A System For Fault-Tolerant Execution of Data and Compute Intensive Programs Over a Network Of Workstations*

J.A.Smith and S.K.Shrivastava

Department of Computing Science,
The University of Newcastle upon Tyne,
Newcastle upon Tyne,
NE1 7RU UK
{jim.smith,santosh.shrivastava}@newcastle.ac.uk

18 february 1996

Abstract

A well known structuring technique for a wide class of parallel applications is the *bag of tasks*, which allows a computation to be partitioned dynamically between a collection of concurrent processes. This paper describes a fault-tolerant implementation of this structure using *atomic actions* (*atomic transactions*) to operate on persistent objects, which are accessed in a distributed setting via a *Remote Procedure Call* (RPC). The system developed is suited to parallel execution of data and compute intensive programs that require persistent storage and fault tolerance facilities. The suitability of the system is examined in the context of the measured performance of three specific applications; ray tracing, matrix multiplication and Cholesky factorization. The system developed runs on stock hardware and software platforms, specifically UNIX, C++.

1 Introduction

A Network Of Workstations (NOW) is commonly employed for general purpose computing to ensure that each user has a good interactive response from their dedicated machine. In such an environment however it has been reported that there are likely to be significant periods of inactivity, e.g., [15]. 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

[0] A shortened version of this paper is to be presented at Euro-Par'96.

of encouraging results obtained using large, and perhaps varying, numbers of workstations for problems executed in this way.

As the problem size increases so too does the duration in any given configuration, and potentially also so too does the number of nodes which may be employed. As the scale of such a distributed computation is increased in this way, the possibility of a failure occurring which might affect the execution of the computation must increase. In the context of a NOW, both power failure to and reboot of an individual workstation are perceived as failures. If it is not possible to tolerate such an event, it is necessary to restart the entire computation.

Many computations manipulate very large amounts of data. Matrix calculations represent one example class. In a Massively Parallel Processor (MPP) such a vast data set is typically partitioned statically between the very many distributed processing elements and moved amongst them as necessary to perform the computation. Such an approach is exemplified in Cannon's algorithm for matrix multiplication [13]. One suggestion is that a NOW be modelled on such an architecture [2]. However, it may be that problem size can exceed even the aggregate memory of all available machines. In such a situation, the problem cannot be statically partitioned between processors. The approach described here provides a solution by implementing a shared store on secondary storage which is shared between a collection of workers.

This shared store is organized as a repository of objects and fault tolerant access to it is supported through atomic actions operating on the contained objects. It is suggested that these mechanisms provide a clear model to the user. The facilities described are supported through the services of an established distributed system which runs on many versions of UNIX and C++, without any alteration to either. An implementation of this approach is investigated through a number of applications of scale appropriate to parallelization and fault-tolerance in a NOW. Performance is shown to be fundamentally limited only in hardware bandwidths.

The paper is organized as follows. An overview of the general computation structure is given in Sect. 2, followed by a brief look at related work in Sect. 3. Then Sect. 4 gives an overview of the fault-tolerance strategy employed before Sect. 5 describes implementation details and the applications themselves. Section 6 describes the measured performance and Sect. 7 presents some conclusions.

2 Bag of Tasks

The essential idea of the structure known variously as "bag of tasks", "task pool", "master worker", "master slave", "process farm" is that the overall computation be divided up into a collection of tasks which are then scheduled dynamically between a collection of concurrent processes. While the Master of the process farm view is a process which actively sends work to slaves and balances work between the slaves, an alternative view is of task definitions being stored in a passive data structure, a bag, and fetched from there by workers which run quite independently. The latter view, shown in Fig. 1, is adopted here as it fits with chosen implementation. In the example a process called Master, M,

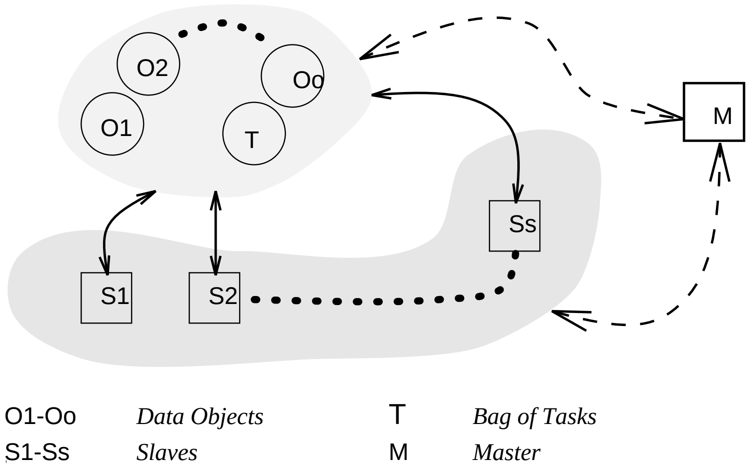

Figure 1: Bag of Tasks Computation Structure

is shown controlling overall startup and shutdown. In addition to the bag of tasks, T, the concurrent slaves, S1-Ss, share a number of data objects, O1-Oo. It is feasible for a computation to be begun with some initial complement of workers and to have workers join and leave the computation, even to the point of stalling the computation for a period.

3 Related Work

The attraction of exploiting a readily available NOW to perform parallel computations is widely acknowledged. It is also recognized that a NOW typically has disadvantages compared to a tightly coupled multiprocessor, including a lower performance interconnect and a greater need for fault-tolerance.

Experiments have been performed to statically partition data intensive computations over a NOW, e.g. [5]. However, the size of the computation is bounded by aggregate memory of the machines. Structuring similar to the bag of tasks is often employed in practice, e.g. for seismic migration in [1], but with limited provision for fault-tolerance and for problems which are less intensive in data.

Mechanisms to support fault-tolerance may be transparent to the application programmer, e.g. [11], [14]. However, a transparent scheme is unlikely to take advantage of points in an application where data to be saved is minimum, such as when data has just been written to disk for instance.

One non transparent scheme for the static partitioning approach [17] maintains a parity copy of distributed partitions of computation state. While performance for a Cholesky factorization of 5000 element square matrix , at 1700 seconds employing 17 Sparc-2 machines, is similar to that recorded here the computation is bounded by total memory and the approach here which employs fewer machines is resilient to a greater number of failures.

An early design study [4] considered the use of atomic actions as a mechanism to support fault-tolerant parallel programming over a NOW.

Fault tolerance for a bag of tasks type structure has been considered before, e.g. [3], [7] but without providing access to large scale data on secondary storage. Plinda [10] which supports access to persistent tuple spaces and a transaction mechanism does have some similarity to this work.

The experiments described here attempt to exploit parallelism in a NOW of modest scale to perform large scale computations in a fault-tolerant way without altering operating system or language.

4 Fault Tolerance Strategy

It is assumed that a workstation fails by crashing and that then any data in volatile storage is lost, but that held on disk remains unaffected. It is also assumed that the network does not partition.

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.
- A machine hosting shared objects may fail. In the most basic configuration all shared objects are co-located on a single machine. In this case a failure prohibits any further progress by any of the slaves and any results not saved to secondary storage are lost.
- The machine hosting the master, which initiates and subsequently waits for completion of the computation may fail. If the required number of slaves have been initiated before the failure, then the result is simply that the user does not know 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.

Atomic actions operating on persistent state provide a convenient framework for introducing fault-tolerance [9] through ensuring defined concurrent behaviour and fault-tolerance. Atomic actions have the well known properties of (1) serializability, (2) failure atomicity, and (3) permanence of effect.

A convenient model is for this state to be encapsulated in the instance variables of persistent objects and accessed through member functions. Within these functions the programmer places lock requests, e.g. read or write to suit the semantics of the operation, and typically surrounds the code within the function by an atomic action, starting with *begin* and ending with *commit* or *abort*. Operations thus enclosed which can include calls on other atomic objects are then perceived as a single atomic operation. The infrastructure manages the required access from and/or to disk based state. Such objects may be distributed on separate machines, e.g. for performance, and replicated to increase availability. The applications are implemented using the Arjuna tool kit [16], an object-oriented programming system that implements in C++ this object and action model.

The following enhancements add fault-tolerance to a bag of tasks application.

1. The slave begins an atomic action before fetching a task from the bag, and commits the action after writing the corresponding result. If the slave fails the action aborts, all work pertaining to the current task is recovered and the task itself becomes available again in the bag.

2. The shared objects are replicated on at least $k+1$ machines, so that the failure of up to k of these machines may be tolerated.

3. A computation object contains a description of the computation and data objects and the computation's completion status. This object may be queried at any time to determine the status of the computation and may be replicated for availability. It is a convenient interface for a process to be started on an arbitrary machine to join in an ongoing computation.

A possible distribution of objects in a fully fault-tolerant parallel implementation of a bag of tasks computation is shown in Fig. 2.

5 Implementation

Arjuna requires an underlying RPC to implement distribution and object server process management and accesses these services through certain interface classes. The RPC implementation employed here supports optional use of the TCP protocol with connection establishment on a per-call basis. Some optimization of this RPC mechanism has been performed to exploit homogeneity of machines. Furthermore, the RPC supports reuse of an existing server process. This facility is exploited in service of the main shared data objects in order to prevent excessive contention in the shared communications medium; the common server is single threaded and therefore serializes all slave requests.

In each application, the main operands are managed as collections of smaller objects. Each task entails computation of some part of the result, which may be one or more of such objects.

At the start of the computation, the shared objects are installed in the object repository. In the fault-tolerant version, a fault-tolerant bag of tasks is created and all task descriptions stored in it. Then the chosen number of slaves is created on separate workstations. In the non fault-tolerant implementation, each slave is informed of a unique allocation of tasks to perform. In these initial

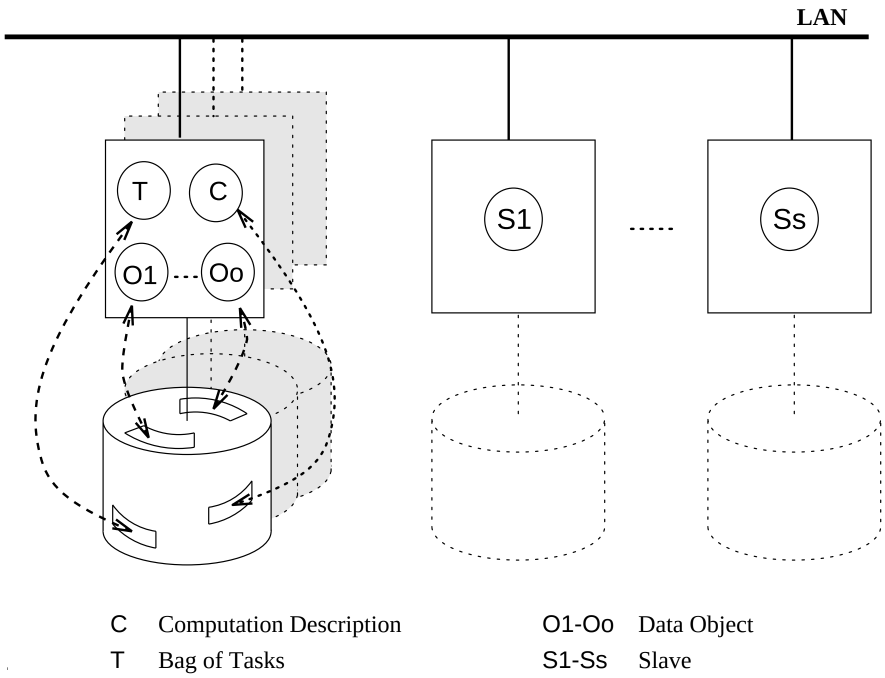

Figure 2: Possible Distribution of Objects in Fault-Tolerant Bag of Tasks Computation.

experiments, a master process is employed to perform these functions and then wait for the completion of the slaves before performing any final processing to the output, such as converting to a desired file format, and finally reporting on the elapsed time. The master takes no active part during the main part of the application, so a shell script replacement is quite feasible. Also at this time the shared objects are not replicated.

5.1 Applications

Three applications are implemented. The first is a port of a publicly available ray tracing package, *rayshade* [12]. Input data comprises only scene description and output is a two dimensional array of red-green-blue pixel values. A task is defined as computation of a number of rows of the output array. To display the output image, it is convenient to copy it to the file format used in the original package, Utah Raster RLE format. In this implementation, this operation is performed serially by the master process. A simple scene provided as an example in the package is traced for the purposes of the test. For comparison, the unaltered package is built and run as a sequential program on one of the workstations.

The remaining applications are dense matrix computations, matrix multiplication and Cholesky factorization. A preliminary description of the former was given in [19]. In linear algebra computations it is common to employ block structuring to benefit from increased locality [8]. In the implementation of both matrix computations here, matrices are composed of square blocks and a task defined as the computation of a single block of the result.

In the case of matrix multiplication, a task entails a block dot product of a row of blocks in the first and column of blocks in the second operand matrices. The implementation of Cholesky factorization employs the Pool-of-Tasks algorithm of [8], §6.3.8.

5.2 Bag of Tasks

The requirements of a recoverable bag are similar to the specification of a semiqueue in [20]. A convenient structure with which to implement the bag in Arjuna is a recoverable queue, similar to that described in [6], 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 [9], so only a brief description is given here through an example.

In figure 3(a), two processes, S1 and S2, are shown having dequeued elements `e1` and `e2` respectively from this queue. In the absence of failures, say S1 completes processing `e1` before S2 completes processing `e2`, then S1 processes `e3`. However, figure 3(b) shows S1 having failed and its partially completed

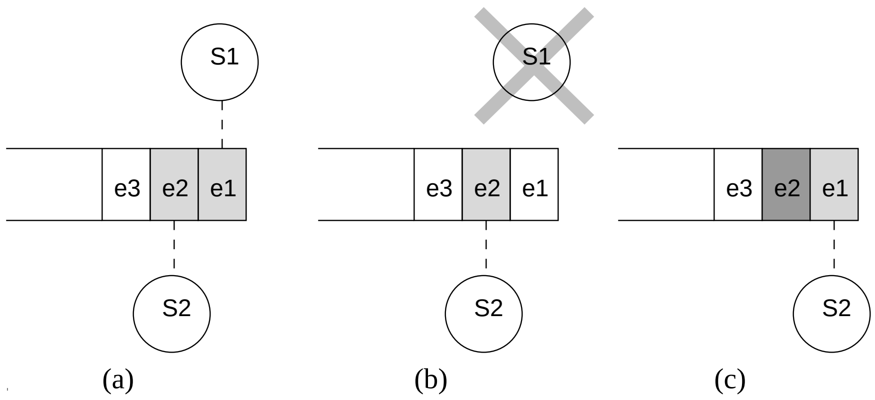

Figure 3: Operation of a Recoverable Queue

work aborted, such that `e1` is unlocked and so available for subsequent dequeue. Figure 3(c) shows S2 having completed processing of `e2`, now processing `e1`.

5.3 Slave

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

In database terms, the slave is coordinator for the atomic action, so that a failure of the slave is failure of the 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 the first phase of the two phase commit protocol. This record is used to ensure the action is either committed or aborted consistent with the user request 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 if the coordinator fails during phase 2 of an atomic action. Such behaviour is not desirable here, where the intention is for an alternative slave to redo such a failed task.

The correctness requirement is that each task description must remain in the queue until corresponding work is completed. Assuming each task entails computing from read only parameters, a unique output and then writing it, idempotency is guaranteed. Both ray tracing and matrix multiplication have this property. In this case, correctness of queue operation may be ensured by careful ordering of updates during commit processing. By contrast in the case of the asynchronous transaction processing referred to earlier, the use of a response queue to reliably inform a human operator of completion status of each queued transaction ensures that operations are not idempotent. [6] suggests use of sequence numbers to avoid duplication of queue entries resulting from this situation.

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.

5.4 Synchronization

The requirement for the data dependencies appearing in the Cholesky factorization algorithm is for a task to be blocked until some prior task has completed and produced output. In a correct execution, a static ordering of tasks may be used to ensure that a needed block will at least be in the process of being computed when a slave computing a dependent block attempts to access it, such that conventional read and write locks are adequate. However, where any slave may fail at any time it is quite possible for a slave to attempt to access a block which is not ready and yet not actually being generated by another slave

The mechanism for inter task coordination employed in the Cholesky factorization algorithm used here is ultimately implemented through a two dimensional array of flags which indicate whether corresponding blocks in the output matrix have been written or not. Concurrent access to the flags is controlled through locks obtained within the scope of atomic actions.

It is important that the controlling flag should only be updated to indicate availability of the corresponding block of the output matrix at or after the time that block is actually written. This behaviour is ensured through the use of atomic actions. However as discussed earlier 5.3, a window of vulnerability arises in phase 2 of commit of a slave action and it is necessary to ensure that during this commit operation, the flag is updated last.

An anomaly is illustrated by the Cholesky factorization application where the input matrix is factored in place, in that an individual task operation is not idempotent. This is because any output block depends not only on the corresponding block in the input matrix, but also on blocks in the output matrix which have already been computed. If factoring in place then, it is necessary to check the flag status explicitly before starting a task. This avoids repeating that task in the event that a slave that had been previously performing it actually failed during commit of the corresponding action.

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

6 Performance

Each experiment is conducted during off peak time in a cluster of HP9000/710 (HP710) machines each with 32 Mbyte memory and 64 Kbyte cache, connected by 10 Mbit/s Ethernet. A small number of HP9000/730 (HP730) machines with 64 Mbyte memory and 256 Kbyte cache have sizeable temporary disk space space available. For the matrix computations a cluster containing a HP730 is used, and the shared objects located on it, but HP710 machines are used otherwise. In this way computations with data requirements of about 200 Mbyte are performed.

6.1 Cost of Queue Access

An indication of the failure free overhead cost may be obtained by comparing fault tolerant and non fault tolerant sequential computations running within a single workstation. This is done for matrix multiplication by locating a single slave and the data objects on the same host, a HP730 machine. The measured results are shown in Tab. 1 for a range of task sizes.

Table 1: Cost of employing queue in sequential multiplication of 3000 square matrices. The times in columns 3 and 4 are averages rounded to integer values.

		Execution time		Fault tolerance Cost		
Items of work	Block width (elements)	Fault-tolerant (seconds)	Non fault-tolerant (seconds)	During queue creation (seconds)	After queue creation (seconds)	Total as % of total time
9	1000	2201	2152	6.5	41.5	2.2
16	750	2254	2224	10.3	20.3	1.4
25	600	2215	2171	15	29	2
36	500	2313	2252	22	38.3	2.7
144	250	3068	2917	93.8	58.1	5.2
225	200	3579	3352	154	73.5	6.8

The fault-tolerance costs represent the following operations:

- The cost of creating the queue and enqueueing one entry per block of the output matrix within a surrounding action, and committing that action.

- The cost incurred by the slave of binding to the queue object, essentially server creation, and then dequeuing an entry describing each piece of work.

The queue entries are simply small job descriptions and their size is independent of the data size so the cost of using the queue should be dependent on the number of tasks, rather than data size. Therefore percentage overheads should reduce for larger scale computations, but even for the size of computation performed, fault tolerance does not appear to be the significant cost.

The queue is implemented as a collection of separately lockable persistent objects, and some breakdown of the costs associated with the use of atomic actions on individual persistent objects is given in [16].

6.2 Parallel Execution

The parallel performance of the applications is shown in Fig. 4.

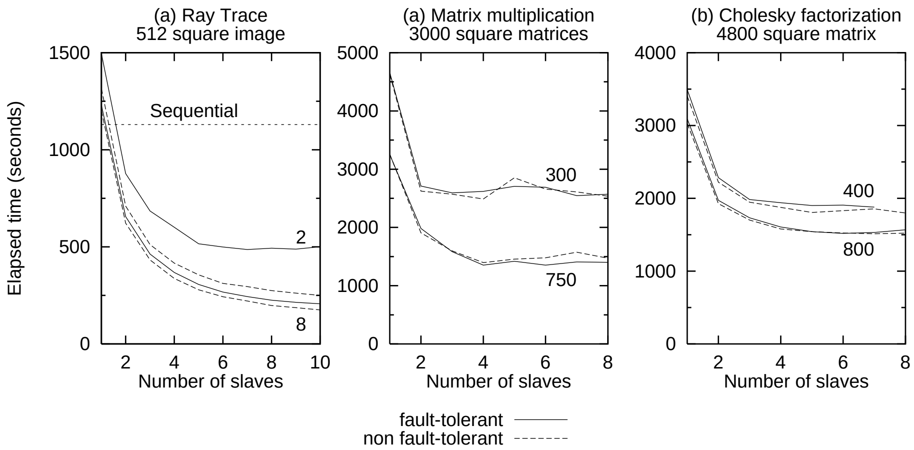

Figure 4: Performance of parallel applications, comparing fault-tolerant and non fault-tolerant versions for indicated task sizes.

In the event of slave failure and immediate resumption, or replacement by a spare, the failure free execution time is increased by a recovery time due to the loss of aborted work. This recovery time is the cost of between zero and one task executions, the *average recovery* being half of the maximum. A computation with non uniform tasks may still be characterized by a simple average recovery cost, though this may be misleading if the cost varies very considerably. If data are cached at a slave which fails, then the slave that takes over the aborted task incurs an extra cost in cache misses. If a slave fails and does not resume and there is no spare, then the increase in overall execution time depends on the exact point of failure, but may be regarded as comprising two components. First, there is the cost of redoing the failed task and secondly, the execution of the remaining tasks is slowed since there is then one less slave.

Table 2 summarizes the performance of the parallel implementations, showing for each application a measure of the performance achieved and estimate of the average recovery time. The table also indicates the total data: input (*input*), written (*put*) and read collectively by slaves during the computation (*get*).

Table 2: Fault-tolerant application parallel performance summary. The speedup shown for ray tracing is absolute, i.e. relative to that of the sequential implementation.

Application	Tasks	Task size (elements)	Data access (Mbyte) *input*	*get*	*put*	Minimum time (seconds)	Performance (speedup or rate)	Average recovery (seconds)
Ray Trace (512^2)	256	2×512	small		6.3	483	2.3	1.5
	64	8×512				204	5.5	9.6
Matrix Multiplication (3000^2)	100	300^2	144	1440	72	2545	21 Mflop/s	24
	16	750^2		576		1353	40 Mflop/s	102
Cholesky Factorization (4800^2)	78	400^2	99	1198	99	1879	20 Mflop/s	23
	21	800^2	108	645	108	1512	24 Mflop/s	74

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

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

The performance of the matrix computations is not exciting, though in the one case the peak performance of the memory based matrix multiplication on a single HP710, measured at 33 Mflop/s, is exceeded. Some intuition for the cost of the parallel computations may be gained by considering the cost of accessing the data. Each data access entails both a memory to memory copy between slave and server machine and a local disk, or filesystem cache access on the server machine. Some potential benefit exists both in pipelining data accesses and in caching blocks at slave machines but neither is attempted here. For block sizes above 250, the low level transfer rates for local memory to remote memory, local disk read and local disk write (new data) are found to be roughly constant at about $1, 1.6$ and 0.2 Mbyte/s. Assuming no benefit is gained from caching blocks between tasks, an estimate for the total time involved in transfers for the matrix multiplication application with larger block size is 1368 seconds. This would then be a lower bound on the parallel computation time and since the implementation described almost achieves this minimum time it seems possible that bandwidth limitation is being observed. Fuller analysis [18] finds that the benefit gained in this particular situation from involuntary filesystem caching is likely to be small, strengthening the case for bandwidth limitation.

7 Summary

The work described here considers the implementation of certain large scale computations each structured as a bag of tasks over a NOW employing Persistent objects and atomic actions to support fault-tolerance. The first application is a public domain ray tracing package with moderate demands for space. Experiment suggests that respectable performance can be achieved if a suitably large granularity is chosen. The other two applications are both dense matrix computations where the space requirement can exceed available memory. In such a case a model which employs a relatively small number of machines sharing large secondary storage space has some attraction. For this type of execution, a realistic all-be-it prototype implementation has shown that the cost of introducing fault-tolerance is small and performance gain through parallelism is limited essentially by hardware bandwidths.

The system described here provides a practical solution to the question as to how to exploit commonly available clusters of workstations for running compute and data intensive programs by providing much needed support for fault-tolerance and moderate speedup. Since the toolkit developed here does not require any special hardware or software facilities other than those already available, it can readily be adapted to exploit new generations of hardware. [18] describes detailed performance analysis of applications reported here and enables prediction of the expected performance under higher network bandwidth. For example, if the communications media is replaced by fast ethernet, at 100 Mbits/s, but the configuration remains otherwise unchanged a performance of 80 Mflop/s is anticipated for matrix multiplication using 4 slaves.

The overall conclusion then is that objects and actions as employed in the computations described seem to be a convenient way to express fault tolerance in parallel applications, and for appropriate scale of computation impose small cost.

Acknowledgements

The work reported here has been supported in part by research and studentship 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 the Arjuna team is acknowledged, and in particular the assistance of M. Little, G. Parrington, and S. Wheater with implementation issues relevant to this work.

References

[1] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 2nd edition, 1994. ISBN 0-8053-0443-6.

[2] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[3] David Edward Bakken. *Supporting Fault-Tolerant Parallel Programming in Linda*. PhD thesis, Department of Computer Science, The University of Arizona, August 1994.

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

[5] A. Benzoni and M. L. Sales. Concurrent matrix factorizations on workstation networks. In A. E. Fincham and B. Ford, editors, *Parallel Computation*, pages 273–284. Clarendon Press, 1991.

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

[7] 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.

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

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

[10] Karpjoo Jeong. *Fault-Tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions*. PhD thesis, New York University, Department of Computer Science, January 1996.

[11] 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.

[12] Craig Kolb. *rayshade*. ftp://ftp.cs.yale.edu, May 1990. version 3.0.

[13] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin Cummings, 1994. ISBN 0-8053-3170-0.

[14] 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.

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

[16] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):225–308, summer 1995.

[17] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. Technical Report CS-94-268, University of Tennessee, December 1994.

[18] J. Smith. *Fault Tolerant Parallel Applications Using a Network Of Workstations*. PhD thesis, University of Newcastle upon Tyne, Department of Computing Science, 1996. In Preparation.

[19] J. Smith and Santosh Shrivastava. Fault-tolerant execution of computationally and storage intensive programs over a network of workstations: A case study. In ESPRIT Basic Research Project 6360 Third Year Report, July 1995.

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