# Fault-Tolerant Parallel Applications Using a Network Of Workstations

James Antony Smith

Ph.D. Thesis

Department of Computing Science
The University of Newcastle upon Tyne

December 1997

**Abstract**

It is becoming common to employ a Network Of Workstations, often referred to as a NOW, for general purpose computing since the allocation of an individual workstation offers good interactive response. However, there may still be a need to perform very large scale computations which exceed the resources of a single workstation. It may be that the amount of processing implies an inconveniently long duration or that the data manipulated exceeds available storage. One possibility is to employ a more powerful single machine for such computations. However, there is growing interest in seeking a cheaper alternative by harnessing the significant idle time often observed in a NOW and also possibly employing a number of workstations in parallel on a single problem. Parallelisation permits use of the combined memories of all participating workstations, but also introduces a need for communication, and success in any hardware environment depends on the amount of communication relative to the amount of computation required. In the context of a NOW, much success is reported with applications which have low communication requirements relative to computation requirements.

Here it is claimed that there is reason for investigation into the use of a NOW for parallel execution of computations which are demanding in storage, potentially even exceeding the sum of memory in all available workstations. Another consideration is that where a computation is of sufficient scale, some provision for tolerating partial failures may be desirable. However, generic support for storage management and fault-tolerance in computations of this scale for a NOW is not currently available and the suitability of a NOW for solving such computations has not been investigated to any large extent. The work described here is concerned with these issues.

The approach employed is to make use of an existing distributed system which supports nested atomic actions (atomic transactions) to structure fault-tolerant computations with persistent objects. This system is used to develop a fault-tolerant "bag of tasks" computation model, where the bag and shared objects are located on secondary storage.

In order to understand the factors that affect the performance of large parallel computations on a NOW, a number of specific applications are developed. The performance of these applications is analysed using a semi-empirical model. The same measurements underlying these performance predictions may be employed in estimation of the performance of alternative application structures. Using services provided by the distributed system referred to above, each application is implemented. The implementation allows verification of predicted performance and also permits identification of issues regarding construction of components required to support the chosen application structuring technique. The work demonstrates that a NOW certainly offers some potential for gain through parallelisation and that for large grain computations, the cost of implementing fault tolerance is low.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I thank Professor Santosh Shrivastava for suggesting that I look at the problems of parallel applications over a NOW, for his subsequent patient supervision and his helpful criticism of early writings.

I thank members of the Arjuna group, particularly Doctor Mark Little, Doctor Graham Parrington and Doctor Stuart Wheater for help with Arjuna specific implementation issues; Mark and Stuart also for sharing their office for what is now a long time.

I thank family and friends for support and encouragement. I thank especially Pauline Pretzel who has done much to make completion attainable.

I thank also the Engineering and Physical Sciences Research Council for financial support in the form of a studentship grant.

# Chapter 1

# Introduction

After a long history of development and achievements parallel computing remains the exception rather than the norm. It tends to be harder to implement a parallel algorithm to solve a given problem than it is to implement a sequential one. The motivation then tends to be the increase in performance which can be achieved.

Following long experience in parallel computing there is a clear trend to build parallel machines out of commodity workstation units as far as possible. At the same time, much technology developed for parallel machines has translated to the general purpose environment, in particular the network technology. Thus more and more attention is being paid to the use of network machines such as a Network of Workstations (NOW) for running parallel computations. As this process continues it is reasonable to consider performing ever wider classes of computation in a NOW environment. One class of computations which are not well suited to parallel execution in a very low specification NOW is that of *out of core* computations, where problem data is so large that it must be based outside of primary memory.

Other than typically a lower specification interconnect a network machine is also distinguished by a greater autonomy of management, raising issues regarding security, load management and fault-tolerance. In the typically single site NOW environment studied the weight of concern is with the latter two issues. On the one hand a NOW typically executes a rather greater variety of jobs with considerable requirement for interactive response. On the other hand there is a rather greater potential for individual component machines being taken out of service or failing separately and at the same time for detecting such events. Both of these events may be regarded as changing resources.

This thesis describes the problems of performing parallel computations on the changing resources of a NOW and a practical approach to addressing these problems both for traditional NOW oriented applications and also for the out of core problems referred to before.

## 1.1 Parallel Computations

Some brief mention of certain terminology is included here for the sake of completeness. For a fuller background in parallel computing see e.g. [55, 74].

### 1.1.1 Decomposition

A single problem is first decomposed into a number of sequential components which can be executed concurrently.

If the problem is characterised by a significant data structure, such as an array, *domain decomposition* emphasises the partitioning of the data. A component of computation is that which is associated with a particular data partition. For example a finite difference computation is typically decomposed by partitioning the difference array between the available processors with each processor being responsible for performing iterations of updates and boundary value communications for its own data partition.

*Functional decomposition* takes the alternative approach of partitioning the work to be performed and letting the data requirements of each resulting computation component be governed by the work partitioning. The well known traveling salesman problem entails finding an optimal tour which visits each of a group of cities once. One approach to solving this problem is branch and bound whereby possible full tours are evaluated from a selection of currently known partial tours and the search bounded by comparison with the currently best known full tour length. A functional decomposition approaches the parallelisation of this problem by defining the unit of work as evaluation of a single partial tour.

It is possible that there may be alternative decompositions and that a choice between them can only be made in the context of a particular target environment. A complex problem may offer opportunities for both domain and functional decomposition at different levels. One example would be a physical simulation comprising separate components, such as a coupled ocean and atmosphere model.

### 1.1.2 Mapping

Having identified a set of concurrent tasks, it is necessary to establish a mapping of them to the available processors. In a simple case all tasks are of the same size and can be initiated at the start of the computation. In other cases where different tasks have different workloads and may have dependencies it may be possible to derive a satisfactory schedule in advance of execution. In some computations the load distribution between processes changes unpredictably between a reasonably fixed set of tasks and it is possible to incorporate a periodic redistribution of load between the processes. Other computations have very variable load characteristics.

## 1.2 Parallel Machines

Historically the development of parallel programs has been meshed closely with the development of parallel architectures since the primary goal is performance. More recently there has been a tendency to try to move away from such a close coupling, but as yet the target architecture still influences the choice between parallel programming structures. A recent overview of parallel architectures may be found in e.g. [66]. The many different types of parallel machines may be classified in various ways, but a particularly well known system is presented in [54]. Machines in this case are distinguished by the number of instruction and data streams. Briefly the classifications are:

**SISD** Single Instruction Single Data machines are uniprocessors.

**SIMD** Single Instruction Multiple Data machines include array and vector processors.

**MISD** Multiple Instruction Single Data machines have multiple processor units receiving distinct instructions yet operating on the same data stream.

**MIMD** Multiple Instruction Multiple Data machines are general purpose parallel machines.

*Data parallelism* is obtained by assigning computation data elements to separate processors and executing the same instructions in each, i.e. in a SIMD machine. It is possible to exploit data parallelism in an MIMD machine by relaxing the synchronisation, so that while processors execute the same program they do not do so in lockstep. The resulting model is referred to as Single Program Multiple Data (SPMD), but in either case a program which obtains parallelism predominantly through data partitioning is referred to as *data parallel.*

The alternative to data parallelism is *control parallelism*, which is obtained by concurrently executing multiple instruction streams, such as in a pipeline, and is suited to MIMD or MISD machines.

Interest here is focussed on the MIMD class of machines which may be further classified as to whether or not they have hardware support for shared memory.

### 1.2.1 Multicomputer

A multicomputer provides no hardware support for a shared address space. All inter processor communication is through message passing and these machines are sometimes referred to as No Remote Access (NORMA). An example of a multicomputer is shown in figure 1.1. Since resources are distributed, it is relatively easy to scale up the number of nodes and practical machines with very large numbers of nodes have been built. However, to exploit such a machine the user must partition the computation between the separate nodes. It is possible to create a virtual shared address space in software, giving a Distributed Shared Memory (DSM), but the access cost is not uniform.

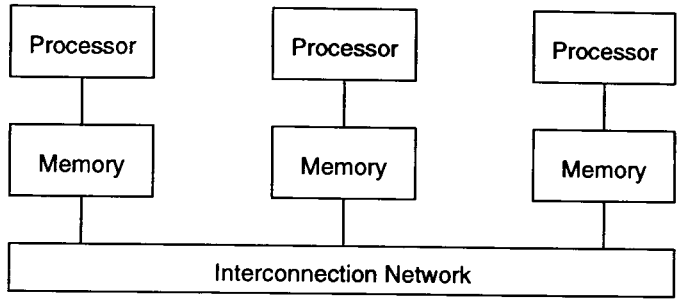



Figure 1.1: Example structure of a multicomputer

### 1.2.2 Multiprocessor

A simple example of a multiprocessor is shown in figure 1.2. The ideal for a shared memory machine,

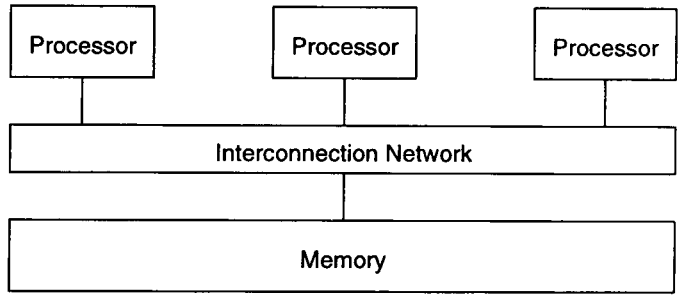



Figure 1.2: Example structure of a multiprocessor

expressed in the well known PRAM model, is of uniform access cost by any processor to all memory. This is a Uniform Memory Access (UMA) machine which may for instance be based on a bus interconnection. Preserving acceptable memory access cost as the number of nodes is scaled up requires a proportionate increase in bandwidth of the interconnection network. Furthermore, even ignoring the effect of caching, the uniform cost model is ultimately limited by physical dimensions.

While there is certainly increased complexity at the hardware level in a multiprocessor, the mapping problem is much simplified. Processes can be moved at will between processors since the processor addressing required in a message passing multicomputer is not needed. Thus processes can be scheduled automatically by an operating system similar to those which might be found on a uniprocessor workstation. A multiprocessor is particularly attractive for irregular problems centred around a dynamic shared data structure.

It is possible to achieve scalability in the number of nodes by distributing memory amongst the

nodes. The overall structure is then similar to that of the multicomputer in figure 1.1, so that again the machine is composed largely of self contained units but with special hardware in each node to support the global address space. Such machines are referred to as Non Uniform Memory Access (NUMA) or DSM machines. Achieving both programmability and scalability is the subject of ongoing work in this area.

### 1.2.3 Large Scale Storage

Beyond primary memory, at one time "core", there is always a need for large scale and possibly longer lasting storage, to hold program sources, documentation, computation input and output data and potentially large intermediate data. The storage hierarchy extends beyond cache memory and primary memory down to secondary storage, typically disk, and subsequently to tertiary storage such as tape or optical jukebox. While random access is possible at any level of the hierarchy, the lower levels are characterised by nonuniform and generally higher access costs. In this work concern is with the large data sets manipulated in out of core computations. This data is assumed to be based at the highest level in the storage hierarchy at which it can be accommodated, typically disk.

Overview coverage of issues related to I/O in parallel computations particularly with regard to achieving high performance in large scale multicomputers may be found in [42, 53].

A widely used approach to increasing disk throughput is to construct a Redundant Array of Inexpensive Disks (RAID) [35] which simulates a single disk with higher performance and/or availability. Various configurations, known as RAID levels, are possible including level 0 (RAID-0) which implements no redundancy, level 1 (RAID-1) which implements disk mirroring. In a multicomputer the parallel processes have to access common secondary storage which is attached to an I/O Processor (IOP) through the interconnection network. To overcome the limiting effect of the single node connection it is typical to stripe data over multiple IOP.

In a conventional general purpose file system, e.g. in UNIX [11], it is reasonable to assume that a file is accessed by a single process at a time. Other than for a database application the concurrency control required for files is quite limited. In parallel computations it is likely that multiple processes manipulate the same data on secondary storage. One option is to structure the data in multiple files, but this introduces a relationship between the file structure and number of processes and the latter may vary between parallel programs or runs of the same program.

To facilitate certain common access patterns by processes of a parallel program to single files, a parallel file system typically allows an access mode to be specified when a file is opened. Typically one such mode defines a single file pointer which is shared between all processes opening the file. This mode can be used to automatically schedule parts of a computation. An alternative allows data to be scattered to, or gathered from, all processes opening the file based on their individual process numbers. In the former case the concurrent access is controlled through the single pointer but in the latter case

it is necessary for all processes accessing the file to synchronise before an access. Such *collective I/O* operations which imply barrier synchronisation points are suited particularly to data parallel program structures. Various strategies may be employed to improve throughput when mapping between different data layouts on disk and memory e.g. [41, 73, 98].

## 1.3 Sharing a Parallel Machine

As a way of increasing throughput for sequential jobs, a UMA multiprocessor may support a similar operating system interface to that of a uniprocessor, such as UNIX. In NUMA and distributed memory machines it is less straight forward to construct such an operating system. Furthermore there is some contention regarding the notion of sharing a parallel machine. Where a user invests in parallelising a significant application, it is not surprising if he seeks exclusive use of a parallel machine. In many cases, the application may justify the cost of a dedicated resource but in the course of an extensive survey [52] makes a case for the growing importance of multiprogramming in parallel systems.

A distinction is drawn between scheduling systems as to whether they are based on space or time slicing. The concept of time slicing is familiar from uniprocessor systems, but in parallel machines it is possible also to partition processors between jobs, this being referred to as space slicing. In space slicing, a parallel job is allocated to a partition of the machine. In principle such a parallel job then has exclusive access to a more restricted machine, but in particular architectures there may still be contention for network resources. In [52] partitioning is classified on the degree to which the partition size is fixed.

**fixed** partitioning entails a system administrator selecting partitions which remain unchanged until the next configuration.

**variable** partitioning sets the partition size to contain a job's requested parallelism, possibly selecting jobs for execution out of queued order to maximise overall machine use.

**adaptive** partitioning is similar to variable partitioning as described above, but also takes account of the load already present on the machine and may load a job into a smaller partition than the degree of parallelism requested. Jobs which fit this computation model are described as *moldable*.

**dynamic** partitioning requires a job to be *malleable* such that its partition size can be varied even during execution.

In a fixed partitioning scheme while no requirement is placed on the parallel jobs submitted fragmentation can lead to wastage of the parallel resource. On the other hand a dynamic scheme cannot give rise to fragmentation, but of the various alternatives this scheme places the greatest constraint on submitted jobs, which must at any time be able to adapt to a varying number of processors. A well

known application structure which fits this requirement is referred to by many alternative names including "workpile", "task queue", "master slave", "farming" and in the remainder of this work "bag of tasks". Essentially the problem is divided into a number of tasks which may each be executed by any of a collection of worker processes. Some central coordinating entity administers the dynamic allocation of tasks to workers.

Time slicing offers the prospect of minimal direct intrusion while allowing full resource utilisation, but there is inevitably an overhead incurred in the scheduling system. Schemes based on time slicing may be categorised as to whether processing elements are scheduled independently or not. An option suited to a UMA machine is to schedule all threads from a central queue. It is also possible to employ local queues or a combination of local and global queues, but separate provision needs to be made for thread placement, i.e. mapping, and load balancing.

It is possible to coordinate processor scheduling in gang scheduling. Typically this achieves time slicing between whole or partial parallel jobs though sharing can be accomplished by space slicing instead. While simultaneous execution of threads supports fine grain interaction as in variable partitioning, the distinguishing feature is that jobs can be preempted so that short jobs can be guaranteed a quick response. In order to accomplish this preemption it is necessary to implement a coordinated context change and the cost of this operation should be small compared to the quantum in a time slicing scheme. Scheduling decisions are typically made at regular intervals, but in lazy gang scheduling jobs execute freely until a waiting job has been delayed for longer than some threshold.

Multiple scheduling disciplines can be combined, for instance supporting different time slicing schemes within different partitions.

Discussion of sharing so far has implicitly assumed that all processes of a job are active whilst the job is scheduled. In a uniprocessor many jobs include a substantial number of blocking requests and if a process issues an I/O request for instance, it waits and another process is readily scheduled to use the processor while the I/O request completes. This carries through to queue based time slicing systems, such as the simple global queue mechanism. In a purely space slicing system however it is up to the application itself to overlap computation with I/O. Gang scheduling systems similarly have no knowledge of application I/O, but again it is possible to ensure overlap between computation and I/O at the application level.

## 1.4 Network Computers

As it has become the norm to allocate a complete machine as a single user resource the possibility of using the spare time on a network of such machines, a NOW, for large scale computations has been considered. The common vision is of essentially free supercomputer scale resources, e.g. [6].

At the scale of a Wide Area Network (WAN), impressive results have been obtained for very large

granularity problems such as number factoring [99]. More recently scientific data processing requiring large scale data transport have been investigated on multiple clusters of workstations distributed over a wide area [34], and interactive wide area supercomputing, particularly incorporating high specification remote visualisation [40].

For a given network computation there is some assembly of separate computers distributed over a local or wide area and connected by a network. When compared with a closely integrated machine, such a resource has a number of identifiable characteristics.

- The individual machines may be of different architecture, or at least of different specification.
- The bandwidth of the interconnection network may be lower than that found in a highly integrated machine. While high speed interconnects such as ATM switches are being used more widely, many existing general purpose clusters are connected via 10 Mbit/s ethernet links. There are some higher speed links available over a wide area.
- The various machines are likely to be administered separately. This means that security may be an issue, but also perhaps that access may be withdrawn at any time, e.g. through reboot. As well as the separate administration it is possible that the machines which are more widely distributed will be more likely to fail independently. In a long running computation machines may be added into the computation at any time.

## 1.5 Changing Resources

Of the various classes of network computer, it is the NOW which this thesis is concerned with. Typically all the machines are at a single site belonging to a single administration. Developments in technology and costs have led to consideration of different models for a NOW. In a commonly used model all a user's interactive processing is performed locally and the user has exclusive rights to the local machine while logged on. In an alternative model, users access a pool of compute processors from a range of machines which would typically be X-terminals but may include workstations which perform jobs such as editing locally. When a user initiates a computation which is to be run remotely, any command if the interface is an X-terminal, that command is scheduled to one of the pool processors. Such a model is assumed for instance in the design of Amoeba [111].

In the first model, if individual users have administrative authority over their machines then it must be assumed that the machines may be rebooted without notice, but in the second model machines can still fail. While exhibiting definite patterns in a particular system, the level of interactive use in a network of privately owned workstations varies considerably [83]. It seems reasonable to assume that there would be significant variation in user activity in the pool of the alternative model. In any case the users may be assumed to expect a response appropriate to interactive work. These considerations make

management of even a constant load of parallel jobs hard since it must be assumed that the resources available to those jobs are changing. The focus of this work lies in the investigation of approaches for utilising resources which are assumed to be changing, but particularly in the context of jobs which make heavy use of non-primary storage.

A change in resources may be notified to the computation, in which case the computation may respond by migrating from one machine. Alternatively if the change occurs and then the computation detects it, the computation must rely on having taken some prior precautionary action, i.e. the provision of fault-tolerance. It is possible to respond to a notified resource change through a fault-tolerance mechanism, but the notification and the likely higher frequency of migration events gives impetus to a search for optimisations not possible in fault-tolerance mechanisms. While these considerations are touched on, this work takes the minimal approach leaving the optimisations for future consideration.

## 1.6 Fault-Tolerance

While fuller discussions of fault-tolerance principles may be found in e.g. [76], certain background is introduced here. A fault can originate in the design or implementation phase and persist undetected until conditions contrive to uncover it. This is typical of software faults whose tolerance is addressed by approaches such as design and data diversity. A fault may instead arise through the in service failure of a correct component. In this case it is sufficient to provide redundancy such that service may continue in the presence of some number of faults and to avoid a fault affecting all redundant components. When a fault is detected it is necessary for some recovery action to be taken to bring the system into a correct state from where correct service can be continued. Recovery actions may be classified as *forward* where some compensating action is taken to bring the system into some new correct state or *backward* where execution is resumed from a *recovery point* where sufficient information had previously been saved to allow restoration. Forward recovery mechanisms are typically employed to respond to particular anticipated faults. In this work provision is for backward recovery.

A distributed system such as a NOW offers potential for such fault-tolerance since the loss of one component can be compensated by those remaining. A common approach to fault-tolerance provision in distributed systems [82] is to define likely failure scenarios, in a failure model, and then include such provision as is necessary to recover from faults which fit the specified model. It may be possible for faults to occur which are outside the chosen model and then recovery is not guaranteed, but this represents a cost trade off. In the Byzantine failure model a fault is regarded as being capable of giving rise to quite arbitrary behaviour, whereby a machine can produce incorrect output or messages. Mechanisms which permit recovery from such faults tend to be quite expensive, so various more restrictive failure models are defined. One such model which is commonly used is the processor crash model, where it is assumed that a failing processor simply stops.

In considering fault detection a distinction can be made between *asynchronous* and *synchronous* systems. In the former no assumption is made regarding the delay in message passing nor about the relative execution speeds of processes. It is then not possible to employ timeouts as a means for accurate failure detection. In the latter it is assumed that both are bounded so as to allow suitable timeouts to be defined for accurate failure detection. In the case where it is impossible to distinguish between a process which is running slow and one which has failed it is necessary to ensure that the recovery action is safe in the event that the diagnosis of a fault was incorrect. This can be accomplished by structuring the computation as a collection of *repeatable*, or *idempotent*, operations.

## 1.7 Structure of Thesis

Chapter 2 considers the issues involved in executing parallel computations in an environment comprising changing resources present in a NOW, outlining typical application environments, approaches used to accommodate resource changes, and the difficulties with managing state on secondary storage. Finally a practical structure based on a shared persistent object store is presented and some example computations described. These example computations are reused throughout the thesis. Chapter 3 presents an approach to modelling computations structured in the way described in chapter 2. This modelling approach can be used both for explaining experimental results and predicting the performance of the applications in different hardware configurations. Chapter 4 describes implementation of the example applications in different workstation configurations and compares performance measured with that predicted. Chapter 5 builds on the work of the previous two chapters in order to make some assessment of the potential for the structuring approach developed. The chapter begins by addressing issues related to organisation of the computation and configuration of hardware including task synchronisation, data caching, problem and hardware scaling and potential benefits of multithreading to overlap communications. After this attention is turned to alternative computation structures and applications. Finally chapter 6 presents some concluding remarks and directions for future work.

# Chapter 2

# Structuring

This chapter considers the execution of a computation in a NOW and what are the implications of the changing resources. Existing work in the various related areas is examined before the emphasis of this work and the practical computation structure are outlined.

## 2.1 Program Models

In physical terms a NOW is certainly a multicomputer, each workstation having its own local memory. Parallel computing over a NOW is commonly based on message passing directly. However, just as a virtual shared address space can be created in software on a closely coupled multicomputer, so too can it be done over a NOW. These approaches can be regarded as supporting parallel programs at a low level. It is also possible to implement an environment at a higher level, either in a structured shared memory or in a certain program structure. The remainder of this section outlines a range of such alternatives.

### 2.1.1 Message Passing

Message send and receive primitives are sufficient for all required communication but there are inevitably certain communication patterns which are used frequently. Furthermore, even if each parallel machine and workstation defines a proprietary message passing interface there is scope for trading off some measure of performance for a degree of portability. Such considerations have led to a number of message passing interfaces.

The p4 system [24] evolved from a need for a higher level interface to synchronisation mechanisms on a shared memory multiprocessor, but now supports a cluster model where groups of processes (clusters) share memory and coordinate internally via monitors but communicate with other clusters via message passing and fits a network of shared memory multiprocessor machines. There is no support for virtual shared memory so that in a NOW where all machines are uniprocessor all communication is via

message passing. A broadcast operation employs a tree rather than a series of unicasts. Global operations such as *sum* and *max* and *min* are implemented by gathering values up a tree and broadcasting the result.

PVM [58] is a widely used message passing system which aims to facilitate interconnection of heterogeneous networked machines into a single Parallel Virtual Machine. Individual machines may be powerful multiprocessors. A computation is divided up into tasks, and each scheduled to a separate machine. As in the case of p4, individual tasks can be parallel computations. Each task has an identifier which can be used to direct communications. Similar global operations are available to those in p4. Shared memory support such as the monitors of p4 is not provided. The popularity of PVM arises because it permits programs to be portable between a wide range of architectures, while the underlying proprietary message passing and task manipulation interfaces vary widely.

MPI [45] represents a standardisation of message passing interfaces, but also places some emphasis on a need for scoping communications activities to facilitate development of parallel software libraries. To this end the standard specifies communicators and contexts which may be passed into library functions or created on the fly within a library function to ensure that communication within a library routine does not conflict with communication outside the routine. The standard refrains from specifying process manipulation mechanisms.

### 2.1.2 Distributed Shared Memory

DSM aims to provide transparent access to any location in a global shared address space simply by address. Separate to the page management there must exist mechanisms to support local and remote process creation and synchronisation, but ideally there is no need to specify the location of data.

The fine granularity at which data is shared makes support for heterogeneity hard to provide. One approach is described for Mermaid [120] where each memory page is restricted to a single type. However special checks are still needed where individual data formats differ between machines and alignment of complex objects to avoid errors in accessing components which might lie in different pages on different machines leads to waste of space.

A simple technique for achieving a global shared address space is to partition the whole address space and locate each partition on a certain node as in Amber [33]. It is then necessary for a thread to move to remote data in order to access it, in Amber by means of RPC calls inserted by a preprocessor. More often data is migrated to a thread in the form of pages. The virtual memory management software may be modified such that a fault to a remote page is intercepted. If the page is not stored locally then it is accessed from its remote location. Read and write faults may be distinguished to enable the DSM support to take different actions in the two cases. It is then necessary to establish mechanisms for finding any desired page and a policy governing its migration. Clearly it is undesirable to transfer a whole page for each separate access, but the alternative of caching a page at the requesting node on first

access leads to the possibility of there being multiple copies. When such a replicated page is updated, it is necessary to either update or invalidate other cached copies, thereby maintaining cache coherence.

The first software implementation of DSM is IVY [77, 110] which experimented with various schemes for managing the page table which tracks movement of shared pages. In a centralised system, a unique manager at a well known location services all requests. By devising a mapping from pages to processors, for instance through block allocation, it is possible to distribute the manager function. The page address can be operated on by the appropriate mapping function to obtain the identity of the manager node. A possible extension allows the ownership of a page to migrate from one node to another, such that the mapping function may only identify the initial owner of the page. The local page table stores a hint to the location of the owner, so that by following such pointers it is possible to locate the page owner.

In the event of local memory exhaustion the policy is to select pages for replacement in the order; not owned, read copies, read owned, write owned, and then according to an LRU policy within each class. It is possible to page to another node's memory rather than disk, and this can be cheap when it entails a simple ownership transfer. Since address space encompasses the network it is allocated on a system wide basis, through a centralised manager, possibly with local managers in addition. A centralised pool of locks is provided for inter process synchronisation.

IVY allowed concurrent readers to cache local copies of the same page and invalidated all out of date copies on a write. This achieves *sequential consistency* [75] whereby a value read is always the latest value written. However the update mechanism can be costly, particularly if the writing process is immediately going to update the same page again. Furthermore it is possible for two separate variables which are updated repeatedly by separate processes to have been placed on the same page at compile time. This situation termed *false sharing* is analogous to the false sharing of cache lines in a shared memory machine. The outcome is that the page or cache line thrashes between the two processes. There is then some incentive to relax the strict consistency requirement and allow writes to be buffered and also to allow concurrent updates to the same page, but the eventual update is more complex.

An example of how this may be done is demonstrated in the Munin system [30]. Munin implements multiple consistency protocols at the level of user objects and provides annotations which the programmer may specify on a per-object basis to tune an application according to the access patterns expected for those particular objects. Programs are written in terms of threads sharing passive objects in a shared memory environment. A shared object is a single variable, though it is possible for objects to be glued together by a special annotation. All synchronisation between threads must be done using the primitives provided by Munin since these contain calls to invoke the underlying consistency protocols. The implementation comprises a preprocessor, a modified linker, library routines and some kernel modifications to support page fault handling and page table maintenance.

Munin introduces the notion of *release* consistency, defined for use in the Stanford DASH processor,

to software DSM. *Release* consistency guarantees that the results of all writes performed by a processor prior to a *release* be propagated before a remote processor acquires the lock which was released. The idea is that where the programmer knows there will be many updates to an object before other threads need to see any of them, he can control the point at which changes are propagated. A thread may use the *acquire* primitive to force through any pending updates from other threads, but the intention is that a thread gains a write share copy of the object through the page faulting mechanism and consequent concurrent modifications are merged at the time of *release*.

Munin supports such write sharing by propagating updates as a set of differences to the altered pages. Thus if two objects are marked as write share, then they cannot give rise to thrashing even if the compiler places them on the same page. If many updates are made to the same object before a *release* these are all coalesced into a single set of differences. The cost incurred is the extra processing required to generate the differences.

The definition does not prescribe that all modifications be propagated at the point of *release*. Such an *eager release* consistency is implemented in Munin however. By contrast *lazy release* consistency [4] defers change propagation until the lock associated with the data item is requested through the *acquire* primitive. This technique allows for reduction in message traffic, but entails maintenance of a precedence relation so that the requesting processor can determine which data needs to be fetched. As in Munin the actual changes are propagated as differences to pages to allow for concurrent writes.

Midway defines *entry* consistency [19] which is similar to *lazy release* consistency in delaying propagation of altered data until it is actually requested. In Midway however, the programmer defines synchronisation objects for shared data. Then when a process acquires one of these variables only the data associated with that variable is fetched.

An alternative approach to supporting a uniform shared address space is to require the user to distinguish between local and global data and local and global accesses. This is the approach taken in Split-C [38] which is a parallel extension to C. The granularity of access is no longer restricted to a page and can be as small or as large as required. Thus at the loss of transparency false sharing is eliminated. Assignment between local and global objects can effect bulk transfer and extra operators support split phase, i.e. asynchronous, get and put operations. The language also supports a variety of synchronisation mechanisms, including barrier. In order to support fine granularity access Split-C typically assumes low latency communications such as Active Messages [115].

### 2.1.3 Structured Shared Memory

Rather than accessing individual addresses in the shared space it is possible to restrict access to be by name to higher level objects. Such objects may be passive data structures or may encapsulate an active element. In this case, false sharing is eliminated because data is accessed by name. Inevitably it is necessary to alter the application level interface, but this can be done without employing a compiler.

Orca [14] is a complete parallel language which supports a shared object programming model where access to shared objects is via method calls. A method which updates the object is guaranteed exclusive access, though to enable synchronisation a method which begins with a guard statement may be concurrently active in the same object. To allow shared object methods to be identified as read only or update, Orca makes restrictions; specifically prohibiting gotos, pointers and global variables. A graph construct is available to support linked structures. This compile time analysis allows Orca to employ shared object replication to implement either update or invalidate strategies and even to switch between the two dynamically according to usage.

Mentat also supports a shared object style of programming since *persistent* Mentat objects retain state in memory between function invocations.

In SAM [96] each shared object is classified as either *value* or *accumulator*. The former have single assignment semantics while updates to the latter are serialised. The implementation is in the form of a runtime library so functions are provided to delineate shared object creation and update, parameterised by the particular shared object. When a shared object update is begun, the object is copied to the requesting processor. In the case of an accumulator update, mutual exclusion is first ensured. A process can read an out of date copy of an object cached on the local processor. SAM provides a mechanism for pushing an object to a remote processor where it will be cached ready for use. Thus SAM supports programming with message passing or shared objects.

Linda [29] defines a small number of extensions to a host language to coordinate access to an associative shared memory space known as *tuplespace*. This space contains tuples which are collections of either values or locations specified by type. Tuples are accessed by matching types and values with the specification in the request. To change an entry in tuplespace, it is necessary first to remove the tuple and then place a new entry in tuplespace with the modified parameters. In this way consistent access to tuplespace is ensured. Conceptually, the tuple matching described above is quite inefficient. However, it is possible to perform significant optimisations in a preprocessor. It is only necessary to seek a match against tuples having the same number and type of parameters as those supplied by the caller; indeed tuplespace may be partitioned by tuple type. Further, it may be that tuple structures may be simplified to reduce or eliminate costly matching. For example, the operations: *out("foo")* and *in("foo")* may be reduced to a semaphore.

### 2.1.4 Large Grain Data Flow

Because of their relatively low level, it is not surprising that message passing systems should easily find widespread application. Again because of the low level though, there may be significant reuse of structuring code possible from one application to another, suggesting scope for higher level programming environments. One approach is based on structuring programs as data flow graphs. Typically a node in such a graph may consist of sequential code which is to be executed as a separate process and

the connecting arcs are communication flows between these processes. The user interface to program development can be through a graphical tool as in Hence [17] or Paralex [9], or a textual language such as PCN [56]. In PCN processes communicate through *definitional* or single-assignment values which are shared through parameter passing. An advantage of such systems is the potential for reusing existing sequential code segments potentially of multiple different languages. PCN also facilitates reuse of the parallel structuring components. Mentat [61] also supports a large grain data flow model, through extensions to C++, but doesn't provide support specifically for code reuse.

### 2.1.5 Data Parallelism

As described earlier, in the purest sense data parallelism implies the SIMD model where each processor operates on an element of the data in parallel. The attraction for the programmer is clearly the single control flow, which also facilitates its inclusion in existing sequential programming languages. To cater for the case where there are not enough physical processors for a large object, it is usual to allocate data elements to virtual processors and map multiple virtual processors to a single physical processor. While logically maintaining strictly a single instruction flow, this mechanism also achieves an increase in computation granularity and can allow such languages to be supported on an MIMD machine which doesn't have the instruction level synchronisation of an SIMD machine. In the same way data parallel languages may be supported on a NOW, e.g. [113, 84]. The result of mapping many virtual processors to a single physical processor lessens the synchronisation requirement a little. Rather than processors running in lockstep at the instruction level, the same program runs in each machine on a different portion of the data. This is the SPMD programming model.

It is not uncommon also to implement an SPMD program directly, using either a standard low level message passing or shared memory environment. Rather than depending on the compiler to choose data partitioning the programmer performs this task by hand. Amongst the enhancements to standard C included in Split-C [38] is provision for spread arrays which gives basic support for data parallel computations. Dome [8] demonstrates how the data mapping can be encapsulated in a class and then evaluated at runtime to allow correct partitioning on heterogeneous machines.

## 2.2 Using Changing Resources

For the purpose of this work, the resource changes a parallel computation may need to be able to react to in a NOW are categorised below.

1. A transient node failure, whereby a machine fails and is restored to service after a short delay.

2. Node replacement, whereby a node is removed from the computing resource, but the total number of nodes is maintained constant through the inclusion of a different node.

3. Node loss or addition, whereby a node is removed or added to the computing resource.

While a transient node failure is an unnotified change, node replacement is notified. Node removal may be notified or unnotified. In general so too may be node addition. If node addition is notified then some global scheduling service is assumed to have allocated the node to the particular computation, whereas an unnotified node addition presumes the computation to have the capability in itself of identifying the new node. It is assumed here that all node additions are notified.

It is correct to say that items 1 and 2 can be regarded as special cases of item 3. If node replacement is not notified, then naturally a mechanism which supports a transient failure is required in addition to that which supports migration from old to new node. Similarly one or both of these mechanisms may be required to support node loss or addition. It is claimed that splitting the requirements in this way permits a natural separation of the mechanisms which tend to be employed.

It is possible to respond to such resource changes either in the context of a single application or in the context of a central scheduling authority by rescheduling whole parallel applications. It is possible that both approaches have use at different granularities of resource change. A given job mix may comprise some jobs which can respond to changes in resource and others which cannot. This thesis is concerned with structuring individual parallel applications so that they can respond to resource changes. The remainder of this section outlines existing approaches to supporting these resource changes.

### 2.2.1 Supporting Transient Node Failure

Tolerating a transient node failure is typically achieved through some form of backward recovery.

In the simplest case, the state of a long running process is saved to disk periodically. Following a failure the application may be restarted from this *checkpoint*. Using a *transparent* checkpointer the application code remains unaltered. Obviously however there is some overhead in time and space. The time overhead is related to the amount of state to be saved, the cost of remote writes and the frequency of checkpoints. Checkpoint size can be reduced by incremental checkpointing whereby only modified data is copied. Even then there may be areas of dead memory checkpointed which increase the overhead significantly. It is possible to provide user callable routines to indicate dead memory which can be excluded [89]. At the cost of increased space, asynchronous checkpointing allows remote writes to proceed concurrently with subsequent computation, by making a local copy of process state. It may be possible also to employ a copy on write mechanism, whereby data is only copied locally if actually altered.

The nonintrusive asynchronous approach to checkpointing is obviously attractive. If computation state is small then this may be quite satisfactory, but if computation state is large it is likely that significant optimisation will be obtained by tuning the checkpointing at the application level, particularly if the computation has a regular cycle. To allow for such tuning a transparent checkpointer may define

routines to call a checkpoint, if a configurable minimum time has elapsed since the last checkpoint, and also to set a maximum time before an asynchronous checkpoint is called [89].

An arbitrary concurrent computation comprises some collection of communicating processes. If one process fails then on recovery it is in general necessary to recover other processes in addition to the one which failed such that the global state of the computation is consistent. Restarting from such a state, it is guaranteed that no inter process communication will be lost. If processes are checkpointed quite independently, then it can be necessary on failure to recover all the way to the start of the computation; the domino effect [94]. A checkpointer for parallel computations must therefore make some provision to ensure that recovery is bounded. One option is to flush all ongoing communications [32] prior to checkpointing all processes. An ordered multicast primitive can be employed to implement this [68]. Alternatively a two phase approach can be employed as in [93]. The disadvantage with a coordinated checkpoint is the need for all processes to recover to the most recent checkpoint after a failure. The approach is conceptually simple however and the interval between checkpoints may be tuned to reduce the failure free overhead to an acceptable level.

For certain computations where the state is evenly partitioned between processors, the checkpointed state can be combined as a parity checkpoint. Such an approach using memory on spare machines is described in [90]. In the simplest case a single spare machine maintains a parity copy of the main data on all active machines such that in the event of a single failure, the lost state can be reconstructed. Tolerance to a greater number of failures can be achieved by increasing the number of parity copies.

It is possible to avoid rollback of all processes on any failure by checkpointing at message exchanges or by logging messages sent. In a pessimistic approach each message is logged before it is sent so that any recovering process can replay from its last checkpoint and fetch messages received prior to the failure from the log. An optimistic approach allows asynchronous logging of messages but maintains sufficient state to ensure that it can be determined which processes apart from the particular one which failed need to be rolled back. In either case, it is necessary that a recovering process replays the same execution during recovery that it performed before failure.

Publishing [91] implements a pessimistic log based checkpointing mechanism in a broadcast based LAN where one node is set aside for all recording functions. Manetho [50] is an example of an optimistic protocol. Each recovery unit, a multi threaded process, constructs a view of the Antecedence Graph which describes a "happened before" description of the overall computation, and piggybacks changes to its graph in messages it sends. The graph has nodes representing recovery points and arcs defining the "happened before" relationship. Each recovery point corresponds to the occurrence of some nondeterministic event and marks the start of a period of deterministic execution leading up to the next recovery point. While Publishing allows independent recovery of an arbitrary number of failed nodes, Manetho optimises the logging process but may require to roll back surviving nodes after a failure.

In a shared memory model it is necessary to ensure redundancy in the shared state. Either the state

is backed up as part of a checkpointing procedure or multiple copies of the state may be maintained. In DSM systems it is usual to take the first approach and save shared state as part of a checkpoint. In a coordinated checkpoint, if each process backs up all pages local to itself, an incremental backup avoids making multiple copies of read only replica pages [25]. Log based checkpointing mechanisms rely on recording interprocess messages to ensure consistency, but inter process interactions through shared data in DSM do not provide such convenient points to build a log. A process execution depends on the precise time at which a request for a cached page arrives from another process. However a log based scheme for sequentially consistent DSM has been demonstrated in [108] and the overheads in this scheme are significantly reduced if lazy release consistency is employed.

In SAM an approach which is similar to pessimistic log based checkpointing is employed [97]. Processes checkpoint independently to memory in other machines whenever they execute one of a set of operations which are not re-executable. Since operations on single assignment values are assured to be re-executable this set includes operations on accumulators and certain operating system calls. All shared objects have a designated main copy which is that held by the owning process. For single assignment values this is the creating process. For accumulators there is only one copy. Making a checkpoint entails saving process local state, comprising stack etc. and all owned shared objects not current with the last checkpoint.

The alternative approach of maintaining replica copies of the shared memory is illustrated by work in Linda. Replication of tuplespace in memory on separate machines is considered in [119, 87]. However, in addition to replicating shared memory it is necessary to implement some sort of process recovery and ensure that the shared state remains consistent in the event of process recovery. In a Linda application, removal of tuples from tuplespace implies certain work needs to be done, and either I/O or the generation of new tuples to output the results of that work. This work and output should not be left incomplete through process failure and recovery.

FT-Linda [12] implements atomic combinations of operations, the ability to define multiple tuplespaces, including stable tuplespaces (through replication) and atomic transfer of tuples between tuplespaces. In the bag of tasks structure, shared data is located in replicated tuplespace. A slave atomically removes a task and replaces it by an *in progress* tuple, such that a monitor process can restore the appropriate work tuple in the event of a slave failing while processing a tuple. As the slave processes a tuple, it writes results into a scratch tuplespace and, on completion of the work, atomically replaces the *in progress* tuple by the contents of the scratch tuplespace. MOM [27] partitions tuples into separate lists, including a *busy* list for work tuples which are being processed and a *children* list for tuples generated by a worker which has yet to call *Done*. The busy list is then similar to the scratch tuplespace of FT-Linda. Plinda [67] backs up tuplespace to disk to ensure availability rather than replicating it and implements transactions as a means of enclosing a worker's operations.

Paralex applies replication to achieve fault-tolerance in data flow style computations. Each compu-

tation node is a replica group with a primary actually performing the computation and the state copied to the secondary.

### 2.2.2 Supporting Node Replacement

In a distributed memory environment all communications between processes on separate nodes must make reference to some node identification. In a NOW this would be the internet number. Furthermore the total resources such as disks etc., are partitioned between the various nodes so that a process can only access directly those which are connected to its current node. In such an environment node replacement, i.e. process migration, has received much attention.

In distributed operating systems, e.g. [15, 47] the aim is to create the image of a single operating environment on top of distributed resources. The precise degree of transparency supported varies from system to system, but is typically high such that an application can make all usual system calls yet not include code to support migration. Ensuring that file and signal operations continue to work after migration just as before requires provision for forwarding system calls, but in the case of a call such as *gettimeofday* it is more appropriate for the call to be always local. Achieving this level of transparency forces a close coupling between migration mechanisms and the underlying operating system. Enabling features include system wide process identification and file name space. Typically it is necessary to make significant changes within the operating system kernel. However, it is possible to make all required changes at the system library level [88]. The modified system library functions maintain the required global name space.

Recently work has been done to implement transparent process migration at a higher level which does not necessitate operating system changes. The approach employed in [31, 103] is to implement wrapper functions for the routines in the message passing library. These wrappers implement the tables necessary for correct message redirection and ordering. State transfer is achieved through coordinated checkpoint and restart. Clearly these systems do not directly support transparent migration of all services, such as continued access to files. It is possible however to achieve such continuity of access if file accesses are all made through the same message passing library, as they are in parallel file systems for NOWs such as [63, 95].

In a shared memory environment replacement of a node needs no extra provision over that required to tolerate a transient failure. The failure typically leads to the program rolling back to the latest checkpoint but then the displaced process is simply run on the new processor and can access shared memory as before. In DSM the same is true at the application level, but within the shared memory support system it may be necessary to take some action, e.g. to update directory information.

### 2.2.3 Supporting Node Loss or Addition

If at a given point a parallel computation comprises a certain number of processes running one per machine then there are simple ways of supporting node loss or addition. Node loss can be supported by migrating the displaced process to another node. The resultant load imbalance can be avoided by provision from the outset of spare nodes, but this extra capacity is wasted in the case of normal execution. An added node can be treated simply as an extra spare, but again this is not fully exploiting the extra resource. These approaches may be acceptable in a closely coupled environment but in the rather more dynamic environment of a NOW, it is desirable for a computation to be more responsive. However supporting such responsiveness within a single parallel application inevitably impacts at the application level. Either the application must explicitly reconfigure in response to a configuration change or it must be implemented in a framework which will respond to the change.

If a computation incorporates dynamic load balancing then reconfiguration is readily supported. If a computation partitions into a number of separate tasks then a convenient organisation is to employ a single master process to schedule the tasks to an arbitrary number of slaves. The master is notified as a slave completes a task and can then schedule a new task to that slave. The structure easily affords tolerance to single slave failures, since control of the computation is centralised on the caller's machine, e.g. [106]. In a batch queuing service, there is no obvious place to centralise control of a computation so the Distributed Resource Manager [37] replicates the master for availability and employs the causal group communications primitives of ISIS to ensure consistent message ordering in exchanges between master and slaves, and master and user interface.

The term *adaptive parallelism* is used in Piranha [69] to describe a programming approach which allows a computation to adapt to changing resources. Piranha implements a load monitor on each node which can signal change of resource. Computations are structured as master, termed *feeder*, and slaves, termed *piranha*. The computation is partitioned into a collection of tasks which may have dependencies. The feeder administers a data structure which controls whether or not any given task may be executed yet or not and feeds the tasks which may be executed to the piranha. The tuplespace of Linda is used for this communication. While not supporting fault-tolerance this approach adds functionality to the basic bag of tasks structure by defining a way of supporting task dependencies. Both FT-Linda [12] and Plinda [67] employ the bag of tasks structure as an example of a computation which can be made fault-tolerant using their respective facilities.

CALYPSO [16] is based on an approach similar to the bag of tasks, an application is structured out of parallel loops possibly separated by serial code. A parallel loop is converted by a compiler to a collection of thread segments, each corresponding to an iteration of the loop. These are distributed between a potentially varying collection of slaves. A slave accesses shared data from a memory server by page faulting but at the end of a thread segment the slave computes a set of diffs for each page and

sends this back to the server. A data structure on the memory server records which thread segments of a given loop have been issued and which completed. While the same thread segment may be issued multiple times if there are not enough for the number of slave processes, it is ensured that all operations are idempotent. It is possible to generalise the approach to employ multiple memory servers [39].

If the number of nodes participating in a data parallel computation is altered it is necessary for the data to be repartitioned. In a data parallel language, this implies changing the mapping of virtual to physical processors. For example, in a Dataparallel C program [84] each processor measures the rate of virtual processor emulations and periodically virtual processors are remapped according to the ratio of emulations on a particular processor to the total rate of emulations over all processors. Redistribution is only done between whole iterations.

Other systems are designed to support adaptive processing for applications implemented directly as SPMD, e.g [8, 49, 92]. These systems operate by redistributing data at suitable remap points, i.e. between complete iterations. It is necessary then either for data to be expressed in a way defined by the system [8, 49] or for the user to provide some definition as to how the data is to be distributed across a given pattern of active processors, e.g. through a procedure called by the system [92].

Many computations are not easily partitioned statically and dynamic load balancing is an inherent feature of any implementation. Cilk-NOW [22] demonstrates how easily such computations can make use of the changing resources of a NOW. Cilk implements a distributed scheduling mechanism for multithreaded computations suited for instance to a game playing program. The NOW version adds a checkpoint mechanism and some facilities to locate free machines.

## 2.3 Management of Secondary Storage

For single host control, RAID storage systems are available as commodity units for attachment to a PC as well as supercomputing storage repositories. There are also implementations based partially or totally in software.

RAID techniques have also been used to implement high performance distributed file systems for workstation networks, such as Swift [26] to support the high data rates required for multimedia data, in Zebra [64] which employs log structured output for higher performance writes and in Xfs [7] which introduces also the use of cooperative caching. These systems support access to high bandwidth storage by single processes.

There has been some development of parallel file systems for NOW environments VIP-FS [63], PIOUS [81], PFSLib [95]. Of these VIP-FS and PFSLib are adaptations from multiprocessor systems. All employ a commonly available message passing library for portable transport and assume the presence of a conventional file system, implementing a mapping from multiple component files to the overall parallel file, but have different emphases. VIP-FS supports the two phase strategy of its parent system

PASSION [41] but also a new strategy called assumed requests which aims to reduce the number of network transfers in a lower bandwidth NOW environment. PIOUS employs a transaction mechanism internally to ensure sequential consistency if multiple clients access the same file. PFSLib is part of a larger system aiming to support Paragon emulation on a workstation cluster for early program development phases and so emphasises support for the file access modes of the Paragon.

While there is much work published on fault-tolerant in core parallel computations very little has been found on applying fault-tolerance to parallel computations which make significant use of secondary storage. In the case of primary memory based shared data a common approach to providing support for transient node failure is some sort of checkpoint mechanism. The size of data on secondary storage is likely to be much larger however, so here the alternative approach seen in Linda systems is assumed. In this case there is some degree of redundancy supported in the storage and also some mechanism which can be employed to ensure application level consistency.

Redundancy at the disk level exists in single host based RAID systems. The network based RAID systems [26, 64, 7] support recovery following failure of a whole node. Assuming that by employing extra storage nodes as hot spares or by replicating data it is possible to achieve any required degree of redundancy in the data, it remains necessary to support application level consistency. Furthermore the issue of adapting to changing resources at the application level needs to be addressed, both in terms of mechanisms and performance impact.

While a file system mechanism such as *sync* in UNIX can ensure a consistent state between separate files in the same file system, it is somewhat crude. Database systems however have long addressed the need to ensure consistency of distributed state at rather finer granularity through the atomic action (transaction) [60] concept. An atomic action has an associated scope within which an update of persistent data is made in a consistent way. A consistent update which may affect multiple possibly distributed objects is intuitively one which is "all or nothing". Through some language level mechanism the programmer can define the scope of an atomic action, typically by some indication of "BEGIN" and "ABORT" or "COMMIT". Formally the properties of atomic actions are described in the following well known way:

**serialisability,** in that an execution consisting of multiple concurrent atomic actions which access shared state appears to execute according to some serial ordering of the atomic actions,

**failure atomicity,** in that all effects of a computation contained within an atomic action are undone on failure of that action,

**permanence of effect,** such that once a state update is committed, it is not lost, barring catastrophic failure.

This work employs atomic actions to implement application level consistency in a series of experiments which form part of a study into the application of a bag of tasks based structuring approach to support exploitation of changing resources by data intensive computations.

A widely used extension to the basic atomic action concept is that of nesting, whereby a number of separate atomic actions are nested within the scope of an enclosing action to achieve the overall effect of a single atomic action [80]. Nesting may be continued to any number of levels. While the effects of nested actions which abort may be undone immediately, effects of committed nested actions may only be made permanent through COMMIT at the top level. A number of systems have been developed to research the use of nested atomic action specifically in a distributed and potentially heterogeneous environment e.g. Argus [78], Camelot [51], Clouds [36], Arjuna [86]. A target application might be a banking system where accounts belonging to a particular branch are maintained on machines local to that branch rather than at a centralised location on a rather larger machine. Without replication of data, the distributed system allows continued access to all data apart from that on a particular failed node. Many further extensions to the basic model are described in the literature, but one which is immediately relevant in this work is the notion of a nested top level action, described in early reports on Argus. The idea is to enable some benevolent side effect to be performed outside of the constraints of an ongoing action hierarchy. The nested top level action steps out of the current hierarchy and begins a new one. Control returns to the original hierarchy on completion of the nested top level action.

Possible approaches to building parallel applications using the atomic action facilities of Argus are described in [13]. For example, a simple implementation of parallel matrix multiplication is described where each slave is a guardian apportioned a part of the output matrix to compute by a master guardian. A slave checkpoints each computed row to stable storage and maintains an integer value determining the next row to be computed. This structure is chosen to optimise performance in a collection of homogeneous machines. However, the strategy prevents other processes from taking over the work of a failed process and can thus degrade performance in the event of a failure occurring. Another application considered is the travelling salesman problem which is less regular than matrix multiplication. Here again the approach suggested is to partition the work statically between slaves, with the master pre-computing the first couple of levels of the search tree. Use of a resilient data type [116], is suggested as a possible way of making the master fault-tolerant, though issues of dynamic load balancing are not addressed. All the computations are structured as in core and no performance results have been published.

Another example of the use of atomic actions to structure parallel programs is Pact [79], but the context is different. Rather than providing for access to distributed persistent state consistently, Pact adds atomic action functions to a sequential language to facilitate fault-tolerant coordination in DSM based programs. However the system is targeted at closely coupled multiprocessors and indeed relies for its log based recovery system on a global clock.

## 2.4 Using A Shared Object Store

The emphasis of the work described here is on implementing parallel computations on the changing resources characteristic of a NOW. Of particular concern are large scale parallel computations which make significant use of state which resides outside primary storage, typically on disk. The computation model which is presented here comprises a shared space of persistent objects which may be distributed over a number of machines. Within this object space individual objects may be replicated to increase availability. A computation is performed by slaves which have no persistent state, but can access the shared objects. Atomic actions are employed to control such concurrent accesses and also to ensure consistency of updates to distributed state. Concurrency between such accesses is controlled by atomic actions. This section begins by describing possible ways of distributing such computations and concludes by introducing a small number of example applications which will be reused throughout the remainder of the thesis.

### 2.4.1 A Static Computation

In a simple static organisation, each slave executes a predetermined allocation of tasks. There is no mechanism for dynamic load balancing and no provision for fault-tolerance. A possible configuration is shown in figure 2.1. In this case all shared data objects are shown residing on a single disk connected to a single machine. If the interconnection bandwidth is sufficiently high, the shared objects may be distributed over multiple machines either in whole or partitioned in smaller constituent objects. Similarly it is possible to distribute objects over multiple disks which are attached to the same machine.

The user starts a Master process, M, which controls the overall computation. The master creates the chosen number of slaves, $S_1$–$S_s$, on separate workstations to perform the computation in parallel, informing each slave of a unique allocation of tasks to perform. Any number of slaves may be created, though if the number exceeds the total number of tasks defined, then some will have no work to do.

### 2.4.2 A Dynamic Computation

If a bag of tasks is added and initially loaded with a description of each task, the load balancing becomes dynamic. The computation may now run more successfully on networks of nonidentical machines, and tolerate better the presence of external load on participating machines. The bag of tasks is simply another object to be located in the shared repository, but need not be disk based.

It is possible for an appropriate slave located on some arbitrary machine to join in an ongoing computation if it knows the identity of all the main objects making up the shared state of the computation, including the bag of tasks. It is convenient to maintain such a list in a single object which will be referred to as a computation object.

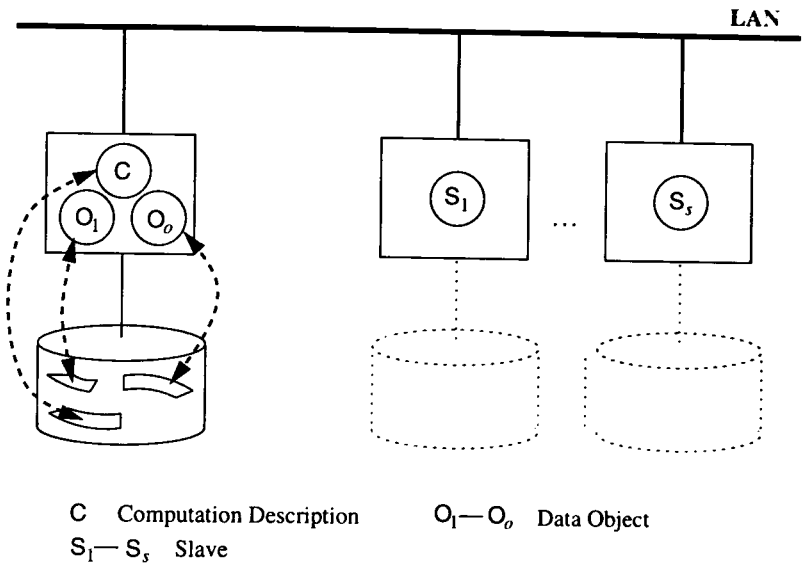



Figure 2.1: Possible distribution of a static computation

### 2.4.3 A Fault-Tolerant Computation

In a static computation there are the following areas of concern:

- A machine hosting a slave may fail. None of the other slaves will take over the unfinished work of the failed slave. In the organisation described so far, there is no record of how much work has been done by the slave.
- A machine hosting shared objects may fail. In the most basic configuration all shared objects are co-located on a single machine. In this case a failure prohibits any further progress by any of the slaves and any results not saved to secondary storage are lost.
- The machine hosting the master, which initiates and subsequently waits for completion of the computation may fail. If the required number of slaves have been initiated before the failure, then the result is simply that the user does not know the outcome of the computation though the computation may progress towards completion. However, if this is not the case, then there may be no further progress although system resources may remain in use.

It is assumed that a workstation fails by crashing and that such crashes can be detected using timeouts in a basic communications layer. In such a crash any data in volatile storage is lost, but that held on disk remains unaffected.

Without a mechanism for dynamic load distribution, it is necessary also to assume that a slave process will be started after a failure to continue work originally allocated to the failed slave and remaining

unfinished at the time of failure.

The following enhancements add fault-tolerance to the static computation.

1. Before starting work on a new task, a slave begins an atomic action and only completes that action after it has finished the work and set a flag in a shared array of recoverable flags to indicate task completion. In the event of slave failure, the action containing an in progress task is aborted and any effects of that task undone. The task completion flag remains unset so that the task can be re-scheduled to a new slave.

2. To ensure tolerance to loss of $k$ copies of a shared object, all objects are replicated on at least $k+1$ machines. This option marks an extreme in a range of possible measures to ensure availability of data. An example of a cheaper option is to depend on parity checking in a local RAID system to tolerate loss of a single disk. The host is assumed to be restarted immediately following a software crash. In the event of a presumed unlikely host failure, it would be necessary to either tolerate longer unavailability or reconnect the storage system to another host.

3. The computation object referred to in section 2.4.2 is sufficient to ensure decoupling of the computation from the user who initiated it. Overall computation status is represented in the array of task status flags, but a possible optimisation is to represent it also directly in the computation object.

### 2.4.4 A Dynamic and Fault-Tolerant Computation

As in section 2.4.3, a slave encloses a task execution within the scope of an atomic action, but in this case a fault-tolerant bag of tasks is employed for task scheduling. It is then not necessary to assume that a slave will be initiated following a failure. Instead the failed slave's unfinished work may be automatically redistributed amongst the surviving slaves.

The slave begins an atomic action before fetching a task from the bag, and commits the action after writing the corresponding result. If the slave fails the action aborts, all work pertaining to the current task is recovered and the task itself becomes available again in the bag, to be performed by any other slave. As in section 2.4.3, the shared objects may be replicated and a computation object defined both to decouple computation and initiator and, in the presence of a bag of tasks, to permit a process to join in an ongoing computation.

Figure 2.2 shows how an application which incorporates these fault-tolerance features may be distributed.

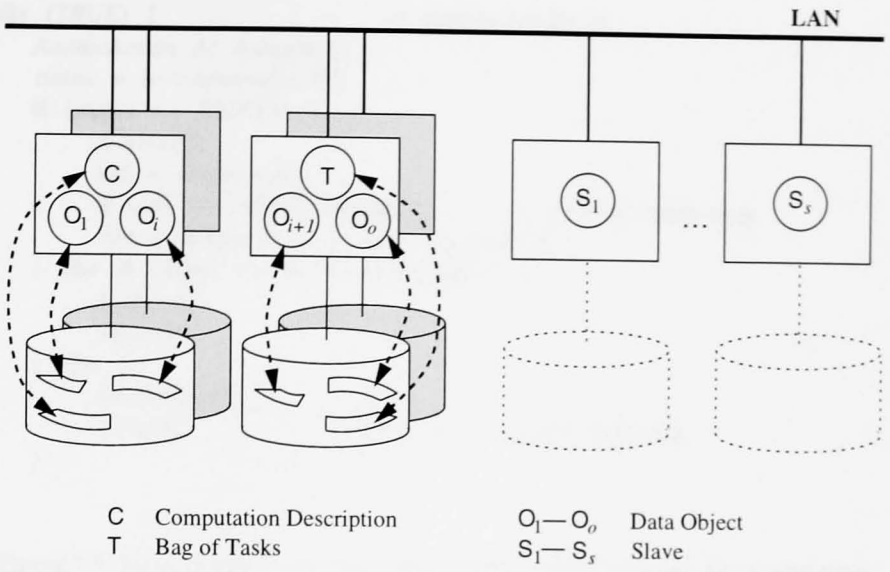



Figure 2.2: Possible distribution of a dynamic and fault-tolerant parallel computation. The shared objects are shown distributed for performance, independent of the replication.

### 2.4.5 Multiple Step Computations

A simple computation can be implemented using a single bag of tasks. However, some computations cannot be decomposed into a single set of independent tasks. It is then possible to divide up a computation into a number of parallel steps as illustrated in figure 2.3. The computation is represented in a sequence of bags of tasks of which one bag is current at any time. All slaves wait, i.e. *pause()*, till the current bag is empty before attempting to select a task, *remove()*, from the next in the sequence. Thus a transfer from one bag to the next marks a barrier synchronisation point. In the example code, an atomic action is implemented as an object and its scope defined by exported operations; *Begin()*, *Abort()* and *End()*. The resulting structure which supports a parallel loop programming style is similar to that described in CALYPSO [16]. However, while in CALYPSO a sequential code segment is executed by the master, it is possible instead to represent such a section of code by a bag containing only a single task.

### 2.4.6 Synchronisation By Side Effect

In some situations it may be preferable to allow tasks to depend on the results generated by other tasks in the same bag. It is then necessary to employ some synchronising mechanism such that a task can be blocked until some prior task has completed and produced output. A simple mechanism is to employ a flag which indicates whether a corresponding task result has been written or not. Concurrent

```
while (TRUE) {                          // dummy condition
    AtomicAction A; A.Begin();
    status = b->remove(work);
    if (status == EMPTY) {
        A.Abort();
        bid = nextbagid();
        if (bid == NONE) break;          // quit enclosing loop
        else { delete b; b = new Bag(bid); }
    } else if (status == NONE_AVAILABLE) {
        A.Abort();
        pause();
    } else {
        dotask(work);
        A.End();          // assume no error in task execution
    }
}
```

Figure 2.3: Slave pseudo code for example multiple step fault-tolerant computation.

access to the flag is controlled through locks obtained within the scope of an action. Where inter task dependencies are satisfied through side effects in this way it is clearly necessary for entries in the bag of tasks to be ordered if deadlock is to be avoided.

### 2.4.7 Multiple Level Computations

Just as it is possible to structure a computation with a sequence of bags of tasks, so too is it possible to employ a hierarchy of bags of tasks. This can be used to encourage each slave to process a sequence of tasks, yet still ensure that all tasks are completed if the collection of slaves changes through the computation. In the simplest case a single higher level bag of tasks contains only references to a collection of bags logically at a lower level. Each slave selects a lower level bag through reference to the higher level bag and then continues processing the tasks in the lower level bag until that is complete. It is necessary for a slave to enclose task executions for the lower level bags in a top level atomic action so that results of completed tasks at this level are not lost on failure of the slave while the entry identifying the lower level bag remains in the higher level bag. In this simple example, illustrated by the slave code in figure 2.4 bags of tasks are used at the lower level for uniformity. However it is possible to place multiple entries in the higher level bag all referring to the same lower level bag to allow a group of slaves to process the same lower level bag concurrently. Clearly it is feasible to generalise further to a hierarchy of bags of tasks. It is also possible to combine elements of the three approaches.

```
    while (TRUE) {
        AtomicAction A; A.Begin();
        status = b1->remove(bid);
        if (status == EMPTY) {
 5          A.Abort(); break;
        } else if (status == NONE_AVAILABLE) {
            A.Abort();
            pause();
        } else {
10          delete b2; b2 = new Bag(bid);
            while (TRUE) {          // process lower level bag till empty
                TopLevelAction T; T.Begin();
                status = b2->remove(work);
                if (status == EMPTY) {
15                  T.Abort(); break;
                } else if (status == NONE_AVAILABLE) {
                    T.Abort();
                    pause();
                } else {
20                  dotask(work);
                    T.End();        // assume no error in task execution
                }
            }
        }
25  }
```

Figure 2.4: Slave pseudo code for example multiple level fault-tolerant computation.

### 2.4.8 Example Applications

Reported applications of a bag of tasks type structure include seismic computations [1, 65], materials science [107] and ray tracing [21]. The main memory requirements are modest and where these examples manipulate disk based state, the data is easily managed by a single disk with all I/O being performed by the master process.

In the work reported here, three examples are employed. The first is a port of a publicly available ray tracing package which might easily be implemented, at least with no support for fault-tolerance, over many alternative infrastructures. The remaining applications are dense matrix computations, matrix multiplication and Cholesky factorisation whose data requirements scale rapidly with the problem size. The latter two serve as examples of computations which, at large scale, may exceed aggregate memory capacity and so be managed as out of core computations.

#### Ray Tracing

One of a number of publicly available utilities to compute ray traced images of a scene described in a simple textual language is *rayshade*. For this package, input data comprises only scene description and output a two-dimensional array containing the red-green-blue values for each pixel. A ray tracing computation can be quite compute intensive even for a fairly simple scene and small image size, where neither data referred to is very large and therefore this must be a good candidate for network parallel execution.

The currently available version of *rayshade* is 4.06 [72], but while this version has attractions in its functionality, an earlier version [71] was modified independently at Yale to run on the network Linda system [21] and therefore offers the possibility of a comparison. Accordingly it is the earlier version that is employed in this work.

The Linda version of the utility is organised as a bag of tasks where the master places in tuplespace a single integer which is set to the highest scanline number. A scanline is a row of pixels in the image and is the unit of work performed by a slave. Each worker process repeatedly reads and decrements this counter, through *in* and *out*, and then computes the pixel values for the required scanline. The worker puts the resulting scanline into tuplespace from where the master retrieves scanlines in the correct order and writes them in sequence to the output file.

The bulk of the Linda code can be embedded directly in the basic master slave structure. A similar approach to that adopted in the Linda version may be employed by wrapping the worker code of the Linda version in a slave structure similar to that employed in the matrix computations. A task is defined as the computation of a certain number of rows of the output array, this number then being the grain size. Each grain of the result is stored as a separate persistent object and computed within a separate task. The Master process which starts up the computation, also copies output data into the correct file

format, Utah Raster RLE format, for further processing such as conversion to postscript.

In a static version each slave is assigned a fixed set of tasks to perform at start up. In an alternative fault-tolerant version, the integer defining the last scanline in the Linda implementation is replaced by a fault-tolerant bag of tasks, with each entry containing the index of the first and last pixel line to be computed. The copy to RLE data format by the Master is not fault-tolerant, but is of quite limited duration compared to the main part of the computation.

### Matrix Computations

Two matrix computations serve as examples of computations which manipulate large state. These are matrix multiplication and Cholesky factorisation. In order to maximise locality and thereby gain greatest benefit from caching at higher levels of the memory hierarchy, it is common to layout matrices according to smaller blocks, or submatrices, and decompose an operation on the original matrix into a combination of operations on the constituent blocks [57]. For convenience, the arbitrary size operand matrices are structured as a collection of square blocks which may be accommodated in slave memory. In each case, a task corresponds to computation of a single block of the result matrix.

### Matrix Multiplication

Conventional block oriented matrix multiplication is performed in which a convenient unit of work to allocate to a task is computation of a block of the result matrix. These tasks may then all be computed in parallel. In the matrix multiplication,

$$\mathsf{C} = \mathsf{AB} \ ,$$

if the $p$ blocks in the $i$th row of A are labelled $\mathsf{a}_{i,1}, \mathsf{a}_{i,2} \ldots \mathsf{a}_{i,p}$, and the blocks in B and C similarly, then

$$\mathsf{c}_{i,j} = \sum_{k=1}^{p} \mathsf{a}_{i,k}\, \mathsf{b}_{k,j}$$

Figure 2.5 shows the essential features of the slave operations. Since all tasks are independent it is again possible to employ a single bag of tasks, each entry specifying the coordinates of the block to be computed within the corresponding task.

### Cholesky Factorisation

Cholesky factorisation computes from a single full matrix two triangular matrices whose product is the original full matrix and which are the transpose of each other; i.e. computing G, given A below.

$$\mathsf{GG}' = \mathsf{A}$$

```
void matmult(int i, int j) {

    sum.set(0);
    for (int k=0; k<A.dl(); k++){
        a = A(i, k);
        b = B(k, j);
        a *= b;
        sum += a;
    }
    C(i, j) = sum;
}
```

Figure 2.5: Function to perform one task in matrix multiplication by bag of tasks.

The example considered here is the factorisation of a dense matrix. This is a relatively simple factorisation to implement yet it demonstrates the potential variety of different ways of structuring a computation which cannot be conveniently partitioned into purely independent tasks. It is convenient to partition the operand matrices into square blocks and employ a left looking algorithm such that each block is written only once. It is only necessary to write half of the result matrix since it is triangular. However there are restrictions on the execution of these tasks. Each block in the output matrix depends on the block in the same block row to its left and if not a diagonal block also on the diagonal block in the same column as shown in figure 2.6.

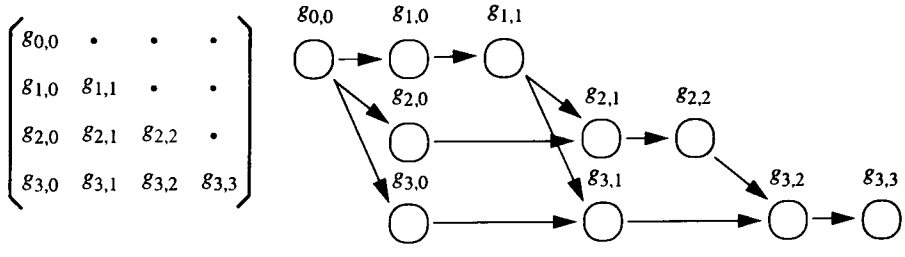



Figure 2.6: Dependencies in left looking blocked Cholesky

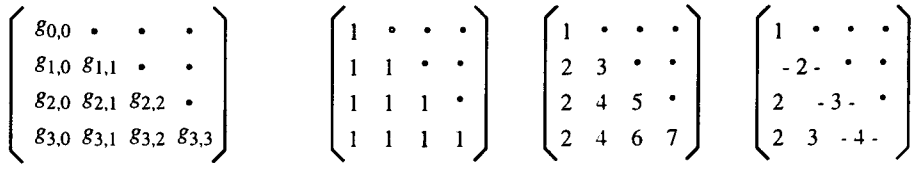
An algorithm which suits this organisation is given in [59, §6.6.5]; the essential features of the slave operation are shown in figure 2.7. Using the same algorithm, there are various ways in which the inter task dependencies may be satisfied, but here just three are highlighted. Conceptually perhaps the simplest is that based on a single bag of tasks, but a synchronisation mechanism must be defined in addition to the bag of tasks and in addition the bag must be ordered in order that deadlock may be avoided. The simplest implementation requires no such synchronisation mechanism but employs

```
void cholesky(int i, int j) {

    sum = A(i, j);
    for (k=0; k<j; k++) {
        // wait for each block if necessary
        w = G(i, k);
        y = G(j, k);
        w *= y.transpose();
        sum -= w;
    }
    if (i == j) {
        sum.chfactor();          // in place block cholesky
    } else {
        diag = G(j, j);
        // solve for x: x*diag = sum, overwriting sum
        sum.solvel(diag.transpose());
    }
    G(i, j) = sum;               // write result
}
```

Figure 2.7: Function to perform one task in Cholesky factorisation by bag of tasks.

multiple bags to achieve barrier type synchronisation points. Each bag contains similar tasks, with alternate bags in the sequence performing block factorisations and block solves. However, there are intervals of sequential code computing the blocks on the diagonal. A third alternative aggregates the task computing a block on the diagonal with the one computing the block immediately to its left to try to achieve some overlap of the sequential work, at least where the preceding column is large. The alternative allocations of tasks to bags are as shown in figure 2.8. The general outline of figure 2.3




(a) **single bag** (b) **multi-bag(1)** (c) **multi-bag(2)**

Figure 2.8: Synchronisation alternatives in Cholesky factorisation by bag of tasks. The order of processing is down columns from left to right. In (b), bag 3 has only a single task, which computes $g_{1,1}$ but in (c) bag 3 has two tasks, the first of which computes $g_{2,1}$ and $g_{2,2}$ and the second of which computes $g_{3,2}$.

may be employed and the single bag of tasks option is simply the special case where the list of bags

specified in the computation object has just one entry. An entry in a bag contains either one or two pairs of integers, defining the coordinates of block or blocks to be computed in the corresponding task.

## 2.5 Summary

This work is concerned with performing parallel computations in a NOW. Specifically the problem addressed is that of exploiting the changing resources available to a parallel computation in a NOW. In order to achieve this, a parallel application should be able to accommodate the unnotified loss and/ or addition of a node. In practice there are seen to be a number of mechanisms which address different parts of this requirement. Checkpointing and replication mechanisms address transient node failures with generally a high degree of transparency. More generally they provide basic support through which an application can tolerate node loss. Migration systems have addressed the particular problem of transferring a process from one machine to another. The general problem impacts all accesses to system resources, but restricted support can be provided in the form of an interposing layer at the interface to a message passing library. On the other hand, using a shared address space for all inter process communication provides support for application migration transparency. Support for reconfiguration to use fewer or more nodes is always a matter for application level structuring.

While approaches exist for structuring in core problems in a suitably dynamic and fault-tolerant way this does not appear to be the case for out of core problems. However there is much work concerned with maintaining availability and consistency of distributed persistent state. Here a parallel computation is structured around a collection of objects in a shared store which are based on secondary storage. The two basic structuring mechanisms are the atomic action and recoverable bag of tasks, the latter having the property that an operation on the bag can be enclosed within the scope of an atomic action and be committed only on completion of that action. Using these mechanisms it is possible to construct various dynamic parallel structures. Three example applications are described which will be used again later.

# Chapter 3

# Modelling

Having defined a computation structure it is necessary to establish an approach to its evaluation. This evaluation is concerned both with the benefit which is obtainable through parallelism and with that obtainable through the fault-tolerance mechanism. In this chapter an analytical model of the computation structure is presented to allow prediction of the overall runtime for any given number of processors. A simple approach is developed which suffices for the class of problems studied here to allow algebraic expressions to be derived for overall runtime in terms of low level parameters, such as disk and communications transfer costs. The intention is not to make a generally applicable model, but purely to support performance prediction for the specific applications picked as examples. In a different example, elements of the approach may be reused, but the detail would be different. Parts of the work in this chapter and early results from the next appear in [102].

While the analytical approach supports estimates of the computation runtime, and allows some consideration of recovery cost, it makes assumptions about the costs associated with the fault-tolerance mechanism. In practice such assumptions seem reasonable if the granularity is large enough, but consideration of this issue is deferred till presentation of experimental results in the next chapter.

## 3.1 Preliminary

A collection of slaves is assumed to be allocated one to each of a collection of processors, which are connected by a network. Each slave has full utilisation of its processor and the collection of slaves have full utilisation of the communications network connecting the processors. The slaves execute concurrently but access main data objects and bag in a shared repository. All accesses to this shared store are serialised, such that any particular access may be delayed arbitrarily. This model is similar to that presented in [1], but emphasises the determination of minimum parallel time and extends the earlier work by showing how to compute bounds on this minimum time, both where there are no inter task

dependencies and where there are dependencies.

Each slave executes a single unique task at a time. The identity of the next task to perform may be regarded as part of the input at the start of a task, and the cost of its access assumed to be small compared to the cost of reading task input data. A task may entail reading and updating objects in shared store. It is assumed that each task comprises three components, summarised in table 3.1: Figure 3.1 shows a

Table 3.1: Bag of tasks basic operations

| Name | Time | Description |
|---|---|---|
| get | $Tget$ | in which input data required to perform the task is read from shared storage. |
| compute | $Tcomp$ | in which computation entailed in the task is performed within the slave machine. |
| put | $Tput$ | in which results computed by the task are written to shared storage. |

visual representation of alternate bag of tasks executions.

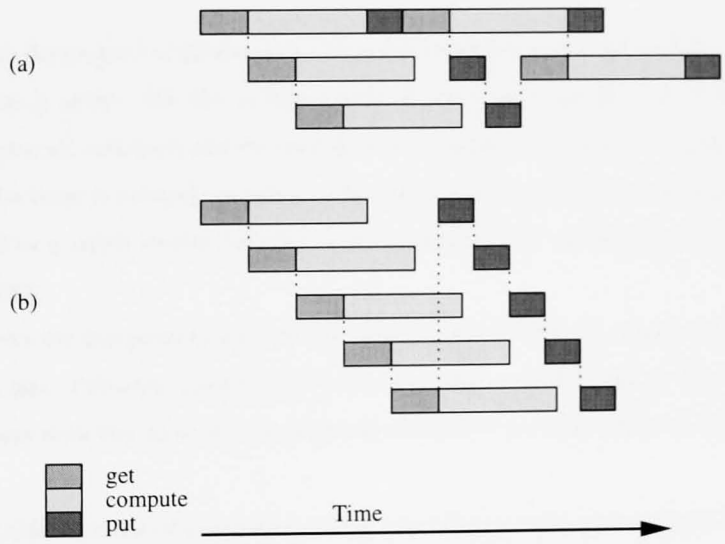



Figure 3.1: Example bag of tasks computation with: (a) 3 processes and (b) 5 processes.

If there is more than one slave, the execution of the computation depends on the pattern of accesses to the shared store. While it is reasonable to assume that the longest waiting slave is served first, no such assumption can be made if two slaves requests are coincident. The overall duration of the computation may vary depending on the order in which slave requests are served. In figure 3.2 when the first process finishes its first task, it competes with the second for access to the shared store.

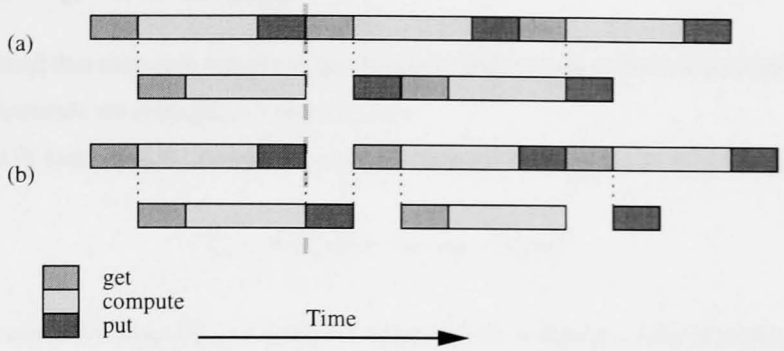



Figure 3.2: Example bag of tasks computation with 2 processes, showing variation in execution time through differing allocation patterns of the shared store.

The task components described above may be of identical duration within different tasks or varied. In the simplest case, they may be contiguous. It is also likely for instance though that data is fetched as it is needed through the task computation rather than all at once, particularly if the data required is large.

The ideal is to determine overall execution time for an arbitrary number of slaves. However in two cases, the analysis is easier. The first is where there is one slave only, the *single slave performance*, when all operations are serialised and the second is the minimum parallel execution time, the *limiting performance*. The latter is certainly achieved when there is a separate slave computing each task, but may be achieved by a rather smaller number of slaves if there is some limiting effect, such as due to network bandwidth.

Ideally all tasks are independent such that an object updated within one task is not read or updated within any other task. However, there may be occasions when it is desirable to incorporate synchronisation mechanisms such that an access to a dependent object is blocked until that object is output by a dependent task.

If there is significant reuse of computation data, it may be possible to benefit through caching such data. Assuming that the shared object store is implemented above a general purpose file system some caching within the file system buffer space may be inevitable.

Finally an overall computation may comprise a number of steps, in which case it is necessary to aggregate performance of all separate steps.

## 3.2 Limiting Performance

First it is assumed that each task comprises get, compute and put components of identical duration and that these components are contiguous within the task.

If there are $N$ tasks overall, a single remote slave performs the computation in time, $T_1$, given by

$$T_1 = N(Tget + Tcomp + Tput) \ . \tag{3.1}$$

Minimum execution time, $T_\infty$, is observed when there is a separate slave processor allocated for each task. All these slaves attempt first to perform initial reads, then computation before writing results. The resultant overall elapsed time depends on the relative values of computation and communication times.

At any point during execution of a task, a slave must be either computing, accessing the shared store or waiting for access to the shared store. Therefore it must be true that

$$T_\infty \leq Tcomp + N(Tget + Tput) \ . \tag{3.2}$$

Otherwise, there must be an interval during which a processor performing some task is neither computing nor accessing the shared store though the task remains uncompleted and access to the store is available. In practice $T_\infty$ is only equal to this upper bound for the trivial case where $Tget = 0$ and/or $Tput = 0$.

Since all communication is serialised, a lower bound on $T_\infty$ is the sum of all communications.

$$T_\infty \geq N(Tget + Tput) \ . \tag{3.3}$$

This bound is closest when computation is dominated by communication; specifically such that

$$Tcomp \leq (N-1)Tput$$

and

$$Tcomp \leq (N-1)Tget \ .$$

A lower bound on $T_\infty$ is defined by the available parallelism,

$$\begin{aligned} T_\infty &\geq \frac{T_1}{N} \\ &= Tget + Tcomp + Tput \ . \end{aligned} \tag{3.4}$$

If computation dominates either input or output, then it overlaps all but one of those operations, i.e.

if

$$Tcomp \geq NTget$$

or

$$Tcomp \geq NTput$$

then

$$T_\infty = Tcomp + N \cdot \max\{Tget,\ Tput\} + \min\{Tget,\ Tput\} \ .$$

Otherwise $T_\infty$ is greater than this value, so it is possible to define the following bound, which is rather closer than (3.4) above;

$$T_\infty \geq Tcomp + N \cdot \max\{Tget,\ Tput\} + \min\{Tget,\ Tput\} \ . \tag{3.5}$$

In general, object store accesses are fragmented through the duration of a task rather than taking place all either at the start or at the end. If the granularity remains the same and if the values $Tget$, $Tput$ and $Tcomp$ are the total times obtained by adding up those for all reads, writes and computation associated with a task then (3.2), (3.3), (3.4) are still valid but (3.5) no longer is.

Combining (3.2), (3.3), (3.4):

$$\text{lwr}\{T_\infty\} \leq T_\infty \leq \text{upr}\{T_\infty\}$$

where:

$$\text{lwr}\{T_\infty\} = \max\left\{\tfrac{T_1}{N},\ N(Tget + Tput)\right\} \ , \tag{3.6}$$

$$\text{upr}\{T_\infty\} = N(Tget + Tput) + Tcomp \ . \tag{3.7}$$

If different tasks have different read, write and compute times then the minimum parallel time is determined by the duration of the longest task, thus:

$$T_\infty \geq \max\{Ttask(i)\} \ .$$

Which task is longest may depend on hardware characteristics, but its duration is bound to be greater than or equal to the average task duration. It is possible then to use the average task length in place of the maximum to determine a lower bound on parallel time. Then similar relationships may be defined to those above. If $Tget(i)$, $Tput(i)$ and $Tcomp(i)$ are the read, write and compute components for the

$i$th task, and

$$TGET = \sum_{i=1}^{N} Tget(i) \ , \ TCOMP = \sum_{i=1}^{N} Tcomp(i) \ , \ TPUT = \sum_{i=1}^{N} Tput(i)$$

then

$$T_1 = TGET + TCOMP + TPUT \tag{3.8}$$

and

$$\text{lwr}\{T_\infty\} = \max\left\{\tfrac{T_1}{N},\ TGET + TPUT\right\} , \tag{3.9}$$

$$\text{upr}\{T_\infty\} = TGET + TPUT + \max\{Tcomp(i)\} . \tag{3.10}$$

The quantities $TGET$, $TPUT$ and $TCOMP$ are the total read, write and compute times obtained by summing those for all tasks.

Corresponding expressions for maximum algorithmic speedup may be obtained, from (3.1), (3.6), (3.7):

$$\bar{S}_\infty \geq \frac{N(Tcomp + Tget + Tput)}{Tcomp + N(Tget + Tput)} ,$$

$$\bar{S}_\infty \leq \min\left\{N,\ 1 + \frac{Tcomp}{Tget+Tput}\right\} .$$

Ideally, if communication cost is negligible then $TGET = TPUT = 0$ and speedup is equal to the number of tasks, i.e. $\bar{S}_\infty = N$. However, if the communication costs are significant with regard to the computation cost, then it is possible for $\bar{S}_\infty$ to be less than $N$.

The maximum number of slaves which may profitably be used is not less than $\bar{S}_\infty$. If the speedup is ideal the number is $\bar{S}_\infty$.

Similarly from (3.8), (3.9), (3.10)

$$\bar{S}_\infty \geq \frac{TCOMP + TGET + TPUT}{\max\{Tcomp(i)\} + TGET + TPUT} ,$$

$$\bar{S}_\infty \leq \min\left\{N,\ 1 + \frac{TCOMP}{TGET+TPUT}\right\} .$$

In this case if communications are negligible,

$$\bar{S}_\infty = \frac{TCOMP}{\max\{Tcomp(i)\}}$$

## 3.3 Non Limiting Performance

Where maximum parallelism is reached analysis is simplified because either serial communication dominates or each task is started in parallel. In the general case each slave executes more than one task but those executions are not constrained to a particular order by the requirements of communication. It is necessary to consider different possible orderings of tasks, though it can be assumed that the longest waiting requester is served first.

Figure 3.3 illustrates example executions for collections of similar tasks, but in this work no attempt

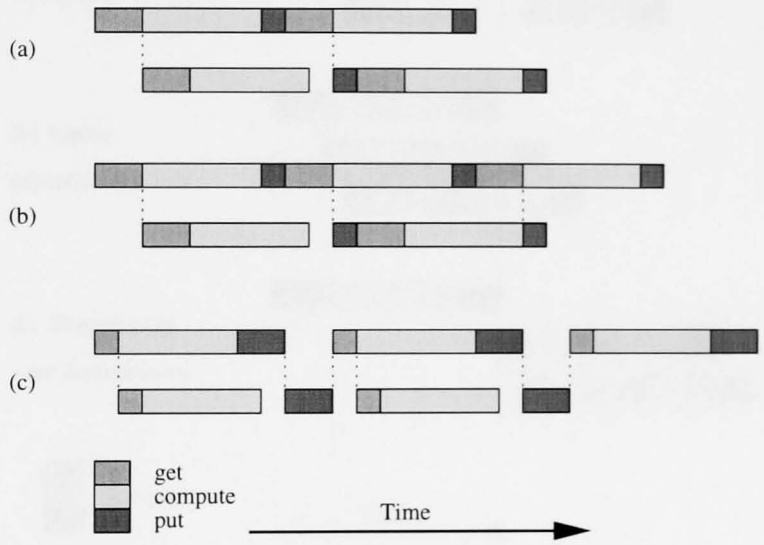



Figure 3.3: Example bag of tasks computations with 2 processes.

is made to determine close bounds on the duration of such executions. Instead the obvious lower bound

$$\mathrm{lwr}\{T_s\} = \frac{T_1}{s}$$

is used exclusively.

## 3.4 Inter Task Dependencies

It is possible to employ a synchronisation mechanism within shared data objects separate to the bag of tasks so that an access to a dependent object is blocked just until that object is ready. Clearly when the computation is performed by a single slave, the synchronisation imposes only an overhead cost. It is however necessary to reconsider the calculation of the execution time when more than one slave participates in the computation. A lower bound on minimum parallel time may be obtained by ignoring

all dependencies such that all tasks execute freely in parallel. A simplification which yields an upper bound is obtained by tightening the dependencies such that rather than relating to data objects, they apply only to complete tasks. The required value is then the upper bound on execution time of this simplified computation.

Figure 3.4 shows a parallel computation divided into three tasks where two of these tasks depend

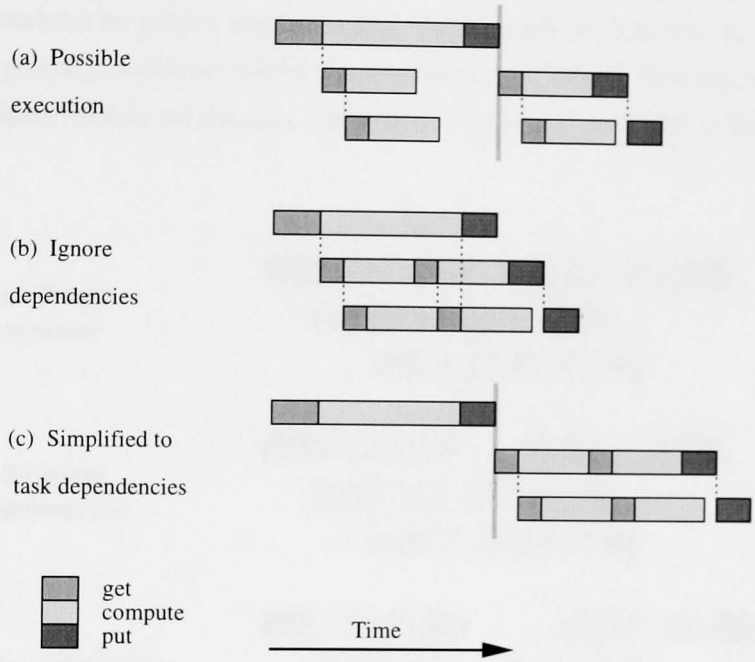



Figure 3.4: Possible bounds on performance of parallel computation in which tasks are not all independent.

at some intermediate point of computation on the output of the third task. Also shown are the lower and upper bounds on computation time obtained by approximation as described above. In this simple example all task computations are all of the same duration and the computed bounds are:

$$\begin{aligned} \text{lwr}\{T_\infty\} &= \max\{3(Tget + Tput),\ Tcomp\}\ , \\ \text{upr}\{T_\infty\} &= \max\{3(Tget + Tput) + Tcomp,\ 2 * Tcomp\}\ . \end{aligned}$$

If the computation is communication bound, such that $Tcomp$ is small then clearly these bounds are very close. However, in the extreme, where communication cost is negligible, such that

$$Tcomp \gg Tget + Tput$$

the upper and lower bounds on computation time may differ by up to a factor of two.

If it is not known how far into a dependent task the dependency occurs then the only sure upper bound is that obtained by assuming that the dependency occurs at the very start of the dependent task. This approach leads to the task dependency approximation above. However, if the computation part of the dependent task may be partitioned into two known amounts, preceding and following the point of dependency, then a tighter bound is possible. The alternative upper bound is obtained by pretending that there is a barrier at the point of synchronisation. In the example of figure 3.4, the actual execution is such that no process proceeds beyond the synchronisation point until all are ready, just as if a barrier were present. However there are situations where this is not the case. One such situation is illustrated in figure 3.5.

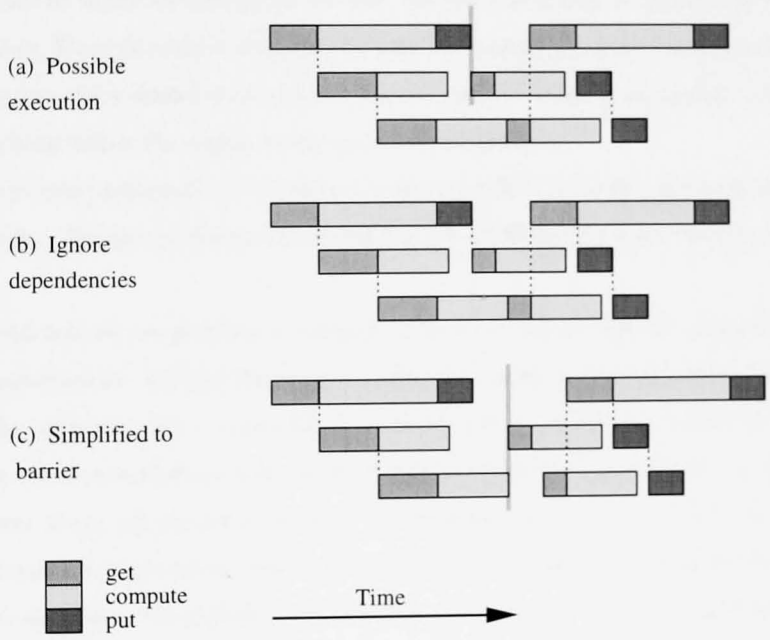



Figure 3.5: Example illustrating alternate bounds on performance of parallel computation which has data dependencies.

In an actual execution the process updating the object depended upon would proceed to compute the next task. Approximating the actual execution with a barrier construct implies pretending that the start of the process's next task execution is delayed till after the synchronisation point but this has the benefit of partitioning the computation into clearly defined phases. The total computation time is then determined simply by adding the computation time of each separate phase. Between barriers, the techniques described in preceding sections may be employed to bound performance of tasks executing freely in parallel.

## 3.5 Cache Effects

So far, all data accesses by application slaves are assumed to be to the shared store, but it is possible to organise the overall computation to attempt to reuse data which has recently been used and may therefore be accessed more cheaply. It is possible to conceive of other levels of storage, but here a three level hierarchy is defined.

1. Slave memory
2. Object server memory
3. Disk

It is assumed that all writes are through to the disk, but that a read may be satisfied within either slave or server memory. Slave memory is obviously private, but server memory is common for all slaves. In a simple realisation of the shared store which is layered above a standard file system, a degree of object server level caching within file system buffer space is involuntary.

In order to predict performance, it is necessary to count the number of reads from each level in the memory hierarchy. The first performance measures to consider are the single slave time and minimum parallel time.

It is assumed that the computation comprises $N$ tasks of various duration and with varying communications requirements. Clearly the memory required to support a given cache strategy may grow with the number of slaves and it is possible in such a case for the execution time to increase as the number of slaves is increased above some value. However, assuming there is sufficient memory to meet the requirements of any chosen cache strategy, the minimum parallel time is still seen when there is an unbounded number of processors such that each task is executed by a separate slave. As before, there can be no time when the shared store is not busy and a slave is neither computing nor accessing the store. The required measures are still given by (3.8), (3.9) and (3.10). However, without detailed knowledge of which blocks are cached throughout the computation, it is no longer practical to identify the maximum task duration. It is then necessary to use the more cautious lower bound on minimum parallel time based on the average task length. The cost of each read which is satisfied within server memory is lower than it would be if requiring a disk access. Each read which is satisfied in local slave memory costs even less and moreover doesn't require access to the shared store.

## 3.6 Recovery

In the event of slave failure and immediate resumption, or replacement by a spare, the failure free execution time is increased by a recovery time. Part of this time is due to the detection of the fault and

depends on communications level timeouts, but the part which is considered here is due to the loss of aborted work. Figure 3.6 shows an example computation where a slave fails but recovers immediately

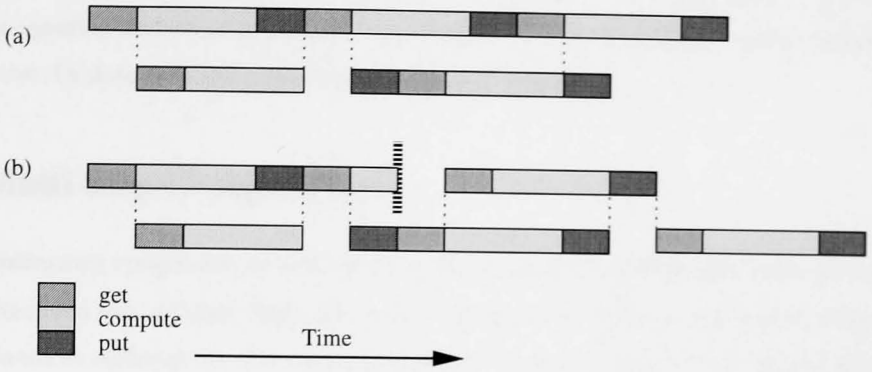



Figure 3.6: Example bag of tasks computation showing (a) a failure free execution and (b) an execution where the slave executing the third task fails but is restored/ replaced immediately.

Clearly at worst where each slave executes only a single task the duration of the computation may be doubled, but in a practical example it is likely that each slave executes a number of tasks. The effect of a failure on overall runtime may be mitigated to some extent if the total number of tasks is not a multiple of the number of slaves. However the worst case occurs where in a failure free execution all slaves finish at nearly the same time. Then assuming that all tasks are of equal cost the actual recovery time is the cost of between zero and one task executions. If a computation is decomposed into tasks of various durations, the recovery cost depends on which task fails. If the computation has inter task dependencies then failure of one task may delay the completion of tasks which are in progress at the time of failure and depend on output from the failed task. If data are cached at a slave which fails, then the slave that takes over the aborted task incurs an extra cost in cache misses.

If a slave fails and does not resume and there is no spare, then the increase in overall execution time depends on the exact point of failure, but may be regarded as comprising two components. First, there is the cost of redoing the failed task and secondly, the execution of the remaining tasks is slowed since there is then one less slave. In the particular case where all surviving slaves are executing tasks which depend on the one which has failed, it is necessary for one of them to abort in order that the failed task may be done and deadlock avoided. In this case the recovery cost due to one slave failure is compounded through the need to complete the task depended on before its dependents.

Clearly the cost of recovery is application specific and then may vary but intuitively if the problem size is large so that there are many more tasks than slaves, then the loss of work on failure should be reasonable. For the purpose of this work a simple measure of the recovery cost which can be derived

from the measured performance is employed as an indication of recovery cost. This measure is computed as half of the maximum recovery on the assumption that all tasks in the computation are of equal cost and is referred to as the *average recovery*. If the total number of tasks is a multiple of the number of slaves then ignoring the cost of any required cache refresh the value referred to is the average expected recovery cost if a slave fails and is immediately restored or replaced.

## 3.7 Multi Step Computations

A computation may comprise more than one basic step separated by barrier synchronisation points.

The execution time of each single step can be predicted in isolation and then combined to give overall bounds on performance. For example a computation may comprise $M$ ideally parallel steps. If the $j$th step has minimum time $Tstep(j)_\infty$ where

$$\text{lwr}\{Tstep(j)_\infty\} \leq Tstep(j)_\infty \leq \text{upr}\{Tstep(j)_\infty\} \ ,$$

then

$$\sum_{j=1}^{M} \text{lwr}\{Tstep(j)_\infty\} \leq T_\infty \leq \sum_{j=1}^{M} \text{upr}\{Tstep(j)_\infty\} \ . \tag{3.11}$$

## 3.8 Examples

The two simple matrix examples described earlier in section 2.4.8 may be employed now to illustrate the preceding discussion. Matrix-matrix multiplication, may be organised into a single bag of similar tasks, i.e. having equal computation, where there are no dependencies between tasks. In contrast, Cholesky factorisation is partitioned into a collection of nonsimilar tasks which do have dependencies.

Table 3.2 lists primitive operations which are used in the following definitions. In much of the discussion which follows all blocks are in fact square. It is convenient to omit the parameters indicating block size in this case, such that the operation times can be written in shorthand *tget*, *tput*, *tmult* etc. to imply that operations are performed on square blocks of size fixed for a given context. A convention adopted in general for such examples is that overall operand matrices are square of size $n^2$ elements and that each such matrix is partitioned into $p \times p$ square blocks of size $b^2$ elements.

### 3.8.1 Matrix Multiplication

Section 2.4.8 describes how block oriented matrix multiplication may be organised as a bag of tasks with a separate task for each block.

Computation of a single block of the result matrix entails performing the dot product of a block row with a block column from the input matrices. This entails $2p$ block gets and 1 block put. In the simple

Table 3.2: Primitive matrix operations.

| Name | Operation time | Description |
|---|---|---|
| put | $tput\{b_1, b_2\}$ | A single block of size $b_1 \times b_2$ is written into shared store. |
| get | $tget\{b_1, b_2\}$ | A single block of size $b_1 \times b_2$ is read from shared store. |
| multiply | $tmult\{b_1, b_2\}$ | Computes the product of matrices x, of size $b_1 \times b_2$, and y, of size $b_2 \times b_1$, e.g. x *= y |
| add | $tadd\{b_1, b_2\}$ | Computes the sum of x and y which are both of size $b_1 \times b_2$, e.g. x += y |
| subtract | $tsub\{b_1, b_2\}$ | Computes the difference of x and y which are both of size $b_1 \times b_2$, e.g. x -= y |
| solve | $tsolve\{b\}$ | In x*l = y or u*x = y, where l and u are lower and upper triangular respectively, x is found. All matrices are of size $b^2$. |
| Cholesky factorisation | $tchol\{b\}$ | In g*g.*transpose*() = a, the Cholesky factor g of a is computed. All matrices are of size $b^2$ and g is lower triangular. |
| LU factorisation | $tlu\{b\}$ | In l*u = a, the LU factors l and u of a are computed. All matrices are of size $b^2$, l and u are lower and upper triangular respectively and one or other has unit diagonal. |

slave implementation above, there are $p$ block multiplications and additions. From (3.1), the value of $T_1$ is obtained:

$$T_1 = p^2(2ptget + tput + p(tmult + tadd)) \tag{3.12}$$

The bounds on $T_\infty$ corresponding to those defined in (3.6), (3.7) are given below:

$$\text{lwr}\{T_\infty\} = \max\left\{\frac{T_1}{p^2}\ ,\ p^2(2ptget + tput)\right\} \tag{3.13}$$

$$\text{upr}\{T_\infty\} = p^2(2ptget + tput) + p(tmult + tadd) \tag{3.14}$$

**Cache Effects**

The following examines the benefit to be gained by caching through some possible application organisations for matrix multiplication. The numbers of reads from the three levels of the memory hierarchy defined are labelled $N_1$, $N_2$, $N_3$. It is also necessary to consider the minimum storage requirement at each level of the hierarchy for a given application organisation and the minimum memory requirements are labelled $m_1, m_2, m_3$.

For the example computation organisations described below, the number of block reads from each of the three possible levels in the memory hierarchy is computed. Also the minimum storage requirement, above that needed only for access, is computed for each level in the memory hierarchy. A summary is given in table 3.3.

1. No attempt is made to benefit from block caching. The memory constraints are that a slave needs to perform an in place multiplication and retain a sum to compute a block dot product and server memory needs to allow transfer between disk and network.

2. If the slaves compute successive blocks of C by row and keep in step through computation of their respective block dot products, then they are each using the same block of A, though a different block of B. This block may be held in server memory.

   If the number of slaves $s$ is less than the number of blocks in a row, then the concurrent computation of $s$ consecutive blocks of C entails $p(s+1)$ block reads from disk and $p(s-1)$ reads from server memory. If the number of blocks in a row is large compared to the number of slaves, then it is reasonable to ignore the effects of transition from one block row to another, and simply assume that there are $\frac{p^2}{s}$ groups of $s$ consecutive blocks to be computed and hence obtain the approximate counts for $N_2$ and $N_3$ given in the table.

   The memory required for this degree of caching at the server is only one block. If more memory is available, but not enough to allow caching of a whole row, then it may be possible for the slaves to become a little out of step, but the performance is the same.

   The same analysis holds for the situation where the slaves compute consecutive blocks in the same column of C or alternatively where each slave is assigned separate block rows of C and computes successive blocks in each of those rows in turn.

3. If as above, the slaves compute successive blocks within the same block row of C, but sufficient space exists within server memory, then the current row of blocks of A may be retained there through computation of a complete block row of C

   Computation of a result block row entails $2p$ disk reads for the first block but for the remaining $p-1$ blocks $p$ block reads from disk and $p$ block reads from server memory.

   Assuming the storage is actually in server file system cache which employs LRU replacement policy, it will not allow retention of the desired block row of A unless there is space for almost all of the block columns of B which are required by $s$ concurrent slaves. The memory requirement is therefore for $p(1+s)$ blocks assuming $s \leq p$.

   However, if the caching were done at the application level, by specifically indicating which blocks to cache, the server would need sufficient memory to accommodate a single block row plus one block to allow access to blocks of B.

4. If the overall size of the operand matrices is small enough then both may be accommodated in server memory.

   The number of unique blocks, and therefore the number of block reads which must be from disk, is $2p^2$. The remainder $2p^2(p-1)$ must all be of already read, and therefore cached blocks.

5. If slaves compute separate block rows of C, then there is potential for benefit in caching part or all of the appropriate block row of A within slave memory. If memory size is quite small, each slave might cache only a few, say the first $c$, blocks of each row read from A. Once cached, these are reused in computation of the remaining $p-1$ blocks of the block row of C

   The memory constraint is for the $c$ cached blocks and one each for; other blocks of the row of A, blocks of the second matrix and the accumulated sum.

6. As before slaves compute separate block rows of C, but here it is assumed that there is sufficient space to accommodate the whole of the required block row of A.

   The storage required is for the $p$ blocks of A. A similar analysis holds if the slaves compute block columns of C and cache block columns of B.

7. Again the slaves compute separate block rows of C, but here it is assumed that there is sufficient space to accommodate all data accessed. This allows reuse not just of the blocks in A, but also those in B

   Each row of A is only read once from remote disk, and thereafter read from slave memory. However, each slave must read all of B into its own memory.

   The slave memory must accommodate all of B and a block row of A.

The block size may be as small as required, but is constrained by an upper bound which depends for the desired level of caching on the main memory available and should be chosen such as to minimise parallel computation time. In table 3.3, expressions are given for the total number of block reads from each level of the memory hierarchy and the minimum memory space required at each level apart from the lowest. These memory requirements are not total requirements for slave and object server, since space required for temporary storage of blocks is ignored.

Performance may be further improved by caching at both server and slave. Where the matrices are relatively small, the best performance is obtained by caching all blocks accessed at slave and server. The memory requirement however, is of order $n^2$ in all machines and highest in the server which needs to accommodate both matrices complete.

For intermediate sized matrices where it is not possible to fit a whole matrix in available memory at slave or server, it may be possible to cache a block row at each slave and a column at the server. Each slave computes a separate block row of the result matrix to optimise local caching, but then the benefit of caching a block column of B at server is reduced as this block column is only used $s$ times before being lost from cache. The memory requirement is of order $n$ in the case of both server and slave.

For arbitrary sized matrices caching a whole block row or column if possible may necessitate setting a very small block size, so an alternative option is to cache a single block at server and a small number of blocks at slave. The memory required is then 1 block in the server and $k$ blocks in each slave.

Table 3.3: Potential benefit achievable using various strategies for exploiting cache reuse in matrix multiplication by bag of tasks. The quantities $N_1$, $N_2$, $N_3$ are the number of block reads from the different levels of the memory hierarchy, respectively slave memory, server memory and disk. The quantities $m_1$, $m_2$ are the minimum space required at levels 1 and 2 of the memory hierarchy.

| | Organisation | Number of accesses and space required<br>Slave memory<br>$N_1$<br>$m_1$ | Server memory<br>$N_2$<br>$m_2$ | Disk<br>$N_3$ |
|---|---|---|---|---|
| 1. | caching no blocks at slave or server | | | $2p^3$ |
| 2. | caching a block of A at server: $s$ slaves | | $\frac{p^3}{s}(s-1)$<br>$8b^2$ | $\frac{p^3}{s}(s+1)$ |
| 3. | caching nominally a block row at server: $s \leq p$ slaves | | $p^2(p-1)$<br>$8nb(1+s)$ | $p^2(p+1)$ |
| 4. | caching all blocks at server | | $2p^2(p-1)$<br>$16n^2$ | $2p^2$ |
| 5. | caching $c < p$ blocks at slave: $s \leq p$ slaves | $cp(p-1)$<br>$8cb^2$ | | $p(2p^2 - cp + c)$ |
| 6. | caching a block row at slave: $s \leq p$ slaves | $p^2(p-1)$<br>$8nb$ | | $p^2(p+1)$ |
| 7. | caching all blocks at slave: $s$ slaves | $p^2(2p-1-s)$<br>$8n(n+b)$ | | $p^2(1+s)$ |

The matrices used are partitioned into $p^2$ blocks of size $b^2$. The computation cost, entailing a block multiplication, is proportional to $b^3$. If $b$ is large, the various transfer costs may be assumed proportional to $b^2$. For all of the cache configurations shown in the table and the three described above, it is seen that the total number of reads is proportional to $p^3$. Since $n = bp$, the total cost of the reads, $Tget$, is in each case proportional to $p$ for fixed $n$. Thus both the single slave time, $T_1$, and minimum parallel time, $T_\infty$, are proportional to $p$, i.e. $1/b$. Even aside from any request overhead associated with transfers it appears profitable for any particular computation organisation to select the largest possible block size. However, it is still necessary to select the optimal cache strategy for given matrix size, memory configuration and primitive costs, and this choice then bounds the block size.

The performance measures for a complete matrix multiplication, i.e. the single slave time and bounds on minimum parallel time, may now be redefined in terms of the values given in the table.

$$\begin{aligned} T_1 &= N_1 tget1 + N_2 tget2 + N_3 tget3 \\ &\quad + p^2(tput + p(tmult + tadd)) \ . \end{aligned} \tag{3.15}$$

The bounds on $T_\infty$ corresponding to those defined in (3.13), (3.14) are given below:

$$\begin{aligned} \text{lwr}\{T_\infty\} &= \max\left\{ N_1 tget1 + N_2 tget2 + N_3 tget3 \right. \\ &\quad \left. + p^2 tput \ , \ \frac{T_1}{p^2} \right\} \ , \end{aligned} \tag{3.16}$$

$$\begin{aligned} \text{upr}\{T_\infty\} &= N_1 tget1 + N_2 tget2 + N_3 tget3 \\ &\quad + p^2 tput + p(tmult + tadd) \ . \end{aligned} \tag{3.17}$$

### 3.8.2 Cholesky Factorisation

Section 2.4.8 describes how block oriented Cholesky factorisation may be organised as a collection of tasks with a separate task for each block. Three alternative schemes were described which differ only in the way in which synchronisation is achieved:

**single-bag** Inter task dependencies are implemented through synchronisation flags employed within shared object methods.

**multi-step(1)** A separate bag of tasks is defined for each task computing a block on the diagonal and again for the tasks computing blocks below the diagonal in each block column. The computation starts with computation of the block at the top left, then those below it, then the next block along the diagonal and so on.

**multi-step(2)** Computation of the block on the diagonal and the one immediately to its left are combined into a single task.

Some of the more lengthy analyses in this section are deferred to appendix A while results are simply quoted here.

First the single slave time is derived. Since synchronisation costs are assumed negligible, the time is the same for each configuration. However, it is separately derived in the **single-bag** and **multi-step(1)** organisations in Appendix A.1.1 and A.2.1.

$$\begin{aligned} T_1 &= \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{2}(p+1)tput \\ &\quad + \frac{p}{6}(p^2-1)(tmult + tsub) + \frac{p}{2}(p-1)tsolve + ptchol \end{aligned} \tag{3.18}$$

**single-bag**

When the factorisation is structured using the **single-bag** organisation, the dependency pattern is as shown in figure 3.7. Each task is composed of a number of block multiplications followed by a factorisation or solve. Where dependencies occur in a task, preceding block multiplications are shown by "✖". Full derivation of performance parameters is given in Appendix A.1.2. Here results are simply

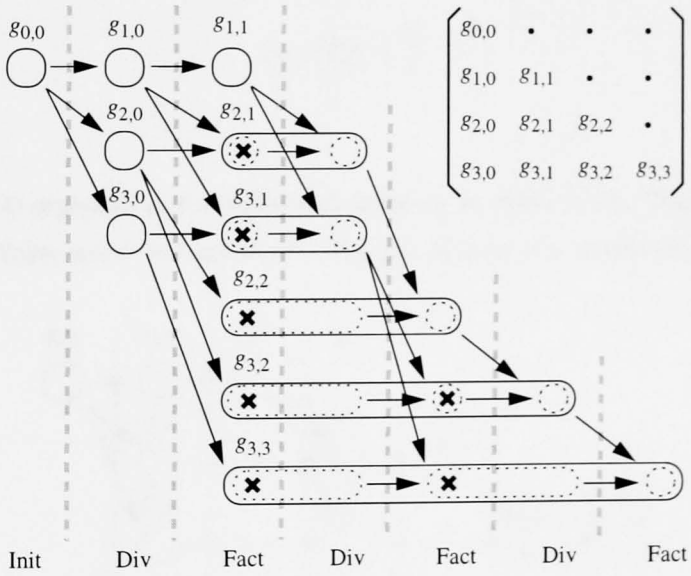



Figure 3.7: Dependencies in **single-bag** organisation of Cholesky factorisation.

quoted.

A lower bound on minimum parallel time may be determined from the hardware bandwidths and length of the longest task, either that computing the block at bottom right or the previous one computing

the block immediately to its left.

$$\begin{aligned}\text{lwr}\{T_\infty\} = & \\ & \max\Big\{\frac{p}{6}(p+1)(2p+1)tget + \frac{p}{2}(p+1)tput\ , \\ & 2ptget + (p-1)\,(tmult + tsub) + tsolve + tput\ , \\ & ptget + (p-1)(tmult + tsub) + tchol + tput\Big\}\end{aligned} \tag{3.19}$$

An upper bound may be obtained by pretending that there are barriers as suggested by the dashed bars in figure 3.7.

$$\begin{aligned}\text{upr}\{T_\infty\} &= \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{2}(p+1)tput \\ &\quad + (p-1)(tmult + tsub + tsolve) + ptchol\end{aligned} \tag{3.20}$$

If there is no bandwidth limiting effect, a lower bound on the time for $s$ slaves to perform the computation is obtained by assuming perfect speedup.

$$\text{lwr}\{T_s\} = \frac{T_1}{s} \tag{3.21}$$

**multi-step(1)**

In the **multi-step(1)** organisation, the dependency pattern is as shown in 3.8. The numbers shown at the bottom of the figure are of the bags of tasks, and correspond to the numbers in figure 2.8(b). Full

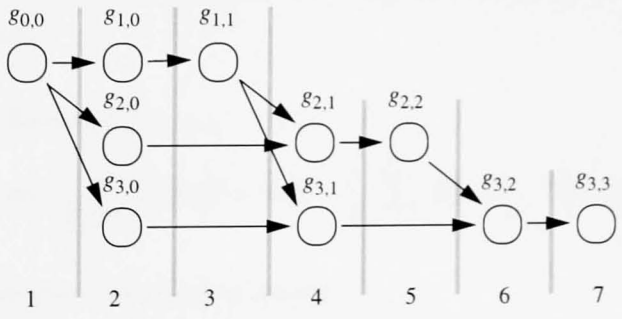



Figure 3.8: Dependencies in **multi-step(1)** organisation of Cholesky factorisation.

derivation of performance parameters is given in Appendix A.2.2. Here results are simply quoted.

A lower bound is obtained by summing the time spent in all serial steps and the lower bound for

each of the parallel steps.

$$\begin{aligned}\mathrm{lwr}\{T_\infty\} = & \\ & \frac{p}{2}(p+1)tget + \frac{p}{2}(p-1)(tmult + tsub) + ptchol + ptput \\ & + \max\left\{\frac{p}{3}(p^2-1)tget + \frac{p}{2}(p-1)tput\ , \right. \\ & \left. p(p+1)tget + ptput + \frac{p}{2}(p-1)(tmult + tsub) + ptsolve\right\} \end{aligned} \tag{3.22}$$

An upper bound is obtained by summing the time spent in all serial steps and the upper bound for each of the parallel steps.

$$\begin{aligned}\mathrm{upr}\{T_\infty\} &= \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{2}(p+1)tput \\ &\quad + p(tchol + tsolve) + p(p-1)(tmult + tsub) \end{aligned} \tag{3.23}$$

This structure of Cholesky factorisation is characterised by significant sequential components. For a small number of slaves a first approximation to the speedup and hence the computation time may be obtained from Amdahl's law [3]. However there is a reducing number of tasks available during each successive current computation step and when the number of tasks falls below the number of slaves the performance attainable during the remaining steps is limited. These latter steps will in fact contain the longest tasks. Furthermore some of the earlier steps, in which the number of tasks is greatest, are more likely to be bandwidth limited. So knowing the characteristics of the computation a better approximation can be made by considering each computation step separately. A lower bound on the time for $s$ slaves to compute the parallel step corresponding to the $j$th column derives from the total communications and the available parallelism.

$$\begin{aligned}\mathrm{lwr}\{Tsolve(j)_s\} = & \\ & \max\left\{\sum_{i=j+1}^{p}\{2jtget + tput\}\ , \frac{1}{s}\sum_{i=j+1}^{p} T(i,j)_1\ , T(i,j)_1\right\}\ . \end{aligned}$$

Then a bound on the time for $s$ slaves can be defined.

$$\mathrm{lwr}\{T_s\} = \sum_{j=1}^{p} T(j,j)_1 + \sum_{j=1}^{p-1}\mathrm{lwr}\{Tsolve(j)_s\} \tag{3.24}$$

**multi-step(2)**

When the factorisation is structured using multiple bags and optimisations are made to try to overlap sequential tasks, the dependency pattern is as shown in figure 3.9. The numbers at the bottom of the figure correspond to those in figure 2.8(c).

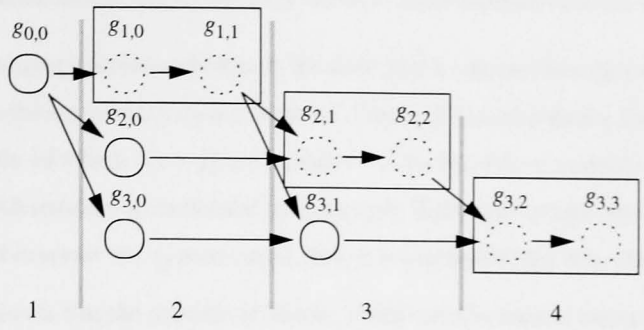



Figure 3.9: Dependencies in **multi-step(2)** organisation of Cholesky factorisation.

The single slave time is the same as that in the **multi-step(1)** organisation. The upper bound on minimum parallel time from that organisation may be used here too and the lower bound on minimum parallel time for the **single-bag** organisation.

In the step corresponding to the $j$th column there is one task computing the block just below the diagonal and the one on the diagonal in the next block column and $p - j$ tasks computing blocks lower in the current block column. A lower bound on the parallel time for $s$ slaves derives from the longest task, total communications and the available parallelism. In a single step

$$\begin{aligned} \mathrm{lwr}\{Tstep(j)_s\} \quad = \quad & \max \left\{ \sum_{i=j+1}^{p} \{2jtget + tput\} + (j+1)tget + tput \ , \right. \\ & \frac{1}{s} \left\{ T(j+1,j+1)_1 + \sum_{i=j+1}^{p} T(i,j)_1 \right\} \ , \\ & \left. T(j+1,j)_1 + T(j+1,j+1)_1 \right\} \ . \end{aligned}$$

Summing for the $p-1$ columns,

$$\mathrm{lwr}\{T_s\} \quad = \quad \sum_{j=1}^{p-1} \mathrm{lwr}\{Tstep(j)_s\} \ . \tag{3.25}$$

**Cache Effects**

Potential performance improvements available through caching are harder to evaluate than for matrix multiplication, but a couple of example situations are described.

1. No attempt is made to benefit from block caching. The memory constraints are that a slave needs to perform an in place multiplication and retain an accumulator block and also to perform an "in

place" block factorisation. Server memory needs to allow transfer between disk and network.

2. If the slaves compute successive blocks in the same block column, then they all read the block row which contains the active block on the diagonal. For the $j$th block column, there are $j(1+2(p-j))$ total block reads, of which $j(p-j)$ are repeats from the block row containing the active block on the diagonal. All remaining reads must be from disk. If the appropriate number of blocks may be accommodated in server file system cache, then it is reasonable that they should be reused.

   The complication is that the number of blocks which must be cached changes during the computation, as the active point moves along the diagonal to the bottom right of the matrix. Firstly the number of blocks in the row containing this point increases. Furthermore the number of entries in the corresponding column decreases so that if a constant number of slaves is working on the computation then eventually they will be working on more than one block column thus necessitating caching of more than one row. However the memory required is certainly not more than 2 full block rows for each slave.

   For a given memory size it is quite possible that all required entries in a block row would be cached during early stages. Later as the computation progresses there may be insufficient space.

3. If the matrix is small enough, then it may all be accommodated in server file system cache and only the first read of any block accesses disk. If no caching is done by the slaves, then this scenario provides an upper bound on performance. The number of disk reads is then equal to the number of unique blocks read, which is $\frac{p}{2}(p+1)$. All remaining reads may be assumed to be from server cache. The number of such reads is $\frac{p}{6}(p+1)(2p+1) - \frac{p}{2}(p+1)$ or $\frac{p}{3}(p^2-1)$.

As in the case of matrix multiplication, the performance predictors for Cholesky factorisation may now be redefined in terms of the values given in the table above. First the single slave time, from (3.18), is:

$$\begin{aligned} T_1 &= N_1 tget1 + N_2 tget2 + N_3 tget3 \\ &\quad + \frac{p}{6}(p^2-1)(tmult + tsub) + \frac{p}{2}(p-1)tsolve + ptchol \\ &\quad + \frac{p}{2}(p+1)tput \ . \end{aligned} \tag{3.26}$$

As described in section 3.5, the lower bound on minimum parallel time appropriate here is that based on average task length rather than that based on the longest task length. For the **single-bag** organisation the bounds on minimum parallel time are: from (A.7)

$$\text{lwr}\{T_\infty\} = \max\left\{ N_1 tget1 + N_2 tget2 + N_3 tget3 \right.$$

Table 3.4: Potential benefit achievable using various strategies for exploiting cache reuse in Cholesky factorisation by bag of tasks. The quantities $N_1$, $N_2$, $N_3$ are the number of block reads from the different levels of the memory hierarchy, respectively slave memory, server memory and disk. The quantities $m_1$, $m_2$ are the minimum space required at levels 1 and 2 of the memory hierarchy.

| | Organisation | Number of accesses and space required<br>Slave memory<br>$N_1$<br>$m_1$ | Server memory<br>$N_2$<br>$m_2$ | Disk<br>$N_3$ |
|---|---|---|---|---|
| 1. | caching no blocks at slave or server | | | $\frac{p}{6}(p+1)(2p+1)$ |
| 2. | caching nominally a block row at server: $s \leq p$ slaves | | $\frac{p}{6}(p^2-1)$<br>$16nbs$ | $\frac{p}{6}(p+1)(p+2)$ |
| 3. | caching all blocks at server | | $\frac{p}{3}(p^2-1)$<br>$16n^2$ | $\frac{p}{2}(p+1)$ |

$$+\frac{p}{2}(p+1)tput \ , \ \frac{T_1}{\frac{p}{2}(p+1)}\Bigg\} \ . \tag{3.27}$$

and from (3.20)

$$\begin{aligned} \text{upr}\{T_\infty\} \ &= \ N_1 tget1 + N_2 tget2 + N_3 tget3 \\ &\quad + (p-1)(tmult + tsub + tsolve) + ptchol \\ &\quad + \frac{p}{2}(p+1)tput \ . \end{aligned} \tag{3.28}$$

Corresponding expressions for the other organisations may be derived in a similar way.

## 3.9 Summary

While experimental measurements are crucial in justifying any computation structure a model helps both in interpreting the measured results and in investigation of possible parameter changes, such as hardware upgrades. For the bag of tasks based structure it is simple to predict single slave time and bounds on bandwidth limiting performance. A lower bound on parallel time for some number of slaves less than that which would lead to bandwidth limiting is obtained easily from the single slave time. For a fault-tolerant computation an important performance measure is the cost of the fault-tolerance. While the failure free overhead will be measured experimentally, it is easy to identify one part of the likely cost of recovery. This is the amount of work which must be redone in the event of a task being aborted. The remaining part which is the cost of detecting the failure is not addressed here but assumed to be

small compared to the former.

In common with accepted practice, the example matrix computations are expressed in terms of block operations to improve locality. Expressions are presented for performance indicators for these computations in terms of the lower level block operations.

# Chapter 4

# Implementation

This chapter considers further the implementation of out of core computations structured to exploit changing resources and compares experimental results against performance predicted by the analysis described in the previous chapter for the example computations introduced in chapter 2.

The implementation employs one of the distributed object systems referred to in section 2.3, namely Arjuna [86]. This particular system is implemented as a class library in C++ and supported over a number of different versions of UNIX including HP-UX and Linux. Arjuna classes provide support for persistent state management and concurrency control with shared read and exclusive write locks. The user encapsulates state in classes which can inherit persistence and concurrency control from certain Arjuna classes. Atomic actions are used for scoping a set of updates, in the same way as transactions. In Arjuna an atomic action is an object which is instantiated and subsequently begun and committed or aborted by calling appropriate operations. Distribution transparency is achieved through a separate underlying layer based on Remote Procedure Call (RPC) [20]. An Arjuna service provides support for transparent replication of user objects. The system has been demonstrated over a number of alternate RPC mechanisms.

## 4.1 Fault-Tolerant Bag of Tasks

A structure which meets the requirements for a fault-tolerant bag of tasks is described in [18] as a recoverable queue and may be regarded as a possible implementation of a semiqueue [116]. Unlike a traditional queue which is strictly FIFO, a recoverable queue relaxes the ordering property to suit its use in a transactional environment. If an element is dequeued within a transaction, then that element is write-locked immediately, but only actually dequeued at the time the transaction commits. Similar use of recoverable queues with multiple servers in asynchronous transaction processing is described in [60], so only a brief description is given here through an example.

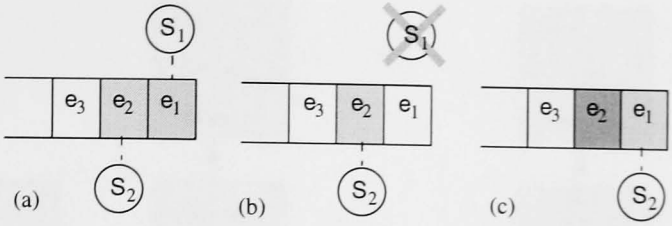



Figure 4.1: Operation of a recoverable queue

In figure 4.1(a), two processes, $S_1$ and $S_2$, are shown having dequeued elements $e_1$ and $e_2$ respectively from this queue. In the absence of failures, say $S_1$ completes processing $e_1$ before $S_2$ completes processing $e_2$, then $S_1$ processes $e_3$. However, figure 4.1(b) shows $S_1$ having failed and its partially completed work aborted, such that $e_1$ is unlocked and so available for subsequent dequeue. Figure 4.1(c) shows $S_2$ having completed processing of $e_2$, now processing $e_1$.

A slave encloses each queue access and corresponding application level task execution within an atomic action. This atomic action guarantees that the slave has free access to the output data corresponding to the task until commit or abort. Any failure of the slave leads to abort of the action, such that any uncommitted output, together with the corresponding dequeue operation is recovered, leaving the unfinished task in the queue to be performed by another slave.

Termination of the computation is detected by testing whether the queue is actually empty or not, as distinct from the condition where no element may be dequeued but the queue is not yet empty.

### 4.1.1 Implementation

A restricted implementation of a recoverable queue has been performed as a collection of persistent objects in Arjuna. A container class **Queue** acts as interface to the object which is represented as a link list of separately lockable instances of **Link** class each with an associated instance of **Element** class. The structure of the composite object is indicated in figure 4.2.

The dequeue operation entails searching along the linked list for an element which is not locked and locking and returning that one. To avoid searching from the start of the list each time, a pointer is maintained in the server process to the first element which is locked. While correctly supporting isolated enqueue and dequeue operations, this approach has the restriction that it cannot support a single client nesting multiple dequeue operations within the scope of a single atomic action, since there is no way to update this marker. However, such a facility is not required in the applications developed here.

A preferable implementation would allow update of the head pointer through nonstandard transaction techniques such as coloured actions [117]. Such an implementation would not suffer from the

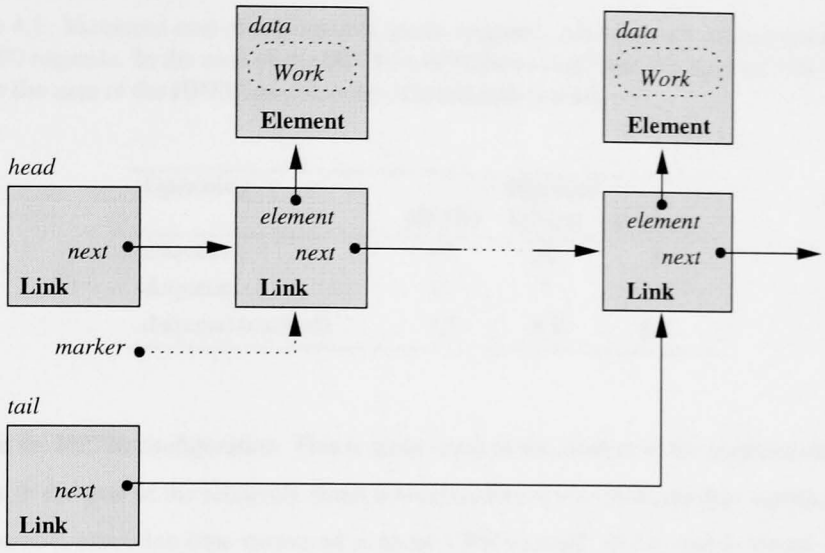



Figure 4.2: Possible implementation of a recoverable queue. Each box is a separately lockable object.

performance penalty of having to start a search from the original head of queue pointer after server failure. Furthermore, it might be possible to avoid making each link a separately lockable object and instead include them all in a single lockable object; the write lock required for update would only be held for a very short time. This would ease the construction of alternative queue structures, such as a priority queue. Each element must remain separately lockable since it is locked by a separate slave throughout execution of the corresponding task.

### 4.1.2 Performance

Since the state of the queue resides on disk, the cost of any queue operation is affected by how much of this state is present in file system cache at the time a request is made. Table 4.1 lists the elapsed time of 100 requests in a number of situations. Initially an empty queue is created and then the time taken to enqueue 100 entries, each enqueue as a separate top level action. The entry type is a pair of integers as used in the applications considered elsewhere. The cost of dequeuing entries is measured both immediately after the entries have been enqueued and also after an attempt has been made to flush file system cash.

The benefit gained in dequeue operations when the whole queue is cached seems to be quite large, but in the applications, the queue resides on the same machine as the main object server. It seems likely that space demanded by large object states will flush the queue state from file system cache rapidly. The measurements then suggest a cost of about 0.4 second per enqueue and 0.07 second per dequeue

Table 4.1: Measured cost of recoverable queue requests. All times are in seconds and are for 100 requests. In the case of the HP710 and Pentium machines the internal disk is used but in the case of the HP730 machine the external disk is used.

| Operation | Machine | | |
|---|---|---|---|
| | HP710 | HP730 | Pentium |
| enqueue | 49 | 38 | 20 |
| dequeue | 11 | 7 | 7.2 |
| dequeue (cached) | 7.5 | 4.5 | 4.2 |

operation on the HP730 configuration. This is quite small in the context of the applications considered. For instance in the case of the relatively small multiplication of two 3000 element square matrices, the minimum parallel execution time measured is about 1300 seconds. If the cost of queue operations in the application setting is as measured then the computation could be partitioned into 100 tasks and the cost of all enqueues and dequeues would be less than 4 %. In the setting of a complete application the variability of the measurement is quite high, partly due to external interference in a general purpose network so it is not easy to make an accurate measurement of the cost of fault-tolerance. However, it is possible to contain an application within a single machine and compare execution time for such runs with and without a queue. Figure 4.3 shows such a comparison for multiplication of two 3000 element square matrices on a HP730. The startup cost of the fault-tolerance provision appears to be roughly

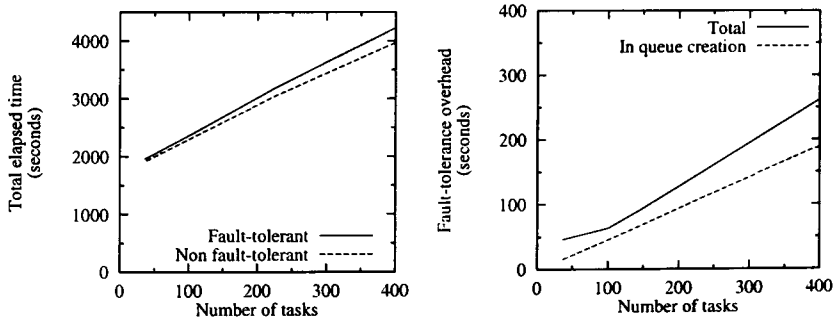



Figure 4.3: Performance of a local fault-tolerant computation

0.5 seconds per task, and the overhead during computation about 0.1–0.15 seconds per task. Both of these measurements are a little higher than the values derived from measurements of queue operations in isolation.

In a parallel execution, the startup cost is unchanged, but the runtime overhead might be expected to vary with the number of slaves. The queue is locked at rather fine granularity and accessed through a

separate server process by each client, so a degree of parallelism is anticipated between two concurrent queue operations. This would suggest that the overall cost of employing the queue should fall as the number of slaves is increased. Such an effect might be countered by costs associated with an increasing number of server processors, including context switches. However, another effect is that as the number of concurrent slaves is increased, so too is the expected number of links each dequeue operation must search along the queue to find the next free entry. For these reasons analysis of queue performance in a general parallel setting is not attempted here. Instead it is left to measurement to show that for a reasonable number of slaves and for appropriate granularity the queue access cost can be small.

## 4.2 Synchronisation

If a computation decomposes into a set of tasks which have inter task dependencies and yet is to be implemented with a single queue, then a synchronisation mechanism extra to the queue is required. The requirement is for a task to be blocked until some prior task has completed and produced output. In a correct execution, static ordering of tasks ensures that a needed object will at least be in the process of being computed when a slave computing a dependent object attempts to access it. However, if the two slaves begin their tasks at about the same time then a race condition arises since there is no mechanism to ensure that the output of the task depended upon is actually locked before the other slave tries to access it. Furthermore, if a slave fails then a dependent slave may attempt to access the output of the aborted task which is neither locked nor ready.

The approach employed here is to allocate a separate object, a recoverable integer, to indicate status of computation objects. Concurrent access to such an integer is controlled through locks obtained within the scope of atomic actions. Updating the appropriate integer within the same atomic action that writes the corresponding state ensures that the computation object remains unavailable in the event of failure of the slave creating it. A fault-tolerant implementation is achieved by making the state of the integer persistent.

In any parallel application, deadlock may arise due to faulty implementation. However if individual process failures are tolerated then any process effectively waiting for completion of such a failed task will block. The queue is ordered so that the first slave to seek work following a failure will take the aborted job. However, if all slaves apart from the failed one are blocked, the application stalls. Rather than including some form of deadlock detection, a simple expedient adopted here is to ensure that slaves do not wait indefinitely. Instead a slave waits only for some application specific interval for any object flag, before aborting its current task and returning to seek work from the queue. This interval should be larger than any period which the slave might genuinely have to wait for, essentially greater than the duration of any task in this application. If a slave incurs such a timeout and then obtains the same task again when going back to the queue, it would be reasonable to increase the timeout, but a collection of

slaves all waiting for a single other might each fetch a different task if they all timed out at about the same time.

## 4.3 Data Transport

A computation is performed in parallel primarily for an improvement in execution time, yet many aspects of distributed systems technology tend to work against this aim, either to provide a safe and easy to use programming environment or to support working in a heterogeneous environment.

At the operating system level, concern for protection between multiple users leads to data in transmit between machines being copied possibly multiple times. Concern regarding this overhead has often been expressed before and many schemes have been proposed to reduce the number of data copies.

A more recent concern regards latency in communication, being signified by the time to transmit a small message rather than the bulk transmission rate for large transmissions. Again various schemes are proposed to reduce the latency.

There are some computations which may be run in parallel with very low communications cost including experiments aimed at harnessing the power of vast numbers of machines across the internet [99, 28]. In such a computation, support for heterogeneity is of considerable importance and the inevitable increase in the communication cost is tolerable because the total volume of communications is low. However if the cost of communications is much more significant, then it is more attractive to optimise data transfers for the assumption of homogeneity.

In distributed systems a common paradigm for communication is RPC [20], which achieves through automatically generated stub codes a user interface to a remote function which is close to a local procedure call. The stub code at one end of the communication path takes care of packing the parameters into a buffer from which they can be unpacked at the other end. Heterogeneity may be supported by converting data items as they are packed into some neutral format.

No attempt is made here to implement optimisations at the operating system level. However, the distributed system employed does allow the choice to be made at application level whether or not heterogeneity is supported and if heterogeneity is not to be supported then saving within the distribution layer is maximum. An application object may define its own marshalling operations. In extreme it may simply redefine the buffer to be a piece of memory in its own data space. This approach allows raw transfer of a single block of memory and is employed in this work.

For complete generality, it would be possible for a user to be able to include a number of such raw blocks together with higher level data which is marshalled in the usual way within a single call. This may be accomplished by linking the raw data blocks into a list structure separate from but associated with the main buffer. At a lower level where system I/O calls are made, scatter gather facilities can be exploited.

## 4.4 Shared Objects

In general to support transfers of many small messages it is desirable to maintain a connection through the period of transfer. The application program actions of object construction, and destruction translate to initiation and termination of a server process which manages the state on the remote machine. The RPC supports a TCP transport option where a connection to a server is established when the corresponding object is initiated and maintained until termination of the object.

However, when making bulk transfers over a shared medium, it is desirable to arbitrate use of that medium. The RPC allows reuse of an existing server for situations where the server does not need to maintain state. In the current single threaded implementation, the object server can be implemented in this way provided a separate connection is established for each call to the server. When data transfers are large the cost of these extra connections is tolerable.

It is possible to increase throughput to the shared object store in many ways. Since the object store is layered above a standard file system one possibility is to employ a proprietary RAID system and stripe files across a number of disks within a single node. Each object store request is then translated to multiple concurrent disk operations, so that a gain is made in the presence of only a single slave. Alternative approaches which exploit parallelism at the granularity of whole requests, make a benefit only when multiple slaves are making concurrent requests.

One alternative is to perform the partitioning within a node at the level of objects. This may be achieved by creating separate object stores on a number of disks. The simplest implementation establishes a separate server process for each store, but in the environment of a shared communications medium, the desired arbitration of that medium achieved by the single process is lost. It is possible however to employ multiple threads to achieve the parallelism, but with a single thread serving the multiple input streams. This option may entail a greater memory usage than the RAID option which seeks parallelism within a single request.

The highest level at which parallelism may be obtained in object store access is the machine level. The objects are distributed between multiple machines and managed by a separate object store server on each machine. In this case, it is necessary to ensure good balance of requests over the separate machines.

### 4.4.1 Management of Large Objects

In common with [80] a state based recovery system is employed in Arjuna. When an object is modified under the control of atomic actions it is necessary to retain multiple state copies in order to support recovery in the event of such actions aborting. For optimal performance in recovery it is desirable to retain these copies in primary memory, but in the work described here these object states can be very large; indeed performance of the matrix computations is best when all physical memory is employed to

support the largest possible block size. There are however various possible strategies for locating these copies.

When an object is updated it is necessary to preserve a snapshot of the state prior to the update to support eventual abort. However on abort of a top level action the state of each object in the action should revert to that before the action started. This is precisely the state which is current for that object on disk throughout the action, so that no volatile state snapshot need be retained through a top level action. While it is necessary to retain separate snapshots through a nested action, the amount of memory retained can be conveniently reduced by updating different objects within different nested actions. By distinguishing between overwrite and update operations, e.g. through different locks, it is possible to avoid an unnecessary initial read of a large state.

When an object is modified within an atomic action, atomicity requires preservation of the newest state of that object in a tentative form up till top level commit. In general there is nothing to stop an application making further updates to any such modified object right up till top level commit, though in practice it may be the case that a series of different objects are updated in sequence. In current Arjuna as in [80] the tentative state of an object is that of the memory resident copy of the object and is written to permanent storage only during top level commit processing. If a number of large objects are updated in the same transaction then it is feasible for memory to be used up leading to undesirable paging. In [80] the possibility of writing state to permanent storage at nested action commit (early writing) is considered as a way of supporting more graceful recovery from failure, and as a simplified form of atomic action checkpointing. However it is there noted that in either case it is necessary to store rather more than simply the object state in order to allow atomic action recovery from that point. Clearly if object state is written to disk at nested action commit and the same object is modified in a number of nested actions within the same top level atomic action then the cost of repeated writes is undesirable. Here however, the application need is to achieve a single early transfer of current state to disk, all be it in a tentative form during the course of a top level action. If this state is not to act as a checkpoint then the issue of saving atomic action context does not arise so it is only necessary to identify a suitable control which is available to the application. Clearly enclosing the object update in a nested top level action achieves the write to permanent storage, but such an action unnecessarily compromises serialisability.

In Arjuna a persistent object is represented by a server in order that operations of that object may be called. This active representation is created at the time of constructor call and removed at the time of destructor call. The call to the object destructor might be a suitable control by which the user can achieve the desired early write. If the object is not locked in the scope of a nested atomic action then the only possible rollback must be to the state current just before the start of the current top level action so that the only actions which need be taken with regard to this object are to either commit the current tentative state or to revert to the latest permanent state on disk. If at the time of the call to the destructor the object is locked within the scope of a nested action then it may be necessary to revert to the state

existing before the start of that nested action. In this case writing the tentative state might be deferred till resolution of the nested action. Similarly if a copy of state current just before the start of the current top level action is preserved in memory at all during that action it could be removed at the point of destructor call. In the example computations studied here there is only one instance where two blocks are written in the same task; in one of the implementations of Cholesky factorisation.

## 4.5 Atomicity

In database terms, the slave is coordinator for the atomic action and the machines hosting shared objects are participants. The coordinator has responsibility for ensuring consistency between distributed state which is updated during the course of an atomic action. Through the well known *two phase commit protocol* [60] the coordinator can ensure that all distributed state is correctly updated eventually regardless of intervening participant failures. In this work however it is important to be able to tolerate failure of the coordinator, i.e. slave. In a simple database system tolerance to coordinator failure can be achieved through the coordinator writing locally a persistent record called an *intentions list* which details the updates to be committed. In the event of coordinator failure it is then possible to ensure that eventual commit is consistent with notification to a human operator, but such failure can lead to "blocking" such that the database items locked during that transaction remain unavailable until the failed coordinator is restored. The general problem of tolerating coordinator failure without the need for such blocking is addressed by nonblocking commit protocols [10]. Here however the application characteristics can be exploited so as to minimise the cost of tolerating coordinator failure without blocking. In the bag of tasks structure the user is concerned only with the outcome of the overall computation, not individual actions. A simple solution then is to always abort any incomplete work in the event of a slave failure and let an alternative slave redo the corresponding task.

The correctness requirement is that each task description must remain in the bag until corresponding work is completed. Assuming each task entails computing, from read only parameters, a unique output and then writing it, idempotency is guaranteed and correctness may be ensured by careful ordering of updates during commit processing. It is sufficient to commit objects in the reverse of the order in which they were touched within the action. By contrast in the case of the asynchronous transaction processing referred to earlier, the use of a response queue to reliably inform a human operator of completion status of each queued transaction ensures that operations are not idempotent. In such a case it is possible to use sequence numbers to avoid duplication of queue entries [18]. The RPC subsystem is responsible for detecting orphan processes and terminating them cleanly [85].

If dependencies exist between tasks of a single bag of tasks or if operations are not idempotent the situation is less straight forward.

- In the simplest case, an object is not depended upon and a write to it can therefore be completed

at any time prior to task completion, i.e. completion of the dequeue operation.

- Alternatively, an object may be depended upon by a task other than that which created it. In this case a flag may be associated with the object and any write to the object must be completed no later than update of the flag. The flag update itself is idempotent and must be completed no later than task completion, i.e. completion of the *dequeue* operation.

- Finally, an object may be written within a task which is not idempotent. An example of a computation where such tasks can arise is an in place factorisation. In this particular example an output object is depended upon by tasks other than that which created it, so it will have an associated synchronisation flag. It is then necessary that completion of a write to the object must be consistent with update to the corresponding flag. To avoid need for such strong consistency requirement, a slave can avoid a repeat execution by explicitly testing the flag status as it begins a task. The consistency requirement is then reduced such that it can be satisfied by ordering the updates as in the previous case.

  The same explicit test on a flag object can be used to ensure correctness where an object is not depended upon, but is not strictly necessary. It is for instance sufficient for the required status to be represented within the object itself.

Overall, it is desirable to achieve a close integration between an object and its associated status flag.

It is typically convenient to structure a computation such that a single object is updated in each task, and therefore each separate top level action. However, it is possible to update multiple objects within a single task. If each update is idempotent then the requirement is that the queue be updated last and that an associated status flag be updated after the corresponding object. If some object update is not idempotent then it must be consistent with the corresponding status flag update, unless a specific check is made as described earlier.

In the case of the object store used in this work, it is easy to be certain of what is an update and what is an overwrite. In general the same approach can be used, but only through reliance on information concerning the particular object store used.

One possible way of overcoming the difficulties associated with operations which are not idempotent is to always transfer commit to a particular machine such as that hosting the recoverable queue.

## 4.6 Experiments

The three applications introduced in section 2.4.8 are implemented. Preliminary results of these experiments were presented in [101]. The first is a port of a publicly available ray tracing package which might easily be implemented, at least without provision for fault-tolerance, over many alternative infrastructures. The remaining applications are dense matrix computations, matrix multiplication and Cholesky

factorisation. These both manipulate large amounts of data and may therefore exceed aggregate memory capacity.

### 4.6.1 Configurations

A major part of the university's general purpose computing provision comprises clusters of HP710 machines connected by separate 10 Mbit/s ethernet segments, the clusters being connected via routers to a common link. A few of these clusters have also one or two HP730 machines. The HP710 and HP730 are both based on the PA-7000 processor but are clocked at differing speeds of 50 and 66 MHz respectively. The installed machines all run HP-UX 9.01 and have 32 and 64 Mbytes of physical memory, 64 and 256 Kbytes of cache memory respectively.

A number of Viglen genie PCI P5/133 machines having 32 Mbytes main memory and 256 Kbytes secondary cache are installed in an experimental laboratory and connected by two alternative networks. In the first case, connection is via Fast Etherlink (PCI) Adaptor from 3Com to LinkBuilder FMS 100 Stackable Fast Ethernet Hub from 3Com. There are two hub units giving a total of 24 ports. Alternative connection is via EN155p-MF ATM Adaptors from Efficient Networks to a single ForeRunner ASX-200WG ATM switch from Fore systems. The switch is configured with 16 155 Mbit/s ports. Hard disks are connected via fast SCSI 2 controllers. Most of the machines have 1 Gbyte IBM Pegasus disks on which a 256 Mbyte scratch partition is defined. Two of the machines have in addition a pair each of MAXTOR MXT-540-SL disks. All the machines are running Linux with kernel version 2.0.23.

Three experimental configurations are employed:

**HP** The first experiments attempt to employ the general purpose computing facility. In the case of the ray tracing application, the data manipulated is small so that the object store can be located in the rather small temporary space available on one of the HP710 workstations. However for the matrix computations, sufficient temporary space for even moderate size examples exists only on the HP730 workstations. In the matrix experiments reported all objects are co-located on a HP730 machine.

**fast** In a relatively small scale experiment using the Pentium machines, the object store is located on the scratch partition of the IBM disk connected to a single machine and communications between slaves and object store is exclusively via the fast ethernet.

**ATM** In the third configuration, which also uses the Pentium machines, the object store is distributed between the four MAXTOR disks, providing an aggregate 2 Gbytes of storage. On each of the two machines hosting objects, the *md* [121] software is used to manage the two local disks as a RAID-0. Communications between slaves and object store is in this case via the ATM network.

A master process is employed directly for computation start up and shutdown, but not actively during the computation. In these experiments, no object is replicated.

Since the HP workstations form part of the university's general purpose computing provision, access is never exclusive. Experiments are performed during off-peak hours within any of the few clusters which is mostly free, and results from multiple runs averaged. By contrast while there is some teaching use of the Pentium based cluster, it is possible to obtain near exclusive access to the cluster for periods.

## 4.7 Preliminary Results

Figure 4.4 shows one of the example scene descriptions included in the *rayshade* distribution. This scene is employed in the experiments described here. Version 3 of rayshade was ported to Linda at

```
/*
 * This file is the result of feeding "balls" from Eric Haines'
 * SPD through nff2shade.awk and then hand-tweaking things.
 */
maxdepth 3
eyep 2.1 1.3 1.7
lookp 0 0 0
up 0 0 1
fov 45
screen 256 256
background 0.078 0.361 0.753
surface s1  0.15 0.1 0.045  1. 0.75 0.33  0. 0. 0.  0. 0. 0. 0.
plane s1 0 0 1 0 0 -.5
surface s2  0.035 0.0325 0.025  0.5 0.45 0.35  0.8 0.8 0.8  3. 0.5 0. 0.
sphere s2 0.5 0 0 0 texture bump 0.3 scale 0.04 0.04 0.04
sphere s2 0.166667 0.272166 0.272166 0.544331
sphere s2 0.166667 0.643951 0.172546 0
sphere s2 0.166667 0.172546 0.643951 0
sphere s2 0.166667 -0.371785 0.0996195 0.544331
sphere s2 0.166667 -0.471405 0.471405 0
sphere s2 0.166667 -0.643951 -0.172546 0
sphere s2 0.166667 0.0996195 -0.371785 0.544331
sphere s2 0.166667 -0.172546 -0.643951 0
sphere s2 0.166667 0.471405 -0.471405 0
light 0.288675 point 4 3 2
light 0.288675 point 1 -4 4
light 0.288675 point -3 1 5
```

Figure 4.4: Example ray tracing scene Description "balls".

Yale, and a small number of HP710 workstations here have support for Network Linda version 2.4.5, so it is possible to compare the performance of the fault-tolerant implementation with the Linda version in the same configuration.

Figure 4.5 shows the measured parallel performance of the ray tracing computation for an output image size of $512^2$ both with and without fault-tolerance and also shows a comparison with the Linda version.

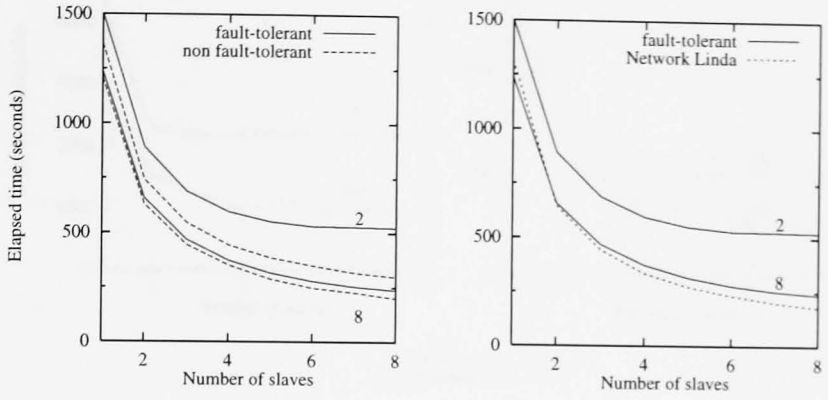



Figure 4.5: Measured performance of parallel ray trace of example image "balls" at size $512 \times 512$ pixels, using task sizes 2 and 8, and showing the cost of fault-tolerance.

The comparison with the Linda version is not completely fair in that the Linda version was designed to employ the somewhat small task size of 1 row and this remains unaltered. However, communication cost in the Linda version is considerably lower than in the fault-tolerant version where output is to disk. The intent is to demonstrate that the implementation is not totally unreasonable in terms of absolute performance so long as the granularity is appropriate.

Figures 4.6 and 4.7 show the performance of the fault-tolerant multiplication of two $3000^2$ matrices and Cholesky factorisation of a $4800^2$ matrix

### 4.7.1 Performance Summary

Table 4.2 summarises the performance of the parallel implementations measured in the **HP** configuration. The table shows for each application a measure of the performance achieved and estimate of the average recovery time. The table also indicates the total data: input (*input*), written (*put*) and read (*get*) collectively by slaves during the computation.

For all three experiments it is seen that increasing the task size improves the performance. In the matrix computations, the increase in total data read with decreasing block size seems to be the overwhelming effect. In the ray tracing example little data is read, but at 25 Kbyte and 98 Kbyte the task output is not so large as to be bandwidth limited and so the larger task is cheaper proportionally.

Noting that the data format conversion for ray tracing mentioned earlier takes about 23 and 13 seconds respectively for the task sizes of 2 and 8, the performance of this easy application appears prom-

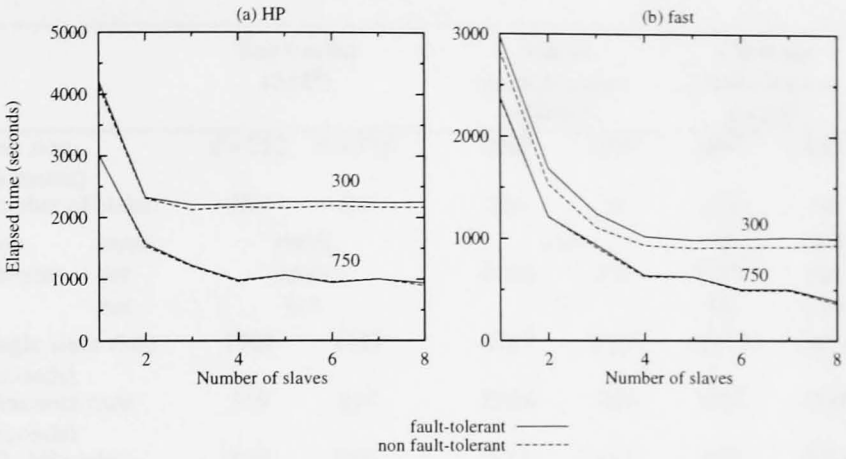



Figure 4.6: Measured performance of fault-tolerant parallel matrix multiplication of 3000 element square matrices, with block sizes 300 and 750, in the **HP** and **fast** configurations.

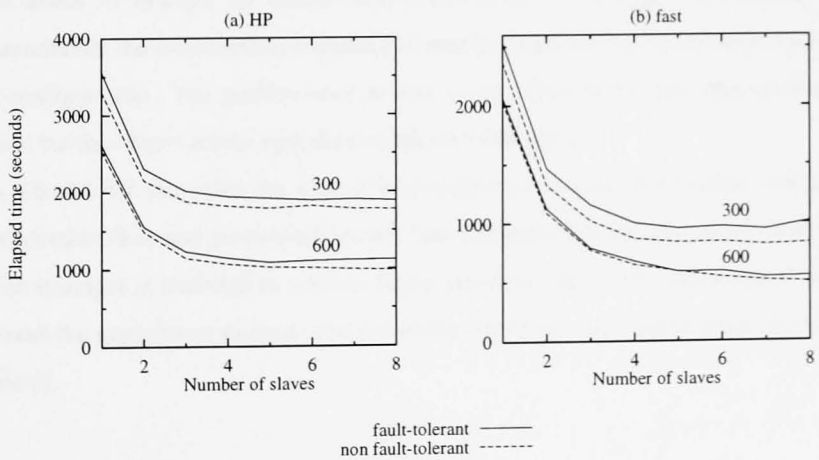



Figure 4.7: Measured performance of fault-tolerant parallel Cholesky factorisation of a 4800 element square matrix, with block sizes 300 and 600, in the **HP** and **fast** configurations.

Table 4.2: Measured performance of fault-tolerant parallel applications in **HP** configuration. Element sizes are 24 bytes for ray tracing and 8 bytes for the matrix computations.

| | Ray tracing ($512^2$) | | Matrix multiplication ($3000^2$) | | Cholesky factorisation ($4800^2$) | |
|---|---|---|---|---|---|---|
| Task size (elements) | $2\times512$ | $8\times512$ | $300^2$ | $750^2$ | $300^2$ | $600^2$ |
| Number of tasks | 256 | 64 | 100 | 16 | 136 | 36 |
| Data (Mbyte) *input* | small | | 144 | | 98 | 104 |
| *get* | small | | 1440 | 576 | 1077 | 588 |
| *put* | 6.3 | | 72 | | 98 | 104 |
| Single slave time (seconds) | 1502 | 1237 | 4214 | 3015 | 3559 | 2613 |
| Minimum time (seconds) | 519 | 213 | 2248 | 937 | 1887 | 1106 |
| I/O (Mbyte/s) | 0.01 | 0.03 | 0.67 | 0.69 | 0.62 | 0.63 |
| Average recovery (seconds) | 3.0 | 9.9 | 22 | 92 | 13 | 36 |
| Performance | absolute speedup | | maximum execution rate (Mflop/s) | | | |
| | 2.2 | 5.3 | 24 | 58 | 20 | 33 |

ising.

In this environment the absolute performance of the matrix computations is not exciting, though the out of core implementations do exceed the peak performance of the lower level in core operations measured at about 33 Mflop/s for matrix multiplication and 25 Mflop/s for Cholesky factorisation. Table 4.3 summarises the measured performance of parallel implementations of the matrix computations in the **fast** configuration. The performance is seen to be rather better than that observed in the **HP** configuration, but the observations regarding block size still apply.

Figures 4.6 and 4.7 show that the cost of fault-tolerance is quite low in both configurations. For this scale of computation and particularly in the **fast** configuration the overall runtime is quite small. However, the structure is intended to scale to larger problems where the total runtime would be much greater. Overall the experiment suggests that for sufficiently large granularity the cost of fault-tolerance should be small.

## 4.8 Validation

This section describes how the performance model introduced in chapter 3 is calibrated and validated through measurements in a number of modest experimental configurations. This is intended to support the use of the model in extrapolating to various alternative configurations in the next chapter. In the case of the matrix computations a small number of low level functions are readily benchmarked to allow

Table 4.3: Measured performance of fault-tolerant parallel applications in **fast** configuration.

| Application | | Matrix multiplication ($3000^2$) | | Cholesky factorisation ($4800^2$) | |
|---|---|---|---|---|---|
| Task size (elements) | | $300^2$ | $750^2$ | $300^2$ | $600^2$ |
| Number of tasks | | 100 | 16 | 136 | 36 |
| Data (Mbyte) | *input* | 144 | | 98 | 104 |
| | *get* | 1440 | 576 | 1077 | 588 |
| | *put* | 72 | | 98 | 104 |
| Single slave time (seconds) | | 2984 | 2390 | 2539 | 2071 |
| Minimum time (seconds) | | 986 | 386 | 925 | 543 |
| I/O (Mbyte/s) | | 1.5 | 1.7 | 1.3 | 1.3 |
| Average recovery (seconds) | | 15 | 75 | 9 | 29 |
| Performance (Mflop/s) | | 55 | 140 | 40 | 68 |

use of the specific models developed for these computations in chapter 3. However, the ray tracing application is largely existing code used in "black box" fashion. Furthermore the amount of computation involved depends on the particular scene, varying within the scene depending on the number of objects in that region. It is still possible to make some use of the basic model by first estimating average values for variable parameters.

### 4.8.1 Benchmarks

Before actual values can be derived for the performance estimates, it is necessary to measure the various primitive operation times, *tadd*, *tsub*, etc. This section describes these primitive operations and the benchmark measurements. The operations themselves are intended to be reasonable given the systems used, rather than necessarily optimal.

#### Computation

The primitive computations, i.e. multiplication, addition etc. are implemented as members of a C++ template class, **Matrix**, instantiated for **double**. Instances of this class are blocks of the matrices which are operands of the whole computation. The member functions[1] described are themselves block structured, the lower level blocks being referred to as sub-blocks, so as to increase locality and thereby gain from processor caching. The *operator*=()* function computes each block row of the product in a tem-

[1] The implementation of the member functions is based on a source for a matrix multiplication function publicly available via *ftp* in Netlib, due to T. Maeno, Tokyo Institute of Technology

porary space before overwriting the source matrix. In this way, it is only necessary to hold a little over three blocks in memory in order to compute a block dot product.

Figure 4.8 shows how the matrix multiplication is tuned to the HP710 machine. It is seen that there

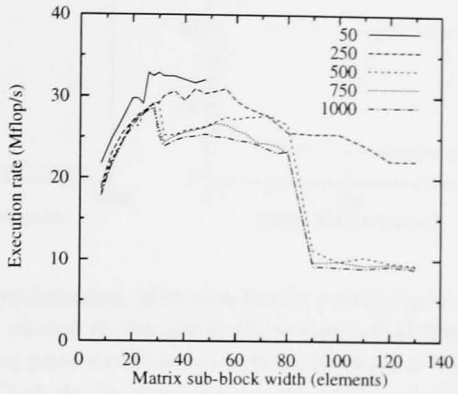



Figure 4.8: Tuning the in core matrix multiplication primitive to the HP710. Separate plots are shown for different matrix sizes. The values plotted in the graph are average of at least 4 measurements. The floating point execution count for multiplication of two square matrices of size $n$ is assumed to be $2n^3$.

is a reasonably consistent peak of about 29 Mflop/s in the performance of matrix multiplication for larger matrix sizes, at a sub-block size of about 28 and consequently, the sub-block size is set to this value, or the matrix width if smaller. The resulting performance is plotted against matrix size in figure 4.9(a) and is seen to be fairly constant apart from a noticeable peak for small data size. Corresponding measurements for the same code on the HP730 show a peak in performance of 43 Mflop/s for the same block size which is also fairly constant for a large range of matrix sizes, and a maximum performance for smaller matrix sizes at least approaching 47 Mflop/s. Also shown in figure 4.9 are the other primitive matrix operations employed, and similar measurements made on a 133 MHz Pentium also with the same code, but employing the slightly smaller block size of 24.

Of the other primitive operations, the Cholesky factorisation and solve operations are tuned in the same way as matrix multiplication. However, no attempt is made to improve performance of the block addition or subtraction operations above that of the simplest loop. There is little to gain from localisation in these operations since there is only one floating point operation to perform per matrix element. The resulting performance of these latter two operations is seen to be quite inferior, at roughly 1.6 Mflop/s on the HP710 and 4.5 Mflop/s on the Pentium. However, the cost of either is of order $n^2$ flop, as compared to $n^3$ for the other operations. Using the rates from the graph, it is seen that the cost of a block addition is under 5% of that for a block multiplication for a block size of 200, and a lower percentage for larger block sizes. In this work, these primitives are used in two large scale computations in both of which the

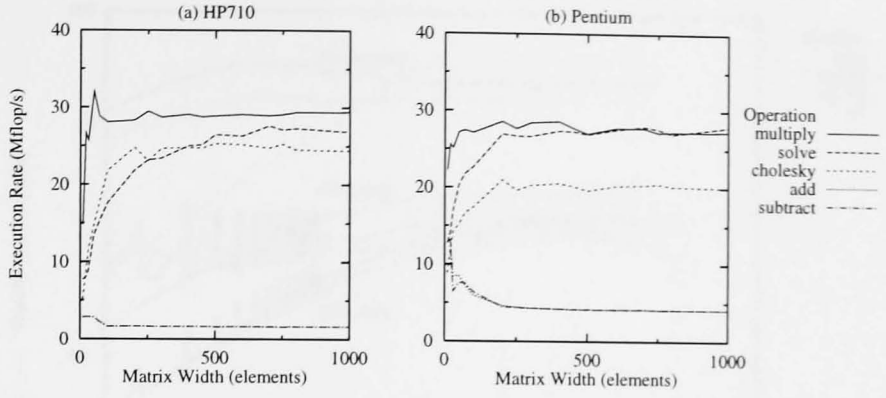



Figure 4.9: Measured performance of in core matrix primitives on the HP710 and Pentium machines. The values plotted in the graph are average of at least 4 measurements. For matrix size $n$ the floating point execution counts assumed are as follows; for multiplication $2n^3$, for solve $n^3$, for Cholesky factorisation $\frac{n^3}{3}$, for addition and subtraction $n^2$.

number of block additions or subtractions is matched by the number of block multiplications.

For comparison, the performance of a HP730 machine running the Linpack benchmark for matrices of width 1000, with the code tuned to the machine, is reported as 49 Mflop/s in [43]. A single precision Linpack measurement of 14.7 Mflop/s is presented in [62] for a 66 MHz Pentium machine with the same cache configuration as in the machines used here. Overall, it is claimed that the primitives as used are of fairly realistic performance, though it is not claimed that no further performance improvement may be gained, either through further levels of blocking or faster algorithm. Ultimately rather than tuning by hand, it would be preferable to employ library primitives, such as from Lapack [5].

**Data Access**

Figure 4.10 compares the performance of local data access operations on HP machines and for the one disk type on Pentium machines.

The operations measured are; read file from file system cache, write new file data to, and read file from locally mounted disk. These basic operations correspond to the object level accesses which are made in the application. A technical specification obtained for the IBM disk quotes a minimum sustained data rate of 3.2 Mbyte/s.

The bandwidth available in a single ATM link greatly exceeds that available to a single disk. However, two of the Pentium machines have a pair of MAXTOR MXT-540-SL disks, so to make better use of the available link bandwidth in larger scale experiments, data is striped over such a pair of disks. The pair of disks are configured as RAID-0, using a software package *md* [121] freely available for Linux. This package implements a pseudo device redirecting I/O requests from the file system layer to

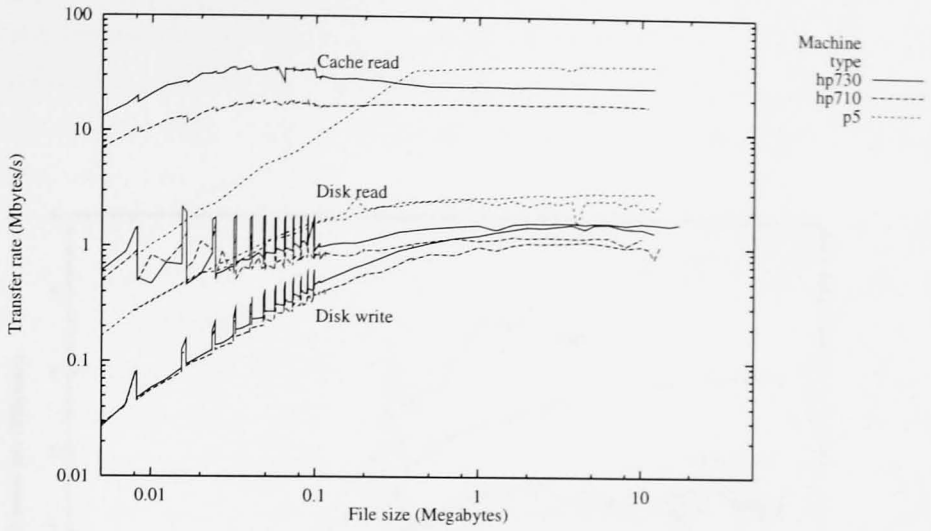



Figure 4.10: Measured performance of data access operations. Both axes are logarithmic. The values plotted in the graph are average of at least 3 measurements. A block of width $b$ corresponds to a file size of $8b^2$ bytes.

multiple individual disk devices. The system supports a linear mode which allows simple concatenation of multiple disks with no redundancy and no performance gain and also RAID levels 0 and 1. Figure 4.11 shows benchmark measurements of various combinations of MAXTOR disks and compares these with the IBM disk.

The MAXTOR disk exhibits different behaviour from either the HP or IBM disks in that the cost of writes is much greater than that of reads. It is speculated that the disk implements a buffered read ahead protocol, though supporting documentation has not yet been located. The maximum read throughput from a single disk is 3.8 Mbyte/s which is rather better than from the IBM disk. A technical specification obtained for the MAXTOR disk quotes a disk transfer rate of between 2.8 and 4.0 Mbyte/s. The performance of writes is somewhat inferior, reaching a maximum of about 2.2 Mbyte/s.

Connecting two disks in linear mode yields effectively a single disk which has the same performance but twice the capacity. The theoretical bandwidth expected over a fast SCSI II link is 10 Mbyte/s. So ideally the throughput to and from a RAID-0 pair should be double that for a single disk. In practice the RAID configuration gives no performance improvement for write operations but improves the maximum read performance to about 5.5 Mbyte/s.

### Communication

Figure 4.12 shows how the transfer rate varies with data size over the three network configurations and for the two request types put and get.

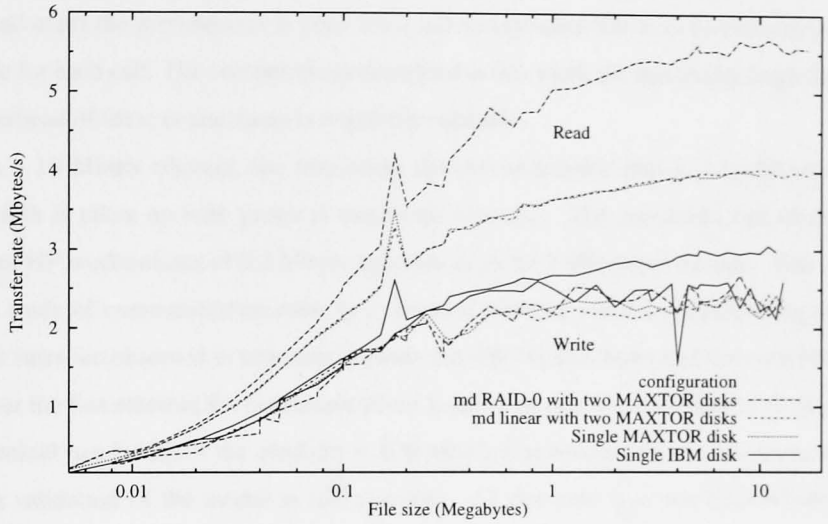



Figure 4.11: Measured performance of the software RAID system using various disk configurations.

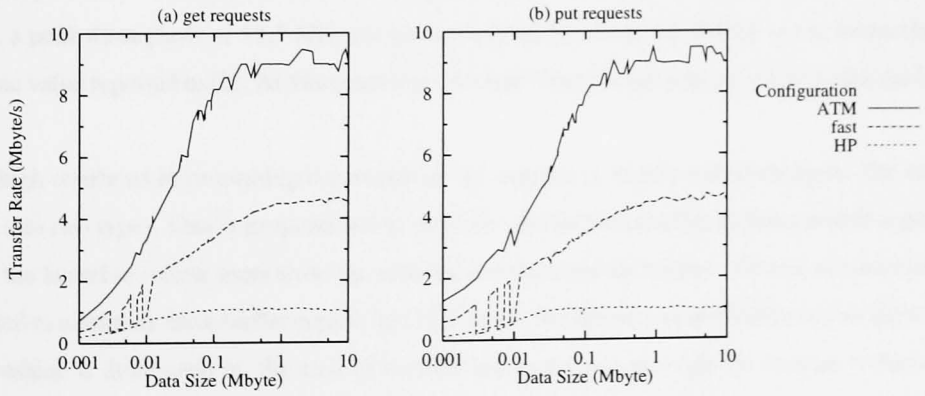



Figure 4.12: measured performance of communications (a) get and (b) put transfers for the three experimental configurations employed.

The operation timed is equivalent to a single RPC call to the shared object store server. The client first establishes a connection to the server, then sends a buffer to the server which returns a separate buffer and finally disconnects. The operation is similar to a single exchange from client to a serial TCP server as described in [104]. In the case of *put* the former is of variable length while the latter is fixed. The reverse is true in the case of *get*.

In all cases the performance is poor for small transfers but this is to be expected since a connection is made for each call. The computations described in this work are marked by large data transfers where the overhead of these connections is hopefully tolerable.

On a 10 Mbit/s ethernet, the maximum theoretical transfer rate is 1.25 Mbyte/s. A part of this bandwidth is taken up with protocol overheads however. The maximum rate observed for transfers between HP workstations of 0.2 Mbyte upwards is about 1 Mbyte per second. This is not inconsistent with a study of communication rates in a range of network parallel programming environments [46]. Similar rates are observed in transfers between two HP710 machines and between HP710 and HP730.

Over the fast ethernet the communications bandwidth reaches a maximum of only 4.6 Mbyte/s. As the nominal bandwidth of the medium is 100 Mbit/s this measurement gives some cause for concern, though validation of the model is still possible. At this time it is not clear where the performance degradation occurs, but a simple echo test using UDP over the same fast ethernet connection achieves also a maximum of 4.6 Mbyte/s.

The performance for transfers between a single client server pair over the ATM network gives only partial indication of the performance that is attainable. The link and switch bandwidths are nominally 155 Mbit/s and 2.5 Gbit/s. Transfers between a single client server pair are therefore bounded by the former. For raw ATM, the maximum throughput is 135.6 Mbit/s [2]. The measured bandwidth over TCP is seen to peak at 9 Mbyte/s or 72 Mbit/s. Using an earlier installation of Linux based on kernel version 1.3.71, a peak throughput of 12.5 Mbyte/s was measured, which at 100 Mbit/s is not inconsistent with the basic value reported in [2]. At this point it is not clear what change may have lead to the performance loss.

A high overhead in processing communications requests is widely acknowledged. The overheads divide into two types. One is proportional to data size and includes buffer copying which is performed within the kernel to isolate users from the network interface and each other. Various schemes have been proposed to eliminate these buffer copies, eg [2]. Even in the absence of any buffer copies there remains a cost which is dominated by the cost of transferring a message through the various protocol layers. This latency can be observed as the time taken to transfer even a small message between application and network. This issue has attracted some attention recently particularly as the idea of employing "off the shelf" workstation clusters, perhaps with retargeted multiprocessor networks such as Myrinet [23], for general parallel computing tasks has lead to comparisons between the communications infrastructure in the two environments. A portable user level communications interface, eg U-Net [114], addresses the

high latency through reducing kernel involvement in message transfer. Such a system is not currently available for the hardware used here[2].

### 4.8.2 Ray Tracing

The cost of the total computation component is roughly equal to that of the sequential implementation, where the disk output is done in parallel with computation. The time for the unmodified software to complete the specific example computation is measured at 1130 seconds. The bulk of the communication is that entailed in writing the completed rows. It is possible therefore to estimate the maximum parallelism as the average task computation divided by the cost of writing the output to shared store. Each row of the output matrix contains 512 **color** entries, each containing 3 **double** values. The size of a **double** here is 8 bytes, and so each row occupies 12 Kbyte. The cost of writing to disk is shown in figure 4.10 and the cost of writing a few rows of the output, say up to 8, is within the leftmost region of the graph which exhibits significant variation with data size. Figure 4.13 shows in greater detail the

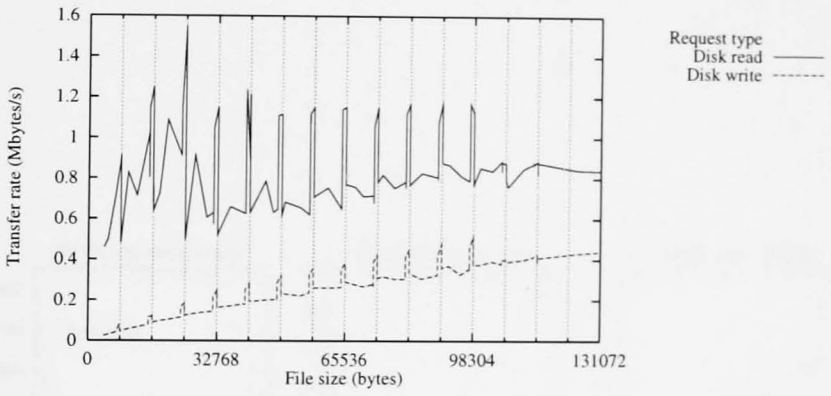



Figure 4.13: Performance of small data transfers to and from HP710 internal disk.

cost of small disk accesses to the HP710 internal disk. It is seen that there is a peak in transfer rate at each multiple of 8192 bytes, the physical block size. While the data contained in 2 and 8 rows is the same size as 3 and 12 disk blocks respectively, the actual data written is slightly larger. This is because each row is stored in a general array structure and the current size of the array is stored with the data itself. The effective transfer rate is thus the value corresponding to the dip just after the appropriate peak. Table 4.4 shows derivation of the expected performance for a range of task sizes.

Figure 4.14 shows the measured performance of the application for grain size of 1, 2 and 8 both as

[2]In particular U-Net version 2.0 does not support HP-UX and supports Linux only with a Fore Systems ATM adapter or DECChip 21140 "Tulip" fast ethernet adapter

Table 4.4: Derivation of maximum performance for ray tracing.

| Task size (elements) | $Tcomp$ (seconds) | Disk Write (Mbyte/s) | $Tput$ | $T_1$ (seconds) | lwr $\{T_\infty\}$ | upr $\{\bar{S}_\infty\}$ |
|---|---|---|---|---|---|---|
| $1 \times 512$ | 2.21 | 0.0658 | 0.187 | 1226 | 95.5 | 13 |
| $2 \times 512$ | 4.41 | 0.125 | 0.197 | 1180 | 50.4 | 23 |
| $4 \times 512$ | 8.83 | 0.226 | 0.217 | 1158 | 27.8 | 42 |
| $8 \times 512$ | 17.7 | 0.375 | 0.262 | 1147 | 16.8 | 68 |
| $16 \times 512$ | 35.3 | 0.555 | 0.354 | 1141 | 11.3 | 101 |
| $32 \times 512$ | 70.6 | 0.774 | 0.508 | 1138 | 8.1 | 140 |

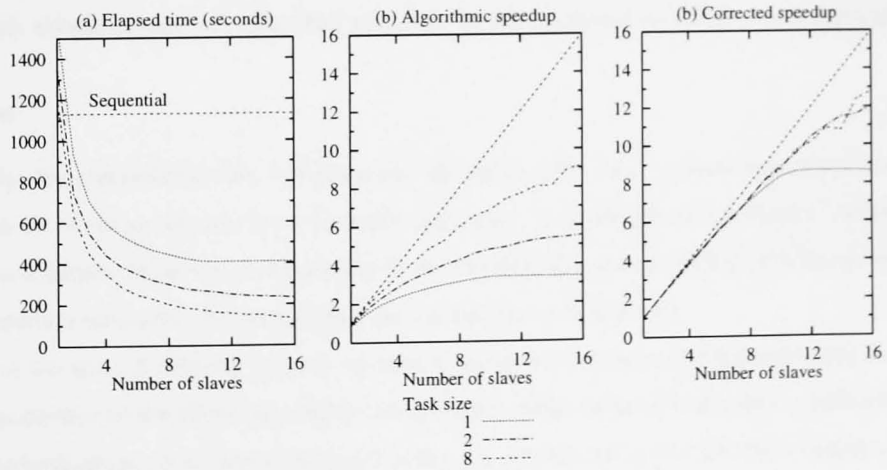



Figure 4.14: Measured speedup of parallel ray trace of example image "balls" at size $512 \times 512$ pixels.

elapsed time and algorithmic speedup. The algorithmic speedup is seen to level off as expected, but at a much lower value than predicted. Similarly the single slave time is somewhat higher than the expected value. However, as remarked before with regard to the matrix computations, the cost of constructing the empty output object on disk is not accounted for in the model. Again, the cost of constructing this object is proportional to the total number of tasks, and not directly to the size of the data. In the three instances described here where the task size is 1, 2 or 8 rows, the cost of constructing the output object is approximately 160, 80 or 20, i.e. about 0.3 seconds per task. Furthermore, there is an extra cost which is not accounted for in the model, namely the conversion of output data to Utah raster format. For the three task sizes referred to, this cost is roughly 44, 22, 5 seconds, i.e. about 0.09 seconds per task. Figure 4.14(c) shows the algorithmic speedup again, but after the initialisation and data reformatting cost have been deducted from the measured time.

### 4.8.3 Matrix Computations

As for ray tracing, it is possible to make a precise comparison for each separate experimental configuration between measured and predicted performance. Though such an exercise verifies the model, it doesn't facilitate extrapolation to alternative configurations or to larger or alternative problems. For this purpose it is more useful if a simple cost function can be defined for each of the primitive operations.

In general it is not possible to guarantee that upgrading disk say will lead to the relevant performance curve being altered by a simple multiplicative factor, but the matrix computations studied earlier admit particularly simple expressions. The discussion that follows is in terms of these two computations.

#### Processor

Previously, the computation rate was measured for each of the basic matrix operations. Given such a detailed set of measurements for a particular processor, it is possible to use them to attain greatest prediction accuracy. However, a comparison between different machines is less obvious as the number of parameters is increased. For such a purpose, a compromise is desirable

One of the aims of deriving a block oriented expression of a matrix computation is to concentrate a large proportion of the effort into highly tuned routines such as the BLAS, and in particular matrix-matrix multiplication. It is then tempting to make the simplification that all block matrix operations run at the rate of matrix multiplication. In such computations the HP710 and Pentium are then of nearly equal performance. Referring back to the measured performance, figure 4.9, it is seen that the computation rate for the tuned matrix primitives are generally not very different. The addition and subtraction operations do have much worse performance, but as noted earlier the operation count in these cases is proportional to the square rather than cube of the block width. For these computations then the parameter describing machine speed is taken as the execution rate for matrix multiplication.

and as observed in practice this rate is assumed to be sustained over most of the possible range of block sizes.

### Disk

For large transfers the disk transfer rate is roughly constant. While transfer rates for read and write transfers are nearly equal in the case of the HP and IBM disks, they differ considerably for the MAXTOR disk. However, in the case of the HP disks the performance of small read requests is rather better than that of small write requests. The cost of a cache read is much smaller than the cost of a disk read and so is conveniently ignored.

### Communication

As in the case of disk large transfers are seen to be at roughly constant rate.

### Model Parameters

For large block sizes, the three bandwidths are all constant, but a convenient approximation for modelling behaviour when the bandwidth is small is to employ piecewise linear models, so that each parameter is represented by a bandwidth and a threshold. Below the threshold the bandwidth is assumed to be proportional to the size of the data for the transfer costs and to the number of basic operations in the case of the processor cost. The values corresponding to the three practical configurations are summarised in table 4.5. In each case, the bandwidth and threshold are given.

Table 4.5: Measured parameters for the various hardware configurations to the performance model used for the matrix computations.

| Configuration | | HP | fast | ATM |
|---|---|---|---|---|
| Computation (Mflop/s:ops) | $P$ | $30{:}30^3$ | $27{:}30^3$ | |
| Network (Mbyte/s:Mbyte) | $C$ | 1:0.05 | 4.6:0.25 | 9.0:0.11 |
| Disk (Mbyte/s:Mbyte) | $Dget$ | 1.6:0 | 2.8:0.11 | 5.5:0.3 |
| | $Dput$ | 1.6:0.3 | 2.8:0.11 | 2.4:0.3 |

### Computation

$$tmult = \frac{2b^3}{P} \ , \ tsolve = \frac{b^3}{P} \ , \ tchol = \frac{b^3}{3P} \ , \tag{4.1}$$

$$tadd = tsub = \frac{b^2}{P} \ . \tag{4.2}$$

### Data Access

$$tget = 8b^2 \left( \frac{1}{Dget} + \frac{1}{C} \right) \ , \ tput = 8b^2 \left( \frac{1}{Dput} + \frac{1}{C} \right) \tag{4.3}$$

Figure 4.15 shows how the estimates of single slave and minimum parallel execution time provide bounds on limiting and nonlimiting performance.

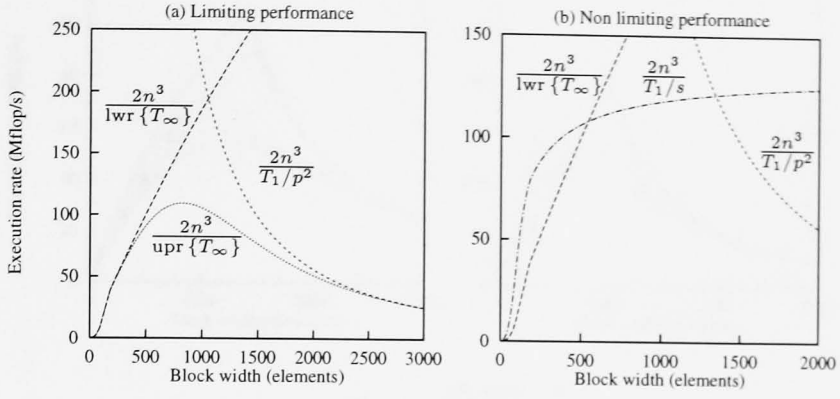



Figure 4.15: Computing bounds on parallel performance of $3000^2$ matrix multiplication. In the nonlimiting example $s$ is the number of slaves. $\frac{T_1}{p^2}$ is the granularity bound, i.e. $\max\{Tcomp(i)\}$, for matrix multiplication.

Figures 4.16 and 4.17 compare expected performance of each of the two matrix computations with measured results. Performance of matrix multiplication is computed using (3.15) to (3.17) and Cholesky factorisation using (3.26) to (3.28). In each case the comparison is made in two alternative hardware configurations.

As well as supporting the validity of the analysis, the graphs show that the greatest uncertainty of prediction occurs where the block size is less than but not much less than the matrix size. Both when the block size is very much smaller than the overall matrix size and in the trivial case when these two sizes are equal, the upper and lower bounds are close together. Obviously in the latter case, there is no parallelism and hence no uncertainty. In the former case, the parallelism is greatest and is therefore only restricted by the number of processors. In realistic large scale computations it is likely that the block size would be significantly smaller than the overall matrix size.

A small number of measurements have been made using the **ATM** configuration for larger scale computations. Table 4.6 summarises the performance of the fault-tolerant parallel implementations, showing for each application a measure of the performance achieved and estimate of the average recovery time. As in the case of earlier results, the table also indicates the total data accessed by slaves. In this configuration it was only possible to make 5 slaves available for the experiment, and clearly bandwidth limits are not reached. The lower bound for the nonlimiting case is used as predicted time and shown in parentheses. The efficiency quoted is simply the ratio of measured time for the fault-tolerant implementation to the expected time.

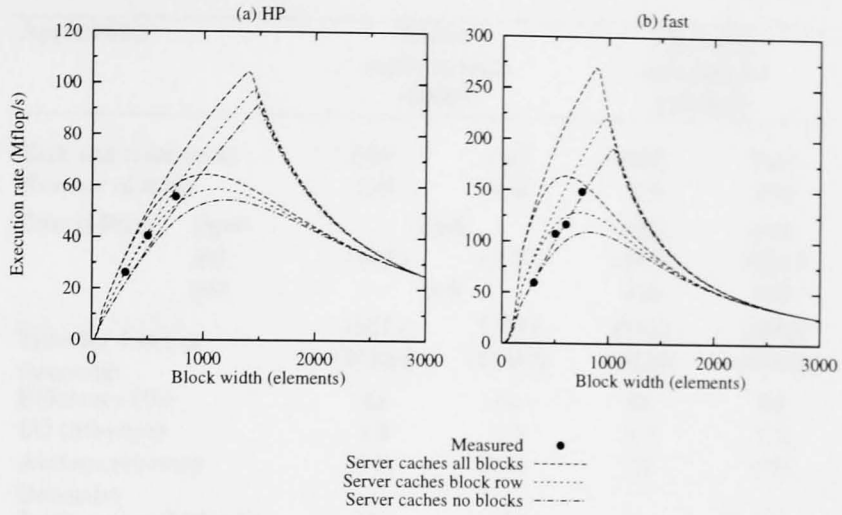



Figure 4.16: Potential limiting performance of $3000^2$ matrix multiplication for varying block size with some measured values.

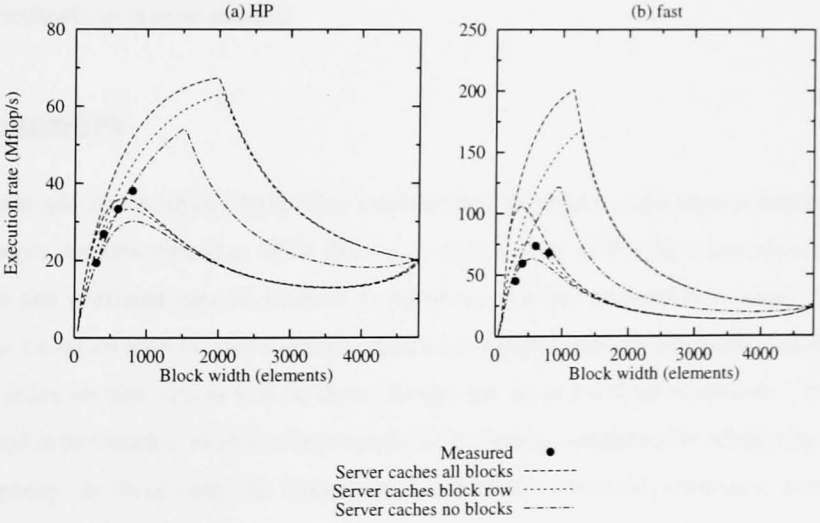



Figure 4.17: Potential limiting performance of $4800^2$ Cholesky factorisation for varying block size with some measured values.

Table 4.6: Measured performance of fault-tolerant parallel applications in ATM configuration.

| Application | | Matrix multiplication ($9000^2$) | | Cholesky factorisation ($15000^2$) | |
|---|---|---|---|---|---|
| Task size (elements) | | $600^2$ | $750^2$ | $600^2$ | $750^2$ |
| Number of tasks | | 225 | 144 | 325 | 210 |
| Data (Mbyte) | *input* | 1296 | | 936 | 945 |
| | *get* | 19440 | 15552 | 15912 | 12915 |
| | *put* | 648 | | 936 | 945 |
| Time for 5 slaves (seconds) | | 13179 (11620) | 12611 (11462) | 11521 (9853) | 10922 (9792) |
| Efficiency (%) | | 88 | 91 | 86 | 90 |
| I/O (Mbyte/s) | | 1.5 | 1.3 | 1.5 | 1.3 |
| Average recovery (seconds) | | 146 | 219 | 89 | 130 |
| Performance (Mflop/s) | | 111 | 116 | 98 | 103 |

In this experiment the computations are of about 4 hours duration, so fault-tolerance is certainly desirable. The efficiency, based on measured performance of a fault-tolerant computation and computed estimate of the single slave time in the absence of fault-tolerance demonstrates that the application of fault-tolerance to this computation structure is successful. Clearly too however, the distribution of computation data is successful, and it is in the consistent update of distributed persistent data that transaction technology is most needed.

## 4.9 Summary

The three example computations have been implemented in various experimental networks. The ray tracing example demonstrates that while there is a cost inherent in this fault-tolerance mechanism as in any other this overhead may be reduced in significance if the granularity is large. Clearly if the problem size increases then in such a computation even a fault-tolerance mechanism such as that used here which relies on disk access will be cheap though the benefit will be significant. The structuring approach used in this work is of more direct benefit to the matrix computations which may exceed main memory capacity. In these cases the failure free cost of fault-tolerance provision is seen to be small provided the granularity is large.

The performance models are calibrated through benchmark measurements and used to show that the measured performance of each of the three computations is close to that predicted. In absolute terms it is seen that significant benefit through parallelism can be gained for the matrix computations in the higher specification network.

# Chapter 5

# Assessment

Previous chapters have described a technique for distributing a parallel application over a collection of workstations using a shared persistent store and using atomic actions to implement various levels of fault-tolerance. Implementation of three large grain examples has shown that the cost of fault-tolerance can be small and yet still allow significant benefit, particularly as the scale of computation grows. This chapter employs the simple modelling techniques described in chapter 3 in an attempt to extend the study beyond the limits of experiment to make some assessment of the potential for the structuring approach. First a comparison between alternate computation structuring approaches is performed in the context of Cholesky factorisation. Following is an analysis of the effect of scaling problem size. Much of the subsequent description is in the context of the simpler application, matrix multiplication, and begins by studying the potential benefit of data caching and continues by addressing issues of scaling hardware configuration. The potential application of multithreading techniques to overlap communications is addressed next, and subsequently a comparison with a data parallel structure. Finally some consideration is given to the potential for further application of the approach.

## 5.1 Computation Structures

It is seen earlier that the dependencies inherent in Cholesky factorisation may be satisfied in various different ways. This issue inevitably arises in a nontrivial application and is clearly specific to each application. While it is possible to define a set of synchronisation mechanisms which are sufficient in general, it is important to understand the impact on performance that a choice of synchronisation mechanism may have. In the case of the example algorithm for Cholesky factorisation which is suited to an out of core computation, three implementations are derived which differ only in the mechanism employed to achieve correct synchronisation. These can be employed both to study the implementations of the alternative structuring techniques and also as an example of the issues arising in selecting a

synchronisation mechanism for a particular application. A brief record of this work appears in [100].

First the bounding performance is compared, specifically that of the **multi-step(1)** and **single-bag** organisations. Subtracting (A.10) from (A.14) and ignoring the second major term of (A.14) the difference is

$$\begin{aligned} &\mathrm{lwr}\,\{Tmultib1_\infty\} - \mathrm{upr}\,\{Tsingleb_\infty\} \\ &\geq \frac{p-1}{2}((p-2)(tmult + tsub) - 2tsolve) \ . \end{aligned} \tag{5.1}$$

This quantity is positive if $p > 2$ and

$$tmult + tsub > 2tsolve \ .$$

In the model (4.1) $tmult = 2tsolve$ and this is found in measurement too, figure 4.9. So for practical factorisation problems, an upper bound on minimum parallel time in the **single-bag** organisation is exceeded by a lower bound on minimum parallel time in the **multi-step(1)** organisation. This algebraic comparison is valid for all hardware configurations.

Figure 5.1 compares the maximum performance for the three computation structures for a matrix size of $4800^2$ in the **fast** configuration. Both upper and lower bounds on maximum performance are shown and so are a number of measurements actually made in this configuration. The computation rate assumes an overall operation count of $\frac{n^3}{3}$. The performance is greatest for the **single-bag** organisation, and the performance of the **multi-step(2)** organisation is not much better than that of the **multi-step(1)** organisation.

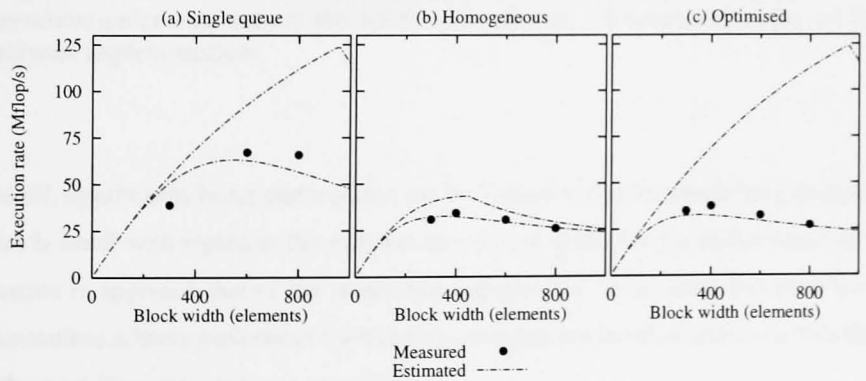



Figure 5.1: Potential performance of parallel Cholesky factorisation using three different synchronisation structures in the **fast** configuration. All measured values are for fault-tolerant implementations.

A study of larger scale computations is based on the **ATM** configuration. Figure. 5.2 shows how the performance varies for increasing matrix size and separately for increasing number of slaves. The graph plotted for each structure derives from the lower bound on the predicted parallel time (3.21), (3.24), (3.25). A number of measurements made in the **ATM** configuration are also plotted.

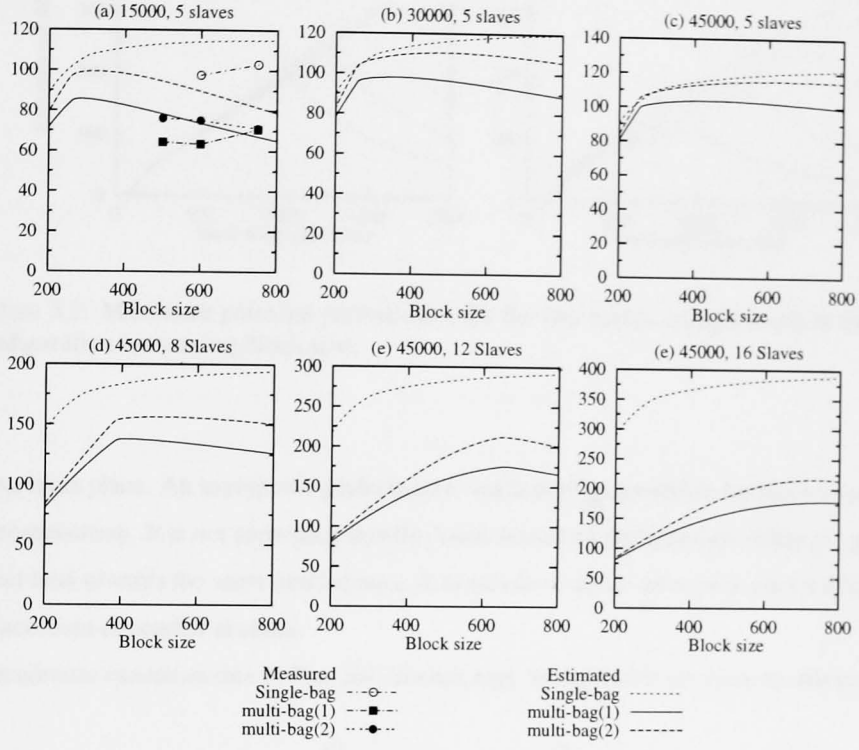



Figure 5.2: Potential performance of parallel Cholesky factorisation using three different synchronisation structures in the **ATM** configuration. All measured values are for fault-tolerant implementations.

Overall, significantly better performance can be obtained using the **single-bag**, though if the number of slaves is small with regard to the problem size it is possible for the performance of **multi-step(2)** organisation to approach that of the **single-bag** organisation. It is noted that even the fault-tolerant implementations achieve performance which is close to that predicted, so that even with the assumptions made, the modelling process is not unrealistic.

## 5.2 Computation Scaling

It is seen already that the limiting performance for the matrix computations increases as the operand size increases. This peak performance is related to the inverse of the total I/O cost. The relationship is

shown more clearly in figure 5.3. Only the upper bound on performance is plotted and it is assumed that

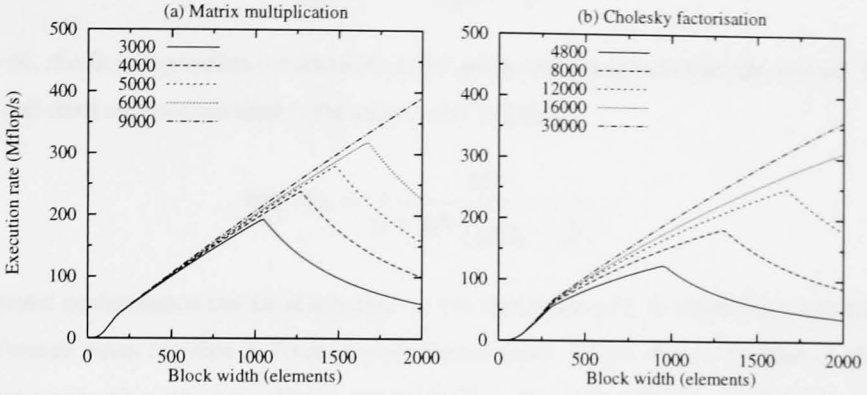



Figure 5.3: Maximum potential performance for the two matrix computations in the **fast** configuration for varying block size.

no caching takes place. An asymptotic performance which is proportional to the block size is apparent for each computation. It is not surprising that the lower bound on performance, while not plotted here, does in fact tend towards the same performance. It is simple to derive an expression for this asymptotic performance from the earlier analysis.

The maximum execution rate is $R_\infty$, and bounds may be derived from those on minimum parallel time.

$$\frac{O}{\mathrm{upr}\{T_\infty\}} \leq R_\infty \leq \frac{O}{\mathrm{lwr}\{T_\infty\}}$$

where $O$ is the overall operation count.

For large $n$, the operation count for matrix multiplication is taken as $2n^3$, i.e. $2b^3p^3$. For large block size, the model parameters $C$, $Dget$, $Dput$, $P$ are constant. The maximum performance for matrix multiplication is derived from the expressions for minimum parallel time, (3.13), (3.14).

$$\begin{aligned}
R_\infty &\leq \frac{2p^3b^3}{2p\frac{8b^2}{Bget} + \frac{8b^2}{Bput} + p(2b^3 + b^2)\frac{1}{P}} \\
R_\infty &\leq \frac{2p^3b^3}{2p^3\frac{8b^2}{Bget} + p^2\frac{8b^2}{Bput}} \\
R_\infty &\geq \frac{2b^3p^3}{2p^3\frac{8b^2}{Bget} + p^2\frac{8b^2}{Bput} + (2b^3 + b^2)\frac{1}{P}}
\end{aligned}$$

where:

$$\frac{1}{Bput} = \frac{1}{Dput} + \frac{1}{C}$$

and

$$\frac{1}{Bget} = \frac{1}{Dget} + \frac{1}{C} .$$

If $p \to \infty$, the first expression continues to grow and is always greater than the second. However, the second and third expressions tend to the same value, which is

$$\lim_{p\to\infty} R_\infty = \frac{2b^3}{2 \times 8b^2 \left(\frac{1}{Dget} + \frac{1}{C}\right)} .$$

This asymptotic performance can be interpreted as the maximum rate at which matrix multiplication can be performed when the data is fetched from shared store. In the case of the **fast** configuration, the asymptotic execution rate for a block size of $750^2$ is then 163 Mflop/s. In the case of the **HP** configuration, the corresponding figure is 58 Mflop/s. In the case of the **ATM** configuration, the disk read and write costs are different, but ultimately only the read cost is relevant. Since the object store is distributed over two machines, the limiting bandwidth is twice that for a single node, i.e. 11 Mbyte/s for disk and 18 Mbyte/s network. The asymptotic performance for block size 750 is then 641 Mflop/s.

For Cholesky factorisation, the overall operation count is taken as $\frac{n^3}{3}$, i.e. $\frac{b^3p^3}{3}$, for large $n$. Expressions for limiting execution rate are defined for the single queue configuration from those for minimum parallel time, (3.19), (3.20).

$$\begin{aligned}
R_\infty &\leq \frac{p^3b^3}{3(2p\frac{8b^2}{Bget} + \frac{8b^2}{Bput} + ((p-1)(2b^3+b^2)+b^3)\frac{1}{P})} \\
R_\infty &\leq \frac{p^3b^3}{3(p\frac{8b^2}{Bget} + \frac{8b^2}{Bput} + ((p-1)(2b^3+b^2)+\frac{b^3}{3})\frac{1}{P})} \\
R_\infty &\leq \frac{p^3b^3}{3(\frac{p}{6}(p+1)(2p+1)\frac{8b^2}{Bget} + \frac{p}{2}(p+1)\frac{8b^2}{Bput})} \\
R_\infty &\leq \frac{p^3b^3}{3(\frac{p}{6}(p+1)(2p+1)\frac{8b^2}{Bget} + \frac{p}{2}(p+1)\frac{8b^2}{Bput} + ((p-1)(3b^3+b^2)+\frac{pb^3}{3})\frac{1}{P})}
\end{aligned}$$

As $p \to \infty$ the first and second expressions grow, but the remaining two approach the same asymptotic performance as seen for matrix multiplication.

It is possible to repeat the exercise for a computation where blocks are cached either at slave or server. As an example the case of matrix multiplication where a block row is cached at server level is considered. From (3.16) and (3.17) bounds on maximum execution rate can be defined as before.

$$\begin{aligned}
R_\infty &\leq \frac{2p^3b^3}{(p-1)\frac{8b^2}{C} + (p+1)\frac{8b^2}{Bget} + \frac{8b^2}{Bput} + p(2b^3+b^2)\frac{1}{P}} \\
R_\infty &\leq \frac{2p^3b^3}{p^2(p-1)\frac{8b^2}{C} + p^2(p+1)\frac{8b^2}{Bget} + p^2\frac{8b^2}{Bput}}
\end{aligned}$$

$$R_\infty \geq \frac{2p^3b^3}{2p^2(p-1)\frac{8b^2}{C} + p^2(p+1)\frac{8b^2}{Bget} + p^2\frac{8b^2}{Bput} + (2b^3+b^2)\frac{1}{P}}$$

where, as before:

$$\frac{1}{Bput} = \frac{1}{Dput} + \frac{1}{C}$$

and

$$\frac{1}{Bget} = \frac{1}{Dget} + \frac{1}{C} .$$

In this case as $p \to \infty$ the latter two equations both tend to a different limit

$$\lim_{p\to\infty} R_\infty = \frac{2b^3}{8b^2\left(\frac{1}{Dget} + \frac{2}{C}\right)} .$$

This can be interpreted as the maximum rate at which matrix multiplication can be performed when one of the matrix operands must be fetched from remote disk and the other from remote memory.

Clearly the performance achievable in the example computations is limited by the cost of fetching data from shared store. Clearly also there is potential to reach a higher performance through caching data in server or slave memory and the following section considers this issue more fully.

## 5.3 Benefit from Caching

Computation performance is seen already to be dependent on block size and caching pattern. In fact these two parameters are not independent. For a given block size it is no surprise that the performance should never be worse where some caching occurs than where no caching occurs. However, such a comparison assumes that memory space is unbounded. A more realistic comparison fixes the memory utilisation and determines the block size from that value.

For matrix multiplication, a comparison can be made directly between a computation where no caching takes place and an alternative where a row of blocks is cached by each slave. It is assumed that all machines have the same amount of memory. The memory required to support the programs themselves is assumed constant and therefore ignored. Similarly memory which should be required for buffering in the communications path is also ignored. The block size is assumed to be the largest which the particular memory size can support. If the main memory size for a slave is $m$, the matrix size $n$ and the chosen block size $b$, then in the case where no blocks are cached three blocks must be held in memory. Matrix elements are assumed to be 8 byte doubles so that, from table 3.3

$$m = 3 \times 8b^2 .$$

This is rearranged:

$$b = \sqrt{\frac{m}{24}} \ .$$

In the case where a block row is cached by each slave, a row of blocks and two other blocks must be stored by each slave so that

$$m = 8(nb + 2b^2) \ ,$$
$$b = \frac{\sqrt{n^2 + m} - n}{4} \ .$$

It is then possible to compare the maximum performance achievable in the two configurations. This is shown for the **fast** configuration in figure 5.4. The drop in performance for large block size arises

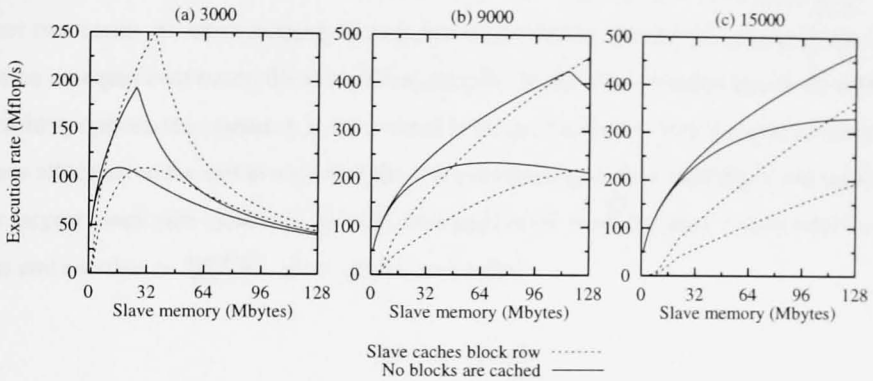



Figure 5.4: Potential performance of matrix multiplication in the **fast** configuration exploiting different patterns of data caching in slave memory.

simply from the reducing parallelism. In practice, the block size would not be increased above the point at which performance is optimal, but this is not important here.

It is assumed that no cost is incurred by a slave in accessing a block which is cached locally. The benefit from caching a block row at slaves is reduced if the number of slaves is greater than the width in blocks of a row because more than one slave will cache the same block row separately. The effect becomes more noticeable gradually as the number of slaves participating increases. This effect is ignored however in the graph so that the performance of the caching option is likely to be an overestimate. Yet still it appears the prospects for caching are not good. The graphs show that for a given problem size and in a given configuration caching a block row cannot give advantage if the memory size is below some threshold. Intuitively if a block row must be accommodated within a given memory size, then the maximum block size must be smaller than if only a single block is required. The volume of I/O performed through the computation is proportional to the block size and this effect tends to counter the

reduction in I/O requirements caused by data reuse.

In this configuration it appears with the 32 Mbyte machines in use, there is potential for benefit through caching a block row at the slaves only for rather small matrices.

Assessing the benefit of caching at the server is not quite so straight forward, since the extra memory for caching is only required at the server. From table 3.3 the extra memory required at the server in order to permit a block row to be cached there is

$$8nb(1+s)$$

where $s$ is the number of slaves participating in the computation and is assumed no greater than $p$. If $s > p$ then to achieve the benefit of block row caching at the server, it is necessary to accommodate as many different block rows as may be computed at once. In general this would be $\left\lceil \frac{s}{p} \right\rceil$. Obviously in the other two cases, no extra memory is required at the server. In order to compare the three caching options on an equal cost basis, the controlling variable is the total memory across all machines. In the slave caching options this memory is all divided between the slaves. In the server caching option some portion is allocated to the server such that the sizes of server and slave memory are in some defined ratio and the largest block size chosen to fit both slave and server memory sizes. Given total memory $M$ and $s$ slaves and $mratio = \frac{mserver}{mslave}$, if no blocks are cached

$$\begin{aligned} M &= 24sb^2 \\ b &= \sqrt{\frac{M}{24s}} , \end{aligned} \tag{5.2}$$

if a block row is cached by each slave

$$\begin{aligned} M &= 8s(nb+2b^2) \\ b &= \frac{\sqrt{s^2n^2+sM}-sn}{4s} , \end{aligned} \tag{5.3}$$

if a block row is cached by the object server

$$\begin{aligned} M &= s\,mslave + mserver \\ mslave &= \frac{M}{s+mratio} \\ &= 24bslave^2 \\ mserver &= \frac{mratio\,M}{s+mratio} \\ &= 8n\,bserver\left(\left\lceil \frac{s}{n/bserver} \right\rceil + s\right) \\ b &= \min\{bslave, bserver\} . \end{aligned}$$

Since

$$\frac{s}{n/bserver} \leq \left\lceil \frac{s}{n/bserver} \right\rceil \leq \frac{s}{n/bserver} + 1 \ ,$$

it is argued that an upper bound on $bserver$ is defined by the expression

$$\begin{aligned} mserver &= \frac{mratio\, M}{s + mratio} \\ &= 8n\, bserver \left( \frac{s}{n/bserver} + s \right) \ , \end{aligned}$$

such that

$$bserver \leq \frac{\sqrt{4n^2s^2 + 2s\, mserver} - 2ns}{4s} \ .$$

Since the performance is proportional to the block size, an upper bound on block size defines an upper bound on performance.

Figure 5.5 compares the maximum expected performance for 5 slaves executing with the three

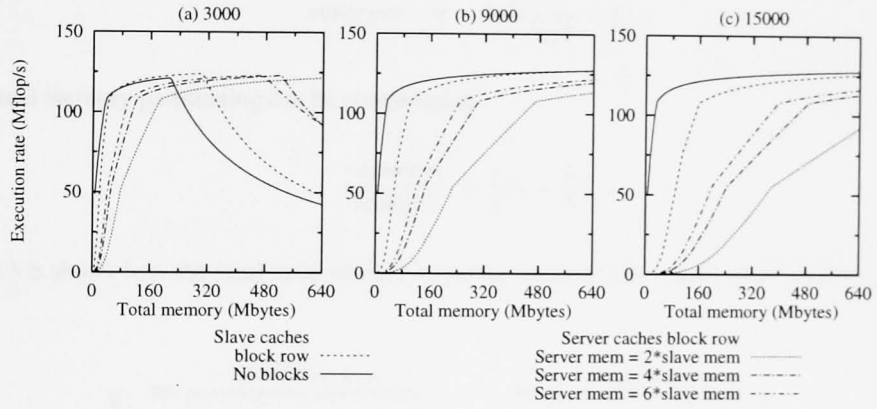



Figure 5.5: Potential performance of matrix multiplication in the **fast** configuration exploiting different patterns of data reuse in slave memory or server level file system cache.

alternate cache patterns. The range of total memory is equivalent to 5 machines with 128 Mbytes. As before memory used by the program code and system, including communications buffers in the slaves is ignored. Also ignored however is temporary buffer memory required in the server machine which is assumed constant for each of the three configurations.

While it seems that avoiding any block caching offers greatest potential, the performance of the server row caching option is quite dependent on the particular memory configuration chosen. Assuming that the overall matrix size is large enough, it is generally best to choose as large a block size as possible. Thus the optimum partitioning of a given total memory allocation is obtained by setting $bslave$ =

$bserver = b$. The maximum block size may then be computed as below

$$M = 8nb\left(\left\lceil\frac{s}{n/b}\right\rceil + s\right) + 24sb^2 \ . \tag{5.4}$$

Again a bound on performance is sought from the modified expression

$$M = 8nb\left(\frac{s}{n/b} + s\right) + 24sb^2 \ ,$$

and hence

$$b = \frac{\sqrt{n^2s^2 + 2sM} - ns}{8s} \ .$$

Since

$$mslave = 24b^2$$

and

$$mserver = 8nb(\frac{s}{n/b} + s) \ ,$$

the actual memory partitioning can be computed as

$$\frac{mserver}{mslave} = \frac{s}{3}(1 + \frac{n}{b}) \ .$$

Figure 5.6 shows how the ideal ratio of slave to server memory size varies with number of slaves for a

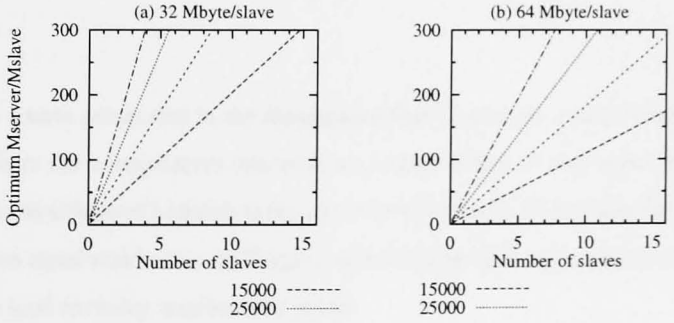



Figure 5.6: Variation of optimum memory partitioning between slave and server with number of slaves for different instances of matrix multiplication.

range of large scale instances of matrix multiplication. Since the optimum memory partitioning depends on the size of the matrix it is clearly difficult to exploit such server level caching in general. Furthermore,

when the performance of the optimum configuration is compared with the other two caching options in figure 5.7, it is clear that from a performance point of view too the best option so far is to avoid block

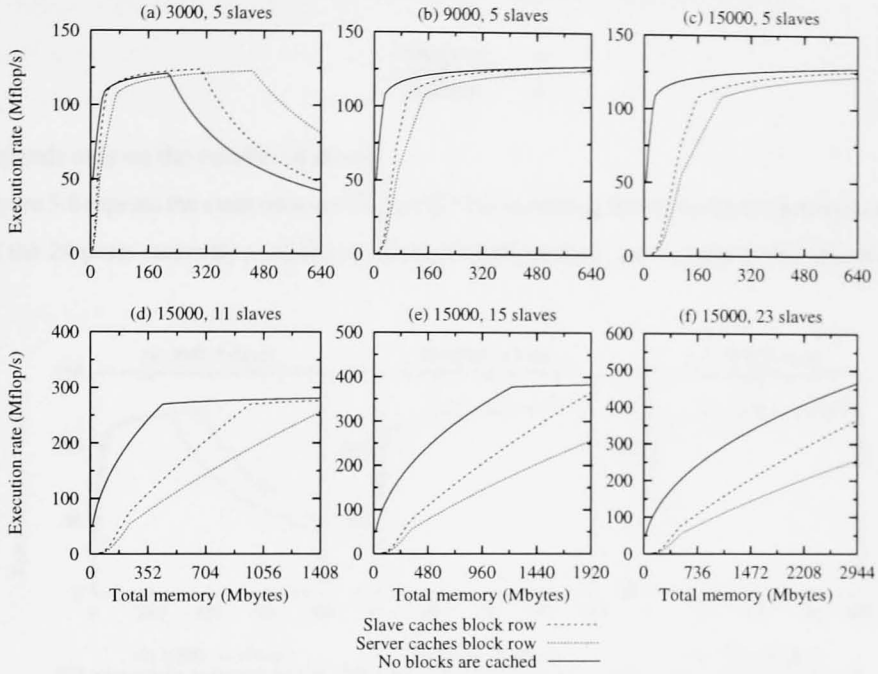



Figure 5.7: Potential performance of matrix multiplication in the **fast** configuration with server level file system caching when the memory is partitioned optimally between server and slaves.

caching.

In (5.4), the $s$ term arises due to the assumption that blocks are cached transparently in file system buffer space. Since the computation operands are implemented as user specified objects, it should be possible to exert some form of control on the server level caching. In this case for instance, it is sufficient for one of the two input matrices to be flagged as permitting caching at object server level, the other by default not. The total memory requirement is then

$$M = 8nb\left\lceil \frac{s}{n/b} \right\rceil + 24sb^2 \ .$$

Once again an upper bound on performance is sought from the modified expression

$$\begin{aligned} M &= 8nb\frac{s}{n/b} + 24sb^2 \\ &= 32sb^2 \ , \end{aligned} \tag{5.5}$$

and hence

$$b = \sqrt{\frac{M}{32s}} \ .$$

Employing user directed caching, the optimum memory partitioning is seen to be

$$\frac{mserver}{mslave} = \frac{s}{3} \ ,$$

and depends only on the number of slaves.

Figure 5.8 repeats the comparison of figure 5.7 but assuming the server level caching is user directed. Out of the 24 ports currently available in the **fast** configuration, one is used by the object server. If the

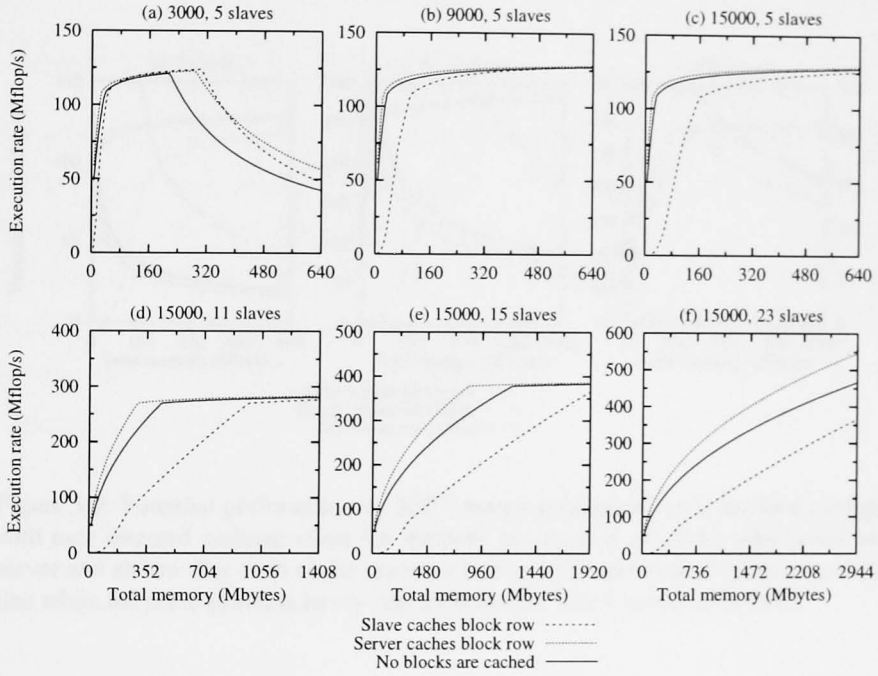



Figure 5.8: Potential performance of matrix multiplication in the **fast** configuration with user directed caching when the memory is assumed optimally partitioned between server and slaves.

memory is 32 Mbyte per slave then there is little to be gained by using more than about 11 slaves at which point the computation reaches bandwidth limit. However, while the presentation has only been in terms of a predicted upper bound on achievable performance, evidence suggests it is worthwhile partitioning application memory between server and slaves to support user directed caching rather than distributing the equivalent total memory purely between the slaves. In terms of computation runtime, caching only gives a significant benefit when the computation is bandwidth limited. This occurs when

the number of slaves is relatively high. Clearly when there is only a single slave the runtime is dominated by computation. The lower bound computed for $T_s$ is $\frac{T_1}{s}$ so clearly the influence of the communication cost remains small and this explains why the difference in execution rate between the organisation which caches a block row at server and that which does no caching is small if neither is bandwidth limited. What actually happens when the computation is not bandwidth limited is that communication by one slave is overlapped by computation performed by another. Figure 5.9 shows how the different caching options compare for the larger problem size of $30K^2$, but also the percentage utilisation of the interconnect,

$$100 \times \frac{\text{lwr}\{T_\infty\}}{T_1/s} \ .$$

If the total memory size is modest there is a significant benefit to be gained through the use of server

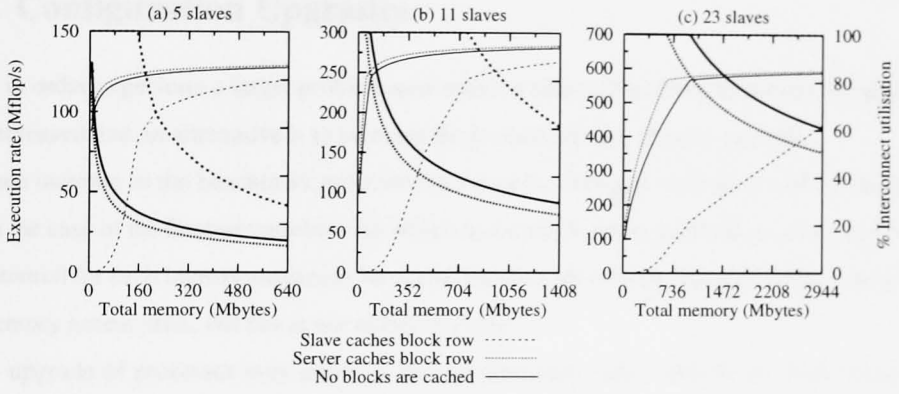



Figure 5.9: Potential performance of $30K^2$ matrix multiplication in the **fast** configuration with user directed caching when the memory is assumed optimally partitioned between server and slaves. For each cache reuse pattern the execution rate is shown by the fainter line while the corresponding heavy line indicates the interconnect utilisation.

level block row caching. For instance, if the memory size is 32 Mbyte per slave and there are 23 slaves an increase in maximum performance from 493 to 568 Mflop/s can be expected, implying a potential reduction in runtime from over 30 hours by 4 hours. For larger memory sizes the performance of these two organisations is very similar, since bandwidth limiting does not occur, but there remains a significant difference in interconnect utilisation.

While the object store is contained within a single machine the memory available for server level caching is fixed. If the server machine is configured with rather more memory than the slave machines then for any matrix multiplication which is structured in this way, it is possible to predict how much memory is required to support this degree of caching. It is possible to increase the server level memory available for caching by distributing the object store over multiple machines, as in the experiments in

the **ATM** configuration.

It is recalled though that to benefit from caching a block row at slave level it is necessary for each slave to be computing a different block row. Yet when all tasks are contained in a single bag any slave may get any task. It is necessary to implement a structure which allows each slave to retrieve a predictable sequence of tasks in the normal case, yet allows correct recovery in case a slave fails. One possible approach is to use the two level bag of tasks structure of section 2.4.7. A slave would select a block row by removing an entry from the first level bag, within an atomic action. Before committing that action, the slave would complete all tasks for that block row, each within a separate top level action. If the slave fails completed tasks for that block row are not undone, but the block row task itself remains available in the higher level bag for continuation by another slave.

## 5.4 Configuration Upgrades

So far, in order to perform a larger problem size within a reasonable time, the number of processors has been increased, but an alternative is to increase the processing rate of each machine.

Some increase in the benchmark processor rate may be expected with more careful tuning, particularly in the case of the Pentium machine for which less corroborative published results have been found. The potential for such improvement may be quantified through consideration of instruction level timings and memory access rates, but this is not attempted here.

An upgrade of processor may imply in the simplest case only a change in clock speed. Pentium processors similar to those used in experimental work may now be clocked at 200 MHz. If suitable published benchmark data is unavailable, a gross simplification for modelling purposes is to assume that increasing the clock speed by 1.5 times in this way translates to increasing the processor rate $P$ used in the model by the same factor. In practice memory or system bus access costs could reduce the impact of the clock rate increase. However, it is reasonable anyway to investigate the performance of the applications in a network of machines of some defined processor rate, perhaps based on measurements such as Lapack [43]. A multiprocessor may be employed simply as a single component within the distributed computation, using parallelism locally to compute a single task at a time.

Figure 5.10 shows the effect on the performance of a large scale matrix multiplication in the **fast** configuration of two alternative processor upgrades. The first assumes a speed increase of a factor of 1.5, which is the best that can be expected from a simple change in clock speed from 133 to 200 MHz. The second alternative assumes the speed increases by a factor of 2 again. The mythical machine is labelled a P5/400, but this is not important. It is in the class of a 50 MHz RS6000 [43], and operates in an equivalent environment to the **fast** configuration.

For the presentation, the number of processors employed is scaled so that the total nominal cpu rate is the same in each case and the range of total memory configurations is 0–128 Mbyte per processor

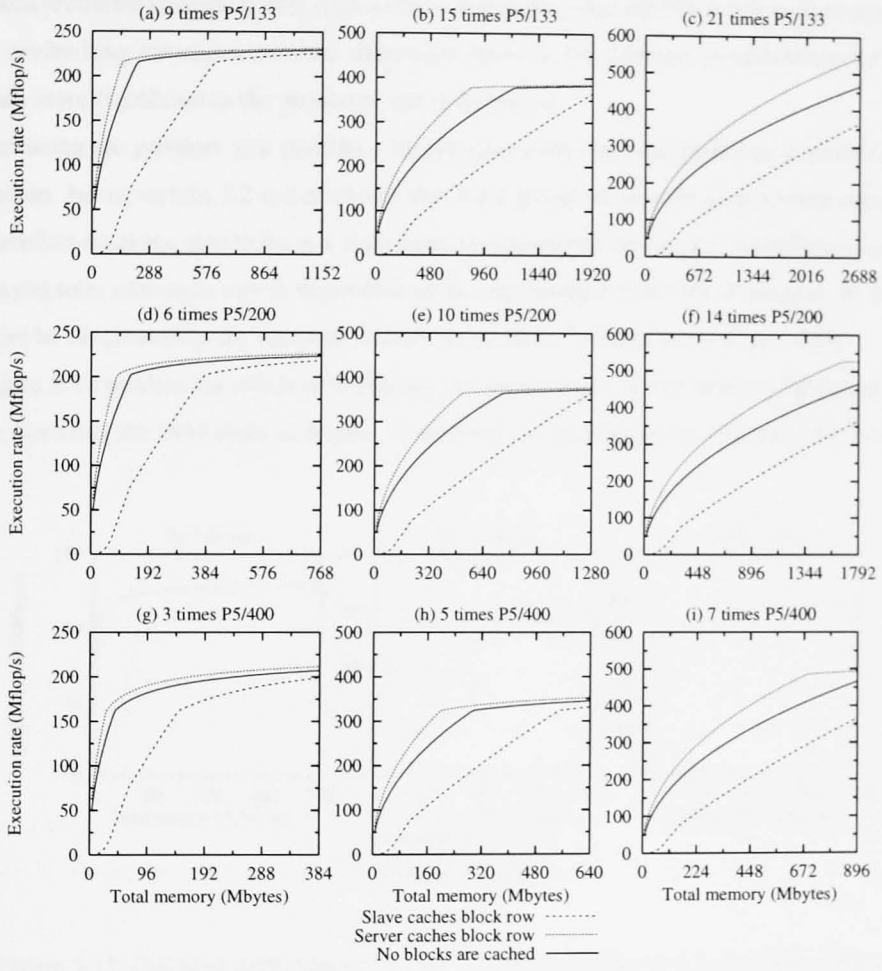



Figure 5.10: Potential performance of $15K^2$ matrix multiplication in the **fast** configuration when alternate processor upgrades are made.

as before. Also as before, the server caching option assumes optimum memory partitioning. Since the memory configured per slave remains the same through the upgrades, so too does the range of block sizes. Thus the bandwidth limiting performance is not altered. However the single slave time depends on both the local computation time and the I/O cost. While the former drops in proportion to the processor rate upgrade, the latter remains unchanged so the single slave time doesn't reduce in proportion to the increase in processor rate. Hence the bound on nonlimiting execution rate falls a little with each processor upgrade in this scaled study. Similarly, since the I/O cost becomes more significant in the nonlimiting execution rate, the difference between the different organisations for cache reuse becomes more significant as the processor rate is upgraded.

Increasing the problem size permits a higher execution rate, and therefore a greater potential for parallelism, but in section 5.2 it was shown that for a given cache utilisation pattern and given block, and therefore memory, size there is a maximum execution rate which is independent of problem size. This asymptotic execution rate is dependent on the aggregate bandwidth of accesses to store and will therefore be dominated by the smallest of the bandwidths of communications and disk.

Figure 5.11 models the effect of replacing the interconnect in the **fast** configuration by an ATM switch, but using the IBM disks as before. Comparing the graphs specifically with (c)–(e) in figure 5.8,

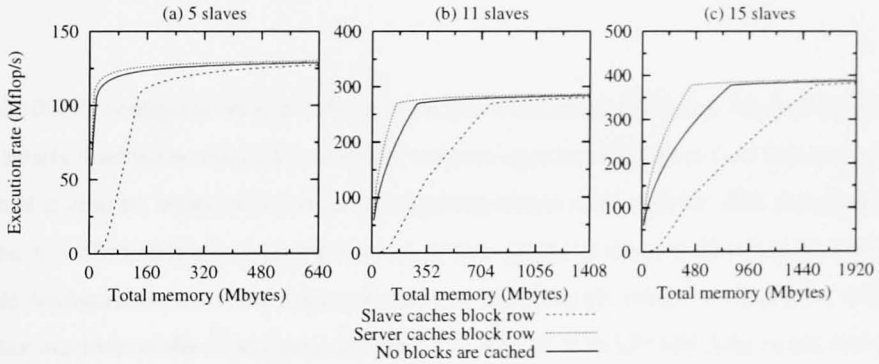



Figure 5.11: Potential performance of $15K^2$ matrix multiplication in the **fast** configuration when the interconnect is replaced by an ATM switch.

the obvious difference is the increase in limiting performance. There is a small difference in nonlimiting performance, but since most communications is overlapped in this region the difference is not obvious. However, there will be a significant reduction in utilisation of the interconnect. If the memory configured is 32 Mbyte per slave and there are 15 slaves the potential performance when using server caching is 379 Mflop/s; a runtime of under 5 hours. Since the communications bandwidth is significantly greater than the disk bandwidth, the benefit to be gained through server level caching is clearly greater. In the

same instance as above, if no blocks are cached the potential performance is 297 Mflop/s; a runtime of over 6.3 hours. However the computation is almost bandwidth limited at this point and is certainly so without the block row caching. An obvious step is to replace the IBM disk by a pair of MAXTOR disks configured as RAID-0, whereupon the performance is as shown in figure 5.12. The read throughput

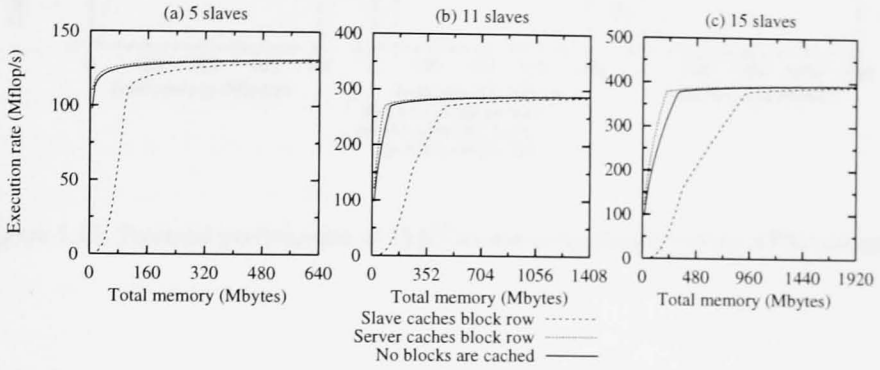



Figure 5.12: Potential performance of $15K^2$ matrix multiplication in the **fast** configuration when the interconnect is replaced by an ATM switch and the single disk by a pair of MAXTOR disks managed using the software RAID system at level 0 but still connected to the single node.

using the RAID configuration is just about twice that of the single IBM disk, but the network throughput is also nearly doubled so network bandwidth remains significantly higher than disk throughput.

While it is to be hoped that further development might yield a higher disk throughput on a single machine, it is ultimately necessary for reasons of throughput or space to distribute the object store. The example configuration used here is then obtained by matching the single machine store configuration on one other machine so the object store is distributed over four MAXTOR disks in all, two per machine. It might be possible to decluster individual objects between object store nodes and obtain parallelism within a request for a single object. Here however this distribution is done at the granularity of whole objects, so each request for a single object communicates with only a single object store node. Consequently, a single slave doesn't benefit from the distribution. When there are two or more slaves, the parallel execution time still cannot be less than the single slave time divided by the number of slaves. However, the minimum achievable execution time is reduced. In calculating this minimum execution time both disk and interconnect bandwidths are multiplied by the number of nodes making up the object store. Using a single ATM switch it is only possible to connect a maximum of 14 slaves, so the maximum performance shown in figure 5.13 is lower than in the previous configuration. Specifically with 14 slaves and 32 Mbyte memory per slave the potential performance in the organisation which caches a row at server level has dropped from 385 to 360 Mflop/s but the computation is now far from being

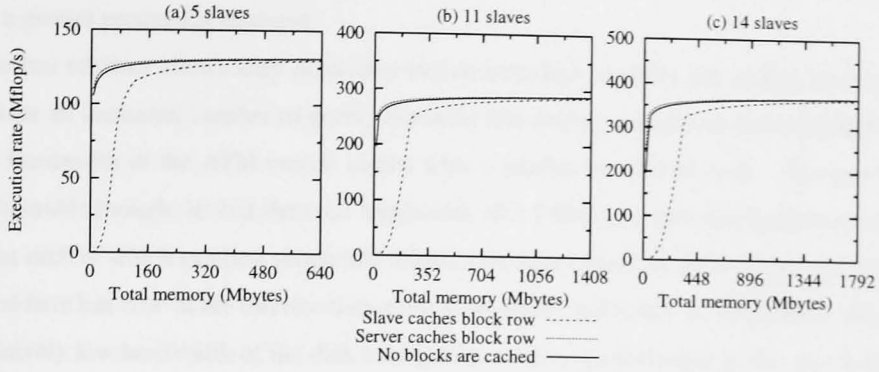



Figure 5.13: Potential performance of $15K^2$ matrix multiplication in the **ATM** configuration.

bandwidth limited so it is possible to consider using higher processing resources. Figure 5.14 indicates performance that may be attainable if up to 14 of the faster processors, nominally P5/400 are used. This configuration is referred to as **ATM/400**. At 32 Mbyte memory per slave, the maximum expected

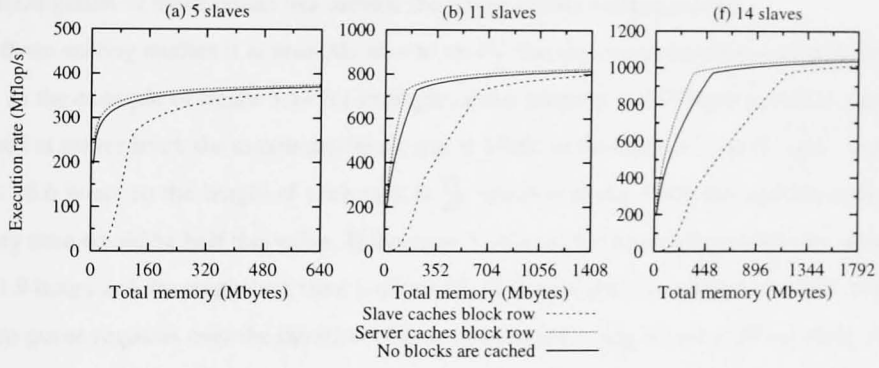



Figure 5.14: Potential performance of $15K^2$ matrix multiplication in the **ATM/400** configuration.

computation rate using 14 slaves and caching a block row at server level is 988 Mflop/s and the overall runtime about 1.9 hours. At this point the computation is once again close to bandwidth limit.

The summary is hardly surprising. To obtain greater performance it is necessary to increase the processing resource, i.e. add more Mflop/s. For a given memory size it is only possible to achieve a certain maximum performance which is determined by the store access bandwidth. In the example computation described increasing memory size allows a larger granularity which reduces the total communication requirement and thereby allows a higher performance for a fixed processing resource and

store access bandwidth. However, ultimately it is necessary to increase store bandwidth in order to utilise a greater processing resource.

The fast ethernet allows easy expansion in that extra hub modules can simply be stacked together to provide an increased number of ports. However this doesn't provide an increased bandwidth. The higher bandwidth of the ATM switch comes with a smaller number of ports. The port arrangement is configurable though, in that the total bandwidth of 2.5 Gbit/s is split evenly between four network modules each of which can be a single 622 Mbit/s link, four 155 or six 100 Mbit/s links [109]. The unit installed here has four of the intermediate network modules and hence 16 155 Mbit/s links. In view of the relatively low bandwidth of the disk configuration, better performance in this application might be achieved through employing the larger number of slower links.

There are more expensive single unit switches which support a greater number of ports, but in scaling the number of processors the point comes eventually when it is necessary to move towards use of multiple switches. The highest speed network link is intended for coupling multiple switches, but simply connecting two units leaves an unbalanced network. A high cost option is to attempt to achieve a large scale well balanced high performance interconnect through mimicking multiprocessor interconnects. Alternatively it may be possible to partition some computations such as to achieve high performance in a less well connected network, even at the cost of replicating some computation data. Full investigation of these issues lies outside the scope of this work however.

In these scaling studies it is possible also to verify that the computation granularity remains appropriate. In the example of figure 5.14 for example, if the memory is 32 Mbyte per slave and a block row is cached at server level, the maximum block size is 1000, so there are $p^2 = 225$ tasks. The single slave time is 26.6 hours so the length of each task is $\frac{T_1}{p^2}$ which is about 7 minutes and the expected average recovery time would be half this value. If there are 14 slaves, the potential parallel execution time is just under 1.9 hours and the single task time is about 6% of the parallel execution time. The average interval between queue requests over the duration of the computation is $\frac{T_{14}}{225}$ which is 30 seconds. In practice all slaves would be making queue requests at about the same time, but the significance of this figure is that the time to dequeue a task should be significantly less than it. If the memory size is 128 Mbyte per node then the maximum block size is 2000 and there are then about 57 tasks. The single slave time is just over 25 hours and the parallel time for 14 slaves just under 1.8 hours. The increased granularity reduces the load on the queue as the expected interval between requests is 114 seconds However the average recovery time at 13 minutes is over 12% of the parallel execution time; the worst case recovery time is nearly a quarter of the parallel execution time.

The significance of the average recovery time can be seen in the fraction $\frac{s}{2N}$, where $N$ is the total number of tasks and $s$ the number of slaves. The expected interval between queue requests is $\frac{T_1}{sN}$. For example, if the matrix size is $20K^2$ and the configuration has 32 Mbytes memory per node with a block row cached at server level then the block size is 1000 again but the total number of tasks is 400. For the

average recovery time to be less than 5% of the parallel run time the number of slaves is $s < 400 \times 0.1$ which is 40. The single slave time is 62.7 hours so if the average interval between queue requests is to be more than 30 seconds for instance the number of slaves must be less than $\frac{62.7 \times 3600}{400 \times 30} = 18.8$. With 18 slaves the best expected execution time is 3.5 hours and the recovery time which is $\frac{62.7 \times 3600}{400 \times 2 \times 60}$ or 4.7 minutes is 2.2% of the parallel execution time.

For both improved performance and lower load on the queue it is desirable to increase the granularity of computation, but for a given problem size this tends to reduce the effectiveness of the recovery mechanism, effectively setting a limit on the degree of parallelism that can be employed. If the granularity remains constant as the problem size increases the number of tasks increases and so the significance of the recovery time for a given degree of parallelism decreases. The structure considered here for matrix multiplication has the desirable property that increasing the problem size allows not just the number of tasks to increase but also the single task time, $\frac{T_1}{p^2}$. This in turn permits the degree of parallelism to be increased without increasing the load on the queue.

## 5.5 Overlapping Communication

### 5.5.1 Server

One obvious way to increase the bandwidth to shared store is to employ multiple buffering in the server. Each slave must take the same time to perform each task, but a greater number of slaves can be accommodated before the computation becomes bandwidth limited. If there is a constant stream of read requests for disk data arriving at the server, then using double buffering it is possible to achieve a maximum 100% increase in throughput if disk and communications bandwidths are equal. The benefit is lessened however if the bandwidths are different. For modelling purposes it is clear that the single slave time is unaltered since it is only possible to achieve overlap at the server in the presence of multiple slave requests. The minimum parallel time is reduced however and a lower bound on the total time to complete all accesses to the shared store is now the maximum of the total cost of all transfers across the communications interconnect and the total cost of all server local disk transfers. Assuming that the cost of fetching a block from slave cache is zero and writing

$$\begin{aligned} tget &= tdget + tcom \ , \\ tput &= tdput + tcom \ , \end{aligned}$$

where $tdget$ and $tdput$ are the local storage access components of corresponding accesses to remote object store and $tcom$ is the cost of the network transfers entailed in such a request, the lower bound on

minimum parallel time, corresponding to (3.16) may be written

$$\begin{aligned} &\mathrm{lwr}\{T_\infty\} = \\ &\quad \max\left\{\max\left\{(N_2+N_3+p^2)tcom\ ,\ N_3tdget+p^2tdput\right\}\ ,\ \frac{T_1}{p^2}\right\}\ . \end{aligned} \tag{5.6}$$

The values for $N_2$, $N_3$ are taken from table 3.3 as before.

To implement such overlapping it is necessary to employ a larger number of buffers. For a single node store, the minimum number required is two, i.e. one more than previously. For the purpose of this analysis it is assumed that two buffers is sufficient in this case. Then, if the store is distributed over $v$ server nodes, the number of buffers required to support double buffering at server level is $v$. A very simple server architecture defines a single thread to monitor for the incoming requests and spawn a separate thread to handle each one, with access to the pair of buffers controlled by locks. The memory then required to support overlap of communications and disk requests at the server are as follows. If no blocks are cached, from (5.2)

$$\begin{aligned} M &= 24sb^2+8vb^2\ , \\ b &= \sqrt{\frac{M}{8(3s+v)}}\ . \end{aligned} \tag{5.7}$$

When a block row is cached by each slave, from (5.3)

$$\begin{aligned} M &= 8s(nb+2b^2)+8vb^2\ , \\ b &= \frac{\sqrt{4s^2n^2+2(2s+v)M}-2sn}{4(2s+v)}\ . \end{aligned} \tag{5.8}$$

When a block row is cached at the server, from (5.5)

$$M = 32sb^2+8vb^2\ , \tag{5.9}$$

and hence

$$b = \sqrt{\frac{M}{8(4s+v)}}\ .$$

Figure 5.15 shows the potential of such double buffering in the configuration of figure 5.8. Clearly, a significant benefit is evident. Without double buffering the largest configuration proved to be bandwidth limited for each of the three caching options and for memory size up to 128 Mbytes per slave. Using double buffering and 32 Mbyte memory per slave it is possible to achieve over 550 Mflop/s. This exceeds considerably the performance expected in the **ATM** configuration, figure 5.13 but not that of the **ATM/400** configuration, figure 5.14

There is a much more noticeable potential benefit available in server level caching than in previous

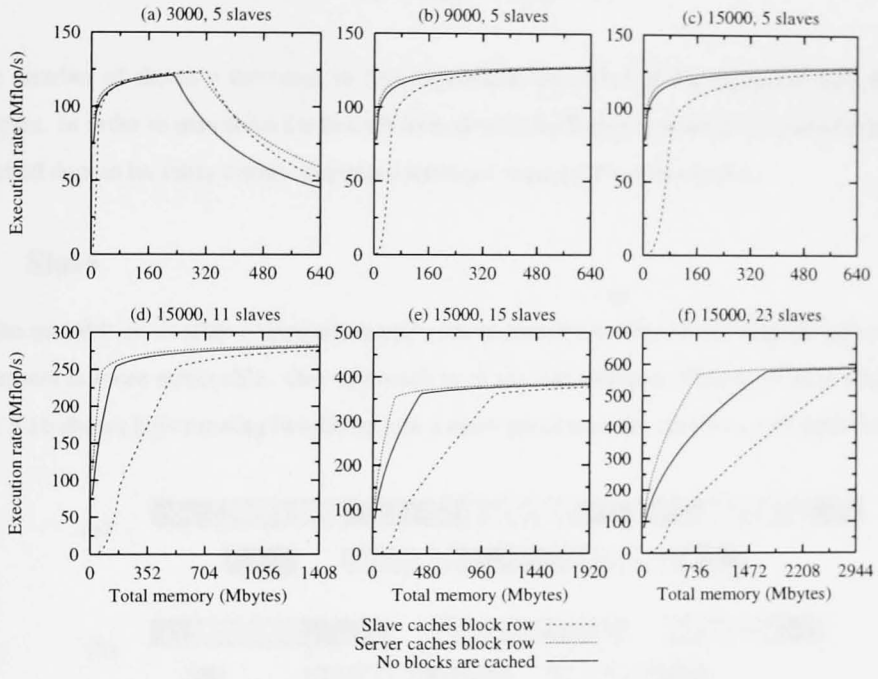



Figure 5.15: Potential performance of matrix multiplication in the **fast** configuration when double buffering is employed at the server level.

configurations. This arises because of the assumption that all disk and communications transfers can be overlapped throughout the computation. In the experimental configuration a disk access is rather slower than a communications transfer, but can be overlapped by a number of accesses to server level cache which do not need to access disk. If the server employs only two buffers then two successive disk requests can occupy those buffers, potentially blocking a series of cache read requests. One way of avoiding such a situation is to allocate a specific buffer just for cache read requests. This would only be required if blocks are cached at server level, but is unlikely to make a great impact on the potential performance since the expression for $b$ in (5.9) changes only slightly to

$$M = 32sb^2 + 16vb^2 \ .$$

As the number of slaves $s$ increases in this expression the effect of the rightmost term becomes less significant. In order to maximise the benefit from double buffering in practice it is desirable for requests for cached data to be fairly evenly dispersed amongst requests for data on disk.

### 5.5.2 Slave

It is also possible to overlap computation and communications at the slaves, but the effect on memory requirement is more noticeable. One approach is to run two separate threads in each slave processor. Figure 5.16 shows how running two threads in a slave processor can achieve communications overlap.

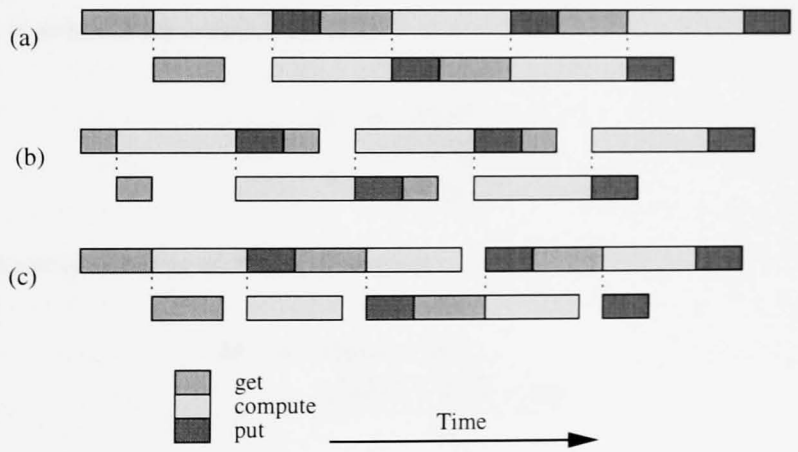



Figure 5.16: Example illustrating potential benefits of employing two threads in a single slave to overlap communication and computation.

In practice, the benefit achieved depends on the relative costs of computation and communication. If the communications cost is not significant with respect to the computation cost the maximum achievable

performance is reduced because of the enforced serialisation of two tasks within each slave. Therefore for a computation comprising $N$ tasks

$$T_\infty \geq \frac{T_1}{N/2} \ .$$

The limit imposed by store access bandwidth remains unchanged, but the single slave time is altered. The effect is for each thread to be able to overlap the other's communications with computation. The slave has only a single processor so all local computation is serialised, as is all communication by the network. Therefore, a lower bound on the time for a single slave with two threads to perform $N$ similar tasks is

$$\max\left\{\sum_1^N Tcomp(i), \sum_1^N Tget(i) + \sum_1^N Tput(i)\right\} \ .$$

Achieving significant benefit from such multithreading in practice depends on the actual distribution of I/O operations through the computation. Matrix multiplication is a favourable example where I/O is regularly spaced.

Assuming as before $tget1 = 0$ the expression corresponding to (3.15) for matrix multiplication is

$$\begin{aligned} T_1 &= \max\left\{N_2 tget2 + N_3 tget3 + p^2 tput \ , \right. \\ &\quad \left. p^3(tmult + tadd)\right\} \ . \end{aligned}$$

Again the values for $N_2$ and $N_3$ can be taken from table 3.3.

Each thread requires the same amount of memory as a single slave in the equivalent single threaded computation. If no blocks are cached, from (5.2)

$$\begin{aligned} M &= 48sb^2 \ , \\ b &= \sqrt{\frac{M}{48s}} \ . \end{aligned} \tag{5.10}$$

When a block row is cached by each slave, from (5.3)

$$\begin{aligned} M &= 16s(nb + 2b^2) \ , \\ b &= \frac{\sqrt{4s^2n^2 + 2sM} - 2sn}{8s} \ . \end{aligned} \tag{5.11}$$

When considering the memory requirement to support block row caching at server level there are effectively twice as many slaves and therefore twice as many blocks being computed at once as in the single threaded organisation. Thus it is not only the slave memory requirement which doubles but also that in the server, so from (5.5)

$$M = 16nb\frac{s}{n/b} + 48sb^2$$

$$= 64sb^2 , \tag{5.12}$$

and hence

$$b = \sqrt{\frac{M}{64s}} .$$

Figure 5.17 shows the effect of employing such techniques in the **ATM/400** configuration. Assum-

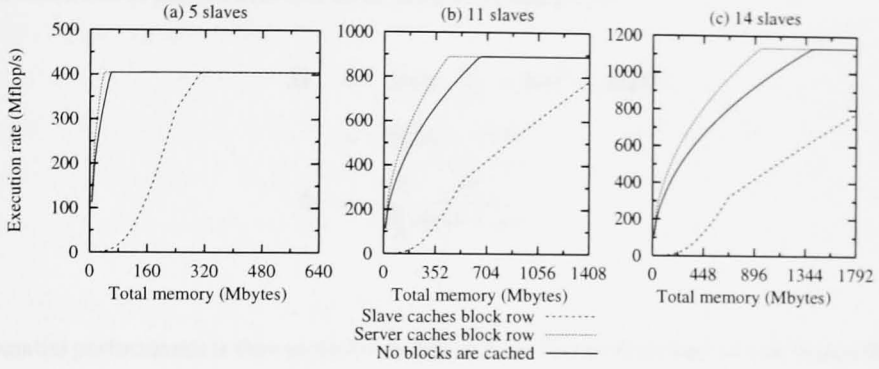



Figure 5.17: Potential performance of $15K^2$ matrix multiplication in the **ATM/400** configuration when two threads are employed at slave level to overlap communications with computation.

ing that full overlap is achievable in each slave the computation rate can in this case approach the total processor resource. Thus when 14 processors are employed, the maximum computation rate is $14 \times 81$ or 1134 Mflop/s. The graphs are then flat because of the assumption that machine speed in the primitive block operations is constant with block size, as observed in practice for large block size. The potential performance for the equivalent single threaded computation is shown in figure 5.14. The greater memory requirement in each slave to support communication overlap ensures the computation is bandwidth limited with fewer slaves than in the earlier configuration. However, it is possible to combine the overlapping mechanisms at server and slave. The memory requirement then increases slightly, such that when no blocks are cached, from (5.7) and (5.10)

$$M = 48sb^2 + 8vb^2 , \tag{5.13}$$

$$b = \sqrt{\frac{M}{8(6s+v)}} . \tag{5.14}$$

When a block row is cached by each slave, from (5.8) and (5.11)

$$M = 16s(nb + 2b^2) + 8vb^2 \ , \tag{5.15}$$

$$b = \frac{\sqrt{16s^2n^2 + 2(4s + v)M} - 4sn}{4(4s + v)} \ . \tag{5.16}$$

When a block row is cached at server level, from (5.9) and (5.12)

$$M = 16nb\frac{s}{n/b} + 8vb^2 + 48sb^2 = 8(8s + v)b^2 \ , \tag{5.17}$$

$$b = \sqrt{\frac{M}{8(8s + v)}} \ . \tag{5.18}$$

The potential performance is then as shown in figure 5.18. The performance is now limited by individual

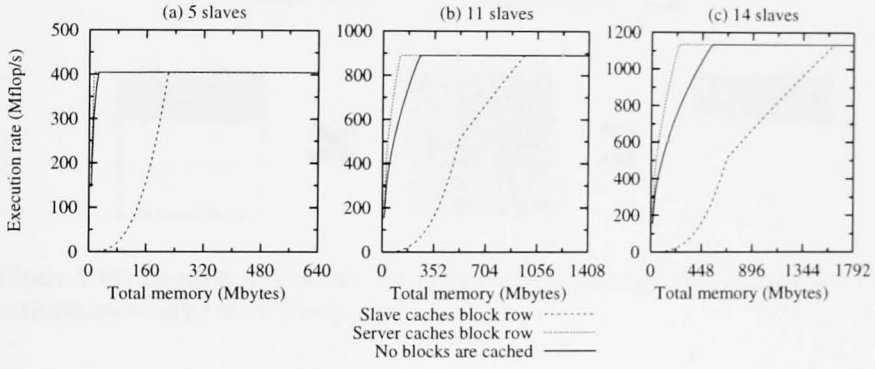



Figure 5.18: Potential performance of $15K^2$ matrix multiplication in the **ATM/400** configuration when two threads are employed at slave level to overlap communications with computation and in addition extra buffering is employed at server level to overlap disk and network transfers.

low level parameters. Other than changing these parameters the only way to improve on the performance is to restructure the computation to reduce the quantity of data accessed to enable use of a greater processor resource.

In the 14 slave configuration with 32 Mbyte memory per node and caching a block row at server level, the block size is 703 and the number of tasks approximately 454. The single slave time is a little over 23 hours and the potential parallel time 1.7 hours. The average interval between queue requests

is rather lower than in the earlier scenario at 13.1 seconds. However if the problem size is larger and the block size the same this interval would also increase. Because two tasks are performed concurrently by each slave machine, the average recovery time must be equal to half the time taken to perform two tasks; $\frac{1.7 \times 3600}{454}$ which is just over 3 minutes, i.e. about 3% of the parallel execution time.

## 5.6 Exploiting Patterns

The bag of tasks structure is very attractive from the point of view of fault-tolerance and load balancing. However, in order to implement such a dynamic execution structure, inter slave cooperation has been minimised. This section examines an alternative more tightly coupled structure described in [112]. This alternative is a data parallel algorithm which caches a block row across the slaves during computation of a block row of the result as shown in figure 5.19. The algorithm partitions the matrices such that a

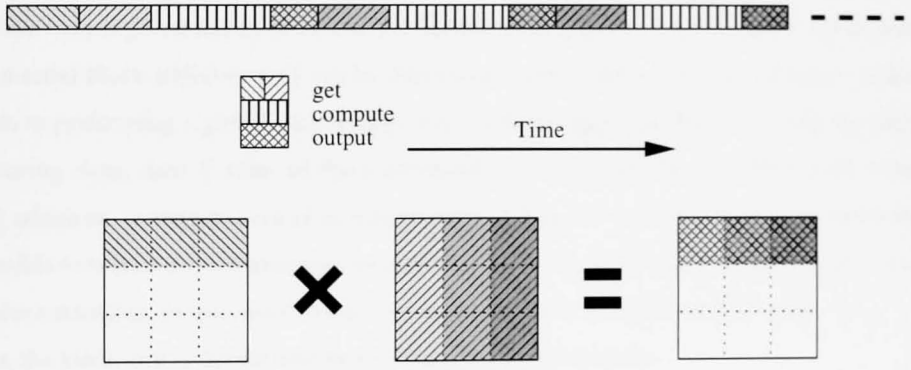



Figure 5.19: Example to illustrate the operation of a data parallel organisation of matrix multiplication using three slaves.

block row of one matrix and block column of the other matrix are accommodated in aggregate memory of available processors, each processor holding the pair of blocks, one from each matrix, which are to be multiplied. After these pairs of blocks are multiplied locally in parallel, a single block of the output matrix is computed by a global sum operation. The computation is coordinated by a master which is here assumed to reside on the same machine as one of the slaves. The key requirement is that the number of blocks across the width of the matrix be determined by the number of processors. For initial analysis the easiest approach is to scale the problem size to fit the number of slaves and total memory configuration. The procedure will then be to fit the problem size to the hardware configuration and compare performance of the bag of tasks and data parallel structures for this problem size. As before each matrix is partitioned into $p^2$ blocks, but here $p = s$, the number of slaves.

In order to compute a block row of the result matrix, it is necessary to load a block row of A into

the slaves, and then step through the block columns of B, loading each column into the slaves at a time, with each slave performing a product. After each such product, the master performs the global sum and writes the result block. The number of disk reads entailed in reading the block row of A is $p$ and that in reading the $p$ block columns of B is $p^2$, the total is then $p(p+1)$. Assuming a need for $p-1$ block transfers over the interconnection network to perform each global sum, and a single write for each block of the result matrix, the I/O count is then the same as in the bag of tasks configuration which employed user directed caching at server level. In this case however it is only necessary to accommodate two blocks in slave memory.

In performing the global sum, the master clearly cannot fetch all the blocks concurrently due to space restrictions. If the master can accommodate two blocks it has to iterate through the slaves one by one, fetching a block at a time and adding this locally to the total. As noted before, the cost of a block addition is proportional to $b^2$ while the cost of a block multiplication is proportional to $b^3$. In terms of scaling, the cost of additions may be small. However in implementing addition, it is more difficult to achieve the very high computation rate of multiplication due to the lower locality. So in practice the cost of $p$ serial block additions may not be insignificant, particularly if $p$, i.e. $s$ is large. A distributed approach to performing a global sum is to perform $\frac{p}{2}$ local additions of a block with the block in the neighbouring slave, then $\frac{p}{4}$ sums of these intermediate results, and so on, reducing the total cost to $\lceil \log_2 p \rceil$ additions. In the presence of an interconnect such as ATM which supports parallel transfer it is also possible to reduce the communications cost entailed in the global sum, at best from $p-1$ to $\lceil \log_2 p \rceil$ single block transfers. In the case of a bus type network this is obviously not the case.

First, the block size is determined by the total configured memory

$$b = \sqrt{\frac{M}{16s}}$$

It is assumed the master and a slave can share the same two buffers within a single machine. Since the matrix width in blocks is equal to the number of slaves, $p = s$, then the matrix size is $n = sb$. Assuming a bus type network, the total time to compute a single block row of the result is

$$\begin{aligned} Tbr_p &= p(p+1)(tdget + tcom) + ptmult \\ &\quad + p\lceil \log_2 p \rceil tadd + p^2 tcom + ptdput \ . \end{aligned}$$

where *tdget* and *tdput* are the costs of local block disk accesses and *tcom* is the cost of a block transfer between local and remote memory as before. The cost of the entire computation is $p$ times this. Figure 5.20 compares a single threaded data parallel computation with the equivalent bag of tasks computation, from figure 5.8. The bag of tasks structure offers best potential performance, presumably because of the need in the data parallel structure for the serial computation of the global sum and block

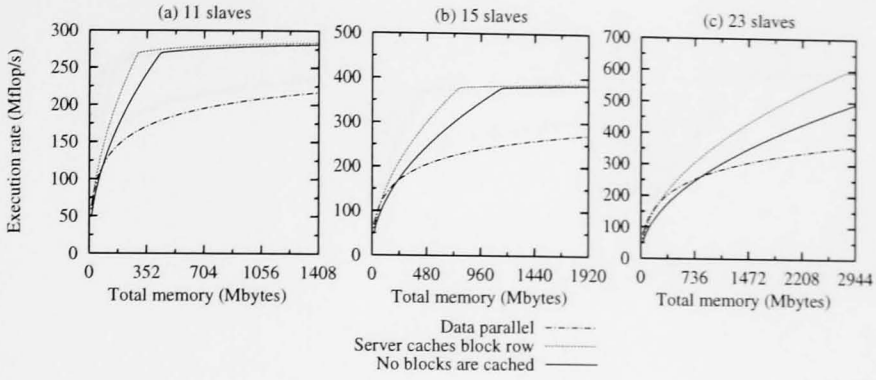



Figure 5.20: Potential performance of single threaded data parallel matrix multiplication and equivalent bag of tasks computation in the **fast** configuration when the problem size is scaled according to the aggregate memory size.

output.

If double buffering is employed at the server then it is possible to overlap disk and network transfer components of all get requests. It is necessary to allocate extra buffer space at server level as described earlier in the case of the bag of tasks structure, so the block size is now

$$b = \sqrt{\frac{M}{8(2s+v)}}$$

where the object store is distributed over $v$ nodes as before. The distributed global sum is performed between the phases of parallel multiplications so no overlap occurs between the two. Similarly, output of the result block is performed serially by the master so no overlap there is possible. Assuming as before a bus based interconnect, a lower bound on the cost of computing a single block row of the result is

$$\begin{aligned} \mathrm{lwr}\{Tbr_p\} &= p(p+1)\max\{tdget, tcom\} + ptmult \\ &\quad + p\lceil \log_2 p \rceil tadd + p^2 tcom + ptdput \ . \end{aligned} \tag{5.19}$$

Again, the cost of the entire computation is $p$ times this. Figure 5.21 compares the performance of such a computation with an equivalent bag of tasks style computation running in the **fast** configuration, similar to that portrayed in figure 5.15. Again, presumably due to the serial component the bag of tasks structure offers the best potential performance.

If the block size is reduced, it is possible to overlap the communications for one output block with the computation for another by running two threads in each slave. Each set of threads computes a

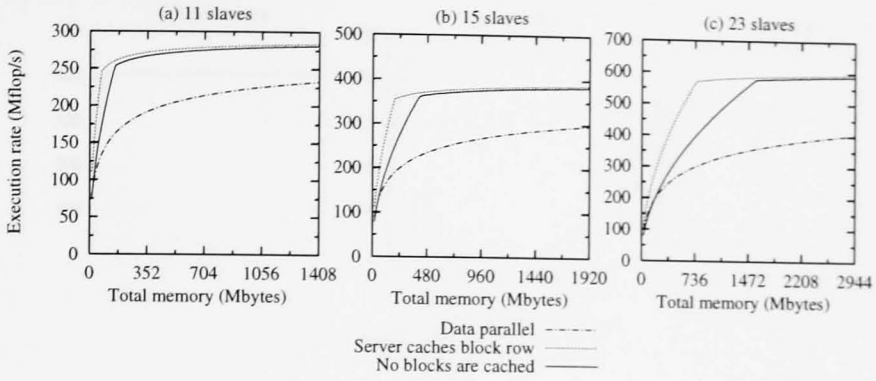



Figure 5.21: Potential performance of single threaded data parallel matrix multiplication employing double buffering at server level and equivalent bag of tasks computation in the **fast** configuration when the problem size is scaled according to the aggregate memory size.

different block row of the output matrix, so there are two block rows cached across the slaves at any time. The block size is given by

$$b = \sqrt{\frac{M}{8(4s+v)}} \ .$$

Once again the overall time is assumed to be the greater of total communications and total computation so, now assuming a switch type network, a lower bound on the cost of computing a single block row of the result is

$$\begin{aligned} \mathrm{lwr}\{Tbr_p\} \quad = \quad & \max\{p(p+1)\max\{tdget, tcom\} \\ & +p\lceil\log_2 p\rceil\, tcom + p(tcom + tdput) \ , \\ & ptmult + p\lceil\log_2 p\rceil\, tadd\} \ . \end{aligned} \tag{5.20}$$

It is assumed that when the object store is distributed over a number of nodes the data parallel organisation can potentially achieve a proportionate increase in throughput of data accesses to the object store. The parallelism of the switch is already taken account of in the communications entailed in the global sum. Figure 5.22 compares the performance of such a computation with that of the approach developed earlier. It turns out that the performance achievable in the data parallel organisation is similar to that achievable in the bag of tasks organisation which caches a block row at server level.

Rather than scaling the problem size to the total memory size a more general analysis begins with the operand matrix size and number of slaves and computes from these the block size. One approach is to partition the matrices into rectangular blocks, $b_1$ by $b_2$ where $b_2 = \frac{n}{s}$. Then a block row $n \times b_1$ is cached and all output corresponding to that block row is computed in $\frac{n}{b_1}$ steps. Figure 5.23 shows

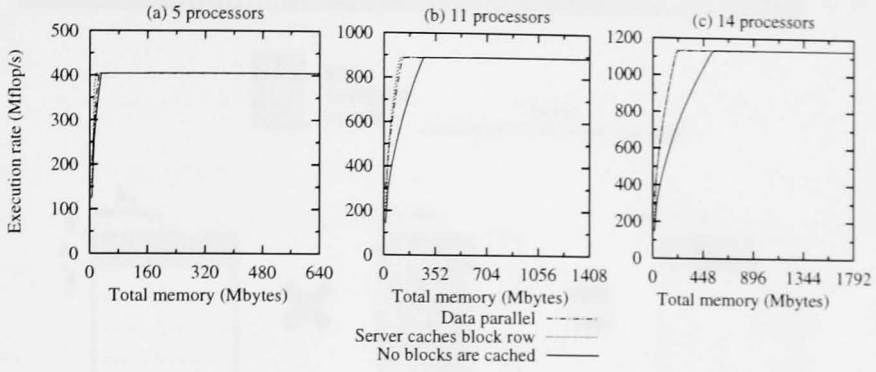



Figure 5.22: Potential performance of multithreading to overlap communications with computation at slave level in the data parallel matrix multiplication and equivalent bag of tasks computation in the **ATM/400** configuration when the problem size is scaled according to the aggregate memory size

a similar computation to that of figure 5.19 but running with two slaves instead of three. In practice the area of the rectangular block in the two input matrices can be slightly larger than that of the square blocks in the earlier example because the block size in the result matrix is smaller, but this is not shown in the example.

In a single threaded computation which exploits double buffering at server level,

$$M = 16b_1n + 8vb_1b_2$$

and so

$$\frac{b_1}{b_2} = \frac{Ms^2}{8(2s+v)n^2} \ .$$

If $Ms^2 > 8(2s+v)n^2$ then $b_1 > b_2$. The matrix blocks are configured so that a single block multiplication fills memory, in order to minimise overall communication, but if $b_1 > b_2$ the product, of size $b_1^2$, is larger than either input block and so cannot be accommodated in memory. If parallel threads are employed to achieve overlap of communication at the slaves the relevant condition is $Ms^2 > 8(4s+v)n^2$. For example if there are 14 slaves performing a $15K^2$ multiplication and the object store is distributed over two nodes, there will be no space problem of this type provided $M < 276$ Mbytes in the single threaded case, or $M < 533$ Mbytes in the multithreaded case. These values correspond to 20 and 38 Mbytes per slave respectively.

The bound can alternatively be interpreted as defining a minimum number of slaves for a given matrix size and memory per slave configuration or a minimum matrix size for given number of slaves

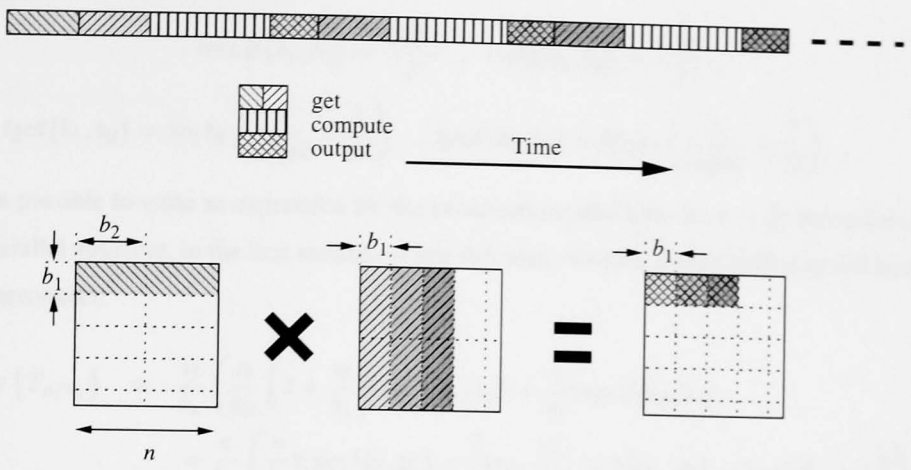



Figure 5.23: Example to illustrate the operation of a data parallel organisation of matrix multiplication using two slaves.

and aggregate memory size. In the case of the single threaded computation above

$$n \geq \sqrt{\frac{Ms^2}{8(2s+v)}} \ .$$

In the **fast** configuration for example if the aggregate memory size is 2944 Mbytes and the number of slaves 23, the minimum matrix size is

$$\sqrt{\frac{2944 \times 10^6 \times 23^2}{8(2 \times 23 + 1)}} = 64359 \ .$$

In the **ATM/400** configuration using 14 slaves with aggregate memory size of 1792 Mbytes the minimum matrix size is

$$\sqrt{\frac{1792 \times 10^6 \times 14^2}{8(4 \times 14 + 2)}} = 27513 \ .$$

If the bound is passed, one option is not to increase the block size proportionately, but this would have an adverse effect on overall performance. Another possibility is to compute more than one block row at a time. However, provided the operand matrix size is large enough this issue does not arise. The following analysis makes this simplifying assumption in a comparison at fixed problem size between the bag of tasks and data parallel structures.

In computing the performance of the data parallel structure where blocks are no longer necessarily square the fuller notation which includes the block dimensions as shown in table 3.2 but abbreviated up till now is employed. The edges of a block are labelled $b_1$ and $b_2$ such that with reference to (4.1), (4.2)

and (4.3)

$$tmult\{b_1, b_2\} = \frac{2b_1^2 b_2}{P} \ , \ tadd\{b_1, b_2\} = \frac{b_1 b_2}{P} \ , \tag{5.21}$$

$$tget\{b_1, b_2\} = 8b_1 b_2 \left(\frac{1}{Dget} + \frac{1}{C}\right) \ , \ tput\{b_1, b_2\} = 8b_1 b_2 \left(\frac{1}{Dput} + \frac{1}{C}\right) \ . \tag{5.22}$$

It is then possible to write an expression for the minimum parallel time for $s = \frac{n}{b_2}$ processors running a data parallel structure, in the first instance single threaded, with no double buffering and using a bus type interconnect:

$$\begin{aligned} \text{lwr}\left\{T_{n/b_2}\right\} \quad = \quad & \frac{n}{b_1}\left(\frac{n}{b_2}\left(1 + \frac{n}{b_1}\right) tget\{b_1, b_2\} + \frac{n}{b_1} tmult\{b_1, b_2\}\right. \\ & \left. + \frac{n}{b_1}\left(\frac{n}{b_2} tcom\{b_1, b_1\} + \left\lceil \log_2 \frac{n}{b_2} \right\rceil tadd\{b_1, b_1\} + tput\{b_1, b_1\}\right)\right) \end{aligned} \tag{5.23}$$

As before, an expression can be derived for a multiple threaded computation which does employ server level double buffering and uses a switch type network.

$$\begin{aligned} \text{lwr}\left\{T_{n/b_2}\right\} \quad = \quad & \frac{n}{b_1} \max\left\{\frac{n}{b_2}\left(1 + \frac{n}{b_1}\right) \max\left\{tdget\{b_1, b_2\} \ , \ tcom\{b_1, b_2\}\right\}\right. \\ & + \frac{n}{b_1}\left(\left\lceil \log_2 \frac{n}{b_2} \right\rceil tcom\{b_1, b_1\} + tput\{b_1, b_1\}\right) \ , \\ & \left. + \frac{n}{b_1}\left(tmult\{b_1, b_2\} + \left\lceil \log_2 \frac{n}{b_2} \right\rceil tadd\{b_1, b_1\}\right)\right\} \ . \end{aligned} \tag{5.24}$$

Figure 5.24 compares the performance of the data parallel structure with that of the bag of tasks for constant problem size of $30K$ in the **ATM/400** configuration. In this case, the bag of tasks structure

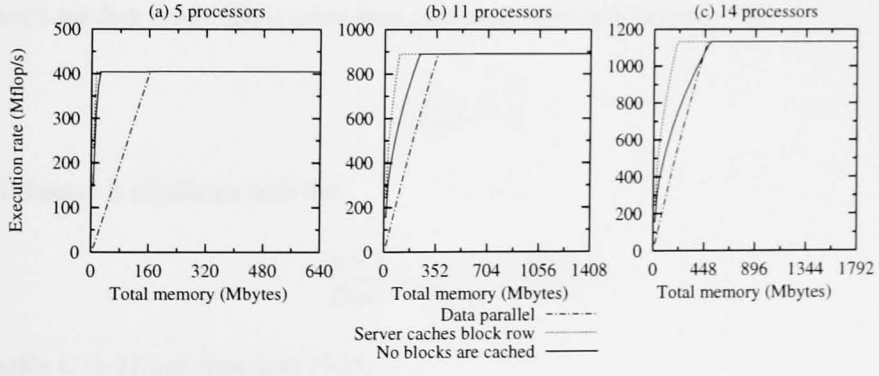



Figure 5.24: Potential performance of data parallel organisation of matrix multiplication with multiple threading at slave level and double buffering at server level computing a fixed problem size of $30K^2$ in the **ATM/400** configuration.

performs better than the data parallel structure. Apparently the communications bandwidth affects the data parallel structure before the bag of tasks structure. This suggests that the I/O cost is higher in the data parallel computation. From (5.6) and (5.24) the total communications cost is picked out for the bag of tasks computation where a row is cached at server level and the data parallel computation.

$$\begin{aligned} Tcbot &= \left(\frac{n}{b}\right)^2 \max\left\{\left(\frac{2n}{b}+1\right) tcom\{b,b\} \ , \ \left(\frac{n}{b}+1\right) tdget\{b,b\} + tdput\{b,b\}\right\} \\ Tcdatap &= \frac{n}{b_1}\left(\frac{n}{b_2}\left(1+\frac{n}{b_1}\right) \max\{tdget\{b_1,b_2\}, tcom\{b_1,b_2\}\}\right. \\ &\quad \left. + \frac{n}{b_1}\left(\left\lceil \log_2 \frac{n}{b_2}\right\rceil tcom\{b_1,b_1\} + tput\{b_1,b_1\}\right)\right) \end{aligned}$$

Substituting for the various low level primitives from (4.2) and (4.3), (5.21) and (5.22), and writing

$$p_1 = \frac{n}{b_1} \ , \ p_2 = \frac{n}{b_2}$$

$$\begin{aligned} &Tcdatap - Tcbot = \\ &\quad 8n^2(p_1+1)\max\left\{\frac{1}{Dget}, \frac{1}{C}\right\} + \frac{8n^2}{C}\left(\lceil \log_2 p_2\rceil + 1\right) + \frac{8n^2}{Dput} \\ &\quad - 8n^2 \max\left\{\frac{2p+1}{C} \ , \ \frac{p+1}{Dget} + \frac{1}{Dput}\right\} \end{aligned} \tag{5.25}$$

In order for the I/O cost to be higher in the data parallel computation it would be necessary for

$$Tcdatap - Tcbot > 0 \ .$$

Commonly the disk bandwidth is lower than communications bandwidth so that

$$\frac{1}{Dget} \geq \frac{1}{C} \ .$$

If the difference is significant such that

$$\frac{p+1}{Dget} + \frac{1}{Dput} \geq \frac{2p+1}{C} \ ,$$

i.e. roughly $C \geq 2Dget$, then from (5.25)

$$\begin{aligned} &Tcdatap - Tcbot = \\ &\quad \frac{8n^2}{Dget}(p_1 - p) + \frac{8n^2}{C}\left(\lceil \log_2 p_2\rceil + 1\right) \ . \end{aligned}$$

This expression is positive for $p_1 \geq p$, and in practice this condition will generally hold.

If the difference between communications and disk bandwidth is not so great such that

$$\frac{p+1}{Dget} + \frac{1}{Dput} < \frac{2p+1}{C} ,$$

i.e. roughly $Dget \leq C \leq 2Dget$, then from (5.25)

$$Tcdatap - Tcbot = 8n^2 \left( \frac{p_1+1}{Dget} - \frac{2p+1}{C} \right) + \frac{8n^2}{C} \left( \lceil \log_2 p_2 \rceil + 1 \right) + \frac{8n^2}{Dput} .$$

In this case the expression is positive for certain values of $p$ and $p_1$ dependent on the relative values of $C$ and $Dget$. For large $p, p_1$ this relationship can be expressed as

$$\frac{C}{Dget} \geq \frac{2p}{p_1} .$$

If $C = 2Dget$ then as before the condition is simply $p_1 \geq p$. If $C = Dget$ then the condition is $p_1 \geq 2p$. Analysis of the extent to which this condition holds in general is deferred to future work, but certainly the point will be reached as the problem size grows. Where the number of slaves and total memory size is modest the point will be reached sooner.

In the experimental configuration $C$ is 9.1 Mbyte/s and $Dget$ at 5.5 Mbyte/s is over half this value. Figure 5.25 compares the I/O cost in the alternative computations for the experiment of figure 5.24. Also

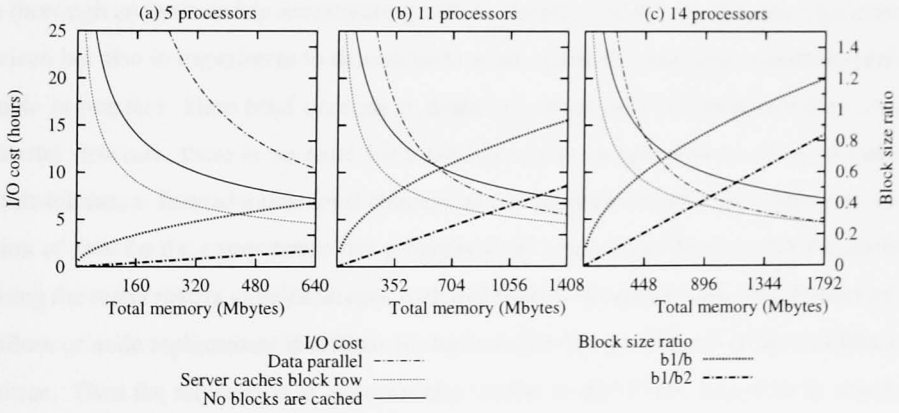



Figure 5.25: Computed I/O cost of data parallel organisation of matrix multiplication with multiple threading at slave level and double buffering at server level computing a fixed problem size of $30K^2$ in the **ATM/400** configuration. A fainter line indicates a total I/O cost while a heavy line indicates a block size ratio.

shown is the ratio of the smaller block dimension in the data parallel organisation to the block size in the bag of tasks organisation which caches a row at server level and the ratio of the two block dimensions in

the data parallel computation. In this configuration the I/O cost of the data parallel structure falls below that of the bag of tasks structure as the ratio of smaller block dimension in the former to block size in the latter approaches 1.2. In this example the matrix size is such that it would be necessary to configure more memory to reach the point at which the block shape in the data parallel organisation is square. As described before it is not possible for $b_1 > b_2$ so if the matrix size were reduced or memory size or number of slaves increased such that the operation moved beyond this point it would be necessary to compute two block rows at once. Then either one block row can be computed with high performance and one with low performance or both with some intermediate performance. As the number of slaves is increased further it becomes possible to compute two block rows at once using square blocks, but on yet further increase it is necessary to compute three block rows at once and the situation repeats itself.

For any particular matrix size there is a relationship between number of slaves and aggregate memory size for which the data parallel computation uses blocks of optimum shape. For a certain region around these optimal points the data parallel structure achieves better overall performance than the bag of tasks structure. In other situations the bag of tasks structure can actually perform better. One such situation is when the computational resources are modest. However, if the data parallel structure does have the advantage for a given configuration then that advantage can be lost if the resources change during the computation. Here the actual value of the performance advantage is not pursued. It is sufficient that the bag of tasks structure can be competitive with the data parallel structure.

It must be noted as elsewhere in this work that the analysis has tended to be in terms of maximum achievable performance and clearly further work is necessary not only as elsewhere in making a more thorough analysis and in investigating further the effect of the configuration parameters on this comparison but also in experiment to determine to what extent the predicted maximum performance is achievable in practice. Here brief mention is made of certain obvious implementation issues. In the data parallel structure, there is no need for a mechanism as complicated as a bag of tasks to implement fault-tolerance. Instead a user level checkpoint can be constructed which maintains on disk some indication of how far the computation has progressed. It is necessary however for the distributed data comprising the result matrix to be consistent with this record. In order to support tolerance of a transient node failure or node replacement it suffices to store on disk the coordinates of the last block output, as it is written. Then the recovery cost is essentially similar to that of the bag of tasks structure. Tolerating permanent loss of a slave is rather more difficult, since the granularity at which the result matrix can be written changes. One possibility is to recover to the start of the current block row though the recovery cost is then clearly greater. Similar difficulties arise when integrating an extra slave into the computation. There is an added complication in the data parallel structure relating to reconfiguration. The calculations have assumed that disk access is at maximum rate in all cases. Clearly after a reconfiguration the accesses in the data parallel structure will be rather more fragmented and therefore may not be at maximum rate.

While at this point the comparison appears favourable in terms of the bag of tasks structure there is scope for furthering the comparison. In the data parallel organisation each slave reads blocks in turn from the same block row of the second input matrix for each block row of the first input matrix and could therefore potentially gain by caching that block row of the second input matrix. In section 3.8.1, the possibility of caching both a block row at slave level and a block column at server level was referred to for the bag of tasks organisation, though it was noted that each block column cached at server level must be reloaded a number of times depending on the number of slaves. Earlier analysis in this chapter for the bag of tasks structure suggests caching a block row at slave level forces too great a reduction in the block size to be worthwhile. However that was on the assumption that the block row was cached in slave memory. If slaves have sufficient local disk space it is possible in either organisation to use this to cache a block row. The cost of accessing data from this store may or may not be cheaper than accessing it from server memory, but there is a cost incurred in loading data into it; the cost of writing from memory to the local disk. Enhancing the earlier cache model and pursuing the comparison between the alternate computation structures is deferred to future work. However it is noted that the use of multithreading at slave level in a multiple level bag of tasks structure offers an opportunity for allowing concurrent access to the same bag of tasks at the second level since both threads running in a single slave can reuse the same block row cached locally.

Ultimately it would be a surprising result if the bag of tasks organisation turns out to be the best choice in general purely from a straight forward performance perspective. Here the claim is that there is likely to be a strong case for it particularly when taken in the context of an environment of changing resources.

## 5.7 Another Example: LU Factorisation

While readily adapted to a bag of tasks based structuring approach, Cholesky factorisation assumes that the input matrix is symmetric positive definite, essentially belonging to a restricted class of well conditioned matrices. This section gives brief consideration to the problems that arise in attempting to structure the rather more generally applicable LU factorisation using the approach described in this work.

In the matrix equation

$$AX = B$$

A becomes LU where L is lower diagonal and U is upper diagonal and one of L and U has unit diagonal. There are a number of possible arrangements of the same basic operations required to perform the factorisation. One such permutation, the Crout factorisation is illustrated in figure 5.26, and figure 5.27. It is possible in all cases to overlay the operand matrix by the two result matrices and compute the factor-

isation "in place", though the example codes suggest the use of separate arrays. All the permutations

```
    void lu(Matrix a, Matrix l, Matrix u)
    {
        for (k=0; k<sz; k++) {
            // complete compution of column k of l
 5          // l[k:sz][k] = a[k:sz][k] - l[k:sz][0:k-1]*u[0:k-1][k]
            for (i=k; i<sz; i++) {
                sum = 0;
                for (j=0; j<k; j++) {
                    sum += l[i][j]*u[j][k];
10              }
                l[i][k] = a[i][k] - sum;
            }
            // complete computation of row k of u
            // u[k][k+1:sz] = u[k][k+1:sz] - l[k,0:k-1]*u[0:k-1][k+1:sz]
15          // u[k][k+1:sz] = u[k][k+1:sz]/l[k,k] — assume l[k][k] != 0
            for (j=k+1; j<sz; j++) {
                sum = 0;
                for (i=0; i<k; i++) {
                    sum +=l[k][i]*u[i][j];
20              }
                u[k][j] = (a[k][j] - sum)/l[k][k];
            }
        }
    }
```

Figure 5.26: Crout's LU factorisation

perform about $\frac{2n^3}{3}$ flops in factorising a matrix of $n^2$ elements.

It is possible as before to improve performance by employing blocking techniques to increase locality and concentrate computation in block level multiplications, as illustrated in figure 5.28. It is necessary to introduce a block level factorisation which must extend all the way to the lower edge of the matrix, but it is possible to implement this operation as a factorisation of the square block on the diagonal and a series of triangular solves of the upper factor and each of the square blocks below the diagonal in the same block column.

The complication that does arise though is the need to employ pivoting to ensure numerical stability. In an arbitrary matrix, the divisor employed in the basic factorisation, figure 5.27 line 21, can turn out to be arbitrarily small; in the worst case it can be zero. Even if division by zero is avoided, the numerical errors accrued in the computation increase rapidly with matrix size. Pivoting entails the application of trivial manipulations, i.e. row or column swaps, to ensure that the value to be used as divisor is non-zero, and ideally as large as possible. A key point is that the manipulation must choose a suitable element to use as divisor out of those available, just as the division is about to be performed. While the set of

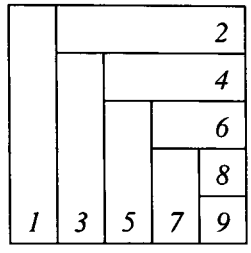



Figure 5.27: Order of computations in Crout factorisation

```
void blocked-lu(Matrix A, Matrix L, Matrix U)
{
    for (k=0; k<p; k++) {
        // complete compution of block column k of L
        for (i=k; i<p; i++) {
            sum = 0;
            for (j=0; j<k; j++) {
                sum += L[i][j]*U[j][k];
            }
            L[i][k] = A[i][k] - sum;
        }
        factor L[k:p-1][k]
        // complete computation of block row k of U
        // U[k][k+1:p] = U[k][k+1:p] - L[k,0:k-1]*U[0:k-1][k+1:p]
        // U[k][k+1:p] = U[k][k+1:p]/L[k,k] — assume L[k][k] != 0
        for (j=k+1; j<p; j++) {
            sum = 0;
            for (i=0; i<k; i++) {
                sum +=L[k][i]*U[i][j];
            }
            tmp = A[k][j] - sum;
            solve L[k][k]*U[k][j] = tmp for U[k][j];
        }
    }
}
```

Figure 5.28: Blocked implementation of Crout's LU factorisation, for a matrix of $p^2$ blocks.

manipulations which are performed under a given pivoting strategy can be equated to a set of a priori manipulations preceding a factorisation without pivoting, it is not obvious a priori which element will be the right one to choose as pivot at any given point in the factorisation. While a definition of a priori matrix manipulations which can alleviate the need for pivoting is highly desirable the current generally accepted approach is to identify pivots on the fly during the factorisation.

The set of values from which a pivot element may be chosen depends on which organisation of the factorisation is chosen [105]. Specifically, organisations such as the Crout factorisation which delay writing to elements in the bottom right corner as long as possible prohibit full pivoting which allows closest bound on error growth. Full pivoting maximises the search space for the pivot element by searching through the whole of the lower right portion of the matrix which is not yet finally written and then swapping both row and column to move the pivot element to the active location on the diagonal. Clearly updates arising from the part of the matrix already factored must be fully applied to this region for the search to be worthwhile, e.g. as in classical Gaussian elimination, but also the assured high accuracy entails a rather high cost in identifying pivots. In practice a more generally applicable approach is employed though the error growth is less constrained. This alternative, namely partial pivoting, restricts the search for a pivot element to the remainder of either the current row or current column. In the case of the Crout factorisation it is convenient to search down the current column, as shown in figure 5.29.

It is possible to introduce pivoting into the blocked algorithm, at the appropriate point in the block level factorisation. The pivot search should be down the whole of the remainder of the current column and swaps made of entire rows. Recent implementors of out of core LU factorisation have tended to organise the out of core data by full columns or column blocks [118, 70, 44]. Each phase of the computation computes a whole block column which is sized according to aggregate memory, typically in a data parallel organisation. Where data is accessed by whole column it is possible to search for a pivot down a column and to apply accumulated pivot swaps to the overall matrix by column or column block. However, such an organisation is less suited to the dynamic computation structures employed in this work. It is not convenient to make arbitrary accesses to the out of core matrix; at times by block, at other times by row or column. The ideal, seen in the other example computations, is where it is possible to choose a single uniform block shape independent of the matrix size, but this is one example application where an optimal data structure is not so obvious. The following claims that it is possible to employ a data organisation which permits use of a bag of tasks based approach. The analysis and verification are both limited however, being mainly deferred to future work; the intention being just to indicate a starting point.

In an out of core factorisation where data is organised by full column or column block it is possible to maintain the out of core matrix in either pivoted or unpivoted form [44]. The pivot swaps may be applied to the out of core data as they are found or saved up and applied globally to the out of core data

```
void lu(Matrix a, Matrix l, Matrix u)
{
    for (k=0; k<sz; k++) {
        // complete compution of column k of l
        // l[k:sz][k] = a[k:sz][k] - l[k:sz][0:k-1]*u[0:k-1][k]
        for (i=k; i<sz; i++) {
            sum = 0;
            for (j=0; j<k; j++) {
                sum += l[i][j]*u[j][k];
            }
            l[i][k] = a[i][k] - sum;
        }
        // perform pivotting
        max = 0;
        for (i=k; i<sz; i++)
            if (abs(l[i][k]) > max) {max = abs(l[i][k]); p = i;}
        swaprows(k, p);
        // complete computation of row k of u
        // u[k][k+1:sz] = u[k][k+1:sz] - l[k,0:k-1]*u[0:k-1][k+1:sz]
        // u[k][k+1:sz] = u[k][k+1:sz]/l[k,k] — assume l[k][k] != 0
        for (j=k+1; j<sz; j++) {
            sum = 0;
            for (i=0; i<k; i++) {
                sum +=l[k][i]*u[i][j];
            }
            u[k][j] = (a[k][j] - sum)/l[k][k];
        }
    }
}
```

Figure 5.29: Crout's LU factorisation with partial pivoting by column.

on completion of the factorisation. It is even possible to avoid explicitly applying derived pivot swaps to the out of core data at all by storing an index array with the out of core data and applying the pivot swaps to the right hand side of a system of equations when using the factorisation to solve that system. If pivot swaps are not applied explicitly to the out of core data as they are found however, it is necessary to apply them during factorisation, to convert between unpivoted out of core data and pivoted in core data. This operation is easily performed if the data is always accessed by full length block column, but in the organisation described here which is intended to support an arbitrary and varying number of slaves, accesses are by block rather than block column and it is anticipated that applying the transformation to and from pivoted form would be costly. Instead an approach is defined for applying the pivot swaps to the out of core matrix as they are identified. The discussion is in terms of an in place factorisation but adapts readily to one producing separate triangular matrices. It is assumed that the matrix is partitioned into $p^2$ square blocks as in other examples developed in this work. Factorisation of the $[k,k]$ block will in general necessitate swapping rows between block row $k$ and all of block rows $k+1$ to $p$. The procedure is to apply pivot swaps by block column. For the $j$th such block column, the block $[k,j]$ is read in. Then in turn each of blocks $[k+1,j]$ to $[p,j]$ is read, necessary pivot swaps applied and the block written back. Finally block $[k,j]$ itself is written back. There is a benefit in that it is only necessary to access the part of the matrix which can have pivot swaps. If the cost of actually swapping rows is ignored, then in the worst case the additional cost of these pivot updates arising from factoring the $k$th block column is

$$p(p-k+1)(tget + tput) \ . \tag{5.26}$$

Secondly it is necessary to identify pivot elements through full column searches. When the leading diagonal block is factorised it is necessary to search down a full column, but with overall performance considerations suggesting the use of a large square block, it seems unlikely that there will be sufficient space to accommodate a full block column in memory. The block level factorisation may be performed at the granularity of sub-block columns, keeping the block being factored in memory but below it only the particular sub-block columns needed to allow full column pivot searches for a single step. While pivot swaps may be applied to the whole block column after completion of the block level factorisation as described before, it is necessary to write the pivot row into the active block during the factorisation. The requirement is then for each sub-block step to read blocks below the diagonal by column but also read blocks by row to allow update of rows within the active block. A simple approach is to perform the block level factorisation at the granularity of sub-blocks; obtaining the data required for each step of the block level factorisation by reading full blocks and update pivot rows before discarding the portion of each block not required in the coming step. As elsewhere the overall matrix is assumed to be of size $n^2$, and partitioned into blocks of size $b = n/p$. As a starting point it is assumed that a full column sub-block

of width $w$ is to fit into the same space as a single block. Then the sub-block width is determined;

$$w = \frac{b^2}{n} = \frac{n}{p^2} \ .$$

For each such sub-block column in a block it is necessary to read all blocks below the diagonal, with cost $(p-k)tget$. The extra cost due to searching for pivots down full columns in the block level factorisation corresponding to the $k$th block column is then

$$\begin{aligned} & \frac{b}{w}(p-k)tget \\ = \ & p(p-k)tget \end{aligned}$$

There are possible ways of optimising these operations. If the matrix is distributed over a number of nodes it is possible to achieve parallelism in the global pivot swapping phase by performing swaps locally. For instance if the matrix were distributed by block column it might be possible to perform pivot swaps for all block columns concurrently. One possible way in which the cost of implementing the pivot searches may be reduced is to actually organise blocks on secondary storage by sub-blocks. Data on secondary storage is only ever written by complete block, but it is possible to reduce the total read cost, even if shorter reads will not be at full disk bandwidth. In the following analysis however, it is simply assumed that all pivoting operations are performed by a single process with data accessed by full blocks. Then the total additional cost due to pivoting in the $k$th block column is obtained from (5.26) and (5.27).

$$\begin{aligned} Tpiv(k)_1 &= p(p-k)tget + p(p-k+1)(tget + tput) \\ &= p(2p-2k+1)tget + p(p-k+1)tput \ . \end{aligned} \tag{5.27}$$

In order to implement an LU factorisation it is also necessary to define task and synchronisation structures. A starting point is to partition the work so that each task writes a single block as before. A task dependency graph for such a computation is shown in figure 5.30. In this arrangement the task which computes a block below the diagonal must access the block above it on the diagonal to compute the final result. For instance the task computing the block $\mathsf{L}[i,k]$ first computes the intermediate result

$$\mathsf{tmp} = \mathsf{A}[i,k] - \sum_{j=1}^{k} \mathsf{L}[i,j]\mathsf{U}[j,k]$$

and then solves

$$\mathsf{L}[i,k]\,\mathsf{U}[k,k] = \mathsf{tmp}$$

to obtain the desired result.

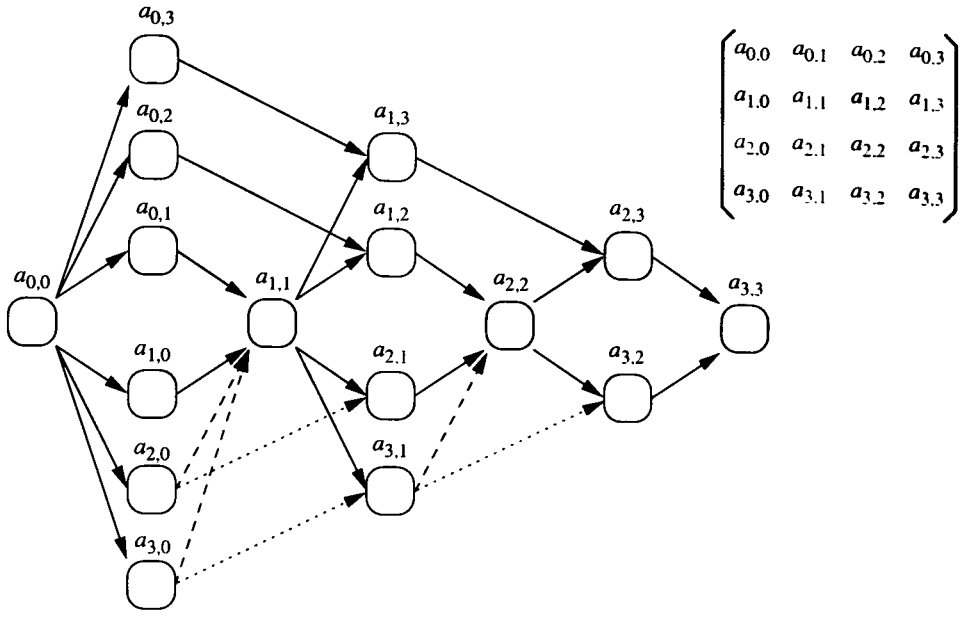



Figure 5.30: Task dependencies for blocked LU factorisation. If pivoting is performed, the dashed arcs are present but not the dotted arcs. If pivoting is not performed the opposite is true.

The task which computes a block on the diagonal must perform first a similar inner product then a block factorisation. It should be possible to perform the first part of the two tasks in parallel. One way this can conveniently be achieved is to split each such task into two and have the first write an intermediate value to be used in the second. It is necessary to introduce some extra indicator to discriminate between the different tasks writing the same block, though for an initial implementation it is convenient to have the task performing the block factorisation also perform the triangular solves to update the blocks below, in addition if required to the global pivot swaps. The example computation then has the task graph shown in figure 5.31 A simple implementation approach is to employ a sequence of bags of tasks to place barrier synchronisation at the indicated points.

An implementation of the computation without pivoting has been performed using the same infrastructure as used before for matrix multiplication and Cholesky factorisation. Also an analysis of the performance has been made to compare with the experimental results. As before the full analysis is contained in an appendix, appendix B, but the principal results are quoted here.

The total extra cost due to pivoting is in all cases

$$Tpiv_1 = p^3 tget + \frac{p^2}{2}(p+1)tput \tag{5.28}$$

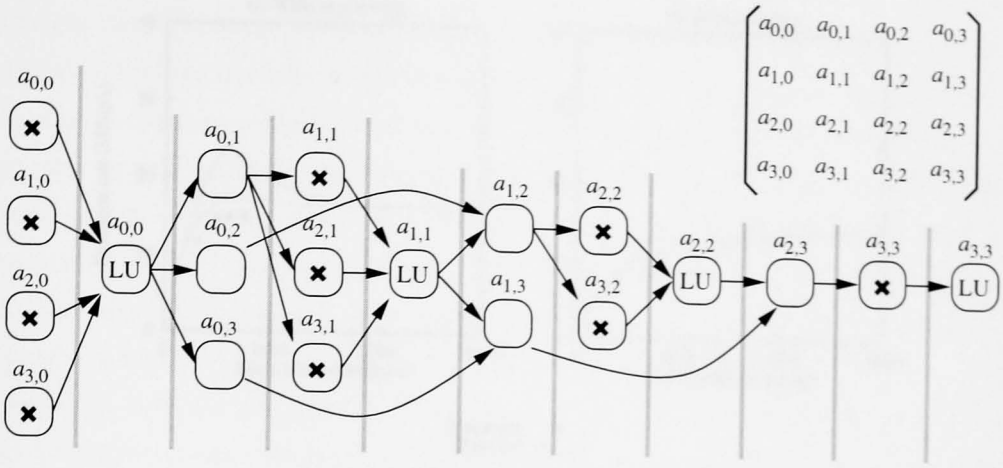



Figure 5.31: Blocked Crout LU factorisation using bags of tasks.

The single slave time and bounds on minimum parallel runtime without pivoting are as follows.

$$TNOPIV_1 = \frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput + \frac{p}{6}(p-1)(2p-1)(tmult+tsub) + p(p-1)tsolve + ptlu \quad (5.29)$$

$$\text{lwr}\{TNOPIV_\infty\} = \max\left\{\frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput + \frac{p}{2}(p-1)tsolve + ptlu\ ,\ \frac{p}{2}(5p-1)tget + \frac{p}{2}(3p+1)tput + (p-1)^2(tmult+tsub) + \frac{1}{2}(p-1)(p+2)tsolve + ptlu\right\} \quad (5.30)$$

$$\text{upr}\{TNOPIV_\infty\} = \frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput + (p-1)^2(tmult+tsub) + \frac{1}{2}(p-1)(p+2)tsolve + ptlu \quad (5.31)$$

Figure 5.32 shows the best measured performance of the Crout LU factorisation of a $3000^2$ matrix without pivoting in the **HP** configuration. Also shown is the expected performance for this case and also for the same computation with pivoting included.

While the performance is not high, it is clear that the I/O cost is considerably greater in the LU factorisation than in the simpler Cholesky factorisation and the hardware parameters of the experimental environment are modest. While possible optimisations to the pivoting operations have been suggested

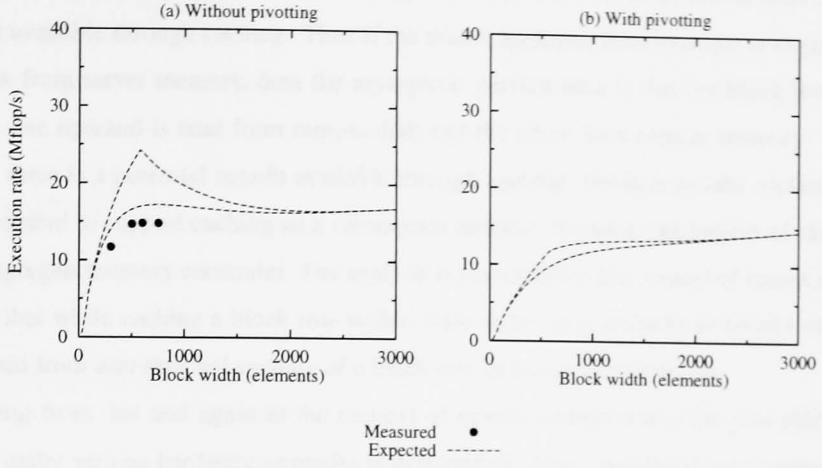



Figure 5.32: Performance of LU factorisation for $3000^2$ matrix in the **HP** configuration.

already, it is also possible to seek improvements in the performance of the basic algorithm. One possible improvement is to move the triangular solves which compute the final results for blocks below the diagonal from the sequential task computing the block on the diagonal to the next parallel step. Each triangular solve might be computed by the task which is responsible for writing the intermediate result for the block just to the right. Then the solves could be executed in parallel rather than serially. Another line of development might be to implement a left looking factorisation where a whole block column is computed in each step. To coordinate concurrent execution of tasks computing blocks in the same block column both above and below the diagonal it is necessary to employ some additional synchronisation mechanism, such as the array of flags used before, but the number of barrier points is reduced. It is anticipated that increasing the problem size should yield a higher potential degree of parallelism. Currently it is shown that the computation can be adapted to a bag of tasks based structure to facilitate use of changing resources. However, further investigation is required to fully assess the potential performance.

## 5.8 Summary

The results of the previous chapter already suggest that the bag of tasks based structuring technique employed in this work provides an effective way to exploit the changing parallel resources of a NOW. However it remains to consider any cost inherent in choosing such a structure instead of another, which might be less dynamic. First issues specific to the bag of tasks based structure are considered, beginning with the performance of alternative synchronisation structures for the Cholesky factorisation example.

Subsequently the issue of problem size scaling is considered and it is shown that there is an asymptotic performance for each of the matrix computations which is equal to the performance of the block

level matrix multiplication operation when the operands are accessed from shared store, but reflecting any benefit available through caching. Thus if the matrix multiplication example is organised to reuse a block row from server memory, then the asymptotic performance is that for block level multiplication where one operand is read from remote disk and the other from remote memory. This analysis shows that there is a potential benefit available through caching, but fails to take account of the extra memory required to support caching so a subsequent analysis considers the benefit of caching under a constant aggregate memory constraint. The analysis is performed in the context of matrix multiplication and shows that while caching a block row within slave memory is unlikely to be of benefit there is a potential gain from user directed caching of a block row in server level memory.

Following from this and again in the context of matrix multiplication the potential performance achievable under various hardware upgrades is determined. Also considered is the appropriateness of the recovery cost and the likely load on the bag of tasks. Consideration is then given to using the well known technique of overlapping communications to increase exploitation of the hardware resources. In the bag of tasks structure two points are identified where such a technique can be applied. At the server level it is possible to overlap network and disk transfers by employing double buffering and at slave level it is possible to overlap communications with computation by employing multithreading.

Next a performance comparison is made between the bag of tasks structure and a data parallel alternative for the example case of matrix multiplication. Each is tuned to a similar degree though there remains a possibility for further tuning through caching on local disk space at slave level. However it appears so far that the bag of tasks structure can be competitive in terms of potential performance and certainly has advantages when the number of slaves participating in a computation changes during the course of that computation.

Finally some consideration is given to a another matrix computation, namely LU factorisation. This example is interesting because it does not permit an overall decomposition so obviously suited to a bag of tasks based structure. It is reasonable to expect this situation to arise in many other example computations.

# Chapter 6

# Conclusions

A NOW is commonly perceived as embodying in its surplus capacity an alternative to a dedicated parallel machine. There is a range of options open to an organisation wishing to exploit this alternative. One example would be to administer a NOW as a dedicated high performance resource overnight. In this case the NOW is certainly prone to partial failures where a tightly coupled machine may exhibit an all or nothing failure pattern. An alternative allows workstations in the NOW to be used for either sequential or parallel computations at any time leading to a highly varied job mix including both interactive and noninteractive jobs. In either case the distinguishing feature is that the resources available to a parallel computation are changing. This work has been directed to the exploitation of changing resources in any NOW configuration. The resource changes addressed here are addition or removal of a workstation to a computation running in a NOW. These changes may be notified to the computation in advance or not. In the latter case it is not essential for the computation to have taken some prior precautionary action; rather it is sufficient for the computation to simply reconfigure. In practice it is probably important to optimise separately for notified and unnotified resource changes.

It turns out however that existing work tends mostly to address various parts of the overall problem. A large amount of work exists on checkpointing and replication techniques which generally address a transient node failure. Several distributed operating systems have addressed the problem of migrating processes from one machine to another. The main problem is ensuring correct interaction between process and system, and thus indirectly also that between processes and thus was originally addressed in the context of completely custom operating systems. In a multiprocessor with truly shared memory the problem does not arise since all communication between a process and outside is through the same shared memory wherever the process runs. Recent work has shown that kernel modifications can be avoided and indeed if all interaction is through a message passing library it is possible to localise considerably the support needed for migration. However no change at the system level can enable a given computation to tolerate loss of a node, notified or otherwise, or exploit an extra node. The computation

itself needs to be structured specifically to support this. Many computations are irregular by nature so that support for dynamic load balancing is inherent to their solution, and elsewhere dynamic frameworks have been developed for computations which would typically be partitioned in a static way, either to support use of a heterogeneous collection of machines or to support withdrawal or removal of machines. One such framework is referred to here as a bag of tasks, a term used often in Linda based publications, though it is synonymous with work pile, task farm, master slave and many others. Generally the resource changes are notified so that extra provision is required to tolerate failures.

Job level scheduling can address the problem of exploiting changing resources by selecting those jobs which will best use the available resources. However, such an approach relies on the jobs currently active exhibiting a sufficient range of resource requirements. It is suggested that there is a place for at least some of the jobs submitted being able locally to fully exploit changing resources, expanding to use more facilities or contracting to use fewer facilities. CALYPSO has addressed the overall problem of supporting use of changing resources by creating a pre-compiler which restructures annotated user code automatically and uses a bag of tasks type approach to execute segments in parallel. The work here also addresses the overall problem of computing on changing resources, but provides library level mechanisms which the user then exploits as required. The approach described in this work is also similar to FT-Linda and Plinda in exploiting a bag of tasks structure. However in general when compared to existing systems the work here addresses a wider range of problems.

Any machine has a restricted physical memory size which ultimately limits the size of problem that can be attempted. An example of a problem where this issue has been reported is a large matrix factorisation. There is therefore a long history of work on out of core structuring techniques for such problems where the problem data is based on secondary storage and physical memory serves only as cache space for this data. More recently interest has mainly focussed on parallel implementations often favouring a data parallel structure for implementation on machines with a large degree of parallelism. This work takes an unusual approach by organising out of core computations in various ways using a number of bags of tasks in order to investigate the potential for performing such computations on the changing resources found in a NOW.

A disk based shared address space is constructed using a distributed object store. Accesses to these shared objects are controlled by the use of atomic actions, employed both within object operations and within application process code. A commit protocol ensures consistency of distributed updates. A fault-tolerant bag of tasks is implemented using a recoverable queue, used previously in queued transaction work. While the basic organisation provides support for simple parallel loops, a number of simple extensions which support more complex parallel structures are described. A problem can be partitioned into phases which can each be implemented using a separate bag of tasks. Since the return status of the dequeue operation distinguishes the case where the queue is truly empty from that where all entries are locked by other processes, it is easy to create the effect of a barrier synchronisation

between computation phases. It is necessary to arrange for slave processes to move on to the next queue only when the current one is empty. An alternative approach is to employ synchronisation variables directly in the computation. It is necessary to ensure that such synchronisation behaves correctly in the event of process recovery. In the case of Cholesky factorisation for example it is convenient to associate a recoverable flag with each block of the result matrix and make updates within the operations of the matrix object. It is also possible to define a hierarchy of bags of tasks if there is a need to encourage each slave to follow a defined sequence of tasks, while not compromising the recoverability. It is claimed that recoverable bags of tasks together with atomic actions make convenient building blocks with which to construct dynamic parallel structures.

Three specific computations have been implemented in three different NOW configurations. Two dense matrix computations, matrix multiplication and Cholesky factorisation, are data intensive. The third, ray tracing, is much less so but enables a comparison with other infrastructures. For a small scale ray tracing computation the fault-tolerance seems a luxury when the performance is compared with that of a commercial grade implementation which is not fault-tolerant. However, for an example of realistic scale the fault-tolerance is not only clearly going to be cheap but also necessary.

The computations have been modelled so as to verify the observed performance, and in the case of the matrix computations to allow the effect of configuration changes to be predicted. The modelling approach is empirical in relying on benchmark measurements of the lower level operations which the applications are built from. These are all at the level of blocks, i.e submatrices, which are accessed as units and include for instance read and write block, multiply two blocks etc. Following calibration of the models, measurement of the performance of complete examples both validates the modelling process and shows that worthwhile absolute performance can be achieved even in the hardware configurations available. For the defined program structures the additional cost imposed by the fault-tolerance mechanism is seen to be small provided the granularity is large.

Following this initial work the next step is to use the models to assess the potential for the structuring approach. First a number of alternative organisations of the same basic algorithm for Cholesky factorisation are compared. These organisations differ only in the approach to synchronisation; in two cases employing sequences of bags of tasks and in another employing separate synchronisation flags manipulated in the operations of the result matrix object. In the former case the model suggests that a structure employing a single queue and additional synchronisation integrated into the result matrix can offer better performance than either of two alternatives which employ multiple queue steps to avoid using the additional synchronisation in the result. Experimental measurements suggest that this higher performance can be realised in practice.

A simple analysis of the scaling properties shows that there is an asymptotic execution rate for a given configuration which is approached as the problem size is increased. This asymptotic rate is equal to the rate at which block level matrix multiplication operations can be performed when the operands are

fetched from store. If the computation is organised to reuse data and so benefit from caching then this asymptotic rate increases proportionately. However the value of data caching is better considered in the context of the overall memory requirement. When compared under a constant memory constraint, for matrix multiplication it appears that caching data in slave memory is not worthwhile, but that a benefit may be gained through user directed block row caching at server memory level.

A practical question arises regarding the usefulness of scaling computation size arbitrarily. There is some evidence [48] however which suggests that dense linear algebra computations of rather larger scale than those considered in this work are performed in practice. While this work makes only a start and tackles with any thoroughness only simple examples it may be that if a way of performing these large scale computations is readily available then increased use will be found.

Using the performance model it is possible to predict the potential performance in the event of some upgrade of hardware configuration. In so doing it is important to verify that the computation granularity remains appropriate, so that recovery time is not too large and the load on a bag of tasks is not too high. Intuitively these requirements can be satisfied if the degree of parallelism is modest and the granularity large. However in the case of matrix multiplication at least there appears to be ample scope for gaining benefit through parallelism while doing so in practice.

Overlapping communications is a standard technique used to exploit more fully any hardware configuration. In a bag of tasks structure this may be achieved at two levels. At server level network and disk transfers can be overlapped through double buffering. At slave level the use of multithreading can overlap communications with computation. Each has some impact on the total memory requirement, ultimately reducing the granularity and so increasing the load on a bag of tasks. Again in the case of matrix multiplication there appears to be scope in practice for exploiting such overlapping techniques for suitably large computations.

After modelling simple hardware upgrades and the potential for overlapping communications, a comparison is made between the bag of tasks structure and a data parallel structure in order to assess the cost of structuring to cope with changing resources. This analysis is performed only in terms of maximum achievable performance and only in the case of matrix multiplication so there remains scope for developing this comparison not just in this example but also for other applications. When both structures are tuned to the same degree, specifically employing both double buffering and multithreading techniques, the evidence so far suggests that the bag of tasks structure can be competitive. Furthermore when operating in an environment of changing resources there are clear advantages to the bag of tasks structure.

Finally brief consideration is given to a new example application, LU factorisation. The significance is that the pivoting required in LU factorisation suggests a rather finer granularity than can be achieved in either matrix multiplication or Cholesky factorisation and so the centralised bag of tasks might be expected to be a bottleneck. It is shown that an organisation based on a bag of tasks type approach is

possible though further work is needed to evaluate the performance potential.

Overall it is claimed that investigation of computation structures which can make full use of changing resources is worthwhile and the study here has made a contribution in this area.

## 6.1 Further Work

The Arjuna system has evolved into a second generation which can employ industry standard transport mechanisms and multithreading. There is then scope for porting the existing applications and optimising servers. Such optimisation in the queue may broaden its range of applications. There is also scope for demonstrating the fault recovery experimentally. The objects used here have been developed in response to application needs. Development of further applications might allow identification of some useful set of tools.

For the type of applications studied in the constrained environment of a NOW the requirement for complex commit protocols to implement nonblocking agreement can be avoided and has permitted development of useful dynamic parallel structures. With appropriate guarantees regarding atomicity it may be that such dynamic structures can find wider application.

As described earlier, structuring a computation using a bag of tasks achieves a form of adaptive parallelism. If the task granularity is small enough it is sufficient to simply kill the slave process executing on a particular machine to cause migration from that machine. A mechanism which allows a slave to save its state to enable cheaper restart would allow the task granularity to be increased.

The analysis of alternate structures for matrix multiplication remains incomplete. In the data parallel structure each slave reuses the same block row of the second input matrix. It is possible to organise the bag of tasks structure also so that each slave reuses a particular block row a number of times. Analysis of the bag of tasks structure suggested that caching a block row in slave memory would not be beneficial because this forces reduction of the block size. However, if slave machines have large local disk space, it would be possible to cache a block row there in either computation structure. On the practical side, it remains necessary to verify the operation of a two level bag of tasks structure to arbitrate separately between block rows and blocks within each block row.

So far the structuring approach has been demonstrated in only two applications where an out of core approach is necessary. It would be interesting to consider structuring other matrix computations in this way and also to consider other data intensive computations. In the case of LU factorisation it is seen that even preliminary investigation highlights areas for further study. The first question concerns the possibility of using the dynamic structure employed here, but there is also the issue of performance particularly when compared with alternative less dynamic approaches.

# Appendix A

# Cholesky Factorisation Performance

## A.1 single-bag

The approach to modelling the **single-bag** organisation is to think of the computation as if there were a barrier at the point of output of a block on the diagonal and at the completion of each block column. Assuming full parallelism is reached, the $2p-1$ steps are as follows.

- In the first step all processes read the appropriate block of the input matrix, $a_{i,j}$ and in addition, the first process computes Cholesky factor of block $a_{0,0}$ and writes it out. This is referred to as the initialisation step.
- In the $p-1$ even numbered steps starting with the second processes computing blocks in the same block column as the diagonal block output in the previous step read that diagonal block, perform the required solve and output the result block. These are referred to as solve steps.
- In the remaining $p-1$ all processes which have not yet output a result, read blocks written during the previous step (one if on diagonal or two otherwise) and perform a multiplication and subtraction. In addition, one process computes Cholesky factor and writes out the result. These are referred to as factorisation steps.

### A.1.1 Single Slave Time

Since there is no parallelism available, the duration of the first step is always

$$Tinit_1 = \frac{p}{2}(p+1)tget + tchol + tput \tag{A.1}$$

There are $p-1$ solve steps for columns $j = 1$ to $p-1$. The step corresponding to column $j$ entails

$p-j$ block solves. The serial time for the $j$th such step is

$$Tsolve(j)_1 = (p-j)(tget + tsolve + tput) \tag{A.2}$$

and the total for all these steps

$$\begin{aligned} Tsolve_1 &= \sum_{j=1}^{p-1} Tsolve(j)_1 \\ &= \sum_{j=1}^{p-1} (p-j)(tget + tsolve + tput) \\ &= \frac{p}{2}(p-1)(tget + tsolve + tput) \end{aligned} \tag{A.3}$$

There are also $p-1$ factorisation steps, but for columns $j = 2$ to $p$. In each of these steps a block multiplication and subtraction is performed for each active block and also a single Cholesky factorisation.

In the step corresponding to column $j$ there are a total of $\frac{1}{2}(p-j+1)(p-j+2)$ blocks of the output yet to be written. Of these $(p-j+1)$ lie on the diagonal and therefore require only a single block read during the step. The remaining $\frac{1}{2}(p-j)(p-j+1)$ require two block reads. The serial time for this step is

$$\begin{aligned} Tfact(j)_1 &= (p+1-j)^2 tget \\ &\quad + \frac{1}{2}(p+1-j)(p+2-j)(tmult + tsub) \\ &\quad + tchol + tput \end{aligned} \tag{A.4}$$

and the total for all these steps

$$\begin{aligned} Tfact_1 &= \sum_{j=2}^{p} Tfact(j)_1 \\ &= \sum_{j=2}^{p} \Big\{ (p+1-j)^2 tget + \\ &\quad + \frac{1}{2}(p+1-j)(p+2-j)(tmult + tsub) \\ &\quad + tchol + tput \Big\} \\ &= \frac{p}{6}(p-1)(p-2)tget + \frac{p}{6}(p^2-1)(tmult + tsub) \\ &\quad + (p-1)(tchol + tput) \end{aligned} \tag{A.5}$$

By summing these three components (A.1), (A.5), (A.3) it is possible to derive an estimate of the

single slave time

$$\begin{aligned} T_1 &= Tinit_1 + Tsolve_1 + Tfact_1 \\ &= \frac{p}{2}(p+1)tget + tchol + tput + \frac{p}{2}(p-1)(tget + tsolve + tput) \\ &\quad + \frac{p}{6}(p-1)(p-2)tget + \frac{p}{6}(p^2-1)(tmult + tsub) \\ &\quad + (p-1)(tchol + tput) \\ &= \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{6}(p^2-1)(tmult + tsub) \\ &\quad + \frac{p}{2}(p-1)tsolve + ptchol + \frac{p}{2}(p+1)tput \end{aligned} \tag{A.6}$$

### A.1.2 Minimum Parallel Time

**Lower Bound**

One lower bound on the minimum parallel time is set by hardware bandwidths.

$$Tcomm = \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{2}(p+1)tput$$

Another is determined by the longest duration task. This may be either the last task which computes the bottom right diagonal block or the previous task which computes the block just to the left in the same block row.

$$T(p,p-1) = 2ptget + (p-1)\,(tmult + tsub) + tsolve + tput$$

and

$$T(p,p) = ptget + (p-1)(tmult + tsub) + tchol + tput \ .$$

The latter is greatest if

$$tchol > ptget + tsolve \ ,$$

but also trivially for $p = 1$ since it is then the only task.

The required lower bound is then the maximum of these separate bounds.

$$\text{lwr}\{T_\infty\} = \max\{Tcomm, T(p,p-1), T(p,p)\} \ .$$

An alternative lower bound may be defined based on the average task length for use when it is not obvious which task is actually longest.

$$\text{lwr}\{T_\infty\} = \max\left\{Tcomm, \frac{T_1}{\frac{p}{2}(p+1)}\right\} \ . \tag{A.7}$$

**Upper Bound**

An upper bound on the minimum parallel time may be obtained by pretending that there is a barrier after each of the computation steps described above and then summing the upper bound on the minimum parallel time for each step.

The time taken to compute the initialisation step is $Tinit_1$ (A.1).

An upper bound on the minimum parallel time to compute the solve step corresponding to column $j$ is obtained from (A.2).

$$\text{upr}\{Tsolve(j)_\infty\} = (p-j)(tget + tput) + tsolve \ .$$

An upper bound on the minimum time to compute all such solve steps is

$$\begin{aligned}
\text{upr}\{Tsolve_\infty\} &= \sum_{j=1}^{p-1} \text{upr}\{Tsolve(j)_\infty\} \\
&= \sum_{j=1}^{p-1} \{(p-j)(tget + tput) + tsolve\} \\
&= \frac{p}{2}(p-1)(tget + tput) + (p-1)tsolve \ . \qquad \text{(A.8)}
\end{aligned}$$

An upper bound on the minimum parallel time to compute the factorisation step corresponding to column $j$ is obtained from (A.4).

$$\text{upr}\{Tfact(j)_\infty\} = (p+1-j)^2 tget + tmult + tsub + tchol + tput$$

An upper bound on the minimum time to compute all such factorisation steps is

$$\begin{aligned}
\text{upr}\{Tfact_\infty\} &= \sum_{j=2}^{p} \text{upr}\{Tfact(j)_\infty\} \\
&= \sum_{j=2}^{p} \{(p+1-j)^2 tget + tmult + tsub + tchol + tput\} \\
&= \frac{p}{6}(p-1)(2p-1)tget \\
&\quad + (p-1)(tmult + tsub + tchol + tput) \ . \qquad \text{(A.9)}
\end{aligned}$$

Finally, the upper bound on $T_\infty$ is obtained by summing values from (A.1), (A.9), (A.8).

$$\begin{aligned}
\text{upr}\{T_\infty\} &= Tinit_1 + \text{upr}\{Tsolve_\infty\} + \text{upr}\{Tfact_\infty\} \\
&= \frac{p}{2}(p+1)tget + tchol + tput \\
&\quad + \frac{p}{2}(p-1)(tget + tput) + (p-1)tsolve
\end{aligned}$$

$$
\begin{aligned}
& + \frac{p}{6}(p-1)(2p-1)tget \\
& + (p-1)(tmult + tsub + tchol + tput) \\
= \; & \frac{p}{6}(p+1)(2p+1)tget + (p-1)(tmult + tsub + tsolve) \\
& + ptchol + \frac{p}{2}(p+1)tput
\end{aligned} \tag{A.10}
$$

## A.2 multi-step(1)

### A.2.1 Single Slave Time

For the $j$th diagonal block the computation time is $T(j,j)_1$. In addition to the original block at this location, it is necessary to read $j-1$ blocks, perform one block multiplication and one block subtraction for each of these blocks and finally factor the result write the resulting factor back.

$$T(j,j)_1 = jtget + (j-1)(tmult + tsub) + tchol + tput \tag{A.11}$$

The remainder of the work on this block column entails computing the blocks below the diagonal. Computation of the $i,j$th block entails $2(j-1)$ reads in addition to that for the original block, $j-1$ multiplications and subtractions, another read and block solve and writing the result.

$$T(i,j)_1 = 2jtget + (j-1)(tmult + tsub) + tsolve + tput \tag{A.12}$$

The time taken by a single slave to compute the $j$th column is then

$$T(j,j)_1 + \sum_{i=j+1}^{p} T(i,j)_1 \; .$$

Finally summing for all columns gives the time taken by a single slave to complete the whole computation.

$$
\begin{aligned}
T_1 &= \sum_{j=1}^{p} \left\{ T(j,j)_1 + \sum_{i=j+1}^{p} T(i,j)_1 \right\} \\
&= \sum_{j=1}^{p} \left\{ jtget + (j-1)(tmult + tsub) + tchol + tput \right. \\
&\qquad \left. + \sum_{i=j+1}^{p} \{2jtget + (j-1)(tmult + tsub) + tsolve + tput\} \right\} \\
&= \sum_{j=1}^{p} \{((2p+1)j - 2j^2)tget
\end{aligned}
$$

$$
\begin{aligned}
& + ((p+2)j - j^2 - p - 1)(tmult + tsub) \\
& + (p-j)tsolve + tchol + (p-j+1)tput\} \\
= \; & \frac{p}{6}(p+1)(2p+1)(3-2)tget \\
& + \frac{p}{6}(p+1)(3p+6-2p-1-6)(tmult + tsub) \\
& + \frac{p}{2}(2p-p-1)tsolve + ptchol + \frac{p}{2}(p+1)tput \\
= \; & \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{6}(p^2-1)(tmult + tsub) \\
& + \frac{p}{2}(p-1)tsolve + ptchol + \frac{p}{2}(p+1)tput
\end{aligned} \tag{A.13}
$$

### A.2.2 Minimum Parallel Time

**Lower Bound**

The computation consists of $2p-1$ steps.

- The first and subsequent odd numbered steps, referred to as factorisation steps, are serial computations which output a single block on the diagonal. The time taken to perform this serial computation for the $j$th column is $T(j,j)_1$ in (A.11).
- The second and subsequent even numbered steps, referred to as solve steps, are parallel computations which output all blocks below the diagonal in the same block column as the block output in the previous step.

The $j$th solve step entails computing all blocks in the $j$th column below the block on the diagonal. The time taken to compute one such block is $T(i,j)_1$ in (A.12). A lower bound on the minimum time taken to compute the $j$th solve step is therefore

$$
\mathrm{lwr}\{Tsolve(j)_\infty\} = \max\left\{ \sum_{i=j+1}^{p} \{2jtget + tput\} \;,\; T(i,j)_1 \right\} .
$$

A lower bound to $T_\infty$ may be obtained by summing the cost of the $p$ serial steps and the $p-1$ parallel steps.

$$
\begin{aligned}
\mathrm{lwr}\{T_\infty\} &= \sum_{j=1}^{p} T(j,j)_1 + \sum_{j=1}^{p-1} \mathrm{lwr}\{Tsolve(j)_\infty\} \\
&= \sum_{j=1}^{p} T(j,j)_1 + \sum_{j=1}^{p-1} \max\left\{ \sum_{i=j+1}^{p} \{2jtget + tput\} \;,\; T(i,j)_1 \right\} .
\end{aligned}
$$

In each of the parallel steps which compute the blocks below the diagonal the cost may be dominated either by the total communications or by the computation. For example, in the case of the first block column the cost of each task is $2tget + tsolve + tput$ while in the case of the second block column the

cost is $4tget + 2(tmult + tsub) + tsolve + tput$. However it is possible to simplify this expression and yet still obtain a lower bound on the overall minimum time, though the bound may be not so close.

$$\begin{aligned}
\mathrm{lwr}\{T_\infty\} &= \sum_{j=1}^{p} T(j,j)_1 \\
&\quad + \max\left\{ \sum_{j=1}^{p-1} \sum_{i=j+1}^{p} \{2jtget + tput\} \ , \ \sum_{j=1}^{p-1} T(i,j)_1 \right\} \\
&= \sum_{j=1}^{p} \{jtget + (j-1)(tmult + tsub) + tchol + tput\} \\
&\quad + \max\left\{ \sum_{j=1}^{p-1} \{(p-j)(2jtget + tput)\} \ , \right. \\
&\quad \left. \sum_{j=1}^{p-1} \{2jtget + tput + (j-1)(tmult + tsub) + tsolve\} \right\} \\
&= \frac{p}{2}(p+1)tget + \frac{p}{2}(p-1)(tmult + tsub) + ptchol + ptput \\
&\quad + \max\left\{ \frac{p}{3}(p^2-1)tget + \frac{p}{2}(p-1)tput \ , \right. \\
&\quad p(p-1)tget + \frac{(p-1)}{2}(p-2)(tmult + tsub) \\
&\quad \left. + (p-1)tsolve + (p-1)tput \right\} \\
&= \max\left\{ \frac{p}{6}(p+1)(2p+1)tget + \frac{p}{2}(p-1)(tmult + tsub) \right. \\
&\quad + ptchol + \frac{p}{2}(p+1)tput \ , \\
&\quad \frac{p}{2}(3p-1)tget + (p-1)^2(tmult + tsub) \\
&\quad \left. + (p-1)tsolve + ptchol + (2p-1)tput \right\} \qquad \text{(A.14)}
\end{aligned}$$

As a check if $p = 1$ in this expression, it is seen that

$$\mathrm{lwr}\{T_\infty\} = tget + tchol + tput \ .$$

**Upper Bound**

An upper bound on the minimum time taken to compute the solve step corresponding to the $j$th column is

$$\mathrm{upr}\{Tsolve(j)_\infty\} = \sum_{i=j+1}^{p} \{2jtget + tput\} + (j-1)(tmult + tsub) + tsolve \ .$$

In the same way as for the lower bound, an upper bound to $T_\infty$ may be obtained by summing the

cost of the $p$ serial steps and the $p-1$ parallel steps.

$$\begin{aligned}
\text{upr}\{T_\infty\} &= \sum_{j=1}^{p} T(j,j) + \sum_{j=1}^{p-1} \text{upr}\{Tsolve(j)_\infty\} \\
&= \sum_{j=1}^{p} T(j,j) \\
&\quad + \sum_{j=1}^{p-1} \left\{ \sum_{i=j+1}^{p} \{2jtget + tput\} + (j-1)(tmult + tsub) + tsolve \right\} \\
&= \sum_{j=1}^{p} \{jtget + (j-1)(tmult + tsub) + tchol + tput\} \\
&\quad + \sum_{j=1}^{p-1} (p-j)(2jtget + tput) + (j-1)(tmult + tsub) + tsolve \\
&= \frac{p}{2}(p+1)tget + \frac{p}{2}(p-1)(tmult + tsub) + ptchol + ptput \\
&\quad + \frac{p}{3}(p^2-1)tget + \frac{(p-1)}{2}(p-2)(tmult + tsub) \\
&\quad + (p-1)tsolve + \frac{p}{2}(p-1)tput \\
&= \frac{p}{6}(p+1)(2p+1)tget + (p-1)^2(tmult + tsub) \\
&\quad + ptchol + (p-1)tsolve + \frac{p}{2}(p+1)tput
\end{aligned} \tag{A.15}$$

Again as a check if $p = 1$ in this expression, it is seen that

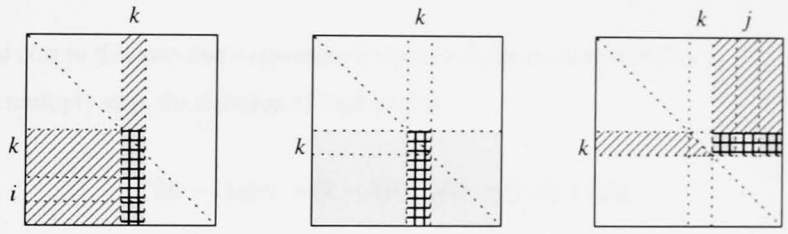
$$\text{upr}\{T_\infty\} = tget + tchol + tput \ .$$

# Appendix B

# LU Factorisation Performance

## B.1 Crout Factorisation

The CROUT organisation computes in each iteration results for a partial block column and row from the diagonal downwards and to the right. Each such iteration comprises three steps. The organisation is illustrated for the $kth$ iteration, corresponding to the $k$th block column, in Figure B.1




(1) multiply (2) factor (3) solve

Figure B.1: Computation steps in Crout LU factorisation.

1. The first step is referred to as a *multiply* step and computes intermediate results for blocks on and below the diagonal. Overall the step performs the computation

$$\mathsf{X}[k:p,k] = \mathsf{A}[k:p,k] - \mathsf{L}[k:p,1:k-1]\mathsf{U}[1:k-1,k]$$

   A single task in this step computes a single block $L[i,k]$ where $i \geq k$.

2. The second step is referred to as a *factor* step as the partial block column $\mathsf{X}[k:p,k]$, computed in the previous step, is factorised. Full column pivoting may be performed during the factorisation, and then during the same step pivot row swaps applied to the whole matrix.

3. The final step writes final result values for all blocks in the current block row but to the right of the diagonal. Clearly there is no need for such a step in the final iteration. The Step computes first

$$\mathsf{X}[k, k+1:p] = \mathsf{A}[k, k+1:p] - \mathsf{U}[k, 1:k-1]\mathsf{L}[1:k-1, k+1:p] \ ,$$

then solves

$$\mathsf{U}[k, k+1:p]\mathsf{L}[k,k] = \mathsf{X}[k, k+1:p]$$

for $\mathsf{U}[k, k+1:p]$. A single task in this step computes a single block $\mathsf{U}[k,j]$ where $k < j \leq p$, without explicitly writing $\mathsf{X}[k,j]$.

### B.1.1 Single Slave Time

Without pivoting the $k$th factorisation step may be accomplished by reading the block on the diagonal, factoring it and writing the result back, then in turn reading each block below the diagonal, solving with the upper half of the block on the diagonal and writing back the result.

$$\begin{aligned} Tfactnp(k)_1 &= tget + tlu + tput + (p-k)(tget + tput + tsolve) \\ &= (p-k+1)(tget + tput) + (p-k)tsolve + tlu \end{aligned} \tag{B.1}$$

The additional cost in this step due to pivoting is derived in the main text (5.27).

In the $k$th multiply step, the duration of each task is

$$(2k-1)tget + (k-1)(tmult + tsub) + tput \ .$$

There are $p-k+1$ such tasks, so the single slave time for the complete step is

$$Tmult(k)_1 = (p-k+1)((2k-1)tget + (k-1)(tmult + tsub) + tput) \ . \tag{B.2}$$

In the $k$th solve step, the duration of each task is

$$2ktget + (k-1)(tmult + tsub) + tsolve + tput \ .$$

There are $p-k$ such tasks, so the single slave time for the complete step is

$$Tsolve(k)_1 = (p-k)(2ktget + (k-1)(tmult + tsub) + tsolve + tput) \ . \tag{B.3}$$

If the computation performs no pivoting the overall single slave time is derived from (B.1), (B.2)

and (B.3).

$$
\begin{aligned}
TNOPIV_1 &= \sum_{k=1}^{p}\{Tfactnp(k)_1\} + \sum_{k=1}^{p}\{Tmult(k)_1\} + \sum_{k=1}^{p-1}\{Tsolve(k)_1\} \\
&= \sum_{k=1}^{p}\Big\{(p-k)(tget+tput) + (p-k-1)tsolve + tlu \\
&\quad + (p-k+1)\left((2k-1)tget + (k-1)(tmult+tsub) + tput\right)\Big\} \\
&\quad + \sum_{k=1}^{p-1}\Big\{(p-k)\left(2ktget + (k-1)(tmult+tsub) + tsolve + tput\right)\Big\} \\
&= \frac{p}{3}(p+1)(p+2)tget + p(p+1)tput \\
&\quad + \frac{p}{6}(p+1)(p-1)(tmult+tsub) + \frac{p}{2}(p-1)tsolve + ptlu \\
&\quad + \frac{p}{3}(p+1)(p-1)tget + \frac{p}{2}(p-1)tput \\
&\quad + \frac{p}{6}(p-1)(p-2)(tmult+tsub) + \frac{p}{2}(p-1)tsolve \\
&= \frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput \\
&\quad + \frac{p}{6}(p-1)(2p-1)(tmult+tsub) + p(p-1)tsolve + ptlu
\end{aligned}
\tag{B.4}
$$

If pivoting is performed during the computation it is necessary to add in the cost of the pivot operations, in (5.27).

$$
\begin{aligned}
TPIV_1 &= TNOPIV_1 + \sum_{k=1}^{p} Tpiv(k)_1 \\
&= \frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput \\
&\quad + \frac{p}{6}(p-1)(2p-1)(tmult+tsub) + p(p-1)tsolve + ptlu \\
&\quad + \sum_{k=1}^{p}\{p(2p-2k+1)tget + p(p-k+1)tput\} \\
&= \frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput \\
&\quad + \frac{p}{6}(p-1)(2p-1)(tmult+tsub) + p(p-1)tsolve + ptlu \\
&\quad + p^3 tget + \frac{p^2}{2}(p+1)tput \\
&= \frac{p}{3}(5p^2+3p+1)tget + \frac{p}{2}(p^2+4p+1)tput \\
&\quad + \frac{p}{6}(p-1)(2p-1)(tmult+tsub) + p(p-1)tsolve + ptlu
\end{aligned}
\tag{B.5}
$$

### B.1.2 Minimum Parallel Time

There are no inter task dependencies within the multiply and solve steps. Bounds on the minimum parallel time of execution of the $k$th multiply step and $k$th solve step are defined in the usual way.

$$\begin{aligned}
\text{lwr}\{Tmult(k)_\infty\} &= \max\{(p-k+1)\,((2k-1)tget + tput)\ , \\
&\quad (2k-1)tget + (k-1)(tmult + tsub) + tput\} \\
\text{upr}\{Tmult(k)_\infty\} &= (p-k+1)\,((2k-1)tget + tput) \\
&\quad + (k-1)(tmult + tsub) \\
\text{lwr}\{Tsolve(k)_\infty\} &= \max\{(p-k)\,(2ktget + tput)\ , \\
&\quad 2ktget + (k-1)(tmult + tsub) \\
&\quad + tsolve + tput\} \\
\text{upr}\{Tsolve(k)_\infty\} &= (p-k)\,(2ktget + tput) \\
&\quad + (k-1)(tmult + tsub) + tsolve
\end{aligned}$$

A lower bound may be obtained by summing appropriate values for all columns.

$$\begin{aligned}
TNOPIV_\infty \geq & \sum_{k=1}^{p} \Big\{ Tfactnp(k)_1 + \text{lwr}\{Tmult(k)_\infty\} \\
& + \text{lwr}\{Tsolve(k)_\infty\} \Big\} \\
= & \sum_{k=1}^{p} \Big\{ (p-k+1)(tget + tput) + (p-k)tsolve + tlu \\
& + \max\{(p-k+1)\,((2k-1)tget + tput)\ , \\
& (2k-1)tget + (k-1)(tmult + tsub) + tput\} \Big\} \\
& + \sum_{k=1}^{p-1} \Big\{ \max\{(p-k)\,(2ktget + tput)\ , \\
& 2ktget + (k-1)(tmult + tsub) + tsolve + tput\} \Big\}
\end{aligned}$$

It is possible to ease computation by relaxing the bound slightly.

$$\begin{aligned}
\text{lwr}\{TNOPIV_\infty\} = & \max\Big\{ \sum_{k=1}^{p} \{2(p-k+1)\,(ktget + tput) \\
& + (p-k)tsolve + tlu\} + \sum_{k=1}^{p-1} (p-k)\,(2ktget + tput)\ , \\
& \sum_{k=1}^{p} \{(p+k)tget + (p-k+2)tput
\end{aligned}$$

$$
\begin{aligned}
& +(k-1)(tmult+tsub)+(p-k)tsolve\} \\
& \sum_{k=1}^{p-1}\{2ktget+(k-1)(tmult+tsub)+tsolve+tput\}\Big\} \\
= \ & \max\Big\{\frac{p}{3}(p+1)(p+2)tget+p(p+1)tput \\
& +\frac{p}{2}(p-1)tsolve+ptlu \\
& \frac{p}{3}(p+1)(p-1)tget+\frac{p}{2}(p-1)tput\ , \\
& \frac{p}{2}(3p+1)tget+\frac{p}{2}(p+3)tput \\
& +\frac{p}{2}(p-1)(tmult+tsub)+\frac{p}{2}(p-1)tsolve \\
& +p(p-1)tget+\frac{1}{2}(p-1)(p-2)(tmult+tsub) \\
& +(p-1)tsolve+(p-1)tput\Big\} \\
= \ & \max\Big\{\frac{p}{3}(p+1)(2p+1)tget+\frac{p}{2}(3p+1)tput \\
& +\frac{p}{2}(p-1)tsolve+ptlu\ , \\
& \frac{p}{2}(5p-1)tget+\frac{p}{2}(3p+1)tput \\
& +(p-1)^2(tmult+tsub)+\frac{1}{2}(p-1)(p+2)tsolve+ptlu\Big\} \qquad \text{(B.6)}
\end{aligned}
$$

If the computation performs pivoting, it is necessary to add in the cost of the pivoting operations, from (5.27).

$$
\begin{aligned}
\mathrm{lwr}\{TPIV_\infty\} &= \mathrm{lwr}\{TNOPIV_\infty\}+\sum_{k=1}^{p}Tpiv(k)_1 \\
&= \max\Big\{\frac{p}{3}(p+1)(2p+1)tget+\frac{p}{2}(3p+1)tput \\
&\quad +\frac{p}{2}(p-1)tsolve+ptlu\ , \\
&\quad \frac{p}{2}(5p-1)tget+\frac{p}{2}(3p+1)tput \\
&\quad +(p-1)^2(tmult+tsub)+\frac{1}{2}(p-1)(p+2)tsolve+ptlu\Big\} \\
&\quad +\sum_{k=1}^{p}\{p(2p-2k+1)tget+p(p-k+1)tput\} \\
&= \max\Big\{\frac{p}{3}(p+1)(2p+1)tget+\frac{p}{2}(3p+1)tput \\
&\quad +\frac{p}{2}(p-1)tsolve+ptlu\ , \\
&\quad \frac{p}{2}(5p-1)tget+\frac{p}{2}(3p+1)tput \\
&\quad +(p-1)^2(tmult+tsub)+\frac{1}{2}(p-1)(p+2)tsolve+ptlu\Big\} \\
&\quad +p^3tget+\frac{p^2}{2}(p+1)tput
\end{aligned}
$$

$$
\begin{aligned}
&= \max\Big\{\frac{p}{3}(5p^2+3p+1)tget+\frac{p}{2}(p^2+4p+1)tput \\
&\quad +\frac{p}{2}(p-1)tsolve+ptlu\ , \\
&\quad \frac{p}{2}(2p^2+5p-1)tget+\frac{p}{2}(p^2+4p+1)tput \\
&\quad +(p-1)^2(tmult+tsub)+\frac{1}{2}(p-1)(p+2)tsolve+ptlu\Big\} \qquad \text{(B.7)}
\end{aligned}
$$

An upper bound may be defined in a similar way.

$$
\begin{aligned}
\text{upr}\{TNOPIV_\infty\} &= \sum_{k=1}^{p}\Big\{Tfactnp(k)_1+\text{upr}\{Tmult(k)_\infty\}\Big\} \\
&\quad +\sum_{k=1}^{p-1}\Big\{\text{upr}\{Tsolve(k)_\infty\}\Big\} \\
&= \sum_{k=1}^{p}\Big\{(p-k+1)(tget+tput)+(p-k)tsolve+tlu \\
&\quad +(p-k+1)\left((2k-1)tget+tput\right)+(k-1)(tmult+tsub)\Big\} \\
&\quad +\sum_{k=1}^{p-1}\Big\{(p-k)\left(2ktget+tput\right)+(k-1)(tmult+tsub)+tsolve\Big\} \\
&= \sum_{k=1}^{p}\{2k(p-k+1)tget+2(p-k+1)tput \\
&\quad +(k-1)(tmult+tsub)+(p-k)tsolve+tlu\} \\
&\quad +\sum_{k=1}^{p-1}\{(p-k)(2ktget+tput)+(k-1)(tmult+tsub)+tsolve\} \\
&= \frac{p}{3}(p+1)(p+2)tget+p(p+1)tput \\
&\quad +\frac{p}{2}(p-1)(tmult+tsub)+\frac{p}{2}(p-1)tsolve+ptlu \\
&\quad +\frac{p}{3}(p+1)(p-1)tget+\frac{p}{2}(p-1)tput \\
&\quad +\frac{1}{2}(p-1)(p-2)+(p-1)tsolve \\
&= \frac{p}{3}(p+1)(2p+1)tget+\frac{p}{2}(3p+1)tput \\
&\quad +(p-1)^2(tmult+tsub)+\frac{1}{2}(p-1)(p+2)tsolve+ptlu \qquad \text{(B.8)}
\end{aligned}
$$

If the computation performs pivoting, it is necessary to add in the cost of the pivoting operations, from (5.27).

$$
\begin{aligned}
\text{upr}\{TPIV_\infty\} &= \text{upr}\{TNOPIV_\infty\}+\sum_{k=1}^{p}Tpiv(k)_1 \\
&= \frac{p}{3}(p+1)(2p+1)tget+\frac{p}{2}(3p+1)tput \\
&\quad +(p-1)^2(tmult+tsub)+\frac{1}{2}(p-1)(p+2)tsolve+ptlu
\end{aligned}
$$

$$
\begin{aligned}
& + \sum_{k=1}^{p} \{p(2p-2k+1)tget + p(p-k+1)tput\} \\
= \; & \frac{p}{3}(p+1)(2p+1)tget + \frac{p}{2}(3p+1)tput \\
& + (p-1)^2(tmult + tsub) + \frac{1}{2}(p-1)(p+2)tsolve + ptlu \\
& + p^3 tget + \frac{p^2}{2}(p+1)tput \\
= \; & \frac{p}{3}(5p^2+3p+1)tget + \frac{p}{2}(p^2+4p+1)tput \\
& + p(p-1)(tmult + tsub) + \frac{1}{2}(p-1)(p+2)tsolve + ptlu \qquad \text{(B.9)}
\end{aligned}
$$

# Bibliography


[1] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/ Cummings, second edition, 1994. ISBN: 0-8053-0443-6.

[2] Werner Almesberger. ATM on Linux. In *International Linux Conference*, EPFL, CH-1015 Lausanne, Switzerland, May 1996. German Unix Users Group.

[3] Gene M. Amdahl. Validity of the single-processor approach to achieving large-scale computing requirements. *Computer Design*, 6(12):39–40, June 1967.

[4] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] Edward Anderson, Zhaojun Bai Bai, Christian Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, Susan Ostrouchov, and Danny Sorensen. *LAPACK Users' Guide - Release 2.0*. Society for Industrial and Applied Mathematics, September 1994.

[6] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[7] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[8] Jose N.C. Arabe, Adam L. Beguelin, Bruce Lowekamp, Erik Seligman, Michael Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user envionment. Technical Report CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, April 1995.

[9] Özalp Babaoğlu, Lorenzo Alvisi, Alessandro Amoroso, and Renzo Davoli. Paralex: An environment for parallel programming in distributed systems. In *Proceedings of the International Conference on Supercomputing*, pages 178–187, New York, NY, USA, July 1992. ACM Press.

[10] Özalp Babaoğlu and Sam Toueg. Understanding non blocking commit. Technical Report UBLCS-93-2, University of Bologna, Department of Mathematics. February 1993.

[11] Maurice J. Bach. *The Design Of The* UNIX *Operating System.* Prentice Hall Inc., Englewood Cliffs, New Jersey 07632, 1986. ISBN: 0-13-201757-1.

[12] David E. Bakken. *Supporting Fault-Tolerant Parallel Programming in Linda.* PhD thesis, University of Arizona, Department of Computer Science, August 1994.

[13] Henri E. Bal. Fault tolerant parallel programming in Argus. *Concurrency: Practice and Experience*, 4(1):37–55, February 1992.

[14] Henri E. Bal and Frans M. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *Conference on Object-oriented Programming Systems, Languages and Applications*, pages 162–177, October 1993.

[15] Amnon Barak, Oren La'adan, and Yuval Yarom. The NOW MOSIX and its preemptive process migration scheme. *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, 7(2):5–11, Summer 1995.

[16] Arash Baratloo, Partha Dasgupta, and Zvi M. Kedem. CALYPSO: A novel software system for fault-tolerant parallel processing on distributed platforms. In *International Symposium on High Performance Distributed Computing*, pages 122–129. IEEE Computer Society Press, August 1995.

[17] Adam Beguelin, Jack J. Dongarra, Al Geist, Robert Manchek, Keith Moore, Reed Wade, and Vaidy Sunderam. HeNCE: Graphical development tools for network-based concurrent computing. In *Scalable High Performance Computing Conference*, pages 129–136, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, April 1992. IEEE Computer Society Press.

[18] Philip A. Bernstein, Meichun Hsu, and Bruce Mann. Implementing recoverable requests using queues. *ACM Special Interest Group on Management of Data*, 19(2):112–122, June 1990.

[19] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *IEEE International Computer Conference*, pages 528–537. IEEE Computer Society Press, February 1993.

[20] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Tranactions on Computer Systems*, 2(1):39–59, February 1984.

[21] Robert Bjornson, Craig Kolb, and Andrew Sherman. Ray tracing with network Linda. *SIAM News*, 24(1), January 1991.

[22] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Annual Technical Conference*. USENIX, January 1997.

[23] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Weng-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[24] Ralph M. Butler and Ewing L. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Parallel Computing*, 20(1):547–64, April 1994.

[25] Gilbert Cabillic, Gilles Muller, and Isabelle Puaut. The performance of consistent checkpointing in distributed shared memory systems. In *Symposium on Reliable Distributed Systems*, pages 96–105, Los Alamitos, Ca., USA, September 1995. IEEE Computer Society Press.

[26] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.

[27] Scott R. Cannon and David Dunn. Adding fault-tolerant transaction processing to Linda. *Software-Practice And Experience*, 24(5):449–466, May 1994.

[28] Clemens Cap. Massive parallelism with workstation clusters - challenge or nonsense. In *High Performance Computing and Networking Conference Europe*, pages 42–52. Springer, April 1994.

[29] Nicholas Carriero and David Gelernter. *How To Write Parallel Programs: A First Course*. MIT Press, 1991. ISBN: 0-262-03171-X.

[30] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[31] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[32] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[33] Jeffrey S. Chase, Franz G. Amador, Edward D. Lasowska, and Henry M. Levy. The Amber system: Parallel programming on a network of multiprocessors. *ACM Operating Systems Review*, 23(5):147–158, December 1989.

[34] Chungmin Chen, Kenneth Salem, and Miron Livny. The DBC: Processing scientific data over the internet. In *International Conference on Distributed Computing Systems*, pages 673–682. IEEE Computer Society Press, May 1996.

[35] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[36] Raymond C. Chen and Partha Dasgupta. Implementing consistency control mechanisms in the clouds distributed operating system. In *International Conference on Distributed Computing Systems*, pages 10–17. IEEE Computer Society Press, May 1991.

[37] Timothy Clark and Kenneth P. Birman. Using the ISIS resource manager for distributed, fault-tolerant computing. Technical Report 92-1289, Cornell University, Computer Science Department, June 1992.

[38] David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steven Lunetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, New York, NY 10036, USA, November 1993. ACM Press.

[39] Partha Dasgupta, Zvi M. Kedem, and Michael O. Rabin. Parallel processing on networks of workstations: A fault-tolerant, high performance approach. In *International Conference on Distributed Computing Systems*, pages 467–474. IEEE Computer Society Press, May 1995.

[40] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, Summer/Fall 1996.

[41] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, December 1993. at International Parallel Processing Symposium.

[42] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[43] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, Computer Science Department, August 1995.

[44] Jack J. Dongarra, Sven Hammarling, and David Walker. Key concepts for parallel out-of-core LU factorisation. In *Collaboration Workshop on Environments and Tools For Parallel Scientific Computing*, August 1996.

[45] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An introduction to the MPI standard. Technical Report UT-CS-95-274, University of Tennessee, Department of Computer Science, January 1995.

[46] Craig C. Douglas, Timothy G. Mattson, and Martin H. Schultz. Parallel programming systems for workstation clusters. Technical Report YALEU/DCS/TR-975, Yale University, Department Of Computer Science, August 1993.

[47] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, 21(8):757–785, August 1991.

[48] Alan Edelman. Large numerical linear algebra in 1994: The continuing influence of parallel computing. In *Scalable High Performance Computing Conference*, pages 781–787, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, May 1994. IEEE Computer Society Press.

[49] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel Saltz. Data parallel programming in an adaptive environment. In *International Parallel Processing Symposium*, pages 827–832, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, April 1995. IEEE Computer Society Press.

[50] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[51] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector. *Camelot and Avalon A Distributed Transaction Facility*. Morgan Kaufmann, San Mateo, 1991.

[52] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems. Technical Report RC 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY10598, October 1994. Second revision, August 1997.

[53] Dror G. Feitelson, Peter F. Corbett, Yarson Hsu, and Jean-Pierre Prost. Parallel I/O systems and interfaces for parallel computers. In *Topics in Modern Operating Systems*. IEEE Computer Society Press, 1997. To appear.

[54] Michael J. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[55] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[56] Ian Foster, Robert Olson, and StevenTuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, Fall 1992.

[57] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H. Sameh. Impact of hierarchical memory systems on linear algebra algorithm design. *The International Journal of Supercomputer Applications*, 2(1):12–48, Spring 1988.

[58] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Parallel Computing*. MIT Press, 1994.

[59] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. John Hopkins University Press, second edition, 1989. ISBN: 0-8018-3772-3.

[60] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.

[61] Andrew S. Grimshaw. Easy to use parallel processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[62] Linley Gwennap. Pentium approaches RISC performance. *Microprocessor Report*, 7(4), March 1993.

[63] Michael Harry, Juan Miguel del Rosario, and Alok Choudhary. VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. *ACM Operating Systems Review*, 29(3):35–48, July 1995.

[64] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

[65] Peter Hoogerbrugge and Ravi Mirchandaney. Experiences with networked parallel computing. *Concurrency: Practice and Experience*, 7(1), February 1995.

[66] Kai Hwang. *Advanced Computer Architecture, Parallelism, Scalability and Programmability*. McGraw-Hill Book Company, 1993. ISBN: 0-07-031622-8.

[67] Karpjoo Jeong. *Fault-Tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions*. PhD thesis, New York University, Department of Computer Science, January 1996.

[68] Marinus Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 297–312, March 1992.

[69] David L. Kaminsky. *Adaptive Parallelism with Piranha*. PhD thesis, Yale University, Department of Computer Science, May 1994.

[70] Ken Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *International Conference on Parallel Processing*, pages III:29–33. IEEE Computer Society Press, August 1995.

[71] Craig Kolb. *rayshade*. ftp://ftp.cs.yale.edu, May 1990. version 3.0.

[72] Craig Kolb. *rayshade*. ftp://ftp.princeton.edu, February 1992. version 4.0.6.

[73] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994.

[74] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, 1994. ISBN: 0-8053-3170-0.

[75] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[76] Peter A. Lee and Tom Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, Austria, second edition, 1990. ISBN: 3-211-82077-9.

[77] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[78] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[79] Joachim Maier. Fault-tolerant parallel programming with atomic actions. In *Workshop on Fault-Tolerant Parallel and Distributed Systems*. IEEE Computer Society Press, June 1994. at International Symposium on Fault-Tolerant Computing.

[80] J. Elliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. Information Systems. MIT Press, 1985. ISBN: 0-262-13200-1.

[81] Steven A. Moyer and Vaidy S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Scalable High-Performance Computing Conference*, pages 71–78, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, May 1994. IEEE Computer Society Press.

[82] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993. ISBN: 0-201-62427-3.

[83] Matt W. Mutka and Miron Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, July 1991.

[84] Nenad Nedeljković and Michael J. Quinn. Data-parallel programming on a network of heterogeneous workstations. *Concurrency: Practice and Experience*, 5(4):257–268, June 1993.

[85] Fabio Panzieri and Santosh K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, January 1988.

[86] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):225–308, Summer 1995.

[87] Lewis I. Patterson, Richard S. Turner, Robert M. Hyatt, and Kevin D. Reilly. Construction of a fault-tolerant distributed tuplespace. In *Symposium on Applied Computing*, pages 279–285. ACM, February 1993.

[88] Stefan Petri and Horst Langendörfer. Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes. *Operating Systems Review*, 29(4):25–36, October 1995.

[89] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. Technical Report UT-CS-96-335, University of Tennessee, Department of Computer Science, August 1996.

[90] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Algorithm-based diskless checkpointing for fault-tolerant matrix operations. In *International Symposium on Fault-Tolerant Computing*, pages 351–360, Los Alamitos, CA, USA, June 1995. IEEE Computer Society Press.

[91] Michael L. Powell and Dave L. Presotto. Publishing: a reliable broadcast communication mechanism. *ACM Operating Systems Review*, 17(5):100–109, October 1983.

[92] Robert Prouty, Steve Otto, and Jonathan Walpole. Adaptive execution of data parallel computations on networks of heterogeneous workstations. Technical Report CSE-94-012, Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering, March 1994.

[93] Jim Pruyne and Miron Livny. Managing checkpoints for parallel programs. In *Workshop on Job Scheduling Strategies for Parallel Processing*, New York, NY, USA, April 1996. pringer-Verlag Inc. at International Parallel Processing Symposium.

[94] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, February 1975.

[95] C. Röder, Stefan Lamberts, and Thomas Ludwig. PFSLib — An I/O interface for parallel programming environments on coupled workstations. In J. Dongarra, M. Gengler, B. Tourancheau,

and X. Vigouroux, editors, *European PVM conference*, pages 59–64, Paris, France, October 1995. Hermes.

[96] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[97] Daniel J. Scales and Monica S. Lam. Transparent fault-tolerance for parallel applications on networks of workstations. In *Winter Technical Conference*, pages 329–341, Berkeley, CA, USA, January 1996. USENIX.

[98] Kent E. Seamons, Ying Chen, Phyllis Jones, John Jozwiak, and Marianne Winslett. Server-directed collective I/O in Panda. In *Supercomputing*, December 1995.

[99] Robert D. Silverman. Massively distributed computing and factoring large integers. *Communications of the ACM*, 34(11):94–103, November 1991.

[100] J. Smith. On synchronisation in fault-tolerant data and compute intensive programs over a network of workstations. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *International Euro-Par Conference*, pages 1025–1029, Berlin, Germany, August 1997. Springer-Verlag Inc. Appears as LNCS No. 1300.

[101] J. Smith and Santosh Shrivastava. A system for fault-tolerant execution of data and compute intensive programs over a network of workstations. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *International Euro-Par Conference*, pages 487–495, volume 1, Berlin, Germany, August 1996. Springer-Verlag Inc. Appears as LNCS No. 1123 and 1124.

[102] J. Smith and Santosh Shrivastava. Performance of fault-tolerant data and compute intensive programs over a network of workstations. *Theoretical Computer Science*, To appear.

[103] Georg Stellner. CoCheck: checkpointing and process migration for MPI. In *International Parallel Processing Symposium*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD, April 1996. IEEE Computer Society Press.

[104] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[105] G. W. Stewart. *Basic Decompositions*, volume 1 of *Matrix Algorithms*. Society for Industrial and Applied Mathematics, To Appear. Draft available online through http://www.cs.umd.edu/ stewart/.

[106] Mark P. Sullivan and David P. Anderson. Marionette: A system for parallel distributed programming using a master/slave model. In *International Conference on Distributed Computing Systems*, pages 181–188. IEEE Computer Society Press, June 1989.

[107] Vaidy S. Sunderam, George A. Geist, Jack J. Dongarra, and Robert J. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, April 1993.

[108] Guarav Suri, Bob Janssens, and W. Kent Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *International Symposium on Fault-Tolerant Computing*, pages 279–288, Los Alamitos, CA, USA, June 1995. IEEE Computer Society Press.

[109] Fore Systems. *ForeRunner ASX-200WG ATM Switch User's Manual*. Fore Systems, Inc., Warrendale, PA, November 1995. MANU0052 - Rev.C, Software Version 3.4.x.

[110] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber. A taxonomy-based comparison of several distributed shared memory systems. *ACM Operating Systems Review*, 24(3):40–67, July 1990.

[111] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van. Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–64, December 1990.

[112] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[113] Charles J. Turner, David Mosberger, and Larry L. Peterson. Cluster-C*: Understanding the performance limits. In *Scalable High-Performance Computing Conference*, pages 229–238, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, May 1994. IEEE Computer Society Press.

[114] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Symposium on Operating Systems Principles*, pages 303–316. ACM, December 1995.

[115] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–267. ACM Press, May 1992.

[116] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[117] Stuart M. Wheater. *Constructing Reliable Distributed Applications Using Actions and Objects*. PhD thesis, University of Newcastle upon Tyne, Computing Laboratory, June 1990.

[118] David Womble, David Greenberg, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *DAGS/PC Symposium*, pages 56–63, Hanover, NH, USA, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[119] Andrew Xu and Barbara Liskov. A design for a fault tolerant, distributed implementation of Linda. In *International Symposium on Fault-Tolerant Computing*, pages 199–206. IEEE Computer Society Press, June 1989.

[120] Songnian Zhou, Michael Stumm, Kai Li, and David Wortmann. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

[121] Marc Zyngier. *md*. ftp://sweet-smoke.ufr-info-p7.ibp.fr/pub/Linux/, April 1996. version 0.35.