization broadcast [F90], 3) sends out initial tasks to all slaves and then 4) goes into the main loop. Therein, it a) waits for slaves to send back results [C++], b) either sends a new task (when available) or "done" to the originating slave, c) recombines [F90] and d) either waits for another result (while pending) or leaves the main loop. When completely done in the main loop, the master 5) finalizes [C++ and F90].*

Figure 6.37: High-level overview of the master F90&C++ code-base after the second re-factoring step



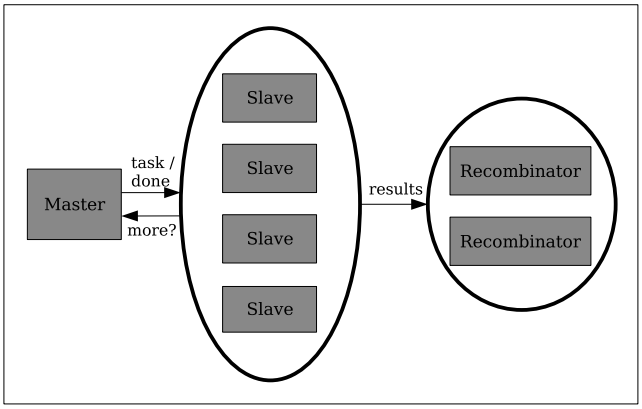
*The **slave** code 1) initializes from own data, and data received from the master [C++ and F90], then 2) waits for a task [C++], 3) calculates a partial result matrix [F90], 4) sends it back [C++] and repeats the process until told by the master to quit. It then 5) finalizes [C++ and F90].*

Figure 6.38: High-level overview of the slave F90&C++ code-base after the second refactoring step



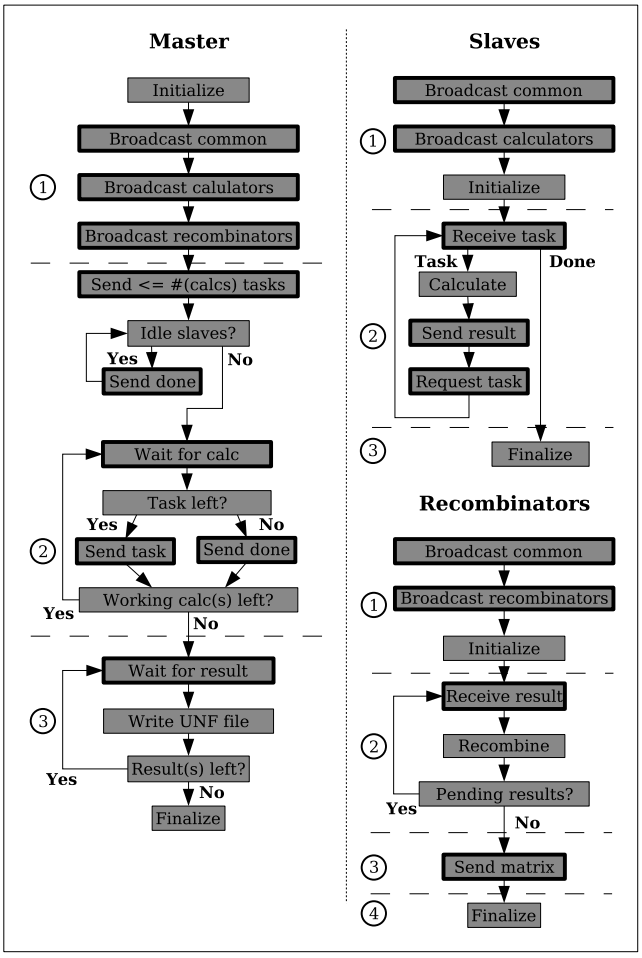
Running the C++-F90 hybrid, again for $L = 0$ and $K = 0, 2, 4, .., 10$, *on the same cluster, using the same setup as in figure* 6.5.

Figure 6.39: A performance graph for the C++ port



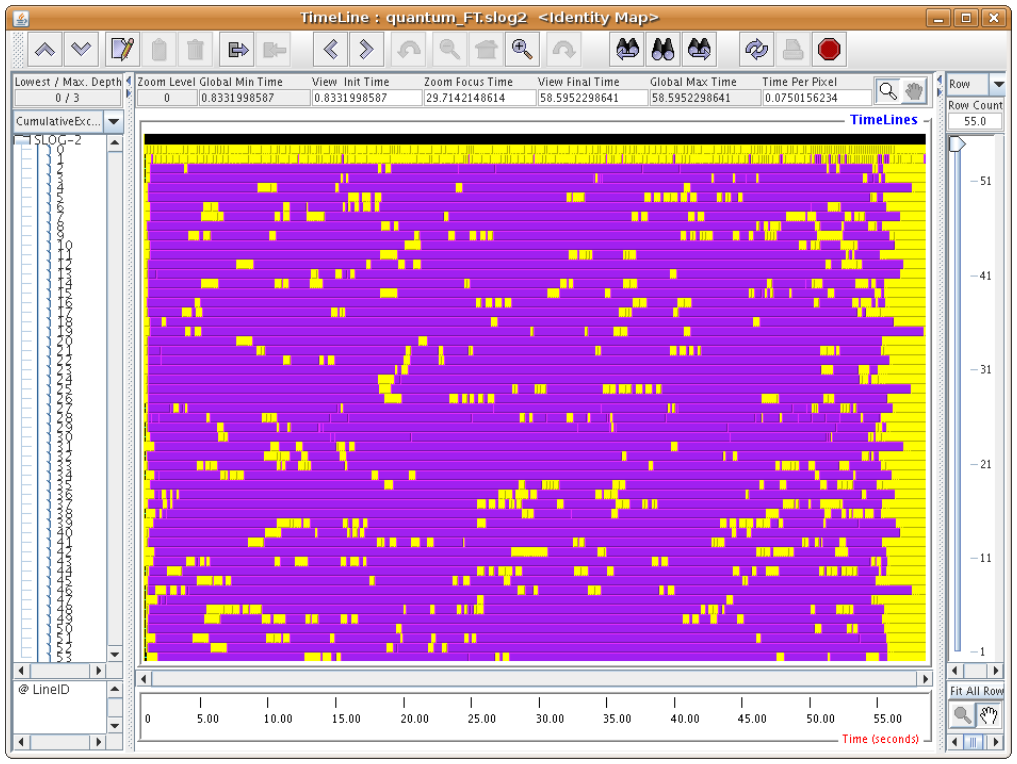
The ^5H *calculation with a master (dispatcher), calculators and recombinators. The partial result matrices generated by the slaves are sent to an appropriate recombinator (instead of the master) now. As soon as the send operation returns, the calculator requests a new task from the master, which it should promptly receive.*

Figure 6.40: The ^5H calculation with parallelized recombination



A detailed overview of the code structure after the third re-factoring step. The separation between C++ and F90 code is retained identical from the previous version, but recombination is now split off into its own process. Writing to / loading from a partial result file is no longer necessary in this version – all data is retained in memory until the end, when results are sent back to the master and written to a Fortran unformatted file in one pass.

Figure 6.41: A detailed overview of the new code using recombinators.



Another calculation for $K = 6$,$L = 2$. Notice the difference with the previous experiment. Parallelization of the recombination process has clearly worked to alleviate the previous problem.

Figure 6.42: Profiling the program for $L = 2$ with parallelized recombination

CHAPTER 7

Re-factoring for Fault-tolerance

Finally, we have arrived at the part where we try to find a suitable solution to our last research question: *With regard to MPI-based software: what – if any – changes need to be introduced to make it fault tolerant, given an MPI implementation that supports it. What would a generic "cookbook" for creating fault-tolerant MPI-based software look like..*

We will first illustrate the process of adapting an existing MPI-based body of code to FT-MPI as an ad-hoc development, by implementing the procedure upon our example application. Along the course of this final refactoring traject, we will come to discover some of the quirks and pitfalls that one will encounter while implementing fault-tolerance in any kind of MPI-based software.

From this process, we will eventually deduce a sort of cookbook, which should prove an aid while refactoring any other piece of software for FT-MPI.

7.1 Implementing Fault-Tolerance with FT-MPI

Now that we have solved the initial parallelization issues, we will investigate the changes needed to make the software fault-tolerant. We resolve this step, basically, by exchanging MPICH-2 for FT-MPI as our MPI implementation, and adding/changing code where necessary to deal with faults wherever they may occur as allowed by the FT-MPI spec.

Before we start out on the details of porting the quantum physics application, we will have a look at some basic issues that we encountered during our design with – and practical use of – FT-MPI. These involve both problems with FT-MPI (in its currently available version), and the essential fact that – at each MPI call – we have to take into account that it might fail.

Going deeper into this section, we will see that developing fault-tolerant MPI software requires a specific frame of mind which is quite different from the thought process that we've been using over the course of the previous sections. Specifically, the work-flow of the software - up until now – used to be more or less linear.

However, when using FT-MPI, each MPI call in the execution sequence can result in three "valid" states, only one of which should eventually lead to abortion of the work-flow. Either 1) the call encounters no problem in the MPI subsystem, and resolves successfully (returning MPI_SUCCESS), or 2) the MPI runtime discovers a recoverable problem in the distributed VM, and returns with the appropriate error message (MPI_ERR_OTHER) or triggers an error handler, or 3) the MPI runtime is confronted with a non-recoverable problem and has to abort, as it would in a regular MPI program. As FT-MPI, by default, handles errors through error-handler MPI_ERRORS_RETURN (not using FTMPI_COMM_MODE_ABORT of course), we should add explicit handling code for this case into the program's work-flow – except if we'd be using an error handler of our own design.

Each of these return states should now be accounted for: a return value of MPI_ERR_OTHER will change the regular control flow, and an unrecoverable error should now explicitly terminate the program.

Over the course of the coming sections, we will attempt to develop a methodic procedure to take on these issues in a consistent, logical manner.

7.2 Problems with the current FT-MPI implementation

First, we will discuss some of the practical issues that we came across while using FT-MPI in practice. Our first point will prove to be quite fundamental. The others mostly concern bugs or unimplemented features in the current version of the software.

7.2.1 Lack of automatic synchronization at key points

The main problem concerns synchronization at certain key points during program execution: initialization, finalization and recovery. Recovery will automatically lead to full synchronization between all processes in the VM – as would be expected, both under recovery modes AUTO and MANUAL. This is not the case, however, for the other two program states.

Even though MPI defines two functions that are required to be called by any valid MPI program (MPI_Init and MPI_Finalize), the MPI standard does not require these to be synchronizing (in fact, it doesn't even mention any developer's advice to do so). Following the standard, FT-MPI doesn't synchronize on these two calls either.

This, however, creates problems with buffering. When sending a series of small-sized messages, the following work-flow scenario will quite frequently happen in the context of a practical program when taking failures into account (as we witnessed over the course of multiple experiments):

1. The master initializes (without synchronizing with the other processes).

2. The master starts sending tasks.
   (a) Each send is buffered (as it is small).
   (b) After being buffered, the MPI call returns without synchronizing with the other processes.
   (In standard mode, an MPI implementation can choose between buffered or synchronized sends according to the needs of the situation.)
3. After sending all tasks, the master calls MPI_Finalize (again without synchronizing).
4. The master stops without ever having synchronized – it is only kept alive to send the buffered messages.
5. The other processes start up, initialize and start running.
6. Sometime during operation, a recoverable error occurs.
7. All processes enter recovery mode, except for the master, as it has called MPI_Finalize and stopped.
8. Recovery is, hence, unable to synchronize.
9. Thus, recovery fails!

7.2.2 Solutions for the synchronization problem

There are a number of possible remedies for this problem.

A naive approach would be to require synchronized mode for every operation : this would force the send calls, which are buffered under standard mode, to block until a receive is posted by a calculator. There are two problems with this work-around, however. From a theoretical p.o.v., it is unacceptable that the MPI implementation would not be allowed to optimize its use of resources in standard mode. (This is, in fact, the very reason why a standard mode exists to begin with.) From a practical angle, we are confronted with the fact that synchronized send mode has not been implemented yet for the current version of FT-MPI.

Another approach would be to make MPI_Init and MPI_Finalize into synchronized calls. This, however, would clash with the following declarations from the MPI Standard Document Version 1.1:

1. regarding MPI_Init: “It must be called at most once; subsequent calls are erroneous.”
2. and regarding MPI_Finalize: “”Once this routine is called, no MPI routine (even MPI_Init) may be called. The user must ensure that all pending communication involving a process completes before the process calls MPI_Finalize.”

Especially the last paragraph suggests that another solution should be searched for. Of course, we have to recognize that it was written without the circumstance of fault-tolerance in mind, and that it might have to be revisited if FT-MPI were ever to get foothold. It is still, by far, the most logical place to do the necessary synchronization.

This means we will have to do our own synchronization at the beginning of the program to “ensure that all pending communication involving a process completes before the process calls MPI_Finalize.” Given the above, such a proper synchronization call should be used in every FT-MPI program. We believe that proper wording as such, at least, should be added to the FT-MPI manual.

A similar synchronization will be performed at the end of the work-flow to allow the master to safely retrieve diagnostics from the written UNF files; with the certainty that all of them have been finished by the recombinators.

The prime candidate for such a synchronization call would be MPI_Barrier. There is one problem with MPI_Barrier however: it is buggy and does not function in this version of FT-MPI. (As a matter of fact, it did not work reliably in version [LOOK UP] of MPICH-2 either, as we experienced from multiple experimental runs!)

The candidate we eventually settled upon was MPI_All_Reduce: complete VM-wide synchronization is absolutely necessary for this call to produce a correct result, and it works properly under both FT-MPI and MPICH-2. By enclosing all MPI calls (except MPI_Init and MPI_Finalize) between two MPI_All_Reduce calls, FT-MPI was properly able to deal with any recoverable error.

7.2.3 FT-MPI mplementation issues

There are some other issues with the current FT-MPI.

Collective recovery modes are currently not implemented, and we have seen broadcasts produce unpredictable results when using FTMPI message mode CONT. Hence, we will not be using this mode in the port. Also, it gives us a good reason to reduce the amount of broadcasts in the software.

As we mentioned earlier, another unsupported feature of FT-MPI is synchronized communication. This will force us to use some unorthodox "trickery" in our upcoming designs, as it will turn out that we have a need for it (lucky for us, it will turn also out that we only need one-sided synchronization – i.e., only one side has to make sure that an MPI operation with the other has properly finished; otherwise, the problem would not be solvable).

Another shortcoming is the lack of support for C++ exceptions as specified in the MPI1.1 C++ language binding. This means that we will have to retain the C bindings for MPI, despite working in a C++ environment. Some lack of forethought in a similar vein can be seen in the FT-MPI C++ binding for MPI_Init. While the C binding has an int return value, the C++ binding (according to the MPI standard) does not have a return value, instead using an exception to signal error. However, in FT-MPI, the return value of MPI_Init doubles up as a means to signal a starting process whether it has been newly started, or has rather been restarted as a result of a recovery procedure. This functionality is not accessible under the C++ binding without change to the MPI standard definition (e.g. by adding another exception type to signal a restart).

RECOVERY_MODE_AUTO has currently not been implemented yet either - we will have to use MODE_MANUAL, which implies a copy (by means of MPI_-Comm_Dup) of the communicator MPI_COMM_WORLD, which synchronizes all processes in the VM. Otherwise, recovery should be identical.

7.3 Software development with FT-MPI

With the practical issues behind us, we can have a look at the other points that require consideration.

Specifically, we will have to make changes to the basic structure of the software to accommodate for fault-tolerance. Given the FT-MPI approach, it is now possible for MPI functions to return a value - *MPI_ERR_OTHER* – which indicates a *recoverable* error.

7.3.1 Recovery issues among multiple processes

This stands contrary to the previous – non fault-tolerant – situation, where a job either a) proceeds successfully, or b) terminates completely on error. Program-flow in regular MPI programs, thus, ends up being relatively linear. The extra return state implies that things will not be as simply when using FT-MPI. Specifically, we will now have to add a mechanism for the different processes within the job, to coordinate among each other after an error in the VM has been discovered, and collaboratively return to a consistent state before resuming calculation.

The complete set of processes might actually be able to assume multiple potentially consistent states after recovery. Some will be more efficient (i.e. lose less data, but harder to create new consistent inter-process state) others will be easier to accomplish (i.e. less work during recovery, simpler coordination mechanisms, but less efficient) than others. The approach taken will influence important attributes of a program like performance, maintainability, as well as the ability to use certain capabilities of FT-MPI.

For example, recovery modes like FTMPI_COMM_MODE_SHRINK or like FTMPI_COMM_MODE_BLANK allow for a lot of flexibility, but add complexity as – during recovery – processes might change rank or become unreachable, thereby completely changing the overall global state of the VM. Under such a scenario, certain processes might have to "migrate" their functionality to another slot (i.e. a master process is needed by many parallel programs to function), while FTMPI_COMM_MODE_REBUILD does not require such complexity.

State-retaining as it may be, it might not always be desirable to re-spawn each unrecoverable process as, for example, this might result in a large number of re-spawned processes running on the same host due to lack of idle (redundant) machines in the VM, leading to detrimental CPU competition and unwanted synchronization between multiple processes.

All of these issues will have to be take into account when redesigning our software, and will influence its structure at a very basic level. Moreover, all of this implies, for any FT-MPI based software, that easy switching between different sets of choices for FT-MPI modes might not be easy, or even possible at all.

7.3.2 Recovery issues within a single process

Another issue to be taken into account, is that recovery might lead to a situation in which some processes have to recalculate data that was lost due to a crash further

upstream in the process-work-flow. Such a situation will force the process to "roll back" to a previous state, and restart calculation from that point. Sometimes, it might even be necessary to "roll forward" a process – in particular, this will happen to many re-spawned processes.

Which brings us to the topic of state-retaining and stateless processes : in many designs, some processes will be more important than others, in that they retain less state than others (or none at all), while others will contain a lot. In the case of the average master-slave application, we even have the extreme situation, wherein a single process holds all of the calculation state, while all of the others hold none.

Recovering / "roll forwarding" a state-poor process is generally easy and comes with relatively small losses in efficiency, while recovering a state-rich process will almost always require one or more processes to roll-back. Thus, distributing state over many processes will heighten the chance of state-loss happening (and require more rollbacks) - but, as less state is retained per process, it also makes for smaller roll-backs, improving efficiency during recovery. Again, these issues lead to choices that will have a profound influence on the structure of any fault-tolerant MPI software.

7.3.3 State-based, state-aware programming

One word has been seeing a lot of use in the previous paragraphs: *state*. More precisely, we discussed three kinds of state.

Different kinds of state

First, state among processes or *inter-process state*, requiring cooperation and coordination among processes during recovery – i.e., how do we bring all processes back into a mutually consistent state.

Second, state within a process or *intra-process state*, which needs to be managed w.r.t. roll-back/forward operations during recovery, as well as during general process flow.

And third, *result state* which changes as a result of process failures, and depends on the state-retaining aspects of the failed processes in question.

This situation will naturally lead to a prolific presence of concrete, visible state information in any FT-MPI based software. Practically speaking, it means that our design approach has been to re-factor the software to become state-based and state-aware: where state used to be part of the software only on an implicit level, the software will now be visibly and concretely divided into multiple states, one naturally flowing into the other during normal operation. Each process now keeps a concrete track of its state evolution.

Keeping track of state

Inter-process state coordination after recovery is managed by an application-specific RECOVERY state. All process synchronize at the start of this process (using the aforementioned MPI_Dup operation on MPI_COMM_WORLD as MODE_AUTO has not currently been added to the FT-MPI reference implementation). According to need, all processes evolve through a sequence of recovery sub-states until all processes agree on a baseline to resume calculation.

Intra-process state is kept as a concrete state-counter, which is checked at each state-change, either between "regular" states or regular/recovery states and vice-versa. Each process is given specific and concrete logic on how to flow from one state into the other, be they regular or recovery states.

In this manner, each process regulates its own roll-back/forward according to the outcome of inter-process coordination during the synchronized recovery state. Thus, roll-back/forward is basically reduced to changing a counter – the built-in state handling logic will handle the rest of the process.

Result state is managed through lists that keeps check of the tasks that have been calculated and exchanged between master and recombinators.

Each state tends to be delineated by an MPI communication operation, which can either result in success (MPI_SUCCESS – proceed as normal), recoverable error (MPI_ERR_OTHER – proceed to state RECOVERING in order to synchronize all processes in the VM) or unrecoverable error (proceed to state ABORTING to take necessary actions to abort the VM). In order to make program flow easier to follow, some purely logical states have been added as well.

In other words, we can look at the whole system as if it were a state machine, with the results of MPI calls as input, and the current state (and the result files that come with it) as output. The results list is then, in its turn, used as input for the recovery process when needed.

Adding state to the software

Adding FT-MPI support to our software takes the following changes related to state management: [intra-process state] 1) delineate states within the work-flow of each process type (master, calculator, recombinator), 2) add state counter to keep track of the current state, 3) add code, defining logic to control state transitions for each process type, [inter-process state] 4) add a recovery state to each process type that, at least, performs the required MPI_Comm_Dup action for synchronizing the recovering processes, 5) define a recovery work flow in terms of state transitions, dependent upon the chosen level of complexity/efficiency for the recovery process, and integrate the new states within the transition logic 6) add the necessary logic to the recovery process to deal with roll-forward/back situations.

In terms of result state, we are faced with a different situation for each distinct type of process.

The master process is surprisingly state-light. During most of its operation, the only state it keeps is the lists of tasks that still need to be computed. As as each of the recombinators also knows which tasks it still needs, this data is covered through redundancy and can be reconstructed on an eventual crash.

The calculator processes retain no state, except for the result of the latest calculation in case that it hasn't been relayed to a recombinator yet. In case of a calculator crash, the data originally on it will need to be recalculated, but the damage will be limited to only that result.

The recombinators are the major state retaining processes in the work-flow. They accumulate the results of all the calculators, and without any kind of custom "self-checkpointing" (i.e. periodically writing the current state of the result matrix to

persistent storage), all calculations for that recombinator will have to be redone after a crash. This will impact the state of the master during recovery - and of course, the duration of the calculation.

7.3.4 Practical choices regarding FT-MPI

When faced with the choices between the different FT-MPI modes, we were forced to make some decisions that would uniquely shape the rest of the re-factoring process.

Recovery mode

The most influential choice that we have to make is definitely that between the different FT-MPI modes.

Choosing the recovery mode is easy: MODE_IGNORE falls back to regular MPI behavior (not fault-tolerant), and is hence useless for our purpose. Mode AUTO would be our mode of choice, but is currently not supported by the reference implementation. The remaining option is RECOVERY_MODE_MANUAL, and only requires one extra operation before recovery proper can start - this is the recovery mode we will be working with.

Communicator mode

Choosing the right communicator mode requires greater consideration.

Discussing the merits of communicator mode BLANK, we came to the conclusion that it provides almost no advantages in our situation. We would have to add logic to the software to deal with non-active processes in case of a calculator. Worse: we would have to provide the necessary means to "migrate" process logic to a new communicator ID in case of failure of the master, or recombinators.

Mode SHRINK will have the same problem with regards to the need to migrate process logic to new communicator IDs. Communicator mode REBUILD, then, seems the logical choice.

It does have one major disadvantage though: sticking to the principle of MPI being hardware-agnostic, the FT-MPI spec does not provide any means for controlling on which processor a process gets to be re-spawned after recovery. It could attempt to restart the process on the same processor/host: this would be preferable, but is obviously not always possible (e.g. when the host has crashed and fails to come up again, or when the network link to the host has suffered a permanent failure). It could also pick a new processor out of a list (e.g. in a round-robin fashion). This is not much of a problem if we have redundant resources available; however, in any case, we are likely to maximally apply any resource we have at any time.

The result will be that, after a number of failures, the system will evolve into a state wherein multiple processes will be sharing the same computational resources. Naturally, this could be detrimental to operational efficiency.

Combining the master with any other process will not be much of a problem, as it does not consume many resources (except when wrapping up during the end phase, but at that moment, all other processes will be idle).

When combining a calculator with multiple recombinators, or multiple calculators, on a single computational resource however, performance will inevitably be negatively affected.

Introducing a shared filesystem

There are a few other things to consider when using COMM_MODE_REBUILD : foremost, it would mean that we would have to use a shared filesystem for the nodes. Specifically, we would have to make sure that result files are all eventually written to the same location, no matter how many times the master gets "migrated" after a failure.

This does not have to be detrimental. In fact using something like a (fast) network-shared raid type 1, 5 or 6 based solution would allow us to add fault-tolerance to the storage medium. It would also allow us to simplify the design of our software: each process would be able to read its configuration for itself, instead of having to rely on the master to do so and spread it through multiple MPI broadcasts. The whole work-flow would become simpler, more streamlined, more reliable and – given the right storage technique – probably even faster.

Furthermore, using a shared storage medium would allow us to have the result files written directly by the recombinators. This would eliminate another send/receive per recombinator, simplifying the software structure even further.

After some arguing back and forward, we came to the following conclusion. For the purpose of our work, we can make the following compromise : given the fact that we are using computational nodes with hyper-treading enabled, we can run our nodes with one process each instead of two (as we did during our previous test runs), and use the resulting (small) redundancy as a means to compensate for any situation in which multiple processes will run on a single node. In exchange for this, we can make our software simpler, easier to test, deploy and maintain as described above.

Given that, under regular circumstances, we would still expect failures to be exceptional situations, this compromise should produce more than acceptable results.

Message mode

As the collective modes have not been implemented yet, our next and last choice is between message mode CONT and RESET.

The advantage of using message mode CONT is, of course, that data in transit when a problem is detected will arrive as originally planned. In other words, if a calculator has produced a result and subsequently crashes, bringing the VM into the ERROR_DETECTED state, the calculated result will not be lost.

There is a complication however: the FT-MPI spec raises an important warning flag about using MODE_CONT in combination with non-deterministic communication patterns when using wildcards like MPI_ANY_SOURCE. Specifically, it illustrates how this combination might lead to deadlock (FT-MPI spec, p. 17-18). The spec is quite clear on the issue: "the usage of MPI_ANY_SOURCE should be avoided wherever possible when using mode FTMPI_MSG_MODE_CONT".

The ANY_SOURCE wildcard is used for proper streamlining of the master's work-flow though. It is not immediately apparent how to work around this in an elegant and proper manner. One way could be to do cyclical polling of asynchronous receive operations in the master process, but this method is – to say the least – far "clunkier" than what we have with the current implementation.

There are other complications as well though: it turns out to be rather hard to

find out whether there are any messages on the fly during the recovery phase. As we use standard send mode wherever possible, we can never be certain that a given send operation has completed - even for a single process, multiple sends might be on the wire at any given time.

This makes recovery using message mode CONT a lot more difficult, as every process will be forced to keep track of all its sends, thereby adding extra state to the processes which is not desirable in the least. (Looking ahead, we will see that using standard mode sends leads to other complications as well later on, during the design process). On top of this, mode CONT will affect the time efficiency of the FT-MPI VM recovery process as well – there is no straightforward answer to the question of whether it will actually be more efficient to use mode CONT, or to resend from the process flow itself during job recovery.

Hence, after some further discussing, we came to the conclusion that – certainly in a first-stage implementation – it would be far more convenient to use MSG_MODE_RESET.

Again, we start out from the idea that failures are exceptional, and will not be occurring on a regular basis. Furthermore – once more looking forward to the design process later on – we will see that some form of (one-sided) synchronization at the calculators will be required anyhow. This will allow us to retain data at the calculator until we are certain that the appropriate recombinator has received it. Hence, we will not lose any data calculated by a stable calculator during a failure, even though it will have to be resent.

The same goes up for messages sent from recombinators to the master while wrapping up. Any other messages will be small, and hence easy to resend – their loss will not be too much of an overhead.

Errors return vs. error-handlers

Starting out from the concept of a state machine, we decided to handle the return value of each MPI call as input to drive state transitions.

Hence, when faced with the choice between using ERRORS_RETURN or registering an error handler function, we chose the former. This runs contrary to the examples distributed with the FT-MPI sources, but we felt that it worked better and resulted in a more comprehensible program layout: recovery states are just "regular" states, that are reached through "regular" state transitions within a single state machine.

Practical crash testing

One more thing that we have to take into account is that we still have to rigorously test each of our designs: not just under regular circumstances, but including actual failures. There are two possible ways to go about this.

One is to manually kill a process through the FT-MPI console, as one can do in PVM. This method is very appropriate to simulate random crashes, but it does not enable us to do focused testing on different parts of our software.

The other method is one we refer to as "time-bombing". Basically, it involves performing a detectable error (e.g. writing to a null-pointer) at a particular chosen time within a specifically chosen process / state / spot. The major advantage of this

method is that we can do precise testing of each state(-change) / MPI operation and for each phase in each process in the software.

We came to the conclusion that this would be more useful in a first phase than simulating uncontrollable random crashes. The only disadvantage being that simultaneous crashes will be harder to accomplish – however, we should be able to accommodate for this given an appropriate setup. Hence we chose time bombing as our primary reliability testing method.

7.4 Approach 1: maximum simplicity

Given the preceding set choices, we thoroughly discussed the diverse possibilities for a fault tolerant design. We ended up with two implementations, allowing us to demonstrate both the potential of FT-MPI, and the design techniques needed to work with it. Specifically, the chosen examples illustrate the way in which complexity rises as soon as you start programming less than trivial software using the application-controlled approach to fault-tolerance.

Our first implementation stresses simplicity in the design, and attempts to introduce fault-tolerance into the software with a minimum of recovery-specific code. The major advantage here is that the design remains transparent, and relatively easy to comprehend, modify and maintain. The major disadvantage is that the recovery procedure has not been extensively optimized, which will lead to data loss during recovery. Any lost data will need to be recalculated.

7.4.1 Design

The design attempts to minimize – even virtually eliminate – recovery code by drawing as many parallels as possible between either regular task resolution and the mechanics needed for recovery, re-organising the software to take maximal advantage of reusable functionality. The only two explicit exceptions we will have to make in this regard are a) specific code for handling re-spawned processes, and b) a virtually empty recovery state that we have to introduce to provide the necessary code for activating recovery under FTMPI_RECOVERY_MODE_MANUAL.

In order to accomplish our design goal, we move the first step of the computation process – task marshaling, i.e. the creation of the tasks and the full task set – from the master to the recombinators. Since we are now using a shared filesystem, each recombinator can read all the necessary config files and create for itself the complete set of tasks for which it needs solutions.

Once each recombinator knows about the set of tasks for which it requires computation, it sends the set to the master process. From this point on, parallel computation proceeds as previously, until either the complete solution is successfully computed or a recoverable error occurs. On a recoverable error, each process transitions into a logical recovery state that kick-starts FT-MPI recovery and then falls back into the task marshaling sequence. There are two spots at which the task marshaling sequence requires some adaptation to allow for error-recovery.

At the master process, the task marshaling state has to take into account that a recombinator can send a NO_FURTHER_TASKS message instead of an actual task set. This messages communicates that all required tasks for that particular recombinator were successfully computed before the error occurred. The recombinator is furthermore ignored, as – under normal circumstances – it has already written its results back to persistent storage and is simply waiting for the finalization sequence to start.

At the recombinators, as a part of the task marshaling sequence, we have to check the results that have already been calculated, and remove them from the task set before sending it again to the master.

In this manner, by "smuggling" a minimal bit of error recovery logic into the regular computational flow for the master and the recombinators, we keep down the number of different states (and state transitions) for each type of process. In fact, the calculator processes will not have any kind of recovery code whatsoever.

7.4.2 Disadvantages of the design

Design-wise, one disadvantage of this approach is that we have actually introduced "covert" recovery code into the regular program flow, thereby obscuring a strict division between regular code and recovery code. We are, however, convinced that this is worth the resulting greater simplicity and ability for code re-use.

A bigger problem in this case, lies with result data that is either "in calculation" (i.e. result data on a calculator that hasn't been forwarded to a recombinator yet) or "in transit" (i.e. in course of being sent to a recombinator). In calculation data will be lost because the calculators are not part of the task marshaling sequence. Hence, they can not inform the master of the fact that they have result data – unknown to any of the recombinators – that does not require calculation. In order to make sure that we do not recombine the same result multiple times into the same result matrix, this result has to be dropped. In transit data is lost because of our previous decision to use FTMPI_MSG_MODE_RESET.

Neither of these problems can be dealt with under the current design. A recovery procedure that fully incorporates the calculators will be necessary to overcome them. This is what we will concentrate on in our next design. We will see, however, that this requires a separate recovery procedure that is far more complex than the one in this design - certainly if we have to take into account that the recovery procedure has to be fault-tolerant as well!

7.4.3 Remarks about states and transitions

The final step in describing our first design is a thorough illustration and state-by-state description of the program flow for each different type of process. Before we start out on the actual state diagrams and descriptions, some more remarks may be in order.

First, each type of process will now have two exit states.

Exit can happen into the regular result state, meaning that we have been able to successfully complete the whole computation and exit successfully, or into the abort state, meaning that we have encountered an unrecoverable error and have not been

able to produce any usable results. This is merely a more explicit specification of the situation as it was in the non fault-tolerant software as well.

Entry into the state-flow happens through an "init" state (at startup), but – from then on – either proceeds to the next regular state or directly to the recovery state. The latter is reached in case that it concerns a re-spawned process. In this case, the code needs to execute the necessary actions to trigger recovery as per RECOVERY_MODE_MANUAL.

Each process also has a new "pre-sync" and a "post-sync" state, as described in sections 7.2.1 and 7.2.2.

Each major state change takes place along the boundaries created by an MPI call. Each of these state changes can result in either a result state of success (the next state in the logical program flow), "recovering" or "abort" – "abort" being an end state, and "recovering" potentially leading back to one (or more) regular states.

A few, purely "logical" states exist as well, that do not influence the MPI-driven state changes (and hence the recovery procedure), but that were entered into the state diagrams to add further clarity to some situations.

7.4.4 Information about the state diagrams

The only task that remains now is to give some practical information w.r.t. the state diagrams following in the next section.

Each of the diagrams consists of a number of numbered states, connected by unidirectional arrows. Each number has an associated name in the accompanying text. This accompanying text also mentions the MPI call at exit of the state. Thick-lined circles signify "regular" states, thin-lined circles "purely logical" states, and double circles indicate an "entry" states or one of the"exit" states. Squares represent "composite" states, i.e. states that are themselves composed of multiple states and transitions. For the sake of clarity, these are elaborated upon in further ("sub-") state diagrams, with their own block of accompanying text.

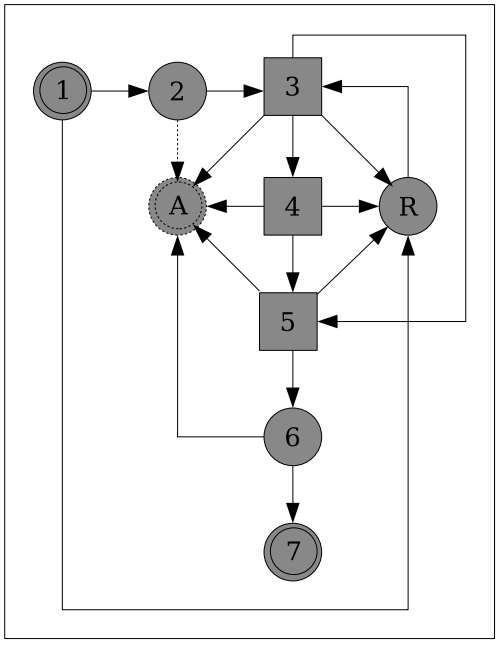
Each main diagram has four special states, directly indicated as such : 1) state "**1**", the initial state at job startup - note that this is also a purely logical state, as no MPI call is performed at this stage that supports error detection or recovery through FT-MPI, 2) state "**R**" or "recovering", the state entered after encountering an MPI_ERR_OTHER and finally, 3) state "**A**" or "abort" – the state we reach when the VM encounters a non-recoverable error.

State transitions are represented by arrows. Within the context of FT-MPI, three kinds of transitions can occur: 1) a transition to the next regular state in case of no error, 2) a transition to state "R" in case of a recoverable error or 3) a transition to state "A" in case of a non-recoverable error. Since there are no other transitions possible, we do not need to label the arrows in the diagrams - the transition's condition will be clear from its destination state.

Sub-diagrams show dotted lines from a dotted circle representing its referring state, and another dotted line towards its destination state on exit.

7.4.5 State diagrams and descriptions

We start out with the state diagrams for the master process. In total, we will need a total of one main and 3 sub state diagrams to describe all the logic contained within the master.



Main state diagram for the master process. Notice that the recovery state is not a composite one: recovery basically comes down to performing all the necessary actions to trigger recovery of the VM under mode MANUAL. Afterwards, we immediately transition back into the task marshaling state.

Figure 7.1: Master – main state diagram

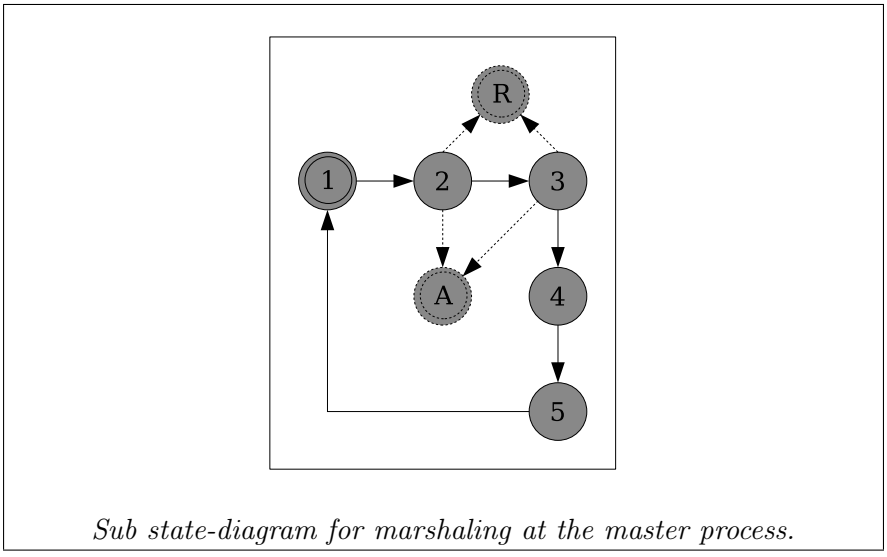
Master – main state diagram 7.1:

The master process is most easily described as a flow over multiple broad composite states, which will be described further in a number of sub state diagrams. Notice that it contains no F90 code anymore whatsoever. All code is C++ now, except the MPI calls which use the C API due to the FT-MPI shortcomings described above.

1. Initializing: entered at job startup. The process takes note of the number of recombinators and calculators. No further data is needed at this stage. (Specifically, note that – under our new design – we have no need to read any config files at the master end.) *No delineating MPI call that triggers state-changing output (logical state).*
2. Pre-synchronizing: will either lead to success or abortion. An error at this stage

will move us into a state wherein it cannot be guaranteed that all processes have synchronized for the purpose of proper recovery, as discussed in sections 7.2.1 and 7.2.2. Hence, no recovery is possible at this stage. *Full N-to-N synchronization is accomplished through a collective call to MPI_ All_ Reduce.*

3. Marshalling: composite state. Task sets are received from all recombinators (or NO_FURTHER_TASKS messages in the case that one or more recombinators have all the results that they need). These tasks are marshaled into a single task bag, to be resolved in the coming states. We elaborate on this composite state in figure 7.2, and the accompanying description block.
4. Resolving task-bag: composite state. The master waits for task requests from the calculators. Tasks are sent until the bag of tasks is empty. We elaborate on this composite state in figure 7.3, and the accompanying description block.
5. Finalizing calculators: composite state. When the task bag is empty, the master sends finalization messages to all of the calculators. We elaborate on this composite state in figure 7.4, and the accompanying description block.
6. Post-synchronizing: put master on hold until all other processes have finish,ed their business, before finalizing to prevent the problems described in sections 7.2.1 and 7.2.2. Like in the pre-synchronization stage, *full N-to-N synchronization is accomplished through a collective call to MPI_ All_ Reduce.*
7. Finalizing: the last "active state". Used for things like garbage collection, writing to log files etc. *No delineating MPI call that triggers state-changing output (logical state).*



Sub state-diagram for marshaling at the master process.

Figure 7.2: Master – marshaling

Master – marshaling sub state diagram 7.2:

1. Marshalling: as long as the master hasn't received either a task set or a NO_FURTHER_TASKS message from all recombinators, wait for incoming

messages from the recombinators. Change state to "waiting for tasks" to wait for unresolved recombinators, or "resolving task-bag" when all recombinators have checked in. A counter is maintained to track the number of recombinators that has checked in. The counter is initialized to 0 when we enter this state for the first time since either startup or the latest recovery procedure. *No delineating MPI call that triggers state-changing output (logical state).*

2. Waiting for nr. of tasks: the first message that the master expects from a recombinator is the number of tasks in the task bag that it is sending. If this number is 0, the master takes this as a NO_FURTHER_TASKS message and flows back into the "marshaling" state. Otherwise, the master proceeds to "waiting for tasks". *This is accomplished by posting an MPI_Receive for an MPI_Integer message from ANY_SOURCE in COMM_WORLD with the proper label. We take note of the sender of the message in order that we can properly complete the next state.*
3. Waiting for task set: knowing the nr. of tasks to expect in the set for this particular recombinator, the master waits for the task set. *This is accomplished by posting a MPI_Receive for an array of the proper amount of MPI_Integers from the proper recombinator. A proper label is used, though it is not strictly necessary due to the guarantees provided by MPI (specifically, that a series of messages from the same source to the same destination will be received in the same order that they were sent).*
4. Adjusting task bag: add the received task set to the bag of tasks. *No delineating MPI call that triggers state-changing output (logical state).*
5. Adjust counter: advance the recombinator counter by 1 before falling back into the marshaling state. *No delineating MPI call that triggers state-changing output (logical state).*

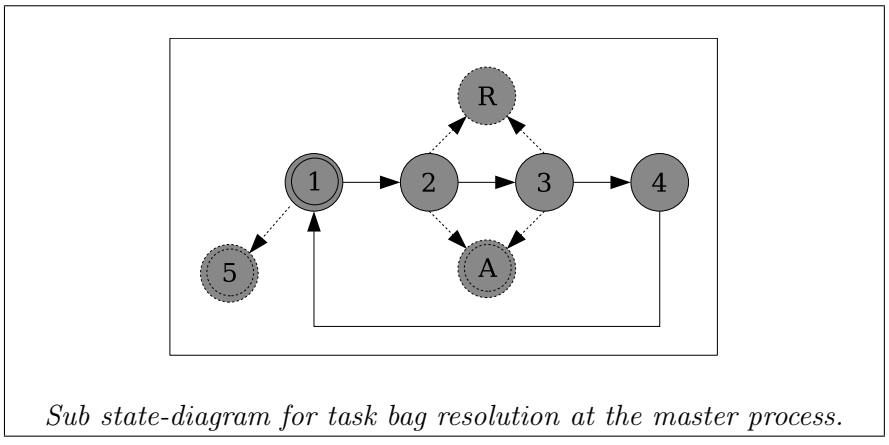


Figure 7.3: Master – resolving task-bag

Master – resolving task-bag sub state diagram 7.3:

1. Resolving: the actual calculation loop at work. As long as we still have tasks

in the bag, flows through to the waiting state. Otherwise, change state to finalizing calcs. *No delineating MPI call that triggers state-changing output (logical state).*

2. Waiting for calculator: wait for a calculator to message that it is ready to accept a new task. *This is accomplished by posting an MPI_Receive for an MPI_Integer message from ANY_SOURCE in COMM_WORLD with the proper label. We take note of the sender of the message in order that we can properly complete the next state. The value sent through the message is a fixed number – whenever the actual value sent should differ from this, fall through to the abort state. On a successful receive of the notification, flow through to "sending task".*
3. Sending task: retrieve the next task from the task bag (stored as a list structure) and send it to the available calculator. *This is accomplished through an MPI_Send operation of an MPI_Integer on COMM_WORLD to the calculator that sent us the confirmation in the previous step. The calculator, through the config files on the shared storage medium, has access to all the data that maps to the particular task number.*
4. Decreasing task bag: remove the sent task from the task bag and flow back into the "Resolving" state. *No delineating MPI call that triggers state-changing output (logical state).*

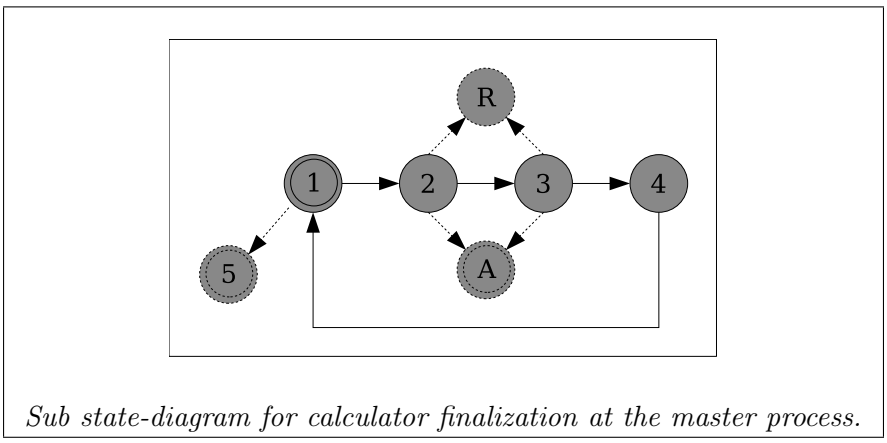


Figure 7.4: Master – finalizing calculators

Master – finalizing calculators sub state diagram 7.4:

1. Finalizing: as long as we still have active calculators, flow through into a waiting state. If all calculators have checked in, advance straight into post-synchronization. Calculators are tracked by a counter, which is initialized to 0 when entering this sub-state for the first time, or after a successful recovery procedure. *No delineating MPI call that triggers state-changing output (logical state).*
2. Waiting for calculator: as in the "resolving" state, wait for a calculator to send

notification of it's availability. *Once more, this is resolved through posting an MPI_Receive for a message containing an MPI_Integer on COMM_WORLD, using MPI_ANY_SOURCE and taking note of the sending calculator. Again, if the notification contains an unexpected value, we immediately go into the abort state. under normal circumstances, we advance to "sending finalization".*

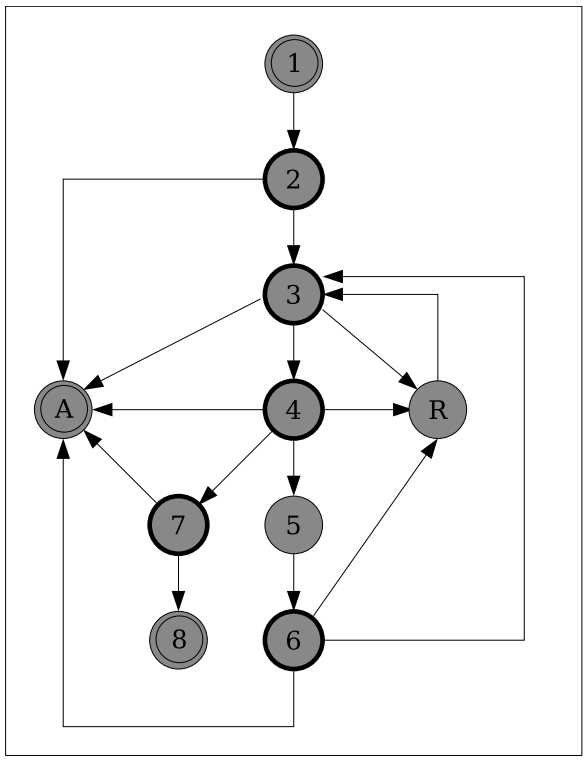
3. Sending finalization: instead of a task, we send a finalization message in the form of a negative integer. On reception, the calculator should interpret this as a message to stop its regular calculation cycle and proceed to post-synchronization itself. *Resolved through a call of MPI_Send with an MPI_Integer message on MPI_COMM_WORLD to the proper calculator.*
4. Increase count: increase calculator count by one and flow back into the "Finalizing" state. *No delineating MPI call that triggers state-changing output (logical state).*

Thereby, we have finished describing the master process. As we can see, putting a very small amount of recovery code within the marshaling state leaves us with a rather elegant state-flow. We shall see that this convenience is maintained within the other process types as well. In fact, looking at our next subject, the calculator processes, we will notice that they do not contain any recovery logic at all.

Calculator – main state diagram 7.5:

The calculator processes are the most straightforward, design-wise. In fact, we only require a main state diagram to describe it, as it is simple enough to be represented without composite states.

1. Initializing: read config files from shared storage and store the necessary data to map task numbers (which will be received from the master) to a set of input data for the calculation functions. Afterwards, flow through to pre-synchronization. *No delineating MPI call that triggers state-changing output (logical state).*
2. Pre-synchronizing: as in the master process. *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce. On success, flow through to the "notifying master" state.*
3. Notifying master: notify the master process that the calculator is ready to accept a task. Recovery flows back into this state, yet it doesn't contain any recovery-specific code. For calculators, recovery simply consists of dumping all their current data and making themselves available again. *The state is concluded by an MPI_Send operation of an MPI_Integer (1) to the master process. On success, make the transition to the waiting state.*
4. Waiting for task: the calculator waits for a response to the previous message. This response can be an actual task nr., or a NO_FURTHER_TASKS message. *Transition on a post to MPI_Recv. On successful receipt, change to "calculating" if the message contains a task nr., or to "post-synchronizing" if it is NO_FURTHER_TASKS.*
5. Calculating: retrieve the necessary input for the calculation from the map structure using the task nr., and call the F90 calculation function. On return from the F90 function, flow through to the next state. *No delineating MPI call*



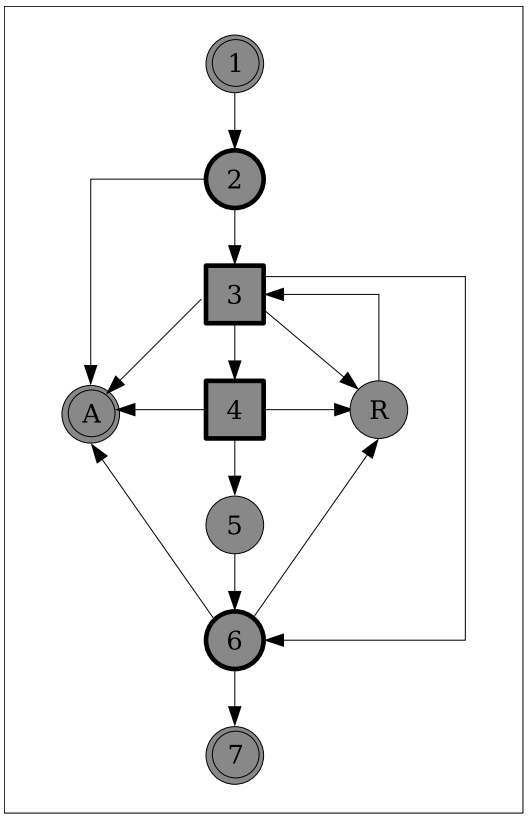
Main state diagram for the calculator processes. Notice the lack of composite states in this diagram: the calculator is the simplest of the process types.

Figure 7.5: Calculator – main state diagram

that triggers state-changing output (logical state).

6. Sending to recombinator: send the partial result matrix, generated by the calculation, to the appropriate recombinator (once more, information that we can retrieve from our memory map). *Change state on MPI_Send of a pointer to the partial result matrix to the appropriate recombinator. Flow back into the "notifying..." state.*
7. Post-synchronization: as in master process. *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce.*
8. Finalizing: except for possible logging and profiling data that needs to be gathered/written, no specific finalization steps need to be taken.

The simplicity of the diagram is large due to the lack of specific recovery code in the calculator. The trade-off is a lack of persistence w.r.t. the data kept in the calculator, which is lost on any MPI error. Most of the action, as we will see, will be happening in the last type of process we describe: the recombinator.



Main state diagram for the recombinator processes. Notice the high number of different transitions going out of the "marshaling" composite state. It is probably the most complex of the states in the whole master-calculator-recombinator system, due to it's built-in recovery code. Still, the diagram as a whole remains largely straightforward.

Figure 7.6: Recombinator – main state diagram

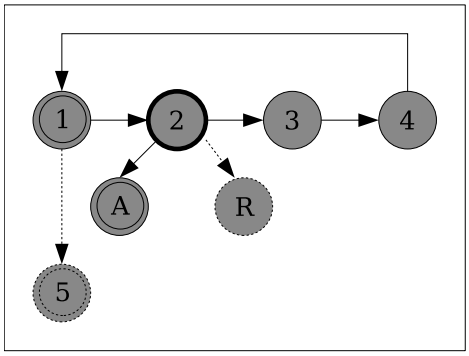
Recombinator – main state diagram 7.6:

The recombinator processes are some of the most crucial in this setup, simply because they are the only truly state-retaining processes within the whole calculation mechanism. Losing one of these means irrevocably losing a potentially major amount of calculated data. In such a case, this data will have to be recalculated.

1. Initializing: read the necessary data from shared storage and create the proper data structures to contain the accumulating partial result matrices. *No delineating MPI call that triggers state-changing output (logical state).*
2. Pre-synchronization: as in the master process. *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce. On success, flow*

through to the "notifying master" state.

3. Marshalling tasks: composite state. At startup, the recombinator determines which tasks it needs calculated, marshals them into a task set and sends it to the master process. This state also contains error recovery logic for determining the progress of the calculations after an error. We elaborate on this composite state in figure 7.7, and the accompanying description block.
4. Processing results: composite state. Task results are received from all calculators. These results are recombined into a result matrix, eventually to be written to file in one of the coming states. We elaborate on this composite state in figure 7.8, and the accompanying description block.
5. Writing UNF: write the completed result matrix to disk as an F90 unformatted file. After this point, the recombinator ceases to be state-retaining and will, hence, no longer suffer from a risk of data loss. *No delineating MPI call that triggers state-changing output (logical state).*
6. Post-synchronization: as in master process. *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce.*
7. Finalize: as in the other processes.



Sub-state diagram from task marshaling at the recombinator. Notice that after sending the NO_FURTHER_TASKS message, we flow straight through to post-synchronization. It is not possible for the recombinator to be in a global state with no further tasks left to be calculated, without having written its UNF data file first.

Figure 7.7: Recombinator - marshaling tasks

Recombinator – marshaling tasks sub state diagram 7.7:

1. Marshalling tasks: the recombinator checks the number of tasks left to be calculated (initialized in the main state and potentially influenced by previous calculations before a recovery) – the set of uncalculated tasks that are left needs to be transferred. *No delineating MPI call that triggers state-changing output (logical state).*
2. Sending nr. of tasks: send the nr. of tasks in the task set to the master. If

there are no tasks left, send a NO_FURTHER_TASKS message. *Delineated by an MPI_Send of an MPI_Integer to the master. Check the nr. of tasks. If it is greater than zero, proceed to "sending tasks". If not, proceed directly to "post synchronization" in the main state diagram virtual state "5"). (Incidently, the latter case can only occur if we have arrived back at this state – through recovery – from the post-synchronizing state, after writing the UNF file. This branch in the task marshaling process flow serves as integrated recovery code to simplify both the recovery process and the overall state diagram.)*

3. Sending tasks: send the task set as an array of MPI_Integers. *Ends in an MPI_Send of #task_set MPI_Integers to the master. Proceed to the "processing" composite state 7.8 (virtual state 4).*

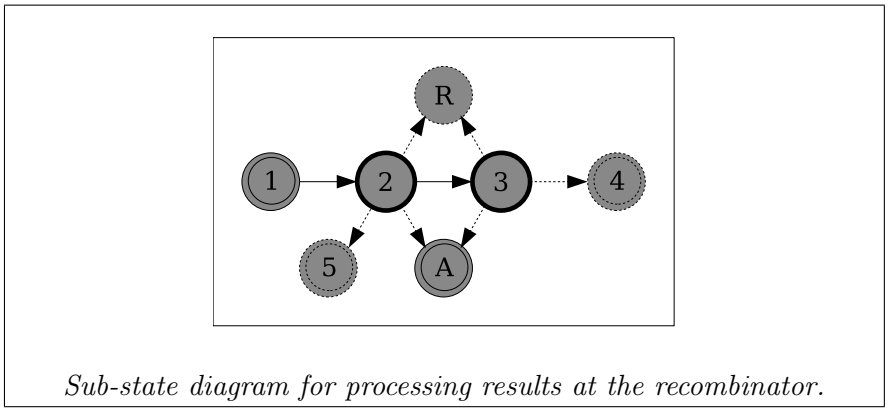


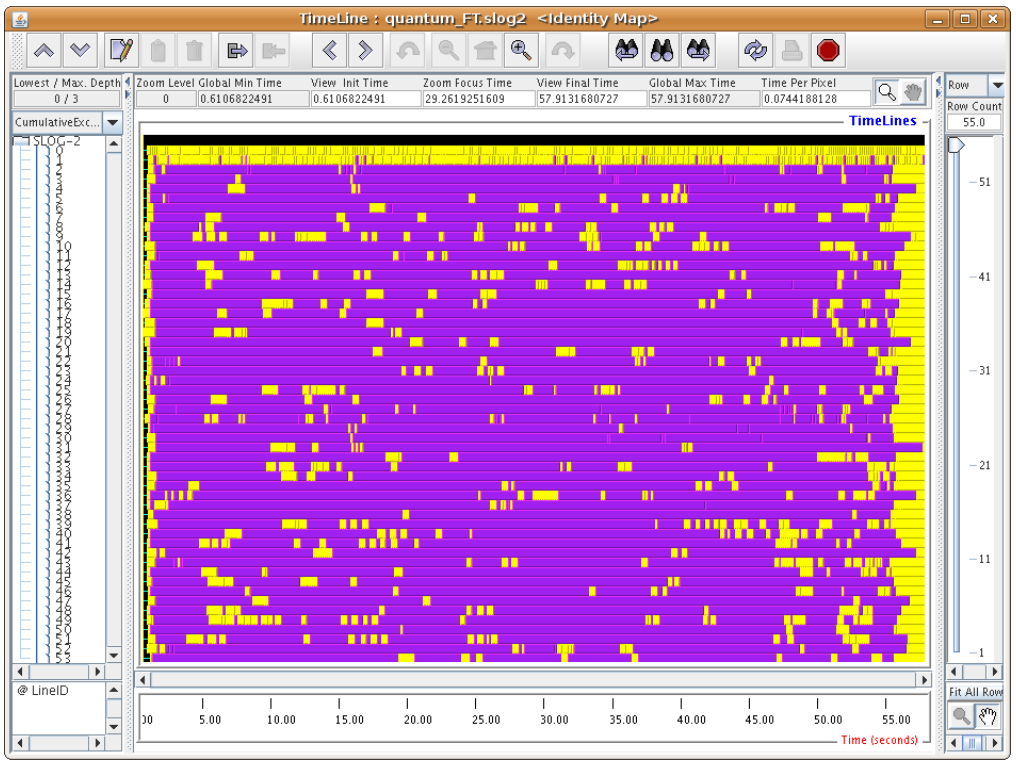
Figure 7.8: Recombinator - processing results

Recombinator – processing results sub state diagram 7.8:

1. Processing results: keep note of the set of results left to be received. As long the set is not empty, proceed to wait for a result. As soon as it is empty, transition to writing the UNF file. *No delineating MPI call that triggers state-changing output (logical state).*
2. Waiting for result: wait for a result from a calculator. *The recombinator posts an MPI_Receive for a partial result matrix from ANY_SOURCE, with ANY_TAG. On receipt, take note of the tag value: it is used to piggy-back the task nr. onto the result message.*
3. Recombining: recombine the partial result matrix into the total result. *No delineating MPI call that triggers state-changing output (logical state).*
4. Decrease task-bag: remove the task nr. received from the set of tasks to be calculated, and flow back into processing results. *No delineating MPI call that triggers state-changing output (logical state).*

Performance

After running the corresponding code through a set of performance tests similar to those we used to evaluate the non fault-tolerant versions of the code (figures 7.10 and 7.9), we came to the conclusion that the measures we took to introduce fault-



Another calculation for $K = 6$,$L = 2$, this time with the first fault-tolerant iteration of the code. Notice that fault-tolerance has not negatively influenced workload processing at higher values of L.

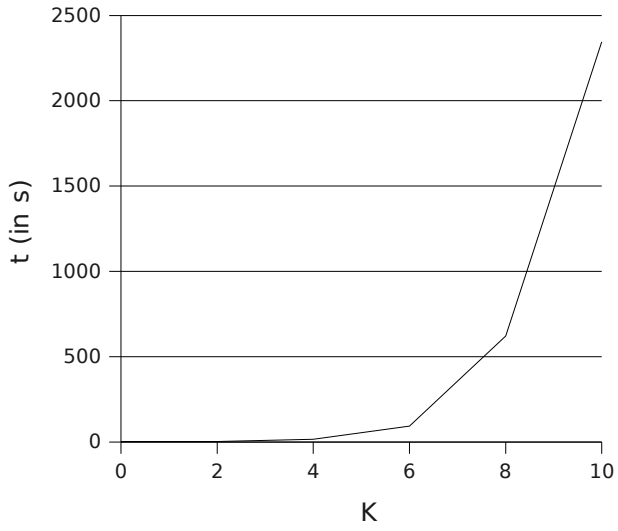
Figure 7.9: Profiling the first fault-tolerant version of the program for $L = 2$

tolerance did not introduce any significant overhead beyond what we could measure for the non fault tolerant versions. Thus, the FT-MPI reference implementation proves to be robust performance-wise, and the addition of fault-tolerance turns out not to impose any significant extra effort under normal (fault-free) operation.

7.4.6 Crash tests

In order to empirically test the fault-tolerance of the whole system, we put the whole program through a series of crash-tests using the above-mentioned 7.3.4.6 time-bombing approach. Each state in each process type within the software was "rigged" with a time-bomb, and each of the time-bombs was set off in multiple series subsequent tests to test robustness. Over a large number of tests, the software proved able to survive all bombing attempts. Hence, we feel confident in the ability of our design to survive process crashes.

The influence of lost data in the recombinators, however, has shown to be potentially huge. During one test, we had the master send a whole "round" of tasks,



From this plot, it is clear that adding fault-tolerance to the software has not influenced its performance.

Figure 7.10: Performance for the first fault-tolerant version of the program.

and then immediately crash. This lead to all the calculators first computing a result, only to be notified of the failure by FT-MPI when trying to pass the result on to the recombinators. Following, the results were dropped and all of the work had to be performed a second time. Depending upon the amount of time needed by individual tasks, this might lead to major time loss.

7.4.7 Summary of approach 1

We have shown that, given some redesign, it is possible to make our software fault-tolerant towards process crashes during all phases of the computation process. More importantly, we have also shown that, for meaningful computational loads, this does not lead to measurable performance loss under a fault-less scenario. Moreover, even for a scenario that goes beyond the basic complexity of the elementary master-slave design, we have shown that it is possible to introduce application-controlled fault tolerance.

However, through our attempt to keep complexity as low as possible, we have introduced some major inefficiency in the recovery process, as calculator processes are currently not being involved in the recovery process. Under the current design, any results awaiting delivery on the calculators are lost during the recovery process.

Moreover, FT-MPI cannot interrupt a calculator: it can only notify the calculators when they contact the MPI layer, i.e. when waiting for a task or sending a result. As FT-MPI can only recover the VM on synchronization between all processes, re-

covery will have to wait until all calculators have finished their current computation, only to discard the results immediately on entering recovery. This is, naturally, an unacceptable waste of resources.

In the next design, we will attempt to fix this problem by actively involving the calculators in the recovery process. We will attempt to do this while still attempting to retain a minimum of simplicity in the design. The design – especially the recovery stage – will necessarily be more elaborate though.

7.5 Approach 2: more efficient, more complex

In order to minimize the amount of work that is "lost" due to the recovery procedures, we will change and expand the previous design to accommodate for a more intelligent recovery procedure, that is able to retain all computed results after a failure. This change in approach will be most visible within the recovery procedures of each type of process: we will see that the recovery state now becomes a composite state for each process type. Moreover, one process type – the calculator – will even have two different recovery procedures, depending on the state of the process at the moment of failure.

We will first give a broad overview of the design with its advantages, disadvantages and general remarks. Afterwards, as above, we will provide full illustrations and accompanying descriptions of the states and state flow for each type of process.

7.5.1 Design

On of the first things we changed about the previous design was to turn the task marshaling system back to the original approach, with the master creating the full to-do list of tasks to be computed at startup. Both the calculators and the recombinators determine at startup which results need to be provided to which recombinator. Recovery-specific logic is completely moved out of the regular states. Regular computation then proceeds pretty much as usual.

After failure, a two-step recovery process is used to re-synchronize all processes: first, all calculators notify the master whether they have result data available that possibly hasn't yet made it to a recombinator. The master scraps all available results from its to-do list. A similar procedure is then followed w.r.t. the recombinators. Each recombinator checks in with the results already in its possession, each of which is scrapped from the to-do list.

From that moment on, the system comes out of recovery. The master starts waiting again for "ready" messages from calculators. Calculators that possessed a result try to send it to the appropriate recombinator again. Calculators without a waiting result send a ready message to the master, requesting a new task. Recombinators start listening for results again. If it receives a result that has already been calculated, it is dropped. Otherwise, it proceeds as normal.

It is possible that a result gets sent twice. Whenever it has finished a calculation, the calculator sends the result to the proper recombinator and waits for the latter

to send a confirmation of receipt. Whenever a failure leads to the loss of such an on-the-wire confirmation messages, the calculator cannot know whether it should send the result again or not without conferring with the recombinator.

This would require first conferring between the recombinators and the calculators or the master. Apart from making the design quite a bit more complex, it would also drive up the number of round-trips, prolonging the recovery procedure. The current approach allows for the quickest possible return to the regular computing state, at the cost of a single potential spurious send per calculator with a waiting result – a risk we consider worth taking.

Moreover, this design could easily be adapted to eliminate the spurious sends, by using a synchronized send for the result. This would offer an atomical approach for the calculator to confirm receipt of the result. As synchronized communication modes haven't been implemented however, this is not possible using the current FT-MPI reference implementation. Thus, the current design is the closest possible to the ideal scenario.

7.5.2 Consequences of a composite recovery state

There is one more thing that needs to be taken into account: the possibility of an MPI failure during the composite recovery procedure. Again, we start out from the idea that – under normal circumstance – failures should be an exceptional occurrence. Given this, the clearest and easiest solution to the problem is to restart the whole recovery procedure. Basically, we flow into a RECOVER_ERROR state which flow straight back into the RECOVERING composite state, restarting the whole recovery process.

In the case that a recovery procedure actually gets to be interrupted by another failure, this will result in a number of spurious sends, as calculators and recombinators will send data to the master that it already received during the interrupted recovery procedure (except, of course, if it was the master itself that failed). However, only limited sets of MPI_Integers (task nrs.) are transferred during this phase (no calculation results), so the loss of efficiency will be limited.

Moreover, it keeps the composite recovery state relatively simple. Otherwise, we would have to write a new customized "meta"-recovery procedure for the recovery code itself. And we would probably have to write further recovery code for that one as well, etc... until we manage to end up with a single-state recovery flow as in our first example. Again, given that failures are supposed to be sparse, the latter proposal would not produce an efficient design. Indeed, given regular circumstances, a fast and large-scale occurrence of multiple successive failures would probably point towards some kind of pathological failure, that will eventually bring down the whole system anyhow.

In any case, by repeating the recovery procedure from scratch, we can still recover from any failure, whether it happens within or outside of the recovery sequence.

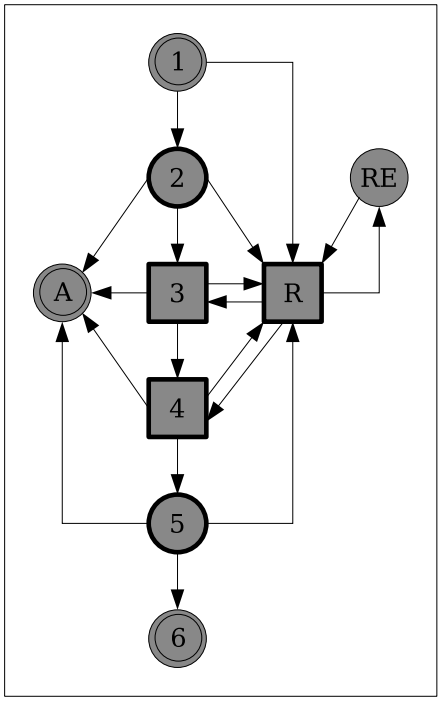
7.5.3 Disadvantages of the design

The most apparent disadvantage of this second design is the greater complexity within the recovery procedure, as recovery becomes a two-stage, composite state

operation. Apart from the extra complexity in itself, this also leads to the issues described above. As well, recovery will generally be taking more time, as each calculator is now being involved in the procedure. While the number of recombinators can be relatively low, the number of calculators can theoretically be as high as the number of tasks.

Given the gain in terms of recovered results – which will be most felt when calculations get longer with higher values for K, the loss in recovery speed and the higher overall complexity should be very much worth the effort however.

7.5.4 State diagrams and descriptions



Main state diagram for the master process. Notice, w.r.t. our first design, that the marshaling state has disappeared, and that the RECOVERING state has become composite. Otherwise, the high-level design of both implementations is quite comparable.

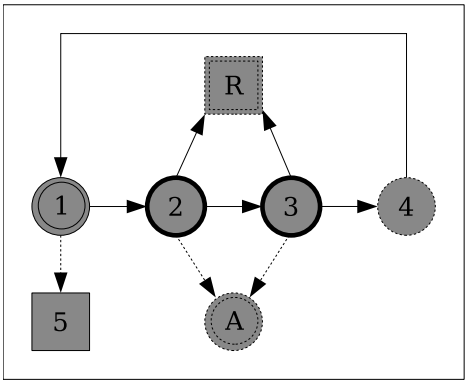
Figure 7.11: Master – main state diagram

Master – main state diagram 7.11:

1. Initializing: we arrive in this state either at startup, or after a restart. In both cases, all of the config files are read from storage, and the data is used to populate a task-bag with the full set of tasks. In case of a fresh start, proceed to PRE-SYNCHRONIZING. In case of a restart, go on to RECOVERING. *No*

delineating MPI call that triggers state-changing output (logical state).

2. Pre-synchronizing: *Full N-to-N synchronization is accomplished through a collective call to MPI_ All_ Reduce. On success, flow through to the "notifying master" state. On success, proceed to RESOLVING_ TASKBAG. On failure, ABORT.*
3. Resolving task-bag: composite state. Send tasks to available calculators until the task-bag has been emptied. We elaborate on this composite state in figure 7.12, and the accompanying description block.
4. Finalizing calcs: composite state. When a calculator requests a new task, notify it that it can quit as all tasks have been sent. We elaborate on this composite state in figure 7.13, and the accompanying description block.
5. Post-synchronizing: *Full N-to-N synchronization is accomplished through a collective call to MPI_ All_ Reduce.*
6. Finalizing: write out the general process data file (contains start time, end time etc.) and quit.
7. Recovering: handle errors by gathering task data from all calculators and recombinators to properly adjust the task-bag. We elaborate on this composite state in figure 7.14, and the accompanying description block.
8. Recovering error: an error occurred during recovery. Transition back to the beginning of the full recovery recovery sequence.*No delineating MPI call that triggers state-changing output (logical state).*



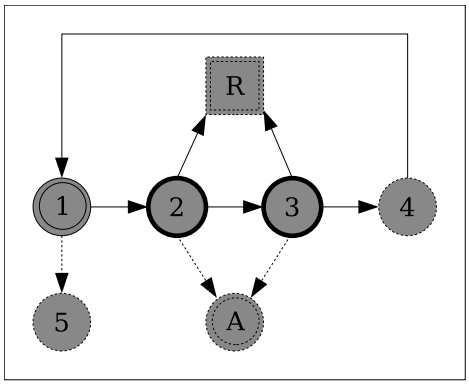
Sub-state diagram for the resolving state. Virtually nothing has changed from the first design.

Figure 7.12: Master – resolving task-bag

Master – resolving task-bag sub state diagram 7.12:

1. Resolving: the actual calculation loop at work. As long as we still have tasks in the bag, flows through to the waiting state. Otherwise, change state to finalizing calcs (virtual state "5"). *No delineating MPI call that triggers state-changing output (logical state).*

2. Waiting for calculator: wait for a calculator to message that it is ready to accept a new task. *This is accomplished by posting an MPI_ Receive for an MPI_ Integer message from ANY_ SOURCE in COMM_ WORLD with the proper label. We take note of the sender of the message in order that we can properly complete the next state. The value sent through the message is a fixed number – whenever the actual value sent should differ from this, fall through to the abort state. On a successful receive of the notification, flow through to "sending task".*
3. Sending task: retrieve the next task from the task bag (stored as a list structure) and send it to the available calculator. *This is accomplished through an MPI_ Send operation of an MPI_ Integer on COMM_ WORLD to the calculator that sent us the confirmation in the previous step. The calculator, through the config files on the shared storage medium, has access to all the data that maps to the particular task number.*
4. Decreasing task bag: remove the sent task from the task bag and flow back into the "Resolving" state. *No delineating MPI call that triggers state-changing output (logical state).*



Sub-state diagram for calculator finalization. Again, virtually nothing has changed from the first design.

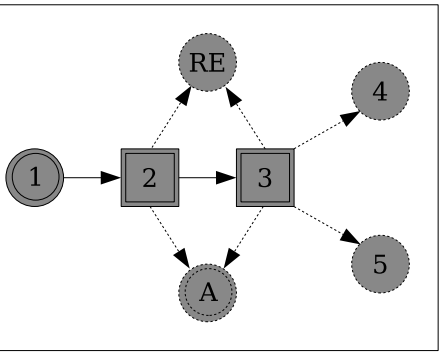
Figure 7.13: Master – finalizing calculators

Master – finalizing calculators sub state diagram 7.13:

1. Finalizing: as long as we still have active calculators, flow through into a waiting state. If all calculators have checked in, advance straight into post-synchronization (virtual state 4). Calculators are tracked by a counter, which is initialized to 0 when entering this sub-state for the first time, or after a successful recovery procedure. *No delineating MPI call that triggers state-changing output (logical state).*
2. Waiting for calculator: as in the "resolving" state, wait for a calculator to send notification of it's availability. *Once more, this is resolved through posting an*

MPI_ Receive for a message containing an MPI_ Integer on COMM_ WORLD, using MPI_ ANY_ SOURCE and taking note of the sending calculator. Again, if the notification contains an unexpected value, we immediately go into the abort state. under normal circumstances, we advance to "sending finalization".

3. Sending finalization: instead of a task, we send a finalization message in the form of a negative integer. On reception, the calculator should interpret this as a message to stop its regular calculation cycle and proceed to post-synchronization itself. *Resolved through a call of MPI_ Send with an MPI_ Integer message on MPI_ COMM_ WORLD to the proper calculator.*
4. Increase count: increase calculator count by one and flow back into the "Finalizing" state. *No delineating MPI call that triggers state-changing output (logical state).*



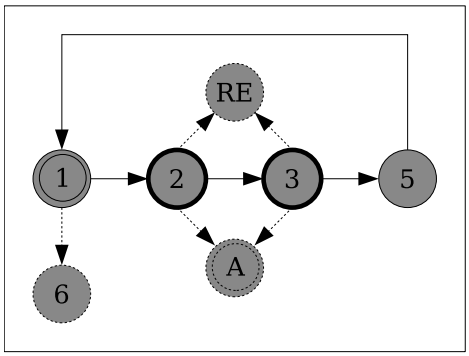
Sub-state diagram for the recovery state. It is clear to see the two-stage inquiry process, each one having its own sub-state diagram.

Figure 7.14: Master – recovering

Master – recovering sub state diagram 7.14:

1. Refilling task-bag: re-create the complete task-bag (from a copy taken during the initialization state). *No delineating MPI call that triggers state-changing output (logical state).*
2. Inquiring calculators: wait for calculators to send in whether they have task results available (max. one per calculator). For each calculator, remove its results (if available) from the task-bag. We elaborate on this composite state in figure 7.15, and the accompanying description block.
3. Inquiring recombinators: wait for recombinators to send in the results that they have available (potentially multiple per recombinator). For each recombinator, remove all recombined tasks from the task-bag. Transitions into Resolving Task-bag in the main state diagram when there are further tasks to be computed (virtual state "4"), or into Finalizing Calculators when it turns out that we are done (virtual state "5"). We elaborate on this composite state in figure

7.16, and the accompanying description block.



Sub-state diagram for the calculator inquiry stage during recovery.

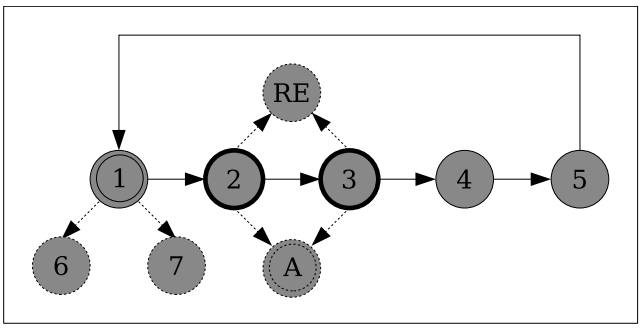
Figure 7.15: Master - recovering / inquiring calculators

Master – recovering / inquiring calculators sub state diagram 7.15:

1. Recovering / inquiring calculators: as long as we haven't received confirmation from all calculators, wait for one. If all calculators have called in, transition into "Recovering / inquiring recombs" (virtual state "6"). The nr. of calculators that has sent some form of confirmation is tracked by a counter. *No delineating MPI call that triggers state-changing output (logical state).*
2. Recovering / waiting for calculator: wait for confirmation from a calculator. *Resolved by posting an MPI_ Receive for an MPI_ Integer, which will either be -1, or a positive integer, corresponding to a task nr. In case of NO_ TASK (-1), transition into "Recovering / advancing calccount". Otherwise, proceed to "Recovering / adjusting taskbag (calculator)".*
3. Recovering / adjusting taskbag (calculator): remove the task nr., received at conclusion of the previous step, from the task bag. *No delineating MPI call that triggers state-changing output (logical state). No delineating MPI call that triggers state-changing output (logical state).*
4. Recovering / advancing calccount: increment the count of calculators, and return to "Recovering / inquiring calculators". *No delineating MPI call that triggers state-changing output (logical state).*

Master – recovering / inquiring recombinators sub state diagram 7.16:

1. Recovering / inquiring recombinators: as long as we haven't received confirmation from all recombinators, wait for one. Keep track of recombinators using a counter. As soon as all recombinators have sent in some kind of notification, advance to "Resolving taskbag" (virtual state nr. "6" in the state diagram) if there are any tasks left to be computed – otherwise, proceed to "Finalizing calcs" (virtual state "7"). *No delineating MPI call that triggers state-changing output (logical state).*



Sub-state diagram for the recombinator inquiry stage during recovery.

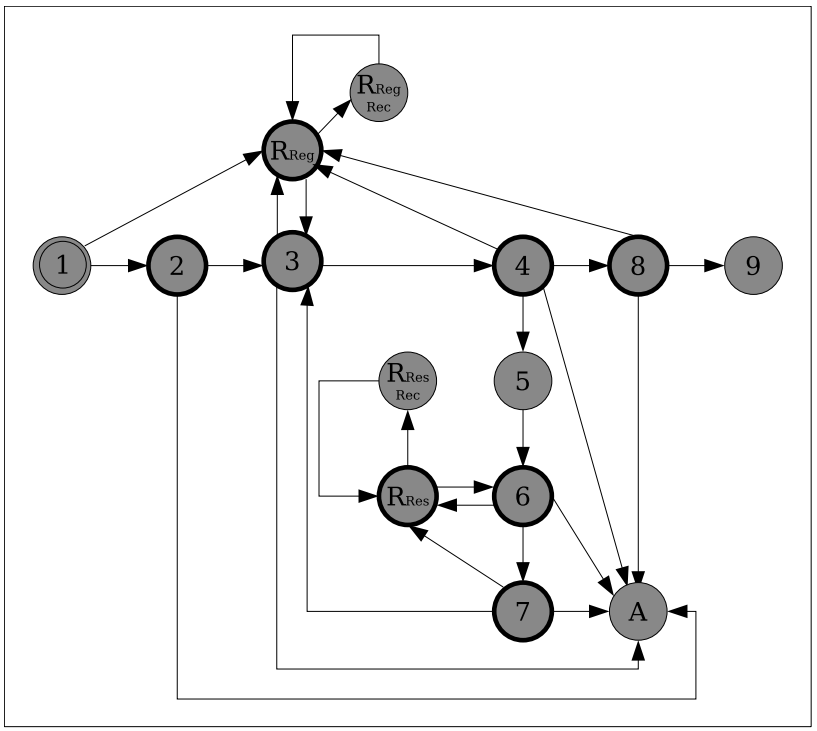
Figure 7.16: Master – recovering / inquiring recombinators

2. Recovering / waiting for NrTasks: wait for a recombinator to call in with the number of results it has already received. *Resolved by posting an MPI_Receive for an MPI_Integer. If this number is 0, transition directly into "Recovering / increasing RecombCount". Otherwise, proceed to "Recovering / waiting for tasks".*
3. Recovering / waiting for tasks: wait for the recombinator from the previous step to send in an array with tasks that it has already recombined. *Resolved by posting an MPI_Receive for NrTasks MPI_Integers, each one corresponding to a task nr. Proceed to "Recovering / adjusting taskbag on successful receipt.*
4. Recovering / adjusting taskbag: remove each of the tasks in the task set – received in the previous step – from the task bag. Then proceed to "Recovering / increasing RecombCount". *No delineating MPI call that triggers state-changing output (logical state).*
5. Recovering / increasing RecombCount: increase the counter for recombinator tracking by one, and return into "Recovering / inquiring recombs". *No delineating MPI call that triggers state-changing output (logical state).*

This last sub state diagram concludes the state flow description of the master process. From the state diagrams, it can be seen that the main state flow has become a little more straightforward than in our first design, while all recovery logic has been removed from it. This comes at the price of a complex recovery procedure – that has already been kept down to a minimum of complexity by simply restarting the recovery procedure at the occurrence of a failure during the recovery procedure itself. Still, the state-flow remains comprehensible, and no tasks should ever be sent to a calculator twice, as long as the calculator in question has not actually failed.

Calculator – main state diagram 7.17:

1. Initializing: as in the previous version – read init files and proceed to pre-synchronization. *No delineating MPI call that triggers state-changing output*



Main state diagram for the calculator process. Notice the added complexity vs. the previous version, as we now have two separate recovery states, both of which are – thankfully – non-composite.

Figure 7.17: Calculator – main state diagram

(logical state).

2. Pre-synchronizing: *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce. On success, flow through to the "notifying master" state. On success, proceed to RESOLVING_TASKBAG. On failure, ABORT.*
3. Notifying master: notify the master that the calculator is ready to perform a calculation. *Resolved through an MPI_Send of an MPI_Integer to the master process.*
4. Waiting for master: wait until receipt of a task from the master. *Resolved by posting an MPI_Receive for an MPI_Integer – the task nr to be calculated. If a task is received, proceed to "calculating". If however, NO_FURTHER_TASKS (-1) is received, transition into "post-synchronizing".*
5. Calculating: calculate the partial result matrix for the given task nr. *No delineating MPI call that triggers state-changing output (logical state).*
6. Sending to recombinator: send the partial result matrix to the correct recombinator. *Resolved through an MPI_Send of an MPI_DOUBLE_PRECISION*

array.

7. Waiting for recombinator: as we described in a previous section, we would have used an MPI_Ssend in the previous section had it been supported. Due to the lack thereof in FT-MPI, we have to wait for confirmation of receipt from the recombinator. *Resolved by posting an MPI_Receive for an MPI_Integer.*
8. Post-synchronizing: *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce. On success, flow through to the "notifying master" state. On success, proceed to "finalizing".*
9. Finalizing: call MPI_Finalize and quit.
10. Recovering regular: the recovery state that is entered in a situation where no (unconfirmed) calculation result is available on the calculator. Just send a NO_TASK (-1) message to the master. *Resolved by an MPI_Send of an MPI_Integer to the master process. On success, return to "notifying master". On failure, go to "recovery / recovering regular".*
11. Recovering / recovering regular: transition back into "recovering regular". *No delineating MPI call that triggers state-changing output (logical state).*
12. Recovering result: enter this state on failure from any state wherein we have an (unconfirmed) result. *Resolved by doing an MPI_Send of an MPI_Integer, containing the task nr. corresponding to the available result, to the master. On success, transition back into "sending to recombinator". On failure, proceed to "recovery / recovering result".*
13. Recovery / recovering result: we end up in this state due to a failure during recovery with a result. Resume the recovery procedure with result. *No delineating MPI call that triggers state-changing output (logical state).*

This is all of the calculator state flow. It is one of the biggest state flow diagrams, but it is also the only one that doesn't have any compound states. The split in recovery states shows clearly how complexity rises when we have to do MPI operations during the recovery procedure (as this might lead to new failures). In this case, luckily, there is only one MPI operation – a send. Redoing this is the most efficient "recovery of recovery" possible. Hence, for the calculator, recovery will actually be optimal.

Recombinator: main state diagram 7.18:

1. Initializing: read in the necessary config files and proceed to "synchronizing". *No delineating MPI call that triggers state-changing output (logical state).*
2. Pre-synchronizing: *Full N-to-N synchronization is accomplished through a collective call to MPI_All_Reduce. On success, proceed to "processing results". On failure, ABORT.*
3. Processing results: composite state. Receive results from calculators and recombine. We elaborate on this composite state in figure 7.19 and the accompanying description.
4. Writing UNF file: when done recombining, write the finished result matrix to an unformatted file (Fortran). *No delineating MPI call that triggers state-changing output (logical state).*
5. Post-synchronizing: *Full N-to-N synchronization is accomplished through a col-*

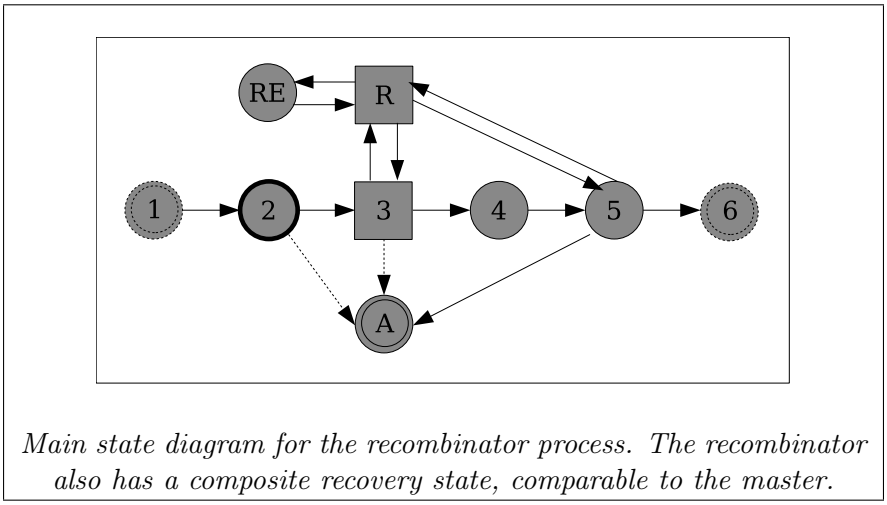


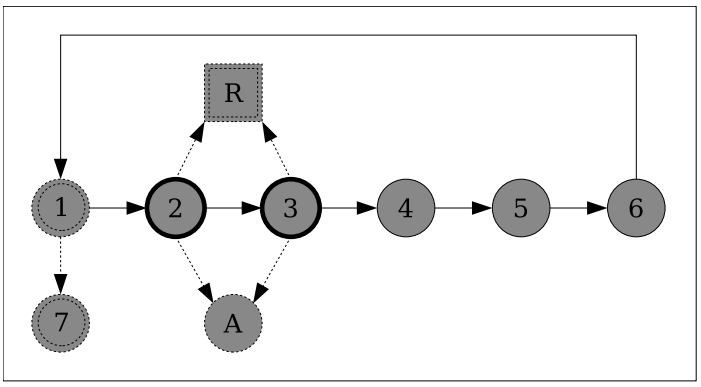
Figure 7.18: Recombinator – main state diagram

lective call to MPI_All_Reduce. On success, flow through to the "notifying master" state. On success, proceed to "finalizing". On failure, ABORT.

6. Finalizing: call MPI_Finalize and exit.
7. Recovering: composite state. Recover from an MPI error at any state, delineated by an MPI call. We elaborate on this composite state in figure 7.20 and the accompanying description.
8. Recovery / error: this state is reached whenever FT-MPI signals a failure during the recovery process. Transition back into "recovering" to restart the recovery process. *No delineating MPI call that triggers state-changing output (logical state).*

Recombinator: processing results 7.19:

1. Processing results: as long as the result matrix remains incomplete, transition into "waiting for results" in order to receive further partial result matrices from the calculators. We keep track of the state of the full result matrix by keeping a list of tasks that have already been performed (which we will refer to as the "result-bag" - it will be used for other purposes than progression tracking as well, later in the state flow). If we have a complete result matrix, proceed to "writing UNF" (virtual state 7). *No delineating MPI call that triggers state-changing output (logical state).*
2. Waiting for result: wait until a calculator sends in a result. On receipt, proceed to "notifying calculator". *Resolved through a call to MPI_Receive for an array of MPI_DOUBLE_PRECISION, listening for MPI_ANY_SOURCE. On receipt, identify and remember the originating calculator for use in the next step. Also, retrieve the task nr. from the tag value.*
3. Notify calculator: notify the calculator that the result has been successfully received, and that it can proceed with further calculations. *Resolved through a call to MPI_Send of an MPI_Integer (value=1) to the originating calculator.*



Sub-state diagram for the processing results stage within the recombinator process.

Figure 7.19: Recombinator – processing results

4. Checking tasknr.: check whether the tasknr. hasn't been recombined yet. Whenever we receive a partial result matrix that has already been incorporated into the final result, drop it. We elaborate on the need for this action after the full description of the state flow. *No delineating MPI call that triggers state-changing output (logical state).*
5. Recombining: recombine the received partial result matrix into the full result matrix. *No delineating MPI call that triggers state-changing output (logical state).*
6. Increasing resultbag: Afterwards, add the task nr. to the "done" list for use during later steps.

The functionality described under the entry for state nr. 4 ("checking tasknr.") is required because, under specific circumstances, the same result might be received twice in a row from the same calculator. Specifically, this situation might occur soon after recovery. Due to FT-MPI's eager send behavior, it is possible for a) a notify to a calculator to return successfully, while b) the message might actually be lost before it arrives at its destination, because of a failure. Hence, the recombinator can never be certain that any calculator has received its notification. This might lead to a calculator resending (but never recomputing) an already recombined partial result matrix.

Recombinator – recovering 7.20:

1. Recovery / sending nr. results: the recombinator starts its recovery procedure by notifying the master of the nr. of partial result matrices that it already recombined into the full result. *Resolved by an MPI_ Send of an MPI_ Integer to the master process. As long as the nr. of results is smaller than the amount required for a complete full result matrix, proceed to "recovery / sending results". Otherwise, transition directly into "post-synchronizing".*

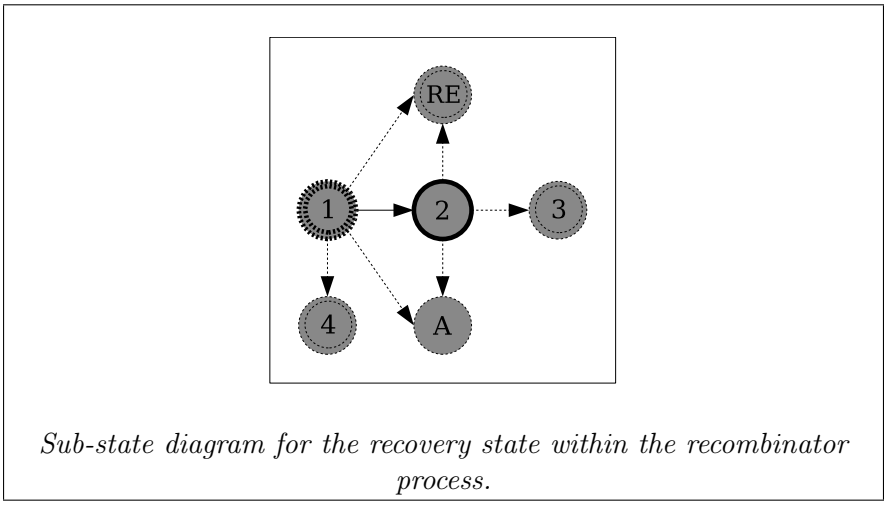


Figure 7.20: Recombinator – recovering

2. Recovery / sending tasks: it then sends the full list of task nrs, corresponding to all the results that have been recombined. *Resolved by an MPI_Send of an array of MPI_Integers. On success, proceed to "processing results".*

This concludes the last state diagram describing the state flow of our second design during run-time. The added complexity in the design should be compensated by the gain in performance during the recovery phase. this is what we will investigate in the next section.

7.5.5 Performance

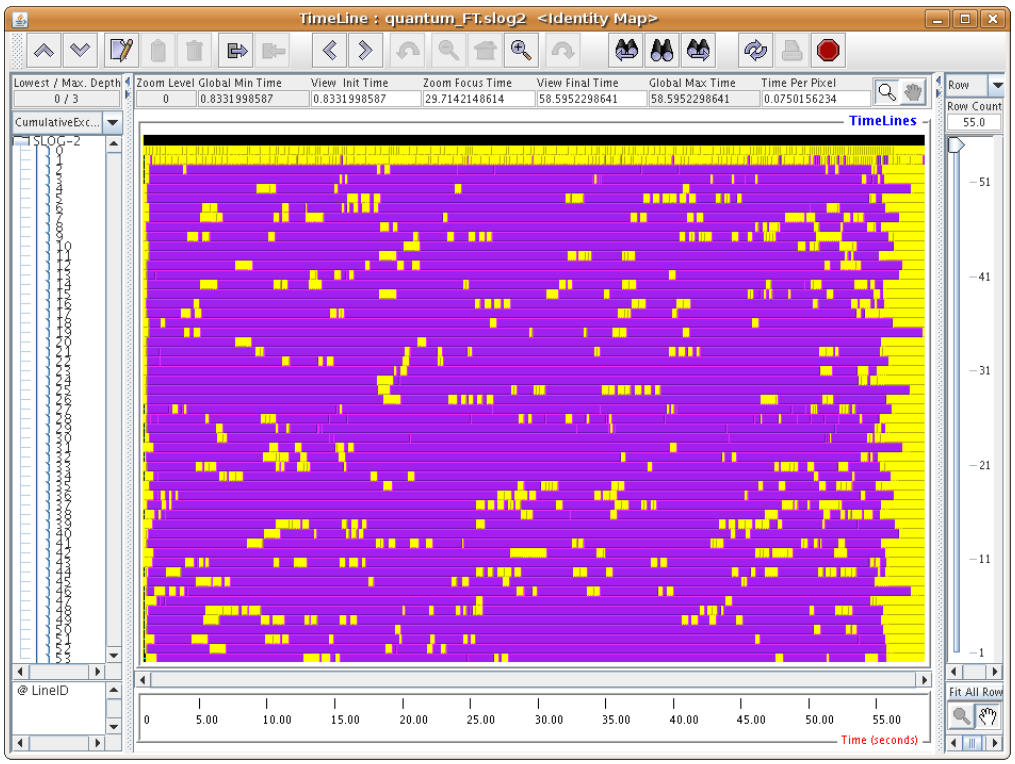
Once more, we put our code through the usual set of performance tests (figures 7.22 and 7.21). Again, we conclude that adding fault-tolerance has not changed anything about the efficiency of the code under non-faulty circumstances, as would be expected.

7.5.6 Crash tests

After the regular performance tests, once again, we instrumented the code with time bombs and ran a number of crash tests "blowing up" under every single state in every process type to test the robustness of the design. Again, the code proved able to survive all attempts at time bombing.

This time, when time-bombing the master after sending tasks to all calculators, 1) the calculators finished their calculations, 2) properly informed the master during recovery and 3) retained and sent their results to the appropriate recombinators after recovery. As designed, no loss of data occurred.

After a number of tests, the only loss of efficiency incurred on a crash would occur when, after a failure, one or more calculators would fail and restart while others where still working on calculations. These calculators will only enter the recovery phase when they finish calculating and attempt to send the result.



Another calculation for $K = 6, L = 2$, this time with the last fault-tolerant iteration of the code. Again, fault-tolerance has not negatively influenced workload processing at higher values of L.

Figure 7.21: Profiling the first fault-tolerant version of the program for $L = 2$

For the moment, this is the best we can do. The only way to eliminate this final loss in efficiency is to make the coupling between the calculating and the communicating parts of the software completely asynchronous. We will further discuss this option in the next section on alternative approaches. As we don't have guarantees on FT-MPI's thread safety, an implementation of such a design is not possible at this moment.

7.5.7 Summary of approach 2

Our previous effort to create a fault-tolerant version of our software was shown to perform sub-optimal due to data-loss during the recovery phase. Hence, we made another attempt at achieving maximum feasible efficiency, albeit at the cost of greater complexity.

We have shown that – given minimal MPI interactions inside the recovery state – it is possible to completely eliminate data loss and redundant recalculation within surviving processes after a failure. We have also shown that further complications

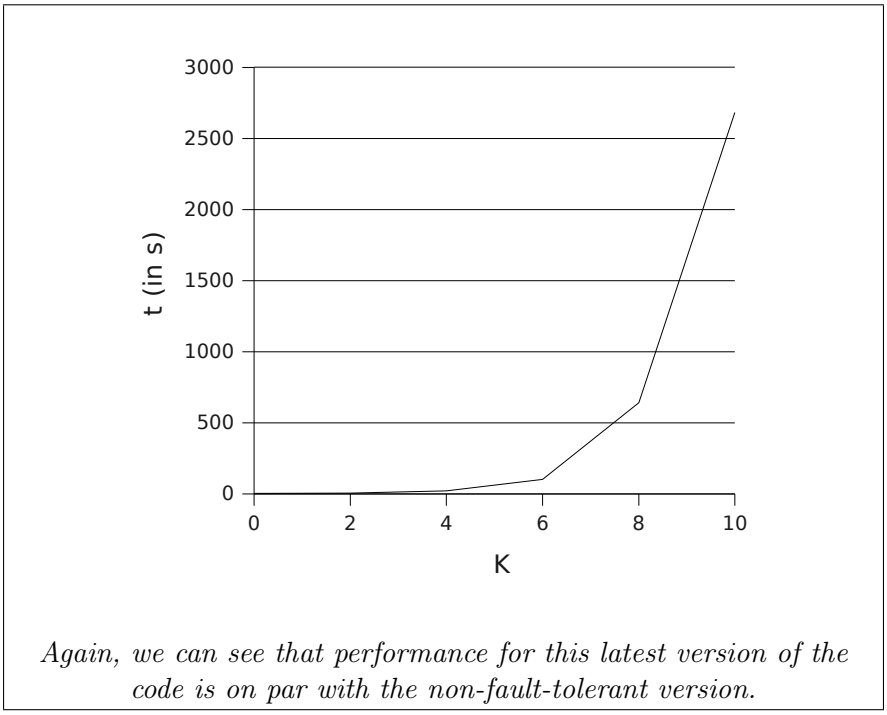


Figure 7.22: Profiling the second fault-tolerant version of the program for $L = 2$

of the recovery process are not necessary as long as we accept failures as being exceptional events. We have shown that, despite an increase in complexity, this can be accomplished in a relatively easy to explain manner, design-wise.

Finally, we have shown that it is not possible to perform any better during recovery, without resorting to unconventional methods like multi-threaded programming. This is one of the scenarios that we will be looking at in the next section on alternative, but further unexplored approaches.

7.6 Alternative approaches

Over the course of the last few sections, we were able to book a number of successes – given a few concrete advance choices with regard to things like FT-MPI communicator modes, message modes, etc.. In the end, we also hit some definite limits in terms of maximum attainable efficiency. In this section, we will be looking at the potential consequences of some alternative design decisions. Decisions that we contemplated, but either a) decided not to implement, or b) weren't able to implement because of technical and temporal limitations.

7.6.1 Solutions without uncontrolled re-spawns

Premise

One of the first decisions we made was to use COMM_MODE_REBUILD. The major advantage of this approach was that we were able to specify a fixed order for the different types of processes (i.e. master = rank 0, recombs = ranks 1...n, calcs = ranks n+1...m), that could also be retained after any recovery procedure.

As we mentioned in the appropriate section, the problem with this approach, however, is that we have no direct control over the re-spawn logic. Consider the case where a) the amount of computational resources available exactly matches the number of processes and b) most failures will be due to failure of the host system it is running on. Any re-spawned process will end up sharing computational resources with at least one existing process without giving the developer any input on the re-spawning policy.

In case of multiple failures among different calculators and / or recombinators, such a situation would force multiple computationally intensive processes to unnecessarily – and un-productively – compete for resources.

Our problem specifically relates to the fact that – give the situation described above – there is process logic that *must* somehow be retained for the whole duration of the computation in order to succeed (the master, recombinators), while there is other process logic isn't crucial (calculators – at least as long as we have a minimum of one). Moreover, whenever a process of one of the necessary types fails, a process of an unnecessary type should somehow be yielding its computational resources.

In this section, we will be taking a look at the means, provided by FT-MPI, to handle this situation. Specifically, we will be looking into the added complexity that comes with any such type of solution.

A solution using COMM_MODE_SHRINK

One natural approach would be to use FT-MPI's communicator mode SHRINK. The major issues that we have to deal with here, are that: a) whenever the master or a recombinator goes down, one of the calculators has to take over its functionality and b) the numbering of the processes will change after recovery of the VM.

Here is an overview of one possible approach to a solution:

1. On startup, have the processes organize themselves in the following manner (given m processes, of which n calculators):
   (a) rank 0: master
   (b) rank 1 ... n: calculators
   (c) rank n+1 ... m: recombinators

2. On startup, have each process store both its own sequence number per process type (i.e. calc 1, calc 2, ..., recomb 1, recomb 2, ...)

3. On startup, have...
   (a) ... the master keep a count of available calculators, and ..

   (b) ... the calculators keep a virtual mapping of the original (real) recombinator ranks at startup to the actual recombinator numbers, dependent upon crashes during execution

4. On a crash:
   (a) surviving recombinators do not have to take any specific measures.
   (b) the master simply substracts one from the calculator count for each process that fails
   (c) the calculators check the rank(s) of the process(es) that failed...
      i. ...in case that the master failed, the lowest-ranked calculator (which now has rank 0 due to the shrink operation) "upgrades" itself to master. This means that all other processes can keep addressing the master as process 0.
      ii. ...if a (or several) recombinator(s) crashed, the highest-ranked calculator takes over its (their) role, and the other calculators adapt their rank mapping accordingly, rearranging the ranks as needed.

Otherwise, this procedure can be combined with both of the solutions described in the previous two sections. The major advantage of this design would be that it doesn't require extra communication during the recovery phase. Therefore, it would not further complicate the state flow of any of the previous designs. The added complexity in its own right, however, did lead to us being unable to implement and test the design in the end.

A solution using COMM_MODE_BLANK

Another approach would be to have the dead processes "blanked out" using the appropriate FTMPI communicator mode. The added complexity here would be greater. Like the above solution, it would require a re-mapping in case of recombinator failure. A similar remapping would be necessary in case of master failure as well however.

On top of this, the master would have to keep a mapping as well, in order to make out which calculators have been "promoted" to recombinator status, and which have been blanked out. A "quick and dirty" way to forget about tracking blanked calculators would be to just attempt a send – that would return 0 immediately. However, since calculators will be taking over recombinator duties, some kind of mapping will be necessary under any circumstance.

All in all, this solution would bring even more extra complexity to the code base than the previous. Yet, it does not seem to offer any advantages over the solution using COMM_MODE_SHRINK. The only potential advantage would be that surviving recombinators would retain their original ranks, but this is being offset by the fact that we need to keep a mapping anyway to identify recombinators that have been remapped onto former calculators.

Hence, we consider this approach the inferior of the two options.

Conclusion

A solution without uncontrolled re-spawns would probably be preferable under any scenario wherein a medium to large number of failures were to be expected. The only means currently at our disposal are FT-MPI comm modes SHRINK and BLANK.

Under such circumstances, a solution using FTMPI_COMM_MODE SHRINK would be preferable over one using COMM_MODE_BLANK. The former only requires major code additions to the calculators and minimal adaptions to the master, while the latter will take some major re-tooling to all process types because the master might end up at a rank other than 0, while recombinators might end up anywhere within the range of calculators due to freakish crash patterns.

Luckily, neither of the two approaches require extra communication during recovery if a strict re-mapping protocol for the master and recombinator processes is held on to by all processes involved.

As an addendum, let us take note that none of these solutions are perfect. A process might crash for non system-related reasons while the machine remains available, or a machine that went down during a previous crash might be up again and available at the time of a next crash. Or a machine, previously in use for another job, might be available at the time of failure. In either case, it would be preferable to use the freshly available machine, but we have no means to do so.

In other words: given that a fault-tolerant MPI implementation, styled like FT-MPI, provides us with a mutable VM model (gaining and losing resources over the course of multiple jobs), a logical evolution for the API would be to eventually become more VM-aware. It would provide the programmer with the necessary means to communicate with the VM from within the boundaries of an individual job – if only within the confines of the recovery procedure.

These means would include operations like requesting new resources from the VM or donating obsolete resources back to it in a more concrete manner than currently allowed by MPI-2 calls like MPI_Spawn. (Perhaps – a bit ironically – more in the vein of PVM.)

7.6.2 Dealing with synchronous recovery

Premise

One of the last remaining problems with both of our fault-tolerant designs lies with the synchronous nature of the recovery procedure. Specifically, recovery has to wait for all processes to make an MPI call in order to be able to initiate collective recovery of the VM. This means that several – available – calculator processes might have to wait for one or more others that are still working on data before they can start recovering.

During this period, the master will not be able to send new tasks, recombinators wouldn't be able to accept new results, and calculators would remain "unemployed" for the whole of the period before full synchronization.

This problem cannot be avoided while holding on to classical, synchronous / single threaded processes. In this section, we will investigate the possibilities of using multiple threads for overcoming this issue.

Design

The crucial part of any design featuring intra-process asynchronous execution, would be to decouple communication from calculation within the calculator processes. The master is constantly listening for either tasks from the recombinators or task requests

from the calculators, and would hence be notified of problems in the FT-MPI VM almost immediately after they were detected. Likewise, the recombinator is constantly accessing the VM as well, waiting for results. It is the calculator that has to be re-tooled to enable it to engage in recovery of the VM, in parallel to any running calculations.

Therefore, we would split up the calculator in two distinct threads of execution, using any of the available tools to do so (e.g. the POSIX threads library or any of its derivatives, the Boost threads library, ...). One thread would take care of calculation, while another would only be concerned with all the MPI-related work.

At startup, set up a process-wide environment with:

1. a shared variable to contain either a task nr. or a NO_FURTHER_TASKS value, as received from the master process
2. a shared memory region to contain a partial result matrix
3. a messaging bit, protected by e.g. a semaphore.

For the calculating thread:

1. Remain in wait state, to be scheduled in only when a task has been received by the MPI thread.
2. Once scheduled in, check the shared variable to decide whether it has a new task, or whether the global calculation process has finished...
3. ...if the job-wide calculation phase has finished (NO_FURTHER_TASKS), finalize (at the thread level).
4. ...otherwise, retrieve the task nr. and start calculating.
5. As soon as a partial result matrix becomes available, notify the MPI thread through the shared bit (e.g. lock it, set it, and unlock).
6. Go back into wait state.

For the MPI thread:

1. Set up a thread-local environment in the following manner:
   (a) create a state variable to indicate either WAITING, CALCULATING or SENDING – start out in the WAITING state
   (b) post a non-blocking send to the master, using a tag that is otherwise unused for any communication from calculator to master. The master will not be posting a receive to any of these sends (from the multiple calculators) until the end of the calculation phase.
2. Wait for a message from the master. During this phase, any failure within the VM will be detected as soon as possible, and the calculator will be able to participate in recovery immediately.
3. On receipt of a message from the master: a) change state to CALCULATING, b) copy the message content to the shared variable and c) notify the scheduler to schedule in the calculation thread.
4. Now, enter an active polling loop:
   (a) Check the status of the MPI VM by doing an MPI_Test on the pending non-blocking send – any failure of the VM should return an appropriate error, with MPI_ERR_OTHER indicating a recoverable error as usual...
   (b) ...if such a recoverable error is caught, participate in recovery immediately

while calculation proceeds as usual in the other thread; after recovery, return to this polling loop.
   (c) ...if an unrecoverable error is detected, abort as usual.
   (d) ...if no failure is detected, check the shared bit (e.g. lock, copy, unlock)...
   (e) ...if the bit is set, change state to SENDING and break from the polling loop.
   (f) ...if the bit is not set, sleep for a short time (e.g. a few seconds) and repeat the polling loop.
5. While SENDING, exchange the partial result matrix with the appropriate recombinator – this involves semi-constant contact with the VM: any failure of the VM will be detected as soon as possible...
6. ...in case of failure, and participate in recovery immediately; after recovery, restart the exchange procedure as usual.
7. ...otherwise, return to WAITING state and recommence the sequence.
8. When the job-wide calculation phase finishes, the master will post a receive for all of the non-blocking sends by the different calculators...
9. ...post a wait for the non-blocking send.
10. Proceed with the rest of the process, as under any of the single-threaded scenarios.

Considerations

The design described above should ensure a proper recovery of the MPI VM under any circumstance, without having to wait for calculation to finish within any calculator process. Moreover, the calculation will not be halted, and the result can be used later on – we can take this into account to adapt our second (single-threaded) fault-tolerant design to make it even more efficient on failure. Recovery should be initiated by all processes at most seconds after detection, and no precious computational resources should be lost in waiting. Overall, CPU cycle loss should be minimal.

Even though the FT-MPI reference implementation is not labeled as being thread-safe, the use of multiple threads in this case should not be an issue. All interaction with the MPI VM is contained within a single thread, and any communication with the other thread happens through different means.

It would even be possible to *fork* the calculation part into another process, enabling both parts to communicate under protection of the OS (pipes) without having to take any further precaution (no need for locks / semaphores / ...). However, this would force us to add yet another layer of fault checking – apart from FT-MPI, complicating the design even further. It would also complicate transferal of the partial result matrix.

One requirement is that MPI_Test returns proper error messages on VM state for the reference implementation. Even though, according to the standard, it is a pure local call – this should not pose a problem.

An alternative solution

The above approach is far from elegant. A far more aesthetically – and practically – pleasing solution would allow us to take on the full problem using only MPI-style

library calls. The reasoning behind this is two-fold: (1) multi-threaded programming adds another layer of complexity which scientific programmers should not have to deal with, but more importantly (2) multi-threaded programming is almost always a platform- specific affair, which goes directly against the MPI philosophy of universal portability.

We think that the best solution, in this case, would be to use an approach similar to that taken with MPI's non-blocking calls, not in the least because they serve a similar purpose: to hide the intricacies of multiple parallel execution paths from the programmer, as well as offering a platform-independent, portable alternative.

This would entail the addition of a few calls to FT-MPI. We suggest the following:

- Add a call FTMPI_Raise_guard: this call would monitor the state of the current job in parallel with any running calculations in the main thread. Whenever a fault occurs, the constant monitoring would detect said fault swiftly and call an appropriate errorhandler.
- At the end of the calculation, there would be a call to FTMPI_Lower_guard. This would cease the frequent parallel monitoring of the job state and return to regular operation. In case that an error handler is active at this time, the call would block until the error handler finishes.
- An appropriate error handler would be called in the case that an error occur under FTMPI_Raised_guard. This error handler would take care of the recovery procedure and return the process to a consistent state after recovery. FTMPI_Lower_guard will block until the error handler resolves this correctly.
- An extra call should be provided to stop the current calculation in the main thread, in the case that its results would be voided by the recovery process for whatever reason. This call – called something like FTMPI_Void_main – would cease executing the current main thread, freeing up CPU cycles for after recovery.

In our opinions, this approach would provide a far more elegant approach to this particular issue, offering maximal transparency combined with minimal additions to the API.

Conclusion

As long as a program can guarantee short computation times in between calls to the FT-MPI VM for every process, synchronous recovery should not pose a problem (as is the case with most example programs delivered with the reference implementation). In any case more complex, however, some solution of the type above is going to be absolutely necessary.

In our case, for high enough values of K, any single computation might take hours before having to do an MPI call – delaying any possible recovery by as much. This delay in recovery is unacceptable, potentially leaving lots of resources unused – waiting for extant work on one or two calculators.

Through the above design, we have shown that it is possible to minimize the effect of synchronous recovery on the overall performance of the recovery procedure. We have shown that, to accomplish this, it is necessary to split up the calculator process into two parallel execution units using multi-threading or similar techniques.

Using one thread for MPI communication, and one for calculation should solve the problem - even if we are using a non thread-safe FT-MPI implementation.

In fact, we expect the above approach to provide a useful pattern that will be of frequent use in any serious fault-tolerant software using FT-MPI – it should probably find its way into the more advanced documenting examples bundled with the reference implementation.

There is one thing that bothers us in this case though: in our example design, we are forced to use a dummy non-blocking send, combined with periodic calls to MPI_Test, as a kludge solution to poll VM state. In any platform that allows for user-driven fault-recovery, the ability to actively investigate global "sanity" of the system will be crucial. Being as important a functionality as it is, we propose that the ability to poll VM state should probably be given its own specific call.

One more minor worry of us concerns the lack of attention to threading cases in the FT-MPI specification. Even though we didn't have to perform MPI calls from multiple threads within the same process, one can certainly imagine cases in which this ability might be desirable. However, how will the multi-threaded setup be handled in case of a failure?

Do all threads within a process have to participate in recovery, as if they were individual processes? What would be the effect of this on process work-flow? Or on the internals of FT-MPI? The complexity of having to implement a thread-safe MPI? Or is there only a need for one to do so? In that case, what happens if one thread posts an MPI call while recovery of the VM remains unfinished? Should it be held back? What happens if an MPI call gets posted during (user-driven) job recovery? Won't this complicate user-driven recovery even further? All of these are questions that we believe should be addressed in the specification.

7.6.3 Handling recombinator crashes

Premise

Any of the previous designs spreads all of the state data of the job over the recombinators. Loss of the master will not lead to state loss at all – its state being fully restorable from data in the other processes. Likewise, loss of a calculator will, at most, lead to the loss of one partial result matrix – which is not dramatic.

Even the loss of a single recombinator does not lead to 100% data loss, as all of the computed data on the other recombinators remains available. One way to minimize concentrated data loss is to interleave tasks for different recombinators as much as possible in the serving sequence – but even in this case, failure of a recombinator late in the calculation process will lead to a lot of lost data / computing time.

The only way to really minimize the impact of a recombinator crash, is to provide some kind of backup. We conceive of two – possibly complementary - ways of containing data loss. One is to provide for "sleeping" redundant recombinator processes. The other is to keep customized, user-controlled checkpoints on persistent storage. We will investigate each of these options further in its own section.

Adding redundancy

One logical way to counter the loss of a recombinator would be to provide other, redundant recombinator processes. These would take over chores in case that their "originals" were to be lost due to failure. The major advantage of this approach would be that it doesn't require any interaction with the filesystem, or another kind of persistent storage.

There are some major disadvantages to this solution, however. The most obvious one, is that – as we mentioned before – a failure will probably mean that one or more computational resources are lost to the VM – either because they unexpectedly went down, or because of network problems. Afterwards, we will also not be able to make out whether these resources return to the VM at one point or another. Also, we are not likely to leave computational resources unused at startup – i.e. there will not likely be any comparable redundancy among the available computational resources to make room for the backup processes.

On top of this, as we also previously addressed, we have absolutely no control over how the computational resources are distributed among the different processes. In other words, these redundant recombinators might end up anywhere at startup. So we might have a recombinator and its redundant copy end up on the same host: in this case, both would probably be lost during a failure. Otherwise, it will end up on another computational resource, already in use for either calculation or another recombinator.

If we set up our system in such a way that the redundant recombinators only function as backup storage – as long as their "originals" have not failed – at least computational power will not be significantly reduced. But, in case of failure, the backup recombinator will end up competing for resources with whatever other process is residing on the same CPU. In this case, we would have to consider the trade-off: whether it is worse to have the redundant recombinator hinder whatever other process runs on the same resource, or whether the saved data is actually worth it.

Another issue to consider is how to keep the redundant copy up-to-date. Necessarily, we would have to send (broadcast) an incomplete, accumulated result matrix from each "original" recombinator to its backup(s). From the recovery p.o.v., this would ideally happen after each recombination. However, it is hard to assess the impact that this might have on overall network performance – specifically on the network traffic necessary for regular operation. As always, ideally, we would like to keep the impact of fault-tolerance / recovery on regular operation as little as possible.

Altogether, this does not seem like the most practical solution. It would perhaps have been a more practical approach under, say, PVM, where the user has tighter control over where processes are to reside, and how the are to act w.r.t. the VM. But MPI, even FT-MPI, simply does not seem to offer the right approach for this kind of solution. On top of this, the network overhead and internal competition for CPU resources would pose a problem on any parallel platform.

Customized checkpointing

The other logical approach would be to regularly make a backup a) of the accumulated incomplete result matrix and b) the list of tasks that have been computed to

persistent storage. The recombinator regularly notifies the OS to do a flush to shared persistent storage, to properly safeguard the data. With this approach, we would effectively be creating a form of user-implemented and -controlled checkpointing.

After a failure plus re-spawn (or, under MODE_SHRINK, re-purposing of a calculator), the renewed recombinator would fetch the most recent version of the accumulated incomplete result matrix from disk, as well as the list of calculated tasks. The "restarted" recombinator could then participate in regular recovery by notifying the master of the tasks that it was able to retrieve from checkpoint, thereby reducing data loss after failure.

This approach is a very attractive one for a number of reasons. For one, it does not require any extra communication overhead, being controlled fully by each individual recombinator. Moreover, any writes from a recombinator to its "checkpoint" could be greatly accelerated through caching by the OS, given enough memory on the appropriate node. Flushes to disk can be taken care of by the OS, in parallel with the recombinator process without greatly hindering it during operation.

The only performance hit would be taken during recovery, when a recovered failed recombinator has to read its checkpoint before being able to participate in global recovery. The gain from retaining a lot of computation history should more than compensate for this however.

All in all, this approach represents the best aspects of user-driven fault-tolerance. It is a case in which all the advantages of checkpointing can be retained where it matters, without wasting any resources on checkpointing messages/ processes that are not of primary importance to global state. It even enables us to cherry-pick the exact necessary data within a single state-retaining process to minimize the checkpointing footprint. And best of all, this can all be done without greatly impacting fault-less operation.

Conclusion

One of the last optimizations left unaddressed pertained to the data loss suffered after failure of one or more of the state-retaining recombinators. Over the course of this section, we had a look at two different, but not necessarily mutually exclusive options.

The first, providing for redundant recombinator processes that periodically synchronize with their respective "originals" seems feasible but not very practical, due to considerations of multiple processes competing for the same CPU resources and the inability to control the execution of re-spawns.

The second option was to implement user-controlled, custom-optimized checkpointing for the recombinators – periodically writing the incomplete result matrix to persistent storage to retrieve it on recovery if needed. This solution does offer a significant advantage, potentially retaining computed data over multiple failures without significantly impacting fault-less performance given enough memory per node to make optimal use of disk-caching.

All in all, customized checkpointing seems like a near-perfect solution, beautifully illustrating the advantages of user-controlled fault-tolerance, and would for certain be an interesting option to add to our software in future re-factoring.

7.7 Conclusions and summary

At this stage, the time has come to draw some final conclusions from the refactoring process and re-iterate the general conclusions that are to be drawn from the example application one final time.

7.7.1 The Fortran phase

For completeness' sake, we start out with the pure Fortran-based part of the work. Allthough it doesn't directly add to the answer for our two last research questions, knowing about it is important to understand the greater scheme of things to come...

A very large number of issues have been discussed during the first refactoring chapter. We shall have a look at them one by one, one last time, and try to summarize our first set of conclusions.

Over the first few sections, we introduced the issue at the base of the research presented in this chapter: the quantum nuclear scattering problem. We had a look at its basic nature, including a description of the important parameters to this discussion: the hyper-momentum K and total angular momentum L. We gave a brief overview of the original software solution to the problem, written in Fortran 77. This software produced a set of result matrices that was written to a number of unformatted Fortran data files – to be further processed by other software. Because a) these files were unformatted, and b) we didn't want to touch the back-end software, at least part of the software would have to be retained in Fortran.

We discussed how, due to memory management issues, we first had to port this code-base over to Fortran 90. After this first – relatively minor – re-factoring iteration, a review was performed. We were able to clearly illustrate, through experimental results, that this solution did not scale to any reasonable values for K or L.

Hence, we decided that a major re-factoring iteration was due. Parallelization was the only practical means to attain the processing power that we needed. Given that MPI has become the de-facto standard for parallel programming solutions over the past years, and that it defines a very clear and complete Fortran interface, it was the natural tool to help parallelize the original code.

A major rewrite on all levels of the code was performed, making the code far more modular and flexible than it had been – a change that would turn out very useful in later steps. More importantly, the application was changed into a master-slave type of parallel program. The master delivered tasks to, and received partial result matrices from the slaves, recombining them into a set of full result matrices that were written to file as previously.

After fully re-tooling and experimentally testing the new F90 code, we came to the following conclusions:

1. First, the parallelization allowed the software to scale to the kind of level that we were looking for, at least for low values of L.

2. Second: for high levels of L, there was a saturation problem at the master end of our architecture that needed to be resolved.
3. Third, in light of the previous, plus the heavy re-factoring that still needed to be done, we wanted to change over to a computer language / programming paradigm that would ease future transitions in the code. An object oriented approach using C++ seemed ideal.

7.7.2 The C++ phase

In this phase of our research, we addressed our general question concerning how to integrate Fortran-based legacy applications with newer, object-oriented C++ code.

While looking over the design at this stage, it was decided to retain three parts of the software in Fortran: calculation, recombination and writing out the UNF files. All other parts of the software – i.e. setup, organization and communication – would be ported over to C++.

This required us to look at the issues that were to be expected when setting up communication between C++ and F90 code. After examining the different complications, it turned out that communicating between C and F77, or even between C++ and F77 did not pose a real problem – given a few caveats.

Communicating some of the features in F90 – specifically pointers – back to C++, however, prove less straightforward. After some thorough research, it turned out that these problems were due to the run-time behavior of F90 arrays. For a range of reasons, F90 requires a rather large amount of run-time meta-data. This meta-data is kept in a compiler-generated, hidden structure called a dope-vector. Contrary to all other (visible) data structures in Fortran – which are passed by reference (pointer) – the hidden dope vector is passed by value (copied) between function calls for performance reasons, producing some rather bizarre behavior when passing it straight back to a C++ function not expecting this.

After looking around for a solution to our problem, we came up with a number of different existing projects that are available for free from the web. However, it turned out that none of these provide the exact functionality that we need. One of the potential solutions - a software package called Chasm – did however provide some fundamentals that we could use for a custom solution – specifically, it offered some much needed insight into the actual structure of the elusive dope vector.

Given this basis, we set out on developing a custom solution to our problem: the *arrays.h* library. After a lot of hard work, we managed to come up with a solution that satisfied all of the following prerequisites:

1. It provides a proper type for Fortran pointers (i.e. dope vectors) which is usable in both F90 and C++.
2. It allows for allocation to a pointer to array in either language.
3. It allows for transparent passing of pointers to array from either language to the other.
4. It allows for deallocation of a pointer to array, allocated in either language, from the other as well.
5. It allows for seamless passing of pointers to array as members of structs or as function return types

6. It supports column-major order indexing in both C++ and F90
7. It supports same-style indexing in C++ and in F90 (i.e. indexing multiple dimensions through a single '()' style operator).
8. It supports different levels of portability, i.e. enables a developer to choose between a faster but low-level, compiler dependent implementation or a higher-level, compiler independent implementation as needed.
9. It supports the use of Fortran array-pointers at similar runtime speeds in C++ as in Fortran itself – again, with both compiler dependent and compiler independent implementations available.
10. It is able to accomplish this through a relatively straightforward interface: a straight library in C++ (internally making heavy use of templates).

This comprises all the functionality that we will need to comfortably deal with the C++ porting problem, and more. First, extra features have been added that should make the library useful for C++-F90 cross-language programming of any nature. Second, performance is not lagging either: through careful application of meta-programming techniques in selected places, we have been able to produce performance figures for read / write accessing of arrays that are most interesting in their own right – beyond just the issue of cross-language programming. To the extent of our knowledge, this is the first piece of software to accomplish all of these goals and offer these kinds of features.

As a side effect, unit-testing of the meta-programming part of *arrays.h* has led to the development of another feature that we haven't been able to find in any of the large available C++ template-meta-programming libraries: a straightforward implementation of template meta-loops, able to do straight iterations without involvement of concepts like meta-collections and other complications that go beyond the needs of many simple applications.

Given all of these elements, we believe we have provided the interested reader with all the necessary information and tools needed to make their Fortran-based legacy code accessible from any C++ code in the most elegant way possible, thereby providing ample material in answer to our third general research question.

With all of the previous out of the way, we were able to to a rather straightforward port to C++. As planned, everything but the actual individual calculations, recombinations, and writing or unformatted result files in this new version is handled by C++ code. After completion of this step, the code-base comes over as cleaner and leaner, and – most importantly – more adaptable than under its previous incarnation. The ease with which we introduced FT-MPI into the code later on proves that this is not just matter of esthetics. Experiments have shown that performance did not suffer from the change of programming language.

One last issue that needed resolving before starting out on adding fault-tolerance was the saturation issue with the master process, that lead to starvation of the calculators for values of L above 1. This problem was solved by taking parallelization further, decoupling the recombination procedure from the master process and wrapping it into its own type of process, one of which will be running for each result matrix to be computed. This moves the code away from the original, relatively sim-

ple master-slave paradigm. Its work-flow is more complex now, which will add its complications for adding fault-tolerance in the next phase because computational state is now spread over a greater number of processes.

7.7.3 The FT-MPI phase

The only research question left unanswered concerned the measures needed to make existing MPI-based code FT-MPI "ready". Given that our example legacy-application had finally, over the course of the past refactoring phases, been worked into the state envisioned at the start of our work, we were finally ready to take it through "the works".

Here, we will present a general overview of our findings during this stage of development. In the final, conclusive part of this thesis, this practical experience with FT-MPI, will be used to eventually attempt to deduce a "recipe" of sorts to aid future developers in refactoring any general piece of MPI-based code for FT-MPI.

Now, during the early development of the FT-MPI-based version of the legacy software, we came to the conclusion that *state* was going to play a major role in the development process. While using FT-MPI in practice, it turned out that state manifested in two important ways: intra-process state, and inter process state. Intraprocess state describes the progress of the work-flow in each type of process, while inter-process state concerns the set of states of all processes in a job, which is considered during recovery.

State became a concrete part of all the designs that were tested during development. It primarily serves to set milestones for: a) rolling forward or rolling back a process during recovery (intra-process state), b) knowing how and where to enter the recovery process when needed (intra-process state) and synchronizing between the different processes during recovery (inter-state recovery).

Two different versions of the example software were developed. A first aimed for maximum simplicity, at the cost of efficiency during recovery. Simplicity was maintained by writing the recovery procedure in such a way that it didn't require any MPI communication. This attempt prove that it was possible to write practical, real-life, usable fault-tolerant software using FT-MPI without overt complexity. Moreover, it was able to provide full fault-tolerance without extra overhead during faultless operation. The major disadvantages to this version were that a) all results, residing on calculators during recovery were lost, and b) recovery could only begin at a moment when all calculators were available for communication over MPI – i.e., recovery had to wait for all calculators to finish their current calculation.

A second version tried to fix the problem of lost results on the calculators by involving them in the recovery process. This lead to a need to use MPI communication during recovery, making it quite a bit more complex as failures could now actually occur during recovery itself. The end result took quite a bit more work than the first version, and is a fair deal more intricate as well. However, it is still maintainable enough, and provides nearly the best possible fault-tolerance available for this particular piece of software – hence, we are satisfied with it. It does not fix the calculator wait problem during recovery however.

A few further optimizations have been thought of. An ideal version of the soft-

ware would take fault-tolerance to the maximum level by doing periodic, software controlled checkpoints of the unfinished result matrices. In this way, only the recombinators have to checkpoint, and even then only a subset of the data: exactly the kind of application we originally had in mind when we contemplated user-controlled fault-tolerance.

Two other optimizations would not so much enhance fault-tolerance, but rather the speed of the recovery procedure and the stability of the post-recovery VM.

Making adaptions to use FT-MPI communicator mode SHRINK would make sure that after recovery, any CPU would never be running more processes than it can handle, using all available resources from the beginning of the job. It would, however, also complicate the software because processes would have to be able to switch rank nrs. during the job, with calculators switching role whenever a recombinator or the master would fail.

A last adaption would fix the calculator wait problem by making the calculators multi-threaded. Any calculator would run their MPI-interactions and calculations in parallel under different threads, using MPI_Tests on dummy non-blocking communications to test the state of the VM. This would allow calculators to take part in recovery while calculations would remain running.

This optimization would greatly optimize the recovery procedure. In fact, we state that it wouldn't just be useful for our software, but for any software using FT-MPI. One other thing came to our attention: due to buffering, some types of processes – specifically "provider" type off processes – might reach MPI_Finalize before a failure can be caught, blocking the VM from proper recovery. To alleviate this, each job should start by synchronizing all its processes. We are of the opinion that these two issues should be elaborated on within the FT-MPI documentation, as well as within the examples delivered with the reference implementation.

7.7.4 Overall conclusion

This concludes our discussion on the work we did to make the quantum clustering software fully fault-tolerant. Throughout this chapter, through discussion and illustration by experiment, we have shown that it is in fact possible to write fault-tolerant MPI-based software using FT-MPI – real, production-level software with an actual application, that both recovers well (with a minimum loss of job data after failure) and efficiently (with fast and safe recovery and optimal usage of the available resources). As far as we know, this is one of the first "for-production-use" projects to take FT-MPI to this stage. It has taught as quite a few things that would be useful as feedback to the original project.

More-so, on the way there we have produced a number of peripheral developments that are interesting in their own right.

Part IV

Conclusions

7.8 Questions and Answers

In this dissertation we have addressed three key research questions:

1. With regard to MPI implementations: what are the current options w.r.t. fault-tolerance and MPI, what would be the best approach for the particular case of small research groups working with low-budget hardware and what yet remains to be done in this area to make fault-tolerant use of MPI valid.

2. With regard to MPI-based software: what – if any – changes need to be introduced to make it fault tolerant, given an MPI implementation that supports it. What would a generic "cookbook" for creating fault-tolerant MPI-based software look like.

3. With regard to Fortan-based legacy code: what needs to be done to make easy integration of said code with modern C++-based code work in a user-friendly manner.

We have provided an answer to each of these questions, which we now summarize:

7.8.1 MPI implementations and fault-tolerance

Concerning MPI implementations: what are the current options w.r.t. fault-tolerance and MPI, what would be the best approach for the particular case of small research groups working with low-budget hardware and what yet remains to be done in this area to make fault-tolerant use of MPI valid.

When seeking out a fault-tolerant MPI implementation, there are two basic options: those implementations that offer transparent fault-tolerance and those that support application-controlled fault-tolerance.

MPI implementations offering transparent fault-tolerance have the major advantage that they don't require the user to make any alterations to his MPI-based

software to accomplish fault-tolerance. This is also the area where, to date, most research on fault-tolerance in MPI has been performed. The major disadvantage to this approach, however, is that it is a very heavy-weight solution which fails to scale – especially under a scenario with heterogeneous, geographically spread resources. Specifically, this solution comes with a fixed overhead per resource which grows with the size of the target parallel platform, even during regular (i.e. faultless) operation.

There is currently one MPI implementation – FT-MPI – which offers the alternative approach, providing the user with the ability to control and implement fault-tolerance from within the application. The apparent disadvantage here is that a limited set of adaptations to the MPI specification is needed for this to work. Likewise, the MPI-based software will have to be adapted as well to make it fault-tolerant. The major advantage is that the developer can tailor any fault-tolerance related features to the needs of both the software and the platform on which it is supposed to run. Most importantly, the amount of work needed during "regular" operation can be minimized, and hence the impact on scalability is significantly suppressed, and can even be negated in a number of cases (as, in fact, we demonstrate in our specific example). All of this makes FT-MPI into the best solution to our needs, and those of other teams with compareable needs and limitations.

One major shortcoming of FT-MPI, however, was in the particular implementation of its state database, which was stored in a centralized, non-fault tolerant name-service. This is the first part where we have produced original work, solving the problem by making the name service "universally pluggable" – allowing any name service (many of which provide fault-tolerance out-of-the-box) to substitute for the current implementation. The major hurdle to overcome here was the issue concurrency within an environment that provides only very limited means for handling complex distributed transactions. Through research, we came up with an unreliable locking scheme which manages to addres all the requirements.

Overcoming these hurdles allowed us to go ahead using FT-MPI for a real-life case.

7.8.2 Creating fault-tolerant MPI-based software

With regard to MPI-based software: what – if any – changes need to be introduced to make it fault tolerant, given an MPI implementation that supports it. What would a generic "cookbook" for creating fault-tolerant MPI-based software look like.

As far as we are aware, we are one of the first research groups to have used FT-MPI for a "real-life" application (apart from some work by Al Geist [57]). Thus, our work on the quantum nuclear scattering software – and specifically the conclusions we can draw from it applying to fault-tolerant programming under FT-MPI in general – serve as our second body of original work. What follows is an overview of (1) some general remarks concerning the use of FT-MPI – mostly items that the FT-MPI designers themselves do no seem to have given a lot of thought to, but which did turn out to be an issue for a "real" application, followed by (2) a "cookbook" of sorts for those who want to design software using FT-MPI.

General remarks

On of the most important changes from regular MPI to FT-MPI is that any communication can now lead to three different results: success, recoverable error or unrecoverable error. Success will lead any program into its next valid state, a recoverable error should bring the software into a recovering state and an unrecoverable error should lead to an abort state. This greater emphasis on internal state-flow will make any FT-MPI based software explicitly more state-based than its non-FT equivalent.

Moreover, whenever there is MPI communication going on within the recovery state, which might produce a recoverable or unrecoverable error itself, it is absolutely necessary to provide "recovery-recover" states as well (etc., until no recovery state contains any more MPI communication). It seems like this has not been given much thought by FT-MPI's developers, as any examples of this are sorely lacking from the distribution's example programs.

One more item that FT-MPI's developers seem to have overseen is the necessity of (at least) a "post-synchronization" mechanism – this to make sure that it is not possible for certain processes to MPI_Finalize before detecting any failures. In some cases, a "pre-synchronization" state can be usefull as well. Due to errors in the current implementation of MPI_Barrier, another synchronizing call should be used to emulate its functionality, e.g. an All_reduce.

One more important item to consider is FT-MPI's COMM_MODES. By far the easiest mode to develop for is COMM_MODE_REBUILD. This is because it allows the developer to assume a stable virtual machine layout. However, the major problem with this approach is that, as a developer, you have no control whatsoever about where (i.e. on which resources) the failed processes are respawned. This might lead to serious problems if too many processes get respawned on the same resource. This is why, in practice, we would recommend using COMM_MODE_SHRINK or COMM_MODE_BLANK wherever possible (in our case, MODE_SHRINK seemed best). Again, FT-MPI development documents do not seem to discuss this issue anywhere.

To the defense of FT-MPI's designers, giving the developer greater control over the respawning mechanism would require far greater adaptions to the MPI specification than we have with the current FT-MPI specification. The same is true if we were to fix the one great omission in FT-MPI: to add the ability to grow MPI_COMM_WORLD during operation. In other words: we can retain the size of the main communicator and shrink it in case that we lose a process – but we cannot grow it in case that we were to add extra resources to the VM during operation. In order to accomplish anything like this, we would have to use MPI-2's intercommunicator mechanisms. These, however, are not yet implemented for the current version of FT-MPI.

One last thing to remark is that FT-MPI is incapable of recovering a job until all processes within it reach a global synchronization point, i.e. until all the processes have reached an MPI communication point after the fault has occurred. In the case where some processes might be spending considerable time between communication points – i.e. in a calculating state – all the processes will have to wait for these to

finish before recovery can commence. This can seriously delay the ability of other processes to recommence working after a fault has occurred. The only practical way to deal efficiently with this situation seems to be a multi-threaded approach: do all calculating within one thread, while the other frequently checks FT-MPI for faults by means of non-blocking communication tests.

Given these general remarks, we offer the reader the following "cookbook" for writing fault-tolerant parallel computing software using FT-MPI...

The Cookbook

Our cookbook will try an take into account all of the previous remarks except the last one (i.e. dealing with "recovery" wait for global synchronization). This will be dealt with in an addendum to the regular "recipe", as it is easily illustrated as an addition to regular development practice without over-complicating the basic approach.

In order to make a given MPI-based program FT-MPI ready, refactor it in accordance with the following steps:

1. Map and explicitize the intra-process state flow.
   - Map-out the state flow within each type of process – the important state transitions happen on each MPI communication.
   - Sometimes, it is usefull to explicitly distinguish a number of non-MPI related "purely logical" state transitions as well - especially to and from calculating states. These states and transitions do not directly influence the fault-tolerance issue, but they make the design more readable and might make it easier to introduce new features later on (like multi-threading in our addendum).
   - Using this adapted design, make your software explicitly state-based. Use the appropriate constructs for your particular programming language, e.g. a state variable plus switch statements for state testing and transitions and functions to represent individual states in C(++).

2. Add a post-synchronization state.
   - Synchronization must happen between all processes before MPI_Finalize. This to make sure that all run-time errors are caught and recovered from before any process finalizes - a situation which can occur because of FT-MPI's (standards compliant) use of eager send mode.
   - Since MPI_Barrier is broken, it is best to use an MPI_Reduce or similar call to accomplish this.
   - Sometimes, it might be usefull to add a similar "pre-synchronization" state as well for the sake of clarity.

3. Add a mapping state at the beginning.
   - The intent is to map actual process ranks to a virtual ranking in order to ease rank re-ordering when using COMM_MODE_SHRINK or COMM_MODE_BLANK.
   - When using either of these COMM modes - especially MODE_SHRINK - these are bound to change. A remapping during the recovery state will

allow the rest of the program to keep functioning as if nothing happened, without the need to be aware of any changes.

4. Add ABORT and RECOVERING states where necessary.
   - In FT-MPI, each MPI communication can result in either success, a recoverable error or an unrecoverable error.
   - Thus, for each MPI communication (i.e. state change), add a transition to RECOVERING or ABORT.
   - Depending on the internal logic of the particular application, there might be more than one RECOVERING state. (In fact, our example software demonstrates this in one instance.)
   - Naturally, there is always only one ABORT state.
5. Map and explicitize the inter-process state flow.
   - Now, it is crucial to find out (1) which combinations of states among the different process type states are valid and (2) which valid combinations of states is (are) most advantageous to return to after recovery.
   - Add transitions from the RECOVERING state(s) to the appropriate regular states for each process type in such a way that recovery will always lead the software to resume operation in a valid inter-process state.
   - This includes the possibility that the job might be able to return to multiple inter-process state sets depending on which is most advantageous. Do try to keep things simple though.
6. While conceptualizing the recovery procedure...
   - Make sure that your recovery state is fault tolerant, i.e. whenever an MPI call is used during recovery, make sure that the procedure accounts for the potential of a fault being detected during this call.
   - This indicates, indeed, the possibility that you will need a "multi-layered" recovery procedure (i.e. recovery-within-recovery).
   - Generally, the easiest approach in this case is to restart the recovery procedure (as we demonstrate in our example case).
   - The best solution by far, in this respect, is to refrain from MPI communications alltogether during recovery.
   - One also has to make sure that only a minimal amount of data must be disposed of (i.e. loss) during recovery.
   - The above two goals are not always reconcilable (again, as demonstrated by our specific example).

The above "recipe" should help a developer make his MPI-based software fault-tolerant while minimizing the chance of any "hitches" occurring along the way. It does not solve the synchronization problem, however – this is to say: before FT-MPI is able to recover, all processes in a job must synchronize on an MPI communication. If and when one or more processes are spending a long time calculating before engaging in the next MPI communication, the other processes in the job might have to wait for quite a long time.

Currently, there is only one way to tackle this problem, and that is by means of a "hybrid" solution, combining MPI's multi-processing model with a multi-threaded approach. The major disadvantage of this approach is the added complexity, and the fact that multi-threading issues are almost always a platform-dependent matter.

This approach requires the following addition to the basic "recipe":

1. Before any calculation state, instead of directly starting with the calculation, launch it in a new thread.
2. Within the original thread, the code must now engage in active polling of:
   - the state of the current job, by means of tests on non-blocking MPI calls
   - the state of the calculating thread, e.g. by checking a semaphore-protected shared variable
3. When a fault occurs, it can thus be discovered relatively soon.
4. Moreover, a fault can be resolved while the calculating thread keeps running (and thus without losing calculation time).
5. Optionally, the calculating thread could be terminated as well if it turns out that its future results would not be usefull after recovery anyway – again, this spares us from waisting CPU time.
6. If calculation is allowed to continue, do not forget to retrieve the results from said calculation(s) properly when they finish after recovery.

We also suggest that a better solution would be to provide a set of FT-MPI specific calls in the same vein as MPI's non-blocking calls, likewise shielding the programmer from the complexities (and – more importantly – the platform-dependent aspects) of multi-threaded programming.

As far as our experience reaches, the above cookbook and addendum represent the optimum approach to making regular MPI-based software fault-tolerant. As far as we know, nothing of the sort has been produced to date, and hence we hope that it might prove usefull to future research.

The only alternative option is to use so-called "naturally fault-tolerant" [57] algorithms, but these basically trade "acceptable" degradation in the final result for fault-tolerance, using COMM_MODE_SHRINK in its most basic form – an approach which could be usefull in case of Monte-Carlo simulations and like techniques.

7.8.3 Integrating legacy Fortran code with C++

With regard to Fortan-based legacy code: what needs to be done to make easy integration of said code with modern C++-based code work in a user-friendly manner.

A lot of extant scientific code is written in Fortran. Despite Fortran's known strengths, especially with regard to the optimization of matrix operations, it also suffers from a number of known weaknesses. Fortran 77's most obvious issue was its lack of dynamic memory management features - an item which becomes most

problematic when dealing with very large array structures, as demonstrated by our example problem.

These problems were mostly fixed in Fortran 90 by adding features supporting pointers and dynamic arrays, but F90 introduced a new problem. In F70, it was relatively simple to exchange data with C(++) code – in fact, back then it was the only option if you really couldn't work without dynamic memory. In F90, however, exchanging data – specifically dynamic memory structures – with C++ has become problematic.

The reason for these problems is that F90 needs to retain meta-data on dynamically allocated memory, allowing programmers to pass dynamic memory structures to functions taking an open array argument and allowing for run-time optimizations. The problem here is that said meta-data, contained within a so-called "dope-vector", is hidden from the user. Moreover, its structure is not specified by any standard, and hence compiler-dependent..

A few projects have previously attempted to tackle this problem, but all of them fall short at one point or another, and they are all very heavy-weight solutions requiring techniques like multiple-pass compilation and code transformations. A more complete and user friendly solution was in order. this is where we present our third body of original work.

To tackle the problems inherent with exchanging dope-vectors between F90 and C++, we developed the "arrays.h" library, which allows developers to exchange dynamic memory structures between F90 and C++ by simple inclusion of a header and linking with the respective library. This library offers the user a multitude of options for balancing compiler-independence with performance as required for each individual project, and uses extensive template meta-programming techniques for optimization wherever possible.

Depending on the options chosen by the developer, "arrays.h"-based code can run at close-to or even effectively native speeds when working with F90 dynamically allocated arrays in C++, with the added benefit that it also allows for choice between Fortran array semantics: column-major ordering vs. row-major ordering, subscripting operators etc. Thanks to the adaptability of C++'s template mechanism (allowing generic and generative programming), no "legwork" is required in C++, and given a sufficient set of pre-written glue code, it should be possible to avoid any extra work in Fortran as well.

7.9 Outline of future work

Despite all of the work that has been done, future work can be thought of for each of the answers to our three questions.

With regard to FT-MPI, we have already mentioned a number of issues:

- a proper solution to the pre- and post-synchronization problem needs to be researched

- the issue of the "recovery-wait syndrome" needs to be researched – we suggest adding a proper set of FT-MPI specific non-blocking calls to solve the problem
- the lack of control over the behavior of REBUILD mode should be further investigated for solutions
- another item of interest would be the ability to grow the number of processes in a job after startup – we suggest investigating the possibility of using MPI-2's intercommunicator-related features for this purpose

With regard to our cookbook:

- each of the above research items will have repercussions for the "recipe" as well
- for more complex tasks than our example, it would probably be interesting to investigate more complex recovery techniques "like made-to-order" in-memory checkpointing and the possibility to wrap these in libraries
- recipe extensions for more "tightly-coupled" processes (e.g. matrix multiplication) should also be further elaborated upon

And finally, with regard to our *arrays.h* library:

- some automization for creating any necessary Fortran "brick-and-mortar" code would be usefull, to compensate for Fortran's lack of generic coding mechanisms
- it might be usefull to investigate the possibility of adding Blitz++ [81] like structures to *arrays.h* to make it more comprehensive and add even more performance-enhancing features to C++ to deal with Fortran-exchangeable dynamic memory
- some extra template-metaprogramming techniques could probably be used to simplify the current implementation – it could be usefull to have a look at the Boost C++ [85] meta-programming library for usefull techniques

Each of these items justifies further research, but would have added simply too broad a scope (and time) to properly deal with in the context of this work.

Appendices

APPENDIX A

Nederlandse samenvatting

A.1 Inleiding

Over de loop van de afgelopen decennia zijn computers gestaag krachtiger geworden. In 1965 deed Gordon Moore, mede-stichter van Intel Corporation, de voorspelling dat technologische vooruitgang er toe zou leiden dat ruwweg elke twee jaar het aantal transistors op een enkele IC (Eng.: "Integrated Circuit" - een geïntegreerd circuit van schakelingen op een enkele chip) verdubbeld zou worden, met een grosso-modo evenredige stijging in performantie tot gevolg. (Zowel in termen van rekenkracht als geheugen.) Deze prognose blijkt - gegeven een aantal kleine bijstellingen - ook grotendeels uitgekomen, en staat onder informatici dan ook informeel bekend als "Moore 's law" (Engels voor: de "wet" van Moore - die dus eigenlijk meer een soort van vuistregel is, die origineel het leven zag als niet meer dan een boude extrapolatie op basis van empirische vaststelling).

En toch: ondanks de spectaculaire vooruitgang in de ontwikkeling van de chips blijkt dat er vandaag, anno 2009, onder wetenschappers nog altijd een tekort aan rekenkracht bestaat. De problemen die de wetenschap probeert aan te pakken groeien, simpelweg, sneller in complexiteit dan wat de IC-ontwikkelaars in dezelfde tijdsspanne aan rekenkracht kunnen bij-creëren om ze op te lossen.

Opvallend is dat het tekort aan rekenkracht zich niet alleen beperkt tot groepen die zich bezig houden met de "grote" problemen. Zelfs bij kleinere onderzoeksgroepen stellen er zich reële problemen op dit vlak. Om een zeer concreet voorbeeld te geven: in onze eigen onderzoeksgroep, CoMP, wordt er gewerkt met software die al gauw 24 uur rekent op problemen van een relatief lage orde wanneer we proberen de berekeningen te doen op een enkele reken-eenheid/kern. Dit geval, de situatie waar kleinere research-groepen zich in vinden, is dan ook degene waarop wij ons in de rest van deze tekst zullen concentreren. Vanuit deze optiek zullen we wat meer uitgebreid

onderzoeken wat de mogelijkheden zijn die we tot onze beschikking hebben, zowel op het gebied van hard- als software...

Natuurlijk blijft de computer-wetenschap niet bij de pakken zitten, en wordt er constant gezocht naar methodes om tóch de gewenste rekenkracht te verwezenlijken. Op dit moment bestaat de meest populaire oplossing voor het rekenkracht-tekort uit het parallelliseren van die berekeningen die te zwaar zijn om binnen een realistische tijd uitgevoerd te worden, op een enkele reken-eenheid. Op dit gebied is er de afgelopen jaren veel vooruitgang gemaakt, en de komst van MPI - een standaard-specificatie voor parallelle programmatie op basis van message passing - heeft deze ontwikkeling nog een stevige duw in de rug gegeven. MPI is onder wetenschappers vandaag zowat de de-facto standaard voor parallelle programmatie, en laat wetenschappers wereldwijd toe om code te delen en uit te wisselen zonder angst voor compatibiliteitsproblemen, en geniet bij CoMP dan ook een zekere voorkeur.

In deze thesis zullen wij ons concentreren op een aantal aspecten van deze constante zoektocht naar rekenkracht, specifiek in de context van kleinere onderzoeksgroepen met beperkte financiële middelen. We zullen zien dat dit een aantal complicaties met zich meebrengt in de keuze van hardware. En deze beperkingen in mogelijkheden van hardware brengt met zich een aantal stringente voorwaarden betreft software. We zullen zien dat vooral fout-tolerantie in de software cruciaal wordt, en dit is dan ook een van de belangrijkste elementen waar wij on in deze tekst op zullen richten.

In dit kader is er ook nog een tweede element dat we nader onder de loep willen nemen... Zeer veel onderzoeksgroepen, inclusief CoMP, werken zeer frequent op basis van oudere software - in het Engels vaak omschreven als “legacy-software”, m.a.w. software die werd “over-geërfd” van voorgaande research, in de wetenschappelijke wereld meestal geschreven in Fortran. Alle parallelle programma’s dienen specifiek geschreven te worden om gebruik te maken van een parallelle reken-omgeving, en dit is met deze legacy-software zo goed als nooit het geval. Deze software dient dan ook herwerkt (Eng: “refactoring”) te worden voor parallellisatie - vaak een zeer ingrijpend proces. Bovendien zijn de oudere programmeertalen en -omgevingen vaak niet de meest optimale om op een praktische manier refactoring in toe-te passen.

A.2 Parallelle reken-platformen: kost versus betrouwbaarheid

Weliswaar zijn er de afgelopen jaren goede vorderingen gemaakt in de ontwikkeling van hardware voor parallelle rekentechnieken: grote hardware-platformen die een zeer groot aantal reken-eenheden verzamelen binnen één structuur, zoals de IBM BlueGene, zijn een realiteit - ze bieden een zeer grote parallelle rekencapaciteit in combinatie met een aanvaardbaar betrouwbaarheidsniveau. Spijtig genoeg ligt het prijsniveau voor dit soort hardware hoog - zéér hoog... prijzen die niet haalbaar zijn voor kleinere, minder vermogende onderzoeksgroepen. Voor deze groepen zijn er gelukkig echter andere mogelijkheden.

Enerzijds heeft men vaak de beschikking over oudere hardware - niet langer in

gebruik voor professionele doeleinden zoals studentenlabs of burotica en volledig afgeschreven, maar wel nog altijd beschikbaar voor gebruik. Deze individuele machines kunnen door middel van reguliere netwerk-technologie gebundeld worden in zgn. "Beowulf-clusters". Dit zijn op het Linux uitbatingssysteem gebaseerde verzamelingen van machines waarop, via een netwerk, parallelle software en de nodige data voor de berekening van de deelproblemen wordt aangebracht. Vervolgens wordt elk van de computers in de cluster gebruikt als parallelle reken-eenheid totdat alle deelproblemen zijn opgelost.

Via clustering is het ook mogelijk om dit type van reken-capaciteit binnen verschillende onderzoeksgroepen, mogelijkerwijs zelfs gespreid over meerdere wetenschappelijke instituten, te bundelen in één parallel hardware-platform.

De Beowulf-aanpak kan ook gebruikt worden op niet-afgeschreven hardware: veel computers in research-instituten worden slechts een gedeelte van de dag gebruikt, d.w.z vooral tijdens de werk- / studie-uren. Daarbuiten blijven zij ongebruikt. Deze machines kunnen op tijdelijke basis worden ondergebracht in een parallel platform op gelijkaardige wijze aan de vorige aanpak.

Ten laatste kan er nog verder worden gegaan in het recupereren van ongebruikte rekentijd: projecten zoals Seti@home gebruiken ongebruikte rekentijd op een computer tijdens de reguliere gebruikstijd, d.m.v. een programmaatje dat monitort wanneer een computer niet in gebruik is en tijdens deze tijd berekeningen draait. Deze aanpak is niet enkel bruikbaar in de context van globale supercomputers zoals het eerder vermeldde Seti@Home of Distributed.Net, maar kan ook op beperktere schaal worden toegepast in kleine organisaties op de machines die daar beschikbaar zijn.

Al deze oplossingen hebben echter een aantal problemen gemeen. Eén aanzienlijk obstakel is een inherent gebrek aan betrouwbaarheid. Beowulf clusters van ouder materieel lopen meer kans op problemen door de natuurlijke sleet waaraan de apparatuur in de cluster onderhevig is. Machines die slechts gedurende een beperkt aantal uren gebruikt mogen worden verlaten de cluster wanneer de "extra-curriculaire" uren eindigen. En met machines waarop men probeert tijdens de reguliere gebruiksuren nog wat rekentijd te "stelen" wordt dit probleem alleen nog maar erger.

A.3 Probleemstelling

Gegeven het voorgaande kunnen we een aantal algemene stellingen poneren. Ten eerste:

- ook kleine onderzoeksgroepen worden vandaag geconfronteerd met een tekort aan rekenkracht; parallelle rekentechnieken vormen hiervoor de enige realistische oplossing
- gegeven de doelstellingen om zo gemakkelijk mogelijk code te kunnen uitwisselen met andere onderzoeksgroepen, alsook om parallelle code op zo veel mogelijk diverse platformen te kunnen draaien is het voor de hand liggend om voor de MPI standaard te kiezen voor het schrijven van parallelle code

- spijtig genoeg beschikken kleinere groepen ook slechts over gelimiteerde fondsen; gelukkig zijn er een aantal technieken die toelaten om allerlei vormen van ongebruikte rekenkracht aan te wenden voor parallelle berekeningen, maar al deze technieken hebben één ding gemeen: het onderliggend hardware-platform is altijd onderhevig aan een verhoogde fout-gevoeligheid - en dit is een probleem: MPI is namelijk niet fout-tolerant

Ten tweede:

- veel onderzoeksgroepen maken nog altijd royaal gebruik van Fortran legacy code
- naar de toekomst, vooral met refactoring op het oog, is het interessant om deze code te integreren met nieuwe, object-georiënteerde code - C++ lijkt hiervoor de meest geschikte kandidaat
- C++en Fortran integreren is niet altijd een voor de hand liggende zaak - zeker niet wanneer het gaat om Fortran 90 code

Het voorgaande brengt ons er toe om de volgende drie onderzoeksvragen naar voor te schuiven:

1. betreft MPI implementaties: wat zijn op dit moment de opties om tóch fout-tolerantie toe te voegen aan het reken-proces, en wat moet er op dit gebied nog gedaan worden om van MPI een geheel valide optie te maken voor parallelle berekeningen op fout-gevoeligere parallelle hardware?
2. betreft MPI-gebaseerde software: gegeven een MPI implementatie met ondersteuning voor fout-tolerantie - welke stappen dienen er ondernomen te worden om deze software fout-tolerant te maken; is het mogelijk om een relatief algemeen "receptenboek" te schrijven voor het fout-tolerant maken van bestaande MPI-gebaseerde software, en hoe zou dit er uit zien?
3. betreft Fortran legacy-code: is het mogelijk om op een gebruikersvriendelijke wijze Fortran - specifiek Fortran90 - code te integreren met C++ code, en wat dient hiervoor te worden gedaan?

Het is de zoektocht naar antwoorden op deze drie vragen waar deze thesis zich op concentreert. Over het verloop van de komende sectie proberen voor elk van de vragen een antwoord en conclusie te formuleren op basis van het door ons uitgevoerde onderzoek.

A.4 Oplossingen en conclusies

A.4.1 MPI-implementaties en fout-tolerantie

FT-MPI

Gezien de aard van de middelen waar wij specifiek in geïnteresseerd zijn - "los" gekoppelde machines, gespreid over meerdere administratieve domeinen, netwerken

en locaties - zijn oplossingen op basis van checkpointing en proces-migratie niet schaalbaar. De enige mogelijkheid die in de praktijk openstaat is om te proberen binnen MPI een gelijkaardige aanpak te gebruiken als bij zijn voorganger PVM: de code zelf zijn fout-tolerantie "op maat" laten ondersteunen. Spijtig genoeg biedt de standaard zelf hiervoor geen afdoende ondersteuning: een uitbreiding van de MPI-specificatie zal nodig zijn. Op dit vlak bestaat er momenteel maar één project dat deze richting ernstig onderzoekt: FT-MPI.

FT-MPI is een project van het Innovative Computing Laboratory (ICL, Engels voor "centrum voor innovatieve berekeningstechnieken") aan de Universiteit van Tennessee, Knoxville (UTK). FT-MPI is, net zoals MPI zelf, geen software maar een specificatie - een uitbreiding op de originele MPI specificatie zelf. De uitbreiding omvat een aantal elementen die tegemoet komen aan de hiervoor gestelde punten. Een essentieel basis-standpunt van FT-MPI is het bieden van een maximale garantie op compatibiliteit - reguliere MPI-programma's zouden zonder probleem moeten werken met FT-MPI, en (in mindere mate) vice-versa: FT-MPI gebaseerde programma's zouden zo veel mogelijk moeten kunnen werken op een niet fout-tolerante MPI implementatie (maar dan zonder fout-tolerant gedrag natuurlijk).

Er zijn echter een aantal problemen met de FT-MPI implementatie die momenteel beschikbaar is van UTK: (1) een aanzienlijk probleem is dat deze implementatie - door haar interne constructie - niet schaalbaar blijkt over verschillende netwerken, en onder deze omstandigheden gevoelig wordt voor flessenhalzen in de communicatie; en (2) er blijkt een stuk van de implementatie te zijn dat niet voorziet in fout-tolerantie binnen de referentie-implementatie zelf.

De kern van het probleem is de name service die de volledige toestand van het FT-MPI systeem bijhoudt: (1) deze moet ten allen tijde beschikbaar zijn - het verlies van deze data betekent het einde van de FT-MPI omgeving, waardoor de name service een single point of failure vormt en (2) alle processen moeten hier ten allen tijde informatie mee kunnen uitwisselen - dit is niet schaalbaar over meerdere netwerken met trage (internet) verbindingen; de name service vormt aldus een flessenhals in het systeem.

Oplossing

In plaats van te voorzien in een nieuwe, aangepaste name service, biedt onze oplossing de gebruiker de mogelijkheid om de bestaande name service te vervangen door een andere name service software naar keuze. Er bestaat namelijk reeds uitgebreid werk in deze context, en er is een hele reeks beschikbare software - vrije en commerciële - die een ruim gamma van mogelijkheden qua fout-tolerantie aanbieden zoals persistente opslag, redundante backup-processen etc. Op deze wijze is er een oplossing voor het single point of failure.

Dit wordt verwezenlijkt door de uitwisseling van data tussen reken-eenheden en de FT-MPI name service op te vangen aan de rand van elk administratief domein - waar dit qua performantie het meest interessant is en waar meestal ook H2O zich zal bevinden. Dit gebeurt via een zogenaamd proxy-proces (één per administratief domein) - een voorziening die naar de individuele reken-eenheden toe optreedt als plaatsvervanger (Eng.: "proxy") van de reguliere FT-MPI name service. In werke-

lijkheid worden van daaruit de oproepen gebundeld en doorgestuurd naar de echte name service - de nieuwe, gekozen door de gebruiker zelf. Hiertoe wordt gebruik gemaakt van een Java-technologie genaamd JNDI, een bibliotheek die communicatie met allerlei diverse name (en directory) services ondersteund op basis van een universele interface. Op deze manier kan er vrij gewisseld worden van name service zonder code te moeten veranderen in de proxy. Dankzij H2O integreert dit geheel vloeiend en schaalbaar in de verschillende administratieve domeinen, en wordt platform-onafhankelijkheid gegarandeerd.

Voordelen van deze aanpak is ook dat het bundelen van alle communicatie in de proxy, en de plaatsing van de proxy processen in elk domein laat toe om de communicatie met de echte name service te stroomlijnen, en om repetitieve uitwisselingen te optimaliseren - dit zorgt voor een aanzienlijke ontlasting van de name service als flessenhals, waardoor ook dat probleem van de baan is.

Hiertoe werd een nieuwe techniek uitgewerkt, die gebruik maakt van de atomiciteit van de zogenaamde "bind" operatie in name services (Eng. - dit staat voor het toevoegen, of "binden", van een nieuwe naam aan de service). Op basis hiervan werd een zogenaamd "unreliable locking" mechanisme uitgewerkt

- locking - Engels voor "afsluiten", d.w.z. het afsluiten van de name service voor aanpassingen door slechts één partij of actor, hetgeen atomiciteit garandeert zoals in de originele name service
- unreliable - Engels voor "onbetrouwbaar", omdat we werken via een onbetrouwbaar medium: een netwerk; dit kan op eenderwelk moment onderbroken worden, waardoor een afgesloten name service niet weer ontsloten zou kunnen worden

Om de onbetrouwbaarheid van het netwerk te overleven moeten er twee dingen gegarandeerd worden.

Ten eerste moet ervoor gezorgd worden dat half-afgewerkte samengestelde operaties niet zichtbaar achterblijven in de name service na het optreden van een netwerk- of andere fout. Dit wordt gegarandeerd op basis van gekende technieken zoals copy-on-write, shadow paging en garbage collection. Ten tweede moet er voor worden gezorgd dat een afgesloten name service, die door een fout m.b.t. de huidige actor niet meer ontsloten wordt, na verloop van tijd automatisch ontsloten wordt om zo terug toegankelijk gemaakt te worden voor andere partijen. Dit wordt gerealiseerd door het combineren van timers met tijds-aanduidingen.

Conclusie

Op basis van dit werk kunnen we stellen dat we een valide oplossing hebben voor ons probleem wat betreft de zoektocht naar een mogelijkheid voor het toevoegen van fout-tolerantie aan MPI: FT-MPI, gebruik makende van de referentie-implementatie van UTK, uitgebreid met H2O en de gezamenlijk door DCL en CoMP uitgewerkte oplossing. In de volgende sectie zullen we zien dat er nog een aantal problemen zijn met de FT-MPI specificatie zelf, maar dat deze momenteel omzeild kunnen worden. We zullen ook suggesties bieden om deze tekortkomingen zo elegant mogelijk op te lossen.

A.4.2 FT-MPI codering

Inleiding

Het nadeel van de FT-MPI aanpak is dat we bestaande MPI programma's zullen moeten aanpassen, alleszins toch als we willen dat fout-tolerantie beschikbaar is. Het voordeel van deze aanpak is echter dat we fout-tolerantie op maat kunnen voorzien, die zo weinig mogelijk rekenkracht en andere middelen verbruikt en die zo snel mogelijk de parallelle taak terug tot een werkbare staat brengt na het optreden van een fout. In deze sectie zullen we een overzicht geven van het onderzoekswerk dat we gedaan hebben met betrekking tot de stappen die genomen moeten worden om van een bestaand MPI-gebaseerd programma een fout-tolerant FT-MPI programma te maken. Dit werk zal ook licht werpen op een aantal eigenschappen van de FT-MPI specificatie zelf.

Oplossing

Over het verloop van ons onderzoek hebben we een "kookboek" opgesteld van zes stappen, waarin een bestaand MPI programma wordt omgezet naar een FT-MPI programma. Bovendien voegen we hier nog een "recept" van zes volgende stappen bij om een oplossing aan te reiken voor het hierboven omschreven "wacht op synchronisatie"-probleem bij de herstel-procedure.

1. Analyseer de verschillende toestanden waar de software door loopt, maak deze expliciet en breng de overgangen tussen de verschillende toestanden in kaart.

   In FT-MPI kan elke MPI-communicatie leiden tot (1) een succesvolle operatie, (2) een herstelbare fout of (3) een niet-herstelbare fout. Aldus worden er drie types van exclusieve toestand geïntroduceerd in de code: geen fout, herstelbare fout ontdekt of niet herstelbare fout ontdekt. Zolang het programma zich in de eerste toestand bevindt dienen er geen speciale maatregelen genomen te worden. Wanneer een herstelbare fout ontdekt werd dient het programma de nodige maatregelen te nemen om de toestand zo efficiënt mogelijk terug te herstellen tot "geen fout". Bij een niet-herstelbare fout dient het programma afgesloten te worden. Deze toestandsveranderingen worden afgebakend door MPI-communicatie. Aldus is het mogelijk om het parallel programma te characteriseren als een reeks toestands-transities, van van MPI-oproep tot MPI-oproep, met eventueel nog een aantal logische stappen er tussen voor een beter begrip of efficiëntere werking.

2. Voeg een post-synchronisatie stap toe.

   Vooraleer een proces wordt afgesloten is het noodzakelijk dat alle processen synchroniseren. Dit is nodig omdat tijdens het herstellen van een fout alle processen zich ook moeten synchroniseren. Wanneer er een fout zou optreden op een moment dat al één of meerdere processen zijn afgesloten, zou deze fout dus niet hersteld kunnen worden. Hier komen we later nog op terug.

3. Gebruik virtuele proces-nummers.

Door het voorkomen van een fout kan het zijn dat de volgorde - en dus ook de nummering - van de processen in een parallelle taak verandert. Daardoor is het beter om in de code zelf geen rechtstreekse proces-nummers te gebruiken. Het aanpassen van de volgorde komt dan eenvoudigweg neer op het aanpassen van de mapping van echte nummers op virtuele nummers, zonder dat de proceslogica zelf moet worden aangepast.

4. Voeg "ABORT" en "RECOVERY" toestanden toe aan de bestaande toestandstransities bij elke MPI-communicatie.

   In ABORT moeten alle processen gestopt worden. In de RECOVERY-toestand moeten alle processen terug in een toestand gebracht worden die zowel intern (intra-proces toestand) als in het geheel van de taak (inter-process toestand) consistent is. Hiervoor maken we gebruik van de opdeling in toestanden uit stap 1. Elk proces dient individueel naar één van die toestanden terug te keren, en de verzameling van toestanden moet een consistente super-toestand vormen - hetgeen ons brengt tot de volgende stap...

5. Breng de evolutie van den inter-proces toestand in kaart.

   Maak een overzicht van welke verzamelingen van toestanden - over de verschillende processen heen - consistent zijn (m.a.w. leiden tot een valide oplossing). Na RECOVERY moet de gehele parallelle taak zich terug bevinden in één van deze toestanden.

6. Bij het uitschrijven van de RECOVERY-procedure:

   (a) let er op dat de procedure zelf fout-tolerant is

   (b) dit houdt in dat het mogelijkerwijs nodig is om voor de RECOVERY-procedure zelf ook een aantal verschillende toestanden te voorzien waartussen getransitioneerd wordt - inclusief een "RECOVERY-RECOVERY" procedure

   (c) over het algemeen wordt dit probleem het gemakkelijkst opgelost door de RECOVERY-procedure te herstarten (dit is wat we zelf hebben gedaan in onze toepassing)

   (d) de gemakkelijkste oplossing voor dit probleem is om überhaupt te vermijden dat er MPI-communicatie voorvalt gedurende de RECOVERY-procedure

   (e) tijdens RECOVERY moet er ook voor gezorgd worden dat slechts een minimum van de reeds berekende data verloren gaat

   (f) de twee bovenstaande doelen zijn niet altijd verenigbaar (ook gedemonstreerd door onze eigen toepassing)

Indien de programmeur deze stappen volgt zou elk op MPI gebaseerd programma omgezet moeten kunnen worden naar een FT-MPI programma. Het enige wat boven-

staande "recept" niet doet is de noodzaak wegnemen om alle processen te laten synchroniseren bij RECOVERY, waarbij enkel naar deze toestand getransitioneerd kan worden na een MPI-communicatie. Indien deze laatste op zich laat wachten (bvb. doordat een proces bezig is met rekenen) moeten dus ook alle andere processen wachten om verder te gaan met RECOVERY.

Dit kan momenteel alleen opgelost worden door "hybride" te gaan werken, d.w.z. door de message-passing aanpak te combineren met asynchrone (oftewel "multi-threaded") programmatie. Nadeel is de grotere complexiteit en het feit dat multi-threading zo goed als altijd platform-afhankelijk is. Indien de ontwikkelaar dit toch wenst te implementeren kan dit gedaan worden door het volgende "recept" toe te voegen aan het voorgaande.

1. Draai berekeningen in een aparte thread.

   Zorg ervoor dat alle langdurige, niet-MPI gerelateerde operaties worden afgewerkt in een nieuwe thread. Alle MPI-operaties gebeuren dan in de main thread, en worden niet geblokkeerd door actieve berekeningen.

2. Gebruik active polling om de status van de parallelle taak te verifiëren.

   Binnen de main thread gaat het proces, gedurende langdurige berekeningen, op regelmatige basis actief de status van de taak nakijken, zowel als de toestand van de reken-thread. Dit kan gebeuren, respectievelijk door het gebruik van MPI non-blocking calls en bvb. het nakijken van door semaforen beschermde gedeelde variabelen.

3. Wanneer een fout voorvalt wordt dit aldus relatief snel waargenomen

4. De fout kan dusdanig worden opgelost terwijl de berekening in de andere thread gewoon verder loopt.

5. Evalueer na een fout het nut van de nog lopende berekening

   Mogelijkerwijs zijn de resultaten van deze berekening na RECOVERY niet langer relevant voor de parallelle taak. In dit geval kan de parallelle reken-thread worden stilgelegd.

6. Recupereer de resultaten van de reken-thread na voltooiing van de berekening.

   Vergeet niet om dit ook te doen na een eventuele RECOVERY.

Conclusies

Het omzetten van "pure" MPI-gebaseerde software naar FT-MPI is op zich geen voor de hand liggende zaak, en UTK voorziet geen echte documentatie naar dit proces toe. Bovendien ontbreken er in de voorbeeld-programma's van UTK zelf een aantal essentiële maatregelen die genomen zouden moeten worden, meer bepaald de "RECOVERY-RECOVERY" procedure en de post-synchronisatie.

Op basis van ons praktisch werk hebben wij wel kunnen voorzien in een efficiënte methodiek voor het omzetten van MPI naar FT-MPI. Daarenboven hebben we ook een aantal huidige tekortkomingen in de FT-MPI specificatie blootgelegd.

Eén van de eerste is de nood aan een synchronisatie op het einde van een proces - dit om te voorkomen dat processen reeds afsluiten terwijl er nog andere processen draaien waarin een fout kan optreden. Bij een fout moeten alle processen synchroniseren - als er reeds enkele taken voltooid zijn kunnen deze niet deelnemen aan het synchronisatie-proces en kan er dus niet begonnen worden aan RECOVERY. Bovendien kan het zijn dat deze voltooide processen na het afhandelen van een fout terug actief dienen te worden, en dit is na afsluiting van een proces niet langer mogelijk. Soms kan het voor de duidelijkheid ook nuttig zijn om een pre-synchronisatie stap in te voeren.

De meest voor de hand liggende oplossing om dit op te vangen in FT-MPI lijkt te zijn door MPI_Begin en MPI_Finalize synchroniserend te maken. Dit is in zuivere MPI niet het geval. Door deze ingreep hoeft er geen FT-MPI specifieke oproep toegevoegd te worden aan de specificatie, wat een minimum aan modificatie en een maximum aan compatibiliteit met zich meebrengt.

Een andere probleem is de nood voor synchronisatie na een MPI-communicatie voor RECOVERY. Dit houdt zoals vermeld in dat processen voor recovery allemaal dienen te synchroniseren, en dat de processen zich enkel bewust kunnen worden van de fout (en dus de nood voor synchronisatie) via een MPI-oproep. Als de eerstvolgende MPI-oproep in een proces lange tijd op zich laat wachten - bvb. omdat het proces aan het rekenen is - moeten alle andere processen ook wachten voor ze aan RECOVERY kunnen beginnen. In ons eigen praktisch werk lossen we dit op via multi-threading, maar deze aanpak is niet erg elegant.

Een oplossing zou zijn om in de FT-MPI specificatie hiervoor asynchrone (non-blocking) oproepen te voorzien in de stijl van de bestaande non-blocking point-to-point communicatie. Dit zorgt voor een elegantere oplossing in de uiteindelijke code, maar vergt wel het toevoegen van nieuwe functies aan de specificatie.

Ook dekt FT-MPI op dit moment alleen nog maar de functionaliteit in MPI-1. Dit heeft als gevolg dat het niet mogelijk is om nieuwe processen toe te voegen aan de parallelle taak tijdens het reken-proces, bvb. wanneer er nieuwe reken-eenheden beschikbaar komen. Door het uitbreiden van FT-MPI met de MPI-2 “spawn” functionaliteit kan ook dit worden opgevangen, maar dan moet er wel ernstig nagedacht worden op de repercussies van de fout-tolerantie voorwaarde op de andere onderdelen van MPI-2.

Wij zijn van mening dat het toevoegen van de bovenstaande suggesties, of alvast het opnemen ervan in discussie over aanpassingen aan de FT-MPI specificatie, zou leiden tot een noodzakelijk verbetering.

A.4.3 C++ integreren met Fortran 90

Inleiding

Heel wat bestaande wetenschappelijke programmatuur is geschreven in Fortran. Fortran beschikt over een aantal gekende troeven - vooral dan op het gebied van matrix-operaties, maar heeft spijtig genoeg ook te leiden van een aantal ernstige beperkin-

gen - vooral dan op het gebied van dynamisch geheugen-beheer (dat in Fortran 77 gewoon niet ondersteund wordt) en ondersteuning voor refactoring (door het gebrek aan bepaalde eigenschappen vergelijkbaar met wat we terugvinden in, bvb., object-georiënteerde talen zoals C++).

Het dynamisch-geheugen probleem is opgelost in Fortran 90, maar met een prijs: in F77 was het vrij gemakkelijk om oproepen te doen vanuit C/C++ naar Fortran en vice-versa. In F90 is dit niet langer triviaal...

Probleemstelling

De reden hiervoor is dat F90 meta-data dient te bewaren over dynamisch gealloceerd geheugen, om programmeurs toe te laten om dynamische arrays uit te wisselen met functies met een open array in hun argumenten-lijst en om de compiler toe te laten om matrix-achtige operaties te blijven optimaliseren. Het probleem is specifiek dat deze meta-data - die tijdens run-time wordt opgeslagen in een zgn. "dope-vector" - verborgen wordt gehouden van de gebruiker. Bovendien is de structuur van deze meta-data compiler-afhankelijk.

Oplossing

Om de problemen aan te pakken die ontstaan bij een poging om F90 dope-vectors uit te wisselen met C++ hebben we de *arrays.h* bibliotheek ontwikkeld. Deze laat toe om dynamische geheugen-structuren uit te wisselen met F90 door het includeren van een header-file in C++ en linken met de binaire vorm van de bibliotheek. Deze oplossing laat de gebruiker toe om te kiezen uit een schare van opties met betrekking tot de verhouding performantie / compiler-onafhankelijkheid, op basis van de noden van elk project. Om dit te realiseren wordt er extensief gebruik gemaakt van C++ template meta-programmatie technieken voor optimalisatie, overal waar dit mogelijk is.

Conclusie

Afhankelijk van de keuze van de ontwikkelaar kan code op basis van *arrays.h* werken aan een snelheid dichtbij de snelheid van effectieve C-arrays, met de toegevoegde waarde dat de gebruiker kan kiezen om te werken met Fortran-array semantiek - d.w.z.: column-major ordering, elementen accederen op basis van "()" i.p.v. "[]" etc. Dankzij het template mechanisme in C++, dat de mogelijkheid biedt tot generieke en generatieve programmatie, is hiervoor geen extra werk benodigd in C++. Gegeven een afdoende verzameling van vooraf geschreven "glue-code" zou het mogelijk moeten zijn om extra werk in Fortran ook te vermijden (dit hangt o.a. samen met het feit dat in Fortran arrays beperkt worden tot zeven dimensies).

A.5 Besluit

Op basis van onze noden: het ontwikkelen van een betaalbaar MPI-gebaseerd parallel reken-platform voor kleine onderzoeksgroepen met beperkte financiële middelen, hebben we een drie-tal vragen vooropgesteld waarvoor een antwoord nodig is willen

we dit plan realiseren. Over het verloop van de voorgaande sectie hebben we één voor één voorzien in een antwoord op deze onderzoeksvragen. Een overzicht:

1. *Betreft MPI implementaties: wat zijn op dit moment de opties om tóch fout-tolerantie toe te voegen aan het reken-proces, en wat moet er op dit gebied nog gedaan worden om van MPI een geheel valide optie te maken voor parallelle berekeningen op fout-gevoeligere parallelle hardware?*

   De enige realistische oplossing voor het type van hardware-platformen dat wij in beschouwing nemen is FT-MPI van de University of Tennessee, Knoxville (UTK). UTK voorziet zowel in de specificatie voor FT-MPI als een referentie-implementatie ervan. Met beide zijn er een aantal problemen die nog dienden opgelost te worden. Voor de praktische problemen - een flessenhals en single point of failure - ontwikkelden we een oplossing in samenwerking met het Distributed Computing Lab van Emory University in Atlanta - Georgia (Emory-DCL). Voor de problemen met de specificatie in haar huidige vorm presenteren we een aantal voorstellen tot oplossing. Mits een aantal conventies kan er momenteel grosso-modo rond deze beperkingen heen worden gewerkt.

2. *Betreft MPI-gebaseerde software: gegeven een MPI implementatie met ondersteuning voor fout-tolerantie - welke stappen dienen er ondernomen te worden om deze software fout-tolerant te maken; is het mogelijk om een relatief algemeen "receptenboek" te schrijven voor het fout-tolerant maken van bestaande MPI-gebaseerde software, en hoe zou dit er uit zien?*

   Op basis van FT-MPI hebben we een "kookboek" uitgewerkt met twee "recepten". Eén "basis-recept" om pure MPI-gebaseerde code om te zetten in FT-MPI gebaseerde code, en een toegevoegd recept om synchronisatie-problemen op te lossen indien dit gewenst is. Het laatste recept voegt wel aanzienlijke complexiteit toe aan de code, en onder onze voorstellen ter verbetering van de specificatie voorzien we dan ook een voorstel om dit punt aan te pakken.

3. *Betreft Fortran legacy-code: is het mogelijk om op een gebruikersvriendelijke wijze Fortran - specifiek Fortran90 - code te integreren met C++ code, en wat dient hiervoor te worden gedaan?*

   Ja - het is mogelijk om op relatief gebruiksvriendelijke wijze te voorzien in een middel om functie-oproepen te overbruggen tussen C++ en F90. Om dit aan te tonen schreven wij de bibliotheek *arrays.h*, die voorziet in de middelen om arrays en pointers naar arrays (het belangrijkste struikelblok) uit te wisselen tussen de twee talen. Door de gebruiker een ruime waaier aan keuze te laten bij het gebruik van deze bibliotheek kan deze maximaal optimaliseren op maat van zijn eigen noden.

Hiermee zijn een aantal ernstige vragen op de zoektocht naar meer rekenkracht - voor een beperkt budget - uit de weg geruimd, met behoud van opties zoals uitwissel-

baarheid van code, flexibiliteit met het oog op refactoring en vergelijkbare argumenten.

Bibliography

[1] G. E. Moore. Cramming more components into integrated circuits. Electronics Magazine, 19 April 1965

[2] G.E. Moore. No exponential is forever... but can we delay 'forever'. Presented at the International Solid States Circuits Conference (ISSCC), Feb. 10, 2003

[3] T.A. Maier, J.B. White, M. Jarrel, P. Kent, T.C. Schulthess. New insights into high temperature superconductivity from a computational solution of the two-dimensional Hubbard model. In Journal of Physics: Conference Series 16 (2005) pp. 257-268

[4] S.R. Alam, J.S. Vetter, P.K. Agarwal, A. Geist. Performance characterization of molecular dynamics techniques for biomolecular simulations. In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 59-68, 2006

[5] D.I. Lewin. A changing climate for climate modeling. In IEEE Concurrency 7(2): pp. 10-12, Apr-Jun 1999

[6] B.R. Rau and J.A. Fisher. Instruction-level parallel processing: history, overview and perspective. The Journal of Supercomputing, vol. 7(1-2), pp. 9-50, May 1993

[7] T.L. Sterling, J. Salmon, D.J. Becker, D.F. Savarese. How to build a beowulf: a guide to the implementation and application of PC clusters. MIT Press Scientific And Engineering Computation Series, 1999.

[8] N.R. Adiga et al, An overview of the BlueGene/L supercomputer. Proceedings of SC2002 + IBM Research Report RC22570 (W0209-033), Nov 2002

[9] I. Foster, C. Kesselman. The grid: blueprint for a new computing infrastructure. Morgan Kaufmann Publishers. ISBN 1-55860-475-8.

[10] SETI@HOME Project. Online at http://setiathome.ssl.berkeley.edu, Space Sciences Laboratory, University of California Berkeley, USA

[11] J.J. Dongarra and D. Walker. MPI: A standard message passing interface. Supercomputer, 12(1): pp. 56-68 (Jan 1996)

[12] L. Lamport. On programming parallel computers. Proceedings of The Conference on Programming Languages and Compilers for Parallel and Vector Machines, pp. 25-33, 1975

[13] B.L. Chamberlain, S.J. Deitz, L. Snyder. A comparative study of the NAS MG Benchmark across parallel languages and architectures. In Proceedings to the ACM/IEEE SC 2000 Conference, pp. 46, 2000

[14] H. Jin, M. Frumklin, J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and its performance. NAS Technical Report NAS-99-011, Oct 1999

[15] J.C. Cummins, J.A.. Crotinger, S.W. Haney, W.F. Humfrey, S.R. Karmesin, J. V.W. Reynders, S.A. Smith, T.J. Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In Object Oriented Methods for Interoperable Scientific Engineering Computing, 1999, SIAM, ch. 29

[16] S. Saunders, L. Rauchwerger. ARMI: an adaptive, platform-independent communication library. In Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 230-241, 2003

[17] A. Benoit, M. Cole, J. Hillston and S. Gilmore. Flexible skeletal programming with eSKel. In Proceedings to Europar 2005, Sep. 2005.

[18] K.L. Johnson, M.F. Kaashoek and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In Proceedings of the Fifteenth Symposium on Operating Systems Principles. December 1995.

[19] H.J. Ehold, W.N. Gansterer, D.F. Kvanicka, and C.W. Ueberhuber. HPF and numerical libraries. Proceedings to the Fourth International APC Conference Including Special Tracks on Parallel Numerics (ParNum 99) an Parallel Computing in Image Processing, Video Processing and Multimedia, 1999

[20] M. Frumkin, M. Hribar, H. Jin, A. Waheed and J. Yan. A comparison of automatic parallelization tools/compilers on the SGI origin 2000. In Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, pp. 1-22, 1998

[21] A. Waheed and J. Yan. Parallelization of NAS benchmarks for shared memory multiprocessors. NAS Technical Report NAS-98-010, Mar 1998

[22] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. In Parallel Computing vol. 22(11), pp. 1513-1526, 1997

[23] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr. and M.E. Zosel. The High Performance Fortran Handbook. MIT Press, Cambridge, MA, 1993

[24] OpenMP Fortran Application Programming Interface, v. 1. World Wide Web, http://www.OpenMP.org, Oct 1997

[25] OpenMP C and C++ Application Programming Interface, v. 1. World Wide Web, http://www.OpenMP.org, Oct. 1998

[26] T. von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schausser. Active messages: a mechanism for integrated communication and computation. In Proceedings of the 19th International Symposium on Computer Architecture, May 1992.

[27] W. Gropp, E. Lusk, A. Skjellum. Using MPI - Portable parallel programming with the Message Passing Interface (2nd ed.). 1999.

[28] Gary L. Mullen-Schultz. Blue Gene/L: Application Development. Technical Report SG-247179-01, IBM, December 1 2005

[29] K. Mani Chandy and Lesli Lamport. Distributed Snapshots: determining global state of distributed systems. ACM Trans. Comput. Syst., 3(1):63-75, 1985

[30] E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang and David B. Johnson. A survey of rollback-recovery protocols in message passing systems. ACM Comput. Surv., 34(3): 375-408, 2002

[31] Jason Duell, Paul Hargrove and Eric Roman. The design and implementation of Berkeley Lab's linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2003

[32] Michael Litzkow, Todd Tannenbaum, Jim Basney and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report CS-TR-199701346, University of Wisconsin, Madison, 1997

[33] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li. Libckpt: Transparent checkpointing under UNIX. Technical Report, Knoxville, TN, USA, 1994

[34] Hua Zhong and Jason Nieh. CRAK: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.

[35] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. Cluster Computing, 6(3):227-236, 2003

[36] George Bosilca et al. MPICH-V: Toward a scalable fault-tolerant MPI for volatile nodes. In SC'2002 Conference CD, Baltimore, MD, 2002. IEEE/ACM SIGARCH. pap298, LRI

[37] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In IPPS '96: Proceedings of the 10th International Parallel Processing Symposium, pp. 526-531, Washington, DC, USA, 1996. IEEE Computer Society.

[38] Sriram Sankaran, Jeffrey M. Squyres, Brian Barett, Andrew Lumsdaine, Jason Duell, Paul Hargrove and Eric Roman. The LAMP/MPI checkpoint/restart framework: system-initiated checkpointing. International Journal of High-Performance Computing Applications, 19(4):479-493, Winter 2005.

[39] B. Randell. System Structure for software fault-tolerance. IEEE Transactions on software engineering, SE-1(2):220-232, June 1975.

[40] Y. Tamir and C.H. Sï¿œuin. Error recovery in multi-computers using global checkpoints. In Proceedings of the International Conference on Parallel Processing, pp. 32-41, Aug. 1984.

[41] E.N. Elnozahy, D.B. Johnson and W. Zwaenepoel. The performance of consistent checkpointing. In proceedings of the Eleventh Symposium on Reliable Distributed Systems, pp. 39-47, Oct 1992

[42] D.L. Russel. State restoration in systems of processes. In IEEE Transactions on Software Engineering, SE-6(2):183-194, Mar. 1980

[43] R. Strom and S Yemini. Optimistic recovery in distributed systems. ACM Transaction on Computer systems, 3(3), pp. 204-226, Aug. 1985

[44] J.F. Bartlett. A non-stop kernel. In Proceedings of the 8th ACM Symposium on Operating System Principles, pp. 22-29, 1981

[45] A. Borg, W. Blau, W. Graetsch, F. Hermann and W. Oberle. Fault tolerance under UNIX. ACM Transactions on Computing Systems, 7(1)):1-24, Feb. 1989

[46] J.P. Banatre, M. Banatre and G. Muller. Ensuring data security and integrity with a fast, stable storage. In Proceedings of the Fourth Conference on data engineering, pp. 285-293, Feb. 1988

[47] D.B. Johnson. Distributed system fault tolerance using message logging and checkpointing. Ph. D. Thesis, Rice University, Dec. 1989

[48] D.B. Johnson and W. Zwaenepoel. Sender based message logging. In Proceedings of the Seventeenth International Symposium on Fault-tolerant Computing (FTCS-17), pp. 14-19, Jun 1987

[49] T. T-Y. Juang and S. Venkatesan. Crash recovery with little overhead. In Proceedings of the International Conference on Distributed Computing systems, pp. 454-461, May 1991

[50] J. Hursey, J.M. Squyres, A. Lumsdaine. A checkpoint and restart service specification for OpenMPI. In Indiana University Computer Science Tech Report TR635, July 2006

[51] R.L. Graham, S.E. Choi, D.J. Daniel, N.N Desai, R.G. Minnich, C.E. Rasmussen, L.D. Risinger and M.W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. International Journal of Parallel Programming 31 (4): pp. 285-303, August 2003

[52] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. Risinger, M.W. Sukalski and M.A. Taylor. Network fault-tolerance in LA-MPI. In Proceedings of the 10th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2003

[53] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. RWoodallisinger, M.A. Taylor and T.S. Woodall. Architecture of LA-MPI, a network-fault-tolerant MPI. In Proceedings to IPDPS 2004.

[54] C. Partridge, J. Hughes and J. Stone. Performance of checksums and CRCs over real data. Computer Communication Review, v. 25 n. 4 pp. 68-76, 1995

[55] T. Angskun, G.E. Fagg, G. Bosilca, J. Pjesivac-Grbovic and J.J Dongara. Scalable fault-tolerant protocol for parallel runtime environments. In Proceedings of the 13th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Sep 2006

[56] R. Butler, W. Gropp and E.L. Lusk. A scalable process-management environment for parallel programs. In Proceedings of the 7th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 168-175, London, UK, 2000. Springer-Verlag.

[57] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100.000 processors. In Journal of Parallel and Distributed Computing, to be published

[58] G. Geist, J. Kohl, R. Manchel and P. Papadopoulos. New features of PVM 3.4 and beyond. PVM Euro User's Meeting, pp. 1-10, September 1995.

[59] G.E. Fagg, A. Bukovsky and J. Dongarra, "Harness and fault-tolerant MPI", Parallel Computing, vol. 27, num. 11, pp. 1479-1495, October 2001.

[60] G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London and J. Dongarra. Extending the MPI specification for process fault-tolerance on high-performance computing systems. In Proceedings of the International Supercomputer conference (ICS) 2004.

[61] G.E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovski and J.J. Dongarra. Fault tolerant communication library and applications for high-performance computing. LACSI Symposium 2003, Santa-Fe, October 27-29, 2003

[62] Available for free download at http://icl.cs.utk.edu/ftmpi/

[63] J. Hursey, J.M. Squyres, A. Lumsdaine. A checkpoint and restart service specification for OpenMPI. Technical Report TR635, Indiana University, July 2006

[64] Y.M. Wang. Space reclamation for uncoordinated checkpointing in message passing systems. Ph. D. thesis, university of Illinois Urbana-Champaign, Aug 1993.

[65] Z. Tong, R.Y. Kain and W. Tsai. Rollback-recovery in distributed systems using loosely synchronized clocks. In IEEE Transactions on Parallel and Distributed Systems, 3(2): pp. 246-251, Mar. 1992.

[66] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. In IEEE Transactions on Software Engineering, SE 13(1):23-31, Jan 1987.

[67] H. Huang and Y.M. Wang. Why optimistic message logging has not been used in telecommunication systems. In Proceedings of the Twenty Fifth International Symposium on Fault-Tolerant Computing (FTCS-25), pp. 459-463, Jun 1995.

[68] E. Gabriel et. al. OpenMPI: goals, concept, and design of a next generation MPI implementation. In Proceedings of the 11th PVM/MPI user's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2004

[69] A. Geist. Too big for MPI? In Proceedings of the 13th European PVM/MPI User's Group Meeting on Recent Advances In Parallel Virtual Machine and Message Passing Interface, pp. 1, September 2006

[70] V.S. Vasilevsky, A.V. Nestorov, F. Arickx and J. Broeckhove. The algebraic model for scattering in three-s-cluster systems: theoretical background. In Phys. Rev. C36 (2001) 034605:1-16

[71] J. Broeckhove, F. Arickx, W. Vanroose and V. Vasilevsky. The modified j-matrix method for short-range potentials. J. Phys. A: Math. Gen. 37 (2004) 1-13

[72] E. Karrels and E. Lusk. Performance analysis of MPI programs. In Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, pp. 195-200, 1994

[73] O. Zaki, E. Lusk, W. Gropp and D. Swider. Toward Scalable Performance Visualization with Jumpshot. In High Performance Computing Applications, vol. 13, nr. 2, pp. 277-288, 1999

[74] C.E. Rasmussen, K.A. Lindlan, B. Mohr and J. Striegnitz. CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability. In proceedings of the LACSI Symposium 2001, October 15-18, 2001, Sante Fe, New Mexico

[75] The Chasm project - website: http://chasm-interop.sourceforge.net

[76] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. Proceedings of the ACM/IEEE 2000 Conference on Supercomputing, Dallas Convention Center, Dallas Texas USA, 04/11/2000 - 10/11/2001, pp. 49- 49

[77] Bernholdt, et. al. A Component Architecture for High-Performance Computing, POHLL-02 New York, NY. 22/06/2002

[78] The Babel project - website:
https://computation.llnl.gov/casc/components/babel.html

[79] C - Fortran linking:
http://arnholm.org/software/cppf77/cppf77.htm#Section3.5.2

[80] D. Abrahams and A. Gurtovoy. C++ Template Metaprogramming: Concepts, Tools and

[81] T. L. Veldhuizen. Arrays in Blitz++. LNCS: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98) Santa Fe, NM, USA, December 8-11, Springer-Verlag, 1998

[82] The Blitz++ Library: http://www.oonumerics.org/blitz/

[83] T. L. Veldhuizen. Scientific Computing: C++ Versus Fortran - C++ has more than caught up. Dr. Dobb's Journal of Software Tools 11 (vol. 22), pp. 34 36-38 91, nov 1997

[84] T. L. Veldhuizen, "Using C++ template metaprograms," C++ Report, Vol. 7 No. 4 (May 1995), pp. 36-43

[85] The Boost C++ Library: http://www.boost.org/

[86] D. Abrahams, A. Gurtovoy: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, ISBN 0-321-22725-5

[87] A. Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied (C++ In-Depth Series), ISBN-0201704315, Addison-Wesley Professional (February 23, 2001)

[88] N. Karonis, B. Toonen and I Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. In *Journal of Parallel and Distributed Computing (JPDC)*, 63(5), May 2003, pp. 551-563

[89] R. Keller, B. Krammer, M.S. Mueller, M.M. Resch and E. Gabriel. MPI development tools and applications for the grid. In *Workshop on Grid Applications and Programming Tools*, 2003

[90] D. Kurzyniec, T. Wrzosek, D. Drzewiecki and V. Sunderam. Towards self-organising distributed computing frameworks: The H2O approach. In *Parallel Processing Letters*, 13(2), 2003, pp. 273-290

[91] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. In *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 41(5), May 1992, pp.526-531

[92] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier and F. Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *ACM/IEEE SC2003 Conference*, 2003, pp. 25

[93] Y. Chen, K. Li, J.S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. 1997. Available at http://citeseer,ist.psu.edu/chen97clip.html

[94] J. Chin and P.V. Coveney. Towards tractable toolkits for the Grid: a plea for lightweight, usable middleware. Available at http://www.realitygrid.org/lgpaper21.pdf

[95] G. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic and J. Dongarra. Process fault-tolerance: Semantics, design and applications for high-performance computing. In *International Journal for High Performance Applications and Supercomputing.* 2004.

[96] T. Imamura, Y. Tsujita, H. Koide and H. Takemiya. An architecture of Stampi: MPI library on a cluster of parallel computers. In *7th European PVM/MPI Users' Group Meeting*, 2000, pp. 4-18

[97] T. Tyrakowski, V. S. Sunderam, M. Migliardi. Distributed Name Service in Harness. In *Proceedings of the international Conference on Computational Sciences - Part 1(LNCS Vol. 2073)*, 2001, pp. 345-354

[98] G. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic and J. Dongarra. Process fault-tolerance : Sematics, design and applications for high-performance computing. In *International Journal for High Performance Applications and Supercomputing.* 2004.

[99] D. Kurzyniec and V. Sunderam. Combining FT-MPI with H20: Fault-tolerant MPI across administrative boundaries. In *Proceedings of the HCW 2005-14th Heterogeneous Computing Workshop,* (accepted), 2005

[100] D. Dewolfs, D. Kurzyniec, V. Sunderam, J. Broeckhove, T. Dhaene, G. E. Fagg. Applicability of Generic Naming Services and Fault Tolerant Metacomputing with FT-MPI. In *Proceedings of the 12th European Parallel Virtual Machine and Message Passing Interface - Euro PVM/MPI (Springer-Verlag Berlin LNCS 3666),* Sorrento (Naples), Italy, 2005

[101] G. E. Fagg, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, J. Dongarra. Scalable Fault Tolerant MPI: Extending the Recovery Algorithm. In *Proceedings of the 12th European Parallel Virtual Machine and Message Passing Interface - Euro PVM/MPI (Springer-Verlag Berlin LNCS 3666),* Sorrento (Naples), Italy, 2005, pp. 67

[102] M. Migliardi and V. Sunderam. The Harness Metacomputing Framework. In *The Ninth SIAM Conference on Parallel Processing for Scientific Computing, S. Antonio*, 1999

[103] I. Foster, H. Kishimoto, A. Savva, Fujitsu (Editors), D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, J. Von Reich. The Open Grid Services Architecture, Version 1.5., 24 July 2005 (http://www.ogf.org/documents/GFD.80.pdf)

[104] R. Eigenmann and M. Voss, editors. OpenMP Shared Memory Parallel Programming. In *Proceedings of the International Workshop on OpenMP Applications and Tools, WOMPAT 2001*, West Lafayette, IN, USA, July 30-31, 2001. (Lecture Notes in Computer Science). Springer 2001.

[105] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: A multiprotocol RMI framework for java. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 140146, Nice, France, April 2003. IEEE Computer Society.

[106] D. Dewolfs, J. Broeckhove, V. Sunderam and G. E. Fagg. FT-MPI, fault-tolerant metacomputing and generic name services: a case study.- In: *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface - Euro PVM/MPI (Springer-Verlag Berlin LNCS 4192),* Bonn, Germany, 2006, pp. 133-140

[107] G. Stuer, V. Sunderam, and J. Broeckhove. Towards OGSA compatibility in alternative metacomputing frameworks. In *Proceedings to the 4th International Conference on Computational Science - ICCS 2004*, Krakow, Poland, June 2004. (Springer-Verlag Berlin LNCS 3036 part I), pp. 51-58