A case study in synchronous parallel discrete event simulation

José Miguel Alonso [1,2,*], José A.B. Fortes[2,§]

[1] Departamento de Arquitectura y Tecnología de Computadores
UPV/EHU
Apdo. 649
20080 San Sebastián
SPAIN

[2] School of Electrical and Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907-1285
USA

acpmialj@si.ehu.es, fortes@ecn.purdue.edu

* This research was conducted while José Miguel was a Visiting Scholar at Purdue University.

§ This research was partially funded by the National Science Foundation under grants MIP-9500673 and CDA-9015696.

Table of contents

1 Introduction 1
2 Predicting the performance of SPED 4
3 Implementation 6
4 Experiments 8
5 Performance results 10
6 Improving the performance 13
7 Conclusions 21
References 22

Abstract

This paper considers the suitability of SPED, a synchronous parallel discrete event simulator, for the study of message passing networks. The simulation algorithm is described, and its potential performance is assessed showing that, under some simplifying assumptions, SPED might offer speedups directly proportional to the number of processors used in the simulation. An implementation of SPED in a distributed memory parallel system is used to study a model of an interconnection network for a multicomputer. Experiments show that SPED performs nearly as expected, as long as the *event density* imposed on the LPs is above a certain threshold. If this is not the case, the overhead due to synchronization plus communication dominates the execution time, and the achieved speedups are not as good.

Some ways to improve the performance of SPED are proposed: a method to reduce the number of messages interchanged during the simulation, and a new algorithm for synchronous PDES, called PTD-NB (Parallel Time Driven No Barriers), which reduces the synchronization overhead by removing barrier operations and can be easily implemented in multicomputer systems without support for global synchronization operations.

Keywords

Parallel discrete event simulation, synchronous PDES, multicomputer networks, performance analysis, barrier synchronization.

1 Introduction

The study of large and complex models of dynamic systems by means of computer simulation, a common activity in many science and engineering fields, is a computationally demanding task. The simulation community is in continuous search of new techniques to accelerate this kind of studies. One of the most promising possibilities comes from the use of parallel computers. An extensive set of techniques can be found in the literature to perform PDES (Parallel Discrete Event Simulation). Most of those techniques have in common the fact that parallelism is exploited by *model decomposition*: the model to simulate is divided into several parts, and each part is assigned to a Logical Process (LP). The resulting collection of LPs can run concurrently, each one simulating its part of the whole. LPs communicate and synchronize by passing messages that contain events scheduled by one LP (the sender) to be processed by another LP (the receiver).

In order to maintain the *causal relationships* among the events in the simulation, a synchronization mechanism is needed. There are two broad groups of synchronization mechanisms, which differ in the way (simulated) time information is perceived by the LPs.

- In *synchronous* methods the simulation clock is global, that is, all the LPs share a common view of time. Only those events with the same (or very close) timestamp are executed in parallel [PWM79, Luba88, SSH89, YTH89, KY91, Soul92].
- In *asynchronous* methods each LP has its own local clock. The objective of these methods is to allow events with different timestamps to be processed in parallel, not necessarily in strict timestamp order but taking care of maintaining the causal relationships of the events to ensure simulation correctness. The best known approaches to asynchronous PDES are those by Chandy-Misra-Bryant [CM79, Brya77], and Jefferson's Time Warp [Jeff85].

This paper studies one particular algorithm in the first group, which we call SPED (Synchronous Parallel Event Driven). Each LP of a SPED simulator keeps the same data structures of a single, sequential event-driven simulator: clock, state variables, statistics and event calendar. The clocks of all the LPs always keep the same value, so it can be said that the LPs share a common clock. The rest of the data structures are private. The event calendar stores self-scheduled events (events scheduled by one LP for its own future) as well as those events scheduled by other LPs and received via messages.

```
clock = 0;
while (clock <= end_of_simulation) {
     t = minimum_timestamp();    /* step 1 */
     clock = global_minimum(t); /* step 2 */
     simulate_events(clock);     /* step 3 */
     synchronize();              /* step 4 */
}
```

Figure 1. Outline of a logical process that forms part of a Synchronous Parallel Event Driven (SPED) simulator.

Each LP performs the basic algorithm depicted in Figure 1. It consists of a loop where iterations are separated by barriers. Each iteration performs four steps. First each LP obtains the timestamp of the earliest message of its event calendar. Then, a global operation is performed to compute the minimum among those values; the resulting minimum is assigned to the clock of all the LPs. In the third step each LP consumes all the events whose timestamp equals the new value of the clock. The last step is the barrier synchronization; it is needed to make the LPs start the next iteration at the same time. The LPs should not proceed to the next iteration until all the messages generated in the previous step have been delivered and safely stored in the corresponding event calendars.

The algorithm guarantees that at least one LP will consume one event in each iteration: the one that was used to compute the new clock. In the worst case, SPED behaves exactly like a sequential simulator, because it is possible that in a given iteration only the LP with the earliest event has something to do, while the others simply await to proceed to the next iteration. However, in a well balanced scenario with a reasonable event density (defined as the average number of events with the same timestamp), SPED can efficiently exploit the available parallelism, with a moderate synchronization cost. Two additional positive aspects can be found in this method: the simplicity of the design (which makes the simulator easy to build and maintain) and the possibility of an efficient implementation in SIMD systems, while other approaches to model distribution simulation are best suited for SPMD or MIMD systems.

The outline of the remainder of this paper is as follows. In §2 a simple analytical model of SPED is used to predict its performance. In §3 an implementation of SPED in an Intel Paragon multicomputer is described. A model of a message passing network for a multicomputer system, described in §4, has been used to test this implementation. The results of a set of experiments, along with the conclusions drawn from them, are presented in §5. In §6 two ways of improving the performance of SPED are proposed and evaluated: a new algorithm called STD-NB (Synchronous Time Driven-No Barriers), and a way to reduce the number of messages interchanged in a synchronous parallel simulation, applicable to both SPED and STD-NB. The paper ends with a summary of conclusions in §7.

2 Predicting the performance of SPED

Felderman & Kleinrock perform in [FK90] a comparison of the potential performance of SPED versus an asynchronous parallel simulator such as Time Warp. The study is simplistic: in the synchronous case the only overhead that is considered is the time spent synchronizing. In the asynchronous case it is assumed that the simulator performs its best and no time is spent in synchronization. Although the main objective of this work is to compare both methods, in this section we will only use some results obtained for the synchronous algorithm.

In order to simplify the analytical study, it is assumed that P processors execute K *tasks* sequentially. Each processor p must perform tasks $T_{p1} \ldots T_{pk} \ldots T_{pK}$ in sequential order. A task will take a random amount of time to complete execution on any processor—this is called the *task time*. Each processor houses exactly one LP, so both terms can be used interchangeably.

Using the synchronous approach, a processor must wait for all other to complete a step before continuing. Each processor must wait until every processor has completed task i before starting with task (i +1). This is a staged execution with K stages, where each stage takes as long as the slowest processor.

Under these assumptions, if the task times are exponentially distributed random variables (with mean 1/ μ), the expected completion time for a synchronous simulator is K times the maximum of P exponentials. This value can be expressed as [FK90]:

$$E[T_{SPED}] \approx \frac{K}{\mu}\left(E + \ln P + \frac{1}{2P} - \frac{1/12}{P(P+1)}\right)$$

where E = Euler's constant ≈ 0.57722.

The previous result depends on the assumption of an exponential distribution for task times. If now it is assumed that task times are uniformly distributed between 0 and X, the expected value of the execution time can be expressed as [FK90]:

$$E[T_{SPED}] = KX\frac{P}{P+1}$$

From these results, and assuming that no method can achieve a speedup greater than P for P processors, Felderman & Kleinrock state that the maximum achievable speedup of SPED is $(P/\ln P)$ under the assumption of exponential task times, and P under the assumption of uniform task times. They also conjecture that the results for exponential task times are due to the infinite tail of the exponential distribution and may therefore be applicable to other distribution with infinite tails. Similarly, the results for uniform task times could be applied to *any distribution with finite support*.

For the kind of models we are studying, a *task* is a set of events with the same timestamp, simulated in the same iteration (stage). The execution time of an event is approximately constant, but the number of events in each stage is random, in such a way that the size of that set (times a constant) is a good approximation of the *task time* as previously defined.

The duration of each stage is bounded: each LP simulates a finite number of model elements, and the number of events that can happen in each of those elements in each cycle of simulated time is also finite. According to the previous discussion, we can expect that, to a first approximation, speedup should be proportional to the number of processors. However, the analysis in [FK90] does not consider other costs of SPED, such as the interchange of messages among LP in order to schedule events. This overhead increases with the number of processors, so it should come as no surprise if the actual performance is not up to our expectations.

3 Implementation

The SPED algorithm described in the introduction has been implemented in an Intel Paragon multicomputer, using the ANSI C programming language with Intel's NX library for parallel programming. This library includes point to point communication in different styles: blocking, nonblocking, interrupts, etc. We have used the blocking functions:

- **Csend**() sends a message. This function returns to the calling process immediately, once the message has been stored in a system buffer.
- **Crecv**() receives a message. This function blocks the calling process until a message is received.

Each message sent must have a *tag* or message type, in such a way that a receive function may select just those messages with a given tag.

NX also provide global communication functions. Among those, a particularly interesting one for our purposes is **gilow** (), which synchronizes all the processing elements that participate in an application and then computes the minimum (integer) value among those provided by the calling processes. **Gsync**() provides just the synchronization (a barrier), without performing any reduction operation.

Using these functions, the basic design of a LP in our implementation of the SPED simulator follows, in broad lines, the description given before. Each LP_i executes a loop of 4 basic operations:

Clock advance. LP_i computes the minimum timestamp among the events stored in the local event calendar, t_i. Collectively, the LPs compute the minimum among all those values, $c = \min(t_i)$. This global operation also performs a barrier synchronization.

Event consumption. LP_i advances its clock to reach c. All the events with this timestamp can be executed safely, because there are no causality relationships among them. During this step, internal events are stored in the local event calendar, while external events are stored in an auxiliary data structure.

Message distribution. LP_i sends the external events generated in the previous step, by means of messages. This is done in two phases: (a) every neighbor is informed about how many messages will be sent to it and (b) the messages are actually sent.

Message gathering. LP_i reads all the external events generated in other LPs and sent to it. Again, this is done in two phases: (a) gather from the neighbors the number of messages to receive and (b) actually receiving the messages.

The resulting code for a LP is sketched in Figure 2. The design of the message distribution and message gathering phases, along with the barrier at the beginning of each phase, ensures that all the messages generated in one iteration are safely received and stored in the same iteration, without interfering with the next one.

```
process SPED-LP:
while (clock <= end_of_simulation) {
     ts = minimum_timestamp();
     clock = gilow(ts);
     while (next_event_time() == clock) {
          m = next_event();
          consume(m);
     }
     send_messages();
     receive_messages();
}
```

Figure 2. Sketch of the main loop of a logical process that uses the SPED synchronization mechanism, as implemented in the Paragon.

4 Experiments

To test our implementation of SPED we used a model of a message passing network for a multicomputer system. The details of this model can also be found in [Arru93]. It is basically a torus network of message routing elements (routers). Each router (Figure 3) is connected to four neighbors and to a processing element, via bi-directional channels. All the messages have the same length. The flow-control mechanism is cut-through and, for this reason, message queues are needed, which are associated to each output channel.

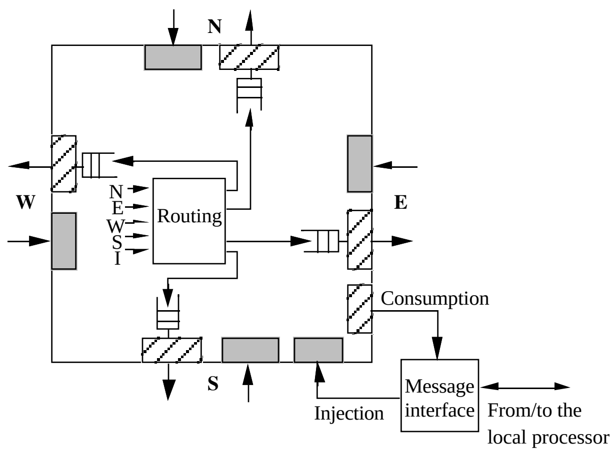


Figure 3. A model of message router. The actual model being simulated is a torus network were each node is a <router, processor> pair, representing a multicomputer system.

Three important parameters of this model are:

- *Message length (M)*, measured in *flits* (flow-control digit). It is assumed that a flit advances from one router to another in one cycle (of simulated time).
- *Load of the network (L)*. It is the amount of information generated by the processing nodes, measured from 0 (none) to 100 (theoretical maximum). The maximum (100) corresponds to the bisection bandwidth (in flits) of the network, that is, to a situation where the channels of the network bisection are continuously utilized. A given load level can be reached with many short messages or with few long messages.
- *Size of the network (S)*. Number of <router, processor> pairs.

These three parameters have a direct influence on the event density of the simulation: it increases with L and S, and decreases with M.

Other characteristics of the model such as the size of the router queues, the communication patterns and the routing strategy were fixed, although they could be easily changed. The transit queues can hold 10 messages, while the injection queue has enough room for 4 messages. A random communication pattern was used: a node can send messages to any of the other nodes, with the same probability. The (time) separation between messages generated at a given node is exponentially distributed, with a mean that is directly proportional to the message length and inversely proportional to the network size and load. The routing strategy is oblivious in order of dimension (first X, then Y), following a technique described in [Arrua93, Izu94] to avoid communication deadlocks. This is needed because the topology is a torus.

To perform a parallel simulation the simulated network is divided in squares of the same size, and each square is assigned to a logical process. This means that each LP simulates an aggregate of routers, not only one. The size of the square depends on the number of available processing elements (PEs): exactly one LP is mapped onto each PE.

A sequential event-driven simulator was also implemented, in order to have a reference point to assess the achieved performance of the parallel simulators under different configurations and workloads. After measuring the execution time of a sequential and a parallel simulation of the same model, a speedup figure can be computed. The model simulated by the sequential and parallel programs are basically the same but, in order to make fair comparisons, some optimizations were included in the sequential version that take advantage of the use of a single memory space.

5 Performance results

An exhaustive set of experiments performed with SPED simulating the model described in the previous section led to the following conclusions: for a given value of P (number of processing elements) the performance of SPED increases with the event density, that is, the best conditions for SPED come with large values of L (load) and S (size), and small values of M (message length); additionally, for a given event density (L, S and M), speedup increases with P.

These results are illustrated in Figure 4, which shows the speedups achieved when performing the following experiments:

- Experiment **1**: using a number of processors P ranging from 1 to 64, a network of $S = 24 \times 24$ routers is simulated during 4000 cycles. The values of the other parameters are $L = 50$ and $M = 4$.
- Experiment **2**: using a number of processors P ranging from 1 to 100, a network of $S = 90 \times 90$ routers is simulated during 1000 cycles. The remaining parameters are the same as in experiment 1.

Separately, each experiment tells us how well the execution time improves with the number of processors. If the attention is fixed on the results of both experiments for a given number of processors, the influence of the event density (in this case, only of S) can be seen.

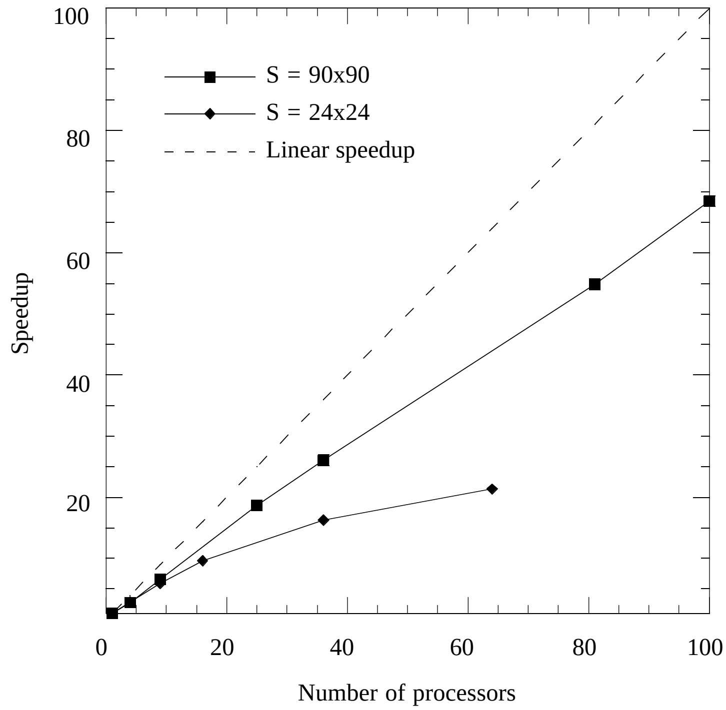


Figure 4. Speedups achieved with SPED running experiments **1** and **2**. The same experiments, using an optimized version of the model, were run with a sequential simulator, whose execution times were used as the reference to compute speedups.

The obtained results are, for the case of high event density (experiment **2**) nearly as expected from the analytical study of §2: linear with the number of processors. However, when this density is smaller, the performance is not very good when many processors are used. We hypothesize that this should be due to overheads not considered in the study of Felderman & Kleinrock, such as communication among processors, which increases with the number of processors.

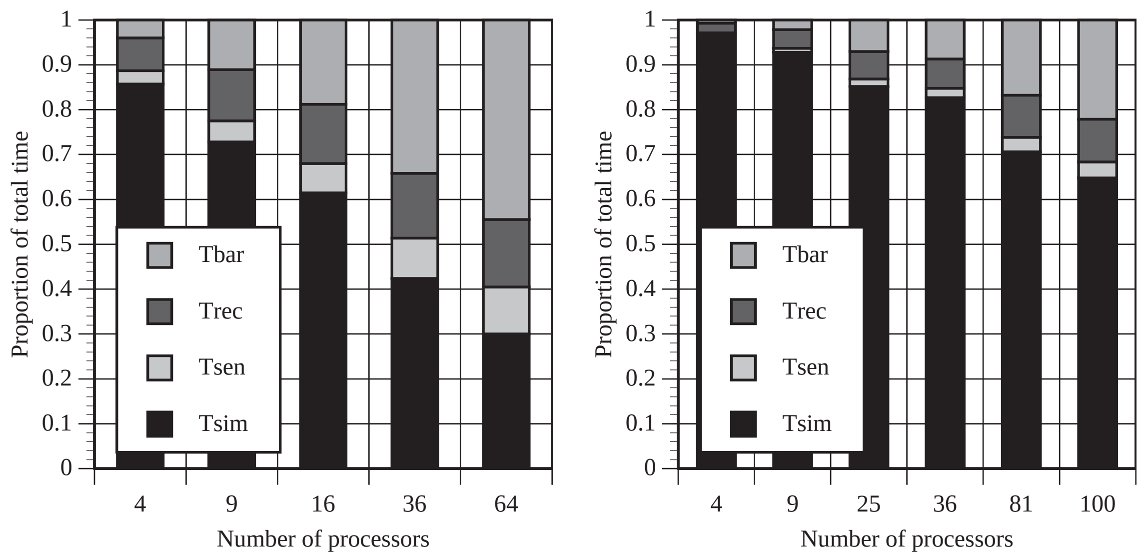
To further understand the behavior of this SPED implementation we instrumented the simulator to measure how much time LPs spend executing events and how much performing synchronization tasks. This is shown in Figure 5. The total execution time is divided into four parts:

Tsim is the time spent managing and executing events; management tasks are event calendar insertions and deletions.

Tsen is the time spent sending messages to other LPs.

Trec is the time LPs spent awaiting to receive, from their four neighbors, messages generated as a result of iteration completions. Trec is basically synchronization time, because it includes the time a LP spends awaiting its neighbors to finish, plus a small overhead due to the invocation of a system call.

Tbar is the time LPs spent computing the value of the clock at the beginning of each iteration. This global minimum operation is a form of barrier, so this time is also devoted to synchronization.



Experiment **1**. Network 24 ×24 routers. Experiment **2**. Network 90 ×90 routers.

Figure 5. Distribution of total execution time among simulation and synchronization for experiments **1** and **2** , expressed as a percentage of the total time. Tbar = time spent barrier-synchronizing; Trec = time spent receiving messages; Trec = time spent sending messages; Tsim = time spent executing events.

It is easy to see how a highly loaded simulator performs better because it is able to spend most of its time executing useful work. If the load is lowered then the ratio of computation time to synchronization time degrades considerably.

6 Improving the performance

The experimental evaluation of SPED presented in the previous section shows that the performance of SPED is quite satisfactory. However, the study of the time that the simulator spends in computation, synchronization and communication activities suggested that performance could be further improved if some of the overheads were reduced. In this section two of such optimizations are analyzed, which have been implemented and tested, giving satisfactory results. The first one reduces the synchronization time by eliminating barrier operations. The second one reduces communication overheads, grouping events to allow the simulator to interchange fewer messages of larger size.

6.1 Removing barriers

The experiments performed with SPED show that, in most cases, the number of barriers executed by the LPs is equal to the number of simulation cycles, which means that the clock advance from iteration to iteration is always one time unit or, in other words, that SPED actually works as a time driven simulator. In fact, we implemented and tested the algorithm of Figure 6, which we will call PTD (Parallel Time Driven), obtaining the same results for the majority of the performed experiments. The exceptions were some experiments with an extremely low event density, where SPED performed slightly better than PTD.

```
process PTD-LP:
clock = 0;
while (clock <= end_of_simulation) {
     gsync();                   /* Sync. 1 */
     while (next_event_time() == clock) {
          m = next_event();
          consume(m);
     }
     send_messages();
     receive_messages();        /* Sync. 2 */
     clock++;
}
```

Figure 6. Sketch of a logical process using the PTD synchronization mechanism.

The PTD algorithm shows that each LP must perform two levels of synchronization, indicated in the figure. The first one is the barrier, which ensures that all the LPs starts the next iteration at the same time. This barrier substitutes the global operation performed in SPED to compute the timestamp of the events to process. With PTD that computation is no longer needed, because the clock always advances one unit. The second point of synchronization is not global, as the barrier, but affects just a LP and its neighbors: function **receive_messages**() returns only after the neighbors have finished simulating the events the current value of the clock, and have executed **send_messages**().

The fact is that the barrier is not necessary at all, if we provide the LPs with a means of guaranteeing that messages generated during two different iterations are never mixed. However, the straightforward implementation of **send_messages**() and, particularly, of **receive_messages**() do not prevent the mixture of messages, thus requiring the use of the barrier.

Fortunately, some minor changes to these functions can make the simulation work properly without the barriers. Figure 7 shows the resulting algorithm, that we will call PTD-NB (Parallel Time Driven-No Barrier).

```
process PTD-NB-LP:
clock = 0;
while (clock <= end_of_simulation) {
     while (next_event_time() == clock) {
          m = next_event();
          consume(m);
     }
     send_messages2();
     receive_messages2(); /* Sync. */
     clock++;
}
```

Figure 7. Sketch of a logical process using the PTD-NB synchronization mechanism.

In PTD-NB the message distribution phase, **send_messages2**() is slightly different from **send_messages**(). Every message is tagged, before being sent, with a label with two fields:

Type of the message. Possible values are ANNOUNCE and USEFUL. Firstly, a message of the ANNOUNCE type is sent to each neighbor, to inform about how many USEFUL messages are being sent as a result of the iteration just finished. Then as many USEFUL messages as previously advertised are sent.

Sending timestamp. Both ANNOUNCE and USEFUL messages are tagged with the current value of the clock.

With this information, the message gathering phase can also perform the necessary synchronization to separate one step from the next. This is done using the message labels. Function **receive_messages2**() executed at the end of iteration i proceeds as follows:

- The LP awaits to receive as messages as neighbours, selecting only those tagged <ANNOUNCE, i>. Then it knows how many USEFUL messages will be received, say m.

- Then the LP awaits to receive exactly *m* messages tagged <USEFUL, *i*>. When the m messages have been received, it is sure that no new messages belonging to iteration *i* will be received.

After **receive_messages2**() returns the LP knows that iteration *i* has finished, so the clock can be incremented and the next iteration starts.

It should be clear that the synchronization effort is very similar with or without barriers. Although under some circumstances it is possible that the LPs need to spend less time blocked (see Figure 8), a slow LP would make its neighbors slow down, and those would do the same, thus slowing down the simulation as a whole. However, removing barriers can reduce the total execution time because it avoids calling, at the beginning of each iteration a costly system call. Another advantage of PTD-NB is its suitability to be implemented in multicomputers that lack support for global operations.

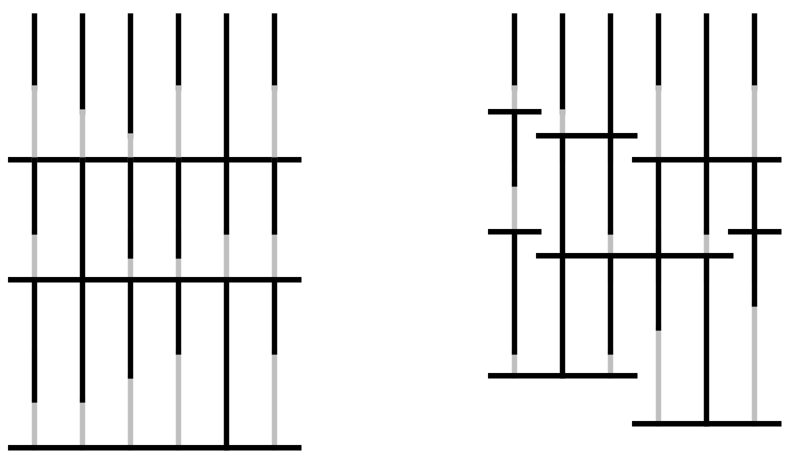


Figure 8. Evolution of a collection of LPs using PTD (left) and PTD-NB (right). Each line represents one LP. The black part is the time spent consuming events (computation phase), while the gray part is waiting time (synchronization phase). In PTD each LP finishes an iteration when all the LPs finish the computation phase. In PTD-NB a LP finish an iteration when it and all its neighbors—in this case, the one at the left and the one at the right—finish their computation phases.

The first point has been confirmed performing some experiments in the Paragon. A reduction in execution time proportional to the number of simulation cycles (i.e., of barriers in PTD or SPED) can be achieved. This reduction is more significant when the number of processors involved in the barriers is high. Figure 9 compares SPED to PTD-NB running Experiment **1**. Execution times of PTD-NB are always shorter, and the time gain improves with the number of processors, the reason being that the cost of a barrier increases with the number of nodes to synchronize.

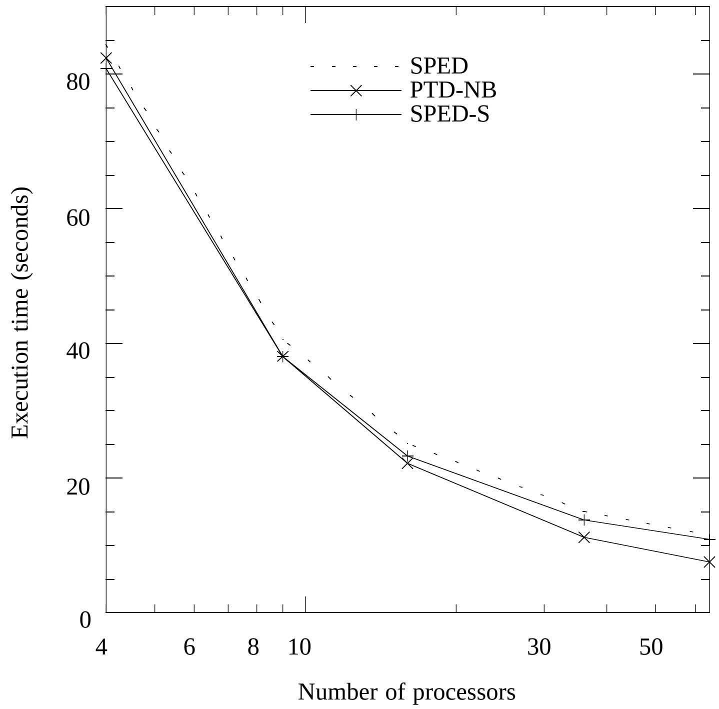


Figure 9. Execution times of SPED, PTD-NB and SPED-S (§6.2) running Experiment 1 (network of 24×24 routers). PTD-NB improves significantly with the number of processors, because the cost of a barrier increases with this number while the number of barriers remains constant. SPED-S takes about 6% less time to complete than SPED, so the effect of this improvement is more noticeable when only a few processors are used and, therefore, execution times are longer.

Figure 10 (left) shows the distribution of execution time among simulation (Tsim), sending messages (Tsen) and receiving messages (Trec) for Experiment **1**. Compare with Figure 5 (left). The proportion of time spent receiving messages is now, as expected, much higher, because it includes all the synchronization costs. The proportion of time spent executing events and sending messages also higher. Although the absolute values of Tsim and Trec are the same for SPED and PTD-NB, in the latter case the cost of performing barrier operations has been removed and, therefore, the overall synchronization overhead has been significantly reduced, specially for large numbers of processors.

To finish this section, we must mention that both PTD and PTD-NB have a pitfall that prevents them from being of general use. This is that LPs can not schedule events with zero timestamp increment, except for self-scheduled events. In other words, a LP it is not allowed to schedule an event timestamped i for another LP when the simulation clock is i. The reason is simple: if that event was scheduled, it would be sent as a message at the end of iteration i, and would not be processed because next iteration would only simulate events timestamped $i+1$. The event-driven simulator, SPED does not have this limitation.

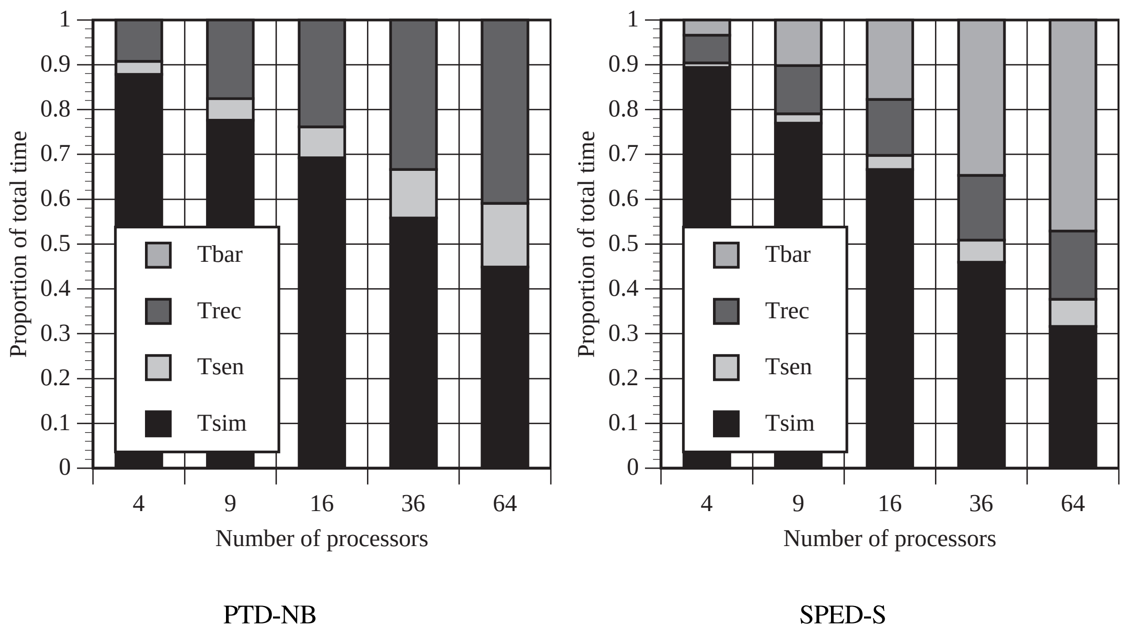


Figure 10. Distribution of the total execution time for PTD-NB and SPED-S running Experiment 1 (compare to Figure 5 - left). For all the cases Tsim remains constant in absolute terms but the overhead reduction leads to a higher efficiency in the use of the processors.

6.2 Reducing the number of messages

In the description of the previous algorithms, SPED, PTD and PTD-NB, it has been assumed that, for each event scheduled by a LP for a different one, a message must be sent. The implementation is this way because it shares most of the code with other parallel, asynchronous simulators, where messages are sent as soon as possible to reduce the synchronization effort. However, SPED allows an alternative way of dealing with messages.

Communication operations are very expensive in most parallel computers, so a reduction in the number of messages immediately results in a performance improvement. This is true even if the amount of information actually interchanged remain fixed, because sending/receiving a message has a significant cost in terms of software overhead, which is specially significant if messages are short. The longer the message, the smaller the overhead, in relative terms. Under these conditions, SPED (PTD, PTD-NB) can be adapted to reduce the number of messages sent by a LP at the end of each iteration, to a number equal to the number of neighbors. This is done by grouping all the events scheduled for a neighbor in a single message of variable size.

In order to realize this message reduction, the **send_messages**() operation must be changed. Now it is not necessary to advertise how many messages are going to be sent to each neighbor, and then send each event in a message. It is enough to send a single message that might be empty or contain many events grouped together.

The **receive_messages**() operation must change accordingly. The main problem is that the length of a message is not known in advance, so it must be obtained at run time. This forces the receive operation to be split in several parts:

- Await for a message to arrive, using function **cprobe**(), available in the NX library.
- Obtain the message length, using **infocount**(). This way it is possible to know how many events are arriving, dividing the obtained result by the size of the data structure that stores an event.
- After allocating the right buffer size, actually receive the message with **crecv**().

This improvement has been implemented and tested running Experiment **1**, the network of 24× 24 routers. Figure 9 shows the execution time of SPED (the version that sends a message per event plus a message per iteration and per neighbor), versus SPED-S (the new version that sends only one message per iteration and per neighbor). SPED-S always runs faster, being the execution times are about a 6% shorter.

When the execution time of the simulator is split into components, it can be seen that SPED-S spends noticeably less time sending messages, because the effort in terms of system calls is considerably reduced. The time spent receiving messages is also reduced, but not as much, because (a) splitting the receive operation now requires more system calls and, (b) most of the time the simulator spends in receive operations is actually waiting time, that is, does not depend on the way messages are received but on the time it takes all the neighbors to finish their computations. The other components of the time (computation, barriers) does not improve. Figure 10 (right) shows the new distribution of time for the case of Experiment **1** .

The advantage of grouping is, in absolute terms, proportional to the event density: if a LP needs to send many messages each iteration, those are longer, and make a better use of the communication mechanisms of the target multicomputer. For this reason the improvement in Experiment **1** is not very important for the case of 64 processors.

7 Conclusions

In this paper we have described SPED, a synchronous, parallel, event driven simulator, which is specially attractive due to the simplicity of its design: it is like a sequential, event driven simulator but many events can be processed simultaneously, if they have the same timestamp (and, thus, they do not depend on each other). Despite its simplicity, it has been shown how it has the potential to offer speedups proportional to the number of processors.

An implementation of SPED in a Paragon parallel computer has been used to simulate a model of a message passing interconnection network, designed to constitute the communication infrastructure of a multicomputer. Experiments with different parameters of the model, and different number of processors, allowed to identify the factors that improves the performance of SPED. If the workload imposed by the model, measured in terms of event density (number of events per unit of simulated time) is high, SPED spends most of its time performing useful computation. In these circumstances, the achieved speedups are nearly as predicted, that is, proportional to the number of processors. However, if the LPs have only a few events to consume at each iteration, synchronization and communication takes most of the time, and the achieved efficiency is not as good as expected.

Although the overall performance of our SPED implementation is satisfactory, it is possible to improve it. We have introduced PTD-NB, a new algorithm to perform time driven parallel simulation without requiring the use of barrier synchronization operations. Although PTD-NB has some restrictions that prevent its use for any kind of models, for the one used in our experiments the behavior of SPED and PTD-NB are identical, being PTD-NB faster. The new algorithm is specially interesting when barrier operations are expensive (including cases where a large number of processors are involved) or, simply, not available.

Another change that can be introduced in our implementations of SPED and PTD-NB is a reduction of the number of messages interchanged by the LPs of the simulator, by grouping several events in a single message of variable size. This optimization is suitable for current parallel computers, where sending messages incurs in severe overheads. Its use improves the execution speed, reducing run time by about 6%, for the performed experiments.

References

[Arru93] A. Arruabarrena. *Análisis y evaluación de sistemas de interconexión para procesadores masivamente paralelos*. PhD dissertation, Departamento de Arquitectura y Tecnología de Computadores, Universidad del País Vasco, Sept. 1993.

[Brya77] R.E. Bryant. *Simulation of packet communications architecture computer systems*. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[CM79] K.M. Chandy and J. Misra. "Distributed simulation: a case study in design and verification of distributed programs". *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, Sept. 1979, 440—452.

[FK90] R.E. Felderman and L. Kleinrock. "An upper bound on the improvement of asynchronous versus synchronous distributed processing". *Distributed Simulation 1990*, 131—136.

[Izu94] C. Izu. *Análisis, evaluación y aportaciones al diseño de arquitecturas para encaminadores de mensajes.* PhD dissertation, Departamento de Arquitectura y Tecnología de Computadores, Universidad del País Vasco, May 1994.

[Jeff85] D.R. Jefferson. "Virtual time". *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, 404—425.

[KY91] P. Konas and P-C. Yew. "Parallel discrete event simulation on shared memory multiprocessors". *Proc. of the 24th Annual Simulation Symposium*, New Orleans, Luisiana, April 1991, 134—148.

[Luba88] B.D. Lubachevsky. "Bounded lag distributed discrete event simulation". *Proc. SCS Multiconference on Distributed Simulation*, Feb. 1988, 183—191.

[PWM79] J.K. Peacock, J.W. Wong and E.G. Manning. "Distributed simulation using a network of processors". *Computer Networks*, Vol. 3, No. 1, 1979, 44—56.

[Soul92] L. Soulé. *Parallel logic simulation: an evaluation of centralized-time and distributed-time algorithms*. PhD dissertation. Stanford Univ. Technical Report CSL-TR-92-527, June 1992.

[SSH89] L.M. Sokol, B.K. Stucky and V.S. Hwang. "MTW: a control mechanism for parallel discrete simulation", *Proc. Int. Conf. Parallel Processing*, Vol. III, Aug.1989, 250—254.

[YTH89] Q. Yu, D. Towsley and P. Heidelberger. “Time-driven parallel simulation of multistage interconnection networks”. *Distributed Simulation 1989*, 191—196.