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

Abstract. The bag of tasks structure permits dynamic partitioning for a wide class of parallel applications. 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 is suited to parallel execution of data and compute intensive programs that require persistent storage and fault tolerance, and runs on stock hardware and software platforms, UNIX, C++. Its suitability is examined in the context of the measured performance of three applications; ray tracing, matrix multiplication and Cholesky factorization.

1 Introduction

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 [14]. One suggestion is that a Network Of Workstations (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.

As the problem size increases so too does the computation time in any given configuration, and in a NOW potentially so too does the number of nodes which may be employed. As the scale of a distributed computation is increased in this way, the possibility of a failure occurring which might affect the execution of the computation must increase. If it is not possible to tolerate such an event, it is necessary to restart the entire computation.

The approach described here provides a solution for these problems by implementing a store on secondary storage which is shared between a collection of concurrent processes. A computation is organized as a bag of tasks type structure [7] where the overall computation is divided up into a number of tasks which

are then scheduled dynamically between a potentially varying collection of concurrent processes. Computation data, including the bag of tasks is located in the shared store, which is organized as a repository of objects and fault tolerant access to it supported through atomic actions operating on the contained objects. It is suggested that these mechanisms provide a clear model to the user.

In this experiment, these facilities are supported through an established distributed system which runs on many versions of UNIX and C++, without alteration to either. The approach is investigated through implementation 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 continues with notes on related work in Sect. 2, a description of the applications and fault-tolerance mechanisms in Sect. 3, measured performance in Sect. 4 and summary in Sect. 5.

2 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. [12], [15]. 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], [8] but without providing access to large scale data on secondary storage. Plinda [11] 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.

3 Implementation

3.1 Fault Tolerance

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.

Atomic actions operating on persistent state provide a convenient framework for introducing fault-tolerance [10] 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.

Arjuna requires an underlying RPC to implement distribution and object server process management; accessing 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. The RPC also 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 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.

The fault-tolerant bag of tasks is implemented as a recoverable queue [6] which relaxes the usual FIFO ordering to suit its use in a transactional environment. If an element is dequeued within a transaction, then it is write-locked immediately, but only actually dequeued at the time the transaction commits. Similar use of recoverable queues in asynchronous transaction processing is described in [10]. The *dequeue* operation returns a status which allows the caller to distinguish between the situation where the queue is empty and that where entries remain but are all locked by other users.

3.2 Applications

Three applications are implemented. The first is a port of a publicly available ray tracing package, *rayshade* [13]. 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 [9]. 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 [9], §6.3.8. The required inter task coordination 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 operations

on the flags are controlled through locks obtained within the scope of atomic actions and are therefore recoverable. A fuller description appears in [18].

4 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.

4.1 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.

Items of work	Block width (elements)	Execution time		Fault tolerance Cost		
		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].

4.2 Parallel Execution

The parallel performance of the applications is shown in Fig. 1.

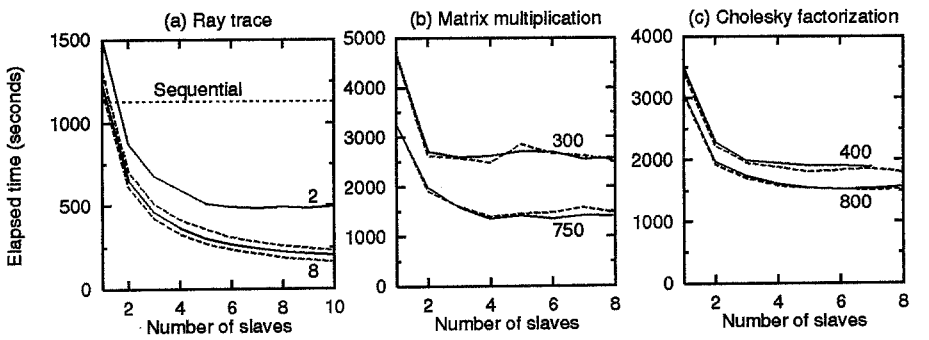

Fig. 1. Performance of parallel applications, comparing fault-tolerant (solid line) and non fault-tolerant (dashed line) versions for indicated task sizes.

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.

5 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 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. Nicholas Carriero and David Gelernter. *How To Write Parallel Programs: A First Course.* MIT Press, 1991. ISBN 0-262-03171-X.
8. 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.
9. Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* John Hopkins University Press, second edition, 1989. ISBN 0-8018-3772-3.
10. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kauffman, 1993.
11. Karpjoo Jeong. *Fault-Tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions.* PhD thesis, New York University, Department of Computer Science, January 1996.
12. 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.
13. Craig Kolb. *rayshade.* ftp://ftp.cs.yale.edu, May 1990. version 3.0.
14. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing.* Benjamin Cummings, 1994. ISBN 0-8053-3170-0.
15. 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.
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.