On Synchronisation in Fault-Tolerant Data and Compute Intensive Programs over a Network of Workstations

J.Smith

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

Abstract. An application structured as a fault-tolerant bag of tasks adapts easily to changing resources. To be represented by a single bag of tasks, a computation must decompose into purely independent tasks. The work summarised here investigates performance of structuring approaches applicable where this ideal is not possible, partly through analysis and partly through measurements of a realistic fault-tolerant computation.

1 Introduction

Where applicable, the well known "bag of tasks" organisation for parallel computations has proved popular particularly in networked environments due to its transparent load balancing property. Examples include seismic computations [1] and materials science [13]. Typically the computation is controlled by a single master process and the data manipulated by the computation is located on a single disk with all I/O being performed by the master. A fault-tolerant version of this structure [6, 2, 10] allows cheap recovery since only the particular task affected by a machine failure needs to be recovered.

It is possible to increase capacity and bandwidth of storage at a single machine using RAID techniques [5], but in some computations the data manipulated outstrips the capacity of a single machine either in terms of volume or bandwidth requirements. It is then necessary to distribute data over multiple machines and then valuable gains may be made by taking advantage of computation structure to optimise I/O [7]. To ensure application fault-tolerance however a further mechanism is required, such as checkpointing.

It is possible however to avoid the single node bottleneck in a bag of tasks computation but it is necessary to take measures to ensure consistency of access to the distributed disk based state. An approach deriving from queued transaction processing [9] is demonstrated in [12] where computation state is persistent and located in a shared distributed object store and a recoverable queue [4] serves as a fault-tolerant bag of tasks. Writes to the shared state are enclosed in an atomic action (transaction), and abort of that action leads to rollback of writes enclosed in the failed action.

One of the computations implemented in the earlier work is dense Cholesky factorisation which does not decompose into purely independent tasks. In the earlier work an algorithm is employed directly from [8, §6.3.8.]. The matrix operands are partitioned

into blocks and each task entails computation of a single block of the output. The order of computation starting at the top left and proceeding down block columns and from left to right is represented in the queue, but it is necessary to delay an accessing slave until a block of the matrix becomes available. This is achieved through synchronisation flags, employed within the operations of the distributed matrix object itself.

Employing an additional synchronisation mechanism together with a single queue is one approach. An alternative requires no such synchronisation mechanism and instead relies on global barrier type synchronisation points to ensure that data is ready when required. A computation consists of a sequence of queues of which one is current at any time. All slaves wait till the current queue is actually empty before attempting to dequeue from the next in the sequence. 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. The resulting structure which supports a parallel loop programming style is similar to that of CALYPSO [3], but operating on disk based state.

One issue in the choice between such possible structures is the ease of programming at the application level, but important also is the achievable performance. This is clearly specific to each application. The work here investigates the performance of alternative realisations of Cholesky factorisation. The algorithm referred to above is used as the basis of two alternative realisations.

Single queue Where all tasks are stored in a single queue, they are all enabled at the start of the computation so it is necessary to employ a synchronisation mechanism separate to the queue so that dependencies may be satisfied. Use of an array of atomic flags is described in [12].

Homogeneous It is possible to avoid the need for synchronisation flags by loading tasks into a number of separate queues. In the structure considered here blocks on the diagonal are computed serially and all blocks in the same block column below the diagonal are computed in parallel.

Because of the dynamic nature of the computation structure it is not simple to predict parallel performance precisely. However where I/O is invariant with the number of slaves it is possible to characterise a computation by its single slave time and bounds on the minimum parallel time. Queue and synchronisation flag access cost is assumed negligible. If tasks are independent, a lower bound is the sum of all communications with the shared store and an upper bound the same plus the longest of all the task computation components. An alternate lower bound is reached when each task is computed by a separate slave. If there are inter-task dependencies then the computation may be modelled as if there are barriers to allow an upper bound to be obtained. Algebraic expressions for these bounds are derived in [12] for the single queue based realisation of Cholesky factorisation to allow extrapolation of the predicted performance to allow for upgraded hardware. Similar analysis for the alternate structure described here is detailed in [11]. Space restrictions here accommodate only the results.

The implementation of the single queue structure on a network of HP9000 based machines was described in [12]. For this work the application was ported to a network of 133 MHz Pentium machines, and the alternative application structure implemented in the same environment.

The Pentium machines have 32 Mbytes main memory and 256 Kbytes secondary cache. Hard disks are connected via fast SCSI 2 controllers. Most of the machines have 1 Gbyte IBM Pegasus disks, though the available scratch space on these disks is limited. Two of the machines have also a pair each of MAXTOR 540SL disks, providing an aggregate 2 Gbytes of storage. All the machines are running Linux version 2.0.23. The machines are connected by both a LinkBuilder FMS 100 Stackable Fast Ethernet Hub from 3Com and a ForeRunner ASX-200WG ATM switch with 155Mbit/s links. The following configurations are used.

Fast The object store is located on the IBM disk connected to a single machine and communications between slaves and store is via fast ethernet.

ATM The object store is distributed between the four MAXTOR disks. On each of the two machines hosting these disks, the two local MAXTOR disks are managed as a RAID-0 pair through the *md* [14] software. Communications between slaves and store is via ATM.

In each case the slaves are located on separate machines from those hosting the shared object store.

For block sizes above 250, the low level transfer rates for local memory to remote memory and to and from disk are found to be roughly constant. While the read and write cost for the IBM disk have nearly equal cost, the cost of writes to the RAID configuration is rather higher than the cost of reads. Clearly in the ATM configuration the limiting I/O rate should be double that offered by a single node, but a single slave can only use the bandwidth of a single node.

The computation rates vary for the different matrix primitives. In the overall computations however, matrix multiplication dominates and as tuned this primitive runs at about 27 Mflop/s for a block size above about 30. The experimental configuration is summarised below.

Configuration			Fast	ATM
Computation	(Mflop/s)		27	
Network	(Mbyte/s)		4.6	9.0
Disk	(Mbyte/s)	*read*	2.8	5.5
		write	2.8	2.4

2 Comparison

Figure 1 compares the maximum performance for the two computation structures for a matrix size of 4800^2 in the **fast** configuration. Upper and lower bounds on maximum performance calculated from the data in the table above are shown and so are a number of experimental measurements. The computation rate assumes an overall operation count of $\frac{n^3}{3}$. The performance is greatest for the single queue configuration and it can be shown analytically that this is generally true [11].

Figure. 2 shows the performance for the larger example of a 15000^2 matrix in the **ATM** configuration. Again both derived and measured performance are plotted. Here

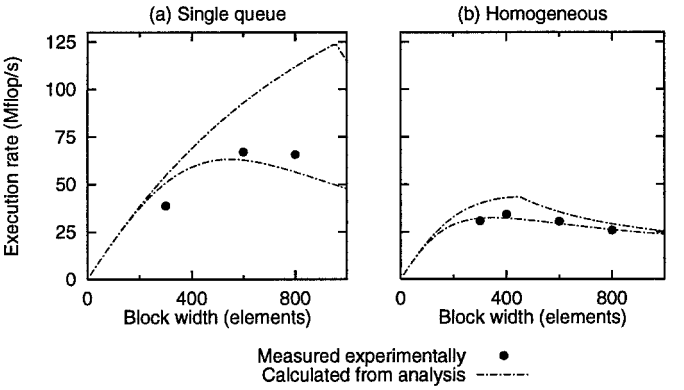

Fig. 1. Maximum performance of parallel Parallel Cholesky factorisation of a 4800^2 matrix for different synchronisation structures in the **fast** configuration.

though, five slaves do not exhaust the available bandwidth, so rather than the bounds on limiting performance, a lower bound on expected performance using just five slaves is plotted in each case. For the single queue this is obtained by dividing the single slave time by five. The approach is similar where multiple queues are used, but each phase of the computation is treated separately. It is seen that even the fault-tolerant

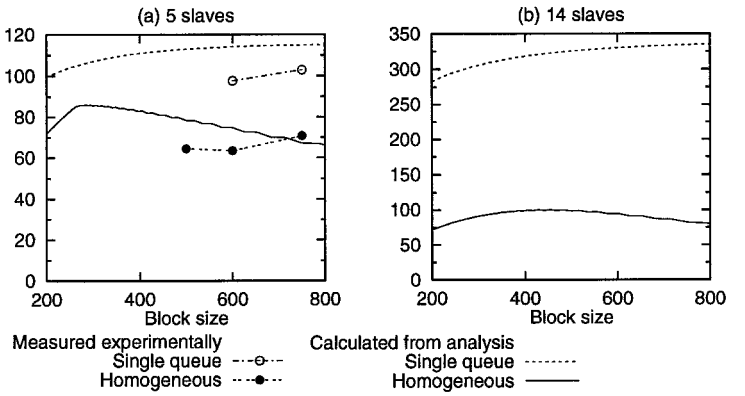

Fig. 2. Performance of parallel Cholesky factorisation of a 15000^2 matrix for different synchronisation structures in the **ATM** configuration.

implementations achieve performance which is close to that predicted, so that even with the assumptions made, the modelling process is not unrealistic.

Significantly better performance can be obtained using the single queue configuration particularly when the number of slaves is high. A simple modification to the homogeneous structure is to include computation of a block on the diagonal in the same task with the block immediately to the left. The new structure is found to perform better than the homogeneous structure, but only approaches the performance of the single queue

structure for small block size, of about 250, and for a small number of slaves [11]. Intuitively the single queue structure offers least restriction to parallelism so long as the cost of synchronisation is small.

Overall the evidence suggests that the single queue approach cannot be abandoned for performance reasons.

Acknowledgements

The support of all the Arjuna team is gratefully acknowledged, and in particular collaboration with S. Shrivastava in earlier experiments which this work builds upon and the assistance of M. Little, G. Parrington and S. Wheater with implementation issues relevant to this work.

References

1. G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 2nd edition, 1994. ISBN 0-8053-0443-6.
2. D. E. Bakken. *Supporting Fault-Tolerant Parallel Programming in Linda*. PhD thesis, The University of Arizona, Aug. 1994.
3. A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A novel software system for fault-tolerant parallel processing on distributed platforms. In *4th International Symposium on High Performance Distributed Computing*. IEEE, Aug. 1995.
4. P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. *ACM SIGMOD*, pages 112–122, 1990.
5. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
6. T. Clark and K. P. Birman. Using the ISIS resource manager for distributed, fault-tolerant computing. Technical Report 92-1289, Cornell University Computer Science Department, June 1992.
7. J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, pages 59–68, Mar. 1994.
8. G. H. Golub and C. F. V. Loan. *Matrix Computations*. John Hopkins University Press, second edition, 1989. ISBN 0-8018-3772-3.
9. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.
10. K. Jeong. *Fault-Tolerant Parallel Processing Combining Linda, Checkpointing, a nd Transactions*. PhD thesis, New York University, Jan. 1996.
11. J. Smith. *Fault Tolerant Parallel Applications Using a Network Of Workstations*. PhD thesis, University of Newcastle upon Tyne, 1996. Forthcoming.
12. J. A. Smith and S. Shrivastava. Performance of data and compute intensive programs over a network of workstations. *Theoretical Computer Science*, 1997. To appear in special issue for Euro-Par'96 papers.
13. V. S. Sunderam, G. A. Geist, J. J. Dongarra, and R. J. Manchek. The PVM concurrent computing system: Evolution, experiences, and t rends. *Parallel Computing Vol. 20(4)*, pages 531–546, 1993.
14. M. Zyngier. *md*. ftp://sweet-smoke.ufr-info-p7.ibp.fr/pub/Linux/, Apr. 1996. version 0.35.