**Richard Doyle**

**BSc (Hons) Software Engineering Management**

**Final Year Dissertation**

**2000**

**Bournemouth University**

# A Framework for Interactive Data Parallel Processing on a Network of Workstations

## Abstract

Parallel processing using networks of workstations is becoming an increasing important area within high performance computing.
However the development of interactive data parallel applications for the environment is currently not well supported.

This project investigates the complexities of implementing interactive data parallel applications on a network of workstations. In particular the development of a parallel volume rendering application that provides visual progress of the rendering will be investigated. From the experience gained a framework is proposed to reduce the time spent in implementing other interactive data parallel applications that require user feedback.

From the analysis of the performance results using the framework, it is shown that 'superlinear' speedup is obtainable for such applications.

## Acknowledgements

I would like to thank my two supervisors Jon Macey and Mike Jones for their guidance throughout this project, and my family for helping me keep my sanity during my final year.

# Contents

# 1 Introduction

## 1.1 Overview

Parallel processing using networks of workstations is becoming an increasingly important area within high performance computing. It is viewed favourably as it can provide supercomputing performance, but at much less expense. With the ever-increasing performance of commodity computer and network hardware, coupled with the decreasing cost in both, the utilisation of computers on networks is viewed as a key area in current and future parallel processing. (Das and Das, 1998; Liu et al 1999; Meyer et al 1997; Warren et al, 1997).

Bournemouth University has a large number of powerful UNIX workstations and PC's. However it has been noted that these resources are often left unused for long periods of time (especially overnight and weekends). If these computers can be used as a single system, then the collective power can potentially offer supercomputer performance at no extra cost.

However the development of programs for these environments requires different skills than sequential programming and is considered a non-trivial task (Steenkiste, 1996). Furthermore the development is far from clear, and there is no universal agreement on how applications should be created.

Although many programming tools do exist for networks of workstations, they are not ideal for the development of interactive graphical applications. Low level programming tools provide encapsulations around communication, but provide little help in the development of applications. High level tools encapsulate the parallel computation, hiding the progress of work, and therefore providing no user feedback until the whole problem is complete.

## 1.2 Aim

This project aims to develop a framework to support the implementation of interactive data parallel applications for networks of workstations, that allow graphical applications to provide feedback on the progress of work.
To do this the environment will be investigated, and characteristics that effect the design of applications identified.
Using this knowledge an interactive parallel volume rendering application will be designed and implemented for the environment, so that problems can be identified in the creation of applications.
From the experience gained in the implementation, a framework will be created to reduce the time required to develop similar interactive data parallel applications for networks of workstations.
The performance of applications written with the framework will be analysed to see if it provides similar performance to custom written applications, and how it compares to sequential processing.

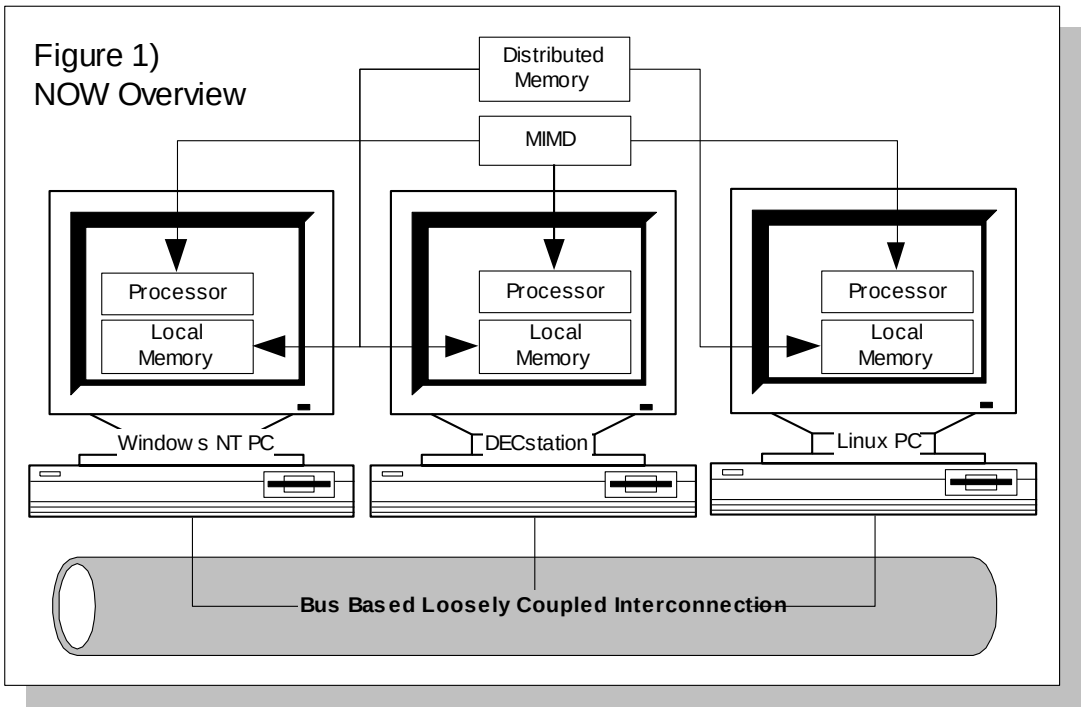
## 1.3 Objectives

1) Develop an understanding of the implementation environment and how it can effect the design and implementation of efficient parallel programs.
2) Investigate approaches to parallel programming for the environment
3) Investigate the design and implementation of a parallel volume rendering application.
4) Identify areas in which a framework could be used to reduce the time spent in development.
5) Develop a prototype framework to address some of these issues
6) Analyse the frameworks computational performance and usability.

# 2 Implementation Environment

## 2.1 Overview

This chapter intends to investigate the characteristics of networks of workstations (NOW) that effect the development of parallel applications (both problems and benefits).
The specific architectural characteristics of the intended implementation environment are not considered here, as the program design would ideally support a large range of environments. However the generic network of workstations (NOW) model proposed by Anderson (1995) has been used as it provides the best match to the intended environment. These systems have a number of characteristics (Figure 1) that can effect the design of efficient parallel programs and these are considered below.




## 2.2 Multiple Instruction stream Multiple Data stream (MIMD)

This basic classification (Flynn, 1972) essentially defines the system as a collection of independent processors, each capable of following their own flow of control, executing their own instructions on their own data.
The parallel implementation of a problem for such an environment must be partitioned into segments that can be executed concurrently by the separate processors. Furthermore as each processor follows its own flow of control, some form of synchronisation is likely to be required in the program.
Programs written for such machines can emulate all of Flynn's (1972) classifications by using software abstractions. This approach can reduce the complexity of parallel program design and implementation, and the Single Program Multiple Data (SPMD) approach is often used for Single Instruction stream Multiple Data stream (SIMD) processing.

## 2.3 Distributed Memory

The distributed memory on NOW's is physically partitioned among the processors, and each processor can only directly access their private local memory. Communication is made by sending messages between processors through the interconnection. The time required sending a message is dependent on the interconnection but is usually an order of magnitude slower than local memory access.
Program implementations (and therefore designs) should try to minimise communication between processors by partitioning problems to gain the maximum use of local memory (Foster, 2000).

## 2.4 Loosely coupled interconnection

The high latency of the interconnection on loosely coupled architectures like NOW's restricts the frequency of task synchronisation in efficient programs to a courser grain than is possible in tightly coupled architectures (Das and Das, 1998; Leutenegger, 1997). Furthermore the comparatively low bandwidth within NOW's restricts the possible efficient implementation of data intensive applications to those where the computational requirements sufficiently outweigh the time required for communication (Steenkiste, 1996).
Squyres et al (1998) identifies that the time required to send smaller messages is dominated by latency, while the time required for larger messages becomes linearly proportional to the message size.

Time = latency + message size/ network bandwidth

From this they conclude that parallel programs 'should strive to minimise the number of communication operations, and at the same time maximize the size of the messages sent'.
However this is not strictly true, as the communication time is only a concern to the process if blocking communication is used or further processing is not possible until the communication is completed (often in synchronisation). Also the message size should not be maximised, as they have identified it increases the communication time.
However with blocking communication it can be quicker to send several agglomerated messages in one operation as a single message, as the communication is delayed by at least twice the latency (send and acknowledgement).
Finally, it can be better to reduce the number of communication operations, as communication abstractions (if used) can require substantial computation in sending messages.

## 2.5 Bus Based interconnection

The bus based interconnection of NOW's means that all computers share a single interconnection (not switched) and therefore share the bandwidth (with further problems for data intensive applications). Furthermore if many processors simultaneous communicate with one another (especially massively parallel NOW's) the bus-based network can become saturated.
Network bandwidth can be saved if messages can be divided without remainder into the network packets (and therefore not sending empty space). This is really only a concern where small messages that are greatly under the minimum packet size are frequently used. In such situations the agglomeration of the messages can result in much less wasted space in the network packets and therefore a saving in network bandwidth.

## 2.6 Non-dedicated (to parallel processing)

Where local workstation processes are not to be slowed by parallel processing, careful dynamic scheduling and process migration (Leutenegger and Sun, 1997; Steenkiste, 1996) is required to achieve good performance (although the performance still depends on workstation load). However the scheduling requires estimates of workstation load and task computational requirements that are often not available to any accuracy (Du and Zhang, 1997). An alternative and simpler approach is to only use free machines for the parallel processing or to allow standard workstation processes to be slowed.
For comparison parallel programs for dedicated machines can often apply static load balancing if accurate estimates for task computational requirements are available. Also dedicated systems can benefit from many tweaks and enhancements to the environment that would not be acceptable on workstation orientated systems (Anderson, 2000; Bruckdal, 1997; Kung et al, 1991). A particular category of note is 'Beowulf' clusters that are dedicated systems made from low cost commodity computers and software (Hoffman and Hargrove, 2000; Sterling et al, 1998; Warren et al, 1997).

## 2.7 Heterogeneous

Programming for heterogeneous systems is often seen as a key requirement in the development of parallel programs (Das, 1998; Squyres, 1998; Steenkiste, 1996) as it provides a much larger range of environments to run the program.
However parallel programs for heterogeneous environments have to contend with different data representations (little and big endian for example). Programmers should seriously consider using communication libraries to provide uniform access (Steenkiste,1996). The task scheduling can be more complicated as each processor can have a different computational power, and the communication latency/bandwidth to each can vary.
In a heterogeneous environment with greatly differing architectures (e.g. mixing MIMD and SIMD) only functional decomposition of the problem may be appropriate as certain machines may be better suited to (or only able to process) certain tasks. In such environments it is possible to gain 'super-linear' speed up (higher than the

individual performance of all nodes) as tasks can be distributed to machines they are suited for (Donaldson, 1994). However in general, heterogeneous environments provide worse performance than homogenous environments (Clematis et al ,1997).
For comparison, clusters of workstations (COW) are usually homogenous (Leutenegger and Sun, 1997) or possess only weak heterogeneity (processor's differ by computational power only). These environments can gain higher performance by adapting to make use of specific software or hardware (Anderson, 2000; Chiola, 2000; Warren et al, 1997). The optimal power available from a homogenous system is the sum of the performance of each individual component (linear).

## 2.8 Summary

This chapter has investigated the characteristics of a network of workstations and identified problems that parallel processing must take into account in order to gain good performance. Below is an overview of the factors considered in this chapter:

2.8.1 Partitioned problem into separate sequential segments
2.8.2 Identify communication between tasks
2.8.3 Maximise local memory use
2.8.4 Minimise synchronisation frequency
2.8.5 Minimise data transfer
2.8.6 Minimise communication operations
2.8.7 Agglomerate small messages
2.8.8 Consider Task Scheduling (Static or Dynamic)
2.8.9 Support heterogeneous communication

This has provided some knowledge of the implementation environment, but the development of parallel programs is still unclear and is considered in the next chapter.

# 3 Parallel Programming Overview

## 3.1 Overview

This chapter aims to give a very brief introduction into the high level approaches available for the development of interactive data parallel programs on networks of workstations.

## 3.2 Sequential vs. Parallel Code

Ideally parallel code could be automatically translated from sequential code. Unfortunately most automatic tools that convert sequential code to parallel concentrate on finely grained computation (in particular loop parallelization). Unfortunately fine-grained parallelism requires frequent synchronisation that has been identified as unsuitable (2.8.4) for the networks of workstations.
Some newer compilers use subsets of languages (C without pointers or structs) to identify course grained computation (Das, 1998), but these are still only research tools and currently have little practical testing.
Current automatic tools only translate the sequential code into parallel equivalents without actually 'understanding' what the problem is. However sequential programs often imply control and flow constraints the are not necessary for the completion of the problem. In many cases a new parallel approach will provide much better performance, and at present this requires that a programmer develop a new parallel program from the problem knowledge.

## 3.3 Parallel Programming Models

The development of parallel programs cannot follow the Von Neuman machine model and 'at present, the parallel computing community does not agree upon a unifying and equivalent abstract model for parallel programming' (Aggarwal and Chillakanti, 1996).

High level parallel languages exist for the development of data parallel applications. The Parallel C++ (pC++,1994) library supports parallel programming for homogenous NOW's through the Parallel Virtual Machine (PVM) communication abstraction. However the language completely encapsulates the parallel processing, and therefore user feedback during the processing cannot be made.

Shared memory abstractions encapsulate the distributed memory and therefore provide a desirable single system image. However efficient implementations on NOW's require explicit allocation of data objects and tasks to processing nodes. This however breaks the shared memory abstraction, and is viewed by many computer scientists as a step back from the message passing programming model (Sterling et al, 1998).

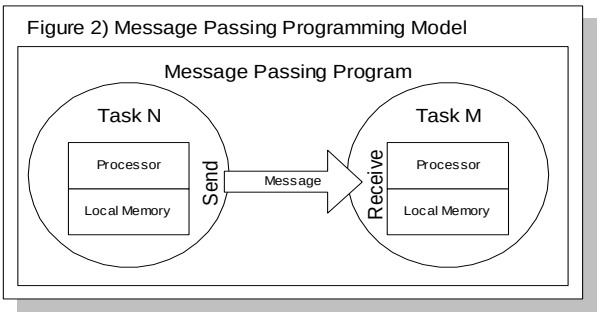
Parallel Object Orientated approaches provides a useful design approach as the encapsulation of data and functions allows easy migration of tasks. However concern has been expressed on the performance gained using such techniques (Parsons, 2000).

The message passing programming model is currently a dominant and well supported approach for parallel processing on network based multicomputers (Foster, 2000; Ragsdale, 1991; Steenkiste, 1996). It is particular well suited for NOW's as the model matches the hardware functionality available and programs can therefore obtain high performance. The model puts very few restrictions on the development of programs and the creation of a parallel program with user feedback is possible.

Due to its dominance in current parallel processing the message passing model was chosen for further development.

## 3.4 Message Passing Programming Model

With the message passing programming model (figure 2) a program is decomposed into a set of sequential tasks (processes) that can be executed in parallel. Each has a unique task id and encapsulated local data (private to all other tasks). Tasks communicate by explicitly sending and receiving messages using the unique task ids.




## 3.5 Summary

This chapter has very briefly reviewed how the popular programming models fit with the development of interactive data parallel applications.

The message-passing model was chosen as it is the dominant approach to parallel program design, and can be used to provide feedback to the user on the progress of parallel processing.

However although a programming model has been chosen, its actual implementation on a NOW is unclear. The following chapter investigates how and application can be developed using this model on the university computers.

# 4 Communication Investigation

## 4.1 Overview

The choice of communication library can drastically change the amount and type of work required implementing a message passing parallel program. Choosing a low-level abstraction or working at the hardware level requires a lot of effort to achieve the basic functionality but higher performance can be obtained as code can be optimised for parallel processing (Chiola, 2000). Higher level abstractions provide lots of functionality but perhaps with an overhead of code unnecessary for the parallel processing in the environment. Furthermore applications have to be moulded to fit the interface provided by the library, which may introduce more complexity than the library encapsulates.

A particular concern is support in the communication library for a heterogeneous environment (2.8.9). This is a concern as it requires significant time to implement, but it provides a much larger number of possible environments for applications to be used in.

An initial investigation was conducted into using remote procedural call (RPC) for parallel processing. It was chosen as it can provide a high level abstraction to communication and heterogeneous communication. Furthermore it is already present on all the university computers and therefore requires no additional effort in installation and configuring. However problems were encountered (explained in 4.2) and therefore alternative libraries were investigated.

When considering possible libraries for parallel processing the research was restricted to only relatively well tested software (relatively as most are still being developed). However it was intended that several freely available libraries (MPI, PVM, MultiRPC) obtained through the Internet would be investigated. Unfortunately the time required for the installation of these libraries in the implementation environment was underestimated. Problems were encountered, as the libraries required certain facilities or functionality that were not already in place. This functionality was easily set up in a dedicated prototype environment (only one machine), but only restricted access was available for the intended environment and workarounds had to be employed.
However the local area multicomputer (LAM) MPI implementation from the university of Notre Dame was successfully installed on 16 Unix workstations and program development investigated (4.3).

## 4.2 RPC Investigation

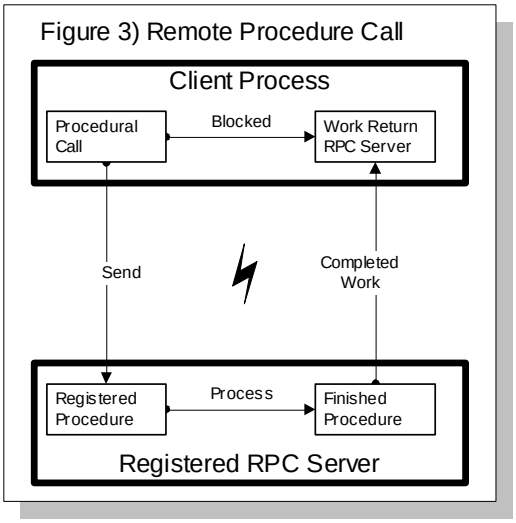


Figure 3) Remote Procedure Call

RPC provides a high level encapsulation of explicit message passing so that remote processing can be viewed in the same way as local subroutines. A client process calls a registered procedure (possibly local or remote) and the client is blocked until the server has completed the procedure (see figure 3). RPC supports communication between heterogeneous computers by using a standard external data representation (XDR).

RPC therefore offers a simple way in which remote servers can be used for processing. If a program could be written to call several registered RPC servers then a simple form a parallel processing would have been created. Unfortunately with standard RPC programming (with RPCGEN) the model is restricted to synchronous communication. This means the calling processes is blocked until the server has finished processing the procedure and therefore the calling process is only able to call servers sequentially. Multithreading would offer a relatively efficient solution, but unfortunately the standard RPC implementations are not thread safe.

Bloomer (1991) creates a distributed image processing application by forking separate child processes for each RPC call the client parallel application intends to make (figure 4). After each child has completed their assigned RPC call they carry out any actions required on the result and terminate. The master receives the exit flag through a registered call back routine and forks a new child to contact the free server. The data is displayed by having each child processes write the data directly to a shared X window.

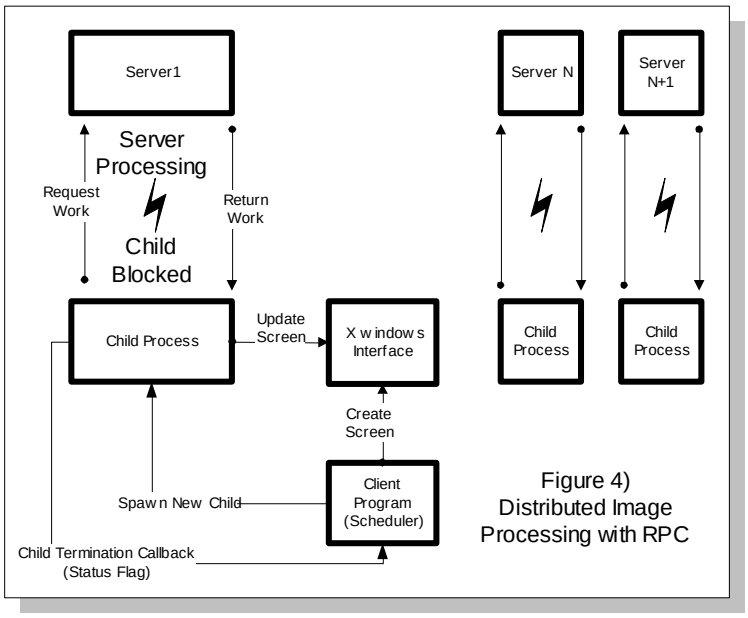
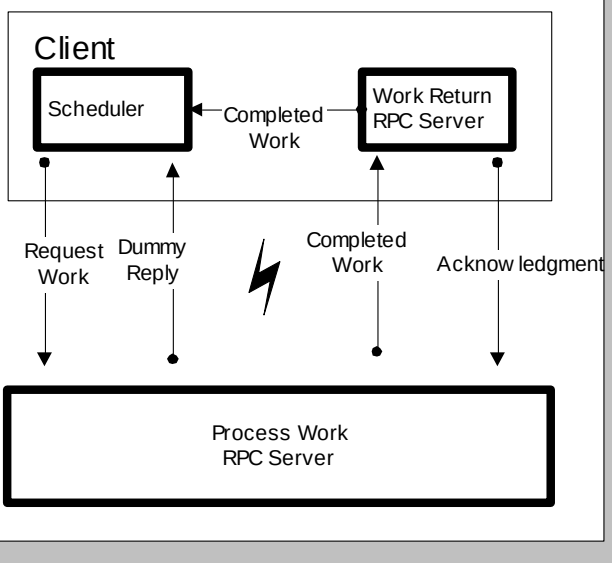


Figure 4) Distributed Image Processing with RPC

Although this approach could be used to produce an interactive graphical application, the writing of results directly to the X window is not desirable. Firstly it restricts the graphical part of the application to an X window environment and the return data must be appropriate for display. These problems can be avoided by having some local inter process communication transport the data back to the parent process.

An alternative approach is to have the user program act as both a client and server (figure 5). The user program makes a standard request to a server, but is given an acknowledgement reply and not the result. The server continues processing the request and when finished contacts the requesting client who is also acting as a RPC server. Unfortunately the implementation of a program to act as both an RPC server and client requires programming below the high level abstraction of RPC. This can be avoided by having the server and client as separate processes and using local inter-process communication to transport data between them.

Figure 5) Followup RPC




Although parallel processing with RPC is possible, it introduces complexity to the implementation of the application. This not only effects the time required for implementation, but also the performance of the application. It is therefore viewed that the benefits of using the library for this project do not outweigh the complexity it introduces to the application.

## 4.3 MPI Investigation

The first message passing interface (MPI-1) was proposed in 1994 by over 40 organisations as a standard message passing interface for MIMD distributed memory concurrent computers and is now considered the de facto standard for message passing (Foster, 2000; Humprey and Coghlan, 1999; Squyres, 1998; Warren et al, 1997)

The standard only defines a communication interface and does not define how the message passing is to be implemented, or even how parallel programs should use the interface. Various implementations of the MPI-1 standard have been created, with many making use of specific hardware to obtain higher performance. Other popular parallel libraries (PVM) are now providing wrappers to support the MPI standard.
Programs written for the MPI can run without modification on all MPI implementations, and therefore obtain very good portability. Also MPI implementations can support transparent communication between heterogeneous computers (although dependant on implementation), and support communication between different MPI implementations through Interoperable MPI (IMPI).

Unsurprisingly the message passing interface provides the simple functionality of the message passing programming model (and much more). The use of the MPI is slightly more complicated than the simple message passing model shows, and a very brief overview for some of the functionality is given in Appendix A. For a more comprehensive introduction to MPI the reader is directed to Snir (1996).

## 4.4 Conclusion

The MPI was chosen for further program development as it offers an excellent match of functionality to the message passing programming model and is extremely well supported. The local area multicomputer (LAM) implementation of the MPI was installed, as it was the only one that currently supported both Unix and Linux. The importance of the library supporting Linux was so that work could be attempted on a dedicated environment before being implemented on the restricted access university computers. This proved a very good decision as understanding the internals to the operating environment and library allowed workarounds to be developed for the installation of the library on the university computers.

## 4.5 Summary

This chapter has investigated the two libraries, remote procedural call (RPC) and the message passing interface (MPI). The RPC library was found to be unsuitable for use with the message-passing model, while MPI provides a superset of the functionality, but at an increased complexity.

Now that the development of message passing programs is clear, the design and implementation of the interactive parallel volume rendering application can be made.

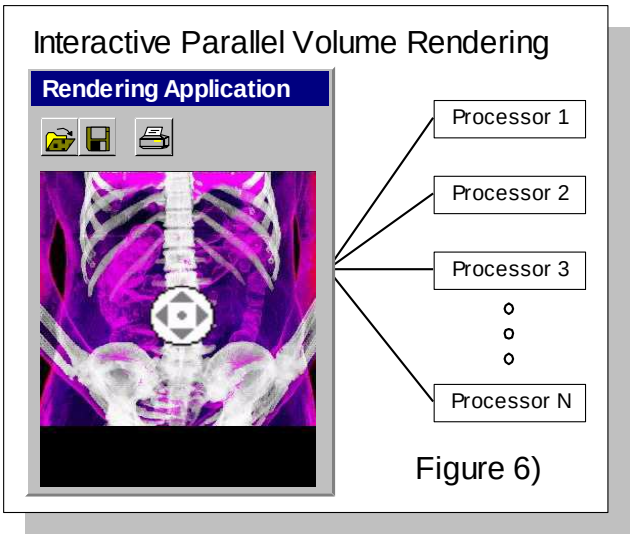
# 5 Parallel Volume Rendering Design

## 5.1 Overview

As with the parallel programming model, there is no agreement on the design approach to be taken, with different authors specify different steps in the design of a parallel program (often with conflicting views). However Foster (2000) puts forward a comprehensive approach covering reiterative stages of partitioning, communication, agglomeration and mapping. This is used for the design of the interactive parallel volume rendering application described in this chapter. The program design is complicated because the goal of the program is to provide higher performance and therefore efficiency is a factor throughout the development.

## 5.2 Application Requirements

The desired application is to have a graphical front end to display a 2D visualisation of 3D data (see figure 6). The application is to support the rendering of volumes and provide visual feedback on the progress.




Volume rendering is a technique for visualising 3 dimensional data by computing 2 dimensional projections of the data as a coloured transparent volume. Due to the large sizes of the volumes, the rendering and processing of data can take substantial amounts of time and therefore a parallel implementation is desirable. However the large size of the data provides difficulties in creating efficient implementations. For a better introduction to volume rendering, and in particular ray casting the reader is directed to Pawasukas (2000).

## 5.3 Partitioning

The partitioning step aims to identify opportunities for concurrent execution in the problem (2.8.1). As the proportion of sequential operations limits the maximum potential speedup of a parallel program (Amdahl's Law), the partitioning phase should aim to identify as much parallelism as possible (Foster, 2000). Later stages consider the practicalities of the implementation of the partitioning.

To simplify the application, functional partitioning is used to break the graphical interface from the volume rendering computation. This reduces the complexity, as the parallel code does need not be concerned with displaying data, but only in the computation of the output image. This follows good software-engineering practices of splitting the GUI from application code as it provides more maintainable software.
Although this appears to be a good approach it is unlikely to provide high display performance (Frames Per Second), as the sequential I/O is likely to become a bottleneck (Squyres, 1998; Li et al, 1997). However the volume rendering application is not expected to achieve performance in FPS (although 1 FPS is achieved), and therefore functional partitioning is viewed as a good decision.

The rendering computation code can be further broken down using data decomposition. The chosen method was to partition the output image data into individual pixels, each of which is computed by a separate ray.
Each ray requires a small subset of the whole volume to compute the output pixel, and can be processed concurrently without communication with other rays. As the partition has been made on the output data, as soon as a task has been completed it can be displayed to the user.

Although alternative decomposition approaches can provide faster parallel rendering as they make better use of local memory, they do not provide as good image quality as ray casting (Pawasauskas, 1997). Furthermore they cannot provide instant visual feedback of the rendering process, as the output data is often only computed with the final task (e.g. splattering).

The chosen partitioning method creates tasks of unequal computational requirements, as rays that pass through empty space require less processing than rays that pass through data. This is undesirable, as static scheduling is likely to provide poor results because the computational requirements are unknown. Fortunately the partitioning results (in all practical situations) with a number of tasks an order of magnitude larger than the available processors and this is identified as useful for the dynamic scheduling of unbalanced tasks (Foster,2000; Ragsdale, 1991). Furthermore the approach is very scalable as an increase in the problem size results in an increase in both the task complexity and number of tasks, and therefore more processors can be used more efficiently (larger tasks) in the computation.

## 5.4 Communication

The communication step attempts to identify the communication (data or control) (2.8.2) that is required between tasks to produce the desired output data. This step assumes that all the tasks are being processed concurrently as the mapping of tasks to processors is considered in the mapping step (5.6).

Each ray requires a subset of the volume data to complete the task, but requires no communication with other rays to produce the output pixel. However the separate pixels produced by the individual rays still need to be combined to produce the complete output image.

A centralised collection point matches the requirement for user feedback, as results can be displayed as soon as they are received. However this approach is not suitable for the current number of tasks, as the sequential processing for receiving results would represent a bottleneck (Foster, 1999, Ragsdale, 1991).
However this view is based on having processors for each task (possibly 1000's). But for the number of computers available in the intended implementation environment (below 50) the centralised approach is unlikely to become a bottleneck (Ragsdale, 1991). However the efficiency depends on the relative costs of receiving the result and executing the problem (Foster, 2000; Steenkiste,1996).
The time spent receiving the results would be minimal, with the biggest expense being computation required for the communication (buffering messages). The current task size is relatively small and therefore the performance is unlikely to be very high. However by increasing the task size (see step 5.5) this balance can be changed and the efficiency increased. Furthermore as agglomerating tasks increases the workload imbalance, the results are unlikely to be sent at the same time and therefore the centralised collection point is less likely to become a bottleneck

A popular alternative approach is to distribute the summation of the results between the tasks, having each task (except first and last) wait for the image from the task before it, and then pass the image with its pixel added to the next task in the line. The first task just sends its output pixel to the second, and the last task notifies the graphical user interface that the image is ready. However as the task sizes are not equal this process could be slower than the master slave approach, as if the first task requires longer to complete than others, the communication between all tasks would be delayed. Furthermore as the pixels are joined to make larger images the distributed accumulation approach requires more data to be transferred compared to the central point approach. Finally the approach gives no feedback to the user on the progress of the computation which is a key requirement of the application.

## 5.5 Agglomeration

The agglomeration step looks at whether it would be worthwhile to increase the task size, as fine-grained computation is not optimal for some types of computers (although by considering the task size we also consider the mapping!).

It has been identified that fine-grained tasks are not suitable for the implementation environment (2.8.4 & 5.4). By agglomerating multiple rays together to form larger tasks the frequency of communication is reduced but the communication size is also enlarged. However the current output from each task is only a single pixel and therefore an increase would actually save network bandwidth (2.8.7).
Caution is needed as the agglomeration of tasks reduces the scalability of the solution, as there are fewer tasks to distribute to processors. Furthermore increasing the computational grain size also increases the likelihood of a larger load imbalance between the processors.

Simple scan line rendering (horizontal line of the output image) was chosen as the computation required for each line is believed to be sufficiently large enough to reduce the communication to a acceptable frequency. But also the time required to complete a single scan line is small enough not to represent a problem in load balancing (i.e. all processors waiting for the last scanline).

## 5.6 Mapping

The final mapping step considers how tasks are to be placed on processors, and aims to keep processors busy (using computation for the problem), while minimising inter-processor communications.

Static scheduling is where decisions are made on the distribution of data only once at the creation of the tasks. This provides the most efficient solution if the task requirements and machine power are known as communication is minimised. However for volume rendering the task computation can vary and this approach would results in unbalanced processors.

For dynamic task allocation to be effective a sufficient number of tasks per processor need to be available. Fortunately even with the agglomeration of tasks into scan lines, the problem still has a sufficiently large enough number of tasks per processors for the intended implementation environment. For example for a small volume (320^3) on 50 machines there are still over 6 times as many tasks as processor. This fits in the 4 to 10 times recommended by Ragsdale (1991).

The master-slave approach was chosen to allocate tasks as it provides a simple implementation of dynamic scheduling that has been shown to work successfully (Allan, 1996; Cooperman, 1995; Smith and Shrivastava, 1995; Steenkiste, 1996; Squyres, 1998). Furthermore it matches the centralised collection approach that has already been chosen. However this has made a single task a weak link in the application, and therefore the implementation of the application should be concerned with the amount of computation and communication that is made on the master node.

## 5.7 Summary

This chapter has followed the design of a parallel volume rendering application. Figure 7 shows the high level design that has been produced based on the message passing program model.

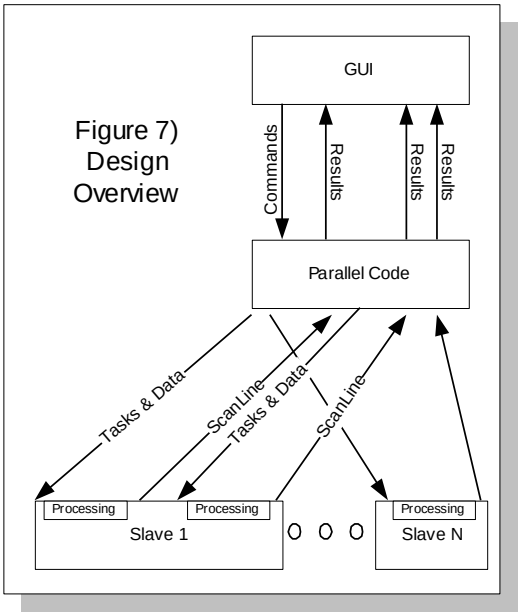


Figure 7) Design Overview

The design has followed one possible option of many, and has highlighted some of the factors that need to be considered when designing a parallel program. However the process is very much reiterative, and the implementation is just as critical an area as the design.

The following chapter will look at how the application was implemented using the LAM-MPI communication library introduced in chapter 4.3.

# 6 Application Implementation

## 6.1 Overview

This chapter is concerned with the implementation of the volume rendering application, and discusses any major changes to the design.

## 6.2 User Interface

The application design does not specify how the user interface is to be created. As discussed earlier, ideally the application would support many environments. However the creation of graphical windowed applications varies considerably between different operating systems.
The OpenGL Utility Toolkit from Silicon Graphics (Kilgard, 2000) was used as it provides a high level encapsulation around OpenGL graphics, and provides programs with a single interface for the creation of windowed applications (programs can be compiled without change for Linux/Unix/Windows etc).
However the library adds a further complication, as with its use a non-returning function (glutMainLoop) must be called, which starts an event processing loop that executes user registered callbacks when required. However to implement an interactive volume rendering application we need to receive results from the slaves and display the graphics at the same time.
This can be achieved by creating multiple threads in the master processes, but at present the usage of threads is dependent on the operating system. The pthreads approach from POSIX (Portable Operating System Interface for UNIX) is common throughout most UNIX/Linux operating systems and work is being made into developing wrappers for other popular operating systems like Microsoft Windows (Johnson, 2000).

## 6.3 Data Distribution

The workstations may not have sufficient resources to open a whole volume (either local disk space or memory). However the distribution of data by the master node either requires the whole volume to be in memory, or that frequent file access is performed (to the larger file server). As already noted the master task should perform the minimum amount of computation and communication (maximum amount distributed) to stop it becoming a bottleneck.
It was observed that the implementation environment has a common network file system between all workstations. In fact all user file access is through the Network File System (NFS). For each piece of data sent to a slave, the data has to first be sent to the master from the NFS, therefore doubling the communication (Figure 8). Where volume segments do not follow the contiguous nature of the file storage, many calls to produce the volume subsection would be required.
The chosen strategy is to allow each slave to read from the NFS (Figure 9). Although the number of messages is not reduced, the amount of data sent across the network is reduced. Also the master is freed of the communication and computation required to create the ray subsection, and therefore less of a bottleneck.

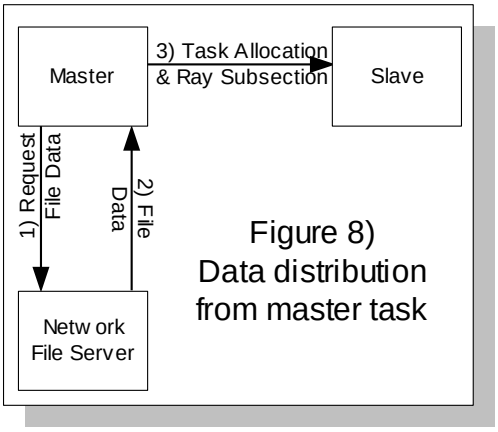


Figure 8) Data distribution from master task

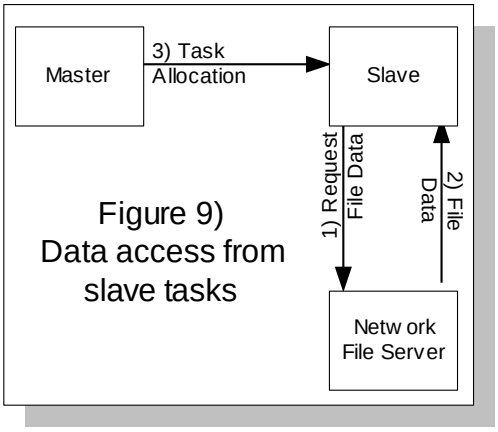


Figure 9) Data access from slave tasks

## 6.4 Volume Caching

3 caching methods were used in the slaves to store the volume data required for rendering.

a) The first approach was to access the volume as a file, and therefore required no additional memory to be allocated to store the volume. However this results in frequent communication with the NFS and therefore poor performance.

b) The second approach loads a segment of volume from the file server each time a task (set of rays) is allocated. This reduces the communication with the file server, as only one request is required per task.

c) The final approach caches the whole volume during initialisation, and therefore no fileserver access is required during rendering. This approach will not usually be possible where large volumes are concerned. It was primarily created to provide a comparison of the optimal approach against the two caching techniques above.

## 6.5 Task Scheduling

A basic First Finish First Serve (FFFS) approach was taken to task scheduling, as it has been used successfully in several other master slave implementations (Allan, 1996; Squyres, 1999).

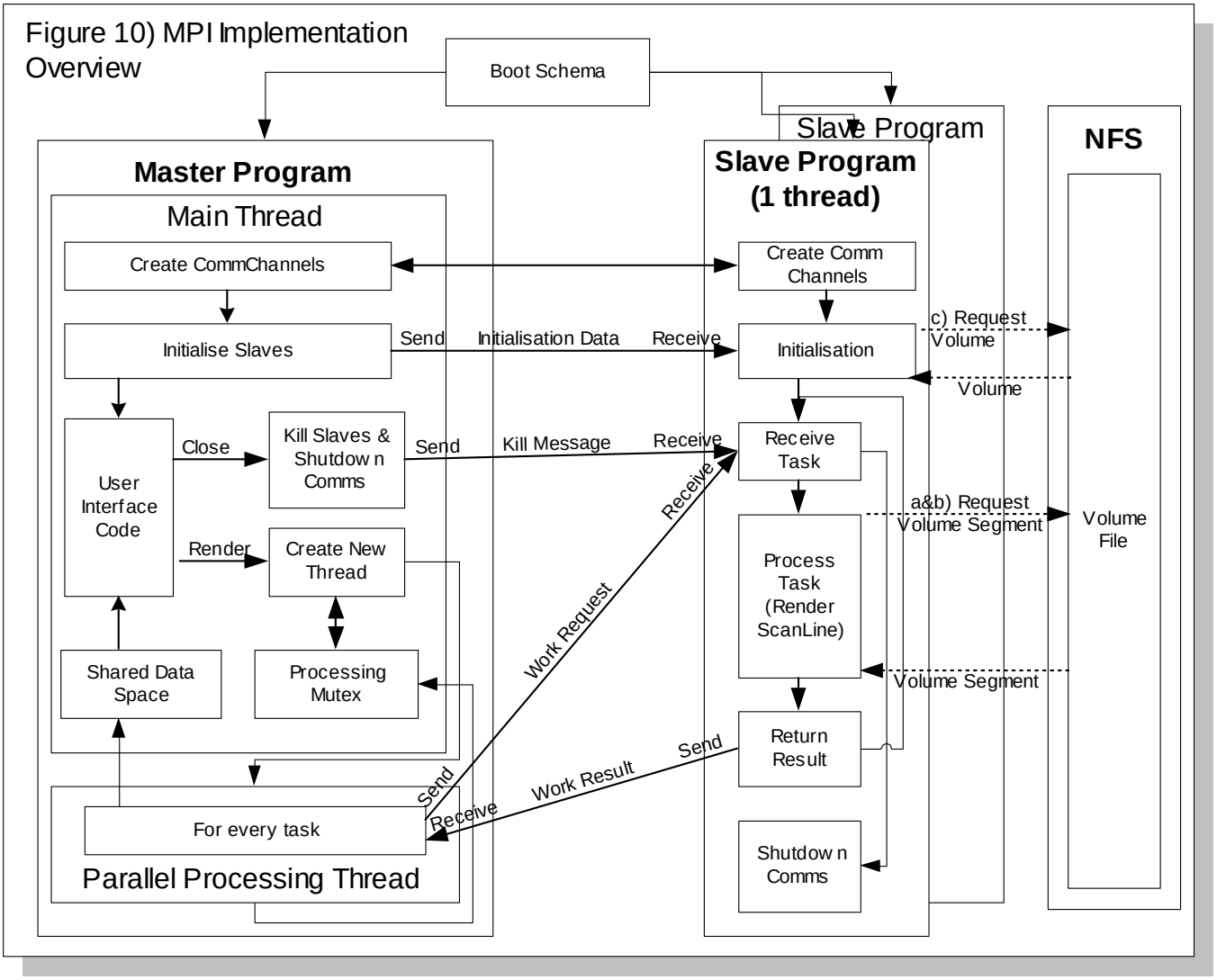
When the rendering is first started, each slave is allocated a task (if there is sufficient tasks). Then the master makes a blocking receive, and waits for a slave to return a result. When a slave returns a result, it is processed, and if there are further unallocated tasks another is allocated to the free slave. If not all the required results have been received the master repeats the cycle and waits for another result.

The work request is a fixed size structure that specifies a collection of rays that are to be rendered through the volume. In this basic application we send a volume segment, a ray direction and processing status flags (e.g. an terminate flag to slaves)

The result message is the image scanline number and the image data.

## 6.6 Execution Model

Figure 10) shows the execution model for the application. Appendix B provides a description of the steps in execution.




## 6.7 Summary

This chapter has described the actual implementation of the interactive parallel volume rendering application. Although the application has been created successfully (performance is analysed in chapter 9.0) the implementation was a rather long process, and it would be desirable to reduce the time required for similar applications.

# 7 Problems Identified

## 7.1 Overview

This chapter identifies problem areas in the implementation of the interactive parallel volume rendering application.

## 7.2 Complex MPI applications require significant development

The development of simple 'Hello World' type applications using the MPI can be quick and simple. However the production of more realistic applications is much more complex and deeper knowledge of the MPI is required. However it is felt that 'learning MPI is a lot like learning a complete new programming language' (Dietz, 1998). A particular area that was frustrating was the use of complex data structures in communication. These are complicated to use and a significant amount of time was initially spent trying to communicate successfully with structures. Another problem with parallel processing using the MPI is that errors are very hard to identify. Functions take numerous (often complicated) parameters and each parameter can be a possible source of a problem. Also communication operations are based on all parties joining in the communication. For point to point communication a possible error could be on one of the two communicating (or not communicating) tasks, while with collective communication any of the numerous tasks could be the source of an error.

## 7.3 Matching Send and Receive

A problem identified in the development of the application, but also identified at the Cornell Theory Center (2000) is in the synchronisation of communication operations. MPI is based on message passing where all parties who are participating in the communication explicitly join in. For communication to be made successfully, the different parties must agree on how and what communication is to take place. Failure to correctly match communication can result in deadlock or buffer overruns and therefore failure of the application. As the different parties cannot communicate at runtime to match communication, the programmer must do this before hand. However although program design is ideally completed before implementation, in many situations the design is modified as further knowledge is gained in the implementation. This causes problems in the development of MPI programs, as when further communication requirements are identified, previously completed code often has to be updated for a new approach.

## 7.4 Generic Scheduling and Fault Tolerance Code

The volume rendering application provided only basic scheduling code and therefore no fault tolerance. However the scheduling code and fault tolerance code is similar between most applications, and therefore could be provide in generic classes.

## 7.5 Lack of structure

One of the goals of MPI was not to restrict the programming to a specific structure (master-slave/SPMD etc) but to only define the communication interface. However this makes initial program development difficult as there is too much flexibility. In the initial volume rendering prototype the application and communication code was mixed together. This followed the learning process that was being used to produce the prototype, but does not provide easily understandable module code.

## 7.6 Summary

This chapter has introduced problems that were encountered in the implementation of the parallel volume rendering application. It was thought that MPI requires a significant amount of effort to develop with and its requirement for matching send and receives caused problems. Furthermore its flexibility also provides too little structure for application development. Generic scheduling code has been used in many other applications, but it still had to be implemented for the application.

These issues are tackled in the following chapter

# 8 Framework Design

## 8.1 Overview

This chapter explains what the framework has tried to achieve to reduce the complexity in the implementation of interactive data parallel applications. An overview of the actual framework is included in Appendix C.

## 8.2 Encapsulation of MPI communication operations

As identified MPI communications can be complex to use. The framework has tried to improve this by encapsulating all the MPI communication operations into the generic classes. This provides a further benefit of giving a more structured approach to data parallel programming. As the communication code is separate from the application code, the communication enhancements can be made without concern over effecting the application code.

However the creation of MPI datatypes has not been encapsulated, and custom complex data structures still require MPI datatype creation. The creation of the datatypes is contained in a single file, which is shared between the master and slave, and therefore ensures matching data structures. Projects have been made on encapsulating MPI datatype creation, but these have not been incorporated into the framework. AutoMap (Goujon, 1998) is a compile time tool that can be used to generate MPI datatypes routines from the complex data structures, and AutoLink (Goujon, 1998) is a runtime tool for creating MPI datatypes from user structures. These would ideally be further investigated (performance results were not available), and its inclusion with the framework could provide the datatype encapsulation.

## 8.3 Synchronisation of Send and Receive

One of the key benefits with the framework is that the application code does not have to worry about matching send and receives. Through the use of the default header, the master process can allocate any kind of task to a slave, and a slave can return any kind of result. This greatly simplifies application development, and makes the enhancement to applications (say loading another volume) much easier as the slaves state does not need to be considered. However the header-data approach used could represent a performance problem for finer grained applications (although not visible in the performance results).

The user still has to adequately describe any custom messages they transmit and also correctly interpret this information on the receiving processor. This would ideally be encapsulated from the user but this was beyond the scope of this project.

## 8.4 Encapsulation of Scheduling Code

The scheduling code is encapsulated in the generic master class, and therefore custom applications only need to submit tasks using ProcessTasks or AllocateTask. The basic FFFS scheduling approach was used, but also a Redundant FFFS (Squyres, 1999). RFFFS only changes from FFFS when all tasks have been allocated to processors.

## 8.5 Unresolved Problems

The framework is primarily designed for data parallel applications that require no communication between slaves. However communication between tasks is possible but requires the use of MPI communication. But given that the slaves do not know the location of other tasks (or even if they have been allocated), the communication between slaves is unlikely to be beneficial.

## 8.6 Summary

This chapter explains the way the framework has attempted to solve the various problems identified in the development of MPI applications (see chapter 7).

# 9 Performance Analysis

## 9.1 Overview

This chapter compares the performance of the parallel volume rendering application with a sequential implementation and a further parallel implementation using the proposed framework.

## 9.2 Testing Method

Performance analysis was conducted using several version of the interactive parallel volume rendering application:

1) Sequential application written without the MPI library
2) Custom application as described in chapters 5&6
3) Modified version of the custom application to use the framework class (the tests were made using FFFS scheduling to match the custom application).

For each one of these the different volume caching methods as described in 6.4 were used to obtain separate results.

All tests were run 5 times using only machines that were not in use, and the results are the average of the best 3 times. Only the best 3 times were used so abnormal results that may occur through the FFFS scheduling will hopefully not be seen in the results.

The sequential application was run on just one machine. The scaled performance is the time taken for the sequential application to complete, divided by the number of processors the scaled performance is referring to. This is viewed as the optimal performance that can be achieved by the parallel program (linear speedup).

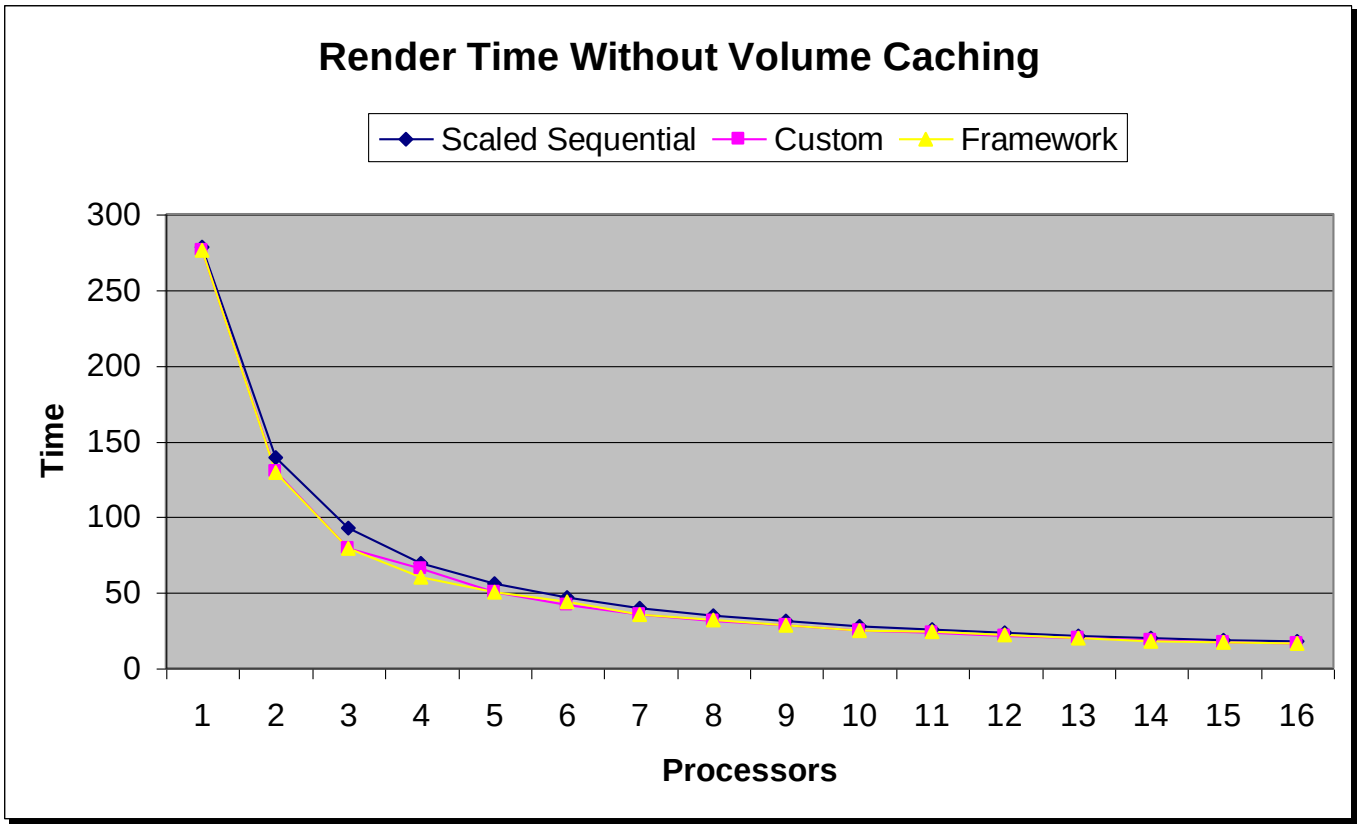
Speedup refers to the decrease in time compared to the original sequential application and is defined as:

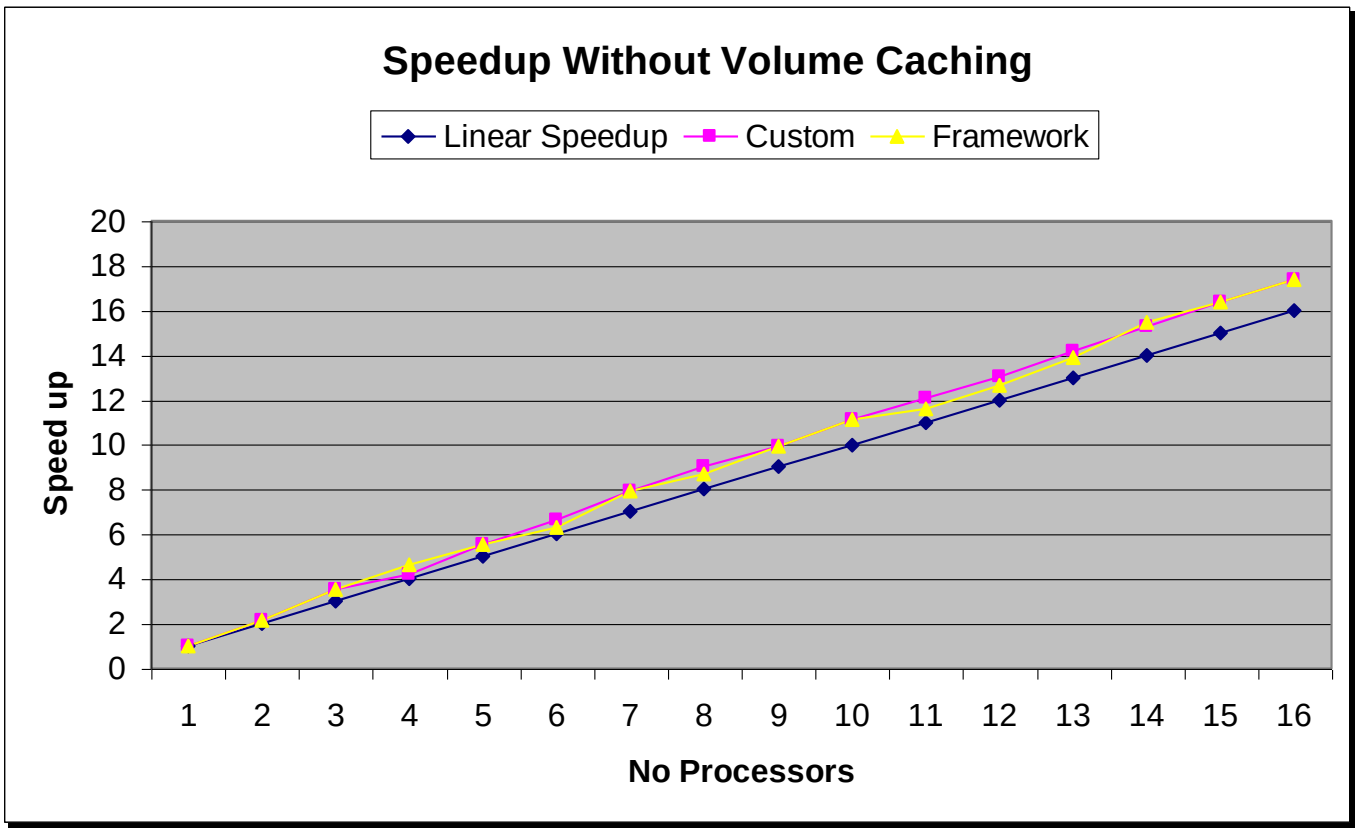
$$Sp(n) = Ts/Tp(n)$$

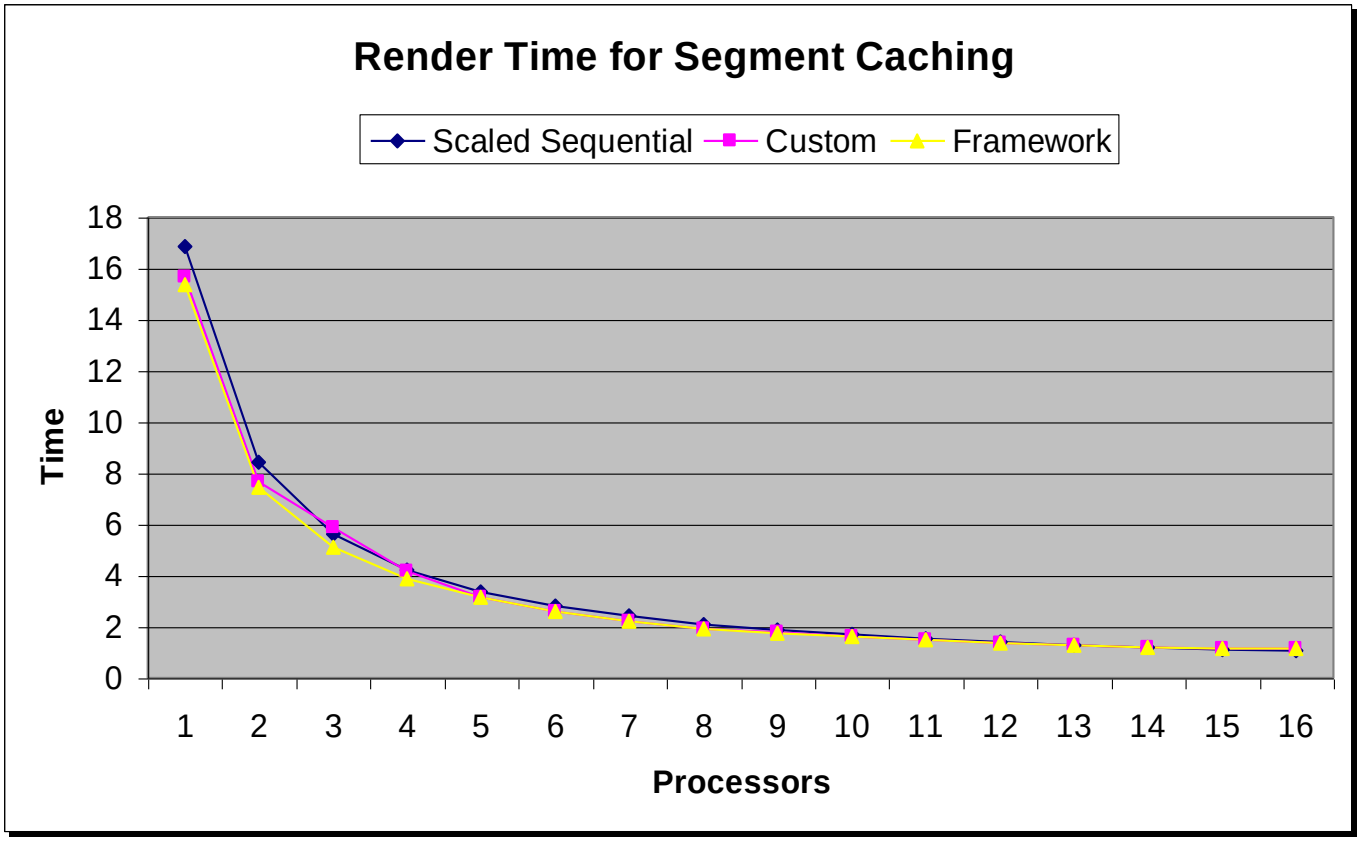
Where Sp is the parallel speedup on n processors, Ts is the time taken to process the problem sequentially without the parallel processing overhead, and Tp is the time taken to run the program in parallel on n processors.

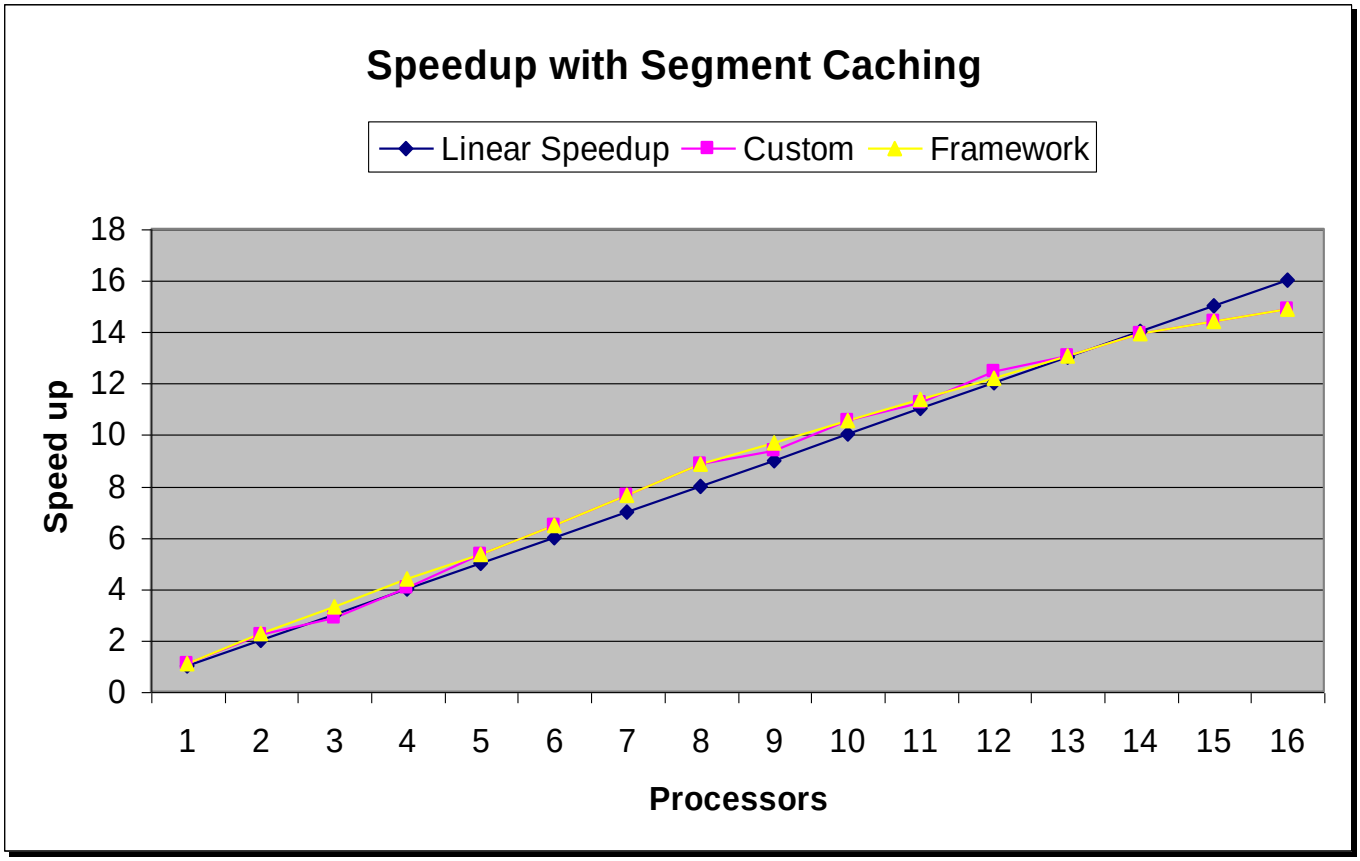
The performances of the parallel implementations were tested on the same environment using 1 to 16 UNIX workstations. The performance results are plotted on graphs on the following pages.

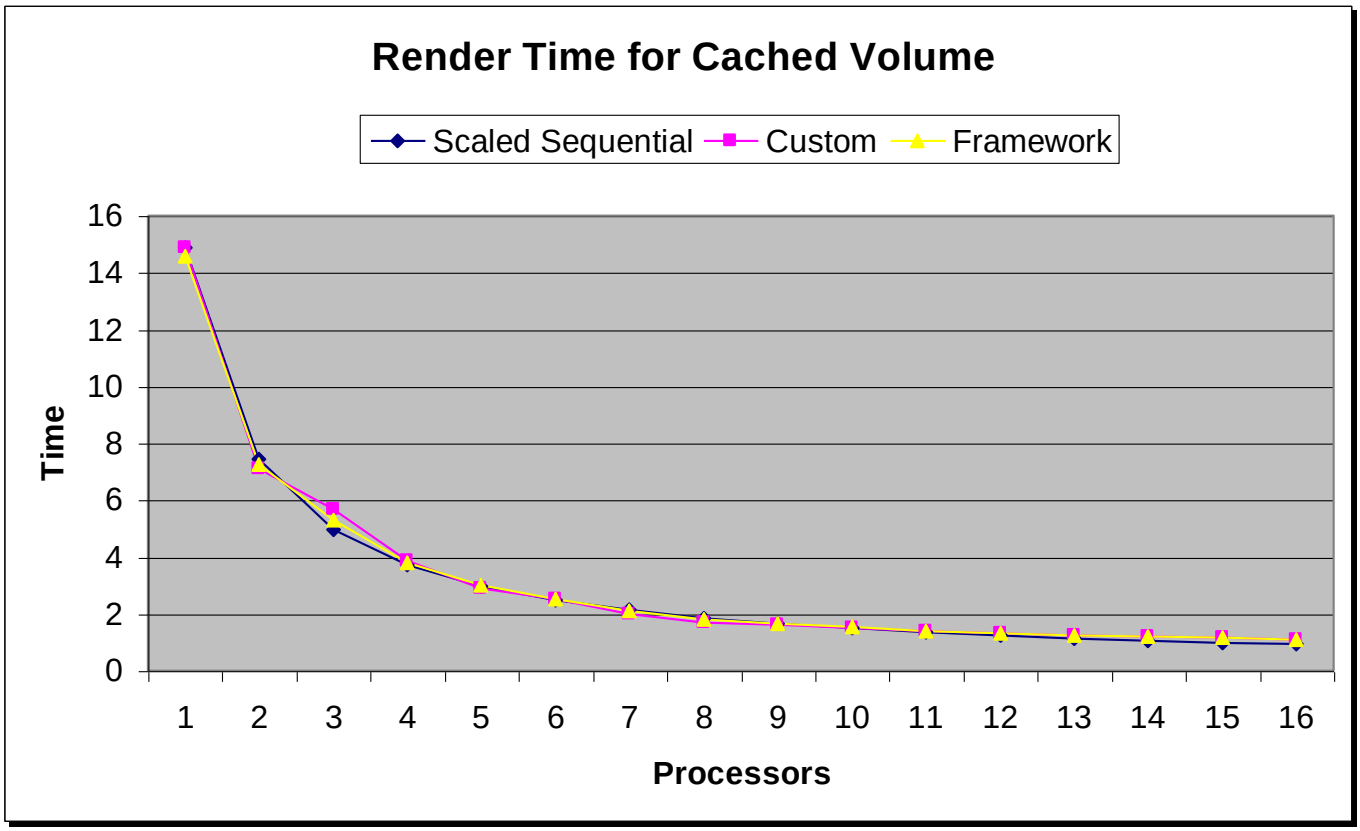
## 9.3 Performance Test Results

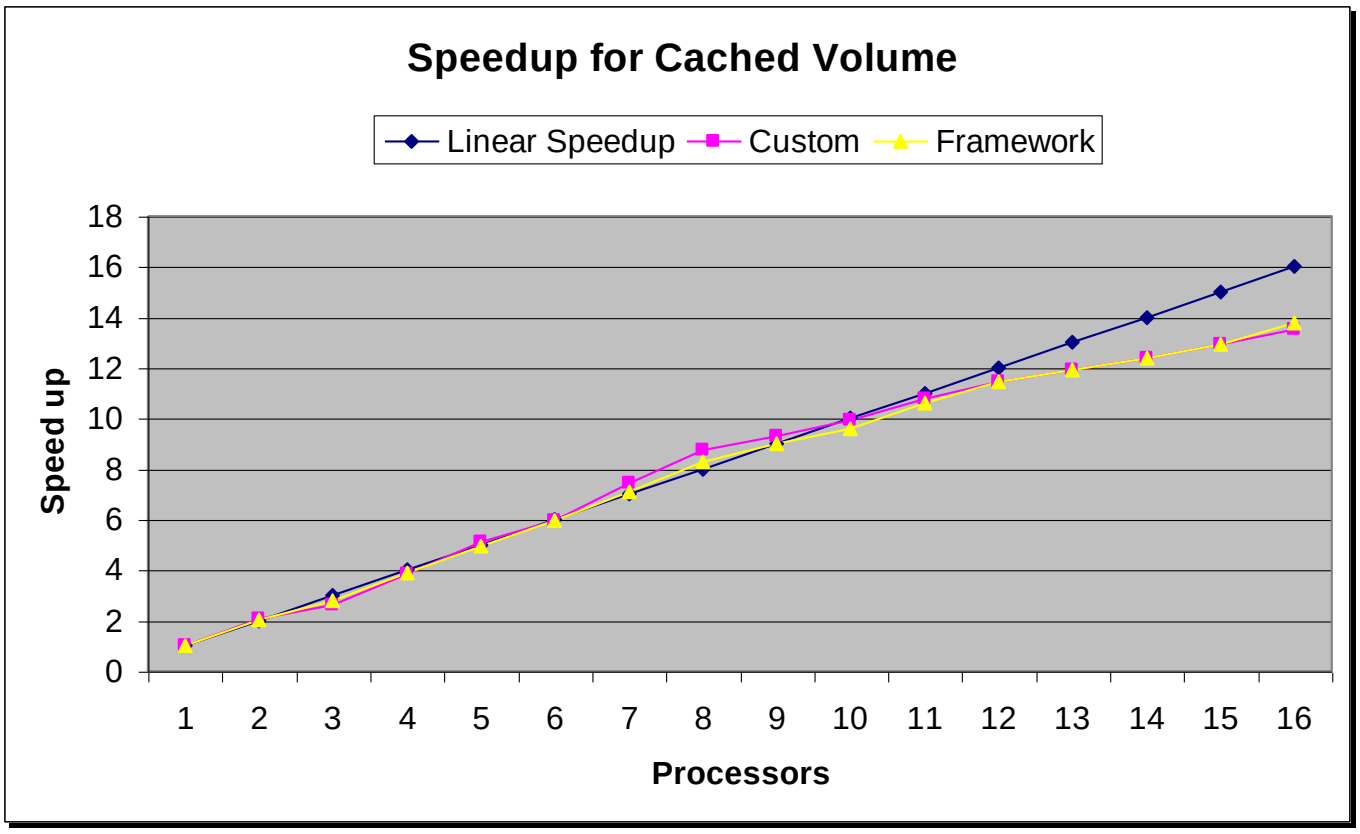



Render Time for Segment Caching
Scaled Sequential
Custom
Framework
Time
0
2
4
6
8
10
12
14
16
18
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Processors


Speedup with Segment Caching
Linear Speedup
Custom
Framework
Speed up
0
2
4
6
8
10
12
14
16
18
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Processors


Render Time for Cached Volume
Scaled Sequential
Custom
Framework
Time
0
2
4
6
8
10
12
14
16
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Processors


Speedup for Cached Volume
Linear Speedup
Custom
Framework
Speed up
0
2
4
6
8
10
12
14
16
18
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Processors

## 9.4 Performance Analysis

**Superlinear Speedup**
As can be seen from the graphs, both the custom and framework implementations provide 'superlinear' speedup. This at first seemed incorrect as how could an application provide higher performance than an optimally distributed sequential application?

However the application is functionally partitioned with a single interface on the host machine. Only the displaying machine needs to have a desktop running (like KDE), and to provide a windowed application for the display of results. The display task does not complete after a certain amount of processing but acts as a constant overhead. Machines that are used purely for data processing do not need to have any display processes running and therefore more processor power is available.

**Framework Overhead**
It was presumed that the 2 messages used in the header-data communication and the polymorphic functions in the framework would add a noticeable overhead to the processing. Surprisingly, it can be seen from the graphs that no significant difference could be identified between the framework and the custom application. In many of the results the framework shows slightly higher performance than the custom application!

**Inaccurate timing facilities**
Problems were encountered as the volume (20mb) was rendered very quickly, and no operating system functions existed to gain the elapsed time below a second interval. As the volume was being rendered in less than 1 second when 16 machines were being used, alternative time measurement techniques (stopwatch) had to be used. This however still does not provide the accurate results that would identify small differences in the performance between the custom and the framework application.

**Master-Slave Potential Bottleneck**
Using the 16 processors available the master-slave approach has not become a significant bottleneck with any of the applications. However the speedup obtained using the volume caching application is not as high as the speed up obtained with the application without volume caching.

As the volume caching approach completes a task in a much smaller time than the file caching, it is believed that the frequency of communication is becoming a problem in the master task.

The time required to complete a task (roughly) using the cached volume can be calculated as:

Task time = Sequential processing time for all tasks /number of tasks
46.46875 ms/task =14870ms / 320 tasks

This is actually below the minimum task size of 50ms that has been used successfully in the Distributed Processing Library created by Allan (1995).

Therefore the use of more than 16 machines with similar small cached volumes should be used with caution, as the master may become a bottleneck. However if the volume size is increased the task duration also increases and therefore the master is less likely to become a bottleneck. Further testing is required with larger volume sizes, which unfortunately were not available before the completion of this project.

**Redundant Scheduling**
In the testing a very occasional abnormal result would be obtained due to one machine not completing its task in the normal time (interference by some outside process). The redundant scheduling within the framework was used as a separate test, and provided identical performance to the FFFS scheduling. However with the redundant scheduling the occasional high results were avoided as tasks were allocated to other machines.

## 9.5 Summary

This chapter has compared the performance of a framework implementation of the interactive volume rendering application against the MPI approach and a sequential implementation. The performance results show little difference between the framework and the parallel implementation. Furthermore all parallel implementations provide very good performance increases, with all gaining ‘superlinear’ speedup.

# 10 Framework Conclusion

This chapter attempts to provide a conclusion on if the framework is a useful way of creating data parallel interactive applications.

## 10.1 Framework Benefits

**Provides high performance**
From the results analysis it can be seen that for some problems the framework can be used to gain very high performance. The benefits of running interactive applications on such an environment gain more benefits than standard batch parallel applications as the overhead of visual processing reduces the processing power available on the host machine. The preliminary performance results have showed no identifiable difference in the speed between the custom and framework parallel application, and therefore based on these results the framework offers great potential.

**Encapsulates MPI communication code**
The framework does provide a higher level interface from the MPI, with all MPI communication encapsulated within the framework. This reduces the complexity of program design, as the complexities of the MPI do not have to be understood. The learning time required for the framework is believed to be much lower than the learning time for MPI.

**Encapsulates scheduling**
The framework encapsulates the scheduling of tasks and provides basic fault tolerance. Even for the simple scheduling currently implemented in the framework, this significantly reduces the overhead of making a parallel application. With more complicated scheduling code for tasks where computational requirements are estimable the approach will offer even stronger benefits.

**Encapsulates matching Send and Receive**.
Through the use of the Header and Data approach in communication the framework provides a partial encapsulation around the need for matching send and receives. This provides much greater flexibility as programs can expand the communication with out regard for the slaves state.

**Structured Approach**
By providing a structured approach, development time is quicker as less confusion is made in the development, and more readable code is created.

## 10.2 Framework Limitations

**Does not provide MPI datatype encapsulation**
The framework currently does not provide encapsulation around the MPI datatypes and therefore still requires that the developer learn some of the MPI.

**Requires the learning of an additional model**
As the framework provides a structured approach to programming it requires that the user understand the structured model. Although the user's view of the framework is relatively simple it still requires learning.

**Master Slave approach a potential bottleneck**
Although the results have shown the master-slave approach to be successful for the current environment, it is believed that it could represent a potential bottleneck for smaller volume sizes and larger numbers of processors.

**Only applicable to problems that fit the master-slave model**
The choice of the master-slave approach makes the communication difficult between allocated tasks. The framework is realistically restricted to problems that can be partitioned into individual tasks that can be computed without communication with other tasks.

## 10.3 Conclusion

The conclusion is not clear. The framework definitely offers benefits but it still isn't perfect. No usability testing has been completed for the project, and this would have identified how people without internal knowledge of the framework felt about it use.

However the framework has proved itself on its ability to solve the problems identified in the MPI implementation and maintain high performance, and therefore meets the requirements of this project. Furthermore the framework has shown its use by reducing the development time for the implementation of an unrelated project (Appendix D).

# 11 Bibliography

Aggarwal, J. and Chillakanti, P., 1996. Software for parallel computing –a perspective. In: A.Y.Zomaya, ed. Parallel Computing Paradigms and Applications. International Thomson Computer Press, 357-375

Allan, D., The Distributed Processing Library (DPL). *Astronomical Data Analysis Software and Systems V*, Vol 101, 1996

Anderson, P. (2000). The Texas Tech Tornado Cluster: A Linux/MPI Cluster For Parallel Programming Education And Research [online].
Available from: http:/www.acm.org/crossroads/xrds6-1/tornado.html [01/05/2000]

Anderson, T., Culler, D., and Patterson, D., 1995. A case for networks of workstations: NOW. *IEEE Micro*, Feb

Baker, M. and Fox, G. (2000). Distributed Cluster Computing Environments [online]. Syracuse University. Available from :-
http://www.npac.syr.edu/users/mab/homepage/cluster-computing/dcce-12.html#RTFToC3 [01/05/2000]

Bloomer, J., 1992. Power Programming with RPC. O'Reilly & Associates.

Bruck, J., Dolev, D., Ho, C., Rosu, M., Strong, R., 1997. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing*, Volume 40(1), pages 19-34

CESDIS. (2000). Introduction to the CESDIS Beowulf Project [online].
Available from: www.beowulf.org/intro.html [01/05/2000]

Chiola, G., 2000, GAMMA Project: Genoa Active Message Machine, Dipartimento di Informatica e Scienze dell'Informazione [online], Available from:-
www.disi.unige.it/project/gamma/[01/05/2000]

Clematis, A. and Corana, A., 1999. Modelling performance of heterogeneous parallel computing systems. *Parallel Computing*, Volume 25, p1131-1145.

Cooperman, G., 1995. STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems, *Proc. of International Symposium on Symbolic and Algebraic Computation*, ACM Press, pp. 126-132

Cornell Theory Center, 2000. Course on Parallel Computing and Programming Languages[online], Available from :-
www.tc.cornell.edu/Edu/Talks/course2.html, [01/05/2000]

Corrie, B., Mackerras, P., 1993. Parallel Volume Rendering and Data Coherence. In: *Proceedings of the 1993 symposium on Parallel rendering.* San Jose CA USA , October 25 – 26 1993.

Das, D., Das, P., PACWON: 1998. A parallelizing compiler for workstations on a network. *Journal of Systems Architecture*, Volume 45, p157-176

Dietz, H., 1998, Linux Parallel Processing HOWTO [online], http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html [01/05/2000]

Donaldson, V., Berman, F., Patori, R., 1994. Program speedup in a heterogeneous computing network, *Journal of Parallel and Distributed Computing*, Vol 21(3), p316-322..

Du, X., Zhang, X., 1997. Coordinating Parallel Processes on Networks of Workstations, *Journal of Parallel and Distributed Computing*, Vol 46,p125-135.

Eadline, D., 1999. Cluster Quick Start (DRAFT)[online], Paralogic, www.xtrememachines.com/x-cluster-qs.html [01/05/2000]

Foster, I., 2000. Designing and Building Parallel Programs[online], www-unix.mcs.anl.gov/dbp/ [01/05/2000]

Flynn, M.J., 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. On Computers*, vol C-2, pp. 948-960,

Giertsen, C., Petersen, J., 1993. Parallel Volume Rendering on a Network of Workstations, *IEEE Computer Graphics & Applications*

Goujon, D., Michel, M., Peeters, J., Devaney, J., 1998. AutoMap and AutoLink: Tools for Communicating Complex and Dynamic Data-Structures Using MPI. Reseau ESIAL, p. 2, vol 4,

Hoffman, F. and Hargrove, W., 2000. Parallel Computing With Linux[online] Available from : http://www.acm.org/crossroads/xrds6-1/parallel.html [01/05/2000]

Hsu, W., 1993. Segmented Ray Casting for Data Parallel Volume Rendering *Proceedings of the 1993 symposium on Parallel rendering*, San Jose, CA USA , October 25 – 26 1993, Pages 7 - 14

Humprey, W., Coghlan, S., 2000, Using a Linux Cluster for Linear Accelerator Modeling [online]. Advanced Computing Laboratory, Los Alamos National Laboratory.
Available: www.extremelinux.org/activities/usenix99/docs/linacc/LinuxAcc2.html [01/05/2000]

Johnson, R., Open Source POSIX Threads for Win32.
Available: http://sourceware.cygnus.com/pthreads-win32/ [01/05/2000]

Kilgard, M., 2000. The OpenGL Utility Toolkit (GLUT) Programming Interface [online].
Available: http://trant.sgi.com/opengl/toolkits/toolkits.html [01/05/2000]

Kokinis, J., Lent, E., Gokhale, N., Bradshaw, S., 1994. A distributed, parallel, interactive volume rendering package, *Proceedings of the Visualization 94 conference*, Washington DC, October 17-18, pp 21-30

Kung, H., Sansom, R., Schlick, S., Steenkiste, P., Arnould, M., Bitz, F., Christianson, F., Cooper, C., Menzileioglu, O., Ombres, D., Zill, B., 1991. Network-Based Multicomputers: An Emerging Parallel Architecture, *IEEE Supercomputing*, p663.

Leutenegger, S., Sun, X., 1997. Limitations of Cycle Stealing for Parallel Processing on a Network of Homogeneous Workstations. *Journal of Parallel and Distributed Computing*, 43, 169-178,

Li, W., Huang, X., Zheng, N., 1997. Parallel implementing OpenGL on PVM, *Parallel Computing*, 23, p1839-1850.

Liu, Y., Cheng, H., King, C., 1999. High performance computing on networks of workstations through the exploitation of functional parallelism, *Journal of Systems Architecture*, Volume 45, p1307-1321

Ma, K., Painter, J., Hansen, C., Krogh, M., 1999. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering, *Proceedings of the 1993 symposium on Parallel rendering*, San Jose, CA USA, October 25 - 26, Pages 15 - 22

Machiraju, R., Yagel, R., 1993. Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors, *Proc. of SUPERCOMPUTING'93*, Portland, Oregon, November pp. 699-708

Mackerras, P., Corrie, B., 1994. Exploiting Data Coherence to Improve Parallel Volume Rendering, *IEEE Parallel & Distributed Technology*

Message Passing Interface Forum. 1994. MPI: A Message-Passing Interface Standard

Meyer, T., Davis, J., Davidson, J., 1997. Analysis of Load Average and its Relationship to Program Run Time on Networks of Workstations, *Journal of Parallel and Distributed Computing*, Volume 44, p141-146,

MPI: A message-passing interface standard, Dept of Computer Science, Univ of Tennessee. 1994

Snir,M., Otto,S., Huss-Lederman,S., Walker,D., Dongarra, J., 1996. MPI: The Complete Reference, Available from :- http://www.netlib.org/utk/papers/mpi-book/mpi-book.html [01/05/2000]

Munro, A., 2000. Moving to Distributed Processing Standards … Remote Procedural Call. University of Bristol, www.ja.net/documents/NetworkNews/Issue44/RPC.html [01/05/2000]

Oberhuber. M., 1998. **ICG - Institute for Computer Graphics and Vision. Technical University Graz, Austria**

**Available:** http://www.icg.tu-graz.ac.at/mober/pub/thesis/html/da.htm

Parsons, P., 2000. Object Oriented Parallel Programming – Summary [online]. Hull University. Available from: -http://www2.dcs.hull.ac.uk/people/pjp/Teaching/08318-9697/overviews/oo.html [01/05/2000]

Pawasauskas, J., 1997. Volume Visualisation With Ray Casting, *Advanced Topics in Computer Graphics*.
Available: www.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.html [01/05/2000]

pC++, 1994, Overview of pC++/Sage++, Available from :- www.exteme.inidiana.edu/sage/overview.html [01/05/2000]

Perrot, R.H., 1987, Parallel Programming, Addison-Wesley Publishers Ltd.

Pfister, H., Kaufman, A., 1996. Cube~4- A Scalable Architecture for Real-Time Volume Rendering, *Symposium on Volume Visualization Proceedings of the 1996 symposium on Volume Visualization*, San Francisco , October 28 - 29

POOMA, 2000, Los Alamas National Laboratory, Available from :- www.acl.lanl.gov/pooma/html/introduction.html [01/05/2000]

Ragsdale, Susann, 1991. Parallel Programming, McGraw-Hill (Intel Series)

Sergei, G., 1996. Stages and transformations in parallel programming. *Abstract Machine Models for Parallel & Distributed Computing*, Vol 48, p147-162

Sharp, J.A., 1987. An Introduction to Distributed and Parallel Processing, *Blackwell Scientific Publications*

Smith, J., Shrivastava, S., 1995. Fault-Tolerant Execution of Computationally and Storage Intensive Parallel Programs Over A Network Of Workstations: A Case Study, Department of Computing Science, The University of Newcastle upon Tyne, June, Available: http://arjuna.newcastle.ac.uk/group/papers/p057.html [01/05/2000]

Steenkiste, P., 1996. Network-Based Multicomputers: A Practical Supercomputer Architecture, *IEEE Transactions on Parallel and Distributed Systems*, Vol 7(8)

Sterling, T., Becker, D., Warren, M., Cwik, T., Salmon, J., Nitzberg, B., 1998. An Assessment of Beowulf-class Computing for NASA Requirements: Initial Findings from the First NASA Workshop on Beowulf-class Clustered Computing. In *Proceedings, IEEE Aerospace Conference*. March 21-28, Aspen CO,.

Squyres, J., Lumsdaine, A., Stevenson, R., 1998. A toolkit for parallel image processing. *Part of the SPIE Conference on Parallel and Distributed Methods for Image Processing II, SPIE Vol* 3452, p69-80

Valiant, L., 1990. Bridging Model for Parallel Computation, Communications of ACM, 33, 8

Warren, M., Becker, D., Goda, Salmon, J., Sterling, T., 1997. Parallel Supercomputing with Commodity Components, *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1372-1381,

# Appendix A – MPI Overview

MPI-1 contains 129 functions for both point to point and collective communication between MPI processes, while MPI-2 is an extension with 152 functions but currently no full implementations. Communication is based around 'communicators' that contain collections of processes, each of which can be identified by their unique rank (id) within that communicator. Three communicators are automatically created (one being for all the processes) and further groups can be derived from these by executing various functions on the default communicators.

To obtain the benefit of transparent heterogeneous communication the data needs to be correctly described otherwise the automatic conversion will not be made. MPI has the ability to send simple data types, array data types (even if not contiguous in local memory), and complex data structures.

Point to point messages are made between two processes in a communicator group using the processes ranks (id's within that group) and message tags. The message tags are integer values used by the applications to identify different messages, and allow processes to restrict receiving messages to those of a certain tag. There are 4 send message functions (more composites) with different buffering employed in each and all these are available in non-blocking and blocking modes. Blocking sends are completed only when the buffer is ready for reuse while non-blocking sends return instantly (when the buffer may still be in use) with a handle to allow checking of the buffer state.

Collective operations provide concurrent point to point communication between all the processes within a given communicator. Every process must execute the same routine for the operation to take place. There are various different routines but they mainly consist of different implementations or combinations of broadcast, gather, scatter and reduce.

# Appendix B – Application Implementation Overview

**Step 1 - Preparation 1**
Firstly and not shown in the diagram (Figure 10), the communication library must be running on each machine. How this is done is not specified in the MPI but is dependent on the implementation. With the LAM-MPI a lamboot command is executed with a list of processors that are to be included in the communication, which then become the total available processors.

**Step 2 - Preparation 2**
When the application is to run a simple boot schema is used to map programs to processing nodes (as created in step1). For this application a master is created on the host node (can be created remotely but as its interactive a bit pointless), and slaves are created on all the available nodes (optionally including the host).

**Step 3 - Initialisation 1**
When the master and slave programs are started, they both create and initialise their communications. For the prototype application the default single communicator for all processes is used, and therefore only a call to MPI_INIT is need to initialise communicators. Further parameters are set-up to gain the task's rank in the communicator, and the number of processes who have joined the communicator.

**Step 4 - Initialisation 2**
When the communications have been initialised, the master process loads the header information (size information) from the volume file and loads the colour map used for rendering into memory. The information is combined into a fixed size structure, and then sent as an initialisation message to each slave (who having been waiting since Initialisation 1).
Upon the slaves receiving the initialisation structure, they either store the information (volume caching a&b) or load the volume into memory from the file server (volume caching c).
When this has been completed the master displays the user interface (GlutMainLoop), and the slaves wait to receive a work request.

**Step 5 - Render**
When some user input indicates that the volume is to be rendered (e.g. key press), a mutex is checked to see if the parallel code is currently rendering. If not then a new thread is created to render the volume, leaving the current thread free to return control to the user interface. The newly created thread distributes the tasks as described in (6.4) updating the shared image data every time a new result is received from a slave. When all the results have been collected, the processing mutex is freed and the thread terminates. An image update mutex was originally created, but removed as only the parallel processing thread actually writes to the image data, and the GUI periodically renders the image.

Slaves upon receiving the work request, may need to obtain volume data to render the scanline (volume caching a&b). This made by using the stored file information passed during the second initialisation phase.

Following this the slave renders the volume into an array of pixels (currently RGBA values are stored for each pixel). The result is returned to the master, with the work request information (as the master needs to identify the work).

**Step 6 – Shutdown**

Under the current implementation the termination of the application is not allowed until the parallel processing has finished (checked with the processing mutex). If no parallel processing is being made, then each slave is sent a default work request structure with a message tag which identifies that it is to terminate. Following this both the master and slave free their communication structures and terminate.

# Appendix C – Proposed Framework

## Overview

This chapter aims to introduce the framework for interactive data parallel applications.

## Task Messages

The framework is based on using messages to describe tasks. The user must package instructions on processing to be completed into a message. This can then be transferred to a slave, which in turn executes the relevant processing as specified in the message.

## Communication

All user communication (initialisation does not follow this model) uses a fixed size default header message and optional variable size data message. Two headers are required in the shared communications file, one for master-to-slave messages and one for slave-to-master messages. These tell the receiving program if there is a following data message, how the message is being sent and the characteristics of the buffer (set by user). Figure 11 gives an example of custom message buffer transfer from the master program to a slave.

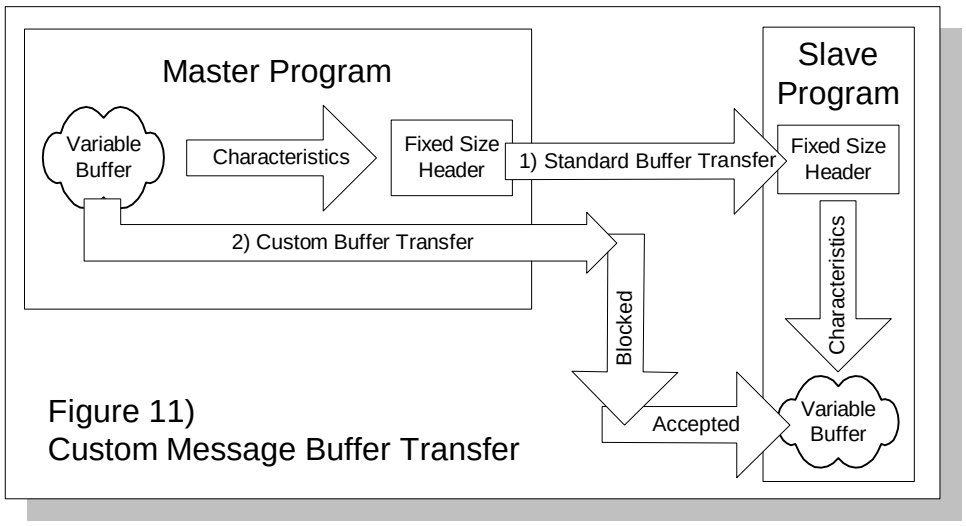


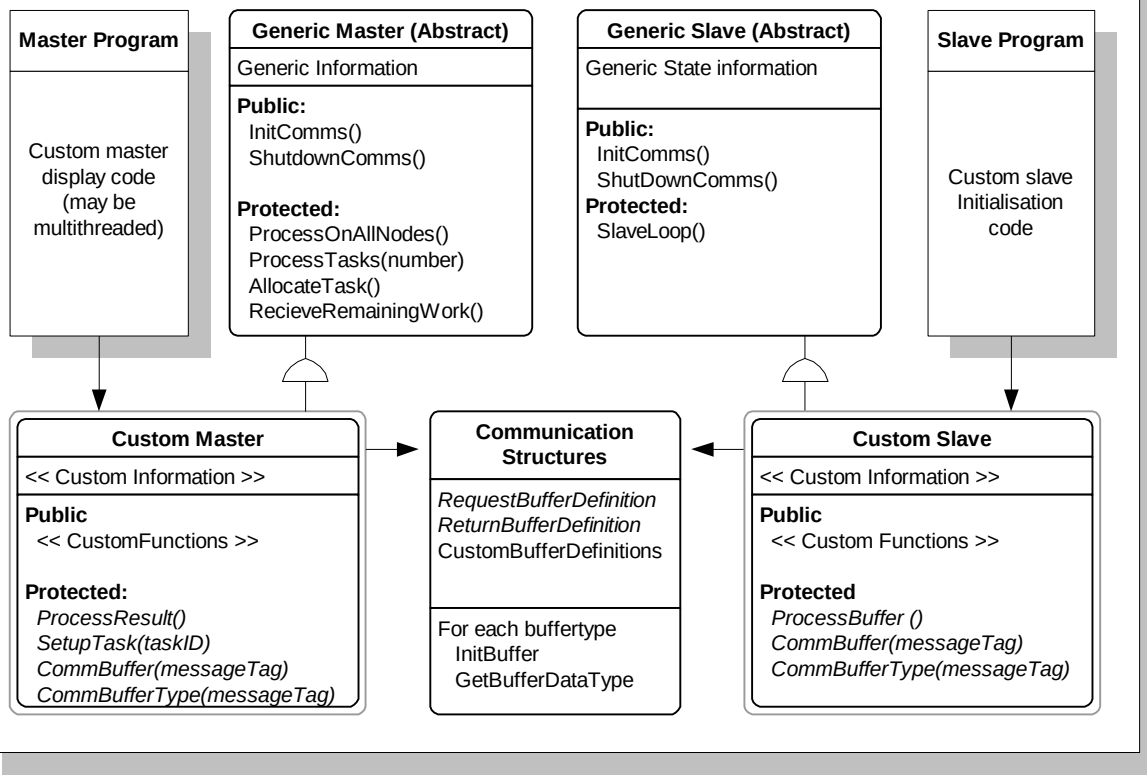

Figure 11)
Custom Message Buffer Transfer

For static data structures (or structures that can be set with a custom initialisation) the characteristics of a buffer may be adequately defined by a single message tag. The header intends to tell the receiving program what structure they need to create to receive the message correctly.

## File Overview

Figure 12 provides an overview of the classes and files that are used to make an application with the framework (private functions are not shown). The functions or data structures in the custom classes printed in italics are required for the instantiation of the class as they are specified in the generic abstract class.
For both the generic master and slave classes there are more protected functions than shown in the diagram, as the classes are intended to be extendable through the use of polymorphism. However for standard use, these functions are considered private and are therefore not shown in the diagram.

**Figure 12 - Framework Overview**

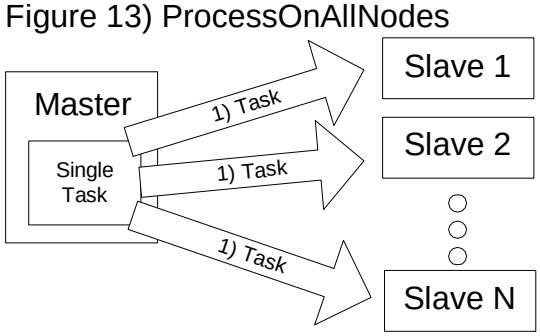



### Generic Master

The InitComms function initialises the communications and creates separate MPI communicators between each slave and the master.
The ShutdownComms sends a kill message to each slave, and then shutsdown the master communication.

ProcessOnAllNodes sends a single user specified task message to all the slaves (Figure 13). The slaves in turn process the message and each return a reply message. The reply messages are processed sequentially on the master node with the custom master function ProcessResult. The function gives no guarantee of the sequence

Figure 13) ProcessOnAllNodes



of processing or sequence of replies. After execution all available slaves will have processed the given task (how they have processed is dependent on the slave implementation).
This is intended for initialisation purposes, however it can also be used for multiple instruction stream single data stream (MISD) applications.
However the current implementation of this function does not provide fault tolerance, since if a slave were to die after accepting the task, the master would wait indefinitely for the result.

The ProcessTasks function (Figure 14) takes two arguments, the first specifying how many tasks are to be processed, and the second an optional argument specify scheduling options. The function allocates tasks to slaves (see task allocation), and only returns when all task results have been processed and the results received.
The order in which tasks are processed is not guaranteed, and neither is the order in which the results are received.

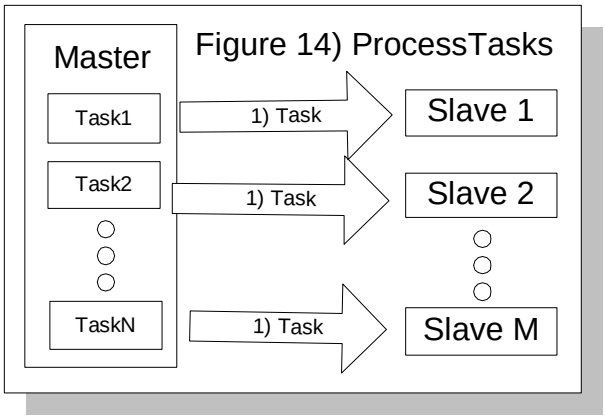


Figure 14) ProcessTasks

AllocateTasks and ReceiveRemainingTasks are lower level functions for task allocation that can be used if the number of tasks is unknown. Tasks are allocated using the AllocateTasks (possible blocking), and outstanding results (when all tasks have been allocated) are collected using ReceiveRemainingTasks

**Generic Slave**
The InitComms function performs similar initialisation to the master, but only keeps a communicator with the master.
The ShutDownComms close the communication and frees the communicators.
The SlaveLoop enters a loop that waits for task messages from the master. On getting a task message the custom class ProcessBuffer function is called to process the task. On completion of ProcessBuffer a result message is returned to the master, and the slave continues in the loop. Only by getting a kill message from the master node does the SlaveLoop exit.

**Custom Classes**
Both the custom master and slave classes must implemented GetBuffer functions to return custom buffers and their MPI datatypes. These are called when a header message is received that states that a custom buffer has also been sent (see communication 9.3).

**Custom Master**
The generic master calls the SetupTask function before a task (specified by the ID) is sent to a slave. The function is to set up the task request message to describe a task item (see communication 9.3). If the task requires a variable custom data buffer, the header must also include sufficient information so that the custom slave buffer functions can allocate a correctly sized buffer (see communication 9.3)

ProcessResult is called when a slave has completed the processing of a buffer, and a result has been returned.

Further custom functions are required in the inheriting classes, since the generic class does not provide any public task related functions (only initialisation and shutdown).

**Custom Slave**
The custom slave contains one additional function of ProcessBuffer that is called when a task request has been sent from the master. This routine is where application code is placed to process the task sent from the master. Before exiting the routine return data can be specified by altering the default return header to specify a custom buffer.

**Standard Master and Slave Programs**
These contain code that is not concerned with the parallel processing. The master can include multithreading if interactive input from the user is required. The slave program will often just create an instance of the slave class and just pass control to the slave loop.

**Communication Structures**
This consists of a header file that contains the definitions of all C structures to be used as communication buffers but also function definitions. The c file contains the actual implementations for all the structure initialisation and MPI data type creation functions.

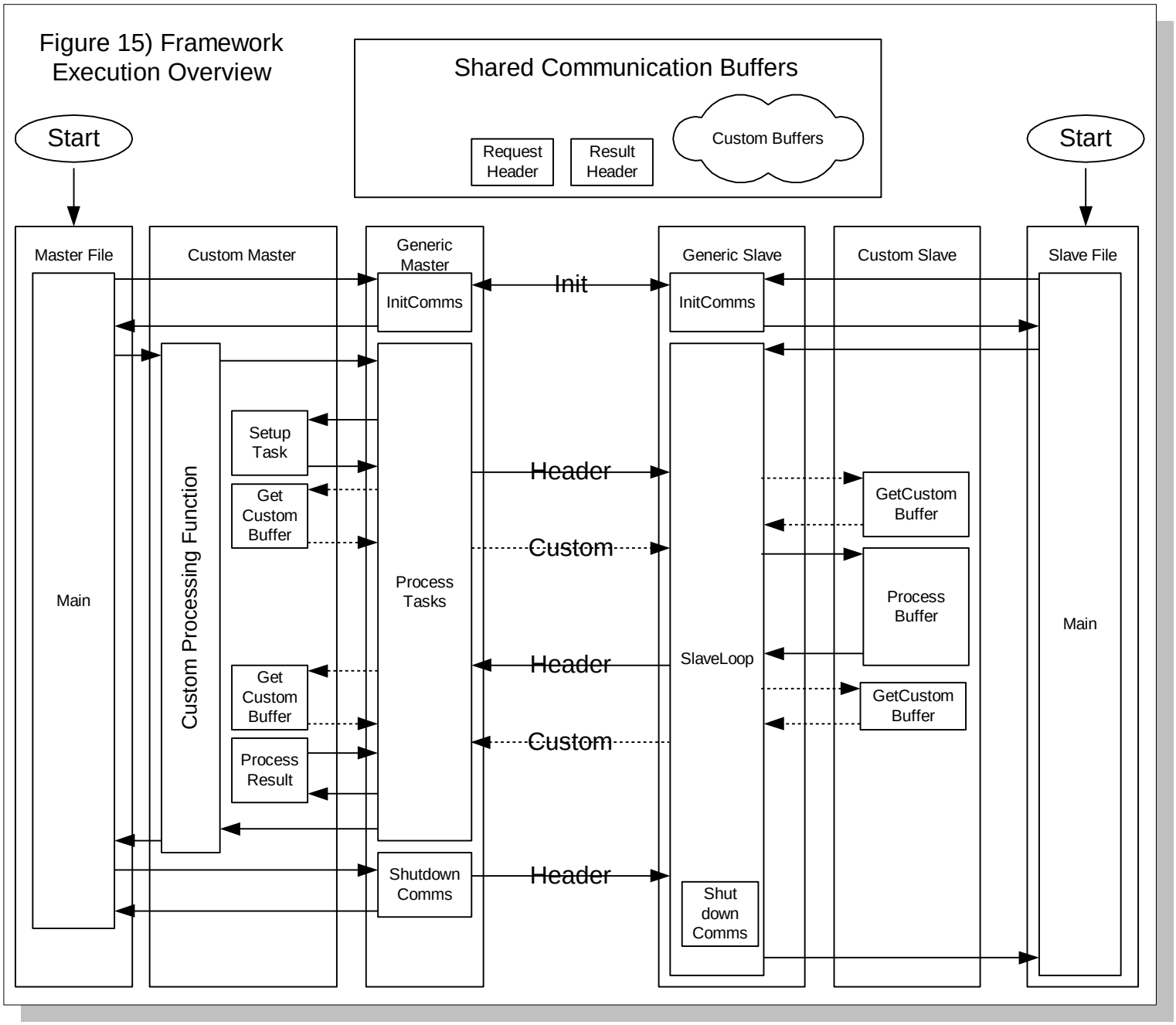
## Task Allocation

The task allocation revolves around a task list and a slave list. The task list is created when the custom code calls ProcessTasks or is incrementally created with AllocateTask. This contains a list of all the tasks that are to be processed and the state of each. Initially a task is given the state of ‘not started’. If it is then allocated to a processor the state is changed to ‘Processing’. If the slave returns a result the task status becomes ‘Completed’.

With FFFS free processors are only allocated to ‘not started’ tasks, while with RFFFS several tasks may execute a ‘Processing’ task.

## Execution Model

Figure 15 provides an execution model for the use of the ProcessTasks function. The diagram shows only the allocation of one task to one slave for simplicity, and multithreading in the master is not considered. Two threads are created one in the main procedure in the master program, and the another in the slave program. The blank arrows show the flow of control, and the named arrows show message passing. Dotted arrows are optional control or messages for custom message buffers. The execution of the program flows downwards, with the top being the start of the program, and the bottom being the termination.



## Procedure for Use

The last few pages has provided an overview of the actual framework, but has not showed the actual use of the framework. Using the template classes parallel processing can be obtained by adding a very small amount of code.

The development of programs can be made by following a few easy steps

1. Identify program segments that can run concurrently and independently
2. Identify the communication messages needed to describe the processing required in these tasks, and create the relevant custom buffers. Create a different MessageTag (ID) for each custom buffer.
3. Modify the message headers to provide adequate definitions of the custom buffers.

Custom Master and Slave:

4. Complete the get CustomBuffer and CustomBufferDataType routines to return a custom buffer pointer and datatype based on the description in the header.

Custom Slave:

5. Fill in process buffer routine to provide some processing based on the buffers received.

Custom Master only:

6. Complete the SetupTask routine to fill in the header and optional custom buffer.
7. Create a custom master function (like StartRenderer) to call the ProcessTasks in the Generic class with the relevant number of tasks.

SlaveFile:

8. Include an object of your custom slave, call InitComms, call the SlaveLoop, and then call ShutDownComms. (InitComms is automatically called with ProcessTask and ShutComms is automatically called on object termination)

Master File:

9. Include an object of your custom type and call the custom function written in step 7 (InitComms and ShutComms are again automatically called).

# Appendix D - Case Study Analysis

An artificial neural network (ANN) was developed using the framework to highlight the flexibility in its application.

A batch back propagation neural network as proposed by Evans and Sanossian (1996) was constructed under the framework. This is based on creating several identical artificial neural networks on several processes, and partitioning the training data set between them. Each network performs the feed-forward and back-propagation for their training vectors but the connection weights are not modified. Instead the calculated weight updates are sent back to a central point where the average weights for all slaves are calculated. All networks are then updated with the averaged weights and the processing loop continues. The frequency of this synchronisation can range from a minimum of each processor running just one training item or to the maximum of the distributed processing of the whole training set. Figure 1, shows the implementation of the batch ANN. Step a, is made with ProcessOnAllNodes, step b uses ProcessTasks with FFFS scheduling, and step c reverts back to ProcessOnAllNodes.

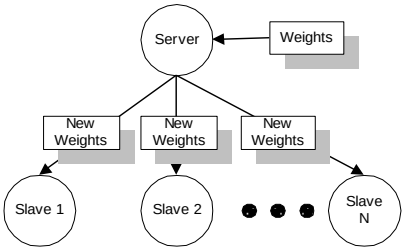



Figure 1 (a) Clients networks are updated

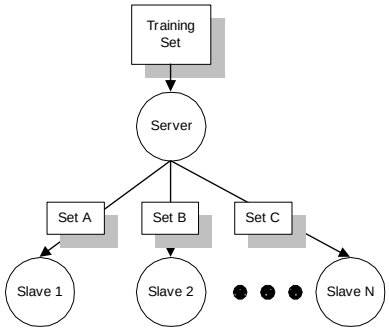



Figure 1 (b) Training sets are sent to the slaves

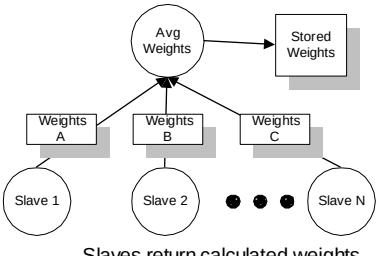
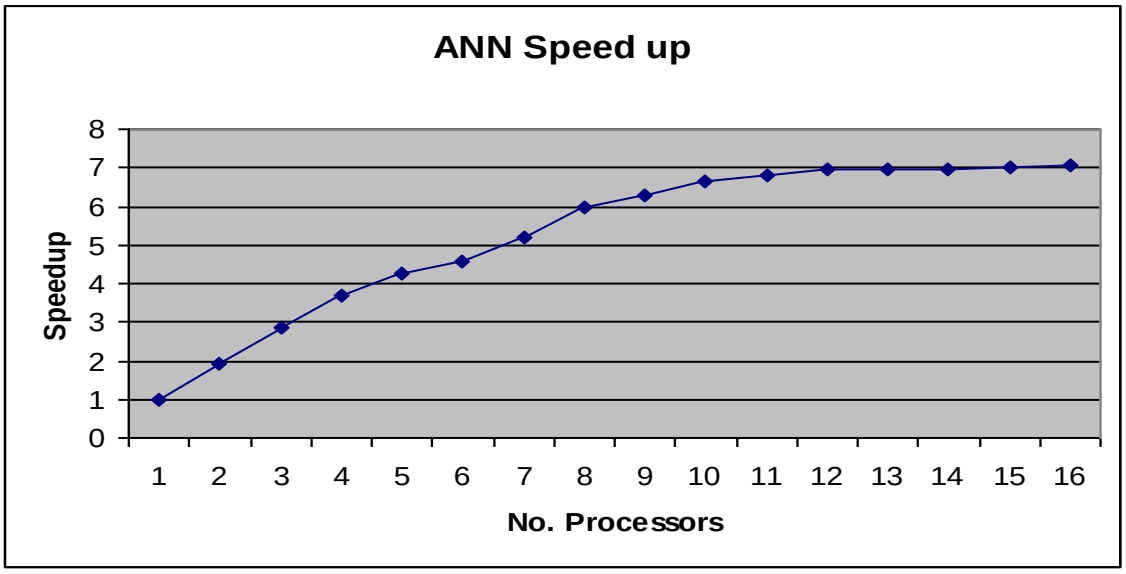



Figure 1(c) Slaves return calculated weights, server generates average weights

Its implementation in the framework requires that only FFFS scheduling is used, as each slave keeps state information. If redundant FFFS was used a calculation for a vector pair would be duplicated, and therefore part C the collection of results would contain invalid information.
The network architecture was 900 inputs, 600 hidden layer neurons and 10 output neurons. This was trained with digit recognition tests using 1920 input patterns, and making only one weight update per pass through the training set.




As can be seen from the graph above parallel speedup was obtained although it is much lower than the volume rendering application. A final test was performed using a large data set size and with 15360 digits processed per update, which on a single machine required 2hrs and 4 minutes to complete, while on 16 machines finished in 9.38 minutes. This represents over 13 times speed increase with using the parallel implementation.

# Appendix E – Original Project Proposal

**BSc Software Engineering Management: Project Proposal**

Student Name: Richard Doyle
Proposed Academic Supervisor: Jon Macey
Date: 2/10/1999

**Project Title**
Investigation into the development of a framework to support parallel graphic pipelines using a Linux/Unix cluster.

**Project Aims and Objectives**
To develop a prototype framework for parallel graphic pipelines on a cluster of Linux/Unix Machines.
To analysis this framework to see what benefits it brings (if any and under what circumstances) over sequential processing.

**University Computing Resources Required**
3+ Linux/Unix Machines connected through a network

**Deliverables**
A requirements specification for the framework
A prototype framework for parallel graphic pipelines on a cluster of Linux/Unix Machines.
A report on the frameworks performance on various algorithms compared to original sequential times.

**Outline of Method**
Investigation into parallel processing and in particular cluster based parallel processing
Investigation into graphical processing techniques, and ways of distributing the data when they are run in parallel
Investigation in distributed computing communication
Requirements elicitation and analysis
Phased Design and Implementation
Tuning
Analysis of performance of parallel algorithms in the framework compared to the original sequential algorithms
Evaluation of the framework

**What is Distinctive honours worthy about the proposed project**
As the maximum processing capability of a single conventional CPU draws closer to its theoretical maximum, cluster based parallel processing is becoming a much more attractive and cheap way to increase the power of computers. At present programmers wishing to run parallel programs have to spend much of their time learning the new environment. The framework should at least provide a quick route for a range of graphical processing techniques.