KTH VETENSKAP OCH KONST

KTH Microelectronics and Information Technology

amadeus

Master Thesis Project

Task Distribution and Monitoring in Distributed Computing

Leo Tingvall (tingvall@kth.se)

June 11th 2003

Academic advisor & examiner:	Vladimir Vlassov Department of Microelectronics and Information Technology Royal Institute of Technology Sweden
Industrial advisor:	Alexis Grandemange Amadeus s.a.s. France

Abstract

The need of computing resources is constantly increasing. This has been a driving force behind the speed improvement of computer hardware that is now closer than ever to the limits set by the laws of physics. Distributed computing has been developed to improve the performance of computer systems by distributing the computations, and lately this has attracted interest since it promises high-performance computing at a lower cost than traditional high-end computer systems. Grid computing attempts to provide a software application framework to computing services.

This project investigates the use of distributed computing for task distribution. We implement a prototype in Linux to provide a single point-of-entry to a cluster of 4 computers that run a problem solving application. Requests are distributed on the cluster using the Message Passing Interface. We also implement a prototype using non-blocking sockets and multiple communication buffers that uses resource availability measurements provided by the Network Weather Service monitoring system. The availability information is used to set weights in a weighted round robin distribution algorithm.

We conclude that the technology is mature and usable for suitable applications. We anticipate further developments in the area of grid services that will provide a higher degree of transparency, functionality and usability of grid resources.

Acknowledgements

This project is my Master’s thesis at the Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden. The project was performed at Amadeus in Nice, France during six months in 2002.

Alexis Grandemange was industrial advisor and provided excellent guidance and information throughout the project. Guglielmo Guastalla gave valuable advice and comments. Vladimir Vlassov was academic advisor. I especially thank the three persons mentioned for their assistance, and I express my gratitude to everyone at Amadeus and KTH who made the project possible.

Table of Contents

Table of Figures

1 Introduction

The development of new applications and increasing requirement of information processing has lead to a constant lack of computing resources. This has been one of the driving forces for the computer and semi-conductors industries and so far the rate of growth of chip and computer speed has kept a steady pace. The speed increase is however accompanied with some problems. Even though circuits are built with technology that is many times more efficient than to older models, the increased performance has lead to higher power consumption, increased heat and power loss and constant struggles with the laws of physics.

Computers with the latest technology have very good performance capabilities when it comes to processor, memory and I/O devices, but another major development has been in the area of networking technologies. Fast network communication has become affordable and widespread.

Distributed computing is the area of computer science that deals with development of applications that run in parallel on multiple processors or computers taking advantage of the increased computational power of each processor. The access to high performance computers and networking hardware has opened up for the extensive use of distributed computing in many areas. The roots of distributed computing lie in the scientific community where high performance was often required at any cost forcing development of new ideas, but distributed computing technology is increasingly interesting and has become accepted in other business areas where it promises cheap, high-performance and fault-tolerant computer systems.

1.1 Topics covered

This project intends to investigate and study distributed computing mainly in the areas of task distribution and monitoring.

1.1.1 Task Distribution

According to www.dictionary.com a “task” can be defined as “A piece of work assigned or done as part of one's duties.” In a distributed computer system a task can be just about anything such as running a database query, analyzing some data or even starting an application. The type of tasks the system should be able to handle must be defined when the system is designed.

The object of the “task distribution” in a distributed system is to assign each task to one (or more) of the nodes in the system that is to execute the task. This should be done (if possible) following an algorithm that enables each task to be executed properly according to its requirements, while respecting the system constraints. A task requirement can for example be to make sure that a task is performed within a certain time limit. System constraints can for example be the number of processors in the system and their speed, which limits how fast tasks can be executed. The task distribution should try to optimize some function, for example minimizing the total execution time or maximizing the resource usage.

This project considers task distribution in the context of distributed computing. We intend to investigate and construct a system that performs task distribution. The task distribution algorithm used in such a system is very important since it responsible for properly using the resources available.

1.1.2 Monitoring

Monitoring a system is essential in order to understand how it behaves. A distributed computer system involves a large number of objects that constantly perform tasks related to networking, processing, memory and running applications. A “monitoring system” should allow the monitoring of some of these parameters, not necessarily by performing measurements itself but rather by using the available system services such as operating system information and providing a framework for collection and distribution of the measurements. The monitoring is vital for a distributed computer system since it is the only way to observe the system and validate its functionality.

The monitoring information of the system should be accessible from more than one single point in the system, and the information should be distributed efficiently with as little penalty as possible to the system performance. The cost of monitoring that will be allowed and the degree of correctness required will vary between systems. Monitoring has previously been investigated in projects such as PARMON [35], MIST [27], XPVM [39] and NWS [13].

1.2 Literature Used

The interest in distributed computing has increased both in the academic and commercial communities, something that has lead to an increased number of projects and products. Project information can often be found on project websites and in research reports, and these are often recent and good resources. As the interest has increased the number of conferences in the subject has also increased. In 2002 many conferences were held such as the SC2002 and ICS 2002, as well as others listed at [14]. Conference papers are often both current and well written, which makes for good descriptions of projects and recent results.

In the study for this project we used mainly material found in reports from research groups (e.g. conference papers), project publications, articles and books.

1.3 Prerequisites

Major parts of this project concern networking and computing on a software level. To read the report some knowledge in this area is required. Some parts of the project tend to be a bit technical, others are perhaps not that technical but rely on models and background knowledge. The reader should be familiar with networking, software development and computer hardware basics.

Distributed computing and networking causes some problems for the application programmer for example concerning bandwidth constraints, latency and fault tolerance. These issues make distributed computing difficult in theory and practice and it is important to have knowledge and an intuitive understanding of this.

1.4 Structure of the Report

The structure of the report is as follows. The chapter Background describes the background of the project, introduces different topics related to the project and describes a number of related previous projects. In the chapter Method we describe the goals of the project and the tasks we intend to perform. The following chapters Implementation, Analysis and Results & Discussion describe the prototype developed, the analysis performed and the results of the analysis. Finally in the chapter Conclusions we discuss some general results and reflections, and hint about topics that might be of interest for further investigations.

2 Background

Although distributed computing has been used and researched during a long time – projects started as soon as computers arrived – the environment has changed during the past few years. One of the major changes is that computers become a commodity where even cheaper components provide good performance, as compared to the rare and expensive computer systems previously available.

There are still some obstacles to be crossed before the vision of distributed computing can be realized. Increased bandwidth in combination with programming languages and components that allow for efficient application development will improve the situation. Some of the obstacles will however remain for example because of physical limits and algorithmic complexity.

In this section we describe some basics of distributed computing to make the reader acquainted with the technology. After some basics we describe some projects, tools and concepts that are related to our project. We also give background information regarding the problems we intend to investigate and describe related problems.

2.1 Distributed Computing

Distributed computing is a term often used to describe programs or computations that run on several processors or computers simultaneously. Often the idea is that some sort of cooperation should take place between the processors or computers. To make this possible there have to be some kind of communication system, often something like a memory if we consider a system of processors or a computer network if we consider a system of computers. Lately the term “Grid computing” has been spread and it has gained wide acceptance as a term representing a concept. Although the “distributed computing” and “grid computing” have a very similar meaning we consider grid computing to be a somewhat more narrow definition than distributed computing.

Throughout this report we may alternate between the terms “cluster”, “grid” and “grid farm”. At a quick glance the terms have about the same meaning: a set of interconnected computers that are used to solve problems cooperatively. A “cluster” is often referred to as a set of computers specifically installed for the purpose of performing computations, often with a dedicated network connecting them. The term “grid” is associated to grid computing and can be seen as a bit more focused on the application framework that the system is installed with. A “grid farm” is thus also often used in the context of grid computing, but in this report we consider a grid farm to be the hardware part of the system, the same as with the cluster term.

2.1.1 Grid Computing

One result of the popularization of distributed computing is the introduction of the concept “Grid computing” (see for example [8]). While distributed computing is more or less a term used to describe applications that work together to solve a specific problem or set of problems, grid computing is a bit more aggressive and revolutionary.

One definition of grid computing can be found at [18]:

“Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across "multiple" administrative domains based on their (resources) availability, capability, performance, cost, and users' quality-of-service requirements.“

Further, at [10]:

“The term ‘grid’ is an analogy to the electric power grid, where one can plug in to any wall plug and get electricity from the collective generators in the power grid, …”

The idea of grid computing is that as computers become widespread and network capacity increases, “the Grid” can be considered an asset of resources or a service that can be used as a commodity. The term does not just include a specific type of application; it is the definition of a system that provides a service. The service is usage of computing resources and applications. Consider the analogy of connecting a non-intelligent device (such as a dumb terminal), and with it gaining access to a much more potent computer system. With grid computing the applications that require should also be able to take advantage of being able to run using the resources that the grid provides.

A long way is still to go before we can see this kind of service widely spread. Traditional distributed applications were usually written to be runnable on a specific platform or a specific set up of hardware. By providing an application framework that is portable, different platforms can cooperate in a computation. A system that consists of computers of different architecture is often called a *heterogeneous* system. By providing a portable framework, an application written for “the grid” can take advantage of the hardware resources available. A heterogeneous network will however cause problems, as different architectures might not produce the exact same result to a given code.

The aim is that the usage of the grid resources should be transparent. The system should be able to configure the application and find resources required before starting the computation on the selected resources. This should be done by a system of application often referred to as the Grid Infrastructure. The only thing the developer should have to worry about is making sure the application uses the

Grid Infrastructure and the user should only require access to a suitable grid environment.

One of the reasons grid computing is gaining popularity is that computers and networks are powerful enough to support a grid infrastructure, and that the grid can be a reality within a foreseeable future.

2.1.2 Programming for Distribution

Setting up and using a set of networked computers for running distributed applications is attractive especially for some applications. One reason is that it might be possible to use existing systems, which would provide a system with good performance at a low cost. Applications that use features such as threads or communicating processes might not at all be suitable for distribution. This is because they are often written with smaller granularity and the increased latency and decreased bandwidth will cause performance problems. When designing and implementing a distributed application these factors must be considered.

2.1.3 Programming Paradigm

A computer program consists of a set of instructions that perform different actions depending on the data set it is executed with. The main problem in a distributed system is that access to data, both static data such as a file but also dynamic data such as interaction with other processes or computers, is sometimes very time-consuming. Reading and writing data of this kind must be minimized in order to get decent application performance.

A programming paradigm suitable for programming a distributed application should provide a programming interface that enables the developer to write correct and efficient code. By forcing the programmer to access “expensive” resources as little as possible the application performance can be maintained. Below we describe two widely used paradigms for distributed programming.

Shared Memory

A multiprocessor computer is a computer that has a number of processors (or computation elements) that all share the same memory usually through (at least logically) the same memory bus. In this system a memory read or write is usually fast since it is done over the fast memory bus. If two processors, or processes running on separate processors, wish to communicate with each other they can do this by simply writing to and reading from some defined memory regions. Figure 1 shows an image of a simple shared memory multiprocessor with three processors.

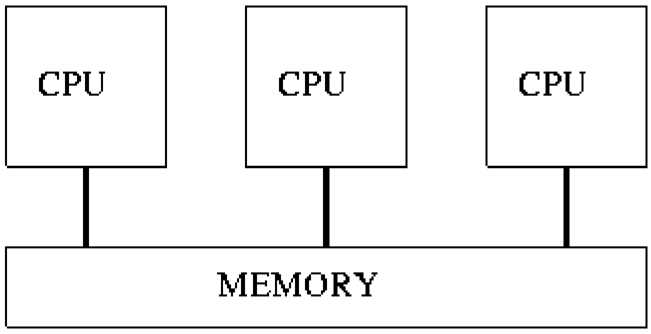

Figure 1. A shared memory multiprocessor.

There are some issues that need to be addressed in a shared memory system. For example some logic has to make sure that two processors do not write to the same memory at the same time since this would produce data inconsistency errors. Modern systems also include a local cache inside each processor. This force the use of cache consistency logic to make sure the local caches are consistent at all times. A multiprocessor system can however handle this locking of memory efficiently because the system is capable of high-speed and low-latency communication through the memory bus.

A shared memory programming paradigm can also be implemented in a distributed memory architecture (see for example [25]) but since it is normally used in an environment where latency and bandwidth is hidden, its use might cause performance problems in a system with higher latency and lower bandwidth.

Message Passing

Message passing is a paradigm that is often used in machines with distributed memory architecture. The inter-process communication is based on passing messages between the processes, and the messages can are transferred either locally trough a memory bus in a multiprocessor system or across a network in a distributed system. Because each message comes with a time-penalty, limiting the number and size of the messages is likely to improve application performance. This will allow the developer to intuitively understand application bottlenecks and circumvent them.

In its basic form message passing requires each message to have a sender, a receiver and some data. There is a small delay between the sender sending a message and a receiver receiving it that depend on the latency of the communication system and the message size in combination with the bandwidth. Special messages such as broadcast may also be possible to perform effectively, or it can be deduced to many messages from one sender.

Since every message is sent explicitly it might be easier to find and minimize the time-penalty of these. Message passing can also be efficiently translated and used in a shared memory system, which makes it a good paradigm for general use.

2.1.4 Issues and Potential

The major issue when designing and implementing a distributed application is that in order to reach an efficient result the design has to solve issues related to large latency, limited bandwidth, asynchronous execution between the nodes and possibility of node failure. Some algorithms and applications are simply impossible to produce efficient distributed versions of, often because of the nature of the algorithm or application. In other cases distributed versions that have good performance can be produced very easily.

There is always some overhead associated with the distribution of an algorithm. The aim of producing good distribution is to minimize this overhead. The ideal case would be that each node addition to the system would increase the performance by the individual performance of one node. This is rarely possible, but still a smaller performance increase will produce a system with better performance, and in some cases it might be the only or at least the simplest way of increasing application performance.

2.1.5 Feasible Problems

A large number of scientific problems can and have been proven to benefit from the use of distributed computer systems. Normally these problems have some special characteristics that make them suitable for running in a multiprocessor system.

A lot of mathematical operations, for example matrix operations, have been proven possible to develop distributed versions of. For some it is even possible to break up the matrices in strips and distribute them to the nodes that perform the calculations required on each strip. The overhead comes from the breakup of the matrices and the merging of the results for the final answer. Often there is a critical size based on problem size and the number of nodes involved that will affect the performance, sometimes decreasing the number of nodes used for the computation can improve total performance.

Various particle simulations are often difficult to execute with good results on a distributed system. Many algorithms in this area rely on pair-wise calculations between any two particles in every step of the simulation, and this requires a lot of synchronization that is expensive in a distributed system. By using approximations, for example dividing up the space into areas, efficient algorithms can be constructed for these problems as well.

Business applications often differ from scientific problems in characteristics. Tasks are often more transaction oriented and each task often has constraints that have to be satisfied and a deadline that has to be kept. Some constraints might be very difficult to keep in a distributed environment.

2.2 Programming Toolkits

Development of a distributed application is often difficult. Not only must the algorithm be constructed in a way that enables it to be distributed, but other practical issues such as how to start the execution, distribute processes, keeping track of the execution, handle data transfers and share resources must also be solved. A programming toolkit is a major aid in this area.

A programming toolkit provides a set of tools that offer ready-made implementations of certain function. A distributed toolkit should provide tools for functions that simplify development of a distributed application, and it should also take advantage of the underlying hardware. The exact functionality that is necessary is based on what the toolkit is to be used for. Various toolkits with different functionality have been developed such as a number of different projects at IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4 and PARMACS, Zipcode, Chimp, PVM [9], Chameleon, PICL and MOSIX [28].

A toolkit was usually developed to satisfy a specific requirement either on from applications or from system hardware. Each toolkit will thus usually have some stronger and weaker points, and which toolkit to use will likely depend on these properties. For a long time there was no standardized toolkit but the interest in distributed computing with cheap computer hardware spurred the development of the free PVM toolkit described in section 2.2.1. PVM became a de-facto standard for distributed computing but the different manufacturers of computer systems had not agreed on a standard, which meant that each system manufacturer often had their own toolkit. Having a standardized toolkit would help both hardware and software developers, and eventually MPI described in section 2.2.2 was accepted as a standard after a joint standardization process between many parties.

Toolkits such as PVM and MPI has made life easier for developers that want to write distributed applications. However the toolkits still operate on a rather low level. The vision of Grid computing spans further when it comes to functionality by adding features for security, data access, resource allocation, monitoring and service discovery. Standardization attempts in these areas have started as well.

2.2.1 Parallel Virtual Machine

The Parallel Virtual Machine (PVM) [9] project began in 1989 at Oak Ridge National Laboratory. The toolkit was based on message passing and was usable both in multiprocessor machines as well as in distributed environments or a mixture of both. The key concept in PVM was that it made a collection of computers appear as one large virtual machine, hence the name.

Application development in PVM was done using the concept of communicating processes. Logically the developer started a number of processes that could communicate with each other through an interface. Whether the processes were executed locally or on a remote host was decided at run-time.

PVM was designed to be versatile and it supported both data parallel programming and function parallel programming. Data parallel programming is when the data set of the problem is split up between the different nodes but they each execute the same basic logic. In function parallel programming different nodes are responsible for different functions. The possibility of using data parallel or function parallel programming, or even a mixture of both, made PVM useful in a different environments and was one key to its success.

PVM was completely free and quickly became the de-facto standard for developing parallel applications. The main competitors of the time were toolkits developed by hardware manufacturers, but unlike PVM applications made with these were hardly portable. PVM provided the basic functionality needed by parallel applications such as a message passing interface, synchronization and the possibility to start and stop applications across the network.

One feature PVM included that was really interesting was the possibility to dynamically resize the virtual machine by allowing nodes to join and leave the computation at run-time. This feature also allowed for the construction of fault-tolerant applications. However, it also adds complexity since an application must handle nodes that join or leave.

PVM has been very successful and popular because developers liked its features and interface. A lot of the functionality was borrowed when designing MPI (which we describe in the next section). PVM never became standardized and thus never became as wide spread as it perhaps could have become, but many of its features were borrowed when MPI (which we describe in the next section) was constructed. We refer to [9] for more information about the functionality and how to use PVM.

2.2.2 Message Passing Interface

The Message Passing Interface (MPI) project started when a number of vendors of concurrent computers, researchers, government laboratories and other parties of industry, joined together to create the Message Passing Interface Forum [26]. The object of the forum was to create a standard for an interface for message passing. The members of the group intended to create a practical, portable, efficient and flexible standard for message passing, and in 1992 the first draft was presented.

During the standardization of MPI the MPI Forum attempted to take advantage of previous experiences and adapt the most attractive features of previous message passing systems. The design of MPI has thus been influenced by many previous projects.

The MPI Forum had a number of goals when constructing MPI, some of which were to:

- design an Application Programming Interface (API) that can and will be used by developers of parallel applications.
- allow efficient communication by avoiding memory copying as well as allowing offloading communication to a communication co-processor if available.
- allow implementations that can be used in heterogeneous environments.
- allow convenient bindings in C and Fortran 77.
- assume reliable communication so that the developer does not have to worry about transmission errors.

The first version, MPI Standard version 1.0, was released in 1994 and the standard became widely accepted. A number of free and commercial toolkits were developed and this also lead to an increased user base. Some of the features missing in the standard were added in the MPI Standard 2 that was released in 1997. Among the important additions were dynamic process management, one-sided communication (message parameters such as message size decided at sender), extended collective operations and new language bindings (C++ and Fortran 90). As of today not many toolkits support all features of version 2 of the standard.

MPI is recognized and supported by most vendors of concurrent computers. These often provide implementations optimized for specific hardware, which results in increased performance. One of the major advantages of using an MPI toolkit is that it is portable. If the hardware is changed a recompilation will often suffice to build the application if it is written with MPI. MPI can also be used both in distributed memory as well as shared memory systems, which makes it highly versatile. Some commercial MPI implementations are MPI/Pro [31], ScaMPI [36], Digital MPI, Sun MPI.

A number of open-source MPI implementations have also been developed, MPICH [29] and LAM/MPI [22] are two of the most ambitious. They both have large user bases and are actively being developed. We describe MPICH, the toolkit used in the prototype of this project, in more detail below. More information on MPI programming can be found in [12] and in the MPI Standard documents found on the website of the MPI Forum [26].

MPICH

MPICH [29] is one of the most used implementations of MPI. It is an open source project that aims at producing a highly portable implementation of the MPI Standard. Major parts of it are developed at the Argonne National Laboratory.

MPICH attempts to follow the MPI Standard very closely and stay portable rather than optimizing performance. Because it has a large user base there are a lot of support available mostly in newsgroups and mailing lists. Some of the features (as of November 2002) include:

- MPI Standard 1.2 compliance.
- Bindings for languages C, Fortran and C++.
- Support for a wide variety of environments such as clusters of single-processor computers, clusters of multi-processor computers or massively parallel computers.
- Parts of the MPI 2 Standard implemented.
- Parallel programming tools such as trace and log file creation as well as performance analyzer.

A communication layer called Abstract Device Interface (ADI) was written as a communication framework. ADI allows MPICH to be ported to different communication systems, and this enables MPICH to be optimized for different hardware such as high-speed network interfaces. Manufacturers of computer hardware can thus use ADI to write communication drivers that optimize performance for specific hardware.

The MPICH version used for a normal cluster of workstations uses TCP and BSD sockets for communication, but communication drivers have also been written for Virtual Interface Architecture (VIA information see [38] and [32]), InfiniBand (see [21]), Myrinet (see [33] and MPICH-GM that is Myrinets port of MPICH for Myrinet) and other high-performance architectures.

In the standard distribution, MPICH comes in four different versions:

- ch_p4 for use with cluster of networked workstations. Can be used in heterogeneous environments and supports multi-processor nodes.
- ch_p4mpd for use with homogeneous clusters of single-processor computers. Because of the fewer features compared to ch_p4 this version provides improved startup time and startup scalability.
- Globus2 which supports the Globus Toolkit (see section 2.2.3) and uses routines found in the Globus Toolkit for startup, such as authentication.
- ch_shmem for use on a single shared memory system such as SGI Origin or Sun E10000. This version uses shared memory systems such as System V shared memory, anonymous nmap

regions for data, System V semaphores or other OS specific routines for mutual exclusion and synchronization.

MPI has a large user base and is being actively developed, which has made it a stable foundation for MPI application development. Currently efforts are being made to include more features from the MPI 2 Standard in the toolkit.

2.2.3 The Globus Toolkit

One problem with developing and running distributed applications in a grid or cluster environment was that some basic tools and functions had to be written for each application. Some of these issues, for example application startup, have often been taken care of by programming toolkits. Modern applications and environments often require a larger set of tools and functions.

The Globus Toolkit (see [17]) is an attempt to collect a set of tools that can be used by developers to easily get access to some functions often required when developing grid applications. Globus provides tools for security, data access, resource allocation and more. These are important parts for distributed applications, and by providing a framework development will be easier because an existing set of tools can be used.

The aim of the Globus project is to provide developers and users functionality that enable them to easily take advantage of a heterogeneous environment. The Globus toolkit provides a number of important services that can be used by any application. These services are made of components in a few key areas:

- Grid Resource Allocation Manager (GRAM) provides resource allocation, process creation, monitoring and management services
- Grid Security Infrastructure (GSI) provides secure authentication and communication over an open network.
- Monitoring and Discovery Service (MDS) is a Grid Information Service that uses the Lightweight Directory Access Protocol (LDAP) for providing and accessing system configuration, network status or the locations of replicated data sets.
- Global Access to Secondary Storage (GASS) implements a number of data movement and data access strategies, enabling programs running at a remote location to get access to local data.
- Nexus and globus_io provide communication services for heterogeneous environments.
- Heartbeat Monitor (HBM) allows detection of failing components or application processes in the system.

The components can be used by an application in order to enable it to run in various environments and across different organizations and locations transparently. The Globus toolkit provides infrastructure that can be used for running applications in a heterogeneous grid environment and enables them to take advantage of the services provided by the environment.

The Globus toolkit aims at becoming a widely-accepted standard for grid computing that will allow applications to more easily become “grid-enabled” and used in any environment. By providing an infrastructure that is standardized and well spread hopefully application developers will be able to take advantage of existing and future infrastructure technology.

2.3 Task Distribution

Task distribution is an optimization problem much like maximum flow, minimal distance, etc. The object is to optimize some function such as the total execution time or the average completion time of a number of tasks. What function to use depends on the characteristics of the problem: the type of service required, how the problem is solved, how to measure solution quality, etc. The distribution algorithm used should optimize this function, but in reality it is likely impossible to use the optimal distribution algorithm (it is too expensive) and approximate methods are used.

The idea is that in some situations it would be beneficial to use a more intelligent, possibly expensive in terms of resources, distribution algorithm that can compensate its cost with improved solution quality. The gain of the algorithm should be higher than the cost of using it. If the tasks are homogeneous and the system has no predictable behavior that can be taken advantage of, a simple task distribution function is most likely good enough. If the tasks or the system have properties that can be taken advantage of, it may be possible to find a more intelligent distribution function.

Also note that if there is no contention for resources, the simplest possible task distribution is the optimal one.

2.3.1 Mainframe Systems and Job Scheduling

Mainframe system commonly run batch-jobs, and the job-scheduling algorithm based its calculations on each jobs parameters such as memory required, CPU-time required, what devices and resources were required and perhaps other data such as priority. The scheduling algorithm decided the running order of the jobs given this input.

Job scheduling is NP-complete both in common single-processor and multi-processor cases (see for example [3]) which means that approximate methods have to be used. Evaluating how well an algorithm performs can be done with simulation data.

2.3.2 Distribution Algorithms

Distribution systems implement different distribution algorithms. Usually the assumption is that we have a set of tasks (may be ordered in time) and a set of workers. Both the tasks and workers may have specific parameters that determine their behavior. The distribution algorithm should minimize some distribution function or be effective in some common situation.

Some common algorithms are:

- Round-robin: The workers are logically ordered in a ring and the task assignment is decided by a orderly ring traversal. This algorithm will provide an even distribution.
- Weighted round-robin: This the round-robin slightly modified so that workers may appear multiple times during one complete ring traversal. This allows for uneven distribution of the tasks to compensate for example workers with different performance.
- Priority: An example of this algorithm would be to send tasks to one specified worker if available, otherwise one of the backup workers is selected.
- Least tasks: Keep track of how many tasks are being processed at each worker and send the current task to the worker processing the smallest number of tasks. Can also be used e.g. with TCP connections.
- Fastest response: Send the current task to the worker that provided the best response time on a recent task. The response time must be specifically defined, perhaps the time it took to respond to a recent request.
- Hash function: This algorithm uses some piece of information (the hash key) from the task such as an identifier and applies a hash function on this to determine the selected worker. The object of the function is to create an even or “almost random” distribution and the hash key as well as the hash function should be selected with this in mind.

The algorithm to use depends on the tasks and the system. It is often very difficult to calculate the best algorithm to use and empiric testing is often required.

2.4 Monitoring

Monitoring is the process of observing a system and taking note of some visible events that take place. It is impossible to monitor every aspect of a system, which forces some restrictions. Which parameters, spanning what time and how to actually perform the monitoring are questions that must be considered. A monitoring system that deals with something as complex as a computer system will have to make many assumptions and simplifications.

In a distributed system there are additional issues as compared to monitoring a single computer. Since every node in the system has its own clock there is no universal time. Also each network transfers is affected by latency so it is impossible to have exact synchronization.

2.4.1 Monitoring Events

An event in a system can be anything that happens that can be observed. Because there is no universal time it is difficult to know at which time an event happened. This is often handled by considering event causality. Causality means that by knowing the order of events, two events can be compared to see either in which order they occurred or if they occurred “in parallel”. If two events occur in parallel it means that it is impossible to know which of the events took place first without a universal clock.

The use of event causality makes it possible to have a logical time, for example with the use of Lamport timestamps (see for example [7]). Vector timestamps and dependency vectors can also be used for understanding the flow of events in a system.

2.4.2 Monitoring Architecture

A monitoring architecture is required to monitor a distributed system. Each node should be able to monitor its own events, but the monitoring architecture is responsible for processing and sharing the monitoring data.

A central monitoring station would collect data from each of the monitored nodes. The simple design and possibly good performance of such a system would make it suitable for many applications. On the other hand a centralized solution might be problematic both in terms of fault-tolerance but also if the monitored nodes are many of geographically spread which can cause communication problems. In this case other strategies might be necessary.

A development might be to use many monitoring stations. Related to this are a number of problems: how many stations, where should they be located, should the monitoring be overlapped between stations, performance problems, fault-tolerance, etc. Depending on what type of resources will be monitored and what service requirements exist on the system, possible solutions might be developed. However, a more advanced solution than the centralized will require added complexity in both the monitored node as well as the monitor stations.

2.5 Data Access

A program consists of logic and data. The logic is the code to be executed and the data is the data that is used during the execution. The data can be from input parameters, read from files, interactive information caused by interaction, network communication, etc. A distributed application might not have the data stored locally where it can be accessed fast with high bandwidth, thus the data access service must be supplied efficiently to make sure the application can get its data quickly and safely.

2.5.1 Distributed File System

Distributed file systems are vital parts of many organizations and extremely important in computer systems. The design of a distributed file system is not trivial because of the large number of parameters and events, as well as the demands of a fault-tolerant system. Common problems that have to be solved are how to handle data consistency, whether a state or stateless protocol should be used and how to make the system secure.

Some examples of distributed file systems are Network File System (NFS), Andrew File System (AFS) and Appletalk. The File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP) are both variations of distributed file systems in the sense that they provide remote access to files, but they are less advanced when it comes to features such as for example file locking. Their simplicity has however also made them popular, especially for simple file-transfers over the Internet.

Distributed applications and grid computing requires a distributed file system with a set of properties such as being usable, allowing localization of files (by using e.g. mirrors) to enable faster access, allowing simple access from multiple places and offering security for file access and transfer. The system must also provide functionality to handle different versions of a file to make sure the data is not corrupted and each instance of an application reads the same data.

2.5.2 Distributed Database

A database is often difficult to run in a distributed environment because some operations require locking of entries and tables, which is very expensive in a high-latency and low-bandwidth environment. There are a number of commercial distributed databases available and their design always tries to limit the effects of this problem. Usually a local cache is used on the nodes, but to make sure the data is consistent the caches must be invalidated after a write, thus many writes will cause bad performance.

Although potentially bad performance for the general case a distributed database can be quite efficient for certain applications, especially if database reads are far more common than database writes.

2.6 Related Work

Distributed computing is not a new subject and has been investigated in different projects of various extents. In this section we describe a few interesting projects related to the task distribution and monitoring, the topic of our project.

2.6.1 SETI@home

The SETI@home project [37] is by far the most successful project that takes advantage of distributed computing. Over 4 million users have downloaded the client application and taken part in the project, and some thousands have actively used it. These are large numbers compared to any other distributed application.

The object of the SETI@home project is to analyze a very large amount of data collected from radio telescopes and to find signs of interesting signals in this data. The problem is well suited for distribution because each work unit distributed to a host for analysis is rather small (about 1 Megabyte) and takes a long time to analyze (about 15 hours on average). The only communication required is when downloading a work unit and when uploading the result. The most important reason for the success of the project is that people actually donate computer power to the project, likely because they are interested in the project as well as the client application is easy to use (can be used as a screensaver on some platforms).

There are multiple other projects that attempt similar strategies such as Folding@home [16] that tries to find out how proteins fold and a number of projects try to find prime numbers, crack encryption keys or solve other mathematical problems. The projects in this category are by many referred to as “Embarrassingly Parallel” because they are by nature very easy to parallelize because of the small communication requirements in relation to the processor time required for each work unit.

2.6.2 NetSolve

NetSolve [2] is an attempt to create a system that lets an application take advantage of remote resources. The aim is to provide a system that allows an organization to keep a set of highly capable resources that can easily be used by applications through a simple programming interface. There are numerous issues that must be identified and solved in order to do this such as knowing hosts are active, what software they are equipped with, if they are available, etc. The NetSolve project has developed a client/server system that enables users to solve scientific problems across a network by making Remote Procedure Calls (RPC). This allows an application to take advantage of remotely located hardware and software.

The system consists of three parts: agent, client(s) and server(s). The agent is updated with information regarding what hardware and software each configures server is equipped with. When using NetSolve the client (or user application) states what resources its problem require and asks the agent for a suitable server (see Figure 2). The agent responds with a server (if any suitable was found) and the client sends the problem statement to the selected server. When the server has solved the problem the answer is returned to the application. All this is done with a single NetSolve function call.

The agent can also use features such as load-balancing to make sure problems are evenly distributed and fault tolerance to restart a problem if a server is found dead.

Being a scientific project NetSolve has language bindings for C, Fortran, Matlab and Mathematica. It supports features such as task farming (distributed one problem instance to each server), request sequencing, non-blocking function calls and includes Kerberos based security. The usage of NetSolve is a function call to a NetSolve routine, upon which NetSolve will automatically find the resources needed and solve the problem. The NetSolve agent can use features such as load-balancing to make sure servers are evenly loaded and fault-tolerance to restart problems if a server is found dead.

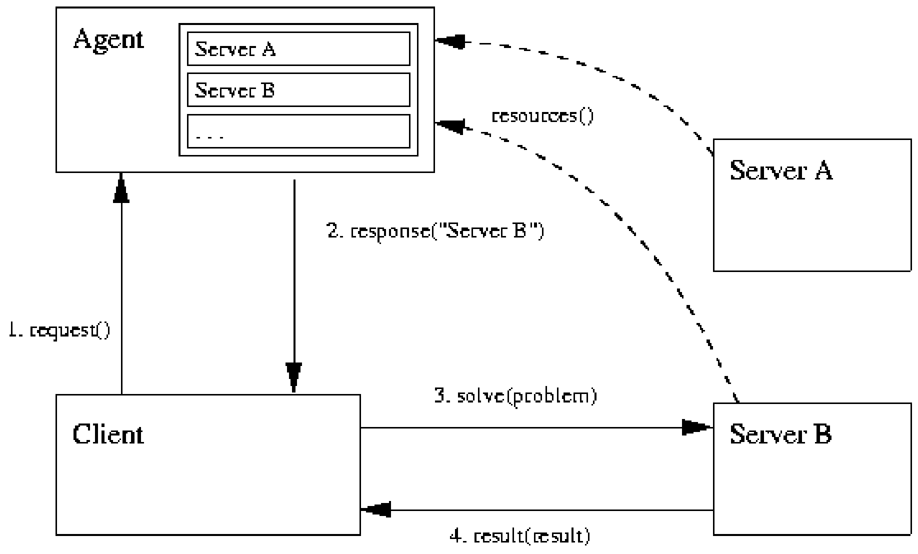

Figure 2. NetSolve in action. The client requests a server from the agent. The server returned is used to solve the problem.

Using NetSolve

Before started NetSolve requires some configuration on the agent, client and server. Starting the agent process on the designated machine sets up the agent. The process will bind to a configured port and wait for servers to connect and update their resource information that comprises of the hardware and software accessible on the server.

The server setup is done by writing a Program Definition File (PDF) that contains information about the software available. The file contains information about each function made available through NetSolve. This information includes a small chunk of code that will actually execute the function when called. The file is translated into source code that must be compiled before the server is started.

The client application setup comprises of including the NetSolve libraries and using NetSolve function calls such as `netsl()` to use the NetSolve software. Parameters to the function are the name of the function to call and application parameters. When the NetSolve functions are called the software will automatically contact the agent and use the retrieved server for problem solving, assuming there is a suitable server that has the required function defined in its PDF.

NetSolve has also been combined with other programs in order to try to get efficient task distribution and making sure that tasks get good service. One of the programs that it has been combined with is Network Weather Service, which we discuss in next.

2.6.3 Network Weather Service

The object of the Network Weather Service (see [13] and [34]) project was to create a system that provided accurate forecasts about dynamically changing performance characteristics from a set of computing resources. A sensor in NWS is a process that repeatedly polls resources on a node to measure the current resource availability. By storing earlier results the system can use the history to attempt to forecast the future resource availability. The forecasts depend on the usage patterns of the resources and of course the accuracy of a forecast will vary.

The aims during the design and development of NWS was to create a system that provided:

- Predictive accuracy: Accurate measurements and estimations of future resource availability.
- Non-intrusiveness: Interfere and load the monitored resources as little as possible.
- Execution longevity: The monitoring should logically be running an indefinite time.
- Ubiquity: The service should be available from any of the potential execution sites in a resource set and should be able to monitor and forecast all available defined resources.

The main parameters that NWS measures are CPU, network and memory resources. The forecasting is done by a generic function that takes as input a series of time-stamped values and from these it produces a short-term prediction. Resource measurement samples are commonly taken with a period of about ten seconds but this will depend on the type of resource monitored. It is also possible to plug in any type of forecasting algorithm that bases its predictions on a number series.

Each monitored host is equipped with a memory process and a sensor. The memory stores reports made by the sensors and also passes them on to the configured forecaster. Forecasts can be requested by an application as depicted in Figure 3.

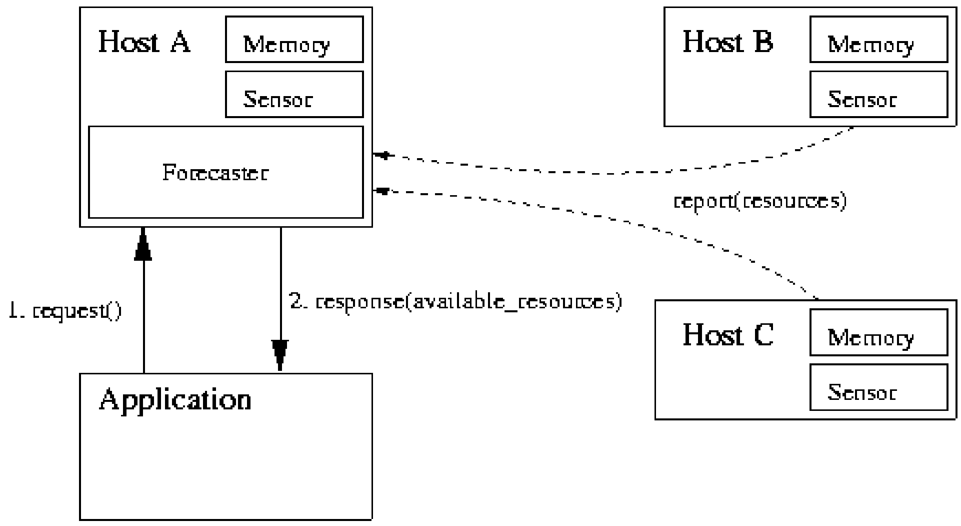

Figure 3. The NWS forecaster collects measurements from the sensors and provides predictions to applications.

Since the system is constantly running and making forecasts, it is able to learn by previous predictions. In normal setup, the system has two prediction algorithms: Mean Absolute Error and Mean Square Error. When calculating forecasts the NWS forecaster will use both methods. Since the forecasts are also stored, during the execution the forecaster can automatically use the forecast method that provided the best result (smallest error). If more forecasting functions are supplied, the system should be able to further increase its forecasting accuracy.

The system is also designed to be open and compatible with other software. This allows it to be plugged in as a module into for example Globus, which can then take advantage of its features.

2.6.4 SUN GridEngine

The SUN GridEngine [19] is a cluster management system developed by SUN Microsystem that is now open-source. The GridEngine is more similar to a mainframe system in its user interface, but is designed to run on a cluster of Solaris or Linux computers. The similarity to mainframe systems comes from the fact that the system is job-based.

In job-based system the problems sent are defined with some job-parameters. The parameters for a job can be requirements on memory, disk space or CPU time. The scheduler of the system is responsible for making sure that each job is matched to a node or set of nodes that will be able to fulfill its requirements. The scheduler can also, with the help of the job-requirements information, attempt to optimize the running of the jobs.

This type of scheduling is quite common in mainframe computing and using a similar system in a cluster is interesting. If for example we have a cluster were the different computers are equipped with different resources (hardware or software) we could use such a scheduler to make sure that job requirements are met, and that the resource usage is optimized.

GridEngine has some interesting features, for example that it is possible to define the load measurement used by the scheduler. Instead of just using the current load value of the system to decide which system is less loaded, it would be possible to define other parameters that might suit the problems better and provide better service.

2.7 Problem Types

In this section we elaborate on some common problem types. We loosely base our categorization of problem types on the communication requirements of a problem in relation to its computation requirements.

2.7.1 Loosely Coupled

The Loosely coupled category is in its most extreme form often referred to as Embarrassingly Parallel Computing (EPC). Applications in this category often perform intense computations on smaller data sets, often of mathematical nature. Examples include the SETI@home project described in section 2.6.1 and other projects mentioned there.

The applications of this category often have a fairly small problem definition, whereas the actual problem solving requires very intense computing resources. This is often because the problem solving is algorithmically complex and requires a lot of processor time. What makes this category loosely coupled is the fact that the communication required between the problem solvers is very small. The nodes involved in the computation often do not have to communicate, and thus there is little synchronization required. A network will thus not have a major influence on the total performance.

These type of problems are often ideal for distribution to a large number of computers and the most successful distributed computing problems are of this category. Unluckily the number of problems in this category is limited.

2.7.2 Extremely Coupled

Extremely coupled problems have the property that the different solvers require a high degree of communication or synchronization in order to solve the problem. Consider for example a particle simulation where every particle will have influence every other particle in every step of the simulation. This causes a lot of synchronization, which will be very costly in a distributed environment. This type of problem is often much more suited for a shared memory system because it is most efficient if a “global memory” that can be read and written by any processor is available.

2.7.3 Somewhat Coupled

The Somewhat coupled problems are problems that required more inter-worker communication that the Loosely coupled category, but they might still, given the proper resources, be runnable with good results on a cluster of computers. The performance of the network and the algorithmic design will have a major impact on the performance of the problem solver. Problems in this category often required some synchronization but at the degree that using multiple solvers connected with a network will still positively affect the performance. Because of synchronization, problems might also arise if too many hosts are involved.

This type of problem is quite common in a number of areas and sometimes Extremely coupled problems described in section 2.7.2 can be approximated to enable them to run in this category. One example of this is when a particle simulation can be divided up in space and approximate values can be used for the regions that are executed remotely. This decreases the synchronization requirements and the approximated problem solver can thus perhaps run with good performance on a cluster system.

2.7.4 Business Transactions

Business transactions are often rather small in size, but each request may involve computations on large amounts of data. Typically the number of requests per time-unit is high, which is the major performance problem for this type of system. Examples of this type of system is a business database, where small queries are sent that potentially require access to very large sets of data.

Since very large amounts of data will require a very large host, attempting to distribute the data to smaller hosts while still allowing the system to be used with equal or better performance is highly interesting. Another interesting aspect is the possibility of fault-tolerance in such a system.

2.8 Difference in Approach

Task distribution can take on different shapes. It might be scheduling of jobs on a set of resources where each job has a specific resource requirement that must be provided by the system. It might also be distributing TCP connections between servers, a task that is also performed by a network level router that can use information in the protocol headers. The task distribution can work at different levels of the OSI stack, and which solution to use depends highly on the type of application that is involved.

In the scenario of this project we are located somewhere in between the two solutions mentioned. The traffic we focus on is of the transaction type, which means that we have a large throughput but a small solve-time for each request. The traffic is injected over TCP connections that send a stream of separable requests, and these requests should be distributed among the workers. Additionally we assume that the cluster we use runs other applications, which means that we are not in complete control over its resources. Thus it is not possible to reserve resources, but rather we have to monitor them to make sure we can get the service we require.

3 Method

The goal of the project was to investigate distributed computing or grid computing to see if it was applicable for some categories of current and future applications. The first step was to get and idea of what we wanted to do, and after this we decided to develop a prototype application.

3.1 Tasks

Examining the tools that are available and learning how they function was an important part of the project. The initial study helped us decide what tools we should use and what type of prototype we should develop.

Based on the study and the foreseeable applications we decided to develop a prototype framework for distribution of tasks. Figure 4 shows the design principles of the prototype. The idea is to use the grid for serving a set of request injectors with a single entry-point that we call the coordinator. The remaining nodes in the grid act as workers that perform the actual computations required to serve requests. The motivation for the design was its appeal to infrastructure requirements.

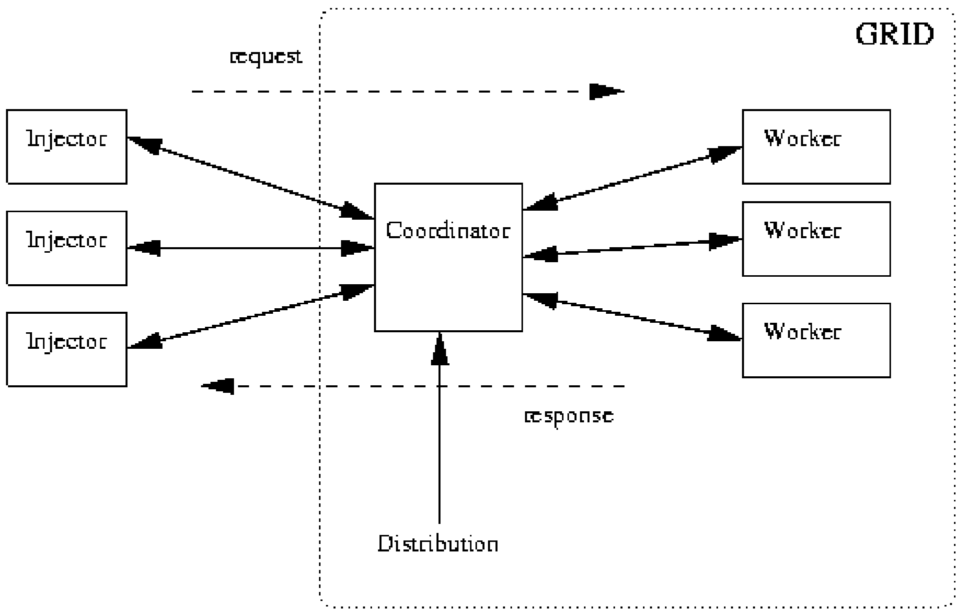

Figure 4. Sketch of the prototype application.

The prototype should leverage on existing working and accepted technologies and should serve as a proof-of-concept that can be used to test ideas, analyze and evaluate performance or other parameters.

3.1.1 Coordinator Responsibilities

The responsibility of the coordinator is to shuffle data between injectors and workers. The value of the coordinator is that it provides a single point-of-entry to the grid to the injectors and it can perform an intelligent distribution based on information in the application layer of the system.

The focus of the coordinator was on the distribution of requests and it is not responsible for more advanced application issues such as maintaining order on requests or handling lost requests.

3.1.2 Worker Responsibilities

The worker has the simple job of executing the requests served by the coordinator and returning the results of those requests. The workers execute a function that solves the given problem (more on this later) for each request received, and they may use static or dynamic data to do this. The type of request is defined at compile-time.

3.1.3 Injector Assumptions

The injector should connect to the coordinator with a TCP connection and start feeding requests. We assume that the order of received replies is not an issue, neither is the loss of requests as a result of errors. The injector can send one request at a time and wait for the answer or it can have multiple requests issued.

3.2 Requirements and Expected Results

Gaining hands-on experience of distributed computing and finding out the problems and possibilities of it was one of the main reasons for developing a prototype. We expect the prototype to provide us with measurable results about design, implementation and usage of applications in distributed computing. By performing the project we will hopefully be able to draw some conclusions to whether distributed computing is usable, what type of problems exist and how to take advantage of the technology.

The prototype is not focused on a specific problem, which gives us a lot of freedom in our investigation to look at the areas that seem most interesting. However we need some restrictions, and those are the project environment described earlier in section 3.1. We expect that this will enable us to draw conclusions to whether distributed computing is useful in this area, and what restrictions will apply to such a system.

3.3 The Prototype

The prototype design required determining what algorithms were to be used in the prototype. In this section we describe the algorithms used.

3.3.1 Problems

In order to test the prototype we required some payload that would consume some resources. We describe the problems we used to load the system with in this section.

Spell Checker

We selected to use a spell checker as one problem in the project. The industrial advisor of the project, Alexis Grandemange, had previously written a spell checker based on Ternary Search Trees (see [5]) in C++ that was well suited for use. Some wrapper functions had to be written to plug the code into the prototype.

The data set for this problem is a file containing the words to be included in the dictionary. We used a dictionary with about 25,000 English words. In this problem there is no need for cooperation between the workers.

Find Route

We implemented a route finder in a network of airports and flights. The problem is as follows: Given a departure airport, a destination airport and a graph consisting of airports (nodes) and flights (edges), the object is to produce a number of possible routes (or set of flights (edges)), that will take us from the departure airport to the destination airport.

The solution is based on a Depth First Search algorithm (see [1]) with some constraints. Algorithm efficiency was not of primary concern. The find route application may require some cooperation between the workers. This is further described in section 6.1.2.

The data set used in the find route was a file with about 300 airports and a set of 30,000 randomly generated flights.

Null Problem

When we evaluate the prototype we use a “null” problem. When any type of request is received, the solution is to return a result with a specific length defined at compile-time. The content is not defined. The motivation for this problem is that it takes little or no time to “solve” the problem but it creates network traffic and we can decide the length of results. This is used to test the performance of the prototype.

3.3.2 Algorithms

Communication Logic Algorithms

Communication and data transfer is one of the most expensive operations in a computer. It is therefore desirable to do this as efficiently as possible, and to try to maximize the throughput by taking advantage of the tools and tricks available. One of the most important tools that can be used to optimize communication is to use non-blocking communication.

The problem with using non-blocking communication is that it often makes the program more complex and difficult to follow. However, the advantage is big when it comes to resources and the possibility to do computations in parallel with the communication.

In the prototype we try both a normal read-write communication, and also a more complex non-blocking communication with multiple buffers.

Request Distribution Algorithms

Two different request distribution algorithms are used. The first algorithm lets the workers decide the pace of the requests by forcing them to actively ask for each request. The second algorithm uses monitoring to find resource availability on the workers and this information is used to decide the distribution of the tasks. These algorithms are described more in detail in section 4.

Data Service Update

The spell checker and find route problems use a set of data, either the word dictionary or the network of flights. The data involved is of limited size, a few hundred kilobytes or so, and we don’t have very tight constraints on the data service required for either problem.

The data used by a general problem is potentially very large, measuring many megabytes. In this case it is often not feasible to reread the entire data set when it is updated. Also, an update of data is for many problems considerably smaller than the entire data set. The periodicity of the updates is also an important parameter.

Since our problems have relatively small data requirements, we allow the application to just reread the entire data instead of adding features for data updates. We also do not require the nodes that take part in the computation to be synchronized when it comes to the data used; our requirements are rather loose. We assume that we need to update about 2 times per second, a figure that is based on existing application requirements.

3.3.3 Implementation Approach

The prototype was developed with an iterative method with added functionality and tests performed at each iteration. This way we were able to test new ideas and decide about new functionality as the project developed.

The problem solving was generalized from the problem solver view to reading a request from a buffer, solving the request and writing the response to a buffer. This made it possible to hook up any type of problem solver, and we could interchange solvers and problems easily during testing.

3.3.4 Hardware Used

A small “grid farm” was constructed and used as a platform for testing the prototype. This small grid consisted of four PC computers each equipped with a 1 GHz processor, 256 or 512 MB or RAM, 16 GB hard drive and an Ethernet card capable of 100 Mbit/s of duplex traffic. The computers were interconnected with a network-switch capable of peak traffic.

The computers were installed with RedHat Linux 7.1 and a number of grid tools were installed such as MPICH, NWS, PVM.

3.3.5 Implementation Language

The first decision regarding the prototype was which programming language to use. First candidates were a solution using C/C++ or Java. Fortran or High Performance Fortran (see [20]) was also considered since it has a history of being used in high performance computing.

Java was appealing because it provides quick development and quick production of applications with high quality. However the main drawbacks were that Java is slower than a C/C++ solutions and it was also not targeted towards high performance distributed computing making toolkits fewer and less mature.

Fortran has been mature since the 1950s and lately has had success in high performance parallel computing with High Performance Fortran (HPF), making it a very interesting choice. The language has many interesting features such as compiler-optimization and HPF provides a portable syntax for data-parallel computations. The fact that we did not have previous experiences with Fortran was one of the showstoppers.

C/C++ was chosen because it is well known, used a lot in grid computing, and we also have previous experience with it. It was also well suited for the Linux/UNIX platform and it had great availability of matured grid toolkits. C/C++ was also accepted in the current infrastructure.

4 Implementation

Using the platform provided and after having selected the programming language we concluded that we wanted to use a toolkit to develop the prototype. This resulted in the first MPI-based prototype.

After implementing the problem solving parts and the prototype using the MPI toolkit, we evaluated it and planned for the next move. We decided that we wanted to use another algorithm for task distribution, which could not easily be added to the current prototype structure. To further investigate this we developed a second prototype that was not based on MPI.

This section describes the implementation of the two prototypes we developed.

4.1 MPI Prototype

We had decided to use MPI to develop the prototype and after comparing different implementations we decided to use MPICH. MPICH was chosen because it had a big user base, wide support and a lot of functionality. Since MPI is a standard interface we could also switch implementation just by sending other flags to the compiler and linker.

MPI communication was used within the grid and between the nodes in the grid (see Figure 5). The communication of data from the injectors was sent over TCP using BSD sockets.

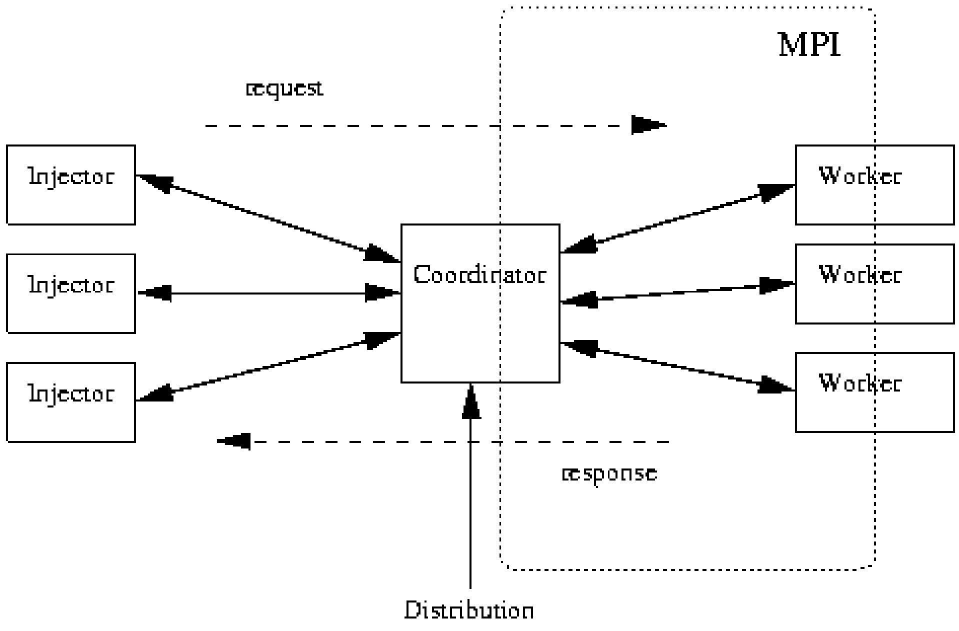

Figure 5. The MPI prototype. MPI is used for communication inside the marked area.

4.1.1 MPI Usage Motivation

MPI is well spread and widely used in grid computing and parallel computing in cluster systems and in larger parallel systems. One of the most important benefits of using MPI is that you will be able to use concepts and features that simplify development. We selected MPI over for example PVM and Globus Nexus because MPI is a standard and widely distributed but also because MPI can be plugged in to the Globus toolkit (see section 2.2.3) at a later stage should some Globus functionality be required.

Some of the important features include guaranteed messaging, defined groups at startup, different message passing protocols depending on message size and non-blocking message passing. These features are important since they provide a framework and foundation on which the application can be developed, and they also provide features that are helpful in creating good application behavior. Non-blocking message passing allows for good CPU resource utilization.

MPI also implements for example broadcasting and more advanced collective operations like Scatter and Gather (scattering/gathering an array of numbers to/from each node in the computation). If the application can benefit from these types of operations they are often implemented efficiently to make sure they are effective and sparse on resources.

MPI was used for communication between the coordinator and the workers, which are the computers that are part of the grid farm. BSD sockets were used for communication with the injectors. Note also that the MPICH version used is based on socket communication, but this is wrapped in the MPI function calls with a resulting small overhead.

4.1.2 Implementation

As described in section 3.1 the prototype consists of two major parts: one coordinator and a number of workers. The coordinator is responsible for forwarding requests to the workers. The workers are responsible for executing requests: they should solve the problem related to each request.

When using MPI each node is given a name represented by an integer. With `p` nodes the nodes are named between `0` and `(p-1)`. In our implementation we use this naming by letting the node named `0` be coordinator and nodes `1` to `(p-1)` are workers.

Request and Response Format

The coordinator accepts one or more incoming TCP connections from injectors. The injectors send requests over the TCP connections and to separate requests each request is prepended by the request length represented by a `short` (a 2-byte integer) in Network Byte Format (see Figure 6). The same format is used for the response. A request or response can thus be between 0 and 65535 bytes long.

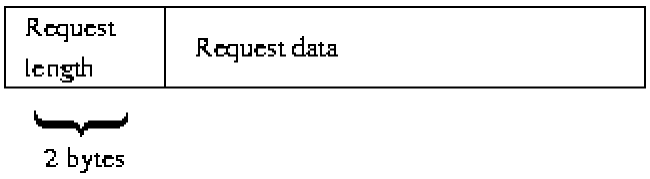

Figure 6. Request and response format.

Coordinator Logic

The coordinator is responsible for receiving requests from injectors on the incoming TCP connections, route them to an available worker and eventually return the response sent by the worker to the injector. Figure 7 shows a schematic image of the coordinator behavior.

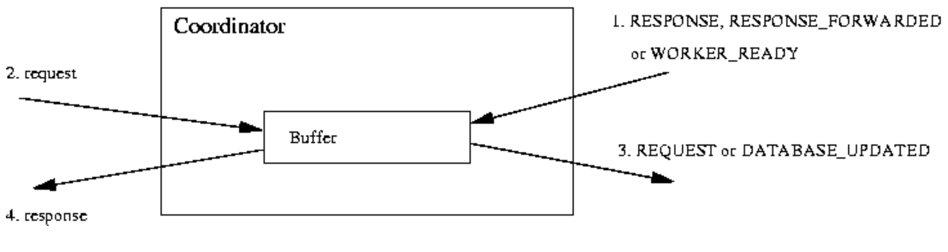
Coordinator
Buffer
1. RESPONSE, RESPONSE_FORWARDED or WORKER_READY
2. request
3. REQUEST or DATABASE_UPDATED
4. response

Figure 7. MPI Coordinator design.

At startup the coordinator first initializes MPI. It then binds to and listens to a port number specified as input parameter for incoming injector connections. When a connection is made the coordinator can begin forwarding requests.

A simplification of the order of the events in the coordinator can be seen in Figure 7. It shows the order of the events in the coordinator, the numbers represent the order in the loop of each event. The figure also shows the three messages that the coordinator may receive:

`WORKER_READY`: This is sent at startup when the worker announces explicitly that it is set up and is ready to receive a request.

`RESPONSE`: The worker sends a response that is to be returned to an injector. This also implicitly states that the worker is ready for a new request.

`RESPONSE_FORWARDED`: The sending worker has forwarded the request to another worker (which worker is stated in the message). The sender of this message is thus ready for a new request. The coordinator must not send a request to a worker that has been forwarded a request by another worker.

When a message from a worker is received the coordinator will read a request from an injector (step 2), which one is selected by round robin on the sockets. The request read is (step 3) sent to an available worker. The coordinator keeps track of which injector sent which request by adding a field with the injectors file descriptor number to the request. This field must by copied to the response by the worker.

After sending the request to the worker the coordinator will wait for a response from any worker (step 1). When a request is received it will return the response to the injector stated in the response.

The coordinator may instead of a `REQUEST` message send a `DATABASE_UPDATED` message with a request as message data. This message informs the worker that some data has changed and is useful if the workers should reread a database, file or other source of information in order to update the application data set. The exact definition of what data has changed is up to the specific problems to decide. The implemented application reads a file over NFS and reinitializes some data structures.

Worker Logic

The workers are very simple in terms of logic. They use MPI to communicate with the coordinator and the other nodes in the grid. The file transfer is made with NFS. Figure 8 shows the worker design.

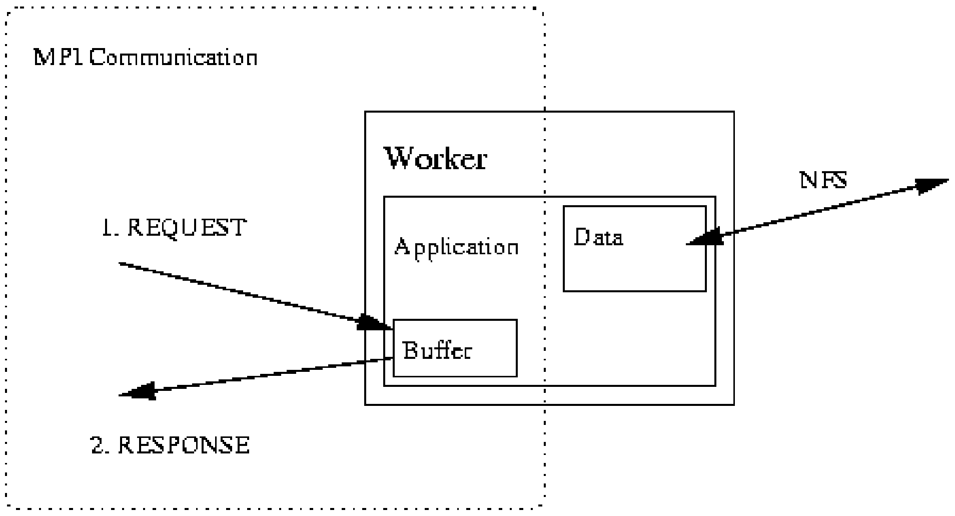

Figure 8. MPI Worker design.

After the worker is started it sends out a `WORKER_READY` message to the coordinator stating that it is ready to receive a request. It waits for requests in a blocking receive. When a request is received (step 1) it is solved by calling a function specified by a function pointer at compile-time. The definition of a solving function should look like this:

```
int solve_function(char *req, char *res);
```

The function takes as arguments a buffer where the request resides and a buffer where the response should be written. The function must return the length written to the result buffer or `-1` if an error occurred.

When the response has been written to the buffer, along with the response length and file descriptor number copied from the request, it is sent back to the coordinator (step 2).

If a `DATABASE_UPDATED` message is received the worker runs a function specified at compile-time. In the spell checker and find route problem implemented the function rereads a file over NFS and reinitializes (reconstructs the data structure) the problem-solving parts of the worker. This function could also be null meaning nothing is done, as is the case with the null-worker.

4.1.3 Fault-tolerance

The version of MPICH that we used implemented version 1.1 of the MPI specification. The specification does not define what is to be done if communication errors arise, so an attempt to communicate with a node that does not respond results in the application exiting with an error.

4.2 Socket Prototype

We wanted to create a prototype that used a task distribution algorithm that took advantage of monitoring information. Since the MPI-based prototype we had developed was not suited for this approach we decided to develop a prototype that did not use MPI but was instead based on pure BSD sockets.

4.2.1 Motivation

MPI is very general since was created to fulfill the requirements of a large number of applications with very general communication requirements. This also results in MPI being too advanced and rich in features that cause unnecessary overhead. Using BSD sockets directly would provide better control of the data flow in the application, allowing us to streamline the buffer handling and use non-blocking communication. Additionally MPI prevented us from using the task distribution algorithm that we wanted.

4.2.2 Implementation

The socket prototype started from about the same code base as a mature version of the MPI prototype.

Request and Response Format

The format of the requests and responses are the same as in the MPI prototype. All data sent is prepended with a short integer stating the length of the data in bytes. We have the same data size limit from 0 bytes to 65535 bytes.

Coordinator

In order to make effective use of non-blocking communication a new buffer design was implemented, schematically depicted in Figure 9.

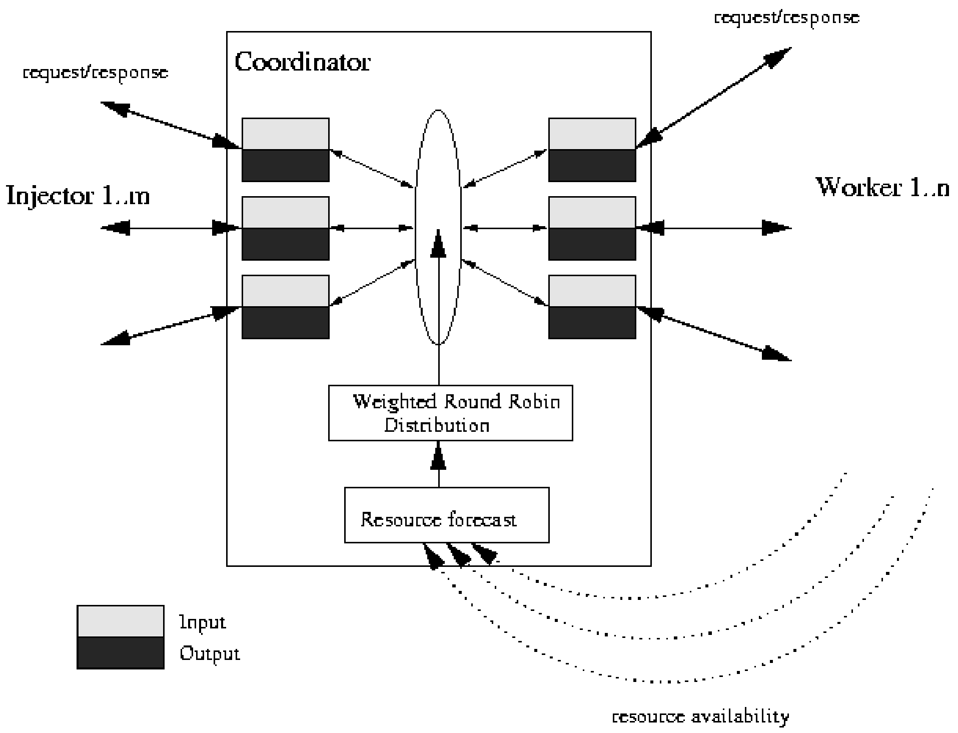

Figure 9. Socket coordinator design.

As can be seen in Figure 9 the coordinator uses two buffers per socket, one input- and one output-buffer. The connected injectors and workers are represented using the same data structure with buffers. The buffers enable the coordinator to shuffle data quickly by using the `select()` function on the socket file descriptors. The function is used to efficiently monitor a number of socket file descriptors for read and write availability, and with repeated calls to `select()`combined with reads and writes very high data throughput can be obtained.

The new non-blocking data flow allowed us to shuffle data very quickly between the injectors and the workers. The full control over the sockets allowed us to let workers join and leave the system and we were also able to take advantage of the network communication buffering of the operating system without having MPICH add overhead.

The new data flow also allowed us to implement the new distribution algorithm that makes distribution decisions for every request taking into consideration the number of requests sent to a worker, whether the socket of a worker is writeable (buffer is not full) and the resource availability of the workers. The resource availability of the workers is measured by the Network Weather Service described in section 2.6.3, and the current CPU resource availability on the workers are fed into a weighted round robin function where the weights are based on the resource availability. A NWS forecaster process that is asked for resource availability is started on the coordinator.

This prototype did not implement different message types such as an equivalence of the DATABASE_UPDATED message found in the MPI prototype, but adding this feature is fully possible.

Worker

The worker of the socket prototype had basically the same functionality as the worker in the MPI prototype. One difference is that each worker is running a NWS sensor process (see Figure 10) that reports CPU resource availability to the NWS forecaster process running on the coordinator. Another difference is that a worker will only communicate with the coordinator and not with other workers. Since the DATABASE_UPDATE message is not implemented there is no support for this feature.

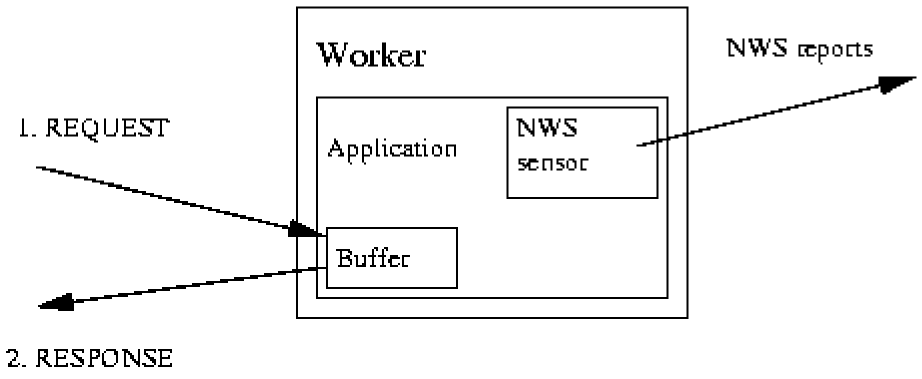

Figure 10. Socket worker. Note the NWS sensor process running on each worker.

4.2.3 Fault-tolerance

Since the socket prototype allowed complete control of the socket communication we were able to check for errors and remove workers that disconnected or died. We could also keep listening on the incoming connection socket for workers, which allowed a worker to join at any time.

The fault-tolerance worked to the extent that the system will not die when a worker died, but any requests dispatched to that worker were lost. A more advanced fault-tolerance mechanism was beyond the scope of this project.

5 Analysis

The analysis of the prototype was important because we needed to get information about how it behaved and if it could satisfy the requirements of the applications. This section describes and motivates the analysis.

5.1 Object of Analysis

The object of the analysis of the prototype was to investigate whether the prototype was usable for the applications it was considered, and to gain a better understanding of what could be expected from the technology. It should also provide information regarding what type of applications can be used and how they should be implemented. We focused the analysis on a few items: application performance, infrastructure overhead, programming concept and problem feasibility.

5.1.1 Application Performance

The purpose of using distributed computing in the first place is to get some benefits, primarily better application performance. The performance is very important, but it is sometimes interesting to trade some performance for other features such as fault-tolerance, complexity or financial cost.

Since we only had access to a small grid farm consisting of 4 nodes, it was of interest to understand how the prototype would scale to a larger grid farm consisting of more nodes and understanding what type of problems would arise from this. This part is especially important since one aim of the project was to get some information about this area before actually scaling up to a larger and more expensive grid farm.

5.1.2 Infrastructure Overhead

The infrastructure overhead is a measurement of the extra cost following the use of a distributed application compared to a non-distributed application. The overhead comes from application startup, communication and from the closedown. In our study we focus on the communication since this is the part that will cost throughout a long-term execution.

Of course the object is to have the smallest possible overhead. We were interested in comparing the infrastructure cost with the added performance. Due to the nature of the application we developed we focused on the network traffic and the task distribution when analyzing infrastructure overhead.

5.1.3 Programming Concept

The programming concept can be described as a general view of the prototype development approach. Analysis of this should include an analysis of the toolkit selection and the benefits related to this. It should also include the analysis of other parts of the implementation such as the general algorithm, the programming method and program evaluation.

5.1.4 Problem Feasibility

Problem feasibility was a very important subject for the project, since the project is not focused on a specific problem. The object was to gain understanding on what type of problem can efficiently benefit from a distributed environment, and what type of constraints and limitations the problem is bounded by. This analysis required us to investigate the different problems that are at hand and understand their properties.

Studying this also opened us up for the investigation of other areas such as data replication, distributed databases as well as data and program consistency aspects.

5.2 Tool Usage

After defining the parameters we wanted to analyze we had to find a way to do this efficiently and with some degree of accuracy. Analyzing the performance of a computer and a computers system is not trivial and we have to rely on the tools we have at hand. Such tools include what we can make out of the operating system, what we experience when monitoring the system as well as trying to use timing data and statistical data in order to understand how the system works. This section describes the tools we used for our analysis.

5.2.1 Counters

One of the simplest mechanisms we used was to embed counters in the code and allow events to trigger these. Events that triggered the counters were for example when a request was received or when a response was returned to the injector. Timers were used to enable us to periodically check or output the current status of the counters.

Counters are very useful because they allow us to see the status of the system at any given moment. However they should be used with moderation since they have a tendency to make the code ugly. It is important to verify both that you are actually counting the correct parameters, as well as that your counters function properly. For example they should not count the same event twice by mistake.

5.2.2 Traffic Injector

Throughout the development and testing of the prototype we used a traffic injector. The traffic injector was written in the Java language and its design is very simple. It opens up a TCP connection to the coordinator on a specified host and port, and then it starts two different threads. The first thread writes requests to the socket and the other thread reads responses. A few different versions were used for different problems, performance tests or verification. We ran the traffic injector on a workstation not part of the grid cluster.

The purpose of the traffic injector was to test the functionality and performance of the prototype. By starting a number of different traffic injectors, on one or many hosts, we also made sure that the prototype could handle more than one injector and scale properly. The design of the injector was motivated by the design of existing applications.

5.2.3 Profiling

The performance of an application can be studied with the aid of a profiling tool. Since we used the GNU Compiler (also know as "gcc") we used gprof, which is a profiler available with GNU Compiler.

The gprof profiler is enabled with the GNU Compiler by passing the flag `–pg` when compiling. This enables the profiler and it will add some code to the binary increasing its size and causing some overhead. When the application is run the profiler will keep track of function calls and the time spent in each function, and from these measurements it will produce an output file that is viewable with the program gprof.

The output file contains information regarding the total running time, the percentage of time spent in each function, the number of times a function was called and a "call graph" showing how often functions called each other. This information can be used to understand where time is spent in the application and what functions should be targeted for optimizations.

The profiling introduces some overhead in running time and application size, and should only be used during the development phase. Also there are very limited guarantees about the accuracy of the information provided.

We used profiling throughout the development of the application. This allowed us to focus optimizations and to verify were the application spends time.

5.2.4 MPICH Logging

Some extra analysis features accompany MPICH. One of the most useful features is the logging. As with the profiling, all that is needed to enable logging is to pass a flag, in this case `–lmpe`, to the compiler and linker.

The logging will give a unique name to each type of MPI event, such as a `MPI_Send()` or `MPI_Recv()`. It will also log on a slightly lower level such as when messages are received. The developer may also add log identifications for any other functions, and they will then be logged as well. Each event is logged with id and timestamp, and the logging takes place on all the nodes in the MPI run. When the program has terminated, the logs of the different nodes will be collected and merged into one single log file.

The merged log file contains information about all the logged events on all nodes, with timestamps. Because each message that is sent is given a unique identification, it is also possible to match sent messages from one node with received messages on another node. This makes it possible to deduce relations between events in the system, the event causality as described quickly in section 2.4.1.

5.2.5 MPI Log Viewing with Jumpshot

The log file is written on a format that is descriptive but not readable; the format is either CLOG or SLOG, two standard log file formats. With the MPICH distribution is included a Java program called Jumpshot. Jumpshot can be used to visualize a log file, and it is possible to zoom in on events (there are often a very large number of events). Figure 11 is a sample view of Jumpshot. We refer to the MPE manual [6] for more information on Jumpshot.

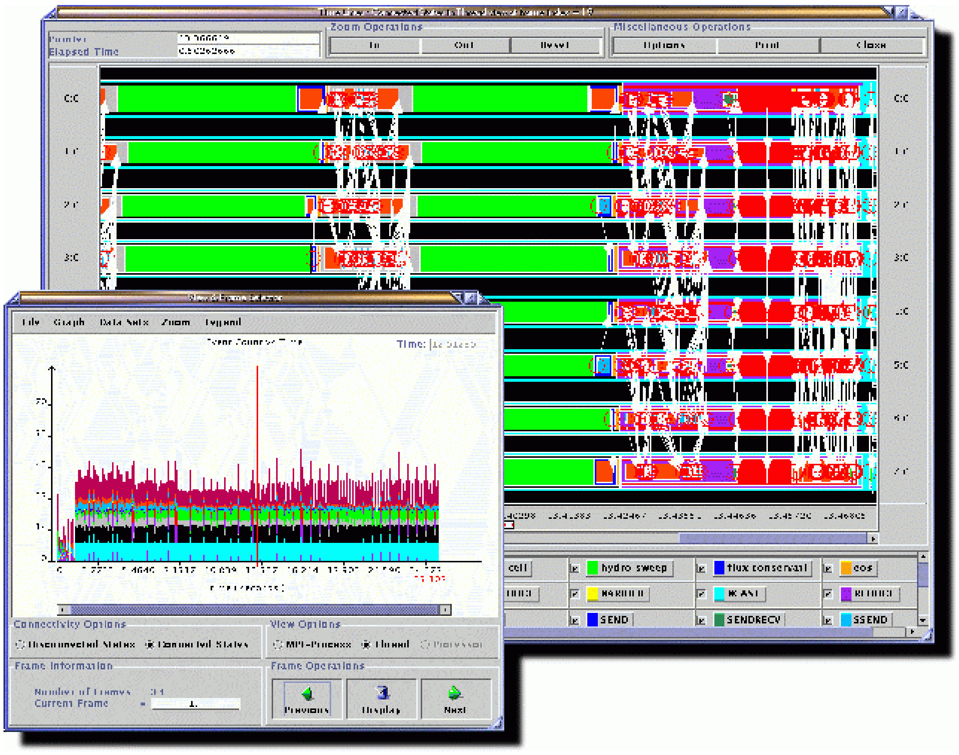

Figure 11. Sample Jumpshot view.

5.3 Method of MPI Prototype Analysis

The MPI prototype was analyzed with the aid of profiling information, counters and also the MPICH logging features. Interactive debugging was also used to some extent to verify the program flow.

5.3.1 Throughput

The primary aid when measuring the throughput of the prototype was the use of the traffic injector described in section 5.2.2. In order to test the performance of the prototype coordinator and not a problem-specific performance we used the “null problem” described in section 3.3.1. The null problems enabled us to analyze the maximum possible data throughput of the prototype on the available software and hardware resources.

The throughput results were obtained by using one traffic injector and all four nodes in the cluster (one coordinator and three workers) by observing the prototype counters representing the number of requests processed. The test was repeated with different result size settings (defined at compile-time with the null problem) and the results of these tests were noted. The counters show the number of requests and replies processed each second, and since the sizes of these were known the numbers could easily be translated into bandwidth usage.

5.3.2 Spell Checker Performance

The performance of the spell checker was tested by starting the traffic injector set to send requests of the correct format. By reading the counters on the prototype we were able to get figures of the request throughput on the coordinator.

5.3.3 Logging and Log Viewing

By enabling logging when compiling the prototype the prototype produced an output file containing the logging information. The information in the file was opened using the Jumpshot log viewer and by inspecting the image presented with Jumpshot we were able to draw conclusions about the application performance and behavior.

5.3.4 Request Distribution Algorithm

By examining the performance figures, profiling information and log view we were able to get information about how the request distribution algorithm behaved. The aim was go get good performance figures, but these tend to depend on many factors. The profiling information allowed us to investigate where performance could be improved, to the extent that we could make sure that the request distribution algorithm worked good with regard to resource usage. The log view allowed us to visually investigate how the algorithm worked, and help us describe its behavior.

5.3.5 Data Service Algorithm

The MPI prototype implemented the `DATABASE_UPDATED` message that was used to inform workers that the data (the file) should be reread. The spell checker was the only prototype that took advantage of this feature by deleting the old data structure from memory and rebuilding it using the file.

The data service was monitored manually by making sure updates were made on time and that they were not taking too long.

5.3.6 Data Flow

The prototype data flow was analyzed through code inspection and with the help of profiling information such as call graphs.

5.4 Method of Socket Prototype Analysis

In the analysis of the socket prototype we relied mostly on counters and profiling information. Since the socket prototype did not use MPICH it was not possible to use any of its logging features. To verify functionality and find bugs we also used some interactive debugging.

5.4.1 Throughput

The throughput of the socket prototype was measured the same way with the MPI prototype described in section 5.3.1. We used one injector sending requests to the coordinator that distributed these to three workers.

5.4.2 Scalability with Find Route

The scalability test was performed by increasing the number of workers used (gradually from one to three), and by increasing the number of injectors used. We tested scalability with the find route problem since it was the only problem we had were the processing power of the worker set the limit on throughput.

The test with increasing the number of workers was performed by starting the prototype with one single worker and using one traffic injector to inject requests. The throughput was recorded and after this we added another worker. The cluster limited us to three workers. With the throughput of requests and responses and their sizes, we are able to compute the network bandwidth used.

By increasing the number of traffic injectors we tested the scaling of this part of the prototype. We performed tests with between one to ten traffic injectors running on one single host.

5.4.3 Non-blocking Communication

The analysis of the non-blocking communication system used in the socket prototype was done by inspecting profiling information, throughput figures and following program flow throughout the development of the prototype.

5.4.4 Request Distribution Algorithm

The request distribution algorithm used in the socket prototype was analyzed with the help of profiling tools and counters. To make sure that the algorithm was efficient we checked the profiling information to make sure the distribution did not take too much time.

Counters that were used to keep track of processed requests and responses were inspected and compared to processor availability statistics as reported by NWS. The idea was to make sure the statistics were reported properly by NWS and to check that the distribution of requests used the statistics properly. This was done by inspecting the periodic output from the prototype.

5.4.5 Data Flow

The data flow in the prototype was analyzed through code inspection and with the help of profiling information.

6 Results & Discussion

In this section we describe the results of the analysis of the prototypes, the problems and the system in general.

6.1 Problems

We used three different problems for the analysis of the prototypes; spell checker, find route and null problem. The problems were introduced in section 3.3.1. This section discusses the results.

6.1.1 Spell Checker

The spell checker was based on Ternary Search Trees. The algorithms searches for a word in the dictionary and suggests alternative words. The suggestions are based on partial match or near-neighbor searching. Matching a word in a Ternary Search Tree has a complexity on the number of comparisons required of $O(\log n+k)$, where n is the number of strings in the dictionary and k is the length of the word to look up.

The time it takes to look up a word depends on the word and the dictionary used. A larger dictionary affects the time, but also the word that is sought. More time-consuming than the matching is however the search for alternative words. We allowed for up to ten alternative suggestions in our application. Some words that consist of many letters or have no near neighbors will end up in a sparse branch of the tree and result in few alternatives. These words are quick to lookup. Other words, often short, have many neighbors and finding alternatives for them takes more time.

We set the maximum possible word length to 20 bytes. Since we allowed up to 10 alternatives the maximum reply length was about 200 bytes.

6.1.2 Find Route

The find route problem was described in section 3.3.1. We used a data set of about 300 airports (vertices) and 30,000 flights (edges). In our algorithm decided to limit the number of flights a trip was allowed to consist of to 4 and a maximum of 100 possible trips were returned. Every flight was represented by 30 bytes, so in total the length of a result of a search could be 30*4*100=12,000 bytes.

The algorithm we implemented was not very time-efficient and it could have been greatly improved in this respect. Since the algorithm itself was not the interesting issue here, we did not do anything about this. However, the internal string handling would benefit from a rewrite, and a non-blocking send from the worker would also increase performance.

In the MPI prototype we attempted to use a more cooperative model for the find route problem. We divided up the airports into regions and put each worker responsible for one region. We had three workers and thus selected three regions. When a flight ended up in another region than the current one, the unfinished solution was sent to the responsible worker for completion. The coordinator was informed of this with the `RESPONSE_FORWARDED` message as described in section 4.1.2. Although this approach was interesting, the distribution algorithm used in the MPI prototype made it inefficient since it did not allow for the coordinator to distribute the request to the node responsible for the region, which caused many redirections.

6.1.3 Null Problem

The null problem was useful for testing the performance of the prototypes. We described it in section 3.3.1 but the basic idea is that the computation required to “solve” the problem is close to zero since it does nothing but writes some data to a response buffer. It allows us to configure any length for the response it produces which is used to test the maximum throughput that we can obtain.

We used the null problem with response sizes of between 1 byte and 18,000 bytes.

6.2 MPI Prototype

6.2.1 Throughput

One important performance measurement is the maximum throughput. This was measured by using the “null problem” (as described in section 3.3.1) and by varying the size of the responses. The result from this test can be seen in Figure 12. Requests of size 3 bytes are sent on the same link. One injector and 3 workers were used for the test.

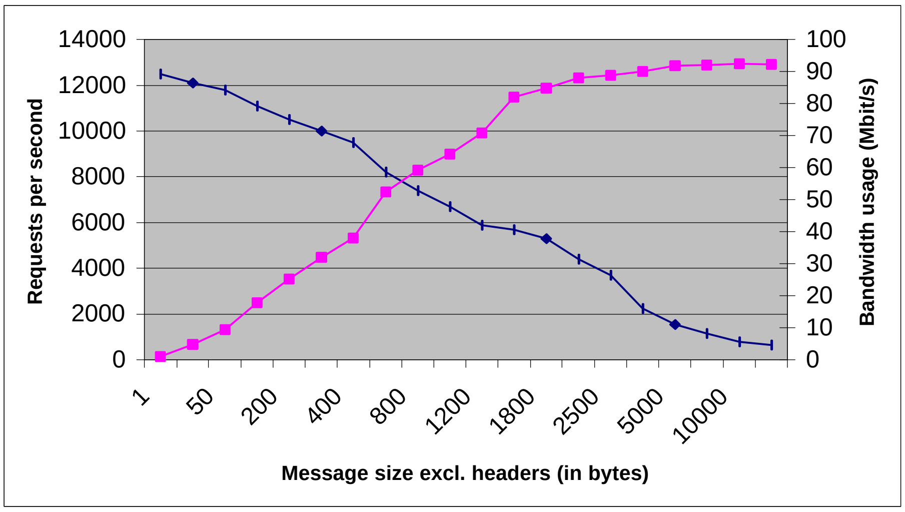

Figure 12. MPI Prototype: Message size vs. Requests per second and Bandwidth usage.

The horizontal axis in Figure 12 shows the response length used in the test. The decreasing graph represents the number of responses returned per second, which equals the number of requests serviced per second. We can read that the maximum is about 12,000 requests per second with one-byte (very small) replies, and it approaches zero as the size of the response increases.

The increasing graph represents the bandwidth usage of the actual data carried in the responses. We see that the graph starts a bit over zero and with very large messages it reaches over 90 Mbit per second. Headers and some data (e.g. we only count the response length in the figure, not the 3-byte requests or other packet data) are not accounted for which is why we can not reach 100 Mbit per second. With 1,000-byte responses the bandwidth usage is about 60 Mbit per second.

The reason for performing this test was to measure the efficiency of the data transfers and to get an upper limit on the number of requests that can be handled by the coordinator. This is important in order to understand the performance of the coordinator but it can also be used to predict scalability.

6.2.2 Spell Checker Performance

When running the spell-checking problem the number of requests per second reached about 9,000. Request size was 3 bytes and response size was about 50 bytes. Note that short words normally take longer to solve because there are a larger number of suggestions possible.

For this test we also used the database update feature of the prototype, which caused a reread of the wordlist about two times per second. This was likely the main reason for the loss of about 3,000 requests per second handled when we compare the results with the null problem with 50-byte responses.

After performing tests with the spell checker we concluded that the spell checking took very little processor time, something that made it less suitable for the MPI prototype and distributed computing in general. By collecting a batch of 10 requests and sending them in one package to the worker we were able to increase the throughput with a factor of 2 to 3. This verifies our intuition that the small requests hinder performance.

6.2.3 Logging and Log View

The logging of MPICH was enabled with the compile flag `–lmpe`. When logging was turned on the process used a lot of memory during the run and the performance was also significantly worse. When the processes had exited, the log files were merged and saved on one of the coordinator nodes. It could be viewed with Jumpshot, and Figure 13 shows a view were the prototype is running the find route problem with cooperation (described in section 6.1.2) enabled.

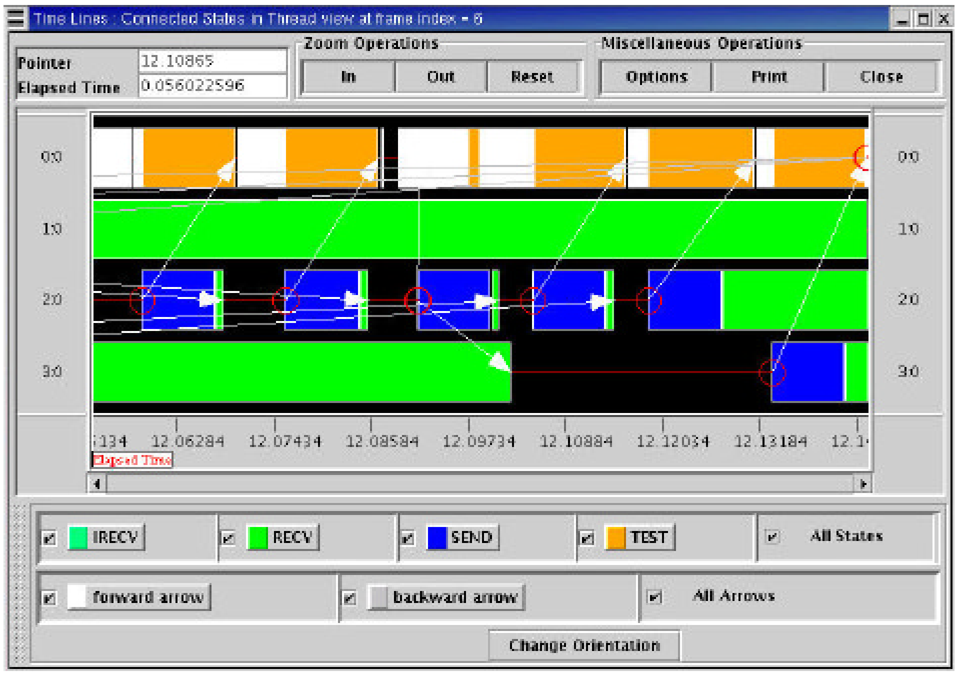

Figure 13. Jumpshot showing a sample run with find route and cooperation.

Figure 13 displays a horizontal time axis and the coordinator is the uppermost node. The colored boxes represent the MPI function that the process is currently executing. For example green means that the process is in receive, and blue means that the process is sending. The black area represents time that is not spent in MPI functions, so this is where the other things such as problem solving or other functions are performed.

Although the logging might not be perfectly accurate, it does give a good understanding of the behavior of the application and how it works algorithmically and performance-wise.

6.2.4 Request Distribution Algorithm

The MPI prototype used a request-driven distribution. This means that when a worker sends a `RESPONSE` or `WORKER_AVAILABLE` message, the coordinator sends the next request to this worker. This algorithm is simple and the worker will never be overloaded because it sets the pace of the reception of requests itself.

The main concern with this algorithm was that especially for small requests or problems that take little time to solve, this algorithm would not provide very good performance. This turned out to be true. This can also be seen in section 6.2.1 where small requests use very little link bandwidth and are thus likely not performing well. On the other hand with requests and responses that require larger transfers and especially if the problems are more difficult and CPU consuming, this self-regulating algorithm should, although it is very simple, provide decent performance.

We also noticed an issue with MPICH over TCP: when more than one worker has sent a response and we do a `MPI_test()` it seems that MPICH will always return the worker with the smallest rank. This seems to originate in a piece of MPICH code where a `select()` is done on the socket file descriptors and the data is read from the first file descriptor available, causing unfair behavior.

6.2.5 Data Service Algorithm

The data service was used with the spell checker. In this case the algorithm worked without problems, mainly due to the fact that the problem had few requirements. The update size was of acceptable (a few kilobytes), no synchronization was required between the workers and the update itself did not consume much processor time so we could keep the simple update strategy.

The file was located on a drive mounted via NFS. This will result in the file being transferred over the network when first read, but subsequent reads will likely use a locally cached file. Also the complete file is reread while a real application would likely only update the parts that have changed, if possible.

It is easy to find situations when the algorithm might prove to be too simple. If there are performance issues regarding the update size or update periodicity the problems could perhaps be addressed and solved within the scope of the current solution. Many interesting applications would however require a more controlled or synchronized data update that would likely require a more advanced solution. A system based on Dynamic Quorums (see for example [7]) could be one possible solution that would provide this functionality.

6.2.6 Data Flow

The data flow of the MPI prototype with a single buffer as seen in Figure 7 was not efficient. The design worked because we used blocking communication but this also made use dependent on the network communication buffering of the operating system, something that is not good style in MPI programming.

6.2.7 MPI Prototype Summary

The MPI prototype is not really efficient for applications like our spell checker. The main reason for this is that the small requests are less suitable for the distribution algorithm used and would be better suited for a system that makes uses more queuing.

Problems that require more processor time should behave decently with the MPI prototype. The main bottleneck seems to be that it can not handle large numbers of requests per second, but with fewer requests per second with a problem that is bounded by processor time and network bandwidth the system should behave quite well. Problems that require more processing power should scale nicely to a larger cluster.

6.3 Socket Prototype

6.3.1 Throughput

The network throughput test with the socket prototype was performed the same way as with the MPI prototype. We used the “null problem” to generate responses of specific lengths did this a number of times resulting in Figure 14. The test was run using requests of 3 bytes with a single injector and 3 workers.

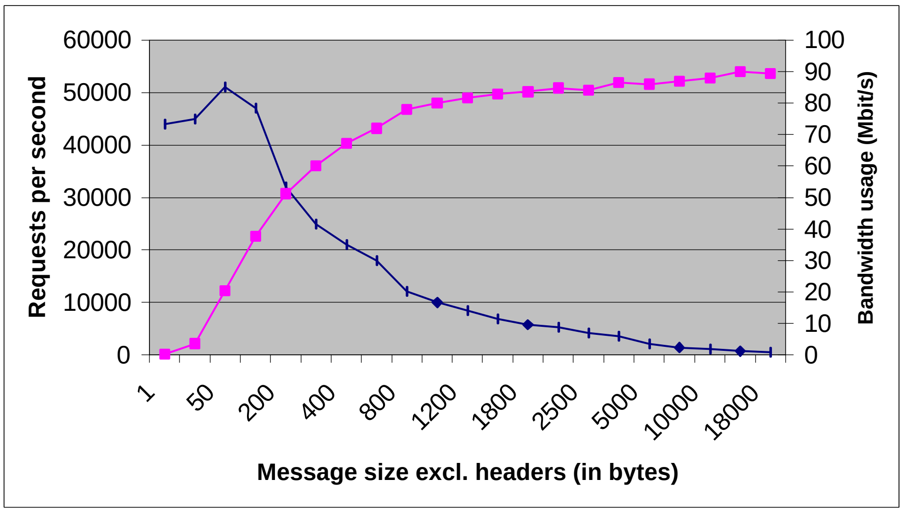

Figure 14. Socket Prototype: Message size vs. Requests per second and Bandwidth usage.

As we see in Figure 14 along the decreasing graph the request or response throughput reaches up to 50,000 requests per second with small response lengths. With 1,000 byte responses the throughput is about 10,000 response per second, a bandwidth usage of about 80 Mbit per second compared to 60 Mbit per second for the MPI prototype. With very large responses we use about 90 Mbit per second which should be about the peak bandwidth.

6.3.2 Scalability with Find Route

We performed some scalability tests with the find route problem. Find route was selected because it is, opposite to the spell checker, very time-consuming and it also potentially produces larger responses. We selected to test with a request that produces a response of about 10,000 bytes to simplify transfer rate calculations. The request (not included in transfer rate calculation) had a length of about 50 bytes. Figure 15 shows the results of the scaling test.

Number of workers	Requests handled per second	Transfer rate (responses of 10,000 bytes)
1	410	32 Mbit/s
2	820	65 Mbit/s
3	1000	80 Mbit/s

Figure 15. Scalability from one to three workers running the find route problem.

According to Figure 15 one worker should be able to handle about 400 requests per second, and we see that this is true for two workers as well. When we scale up to three workers the bandwidth seems to hinder higher throughput.

We also tested varying the number of traffic injectors between one to ten injectors running on the same hosts, but it showed no major difference on the throughput.

6.3.3 Non-blocking Communication

The non-blocking communication appeared effective for increasing the performance, considering the throughput figures of section 6.3.1. One drawback with non-blocking communication was that it required a lot of code to handle the buffers, and the increase in buffers will of course increase the memory footprint of the process.

6.3.4 Request Distribution Algorithm

The socket prototype used NWS (described in section 2.6.3) to get CPU resource availability information from the workers. Each worker was thus equipped with a NWS sensor (see worker setup in section 4.2.2) that reported CPU availability to the forecaster located on the coordinator.

The information provided by NWS was used as weights in the weighted round robin distribution algorithm. Round robin uses very little resources and the use of resource information to set weights worked well. The idea behind this was that if we did not know what applications were running on the nodes in the cluster, using an independent application to monitor the resources would enable use to make better decisions.

One issue we found was that NWS sometimes did not report new resource availability values properly, which sometimes led to non-optimal settings of the weights. It would be possible to check timestamps to determine when this happened. Most other problems concerning weights and measurements could be eliminated by using constraints and making sure the weights are sane.

6.3.5 Data Flow

The data-flow design with multiple buffers probably had a lot to do with the performance increase when compared to the MPI prototype. The multiple buffers were what made the non-blocking communication possible.

The drawback of using multiple buffers was that it resulted in lots of more code and a more complex application.

6.3.6 Socket Prototype Summary

The socket prototype provided good performance with large number of requests and one benefit is that it was able to process many data streams simultaneously. The throughput when using small requests and responses proved that the distribution mechanism was, as expected, more efficient compared to that of the MPI prototype. The big difference here was likely not the buffering system and the non-blocking communication, but also the fact that requests were fed to the workers rather than asked for. This increased buffering on both the coordinator and the worker, as well as limited the impact of the network latency.

Overall the socket prototype was better suited for our application since it used a rather simple communication flow. However, it lacked the access to MPI features, which greatly limits the type of applications it could be used with.

6.4 General Discussion

6.4.1 Profiling

Profiling was used throughout the development of both prototypes to make sure the functions worked properly and did not steal processor time unnecessarily. The workers should ideally spend most of their time solving problems while the coordinator should spend most of its time shuffling data. Other functions should only account for small amounts of processor time. Figure 16 display were the prototypes spent most of their time. We only show the most time-consuming functions.

	MPI prototype	Socket prototype
Worker	1. Send, receive and wait 2. Problem solving	1. Problem solving 2. Send and receive
Coordinator	1. Send, receive and wait	1. Loop and select 2. Send and receive

Figure 16. Profiling results for MPI prototype and Socket prototype. In the boxes we order the most time-consuming operations when running the spell checker.

The coordinators in the two prototypes have about the same profile. The main difference is that the MPI coordinator spends a lot more time waiting for new requests than the socket prototype.

For the workers the socket prototype spends more time solving problems than communication, while with the MPI prototype it is the other way around. This is because of the distribution: the socket prototype will almost always have requests to read quickly from the local network buffer, while the MPI prototype will have to send its response and wait for a new request.

6.4.2 MPI Usage

MPI provides a lot of functionality and is very useful when implementing a distributed application, mainly because it provides a simple and consistent interface to message passing between the different nodes. If applications have special requirements or can take advantage of advanced group communication routines such as barriers, broadcast, gather and scatter, MPI will be even more useful since it efficiently implements these features. Although we did not use it, the non-blocking communication system and buffering of MPI would probably provide both good performance as well as a nice programming environment.

Our prototypes did most of their communication between the coordinator and a worker (except the cooperating find route), and we used a very small set of the functionality of MPI. Our prototype also did not need any advanced message types provided in MPI, so the unused parts of MPI was mostly a cost of data overhead and more logic to process.

The fact that the MPI version we used did not provide us with the possibility to do fault-tolerance was also a problem since it would be a very nice feature for the application. In the socket prototype when we had control over the sockets, we were able to implement a simple fault-tolerance system that would at least allow workers to join and leave.

6.4.3 Socket Usage

Using sockets directly causes some extra work when writing the communication system. The main benefit is the complete control over the sockets and communication system, and we should thus be able to take better advantage of the resources. This requires knowledge and experience of programming sockets on a low level. If we required communication between workers, the problem would immediately grow in complexity, and in this case MPI is probably well worth the overhead that it compensated with simpler programming.

The use of sockets directly removes the packaging of messages done by MPI. This reduces the processor resources used for each message sent and received, and also the size of the message. With very little message data the packaging information might be the significant part of the message.

6.4.4 Scalability

We performed some simple scalability tests but what it proved to us was mostly that the bandwidth quickly became the bottleneck with our problems. The positive thing is that the number of requests per second should be sustainable for the coordinator when using more workers, in which case a more difficult problem than what we used would scale rather nicely up to more workers. In this context it would also be interesting to test faster networking technologies.

6.4.5 Monitoring

The NWS monitoring architecture is built as a general tool that should be usable in many different environments. Setting up and using NWS was not painless but the design made sense. Our main concern was whether the measurements from NWS that we used as weights in our weighted round robin algorithm, were actually the best thing to use as weights. The performance of the application is likely to depend not only on the CPU reports but also on other parameters such as what other applications are running on the monitored computer.

One alternative that came up was to do the monitoring in the prototype itself instead and also send results with the same communication system as the other application data. In this case we motivated the use of NWS with the value of having an independent system for monitoring that was not bound to a specific application.

6.4.6 Networking Technologies

There are other networking technologies that provide better performance, but these are usually not as common or as cheap as Ethernet. If the application uses heavy communication or has large data requirements it might be necessary to investigate other technologies further. Storage Area Networks (SAN) often use technologies such as Firewire to increase performance, and it is also possible to improve the latency of the network.

The IP protocol stack causes work for the processor which inhibits computation performance and increases latency. Using a dual-processor machine where one processor runs the application and the other takes care of other jobs such as I/O and operating system calls might be a solution.

6.4.7 Accuracy

The fact that the figures stated have been found on a set of computers that have been set up the same while performing the test allows for comparison. In a different environment the figures will change but we believe that the conclusions we make from the figures are valid. Since we did not have an existing implementation for comparison we have not focused the analysis on individual figures.

7 Conclusions

The intention of the project was to investigate the use of distributed computing for business applications. After the initial investigation we realized that we needed to develop an application that we could evaluate. The application we developed was the task distribution engine that would allow us to distribute a single-host application to multiple hosts with a single point of entry.

7.1 Problem Types

The aim of the project was not to solve a specific business problem but to do a more general evaluation. The problems we selected, spell checking and find route, are thus examples.

As we saw in the analysis the spell checker did not really perform efficiently when distributed. The major problem was that the algorithm used for spell checking was too fast, which caused the cost of distribution to be higher than the gain.

The find route required a greater amount of computing resources, and the results were also larger, a few kilobytes. As our scalability test showed, the find route problem could actually take advantage of the distribution. By increasing the number of computers available the throughput increased. The test also showed that even for find route, three nodes would saturate the link bandwidth.

The find route that took advantage of cooperation between the workers was more a quick test than a thorough investigation. It showed that the implementation of this feature that we did was a bit too easy and did not perform that well neither in throughput or in behavior. On the other had it did show us that it was a viable technique and with the correct problem and in the correct environment it could be successful.

The problem the distributed system is used to solve will set the limits on how the system should be implemented and how much infrastructure cost is tolerable. It is clear that adding one computer processor is a big resource and its added performance must be used. There surely exists a number of interesting problems that can greatly benefit from distributed computing, and we are sure that as the technology is being accepted on a wider base the usage for many business problems will increase.

7.2 Cluster Hardware

The grid cluster we used in the project consisted of 4 PC computers. This is a rather small setup and the intention was to consider about 30 computers, a significant resource. By using cheap hardware and open-source software such as Linux and the other packages installed, the cost of the grid is easy to calculate. If more performance is needed it is perhaps also interesting to look at dual-processor machines.

With a larger grid there is also more problems with administration. This requires both knowledge of how to administer such a set up, but also perhaps some extra hardware such as console and screen switches. With a larger cluster it might also be necessary to use some type of cluster management tool, something this project did not look into.

7.3 Message Passing Interface Usage

MPI was used because we wanted to use standard tools that had a proven success record. The MPI toolkit provided quick, easy and standard-compliant development of distributed applications. Although we only used a few of them, MPI comes with a large number of features that can be used to simplify programming, improve performance and produce quality programs.

For the applications we looked at, one of the drawbacks was the lack of support for a dynamic environment where nodes could join and leave. This is a rather complex function but since we considered business applications that require uptime and serviceability it is a major problem. We are happy that this functionality is added to the MPI-2 Standard and hope to see it implemented in toolkits soon. Since fault-tolerance is one of the important benefits of a distributed system we consider it very important for the acceptance of MPI and distributed computing in business environments.

Having a standardized toolkit is very important if grid computing is to be widely spread, and we believe that MPI is a good basis for this. However we think that often the use of MPI in its current form is rather low level and we would encourage toolkits on top of the MPI Interface that provide an even more usable environment.

7.4 Socket Usage

Socket programming increases the control over the communication, but it is also more difficult and causes complex code. Sockets do provide high-performance and safe communication when used properly but it is difficult to master the system to take full advantage of it. Still many distributed applications are written using these low-level communication tools because they provide full control over the system and the developer can do exactly what he wants.

As mentioned before, programming sockets makes it very difficult to provide group communication that is often required or practical in a distributed computing environment. Applications that use group communication require a great deal of logic, logic that is usually provided by a toolkit such as MPI.

In our socket prototype we saw greatly improved performance compared to our MPI prototype. However, we believe that with some optimizations and a nice buffering architecture in the MPI version we would be able to match the performance since MPI does provide features both for buffering and non-blocking communication.

7.5 Resource Availability with Distribution

In a grid cluster of decent size we consider it likely that there might be a number of different applications running at any time that use the cluster. If we don’t know exactly what applications are running, where they are running and how they behave, a system that monitors the processor usage and other parameters might prove very useful. This information could likely be used to provide better service to an application by allowing it to use the least-used nodes in the cluster.

In this project we used the resource availability information to control our task distribution. There are a number of parameters involved here that make the problem quite difficult such as other applications, the resource reports, how measurements are made, the service requirements of the application and the setup of the nodes. To successfully provide a distribution, based on the resource availability numbers we get on problems with the small granularity of our prototype requests, is very difficult.

Using a resource monitor is very important, but it might be better to use it on a higher level, perhaps to select which set of nodes are to be used (or reserved) for an application.

7.6 Network Usage

The performance of the processors on the nodes in the cluster was very good. This resulted in problems being solved very quickly and thus the network bandwidth became an issue. This verified our intuition that it is valuable to be able to perform smart but computationally expensive functions on the nodes, and to try to limit the network resources required.

It will probably not take that long before there is a shift from current 100Mbit/s Ethernet to 1Gbit/s Ethernet or other technologies. Current

Storage Area Network solutions (quickly mentioned in section 6.4.6) often use other technologies to provide better performance, but these are still rather expensive and would increase the cost of the cluster.

7.7 The Next Step

Distributed computing is very interesting and a lot of developments in the area will take place in the near future. There are many interesting aspects that require research and development, and many organizations and companies are involved in this.

One interesting area is the construction of a more complete grid cluster setup that would allow for the running of multiple applications and application types. This problem is somewhat being issued by for example the Globus toolkit but many things remain to be done.

Extraction and use of monitoring information is also an area in need of further investigation. Further investigations about what type of monitoring information that should be used to predict the performance of a specific application would be interesting to enable more accurate measurements and predictions.

Perhaps the most ambitions but also interesting research area would be the development of autonomous clusters that run applications, self-heal and provide services to users safely and transparently. This are is heavily researched and will be very important in future computing systems.

8 References

8.1 Publications

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cli_s, NJ, 1993.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. *Users' Guide to NetSolve V1.4.1.* Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[3] G. Ausiello, P. Crescenzi, G.Gambosi, V. Kann, et al. *Complexity and Approximation*. Springer, 1999.

[4] Roberto Baldoni, Michel Raynal. *Fundamentals of Distributed Computing...* http://dsonline.computer.org/0202/features/bal_6.htm

[5] Jon Bentley and Bob Sedgewick. *Ternary Search Trees*. Dr. Dobb's Journal April 1998 http://www.ddj.com/documents/s=921/ddj9804a/9804a.htm

[6] Anthony Chan, William Gropp, and Ewing Lusk. *User's guide for* `mpe` *extensions for mpi programs*. Technical Report ANL-98/xx, Argonne National Laboratory, 1998. ftp://ftp.mcs.anl.gov/pub/mpi/mpeman.ps.

[7] Randy Chow and Theodore Johnson. *Distributed Operating Systems &Algorithms*. Addison-Wesley, 1997.

[8] Ian Foster, Carl Kesselman, and various authors. *The Grid: Blueprint for a New Computing Infrastructure.* 1st edition. Morgan Kaufmann Publishers, November 1998.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. http://www.netlib.org/pvm3/book/pvm-book.html

[10] Kenneth W. Neves. *Industrial "Power Grid" Computing: The Next High Performance Challenge*. http://www.atip.or.jp/ts/neves.html

[11] Norbert Sensen. *Algorithms for a Job-Scheduling Problem within a Parallel Digital Library*. Department of Mathematics and Computer Science University of Paderborn, Germany. http://www.uni-paderborn.de/cs/sensen/Scheduling/icpp/icpp.html

[12] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. *MPI: The Complete Reference*. http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

[13] Rich Wolski, Neal Spring, and Jim Hayes. *The network weather service: A distributed resource performance forecasting service for metacomputing.* Journal of Future Generation Computing Systems, 15(5-6):757-768, October 1999.

8.2 Internet Websites

[14] *Google conference directory*. http://directory.google.com/Top/Computers/Parallel_Computing/Conferences/2002/

[15] *distributed.net*. http://www.distributed.net/.

[16] *Folding@home*. http://folding.stanford.edu/

[17] *The Globus Project*. http://www.globus.org/

[18] *Grid Computing Info Centre (GRID Infoware)*. http://www.gridcomputing.com/

[19] *GridEngine*. http://gridengine.sunsource.net/

[20] *High Performance Fortran Forum*. http://www.crpc.rice.edu/HPFF/home.html

[21] *An InfiniBand™ Technology Overview.* http://www.infinibandta.org/ibta

[22] *LAM/MPI*. http://www.lam-mpi.org/.

[23] *The Linda Group*. http://www.cs.yale.edu/Linda/linda.html.

[24] *Linux Cluster HOWTO*. http://www.tldp.org/HOWTO/Cluster-HOWTO.html

[25] *mEDA-2: A Virtual Shared Memory for PVM*. http://vvv.it.kth.se/labs/cs/meda2/

[26] *Message Passing Interface Forum*. http://www.mpi-forum.org/.

[27] *Migration and Integrated Scheduling Tools for Concurrent Processing Environments on Multiuser Heterogeneous Networks (MIST)*. http://www.cse.ogi.edu/DISC/projects/mist/

[28] *MOSIX*. http://www.mosix.org/.

[29] *MPICH-A Portable Implementation of MPI*. http://www-unix.mcs.anl.gov/mpi/mpich/.

[30] *MPICH-G2*. http://www.hpclab.niu.edu/mpi/.

[31] *MPI/PRO*. http://www.mpi-softtech.com/products/mpi_pro/

[32] *MVICH – MPI for Virtual Interface Architecture*. http://www.nersc.gov/research/FTG/mvich/

[33] *Myricom*. http://www.myri.com

[34] *Network Weather Service*. http://nws.cs.ucsb.edu/.

[35] *PARMON*. http://www.csse.monash.edu.au/rajkumar/parmon/index.html

[36] *SCALI*. http://www.scali.com

[37] *SETI@home*. http://setiathome.ssl.berkeley.edu/.

[38] *Virtual Interface Developer Forum*. http://www.vidf.org/

[39] *XPVM*. http://www.netlib.org/utk/icl/xpvm/xpvm.html.