Performance of Discrete Event Simulation on a Multiprocessor using Optimistic and Conservative Synchronization

Bruno R. Preiss

Department of Electrical and Computer Engineering
University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1

Copyright (c) 1990 by The Pennsylvania State University

Abstract

This paper presents an empirical study of the performance of various approaches to distributed discrete event simulation on a message-passing multiprocessor. This study compares the performance of distributed simulation to a sequential simulation of exactly the same system.

Four simulation benchmarks are presented. Each is a closed queueing network with a different topology. Four implementations of each benchmark are described — each with a different lookahead. Each benchmark was run under three customer loads. Each benchmark was run using one, two, four, and eight processors. This paper presents data showing the relationships between speedup, number of processors, lookahead, and computation load.

1. Introduction

This paper presents the results of an empirical study of the performance of various approaches to distributed discrete event simulation on an Inmos Transputer-based multiprocessor. In this study we compare the performance of the distributed simulations to a uniprocessor simulating exactly the same system using the ‘‘best’’ uniprocessor event sequencing algorithm. This study was done using the Yaddes simulation system[1-4]. Yaddes provides an environment in which the system to be simulated can be specified *independently of whether the simulation is done on a uniprocessor or a multiprocessor*. This means that exactly the same system is simulated on the uniprocessor and the multiprocessor. Furthermore, this means that the most appropriate algorithms for event sequencing can be applied to the uniprocessor and multiprocessor simulations leading to more meaningful speedup data. Specifically, the uniprocessor and multiprocessor algorithms need not be the same. (See also [5] and [6].)

2. The Yaddes Simulation System

The Yaddes system is a tool for constructing discrete event simulations. The user prepares a specification of the desired simulation. The Yaddes compiler translates the specification into a C language program. This program is then compiled and linked to a synchronization method library to form an executable program that performs the desired simulation.

The advantage of the Yaddes system is that it uses a programming model that supports several different simulation synchronization methods. In particular, Yaddes supports sequential simulation using an event list and three distributed simulation synchronization methods — distributed simulation using multiple event lists; conservative (Chandy-Misra-Bryant) distributed simulation[7]; and, optimistic (virtual-time-based) distributed simulation[8].

The programming model used in the Yaddes system is derived from Chandy-Misra-Bryant distributed discrete event simulation. The real-world system is modeled by a collection of physical processes (PPs) that periodically exchange information. This exchange of information occurs

This work was supported in part by the Information Technology Research Centre (ITRC) of the Province of Ontario (Canada) and by the Natural Sciences and Engineering Research Council (NSERC) of Canada under grant OGP36635.

Copyright (c) 1990 by The Pennsylvania State University

at discrete points in time. Every instant at which one process provides information to another is called an *event*.

The PPs of the real-world system are mimicked in the simulation by computer programs called *logical processes* (LPs). The exchange of information by the PPs is mimicked in the simulation by the exchange of messages by the LPs. Since the computer simulation does not execute in real time, each LP has its own notion of time and each message is timestamped with the time of the corresponding real-world event.

2.1. Event Combinations and Model Calls

Each Yaddes LP is driven by the occurrence of *event combinations*. An event combination is a collection of one or more input events *having the same timestamp*. An LP is invoked *once per event combination*. This activation of an LP is called a *model call*. In response to an event combination, an LP may change its state and produce zero or more output events. The Yaddes system ensures that event combinations are presented to each LP in increasing timestamp order. (N.B., *all* events with the same timestamp are presented to an LP in a single event combination.) Output events generated by an LP may be timestamped with any time greater than the timestamp of the input combination. In addition, output events must be produced in increasing timestamp order on each LP output.

3. Simulation Synchronization Methods

The fundamental issue in discrete event simulation is to process event combinations in a valid order. (In general, there are many possible sequences that are valid.) In the following sections, four synchronization methods for obtaining valid execution sequences are described.

3.1. Sequential Simulation

The sequential simulation synchronization method (EL) is implemented using the traditional discrete event simulation method. A single data structure, called the *event list* holds future events. The event list is sorted by time and by the ID number of the target LP. The basic execution cycle involves removing events from the head of the event list, forming event combinations, and causing the target LP to do the action associated with the given input event combination. When an action causes output events, those events are inserted into the event list.

The current implementation of the EL method uses a linear linked list to hold the future events. It is well known that this is not an optimal data structure for this application. To obtain uninflated speedup figures, a more efficient data structure such as calendar queues[9] should be used.

3.2. Distributed Simulation Using Multiple Event Lists

The multiple list synchronization method (ML) is a simple extension of the EL method to a multiprocessor. The EL method uses a serial algorithm in which event combinations are processed one at a time. However, it is not strictly required that the event combinations be executed one at a time. In particular, all the event combinations having the same time value as the first event in the list could be executed in parallel. The ML method attempts to exploit the parallelism at the head of the single event list by partitioning the event list. In ML, each processor has its own local event list. In addition, one processor has special status and acts as a

Copyright (c) 1990 by The Pennsylvania State University

global synchronizer. The synchronizer ensures that simulation time advances monotonically and that all processors are at the same point in simulation time.

3.3. Distributed Simulation Using Conservative Synchronization

In the conservative synchronization method (CM), each Yaddes LP runs as a separate task. Each Yaddes LP is managed by an *envelope*. The basic execution cycle begins when the envelope receives a message. The envelope buffers messages until an event combination can be formed. The decision regarding when an event combination can be formed is made purely on information available to the envelope. Specifically, an event combination with simulation time T can only be formed when the envelope has received an event message for each input of the LP having time T', such that $T' \geq T$. When an event combination is formed, the appropriate action is invoked. When an action causes an output event, a message is sent to the envelope of the appropriate LP.

The CM method has the potential for deadlock. That is, simulations that do not deadlock when using the other synchronization methods (EL, ML, and VT) may deadlock under CM. Although it is possible to implement deadlock detection and recovery methods, the current implementation of CM requires that the simulation programmer explicitly code the simulation in a way that prevents deadlock (if CM is to be used). This can be done by calling special library routines to send *null messages*. The meaning of such messages is ‘‘no event has occurred at time t,’’ where t is the timestamp of the message. This allows an envelope to determine that it has received all the events up to a given time t on a given LP input.

The routines provided to assist in the prevention of deadlock have no effect when the other synchronization methods are used. Hence, a simulation written for CM can be executed using the other synchronization methods *without change*.

3.4. Distributed Simulation Using Optimistic Synchronization

The optimistic synchronization method (VT) is based on the virtual-time concept used in the Time Warp Operating System[10]. As in the CM method, each Yaddes LP runs as a separate task and is managed by an envelope. However, the VT method differs from CM in that it uses an event synchronization method that does not deadlock.

Every envelope has a local clock. When an event message is received by an envelope there are three possibilities — its timestamp is either before, after, or equal to the local value of simulation time. If its timestamp is after the local time, an input event combination (consisting of the single event) is formed and the appropriate action is invoked. However, if the timestamp of the received event message is less than or equal to the local clock value, the envelope has already processed an event combination with time greater than or equal to the incoming event. The envelope must back up to the time of the incoming message. This backing-up is facilitated by an elaborate checkpointing procedure that allows earlier states to be recovered. An earlier state is restored, input event combinations are rescheduled, and output events are cancelled by sending *antimessages*. The envelope has buffers that save past inputs, past states, and antimessages.

Copyright (c) 1990 by The Pennsylvania State University

4. Queueing Network Benchmarks

The suite of benchmark programs used in this study was adapted from Nicol[11]. The suite consists of simulations of four different closed stochastic queueing networks.

Each queueing network contains a fixed number of nodes (64 in all cases). A node consists of a single queue and server. The network is populated with a fixed number of customers. A customer arrives at a node, waits for service, receives service, and then departs for another node.

The queueing discipline at each node is first-come, first-served (FCFS). The service time at each node is a biased, exponentially-distributed random variable with a minimum service time (T_{min}) of two time units and a mean service time (T_{mean}) of ten time units. Service is non-preemptive.

The routing of customers between nodes is random and uniformly distributed. Departures are equally distributed over all outputs and are independent of the input on which the customer arrived.

4.1. Topologies

The four simulation benchmarks are differentiated by the topology of the interconnection between the nodes. The topologies used in this study are Ring, Mesh, Multistage, and Hypercube.

The Ring topology contains 64 nodes. The ring is bidirectional.

The Mesh topology is a two-dimensional, 8×8 grid of nodes. Each node is connected to its four nearest neighbours. The connections between nodes are all bidirectional. The side and top edges of the grid are wrapped around to form a torus.

The Multistage topology consists of eight columns of eight processors. Each node in one column is connected to two nodes in the next column. The connections between columns are unidirectional. The topology of the interconnect between columns is a perfect, two-way shuffle.

The Hypercube topology is a six-dimensional Boolean cube. The connections between nodes in each dimension are bidirectional.

Each topology is characterized by the number of inputs and outputs at each node and by the size of the smallest loop in the network. The number of inputs that a node has affects the number of synchronizations that must be done, since event cominations must be formed and processed in order. The number of outputs that a node has affects the number of null messages that must be sent when using the CM synchronization method. The size of the smallest loop affects the rate of feedback which impacts on the performance of the synchronization method[1, 12].

4.2. Communication and Computation Loads

The number of customers in each queue at simulation time zero is a measure of the communication and computation load of the benchmark. In this paper, we present results for loads of one, four, and eight customers in each queue at time zero. This corresponds to a total of 64, 256, and 512 (respectively), customers in the system.

Copyright (c) 1990 by The Pennsylvania State University

5. Server Models

In this study, we compare the performance of four different implementations of the benchmark systems described above. The four implementations differ in the degree of lookahead used. Lookahead for an LP is the largest value, τ, such that if an LP knows all events up to and including some simulation time T, it can predict all actions that the LP will take up to time $T+\tau$[13, 14].

The degree to which an LP can predict depends on the system being simulated and on the implementation of the simulation. The simulation programmer can affect the latter by incorporating more ''knowledge'' into the implementation of the simulation[3, 15, 16].

5.1. Epsilon Lookahead

The smallest possible lookahead is zero. However, distributed simulation using conservative synchronization will immediately deadlock if all processes have zero lookahead. To avoid this deadlock, the epsilon lookahead model uses a lookahead equal to the smallest possible non-zero simulation time increment, ε. E.g., if a customer is to depart from an LP at simulation time T_{depart}, a message is sent from that LP to its successor with a timestamp T_{depart} when the LP's simulation time has reached $T_{depart}-\varepsilon$.

This model does not incorporate any knowledge about the queueing discipline (FCFS) or the non-preemptive service discipline. However, it is possible to predict the precise time when a customer will depart at the point when the customer arrives at a node. This can be done by sampling the random number generator and computing the service time for the customer when it is placed in the queue. However, the transmission of the customer bearing message is delayed until the LP's local time has advanced to $T_{depart}-\varepsilon$. To trigger a delayed event such as this, an LP merely sends a message to itself with timestamp $T_{depart}-\varepsilon$.

For every customer-bearing message sent from an LP to one of its successors, a null message with the same timestamp is sent to all the other successors of that LP.

5.2. Service-Time Lookahead

The service-time lookahead model takes advantage of the non-preemptive service. Once a customer has commenced service, the departure time for that customer is fixed. In the implementation of the service-time lookahead model, the time of departure, T_{depart}, is predicted when a customer arrives at a node. However, the transmission of the customer bearing message is delayed until that customer commences service.

When using conservative synchronization, null messages are sent to all the successors of an LP that do not receive a customer. If a customer departs from a node at time T_{depart}, then the earliest that another customer can depart is at time $T_{depart}+T_{min}$, where T_{min} is the minimum service time. A customer-bearing message is sent from an LP at time $T_{depart}-T_{service}$ with timestamp T_{depart} to one successor of that LP. At the same simulation time, null messages will be sent with timestamp $T_{depart}+T_{min}-\varepsilon$ to all the other successors of that LP.

Copyright (c) 1990 by The Pennsylvania State University

5.3. System-Time Lookahead

This lookahead model takes advantage of the queueing discipline (FCFS) and of the non-preemptive service. If a customer arrives at a node at time T, it will depart from the node at time $T_{depart}=T+T_{queueing}+T_{service}$, where $T_{queueing}$ is the queueing delay and $T_{service}$ is the service time for that customer. A customer-bearing message is sent from the LP at time T with timestamp T_{depart}.

When using conservative synchronization, null messages are used to predict the earliest possible arrival of a customer. If a customer departs from a node at time T, with timestamp T_{depart}, then the earliest that another customer can depart is at time $T_{depart}+T_{min}$, where T_{min} is the minimum service time. Thus, null messages are sent at time T with timestamp $T_{depart}+T_{min}-\varepsilon$ to all the successors of an LP that do not get a customer.

5.4. Presampling Lookahead

The presampling lookahead model takes advantage of the queueing discipline (FCFS), the non-preemptive service, the random routing, and the inability to distinguish between customers. This model presamples the random number generator to ‘‘predict’’ service times and routing for customers that have not yet arrived. This model is implemented using a futurelist[11].

The presampling lookahead model treats customer-bearing messages exactly the same as the system-time lookahead model. The presampling of the random number generator to predict service times and routing only affects the timestamps on the null messages.

In the presampling model, when a customer-bearing message is sent from an LP at time T with timestamp T_{depart} to a particular successor, null messages bearing timestamps $T_{depart}+T^{i}_{prediction}-\varepsilon$ are sent to all the successors of that LP. The prediction, $T^{i}_{prediction}$ is obtained by computing the earliest possible time at which a customer could arrive at the ith successor of an LP assuming an infinite number of customers arrived at that LP at time $T+\varepsilon$. Note that the prediction is different for each output of the LP.

The null messages are only sent if the prediction at time T differs from a prediction made at an earlier time. Note that each customer-bearing message is followed by a null message predicting the earliest possible arrival of the next customer.

6. Results

6.1. Experiments

All the results presented in this paper were measured on a network of Transputers. The network consists of eight 32-bit T414 Transputers connected in a three-dimensional hypercube. Each Transputer has 1 Mbyte of external RAM memory. The simulations were compiled using the Yaddes translator and the 3L Parallel C compiler (Rev. 2.0.1). The simulation programs run on the bare Transputer. A ninth 32-bit T800 Transputer with 2 Mbyte of external RAM memory serves as the interface between the network of Transputers and the host. (The interface Transputer does not participate directly in the execution of the simulation.) The host is an Intel-80286-based (DOS) coprocessor to an Apollo DN3010

Copyright (c) 1990 by The Pennsylvania State University

workstation (Motorola 68020 with 8 Mbyte RAM and a 70 Mbyte hard disk) running DOMAIN/OS.

This paper presents results derived from a total of 480 different simulations — 432 distributed simulations and 48 sequential simulations. For the distributed simulations, each of the four benchmark topologies (Ring, Mesh, Multistage, Hypercube), was simulated at three different initial loads (one, four, and eight customers initially in each queue), using each of the four lookahead models (epsilon, service time, system time, presampling), using each of the three distributed synchronization methods (ML, CM, and VT), on two, four, and eight processors (T414 Transputers).

For the sequential simulations, each of the four benchmark topologies (Ring, Mesh, Multistage, Hypercube), was simulated at three different initial loads (one, four, and eight customers initially in each queue), using each of the four lookahead models (epsilon, service time, system time, presampling), using a traditional event list on one processor (T414 Transputer).

Of these 480 simulations, 37 did not run to completion. The simulations that did not run to completion failed due to memory exhaustion. Typically, this occurred using presampling or service time lookahead with the VT synchronization method. This is because the current VT implementation does not provide any flow control.

In some cases the CM method also failed. Typically, CM failed due to memory exhaustion when using presampling lookahead on the hypercube topology. The large fan-out of the hypercube topology leads to a proliferation of null messages that exhaust the message buffer pool. The current CM implementation does not provide null message cancellation.

6.2. Verification of Correctness of Implementation

Each LP collects the following simulation statistics: (1) mean and standard deviation of service times, (2) mean and standard deviation of the time a customer spends in the queue, (3) mean and standard deviation of the length of the queue, (4) mean and standard deviation of the number of the output from which a customer leaves a node. For a given topology and load, these data should be the same for all simulations regardless of the synchronization technique, lookahead model, or number of processors used. In all the cases reported herein, *without exception*, the simulation statistics were identical (for the simulations that ran to completion).

6.3. Speedup vs. Number of Processors

Let $T_n(s,t,l,m)$ represent the execution time for a given synchronization method (s), topology (t), load (l), and lookahead model (m) on n processors. Then speedup is given by

$$S_n(s,t,l,m)=\frac{T_1(EL,t,l,m)}{T_n(s,t,l,m)}.$$

Note, speedup is measured against a single processor using the EL synchronization method (i.e., the ‘‘best’’ uniprocessor algorithm for discrete event simulation). The special case, $S_n(s,t,l,m)=n$ is called *linear* speedup. Figures 1, 2, and 3 show the measured speedup for the ML, CM, and VT synchronization methods (respectively).

Copyright (c) 1990 by The Pennsylvania State University

Figure 1 shows that, regardless of topology, load, or lookahead model, the ML synchronization method never achieves speedup greater than two. Furthermore, in some cases ML on $n \geq 2$ processors sometimes runs more slowly than EL. This data indicates that for the given set of benchmarks, the intrinsic parallelism is less than two. Thus, a synchronization method that keeps all of the processors at the same value of simulation time will not show good speedup.

Figure 2 shows the speedup data for the CM synchronization method. This figure clearly shows that near-linear speedup can be achieved for certain topologies, loads, and lookahead models. (In fact, the curves show a super-linear speedup in one case. This can be attributed to the inflation of speedup due to the less than optimal implementation of EL.) Figure 2 shows that the best speedup can be achieved under heavy loads (four and eight) when using the system-time and presampling lookahead models. Because there are more customers in the system under a heavy load, it is less likely that a processor will become idle. Furthermore, the system-time and presampling lookahead models are more predictive, and consequently, fewer null messages need to be processed.

Figure 2 also shows that the CM synchronization method is totally inappropriate for certain loads and lookahead methods. This is because the added computation needed to process null messages outweighs any possible benefits of distributed computation. Figure 2 also shows that as the fan-out of the topology increases, the magnitude of the speedup decreases. This is again due to the added null message processing.

Figure 3 shows the speedup data for the VT synchronization method. This figure clearly shows that near linear speedup can be achieved for certain topologies, loads, and lookahead models. In contrast with the CM method, Figure 3 shows that the VT is able to achieve speedup in most cases on $n \geq 4$ processors for all loads, topologies, and lookahead models. This is mainly due to the absence of null message processing in the VT method. A close comparison of Figures 2 and 3 also shows that, while VT is able to achieve speedup in almost all cases, CM achieves better speedup under heavy loads using system-time and presampling lookahead and for small fan-out. In these cases, it appears that the erroneous computation done by the optimistic VT algorithm impacts execution time more adversely than the null message processing of the conservative CM algorithm.

7. Conclusions

This paper presents the results of an empirical study of the performance of distributed simulation on a Transputer-based multiprocessor. A suite of four different closed stochastic queueing network benchmarks has been studied. Four different implementations of the simulation models have been examined — each with a different degree of lookahead. Each benchmark has been run on a uniprocessor using the traditional sequential algorithm for discrete event simulation and on a multiprocessor with two, four, and eight processors using three different distributed simulation synchronization methods. Each benchmark was run under three different customer loads.

The results of these simulations show that distributed simulation using multiple event lists will not give useful speedup for these benchmarks. The two synchronization methods that do give good speedup, CM and VT, behave differently with respect to network topology, lookahead

Copyright (c) 1990 by The Pennsylvania State University

model, and load. CM gives the best speedups for low-fan-out topologies using large lookahead (system time and presampling) and under heavy customer loads. Furthermore, the null message processing overheads lead to fractional speedup (slowdown) in many cases. On the other hand VT is able to achieve good speedups for almost all cases with $n \geq 4$ processors. However, VT does not match the performance of CM in thoses cases where CM does extremely well.

8. References

1. Preiss, B. R., Loucks, W. M., and Hamacher, V. C., ''A Unified Modeling Methodology for Performance Evaluation of Distributed Discrete Event Simulation Mechanisms,'' *Proc. 1988 Winter Simulation Conf.*, pp. 315-324, December 1988.
2. Preiss, B. R., ''The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments,'' *Proc. SCS Multiconf. — Distributed Simulation*, Vol. 21, No. 2, pp. 139-144, March 1989.
3. Loucks, W. M. and Preiss, B. R., ''The Role of Knowledge in Distributed Simulation,'' *Proc. SCS Multiconf. — Distributed Simulation*, pp. 9-16, January 1990.
4. Preiss, B. R. and Loucks, W. M., ''Prediction and Lookahead in Distributed Simulation,'' CCNG Technical Report E-191, Computer Communications Networks Group, University of Waterloo, Waterloo, Ontario, CANADA, 1989.
5. Reed, D. A., Malony, A. D., and McCredie, B. D., ''Parallel Discrete Event Simulation Using Shared Memory,'' *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, pp. 541-553, April 1988.
6. Fujimoto, R. M., ''Time Warp on a Shared Memory Multiprocessor,'' *Proc. Int. Conf. on Parallel Processing*, Vol. III, pp. 242-249, August 1989.
7. Misra, J., ''Distributed Discrete-Event Simulation,'' *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39-66, March 1986.
8. Jefferson, D. R., ''Virtual Time,'' *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, July 1985.
9. Brown, R., ''Calendar Queues: A Fast O(1) Priority Queue Implementation of the Simulation Event Set Problem,'' *Communications of the ACM*, Vol. 31, No. 10, pp. 1220-1227, October 1988.
10. Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S., ''Distributed Simulation and the Time Warp Operating System,'' *Proc. 12th SIGOPS — Symposium on Operating Systems Principles*, pp. 77-93, 1987.
11. Nicol, D. M., ''High Performance Parallelized Discrete Event Simulation of Stochastic Queueing Networks,'' *Proc. 1988 Winter Simulation Conf.*, pp. 306-314, December 1988.
12. Leung, E., Cleary, J., Lomow, G., Baezner, D., and Unger, B., ''The Effects of Feedback on the Performance of Conservative Algorithms,'' *Proc. SCS Multiconf. — Distributed Simulation*, Vol. 21,

Copyright (c) 1990 by The Pennsylvania State University

No. 2, pp. 44-49, March 1989.

13. Fujimoto, R. M., ‘‘Performance Measurements of Distributed Simulation Strategies,’’ *Trans. Society for Computer Simulation*, Vol. 6, No. 2, pp. 89-132, April 1989.

14. Wagner, D. B. and Lazowska, E. D., ‘‘Parallel Simulation of Queueing Networks: Limitations and Potentials,’’ *Performance Evaluation Review*, Vol. 17, No. 1, pp. 146-155, May 1989.

15. Chandy, K. M. and Misra, J., ‘‘Conditional Knowledge as a Basis for Distributed Simulation,’’ Technical Report 5251:TR:87, Computer Science Department, California Institute of Technology, Pasadena, California, 1987.

16. Baezner, D., Cleary, J., Lomow, G., and Unger, B. W., ‘‘Algorithmic Optimizations on Time Warp,’’ *Proc. SCS Multiconf. — Distributed Simulation*, Vol. 21, No. 2, pp. 73-78, March 1989.

Copyright (c) 1990 by The Pennsylvania State University

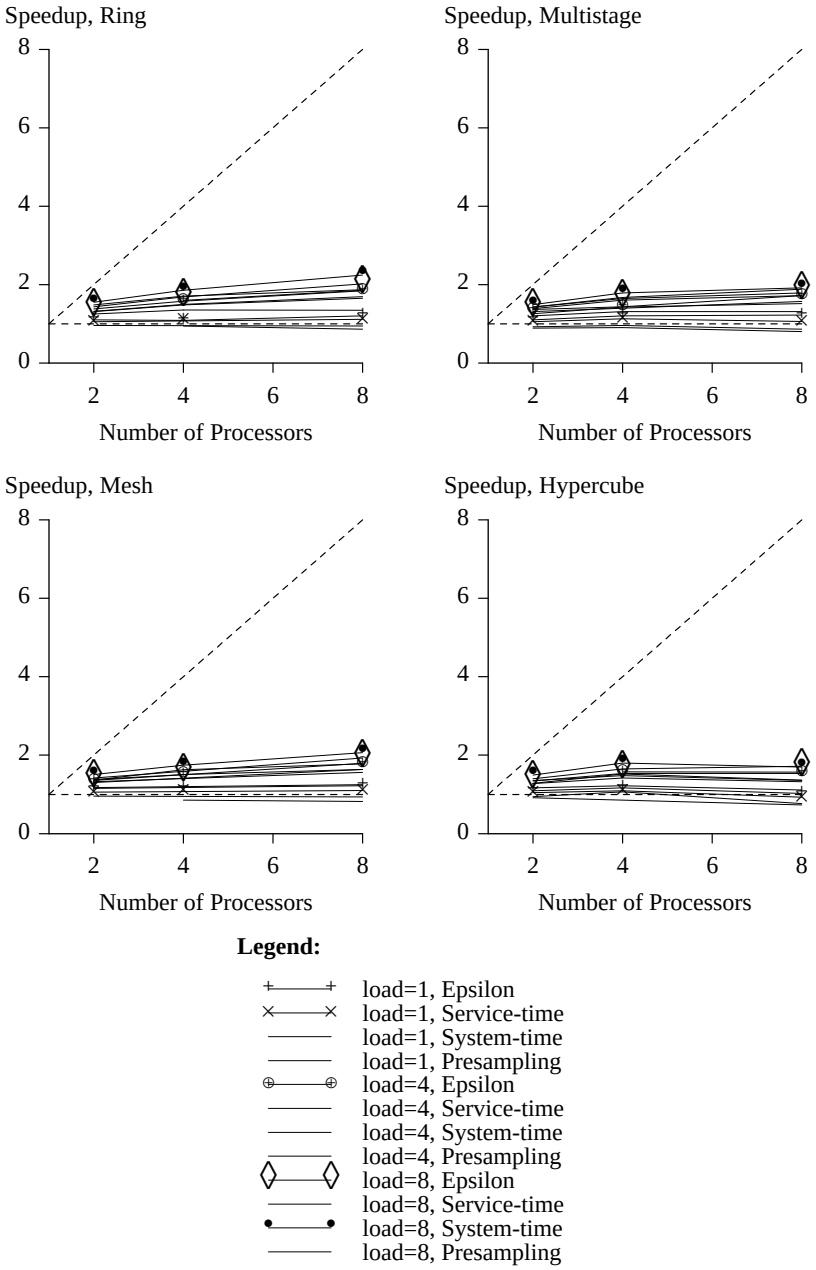

Figure 1. Speedup vs. Number of Processors, ML.

Copyright (c) 1990 by The Pennsylvania State University

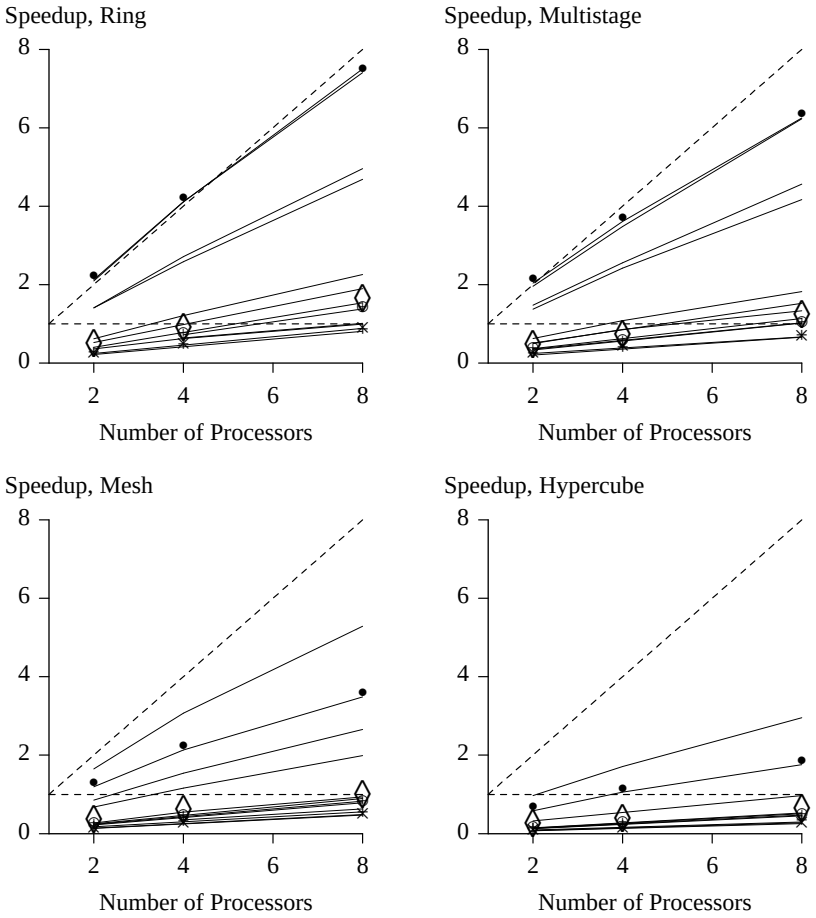

Figure 2. Speedup vs. Number of Processors, CM.

Figure 3. Speedup vs. Number of Processors, VT.

Copyright (c) 1990 by The Pennsylvania State University