Scientific applications in distributed systems

Kenneth Pedersen <kennep@student.ntnu.no>
Kjetil Pedersen <kjetip@stud.ntnu.no>

23rd November 2001
Abstract

Computerized solutions to numerical problems, commonly used to predict the properties of physical systems, require large amount of processing power. Traditionally, the solution has been large, expensive, special-purpose computers. Since the early 1990's another solution became viable; using the combined power of commodity-hardware personal computers in a distributed system.

However, implementing a distributed system is not easy. One needs to be able to communicate and synchronize simply and efficiently. Many different approaches exists, such as the message-passing libraries MPI, OOMPI and PVM, the distributed shared memory environments OpenMP and Linda, and the object-oriented middleware libraries CORBA, DCOM and Java RMI. Still, implementing a distributed system can be a complex task.

In this report, we try to find out if using a special-purpose framework to implement a distributed system is better than creating a customized solution using a communication library directly, with respect to usability and performance. We have compared MPI, OOMPI and CORBA to a framework designed for creating partial differential equation solvers using the finite element method (FEM). For performance and usability testing, we have implemented a simple problem (surface area computation of a three-dimensional mesh), that still requires most of the elements from FEM solvers. For distributed systems were created, using each of the four approaches: MPI, OOMPI, CORBA and Framework.

Our results indicate that with respect to usability, using the Framework is vastly superior to the other approaches, as it required only a tenth of the code required to implement the system using the other approaches. The framework handled all parallelization aspects transparently, while these has to be handled explicitly by the programmer using the other approaches.

However, the performance of the Framework was worse than that of the MPI and OOMPI versions, with the largest difference between Framework and MPI implementations being about 27%. The Framework performance was comparable to that of the CORBA version. Still, in our opinion, the performance gained is not worth the extra effort required in a customized solution, and we expect the Framework to have better performance in tests with more complex real-world problems.
Contents

Abstract ............................................................... iii
Table of Contents ................................................... vi
Preface ............................................................... vii

1. Introduction ..................................................... 1
   1.1. Motivation ................................................ 1
   1.2. Structure of this report ................................. 1

2. Background ..................................................... 3
   2.1. Distributed Systems .................................... 3
   2.1.1. Examples of distributed systems .................. 3
   2.2. Parallel Algorithms .................................... 5
   2.2.1. Parallel programming paradigms .................. 5
   2.3. Data Distribution ...................................... 8
   2.3.1. Topologies ......................................... 8
   2.3.2. Communication Patterns ........................... 9
   2.4. Parallel Programming Environments ................. 11
   2.4.1. Inter-Process Communication and Synchronization Primitives . 12
   2.4.2. Distributed shared Memory Environments ........ 13
   2.4.3. Distributed Memory Environments ................. 14
   2.5. Parallel Programming in Distributed Memory Environments .... 14
   2.5.1. Message-passing libraries ........................ 14
   2.5.2. Distributed Shared Memory Environments ....... 21
   2.5.3. Object-oriented Middleware ....................... 24
   2.5.4. Other approaches ................................ 26
   2.5.5. Applicability of distributed memory environments .... 27

3. Context .......................................................... 32
   3.1. Finite Element Method .................................. 32
   3.2. Using the FEM in a parallel program ................ 36
   3.3. A framework for solutions of PDEs .................... 36
   3.3.1. Overview .......................................... 37
   3.4. Implementation approaches for PDE solving ........ 40
   3.4.1. MPI ................................................ 40
   3.4.2. OOMPI ............................................ 45
   3.4.3. CORBA ............................................ 48
3.4.4. Framework .................................................. 53
3.5. Research agenda .............................................. 55
  3.5.1. Hypothesis ................................................. 55
  3.5.2. Method .................................................... 55
  3.5.3. Problem description ...................................... 56
  3.5.4. Usability metrics ......................................... 58
  3.5.5. Performance metrics ...................................... 59
  3.5.6. Planning .................................................. 61

4. Implementation .................................................. 62
  4.1. Common structure and shared elements .................... 62
    4.1.1. Shared elements ....................................... 64
  4.2. MPI .......................................................... 64
  4.3. OOMPI ........................................................ 65
  4.4. CORBA ........................................................ 65
  4.5. Framework .................................................. 66

5. Test results ................................................... 67
  5.1. Usability .................................................... 67
  5.2. Performance ................................................ 67
    5.2.1. About the Cluster ...................................... 67
    5.2.2. About the Software .................................... 67
    5.2.3. Test files ............................................... 67
    5.2.4. Results ................................................ 68

6. Evaluation .................................................... 72
  6.1. Evaluation .................................................. 72
    6.1.1. Usability ............................................... 72
    6.1.2. Performance ............................................ 73
  6.2. Validation .................................................. 74
  6.3. Suggested improvements .................................... 75

7. Conclusion ................................................... 76
  7.1. Conclusion .................................................. 76
  7.2. Further work ............................................... 76

Appendix .......................................................... 80

A. Results ........................................................ 80
  A.1. Summary of results ......................................... 80
  A.2. Detailed results ........................................... 80
B. Sample code

B.1. GetSubDomain(): Prepare data for sending .......................... 86
B.2. triangle_area(): Calculate triangle area ............................. 88
B.3. calculate_area(): Calculate total area of the mesh .................. 89

C. Testing scripts and commands .......................... 90
C.1. Line count ................................................................. 90
C.2. Testing script ............................................................. 90

Glossary ................................................................. 92

Bibliography ............................................................. 94
Preface

This report is the result of a project performed during the autumn term of 2001 by fifth-year students at the Norwegian University of Science and Technology in Trondheim.

During the course of the project we have learned much about distributed systems and parallel programming, and gained valuable insight into several approaches for interprocess communication. We have discovered that creating parallel programs is difficult, but there exists tools and technologies that can make this job simpler.

We would like to extend our thanks to our mentor, Roxana E. Diaconescu, who provided us with valuable feedback and helpful suggestions. We would also like to thank our colleagues Carl Erik Hauge and Per Traen, with whom we worked during this project.

An informational page for the project, along with a complete copy of the source code created during the project, can be found at http://faeurn.dhs.org/project/.

Trondheim, 23rd November 2001

Kenneth Pedersen                        Kjetil Pedersen
1. Introduction

In the 1980's most people believed faster hardware and more efficient software would be the best way of improving computer performance. Since the early 1990's another idea became interesting; using the combined power of personal computers to gain a cheap and efficient solution. The driving force behind this is cheaper and better computer components that are available for PCs. Making a cluster of computers cooperate like this gives a high-powered solution at a low cost. Such a cluster would need an efficient way of communication so that the computers' resources can be utilized fully[4].

1.1. Motivation

Our perspective is to consider how numerical problems like the partial differential equations can be solved in a distributed environment. Different approaches to creating distributed systems exist, each with different advantages and disadvantages. When determining which approach to use, several factors must be considered. Two important factors are the ease of which a solution can be implemented, and the performance of the resulting solution. One seldom get one without sacrificing the other, i.e. to improve the performance one may need to create a more complex system, and this takes additional time.

There are many situations in which a user might want to choose simplicity before efficiency, but for computation-intensive numerical applications, performance remains important. Is it really possible to get a solution that involves more reusable code and less implementation work, without totally destroying the performance? Such a solution would preferably come in the form of a ready-made library or framework that makes it easier to solve the problem at hand, while having only slightly lower performance than a customized, built-from-scratch solution.

In order to try to answer the above question, we will evaluate a framework for the distributed solution of partial differential equations against customized solutions based on traditional communication libraries.

1.2. Structure of this report

In chapter 1 we give a short introduction to distributed systems and state our motivation for writing this report. Chapter 2 consists of general background information on distributed systems and information on the different parallel programming paradigms, as well as a survey of the different methods of communication used in distributed systems.
In chapter 3 we describe the finite element method (FEM) and describe a method of implementing a FEM solver in a distributed system. We formulate a hypothesis for the performance and usability of different approaches, and describe how we plan to evaluate them. To aid in the evaluation, a small problem is defined. This problem will be solved using four different approaches.

The different implementations solving the problem are described in detail in chapter 4. We also point out the similarities and the difference between the implementations.

The key results of our tests are presented in chapter 5, and are evaluated in chapter 6. In chapter 6 some suggested improvements are also listed.

Finally, in chapter 7 we make the conclusion of our work, with a short summary of the results achieved. Lastly, some suggestions for further work is made.
2. Background

In this chapter we give a brief outline of the fundamentals of distributed systems and parallel programming. First the term “distributed system” is defined and some examples of different types of distributed systems are given. Then different paradigms for constructing parallel algorithms, and the data distribution issues in parallel systems, are discussed. Lastly, a survey of different parallel programming environments, libraries, and languages is conducted.

2.1. Distributed Systems

A distributed system consists of a collection of two or more independent computers working together as an integrated unit. The computers have some means of communicating with each other, i.e. a computer network, and coordinate their actions by passing messages to each other.

That the computers are independent means that architecturally, they are capable of operating as autonomous units. That they work together as an integrated unit means that this set of computers appears as a single computer system to the users of the system. This is known as the single system image and is a major goal in designing distributed systems that are easy to maintain and operate [3].

There exists several definitions of the term “distributed system” in literature. In [1] a distributed system is defined as

“A system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing.”

[2] has another definition of the term:

“A distributed system is a collection of autonomous computers linked by a network with software designed to produce an integrated computing facility.”

2.1.1. Examples of distributed systems

Many types of distributed systems exist, and they vary both in the coupling of the computers themselves, and in the coupling of the computer software.

A loosely coupled computer system consists of many computers located at different locations connected by a low-bandwidth, high-latency, unreliable network, while a tightly
coupled system consists of (relatively) few computers located in close proximity connected by a high-bandwidth, low-latency, reliable network.

A system with tightly-coupled software appears to the users as a single computer, while a system with loosely-coupled software appears as a collection of distinctly separate entities.

A few examples of different distributed systems follow.

**SETI@home** SETI@home\(^1\) is an experiment which aims to use the spare processing cycles of computers connected to the Internet throughout the world to analyze radio telescope data in the Search for Extraterrestrial Intelligence (SETI). SETI@home is a loosely-coupled system with loosely-coupled software, but its raw computing power is impressive; the total computing power of SETI@home is higher than that of the world's most powerful supercomputer, the IBM ASCI White\(^2\).

**Beowulf** A beowulf\(^3\) cluster is a collection of PC's or workstations dedicated to running high-performance computing tasks. The computers in the cluster run a free-software operating system like Linux or FreeBSD, together with special clustering software, and are connected by a private high-speed network.

A beowulf cluster is a medium-coupled system with medium-coupled software. It falls somewhere between massively parallel processors (true parallel computers with multiple processors and possibly shared memory), and a network of workstations. The computers are connected with a fast dedicated network and are dedicated to running cluster jobs, but are still commodity PC's or workstations running a traditional operating system (albeit with special clustering modifications).

---

\(^1\)http://setiathome.ssl.berkeley.edu

\(^2\)According to the SETI FAQ at http://setiathome.ssl.berkeley.edu/faq.html#q1.21

\(^3\)http://www.beowulf.org
**Amoeba** Amoeba\(^4\) is a microkernel-based general-purpose operating system that turns a collection of workstations or single-board computers into a transparent distributed system. The Amoeba software is tightly-coupled, and usually runs on medium-coupled hardware (ie a dedicated network of workstations).

Amoeba accomplishes the goal of the single computer image; to the casual user, an Amoeba system looks like a single old-fashioned time-sharing system, albeit with multiple processors. The operating system and supporting software automatically takes care of location, migration, replication and concurrency transparency (these terms are defined in [3]).

### 2.2. Parallel Algorithms

Ian Foster\(^4\) suggests that the design process of a parallel algorithm should be organized in four separate phases: partitioning, communication, agglomeration, and mapping.

**Partitioning** deals with the decomposition of the computational processes and the data which it operates into several small tasks. Tasks are pieces of work that can be executed sequentially, on a single processor, and are typically implemented as processes or threads on a computer. Domain/data decomposition is the decomposition of the data associated with the problem, and functional decomposition is the decomposition of computation into disjoint tasks. The issues related to decomposing and distributing data will be discussed further in section 2.3.

**Communication** focuses on the flow of information and coordination between the tasks created during the previous phase. There are four popular communication patterns: local/global, structured/unstructured, static/dynamic and synchronous/asynchronous.

**Agglomeration** involves evaluating the the task and communication structure vs. performance requirements and implementation cost. Tasks are grouped into larger tasks if required to improve performance or reduce the cost. Communication may be gathered into a super communication, which will help reduce the communication cost by increasing computation and communication granularity.

**Mapping** means mapping each task to each processor in a way that optimizes the use of system resources. This is done either at run time or before, in the first case the mapping is determined through load-balancing methods.

#### 2.2.1. Parallel programming paradigms

Most parallel applications can be classified into some well defined programming paradigms. A few programming paradigms are used repeatedly to develop parallel programs. Each paradigm is a class of algorithms that have the same control structure.

The choice of what parallel paradigm to use depends much on what type of algorithm the system uses, and what sort of model to use. For example a message passing system like PVM has a dynamic task spawning capability, where the tasks spawn other tasks. This fits very well with the master/slave paradigm. A master task spawns new slave

---

\(^4\)http://www.cs.vu.nl/pub/amoeba
tasks, which spawn their slave tasks etc. This results in a hierarchy of tasks and subtasks with the first task as the master.

The following paradigms are popularly used in parallel programming:

- Task Farming, or Master/Slave
- Single Program Multiple Data (SPMD)
- Data Pipelining
- Divide and Conquer
- Speculative Parallelism

These paradigms are described briefly in the following.

**Figure 2.2:** The basic structure of a master/slave program

**Task Farming (Master/Slave)** This paradigm consists of two classes of processes: one master process and multiple slave processes. The master is responsible for decomposing the problem into small tasks, and hands these tasks to a farm of slave processes. After the slaves have finished processing, the master gathers the partial results and computes the final result.
**Single Program Multiple Data**  In the SPMD paradigm, the same algorithm run concurrently on different pieces of data. This involves splitting the data amongst the available processors, running the algorithm on each piece of data (possibly with some communication between the processes), and assembling the final result in the end.

Many physical problems lend themselves well to this method of processing, and this paradigm may be the most widely used programming paradigm.

**Data Pipelining**  This paradigm is based on a functional decomposition. The tasks of the algorithm that can be run in parallel are identified, and each processor executes a small part of the total algorithm. The processes are organized in a pipeline, with each process handling one step of the algorithm. Communication flows from one step of the algorithm to the next, and the communication may be completely asynchronous.

The simplicity and robustness of this paradigm makes it one of the most popular functional decomposition paradigms.

**Divide and Conquer**  The divide and conquer approach is well known in sequential algorithm development. One solves a particular problem by dividing it into subproblems. These subproblems can then be decomposed into even simpler problem, and so on, until each problem has a simple solution. The results from the subproblems are then combined to give the final result. The collection of problem and subproblems form a tree with the main problem at the root, and the subproblem further down the branches.

This method can also be used in parallel algorithms, with each subproblem being run concurrently on different processors.
Speculative Parallelism  This method is used when it is difficult to achieve parallelism through one of the previous methods. Speculative parallelism involves using some speculation or optimistic execution to facilitate the parallelism. This can include using look-ahead-execution of related activities in case one or more of them needed to be done anyway, or concurrently applying different algorithms on the same problem and selecting the one that gives the final answer first.

Hybrid Models  Sometimes it is necessary to mix elements from the previous paradigms in a hybrid model. One example include using the task control of the master/slave paradigm with the multiple data/same algorithm aspect of the SPMD paradigm.

2.3. Data Distribution

One of the key issues when developing parallel algorithms is determining how data will be distributed amongst the tasks. This depends on the chosen programming paradigm and the data to be processed as well as the nature of the application.

There are two main goals to consider when determining the data distribution in a parallel program. The first is to distribute the computation load as evenly as possible amongst the available processors, thereby enhancing performance. The second is to reduce the amount of data that need to be communicated in the application, in order to eliminate communication bottlenecks and improve scalability. These goals are often conflicting, and one needs to find a reasonable compromise.

2.3.1. Topologies

When deciding how to divide data between tasks, the inherent topology of the data must be considered. Data topology describes the geometric shape of the system’s data flow.
graph. This is determined by the data input, and can be either regular or irregular.

A regular topology is defined in terms of some sort of regular graph structure, such as a regular grid; an irregular topology isn’t. It is possible to analyze a regular topology reasonably, while even discovering the topology of irregular data can be a challenge. With regular data one often knows the structure of the data at compile time, and one can adjust the algorithm to fit. With irregular data one doesn’t know the structure, and the algorithm needs to be more general.

Developing parallel programs that work with irregular data is more challenging than developing programs that only work with regular data, because the structure of the data is not known in advance, and therefore one has to analyze the data before one can determine how to split the input data amongst the nodes. Different algorithms for splitting data exists, one example is the family of partitioning algorithms implemented by the METIS library [15].

2.3.2. Communication Patterns

The different tasks of a parallel application obviously needs to exchange information in some way, and when deciding how, it can be useful to use already defined communication patterns as a starting point. A communication pattern is a “blueprint” for communication that specifies how many senders and receivers participate in the communication, and what their roles are. Often, a set of communication patterns is implemented in a particular parallel programming environment, so the set of patterns to choose from is restricted to what the environment provides.

One class of communication patterns is the class of collective communication patterns. Collective communication is defined as a communication pattern involving a group of processes. Collective communication is notorious for its demands on network bandwidth and its consequent impact on performance.

The commonly used collective communication patterns are broadcast, scatter, gather, all-to-all broadcast, and all-to-all personalized exchange.

• Broadcast communication involves a single sender and multiple receivers. The same message block is transmitted to a number of receivers. Only one message is
Figure 2.7: This figure illustrates the difference between the collective communication patterns. Note that the number of transmitted network messages is much higher for all-to-all personalized exchange than for broadcast communication.
transmitted in a single communication. (This assumes that the nodes are on the same subnet on a bus network, such as Ethernet. If this condition is not true, one message per receiving node may be required.)

- Scatter communication involves a single sender and multiple receivers. Data is divided and distributed amongst a number of tasks. The total number of transmitted messages in a single communication is equal to the number of receivers.

- Gather communication involves multiple senders and a single receiver. Partial data is gathered from multiple tasks and reassembled into a coherent data set. The total number of transmitted messages in a single communication is equal to the number of senders.

- All-to-all broadcast involves that every task communicates a message block to every other task in the system. The total number of transmitted messages in a single communication is \( n \), where \( n \) is the number of tasks.

- All-to-all-personalized exchange, or complete exchange, involves that every task communicates a distinct message block to every other task in the system. The total number of transmitted messages in a single communication is \( n \cdot (n - 1) \), where \( n \) is the number of tasks. This is generally the most demanding communication pattern.

In contrast, point-to-point or unicast communication involves a single transmitter and a single recipient.

2.4. Parallel Programming Environments

There exists various approaches and models for parallel programming. At the system level, distributed operating systems and virtual shared memory systems support parallelization transparently. At the language level, parallel languages can be used to create explicitly parallel program code, and parallelizing compilers can analyze programs and determine which sections can run concurrently. At the application level, libraries that support parallel programming, such as MPI, PVM and the various object-oriented middleware libraries, can be used to manually implement parallelism, and programming skeletons provide abstractions that makes creating parallel programs easier.

Parallelizing compilers are limited to regular parallel applications, such as computations in loops. There are also difficulties due to the non-uniform memory access time for distributed memory machines, which means that this approach rarely provides adequate speedup.

Many developers feel that using a special parallel language seems like a waste of effort when one can use existing languages like C++ to achieve the same effect[4].

In this section some of the most common programming environments and mechanisms for synchronization and information exchange in parallel systems are discussed. A list of different parallel programming environments and languages can be found at [5].
2.4.1. Inter-Process Communication and Synchronization Primitives

The earliest mechanisms for task communication and coordination in parallel systems were simple primitives, primarily borrowed from operating systems design [4]. These primitives are very low-level, and the programmer has to worry about the many details of inter-process communication and synchronization. For tasks communicating via shared memory, synchronization primitives such as semaphores, critical sections and monitors, were popular. For task communication via distributed memory, the more popular methods were message-passing primitives such as sockets, remote procedure calls, and rendezvous.

Semaphore A semaphore is a synchronization primitive used to control access to a shared resource, and is implemented as a non-negative integer-valued variable that indicates the availability of this resource. There are two operations that can be performed on a semaphore, decrement and increment.

The decrement operation is used by a task to gain access to a shared resource. If the semaphore has a positive value, the task gains access to the resource, and the value of the semaphore is decremented by one. If the semaphore has the value zero, the task blocks until this value becomes nonzero.

The increment operation is used to relinquish access to a resource, and increments the value of the semaphore by one. If other tasks have been waiting on the semaphore, one of them gets a chance to run.

Conditional Critical Regions A conditional critical region is a primitive for mutual exclusion, and consists of a sequence of instructions. Only one of the task can be in a critical region at any one time. Each critical region entry is associated with an entry condition, and only if the entry condition is true can the task compete for entry to the critical region.

Monitors In a monitor a set of resources and operations that can manipulate these resources are defined. The operations defined inside a monitor can only access local variables defined within the monitor and parameters passed to it, and communication between a monitor and the outside world is via parameters to these operations. Only one task may be active in a monitor at the time (similar to critical regions).

Remote Procedure Calls Seen from the programmer’s perspective, remote procedure calls works just like normal (local) procedure calls. The programmer calls a procedure, and gets back a result. However, the actual processing happens on another computer.

When a procedure call is made, a stub procedure on the client (calling computer) takes the parameters, packs them (a process known as marshaling), and sends them over the network. A server stub procedure recieves the parameters, unpacks them, and calls the appropriate server procedure. When processing is complete, the return value is transferred in the opposite direction using the same method.
Figure 2.8.: This figure illustrates the primary difference between the RPC (left) and the rendezvous (right) model. In the rendezvous model, the called task can execute concurrently with the calling task before and after the accept.

Rendezvous  In the rendezvous model, used by the language Ada, two tasks communicate with each other by synchronizing as follows: one task calls an entry on the other one, and the latter responds with an accept for that entry. A task that arrives earlier is required to wait for the task that it needs to communicate with.

Rendezvous is similar to RPC in that one task can call another task, but is more general, since the calling task can do any computation before and after the accept.

2.4.2. Distributed shared Memory Environments

Many special-purpose parallel computers have shared memory, that is, memory that is physically accessible to all processors in the computer. If one processor in a shared memory environment writes a word to a specific location in the shared memory, another processor can expect to read back this word. Communication between and coordination of the different tasks running in a system is via this shared memory.

In a distributed memory environment, memory is local to each processor, and there is no shared memory. Nevertheless it is possible to create an illusion of a shared memory system on a distributed memory system by using special software that translates memory accesses into communication requests. Such an environment is called a virtual shared memory environment, or a distributed shared memory environment.

A distributed shared memory environment frees the programmer from having to explicitly think about data distribution, information moving, etc., but the programmer still has to be aware of the bandwidth and latency problems caused by the moving of information from one processor’s memory to another processor’s memory. Distributed shared memory environments are discussed further in section 2.5.2.
2.4.3. Distributed Memory Environments

In a distributed memory environment, the memory is local to each processor, and the tasks running on different processors need to be coordinated using message passing or similar methods. This architecture is scalable because with each additional CPU added to the system, there is memory local to that CPU, which does not present a bandwidth bottleneck for communication between CPU's and memory. An overview of different distributed memory environments is presented in section 2.5.

2.5. Parallel Programming in Distributed Memory Environments

In this section an overview of common distributed memory programming environments is given. The most popular of these high-level environments are MPI (message passing interface) and PVM (parallel virtual machine). Other approaches like object-oriented programming will also be covered.

2.5.1. Message-passing libraries

Message passing systems allows efficient parallel programs to be written for distributed memory systems. Almost every hardware vendor supports MPI, and the communication libraries provide an interface for C and Fortran. The problem is that most of the task of parallelization is still left to the application programmer. The programmer has to develop a significant amount of software to manage the entire parallelization process.

The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability[6].

PVM

PVM, or Parallel Virtual Machine [4, 7], was the first widely accepted message passing environment that provided portability and interoperability across heterogenous platforms. The environment allows a network of heterogenous computers to be used as a single computational resource. The participating processors may be scalar machines, multiprocessors or special-purpose computers, and this enables application components to run on the architecture most appropriate to the algorithm.

PVM contains functionality for concurrent, sequential, or conditional execution of application components, and supports certain forms of error detection and recovery. It also provides facilities for dynamically reconfiguring the virtual machine, including spawning new tasks, forming groups and performing load balancing.
The PVM system consists of a daemon (pvmd) that is running on all computers in
the virtual machine, a library of function calls, and a PVM console to interactively start,
stop, monitor and control the virtual machine and applications. The first commu-
nication instance between two user processes is routed through the pvmd daemon, which is
located at a well-known communication port. Pvmd designates source and destination
communication ports and subsequent communication can happen directly between the
two processes without interference from pvmd.

The biggest advantage PVM has over competing environments is the high degree of
portability and interoperability. Not only can PVM applications run unchanged on the
many platforms and systems supported by PVM (PVM runs on most Unix derivatives and
Unix clones, as well as on the Windows platform), but different application components
can run on different platforms as part of the same application.

One notable disadvantage is that the performance of PVM is not always as good as
other message passing systems such as MPI\(^5\). This is mainly because efficiency has been
sacrificed for flexibility.

**MPI**

Message Passing Interface is one of the two most popular high-level message passing
systems and was standardized by the MPI Forum. This standardization effort involved
about 60 people from 40 organizations mainly from the United States and Europe. The
preliminary draft proposal, MPI1, was out first in 1992 and later revised in 1993.

The MPI Forum's goal for the Message Passing Interface is to develop a widely used
standard for writing message-passing programs. As such the interface should establish a
practical, portable, efficient, and flexible standard for message passing.

**MPI Procedures, Processes and Transmissions** The following terms are used when
discussing MPI procedures:

- **nonblocking** If the procedure may return before the operation completes, and before the
  user is allowed to re-use resources (such as buffers) specified in the call.

- **blocking** If return from the procedure indicates the user is allowed to re-use resources
  specified in the call. local If completion of the procedure depends only on the local
  executing process. Such an operation does not require communication with another
  user process.

- **non-local** If completion of the operation may require the execution of some MPI pro-
  cedure on another process. Such an operation may require communication occurring
  with another user process.

- **collective** If all processes in a process group need to invoke the procedure.

\(^5\)However, [8, 9] indicate that the performance of PVM implementations is comparable to that of MPI
implementations.
An MPI program consists of autonomous processes, executing their own code, in an MPMD style\(^6\) (multiple programs multiple data). The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

MPI does not specify the execution model for each process. A process can be sequential, or can be multi-threaded, with threads possibly executing concurrently. Care has been taken to make MPI 'thread-safe', by avoiding the use of implicit state. The desired interaction of MPI with threads is that concurrent threads be all allowed to execute MPI calls, and calls be reentrant; a blocking MPI call blocks only the invoking thread, allowing the scheduling of another thread.

MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their binding to physical processors. It is expected that vendors will provide mechanisms to do so either at load time or at run time. Such mechanisms will allow the specification of the initial number of required processes, the code to be executed by each initial process, and the allocation of processes to processors. Also, the current proposal does not provide for dynamic creation or deletion of processes during program execution.

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, timeouts, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures. Error handling can be used to restrict the scope of an unrecoverable error, or design error recovery at the application level.

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as I/O or signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

**Point-to-point communication** Sending and receiving of messages by processes is the basic MPI communication mechanism. One can think of message transfer as consisting of the following three phases:

1. Data is pulled out of the send buffer and a message is assembled.

2. A message is transferred from sender to receiver.

\(^6\)It is of course possible to create MPI programs based on any of the programming paradigms mentioned in section 2.2.1, as these are merely specific cases of the MPMD paradigm. (The SPMD style can be thought of as a degenerate version of the MPMD paradigm in which there is only one program.)
3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous. To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The type matching rules fall into the three categories:

- Communication of typed values. The datatypes of the values are transmitted in the communication, and the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.

- Communication of untyped values. In this case, the transmitted information is treated as a contiguous byte stream, and there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.

- Communication involving packed data.

One of the goals of MPI is to support parallel computations across heterogeneous environments, which may require data conversions.

**Message ordering**

Messages are non-overtaking. If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message if the first one is still pending.

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one
mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use nonblocking communication. A nonblocking send start call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

**Collective Communication**  Collective communication is defined as communication that involves a group of processes. The functions of this type provided by MPI are the following:

- Barrier synchronization across all group members
- Broadcast from one member to all members of a group
- Gather data from all group members to one member
- Scatter data from one member to all members of a group
- A variation on Gather where all members of the group receive the result
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all)
- Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members and a variation where the result is returned to only one member
- A combined reduction and scatter operation

**Groups, Contexts, and Communications**  Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a ‘higher level’ of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies
depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

The key features needed to support the creation of robust parallel libraries are as follows:

- **Safe communication space**, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library

- **Group scope** for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code)

- **Abstract process naming** to allow libraries to describe their communication in terms suitable to their own data structures and algorithms

- The ability to ‘adorn’ a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- Contexts of communication
- Groups of processes
- Virtual topologies
- Attribute caching
- Communicators

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called ‘inter-communication’ and the communicator used is called an ’inter-communicator’.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the ‘local group’ that is, the sender in a send and the receiver in a
receive. The group containing the target process is called the 'remote group' that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint in order to avoid deadlock.

MPI provides a caching facility that allows an application to attach arbitrary pieces of information, called attributes, to communicators. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator
- quickly retrieve that information
- be guaranteed that out-of-date information is never retrieved, even if the communicator is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

**Process Topologies** A topology is an extra, optional attribute that one can give to an intra-communicator. Topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

A process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. This logical process is referred to as virtual topology.

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are proposed in this chapter deal only with machine-independent mapping.
The communication pattern of a set of processes can be represented by a graph. The nodes stand for the processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a 'missing link' in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

**OOMPI**

OOMPI[11] (Object Oriented MPI) is an object oriented class library to the Message Passing Interface. It is implemented as a lightweight layer on top of an existing MPI implementation, and provides all functionality defined by the MPI-1 standard [10]. All standards-complying MPI implementations can usually be used with OOMPI.

When specifying a C++ class library, one moves away from the one-to-one mapping of MPI function to language binding (as exists in the C interface), and runs the risk of adding, losing, or changing MPI specified functionality. It is the goal of the authors that OOMPI remains a syntactically and semantically correct interface to MPI.

OOMPI makes use of C++ features to make the life of the application programmer easier. This includes:

- Using C++ exceptions to handle error conditions. All OOMPI functions can then have specified return values, typically corresponding to MPI “out” parameters.
- Implementing stream-based communications. The standard stream I/O operators (>> and <<) can be used to pass messages to and from communication ports.
- Implementing Communicator Objects that encapsulate the functionality of MPI communicators and forms the basis for all communications.
- Including a wide range of message and data objects, used to send and receive messages of different types, and making it possible for users to create their own message/data object types.
- Implementing copying and assignment operators for all objects.

A report[12] by the team behind OOMPI concludes that the overhead added by OOMPI to the message sending time of MPI is negligible.

**2.5.2. Distributed Shared Memory Environments**

Other programming environments are based on the virtual shared memory model, or the distributed shared memory model. These environments implement a shared-memory
programming model on top of a distributed memory architecture. The shared data space is flat and available with normal write and read operations. A process in a DSM system that wants to fetch some data values does not have to know its location like a process in a message passing system must.

The main advantage distributed shared memory systems have over distributed memory systems is that programmers generally feel more comfortable with the shared memory environment, and usually find it easier to program applications in this environment. This style of programming is more similar to the traditional sequential programming model, and it frees the programmers from having to worry about message passing and related issues.

The main disadvantage of distributed shared memory systems is that they typically have lower performance than a pure distributed memory system, because the communication patterns are not made explicit in the program, and hence the programmer has no control over them and no chance to optimize them.

Two examples of distributed shared memory environments are the Linda language and the OpenMP API.

Linda

Linda[18] is coordination language that relies on an asynchronous and associative communication mechanism based on a shared global space called Tuples Space (TS), a multiset of tuples. Coordination languages are intended to simplify the development of complex software systems by managing specifically the coordination aspects of an application. The term coordination is used to refer to the process of building programs by gluing together their independent active pieces (e.g. a process, a task, a thread, etc.) . The "glue" allow those independent pieces to communicate and to synchronize with each other exactly as they need to.

Linda does not include the notion of component directly; nevertheless, the Tuple Space that it defines can be viewed as a generic connection component that can be used to integrate and coordinate computational components defined in some other programming language.

There are two kind of tuples in Linda: Process tuples are under active evaluation; data tuples are passive. The process tuples (which are all executing simultaneously) exchange data by generating, reading and consuming data tuples. A process tuple that is finished executing turns into a data tuple, indistinguishable from other data tuples. A tuple is a sequence of actual fields(values) and formal fields(variables). There is a unique constraint on a tuple: the first field must be an actual field, called logic name (or tag). The basic interaction mechanism in Linda is pattern-matching. It is used to select a tuple in TS that matches a given tuple t.

Linda has four basic tuple-operations:

out(t): the process executing this operation evaluates t, adds it to TS and continues immediately; this is a non-blocking operation.
in(t): the process executing this operation evaluates t and then looks for a tuple t’ in TS matching t. Once in(t) has found a matching t’, the values of the actuals in t’ are assigned to the corresponding formals in t, and the executing process continues. If no matching t is available when in(t) executes, the executing process suspends until one is, the proceeds as before. If many matching t’s are available, one is chosen arbitrarily.

d(t): is the same as in(t), with actuals assigned to formals as before, except that matched tuple remains in tuple space.

eval(t) is the same as out(t), except that t is evaluated after rather than before it enters tuple space; eval implicitly creates one new process to evaluate each field of t.

Linda is a language that enables dynamic composition of computational components. The communication model that it implements is what makes this language interesting and adapted to flexible couple/uncouple computational software components. The problem with this approach is that it is limited in scope. This coordination language proposes a set of coordination abstractions that realize a particular paradigm for realizing coordination. Coordination problems, however are not always well-suited to a particular paradigm.

OpenMP

OpenMP(multi processing) is an application programming interface (API) that gives shared-memory parallel programers a better interface when developing parallel application from various platforms. The 'Open' in the name 'OpenMP' means simply that it provides an open specification for Multi Processing. The specification consist of a set of compiler directives, library routines, and environment variables.

Motive behind developing the OpenMP specification was the lack of standardization in the industry. Previous work on this include the ANSI X3F3 standardization, but that one was never fully adopted. The lack of one standardization always lead to the use of different directives from different vendors, which is not a good thing. Parallel Computing Forum (PCF), an informal industry group, was behind the X3F3 effort and proposed a draft standard, but never completed it. The difference between with this effort and OpenMP is that OpenMP actually was completed and gained industry-wide support, so its an agreement between vendors and users rather than a formal standard.

Considering MPI, for example, message-passing has become accepted as a portable style of parallel programming, but has limits in effectiveness and scalability. Message-passing systems are difficult to program in general, and do not support incremental parallelization of an existing sequential program. Message-passing was at first meant for client/server applications running across a network, so it includes a costly semantics (including message queuing and selection and the assumption of wholly separate memories) that are often not required by tightly-coupled scientific applications running on modern scalable systems with globally addressable and cache coherent distributed memories.

OpenMP on the other hand can be used for scalable applications, and it both supports nested parallelism and looped-parallelism. The latter is highly usable for applications
that have a lot of coarse loop-level parallelism, especially those that will be never be run on large numbers of processors or for which restructuring the source code is either impractical or disallowed.

OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran for a number of platforms, including Unix and Windows NT. Actually OpenMP is designed for Fortran, C and C++ to support the language that the underlying compiler supports. The OpenMP specification does not introduce any constructs that require specific Fortran 90 or C++ features. OpenMP cannot be supported by compilers that do not support one of Fortran 77, Fortran 90, ANSI 89 C or ANSI C++.

2.5.3. Object-oriented Middleware

The object-oriented programming model is an established state-of-the-art method for sequential programming, but according to various tests it does show promise in parallel programming as well. The lack of acceptance may be because Fortran has been the traditional language for this.

An emerging category of programming environments is object-oriented middleware\(^7\), like CORBA, DCOM and Java RMI. These environments are based on the remote object paradigm, which is similar to the remote procedure call paradigm mentioned in section 2.4.1, but extends it with object-oriented concepts. See figure 2.9.

CORBA, DCOM and Java RMI applications are usually client-server, with several tasks (clients) calling methods of objects located at a central repository (the server), but it is also possible to create peer-to-peer\(^8\) applications.

To the client, remote object method calls works just like calling local object methods, but behind the scenes, an actual method call consists of the following steps:

1. The client requests a reference (a handle or pointer) to a remote object (remote interface in DCOM). This can be done using some kind of directory service, via other remote object, via hardcoded references, or some other means. As a result of this process, a local stub object is created on the client. This object then be used to call the remote object’s methods.

2. The client calls one of the stub object’s methods.

3. The stub object marshals (packs) the method call. This involves packing the method name, object reference and method arguments in a network packet, and sending them to the server via the network.

4. At the other end, a corresponding server stub object unmarshals the method arguments, determines which method to call, and invokes the method call on the appropriate object using the arguments received.

5. The method call executes, just like a local method call, and returns a value.

\(^7\) Simply speaking, middleware is application components that allow processes located in a distributed environment to talk to each other.

\(^8\) Meaning, all nodes are equal.
6. The return value is marshaled by the server stub object and sent back to the client.

7. The client stub object receives the return value and returns it to the calling client.

Figure 2.9: This figure illustrates the remote object calling process. The numbers inside circles refer to the calling sequence described in the text.

This process is very similar to the RPC process described in section 2.4.1\(^9\), but with the addition of object references.

The use of object-oriented middleware environments has not gained widespread acceptance in the scientific computing community. This can be because they are relatively new technologies (the CORBA specification was first released in 1992, and DCOM and Java RMI were released even later), and because the performance of these environments may not be as high as the traditional parallel programming environment. In the case of Java RMI scientific developers might also be apprehensive to using Java, which does suffer from poor performance due to Java programs being run on a virtual (emulated) machine.

CORBA

CORBA [13, 14] (Common Object Request Broker Architecture) is a open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. It uses the Internet Inter-ORB Protocol (IIOP) for remoting objects.

The entire CORBA architecture is based upon Object Request Broker (ORB). The ORB is can be perceived as a central object bus, which all other CORBA objects interacts with whether they are local or remote. The ORB is responsible for finding a CORBA object’s implementation, preparing it to receive requests, communicate requests to it and

\(^9\)Actually, DCOM is implemented on top of DEC-RPC, an RPC implementation.
act as the messenger between the object and client. To communicate with the ORB the object uses an ORB interface or Object Adapter (basic OA or portable OA).

A benefit with using CORBA is that since it's just a set of specifications it can be used on a wide variety of operating systems.

**DCOM**

DCOM [13] (Distributed Common Object Model) is Microsoft's distributed extension to its COM component infrastructure architecture. COM is the "glue" that enables local components on the Microsoft platforms to talk to each other, and with DCOM, these components can also communicate across network borders.

DCOM supports remoting objects by running on a protocol called the Object Remote Procedure Call (ORPC). This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support multiple interfaces each representing a different behaviour of the object.

As specified by COM, a server object’s memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. However, DCOM has only been implemented on Windows platforms with the addition of one or two flavours of Unix.

**Java RMI**

Java RMI [13] (Remote Method Invocation) is Sun's distributed object solution for the Java language. Java RMI relies heavily on Java language features like Java Object Serialization and use the Java Remote Protocol (JRMP) for remoting objects. The Java Object Serialization allows objects to be marshaled (or transmitted) as a stream, and since it's specific to Java both the Java RMI object and the client object have to be written in Java. Each Java RMI server object defines an object that can be used to access a server object outside the Java Virtual Machine (JVM) and on another machine's JVM.

When a client wants to locate a server object for the first time, RMI uses a naming mechanism called RMIRegistry that runs on the server machine and holds information about available server objects. A client acquires an object reference by performing a lookup for a server object reference and invokes methods on the server object as if the Java RMI object resided in the client's address space. Java RMI relies on Java and therefore it can be used on diverse operating system platforms as long as there is a Java Virtual Machine (JVM) implemented on the platform.

**2.5.4. Other approaches**

There has been done some work on integrating PVM and MPI; recently, work has been done at the University of Tennessee by Dongarra, one of the earliest proponents of PVM,
and Flagg, to integrate PVM and MPI functionality in a hybrid PV MPI system[17]. With this they hope to exploit the strengths of these two and in some degree erase the weaknesses.

One programming method, which can be used alone or together with a programming environment, is to create a skeleton, or a framework. This is implemented on top of a message-passing, object-oriented, shared-memory or distributed memory system, and provides increased support for parallel programming. A skeleton hides the specific implementation details from the user and provides a high-level abstraction which supports a particular parallel paradigm. This leads to a style of skeleton-oriented programming (SOP) that allows the user to specify the computation in terms of an interface tailored to the paradigm[4].

2.5.5. Applicability of distributed memory environments

There exists a lot of approaches to parallel programming, as one can see from the various programming environments and libraries we have presented. But one question remain unanswered. Which solution should be used in a particular parallel application? Clearly the different environments and libraries have differing characteristics, and none have better performance in all situations. In this section we will try to outline some of the key differences between PVM, MPI, CORBA, DCOM and Java RMI, and formulate some recommendations based on those facts.

For scientific computing, PVM and MPI are more widely used than object-oriented middleware standards like CORBA, DCOM and Java RMI. This is partly because the PVM and MPI standards initially were more targeted for traditional special-purpose parallel computers, primarily used by the scientific community, while CORBA, DCOM and Java RMI started their life as intercommunication standards for businesses and desktop computers. Consequently, it makes sense to first compare PVM to MPI, and CORBA to DCOM and Java RMI, before the two different approaches (traditional message-passing versus object-oriented middleware) are compared.

Comparison of PVM and MPI

Table 2.1, taken from [16], lists the most important differences between PVM and MPI.

An important distinction between PVM and MPI is that PVM applications can be run in heterogenous environments, and different parts of the same application can run concurrently on different platforms. Different parts of the same application can also be written in different programming languages without it causing any interoperability problems. With MPI, this is not possible. All nodes that run an MPI application must run the same operating system on the same hardware platform using the same MPI implementation, and different parts of the same application cannot be written in different languages, as a rule. This means that if your application will run in a heterogenous environment, with multiple operating systems and diverse hardware and software, PVM is a strong contender.

On the other hand, MPI implementations are optimized for performance, have rich
<table>
<thead>
<tr>
<th>PVM</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual machine concept</td>
<td>no such abstraction</td>
</tr>
<tr>
<td>supports heterogenous NOW* and MPP*</td>
<td>intended primarily for MPP*,</td>
</tr>
<tr>
<td></td>
<td>but NOW* support is growing</td>
</tr>
<tr>
<td>simple message passing</td>
<td>rich messaging support</td>
</tr>
<tr>
<td>communication topology unspecified</td>
<td>supports logical communication topologies</td>
</tr>
<tr>
<td>PVM implementations interoperate</td>
<td>MPI implementations do not interoperate</td>
</tr>
<tr>
<td>across host architecture boundaries</td>
<td>across architectural boundaries</td>
</tr>
<tr>
<td>portability over performance</td>
<td>performance over flexibility</td>
</tr>
<tr>
<td>contains resource management,</td>
<td>primarily concerned with messaging</td>
</tr>
<tr>
<td>load balancing and process control</td>
<td></td>
</tr>
<tr>
<td>primitives</td>
<td></td>
</tr>
<tr>
<td>programs in C, C++, or Fortran</td>
<td>interlanguage communication not supported</td>
</tr>
<tr>
<td>may freely intercommunicate</td>
<td></td>
</tr>
<tr>
<td>robust fault tolerance</td>
<td>more susceptible to faults</td>
</tr>
</tbody>
</table>

Table 2.1.: PVM and MPI key differences.
*NOW: Network of workstations, i.e. a network of normal PC’s.
*MPP: Massively parallel processor, i.e. a special-purpose supercomputer

messaging support and allows the implementor to select from a wide variety of different communication topologies. Therefore, if your application is primarily message-oriented and will run in a homogenous environment, or if performance is of the utmost importance, MPI may be a good choice.

Comparison of CORBA, DCOM and Java RMI

Table 2.2 lists some of the main differences between CORBA, DCOM and Java RMI.

<table>
<thead>
<tr>
<th></th>
<th>CORBA</th>
<th>DCOM</th>
<th>Java RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organisation/Vendor</td>
<td>OMG</td>
<td>Microsoft</td>
<td>Sun</td>
</tr>
<tr>
<td>Type</td>
<td>Formal standard</td>
<td>Vendor specification</td>
<td>Vendor specification</td>
</tr>
<tr>
<td>Impl. vs. spec.</td>
<td>Basic</td>
<td>Most/all</td>
<td>Most/all</td>
</tr>
<tr>
<td>Platforms</td>
<td>Unix,Windows,++</td>
<td>Mostly Windows</td>
<td>Unix, Windows, ++</td>
</tr>
<tr>
<td>Language-indep.</td>
<td>Yes</td>
<td>Yes</td>
<td>No, Java only</td>
</tr>
<tr>
<td>Interface definition</td>
<td>IDL</td>
<td>IDL</td>
<td>Java files</td>
</tr>
<tr>
<td>Error handling</td>
<td>Exceptions</td>
<td>Return value</td>
<td>Exceptions</td>
</tr>
<tr>
<td>Performance</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 2.2: Differences between CORBA, DCOM and Java RMI. The “impl. vs. spec.” category gives an indication on how much of the complete specification is commonly implemented.

A common characteristic between CORBA and DCOM is the use of IDL, while Java RMI extracts the interface definition directly from the objects themselves. IDL (interface
definition language) is a declarative language used to specify the public interface of a component. The actual IDL syntax, however, is different for CORBA and DCOM, so you cannot simply use the same IDL files when porting an application from DCOM to CORBA or vice versa.

The IDLs of CORBA and DCOM refuse to allow any data members of the object to be specified, keeping objects fully encapsulated. The use of an interface specification written in IDL clearly separates interfaces from implementation and is a significant aid to software construction. Another benefit of using IDL is that, after the interface has been expressed in IDL, it is possible to use code generators to produce code that deals with creating, maintaining and releasing connections between distributed components of the system as well as marshalling and de-marshal ling parameters. On the other hand, it is simpler to get started writing a simple application in Java RMI because you don’t have to describe the interface separately, and because you don’t have to learn a new language for specifying interfaces.

One area of major difference between CORBA and DCOM is the error handling. DCOM makes use of return values in a 32-bit integer field called an HRESULT. Using return values to report errors means that the programmer needs to explicitly test for errors after each function call. This gives an somewhat unnatural style of programming, and makes it harder to maintain a single error handler for related errors. On the other hand CORBA and Java error handling is achieved through the use of exceptions. This is vastly superior to using return values, because error handling can be done consistently and more clearly separated from the rest of the program code. Also, since DCOM methods always return an HRESULT value, normal return values must be mapped to output parameters in DCOM. In figure 2.10 a small example is given that illustrates this. The example code shown is pseudocode C++. As can be seen, the DCOM code includes two separate error handlers. This could also be coded as one error handler and two goto statements, but that’s considered bad programming practice in object-oriented languages. Java RMI error handling is similar to the CORBA case.

<table>
<thead>
<tr>
<th>CORBA</th>
<th>DCOM</th>
</tr>
</thead>
</table>
| try {  
  int a = myObject.op1();  
  myObject.op2(a);  
}  
catch (...) {  
  cout << "An error occurred!\n";  
} | int a;  
HRESULT r1 = myObject.op1(&a);  
if(FAILED(r1)) {  
  cout << "An error occurred!\n";  
}  
if(FAILED(myObject.op2(a))) {  
  cout << "An error occurred!\n";  
} |

Figure 2.10: Differences between CORBA and DCOM error handling.

A large difference between CORBA’s and DCOM’s infrastructure is that most of DCOM’s infrastructure is defined and implemented, while a large portion of CORBA’s

---

10 i.e. what modules are available, what interfaces are supported by those modules, what methods are supported by those interfaces, and what arguments are taken by those modules.
has not been implemented yet. And also there is a difference in the integration of the
environment. All of Java RMI has also been implemented.

This difference in the degree of implementation results from Java RMI and DCOM
being controlled by corporation, where implementation and specification goes hand-in-
hand, while CORBA is a formal standard. Only a minimal subset of the full set of
CORBA specifications needs not be implemented in order to comply with the standard.

Similarities between CORBA, DCOM and Java RMI includes the client/server based
programming style and the use of a proxy to hide the lower level details of transmission
protocols. CORBA and DCOM is also similar in that they use IDL to specify the
interface to objects and store information about objects in an information repository
which is external to the implementation files.

While the performance of CORBA and DCOM is comparable, the performance of Java
RMI is poor due to the inherent performance limitations of the Java Virtual Machine.

The question of which object-oriented middleware standard to use in an application
depends largely on what the performance demands are, what platforms the application
will run on, and what languages the application is written in. If the performance needs
to be high, CORBA and DCOM are better choices than Java RMI. If the application is
to be written in a language other than Java, Java RMI is also out of the question. If the
application will run on a non-Microsoft platform, DCOM is a weak candidate.

**Comparison of message-passing libraries to object-oriented middleware**

Table 2.3 lists the most important differences between message-passing libraries and
object-oriented middleware.

<table>
<thead>
<tr>
<th>Programming style</th>
<th>Message-passing libraries</th>
<th>Object-oriented middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming style</td>
<td>Procedural</td>
<td>Object-oriented</td>
</tr>
<tr>
<td>Origin</td>
<td>Scientific computing on MPP’s</td>
<td>Integration of applications in NOW’s</td>
</tr>
<tr>
<td>Performance</td>
<td>High/medium</td>
<td>Medium/low</td>
</tr>
<tr>
<td>Architecture</td>
<td>Multiple</td>
<td>Client/server</td>
</tr>
</tbody>
</table>

Table 2.3: The primary differences between message-passing libraries and object-
oriented middleware.

Message-passing libraries usually use a procedural programming style, and most
commonly contain bindings for Fortran and C. If C++ bindings are offered, they usually are
very similar to the C bindings and typically do not exploit any C++-only language fea-
tures like operator overloading and exception handling. However, class libraries do exist
that make programming message-passing applications in C++ more natural.

On the other hand the various object-oriented middleware standards do make use
of the special features available in object-oriented languages, and borrow much termin-
ology and many of their most important concepts from object-oriented language de-
sign. Object-oriented middleware libraries usually contain bindings for object-oriented
languages like C++, Java and Eiffel. It is possible to e.g. create a CORBA-based ap-
plication in a procedural language like C, but the programming style can be a little unnatural.

Message-passing libraries were designed for scientific applications on massively parallel processors, but have since been ported so they can run on a network of normal workstations or PC’s. The object-oriented middleware libraries originated on network-connected workstations and PC’s and were primarily intended for the integration of different desktop and server applications.

The performance of message-passing libraries tends to be higher than that of object-oriented middleware libraries, especially on the high-speed networks commonly used for cluster computing. [19] suggests that the low performance of different CORBA implementations on high-speed networks is due to presentation layer conversions, data copying, demultiplexing, and memory management overhead. This overhead is often masked on lower-speed networks. One can assume that the same will be the case also for DCOM, Java RMI and other object-oriented middleware solutions, as they all have similar architectures.

Object-oriented middleware libraries lend themselves well to client/server applications. It is possible to create applications based on other architectures, but this is more difficult. Applications using message-passing libraries are more architecture-neutral.

For scientific numerical applications where computing performance is the most important factor and one have high-speed networks available, it seems that using message-passing libraries is still the best approach. For many message-passing libraries object-oriented class libraries are available that make programming message-passing applications in C++ easier.

On the other hand, for loosely-coupled applications running on lower-speed networks, or for applications needing to be integrated with other applications already offering an object-oriented middleware interface, using CORBA or DCOM may be a viable solution. Indeed, many research projects and scientific applications do use CORBA. For a list of companies, organisations and applications using CORBA, see [20].

For all-Java applications where performance is not important, Java RMI may be used.
3. Context

Most phenomenon in nature, whether mechanical, geological or biological, can be described, with the aid of the laws of physics, in terms of algebraic, differential, or integral equations relating various quantities of interest. Solving these equations enables us to predict how these phenomenon will behave in the real world.

An important class of phenomenon is those described by differential equations. Problems that can be expressed in terms of differential equations includes determining the distribution of heat in a material, calculating tensions and deformations in a bridge, building or other construction, simulating the flow of liquid in a pipe, or even simulating the weather. While deriving the governing equations for these problems may not be difficult, solving these equations by exact analytical methods often is unfeasible [21]. In those cases solving the problem by approximative methods of analysis is an alternative. One such frequently-used method is the finite element method.

In this chapter a brief overview of the finite element method is given, and then different approaches for solving partial differential equations (PDE’s) using the finite element method in a distributed system are presented. Although the material presented in this chapter is specialized for solving PDE’s using the finite element method, most of the basic principles of program structure and organization can be generalized to other numerical methods that are based on dividing the problem domain into a discrete set of elements, vertices and faces in a three-dimensional coordinate system.

3.1. Finite Element Method

The finite element method (FEM) is a numerical approach for solving boundary value problems, that is, problems defined by differential equations and their boundary values. The equations are assumed to hold over a certain region, which may be one-, two, or three-dimensional. The basic idea of the finite element method is that instead of seeking approximations that hold over the entire region in question, the region is divided into smaller parts, so-called finite elements, and the approximations is then carried out over each element [22].

This approach is often used when calculating the strength of complicated constructions. The following example illustrates how one can use the method to calculate tensions and deformation in a construction.

In figure 3.1(left) we have a frame, for example two iron bars fastened to the ground with three iron bars welded across it. The arrows from the left represent the wind, and
Figure 3.1: An example on the use of finite element method

to ensure that the construction does not fall apart we need to calculate the momentum, shear- and axial force on each point along the bar. That problem can be expressed and solved with the following differential equation:

\[
\frac{d^2w}{dx^2} = \frac{p(x)}{EI}
\]

where:

\( w = \text{displacement} \)

\( p = \text{outer load} \)

\( EI = \text{the stiffness of the bar} \)

This equation is easily solved, although solving it is not the purpose here. If we consider figure 3.1(right), however, it represents a flat solid block of iron. The goal here is to find the stress in x- and y-direction along with the shear stress. The stress of an object is force divided by area. In this example we cannot use the previous method since that is only usable on object that qualifies as a bar. This object is too flat compared to the width and breadth. The problem we get when not dealing with a simple bar is that bending the object will cause considerable stress is in both x- and y-direction, while the stress in the z-direction of a "thin bar" can be ignored. Because of all this we do not get a simple equation as in the previous example; it will be a lot more complicated. The problem can be described with the following equation:

\[
\frac{\delta^4 F}{\delta x^4} + \frac{\delta^4 F}{\delta x^2 \delta y^2} + \frac{\delta^4 F}{\delta y^4} = 0
\]

where \( F \) are external forces.
This equation can only be solved in simple special cases, otherwise it needs to be solved with the help of the finite element method. The idea here is to divide the block into smaller elements with equal shape and predetermined behaviour. These can be put together in a continuous network, a grid of elements together forming the object.

Figure 3.2: An element of the block

Figure 3.1 shows a typical element. The arrows indicate the degrees of freedom, i.e. the direction the points may move. The nodes represent the corners in the rectangle. This element has 8 degrees of freedom, two on each node. Since the elements has to be continuous and autonomous, it can only describe linear movement between the nodes.

The shape of the elements can be chosen, and is chosen depending on what kind of problem one is dealing with. Choosing the best shape for a particular problem requires some knowledge beforehand. The smaller the elements are the closer it will be to reality, though the size should be chosen depending on the computing power available.

For each element we can use the following equation to describe the relation between forces, stiffness and displacement:

\[ S = kv + S^0 \]

- \( S \) = the forces on the node of the elements
- \( S^0 \) = initial forces or state of the elements
- \( v \) = the displacement of the nodes
- \( k \) = the element’s stiffness matrix

For the system as a whole (elements combined) we can set up a similar equation:

\[ R = K\tau + R^0 \]
\[ R = \text{the forces on the nodes of the system} \]
\[ K = \text{the system’s stiffness matrix} \]
\[ r = \text{the system’s total displacement on the nodes} \]
\[ R^0 = \text{initial forces} \]

The transition from elements to system level can be described through the matrix relation:

\[ v_i = a_i r \]

\( v_i \) normally contains all the displacement in the node for element number \( i \). This way, the relation gives the relation between one element’s displacement and the displacement of the whole system. The matrix usually represents the topology matrix.

A general overview of this method in a serial program can be presented step by step:

1. Start reading the data of the system:
   - geometry (the physical “shape” of the region in question)
   - stiffness
   - load
   - boundary conditions

2. Calculate the stiffness matrix \( K \).

3. Calculate the displacement \( r \) (in theory \( K \) inverted multiplied with \( R \)).

4. With the displacement known we can calculate tensions and deformations.

For problems not related to construction (such as weather simulation or fluid flow simulation), the steps 2-4 are replaced with appropriate steps to calculate what we’re interested in. But no matter what the problem the following basic structure can be used:

1. Start reading the data of the system:
   - geometry
   - additional data appropriate for the problem in question
   - boundary conditions appropriate for the problem in question

2. Iterate over the elements, faces or vertices and do a problem-dependent calculation for each element, face or vertex, using approximations allowed by the finite element method.

3. Collect the computed data and formulate a solution based on this data.
3.2. Using the FEM in a parallel program

In a parallel program, the basic steps for using the finite element method to solve a problem are the same. However, because multiple processors are available, the calculations in step (2) in the previous list can be done in parallel. While dramatic speed increases are possible, parallel programming adds two problems to the solution process:

- How should the available faces, elements and vertices be divided among the available processors in order to balance the computing load as evenly as possible, while at the same time reducing the amount of inter-processor communication necessary?
- How can we ensure that all data in the system are kept in a consistent state? A local calculation on a face, element, or vertex often requires data from another element, and that data is possibly located on another processor. A common requirement is that each element requires access to the data of its neighbors.

With this in mind, a modified list of steps for distributed FEM solvers can be defined as follows:

1. Start by reading in the data of the system
2. Divide the data (elements, faces, vertices, additional data) according to a set of criteria (load-balancing, communication, etc.), using a partition algorithm.
3. For each processor, iterate over the elements, faces or vertices local to this processor, and do a problem-dependent calculation for each element, face or vertex, using approximations allowed by the finite element method. On pre-defined points in the process, exchange data among the processors to ensure that all data in the system is kept in a consistent state (this is called the Update phase).
4. When all processors have finished executing, collect the computed data from each processor and formulate a solution based on this data.

3.3. A framework for solutions of PDEs

Because many of the steps in distributed FEM solvers are the same no matter what problem the solver is used for, [23] proposes a framework for solutions of PDE’s. The framework is based on a hybrid Single Program Multiple Data (SPMD) and master/slave computing model. The input data is a mesh that consists of connected nodes that form an irregular topology, and the data of the PDE is mapped upon the elements in this topology.

Much of the work already done on distributed systems and solving of PDEs is requires a lot of effort from the user who has to explicitly code communication patterns and data distribution. In the framework, this work is done once and for all, and the user only has to provide the algorithm that will do the required calculation on the elements local to a
single processor. The framework automatically reads in and divides data, and the user-provided algorithm is then run on each available processor so that in total, all elements are covered in the calculation. To keep data in a consistent state, the framework requires the user algorithm to call a function that automatically updates all data in the system. Since all communication and data distribution issues are performed automatically, the user is shielded from having to worry about the parallelization aspects of distributed computing, and can more or less program the algorithm as if he or she were implementing it on a single-processor computer.

3.3.1. Overview

Figure 3.3 shows a simplified class diagram for the framework classes\(^1\).

**Master**

The *Master* class includes the functionality required to run the master task. This includes

- reading data from file
- getting environment information, such as the number of nodes in the cluster, and information on the data read
- partitioning the data
- sending the partitioned data to each worker task.

**Worker**

The *Worker* class implements worker task functionality. Each instance of the Worker class contains an instance of the Subdomain class to hold the worker task’s data, and the Worker class delegates much of its work to the Subdomain class.

**Subdomain**

The *Subdomain* class represents a partition of the total data. It contains methods for

- receiving partitioned data from the master task
- getting environment information, such as the number of nodes in the cluster, and the node ID that the current task is running on
- getting information on the data received from the master task, such as the number of elements, faces and vertices
- updating data between processors, and renumbering elements as necessary

\(^1\)This diagram was accurate for the framework when we used it, but the framework is under heavy development, and the details of the classes may change as development progresses.
Figure 3.3: Framework class diagram. White classes are those that must be supplied by the user.
Ghost

Figure 3.4: This figure illustrates ghost elements. Local elements are dark, ghost elements are dotted.

The *Ghost* class represents the “ghost elements” that form the boundary of the subdomain. The ghost elements are outside the subdomain, but information on them is needed in the calculations, so they need to be considered. Ghost elements are illustrated in figure 3.4.

MeshStruct

The *MeshStruct* class represents the structure of the mesh and is used to store mesh data. It includes methods for allocating memory for data, setting and retrieving the number of elements, faces, and vertices, and for storing and retrieving element, face or vertex data.

IntBndry

The *IntBndry* class represents the “internal boundary” between elements local to one node.

UserAlg

The *UserAlg* class represents the algorithm that the framework user will supply. It is a virtual class that is meant to be subclassed in an actual implementation. In figure 3.3, UserAlg is subclassed by the *theAlg* class.
UserData

The *UserData* represents the data that the user algorithm uses. In order to enable the framework to deal with arbitrary data, the UserData class is meant to be subclassed in an actual implementation. In figure 3.3 this is done by the `theData` class.

3.4. Implementation approaches for PDE solving

When creating a parallel program for solving a particular problem, one has the choice between different implementation approaches for managing interprocess communication. At one extreme, one can use low-level communication primitives like TCP/IP sockets to handle IPC. At the other extreme, one can use high-level frameworks specifically tailored for the application domain in question. Occupying a middle ground are communication libraries like MPI, PVM and CORBA. This section discusses different approaches for developing PDE solvers using the finite element method. Four different approaches are covered, ordered from most low-level to most high level: MPI, OOMPI, CORBA, and the framework described in section 3.3.

3.4.1. MPI

Many of those who use the MPI library for message passing use either FORTRAN or C. In this chapter we will discuss MPI commands in general, but when dealing with code we will use C/C++, since C++ is the language used for the other implementation approaches in sects. 3.4.1-3.4.4, and it is therefore easier to see the similarities.

Using the MPI library in C

The functionality of the library is the same for both FORTRAN and C, the differences are purely in the syntax. For C the main MPI header file is called mpi.h and in FORTRAN, it is named mpif.f. Also, C is case-sensitive and all MPI functions have the form MPI_Function, while all constants are named in uppercase letters. Return values are given as integer return values of that function, instead of as an output parameter. (e.g. `rc = MPI_Function()` instead of `MPI_FUNCTION(ierr)`). There are other minor differences in syntax as well.

Environmental Subroutines

These routines are the basic routines needed to handle the MPI environment, like initializing, ranking processors and other routines that do not include methods for communication. As an example, figure 3.5 lists a small program that includes some of these environmental routines. The significant lines are numbered.

1. This line includes the MPI library

2. This line declares the variables needed.
```
#include <mpi.h>  //1

void main(int argc, char **argv) {
  int nprocs, myrank, tag, rc;  //2
  rc = MPI_Initialize(&argc, &argv);  //3
  rc = MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  //4
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  //5
  rc = MPI_Finalize();  //6
}
```

Figure 3.5.: MPI program illustrating the environmental routines

3. **MPI_Init** initializes the MPI environment, this routine must be called once and only once before calling any other MPI routines. rc stores the return value used for error handling.

4. The **MPI_Comm_size** gets the number of processors in the group specified in the parameter `MPI_COMM_WORLD` (where `MPI_COMM_WORLD` means all processors in the cluster).

5. With the **MPI_Comm_rank** function a process can find its own identification number, or rank. The rank ranges from 0 to (size-1) where `size` = number of processors in that communicator group.

6. The **MPI_Finalize** routine terminates MPI processing and no other MPI calls can be made afterwards.

**Point to Point Communication**

MPI also have a set of routines for collective communication, but these are not applicable to problem. We use point to point communication.

**Unidirectional Communication**  Even when a single message is sent from process 0 to process 1, there are several steps involved in the communication. At the sending process, the following events occur one after another.

1. The data is copied to the user buffer by the user.
2. The user calls one of the MPI send subroutines.
3. The system copies the data from the user buffer to the system buffer.
4. The system sends the data from the system buffer to the destination process.

The term *user buffer* means scalar variables or arrays used in the program. The following occurs during the receiving process:
1. The user calls one of the MPI receive subroutines.

2. The system receives the data from the source process and copies it to the system buffer.

3. The system copies the data from the system buffer to the user buffer.

4. The user uses the data in the user buffer.

When you send a message from process 0 to process 1, there are four combinations of MPI subroutines to choose from depending on whether you use a blocking or non-blocking subroutine for sending or receiving data.

The four combinations, with small code examples are listed in the following:

**Blocking send and blocking receive**

```c
if (myrank==0)
  rc = MPI_Send(sendbuf, icount, MPI_REAL8, 1, itag,
                MPI_COMM_WORLD);
else if (myrank==1)
  rc = MPI_Recv(recevbuf, icount, MPI_REAL8, 0, itag,
                MPI_COMM_WORLD, istatus);
```

**Non-blocking send and blocking receive**

```c
if (myrank==0) {
  rc = MPI_Isend(sendbuf, icount, MPI_REAL8, 1, itag,
                  MPI_COMM_WORLD, ireq);
  rc = MPI_Wait(&ireq, &istatus);
} else if (myrank==1) {
  rc = MPI_Recv(recevbuf, icount, MPI_REAL8, 0, itag,
                 MPI_COMM_WORLD, istatus);
}
```

**Blocking send and non-blocking receive**

```c
if (myrank==0) {
  rc = MPI_Ssend(sendbuf, icount, MPI_REAL8, 1, itag,
                 MPI_COMM_WORLD);
} else if (myrank==1) {
  rc = MPI_Irecv(recevbuf, icount, MPI_REAL8, 0, itag,
                 MPI_COMM_WORLD, &ireq);
  rc = MPI_Wait(&ireq, &istatus, &ierr);
}
Non-blocking send and non-blocking receive

if (myrank==0) {
    rc = MPI_Isend(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq);
} else if (myrank==1) {
    rc = MPI_Irecv(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq);
        rc = MPI_Wait(ireq, istatus, ierr);
}

Note that you can place MPI_Wait anywhere after the call of non-blocking subroutine and before reuse of the buffer.

Bidirectional Communication When two processes need to exchange data with each other, you have to be careful about deadlocks. When a deadlock occurs, processes involved in the deadlock will not proceed any further. Deadlocks can take place either due to the incorrect order of send and receive, or due to the limited size of the system buffer.

There are essentially three cases depending on the order of send and receive subroutines called by both processes.

Case 1 Both processes call the send subroutine first, and then receive.

Case 2 Both processes call the receive subroutine first, and then send.

Case 3 One process calls send and receive subroutines in this order, and the other calls in the opposite order.

Case 1: Send first and then receive. Consider the following code:

if (myrank==0) {
    rc = MPI_Send(sendbuf, ...);
    rc = MPI_RecvSend(recvbuf, ...);
} else if (myrank==1) {
    rc = MPI_Send(sendbuf, ...);
    rc = MPI_Recv(recvbuf, ...);
}

Remember that the program returns from MPI_Send when the copy from sendbuf variable to the system buffer has finished. As long as the system buffer is larger than sendbuf, the program ends normally. What if the system buffer is not large enough to hold all the data in sendbuf? Process 1 is supposed to receive data from process 0, and that part of process 0's system buffer that has been received by process 1 can be reused for other data. Then the uncopied data fills up this space. This cycle repeats until all the data in sendbuf of process 0 has been copied to the system buffer. And only at that time, the program returns from MPI_Send. In the previous program example, process 1 does the same thing as process 0: it waits for process 0 to receive the data.
**Case 2: Receive first and then send.** The following code leads to a deadlock regardless of how much system buffer one has:

```c
if (myrank==0) {
    rc = MPI_Recv(recvbuf, ...);
    rc = MPI_Send(sendbuf, ...);
} else if (myrank==1) {
    rc = MPI_Recv(recvbuf, ...);
    rc = MPI_Send(sendbuf, ...);
}
```

If you use MPI_Isend instead of MPI_Send, deadlock still occurs. On the other hand, the following code can be safely executed.

```c
if (myrank==0) {
    rc = MPI_Irecv(recvbuf, ..., ireq, ...);
    rc = MPI_Send(sendbuf, ...);
    rc = MPI_Wait(ireq, ...);
} else if (myrank==1) {
    rc = MPI_Irecv(recvbuf, ..., ireq, ...);
    rc = MPI_Send(sendbuf, ...);
    rc = MPI_Wait(ireq, ...);
}
```

**Case 3: One process sends and receives; the other receives and sends** It is always safe to order the calls of MPI_(I)SEND and MPI_(I)RECV so that a send subroutine call at one process and a corresponding receive subroutine call at the other process appear in matching order:

```c
if (myrank==0) {
    rc = MPI_Send(sendbuf, ...);
    rc = MPI_Recv(recvbuf, ...);
} else if (myrank==1) {
    rc = MPI_Recv(recvbuf, ...);
    rc = MPI_Send(sendbuf, ...);
}
```

In this case, you can use either blocking or non-blocking subroutines. Considering the previous options, performance, and the avoidance of deadlocks, it is recommended to use the following code:

```c
if (myrank==0) {
    rc = MPI_Isend(sendbuf, ..., ireq1, ...);
    rc = MPI_Irecv(recvbuf, ..., ireq2, ...);
} else if (myrank==1) {
    rc = MPI_Irecv(recvbuf, ..., ireq2, ...);
    rc = MPI_Isend(sendbuf, ..., ireq1, ...);
}
```
rc = MPI_Isend(sendbuf, ..., ireq1, ...);
rc = MPI_Irecv(recvbuf, ..., ireq2, ...);
rc = MPI_Wait(ireq1, ...);
rc = MPI_Wait(ireq2, ...);

3.4.2. OOMPI

Using OOMPI if one already knows MPI is actually quite easy. As was stated in section 2.5.1 OOMPI is merely a thin layer of object orientation on top of MPI, and both offer basically the same functionality. There are some differences in programming style since MPI is a procedure library, while OOMPI is a class library. Some of the most important differences are listed below.

Sending and receiving data

In OOMPI, the send and receive methods do not need to know where to send data or where to receive data from. Instead, the send and receive methods are methods of the Port class (OOMPI_Port), which has source and destination addresses “hardwired” into it. This means that we can send and receive data like this:

int i, j;
OOMPI_Port Port;
...
Port.Send(i); // Send the integer i
Port.Receive(j); // Recieve the integer j

The OOMPI_Port object contains information about its MPI communicator and the rank of the process to whom the message is to be sent. Note, however, that although the expression Port.Send(i) is a very clear statement of what we want to do, there is no explicit construction of a message object. Rather, the message object is implicitly constructed. Port objects are very closely related to communicator objects, a port is said to be a communicator’s view of a process. Thus, a communicator contains a collection of ports, one for each participating process.

Individual ports within a communicator are accessed with the array index ([ ]) operator, i.e., the i th port of an OOMPI communicator object c is c[i]. The following code fragment shows an example of sending and receiving:

int i, j, m, n;
OOMPI_Intra_comm Comm;
Comm[m].Send(i); // Send integer i to process with rank m
Comm[n].Receive(j); // Recieve integer j from process with rank n

MPI-I provides the capability for users to pack their own messages. In OOMPI, this is represented using the familiar stream interface (also used for console and file I/O in C++). In the following example, a message of 400 integers is constructed and sent:
int i, rank = OOMPI_COMM_WORLD.Rank();
OMPI_Port Port;
OMPI_Packed msg(OMMPI_COMM_WORLD.Pack_size(i, 400),
    OOMPI_COMM_WORLD, PACK_TAG);
msg.Start();
for (i = 0; i < 200; i++) msg << i << rank;
msg.End();
Port << msg;

The arguments to the OOMPI_Packed constructor are the size of the buffer to be created, the communicator, and the tag to be used for sending and receiving this instance. Note that no count argument is passed to the Port when sending the object. An OOMPI_Packed object inherently knows its count. That is, sending an OOMPI_Packed object will send as many bytes as were packed. Receiving an OOMPI Packed object will attempt to receive a message as long as the entire buffer. MPI-1 allows the normal receipt of a shorter-than-expected message.

**Communicator classes**

The classes associated with communicators are:

- OOMPI_Comm
- OOMPI_Comm_world
- OOMPI_Group
- OOMPI_Intra_comm
- OOMPI_Cart_comm
- OOMPI_Port
- OOMPI_Inter_comm
- OOMPI_Graph_comm
- OOMPI:Any_port

These objects encapsulate the functionality of MPI communicators and are the basis for all communication (point-to-point and collective). The communicator objects contain the algebraic group object used to create the communicator, a port object for each rank in the communicator, and an error handler (if there is one). The OOMPI_Comm object is an abstract base class from which the classes OOMPI_Intra_comm, OOMPI_Inter_comm, and OOMPI_Comm_world are derived. These classes provide the functionality associated with intra-communicators, inter-communicators, and MPI_COMM_WORLD, respectively. Note that the class OOMPI_Comm_world has only one instance of an object, the global variable OOMPI_COMM_WORLD.
Message classes

The OOMPI classes associated with messages are:

- OOMPI_Message
- OOMPI_User_type
- OOMPI_Request
- OOMPI_Array_message
- OOMPI_Packed
- OOMPI_Status
- OOMPI_Datatype
- OOMPI_Op

The MPI-1 C bindings specify that all data buffers are of type void *. Since the type of the data is not inherent in the argument, a second argument must be specified to provide the type. In C++, functions can be overloaded based on the type of their formal parameters, but there are two problems with this approach: it leads to a function explosion and user-defined types are not included in this scheme. Using OOMPI_Message as a base class with lightweight default promotions for all base types provides a clean and efficient way of not having to overload functions for each type.

The OOMPI_Message object is a base class that is used to unify diverse data types (base C++ types and user-defined types) into one object type. That is, every MPI-1 function that included a void * data buffer argument is replaced with an OOMPI_Message argument. Since the OOMPI_Message object includes the MPI datatype and a pointer to the top of the data, functions that have OOMPI_Message arguments inherently know the data’s type and where it resides in memory.

Implementing a FEM solver using OOMPI

Implementing a FEM solver or similar application using OOMPI should be quite straightforward. The basic program structure listed in section 3.2 can be used. For communication among processes, use Port and Communicator objects (for communication among all the processes in the cluster, the global OOMPI_COMM_WORLD object can be used). For example, data can be distributed from a master process to a set of worker processes by the following code segment:

```c
int numprocs = OOMPI_COMM_WORLD.Size();
// master process is rank 0, worker processes is in the range
// 1..numprocs-1
for(int i=1; i<numprocs; i++) {
```
... // prepare the data to be sent
OOMPI_COMM_WORLD[i].Send(the_data); // the_data is a message
    // object
}

When sending a large amount of data one must try to pack the data into suitably-sized messages. This helps reduce communication overhead and improves performance. Data can be packed using the OOMPI_Packed message type, or by using user-defined data types which contain the data to be sent.

### 3.4.3. CORBA

Writing an application using CORBA usually consists of the following steps:

1. Design the interfaces between components using IDL.
2. Compile the IDL files using the IDL compiler, which produces server and client skeleton code.
3. Implement the server and client. Implementation on the server side involves writing the implementation of the methods and interfaces defined in step one.

#### Interface

When designing a distributed application using CORBA, one of the first things that need to be considered are the interfaces of the remote objects that form the different processes in the application. As stated in section 2.5.3, these interfaces are specified using IDL.

Since the CORBA interfaces define not only the programmatic interface between objects (i.e. what objects exist, and what methods can be called), but also the communication that takes place between processes, great care must be taken so that the interfaces don't end up being inherently inefficient. As an example, consider a hypothetical server process whose task it is to transfer data for a set of vertices to worker, or client, tasks. Each vertex has x, y, and z coordinates, expressed as floating-point values. Two interfaces, written in IDL, are listed in figures 3.6 and 3.7.

```java
interface DataServer {
    int GetNumberOfVertices();
    void GetVertexData(in int vertex_number,
        out float v_x, out float v_y, out float v_z);
};
```

Figure 3.6.: CORBA interface for transferring element data (1)

When using the interface in figure 3.6, one is supposed to first call the GetNumberOfVertices method to get the total number of vertices, and then call GetVertexData in a loop to get the coordinates of each vertex. This means that one message
interface DataServer {
    typedef sequence<float> float_array;
    void GetVertexData(out float_array v_x, 
    out float_array v_y, out float_array v_z);
};

Figure 3.7: CORBA interface for transferring element data (2)

is potentially transferred from the master task to the worker for each vertex. CORBA implementations don’t necessarily assemble many method calls into one large message, although some implementations try to. Each message contains the method name and other control information in addition to the actual data, and for each method call one has the added overhead of marshalling the call on the server side, and demultiplexing and demarshalling on the client. Clearly this is an inefficient approach for transferring large amounts of data.

In the interface in figure 3.7 the \texttt{GetVertexData} method transfers the data for all vertices in one call. This is accomplished by first defining a sequence type\footnote{A sequence is a CORBA data type which works like a C/C++ array, except that the length of the sequence doesn’t need to be specified.}, and then using that type in the definition of \texttt{GetVertexData}. This means that all vertex data is transferred in a single message. The method name and control information is included once in the message, and the rest of the message consists of the vertex data. Demultiplexing on the client side is only done once. When transferring a large number of vertices, this is much more efficient than using the interface in figure 3.6. (Of course, if only one or two vertices are needed, and not the whole array, the first approach would be the most efficient.)

\textbf{IDL Compiling}

When starting to implement the application, one needs to select a suitable implementation language. The OMG has defined a series of language mappings that define how IDL features (keywords and data types) are mapped to a programming language. Language mappings for CORBA exist for Ada, C, C++, COBOL, CORBA SCRIPT, Java, Lisp, Python and Smalltalk. For high-performance CORBA applications C or C++ is commonly selected. A notable omission from the list of language mappings is support for FORTRAN.

The IDL specification previously created is run through a special IDL compiler which outputs header and code files that can form the basis for implementation of both server and client. Because server and client code is generated from the same IDL file, interoperability is guaranteed. In the IDL compiler, each IDL feature is mapped to its C++ counterpart. For the interface in figure 3.7, a simplified C++ header file is listed in figure 3.8. As can be seen, interfaces are mapped to classes, sequences are mapped to arrays\footnote{In an actual implementation, sequences are commonly implemented as classes that employ operator}, \ldots
and output arguments are mapped to reference arguments in C++.

    class DataServer {
    public:
        typedef float[] float_array;
        void GetVertexData(float_array &v_x, float_array &v_y,
                            float_array &v_z);
    };

Figure 3.8: Header file (pseudocode) resulting from running the interface in fig. 3.7 though the IDL compiler.

In a client application, it would be possible to create DataServer instances, and in a server application, the DataServer class would be subclassed by an actual implementation class.

**Implementation - Server**

When implementing a server, one subclasses the classes created by the IDL compiler from the IDL sources. A typical implementation of the DataServer class is listed in figure 3.9. As can be seen, the implementation class Impl_DataServer inherits from the IDL-generated DataServer class.

In addition to creating an object implementation, one needs to properly initialize the CORBA library and register the object with the ORB so that clients can find them. The most used methods that clients use for locating objects are

- using hard-coded object references that the server prints out or otherwise makes available, and the client uses via a command-line parameter or similar mechanism
- using the CORBA naming service, which provides a hierarchical name space that can be used to convert from symbolic object names to object references. This works similarly to the DNS service.

Figure 3.10 shows an example main program that does all the necessary initialization. This includes

1. initializing CORBA,
2. creating an instance of the Impl_DataServer class,
3. registering the object with the naming service, and
4. activating the object, so that clients can begin calling its methods.
class Impl_DataServer : public DataServer {
    private:
    float vertex_X[1024];
    float vertex_Y[1024];
    float vertex_Z[1024];

    public:
    Impl_DataServer { // Constructor
        // load the vertex data here
    }
    void GetVertexData(float_array &v_x, float_array &v_y,
                        float_array &v_z) { // GetVertexData implementation
        v_x.size(1024); // initialize the sequences to hold
        v_y.size(1024); // 1024 elements each.
        v_z.size(1024);
        for(int i=0; i<1024; i++) {
            v_x[i] = vertex_X[i];
            v_y[i] = vertex_Y[i];
            v_z[i] = vertex_Z[i];
        }
    }
};

Figure 3.9.: Simple implementation of the DataServer server.
int main(int argc, char *argv[]) {
  // 1. Initialize CORBA
  CORBA::ORB_var orb = _ORB_init(argc, argv);
  CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var poa =
    PortableServer::POA::_narrow(obj);
  // 2. Create instance of the Impl_DataServer class
  Impl_DataServer ds* = new Impl_DataServer;

  // 3. Register object with naming service
  // This binds our object to the symbolic name
  // ’/Objects.dir/DataServer.object’
  // define the name
  CosNaming::Name ds_name;
  ds_name.length(2);
  ds_name[0].id = ’Objects’;
  ds_name[0].kind = ’dir’;
  ds_name[1].id = ’DataServer’;
  ds_name[1].kind = ’object’;
  // get reference to root naming context
  // (the ’root directory’ of the naming tree)
  CORBA::Object_var ns_obj;
  ns_obj =
    orb->resolve_initial_references("NameService");
  CosNaming::NamingContext rootNaming =
    CosNaming::NamingContext::_narrow(ns_obj);
  // create an reference to our object and
  // bind the name to this reference
  CORBA::Object_var ds_obj = ds->_this;
  rootNaming->bind(ds_name, ds_obj);
  ds->_remove_ref();

  // 4. Activate the object
  PortableServer::ObjectId_var objid =
    poa->activate_object(obj);
  // ... and serve requests
  PortableServer::PosManager_var pman =
    poa->the_POAManager();
  pman->activate();
  orb->run(); // the program blocks here!
}

Figure 3.10: Example main program for CORBA server
When waiting for requests from the client, the server blocks. This fits well with the client-server programming paradigm. If another programming paradigm is to be used, the server may need to create another thread of execution to continue processing while the CORBA thread waits for client requests.

**Implementation - Client**

Implementing a CORBA client usually is quite simple. After getting a reference to a server object, the client can use this object reference as if it were a local object. The ORB together with the marshaling/demarshaling code created by the IDL compiler handles the rest of the process transparently. Figure 3.11 shows a small client program that connects to the **DataServer**, transfer vertex data from the server, and prints it out.

As can be seen from the code, the initialization procedure can seem a little complex, but it’s mostly boilerplate code. After the object reference has been retrieved the **DataServer** object can be used like any C++ object.

**Implementing a FEM solver using CORBA**

Using the above steps, it is possible to implement a FEM solver using CORBA. When doing so, one first needs to decide if a client/server programming paradigm will be used. If not, additional work may be required to make the CORBA client/server architecture to fit with the overall architecture of the system.

Then interfaces need to be defined, for distributing raw data to each processor, for updating data between processors while the computation is proceeding, and for collecting results in the end. The interfaces must be set up in such a way that they

- provide a logical and easy-to-use way for processes in the system to exchange information, while they
- provide efficient communication between processes.

The last step is the actual implementation, and as long as the interfaces are well-defined, and the system architecture fits with the CORBA architecture, the implementation should be fairly straightforward.

**3.4.4. Framework**

To implementing a FEM solver or similar application using the framework described in section 3.3, one needs to subclass the **UserAlg** and **UserData** classes. The **UserData** subclass should contain the data needed for the computation, while the **UserAlg** subclass should provide the actual algorithm to be run. The **UserAlg** class contains a `main()` function.

4 Object Request Broker, a software component that sets up connections between client and server and handles communications between them.

5 The DNS services is used on the internet to convert hostnames to IP addresses and vice versa.
int main(int argc, char *argv[]) 
{
 // 1. Initialize CORBA
 CORBA::ORB_var orb = _ORB_init(argc,argv);

 // 2. Retrieve the object reference from the
 // naming service.
 // This looks up our object from the symbolic name
 // '/Objects.dir/DataServer.object'

 // define the name
 CosNaming::Name ds_name;
 ds_name.length(2);
 ds_name[0].id = 'Objects';
 ds_name[0].kind = 'dir';
 ds_name[1].id = 'DataServer';
 ds_name[1].kind = 'object';
 // get reference to root naming context
 // (the 'root directory' of the naming tree)
 CORBA::Object_var ns_obj;
 ns_obj =
 orb->resolve_initial_references("NameService");
 CosNaming::NamingContext rootNaming =
 CosNaming::NamingContext::_narrow(ns_obj);
 // Look up the name in the directory, and
 // retrieve the object reference.
 CORBA::Object_var ds_obj =
 rootNaming->resolve(ds_name);
 DataServer_var ds = DataServer::_narrow(ds_obj);

 // 4. Use the object’s methods to retrieve vertex
 // data.
 float_array x;
 float_array y;
 float_array z;
 ds.GetVertexData(x,y,z); // Remote object method call!

 // 5. Print vertex data.
 for(int i=0; i<x.size(); i++) {
   cout << "Vertex " << i << " (" <<
       x[i] << ", " << y[i] << ", " << z[i] << 
   "\n";
   }
}

Figure 3.11.: Example main program for CORBA client
method that can be overridden for this purpose. This algorithm should be designed to
run on an arbitrary subdomain of the problem domain.

The framework will read in model data, partition the domain into subdomains, and
run the main() method of the UserAlg subclass in each worker process (i.e. on all
processors except on the first). The framework will automatically provide the worker
process with the element data of a subdomain, as well as information on the elements’
neighbours. When the computation running in a worker process needs updated data from
its neighbours, it calls the framework Update() method. This method will automati-
cally exchange data as needed with worker processes corresponding to the subdomain’s
methods.

3.5. Research agenda

In section 3.4 we have covered four different approaches to implementing a distributed
FEM solver: using an MPI library directly, using an object-oriented class library such as
OMPI, using CORBA, and lastly, using a special-purpose object-oriented framework.
All these approaches have different advantages and disadvantages, and we would like to
compare them to see which one is best suited to the problem at hand.

3.5.1. Hypothesis

Our hypothesis is that using a special-purpose framework is the best approach for creating
FEM solver or similar applications. We believe that it is much simpler to implement
such an application using the framework, than using other approaches, because many of
the typical problems encountered are already solved once and for all in the framework.
While there may be overhead (e.g. communication overhead, OO language overhead,
method call overhead) associated with this approach that can be eliminated with the
other approaches, we do not believe that this overhead is sufficiently large to warrant the
extra work needed to implement the application.

3.5.2. Method

In order to be able to get an indication of which approach is best suited, we plan to write
four different implementations, one for each approach, and compare these with respect
to the usability and performance of the library or framework.

For usability, we wish to find out how difficult it is to implement a solution to a
problem based on a particular approach. We will find this by answering the following
two questions.

• How much code must be written in order to implement a problem using this ap-
  proach?

• How much does one have to think about the fact that one is writing a parallel
  program?
We will try to find an indication of the general performance of the library or framework by measuring the following:

- How much time, in total, does the execution of a particular implementation on a given hardware platform take?
- If one increases the number of available processors, does the execution time decrease accordingly?

In order to be able to create four different implementation of the same problem in the time available, we have not chosen to try to implement a FEM solver. Instead, we have chosen a simpler problem, but one that can be solved in a similar way.

### 3.5.3. Problem description

The problem we have selected is the following:

For a three-dimensional mesh, consisting of elements, faces, and vertices, calculate the total surface area of the mesh. The total surface area is the sum of the areas of all faces that form the external boundary of the mesh. The faces containing internal boundaries are not counted in the total.

Figure 3.12 illustrates area calculation of a cube. The black edges line the faces that are included in the calculation, while the light gray edges line faces that are internal to the mesh and are not calculated towards the total.

![Area calculation of a cube](image)

**Figure 3.12:** Area calculation of a cube.

The area of an individual triangular\(^6\) face can be calculated using the cross product. See figure 3.13. If we call the vertices that make up a triangular face \(v_1, v_2, \) and \(v_3,\) and we call the vector from \(v_1\) to \(v_2\) \(\vec{v}_1\vec{v}_2\), and the vector from \(v_1\) to \(v_3\) \(\vec{v}_1\vec{v}_3\), we can calculate the cross product \(\vec{v}_1\vec{v}_2 \times \vec{v}_1\vec{v}_3\). We then know that that vector's length is equal

---

\(^6\)Theoretically, the faces could be of any shape, but we have particularized the solution to triangular faces, since the mesh data that we had available contained triangular faces.
to the area of the parallelogram formed by the vectors $\vec{v}_1 \vec{v}_2$ and $\vec{v}_1 \vec{v}_3$. Since the area of the parallelogram is twice the area of the triangular face, the area of the face can be calculated by the following equation:

$$A = \frac{|\vec{v}_1 \vec{v}_2 \times \vec{v}_1 \vec{v}_3|}{2}$$

Figure 3.13.: Area calculation of a face using the cross product

The total surface area of a mesh can be calculated as follows: for all elements and all faces, if a face of an element does not have another element as a neighbour, its area is counted towards the total surface area of the mesh. The neighbour test ensures that the areas of faces that are internal to the mesh are not added to the total.

The pseudocode C++ algorithm below shows how the surface area can be calculated in a serial program.

```cpp
double area = 0.0;
for(int i=0; i<number_of_elements; i++) {
    for(int j=0; j<number_of_faces_per_elem; j++) {
        if(!elements[i].faces[j].hasNeighbour()) {
            area += elements[i].faces[j].area;
        }
    }
}
```

To solve the area problem using a distributed system, one must use all four steps for a parallel FEM solver listed in section 3.2, however, one does not need to update data between tasks in step three as in a FEM solver. In section 6.3, we will discuss how the absence of this requirement impacts the generalization of our results to FEM solvers.
3.5.4. Usability metrics

In order to evaluate the usability of the different libraries and frameworks, we will use the following metrics:

- the learning time
- the approximate number of lines of code that our implementations consists of
- the number of classes our implementations contain
- how many parallel decision points there are in each program

Learning time

The learning time is the amount of time a programmer must spend familiarizing himself/herself with the library or framework before he or she can use it productively. This can be difficult to determine exactly. It depends largely on the experience the programmer has with similar implementation approaches. In addition to the the learning time of a particular library/framework, there is the learning time of C++ to consider. For a person with some knowledge of other programming languages, we estimate that about two months is required in order to become an efficient C++ programmer, but the actual learning time depends largely on the programming skills of the user.

Lines of code

We have defined a line of code as any line in a C++ implementation (*.cc) or header (*.h) file, excluding blank lines and single line comments (comments starting with //). Opening or closing braces were included in the count, and for practical reasons, C-style multi-line comments (beginning with /* and ending with */) were also included (these were not used to a great degree in the program code).

Number of classes

The number of classes is a measurement of both the complexity and order of an implementation. Having to create a large number of classes in order to solve a problem is time-consuming work. Of course one need to strike a balance between having a few, large classes and having many, small classes. Few classes and many codes of line per class makes it hard to navigate and make changes, while creating many classes compared to the number of code lines might mean much unneeded work.

Parallel Decision Points

Parallel decision points are places in a program where the user has to think about the parallelization of his or her program. Examples of such points are when:

- a communication library must be initialized
• it must be decided how to divide data between processes
• data must be distributed among processes
• data must be updated between processes
• results must be collected from other processes
• processes running on different tasks must be synchronized

The amount of work required to deal with a parallel decision point varies. Initialization of a communication library and synchronizing processes usually is quite simple, but dividing, distributing and updating data can be hard.

3.5.5. Performance metrics

These are the metrics that will be used in the performance evaluation.

Total Execution time

The total execution time \( T(n) \) for a parallel program on \( n \) processors, is the time it takes from the first process starts its execution, to the time the last process finishes. If the processes start simultaneously, the execution time is equal to the execution time of the process that takes the longest time. Execution time can be divided into code execution time \( C_i \) (for the \( i \)'th processor) and communication time \( K \). \( K \) is the sum of all the communication that is done throughout the execution. The communication time will obviously increase as the number of processes increase.

\[
T(n) = K + \sum_{i=0}^{n-1} C_i
\]  

Communication time

The time needed to transfer a single message over a network typically consists of a constant and a variable term. The variable term accounts for the size of the message and the capacity of the network, and is proportional to the size of the message, and inversely proportional to the bandwidth of the network. The constant term is called the network latency, and accounts for the time from the first bit of information is sent from the source computer to the time when that bit is received by the destination computer.

Thus we get the following equation:

\[
K = L + \frac{M}{B}
\]  

\( K \) is communication time, in seconds, \( L \) is latency, in seconds, \( M \) is the message size, in bits, and \( B \) is the network bandwidth in bits per second.
Speed-up

It is known that it is unrealistic that the entire serial program can be divided and run on a distributed system. Only a percentage of the serial program will be affected by adding more processors to the system. This means there exist some threshold over which it is no longer cost efficient to add more processors. We can estimate the change of execution time from the serial to the parallel version of a program by using the following equation

\[ R(p, n) = (1 - p) + \frac{p}{n} \]  

(3.3)

Here \( p \) is the percentage that can be parallelized, \( n \) is the number of processors and \( R \) represent the ratio of parallel execution time to the serial execution time. For example if 60% of the task is parallelizable, and we have 4 processors, we get:

\[ (1 - 0.6) + \frac{0.6}{4} = 0.4 + 0.15 = 0.55 \]  

(3.4)

From equation.3.4 we can see that the parallel execution time is 55% of the serial execution time.

The equation is usable only for estimation and cannot be used to calculate any real results since it is based on a very simplified cost/time model. The communication time, for example, is not included. Still, in the cases where the code execution time greatly outweighs the communication time, it gives results that are close to reality. A difficult problem with the usage of this formula is to determine the percentage of the code that is parallelizable.

What we are interested in is a cost/time model that can give us an indication of when it is no longer economical to add more processors. Speed-up is the degree of performance difference between a serialized program and the same program run in a cluster with a given number of processors. The equation looks like this:

\[ S(n) = \frac{t}{T(n)} \]  

(3.5)

where \( t \) is the serial execution time and \( T(n) \) is the runtime of the parallel version. As you can see an ideal case is when \( S(n) = R(1, n) \), which gives us \( S(n) = p \). This means that (near) 100% percent of the program can be parallelized and the communication time is negligible. By doubling the number of processors, the speed-up is doubled, which gives us a linear graph for \( S(n) \). For non-trivial problems, it's usually impossible to parallelize the whole program, and the speed-up will be worse than the ideal case.

Efficiency

Efficiency is defined as the speed-up divided by the number of processors:

\[ E(n) = \frac{S(n)}{n} \]
This value gives an indication of how much the processors are kept occupied, or in other words, how good the speed-up result really is compared to the change in the number of processors. In most cases, one can easily see that the speed-up increases as the number of processors increases, but is it increasing enough? By plotting the efficiency in a graph for program executions with different number of processors we can find an efficiency maximum. This maximum is where the performance of the system compared to the number of processors is the highest. Beyond this maximum, total execution time will still decrease as more processors are added, but it will not decrease by as much.

The efficiency gives a good indication on when a parallel program works at its most efficient. Some parallel software is optimized for many processors, and performs poorly when run on only a few processors, while other software works great with a few processors, but performs poorly with many processors.

3.5.6 Planning

We will test the execution time by executing a large number of program runs in total on the same six-processor cluster. There are 4 implementations and 3 test files and the programs will be run on 3, 4, 5 and six processors. For each combination we plan on doing 20 runs, which gives us a total number of 960 program executions. For each run we will measure total execution time, and calculate speed-up and efficiency based on these numbers.

We have selected 20 runs because it seemed many enough to eliminate variations due to fluctuating CPU and network usage not related to our implementations, while at the same time was few enough so that the test would be finished within an acceptable amount of time. A few test runs done before testing indicated that the execution time for the same input and the same number of processors didn’t vary significantly.

We will also range the implementation approaches from worst to best from a usability point of view by using the usability metrics defined in section 3.5.4.
4. Implementation

This chapter documents the four different implementations that we have made solving the area computation problem.

4.1. Common structure and shared elements

Regardless of which method of communication is used, the general structure in the program remains the same. All implementations use a Master/Worker implementation paradigm, in which there is one Master process and several Worker processes. The Master process controls the calculation process while the Worker processes carry out the actual calculations. Figure 4.1 shows a general flowchart for both the Master and the Worker processes.

As can be seen from the figure, the Master process is responsible for:

- reading the data,
- partitioning the data,
- sending data to Worker processes,
- collecting results, and
- printing the final result.

The Worker process

- receives data from the Master
- calculates the surface area of all its elements’ faces, and
- sends this result to the Master.

In the framework version of the program, the structure is embedded in the framework itself, and is not directly visible for the user. In all other versions, this structure is explicit in the program.
Figure 4.1.: Flowchart of the Master and Worker processes. Dotted lines indicate activities that must happen simultaneously (i.e. sends and receives which correspond to each other).
4.1.1. Shared elements

Some program code could be shared between the implementations. This includes the procedures that

- read in the mesh data from file
- partitions the domain (i.e. the mesh) into subdomains
- prepares the data that will be sent to each processor
- calculates the area of a given subdomain

For the framework version of the program, all items except the last is handled by the framework. In the non-framework-versions, some code from the framework was used. This includes

- from the Master class, code that reads in the mesh data and partitions the domain (the GetDomain() and GetPartition() methods, respectively)
- the MeshStruct class, which was used to store subdomain data.

For the non-framework versions, a procedure that prepares the data that will be send to each processor was implemented. This procedure picks elements from the total data based on information from the domain partitioning procedure, and packs the element, face and vertex data into arrays, taking care not to duplicate face or vertex data (in case the same face or vertex is referenced two or more elements or faces). The code for this procedure is listed in section B.1 of the appendix.

The program code for the functions that calculates the area of a given subdomain is listed in sections B.2 and B.3 in the appendix.

4.2. MPI

MPI has been covered in detail in sections 2.5.1 and 3.4.1. Here we will briefly cover the routines that we used in our implementation.

First of all we have the enviromental procedures that needs to be called. These are MPI_Init(), which initializes the library, MPI_Comm_size(), which returns the number of processors and MPI_Comm_rank(), which returns the rank of the actual processor. This rank is the ID of the process. Each worker_task called gets its own rank (master has rank 0), and the number of worker_tasks called is equal to the number of processes that will run. Last in the procedures MPI_Finalize() are used to end the MPI communication session. No further MPI calls can be made after this call.

The master distributes the domain data to several workers that calculate a partial result and sends this back to the master. When sending and receiving we use the basic MPI_Send() and MPI_Recv() calls. The master task also need to wait for all worker
tasks to finish receiving data before it can start collecting results. MPI has a call that does exactly this, \texttt{MPI\_Barrier()}.

One of the more troublesome issues with MPI was the packing of data to be sent. The elements, faces and vertices need to be packed into arrays, and in the receiving end, the worker unpack the data much much in the same way it was packed. The MPI calls that were used for this are \texttt{MPI\_Pack()} and \texttt{MPI\_Unpack()}.

MPI includes all the functionality needed for communication in our program, and the library was generally easy to use. However, most MPI functions need to be called with many parameters that will be the same for all function calls in the program, and this cluttered the program with unnecessary detail. In OOMPI, these parameters are given defaults and need not be specified.

### 4.3. OOMPI

As one might suspect, using OOMPI for communication is very similar to using MPI. The function calls are basically the same. This means that it is very easy to convert a program which uses MPI to use OOMPI.

The environmental functions in OOMPI belong to the \texttt{OMPI\_Intra\_comm} class, initiated as \texttt{OMPI\_COMM\_WORLD} in our program. These function include \texttt{Init()}, \texttt{Size()}, \texttt{Rank()} and \texttt{Set\_error\_action()}. The latter is specific for OOMPI and specifies how errors should be handled. The \texttt{OMPI\_COMM\_WORLD} object also holds an array of \texttt{OMPI\_Port} objects for all processes in the cluster. The \texttt{OMPI\_Port} class has \texttt{Send()} and \texttt{Recv()} methods which are used for communication. See \S 3.4.2 for more information on this.

In OOMPI packed data messages are created a bit differently from MPI; a separate class is used for this, \texttt{OMPI\_Packed}. The method \texttt{Pack()} adds elements to the package to be sent. Sending in packages is an advantage since only one big transfer is needed. The package can be sent using a normal \texttt{Send()} call. The unpacking does correspondingly with the \texttt{Unpack()} method.

### 4.4. CORBA

As previously stated, CORBA uses a Client-Server architecture, and for our problem, this architecture could fairly easily be mapped to the Master/Worker architecture that we were using fairly. It was natural for the Master to function as the Server and the Workers to function as Clients. The server implements the following IDL interface:

```idl
interface master {
  typedef sequence<long> long_array;
  typedef sequence<double> double_array;
  void getSubDomain(in long pid,
                     out long no_elems, out long no_faces, out long no_verts,
                     out long_array elems, out long_array faces,
```

65
out double_array vertices;
void reportResult(in long pid, in double area);

All methods take the process ID of the worker task as the first argument (worker process ID’s range from 1 to total number of tasks minus one).

The workers call the getSubDomain() method to get the subdomain data. The data is returned as sequences of long integers and doubles. When the workers have completed their calculation, they call the reportResult() method with the total area as an argument. When all workers have reported their result, the master prints out the total area and exits.

A separate class (called master_i) was created which implements the interface. In the master main() method an instance of this class is created and activated.

The simple mapping between the Client/Server and the Master/Worker architectures is only possible if communication only occurs between the master task and one worker task at a time. If the workers need to communicate with each other, or the master needs to communicate with several worker tasks simultaneously, it can be difficult to get the strict CORBA Client/Server architecture to fit in. This issue is discussed further in section 6.3.

4.5. Framework

Solving our problem with the framework was extremely simple. We simply subclassed the UserAlg algorithm and provided the Main method as shown below:

void TheAlg::Main {
    double area = calculate_area();
    OOMPI_COMM_WORLD[0].Send(area);
}

The calculate_area() function call on the second line calls the area-calculation function listed in section B.3, which was the same for all implementations.

As can be seen, the last line of the Main method uses OOMPI directly to send back the result to the master. This is because the framework, in its current version, has no simple way of sending back the result. In addition to adding this line, we had to modify the main() function of the master task so that it collects the results from all worker tasks by calling the corresponding Receive() method in a loop, and prints out the total area in the end.
5. Test results

5.1. Usability

The results from the evaluation are presented in table 5.1.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Learning Time</th>
<th>LOC</th>
<th>NOC</th>
<th>Avg. LOC/class</th>
<th>Decision Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>2-3 days</td>
<td>751</td>
<td>2</td>
<td>375</td>
<td>4</td>
</tr>
<tr>
<td>OOMPI</td>
<td>2-3 days</td>
<td>728</td>
<td>2</td>
<td>364</td>
<td>6</td>
</tr>
<tr>
<td>CORBA</td>
<td>7 days</td>
<td>952</td>
<td>3</td>
<td>317</td>
<td>6</td>
</tr>
<tr>
<td>Framework</td>
<td>1 day</td>
<td>74</td>
<td>1</td>
<td>74</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1.: Results from the usability evaluation. LOC is the number of lines of code for a particular implementation, NOC is the number of classes.

5.2. Performance

5.2.1. About the Cluster

For performance testing, we used the ITEA Beowulf Linux cluster located at NTNU. This is a 6-processor cluster consisting of 3 nodes with two Intel Pentium III 500 MHz processors and 512MB RAM each. In addition, the cluster includes a front-end host (for communicating with the outside world) with a Cyrix MII 230 MHz processor and 128MB RAM.

5.2.2. About the Software

Table 5.2 lists the software we used. In addition to the listed software, *bash* shell scripts and *Perl* scripts were used to perform the actual test runs automatically and to collect and compile results. Some of these scripts are listed in appendix C. GNU *make* was used to build the software.

5.2.3. Test files

Three different files were used as input to the area-calculating algorithm. Table 5.3 lists details on these files. The cube.dat file results are not discussed in the report, because it is a very small file and not at all representative for the kind of problems distributed
<table>
<thead>
<tr>
<th>Software type</th>
<th>Name</th>
<th>Version</th>
<th>Authors/vendors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System Kernel</td>
<td>Linux</td>
<td>2.2.16-1</td>
<td>The Linux Kernel Team</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC</td>
<td>2.95-2</td>
<td>The GNU Project</td>
</tr>
<tr>
<td>C library</td>
<td>glibc</td>
<td>2.1.3</td>
<td>The GNU Project</td>
</tr>
<tr>
<td>CORBA Implementation</td>
<td>omniORB</td>
<td>3.0.4</td>
<td>AT&amp;T Laboratories, Cambridge</td>
</tr>
<tr>
<td>MPI Implementation</td>
<td>LAM</td>
<td>6.5.5</td>
<td>LAM Team, University of Notre Dame</td>
</tr>
<tr>
<td>OOMPI</td>
<td>OOMPI</td>
<td>1.0.3</td>
<td>Open Systems Laboratory, Indiana University</td>
</tr>
<tr>
<td>Partitioning library</td>
<td>METIS</td>
<td>4.0</td>
<td>George Karypis, University of Minesota</td>
</tr>
<tr>
<td>Framework</td>
<td>OODFEM</td>
<td>-</td>
<td>Roxana E. Diaconescu, IDI, NTNU</td>
</tr>
</tbody>
</table>

Table 5.2.: Software used for performance evaluation

Systems will be used for. For such small problems, a serial implementation will always be faster. However, full results are included in the appendix, for completeness.

<table>
<thead>
<tr>
<th>Filename</th>
<th>No. verts</th>
<th>No. faces</th>
<th>No. elems</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cube.dat</td>
<td>9</td>
<td>30</td>
<td>12</td>
<td>A simple cube</td>
</tr>
<tr>
<td>tiny.dat</td>
<td>1331</td>
<td>12000</td>
<td>6000</td>
<td>A larger cube</td>
</tr>
<tr>
<td>small.dat</td>
<td>2933</td>
<td>31818</td>
<td>15581</td>
<td>A sphere</td>
</tr>
</tbody>
</table>

Table 5.3.: Test files used for input

5.2.4. Results

In this section, results from the performance testing are presented in the form of bar charts. Tables containing the exact results can be found in the appendix.

Average execution time

Figures 5.1 and 5.2 show the average execution time for test runs with the two input files tiny.dat and small.dat. For each figure, the average execution time for the serial version of the algorithm, run on one processor, is also shown.

Speed-up

Figures 5.3 and 5.4 show the speed-up (defined in section 3.5.5) for the tiny.dat and small.dat files.
Figure 5.1.: Average execution time for the tiny.dat file. Average serial execution time was 8.69 seconds.

Figure 5.2.: Average execution time for the small.dat file. Average serial execution time was 29.83 seconds.
Figure 5.3.: Speed-up for the tiny.dat file

Figure 5.4.: Speed-up for the small.dat file
Efficiency

Figures 5.5 and 5.6 show the efficiency (defined in section 3.5.5) for the tiny.dat and small.dat files.

![Efficiency for the tiny.dat file](image1)

![Efficiency for the small.dat file](image2)

Figure 5.5: Efficiency for the tiny.dat file

Figure 5.6: Efficiency for the small.dat file
6. Evaluation

6.1. Evaluation

This report has two foci; it addresses both the usability and the performance of distributed communication libraries and frameworks. Usability and performance will be discussed separately in the following. They are often conflicting goals, and a potential user needs to weigh one against the other.

6.1.1. Usability

We look at usability from the user's perspective, i.e. from the perspective of the person or persons who have a particular problem which is to be solved using a distributed system. The typical user isn't interested in the inner workings of a particular library or framework; what is important for the user is that it does its job with as little work as possible required on the user's part, while at the same time providing adequate performance and flexibility and accurate results.

A typical user of a distributed system used for numerical computing has a strong mathematical background, but poor or mediocre programming skills. Therefore, it becomes important that the framework or library is easy to learn and doesn't bury the user in intricate details.

In table 5.1, we have tried to assess the learning time of the different libraries and frameworks. As can be seen, the Framework has the shortest learning time, followed by MPI and OOMPI, and lastly comes CORBA. We have based these approximations on the following facts:

- The Framework shields the user from most of the parallel computing issues.
- In the Framework, the user only has to deal with three things:
  - Subclass UserAlg
  - Subclass UserData
  - Call Update() at predefined times.

- The MPI and OOMPI libraries have a lot of functions available, but only four or five is required in order to get started.

- OOMPI offers an object oriented programming style and provides default arguments for many method calls. This makes code more readable.
• When using CORBA, one has to learn a new language (IDL) for specifying interfaces.

• The CORBA programming style, while object-oriented, can be very convoluted. The header and code files created by the IDL compiler usually are practically illegible.

• Implementing CORBA remote objects on the server side can be difficult for a user with no O.O. middleware experience.

The Framework also comes out on top when one looks at the number of lines of code required for solving the area computation problem. At 74 lines, it required only a tenth of the amount of code required for MPI and OOMPI, while the CORBA implementation required nearly 1000 lines of code. Much of the code for the MPI, OOMPI and CORBA implementations could be shared between implementations, and it wasn’t difficult to change the program to use a new method of communication, but having to write nearly 1000 lines of code in order to solve a simple problem is a formidable task.

The average number of lines of code per class was about the same for the MPI, OOMPI and CORBA versions. This was expected, because there is a length limit over which the classes become too hard to understand and maintain.

The Framework also had the fewest number of parallel decision points - only one. At the current phase of development, the Framework has no provisions for collecting the results, and therefore, an explicit OOMPI call was required for transferring the result\(^1\). CORBA, MPI and OOMPI required that the user synchronized the tasks and transferred all information explicitly. The only reason that the CORBA version have fewer parallel decision points, was that to our knowledge, CORBA does not have any easy ways of synchronizing processes. This made the CORBA version more susceptible to failures caused by the processes getting out of sync.

6.1.2. Performance

From the results, we can see that MPI seems to have the highest overall performance, with OOMPI not far behind. The Framework does well compared to CORBA, and while the CORBA version for the implementation performs better than the Framework version on larger files like small.dat, it is slower than the Framework and has the lowest performance for the tiny.dat file.

Average Execution Time

The average time for one execution gives an indication of the performance of a particular implementation for a given combination of number of processors and input file. Based on the execution time we are able to calculate speed-up and efficiency (defined in section 3.5.5).

\(^{1}\)The author informs us that the Framework will have a method for collecting the results in the future.
The trend is that the MPI implementation looks like the fastest, with OOMPI close behind. CORBA and the Framework does comparatively well, and has about the same performance.

The standard deviation (included in the detailed results in appendix A.2) shows that the differences from run to run are small. We have excluded one of the CORBA runs from the result sets because the process froze, but other than that all results are included. If there were larger fluctuations we would assume those to be caused by changing network or CPU load caused by external factors, as the computation process itself is fairly deterministic.

**Speed-up and efficiency**

CORBA is the only implementation that has a poor increase in speed-up with tiny.dat. Its efficiency drops significantly here (18.5% from 4 to 5 processors and from 5 to 6), and we would say the scalability for that file is poor. The increase from 3 to 4 processors with CORBA seems okay though. The three other implementations do well here. With small.dat all four implementations perform well, even CORBA, so it we assume that as long as the problem is big enough, CORBA is scaleable. The larger file small.dat really prove the champion in performance. MPI does very well here, especially at 6 processors. CORBA and the framework seems to have an efficiency top at 4 processors, while it seems like OOMPI has its top at 5. MPI's efficiency seem to flat out at 5 and 6, and it would be interesting to see how it would perform at 7 processors, the guess is it would decrease slightly. The framework behaves very much like CORBA on the small.dat file, though slightly behind. It looks like MPI and OOMPI has a smoother curve than CORBA and the Framework on efficiency, but it is hard to tell since we only has 6 processors to test on. It would be really interesting to test on 10 or maybe 12 processors and perhaps see if the efficiency curve was irregular or if there would be a smooth curve with a efficiency top for any of the implementation types.

### 6.2. Validation

Our hypothesis 3.5.1 predicted that although the framework would be somewhat slower than the other implementations, it has much greater usability. As we can see from our usability results, using the Framework involves much less work than making an MPI, OOMPI or CORBA implementation. From a user’s perspective, all that is needed is to subclass the `UserAlg` class, and one does not have to think about the parallelization aspects of the problem. It was significantly harder to implement the solution using MPI and OOMPI, because parallelization needs to be handled explicitly in the program. The CORBA implementation was the hardest to write, and it required the highest amount of code, as can be seen in figure 5.1.

When it comes to performance the Framework performed the worst, as expected (see figures 5.1-5.6), but we think that the performance loss is not enough to outweigh the usability benefits of the Framework. Compared to CORBA, it had about the same performance.
In our opinion, the results of the usability and performance evaluation support our hypothesis.

6.3. Suggested improvements

The toy problem that we have selected is not a real-world problem. Its primary shortcoming is that it does not require an update phase (see 3.2), which means that once the data has been distributed, no further communication is needed before the result is calculated. Communication need only happen between the master process and worker processes, and not between two master processes. With an update phase, worker processes would need to exchange data with each other, and this would create problems with the CORBA implementation since it is squarely built upon a Client/Server programming paradigm and clients cannot easily communicate with each other.

To solve this, worker processes need to be both clients and servers, and this would require that each worker process contains two threads of execution: one that performs the computation, as usual, and one that has the role of server and listens for method invocations from other workers. This would create additional complexity in the CORBA implementation.

Other improvements include testing with larger datafiles to really test how the Framework would act in a more realistic situation. This test should involve a larger number of processors as well, for example 3 to 12 processors, and this would give us a more complete image of how the efficiency would change as the number of processors increased. It could also be interesting to test the implementations on another hardware platform, to see how performance and scalability is affected.

It could also be useful to measure memory usage, memory leakage, communication, and the amount of data transmitted. This would give a more complete picture of how the different communication libraries, and the Framework, performs.
7. Conclusion

7.1. Conclusion

In this report, we have looked at three libraries; the message-passing library MPI, the object-oriented MPI layer OOMPI, and the object-oriented middleware standard CORBA, and we have compared these, with respect to usability and performance, to a framework specially designed for solving PDE's using the finite element method. In order to be able to compare these we have written four different implementations of the area-computation problem described in section 3.5.3, using each of the four approaches. These implementations were tested on a six-processor cluster.

For usability, our results indicated that the Framework was vastly superior to any of the other approaches. The Framework transparently handles all parallelization issues in the application, and the user only has to implement his or her algorithm for solving the problem. Our example problem could be solved with a tenth of the amount of code required for the other approaches.

Performance-wise, the Framework performed notably worse than the MPI and OOMPI versions, but had about the same performance as the CORBA version. The lower performance of the Framework is to be expected, because the Framework does more work "behind the scenes" than the MPI and OOMPI implementations, which are optimized for our simple problem. We expect that the performance difference will be less with a problem that is more representative for the kind of problems that the Framework will be used for.

In our opinion, using the Framework for FEM solvers and similar applications is better than implementing a solution from scratch using lower-level libraries. By using the same code base for several applications, that code will be more general, and better tested and debugged than if the same problems are solved over and over again each time. The extra performance gained from a customized solution is not worth the extra effort, except in the most performance-critical applications.

7.2. Further work

As stated in section 6.3, the toy problem that we have selected is not representative for the kinds of problems numerical distributed systems will be used for. Implementing a FEM solver or similar application would give more realistic results.

Also, we have only considered C++ as the implementation language. With the advancements of modern compilers, C++ has become a good alternative for use in numerical
computation software, but other languages, such as Java, FORTRAN and C should be considered.

We have covered CORBA, MPI, OOMPI, and the OODFEM Framework in this report. Other approaches to creating distributed systems exist, and it could be interesting to test some of them to see how well they perform in comparison. Of particular interest would be to test PVM against MPI, and the distributed shared memory environments (OpenMP, Linda, et al) against message-passing libraries. It could also be interesting to compare the performance of low-level communication methods (raw TCP/IP sockets, for instance) to higher-level communication libraries.
Appendix
A. Results

This appendix contains the results for the performance test. For the cube.dat file, results are missing for the MPI, OOMPI and Framework versions when the number of processors is five, and for the Framework version when the number of processors is six. This is because the programs didn’t work correctly for these combinations of input data and number of processors. We suspect this is due to the small mesh size of the cube.dat file, which caused some processors to be assigned an empty subdomain (i.e. containing zero elements).

A.1. Summary of results

The following tables summarizes the average execution time, the speed up, and the efficiency for all three input files. The average execution time is given in seconds.

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.05</td>
<td>1.02</td>
<td>0.62</td>
<td>0.59</td>
<td>0.62</td>
</tr>
<tr>
<td>4</td>
<td>0.05</td>
<td>0.97</td>
<td>0.57</td>
<td>0.57</td>
<td>0.63</td>
</tr>
<tr>
<td>5</td>
<td>0.05</td>
<td>1.03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.05</td>
<td>0.88</td>
<td>0.59</td>
<td>0.59</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table A.1: Average execution time for the cube.dat

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.69</td>
<td>4.25</td>
<td>4.34</td>
<td>4.34</td>
<td>4.62</td>
</tr>
<tr>
<td>4</td>
<td>8.69</td>
<td>3.32</td>
<td>3.19</td>
<td>3.21</td>
<td>3.32</td>
</tr>
<tr>
<td>5</td>
<td>8.69</td>
<td>3.25</td>
<td>2.55</td>
<td>2.55</td>
<td>2.68</td>
</tr>
<tr>
<td>6</td>
<td>8.69</td>
<td>3.34</td>
<td>2.22</td>
<td>2.23</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Table A.2: Average execution time for the tiny.dat file

A.2. Detailed results

The following tables show the execution times for all runs, in seconds. 'P' in the tables is the number of processors the run was made with, and each run is labeled from 1 to
<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>29.83</td>
<td>12.13</td>
<td>12.51</td>
<td>12.08</td>
<td>13.32</td>
</tr>
<tr>
<td>4</td>
<td>29.83</td>
<td>7.55</td>
<td>7.14</td>
<td>7.38</td>
<td>8.16</td>
</tr>
<tr>
<td>5</td>
<td>29.83</td>
<td>7.03</td>
<td>5.28</td>
<td>5.36</td>
<td>7.73</td>
</tr>
<tr>
<td>6</td>
<td>29.83</td>
<td>5.48</td>
<td>4.42</td>
<td>5.11</td>
<td>6.33</td>
</tr>
</tbody>
</table>

Table A.3.: Average execution time for the small.dat file

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.05</td>
<td>0.08</td>
<td>0.09</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>0.05</td>
<td>0.09</td>
<td>0.09</td>
<td>0.08</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.05</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>0.06</td>
<td>0.09</td>
<td>0.09</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A.4.: Speed-up for the cube.dat file

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>2.04</td>
<td>2.00</td>
<td>2.00</td>
<td>1.88</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>2.62</td>
<td>2.73</td>
<td>2.71</td>
<td>2.62</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>2.67</td>
<td>3.41</td>
<td>3.40</td>
<td>3.24</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>2.60</td>
<td>3.91</td>
<td>3.90</td>
<td>3.69</td>
</tr>
</tbody>
</table>

Table A.5.: Speed-up for the tiny.dat file

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>2.46</td>
<td>2.39</td>
<td>2.47</td>
<td>2.24</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>3.95</td>
<td>4.18</td>
<td>4.04</td>
<td>3.65</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>4.24</td>
<td>5.65</td>
<td>5.57</td>
<td>3.86</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>5.44</td>
<td>6.75</td>
<td>5.83</td>
<td>4.71</td>
</tr>
</tbody>
</table>

Table A.6.: Speed-up for the small.dat file

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.02</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.01</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A.7.: Efficiency for the cube.dat file
<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.68</td>
<td>0.67</td>
<td>0.67</td>
<td>0.63</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>0.65</td>
<td>0.68</td>
<td>0.68</td>
<td>0.65</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.53</td>
<td>0.68</td>
<td>0.68</td>
<td>0.65</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>0.43</td>
<td>0.65</td>
<td>0.65</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table A.8.: Efficiency for the tiny.dat file

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.82</td>
<td>0.80</td>
<td>0.82</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>0.99</td>
<td>1.04</td>
<td>1.01</td>
<td>0.91</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.85</td>
<td>1.13</td>
<td>1.11</td>
<td>0.77</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>0.91</td>
<td>1.13</td>
<td>0.97</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Table A.9.: Efficiency for the small.dat file

20. In addition, the average execution time and the standard deviation is calculated.
<table>
<thead>
<tr>
<th>Ver.</th>
<th>P</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Avg</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>0.06</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>CORBA</td>
<td>3</td>
<td>194.07</td>
<td>1.14</td>
<td>1.12</td>
<td>1.13</td>
<td>1.13</td>
<td>1.13</td>
<td>1.14</td>
<td>1.13</td>
<td>1.13</td>
<td>1.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORBA</td>
<td>4</td>
<td>0.14</td>
<td>1.14</td>
<td>1.14</td>
<td>1.14</td>
<td>1.15</td>
<td>1.15</td>
<td>1.15</td>
<td>1.15</td>
<td>1.14</td>
<td>1.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>1.18</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.23</td>
<td>1.16</td>
<td>1.18</td>
<td>1.17</td>
<td>1.17</td>
<td>1.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORBA</td>
<td>6</td>
<td>1.23</td>
<td>2.65</td>
<td>1.14</td>
<td>1.21</td>
<td>1.13</td>
<td>0.14</td>
<td>0.18</td>
<td>0.15</td>
<td>0.14</td>
<td>1.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>3</td>
<td>0.81</td>
<td>0.56</td>
<td>1.35</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
</tr>
<tr>
<td>MPI</td>
<td>4</td>
<td>0.56</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MPI</td>
<td>6</td>
<td>0.58</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td>OOMPI</td>
<td>3</td>
<td>0.89</td>
<td>0.56</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
</tr>
<tr>
<td>OOMPI</td>
<td>4</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OOMPI</td>
<td>6</td>
<td>0.60</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.60</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td>Framew.</td>
<td>3</td>
<td>0.65</td>
<td>0.60</td>
<td>0.65</td>
<td>0.61</td>
<td>0.63</td>
<td>0.63</td>
<td>0.57</td>
<td>0.57</td>
<td>0.64</td>
<td>0.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Framew.</td>
<td>4</td>
<td>0.59</td>
<td>0.74</td>
<td>0.58</td>
<td>0.59</td>
<td>0.62</td>
<td>0.64</td>
<td>0.62</td>
<td>0.58</td>
<td>0.65</td>
<td>0.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Framew.</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Framew.</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A.10: Detailed results for the cube.dat file
<table>
<thead>
<tr>
<th>Ver.</th>
<th>P</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>8.66</td>
<td>8.49</td>
<td>7.97</td>
<td>8.89</td>
<td>9.18</td>
<td>8.71</td>
<td>9.14</td>
<td>9.16</td>
<td>8.56</td>
<td>8.88</td>
</tr>
<tr>
<td>CORBA</td>
<td>3</td>
<td>3.81</td>
<td>3.76</td>
<td>4.80</td>
<td>3.76</td>
<td>3.82</td>
<td>3.79</td>
<td>4.80</td>
<td>3.80</td>
<td>3.88</td>
<td>4.78</td>
</tr>
<tr>
<td>CORBA</td>
<td>4</td>
<td>3.76</td>
<td>2.77</td>
<td>3.83</td>
<td>2.79</td>
<td>3.85</td>
<td>2.80</td>
<td>3.85</td>
<td>2.89</td>
<td>2.81</td>
<td>3.88</td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>3.35</td>
<td>2.52</td>
<td>3.64</td>
<td>2.35</td>
<td>3.53</td>
<td>3.43</td>
<td>3.81</td>
<td>3.20</td>
<td>2.53</td>
<td>3.44</td>
</tr>
<tr>
<td>CORBA</td>
<td>6</td>
<td>3.69</td>
<td>2.94</td>
<td>3.86</td>
<td>3.52</td>
<td>4.04</td>
<td>3.38</td>
<td>2.42</td>
<td>3.49</td>
<td>3.48</td>
<td>3.44</td>
</tr>
<tr>
<td>MPI</td>
<td>3</td>
<td>4.33</td>
<td>4.39</td>
<td>4.30</td>
<td>4.21</td>
<td>4.24</td>
<td>4.75</td>
<td>4.25</td>
<td>4.36</td>
<td>4.27</td>
<td>4.24</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>2.53</td>
<td>2.55</td>
<td>2.54</td>
<td>2.54</td>
<td>2.54</td>
<td>2.59</td>
<td>2.54</td>
<td>2.55</td>
<td>2.54</td>
<td>2.57</td>
</tr>
<tr>
<td>MPI</td>
<td>6</td>
<td>2.22</td>
<td>2.23</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>2.58</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.56</td>
</tr>
<tr>
<td>OOMPI</td>
<td>6</td>
<td>2.22</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
</tr>
<tr>
<td>Framew.</td>
<td>3</td>
<td>4.69</td>
<td>4.76</td>
<td>4.54</td>
<td>4.77</td>
<td>4.49</td>
<td>4.76</td>
<td>4.49</td>
<td>4.77</td>
<td>4.49</td>
<td>4.77</td>
</tr>
<tr>
<td>Framew.</td>
<td>4</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.57</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.33</td>
<td>3.32</td>
</tr>
<tr>
<td>Framew.</td>
<td>5</td>
<td>2.68</td>
<td>2.67</td>
<td>2.67</td>
<td>2.68</td>
<td>2.71</td>
<td>2.67</td>
<td>2.67</td>
<td>2.68</td>
<td>2.67</td>
<td>2.70</td>
</tr>
<tr>
<td>Framew.</td>
<td>6</td>
<td>2.36</td>
<td>2.35</td>
<td>2.35</td>
<td>2.35</td>
<td>2.35</td>
<td>2.34</td>
<td>2.34</td>
<td>2.35</td>
<td>2.41</td>
<td>2.34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver.</th>
<th>P</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>Avg.</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>7.96</td>
<td>8.71</td>
<td>8.93</td>
<td>8.56</td>
<td>7.93</td>
<td>8.92</td>
<td>8.93</td>
<td>7.88</td>
<td>9.66</td>
<td>8.68</td>
<td>8.69</td>
<td>0.46</td>
</tr>
<tr>
<td>CORBA</td>
<td>3</td>
<td>3.96</td>
<td>4.75</td>
<td>4.85</td>
<td>4.74</td>
<td>4.75</td>
<td>4.75</td>
<td>3.79</td>
<td>3.78</td>
<td>3.82</td>
<td>4.80</td>
<td>4.25</td>
<td>0.48</td>
</tr>
<tr>
<td>CORBA</td>
<td>4</td>
<td>2.81</td>
<td>3.80</td>
<td>2.83</td>
<td>3.79</td>
<td>2.78</td>
<td>2.80</td>
<td>3.80</td>
<td>2.84</td>
<td>3.89</td>
<td>3.82</td>
<td>3.32</td>
<td>0.51</td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>3.12</td>
<td>2.50</td>
<td>3.51</td>
<td>3.43</td>
<td>3.33</td>
<td>3.45</td>
<td>3.47</td>
<td>4.06</td>
<td>3.00</td>
<td>3.36</td>
<td>3.25</td>
<td>0.45</td>
</tr>
<tr>
<td>CORBA</td>
<td>6</td>
<td>3.41</td>
<td>3.30</td>
<td>3.33</td>
<td>3.26</td>
<td>3.32</td>
<td>2.88</td>
<td>3.33</td>
<td>3.29</td>
<td>3.39</td>
<td>2.95</td>
<td>3.34</td>
<td>0.34</td>
</tr>
<tr>
<td>MPI</td>
<td>3</td>
<td>4.25</td>
<td>4.55</td>
<td>4.26</td>
<td>4.53</td>
<td>4.25</td>
<td>4.31</td>
<td>4.29</td>
<td>4.22</td>
<td>4.44</td>
<td>4.35</td>
<td>4.34</td>
<td>0.13</td>
</tr>
<tr>
<td>MPI</td>
<td>4</td>
<td>3.18</td>
<td>3.17</td>
<td>3.17</td>
<td>3.17</td>
<td>3.16</td>
<td>3.16</td>
<td>3.17</td>
<td>3.46</td>
<td>3.17</td>
<td>3.17</td>
<td>3.19</td>
<td>0.06</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>2.59</td>
<td>2.54</td>
<td>2.54</td>
<td>2.55</td>
<td>2.54</td>
<td>2.54</td>
<td>2.54</td>
<td>2.54</td>
<td>2.53</td>
<td>2.53</td>
<td>2.55</td>
<td>0.02</td>
</tr>
<tr>
<td>MPI</td>
<td>6</td>
<td>2.23</td>
<td>2.21</td>
<td>2.22</td>
<td>2.22</td>
<td>2.23</td>
<td>2.25</td>
<td>2.22</td>
<td>2.21</td>
<td>2.22</td>
<td>2.23</td>
<td>2.22</td>
<td>0.01</td>
</tr>
<tr>
<td>OOMPI</td>
<td>3</td>
<td>4.23</td>
<td>4.45</td>
<td>4.72</td>
<td>4.26</td>
<td>4.99</td>
<td>4.33</td>
<td>4.21</td>
<td>4.25</td>
<td>4.31</td>
<td>4.25</td>
<td>4.34</td>
<td>0.19</td>
</tr>
<tr>
<td>OOMPI</td>
<td>4</td>
<td>3.17</td>
<td>3.17</td>
<td>3.17</td>
<td>3.17</td>
<td>3.37</td>
<td>3.32</td>
<td>3.20</td>
<td>3.51</td>
<td>3.16</td>
<td>3.17</td>
<td>3.21</td>
<td>0.09</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.56</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.56</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>0.01</td>
</tr>
<tr>
<td>OOMPI</td>
<td>6</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.24</td>
<td>2.23</td>
<td>2.23</td>
<td>2.23</td>
<td>2.22</td>
<td>2.22</td>
<td>2.23</td>
<td>0.01</td>
</tr>
<tr>
<td>Framew.</td>
<td>3</td>
<td>4.50</td>
<td>5.01</td>
<td>4.49</td>
<td>4.58</td>
<td>4.52</td>
<td>4.57</td>
<td>4.52</td>
<td>4.56</td>
<td>4.50</td>
<td>4.57</td>
<td>4.62</td>
<td>0.14</td>
</tr>
<tr>
<td>Framew.</td>
<td>4</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.30</td>
<td>3.32</td>
<td>0.06</td>
</tr>
<tr>
<td>Framew.</td>
<td>5</td>
<td>2.68</td>
<td>2.68</td>
<td>2.68</td>
<td>2.68</td>
<td>2.67</td>
<td>2.71</td>
<td>2.67</td>
<td>2.68</td>
<td>2.68</td>
<td>2.75</td>
<td>2.68</td>
<td>0.02</td>
</tr>
<tr>
<td>Framew.</td>
<td>6</td>
<td>2.43</td>
<td>2.37</td>
<td>2.34</td>
<td>2.36</td>
<td>2.35</td>
<td>2.35</td>
<td>2.35</td>
<td>2.35</td>
<td>2.34</td>
<td>2.34</td>
<td>2.36</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table A.11.: Detailed results for the tiny.dat file
<table>
<thead>
<tr>
<th>Ver.</th>
<th>P</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>29.74</td>
<td>29.97</td>
<td>29.59</td>
<td>30.52</td>
<td>29.57</td>
<td>29.81</td>
<td>29.49</td>
<td>29.57</td>
<td>30.10</td>
<td>30.20</td>
</tr>
<tr>
<td>CORBA</td>
<td>3</td>
<td>12.72</td>
<td>11.11</td>
<td>12.20</td>
<td>11.39</td>
<td>12.63</td>
<td>11.85</td>
<td>12.59</td>
<td>12.08</td>
<td>11.18</td>
<td>13.84</td>
</tr>
<tr>
<td>CORBA</td>
<td>4</td>
<td>7.47</td>
<td>6.84</td>
<td>7.36</td>
<td>8.28</td>
<td>7.83</td>
<td>7.42</td>
<td>9.00</td>
<td>7.46</td>
<td>7.64</td>
<td>6.53</td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>6.27</td>
<td>7.73</td>
<td>7.39</td>
<td>7.23</td>
<td>6.55</td>
<td>6.32</td>
<td>6.20</td>
<td>7.23</td>
<td>9.52</td>
<td>7.70</td>
</tr>
<tr>
<td>CORBA</td>
<td>6</td>
<td>5.57</td>
<td>6.45</td>
<td>5.32</td>
<td>6.17</td>
<td>6.26</td>
<td>6.22</td>
<td>3.76</td>
<td>5.03</td>
<td>6.46</td>
<td>4.92</td>
</tr>
<tr>
<td>MPI</td>
<td>4</td>
<td>7.47</td>
<td>7.00</td>
<td>7.87</td>
<td>6.38</td>
<td>6.61</td>
<td>7.39</td>
<td>7.58</td>
<td>6.64</td>
<td>7.00</td>
<td>6.53</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>5.22</td>
<td>5.47</td>
<td>5.47</td>
<td>5.49</td>
<td>5.03</td>
<td>4.99</td>
<td>4.88</td>
<td>4.94</td>
<td>5.47</td>
<td>5.23</td>
</tr>
<tr>
<td>OOMPI</td>
<td>3</td>
<td>11.90</td>
<td>11.75</td>
<td>11.77</td>
<td>12.66</td>
<td>12.33</td>
<td>11.79</td>
<td>12.26</td>
<td>12.60</td>
<td>11.78</td>
<td>11.81</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>4.98</td>
<td>6.52</td>
<td>5.08</td>
<td>5.12</td>
<td>5.47</td>
<td>5.47</td>
<td>5.73</td>
<td>5.25</td>
<td>5.26</td>
<td>5.41</td>
</tr>
<tr>
<td>OOMPI</td>
<td>6</td>
<td>4.36</td>
<td>4.25</td>
<td>4.30</td>
<td>4.25</td>
<td>4.16</td>
<td>4.15</td>
<td>4.32</td>
<td>5.57</td>
<td>5.67</td>
<td>5.76</td>
</tr>
<tr>
<td>Framew.</td>
<td>4</td>
<td>9.10</td>
<td>8.16</td>
<td>7.15</td>
<td>8.20</td>
<td>8.23</td>
<td>9.45</td>
<td>7.67</td>
<td>9.54</td>
<td>7.57</td>
<td>7.17</td>
</tr>
<tr>
<td>Framew.</td>
<td>5</td>
<td>7.66</td>
<td>7.80</td>
<td>-</td>
<td>7.75</td>
<td>7.72</td>
<td>7.61</td>
<td>7.78</td>
<td>7.69</td>
<td>8.35</td>
<td>7.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ver.</th>
<th>P</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>Avg.</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>29.65</td>
<td>29.64</td>
<td>30.22</td>
<td>29.82</td>
<td>29.88</td>
<td>29.65</td>
<td>30.11</td>
<td>30.04</td>
<td>29.67</td>
<td>30.34</td>
<td>29.83</td>
<td>0.35</td>
</tr>
<tr>
<td>CORBA</td>
<td>4</td>
<td>7.46</td>
<td>7.15</td>
<td>7.30</td>
<td>7.56</td>
<td>8.00</td>
<td>7.19</td>
<td>7.89</td>
<td>6.46</td>
<td>9.03</td>
<td>7.19</td>
<td>7.55</td>
<td>0.65</td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>6.69</td>
<td>6.80</td>
<td>8.16</td>
<td>7.54</td>
<td>6.80</td>
<td>6.56</td>
<td>6.64</td>
<td>6.65</td>
<td>6.26</td>
<td>6.38</td>
<td>7.03</td>
<td>0.80</td>
</tr>
<tr>
<td>CORBA</td>
<td>6</td>
<td>3.90</td>
<td>4.97</td>
<td>4.22</td>
<td>3.83</td>
<td>5.00</td>
<td>5.32</td>
<td>6.58</td>
<td>6.12</td>
<td>6.45</td>
<td>5.02</td>
<td>5.48</td>
<td>0.84</td>
</tr>
<tr>
<td>MPI</td>
<td>3</td>
<td>11.42</td>
<td>12.34</td>
<td>11.43</td>
<td>12.37</td>
<td>11.36</td>
<td>12.07</td>
<td>11.82</td>
<td>12.34</td>
<td>11.99</td>
<td>12.05</td>
<td>12.51</td>
<td>1.08</td>
</tr>
<tr>
<td>MPI</td>
<td>4</td>
<td>6.49</td>
<td>7.00</td>
<td>7.75</td>
<td>7.05</td>
<td>6.99</td>
<td>7.04</td>
<td>7.37</td>
<td>7.44</td>
<td>7.65</td>
<td>7.50</td>
<td>7.14</td>
<td>0.44</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>5.00</td>
<td>5.03</td>
<td>5.05</td>
<td>5.93</td>
<td>5.91</td>
<td>5.40</td>
<td>5.42</td>
<td>5.55</td>
<td>4.87</td>
<td>5.20</td>
<td>5.28</td>
<td>0.31</td>
</tr>
<tr>
<td>MPI</td>
<td>6</td>
<td>4.21</td>
<td>4.16</td>
<td>4.34</td>
<td>4.25</td>
<td>4.67</td>
<td>4.67</td>
<td>4.76</td>
<td>4.64</td>
<td>4.11</td>
<td>4.28</td>
<td>4.42</td>
<td>0.33</td>
</tr>
<tr>
<td>OOMPI</td>
<td>3</td>
<td>11.87</td>
<td>12.21</td>
<td>11.70</td>
<td>12.01</td>
<td>11.72</td>
<td>12.00</td>
<td>12.13</td>
<td>12.58</td>
<td>12.41</td>
<td>12.23</td>
<td>12.08</td>
<td>0.31</td>
</tr>
<tr>
<td>OOMPI</td>
<td>4</td>
<td>7.20</td>
<td>7.06</td>
<td>7.32</td>
<td>7.02</td>
<td>8.33</td>
<td>7.87</td>
<td>7.21</td>
<td>8.42</td>
<td>7.20</td>
<td>7.12</td>
<td>7.38</td>
<td>0.51</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>5.21</td>
<td>5.06</td>
<td>5.15</td>
<td>5.24</td>
<td>5.00</td>
<td>5.31</td>
<td>6.01</td>
<td>5.09</td>
<td>5.68</td>
<td>5.06</td>
<td>5.36</td>
<td>0.37</td>
</tr>
<tr>
<td>OOMPI</td>
<td>6</td>
<td>4.90</td>
<td>5.60</td>
<td>5.53</td>
<td>5.99</td>
<td>5.63</td>
<td>5.49</td>
<td>5.56</td>
<td>5.50</td>
<td>5.58</td>
<td>5.71</td>
<td>5.11</td>
<td>0.66</td>
</tr>
<tr>
<td>Framew.</td>
<td>3</td>
<td>12.73</td>
<td>13.16</td>
<td>12.75</td>
<td>12.43</td>
<td>13.95</td>
<td>13.16</td>
<td>13.41</td>
<td>13.34</td>
<td>13.57</td>
<td>13.69</td>
<td>13.32</td>
<td>0.48</td>
</tr>
<tr>
<td>Framew.</td>
<td>4</td>
<td>7.52</td>
<td>7.63</td>
<td>7.64</td>
<td>7.88</td>
<td>8.61</td>
<td>8.29</td>
<td>7.56</td>
<td>9.53</td>
<td>8.36</td>
<td>8.00</td>
<td>8.16</td>
<td>0.73</td>
</tr>
<tr>
<td>Framew.</td>
<td>5</td>
<td>7.80</td>
<td>7.59</td>
<td>7.75</td>
<td>7.66</td>
<td>7.48</td>
<td>7.50</td>
<td>7.58</td>
<td>8.04</td>
<td>7.69</td>
<td>7.59</td>
<td>7.73</td>
<td>0.19</td>
</tr>
<tr>
<td>Framew.</td>
<td>6</td>
<td>6.43</td>
<td>6.59</td>
<td>6.37</td>
<td>6.37</td>
<td>6.97</td>
<td>6.10</td>
<td>6.71</td>
<td>6.09</td>
<td>6.39</td>
<td>6.16</td>
<td>6.33</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Table A.12: Detailed results for the small.dat file
B. Sample code

B.1. GetSubDomain(): Prepare data for sending

The GetSubDomain() method, listed below, prepares data for sending. It uses as input data

- the elements, faces and vertices arrays which contain the mesh data
- the partition array which contains a mapping from elements to process ID’s and were obtained through the use of the GetPartition() method (taken from the framework, not listed in this document)
- the process ID of the process that is to receive the data

The method returns a pointer to a MeshStruct object which will contain the data that is to be sent to the target process. This object is dynamically allocated (using the new operator), and must be deleted after use.

```c
MeshStruct * Master::GetSubDomain(int pid) {  
    const int nf_e = 4; // Number of faces per element  
    const int nv_f = 3; // Number of vertices per face  
    const int nd_v = 3; // Number of coordinates per vertex  
    const int nv_e = 4; // Number of vertices per element  

    const int e_s = nf_e*2+2; // No. of entries per element in elements array  
    const int f_s = nv_f+1; // No. of entries per face in faces array  
    const int v_s = nd_v+1; // No. of entries per vertex in vertices array  

    MeshStruct *ms = new MeshStruct();

    int *elems = GetElems();  
    int *faces = GetFaces();  
    double *verts = GetVerts();  
    int *part = GetPart();

    int ne = GetNElems();  
    int nf = GetNfaces();  
    int nv = GetNVerts();
```
int i;

// Allocate space for subdomain data.
// To be sure, space allocated is enough to hold complete domain data.
// As the variables are not cleared (we use the new operator instead of
// the AllocInit_* functions), smart operating systems, such as Linux,
// can defer the actual allocation of memory pages to when memory is
// accessed the first time.
int *subd elems = new int[e_s*ne];
int *subd_faces = new int[f_s*nf];
double *subd verts = new double[v_s*nv];

int *globVidx = new int[ne]; for(i=0; i<ne;i++) globVidx[i]=-1;
int *globFidx = new int[nf]; for(i=0; i<nf;i++) globFidx[i]=-1;

int subd_ne = 0;
int subd_nf = 0;
int subd_nv = 0;

int e,f,v,d;
for(e=0; e<ne; e++) {
  if(part[e]==pid) {
    // This element needs to be part of the sub-domain
    // Start by copying the vertices
    for(f=0; f<nf_e; f++) {
      int face = elems[e_s*e+f];
      if(globFidx[face]==-1) {
        // face has not been copied earlier
        for(v=0; v<nf_f; v++) {
          int vertex = faces[f_s*face+v];
          if(globVidx[vertex]==-1) {
            // vertex has not been copied earlier
            for(d=0; d<v_s; d++) {
              subd_verts[subd_nv*v_s+d] = verts[vertex*v_s+d];
            }
            subd_faces[subd_nf*f_s+v] = subd_nv;
            globVidx[vertex]=subd_nv;
            subd_nv++;
          } else {
            // vertex has been copied
            int subd_vertex = globVidx[vertex];
            subd_faces[subd_nf*f_s+v] = subd_vertex;
          }
        }
      }
    }
  }
}
B.2. triangle_area(): Calculate triangle area

The triangle_area() function calculates the area of a single triangular face using the cross product. The function uses as input arguments three one-dimensional arrays with three elements each. Each array corresponds to one of the three vertices in the triangle, and the array elements contain the x, y and z coordinates of the vertices. The function returns the area of the triangle.

```c
double triangle_area(double p1[3], double p2[3], double p3[3])
{
    int i;
    double v1[3]; // vector one
    double v2[3]; // vector two
    double cp[3]; // cross product
    for(int i=0; i<3; i++) {
        v1[i] = p2[i] - p1[i];
        v2[i] = p3[i] - p1[i];
```
B.3. calculate_area(): Calculate total area of the mesh

The calculate_area() function calculates the total surface area of a (part of) the mesh. It takes as input argument a pointer to a MeshStruct object which contain the element data, and returns the total area of the mesh. It uses the triangle_area() function internally.

double calculate_area(MeshStruct *m) {
    int nel = m->GetNElems();
    int *elems = m->GetElems();
    int *faces = m->GetFaces();
    double *verts = m->GetVerts();
    double area = 0;
    for(int i=0; i<nel; i++) {
        int *f = &elems[i*10]; // f[0] = index of face 0,
        // f[1] = index of face 1, etc.
        int *n = &elems[i*10+4]; // n[0] = index of neibg 0,
        // n[1] = index og neibg 1
        for(int j=0; j<4; j++) {
            if(n[j]==-1) {
                // no neighbour
                int *fv = &faces[f[j]*4];
                // fv[0] = index of first vertex, etc....
                double *p1 = &verts[fv[0]*4];
                double *p2 = &verts[fv[1]*4];
                double *p3 = &verts[fv[2]*4];
                area += triangle_area(p1,p2,p3);
            }
        }
    }
    return area;
}
C. Testing scripts and commands

C.1. Line count

The following command line was run in order to determine the number of lines of code for a particular code file:

```
cat <filename>
   | grep -v '^[[:space:]]*$'
   | grep -v '^[[:space:]]*//'
   | wc -l
```

The first `grep` command filters out single-line comments (lines starting with `//`, excluding whitespace), while the second command filters out blank lines, or lines containing only whitespace.

C.2. Testing script

The following script was used to run all test runs on the cluster.

```
COMMAND="repeat 20 time --portability"
# Serial version
echo "Serial version"
function proclist
{
   case $1 in
      3)
         echo c1,3,5
            ;;
      4)
         echo c1,2,3,5
            ;;
      5)
         echo c1,2,3,4,5
            ;;
      6)
         echo c1,2,3,4,5,6
            ;;
```
esac
}

function try
{
  echo "$@
$@
}

for NAME in cube tiny small; do
  echo "$NAME.dat"
  try $COMMAND lamexec n1 src/area_serial/task \ 
  src/data/$NAME.dat 2>results/exec_serial_$NAME
done
echo "CORBA version"
for NAME in cube tiny small; do
  for PROCS in 3 4 5 6; do
    echo "$NAME - $PROCS processors"
    try $COMMAND lamexec appschemas/$NAME.$PROCS \ 
    2>results/exec_CORBA_${NAME}_${PROCS}
  done
done
for VERSION in MPI OOMPI Framework; do
echo "$VERSION version"
  for NAME in cube tiny small; do
    for PROCS in 3 4 5 6; do
      echo "$NAME - $PROCS processors"
      try $COMMAND $HOME/bin/mpiexec 'proclist $PROCS' \ 
      src/area_$VERSION/task src/data/$NAME.dat \ 
      2>results/exec_${VERSION}_${NAME}_${PROCS}
    done
  done
done

done
Glossary of abbreviations

**COM** Component Object Model, Microsoft’s object model for the Windows family of operating systems.

**CORBA** Common Object Request Broker Architecture, an open standard allowing for the creation of distributed objects which can interact with one another even if they are written in different languages, running on multiple computers, running different operating systems on different hardware.

**DCOM** Distributed Component Object Model, a distributed extension to COM. See: COM.

**FEM** Finite Element Method, numerical method used for solving PDE’s by dividing the problem domain into a set of discrete elements and performing approximations for each element.

**IDL** Interface Definition Language, declarative language used for specifying interfaces in DCOM and CORBA.

**IPC** Inter-process communication, methods for communication between different computer processes, possibly located on different computers.

**Java RMI** Java Remote Method Invocation, method of remote object method call in Java. Offers functionality similar to that of DCOM and CORBA.

**MPI** Message-Passing Interface, standard for message-passing in scientific computing.

**MPMD** Multiple Programs, Multiple Data, a parallel system in which the processes run different programs on different sections of the data

**MPP** Massively Parallel Processor; special-purpose multiprocessor computer with many processors.

**NOW** Network of Workstations; a network of normal workstations forming a distributed system.

**OODFEM** Object Oriented Distributed FEM, a framework for creating distributed FEM solvers

**OOMPI** Object-oriented MPI, a class library that forms a thin layer above MPI and gives a more natural C++ interface to MPI.
PDE Partial Differential Equation, an equation involving the partial derivatives of one or more unknowns.

PVM Parallel Virtual Machine, a message passing library that creates an illusion of a single virtual machine.

RMI See Java RMI

RPC Remote Procedure Call

SPMD Single Program, Multiple Data, a parallel system in which all processes run the same program on different sections of the data.
Bibliography


[23] R. Diaconescu, R. Conradi: A Component-Based Framework for the Distributed FEM Solution of PDEs, Department of Computer and Information Science, Norwegian University of Science and Technology.