Performance Analysis of Bulk Synchronous Parallel Algorithms

Wellington Santos Martins
e-mail: wsm@dei.ufg.br
Departamento de. Estatística e Informática
Universidade Federal de Goiás, Goiânia, Brazil

Abstract. In the last few years, there has been considerable interest in general purpose computational models of parallel computation to permit independent development of hardware and software. The BSP and related models represent an important step in this direction. This paper presents a methodology for the performance analysis of bulk synchronous parallel algorithms based on parameters which reflect the two-level memory hiearchy advocated by these models. A parallel sorting algorithm is taken as a case study where it is shown a close agreement between theoretical and experimental results

1. Introduction

Parallel machines are in widespread use but most applications use architecture dependent software which is thus not portable and quickly becomes obsolete with a new generation of parallel machines. Some researchers [1, 3, 4, 9] argue that a parallel model of computation is required which separates hardware and software development so that portable software can be developed for a range of different parallel architectures.

Parallel models of computation can broadly be classified into two categories: PRAM (shared memory) based models and network (nonshared memory) models. The former type has been extensively used for analysing the complexity of parallel algorithms. In this idealised model processors operate completely synchronously and have access to a single shared memory whose cells can be accessed in unit time. Although these assumptions ease the design and analysis of parallel algorithms they usually result in algorithms which are unsuitable for direct implementation on current parallel machines. Network models are based on real parallel machines but, by including realistic costs for operations, they make analysis more difficult. In particular, by including the topology of the architecture in the model, algorithms map with different degrees of difficulty on to machines with different topologies. The Bulk-Synchronous Parallel (BSP) model developed by Valiant [9] attempts to bridge the gap between these two types of model.

The BSP model - and related models - define a general purpose computational model in which the programmer is presented with a two-level memory hierarchy. Each processor has its own local memory and access to a common shared memory. Logically, shared memory is a uniform single address space although physically it may be implemented across a range of distributed memories. The model abstracts the topology of the machine by characterising the communication network via two parameters (*L* and

g), which are related to the latency and bandwidth respectively. Global accesses are costed using these two parameters and the lower the values the lower the communication costs. This is the case in the idealised PRAM model where global operations are assumed to take the same time as local operations. In existing parallel machines, however, these values are considerable higher and dependent on the access patterns to global memory. Valiant [8] has shown that by introducing some random behaviour in the routing algorithm (two-phase randomised routing) it is possible for real machines to maintain good bandwidth and latency. This enables real machines to reflect a two-level memory hierarchy, thus hiding physical topology from the programmer.

2. BSP and Related Models

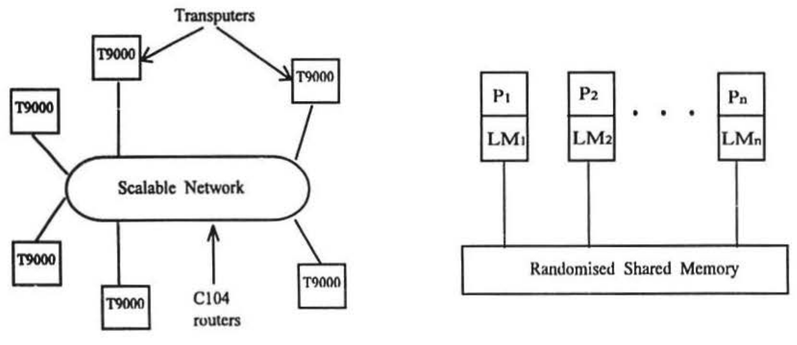
The BSP and related models aim to provide a simple view of a parallel machine such that the programmer is able to design and analyse algorithms whose performance is predictable in real machines. However, the way a computation is viewed, as well as the way performance analysis of algorithms is accomplished, may differ from one model to another.

A computation in the BSP model consists of a sequence of supersteps separated by barrier synchronisation. In a superstep local operations can be carried out and messages can be sent and received to implement global operations read and write. These communication events, however, are not guaranteed to terminate before the end of a superstep. Therefore remote data requested in a superstep can only be used in the next superstep. The performance of a BSP algorithm is determined by adding the costs of its supersteps. Individual supersteps are costed by analysing their total computation and communication demands. The global parameters L and g, together with the number of processors p and the problem size n, are used in this analysis. The parameter L represents the synchronisation periodicity of the machine, whereas g is related to the time required to realise h-relations in a situation of continuous message traffic; a h-relation is defined as a routing problem where each processor sends and receives h messages. The cost of a superstep is then given by $\max\{ L, c, g.h_s, g.h_r \}$ where c is the maximum time spent in local operations, h_s is the number of messages sent and h_r is the number of messages received in that superstep. Alternative cost definitions can be used, depending on the assumptions made about the implementation of the supersteps.

In contrast with the bulk synchronous nature of BSP algorithms, the execution of algorithms on the LogP model [1] can be viewed as a number of separately executing processes which are asynchronous with respect to each other. Communication and synchronisation among the processes is performed via message passing. A message sent to a processor can be used as soon as it arrives, instead of having to wait a barrier operation as in the BSP model. The LogP model defines four main parameters: P, o, L and g. P represents the number of processors. The parameter o is defined as the overhead associated with the transmission or reception of a message. The parameters L and g, although using the same name as in the BSP model, have different meanings. The parameter L sets an upper bound on the latency incurred in sending a small message whereas g is defined as the minimum time period (or gap) between consecutive message transmissions or receptions; the reciprocal of g being the bandwidth per processor. The parameter g is similar to that in the BSP model, in that it provides a measure of the efficiency of message delivery. However, since there is no implicit synchronisation in the LogP model, the notion of supersteps performing h-relations does not apply to this model. The model assumes that the network has a finite capacity, i.e. each processor can have no more than L/g outstanding messages in the network at any one time. Processors attempting to exceed this limit are stalled until the message transfer can be initiated. This is in contrast to the BSP model where any balanced communication event can be done in $g.h$ time. The performance of LogP algorithms can be quantified by summing all computation and communication costs of the algorithm. Communications are costed in terms of primitive message events. For example, the cost for reading a remote location is $2L + 4o$. Two message transmissions are required, one requesting and another sending the data. In each transmission each processor involved in the operation spends o time units interacting with the network and the message takes L time units to get to its destination. The cost of a writing operation is the same, although in this case the response is the acknowledge required for sequential consistency. This analysis assumes the data fits into a single transmission block. When dealing with a block of n such basic data, the cost becomes $2L + 4o + (n-1)g$, assuming $o < g$. This is because after the first transmission, subsequent transmissions have to wait g time units. The LogP model encourages the use of balanced communication events so as to avoid a processor being flooded with incoming messages; a situation where all but L/g of the sending processors would stall.

The WPRAM model [6] views the BSP and LogP models as architectural models. The following description is restricted to this level of abstraction. The WPRAM model attempts to extend the BSP model to a more flexible form. One important difference is that barrier synchronisation is not directly supported, instead message passing can be used to implement any synchronisation operation. This makes the WPRAM model closer to the LogP model. However, while the BSP and LogP models are applicable to a broad range of machine classes, the WPRAM model has been designed for a class of scalable distributed memory machines. This means that network latency should increase at a logarithmic rate with respect to the number of processors, i.e. $D = O(\log(p))$, and that each processor should be able to send messages into the network at a constant frequency, i.e. $g = O(1)$. The global parameters D and g are similar to L and g defined in the LogP model. However, instead of an upper bound given by a constant, the parameter D represents a mean delay which increases logarithmicly with the number of processors. It is modelled by the mean network delay resulting from a continuous random traffic and it thus includes the effects of switch contention and contention for shared data at the destination processor. The parameter g is the same as that of the LogP model, though no limit is imposed on the total number of outstanding messages a processor can have in the network. Because the network is capable of handling a constant maximum frequency of accesses per processor, the analysis of WPRAM algorithms is facilitated since the programmer does not have to be concerned with a network limit capacity, as is the case with LogP algorithms. Besides the global parameters L and g, the WPRAM model defines a number of other machine parameters. These parameters have been incorporated in a simulator so that execution time of WPRAM programs can be obtained. This allows one to determine the relative importance of various low level parameters on the performance of algorithms.

The models described have in common the two-level memory structure with uniform global access. Despite some differences as the number and meaning of the parameters, performance analysis in these models becomes very similar when barrier synchronisations are used in the algorithms. The performance of one such bulk synchronous algorithm, a parallel sort with balanced merge, is analysed in this paper. The analysis follows a proposed methodology which is based on performance parameters reflecting the two-level memory hierarchy advocated by these models. The theoretical results are then compared with the more accurate results produced by the WPRAM simulator.



(a) Target Architecture (b) Programmer's View

Figure 1 - WPRAM simulator

3. WPRAM Computational Model and WPRAM Simulator

An architectural level description of the WPRAM was given previously. At a higher, computational level, the WPRAM (computational) model can be described as consisting of P processors which operate asynchronously and which have access to a shared address space with weak coherency semantics (the 'W' in WPRAM stands for Weak coherency). This means that newly written data is only guaranteed to be visible to other processes when the writer and readers synchronise in some way. Forms of synchronisation provided include process creation, barrier synchronisation and tag synchronisation, which are used to co-ordinate processing and maintain shared data consistency. Also important is the fact that the shared address space distinguishes two forms of data: global data which is randomly distributed amongst all processors and local data which is mapped to a single processor memory. The WPRAM (computational) model can be mapped to real machines conforming to the WPRAM (architectural) model requirements, that is $D = O(\log(P))$ and $g = O(1)$; the model also assigns performance $O(1)$ to local operations and $O(D + X)$ to the access of X remote words. One such mapping was done by implementing the WPRAM model on a simulator (WPRAM simulator) using performance figures for the T9000 transputer and C104 packet router. A full description of the simulator and mapping can be found in [6].

The WPRAM simulator mentioned earlier was used to obtain the experimental results given in this paper. The simulator is based on the interaction of processes that are used to represent both the nodes of the target machine and the user processes. Algorithms are implemented using a programming interface and can be subsequently executed directly on the simulator. This way the sequence of operations generated by the program drives the simulator (execution-driven discrete-event simulation). The WPRAM target architecture for the simulator is a distributed memory machine, which supports uniform global access by the use of data randomisation[1], see Figure 1. The simulator includes a

[1] The use of data randomisation, where data is distributed throughout the local memories, obviates the need for randomised routing.

detailed performance model which costs operations based on measured performance figures for the T9000 transputer processor and simulations of the C104 packet router. Local operations modelled by the simulator include arithmetic calculation, context switching, message handling and local process management. Messages entering the network are assumed to be split up to guarantee that no one message ties up a switch for long periods of time, while global data is assumed to be randomised based on the unit of a cache line. Global operations are costed based on the high level parameters g and D. The value for g was obtained from the Esprit PUMA Project while D was derived from a simulation of the C104 router carried out at Inmos. Work on validating the simulator results has been carried out at Leeds with a close match being found between theoretical predictions and experimental results.

The simulator is written in C and provides a rich programming interface to execute algorithms written in C. The programming interface consists of a set of library procedures which support process management, shared data access and process synchronisation. Only a small subset of these library calls were used in this research, including: procedures for process management (fork, join, my_node and my_index), read and write procedures for data access, and a procedure to barrier synchronise processes.

4. Experimental Methodology and Performance Composition

The methodology proposed here consists of two parts. First, parameters describing local and global performance are defined and, based on them, performance equations for the algorithms are derived. This mathematical model is used to predict the performance (execution time) of the algorithms by specifying the chosen number of processors and the size of the input data. Secondly, the algorithms are implemented on a machine and their performance measured. Close agreement between prediction and experimental results validates the mathematical model.

Due to their well-defined structure, most bulk synchronous algorithms are easy to analyse. The total execution time can be obtained by simply summing up all computation and communication contributions of each (super)step. The analysis assumes that, in all segments of the program, the computation and communication operations do not overlap and can thus be simply added to give the total execution time of each segment.

5. A Case Study: Parallel Sort using a Balanced Merge

The use of the proposed methodology is illustrated in this section by analysing the performance of a parallel sorting algorithm due to Francis and Mathieson [2]. The algorithm removes the linear time bottleneck inherent in simple extensions of sequential algorithms by employing a balanced merge algorithm that utilises all processors in all steps of the merge phase.

Figure 2 illustrates the algorithm for $n = 16$ and $p = 4$. Each processor initially sorts n/p data items and at every following stage each processor finds n/p data items in a set of data items and merges them. For example, for a system of two processors, the data would initially be split into two data sets and each processor would sequentially sort its own data set. After this each processor would, in parallel, find the $n/2$ smallest data items and the $n/2$ largest data items in the complete set. This is equivalent to partitioning the two smaller data sets each into two segments where the number of data items in the pair of segments containing the smaller values is the same as the number of values in the pair of segments containing the larger values. The two segments containing the smaller values are then merged as are the two segments containing the larger values. The resultant data sets are then concatenated to form the result.

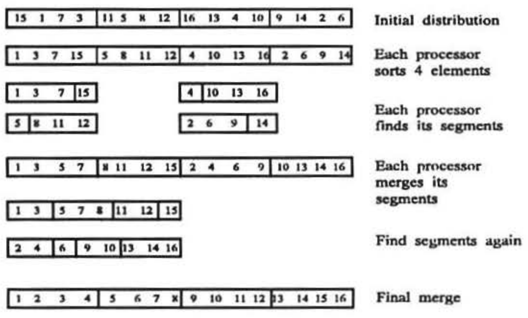


Figure 2 Balanced mergesort for n = 16 and p = 4

The calculation of the partition position in a data set is complex; the algorithm is defined in [2].

Analysis

The parameters describing local (k_s, k_m) and global (k_i, k_g) performance were obtained by experimentation.

The first two parameters, k_s and k_m, were used to model local computations when sorting and merging respectively. Since the bound of sequential mergesort (which was the sequential sorting used) is $O(n.\log(n))$ and that of sequential merge is $O(n)$, it is reasonable to assume that the time of sorting n integers is $t_s(n) = k_s.n.\log(n)$ and the time of merging n integers is $t_m(n) = k_m.n$. A simple approximation to k_s and k_m was obtained by dividing the total run time (for n = 200, 400, 600, 800 and 1000) in microseconds by $n.\log(n)$ and n respectively. Using a single processor, each implementation (sort and merge) was run on 5 different data sets generated randomly, and the average run time was used in the calculation of k_s and k_m.

The parameters k_i and k_g were used to model global communications when accessing a single integer and a sequence of integers in global memory (remote access) respectively. The bound of accessing X remote words in the WPRAM is $O(D + X)$ where D corresponds to the network latency $D = O(\log(P))$. However in order to facilitate the performance analysis conducted, and because most remote accesses required by the algorithms are pipelined, the latency was modelled as a constant. The bound of accessing n integers then becomes $t(n) = O(n)$. To obtain an approximation for $t(n)$ a simple program was developed in which n integers were read/write from/to global memory. It was observed that $t(n) = 23.\text{ceiling}(n/4) + 24$. In the WPRAM simulator remote data is moved in blocks of 16 bytes (corresponding to 4 words), what explains the step function found. Thus the value for k_i, access to a single integer, and k_g, access to n integers, were approximated to 47 and 5.7 microseconds respectively.

Computation

As expected, the computation time of this algorithm does not have any linear component in n. Initially, each processor sorts n/p elements and then they all participate in each step of the merge phase, each one producing exactly $1/p$th of the final merged data. The computation time is given by:

$$(1)\quad \mathrm{comp}(n,p) = \mathrm{sort}\left(\frac{n}{p}\right) + \mathrm{merge}\left(\frac{n}{p}\right)\log(p)$$

Using the parameters defined, this equation can be simplified to:

$$(2)\quad \mathrm{comp}(n,p) = k_s\,\frac{n}{p}\log\left(\frac{n}{p}\right) + k_m\,\frac{n}{p}\log(p)$$

Communication

Figure 4 illustrates how such implementation can be done. Each processor starts by reading $n/8$ elements, sorting them and writing the result to global memory. The merge phase has, in this case, three steps. In each step, the processors merge and write $n/8$ elements. The partition's boundaries are calculated using a binary search. The decision whether to copy the entire segments to be searched to local memory or to read the elements individually as they are required, is dependent on the machine parameters and data input size. For the machine (WPRAM simulator) and data input size (1k, 5k and 10k) considered, it was verified experimentally that the first option is better only when the number of processors is small (less than 4). Hence it was decided to use the second alternative, where elements are read individually. The search space size varies in each

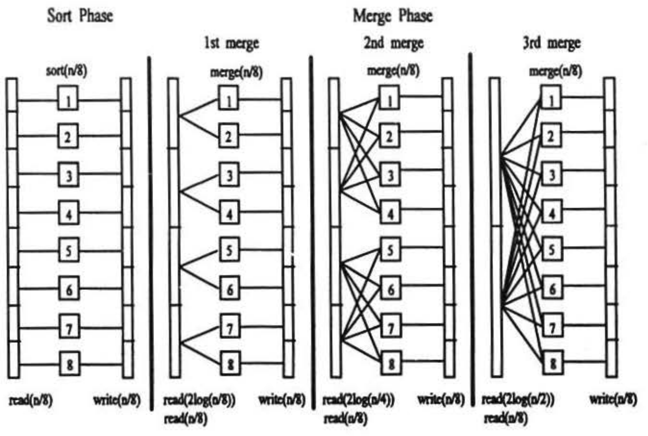


Figure 4 Action of 8 processors in the balanced mergesort algorithm

step of the merge phase thus, for the example considered, 2log(*n*/8) are read in the first step, 2log(*n*/4) in the second and 2log(*n*/2) in the final step. In each of these steps, once the partition's boundaries have been calculated, the corresponding *n*/8 elements are read, merged and written back. The communication time of this algorithm is given by:

(3)

$$\mathrm{comm}(n,p) = \mathrm{read}\left(\frac{n}{p}\right) + \mathrm{write}\left(\frac{\mathrm{n}}{\mathrm{p}}\right) + \sum_{i=1}^{\log(p)}\left(\mathrm{read}\left(2\log\left(\frac{n}{2^i}\right)\right) + \mathrm{read}\left(\frac{n}{p}\right) + \mathrm{write}\left(\frac{n}{p}\right)\right)$$

which can be simplified to

(4) $$\mathrm{comm}(n,p) = \left(2\log(n) - \log(p) - 1\right)\log(p)k_i + \left(2\frac{n}{p} + 2\log(p)\frac{n}{p}\right)k_g$$

The total execution time of the balanced mergesort algorithm is then found by adding (2) to (4):

(5) $$\mathrm{T}_{exec}(n,p) = \left(k_s\log\left(\frac{n}{p}\right) + 2k_g + (k_m + 2)\log(p)\right)\frac{n}{p} + \left(2\log(n) - \log(p) - 1\right)\log(p)k_i$$

By substituting the coefficients k_s, k_m and k_g for their values, equation (5) can be simplified to:

(6) $\mathrm{T}_{exec}(n,p) =$

$$11.4\frac{n}{p} + 1.7\frac{n}{p}\log\left(\frac{n}{p}\right) + 3.7\frac{n}{p}\log(p) + 94\log(n)\log(p) - 47(\log(p))^2 - 47\log(p)$$

which gives the total execution time in microseconds.

Results

The predicted and measured time for the balanced mergesort algorithm is shown in figure 5. The results were obtained for input sizes of 1K, 5K and 10K, using 2, 4, 8, 16 and 32 processors. As shown in the graph, the balanced mergesort gives a good performance overall. It removes the bottleneck existing in other sorting algorithms [7], improving performance more uniformly with the number of processors.

The graph also shows that there was close agreement between predicted and experimental results; measured values being all within 1% of the predicted values. However, there is no guarantee that the WPRAM simulator mimics the WPRAM model and hence the simulator needs to be validated against the machine model. Recently, a mapping of the WPRAM model has been implemented to a real parallel machine [5], the KSR machine, and results obtained with this implementation have been shown in accordance with corresponding results obtained with the WPRAM simulator, thus validating the mapping.

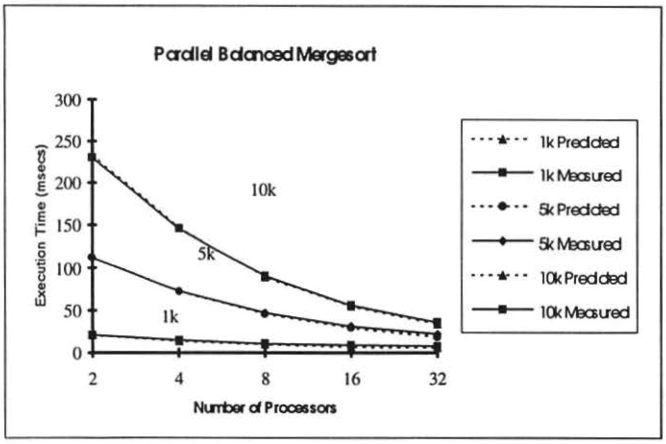


Figure 5 Results for Parallel Balanced Mergesort

6. Conclusions

Recent proposed models of parallel computation abstract the topology of the machine by characterising the communication network via parameters related to latency and bandwidth. This paper presented a methodology for the performance analysis of bulk synchronous parallel algorithms based on parameters which reflect the two-level memory hierarchy advocated by these models. The proposed methodology was illustrated with the performance analysis and implementation of a parallel sorting algorithm. By deriving performance equations for the algorithm it was shown that similar results to those produced by the simulator, which includes a much more detailed performance model, can be obtained without the step of coding and simulating the algorithm. It is then expected that such analysis can be helpful when developing algorithms for real parallel machines conforming to the WPRAM's requirements. In the specific case of the sorting algorithm implemented, for example, the decision of whether to copy a whole segment (n/p) to carry out a binary search or to retrieve each item individually, will depend on their costs, $(n/p).k_m$ and $\log(n/p).k_i$ respectively. Given a particular machine, there will be certain values for n and p for which one approach will outperform the other.

7. Acknowledgements

I would like to thank Jonathan Nash (from University of Leeds, UK) for the WPRAM simulator and the support of the Depart. Estatística e Informática (UFG, Goiânia, Brazil).

References

[1] Culler, D., Karp R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R. and vonEicken, T. "LogP: Towards a Realistic Model of Parallel Computation". In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP, San Diego, California*, volume 28, pages 19-22, ACM Press, may 1993.

[2] Francis, R. S. and Mathieson, I. D. "A Benchmark Parallel Sort for Shared Memory Multiprocessors", *IEEE Transactions on Computers*, Vol. 37, No. 12, pp. 1619-1626, December 1988.

[3] McColl, W. F. "General purpose parallel computing". In A. M. Gibbons and P. Spirakis editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation, volume 4 of Cambridge International Series on Parallel Computation*, pages 337-391. Cambridge University Press, Cambridge, UK, 1993.

[4] Nash, J. M. and Dew, P. "Parallel Algorithm Design on the XPRAM model". In *Proceedings of the 2nd Abstract machines Workshop*, Leeds, 1993.

[5] Nash, J. M., Dyer, M. E. and Dew, P. "Designing Practical Parallel Algorithms for Scalable Message Passing Machines". In *Proceedings of the 1995 World Transputer Congress*, pages 529-541, September 1995.

[6] Nash, J. M. "A study of the XPRAM Model for Parallel Computing", *PhD Thesis, University of Leeds*, 1993.

[7] Dowsing, R. D. and Martins, W. S. "Performance of a Selection of Sorting Algorithms on a General Purpose Parallel Computer", To appear in *Concurrency: Practice and Experience.*

[8] Valiant, L. G. "General Purpose Parallel Architectures". *In the Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. North Holland 1990, pp. 944-971.

[9] Valiant, L. G. "A Bridging Model for Parallel Computation", *Communications of ACM*, pp. 103-111, August 1990.