Testing of a Novel Distributed Shared Memory Framework

Laurence Markey

A dissertation submitted to the University of Dublin,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

1999

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Laurence Markey
17 September 1999

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request

Signed: _____

Laurence Markey
17 September 1999

Acknowledgments

I would like to thank my supervisor, Brendan Tangney, for his guidance, encouragement and advice throughout the work on this dissertation.

Also Stefan Weber who developed the DSD framework that was the focus of this dissertation, and never lost his patience or good humour during many many consultations.

And finally Brian Coghlan for allowing me to use the CAG Cluster for testing the programs implemented using the framework.

Summary

A novel object oriented framework for software Distributed Shared Memory (DSM) has been developed by the Distributed Systems Group in Trinity College, Dublin for programming parallel applications on a group of loosely coupled workstations. It offers the developer the ability to apply different rules for how accesses to shared data are seen by each process (consistency models) which can be implemented by different means of managing shared data (coherence protocols). Different combinations of models and protocols can be applied to individual objects within an application, in addition the protocols that the framework uses can be customized by the developer.

By providing this range of facilities it is hoped that developers will be able to exploit application specific semantics to achieve improvements in performance and speed-up characteristics while maintaining relative ease of programming.

The objective of this project is to test to what extent this DSM framework delivers the flexibility, customizability and programmability in design of parallel applications that it promises and whether it can offer appropriate speedup characteristics for different classes of problem.

Two applications with significantly different data sharing characteristics were designed and implemented as sample cases to test the framework: the Travelling Sales Person (TSP) problem which has a low communication-to-computation ratio and for which workers can work independently of each other; and the LU Decomposition problem which has a very a high communication-to-computation ratio and an algorithm where workers are highly interdependent

The thesis will show that the framework did achieve appropriate speed-up characteristics. It also allows significant flexibility and programmability: protocols can be easily set and changed for individual objects; the protocols themselves were customized to improve the performance of each of the applications.

Table of Contents

List of Tables and Illustrations

ILLUSTRATIONS

TABLES:

1 Introduction

Parallel programming has been developed in the attempt to improve the speed at which programs can be executed, particularly those that demand a great deal of processing power. It describes the attempt to improve the speed at which a given problem is processed by having a number of processors or workstations working in co-operation on the problem. It is contrasted with sequential programming where only one processor is used and all operations take place in a linear sequence. One of the measures of the performance of a parallel programming system is its speed-up characteristic i.e. how much faster a program is processed as more processors are recruited to work on it.

Distributed Shared Memory (DSM) is a technique that allows parallel programming of applications on a network of computers. It relies on the "*shared memory*" abstraction which enables computers that do not share memory to have shared access to data by a means which appears similar to how they access data in their own memory. The shared memory consists of items of shared data which are replicated at each workstation that needs to use them, thereby eliminating the need to read these data across a relatively slow (in terms of processor speeds) network.

However having multiple copies of items of data introduces the issue of how to maintain all of these copies in a consistent state. This can be separated into two elements the Coherence Protocol which determines what mechanisms are used to maintain the copies in a consistent state, and the Consistency Model which defines what constitutes a consistent state by constraining the event-orderings are allowable.

Different coherence protocols will use difference means of passing updates between workstations. The optimal coherence protocol will be different for different applications, being determined by the different communications requirements of the applications.

Maintaining strict consistency where all reads at all nodes return the most recently written value introduces extra delays because no processor can read a data item when any other processor is updating a copy of that data item. These delays can be reduced by allowing a more "relaxed" consistency model. Many such models have been proposed (e.g. Goodman,

1989; Dubois et al., 1988; Gharacharloo et al., 1990; Keleher et al., 1996a), the more relaxed they are the better performance they offer. However application semantics often demand that some of these are not acceptable for any data item in the application, or are not acceptable for certain key data items.

In order to optimise the performance of parallel programmes it would be desirable to be able to separate the means of applying coherence protocols and consistency models so that an appropriate combination could be selected for each application from a range of differing consistency models and coherence protocols. In addition further optimisation would be promised by a system that could apply consistency on a per-object basis.

This is what has been attempted in a framework that has been developed in Trinity College Dublin. The framework (Weber et al., 1998) consists of a set of Java classes that can be selected and extended to provide combinations of consistency models and coherence protocols on a per-object basis for parallel programming. It allows the flexibility of easily choosing and changing the combination of coherence protocol and consistency model that have been chosen for any data item. In addition it allows existing protocols to be customised, or new ones defined in order to achieve optimal performance for specific applications. It does this in an environment that is relatively easy to program.

Objective

The objective of this dissertation was to test how well the framework delivers on the benefits that it promises. This was done by using the framework to implement two parallel applications with different sharing semantics and then testing the performance of these applications over the network in order to determine what type of speed-up was offered. In the process of developing and testing these applications it was possible to analyse how well the framework delivered on the flexibility, programmability and customisability that it claims to offer.

Results

It was found that developing parallel applications for the framework was relatively easy, particularly if the problem space could be partitioned into a list of work items before the commencement of processing. There was a significant range of coherence protocols and

consistency models provided in advance. It was possible to customise these, or develop new ones by creating new classes that extend the base CoherenceProtocol or ConsistencyModel classes. The steps required to assign and change consistency models and coherence protocols for individual variables were clear and easily programmable. The speed-up characteristics obtained were very different for the two applications reflecting the different sharing semantics and communication/computation ratios of the applications. Selection of appropriate ConsistencyModels and CoherenceProtocols had a significant affect on the levels of speed-up obtained.

Therefore we have concluded that the framework has delivered the capabilities that it has promised.

2 Background to Distributed Shared Memory

2.1 Introduction

The purpose of this chapter is to introduce some of the concepts and approaches that form the background against which this investigation was carried out. It introduces Distributed Shared Memory (DSM), explains parallel programming and surveys some of the approaches other than DSM that have been used for parallel programming. It then looks at some of the different approaches that have been proposed based on DSM, the problems that they encountered and how some of these can be overcome. It lays out a conceptual framework that can be used for classifying and understanding DSM systems and for optimising the performance of parallel programs run over DSM.

2.2 What is DSM

Distributed Shared Memory (DSM) is a technique that allows parallel programming of applications on a network of computers. It relies on the "*shared memory*" abstraction which enables computers that do not share memory to have shared access to data by a means which appears similar to how they access data in their own memory.

The objective of this dissertation is to evaluate a novel framework that has been developed for developing programs to run over DSM. In order to clarify exactly what benefits are promised by this new framework-based approach to DSM it is worthwhile to survey the different approaches that have been applied to parallel programming in the past.

2.3 Parallel Programming

Parallel programming is the name for the attempt to improve the speed at which a given problem is processed by having a number of processors working in co-operation on the problem. It is contrasted with sequential programming where only one processor is used and all operations take place in a linear sequence.

Developing parallel programs involves many concerns that are not encountered while developing sequential programs. The greatest of these are process communication and

synchronisation. Since a parallel program uses multiple sequential processes executing and interacting in parallel to solve a problem, developers have to ensure that a consistent view of the problem data is maintained across all these processes, this is called synchronisation. However in the interests of efficiency this must be achieved without requiring excessive communication between processes.

2.3.1 MultiProcessing

The initial approach used to implement parallel programming was multi-processing. It required the construction of a computer with a number of CPUs that share the same bus and that use that bus to access a shared memory. By having a single bus and a single shared memory that contains only one copy of each data item and the semaphore that controls access to that data, it provides a system whose structure implements data consistency and synchronisation. Consistency is the requirement that all of the processors see a consistent view of the data. This is ensured by the fact that there is only one set of data which all processes have access to, thus no two processes can have a different view of the data. Synchronisation is achieved because the system implements semaphores, it ensures that when any one process is updating a data item all other processes have no access.

However these bus-based multiprocessors suffer from a lack of scalability, they cannot be used with more than a few dozen processors because the bus (which only one CPU can use at a time) tends to become a bottleneck. This is because when any one processor accesses a data item in the shared memory they prevent all other processors from accessing any data item because only one processor can access the bus at any time. In addition, in its simplest form, this architecture makes no distinctions between non-shared data and shared data and between read-only data and read-write data. In each of these cases non-shared data and local copies of read-only data could be stored in local memory if that were available without compromising the consistency of the overall data set. This would allow performance improvement because shared-bus access would not be required for access to these data types and so frequency of demand for the shared-bus bottleneck would be reduced.

Switched multiprocessors, such as DASH (Lenoski et al., 1992) can be made to scale, but they are relatively expensive, complex and difficult to build and maintain. A switched

processor tries to overcome the limitation on the number of processors that can efficiently share use of a single bus by having several clusters of processors, each cluster with its own bus and the clusters connected by slower (relative to buses) intercluster links. When a processor wants to access a data item located on the local memory it accesses just as it would in the multiprocessor situation described above. If however the item's address indicates that it is located on the memory of one of the other clusters then the data item is fetched across the intercluster link. This topology assumes that most accesses by processors will be to their local memory and so the slow intercluster links will be relatively infrequently used thus having little affect on performance and allowing greater scale.

However, as mentioned already, these systems are expensive and complex to build and maintain. Given that many locations had networks of workstations it became more attractive to see if the existing processing power on these networks could be used to perform parallel processing, rather than purchasing expensive hardware that performed a specialised function. This approach of attempting to use a network of standalone computers for parallel processing is called multicomputing.

A multicomputer consists of a number of separate standalone computers, each with their own CPU and their own memory, which communicate via a network. This makes larger scale feasible: because each machine accesses its own memory there is no bottleneck for access to memory, they only use the network to propagate updates to data shared with other machines. Large multi-computers are easier to build and cheaper because they use standard workstation and network hardware. However they create a more difficult programming environment, because the programmer now has to deal with issues of consistency and synchronisation. In addition they have to cope with extra time required to communicate across a network. There have been two main approaches to multi-computing: Message passing and Distributed shared memory.

2.3.2 Message Passing

Parallel programming requires that the processes running the application must share data. Message passing achieves this by passing between processes messages containing the data to be shared (Coulouris, et al., 1994). By using this approach where, provided messages are passed between processes in the agreed format, the actual processing can in principle be

performed on any type of computer message passing can implement parallel programming over a heterogeneous network.

Message passing requires extra work on the part of the programmer as compared to multiprocessing. Variables to be passed between processes must be marshalled into messages, messages transmitted and variables unmarshalled at the receiving side. If a heterogeneous system is to be used then different marshalling will be required for each different kind of computer being used on the system. In addition there is a requirement to monitor that messages have been received by their intended recipient, and to manage issues such as flow control, buffering and blocking.

Because each process maintains its own copy of the data, message passing must implement synchronisation to ensure that all processes are using a consistent set of data. However it cannot do this using normal synchronisation constructs such as locks, instead synchronisation must be implemented using special primitives. Message passing does not offer the same flexibility in the selection and changing of policies as will be offered by some of the other systems we will discuss.

2.3.3 DSM

Distributed Shared Memory (DSM) is a technique which allows parallel programming of applications on a network of computers. It relies on the "*shared memory*" abstraction which enables computers that do not share memory to have shared access to data by a means which appears similar to how they access data in their own memory.

Each of the computers is a standalone computer, but using DSM they can all work in parallel on a single application. Each computer works on the problem as it would work on a standalone application, because each has accesses shared data in a way similar to how it accesses its own local data. It is up to the DSM system to ensure that that each workstation sees a view of the shared data that is consistent with that seen by all the other workstations.

Shared Memory

Unlike message passing DSM does not require the programmer to manage communication between processes by passing data messages, instead it provides processes with a shared

address space. A distributed application consisting of processes running on several nodes can access the data in the shared address space using operations like "read" and "write" which, combined with synchronisation primitives, are applied to a specific address. Thus each process uses the shared address space in a similar way to how they use their local memory space with the addition of synchronisation management.

The shared address space consists of local memory on each machine in the system. When a process accesses data in the shared address space, a mapping manager maps the shared memory address to the physical memory of one of the nodes on the system. The process can then read that data and make a local copy. Each computer can maintain local copies of recently accessed data items. Changes made by one process to items in the shared address space are propagated by the DSM system to the local memory of all other machines with copies of that data. The DSM system ensures a consistent overall set of data, the programmer need not worry about this.

Thus it appears to distinct processes on different nodes that they have concurrent access to a shared set of data. And this is achieved while allowing most accesses to data to be to local memory. This enables DSM to improve performance by using local memory accesses whenever it wishes to access a shared variable. The DSM system only needs to propagate data when a change is made to a shared variable. In addition communication between processes and across node boundaries is transparent, it is handled by the DSM and no marshalling is required.

However enforcement of consistency does not come without cost. If we are to ensure that all processes sharing some data see changes to that shared data in the same order then we will have to implement distributed locks. This will require several stages of communication to establish locks on all copies of data, and update all copies of data; only when this has been committed can any processes have access to this data. This introduces significant delays, particularly when communicating across a network. However, as we shall see it is possible to significantly increase performance if we relax the level of consistency being enforced.

2.4 Different DSM Systems

Having outlined the concept of a virtual shared memory upon which DSM relies, we shall now look at different DSM systems that have been developed and the different approaches they have used to implementing the virtual shared memory. They can be broadly grouped into two categories: page-based DSM and object-based DSM.

Page-Based DSM

The initial attempts at DSM were attempts to simulate the operation of multi-processors using a distributed system. This was partly because one of the goals of these original DSM systems was to be able to run existing multi-processor programs on larger DSM systems with the minimum of modification. Multi-processors use a memory shared by all processors in a cluster located on one bus, with one copy of each data item stored in the memory (Tanenbaum, 1995). Each processor can access each address in the memory directly. As we have seen one way of coping with the limitations on access to the shared bus was by passing data along inter-cluster links.

The network hardware that was available for these early DSM systems was optimised for passing pages across the network. So the early attempts at DSM married these two approaches, they divided the address space into pages, each page being located in the local memory of one machine in the system. When a processor tries to access an address in a page that is not local the DSM software identifies the page, locates it and fetches it from the machine it is currently stored on.

Thus the virtual memory consists of pages, stored in local memory of a machine on the network, that migrate across the network as different processors need to access their contents. This migration is required for read accesses as well as write accesses, because the only way data can migrate across the network is if the page it is located on is transferred to another machine.

Because it takes longer to access a page that is located on another machine, performance can be improved by locating particular pages on the machines that use them most frequently. However this approach is not adequate if there is more than one processor that needs frequent access to a given page, because it can lead to a problem called thrashing or alternatively the ping-pong effect.

Thrashing occurs when there are several processes that have frequent needs for access to the same page, this will mean that ownership will be difficult to determine and the page will be transferred back and forth across the network at a very high frequency. These processes will spend much of their time waiting for the page to be returned to them and so their performance will suffer.

The solution to thrashing it is to allow several copies of the page to exist simultaneously on a number of machines and to institute some means of ensuring that they all display a consistent view of the data that they contain. This is the approach that was adopted in the IVY system.

IVY (Integrated Shared Virtual Memory at Yale) was implemented on a network of Apollo workstations connected by a network (Li and Hudak, 1989). The memory was divided into pages, 1 Kbyte in size, which were transferred across the network. There could be more that one process on each workstation. Each process saw the address space as divided into two areas:

1. A shared virtual memory address space which can be accessed by any process
2. A private space which can only be accessed by processes on the same workstation

Each workstation has a mapping manager that controls the mapping between the shared virtual memory space and the local memory of the workstation. When a process raises a page fault a check is made to see if the page is located locally at the workstation. If the page is not local a remote memory request is made and the page is acquired from another workstation.

IVY follows a multiple readers-single writer semantics. There can be multiple copies of a page, each of which can be read separately, thus eliminating thrashing when several processes want to read from the same page. All readers always see the latest value written, which means that IVY enforces “strict” consistency (see below). This consistency model is implemented using the “write-invalidation” protocol. This means that when a writer wants to write to a page which is copied elsewhere, then all other copies of that page are invalidated before the update is made. If a reader attempts to read from an invalidated page the system will obtain a copy of the updated version of the page. In order to enforce strict

consistency where all readers see the latest update, it is necessary to complete the invalidation of copies of an updated page before another update can commence. Therefore there can only be one writer to a page at any time.

Write invalidation saves communication because copies of pages are only updated when a process attempts to read from them. So communication is reduced by only updating those copies that are used subsequently; and by only updating them when they are to be read which may be after several writes have been made.

Ivy resolved the thrashing problem for reads, but not for writes. Each time a write is performed all other copies of the page are invalidated, so subsequent readers or writers will have to receive a copy of that page over the network.

Mirage uses a different approach to reducing thrashing (Fleisch & Popek, 1989). It adds a parameter to the sharing protocol that sets the minimum time for which a page will be available at a node. By ensuring that a page remains at that location for at least a certain time it allows one process complete a sequence of updates to a page before the page is passed across the network. This parameter can be tuned dynamically to ensure that the page does not stay at a node longer than is needed. It also provides a "yield" primitive which a process calls when it has completed a sequence of updates. Thus ensuring that the page does not stay fixed for the duration of the period if there are no further accesses to be made.

However even if these measures do allow page-based systems to reduce thrashing, there is another problem which they have more difficulty in coping with: false sharing.

Granularity and false sharing

Granularity describes the size of the minimum unit of shared memory. In page-based systems it is the page size. The underlying protocols and hardware that are used to propagate updates will have an influence on the choice of granularity. From their point of view efficiency is maximized by making the granularity into a multiple of the size used by these. Page-based systems are designed to optimise the passing of pages around the network so the page size will be determined by the underlying hardware. This ensures that there is limited room for manoeuvre in selection of granularity in page-based systems.

A larger page-size will take advantage of the locality of reference exhibited by processes. A process that accesses one variable is very likely to access other variables located near to it in memory. However this consideration is often overridden by the fact that a large page-size increases the likelihood of false sharing. This occurs when a number of processes seek to access unrelated data items that happen to be located on the same page. Thus there will be delays caused by this contention even though the variables themselves are unrelated. False sharing particularly occurs if a page contains a key variable that is accessed very often, this will mean that processes will experience considerable delays in accessing any other variables located on the same page.

This problem has been moderated somewhat for page-based systems by adopting a mode of operation where key variables are stored alone on one page, so that accessing them will not cause any difficulties in accessing other unrelated data items. However this is only a work-around and an attempt to replicate what is achieved more effectively in shared-variable DSM systems. In particular it wastes valuable memory space, and bus time; a whole page has to be maintained and passed around in order to manage the state of only one variable.

Shared Variable DSM

Shared variable systems were proposed as a means of overcoming some of the problems with page-sharing. By focussing the system on the management of v*ariables* rather than pages they eliminate the problem of false sharing. Processes seize individual variables, thus there is no possibility that accessing one variable will prevent others accessing an unrelated variable that happens to be located beside it in memory.

In addition they ensure that only those variables that need to be shared are passed across the network. Rather than having the same mechanism for all of memory, variables that need to be shared are treated differently from the rest of memory.

Shared variable systems allow several processes to maintain replicates of individual variables in their memory. This solves the problem of thrashing, and ensures that individual processes have faster access to shared variables. However, it calls for a mechanism to ensure that all replicates of a variable are maintained in a consistent state.

This can be optimised by allowing different levels of consistency for different variables which can be achieved by performing specific annotations on the variables that declare what policy applies to them.

Munin is a system that uses this approach (Bennett et al., 1990, Carter et al. 1991). It has a fixed range of types that can be applied to shared data: write-once, write-many, private, migratory etc. Different protocols are used for updating shared data that have been assigned different types. Thrashing caused by competing writers can be avoided by specifying the type as write-many.

A shared variable system puts the responsibility on the programmer to annotate each data item to indicate which variables are shared together and what type of policies apply. This was not necessary in page-based systems where the system treated all data similarly. This flexibility does offer improved performance, however because of the need to annotate each data item it is difficult to avoid costly mistakes. What was needed was some means of simplifying the application of different sharing policies to data items. This is what object-based DSM aims to do.

Object-based DSM

An object is a programmer-defined encapsulated data structure (Booch, 1994). It consists of internal data and associated methods. The methods are procedures that operate on the object state, they can be called by any program that has a reference to the object. Thus an object is a data abstraction that comes supplied with methods to operate on it. One of the key properties of an object is that it implements *information hiding* which means that direct access to the internal state is not allowed. The only means by which the internal state can be accessed or operated on is via the defined methods. Forcing all accesses to an object's data to go through the methods helps structure the program in a modular way and allows the programmer to control the means by which the object can be changed. Objects offer possibilities for optimisations that are not available with a shared memory composed only of pages or of shared variables.

Another key property of objects is *inheritance*. This allows an object to inherit the data and methods of another object. This can be used to simplify the management of how different policies are applied to different classes of data items.

In an object-based distributed shared memory processes on several machines share an abstract space filled with shared objects. The management of these objects is handled by the DSM system. Any process can invoke any object's methods once it has obtained a reference to that object. The process and object need not be located on the same machine, once the method is called it will be run at the location where the object resides.

The issues of replication and managing updates and/or invalidations in order to maintain consistency still have to be addressed in this system. However it does offer number of advantages. In particular the fact that the access methods can be used to restrict how accesses can be made may ensures that synchronisation can be built into the access mechanism. This reduces the possibility of error and makes programming different sharing policies for different data items much easier. Because of the fact that synchronisation and access are controlled at the level of the object the underlying implementation can be more flexible. This has led to the separation of what we have so far called sharing policies into two separate elements which can be defined separately for an object-based DSM system (Weber et al., 1998). The Consistency Model describes the type of consistency that is being enforced for a given variable and the Coherence Protocol describes the mechanisms that are being used to implement that model.

And of course an object-oriented approach, by using inheritance, allows the development of frameworks, which are sets of co-operating classes embodying abstract designs that can be reused to provide a structure for developing applications within a certain domain (frameworks are discussed fully in the next chapter). Using a successful framework should simplify the development of individual applications within the framework's domain.

These three concepts: consistency model, coherence protocol and framework and the benefits that using them together can offer are the key ideas driving the design of the DSM system that has been developed in Trinity College. The remainder of this chapter will be devoted to explaining consistency models and coherence protocols; frameworks are explained in the next chapter.

2.4.1 Consistency models

A consistency model is a system of rules that puts constraints on what orderings of events are allowed in a DSM system (Weber et al., 1998). It is used to ensure that at all times every process with access to a shared data set has a view of that data set that is coherent with the view of all other processes. However for the views of all processes *to be coherent it is not necessary that they be the same.* A shared memory is coherent if the value returned by a read operation is always the value that the programmer expected. This means that a shared memory can be coherent when different processes see different values of a given variable at the same time provided that this is what the programmer had expected. By "what the programmer expected" we mean in accordance with the consistency model that the programmer has specified. As long as the programmer has allowed that different users may see different values of a variable, then different values of the same variable coexisting in the same system are coherent.

Intuitively we tend to think that there can be only one consistency model that can be adequate, all processes must have the same view at any one time (which can be implemented by saying that all reads from data must return the most recently written value). This is based on the premise that if I read a value that does not incorporate an update that has been made by another user, then my view of that value is inconsistent with the view of the writer of that value. However this need not be the case, if for example the values being entered are only estimates of something that is unknown, then the fact that two processes have different estimates may not constitute a major problem.

The classic example that illustrates this point is that of newsgroups. There are many users of a newsgroup and there are many messages posted. However it is not essential that new messages go out to all users at exactly the same time and so are seen by all users in exactly the same order.

Strict consistency may be abandoned because it often confers limited benefits in terms of correctness, i.e. it does not produce a better result for many applications, while it always exacts a heavy penalty on performance. DSM systems achieve gains in performance by enforcing more "relaxed" consistency models that require less communication overhead. These are achieved by allowing the consistency model to specify allowable variations in the

order of accesses. Some of the consistency models that have been proposed are outlined below.

Strict Consistency

The most stringent consistency model is called strict consistency or alternatively sequential consistency. It is defined as:

> "Any read from a variable must return the value stored by the most recent write to that variable." (Lamport, 1979)

This model requires the total ordering of requests, so that all requests are seen in the same order throughout the system. So the distributed system of copies of the variable appears as if there was only one copy of the variable which all processes were accessing. This leads to great reduction in efficiency because any update to a variable can only be completed not merely when that update has been propagated to all processes, but only when their confirmation that they have received the update has been received. Thus this works against the key advantage of DSM i.e. maintaining local copies of shared data, thus allowing local memory accesses to that data. Now every update is not complete until several sequences of messages have been passed back and forth across the network between the initiator and every other copy of that variable on the system. Once one process starts an update then no other process can access the variable until this process is complete.

Note also that because of the delays that this necessarily imposes processes are not guaranteed to be able to update or even read from a variable at will, thus this intuitive consistency model may not even be an unattainable ideal, but could actively interfere with applications that rely on real-time data access.

Processor Consistency

Processor consistency is a relaxation from strict consistency, which was defined by its author as:

> "Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes." (Goodman, 1989)

All processes see the writes of any one process in the order in which they were sent. However there is no guarantee as to what order a sequence of writes by different processors will be seen in at different processors. This means that writes by different processors that had a causal effect on each other may be seen by some processors in an order that breaks this causal relation. This means that writes by a single process can be queued, and so long as they are performed in the right order then other processes need not break off processing in order to perform these writes. This allows considerable freedom to nodes on a system as to when they need to perform updates, however even more relaxed consistency models have been proposed.

All the models we have seen so far have been *uniform* models, i.e. they treat all memory accesses as being equally important and apply the same restrictions to them all. However once we understand the semantics of particular applications we will see that very often only a small proportion of accesses to memory are critical. *Hybrid* models define classes of accesses that are treated differently in an attempt to utilise this knowledge.

Release Consistency

Release consistency exploits the fact that programmers use synchronisation constructs in many applications, these are used to manage critical sections of the program execution or access to critical variables. Release consistency assumes that as long as these are synchronised then synchronisation will not be an issue for the rest of the program. This system relies on the programmer's ability to use these constructs correctly because synchronisation is only applied where the programmer has asked for it. Synchronised accesses can be declared using acquire and release accesses, which indicate the start and end of critical sections of code. The guarantee that is offered is:

> "All ordinary memory accesses issued prior to a release will take effect at all other processes before the release completes – including those accesses issued prior to the preceding acquire" (Gharachorloo et al., 1990)

Updates need not be forced until a critical section is being left, hence the title release consistency. For this reason release consistency can be used to share several updates in one message, or to share only the last of a series of updates.

Gharachorloo has shown that release consistency gives results equivalent to strict or sequential consistency, provided the synchronisation accesses are used correctly. Thus increased performance can be achieved without compromising application semantics through optimising the use of synchronisation by only applying it where needed and allowing processes to proceed without delay where it is not applied in the program.

Lazy Release Consistency

There have been a number of proposals for a relaxation of Release Consistency. These are all responses to the observation that release consistency might be too "eager" in some situations. It is eager in that it sends updates as soon as the updating process releases its lock on a critical section. The "lazier" versions propose different rules which ensure that updates can be delayed or reduced in scope. For instance Keleher describes a lazy release consistency that propagates an update, not when a lock is released, but only when it is next acquired; it only needs to send the update to the process acquiring the lock, and can include the update in the response to the lock request (Keleher et al., 1996a).

The lazy release consistency that has been implemented in the DSM system to be studied here only propagates an update when a subsequent read is attempted on the same portion of data. This is particularly useful for applications where different processes are performing independent operations which do not need to read the results obtained by other processes. In this case updates need not be processed at all until the end of the application when only the master needs to be updated with the results obtained by each process.

2.4.2 Coherence Protocols

The coherence protocol describes the mechanisms that are used to ensure that the virtual shared memory in DSM is maintained in a coherent state. The consistency model defines what orders of events (i.e. accesses to memory) are allowed. The coherence protocol is responsible for ensuring that these accesses are performed correctly so that the consistency model that has been specified is implemented.

Li and Hudak (1989) is the classic article that identifies the key issues that have to be addressed in ensuring that memory coherence is maintained in a DSM system. Their

analysis focuses on the requirements of a page-based system, and does not separate consistency model issues from coherence protocol issues. However we can identify in their article issues that are not addressed by consistency models, as outlined above, these are the issues that the coherence protocol must manage. If coherence is to be achieved the following must be implemented: managing the local copies of data items in the virtual shared memory in a way that ensures the whole data set remains consistent; keeping track of the ownership and location of copies of the data; and achieving the distribution of updates to all nodes with copies of the updated data. We will look at each of these in turn.

Memory Management

Managing the local copy of the virtual shared memory space involves handling local accesses (writes/reads to/from the local copy) and handling updates. Updates can be achieved either by overwriting all copies of the data to be updated (write update protocol), or by updating one copy and invalidating all others (write invalidate protocol). Invalidation of a local copy will prevent the protocol from returning the value of that local copy as the result of a read.

For a read operation memory management must return a value when supplied with the location of the local copy. However this does not mean that the value of the local copy is what will be returned. If the local copy has been invalidated because another copy of that data has been updated then the value of the updated copy will be returned.

Tracking Ownership

Ownership refers to tracking where the most up-to-date copy of a data item is located. This is a service that must be visible to all nodes in the system. In order to assist in the management of the complete set of copies of a data item it must also keep track of all local copies of each data item so that they can be located should they need to be either updated or invalidated.

This can be achieved by using a broadcast message which instructs nodes to invalidate their copy of a given item if they have one. However this depends on being confident that all nodes will always receive such broadcast messages. A more robust alternative is to have an ownership manager which maintains a list of copies for each item and, by communicating

with each node where a copy resides, it can receive confirmation that every copy has been invalidated.

Distribution Management

Distribution management involves the interface with the communication mechanism. It must ensure that updates are sent out to whatever nodes are deemed to require the update. In addition it must handle incoming updates from other nodes. As well as handling updates it must also deal with requests for updates, i.e. it must be able to send them out to particular nodes and be able to handle them when they come in from another node.

Once these three issues have been dealt with in an interlocking manner then the protocol should be able to correctly perform updates to the virtual shared memory in the order specified by the consistency model.

2.5 Summary

The purpose of this chapter was to introduce some of the concepts and approaches that form the background against which this investigation was carried out. It explained parallel programming and surveyed some of the approaches have been used including DSM. It looked at some of the different approaches to DSM that have been proposed, the problems that they encountered and how some can be overcome. It identified a conceptual framework of consistency model and coherence protocol that can be used for understanding the functionality provided by DSM systems and for optimising the performance of parallel programs by managing these issues separately. In the next chapter frameworks are introduced and their application to DSM is outlined.

3 Frameworks

3.1 Introduction

The purpose of this chapter is to introduce the concept of a framework, the characteristics of frameworks, their uses and to indicate how they might be relevant to programming DSM applications.

Then the DSO framework that has been developed in Trinity College is introduced. Its links to the major DSM concepts identified in the last chapter are explained in terms of its major elements, and the relationships between them. Finally the steps required to prepare an application so that it can be run as a parallel application over the DSO framework are explained.

3.2 What Frameworks Are

Frameworks are structures of classes that are used to assist in code reuse. However this does not meant that they just another name for class libraries. The components in a class library can serve as a standalone tools each of which can be recruited as a solution to a particular problem, without needing to call on any other element of its class library. They are designed to provide their functionality in more or less any context. A class library is effectively a toolbox.

A framework on the other hand is a high-level plan that can be applied to create a family of solutions to a range of problems posed by a particular application domain. It can do this because it addresses key issues in the domain in a systematic and interrelated way. This is explained as:

> " A framework is a set of classes that embodies an abstract design for solutions to a family of related problems … frameworks provide for reuse at the largest granularity." (Johnson & Foote, 1988)

This means that a framework is a set of cooperating classes that must be reused together in order to provide a reusable outline design for solutions to a specific class of problems. It achieves this by defining a set of abstract classes and defining their responsibilities and

collaborations in terms of key methods that they implement and interface methods through which they interact. In order to apply the framework to a particular problem a programmer must customise it to provide the particular functionality required for the application. This is done by creating subclasses that extend the abstract classes of the framework, adding particular functionality without compromising the inter-relating structure that has been laid down in the framework. The programmer has the option of creating a suite of such subclasses for each class in the framework, thus providing a modular system where different policies and designs can be applied within the structure of the framework.

The objective in developing a framework is to capture a range of domain specific knowledge in a structure that ensures that all of the key issues of this domain have been addressed. In effect it is to achieve design reuse. By applying a framework developers should be able to proceed more quickly to the details of the application they wish to create, rather than first having to identify what general concepts apply in this domain and then develop a structure which is capable of integrating them into a workable whole. Through using workable structure that has been laid out code reuse is also achieved.

The framework allows the programmer to work within a structure that has been tested for many different applications and that has been constructed with the benefit of the accumulated expertise of the framework's developers.

3.2.1 Other types of reuse

In order to clarify exactly what kind of reuse is offered by frameworks, we will consider in this section the similarities and differences between frameworks and some other types of reuse, namely class libraries, design patterns and components.

Class libraries

As mentioned above class libraries are a common approach to software reuse that is less domain-specific than frameworks (Johnson and Foote, 1988). Class libraries provide a range of standalone tools to be applied to wherever a particular functionality is required. In general they can be applied in any context. Class libraries provide code reuse. Frameworks provide reuse at a much higher level of abstraction. A framework is an outline for the development of a certain class of applications, whereas a class library could be used in many different types of application.

Design patterns

Design patterns are another approach to software re-use that is more general than class libraries (Gamma et al., 1995). However patterns are not domain specific in the way that frameworks are. Patterns are designs that can be reused in any number of domains. They do not offer code reuse, it is up to the developer to create code that implements the pattern. Patterns are generally much smaller in scope than frameworks, a pattern will define a type of interaction between a small number of elements. For instance the barrier that was added to the consistency model for one of the test applications for the framework is an example of a design pattern. It is a technique used to co-ordinate a number of processes. In contrast a framework is an outline for a complete system.

Components

Components are self-contained instances of abstract data-types (Fayad & Schmidt, 1997). They provide a defined interface that allows any application, or indeed another component, that has a reference to them to be able to interact with them via this interface. Reuse of a component is achieved by leveraging knowledge of its external interface, no understanding of the implementation of this interface is required. In contrast reuse in a framework is a matter of extending the super-classes that have been provided. This requires a detailed understanding of the internal structure of these classes and of their interaction. In extending the classes the programmer will often need to override methods of the super-classes without compromising the functionality that they were intended to provide. Reuse of components requires a much lower level of expertise.

Many computer environments that are designed for use by non-experts consist of a collection of components that can be plugged together to provide more complicated functionality. However each component in itself is only an instance of an abstract type, whereas a framework is an overarching design encompassing a set of classes. However, as we shall see, mature frameworks can often look like systems of interchangeable components.

3.2.2 Features of frameworks

To summarise what has gone before we can see that frameworks have the following features:

Reusability

The developer is able to save time and improve the quality of applications by reusing the structure that has been defined using the domain-specific knowledge of the builders of the framework. Frameworks provide reuse of both design and code.

Modularity

By allowing extensions of the classes in the framework, while still ensuring that they implement the key interfaces and methods a developer can create a suite of modules from which combinations can be selected to provide alternative policies and designs within the application domain.

Extensibility

The framework is extensible in that the developer is free to extend the abstract classes to achieve extra functionality that is not included in the framework. However this can only be done if the developer understands the role of the classes that are to be extended and retains this in the new subclasses.

Inversion of Control

The framework provides an overarching system. The extensions and additions by the developer will be to methods that are called by the framework. Therefore the framework tends to control the order of execution. Thus unlike class libraries a framework is a system which executes the modifications the user supplies, not a set of modifications which the user may apply to an application.

Types of Framework

Johnson and Foote (1988) identified two kinds of framework: whitebox and blackbox frameworks. The key difference between the two is that in whitebox frameworks application specific behaviour is added by subclassing from the classes of the framework. This is very much the classic framework scenario as outlined above. These methods must be designed and implemented in such way as to retain the functionality intended by the

designer of the superclasses. This requires the application developer to have an understanding of the framework's implementation.

In blackbox frameworks, on the other hand, various components which extend each of the main classes of the framework are already available. They have been supplied by the creators of the framework, or perhaps by previous users who have extended the framework. An application developer may then be able to create a particular application by selecting from the components that have been made available. In order to create an application the developer need only know the public interfaces of the components and the functionality that they provide, there is no need to understand the details of their implementations or of the detailed workings of the framework.

For this reason blackbox frameworks are more easy to use and require much less of a learning curve, because the user only need only learn about the particular functionality provided by different components, not about the mechanisms by which the different kinds of classes are inter-related to form a working framework.

Johnson and Foote point out that this demarcation is not strict, there is a continuum stretching from whitebox to blackbox frameworks. Indeed the tendency is that a framework will move from being a whitebox to being a blackbox framework as more subclasses are developed to provide particular functionality. If someone has already developed a given functionality for a whitebox framework then there is no obligation on a user to go and define their own subclass with similar functionality. Instead they can simply reuse the functionality that has already been defined. As time goes on and more such classes with different kinds of functionality are defined using the framework will tend to become more a matter of selecting appropriate combinations rather than subclassing to create new functionality.

Conversely for a blackbox framework. If the developer finds that the components available do not provide the functionality needed then, provided the developer has access to the super-classes and understands their rationale, it is possible to develop subclasses that provide the functionality required.

It might be better to say that whitebox and blackbox refer to different ways in which a framework can be used. Depending on a developer's programming expertise and understanding of the framework, it may be open to that developer to extend the framework in a whitebox manner. If that expertise is lacking then the developer is restricted to using the framework in a blackbox manner.

3.2.3 Problems with using Frameworks

Using frameworks for application development offers many advantages such as code and design reuse, and achieving quality applications. Very often frameworks are defined precisely because it has been found very difficult to design a robust application for a particular type of domain. However frameworks also carry their own disadvantages (Fayad and Schmidt, 1997), only some of which come from the fact that they will most likely be dealing with a difficult application domain.

Development effort

The effort and domain knowledge required to successfully develop a framework for a given application domain is greater than that required to develop an application in that domain. This of course is no surprise. A framework is intended to be general and to cope with variations that may not apply in a given application. Also the time invested in developing a framework is intended to be an investment that will yield reduced development time and better quality for a whole range of applications to be developed in the domain.

Learning curve

The learning curve involved in learning to use a particular framework will be quite high. This is certainly the case if the framework is to be used as a whitebox framework.

Difficult to debug

Debugging applications created with a framework is often difficult. The framework controls the flow of control, so it may be difficult to identify what is the cause of a given fault without and understanding of how the framework controls the execution.

3.3 Frameworks applied to DSM

Having looked that the characteristics and features of frameworks, and before explaining the DSD framework it will be worthwhile to consider another framework that has been proposed to aid in the development of DSM parallel applications – DISOM.

3.3.1 An existing framework

DISOM (Castro et al., 1996) is an example of an object-oriented framework developed for building DSM applications based on shared objects. Classes become shareable by extending super-classes that have methods to pass updates to relevant nodes. The framework provides one consistency model (entry consistency). Others can be added by defining new classes with functionality that differs from the entry consistency class. However DISOM does not offer a similar interface to the coherence protocol therefore there is no capacity for the user to modify the coherence protocol.

DISOM allows consistency models to be applied to classes. Thus all instances of a class will have the same consistency rules applied to them. However in many applications although there will be many instances of a given type of data item, there may be performance benefits to be gained by relaxing the consistency applied to many of them and only applying more strict consistency to those that perform key roles in the algorithm.

This framework is offering flexibility in only one (consistency models) of the two major areas where it can be applied in DSM, in addition it only allows consistency to be applied on a per-class basis, not on a per-object basis. DISOM only provides one pre-defined consistency model and leaves it up to the user to define any others that they might wish to use, therefore it makes no concessions towards black-box use and demands that the user develop a full understanding of the structure of the DISOM framework before they can attempt to apply alternative consistency models.

Possible improvements

One would expect that a framework that could overcome all these limitations would offer considerable benefits in terms of improved performance of applications, through selecting and customising both consistency models and coherence protocols that are appropriate and through applying them on a per-object basis. In addition if there is a

selection of predefined consistency models and coherence protocols and the mechanism for choosing and changing them is flexible then the system would be much more usable for non-expert users. Such a system should make parallel applications as programmable as the semantics of shared memory will allow.

This is precisely what has been attempted in the design of the DSM framework that has been developed in Trinity College. It's structure and use are outlined in the rest of this chapter.

3.3.2 The Distributed Shared Data (DSD) Framework Developed in Trinity College

As we have seen a framework is a set of cooperating classes that must be reused together in order to provide a reusable outline design for solutions to a specific class of problems. The design which this framework attempts to provide in outline is based on the concepts that have been extracted from our consideration of object-based DSM. It aims to allow the user design DSM applications by assigning combinations of consistency models and coherence protocols to individual objects.

The rationale for the DSD framework is to provide a flexible structure for building DSO systems from the most appropriate elements (Weber et al., 1998). In particular it should be able to deliver the following benefits:

Provide a selection of predefined consistency models and coherence protocols
So allowing non-expert users a range of policies without requiring them to extend the framework.

Allow Consistency Models and Coherence Protocols to be combined on a per-object basis
This will enable programmers to take advantage of application-specific semantics on particular applications in a way which is not allowed by other DSM systems. Other systems do not allow individual instantiations of classes to have different protocols associated with them, thus they cannot take advantage of optimisations that can be achieved

through treating each instance of a class type in the manner most appropriate to its role in the program.

Allow customisation of coherence protocols and consistency models

The framework allows expert users who understand the workings of the classes of the framework to extend the base classes to create their own protocols and models in order to achieve optimum performance for particular applications.

Flexibility

It allows easy choosing and changing of the protocols and models that are applied to different variables.

Programmability

It is intended that the framework should make it relatively easy to modify an existing application in order to get it to run over the DSM framework. It should minimise the amount of further difficulty in programming parallel applications other than those that are inherent in parallel programming.

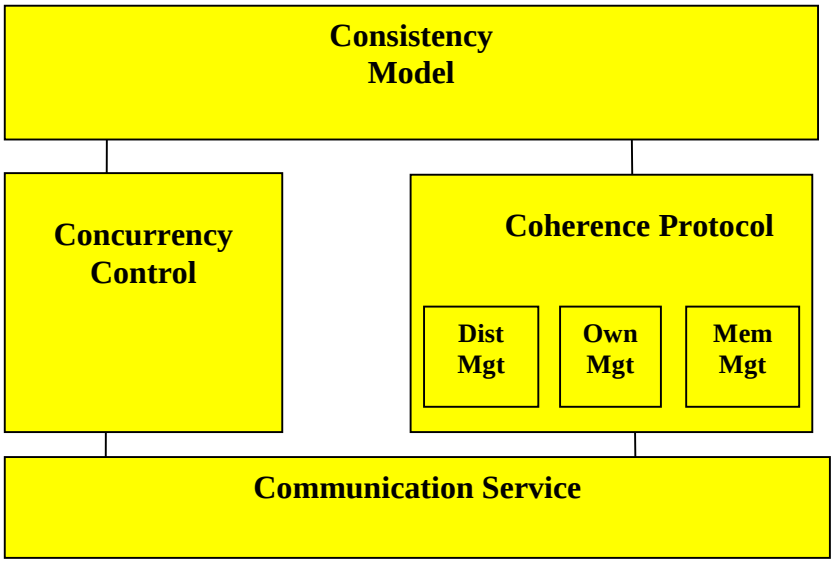
Relative speed-up

The objective of parallelising an application is to reduce the time taken to execute the application by enabling more processing power to be applied to the problem. The relative speed-up is the rate at which the time taken is reduced as more workers are added.

3.3.3 Structure of the Framework

The structure of the framework is as follows (Weber et al., 1998). There are three elements in the DSM: Consistency Models, Coherence Protocols and Concurrency Control. The consistency models and coherence protocols perform the roles that were identified in the last chapter, defining acceptable event-orderings and implementing those event-orderings. The concurrency control implements locks and barriers that are used by the consistency model to control in what order events are seen.

Fig. 3.1 Structure of the Framework



Consistency Model

All consistency models are derived from a base class that defines the common interface for consistency models: the operations read, write, lock and unlock. This ensures that they all can perform the function of a consistency model in the framework. Lock and unlock have corresponding methods in concurrency control. Read and write have corresponding methods in the coherence protocol base class that perform the actual accesses. There are subclasses of the base class for strict and relaxed consistencies. The classes for actual implementations of particular consistency models can then be derived from either the strict or the relaxed consistency model. As we will use Release Consistency and Lazy Release Consistency, these two are explained in more detail below.

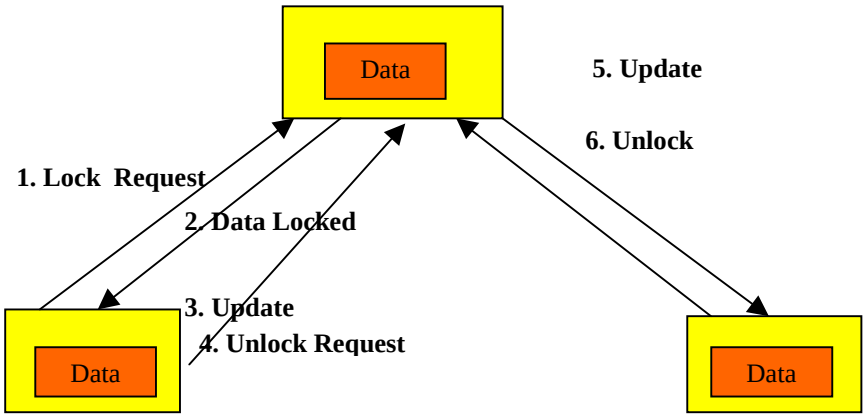
Release Consistency

Release consistency defines two primitives that stand at either end of a critical piece of code: acquire and release. Updates are not forced until the release is encountered, the hope is that several updates will have taken place, and only one message will be required to deliver them.

This is implemented as shown in Fig. 2. A worker wishing to update shared data makes a lock request. The master replies when the lock has been established. The worker updates the master with the new value of the shared data and requests an unlock, (i.e. release is

called) this causes the master to issue an update to all relevant workers and then remove the lock.

Fig. 3.2 Release Consistency

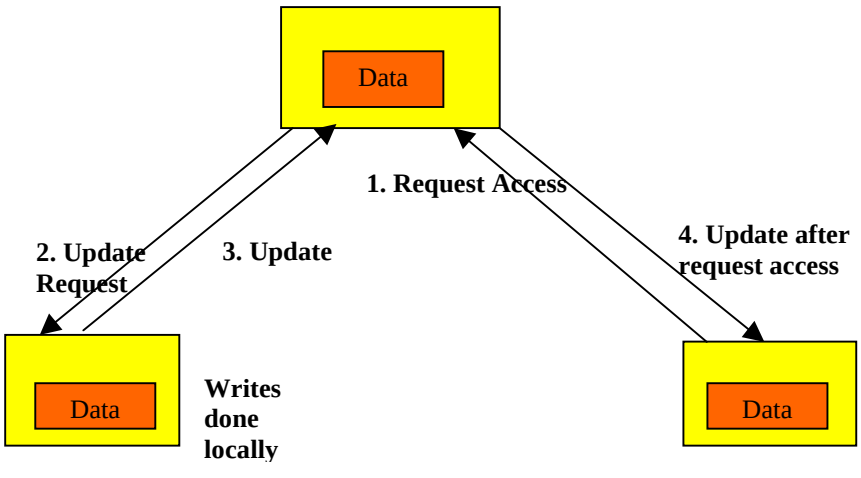


Lazy Release Consistency

Lazy Release Consistency is similar except that updates are not propagated to other workers until they attempt to access the variable. The hope is that many updates will have been made before an access and only the last update will be sent to the worker in question.

This is implemented by having each worker make its updates locally. Then when another worker requests access the master gets the update from the updating worker and it is passed to the worker requesting access.

Fig. 3.3 Lazy Release Consistency



Coherence Protocol

The coherence protocol describes the mechanisms that are used to ensure that the virtual shared memory in DSM is maintained in a coherent state. It is responsible for ensuring that accesses are performed correctly so that the consistency model that has been specified is implemented. There are three elements to the coherence protocol as outlined already: memory management, tracking ownership, and distribution management. They perform the roles that have been outlined above.

Concurrency control

This provides implementations of distributed locks and barriers for use by the consistency model. When a lock is requested it locks all copies of a data item according to the semantics of the particular type of lock requested. At the moment this implements a concurrent read, exclusive write (CREW) lock and an exclusive read, exclusive write (EREW) lock. It also implements a barrier.

Master and Workers

In addition to the DSM per se the framework also includes the master and worker classes. The master issues work to the workers and receives updates back. The workers perform the work on each work item they are sent. At the moment this is the only means of allocating work that has been implemented. However there is no reason within the DSM classes why there should not be some other system, for example workers working together without a master.

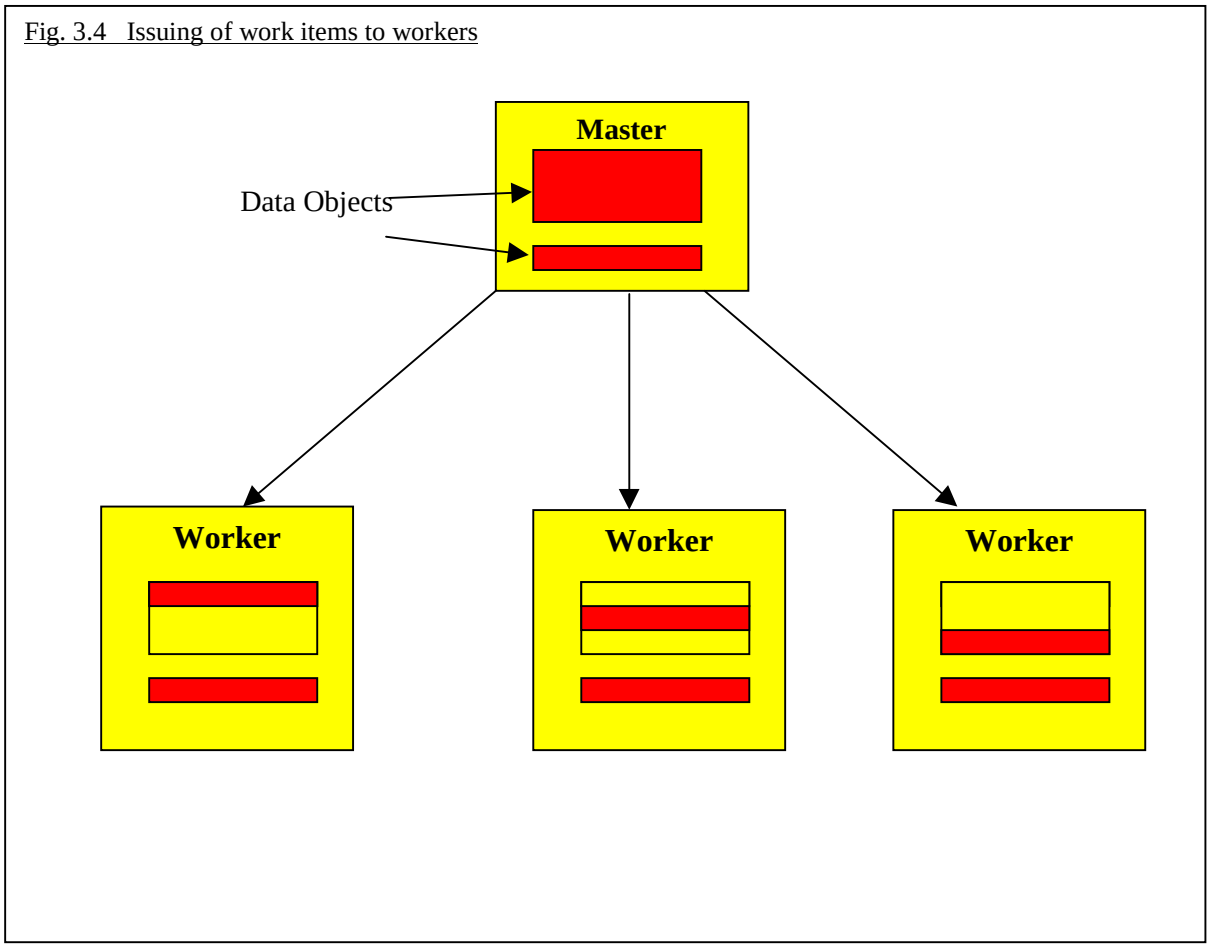
When workers are being initialised at the start of the program there are three modes in which they can be issued data:

Full Distribution:	all data issued at initialisation
Empty Distribution:	update at access time
	(data sent out as worker tries to access it)
Empty Distribution:	update at sharing time
	(data sent out when share method called on that data)

When workers are being initialised they may or may not be provided with full copies of all the data they need to set up the problem, depending on the initialisation mode.

When a worker requests a work item it is supplied with a section of the problem data. This is inserted into its location in the data structure that is has been extracted from (i.e. a matrix row is inserted into the appropriate row of the workers copy of the matrix. The problem is solved by workers requesting and processing work items until all work items have been processed.

Fig. 3.4 Issuing of work items to workers



Not fault tolerant

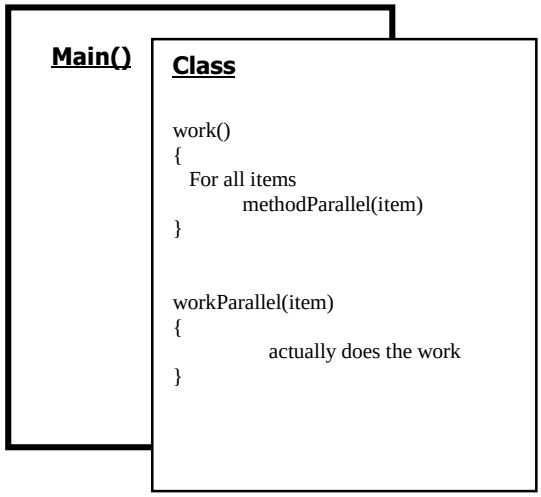
A point worth bearing in mind in regard to the framework is that it has not been designed to be fault tolerant. It was designed to investigate how well the consistency model / coherence protocol structure would work.

3.4 How to Parallelise a Program on the Framework

A simple manual or instruction sheet would ensure that it would be feasible to parallelise a program without needing to understand exactly how the framework handles the different options that are being selected. Of course this would not eliminate the requirement for a general understanding of DSM and of the effect of the different protocols and consistency models, which is required to ensure that the appropriate options are selected when parallelising a program. Nor does it eliminate the need to decide at exactly what points in the execution of the program data needs to be propagated across the system

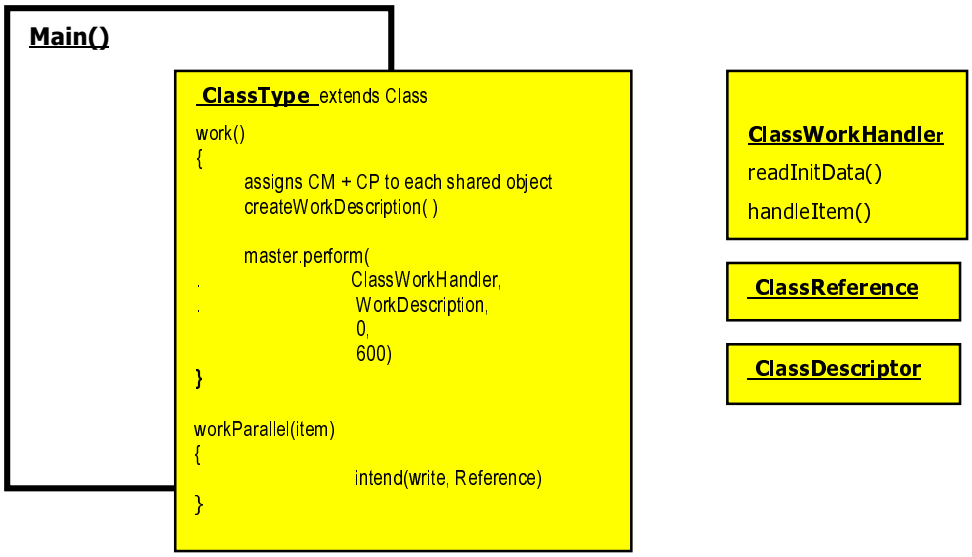
For the application to be parallelisable at all it is necessary that the total block of work to be done can be broken into a number of portions which can be performed in parallel by the different workers, as illustrated in the diagram below. This is not a requirement of this particular system but is necessary for any parallel system, if the work cannot be broken up then it cannot be performed in parallel by a number of processors working simultaneously. I have called these portions of work "items".

Fig. 3.5 Structure of applications that can be run in parallel



The changes and additions that are normally needed to convert a standalone application into a parallel application ready to be run on the framework are illustrated in the diagram below. In certain cases more changes than these may be required, as we shall see later on, however these changes are effective for the majority of applications.

Fig. 3.6 Changes required to run an application on the framework



1 Some lines of code that are the same for all applications to be parallelised must be added to the main method, these provide that method with a reference to the framework's Master which is responsible for managing the workers, and identify the ClassType (see below) either by reading it from a config file or from the command line.

2 The ClassType class must be defined, it indicates the set of coherence model and consistency protocol characteristics to be applied to the variables contained in the problem. Alternative versions of ClassType can be created with different sharing characteristics. In addition it creates a WorkDescription that describes the initialisation data and the method and variables for the workParallel() that the workers are to use.

3 The ClassType class must extend the class defined for the standalone problem and must override its work() method so that instead of calling the workParallel() method for each item that is to be processed it calls the perform() method on the master. This will ensure that the method described in the workDescription is processed by the workers, with the number of the starting item and the number of items to process.

4 The share() method must be called on objects to be shared around the DSM before the start of processing

5 In order to ensure that updates to shared data in the workParallel method are propagated throughout the DSM system it is necessary to declare an “intention” before updating the variable and close the intention afterwards.

6 The WorkHandler class must also be defined. This contains three important methods.

- readInitData() reads the initialisation data coming from the master. It must be defined to extract data in the order in which it was packed in the initialisation parameter.
- handleItem() accepts a work item from the master and then gets the worker to invoke on this item the method specified by the workDescription.
- readEndData() is performed when the master indicates that there is no more work to be done. It does any tidying up that is needed at the end including unsharing shared variables so that remaining updates will be propagated back to the master.

7 The ClassDescriptor and ClassReference classes are two relatively short classes. The ClassDescriptor describes a range of shared data that is to be accessed and the type of access intended. The ClassReference uses that description to perform a read or a write to/from a range of data that has been described.

Overview of changes required

The only changes that are complicated are those to the ClassType, and these can be clearly defined in terms of the steps to be carried out for each variable to be shared across the DSM system. If it is desired to change the sharing characteristics of any one variable, then only four lines need to be changed to select and publish a different pairing of consistency model and coherence protocol for that variable.

Different ClassTypes can be defined in advance with varying allocations of consistency model and coherence protocol to variables. One of these can then be selected simply by passing it as a parameter to the main program, or by including it in a .config file. This helps to ease the process of testing applications for a range of different combinations of consistency model and coherence protocol.

3.5 Summary

The purpose of this chapter was to introduce the concept of a framework, the characteristics of frameworks, their uses and to indicate how they might be relevant to programming parallel applications. Then the DSO framework that has been developed in Trinity College was introduced. Its links to major DSM concepts identified in the last chapter are explained in terms of its major elements, and the relationships between them. Finally the steps involved in the use of the framework in parallelising applications was explained.

4 Testing the Framework

4.1 Introduction

This chapter explains the approach that was used to test the framework and the results obtained. It begins with an explanation of relative speed-up which is the most appropriate measure of how well a parallel programming system provides gains in performance as more nodes work in parallel to execute an application. Then it reiterates the claims put forward for the framework, both in terms of performance and usability. These then provide the focus for the tests that are performed. The two algorithms selected to test the framework are then are outlined and in each case the results and conclusions actually obtained from the process of programing and testing the algorithm are laid out. Finally conclusions are drawn as to how well the framework has delivered on the benefits it claims to offer.

4.2 Relative Speed-up

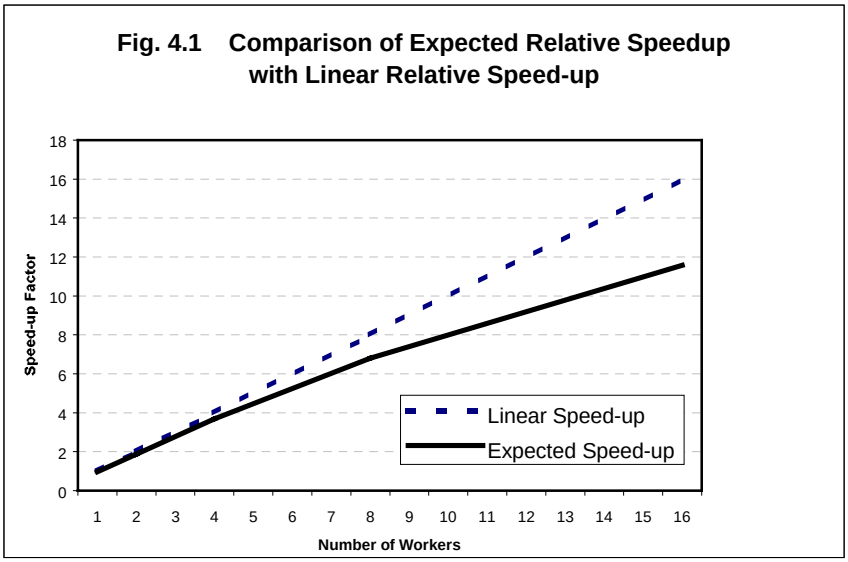
The objective of parallelising an application is to reduce the time taken to execute the application by enabling more processing power to be applied to the problem. Thus the goal in testing a parallel programming system such as the DSD framework would be to produce measurements that indicate how well it meets this objective. The most appropriate indicator of this is a *relative speed-up curve,* which shows how much faster a given problem is processed as more processors work on its solution.

A speed-up curve is obtained by plotting the speed-up value obtained against the number of workers as the number of workers is changed. The speed-up value for *n* workers is defined as

Speed-up factor = the time for 1 worker / the time for *n* workers

Ideally the speed-up curve should be linear and should maintain a ratio of 1:1 against the number of workers. However, because the implementation of a parallel programming system (DSM or any other kind) involves an overhead in terms of management of shared data structures and communication delays between workers which would increase as the number of workers increase we would expect that the speedup will fall away from linear as the number of workers increases in a manner similar to that shown in Fig. 4.1.

Fig. 4.1 Comparison of Expected Relative Speedup with Linear Relative Speed-up



The extent to which the curve should fall away from linear would be affected by the nature of the application. If an application required a great deal of information sharing between workers then it would experience delays due to the communication time required to perform this information sharing. In addition, if the application had certain key global variables which only one worker could access at a time, then there could be considerable latencies introduced with other workers having to wait idle unable to access a variable if another worker is already accessing it.

Amdahl's Law

Amdahl's law defines the maximum relative speed-up that can be gained by parallelising an application, and explains why relative speed-up curves tend to fall away from linear speed-up (Amdahl, 1967). It does this by recognizing that for any parallel application parts of the application will run as a sequential process and only the remainder will run in parallel. Performance improvements due to parallelisation will only apply to the portion of the algorithm that is running in parallel. As more processors are added the time to perform the parallel sections will reduce, but the time to perform the sequential sections will remain the same and so this time will become a larger and larger proportion of the overall time and will limit the relative speed-up that can be attained.

The factors that introduce serial sections into a parallel application are synchronization (which only allows one processor to access a synchronized particular variable at any one time) and communication delays.

Relative speed-up on a parallel programming system is influenced by how much communication and data sharing is involved in the test application. Therefore it is better to test it using applications with significantly different communication and data sharing requirements in order to get results that reflect the capabilities of the system and not just the characteristics of one application.

Gustafson's Law

Gustafson showed how Amdahl's law could be circumvented in practice (Gustafson, 1988). He does not deny its theoretical validity, he just points out that it does not measure what tends to occur in practice. In practice when greater processing power is made available to apply to a problem we do not normally treat this as an opportunity to reduce the time required, rather we use it to increase the complexity and sophistication of our application. He introduced the idea of scaled speed-up. Even if relative speed-up tends to fall off as more processors are applied to a problem, good speed-ups can be obtained when this extra processing power is applied by increasing the size of the problem. Thus by scaling the size of the problem upwards as more processors are added we can still achieve worthwhile speed-ups when we have large numbers of processors involved in an application.

4.3 The Thesis

The thesis of this dissertation is that the framework delivers on the benefits that it promises. As we have seen these are

- Per-object combination of Consistency Models and Coherence Protocols
- Flexibility
- Customisation
- Programmability
- Speed-up

In order to test this thesis, two applications with differing communications and data sharing requirements have been implemented on the framework, they are the Travelling Sales Person and LU Decomposition. The Travelling Sales Person requires very little sharing of data between workers and requires little communication between master and workers. LU Decomposition in contrast requires a great deal of data sharing between workers and a great deal of communication because the data to be shared is substantial. In the following sections on each of these test applications I outline the test problem, the algorithm used to solve it and the speed-up results that were obtained. In addition I will report on how well the framework delivered on the other benefits that it promises to the user.

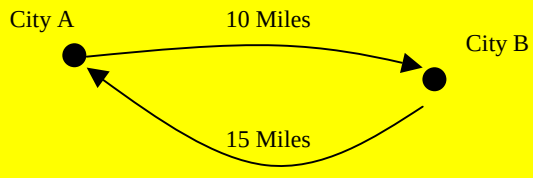
4.4 The Travelling Sales-Person Problem

The travelling Sales-Person problem (TSP) (Taha, 1992) is much more easily described than it is solved. We are asked to imagine that a travelling Sales-Person located at a starting point has to make a tour of a number of cities *n* in order to visit all his clients, his tour must include only one visit to each of the cities and must finish at the city from which he originally departed, city 0. In the interests of efficiency it is desired that we should identify the shortest route that meets the requirements.

This is an instance of a cost minimization problem, the problem stands for any problem where a number of items must be combined in a way that minimizes the total cost, i.e. only the very best solution is considered adequate. For the Travelling Sales-Person the cost is measured in terms of time spent travelling which is non-productive and which the Sales-Person would wish to minimize.

We are provided with a list of cities and distances between them. The distances are supplied in the form of a matrix in which the elements in each row are the distances from the city the row represents to each city in the problem, including the distance to itself on the diagonal. There is one such row for every city. This matrix need not be symmetric, i.e. the direct route from A to B, passing through no other cities on the way, need not be the same length as that from B to A as shown in Fig. 4.2.

Fig. 4.2 Distances between cities in assymmetric TSP problem



All algorithms that have been proposed for solving this problem have shown that the amount of processing required to solve it increases exponentially as the number of cities is increased. This is because the number of possible tours through *n* cities is (*n*-1)!. For example for 16 cities there are 1.3×10^{12} possible routes while for 17 cities the problem is 16 times larger with 2.1×10^{13} possible routes. This means that the TSP is an NP complete problem (Karp, 1972), these are a class of problems for which no efficient solution has yet been found. The time taken to solve the problem increases exponentially for all known

solutions. NP-complete problems are contrasted with so-called tractable problems which can be solved in polynomial time (i.e. the time to solve increases as a polynomial function of the problem size, which increases much more slowly than exponential).

At the moment there are no efficient algorithms for solving NP-complete problems. For this reason they are often used as benchmark problems to test the performance of computing systems. The TSP is one of the most regularly used benchmark NP-complete problems.

There are many heuristic algorithms which can get around the problem of NP-completeness by fairly quickly obtaining a good, but not necessarily optimal, solution to the problem. However our problem is an optimisation problem, we are required to find the very best solution, not merely a "good" solution. Thus our algorithm must be one that can definitively exclude all routes other than the one returned as the shortest route. It must in effect account for every possible route in the problem space, although this does not mean that it has to test every single possible route.

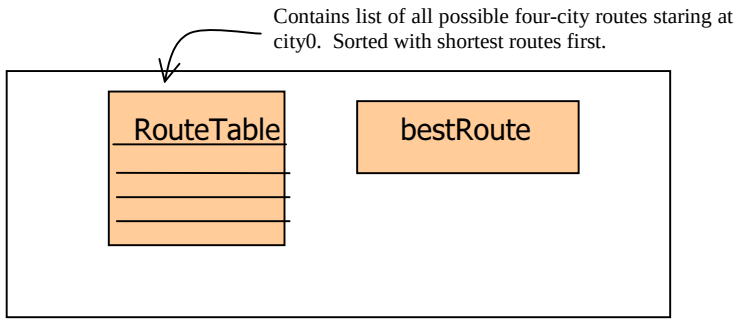
4.4.1 The Algorithm

The algorithm which I designed and implemented for this test was chosen because it fits well with the DSM framework and should offer very good speedup characteristics. It is a depth first tree pruning using a branch and bound algorithm to eliminate non-optimal routes. I have added a number of TSP specific optimisations to reduce the processing time, the details are as follows:

Initial sorting of partial routes

In order to optimise the order in which routes are tested all possible four-city-long partial routes are generated and sorted into a list called the routeTable, with the shortest routes at the start of the list. Then the we repeatedly extract the first (i.e. shortest remaining) route from the routeTable and test to see whether any continuations of that partial route will yield the shortest route (see Fig 4.3 overleaf). The algorithm we use to test this is a branch-and-bound tree pruning algorithm (Taha, 1992).

Fig. 4.3 Outline of TSP algorithm – branch and bound tree pruning



- Extract shortest partial route from RouteTable and test all possible extensions of it until they are longer than bestRoute.
- If a complete route is shorter than bestRoute then it becomes new bestRoute.

Branch-and-bound tree pruning

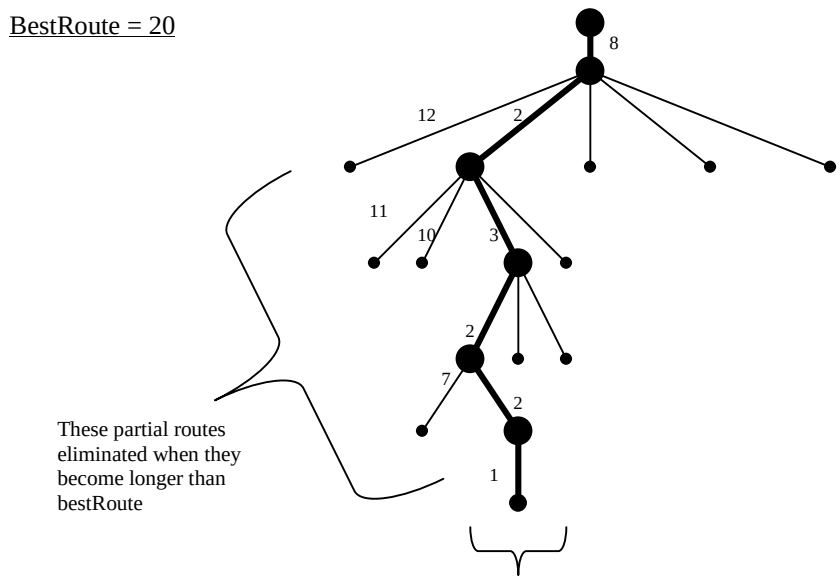
The principle of this algorithm is to eliminate any branches from the tree of possible routes which cannot lead on to the shortest route. This is determined by the branch and bound method. The algorithm starts with a partial route and tests each possible branch on the route, i.e. it considers each possible continuation of a route that could follow from a partial route. It does this depth first, identifying a complete route and then attempting to find other routes that are shorter than it, by working back from the complete route checking other similar routes. At all times the algorithm retains a reference to the shortest route that has been found so far, called the "bestRoute". Each route generated during the tree traversal is compared to it.

If a partial route is longer than the bestRoute then there is no need to test any of the continuations of it because they will also be longer than the bestRoute and so cannot be candidates for the shortest possible route that is being sought. Thus all routes that are connected to this branch are eliminated (i.e. bound out). Through this means, the sooner a short bestRoute can be identified then the more quickly large portions of the problem space can be excluded without having to check each route individually. This produces significant savings in terms of processing time.

If a complete route is shorter than the bestRoute then it becomes the new bestRoute. At the end of the program when all possible routes have been considered the route contained in the bestRoute will be the best possible route for the data set given in the problem.

Fig. 4.4 illustrates how this works. Initially the bestRoute is set at 20. Routes are extended until such time as they become longer than 20, at which time they are eliminated and the next route is tried. When a complete route is less than the bestRoute value as is the case with the route shown in bold, then that route becomes the new bestRoute and any routes longer than it can be eliminated from now on.

Fig. 4.4 Illustration of branch and bound algorithm



Optimisations

A number of optimisations have been added that reduce the number of routes that have to be tested.

- Partial routes are not considered if they are already longer than the best complete route that has been found so far.

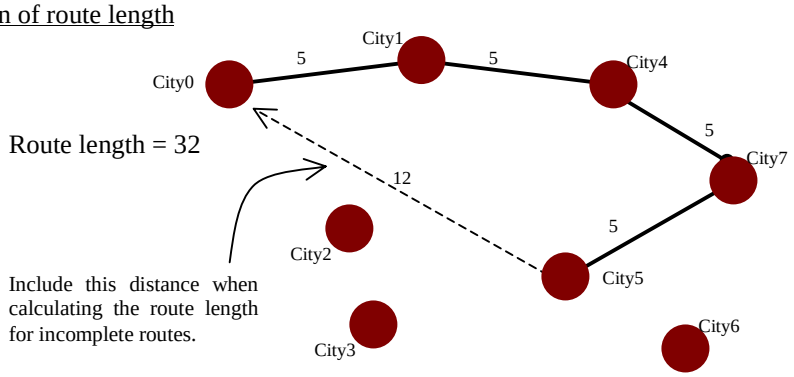
- Before testing the routes an initial guess at a short route is made using the "hillclimb" method (Taha, 1992): starting at city0 we always choose the city closest to the last city to be the next city on the route. This very quick algorithm allows us to produce a reasonably

short route. This ensures that at the start of the algorithm we already have a pretty good route as our best route, thus right from the start we will be able to exclude a large proportion of the non-optimal partial routes: those which are longer than the route found by the hillclimb method.

(It should be noted that when we produce a random distribution of cities the hillclimb will often find the right route without any need for the rest of the algorithm, for this reason the data sets that have been used for testing the algorithm need to have very extreme data in order to ensure that the problem tests all elements of the algorithm. For this reason the data used is that published by Pete Keleher (Keleher, 1996b) as sample data for parallelising the TSP problem.)

- Route lengths are calculated to include the minimum possible distance from the end of the route (if it is a partial route) back to the start. This ensures that long routes are eliminated more quickly, because it accounts not only for the length of the route so far but also for the minimum possible length to complete the route back to the starting point. Thus for the four-city partial route shown in Fig. 4.5 the routelength includes the minimum distance back to the start from the end of the route (12), because any possible completion of this route would be longer than the resulting length: 32

Fig 4.5 Calculation of route length



- In order to ensure that the best candidate routes are tested first partial routes, four cities long, are sorted in order of their length. Each of these are tested in turn starting with the shortest. It is, of course, not necessarily the case that the shortest partial route will be the start of what will be the shortest route, however working on the shortest partial routes first

should allow a short bestRoute to be identified more quickly, thereby reducing the processing required on all subsequent routes. This optimisation is not intended to focus on the best, but rather to screen out the worst.

- Each time a route is being extracted from the routeTable it is tested to see if it is longer than the current bestRoute, if it is longer it is discarded and the next route is extracted. This ensures that workers do not waste time working on routes that cannot provide a new bestRoute.

Features that should enhance relative speed-up

There are three aspects of this algorithm that should help to ensure a good relative speed-up curve: independence of workers, sharing of bestRoute data, and a high computation-to-communication ratio.

Independence of workers

In the case of the TSP algorithm the workers are independent of each other in that each worker can process successfully the partial routes it receives from the master without having any knowledge of the status of any of the other workers. The extension and testing of the partial routes can be performed by comparing them with a local copy of the bestRoute which may be longer than that held by some other workers. This is because the master allows a worker to update its copy of bestRoute only if the worker's copy of bestRoute is shorter than its own copy. Thus eventually the master will be updated with the best possible route.

The independence of the workers means that workers need not wait on each other. If one worker is held up for some reason the others can continue regardless, without affecting the result. This should help to ensure good speedup characteristics. If one worker has a partial route that requires a lot of processing to eliminate routes then the others need not stand idle while it finishes this route.

Sharing of bestRoute data

The more workers we have working in parallel processing candidate routes, the more quickly a short route should be found. If this short route is propagated to all the other workers, they will be able to exclude more routes than before because they will have been provided with a shortest route that is shorter than the one that they have been able to find on their own. At every stage of the algorithm the shorter our best route is the more quickly we will be able to identify that candidate partial routes are wrong and thus exclude them and all extensions of them.

Because the BestRoute does not need to be updated very often and is a very small item of data, there is little communication delay involved in performing this sharing. By ensuring that all workers share the best available version of bestRoute, the parallelised algorithm should achieve close to linear speedup. Indeed as we shall see better than linear speed-up is possible for some data sets.

High computation to communication ratio

Because the items to be passed around the network, i.e. routes, are very small and because the amount of computation required to process each one is relatively large, because the TSP is NP-complete, then there should be low relative delays due to communication so the relative speed-up should be quite good.

4.4.2 Parallelisation of the TSP

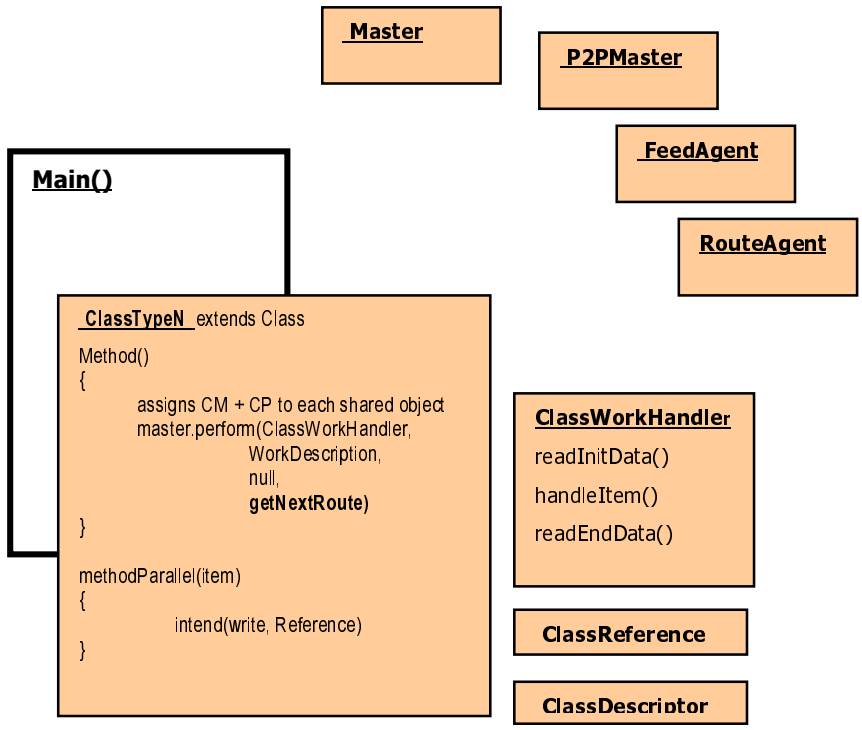
The parallelisation of the TSP algorithm described above proved to be much more complex than the relatively straightforward steps outlined in section 3.4 on parallelising applications. This was because the means by which new items of work (in this case partial routes from the routeTable) were obtained by the workers differed from the standard method.

The standard method assumes that the items are known before any work is done by the workers, and so they can be arranged in an array and selected one by one by passing to each worker the index of the item it is assigned and allowing it to extract the item from the array.

However in the case of the TSP this is not adequate because we do not want to extract every one of the items that is originally added to the routeTable. If any of them are longer than the current value of bestRoute then we want to discard those. This is done by calling a method on the routeTable that performs the check. There is no way in which we can know in advance which partial routes can be ignored by this means.

This required a redesign of the whole mechanism by which work items are issued to the workers, which involved changes to the classes indicated in Fig. 4.6 below.

Fig 4.6 Classes changed in order to parallelise TSP



The key difference was that the perform() method had to be provided with a reference to the routeTable and the name of the getNextRoute() method that was to be called on it, instead of the index of the first item and the total number of items. This change caused a cascade of changes through all the classes that feed work items to the worker. For instance the WorkHandler class had to be redesigned to be provided with a reference to the routeTable on the master during the initialisation stage. It had to call the getNextRoute() method on the routeTable whenever it receives a handleItem from the master.

Customisation of protocols for TSP

A new intention called an "Access" intention had to be added to the read and write intentions. It describes an intention that first reads the value at the master and then decides whether it should overwrite it. This addition was necessary in order to parallelise the setRouteIfBest method which ensures that a worker only updates the bestRoute value of the DSM system as a whole if its value is lower than that currently prevailing.

This change was very easily made. The new intention class was created by combining the functionality contained in the two existing intentions: "read" and "write". The read functionality is performed first. Then the write may or may not be performed depending on the result of the read.

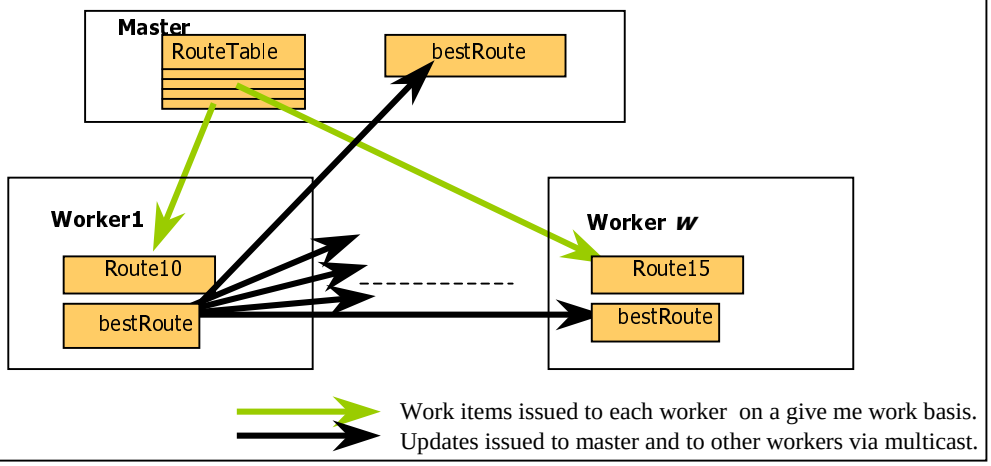
4.4.3 Predicted effect of the protocols

The TSP program was run with both Replication and HomeBased protocols. We would expect significantly different performance for these different types of configurations for reasons which are outlined below.

Replication

The replication protocols perform updates by means of messages that are broadcast to all nodes that are part of the DSM system, as illustrated in Fig. 4.7 below.

Fig. 4.7 Replication protocols



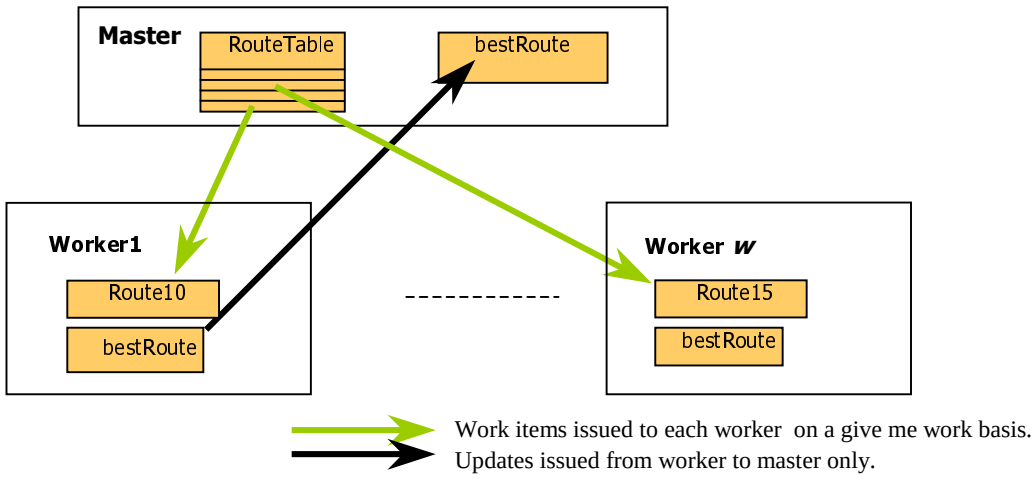
This means that when a worker publishes an intention that it wishes to send an update to the master this update is also sent to the other workers. So all the workers see all updates to the bestRoute as soon as they are propagated to the master. Thus with the replication protocols the DSM system implements sharing of bestRoute data, thus ensuring that the speed-up benefits resulting from this sharing are obtained by the parallel program. This sharing is achieved without having to sacrifice or degrade any of the other optimisations that have been built into the program. For this reason using the replication protocols should offer close to linear speedup.

The TSP algorithm requires only small amounts of communication between the workers and the master, therefore we would expect that there would be little traffic on the network and so Replication2 which uses UDP multicast which is unreliable but lightweight should be faster than Replication3 where LRMP enforces greater reliability at the cost of some management overhead.

HomeBased

The homebased protocols only allow communication between each worker and the master, as illustrated here.

Fig. 4.8 HomeBased protocols



This means that when a worker issues a new lower bestRoute to the master this is not shared with the other workers. Thus we would expect poorer speed-up for either of the two Homebased protocols because as more workers are added all except one will be operating with a sub-optimal value of bestRoute, the proportion of workers working with a sub-optimal bestRoute will increase as workers are added so speed-up should tend to fall further away from linear as more workers are added.

This could be overcome by adding an invalidation feature to the coherence protocol. Whenever one worker updates the master's copy of bestRoute then the other workers copies of the same variable should be invalidated. This would mean that when they next attempted to access their local copy the fact that the local copy was invalidated would cause the protocol to be invoked to obtain the new better value from the master. However at the time that the TSP was being tested this feature had not been added to the framework. The performance of this version would still be slower than that of the replication protocols because performing this invalidation across all copies of the data on the network would require extra time, and would not eliminate the need to perform a separate transfer of data to each worker when they actually want to read an invalidated data item.

This version of the HomeBased protocol would be more efficient that the Replication protocols if it were the case that updates were being made to the bestRoute more often than it was being read. Because then updates would only be propagated to the workers on the rare occasions when they would attempt a read. However this does not apply for the TSP, the bestRoute is only updated a small number of times (less than 15 for our 17-city TSP data set) while the bestRoute is read many thousands of times during the application.

The TSP algorithm requires only small amounts of communication between the workers and the master (because there are few updates and the data to be updated is very small), therefore we would expect that there would be little traffic on the network and so HomeBased1 which uses UDP which is unreliable but lightweight should be faster than HomeBased2 which, using TCP, enforces greater reliability at the cost of some management overhead.

Release Consistency & Lazy Release Consistency

Release consistency should offer better speed-up than lazy release consistency for reasons related to when updates are published. Release consistency publishes an update as soon as it is made. Lazy Release waits until each worker tries to access the shared data and then publishes the update. This takes extra time and so is less efficient in an algorithm like the TSP where the data are being read many more times than they are being updated. If the data were updated more often than they were read Lazy Release would offer significantly better performance.

4.4.4 Results of speed-up tests

The objective of parallelising an application is to reduce the time taken to perform its function by enabling more processing power to be applied to the problem. Thus the objective of testing a parallel programming system such as the DSM framework would be to produce measurements that indicate how well it meets this objective.

As we have seen in section 4.2 the most appropriate indicator of how well a parallel programming system delivers increases in processing power with the addition of more workers is the relative *speed-up curve* which shows how much faster a given problem is processed as more processors work on its solution.

The performance of the parallelised TSP program was tested by recording how long it took to complete the search for the best possible route. This test was repeated ten times for a given number of workers and the speed-up factor was calculated for each test. The average and standard deviation of these ten speed-up results was calculated. A set of ten tests was done for each of the following numbers of workers: 1, 3, 6, 9, 12 and 15.

This procedure was performed for the parallelised TSP program for each of the following configurations of Consistency Model and Coherence Protocol:

Release Consistency, HomeBased Protocol 1
Release Consistency, HomeBased Protocol 2
Release Consistency, Replication Protocol 2
Release Consistency, Replication Protocol 3

The full results are tabulated in appendix 6.1 and summarized in table 1 below:

Table 1

17 City TSP

SUMMARY TABLE

Average Time (s)	**Number of Workers**					
	1	**3**	**6**	**9**	**12**	**15**
RC9 Replication3	686	213	114	81	67	52
RC9 Replication2	678	211	118	85	68	53
RC9 HomeBased1	694	303	191	135	111	99
RC9 HomeBased2	691	315	201	137	120	100

Speedup Factor	**Number of Workers**					
	1	**3**	**6**	**9**	**12**	**15**
RC9 Replication3	1.0	3.2	6.0	8.5	10.2	13.2
RC9 Replication2	1.0	3.2	5.7	7.9	10.0	12.7
RC9 HomeBased1	1.0	2.3	3.6	5.2	6.2	7.0
RC9 HomeBased2	1.0	2.2	3.4	5.0	5.8	6.9
Linear Speedup	1.0	3.0	6.0	9.0	12.0	15.0

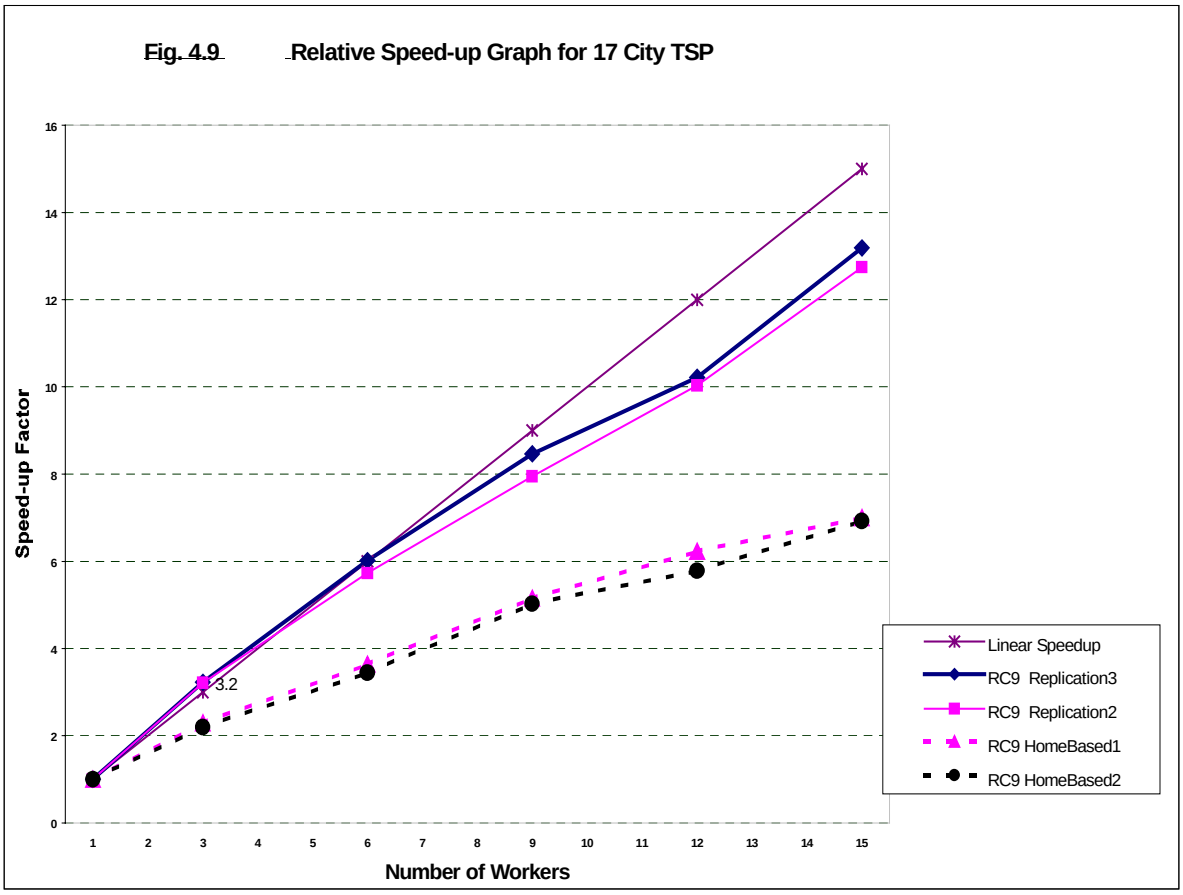
When the speed-up factor is plotted against the number of workers the graph is as illustrated overleaf. The straight line represents linear speed-up where for every doubling of the number of nodes the processing time is cut in half. The other lines show the speed-up achieved by the different configurations of the TSP program.

4.4.5 Analysis of Results

These results allow us to draw a number of conclusions about the speed-up performance of the framework for the TSP algorithm.

Replication protocols better speed-up than Homebased protocols

The two replication protocols achieve near-linear speed-up whereas the two homebased protocols fall away quite sharply from the linear speed-up line. This is in line with what was expected. The homebased protocols do not allow workers to communicate with each other, therefore they do not allow the program to benefit from the optimization that is available through sharing a new bestroute when one is found.



This is the strongest effect that is to be observed from the graph, the difference between homebased and replication protocols is greater than that within these classes of protocols by a ratio of over 20:1. Replication2 and Replication3 differ by 2—3% whereas Homebased1 and Homebased2 differ by over 60%.

Differences within protocols not significant

The differences between Replication2 and Replication3 and between HomeBased2 and HomeBased3 are not significant.

For the homebased protocols this is because both of the homebased protocols have significant levels of variation in the results, the average standard deviation for both of them is over 4% which is greater than the difference in their average times which is only 2—3%.

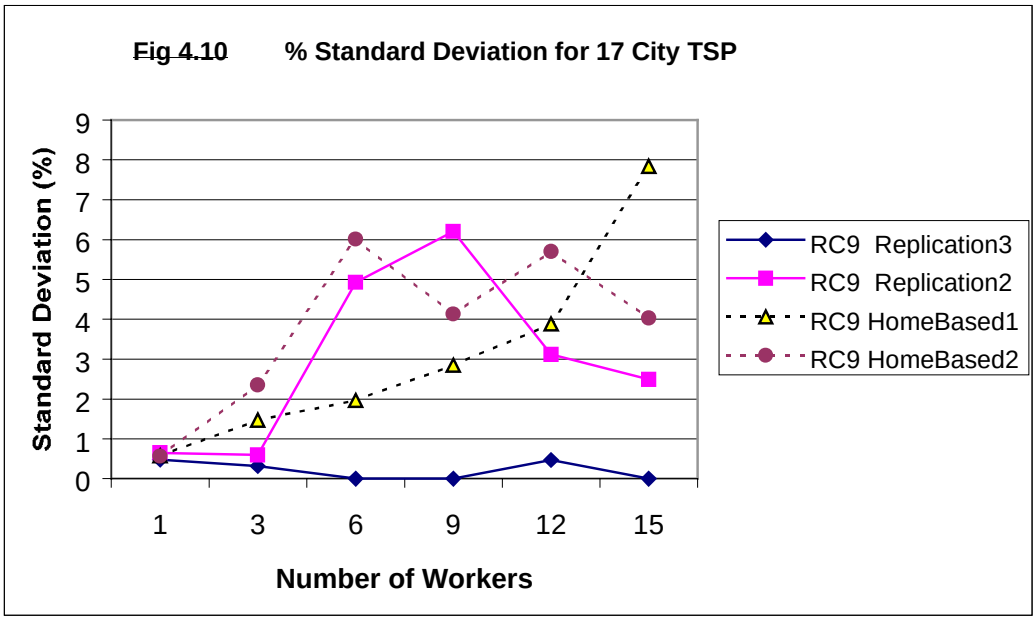
For the replication protocols, although the standard deviation of Replication3 is very low, starting at less than 0.5% and then falling to 0, the standard deviation of Replication2 is

much greater at an average of 3%, enough to cross the difference between the average values

Vulnerability to network traffic

The % standard deviations of the times to complete the TSP algorithm (shown in the graph below) indicate that for all of the configurations apart from Replication3 adding workers tends to increase the variability of the results, this is particularly clearly the case for the Homebased1 protocol. However Replication3, which uses the reliable LRMP multicast protocol, generally maintains a low % standard deviation as more workers are added. Throughout the testing of the TSP program we suffered from external network traffic leaking onto the network where the worker nodes were located (CAG cluster of Linux workstations) due to the fact that this network was only partially isolated from the rest of the college network. These results indicate that the LRMP protocol was most effective at coping with this traffic.

~~Fig 4.10~~ % Standard Deviation for 17 City TSP



Only very slight difference between protocols for one worker.

The fact that there are only very small differences in the processing times for one worker demonstrates that the differences between the protocols for more workers are due to issues with communication and sharing of data, not due to the problem being processed at different speeds. When only one worker is working on the problem there is only communication with the master and there is no sharing of data with other workers. All of

the protocols include communication with the master so they are all doing the same thing for one worker, the slight differences that do exist may be due to the slight differences in the efficiency of the communication mechanisms being used by the different protocols. To ensure a consistent environment for calculating speed-ups the master is always on a separate node with no workers on that node. When there is only one worker then that worker is on a separate node and has to communicate with the master across the network.

Super-linear speed-up possible

The second really noticeable result to come from this data is that it is possible to achieve better than linear speedup. This is a remarkable result because most literature expects some fall off away from linear speed-up due to communication overhead.

On the data that has been presented above there is only one point (3 workers) on each of the replication protocol speed-up graphs that shows super-linear speed-up. In order to investigate this more closely tests were performed for each of these protocols on a greater number of points. The results of these tests are indicated in Table 2 below and on the graph overleaf:

Table 2

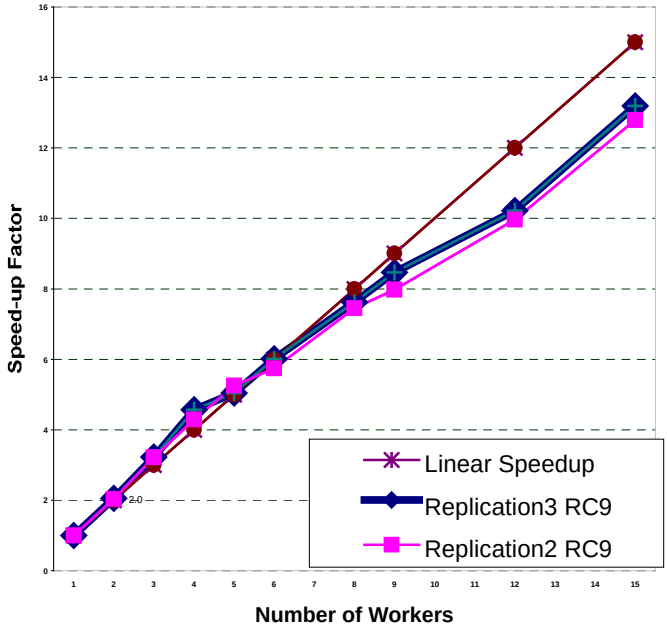
17 City TSP – Additional Data

SUMMARY TABLE

Average Time (s)	**Number of Workers**									
	1	**2**	**3**	**4**	**5**	**6**	**8**	**9**	**12**	**15**
Replication3 RC9	686	334	213	150	135	114	90	81	67	52
Replication2 RC9	678	333	211	158	129	118	91	85	68	53

Speedup Factor	**Number of Workers**									
	1	**2**	**3**	**4**	**5**	**6**	**8**	**9**	**12**	**15**
Replication3 RC9	1.0	2.1	3.2	4.6	5.0	6.0	7.6	8.5	10.2	13.2
Replication2 RC9	1.0	2.0	3.2	4.3	5.3	5.7	7.5	8.0	10.0	12.8
Linear Speedup	1.0	2.0	3.0	4.0	5.0	6.0	8.0	9.0	12.0	15.0

Fig 4.11 Relative Speedup Graph for 17 CityTSP Additional Data



These extra points confirm clearly that the one super-linear point in the previous graph was not an aberration, because now we have other points on either side of 3 workers that show better than linear speedup, for both of the replication protocols. Super-linear speedup is clearly possible for the TSP algorithm, but it is not guaranteed, because it is very dependent on the nature of the particular data set being used.

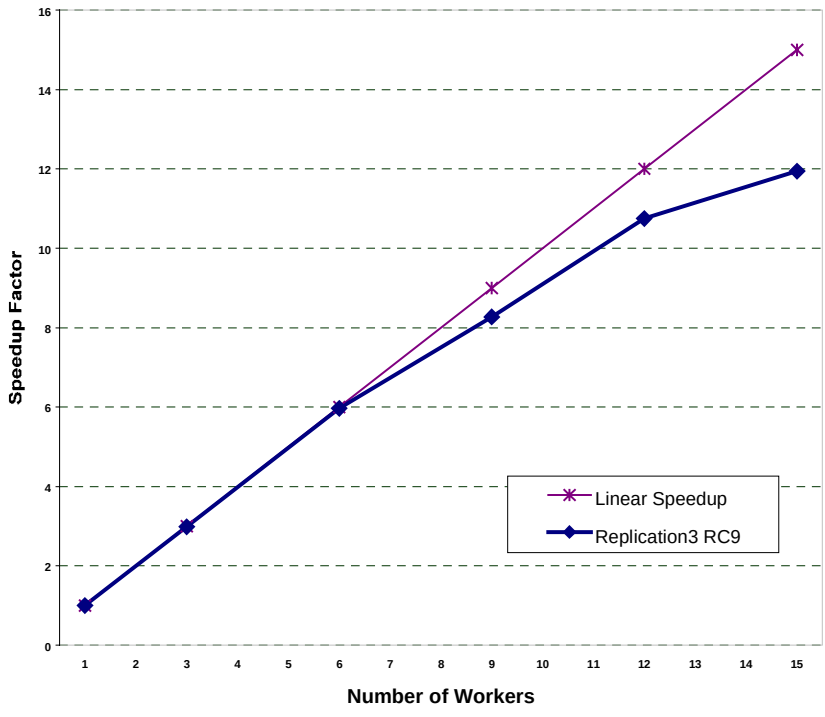
That the super-linear speed-up is not a statistical aberration is demonstrated by the fact that for up to six workers the standard deviation has a maximum of 0.5% for Replication3. Yet the speedup factor at 3.2 for 3 workers is 6.7% above linear more than 13 times higher.

In the next section a number of ways in which super-linear speed-up is possible are outlined, all of which centre around the timing of the finding of a new bestRoute. Finding a new bestroute significantly reduces the time to process a route for all of the workers (when a replication protocol is used). However the actual amount of speedup can vary considerably depending on the circumstances in which the bestroute is found as we shall see in the next section. Given that a new bestroute is found a small number of times during

the running of the program the particular circumstances in which one is found can have a significant impact.

The fact that the levels of speed-up are data dependent is illustrated by the speed-up curve obtained for a 16-city data set which came from the same source as the 17-city data set we have been concentrating on (Keleher, 1996b). In this case, when we use Replication3 and Release Consistency 9, very close to linear speed-up is obtained at the start of the graph, but it never achieves the superlinear speed-up obtained for the 17-city data set. Thus for a different data set we have also achieved very good speed-up but not the superlinear speed-up we obtained before. The 16-city speed-up results graph is shown below. The results are tabulated in appendix 6.3 and the graph is shown below.

Fig 4.12 **Speedup Graph for 16 City TSP**

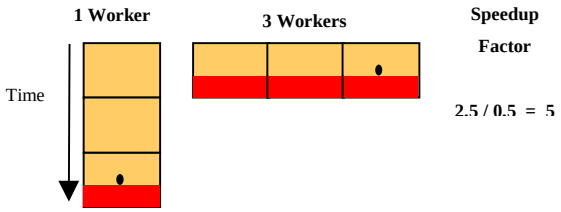


4.4.6 Explanation of Super Linear Speedup

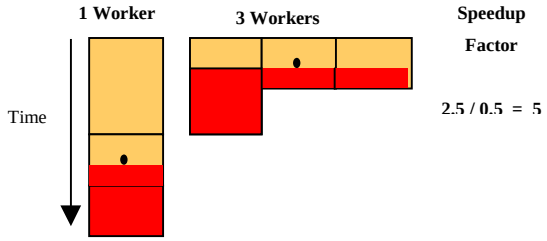
In this section three ways are outlined in which it is possible to achieve super-linear speed-up for a portion of the TSP problem space. Given that the items of work which include a new bestRoute tend to take significantly longer than other items any speedup obtained from them will tend to have a disproportionate effect on the speedup obtained for the problem as a whole.

In order to make the examples clear it is assumed that apart from exceptional routes all routes have the same length, and that the effect of getting a bestRoute is to eliminate any further processing on the current routes.

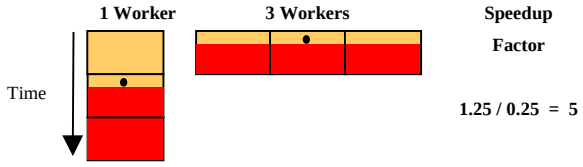
1. A single worker processes routes serially, whereas multiple workers process routes in parallel. When a multi-worker system finds and shares a new bestRoute it could save processing on routes that the single worker system would have processed before finding that new bestroute.



2. The new bestRoute could have been discovered in a route that requires little processing while there is another route that requires far more processing. If there is only 1 worker then it might have to complete the processing of the slow route whereas with more than 1 worker the benefit of the new bestRoute would apply for most of the processing of that route.



3. The new bestRoute could be discovered very early in the processing of one route, in that case the benefit of the new bestRoute would apply to the current route of each worker from a very early stage, whereas with 1 worker many of these routes will have been completely processed before the bestRoute is found.



4.4.7 Analysis of the use of the framework

Flexibility in consistency models and coherence protocols

The mechanisms for selecting consistency models and coherence protocols were found to be very simple and flexible, this aspect of the framework works extremely well. Alternative ClassType classes could be defined with different combinations of consistency models and coherence protocols for different variables. Each of these could be applied from the command-line. This made the process of changing the setup of the program quite simple.

Programmability

Parallelising the TSP algorithm required a great deal more programming than was anticipated. This was because the work items could not be pre-sorted into a list, therefore the already-implemented mechanism for allocating work items (i.e. extracting them from an array using array indices) could not be used. It was necessary instead to provide an alternative mechanism in the framework which allows work items to be extracted by calling a method on the routeTable.

The modifications to be performed to the TSP program itself to were exactly as indicated in section 3.4. The only changes that were required to the body of the TSP's work method

were to call the share() method on the bestroute at the start of the program and to declare an intention before an attempt to update the bestroute, and close this intention afterwards.

Customisations

A new “Access” intention added in addition to the “read” and “write” intentions. This new intention describes an intention that first reads the value at the master and then decides whether it should overwrite it. This addition was necessary in order to parallelise the setRouteIfBest method which ensures that a worker only updates the bestRoute value of the DSM system as a whole if its value is lower than that currently prevailing.

This change was very easily made. The new intention class was created by combining the functionality contained in the two existing intentions: “read” and “write”. The read functionality is performed first. Then the write may or may not be performed depending on the result of the read.

The Release Consistency model was customised to use CREW locks (concurrent read, exclusive write). This was required so that all workers could check the bestRoute value, but only one could update it at a time. The pre-defined class for the distributed CREW lock contained in concurrency control eased the process of making this change. All that was required was for the lock to be applied to the data being managed by the consistency model.

4.4.8 Conclusions for TSP

The conclusions that we have reached from the testing of the parallel TSP program are:

1. Replication Protocols achieve very close to linear speed-up for this algorithm over the range we tested of 1 to 15 workers.

2. Homebased protocols achieve much less speedup because a bestRoute found by one worker is not shared with the other workers.

3. Better than linear speed-up is possible for the Replication protocols. However it is only achieved for certain data sets. Its achievement depends on the fact that finding a bestRoute reduces the problem size for all workers. The data-dependent factors that influence its achievement are: the order in which work items are passed out to different workers, the size of the work items other workers are using when a new bestRoute is found, and how early in the processing of a work item the bestRoute is found.

4. Difficulties with network traffic restricted the amount of time available for testing. Thus full tests were completed only on Release Consistency, no full tests were completed on Lazy Release Consistency.

5. A number of customisations were introduced in order to optimise the Release Consistency model for this application. These changes were eased by being able to reuse existing code for the read and write intentions and for the CREW lock.

6. The flexibility in choosing and changing combinations of Consistency model and coherence protocol enables a range of coherence protocols to be tested. Choosing a replication protocol over a homebased protocol was the factor that had the biggest affect on the speed-up curve. DISOM, the other DSM framework that we mentioned, would have been unable to apply different coherence protocols and so would not have been as appropriate for this application.

7. The framework was found to be flexible and customizable, however it was quite difficult to program this particular application because it does not allocate work items in the manner the framework was designed to accommodate.

8. The difficulties we did have in programming this application were due to the mechanism by which work items were issued. Other than this it was relatively easy to program. Indeed at this stage, with a deeper knowledge of the framework, we can see that there were easier ways to solve this problem.

4.5 LU DECOMPOSITION

LU Decomposition is a matrix transformation problem. Given any square matrix it is possible to transform it into two triangular matrices, one upper triangular and one lower triangular. This transformation is useful for simplifying many matrix algebra problems, e.g. solving a set of linear equations.

In fact because the lower-triangular matrix always has a value of 1 in each diagonal element, it is possible to represent the two triangular matrices using one full square matrix. Because we know the lower-triangular always has 1 on the diagonal we can enter the diagonal elements from the upper-triangular matrix into the diagonal elements of the square matrix. This is convenient because there is an algorithm that can compute this combined lower and upper matrix (hence the description LU decomposition).

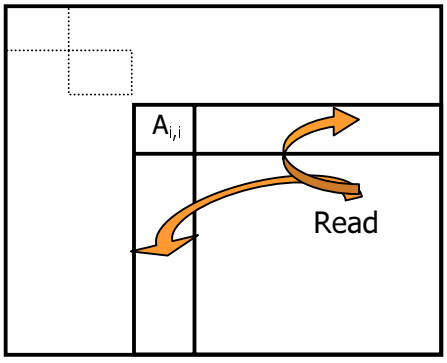
4.5.1 The Algorithm

The algorithm as normally performed requires two stages of computation to be performed on a large part of the whole matrix for each diagonal element in the matrix. These are:

For each diagonal element in the matrix

1. Divide each element in the column below the current diagonal element by that diagonal element.
2. For each element in the square below and to the right of the diagonal subtract from it the product of the corresponding elements in the column and row on which the diagonal lies.

Fig. 4.13 Simple algorithm for LU Decomposition



This algorithm is best subdivided into subtasks for parallelisation by dividing the matrix into columns and treating each column separately. However because it is more convenient to manipulate rows than columns in Java the matrix was inverted before performing the calculations and at the end reinverted to produce the correct result. This allows all operations to be carried out on rows. From this point on the algorithm will be explained in terms of how changes were made to the rows of the inverted matrix.

This algorithm could be programed as:

```
for each diagaonal element a(i)(i)
{
      for j=i+1 to end of row
      {
            a(i)(j) = a(i)(j) / a(i)(i)
      }


      for l=i+1 to end of row
      {
            for m=i+1 to end of column
            {
                  a(l)(m) = a(l)(m) - a(l)(i)* a(i)(m)
            }
      }
}
```

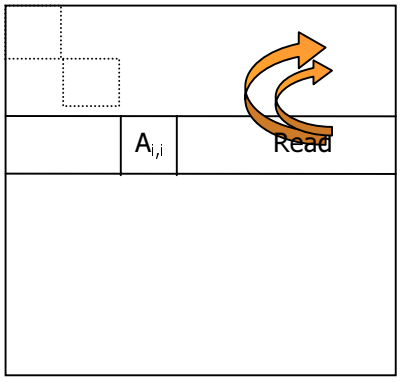
This algorithm could be parallelised by having the workers perform the calculations for a given diagonal on one row. This algorithm is very unsuitable for parallelising because the number of calculations to be done on each row is *n-d* where *n* is the number of elements in the row and *d* is the current diagonal. Each row would have to be updated and returned to the master once for every row above it. This means that the computation to communication ratio is of the order of *n* computations to 1 communication which is too low. The workers in a parallel program using this algorithm would spend most of their time waiting for their results to be returned to the master and for new work items to be delivered to them, not actually working on the problem. The parallel program would be many times slower than a standalone program, thus defeating the object of parallelising the application in the first place.

In order to improve the parallelisability of the algorithm, rather than performing one iteration for each row with each worker updating one row at a time, it would be better to get each worker to perform all the calculations for one row. Each worker is given a row and then performs all the transformations for that row that would have been performed for each of the diagonal elements above that row in the matrix. Thus instead of each worker performing one iteration of row transformation for one row, it performs all the transformation iterations for the row it has been assigned.

Thus for each row a worker will perform the following calculations

```
// For each previous diagonal subtract corresponding row and column
// elements from each element to right of that diagonal on this row.

      for (k=0; k<row; k++){

            for (;rowNotUpdated[k];) {// forces worker to wait until
                                      // other workers have finished
            }                         // updating the row it wants to read

            for (j=k+1; j<n; j++) {
                  A[row][j] = a[row][j] - A[row][k] * A[k][j]
            }
      }


      // Divide all to the right of the diagonal by the diagonal
      for (j=row+1; j<n; j++) {
            A[row][j] = A[row][j]/ A[row][row]
      }

      // Set flag so others will now be able to read this row
      rowNotUpdated[row] = false;
```

Fig 4.14 Better algorithm for LU Decomposition



Thus the number of divisions or multiplications to be done for each row *i* in an *n*x*n* matrix is:

n-j-1 multiplications for each previous row *j*

n-j divisions for each previous row *j*

so the total for each row *i* is:

$$\sum_{j=0}^{i} (n-j-1)$$

$$= \frac{((n-1)^2 + n-1) - ((n-i-2)^2 + (n-i-2))}{2}$$

$$= \text{Order } (n^2/2)$$

This has greatly increased the number of computations per row and ensured that each row only has to be updated once. This should enhance the parallelisability of the algorithm by increasing the number of computations in one visit to a row and reducing communication by only requiring one visit to each row. The combination of these two changes reduces the computation-to-communication ratio which is a good indicator of parallelisability.

However the speed-up characteristics that this algorithm can offer are limited by two facts: the interdependence of workers and the low computation-to-communication ratio.

Interdependence of workers

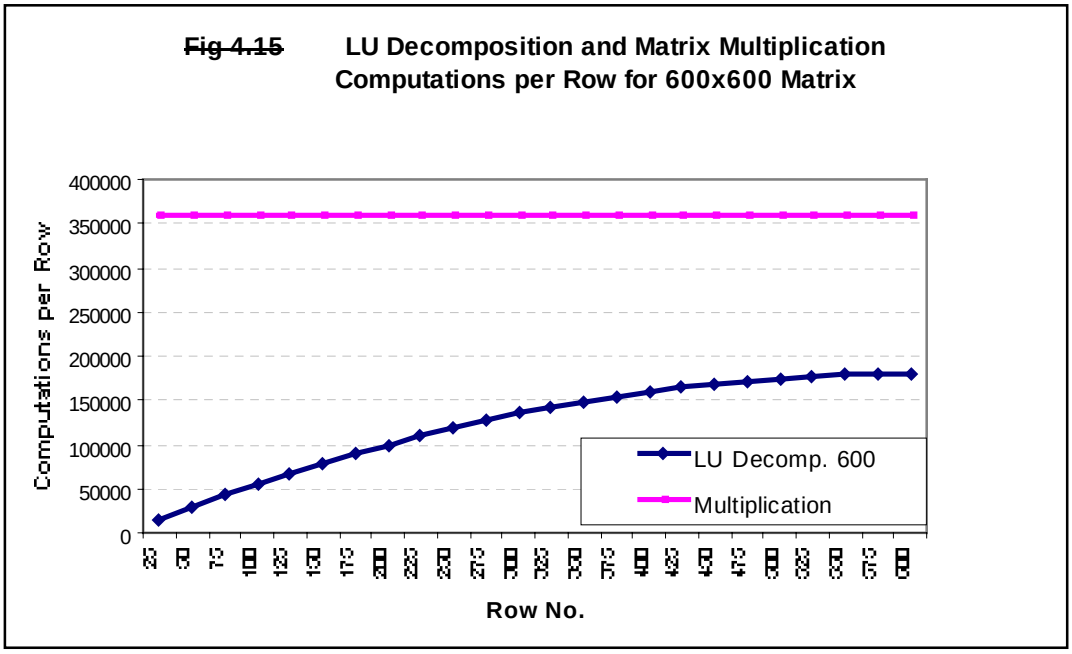
The workers are highly interdependent because, as the fig. 4.14 above shows, in order for a worker to complete the transformations on any particular row it must be able to read the transformed values of every one of the rows above it. Thus no worker can complete a row until all the rows above it have been completed, and so it must wait until all workers have completed all rows above it in the matrix. *If one worker is slow then all the other workers will be delayed*, none of them will be able to complete a row below that workers row until that worker has completed his current row.

Low computation-to-communication ratio

Even with the redesign of the algorithm to improve the communication-to-computation ratio it is still much lower for LU decomposition than it for other matrix applications that have been successfully implemented over the framework such as matrix multiplication (Weber et al. 1998).

The matrix multiplication algorithm that was parallelised involved issuing rows of a matrix A out to workers where they would multiply it by each of the columns of a matrix B in order to calculate one row of the result matrix C. Thus this involved issuing one row of 600 elements for each work item and receiving one row of 600 elements back as the result. (Note that it has been found that an array of 600 elements is the largest that can be reliably passed around the network).

Plotting the computations per row for a 600x600 matrix multiplication and for LU decomposition for a range of matrix sizes results in the graph shown below:



For a 600x600 matrix the number of LU computations per row is at maximum one half of that for every row of the matrix multiplication algorithm. Indeed for the first 100 rows of the matrix the computations per row only get up to one seventh of the computations for each row of the matrix multiplication.

However the amount of communication per row is actually higher in the case of LU Decomposition. For each row of the LU the results of the computation must be propagated to *all* of the workers, and to the master. For matrix multiplication the result need only be sent to the master, not the other workers. Indeed if Lazy Release Consistency is used updates to the master can be delayed until the end for matrix multiplication thereby achieving close to linear speed-ups on this framework, however this is not possible for LU decomposition. LU decomposition cannot allow updates to be delayed until all rows are processed because all workers need to see the results for all previous rows.

Thus LU decomposition imposes a much higher communications burden on the system than does matrix multiplication while requiring only a fraction of the computation involved. Its computation-to-communication ratio is much lower than for matrix multiplication.

For these two reasons we would expect that LU decomposition will not produce speed-up graphs that are anything like as close to linear as those that have been obtained for matrix multiplication and for the travelling sales person.

Attempts to improve the speed-up characteristics

In an attempt to improve the speed-up characteristics by reducing the communication-to-computation ratio the algorithm was changed so that each worker would process several contiguous rows as a single item of work. This reduced the amount of communication by sending a group of rows at less frequent intervals and by ensuring that towards the end of a batch of rows the worker would be reading rows that it itself had processed, so there would be no communication delay in receiving the updated status of those rows. However this meant that the span of the matrix across which workers were currently working was much larger and so the potential for latencies due to workers waiting for other workers to complete their work was much greater. It was found that this change produced no improvement in speed-up.

4.5.2 Parallelisation of LU Decomposition

The LU decomposition algorithm was much simpler to parallelise than was the TSP algorithm. This is because the list of work items that are to be allocated to the workers (for the LU this each row of the matrix) is known at the start of the algorithm. Therefore they can be referred to by their index number in this list and so the method of allocating work that has already been implemented can be used. Thus only the standard classes described in section 3.4 have to be created for LU decomposition.

Customisation of protocols required for LU Decomposition

It was found necessary to customize the Release Consistency model in order to get the application to run properly over the framework. The consistency model, as implemented for previous applications, had been designed so that as soon as a worker had received all the initialisation data it needed to set up a problem it could immediately proceed to processing work items. This eliminated any waiting latency that could be introduced by making each worker wait until all workers had received the initialisation data before any worker could start doing work.

In the case of the LU Decomposition the amount of computation to be done on the first few rows is very low (599 divisions for the first row). This meant that when the first worker to receive all its initial data started working on the first row it would actually finish processing this row while some other workers still had not received all the initial data. Therefore when it published this data, those workers were not ready to listen for that data.

This meant that those workers would never see that the first row had been updated. So they could never process any item because in order to start processing an item they would have to read the updated state of the first row, something which they had missed when it was sent out. All other workers would then be stopped in turn because the stopped worker would never process its item the result of which all the other workers would need to access if they were to process any item further down in the matrix. Thus the whole program would come to a halt after only one or two items had been processed.

This problem was overcome by implementing a barrier which did not allow any worker start processing items until all had received the initialisation data. This was quite simple as a base class for handling barriers has been provided in the barrier server. So this class was just extended to provide the specific functionality of getting all the workers to wait at the barrier.

4.5.3 Predicted Effect of the Protocols

HomeBased Protocols

The HomeBased protocols allow workers to perform updates by means of messages that are sent only to the master, not the other workers that are part of the DSM system. This means that each worker does not see the effect of updates made by other workers, because the worker's local copy of the data is never updated when another worker updates the master. For this reason the HomeBased protocols cannot be used for the LU decomposition algorithm, because this algorithm requires that each worker sees all of the updates made by all other workers.

Replication Protocols

The replication protocols perform updates by means of messages that are broadcast to all nodes on that are part of the DSM system. Thus when one worker updates the master with its transformations on one row then the same data is also multicast to all the other workers as well. This sharing of updated data with other workers is essential for this algorithm because workers need to read the results obtained by all other workers on previous rows in order to complete the transformations for any given row.

We would expect that Replication3 which uses the reliable LRMP multicast protocol would perform better than Replication2 which uses UDP multicast, because of the large volume of data to be multicast (each worker has to multicast a row of 600 doubles at the end of processing one row, and all of the other workers need to receive this data). The replication protocols suit more closely integrated and interdependent algorithms which is exactly what this algorithm is like.

4.5.4 Results of speed-up tests

As already noted the objective of parallelising an application is to reduce the time taken to perform its function by enabling more processing power to be applied to the problem and the most appropriate indicator of how well a parallel programming system delivers increases in processing power with the addition of more workers is the *speed-up curve*.

The performance of the parallelised LU Decomposition program was tested by recording how long it took to complete the transformation of a 600x600 matrix into a matrix combining its lower and upper triangular matrices. This test was repeated ten times for a given number of workers. A set of ten tests was done for the following numbers of workers: 1, 3, 6, 9, 12, 15.

This procedure was performed for the parallelised LU decomposition program for each of the following configurations of Consistency Model and Coherence Protocol:

Release Consistency, Replication Protocol 2

Release Consistency, Replication Protocol 3

The full results are tabulated in appendix 6.4, and summarized in Table 3 below:

Table 3

600x600 LU Decomposition

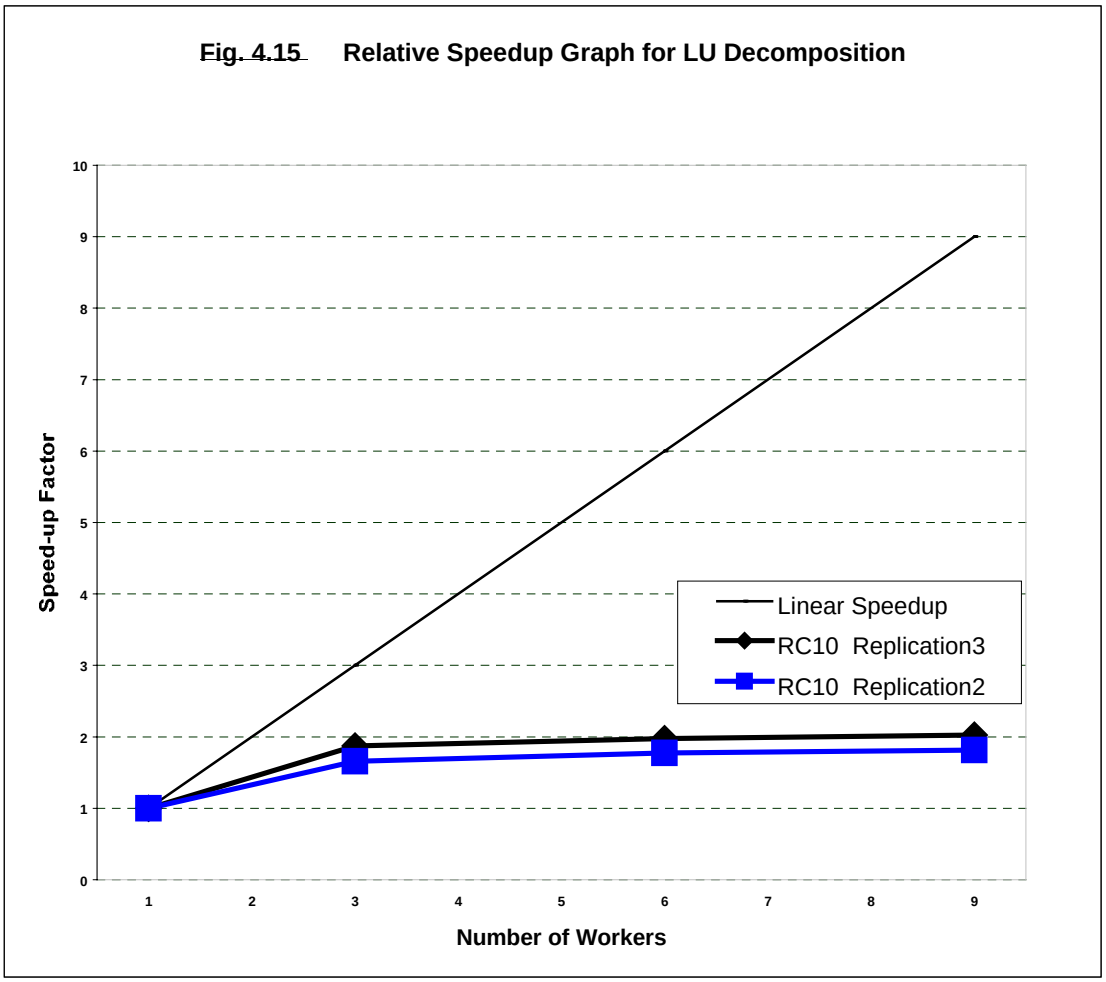
Summary Table

Average Time (s)	**Number of Workers**					
	1	**3**	**6**	**9**	**12**	**15**
RC10 Replication3	75	40	38	37	37	37
RC10 Replication2	78	47	44	43	43	43

Average Speedup Factor	**Number of Workers**					
	1	**3**	**6**	**9**	**12**	**15**
RC10 Replication3	1	1.88	1.97	2.04	2.05	2.04
RC10 Replication2	1	1.66	1.77	1.79	1.80	1.80

The graph of the speed-up factors obtained against the number of workers working on the problem is shown in Fig 4.15 below:

Fig. 4.15 Relative Speedup Graph for LU Decomposition



4.5.5 Analysis of Results

These results allow us to draw a number of conclusions about the speed-up performance of the framework for the LU Decomposition algorithm.

Very little speed-up

There is very little speed-up offered for this algorithm, indeed for more than three workers there is almost no additional speed-up gained by adding more workers. Between one worker and three workers the speed-up is 1.88 for Replication3 and 1.66 for Replication2. The value of 1.88 represents a reasonable speed-up for three workers.

This low value is due to the issues that have already been identified for this application: a very low computation-to-communication ratio (particularly for the first 100 rows of the matrix) and the very high interdependency between the workers.

The difference between the speed-up for the replication protocols and linear speed-up is many times greater than the differences between the two replication protocols as outlined below. For as few as three workers it is six times greater. This indicates that the delays are not primarily due to lost packets, but rather due to latencies introduced by waiting on other workers, the frequency of communication and the volume of data to be transferred on each occasion.

This is reinforced by the fact that the time between rows being processed changes little over the course of the execution of the program, even though the amount of computation for later rows is many times larger than that for the early rows.

Gustafson's Law

By applying Gustafson's scaled speedup (Gustafson, 1988) we should be able to ensure that for larger problem sizes the programme could operate within the portion of the speed-up graph that does produce speed-up. However we were unable to test this due to limitations on the size of arrays that could be passed reliably around the network.

Faster incorporation of updates

The speed-up performance of the framework would be considerably enhanced if the time taken to incorporate new updates at the workers and at the master could be reduced. This code is called hundreds of thousands of times during the application, optimisations to it should have a significant effect. However no work has been done on this at this time.

Reliable multicast faster than unreliable

The Replication3 protocol which uses the reliable LRMP multicasting is on average 15% faster than Replication2 which uses UDP multicast. This difference is much greater than it was for the TSP algorithm and reflects the greater amount of communication that is required by this algorithm. Replication3 reduces the need to resend packets that were not received by enforcing greater reliability.

4.5.6 Analysis of the use of the framework

Flexibility in consistency models and coherence protocols

The mechanisms for selecting consistency models and coherence protocols were found to be very simple and flexible, this aspect of the framework works extremely well.

The availability of the replication protocols was necessary to this application. The homebased were unsuitable for the type of data sharing required in the program. The option of changing coherence protocols was not available in the DSM framework DISOM, so it would have been more restricted in dealing with this application.

Programmability

The programming of the LU Decomposition algorithm so that it could run in parallel over the framework was quite straightforward. This is because the work items are known in advance, i.e. each row of the matrix is one work item, and they can be accessed using array indices i.e. each row can be accessed using its row number. This was simplified by the inversion of the matrix that was performed at the start of the program so that operations

were performed on rows instead of on columns. Therefore the already-implemented mechanism for allocating work using array indices could be applied directly.

The modifications to be performed to the LU program itself to were exactly as indicated in section 3.4. The only changes that were required to the body of the LU's work method were to call the share() method on the matrix at the start of the program and to declare an intention before an attempt to update a row, and close this intention afterwards.

Customization of protocols

As we have seen, in order to get the application to run even with the replication protocols it was necessary to customize the Release Consistency model in order to ensure that all workers had received the initialisation data before the results for the first rows were published. This change was needed because the amount of processing on the first rows of the LU decomposition was almost zero when compared to that required for the TSP and other algorithms which had been parallelised in the past such as matrix multiplication.

In this case the capacity to customise the consistency model was essential. Without doing this the program would not have run at all. The addition of the barrier was the means of stopping the program locking with all workers waiting for a worker that could not proceed.

Programming of the barrier was relatively easy because a barrier class had already been defined in the concurrency control.

4.5.7 Conclusions for LU Decomposition

The conclusions that we have reached from the testing of the parallel LU Decomposition program are

1. The speed-up values obtained for this application were very low. This was due to the very low computation-to-communication ratio.

2. Only the replication protocols were suitable for this application because it requires all workers to see the updates of all other workers, which the homebased protocols cannot provide as they are currently designed.

3. Reliable multicasting produced a 15% performance improvement over unreliable multicasting.

4. The problem was quite easily programmed to run over the framework. No modifications to the program other than those intended were required.

5. The framework was found to be flexible and customizable. The severe communication and synchronisation demands of this application meant that it could never have been run without the ability to customize the framework. For this application the option to use the framework as a whitebox framework was essential to a successful implementation of the parallel program.

4.6 Summary

This chapter explained the approach that was used to test the framework and the results obtained. The two algorithms selected to test the framework were outlined and in each case the results and conclusions actually obtained from the process of programing and testing the algorithm are laid out. Finally conclusions are drawn as to how well the framework has delivered on the benefits it claims to offer.

5 Review of the DSD Framework

5.1 Introduction

In this chapter an overall assessment is made of how well the framework has delivered on the benefits that it promised for developing DSM applications for parallel programming. First it looks at whether the DSD really is a framework. Then what kind of a framework and finally at how well it delivers the benefits it claims to offer to developers of DSM applications.

5.2 Is DSD a genuine framework?

In other words does it meet the definition that we have taken:

> "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems … frameworks provide for reuse at the largest granularity" (Johnson & Foote, 1988)

We can clearly accept that it meets this definition. The DSD is a set of classes that we have used to implement DSM parallel applications with differing sharing and communication characteristics. We have done this by using alternative classes derived from the base classes for consistency models and coherence protocols. Thus the key classes that we have used have embodied this abstract design involving a partition of the problem into two co-operating elements (consistency model and coherence protocol) which can be implemented in many different ways. The reuse of this design is reuse at a large granularity, because what we are reusing is a structure that can be applied to all DSM applications, because it is based on an analysis of the fundamental elements required to deliver DSM.

The characteristics that we identified frameworks as having were: reusability, modularity, extensibility, and inversion of control. DSD has demonstrated all of these features in the development and testing that we have performed. Its basic structure of consistency model and coherence protocol is reused in every application developed using the framework. In addition it has a suite of implemented consistency models and coherence protocols that are available for re-use in parallel applications.

It is modular in that the user can easily replace one consistency model with another in an application, and it is extensible in that the user can develop a new consistency model or coherence protocol which can be applied to variables in exactly the same way as the pre-defined ones. The framework delivers inversion of control because it is the framework that assigns work items to the worker processes and that determines what view they have of the shared data. Therefore we can at least DSD is a genuine framework.

5.3 What kind of a framework is it?

Two kinds of framework were identified: whitebox and blackbox. Whitebox frameworks are used by extending the classes of the framework to create specific functionality. We have seen this done to create new varieties of consistency model and coherence protocol. However the framework can also be used in a blackbox mode because there is a good range of protocols and models predefined and ready to be used and the mechanism for choosing and changing them is relatively easy to use. Therefore we can conclude that the framework combines both approaches, and this is because it is a reasonably mature whitebox framework.

5.4 How effective is it at supporting DSM?

We identified a number benefits that could be expected from the framework given how it was defined.

Providing a selection of predefined consistency models and coherence protocols

We have found that the coherence protocols and consistency models that have been supplied offer significantly different characteristics to the developer. Indeed for the LU program some of them could not be used at all. This demonstrates the benefit of having a range to select from. For the TSP program we saw that there were very significant performance differences between the replication and homebased protocol, showing again the benefits of being able to select a model and a protocol that are appropriate to the application.

In addition the range of pre-defined protocols allows non-expert users a range of policies without requiring them gain the expertise to extend the framework in order to have options in this area.

Per-object combination of consistency models and coherence protocols

This enables programmers to take advantage of application-specific semantics on particular applications in a way which is not allowed by other DSM systems. This feature was not tested in the applications that we used. Many different items were shared but there was no need to have different sharing characteristics for different instances of the same type in either of the test applications that we ran.

Customisation of coherence protocols and consistency models

This feature was particularly important in the LU application, where it enabled the modification of the Release Consistency model to allow the program to run successfully. Also in the case of the TSP optimisations were introduced that allowed each worker to compare routes with the best route that had been found across the whole DSM network.

Flexibility

The mechanism for choosing and changing the combination of coherence protocol and consistency model for different variables worked well and aided the testing process.

Programmability

For both applications it was found that the framework was relatively easy to program, with the exception of the manner in which the work allocation mechanism had to be reprogrammed for the TSP.

Speed-up

Impressive speed-up curves were obtained for the TSP. Speed-up was obtained for the LU decomposition only up to three workers. Again the ability to combine coherence protocols and consistency models and to customise them to the needs of the application were key to obtaining improved speed-ups.

5.5 Problems

Learning curve

There was a very steep learning curve to be climbed at the start in terms of coming to understand how the framework handles an application and passes various data to the workers. However this may have been exaggerated in this case because the application that was done first was the TSP which required going much deeper into the framework than did the LU program. Perhaps if the order had been reversed the learning curve would have been more easily negotiated.

Network interference

The framework was not fault tolerant in respect of high network traffic. However fault tolerance was not one of the key issues that the framework was built to test. For this reason we can say that this is problem does not affect our assessment of how well the framework meets its objectives.

5.6 Conclusion

The DSD framework was very successful at supporting DSM parallel applications. We were able to implement two applications, one of them very demanding on the network. By applying the flexibility that the framework offered we able to run both as parallel applications and to achieve very impressive speed-ups on one of them. According to its authors “the rationale for the DSD framework is to provide a flexible structure for building DSO systems from the most appropriate elements” (Weber et al., 1998). The conclusion of this dissertation is that this rationale has been delivered to a very high extent.

6 Appendices

6.1 Travelling Sales Person 17 Cities

RC9 Replication3	**Nodes**					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	684	213	114	81	67	52
Reading 2	686	213	114	81	67	52
Reading 3	688	213	114	81	67	52
Reading 4	687	212	114	81	67	52
Reading 5	688	214	114	81	67	52
Reading 6	692	212	114	81	68	52
Reading 7	686	212	114	81	67	52
Reading 8	680	212	114	81	67	52
Reading 9	684	213	114	81	67	52
Reading 10	683	213	114	81	67	52
Average	**686**	**213**	**114**	**81**	**67**	**52**
Standard Deviation	**3**	**1**	**0**	**0**	**0**	**0**

RC9 Replication2	**Nodes**					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	680	211	122	85	72	55
Reading 2	674	210	113	99	64	52
Reading 3	681	210	128	80	68	54
Reading 4	674	212	113	84	68	52
Reading 5	679	210	113	80	70	52
Reading 6	675	210	120	85	67	55
Reading 7	687	211	113	86	67	52
Reading 8	683	214	122	81	68	52
Reading 9	674	210	113	89	66	55
Reading 10	674	212	127	84	66	53
Average	**678.1**	**211**	**118.4**	**85.3**	**67.6**	**53.2**
Standard Deviation	**4**	**1**	**6**	**5**	**2**	**1**

RC9 HomeBased1	Nodes					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	699	296	195	130	110	107
Reading 2	688	299	189	133	104	92
Reading 3	694	297	187	138	116	112
Reading 4	690	305	186	136	110	92
Reading 5	698	308	193	129	118	92
Reading 6	690	299	196	139	115	106
Reading 7	691	307	190	140	111	89
Reading 8	697	305	187	133	109	102
Reading 9	694	306	189	137	114	98
Reading 10	698	305	195	132	107	101
Average	**694**	**303**	**191**	**135**	**111**	**99**
Standard Deviation	**4**	**4**	**4**	**4**	**4**	**8**

RC9 HomeBased2	Nodes					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	691	324	215	146	110	103
Reading 2	694	304	184	132	111	101
Reading 3	683	315	200	132	122	94
Reading 4	686	321	188	130	116	99
Reading 5	695	323	217	138	118	103
Reading 6	692	310	204	136	122	104
Reading 7	690	307	187	144	117	102
Reading 8	685	322	210	143	129	95
Reading 9	688	308	193	133	131	103
Reading 10	690	313	208	140	119	94
Average	**691**	**315**	**201**	**137**	**120**	**100**
Standard Deviation	**4**	**7**	**12**	**6**	**7**	**4**

6.2 Travelling Sales Person 17 Cities - Additional Data

Replication3 RC9 Nodes									
	1	**2**	**3**	**4**	**5**	**6**	**9**	**12**	**15**
Reading 1	684	350	213	151	137	114	81	67	52
Reading 2	686	346	213	152	133	114	81	67	52
Reading 3	688	351	213	154	141	114	81	67	52
Reading 4	687	344	212	148	133	114	81	67	52
Reading 5	688	347	214	150	137	114	81	67	52
Reading 6	692	345	212	147	134	114	81	68	52
Reading 7	686	343	212	148	136	114	81	67	52
Reading 8	680	345	212	152	135	114	81	67	52
Reading 9	684	346	213	151	136	114	81	67	52
Reading 10	683	348	213	148	137	114	81	67	52
Average	**686**	**347**	**213**	**150**	**136**	**114**	**81**	**67**	**52**
Standard Deviation	**3**	**3**	**1**	**2**	**2**	**0**	**0**	**0**	**0**

Replication2 RC9 Nodes									
	1	**2**	**3**	**4**	**5**	**6**	**9**	**12**	**15**
Reading 1	674	336	211	158	129	122	85	72	55
Reading 2	679	337	210	159	128	113	99	64	52
Reading 3	675	335	210	159	129	128	80	68	54
Reading 4	687	332	212	157	130	113	84	68	52
Reading 5	683	333	210	158	129	113	80	70	52
Reading 6	674	331	210	159	129	120	85	67	55
Reading 7	674	332	211	158	129	113	86	67	52
Reading 8	674	334	214	160	128	122	81	68	52
Reading 9	682	333	211	158	128	113	89	66	55
Reading 10	677	338	212	157	130	127	84	66	53
Average	**678**	**334**	**211**	**158**	**129**	**118**	**85**	**68**	**53**
Standard Deviation	**5**	**2**	**1**	**1**	**1**	**6**	**6**	**2**	**1**

6.3 Travelling Sales Person 16 Cities

Replication3 RC9	**Nodes**					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	108	36	18	13	10	9
Reading 2	107	36	18	13	10	9
Reading 3	107	36	18	13	10	9
Reading 4	108	36	18	13	10	9
Reading 5	108	36	18	13	10	9
Reading 6	107	36	18	13	10	9
Reading 7	108	36	18	13	10	9
Reading 8	108	36	18	13	10	9
Reading 9	107	36	18	13	10	9
Reading 10	107	36	18	13	10	9
Average	**108**	**36**	**18**	**13**	**10**	**9**
Standard Deviation	**1**	**0**	**0**	**0**	**0**	**0**

6.4 LU Decomposition

Table 4

Replication3 RC10	**Nodes**					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	76	41	38	37	37	37
Reading 2	75	39	38	37	37	37
Reading 3	78	40	37	37	37	36
Reading 4	76	38	39	36	37	37
Reading 5	74	40	38	37	36	38
Reading 6	73	42	39	37	37	36
Reading 7	74	40	38	38	37	37
Reading 8	76	40	38	37	36	37
Reading 9	75	41	39	37	37	37
Reading 10	77	40	38	37	37	37
Average	**75**	**40**	**38**	**37**	**37**	**37**
Standard Deviation	**2**	**1**	**1**	**0**	**0**	**1**

Table 5

Replication2 RC10	**Nodes**					
	1	**3**	**6**	**9**	**12**	**15**
Reading 1	75	46	44	43	43	43
Reading 2	79	47	45	44	43	43
Reading 3	79	46	43	43	43	43
Reading 4	77	48	43	45	44	43
Reading 5	80	46	44	43	43	44
Reading 6	76	47	44	43	43	44
Reading 7	79	47	45	43	44	43
Reading 8	77	48	45	44	43	43
Reading 9	78	46	44	43	43	43
Reading 10	79	48	43	43	44	44
Average	**78**	**47**	**44**	**43**	**43**	**43**
Standard Deviation	**2**	**1**	**1**	**1**	**0**	**0**

7 Bibliography

Amdahl, G.M., (1967). *Validity of the single-processor approach to achieving large scale computing capabilities.* In: AFIPS Conference Proceedings , vol. 30, April 1967, pp. 483—485.

Bennett, J.K., Carter, J.B., Zwaenepoel, W., (1989*). Adaptive software cache management for distributed shared memory architectures.* In: Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90), May 1990, pp. 125—135.

Carter, J.B., Bennett, J.K., Zwaenepoel, W., (1991*). Implementation and performance of Munin.* In: Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP-13) October 1991, pp. 152—164.

Booch, G., (1994). *Object-Oriented Analysis and Design, with Applications. 2nd Ed.* Rational.

Castro, M., Guedes, P., Sequeira, M., Costa, M., (1996). *Efficient and flexible object sharing.* In: Proceedings of the 1996 International Conference on parallel Processing (ICPP'96), August 1996, vol. 1, pp. 128—137.

Coulouris, G., Dollimore, J., Kindberg, T., (1994**)**. *Distributed Systems: Concepts and Design. 2nd Ed.* Addison Wesley.

Dubois, M., Scheurich, C., Briggs, F.A., (1988). *Synchronization, coherence, and event ordering in multiprocessors.* IEEE Computer, 21(2) February 1988, pp. 9—21.

Fayad, M., & Schmidt, D. (1997). *Object-Oriented Application Frameworks.* Communications of the ACM (Guest Editorial), Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, Oct. 1997.

Fleisch, B.D. & Popek, G.J. (1989). Mirage: A Coherent Distributed Shared Memory Design, SOSP12, December 1989, pp. 211—223.

Gamma E., Helm, R., Johnson, R., Vlissides, J., (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J., (1990). *Memory Consistency and Event Ordering in Scaleable Shared-Memory Multiprocessors.* In: Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990, pp. 15—26.

Goodman, J.R., (1989). *Cache consistency and sequential consistency*. Technical Report 61, IEEE Scaleable Coherent Interface Working Group, March 1989.

Gustafson, J.L., (1988). *Reevaluating Amdahl's Law.* Communications of the ACM, vol. 31(5), May 1988, pp. 532—533.

Johnson, R., & Foote, B., (1988). *Designing Reusable Classes.* Journal of Object-Oriented Programming, June/July 1988.

Karp, R.M., (1972). *Reducibility Among Combinatorial Problems.* In: Complexity of Computer Computations, 1972 (Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., Plenum Press, 1972), pp. 85—103.

Keleher, P., Cox, A.L., Dwarkadas, S., Zwaenepoel, W., (1996a). *Treadmarks: Shared Memory Computing on Networks of Workstations.* IEEE Computer, vol. 29(2), February 1996, pp. 18—28.

Keleher, P. (1996b). *The Relative Importance of Concurrent Writers and Weak Consistency Models.* ICDCS16, May, 1996, pp. 91—98.

Lamport, L., (1979). *How to make a multiprocessor computer that correctly executes multiprocess programs.* IEEE Transactions on Computers, vol. C-28(9), September 1979, pp. 690—691.

Lenoski, D.E., Ludon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J.L., Horowitz, M., Lam, M.S., (1992). The Stanford DASH Multiprocessor. IEEEC, vol. 25(3) March 1992, pp. 63—79.

Li, K., & Hudak, P., (1989). *Memory Coherence in Shared Virtual Memory Systems.* ACM Transactions on Computer Systems, vol. 7(4), November 1989, pp. 321—359.

Lipton, R.J., & Sandberg, J.S., (1988). *PRAM: A Scaleable Shared Memory.* Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, September 1988.

Taha, H.A., (1992). *Operations Research.* 5th Ed. Macmillan, 1992.

Tanenbaum, A.S., (1995). *Distributed Operating Systems.* Prentice Hall, 1995.

Weber, S., Nixon, P.A., Tangney, B., (1998). A flexible framework for consistency management in object oriented distributed shared memory. Submitted for publication.