Appendix B: Conservative Parallel Simulation of ATM Networks

Department of Computer Science, University of Waikato, New Zealand.
email: {jcleary,jjtsai}@cs.waikato.ac.nz

Abstract

A new conservative algorithm for both parallel and sequential simulation of networks is described. The technique is motivated by the construction of a high performance simulator for ATM networks. It permits very fast execution of models of ATM systems, both sequentially and in parallel. A simple analysis of the performance of the system is made. Initial performance results from parallel and sequential implementations are presented and compared with comparable results from an optimistic TimeWarp based simulator. It is shown that the conservative simulator performs well when the "density" of messages in the simulated system is high, a condition which is likely to hold in many interesting ATM scenarios.

Introduction

ATM is a recent standard for broadband communications [3,7]. It is a rate independent system which uses fixed size (53 byte) *cells* for all transmission. The motivation of this work is the need to simulate large numbers of cells in ATM systems. Such simulations, which need to explicitly model the passage of cells through the fabric of switches and over the physical communications links, are particularly demanding for a number of reasons. There are indications that it is necessary to simulate in the order of 10^9 to 10^{12} cells in one run to measure some effects such as cell loss rates. Also the models of ATM at the individual cell level have very low compute grains. For example ATM-TN a detailed ATM model produced by the Telesim project [15] has average per simulation event granularities of 17 μseconds on a Sparc-2 55MHz CPU [11]. These granularities are close to the per event overheads that can be expected from sequential simulators. For example, the sequential simulator used to obtain the granularities has a measured per event overhead in the range of 11 to 18 microseconds (depending on the size of the event list).

The need for large numbers of high fidelity simulation events motivates the requirements for a high performance simulator. The low granularities imply that any simulator must be very efficient with per event overheads close to those of the best sequential simulators. Results reported later will show that expected simulation scenarios have significant available parallelism. Thus there is a strong motivation to construct parallel simulators to take advantage of this. The difficulty here is to keep the overheads low enough that the parallelism can be effectively used. To this end the conservative simulator is designed for a shared memory multiprocessor environment. This follows the lead of recent TimeWarp parallel simulators [8].

Any such simulation splits into two main parts: the models of the ATM switches, access points and links; and the models of the traffic generators. This paper is concerned mainly with the interior ATM objects and not so much with the traffic generators. There are two reasons for this: the ATM part is by far the most compute intensive; and it is more stereotyped and consistent than the generators.

ATM-TN is a generic cell-level ATM model [15]. The model deals with individual cells and passage through switches. As well it includes high-level models of TCP/IP, World-Wide-Web traffic, and self-similar traffic [1,2]. The current work is based around this model.

The proposed algorithm belongs to the class of conservative simulations. These, by definition, block until a process can ensure that it will not violate causality by processing the next event [10]. Nil messages, as proposed by Chandy and Misra [5,13], provide a method for communicating processes to exchange information regarding the lower bound on the time stamps of future messages. The effectiveness of nil messages depends greatly on the amount of lookahead available [9]. This is apparently application dependent [14,16]. In any case, the possible imposition of extra overhead by the use of nil messages has caused much criticism of the usefulness of such approaches.

Several mechanisms based on nil messages have been developed since it was first introduced. Chandy and Sherman [6] present an alternative. Instead of nil messages, a conditional message is sent to the receiving process only when the sender will otherwise become blocked. As long as there are (real) messages waiting in the receiver's input buffer, conditional events are not used. Also, Jha and Bagrodia [12] suggested a scheme combining the above approach and conventional nil messages. Cai and Turner [4] proposed a "carrier null message" approach. In this approach, an additional field is included to the nil message so that a process can identify a nil message initiated by itself. When such a message is detected, the blocking is ended. The effectiveness of this approach depends on how much earlier such a carrier can be detected compared to regular nil messages. Although performance speedup can be observed in certain cases with these approaches, the applicability of these to large complicated communication networks (eg ATM) remains unknown.

In the next section the simulation algorithm itself is described, together with both sequential and parallel implementations. Section 3 focuses on a key data structure and its optimisation. Section 4 considers the theoretical performance of the conservative simulator and compares this with event based simulators including TimeWarp. As well actual performance results from a toy problem on a distributed version of the conservative simulator are reported together with comparable results from a TimeWarp based simulator. Initial results from a full port of ATM-TN are also provided. The paper concludes with a summary.

Simulator

In the simulator all *processes* receive (0 or more) *input links* and generate (0 or more) *output links*. *Messages* are sent down links between processes. Each message has a *send* and *receive* time. The critical part of the proposed algorithm is that each time a process is executed it merges all its input lists and executes one *event* for each incoming message, this continues until one of the input links is empty where-upon the process *suspends*. Because the merge is done in time order the process suspends on the earliest empty link, the earliest time that a message can be received down that link is the process's *current time* (CT).

The generic algorithm is:

```
while there are messages in system
   select next process to execute;
   while lowest time stamped
    incoming link has a message
    select lowest message;
    process message;
   end while;
   set current time of process to
    minimum time on any in-link;
   update last time on all outgoing
    links to minimum possible
    receive time of next message
(will be ≥ current time +
lookahead of link);
   suspend process;
end while;
```

This algorithm is different from many event driven simulators, in that, when a process is selected for execution it consumes all of its incoming messages before suspending. That is, scheduling is not done on an event by event basis. In the worst case no event will be executed when a process is selected, although the current time of the process may be advanced which will cause the time on the output links to also advance. Such an "empty" execution corresponds roughly to a nil message in a Chandy-Misra conservative simulator [5] and to the conditional messages used in [12]. Many algorithms are possible for selecting the next process to be executed. The aim is to minimise the cost of scheduling both by minimising the number of suspension / scheduling steps and by minimising the cost of suspension and scheduling.

It is assumed that the links are monotonic - that is messages are received in the order of their receive times which is the same order that they are sent (physically this can be interpreted as "messages cannot pass each other in the links"). Of concern is the *lookahead* of each link which is the minimum difference between the receive and send time of a message. This must be positive (it may be 0 in some cases), and in many interesting cases will be non-zero.

Consider a simulation model as a directed graph where the links are arcs and the processes are nodes. The arcs are labelled with the lookahead of the link. For any loop (sequence of nodes and arcs that arrives back where it starts) the *loop time* (LT) is the sum of the lookaheads along the loop. The *Minimum Lookahead Time* (MLT) for a link or process is the minimum LT of any loop which passes through the link or process. I will assume that no process (and by implication link) has a zero MLT. In particular all ATM models will be seen to have non-zero MLTs.

The algorithm above can be executed by randomly selecting a process, executing it, and then selecting the next process and so on. So long as this process is fair - that is every process is eventually selected for execution - then it is easily shown that the simulator will make progress (the GVT or minimum time of any unprocessed message in the system will increase). The problem is to minimise the overheads of suspension, that is, the aim is to maximise the number of events executed and minimise the number of suspensions.

Sequential Execution

The first (sequential) algorithm always schedules (one of) the processes with the smallest CT. In such a simulator the best that any process can do is to advance its simulation time by its MLT on each execution/suspension cycle[1]. If the MLT of the i'th process is given by MLT_i then a lower bound for the average number of suspensions per simulation time unit, C, will be:

$$C \geq \sum_i \frac{1}{MLT_i} \qquad (2.1)$$

The process of scheduling on the basis of the CT is susceptible to many optimisations. Given that the processes can be scheduled at random, clearly scheduling on the basis of the CTs can be sloppy and still the algorithm will work. Note that selecting for execution on the basis of CT is different from using an event list (in particular, the times used for scheduling may correspond to no actual events in the system and the number of entries in the scheduling list is always equal to the number of processes in the system).

Static Schedules

Another possible algorithm is to have a loop in which each process is executed exactly once. This will clearly be sub optimal if the processes have differing MLTs - those with large MLTs will be executed too often. However, it seems that static schedules in which a process appears in inverse proportion to its MLT can approach close to the lower bound. The example systems in Figure 1 have close to optimum static schedules. In Fig 1a the MLTs for the three processes are: A=1, B=2, C=3. From formula (2.1) the lower bound for the number of suspensions per unit of simulation time is C=1.83.. Using either of the static schedules below achieves C=2..:

C B A A C B A A ..

or

C B A A B A C B A A B A ..

For Fig 1b the MLTs are: A=1, B=3, C=3 and the optimum of C=1.67.. is actually attained by

C B A A A C B A A A ...

Clearly the overheads of such static schedules can be made very small. It is not necessary to maintain an event list or any dynamic data structure. For small numbers of processes they could

---

[1]Consider the loop passing through the process that has the minimum LT. Assume that all the incoming links along the loop other than the one under consideration have minimum receive times that are infinite (or sufficiently far in the future that they do not contribute to the minimum). Then the minimum receive time on the incoming link will be equal to the sum of the lookaheads around the loop, that is, the MLT. It is this time that will limit the advance of the process when it is executed.

be directly compiled as an unrolled sequence of procedure calls.

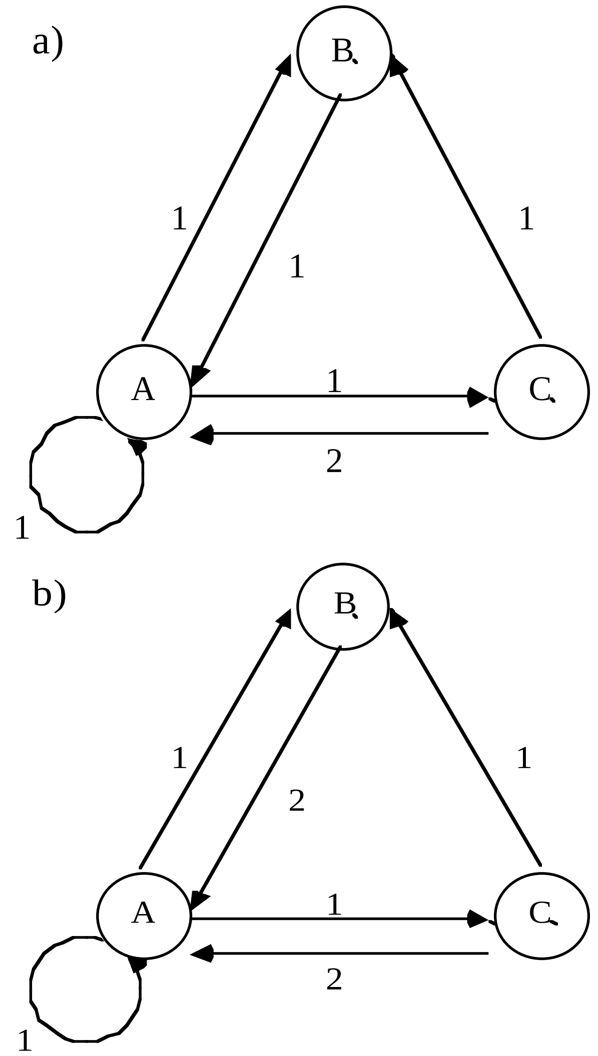

Figure 1. Example Process Graphs.

Parallel Execution

Consider a distributed system where the processes are each mapped to a processor. A static schedule can be followed locally on each processor. In the limit when each process is mapped to one processor the static schedule will be the repeated execution of the same process. In this case the process is effectively polling its input links.

For ATM systems this parallel case (with all processes mapped to their own processor) may be **more** efficient than the sequential one. The critical process, that is the one that on average runs the slowest may never suspend, as the other (lightly loaded) processes will run faster than the critical one and so the critical process will always have messages on all its input links.

Thus the critical part of the computation will be essentially free from any suspension overheads, it will spend all its time executing in the user modelling code.

Link Data-structures

A key data-structure in any simulator based on these ideas is the one that holds the messages in the links. It should allow fast insertion and extraction of events and also be able to be shared between processes in a parallel system. The sharing is made easier by the fact that only two processes ever access the link - the sender and the receiver. The rest of this section consists of a series of refinements of an initial simple data structure. The refinements are designed to give greater efficiency and to provide effects such as bounding the time advance of the sending process. It is notable that the resulting data structure does not need any locking to work correctly, thus avoiding one important source of overheads.

Linked-list

Figure 2 gives a description and pseudo-code for a simple version of the link data structure. It uses single unidirectional pointers between the messages in the link together with two external pointers into the link: Top and Bottom.

This code works correctly even in the case where the sender and receiver are running in parallel. In this case `Top` is modified only by the sender and is read by the receiver. `Bottom` is used only by the receiver. To ensure correct parallel execution the order of assignments in `send` is important, in particular `Top` must be assigned after all the other fields have been set up. As well storing a pointer or a time-stamp must be an atomic operation.

Optimized

The first optimisation is to make the length of the list in the link constant. This achieves two things: it removes the overheads of allocating and deallocating links and it also allows the length of the list to be used as a flow control mechanism. For example, a process with no input links (a message generator) will have to suspend if its output links become full. Fixing the length allows the `next` fields to be initialised so that the list forms a circular loop and thus `next` need never be updated. There is one major change needed to the code for `send` to ensure that `Top` does not over-run `Bottom` (that is that too many messages are not inserted onto the link). This also means that the sender must be prepared to deal with the `send` function returning an indicator that the message cannot be sent - in this case the sender must suspend. The other code change is to `initialise` which now creates a fixed loop of links. This code works correctly in parallel.

The final optimisation is to put all the links adjacent to one another in a single block of memory. The main advantage of this is that it minimises cache misses when accessing the messages. The only code that needs to be changed is the initialise function (this is left as an exercise for the reader).

Two further refinements have been added to the implementation. First, links from an LP back to itself are treated specially - it is not necessary to delay on them when they are empty. Second, LPs forming a cycle of small LT can be clustered into a single LP. This optimisation has great impact on ATM-TN because a switch in ATM-TN is modeled by a number of LPs, which exchange control signals with small lookaheads forming low MLT loops.

Performance

Comparison With Event Driven Simulators

It is possible to make a simple comparison between the amount of computing expected in the type of conservative system outlined above and other event driven simulators such as TimeWarp. Let E be the maximum number of events possible in one (simulated) time unit, χ the cost of doing one suspension, ε the cost of executing one event and λ be a parameter that expresses the fraction of the maximum possible number of messages that are actually sent. Then the total cost of advancing one simulated time unit in the conservative scheme is given by the expression:

$t_c = \lambda\varepsilon E + \chi C$

For an event driven system the cost is proportional only to the number of events:

$t_0 = \lambda\varepsilon E$

Clearly as λ decreases the event driven system will eventually win out over the conservative as the costs of an event driven system are proportional to the number of events. The per event overheads of the conservative simulator will be inversely proportional to λ - an effect which is clearly seen in the empirical results below. This analysis is generous to real parallel event driven systems such as TimeWarp where the per event overheads will be higher than the conservative system because of the need to do state saving and rollback.

Empirical Results

The conservative simulator was implemented on a SPARC-1000 platform with 8 55MHZ Sparc processors. An initial series of experiments were conducted to collect performance measurements from the proposed mechanism in order to compare against a standard event list based sequential simulator, as well as a parallel simulator based on Time Warp. The benchmark used was a 4-dimensional hypercube communication network. Upon receiving a message, a node forwards the message, with an exponential delay with mean 1.2, to a neighbouring node. The selection of a destination node is random. The transit time on each link is set constant to 1.2 time units. As in ATM systems it is assumed that a message prevents another message being sent for one time unit. The experiments use various message populations, ranging from 1 to 1000. Thus the number of pending events per process will vary from 1/16 to 64.

The event granularity for the basic sequential simulation varied from 11 µsecs to 18 µsecs (for a large event list). It is difficult to separate the user code and the simulator overheads but the user code takes about 4 µsecs (mainly for random number generation). A spin-loop is also employed in some cases to investigate the impact of larger event grains.

The average execution time per event versus the message population is shown for a single processor in Fig. 4a. Following the theoretical predictions above, the execution time per event for the conservative simulator increases as the message popu lation decreases. The execution time for the sequential and TimeWarp simulators show an opposite trend increasing slightly as the message population increases - this is probably caused mainly by an increase in the size of the event list (the sequential simulator uses a splay tree and the TimeWarp simulator a calendar queue).

The other three subgraphs of Fig 4 show the overhead on 2, 4 and 8 processors. The performance of the conservative simulator improves smoothly in the parallel execution. Interestingly the TimeWarp simulator has difficulties at low message populations on the parallel runs. There has not been a chance to investigate this more closely but it may indicate the onset of some form of dynamic instability.

The results from the proposed mechanism are encouraging. Even at very low granularities the conservative mechanism shows speedup over a significant range of message populations. Preliminary estimates indicate that average message populations in ATM simulations will be well above the cross-over where the conservative system performs better than TimeWarp.

Fig. 5a shows the absolute speedup of the conservative simulator and Time Warp compared to the sequential simulator without any added granularity. Four different message populations, namely, 20, 100, 200 and 1000, were used. For large message populations, the proposed mechanism shows a speedup of 3.96 when running on 8 processors. Even with moderate message population, eg 100-200, the speedup observed on 8 processors is 2.82. Interestingly, the single processor version of the distributed conservative algorithm shows a slow down of about 37% with a message population of 1000. In principle it should run at about the same speed as the sequential simulator (indeed a sequential implementation of the conservative simulator runs slightly faster than the standard sequential simulator for large message populations). The problem seems to lie in the implementation of the threads package on the Solaris operating system. This requires a system call to access memory which is local to a thread. The structure of the C++ system we are using forces such a call on every send. This seems to account for the slow down (investigations are proceeding on how to avoid this overhead).

In Fig. 5b-d, the same set of experiments are repeated, but with different event granularities. Each figure shows the speedup when a spin-loop of granularity around 10 µS, 100 µS, 1000 µS is added to each event (in addition to the original computation of the application). As seen from the figures, when the granularity increases, the speedup increases. Even with a small increase in event granularity (10µS), the speedup is elevated to about 4.91 on 8 processors when message population is high. Recent timing results on an implementation of the ATM-TN model indicates that the average event grains are about 17µS slightly higher than the grain used in Fig 5b. Time Warp also improves its performance drastically on larger event grains, leading to smaller differences between the two techniques. The results here are kind to TimeWarp in that the added compute grains include no provision for any additional state saving overhead.

ATM-TN Port

The full ATM-TN model has been ported to the a parallel version of the new simulator with encouraging preliminary results. A benchmark consisting of a per-port switch connecting three endnodes has been used to test the system. Two of the endnodes have ethernet traffic source / sink modules running. The switch has three ports, at 45 Mbps each. The collected data in Table 1 shows that when the load on ethernet is low (λ=0.045), both the sequential simulator and the Time Warp simulator outperformed the conservative algorithm. However, as the ethernet load increased (λ=0.11 and 0.18), the execution time per event for the conservative simulator decreases greatly while the Time Warp based simulator and sequential simulators remain the same. The real strengh of the conservative simulator appears on executing on multiple processors. In one experiment, the speedup (vs. sequential simulator) on 4 processors is close to 2.5. Further experiments are planned for more realistic ATM networks. Table 1 shows the execution time per event (in µS) for the different simulators.

Summary

A new shared memory based conservative simulator has been proposed and implemented. It is capable of exhibiting high efficiencies with overheads on a parallel implementation less than twice that for an optimised sequential simulator. It is well suited to its proposed domain of application in ATM models with its high efficiency overcoming problems of very low compute grain sizes. The parallel simulator compares well with a TimeWarp simulator over a wide range of parameters.

λ	con	con (4)	TW	TW (4)	Seq
0.045	32.02	15.69	40.16	27.22	25.37
0.110	26.53	12.01	42.40	27.06	25.52
0.180	24.74	11.80	42.15	26.97	28.07

Table 1. Run Times for ATM-TN

The conservative algorithm relies on a number of features of ATM networks to achieve good performance:

- all nodes (switches) have a small number of incoming and outgoing links;
- the capacity of links is limited by the (fixed) length of cells (this ensures a minimum lookahead for links);
- most “interesting” simulations will have links loaded close to capacity.

Clearly the algorithm is not universal as there will be problems where other mechanisms including TimeWarp will outperform it, however, it is well suited to ATM models and similar communications systems which are demanding and economically important.

Acknowledgments

This work was supported by grant 95-UOW-S14-4321 from the New Zealand Public Good Science Fund and by Science Applications International Corp. We would like to thank Xiao Zhonge and the rest of Telesim team for providing us with both sequential

and parallel simulators and the ATM-TN model.

References

1. Arlitt, M., and Williamson, C.(1995a) *"A Synthetic Workload Model for Internet Mosaic Traffic,"* Proc. Summer Computer Simulation Conf., Ottawa, June.

2. Arlitt, M., Chen, Y., Gurski, R., and Williamson, C.(1995b) *"Traffic Modelling in the ATM-TN Telesim Project,"* Proc. Summer Computer Simulation Conf., Ottawa, June.

3. Boudec, J.L.(1992) *"The Asynchronous Transfer Mode: a Tutorial,"* Computer Networks and ISDN Systems, pp. 279-309.

4. Cai, W., and Turner, S.J.(1990) *"An Algorithm for Distributed Discrete Event Simulation,"* Proc. Distributed Simulation Conference, San Diego California, pp. 3-8, JanuaryJanuary.

5. Chandy, K.M., and Misra, J.(1979) *"Distributed Simulation: a case study in design and verification of distributed programs,"* IEEE Trans.Software Eng., **5**(5), pp. 440-452, September.

6. Chandy, K.M., and Sherman, R.(1989) *"The conditional event approach to distributed simulation,"* Proc. Distributed Simulation Conference, San Diego, California, pp. 93-99, March.

7. Comm. ACM.(1995) *"Special Edition on Issues and Challenges in ATM Networks,"* Comm. A.C.M., February.

8. Das, S., Fujimoto, R., Panesar, K., Allison, D., and Hybinette, M.(1994) *"A Time Warp System for Shared Memory Multiprocessors,"* Winter Simulation Conference, December.

9. Fujimoto, R.M.(1988) *"Performance measurements of distributed simulation strategies,"* Proc. Distributed Simulation Conference, San Diego, California, pp. 14-20, February.

10. Fujimoto, R.M.(1990) *"Parallel Discrete Event Simulation,"* Comm. A.C.M., **33**(10), pp. 30-53, October.

11. Jade Simulations International Corp.(1995) *"Deliverable for ATM-TN Performance Project,"* Science Applications Internationl Corp., August.

12. Jha, V., and Bagrodia, R.L.(1993) *"Transparent implementation of conservative algorithms in parallel simulation languages,"* Winter Simulation Conference, Los Angeles, pp. 677-686, December.

13. Misra, J.(1986) *"Distributed Discrete-Event Simulation,"* ACM Computing Surveys, **18**(1), pp. 39-65.

14. Nicol, D.M.(1988) *"Parallel discrete-event simulation of FCFS stochastic queuing networks,"* ACM SIGPLAN Notices, **23**(9), pp. 124-137.

15. Unger, B.W., Gomes, F., Zhonge, X., Gburzynski, P., Ono-Tesfaye, T., Ramaswamy, S., Williamson, C., and Covington, A.(1995) *"A High Fidelity ATM Traffic and Network Simulator,"* Winter Simulation Conference, Washington, D.C., December.

16. Wagner, D., and Lazowska, E.(1989) *"Parallel simulation of queuing networks: limitations and potentials,"* Proc. International Conf. on Measurement and Control, pp. 146-155.

```
Top: pointer to next location to have a message stored;
Bottom: pointer to next message to be received;
structure  link:
       time: receive time of message (or earliest time of next
              message if pointed at by Top);
       next: pointer to next message (undefined if
               pointed at by Top);
       data: user defined content of message (undefined if
               pointed at by Top);
end structure;
initialise: %Initialise link.
       t:= new link;
       t.time:=0;
       t.next := nil;
       Top:=t;
       Bottom:=t;
end initialise;
send(receive_time,next_time,msg): %Send a message.
next_time is the earliest possible receive time of the next message;
       Top.data:=msg;
       Top.time:= receive_time;
       t:= new link;
       t.time:= next_time;
       Top.next := t;
       Top:= t;
       send := true;
end send;
early_time(time): %Update earliest possible receive time:
       Top.time:= time;
%Receive a message - returns true if Bottom is left pointing at a valid message - in
any %case Bottom.time is left as the earliest time a message can arrive down the link;
receive:boolean;
       if Bottom = Top then
              receive:=false;
       else
              t:=Bottom;
              Bottom:=Bottom.next;
              deallocate t;
              receive:= boolean:(Bottom<>Top);
       end if;
```

Figure 2. Code for Link Datastructure.

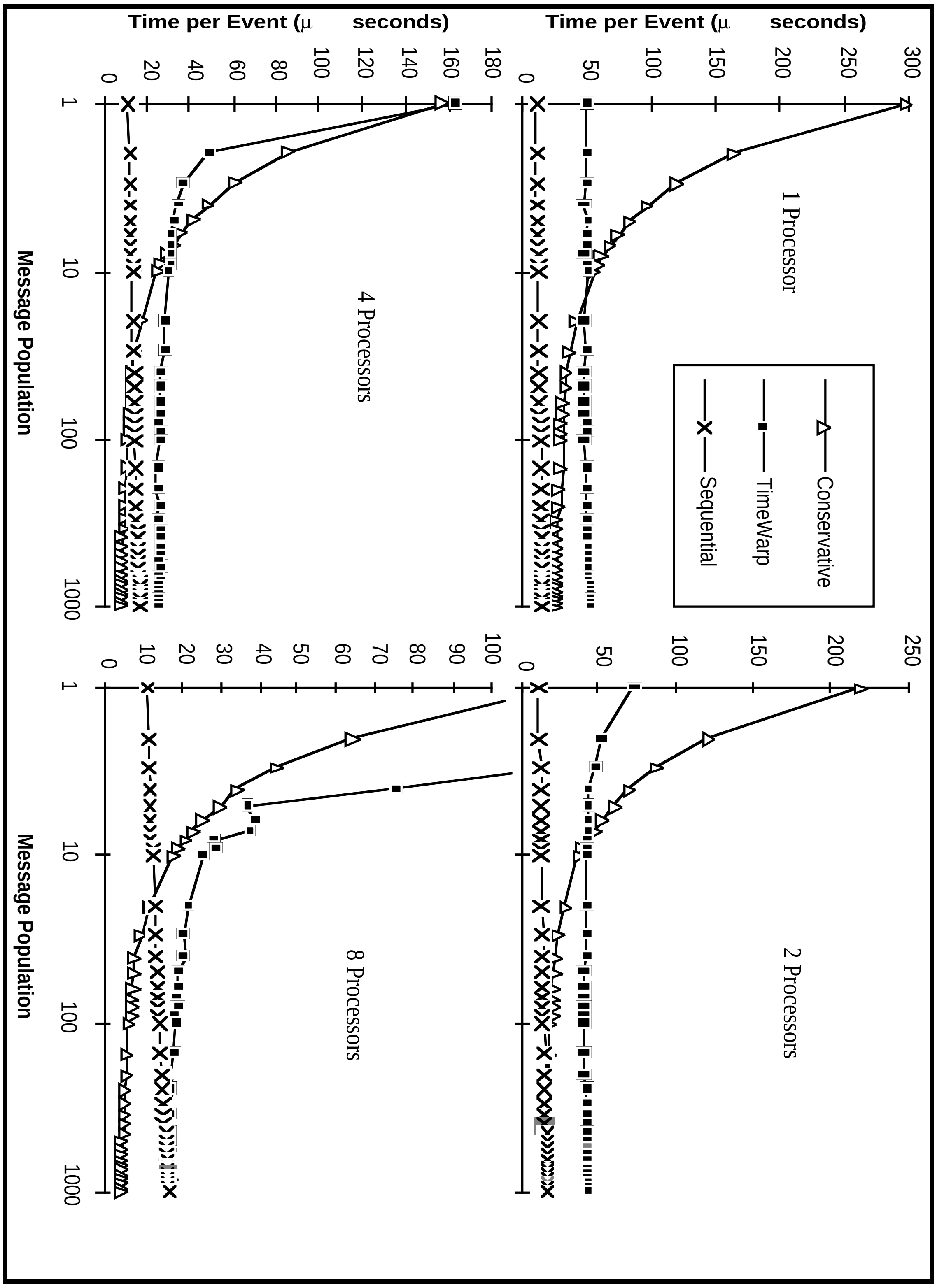

Figure 4. Event Overhead

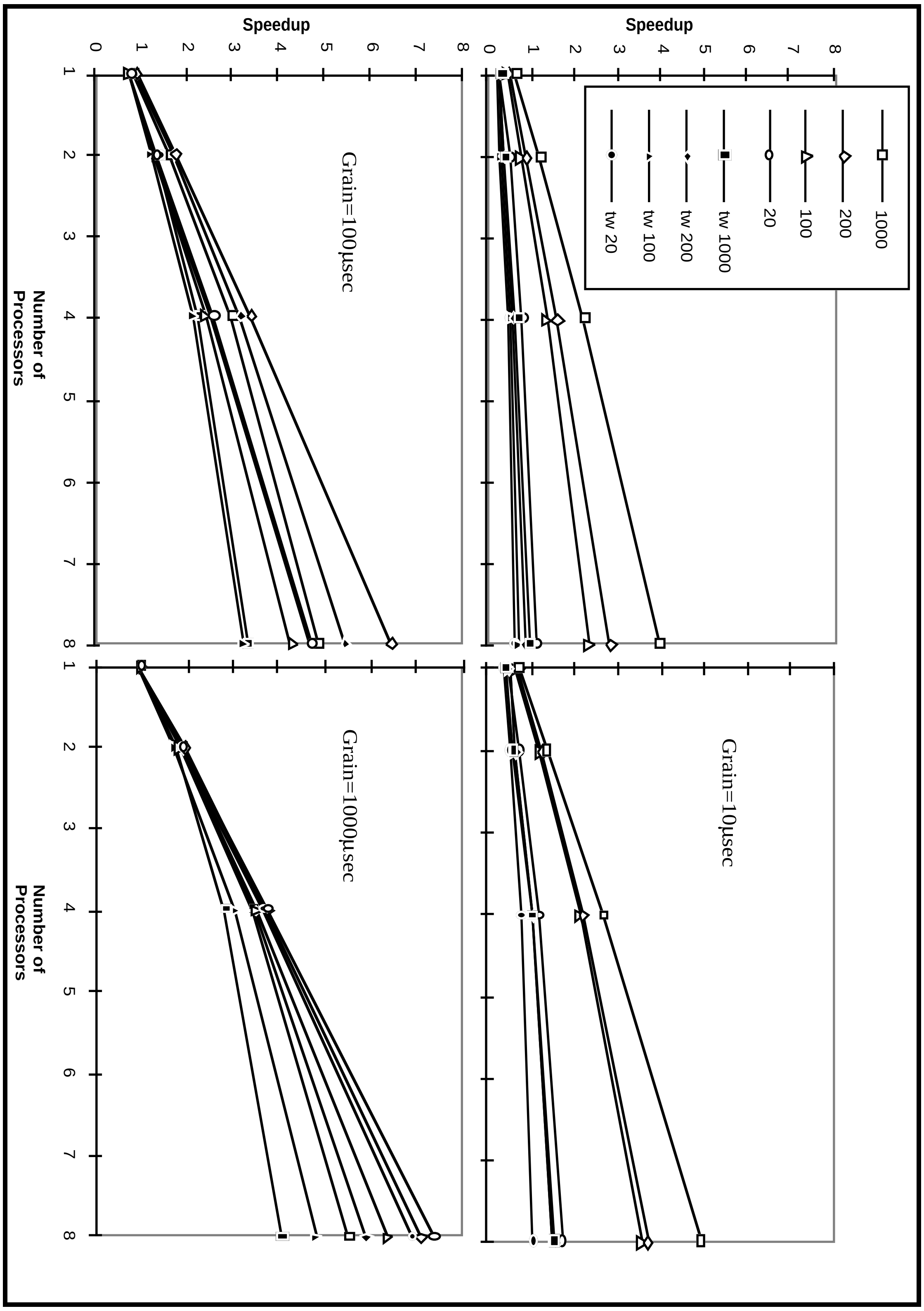

Figure 5. Speedup