Fault-Tolerant Execution of Computationally and Storage Intensive Parallel Programs Over A Network Of Workstations: A Case Study

J.A.Smith
S.K.Shrivastava

Broadcast Technical Report 103
Esprit Basic Research Project 6360
June 14, 1995

Fault-Tolerant Execution of Computationally and Storage Intensive Parallel Programs Over A Network Of Workstations: A Case Study

J.A.Smith **S.K.Shrivastava**

Department of Computing Science, The University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU UK

The paper considers the issues affecting the speedup attainable for computations that are demanding in both storage and computing requirements, e.g. several hundred megabytes of data and hours of computation time. Specifically, the paper investigates the performance of matrix multiplication. A fault-tolerant system for the bag of tasks *computation structure using* atomic actions *(equivalent here to* atomic transactions*) to operate on persistent objects. Experimental results are described. Analysis, backed up by the experimental results, shows how best to structure such computations for obtaining reasonable speedups.*

1 Introduction

Where a network of workstations is employed for general purpose computing to ensure that each user has a good interactive response, it is observed that there are significant periods of inactivity, e.g. [19]. This gives rise to the desire to exploit the idle workstations in a general purpose network to perform computationally intensive work. Indeed there are many reports of encouraging results obtained using large, and perhaps varying, numbers of workstations for problems executed in this way.

Within a similar context, the work described here focuses on large computations where the data manipulated and communicated is also very large, and scales with the problem, potentially exceeding bounds of primary storage, and so employing disk storage. This paper addresses the question as to whether there is potential gain to be made from executing such computations in such an environment. The example application considered here is matrix multiplication. As well as being data intensive, this is one of the class of *embarrassingly parallel* applications which do not require synchronization between concurrent processes during the course of the computation.

A simple analysis, backed up by experimental results, shows what sort of speedup may be expected from the experimental configuration. The network used consists of HP710 and HP730 workstations connected to a 10 Mbps ethernet and is used in this establishment for general purpose computing.

As the scale of a distributed computation is increased in either the number of participating nodes or its duration, the possibility of a failure occurring which might affect the execution of the computation must increase. For example, the owner of a workstation which is participating in a computation may choose to reboot his machine. If it is not possible to tolerate such an event, it is necessary to restart the entire computation. Whether the computation continues after the fault or is restarted, application data structures on disk should be made consistent.

The second aspect of the work described here attempts to address such issues of fault tolerance through a fault-tolerant implementation of the well known *bag of tasks* structure, which is described in [7]. The fault tolerance is achieved through the use of *atomic actions* (equivalent here to *atomic transactions*) to operate on persistent objects, encapsulated in C++ classes. An overview of the use of objects and actions in structuring reliable distributed systems is given in [21].

From the two threads of investigation in the paper it is finally possible to make some comments on the potential for performing a computation such as matrix multiplication in the way described.

For the purpose of the following discussion, a computation is described as *in-core* if all state is accommodated in physical memory for its duration. Likewise, a computation is referred to as *out-of-core* when, of the whole state residing on secondary storage, only a part is ever resident in physical memory at any time. The distinction is blurred in an environment which supports a virtual address space exceeding physical memory size since data may be declared larger than physical memory size and transparently paged in and out as it is accessed. However, it is convenient to regard this configuration as out-of-core.

2 Speedup Analysis

In absolute terms, the appropriate measure of speedup is that which relates the performance of the parallel computation running on k processors to that of the best possible sequential implementation. This is denoted

$$S_k = T_0/T_k \tag{1}$$

where T_0 is the elapsed time taken by the best sequential algorithm and T_k is the time taken by the parallel algorithm running on k processors. This measure describes the speedup realised through parallelizing the computation. An alternative metric is the algorithmic speedup

$$\bar{S}_k = T_1/T_k \tag{2}$$

which relates the performance of the parallel implementation running with one processor to the same implementation running with k processors. For an ideally parallel application, when plotted against the number of processors, either measure should exhibit a linear relationship with unit gradient, though the latter has the value 1 for a single processor. For the purposes of this paper the speedup referred to by default is the algorithmic speedup $\bar{S}_k$ and the former referred to when necessary as the absolute speedup.

Where hierarchical memory storage is employed, it is found attractive to employ blocked techniques, [12], to maximise locality and thereby gain greatest benefit from caching. Such techniques decompose an operation on large matrices into a combination of operations on smaller submatrices, or blocks. These considerations have led to the development of high performance matrix primitives such as the BLAS library. Also, a block structuring appears to scale well, and so since the work here is concerned with operations on large matrices a block oriented operation seems appropriate. In block oriented matrix multiplication, each block in the product matrix is computed as the block dot product of appropriate row of blocks in the first and column of blocks in the second operand matrices. Each such block dot product may be computed in parallel.

The computation is modelled in two stages. First it is assumed that all matrices reside in memory on one machine, then subsequently the model is refined such that all matrices reside on a disk connected to a single machine. In the model, the user starts a Master process, M, which creates a collection of slaves, S1-Sn, on separate workstations to perform the computation in parallel, see figure 1. The master statically partitions the computation and informs each slave of its unique allocation of work. The three matrix objects are located on one machine, initially assumed to reside in memory, but subsequently assumed to reside on disk.

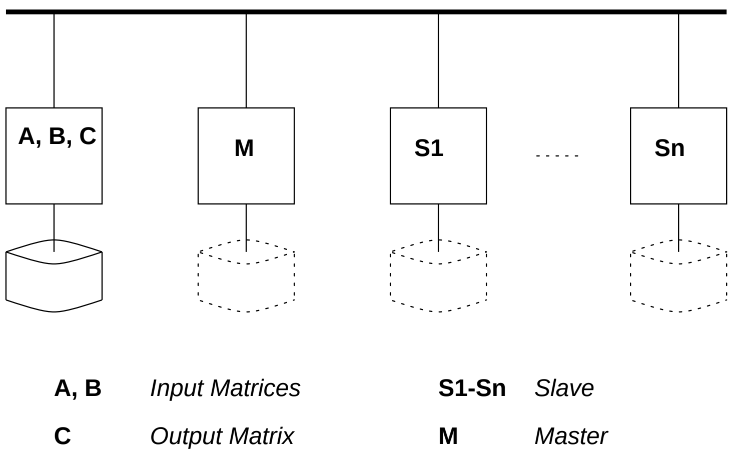

Figure 1. A Model for Distribution of Parallel Matrix Multiplication

It is assumed that all three matrices are square with n^2 elements and are partitioned into p^2 blocks, each containing n_0^2 elements. A block is accessed remotely with a cost t_x this being the memory to memory transfer time between two machines, and assumed equal for put and get operations. It is assumed that communication is through a common medium, here the ethernet and that all such transfers are serialised so that the full bandwidth of the medium is exploited for the duration of the transfer. This is because it is anticipated that the potentially large number of collisions resulting from uncoordinated access to the ethernet used in this experiment would be detrimental to performance. Thus each block access operation implies sole access to the ethernet for its duration. The time to compute the product of two blocks is t_c. A single block of the output matrix is the dot product of entries in a block row of the first, and block column of the second, input matrices. Therefore, the time to compute a single block of the product of two square matrices of width p blocks is $(2p+1)t_x + pt_c$. Since there are p^2 blocks in the output matrix to be computed, the time taken by a single slave to complete the computation when accessing objects remotely is

$$T_1 = p^2((2p + 1)t_x + pt_c) \tag{3}$$

The other parameter of interest is the attainable speedup with addition of extra slave processes. As suggested earlier, for an ideally parallel application, the speedup should be linear with unit gradient. However, in this experiment, all communication is serialised and in addition the communication medium has a limited bandwidth. In the presence of a solitary slave, the utilisation of this medium is given by the ratio:

$$\frac{task\ communication\ time}{task\ elapsed\ time}$$

As the number of slaves is increased to k, this fraction increases by a factor k. A limit occurs when the shared communications medium is fully utilized, such that

$$k = \frac{task\ elapsed\ time}{task\ communication\ time} \tag{4}$$

If more slaves are added beyond this point, then the total time taken in data transport throughout the computation cannot decrease. In this very simplified model, speedup grows linearly up to the cut off point and then remains constant. In the example of matrix multiplication described above, the task communication time is $(2p+1)t_x$ and the maximum speedup is:

$$\bar{S}_{max} = 1 + \frac{pt_c}{(2p+1)t_x} \tag{5}$$

The estimates computed so far do not take into account the cost of disk access. For the second stage of analysis, a single disk store is assumed attached to one of the machines which acts as a server. The input and output matrices now reside on this disk. When accessed through a file system such as that of UNIX, disk transfers are buffered and writes not necessarily completed before a synchronising primitive such as *sync()*. Similarly input is via buffers which effectively act as a cache. The filesystem cache is at the level of disk rather than application level blocks, but clearly reading from cache is likely to be cheaper than reading from disk. In following discussions unqualified references to blocks always refer to application level blocks, i.e. submatrices. Separate terms are introduced; t_{cr} for reading a block from cache, t_{dr} for reading a block from disk and t_w for writing a block to disk. Initially, it is assumed still that complete block accesses are serialised, such that the time to read a remote block is $t_{cr} + t_x$ from cache or $t_{dr} + t_x$ from disk and to write a remote block $t_w + t_x$. There are two limiting cases, the first where all required blocks may be accomodated in the cache such that the only disk access cost is to initialize the cache, and the second where no benefit is gained from the cache.

- If it is assumed that all blocks are accomodated in the cache, then only the initial read of a block is from disk. For this to be the case, the overall size of the operand matrices must be small enough that both nay be accomodated in cache. The total number of block reads is $2p^3$ and the number of unique blocks, and therefore the number of block reads which are from disk, is $2p^2$. Then the single slave computation time is given by the following.

  $$p^2(2(pt_x + (p-1)t_{cr} + t_{dr}) + t_x + t_w + pt_c) \tag{6}$$

  Similarly, the estimate for the maximum speedup is now

  $$1 + \frac{pt_c}{2(pt_x + (p-1)t_{cr} + t_{dr}) + t_x + t_w} \tag{7}$$

  In this case, as the matrix size is increased and p becomes large, the limiting speedup tends to:

  $$1 + \frac{t_c}{2(t_x + t_{cr})} \tag{8}$$

- The other extreme is where all reads are from disk. This is the case if the cache is not large enough to accomodate a single block. However, if even a single block is accomodated in cache then there is some chance that two slaves each request the same block, one after the other. Assuming, all reads are from disk however, the single slave time is:

$$p^2(2p(t_x + t_{dr}) + t_x + t_w + pt_c) \tag{9}$$

and the maximum speedup:

$$1 + \frac{pt_c}{2p(t_x + t_{dr}) + t_x + t_w} \tag{10}$$

with the limiting speedup being:

$$1 + \frac{t_c}{2(t_x + t_{dr})} \tag{11}$$

It is also possible to compute the expected time to perform the computation out-of-core on a single machine, simply by removing the communication cost from equation 6 or 9 respectively:

$$p^2(2((p-1)t_{cr} + t_{dr}) + t_w + pt_c) \tag{12}$$

and

$$p^2(2pt_{dr} + t_w + pt_c) \tag{13}$$

In the parallel configuration, performance is limited by the least of disk and communication bandwidths. While write operations must at least eventually be to remote disk, there is a possibility for allowing read operations to progress in parallel with write operations if the server machine has large memory allocation. While caching inevitably occurs through file system buffer space, explicit block level caching is not considered further in this paper.

Before actual values can be derived for the estimates above, it is necessary to measure the various primitive operation times, t_c, t_x, etc. This is described, in the remaining parts of this section.

2.1 Experimental Values

The machines to be employed in the computation are HP710 and HP730 workstations with 32 Mbytes and 64 Mbytes of physical memory respectively and running HPUX 9.01. The block size is effectively bounded by memory availability. To perform a block product it is necessary to hold a little over two blocks in memory. In addition, to compute a block dot product, the sum must be held in memory. Since the compute machines are all HP710 workstations with 32 Mbytes of memory, it seems unlikely that a block size of more than 1000 square, i.e. 8 Mbytes, may be employed.

2.1.1 Computation Cost

A matrix multiplication primitive is implemented as *operator*=()*, using the conventional dot product algorithm, for a C++ template class, **Matrix**, instantiated for *double*. The performance of this operation is shown in figure 2. The duration of the entire computation is expected to be proportional to the cube of the matrix width. For this selection of matrix sizes, this appears to be so, though performance is better for small matrices, where presumably processor level caching is of benefit. Since a 250 square matrix computation is completed in about 3.38s, it appears that the effective computation rate for larger matrices is about 4.6

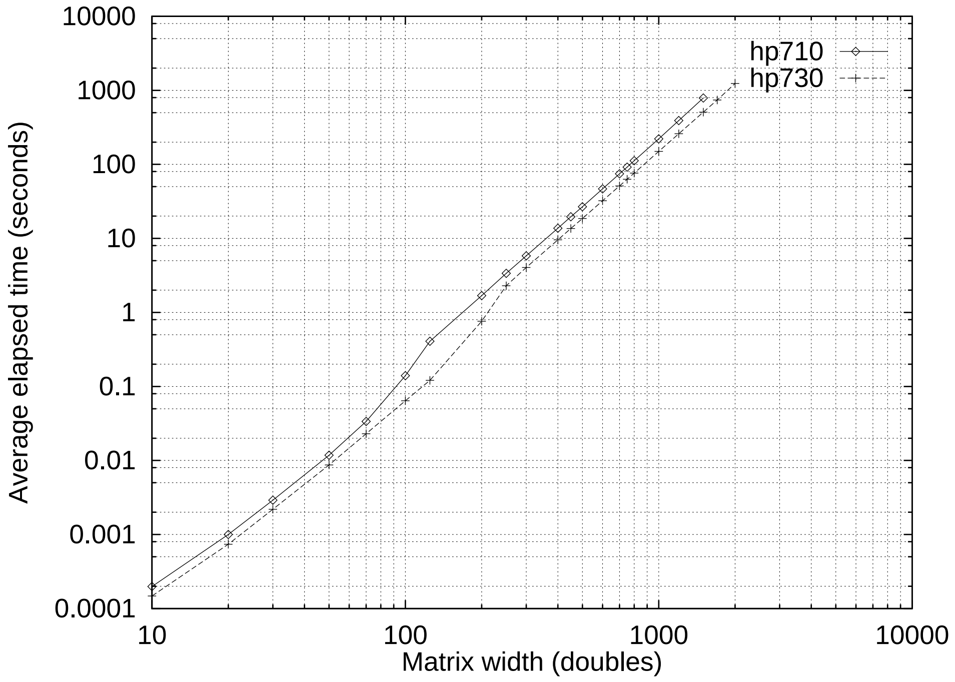

Figure 2. In-core Matrix Multiplication. Both axes are logarithmic. The values plotted in the graph are average of at least 10 measurements.

million floating point multiply operations per second. The implementation of this operation aims to minimize the memory requirements for large matrices. While some care is taken over the implementation, it is not claimed that no further performance improvement may be gained, either through lower level blocking to exploit processor cache or faster algorithm. Ultimately, it would seem preferable to employ a library primitive, such as from BLAS.

2.1.2 Communication Cost

Assuming that the ethernet bandwidth of 10 Mbps may be used completely, a transfer rate of 1.25 Mbytes per second is possible. A part of this bandwidth is taken up with protocol headers however. A simple experiment is performed to measure the practical transfer rate. The procedure is to make transfer by TCP, setting up a new connection for each transfer. In general, such a procedure is likely to be expensive, but here where all transfers are sizeable the cost is acceptable. Employing no buffer copying at either end, the maximum rate observed for transfers of 1 Mbyte upwards from memory on source machine to memory on the destination machine is about 1 Mbyte per second. Both machines are HP710. This is not inconsistent with a study of communication rates in a range of network parallel programming environments, [10].

2.1.3 Disk Access Cost

The performance of various primitive disk operations is measured and the results presented in Figures 3-5. Locally mounted disks are used for these and subsequent tests, to avoid any further cost associated with NFS paths. The operations measured are; write new file data, read file from disk cache, read file from disk. It is assumed that application level block access equates to these basic operations. These operations are timed on the two alternative disk configurations used in subsequent experiments, i.e. 710 boot disk and a larger

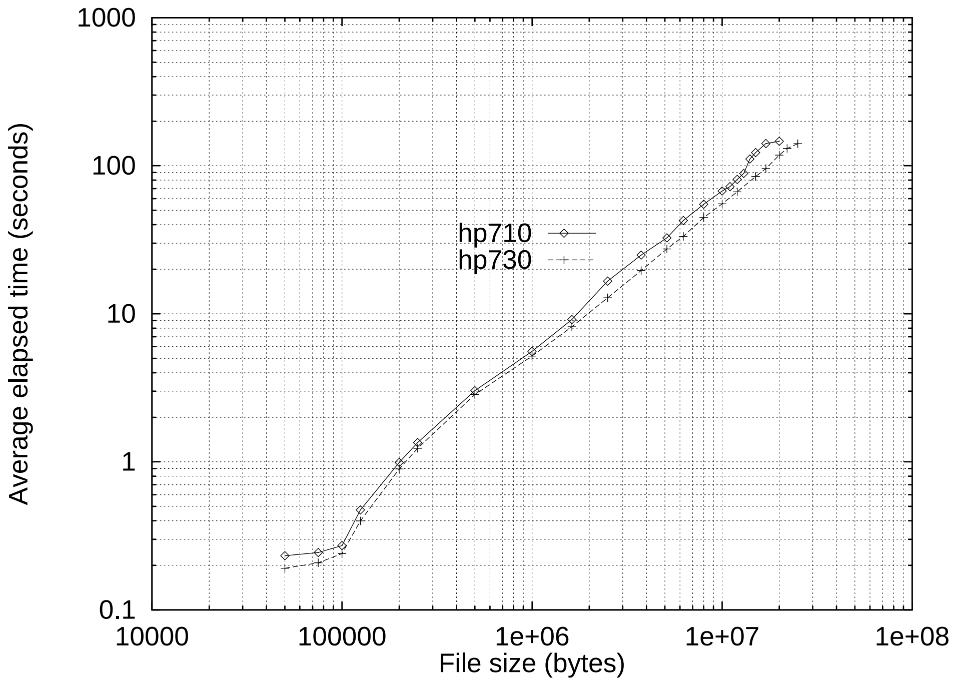

Figure 3. Performance of write new file operation. The values plotted in the graph are average of at least 3 measurements.

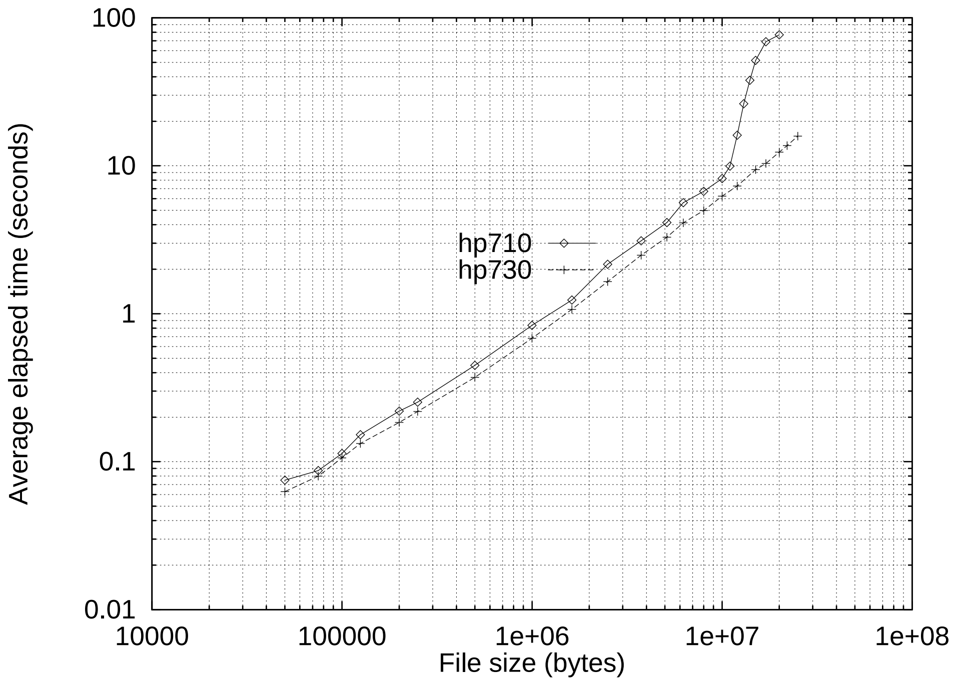

Figure 4. Performance of disk file read operation. The values plotted in the graph are average of at least 3 measurements.

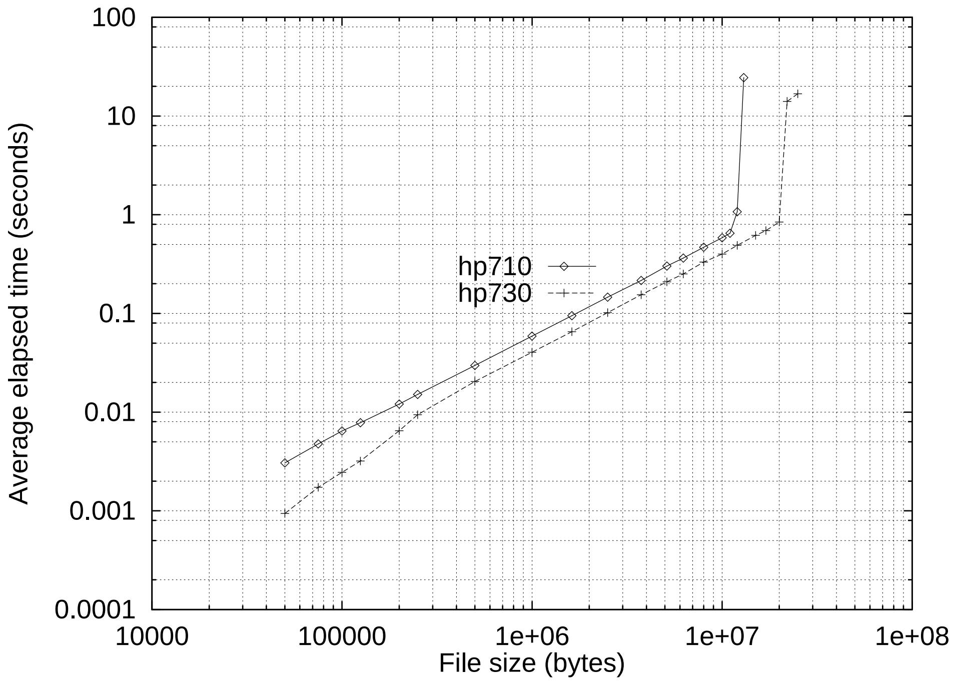

Figure 5. Performance of cached file read operation. The values plotted in the graph are average of at least 3 measurements.

disk mounted locally on /tmp on the HP730 machines.

It is seen that reading a cached file is over 10 times cheaper than reading the same file from disk. The cost of reads in a particular computation is dependent on the scale of the computation, i.e. the extent to which benefit is gained from caching. This provides justification for considering the two extreme cases earlier, though clearly it would be desirable to maximise cache benefit.

The remaining basic operation, which entails overwriting existing file space with new data appears, through measurement, to have similar cost to the disk read, but is not considered further in this particular application which is creating a new matrix as the product of two existing matrices. The reason for the significantly higher cost of writing new file space is presumably the need to find free blocks and update file structure data.

For a large range of file sizes these results appear roughly linear. For a first approximation it is possible to derive expressions in terms of n_0, the block width in elements, for the disk access parameters, t_w, t_{dr}, t_{cr} by fitting lines to the most linear range of each set of measurements. The range 100 Kbytes to 10 Mbytes is employed in this work. While for smaller block sizes, there is certainly some error, the smallest block size used in subsequent experiments is 125.

2.2 Application Level Measurements

A prototype of the computation is implemented based on the Arjuna tool kit, [20]. Arjuna is a programming system which supports construction of distributed applications which manipulate persistent objects (encapsulated in C++ classes) using atomic actions. Typically, these facilities are employed to implement a fault tolerant application, but this first implementation exploits only support for persistent objects. The shared matrices A, B and C are each implemented as a collection of persistent objects. Each such object encapsulates an instance of the class **Matrix** mentioned in section 2.1.1. When not in use, the state of a persistent object is held on disk storage in an object store. When an object is activated, its state is loaded into an object server automatically. A single server is employed to serve all matrices, thereby implementing the desired arbitration of the ethernet. Distribution in Arjuna is supported through an RPC, and the version employed here supports optional use of the TCP protocol with connection establishment on a per-call basis. Some optimisation of this RPC mechanism has been performed to exploit homogeneity of machines. In this implementation, the slaves are created within separate processes forked by the master at computation startup.

Available local disk space is limited to about 30 Mbytes on the hp710 machines. Since doubles are 8 bytes, a convenient size to choose for the matrices is 1000 square. The total size of the data contained in the three matrices is then 24 Mbytes and some space is required for object store overheads and the bag object. Since the overall matrix size places an upper bound on the block size for a given number of workers, it would be preferable to employ larger matrices, but for even 1100 square matrices, the data space alone would be over 29 Mbytes. For the somewhat small size of 1000 square, the in-core sequential time is measured at 221s on the HP710 machine.

In the graph shown in figure 6 measured times for the distributed computation run with two alternative block sizes are shown. The experiment employs

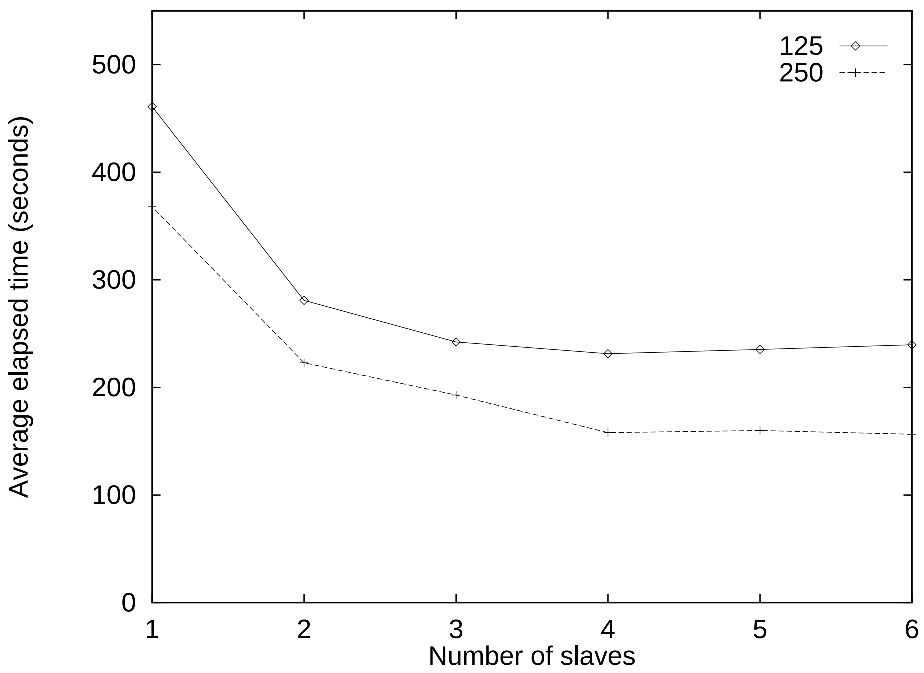

Figure 6. Parallel Multiplication of 1000 square Matrices with Block Sizes 125 and 250. The values plotted are averaged over 5 measurements.

1000 square matrices and block sizes, or granularities, of 125 and 250. The two input matrices exist already on disk at the start of the computation, but the output matrix is created in the course of the computation. Each slave is parameterized with the total number of slaves and a unique integer value between 0 and the number of slaves - 1. Then each slave computes a unique set of blocks of the output matrix, and in the absence of failure, the computation completes. From equations of section 2.1.3, the expected values of single slave time and maximum speedup for both maximum and minimum disk cache benefit are computed and tabulated in table 1 along with the measured values. For this

Block Size (bytes)	Derived				Measured	
	all blocks cached		no blocks cached			
	T_1	$\bar{S}_{max}$	T_1	$\bar{S}_{max}$	T_1	$\bar{S}_{max}$
125	381	2.31	476	1.83	461	2.0
250	348	2.64	386	2.27	368	2.3

Table 1. Comparison Of Derived And Measured Results: Single slave time, $T1$ in hours and maximum speedup $\bar{S}_{max}$ for multiplication of 1000 square matrices

size of matrix, it seems reasonable to expect some benefit from cache hits.

Figure 7 shows measurements of the fault-tolerant multiplication of two 3000 square matrices, using block sizes 750 and 250. In this case, all shared objects are co-located on a single HP730 machine. As before, derived and

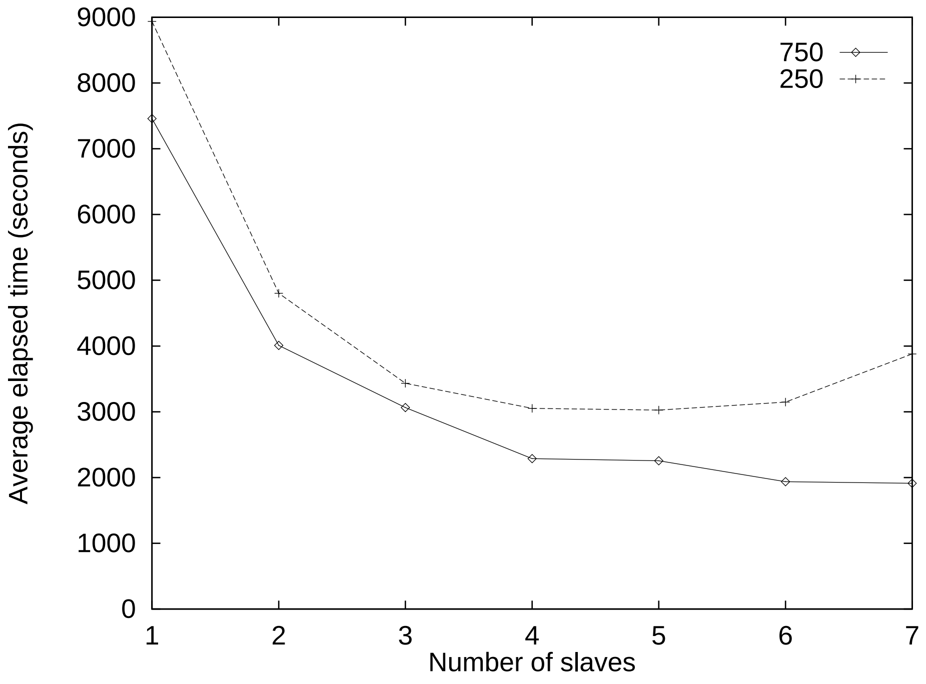

Figure 7. Parallel Multiplication of 3000 square Matrices with Block Size 750. The shared objects reside on a single HP730 machine.

measured values are compared in table 2. The measured single slave time is

Block Size (bytes)	Derived				Measured	
	all blocks cached		no blocks cached			
	T_1	$\bar{S}_{max}$	T_1	$\bar{S}_{max}$	T_1	$\bar{S}_{max}$
250	2.2	3.6	2.6	2.7	2.5	3.0
750	1.9	6.1	2.0	5.1	2.1	3.9

Table 2. Comparison Of Derived And Measured Results: Single slave time, $T1$ in hours and maximum speedup $\bar{S}_{max}$ for multiplication of 3000 square matrices

seen to be about 2.1 hours and the maximum speedup estimated at about 3.9

The estimated in-core computation time is about 1.6 hours on a HP710 machine, suggesting an absolute speedup of about 3.0. On a HP730 machine the estimated in-core computation time is about 1.1 hours, suggesting a speedup of 2.0. However, neither machine has sufficient memory to perform the computation in-core. Estimates of the expected duration of an out-of-core implementation of the computation on a HP730, with block size 750, may be computed for either maximum or minimum cache benefit being approximately 1.25 and 1.33 hours respectively. These are compared with a measured value of about 1.4 hours. The most favourable speedup figure that may be claimed then is 2.6 over this latter value, though it is acknowledged that considerable resources have been exploited in the effort. Work proceeds to attempt to explain observed discrepancy between measured and derived results, hopefully identifying potential savings in the implementation.

3 Fault Tolerant Matrix Multiplication

3.1 General Approach

The model shown in figure 1 is modified such that the master M places a description of each of the tasks making up the whole computation into a shared repository, called a bag. At the start of the computation, the master notifies slaves of the location of the bag object as well as the computation data objects. The slaves repeatedly withdraw a single task description at a time from the bag and complete that task before returning to the bag for another task. The work is thereby balanced between potentially heterogeneous machines. The simplest configuration locates the bag of tasks on the same machine as the three matrix objects.

A fairly common occurrence in a workstation cluster environment is the failure, i.e. crash, or reboot of a single machine, so it seems desirable to be able to tolerate such failures in a way other than restarting the whole computation. It is assumed that any data in volatile storage is lost, but that held on disk storage remains unaffected. There are then three areas of concern:

- A machine hosting a slave may fail between the point at which the slave extracts a task from the bag and the point at which it completes writing the outputs. In the event of such a failure, the task being performed is not completed, though partial results may have been written.
- The machine hosting the shared objects may fail. Such a failure prohibits any further progress by any of the slaves and any results not saved to secondary storage are lost.
- The machine hosting the master, which initiates and subsequently waits for completion of the computation may fail. If the required number of slaves have been initiated before the failure, then the result is simply that the user does not know of the outcome of the computation though the computation may progress towards completion. However, if this is not the case, then there may be no further progress although system resources may remain in use.

There is thus a range of levels of fault tolerance which may be implemented in a bag of tasks application. As mentioned earlier, fault tolerance in this work is implemented through the use of atomic actions operating on persistent objects. In object member functions, the programmer places lock requests, i.e. read or write, to suit the semantics of the operation, and typically surrounds the code within the function by an atomic action begin and end, i.e. commit, or abort. The infrastructure manages the required access from and/or to disk based state. Such objects may be distributed on separate machines. By enclosing calls to multiple such distributed objects with another atomic action, it is possible to ensure distributed consistency. An atomic action which is begun within such a called function is said to *nest* within the surrounding action in the callee function. If such a nested action is aborted, then locks obtained within the action are released immediately. If the nested action is committed, such locks are passed up to the *parent action* and so on and only actually released finally when the top level, i.e. outermost, action commits. It is only at this point that the effects of nested actions become visible. By contrast, if the top level action is aborted, then the effects of all nested actions which have been committed are recovered, i.e. undone. In the event of a callee machine failing part way through a nested action at a remote site, that action is assumed to be aborted.

1. To tolerate failure of a slave, it is desirable for the task currently being performed by that slave at the time of the failure to reappear in the bag, while any work already done towards completion of that task by the failed slave is recovered. The approach employed is therefore for the slave to begin an atomic action before accessing an item of work from the bag, and commit the action after writing the output generated by that work.
2. If the shared objects are replicated on multiple machines, then the failure of such a machine may be tolerated.
3. To tolerate failure of the master process, the favoured approach here is to define a computation object which contains a description of the computation and maintains its completion status. This object may be queried at any time to determine the completion status of the computation, and may be replicated for availability. Use of such a computation object also allows processes on arbitrary machines to join in an ongoing computation.

A possible distribution of objects in a fully fault-tolerant parallel implementation of matrix multiplication which employs a bag of tasks for load balancing is shown in figure 8.

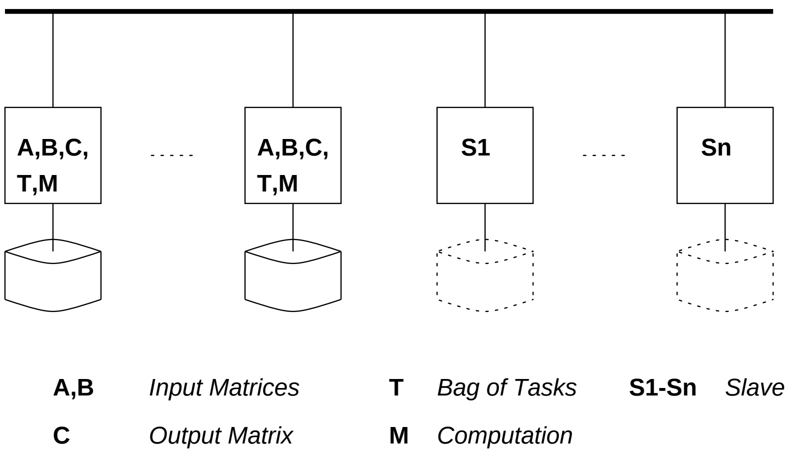

Figure 8. Possible Distribution of Objects in Fault-Tolerant Bag of Tasks Parallel Matrix Multiplication. Shared objects are shown replicated for availability. Disks which are shown dotted are not used explicitly by the application

In this experiment a block structured algorithm is employed, so a task entails computation of a block of the output matrix, which is the block dot product of a block row of the first input matrix and block column of the second input matrix.

3.2 Implementation

The implementation of parallel matrix multiplication described in section 2.1 has been enhanced by the addition of a prototype recoverable queue, implemented as a composite persistent object containing separately lockable links and elements. A task is executed by a slave within the scope of an atomic action and involves calling the *dequeue()* operation of the queue to obtain the next available task description, performing the corresponding calculation, then storing the results in the output matrix.

3.2.1 Bag of Tasks

The requirements of a recoverable bag are similar to the specification of a semiqueue in [23]. A convenient structure with which to implement the bag in Arjuna is a recoverable queue, similar to that described in [5], which may be regarded as a possible implementation of a semiqueue. Unlike a traditional queue which is strictly FIFO, a recoverable queue relaxes the ordering property to suit its use in a transactional environment. If an element is dequeued within a transaction, then that element is write-locked immediately, but only actually dequeued at the time the transaction commits. Similar use of recoverable queues with multiple servers in asynchronous transaction processing is described in [13], so only a brief description is given here through an example.

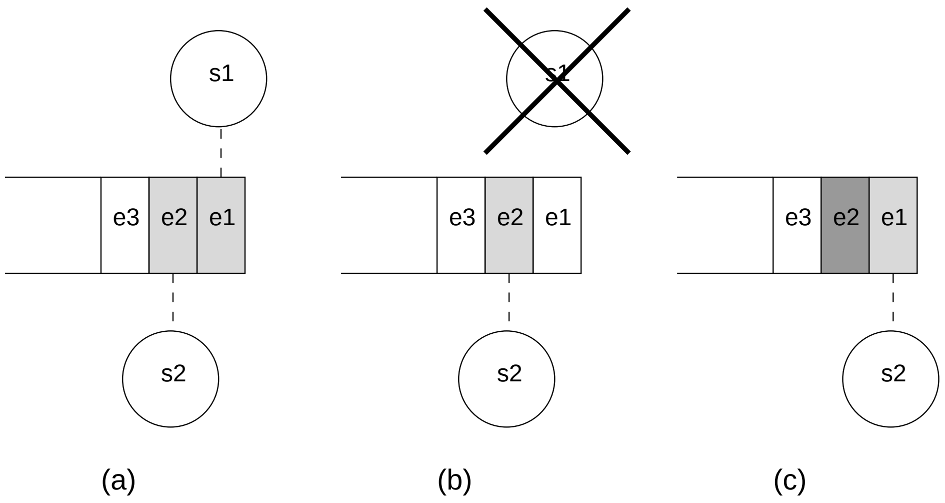

Figure 9. Operation of a Recoverable Queue

In figure 9(a), two processes, s1 and s2, are shown having dequeued elements from this queue, e1 and e2 respectively. In the absence of failures, say s1 completes processing e1 before s2 completes processing e2, then s1 processes e3. However, figure 9(b) shows s1 having failed and its partially completed work aborted, such that e1 is unlocked and so available for subsequent dequeue. Figure 9(c) shows s2 having completed processing of e2, now processing e1.

3.2.2 Slave

After binding to a set of shared objects, the slave executes a loop which repeatedly dequeues a task from the queue, fetches the appropriate parts of the input matrix and computes and writes out the corresponding part of the result. Each such iteration is contained within an atomic action. This atomic action guarantees that the slave has free access to the corresponding block of the output matrix until commit or abort. Any failure of the slave leads to abort of the action, such that any uncommitted output, together with the corresponding *dequeue()* operation is recovered, leaving the unfinished task in the queue to be performed by another slave.

In database terms, the slave is coordinator for the atomic action, so that a failure of slave is failure of coordinator. In a database application, the coordinator is required to ensure eventual outcome is consistent with notification to an operator and achieves this through a persistent record called an intentions list written during first phase of two phase commit protocol. This record is used to ensure the action is completed consistently with operator wishes, or a failure notice given, in the event of crashes at either coordinator or participant sites. However, complete knowledge of the action resides only at the coordinator site, so the action blocks during failure of its coordinator. Such behaviour is not desirable here, where the intention is for an alternative slave to redo such a failed task. The application described here is similar to the asynchronous transaction processing referred to earlier in that the coordinator is driven entirely by the contents of the queue entry. In the former case, a response to an operator may be placed in a separate reponse queue, but in an application of the type described in this work this does not appear generally useful. The correctness requirements are similar however, in that the work description must remain in the queue until corresponding work is completed. Since each task entails computing from read only parameters a unique block of the ouput matrix and then writing it, idempotency is guarenteed. Therefore correctness of queue operation in this application may be ensured by careful ordering of updates during commit processing.

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

3.3 Measurements

A rough measurement of the cost of employing this implementation of a recoverable queue may be obtained by performing fault tolerant and non fault tolerant sequential computations and recording the difference in elapsed times. This is done for multiplication of 1000 square matrices on HP710 and the results shown in table 3. In figure 10 it is seen that the cost of fault tolerance remains

Block Size (bytes)	Items of Work	Fault Tolerance Overhead		
		During Queue Creation (seconds)	During Computation (seconds)	Total as % of Total Elapsed Time
125	64	55	37	18
200	25	23	15	9.6
250	16	15	17	8.5
500	4	5	18	6.4

Table 3. Cost of Employing Queue in Sequential Multiplication of 1000 Square Matrices

fairly constant for varying number of slaves.

The percentage overhead is relatively high in the above examples because the computation is of small scale. The cost of using the queue is dependent on the number of tasks, rather than the matrix size.

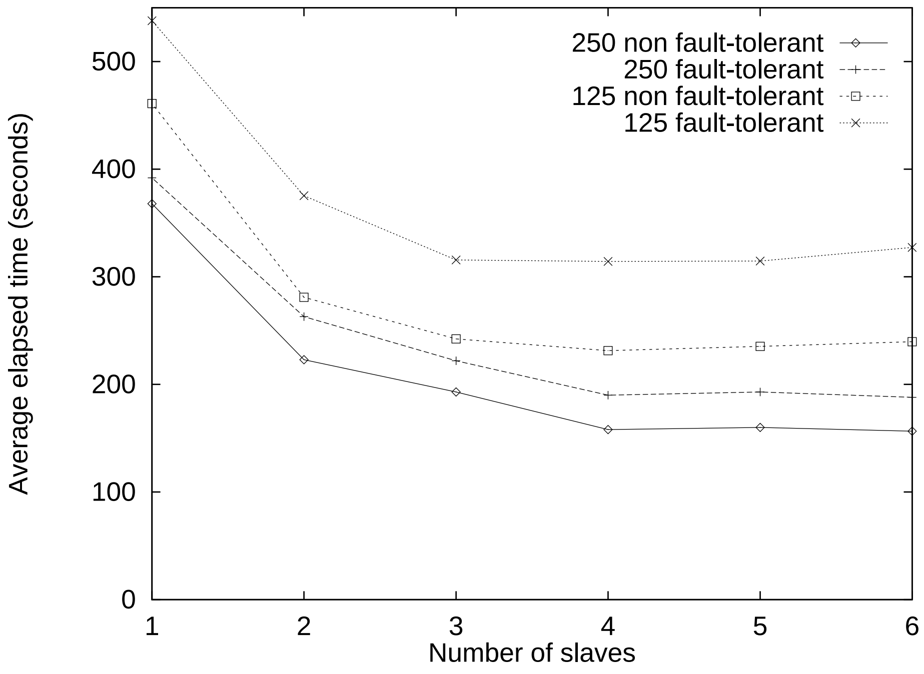

Figure 10. Fault Tolerance Cost in Parallel Multiplication of 1000 square Matrices with Block Sizes 125 and 250.

- There is a cost associated with creating the queue and initializing it with one entry to describe each piece of work. This initialization entails enqueueing one entry per block of the output matrix, within a surrounding action, and committing that action.
- Secondly, there is the cost incurred by the slave of binding to the queue object and subsequently dequeuing an entry describing each piece of work.

4 Related Work

There is growing interest in the use of workstations to perform parallel computations, as demonstrated by the increasing number of systems which support such programming, such as: Munin [8], Linda [7], Mentat [14], PVM [22].

Mechanisms to support fault tolerance may be transparent to the application programmer or explicit. Checkpointing schemes such as the globally consistent mechanism of Orca, [15], and the optimistic scheme of Manetho, [11] are examples of the former category. While these systems require operating system level support, a scheme proposed for the widely used distributed parallel programming environment PVM, [16] is portable. However, a global checkpointing scheme is unlikely to take advantage of the structure of a bag of tasks application, where there is an optimal point for checkpointing state, namely at task completion. An alternative approach relying on primary backup process replication is implemented in Paralex, [2]. However, Paralex is suited to data flow type applications.

Pact, [17], defines a set of extensions to a sequential programming language to enable implementation of fault-tolerant task based parallel programs. The main program defines an execution graph where the nodes are atomic action functions where entry constitutes beginning the action, and exit constitutes ending the action. In widely used terminology of atomic actions, all actions in Pact are top level actions. The edges in the graph are user defined event dependencies, such that for instance one function is to be started on occurrence of the termination event of another function. Actions share data defined in the main program, subdivided into *dataunits*, through an implementation of DSM, distributed shared memory. Locks, according to single writer multiple reader policy, are acquired on *dataunits* as they are used within an action. At the same time a required *dataunit* is migrated if necessary. An action's termination event is not triggered until modified *dataunits* have been written to a log. The run-time system, described in [18], employs an Execution Manager to coordinate the overall program execution among a number of Supervisors, each responsible for management of a portion of the *dataunits* and a set of multithreaded server processes. An action call may include a description of which *dataunits* are to be used for optimal Supervisor selection, and the supervisor allocates the call to a server process based on its accumulated resource usage information. Periodically, the Execution Manager forces global checkpoints which optimize restart recovery and allow truncation of log records. The log is distributed and entries are timestamped to allow global ordering in the necessity for restart. Recovery from failure of a single server node is possible through the logged information, though other failures appear to necessitate restart recovery, i.e. from the last global checkpoint. While it is possible to express matrix multiplication in a bag of tasks style in Pact, Pact does not currently provide support for programming parallel computations which are distributed over a network.

The Distributed Resource Manager which has been implemented using ISIS, [9], does provide support for execution of large scale computations in parallel over a network of heterogeneous machines. The resource manager, i.e. master, is replicated for availability and the causal group communications primitives of ISIS are employed to ensure consistent message ordering for exchanges between master and slaves, and master and user interface. The system is presented as a fault tolerant distributed batch scheduling system. While it may be possible to express a large scale matrix multiplication as repeated execution of a slave program with different parameters, each task is run as a separate executable, entailing a noticeable overhead in the associated *fork()*. Furthermore, the system provides no explicit support for access to the large remote objects.

The bag of tasks structure is well suited to Linda, and an implementation of Linda for a network of workstations is available. FT-Linda, [3] implements additional primitives to support fault tolerance. The extensions include atomic combinations of operations, the ability to define multiple tuple spaces, including stable tuple spaces (through replication) and atomic transfer of tuples between tuplespaces. In FT-Linda, each such replica is updated independently, but a consistent multicast mechanism is used to ensure that identical update requests arrive at each replica in an identical order. In the bag of tasks structure, shared data is located in replicated tuple space. A slave atomically removes a task and replaces it by an *in progress* tuple, such that a monitor process can restore the appropriate work tuple in the event of a slave failing while processing a tuple. As the slave processes a tuple, it writes results into a scratch tuple space and, on completion of the work, atomically replaces the *in progress* tuple by the contents of the scratch tuple space. MOM, [6] partitions tuples into separate lists, including a *busy* list for work tuples which are being processed and a *children* list for tuples generated by a worker which has yet to call *Done*. The busy list is then similar to the scratch tuple space of FT-Linda, and in both cases, there is an analogy to the use of nested atomic actions in the work reported in this paper. Facilities for accessing very large objects are not supported however. However, Plinda, [1], does propose access to persistent tuple spaces and extensions to Linda aimed at supporting efficient access to large data items. Performance measurements however are not published, so that a comparative evaluation is not possible.

Experiments have been conducted in which the fault tolerance facilities provided by Argus were used to program a number of applications, [4] Argus is a language and runtime system for programming reliable distributed applications and implements recovery units called *guardians*. A guardian runs on a single node and contains data objects and *handlers* to operate on them, internally coordinating as necessary through *mutex*. Some of the data objects may be *stable* and are backed up to stable, e.g. secondary, storage, so as to be recovered after failure, while volatile objects are re-initialized. Similarly to Arjuna, Argus implements a nested atomic action model to allow users to ensure distributed consistency. For example, a simple implementation of parallel matrix multiplication is described where each slave is a guardian apportioned a part of the output matrix to compute by a master guardian. Each slave checkpoints each computed row to stable storage and maintaining an integer value determining the next row to be computed. This structure is chosen to optimize performance in a collection of homogeneous machines. However, the strategy prevents other processes from taking over the work of a failed process and can thus degrade performance in the event of a failure occurring. Another application considered is travelling salesman problem which is less regular than matrix multiplication. Here again the approach suggested is to partition the work statically between slaves, with the master pre-computing the first couple of levels of the search tree. Use of a resilient data type, as in [23], is suggested as a possible way of making the master fault-tolerant, though issues of dynamic load balancing are not addressed. The issues related to managing very large secondary storage based data sets have not been addressed explicitly and no performance results have been published, so that, as in the case of Plinda, a comparative evaluation is not possible.

5 Concluding Remarks

Interest in exploiting the parallel processing power of a network of workstations to perform large scale computations is growing. Typically workstations are allocated as single user machines, with storage resources sufficient only for a single user. Obvious applications to consider first for execution in such a network environment are ones which make relatively modest demands for storage, or which can be partitioned into sufficiently small pieces. By contrast, the experiment described here is an attempt to perform a computation which manipulates large amounts of data This work has considered the computation of large scale matrix multiplication in a general purpose computing environment consisting of a network of commonly used workstations.

A prototype of the application has been implemented using the services of a class library for building fault tolerant distributed applications. Optionally, a recoverable queue is employed to implement a fault tolerant bag of tasks structure. The examples considered have been limited by available disk space, but square matrices of width 3000 elements have been multiplied. In this case, the size of each matrix is 72 Mbytes which exceeds available memory even on the more powerful HP730 workstation, of which there is a much smaller number at this establishment. Even for this example then, the computation requires out-of-core techniques on these workstations. Experiment with the prototype has yielded real though modest speedup.

For scalability, the computation is preferably block structured. The block size identifies a compromise with regard to performance of the computation. For a given matrix size, say n, the total amount of data accessed depends on the block size. For a matrix partitioned into p^2 blocks of size n_0^2, the total data transferred is the product of the number of block transfers and the size of a block, i.e. $p^2(2p+1)n_0^2$ or $n^2(2p+1)$ elements. Assuming the access cost itself increases linearly with the block size, then the overal time for a single remote slave to perform the computation decreases in inverse proportion to the block size. Similarly, the maximum speedup increases in proportion to the block size. The block size employed has been limited by available memory, but for the range of block sizes considered, the experimental results appear to confirm this result.

While the cost of reading a block from filesystem cache is certainly much lower then the cost of reading from disk, the cache can provide little benefit for large block size. Blocks of the ouput matrix are computed by block row, so that a number of successive tasks, for the same block row of the output matrix, read the same block row of the first input matrix. For the 3000 square matrix, a block row is 18 Mbytes for 750 square blocks and 6 Mbytes for 250 square blocks. It seems more likely that the latter will be held in cache, so that on this basis it would be preferable to employ a smaller block size, but large enough to make maximum use of cache space. However, as the overall matrix size is increased, the size of block of which a row may be held in cache will decrease. For the example referred to above, the analysis suggests that the faster time for this computation should be obtained by using 750 square blocks rather than 250 square, even if all blocks are cached in the latter case, as shown in table 2.

The cost of provision for fault tolerance in failure free execution is found to reduce as the block size is increased. Clearly, there is one entry added to the queue for each block of the output matrix at startup and one queue access for each block of the output matrix computed by a slave. This is confirmed through experiment. Clearly also though, the cost incurred though recovery following a failure increases as the block size increases. The normal execution time is increased by the cost of one block execution in the event of failure and immediate resumption. Since there are p^2 blocks, this overhead is $100/p^2$%. If a slave in a parallel execution fails and does not resume, then the increase in overall execution time depends on the exact point of failure, but the same value as above may be regarded as a measure of the cost of recovery, separate to the issue of changing the number of slaves.

Acknowledgements

The work reported here has been supported in part by grants from the UK Ministry of Defence, Engineering and Physical Sciences Research Council (Grant Number GR/H81078) and ESPRIT project BROADCAST (Basic Research Project Number 6360). The support of all the Arjuna team is gratefully acknowledged, and in particular the assistance of M. Little, G. Parrington, and S. Wheater with implementation issues particularly relevant to this work.

References

[1] Brian G. Anderson and Dennis Shasha. Persistent linda: Linda + transactions + query processing. In *Workshop On Research Directions In High-Level Parallel Programming Languages*, pages 129–141, Mont Saint-Michel, France, June 1991.

[2] Ozalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. Paralex: An environment for parallel programming in distributed systems. Technical Report UB-LCS-91-01, Univerity of Bologna, Laboratory for Computer Science, April 1991.

[3] David Edward Bakken. *Supporting Fault-Tolerant Parallel Programming in Linda*. PhD thesis, Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, August 1994. Available as technical report TR94-23.

[4] Henri E. Bal. Fault tolerant parallel programming in argus. *Concurrency: Practice and Experience*, 4(1):37–55, February 1992.

[5] Philip A. Bernstein, Meichun Hsu, and Bruce Mann. Implementing recoverable requests using queues. *ACM SIGMOD*, pages 112–122, 1990.

[6] Scott R. Cannon and David Dunn. Adding fault-tolerant transaction processing to linda. *Software-Practice And Experience*, 24(5):449–466, May 1994.

[7] Nicholas Carriero and David Gelernter. *How To Write Parallel Programs: A First Course*. MIT Press, 1991. ISBN 0-262-03171-X.

[8] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Operating Systems Review, pages 152–164, Pacific Grove CA (USA), October 1991.

[9] Timothy Clark and Kenneth P. Birman. Using the isis resource manager for distributed, fault-tolerant computing. Technical Report 92-1289, Cornell University Computer Science Department, June 1992.

[10] Craig C. Douglas, Timothy G. Mattson, and Martin H. Schultz. Parallel programming systems for workstation clusters. Technical Report YALEU/DCS/TR-975, Yale University, Department Of Computer Science, August 1993.

[11] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers*, May 1992.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. John Hopkins University Press, second edition, 1989. ISBN 0-8018-3772-3.

[13] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.

[14] Andrew S Grimshaw. Easy to use parallel processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[15] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerance in parallel orca programs. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, Newport Beach, CA, March 1992.

[16] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, February 1993.

[17] Joachim Maier. Pact - a fault tolerant parallel programming environment. In *1st International Workshop on Software for Multiprocessors and Supercomputers: Theory, Practice, Experience*, St Petersburg, February 1993.

[18] Joachim Maier. Fault-tolerant parallel programming with atomic actions. In *4th Workshop on Fault-Tolerant Parallel and Distributed Systems*, College Station, Texas, June 1994. IEEE, IEEE Computer Society Press.

[19] M.W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, July 1991.

[20] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of arjuna. Technical report, University of Newcastle upon Tyne, Computing Laboratory, 1995.

[21] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. Objects and actions in reliable distributed systems. *IEEE Software Engineering Journal*, 2(5):160–168, September 1987.

[22] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), Dec 1990.

[23] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

Fault-Tolerant Execution of Computationally and Storage Intensive Parallel Programs Over A Network Of Workstations: A Case Study

J.A.Smith
S.K.Shrivastava

Broadcast Technical Report 103

The paper considers the issues affecting the speedup attainable for computations that are demanding in both storage and computing requirements, e.g. several hundred megabytes of data and hours of computation time. Specifically, the paper investigates the performance of matrix multiplication. A fault-tolerant system for the bag of tasks *computation structure using* atomic actions *(equivalent here to* atomic transactions*) to operate on persistent objects. Experimental results are described. Analysis, backed up by the experimental results, shows how best to structure such computations for obtaining reasonable speedups.*

ISSN 1350-2042
Esprit Basic Research Project 6360
Broadcast Technical Report Series