Parallelising the Mean Value Analysis Algorithm

Claudio Gennaro* and Peter J.B. King**

** Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy; E-mail: gennaro@elet.polimi.it; ** Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, United Kingdom; E-mail: pbjk@cee.hw.ac.uk*

The Mean Value Analysis (MVA) algorithm is one of the most popular for evaluating the performance of separable (or product-form) queueing networks. Although its complexity is modest when jobs are indistinguishable, the introduction of different customer classes rapidly increases its computational cost. The problems of parallelising the algorithm while retaining its conceptual simplicity are examined. In particular, a parallel implementation of MVA on a distributed memory machine is developed using the MPI library for communication.

Keywords: Mean value analysis, queueing networks, product form, pipelined algorithm, parallel simulation

1. Introduction

Ever since the pioneering work of Scherr [1] in applying the machine repairman problem to analysing the performance of computer time shared systems, queueing theory has been a tool of the computer performance analyst. A queueing network is a collection of stations (or service centers) arranged in such a way that customers proceed from one to another in order to fulfill their service requirements. When applying queueing networks to computer systems, the stations represent the various system resources (e.g., CPUs, channels, disks, etc.), while the customers represent jobs in the system. Many queueing networks do not have closed-form analytic solutions and cannot be evaluated other than by Monte Carlo simulation. Simulations are expensive to perform, and the results which are calculated are only known to fall within certain confidence intervals. Queueing analysts found closed-form solutions for increasingly complex queueing networks through the 1960s, and in 1975 the discovery of the class of separable or product-form queueing networks was announced [2]. This class of networks represents the most complex queueing networks that can be effectively evaluated numerically today. Several algorithms exist to evaluate the performance of such networks with complexities that depend linearly on the number of customers and the number of stations in the network.

Numerical algorithms for evaluating these networks are now widely known, but few parallel implementations are known. This paper describes the design and implementation of a Mean Value Analysis (MVA) algorithm on a distributed memory system, using the MPI library for communication.

We start by describing previous related work. The MVA algorithm is described, and its complexity is demonstrated. The following section gives our approach to parallelisation. The implementation is described next, followed by practical evaluation of the code on some test problems. Finally, conclusions are drawn and suggestions for further work are made.

Received: January 1998; Revised: July 1998; Accepted: September 1998

TRANSACTIONS of The Society for Computer Simulation International
ISSN 0740-6797/99
Copyright © 1999 The Society for Computer Simulation International
Volume 16, Number 1, pp. 16-22

2. Previous Work

Queueing networks in which customers circulate between service centres have the potential to be extremely difficult to analyse because of the interdependency of the different service centres. Jackson [3] showed that open networks with all services exponentially distributed and Poisson arrivals could be analysed as if the service centres were independent M/M/1 queues. Baskett et al. [2] were able to extend this analysis to a more general framework, allowing non-exponential services in some instances, and also different classes of customers with their own routing behaviours. They showed the probability of the network being in a state $\mathbf{n} = (n_1, n_2, ..., n_M)$, when there are M service centres, is of the form:

$$\Pr(\mathbf{n}) = \frac{1}{G} \prod_{i=1}^{M} f_i(n_i)$$

If the network is open, the factor G is 1, and the service centres are essentially independent of one another, only interacting through the routing of customers to other centres after service. When the network is closed, however, the factor G, which ensures that the probabilities are normalised, introduces a dependency between the service centres. If the total customer population is n, say, then the fact that centre 1, say, has five customers present, implies that there can only be $n - 5$ to be distributed around the other service centres. Obvious approaches to calculating G, for example, by summing over all possible states, soon run into practical problems because of the number of states involved, not to say numerical difficulties such as round-off.

Algorithms to numerically evaluate these networks have been the subject of much interest. Buzen [4] developed the first algorithm, known as the convolution algorithm. This algorithm finds the normalisation constant G for a network of M centres and N customers using a simple recurrence relating $G(M, N)$ to $G(M - 1, N)$ and $G(M, N - 1)$. Other performance metrics, such as mean queue lengths, centre utilisations, etc., are found using G. Although very efficient, the convolution algorithm is not very intuitive, and its computations can be affected by overflow or underflow for large networks. Reiser and Lavenberg [5] developed a new algorithm, Mean Value Analysis (MVA), that uses only meaningful metrics of network performance in its calculation. Essentially, the performance of the network when N customers are present is evaluated using the performance of the network when there are $N - 1$ customers. Metrics such as utilisation and mean queue length are produced as a side effect. The normalisation constant is not calculated. MVA is of similar complexity to convolution. Other algorithms such as LBANC and CCNC [6] and RECAL [7] have also been developed. They all have similar complexities, although in particular cases, one algorithm or another may be favoured.

A major advance in speeding up these algorithms has been the recognition of the tree structuring apparent when different classes of customer only visit subsets of the service centres. It is then possible to significantly simplify the calculations for those stations which are not visited by particular classes of customer. The full complexity of the algorithm needs to be applied only to service centres where different classes of customer interact. This simplification was originally discovered by Lam and Lien [8] and applied to the convolution algorithm. It can also be applied to MVA and RECAL.

Parallel implementations of a number of these algorithms have been proposed. Greenberg and McKenna [9] developed a parallel version of RECAL for use on shared memory multiprocessors. Pace and Tucci [10] worked with MVA. Greenberg and Mitrani [11] have developed a technique using fast Fourier transforms to evaluate the normalisation constant G in parallel. Hanson et al. [12] also used MVA. Most of these algorithms have been implemented or proposed for a shared memory environment. It is a feature of all the algorithms that they build up their solutions iteratively, either from the solutions of the same network with smaller populations, or from solutions of a smaller network with the same population. Shared memory means that earlier results are easily available on all processors.

Our interest is in the development of an algorithm which is effective in a distributed memory environment. Here each processor has its own storage, and data calculated on one processor is not available to other processors without explicit transmission to the other processor's memory. Inter-processor communication needs to be minimised, because it will typically be several orders of magnitude slower than computations.

3. MVA

Mean value analysis operates by relating the performance of the network when n customers are present to the performance when $n - 1$ are present. Since the performance when there are 0 customers is known trivially, calculations proceed using increasing populations, from 0 to N. The evaluation of performance for a particular population involves iteration over all stations in the network. When the population is N, and there are M stations, the complexity of the algorithm is $O(MN)$.

If there are R classes of customer, then the performance when the population vector is $\mathbf{n} = (n_1, n_2, ..., n_R)$ is calculated using the performance at populations $(n_1 - 1, n_2, ..., n_R)$, $(n_1, n_2 - 1, ..., n_R)$, ..., $(n_1, n_2, ..., n_R - 1)$. Given a final population for which we wish to calculate the performance, the calculations needed give a precedence relationship between the different populations. The population $\mathbf{0}$ precedes all other populations, and the other populations must be calculated. The order of calculation is not totally determined, since the precedence relationship is only a partial ordering.

4. The Algorithm

The aim of a parallel algorithm for MVA must be to calculate the same results as a uni-processor MVA algorithm, while gaining significant speedup by performing some of the calculation in parallel. The precedence graph will put an upper bound on the amount of parallelism that is possible.

We allocate a processor to be responsible for each population. Each processor may be allocated more than one population. As soon as the preceding populations have been calculated,

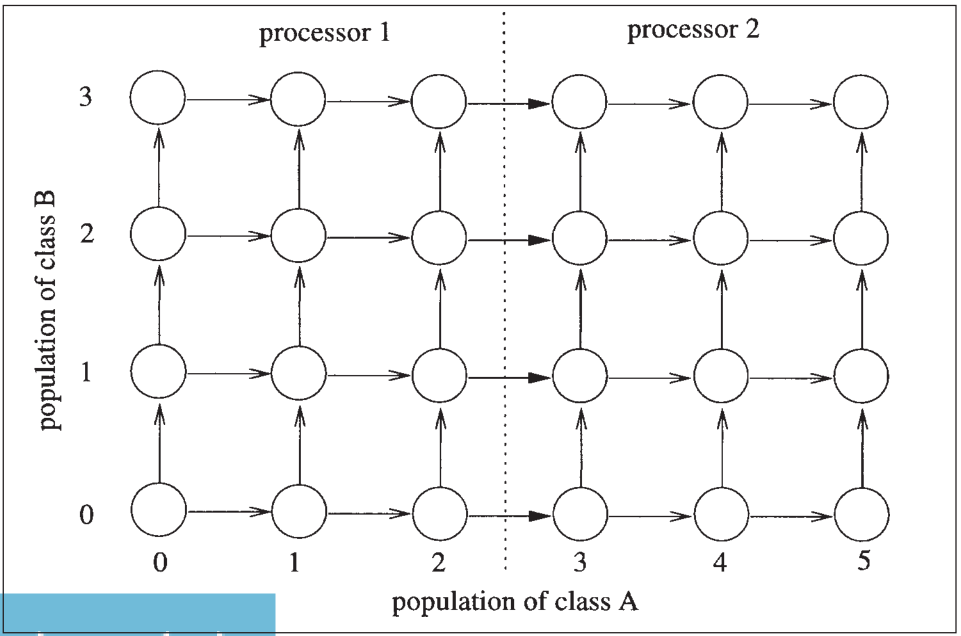

Figure 1. Example of the PMVA algorithm with two classes and two processors

calculations can start. If the preceding populations were allocated to different processors, then the performance vector must be transmitted between the processors. If the preceding population was calculated on the same processor, then no communication is needed.

Even if an unbounded number of processors were available, it would not be sensible to allocate only a single population to a processor. The communications cost in that case would overwhelm most of the speedup obtained by parallelism of the computation. We anticipate that a modest number of processors will achieve an almost linear speedup.

For simplicity, we allocate a population to a processor based only on the population of the first class of customers. This is easily implemented and gives a significant speedup. It might be possible to run some form of processor allocation algorithm which toured the precedence graph in order, allocating the population to a particular processor depending on whether the processor was already allocated, and on the identity of the processor used for neighbouring populations in the graph.

4.1 The Implementation

Figure 1 shows the precedence graph of computation in the case of queueing network with two customer classes (named A and B). Class A has a population of 5 customers and class B, 3 customers. The node at coordinates (r, s) corresponds to the computation of the statistics (queue lengths, response times, etc.) when in the network has r customers of class A and s customers of class B. The calculation ends when the node at coordinates (5, 3) has been evaluated.

In the case of a two-processor implementation, the nodes are partitioned solely on the basis of the population of class A customers. Processor 1 calculates those nodes that have class populations from 0 to 2 inclusive, and processor 2 calculates those with class A populations of 3 or more. In general, when the final population is (m, n), processor 1 is allocated populations (i, j) for $0 \le i \le k$ and processor 2 is allocated $k < i \le m$, where $k = m/2$. In Figure 1, $k = 2$, and processor 1 starts alone and executes the computation at nodes $(0, 0), (1, 0), ..., (k, 0)$, in order. It then sends the results of node $(k, 0)$ to processor 2, which has been idle until this time. Processor 2 executes the nodes from $(k + 1, 0)$ to $(m, 0)$, and simultaneously processor 1 executes the nodes from $(0, 1)$ to (k, l). A pipeline is established with processor 1 executing node (r, s) with $0 \le r \le k$ and simultaneously processor 2 computes the results for $(t, s - 1)$ with $k + 1 \le t \le m$. Eventually, processor 1 reaches node (k, n), calculates the performance for that population and transmits it to processor 2. Processor 1 is then idle while processor 2 works on nodes $(k + 1, n)$ ... (m, n). When processor 2 reaches the node (m, n) and executes the corresponding computations, the algorithm terminates.

When more processors are available, the nodes are still partitioned between processors on the basis of their class 1 population. If a total class population of m customers is to be calculated, and there are p processors available, then each processor is assigned m/p values of class 1 population.

Networks with R classes of customer generate an R-dimensional precedence graph. Although more complex processor assignment algorithms would be possible, we have extended the two-dimensional algorithm. The nodes are partitioned on the basis of the population of class 1 customers. Processor 1 starts by evaluating the nodes for populations of class 1 from 0 to k, while all other classes have populations of 0. When population k is reached, processor 2 starts with population $k + 1$. Meanwhile, processor 1 has started the calculation of results for a population of 1 in class 2, again taking the class 1 population from 0 to k. When it reaches k, processor 2 should have finished computations for class 1 populations up to m, and be ready to calculate for population $k + 1$ again, but with a population of 1 in class 2.

4.2 Performance Prediction of the Algorithm

Letting N_i be the population of class i for $1 \le i \le R$, the execution time $T(1)$ of the algorithm on a single processor is given by:

$$T(1) = h(1) \prod_{i=1}^{R} (N_i + 1) \tag{1}$$

where $h(1)$ represents the mean time spent computing a node of the MVA algorithm in the case of $p = 1$. Since the parallel machines exploit cache mechanisms during the computations, we assume that h may depend on the number of processors. The execution time $T(p)$ with $p > 1$ is given by:

$$T(p) = T_{Calc}(p) + T_{Comm}(p) \tag{2}$$

where $T_{Comm}(p)$ is the time spent communicating and $T_{Calc}(p)$ the time spent computing. The term $T_{Calc}(p)$ can be estimated from the time to calculate one node, the number of processors, and careful accounting for the periods when not all processors are active. Each processor is responsible for a range of class 1 subscripts. Most processors deal with a range of size $\left\lceil \frac{N_1 + 1}{p} \right\rceil$ subscripts. The processors that are responsible for a smaller range will be idle for the processing of a single node. If we assume that the calculation of a single node takes $h(p)$, then the first processor will take time $h(p). \left\lceil \frac{N_1 + 1}{p} \right\rceil$ to calculate while processor 2 is idle. It will then proceed to calculate for the remaining populations of class 2 through to class R. Hence processor 1 will be busy for a time given by:

$$h(p) \left\lceil \frac{N_1 + 1}{p} \right\rceil \prod_{i=2}^{R} (N_i + 1)$$

Before the computation is complete, the pipeline must empty. This involves the remaining $p - 1$ processors each calculating for a time of $h(p) \left\lceil \frac{N_1 + 1}{p} \right\rceil$. Adding these terms we get:

$$T_{Calc}(p) = h(p) \left\lceil \frac{N_1 + 1}{p} \right\rceil \left(\prod_{i=2}^{R} (N_i + 1) + p - 1 \right) \tag{3}$$

This formula will be a slight overestimate if the processors are not all responsible for the same number of subscripts. Dividing them equally, if $N_1 + 1$ is not exactly divisible by p, one should give $\lceil N_1 + 1/p \rceil$ to processors 1, 2, 3, ..., k, and one fewer subscripts to processors $k + 1$, ..., p. The correction term for the pipeline emptying will be slightly smaller than that given above when the later processors have fewer subscripts for which to calculate.

In a similar manner, the communication time can be derived, assuming that all communications are synchronous.

$$T_{Comm}(p) = k(p)\left(\prod_{i=2}^{R}(N_i+1)+p-2\right) \qquad (4)$$

where $k(p)$ represents the mean time spent communicating the queue lengths of a node of the MVA algorithm from a processor to its successor in the pipeline when the number of processors is p[1]. Finally we can give the expression of the total time $T(p)$ by means the equations (2), (3) and (4):

$$T(p) = h(p)(N_1+1)\left(\frac{1}{p}\prod_{i=2}^{R}(N_i+1)+\frac{p-1}{p}\right) + k(p)\left(\prod_{i=2}^{R}(N_i+1)+p-2\right) \qquad (5)$$

5. Experimental Results

The pipelined implementation described above has been implemented on the Cray T3D machine at the Edinburgh Parallel Computing Centre. The algorithm was restricted to the case of load-independent service centres. This was only for implementation convenience, and does not represent a restriction on the applicability of the method. The C programming language was used, with all real values being expressed as **double** variables, which occupy eight bytes. The MPI message-passing library was used to provide interprocessor communications. Synchronous communication was used, so that the sender of a message would block until it had been successfully received.

5.1 The Case Study

The following parameters were chosen so that the program takes about 30 minutes elapsed time when running on a single processor:

- $R = 3$;
- $M = 5$;
- $N_1 = 4095$, $N_2 = 177$, $N_3 = 127$.

where M is the number of service centers in the queueing network.

5.2 Performance Obtained

The execution times[2], the speedups and the relative efficiency of the program with increasing number of processors are shown in the Figures 2, 3 and 4, respectively[3].

Figure 5 shows the average $k(p)$ with p from 2 to 512. Notice the difference between $k(2)$ and $k(p)$ when $p > 2$. This is due to the different behaviour of the pipeline and the use of synchronous communications among the processors (see Figure 6).

1 Notice that $k(p)$ includes the waiting time of the synchronous communications.

2 Because the T3D machine doesn't accept job running with only one processor, the execution time for the case $p = 1$ is considered to be equal to the time obtained with the program running on a two-processor job with population of class 1 given by $2(N1 + 1) - 1$, without communications, and keeping one of two processors idle.

3 The execution times don't include the I/O times for loading/writing the results of the MVA algorithm.

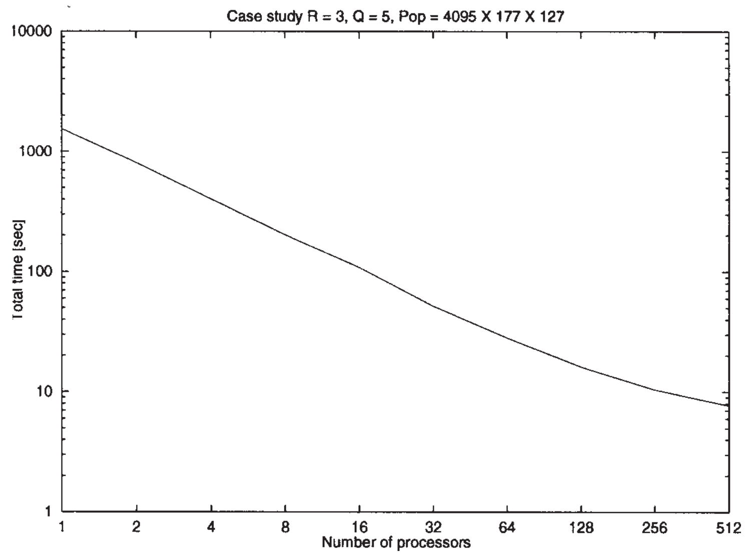

Figure 2. Execution times of the Pipeline MVA algorithm on Cray T3D machine

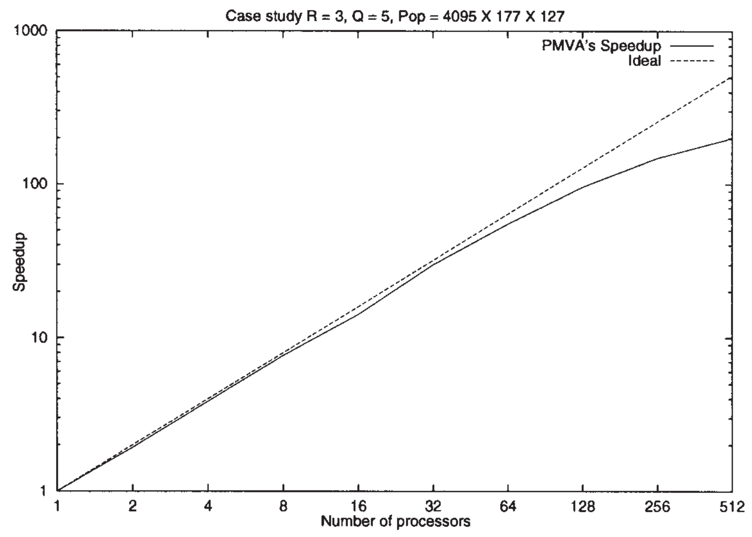

Figure 3. Speedup of the Pipeline MVA algorithm on Cray T3D machine

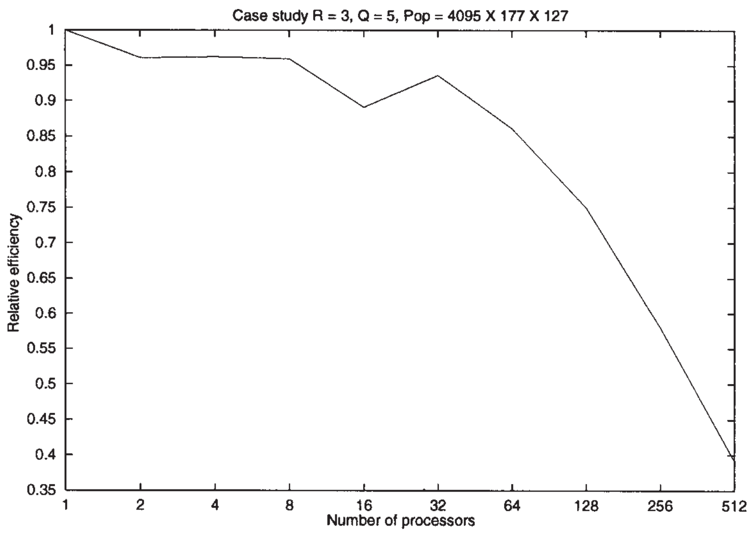

Figure 4. Relative efficiency of the Pipeline MVA algorithm on Cray T3D machine

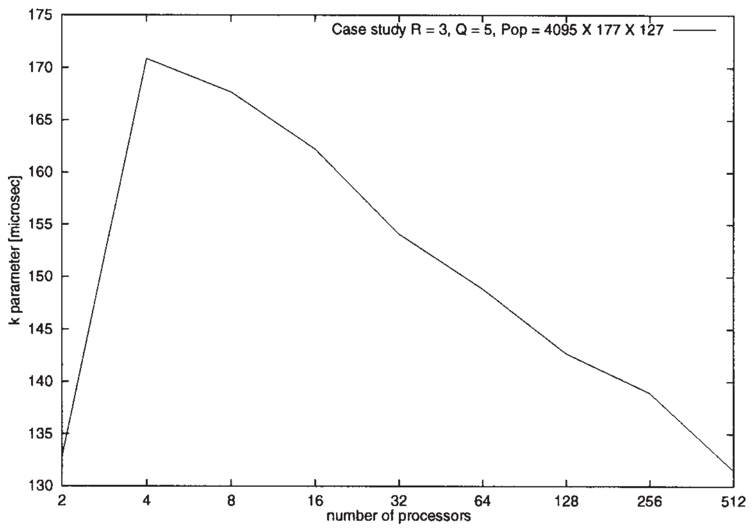

Figure 5. The k parameter of the Pipeline MVA algorithm on Cray T3D machine

Further when $p > 2$, the mean time communicating $k(p)$ decreases as the number of processors increases. This is because the relative amount of load imbalance decreases as the amount of work distributed amongst more processors decreases. Suppose a processor finishes a lot sooner than all the others; it will start to wait until the receiver is ready to receive; hence, its communication time will reflect this. As the amount of work is distributed, any potential load imbalance will decrease in proportion; hence, it will appear that the communication time is coming down.

5.3 Interpretation of the Results

Using Equation (1) and the experimental results, we can evaluate the parameter $h(1)$:

$$h(1) = \frac{T(1)}{\prod_{i=1}^{R}(N_i + 1)} = 16.6\ \mu sec. \tag{6}$$

In order to evaluate $h(p)$, when $p > 1$, we can exploit the measured time $K_i(p)$, that is, the total time spent communicating from the processor i. Since the measured time $T(p)$ is in practice the execution time of p-th processor, we obtain:

$$h(p) = \frac{p(T(p) - K_p(p))}{\prod_{i=1}^{R}(N_i + 1)} = 16.6\ \mu sec.^{4} \tag{7}$$

Figure 7 shows the parameter $h(p)$ obtained in this way. We observe that the computation time $h(p)$ depends on the number of processors. This effect is because of differing data being stored in the high-speed memory cache.

5.4 Improving the Performance of the Program

The performance of the Pipeline MVA algorithm depends strongly on the input problem. The speedup $S(p)$ of the program is given by dividing $T(1)$ by $T(p)$, that is:

$$S(p) = \frac{h(1)\prod_{i=1}^{R}(N_i+1)}{h(p)(N_1+1)(\frac{1}{p}\prod_{i=2}^{R}(N_i+1)+\frac{p-1}{p})+k(p)(\prod_{i=2}^{R}(N_i+1)+p-2)} \tag{8}$$

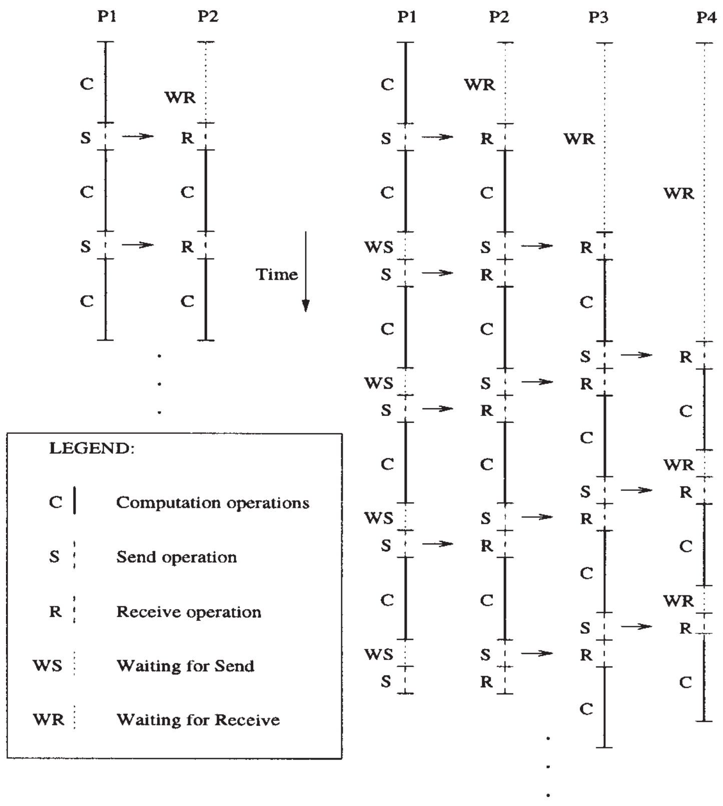

Figure 6. Behavior of the Pipeline MVA algorithm with two and four processors

Dividing both numerator and denominator of (8) by $k(p)$ and supposing $\alpha(p) = \frac{h(p)}{k(p)} \approx \frac{h(1)}{k(p)}$[5] we obtain:

$$S(p) = \frac{\alpha(p)\prod_{i=1}^{R}(N_i+1)}{\alpha(p)(N_1+1)(\frac{1}{p}\prod_{i=2}^{R}(N_i+1)+\frac{p-1}{p})+\prod_{i=2}^{R}(N_i+1)+p-2} \tag{9}$$

When the effects of the pipeline delays are negligible, that is, when $\prod_{i=2}^{R}(N_i + 1) \gg p$, Equation (9) becomes:

$$S(p) = \frac{\alpha(p)(N_1 + 1)}{\frac{\alpha(p)}{p}(N_1 + 1) + 1} \tag{10}$$

Then the relative efficiency $e(p) = \frac{S(p)}{p}$ is:

$$e(p) = \frac{\alpha(p)(N_1 + 1)}{\alpha(p)(N_1 + 1) + p} \tag{11}$$

Hence when $\alpha(p)(N_1 + 1) \gg p$, we can obtain from the program a relative efficiency near to 1.

Equation (11) implies that one should arrange the classes such that class 1 has the largest population. This will ensure that $\alpha(p)(N_1 + 1)$ is as large as possible with respect to p. The

4 The time $K_i(p)$ also includes the time spent waiting in the communication operations.

5 This is true when we can neglect the effect of the cache. In our case study, this is true when $p \le 256$.

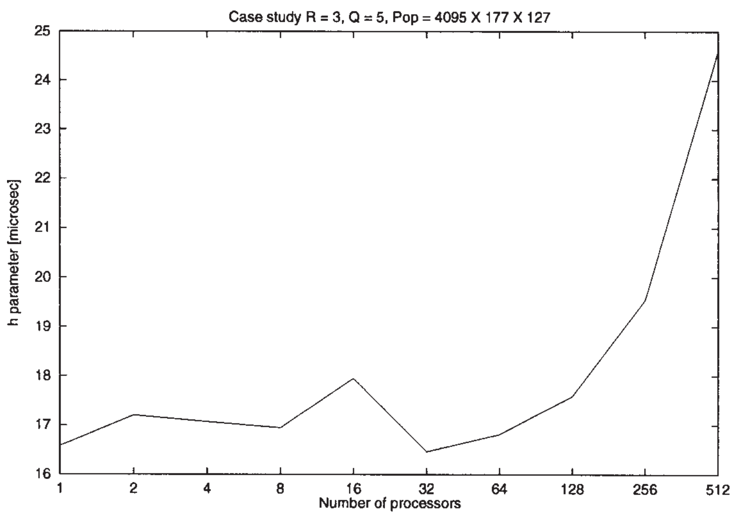

Figure 7. The *h* parameter of the Pipeline MVA algorithm on Cray T3D machine

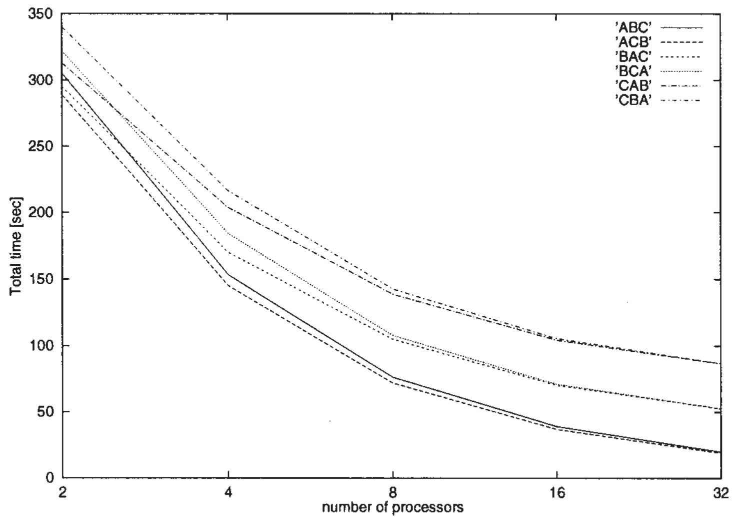

Figure 8. Execution time of the PMVA with different order of classes in the algorithm; $N_A = 4095$, $N_B = 127$, $N_C = 63$

effect of the ordering of the other classes will be minimal, although cache effects may give rise to small differences in performance.

Figure 8 shows the execution times when the same queueing network is analysed, but with the classes presented in a different order. For instance, the case ABC means that class A corresponds to the class with index $i = 1$ in the MVA algorithm, class B with index $i = 2$, and so on. We see that the two cases ABC and ACB have the best execution times, that is, when the class with the largest population has index $i = 1$.

6. Memory Overhead

The memory requirements of the program are dominated by the array of queue lengths. The amount of memory, expressed in bytes, allocated to a processor is given by:

$$Q(p)= \frac{D_f M}{p} \prod_{i=1}^{R-1} (N_i + 1) \qquad (12)$$

where D_f is the number of bytes of float type in the machine where the PMVA is running. This means that in some cases it could be necessary to increase the number of processors of the algorithm just to make possible the running of the program. Sometimes it could be useful to change the order of classes in order to minimize the the term $\prod_{i=1}^{R-1}(N_i + 1)$ expressed in Equation (12).

For instance, in the T3D implementation we have 512 CPUs each with 32Mb of memory available. Because $D_f = 8$, from Equation (12) we can say that if [6]:

$$M \prod_{i=1}^{R-1} (N_i + 1) > \frac{512 \times 32 \times 1024 \times 1024}{8} \approx 2.147 \times 10^9 \qquad (13)$$

the model cannot be solved with the PMVA running on the T3D.

7. Conclusion

We have demonstrated, both theoretically and experimentally, that a parallel implementation of the MVA algorithm based upon a simple partition of the population vectors to be evaluated produces a significant speedup, and is simple to implement and analyse. It is possible that other algorithms to partition nodes between processors might improve the performance of the algorithm a little. It appears from our experimental results that this would only be worthwhile when more than 64 processors are available.

When the problem to be solved satisfies the condition $\prod_{i=2}^{R}(N_i + 1) \gg p$, the pipeline imbalance is negligible, and when $\alpha(p)(N_1 + 1) \geq p$ we can obtain a relative efficiency $e(p) \geq 0.5$.

Further, we have shown that the cost of parallelisation with the pipeline algorithm is small. More complex processor allocation algorithms might obtain better performance, but at the cost of increased complexity. Our results indicate that for fewer than 64 processors, the simple allocation based on the subscript of the most numerous class is adequate.

8. Acknowledgements

Claudio Gennaro was supported by a grant from the EPCC TRACS scheme.

9. References

[1] Scherr, A.L. *An Analysis of Time-Shared Computer Systems*, MIT Press, Cambridge, MA, 1967.

[2] Baskett, F., Chandy, K.M., Muntz, R.R., and Palacios-Gomez, F. "Open, Closed and Mixed Networks of Queues with Different Classes of Customers." *Journal of the ACM*, Vol. 22, No. 2, pp 248-260, 1975.

[3] Jackson, J.R. "Jobshop-Like Queueing Systems." *Management Science*, Vol. 10, No. 1, pp 131-142, 1963.

[4] Buzen, J.P. "Computational Algorithms for Closed Queueing Networks with Exponential Servers." *Communications of the ACM*, Vol. 16, No. 9, pp 527-531, September 1973.

[5] Reiser, M., Lavenberg, S.S. "Mean Value Analysis of Closed Multichain Queueing Networks." *Journal of the ACM*, Vol. 27, No. 2, pp 313-322, 1980.

[6] Sauer, C.H. "Computational Algorithms for State-Dependent Queueing Networks." *ACM Transactions on Computer Systems*, Vol. 1, No. 1, pp 67-92, 1983.

[6] 8 is the size in bytes of the C type **double** on the T3D.

[7] Conway, A.E., Georganas, N.D. “RECAL—A New Efficient Algorithm for the Exact Analysis of Multiple-Chain Closed Queueing Networks.” *Journal of the ACM*, Vol. 33, No. 4, pp 768-791, 1986.

[8] Lam, S.S., Lien. Y.L “A Tree Convolution Algorithm for the Solution of Queueing Networks.” *Communications of the ACM*, Vol. 26, No. 3, pp 203-215, March 1983.

[9] Greenberg, A.G., McKenna, J. “Solution of Closed, Product Form, Queueing Networks via the RECAL and Tree-RECAL Methods on a Shared Memory Multiprocessor.” *Performance Evaluation Review*, Vol. 17, No. 1, pp 127-135, 1989.

[10] Pace, L., Tucci, S. “A Parallel Algorithm for Distributed Computer Performance Evaluation Environments.” In *Proceedings of the 1990 Summer Computer Simulation Conference*, pp 797-802, 1990.

[11] Greenberg, A.G., Mitrani, I. “Massively Parallel Algorithms for Network Partition Functions.” In *International Conference on Parallel Processing*, Chicago, January 1991.

[12] Hanson, F.B., Mei, J.-D., Tier, C., Xu, H. “PDAC: A Data Parallel Algorithm for the Performance Analysis of Closed Queueing Networks.” *Parallel Computing*, Vol. 19, No. 12, pp 1345-1358, 1993.

Claudio Gennaro received his degree in Electronic Engineering from the University of Pisa, Italy, in 1994. Presently he is a PhD Student in Computer Engineering at Politecnico di Milano, Italy. His current research interests include performance evaluation of computer systems, parallel and distributed systems and optimizing and parallelizing compiling techniques.

Peter King has a Mathematics Degree from University College, London, and MSc and PhD degrees in Computer Science from the University of Newcastle-upon-Tyne. He has been a lecturer in Computer Science at Heriot-Watt University in Edinburgh since 1982. His main research interests are the construction and evaluation of performance models of computer systems and communication protocols. He is the Chairman of the British Computer Society Performance Engineering Specialist Group.