In general, we have confirmed that our preliminary ideas about which factors would improve the performance of the conservative parallel simulator were right. In order to take advantage of this sort of simulator, we need a model with a high degree of internal communication, that allows the processes to remain synchronized. In the model used in this study, this happens when the load is high and the messages are short. These two parameters have the strongest influence in the performance of the simulator.

The knowledge of the router behaviour may allow the simulator to exploit the lookahead property, which helps to maintain a good performance when the simulator has not useful work to do. Unfortunately, the lookahead ability of our message routing network is poor, generally speaking. It can not be exploited when it is most needed: in the case of low load. Nevertheless, it can help to make the speedup even better in highly loaded systems, especially when the messages are long.

Finally, if the simulator is running in a set of processes statically assigned to a set of processors, it is important to use coarse grain processes, that is, assign the maximum amount of work to one process. This way, less processes are used to run the model, reducing the synchronization effort. This idea should not led us to the extreme of assigning only one process to a processor, because if the process blocks, the processor stays idle. So, more than one process should be mapped onto a processor. According to this idea, we have seen that using intermediate grain size in our experiments led us to the best performance.

References

[1]

[2] J. Banks and J.S. Carson. "Discrete-Event System Simulation". Prentice Hall, Inc. 1984. ISBN 0-13-215582-6.

[3] R.E. Bryant, "Simulation of Packet Communications Architecture Computer Systems". MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[4] 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, pp. 440—452.

[5] R.M. Fujimoto, "Lookahead in Parallel Discrete Event Simulation". Proc. Int. Conf. Parallel Processing, Aug. 1988, pp 34—41.

[6] R.M. Fujimoto, "Parallel Discrete Event Simulation". Comm. of the ACM, Vol. 33, No. 10, Oct. 1990, pp. 31—53.

[7] "The Transputer Databook" INMOS Databook Series, INMOS Ltd., Bristol, U.K., 1989.

[8]

[9] D.R. Jefferson, "Virtual Time". ACM Trans. on Programming Languages and Systems, Vol. 7, No. 3, July 1985

[10] P. Konas, P-C. Yew, "Parallel Discrete Event Simulation on Shared Memory Multiprocessors". Proceedings of the 24th Annual Simulation Symposium, New Orleans, Luisiana, April 1991, pp. 134—148.

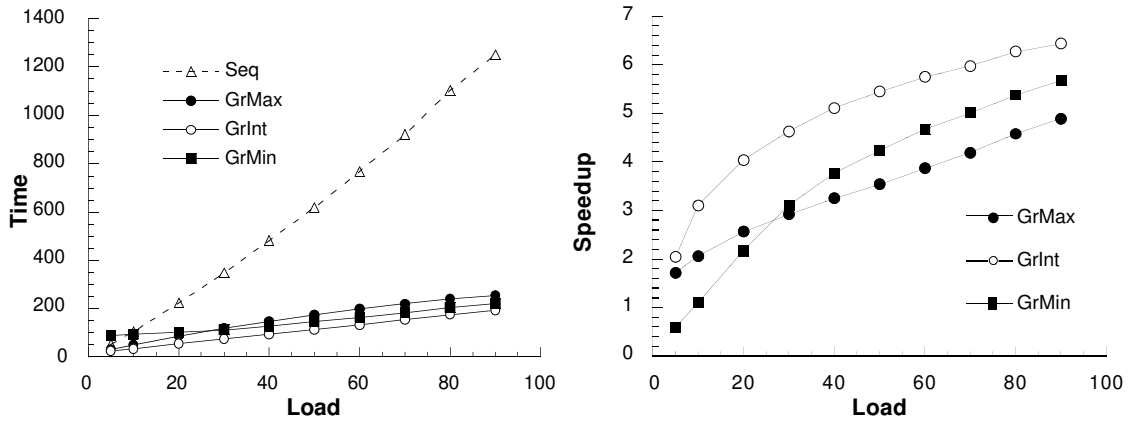
[11]

[12]

[13] J. Misra, "Distributed Discrete-Event Simulation". Computer Surveys, Vol. 18, No. 1, March 1989, pp. 39—65.

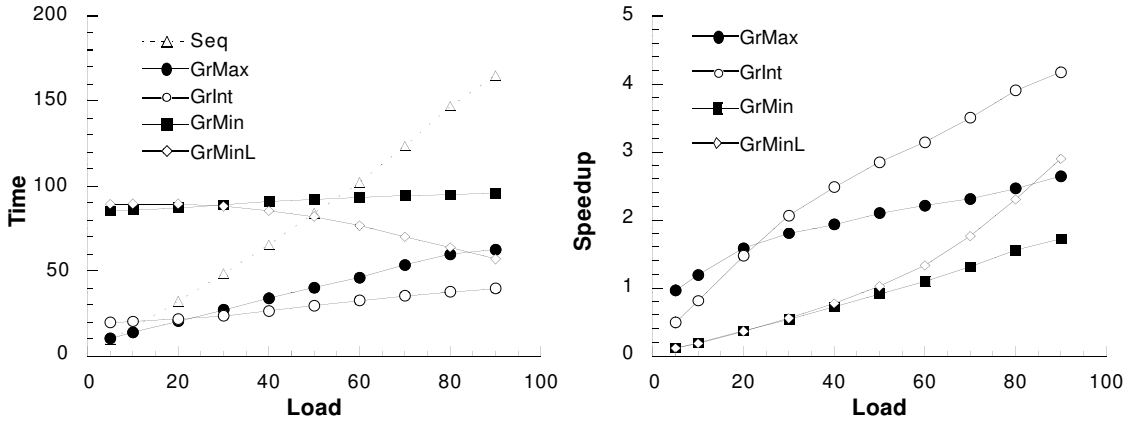
[14] D.B. Wagner, "Conservative Parallel Discrete-Event Simulation: Principles and Practice". Ph.D. dissertation, Univ. of Washington, 1989.

performance gains. The behaviour of the model being studied can be exploited effectively only for highly loaded networks routing long messages. This can be clearly seen in Figure 9 (GrMinL). We made some experiments trying to exploit the lookahead using messages of length 4, but the results were extremely disappointing: the execution times were slightly worse.



(a) (b)

Figure 8. Simulation of a network of 16x16 routers using 16 transputers.
Message length L = 4. (a) Execution time, in seconds.
(b) Speedup.



(a) (b)

Figure 9. Simulation of a network of 16x16 routers using 16 transputers.
Message length L = 32. (a) Execution time, in seconds.
(b) Speedup.

5 Conclusions

In this paper we have presented our experience using a conservative, parallel discrete-event simulator for evaluating a model of a message router to be used in multicomputers. The knowledge obtained in previous experiments with a very simplistic model has been applied, when possible, with this more realistic model.

4.2 Load and message length

These two parameters are closely related. For a given value of load, the average time between injections in each router is calculated using the message length and the topology of the network. A 100% load means that all the bandwidth of the network bisection is used [1]. The network is, thus, saturated. A given load rate, measured in bytes/cycle, can be achieved using few, long messages or many short messages.

The number of managed messages is especially critical in the conservative simulator. For a certain level of load C and a given message length L, the number of messages managed by the simulator during a given interval of (simulated) time is proportional to C/L. If L is small and the load C is high, then the number of messages is high, so the LPs are able to synchronize often, at the same time that they perform useful computation. If C is low or L is big, most of the time the LPs have not useful work to do, so the simulator advances mainly by means of null messages, that impose a serious overhead. This results in poor performance figures when comparing with the sequential simulator, that performs very well in this situation.

The reader can see in all the figures how the performance improves with the load. Comparing Figures 8 and 9 it can be seen that the parallel simulator behaves much better with short (simulated) messages, that increase the total number of (actual) messages managed in the simulation.

4.3 Lookahead

When a LP sends null messages to its neighbours, the best thing it can do is to maximize the value of the timestamp of the messages, in order to allow the maximum clock advance. Unfortunately, sometimes it is impossible to do that.

If a LP simulates only one router, then the behaviour of the simulation is quite *predictable*, and some *lookahead* can be extracted. Let us see the case that, at time t, a router has sent a header of a message through an output port; the simulator *knows* that no new header will be sent at least until $t+L$, where L is the length of the message. The difference between the current value of the clock and $t+L$ is the lookahead. However, when a LP simulates an aggregate of routers the computation of the lookahead is too expensive for the achieved result: most of the times the obtained value is 1, a minimum value that can be assumed without any computation. For this reason, we only show the effect of the lookahead for the case of minimum grain size.

It is clear that the lookahead is greater if the LP has recently interacted with its neighbour. Therefore, in highly loaded networks, where the number of interactions is big, the probability of computing high values for the timestamp of null messages is high too. So we have that (1) the ability to exploit the lookahead is especially interesting when the LP blocks frequently; (2) the blocking probability is higher when lower is the load and (3) with low load, the average value of the calculated lookahead is very low. In short: we have not been able to make simulations faster when we most needed it. Low load is fine for the sequential simulator, but it is the worst possible case for the parallel one. When the load is high the parallel simulation shows its ability to reduce execution times, because very few null messages are needed. If, in addition to that, we efficiently exploit the lookahead, the execution time is further reduced.

Although it has not been mentioned before, it is obvious that the length of the messages also influences the achieved lookahead. When L is big, the computed timestamps for null messages may be big too, but it should be reminded that, for a given load, the length of the messages determines the number of messages managed in the simulator. Again, we have some contradictory facts: (1) if the messages are long, the number of messages (and, therefore, interactions) is low; (2) if the messages are short, the number of messages is high, but the lookahead ability is low.

The practical result is that, if messages are short, any effort to exploit lookahead is useless. The needed effort to keep information about the recent interactions, and the overhead of computing timestamps from this information, surpass the potential

4.1 Grain size

Previous experiments reported in [11, 12], led us to think that the performance of a conservative parallel simulator running in a network of transputers, where communication overhead is very big [8], would be better with greater the grain size of the simulation. This way, the computation/communication ratio would be more balanced.

For a given number t of available transputers and a number r of routers to simulate, it is possible to decide how many message routers are assigned to a LP, say g, and then place the resulting number of LPs, calculated as r/g, in each transputer. The *grain size* is the value of g. Figure 7 shows the grain size possibilities for t=1 and r=16. The reader should notice that, in general, several intermediate values of grain size are possible, although in the selected examples only one can be chosen, if squares are used for the assignations.

Typical values for t and r are 16 and 256, respectively. For this case, the following configurations have been tested: g = 16 (maximum grain size, GrMax in Figures 8 and 9), g = 4 (intermediate grain size, GrInt) and g = 1 (minimum grain size, GrMin).

We first compared maximum vs. minimum grain size. We noticed that, when messages are long, coarse grain is clearly more effective than fine grain (Figure 9). On the contrary, when the messages are short fine grain is slightly better than coarse grain (Figure 8). Then we tried with the intermediate grain size. We found that this is *the best* alternative for intermediate and high loads. Only for low loaded systems the maximum grain size may give better performance. This behaviour can be explained analyzing the time spent for syncronization in the simulator:

- With maximum grain size, a LP can evolve quite autonomously, due to the big number of interactions between the routers assigned to that particular LP. Nevertheless, sometimes the LP has to co-ordinate with the others, and that leads the LP to send null messages and then block. This is extremely inefficient because, as the LP is the only *user* of the transputer, if it blocks, the freed CPU power is wasted.
- With minimum grain size, if a LP blocks plenty of others are awaiting to get the CPU. Here the problem is that, the bigger the number of LPs, the bigger is the number of null messages needed to keep the system synchronized and, what makes things even worse, as every fine-grain LP has very little work to do, it blocks very often.
- With intermediate values of the grain size we try to find a balance: there are few LPs, each one with enough work to do, so null messages are not needed very often. Additionally, as several LPs share a transputer, the probability of having a stopped CPU decreases.

As a conclusion, we were right when advising to use coarse grain simulations, and now we can add that it is very efficient to share a processor among several LPs, to avoid idle processors and make the maximum usage of the available CPU power.

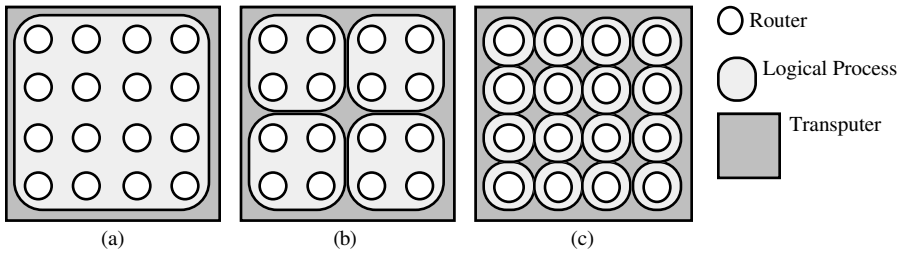


Figure 7. Grain size alternatives for 4x4 routers on a transputer.
(a) Maximum. (b) Intermediate. (c) Minimum.

```
function deliver_message:

h = acceptance_horizon();
ts = minimum_timestamp();
if (ts <= h) {
        s = message_with_minimum_timestamp();
        send(sig, s);
}
else if (h >= clock) send(sig, null_message);
else blocked_simulator = TRUE;
```

The function deliver_message() computes the message-acceptance horizon, as well as the value of the minimum timestamp among all the messages awaiting to be consumed. If this timestamp falls below the acceptance horizon, then the corresponding message can be safely removed from its queue and consumed (sent by *sig*). This is the behaviour that makes the simulator a *conservative* one.

When no message is ready to be consumed a null message is sent to the simulator, in order to increment its clock and inform the other LPs about an incoming blocking. Next time the simulator ask for a message, no one will be delivered, unless a new arrival modifies the acceptance horizon, so the simulator will block. The input process activates the flag *blocked_simulator*.

If a new message arrives, the corresponding linkclock is advanced and, if it is not a null message, it is inserted in its queue. Null messages don't need to be stored, because their only function is the advance they produce in the linkclocks. A linkclock advance (due to a null or a useful message) might increase the message-acceptance horizon, so it may allow an awaiting message to be consumed. For this reason, each time a new message is received the function check_blocked_simulator() is run, which eventually may unblock a blocked simulator process, and allow the consumption of a message. This function is much like deliver_message(). When the recently calculated horizon surpasses the local clock, the simulator is awaked by means of a null message. This way no useful message is consumed, but at least the clock is advanced and this advance can be communicated to other LPs by means of null messages.

3.3.3 Output process

This process has to accept messages from the simulator and send them to the other LPs. The routing effort is minimum, because all the messages are labelled with a *port number* that clearly states which channel the messages have to be sent through.

4 Influence of the basic parameters on the performance of the parallel simulator

In this section we will study, using the data obtained from a set of experiments, several parameters that can improve its performance, measured as the achieved *speedup*: how faster is the simulator comparing with a similar sequential version running the same model. These parameters are the load, the message length, the lookahead and the grain size.

To graphically see what it is being described, four graphs at the end of the section show the results obtained when simulating a network of 16x16 routers over 16 transputers during 4,000 cycles with message lengths 4 and 32. Two of the grahps represent execution times, measured in seconds, while the other two represent the achieved speedups, when comparing with the sequential simulator. In each figure, results with different values of load, grain size and lookahead are shown. The load is expressed as a percentage of the load that saturates the network with random traffic.

```
process simulator:

repeat {
        send(pet, " ");
        receive(sig, m);
        clock = m.timestamp;
        if (m.type == NULL) send_nulls();
        else consume(m);
        if (clock > end_of_simulation)
                build_and_send_statistics();
}
```

The simulator process runs concurrently with the input and output processes. The consume() function first determines the type of the message and then proceeds simulating its effect in the (simulated) routing network. This is done in three phases: first, the status of the appropriate router is examined, then this status is modified and, finally, events for the same or other routers are scheduled, if needed. If the destination router is being simulated in the same LP, then the event is internal, so a message is sent through *loc*. Otherwise, it has to be sent to another LP, so a message is sent through *s2o* for the output process to manage it.

The loop never stops. When the clock reaches the *end-of-simulation* value, the simulator collects a block of statistics, that is sent to the output process in order to reach the monitor. This operation is done only once, although the simulator goes on working.

A conservative simulator must block when no message is able to be consumed. This happens when no stored message has a timestamp below the message-acceptance horizon. When this situation arises, the input process sends a null message to the simulator process, instead of a useful message. This means that, if no new messages are received, the next time an event to consume is requested no one will be given—so the simulator would block. The simulator, as always, updates its clock, and then sends null messages to its four neighbours. The timestamp of those null messages is calculated as the local clock plus one, unless an special effort is devoted to exploit the lookahead of the simulated model. To optimize the number of null messages, a null message is not sent if the linkclock of the corresponding channel is not incremented.

Once the null messages have been sent, the simulator process blocks waiting for an input from *sig*. Only the reception of new messages from other LPs would be able to wake-up this process, by means of an increment of the message-acceptance horizon of the LP. This is a task for the input process.

3.3.2 Input process

As mentioned before, the input process manages all the messages that will be consumed in a LP. It maintains four input queues, plus a local queue. Requests of messages to be consumed are received from *pet*, messages for the local queue are received from *loc* and messages from the neighbouring LPs are received from the four input channels.

```
process input:

in = wait_for_input();
if (in == pet) deliver_message();
else {
        receive(in, m);
        if (in == loc) insert_local_queue(m);
        else if (is_external_channel(in)) {
                insert_input_queue(in, m);
                check_blocked_simulator();
        }
}
```

3.3 The Logical Processes (LPs)

The set of LPs are the core of the simulator. LPs follow the conservative algorithm by Chandy-Misra-Bryant, using null messages to avoid deadlocks [4]. The algorithms are derived from those described in [14].

Internally, an LP is composed of three processes (Figure 6):

- An *input* process, that manages the input queues and the internal event queue of the LP. It receives the messages (events) from the neighbouring LPs and inserts them in the appropriate queue, updating important information as the linkclocks or the message-acceptance horizon[1].
- A *simulator* process, that consumes the events. It interacts with the input process using the channel *pet* to demand messages and receiving them through *sig*. When an event is consumed, new events might be scheduled. Those that will be consumed in this LP are sent to the input process using the channel *loc*. Those for other LPs are sent to the output process through *s2o*.
- An *output* process, that manages the messages to be sent to other LPs or to the monitor. It implements minimum routing mechanisms for the case of statistics blocks.

When the simulator reaches the *end-of-simulation* time, a block with statistics is generated and sent to the output process. An input process may receive statistics blocks from other LPs, that the output process forward to the monitor. The communication between these two processes is done by means of a shared data structure.

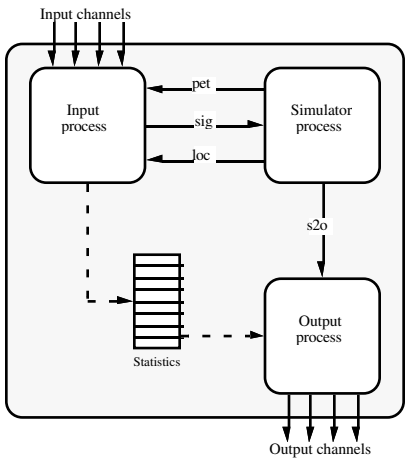


Figure 6. Structure of a logical process.

The main characteristic of the process structure shown in the figure is that already stored messages are processed while messages from the external world are accepted and managed as soon as possible. This way, processes are blocked by communication the minimum possible time.

Next, we will describe how is the implementation of the main elements of a LP: the input, simulator and output processes. We will pay special attention to the input process, that determines the *conservative* character of the simulation. A C-like language is used to express the algorithms.

3.3.1 Simulator process

The basic scheme of the simulator process is as follows:

[1] The linkclock of a channel is the value of the timestamp of the last message received through the channel. The message-acceptance horizon of a LP is the minimum among the linkclocks.

blocks) received from the east/west has to be sent to the west/east. This way the workers will not notice its presence in the network. The structure of the monitor is shown in Figure 4. When all LPs have sent the statistics block, the monitor signals the end of the simulation.

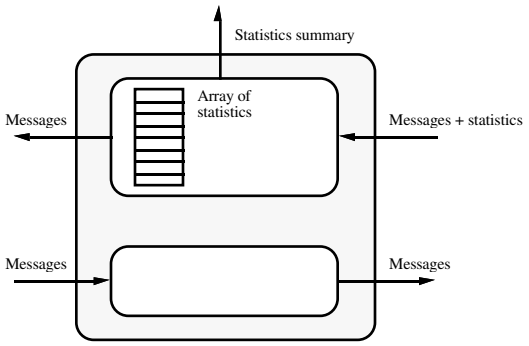


Figure 4. Structure of the monitor.

3.2 The multiplexers

A multiplexer has to provide simultaneous communication between pairs of neighbouring LPs that have been placed in different (but adjacent) transputers. Its structure is shown in Figure 5. As communication is always between immediate neighbours, a more general software router is not needed.

As shown in Figure 3, only peripheral LPs are connected to a multiplexer. Two communicating multiplexers maintain a simple protocol, in such a way that messages from different pairs of LPs are never mixed. The internal channels connected to a multiplexer are numbered from 0 to N-1. When a message m is received from the internal channel x, a message (m, x) is sent to the external link. When a message (m, x) is received from the external link, m is sent to the internal channel x.

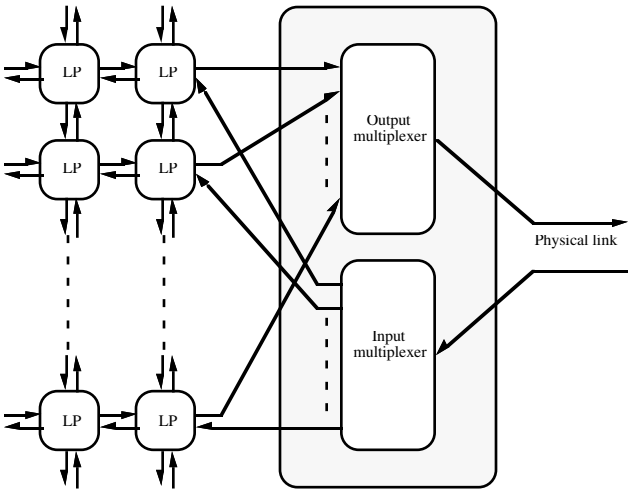


Figure 5. Structure of a multiplexer.

3 General structure of the simulators

We have used a network of transputers [7] in a Parsys SN1000 reconfigurable multicomputer. The transputers are configured in a toroidal network, with a monitor processor inserted in one of the wraparound links [8]. Figure 2 shows a configuration for a 2x2 network of transputers. Another transputer acts as a host, providing I/O capabilities.

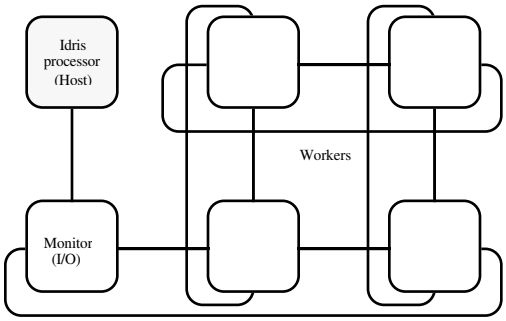


Figure 2. Configuration of a network of transputers for the simulators.

The number of *logical processes* (*LP*s) [4] assigned to a worker transputer depends on the grain size of the simulation. The general structure is shown in Figure 3. The LPs in a transputer communicate by means of internal channels. If the neighbouring LP is in another transputer, external channels are needed. As several pairs of LPs need to share an external channel, communication multiplexers are used.

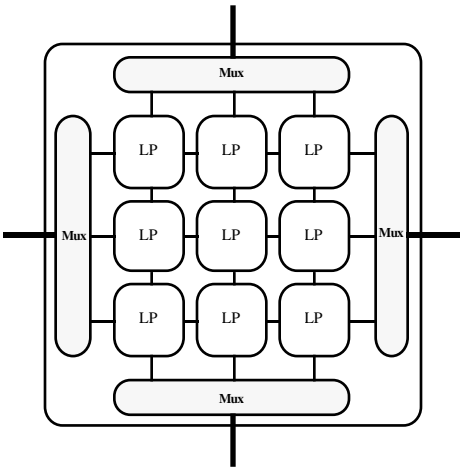


Figure 3. Configuration of each worker transputer.

If maximum grain size is used (only one LP per transputer), multiplexers are not needed—so they are not used. Each LP is a program, linked and configured separately. Internally, the LPs are composed of three program-created processes. The multiplexers as well as the monitor also exhibit internal concurrency. Next, we describe these programs.

3.1 The monitor

The monitor, placed in a separate transputer outside the workers' network, collects statistics, summarizes them and sends the final result to the host. Additionally, due to its position in the network, it has to act as a *bypass*: every message (but the statistics

All the messages have the same length. The flow-control technique is *cut-through*; no other alternative has been considered. The routing is oblivious, in order of dimension, but other strategies could have been implemented easily. Only random communication pattern have been used. The considered topologies are 2-dimensional tori and meshes. In the case of the torus, a deadlock-avoidance strategy based on restricting the injection of new messages and the change of dimension is used [1].

2.2 The simulators

We have developed two event-driven simulators: sequential [2] and parallel conservative [3, 4]. An optimistic parallel simulator [9] in under development. We tried to make the two simulators as similar as possible, to be fair when comparing the performance.

The parameters of the model and of the simulator are described in Table 1 and Table 2 respectively.

Parameter	Meaning	Values used in the experiments
Topology	Topology of the simulated routing network	Torus
Size	Size of the network (always a square)	16x16, 32x32
Message length	Size of the messages managed by the routers	4, 32
Load	Load of the network of routers, expressed as a percentage of the bandwidth of the network bisection using random traffic	5, 10, 20, 30, 40, 50, 60, 70, 80, 90

Table 1. Parameters of the model

Parameter	Meaning	Values used in the experiments
Cycles	Simulated amount of time	4000, 40000
Processors	Number of transputers used	1 (seq.), 4, 16 (par.)
Grain size	Number of routers assigned to each logical process of the parallel simulator.	Maximum, minimum, intermediate
Lookahead	Whether or not special effort is done to exploit lookahead from the model.	With, without

Table 2. Parameters of the simulator

The last three parameters of the simulator are only meaningful for the parallel simulator. The exact meaning of *grain size* and *lookahead* and their influence in the execution time of the simulator are explained later.

The output of the simulation is a compilation of statistics that allows us to assess the behaviour of the model, and also validate the parallel simulator when comparing with previously obtained results using sequential simulators. For our purposes, the main measurement is the *execution time* of the simulators, which allows to compute the achieved *speedup* when using the parallel simulator, that is, how faster is the parallel implementation compared with the sequential one. Additionally, other measurements such as the number of null messages and the number of useful messages are taken.

Current advances on VLSI design result in faster machines in the market every year. Additionally, parallel computers are also available with a reasonable prize. We want our simulators to go faster, taking advantage of these architectural progress, but we have to face two problems:

- *Effective* development environments for parallel systems, including methodologies to produce good parallel software, have not been developed yet.
- In particular, the most used algorithms for discrete-event simulation are intrinsically sequential. New algorithms have to be developed.

Several alternatives for parallelizing discrete-event simulations have been proposed and tested in the literature (quite complete surveys can be found in [6, 13]). In most of the cases, shared memory parallel computers have been used in the experiments [5, 10, 14]. We have some experience running simulations in a distributed memory parallel system [11, 12]. Many of the experiments, including ours, use simplistic models to evaluate the behaviour of the system. We now report results with a realistic model, briefly described in Section 2. As detailed data about this model can be found in other works by our group [1], this paper will be centered on the parallel simulator.

The paper is organized as follows. First, a brief description of the model under study and the simulators being used is done. Then, we make an in-depth description of the parallel simulation, followed by a detailed explanation of the algorithms in use. After that, an analysis is done about the effect that some parameters of the model and of the simulator have on the performance of the simulation. Finally, some conclusions are drawn out.

2 Description of the model and the experiments

2.1 The model

We have been running simulations of message routing networks, designed as the communication backbone for building a multicomputer system. Each router (Figure 1) has 4 input ports and 4 output ports. A waiting queue is placed at each output port, for those messages that have been routed and are awaiting the output port to be freed. Additionally, an injection queue is used when a locally generated message can not be inserted in the appropriate output queue. A more detailed description of the model can be found in [1].

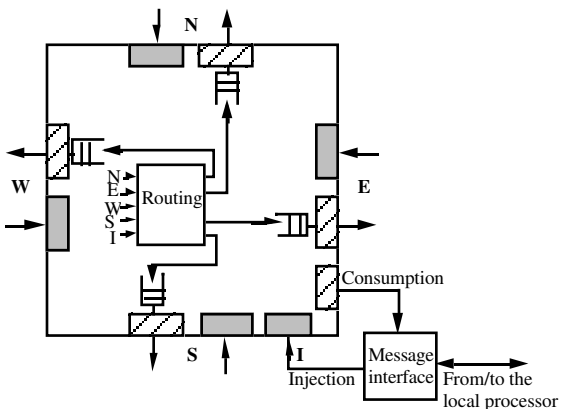


Figure 1. Model of message router.

Parallel simulation of message routing networks

Keywords

Discrete-event simulation, conservative parallel simulation, message routers, interconnection networks.

Abstract

A realistic model of a message router designed to be used in a distributed memory parallel computer is presented. The behaviour of this model is analyzed using two discrete-event simulators: a classical, sequential one and a parallel one that uses a *conservative* synchronization strategy. The use of parallel simulators allows us to run faster simulations, if a parallel computer is available. Different factors that improve the performance of the parallel simulation are discussed, focusing in the model under study and the available computer: a network of transputers. These factors are the *load* of the model being simulated, the *grain size* of the simulator and the simulator ability to exploit the *lookahead* property of the model.

1 Introduction

Our research group is interested in the analysis and design of message routers to be used in distributed memory multicomputer systems. The effectiveness of our proposals has, of course, to be demonstrated. Three possibilities are commonly considered: analytical models, actual implementations and computer simulation.

The first approach can be viewed as the ideal one: it is cheap and fast. The problem is that, in most cases, the analytical model has to be simplified in order to be able to tackle the problem. Those simplifications may hide some important characteristics of the real model, thus obtaining only approximate results. The second approach offers the highest accuracy: we work with the *actual* implementation of our proposals. The problem here is that building prototypes may be extremely expensive, especially when several versions are needed to reach the final solution.

For these reasons, computer simulation is *essential* for us, because it is a very flexible tool to evaluate our proposals, and much cheaper than making an actual hardware implementation and testing it. Unfortunately, everybody who works with computer simulation knows that this is a very CPU consuming activity. Faster CPUs are *always* welcome, but very fast is never fast enough: a bigger problem or a more detailed view of an existing one is always awaiting.

All necessary clearances have been obtained for the publication of this paper. If accepted, the authors will prepare the final camera-ready manuscript in time for inclusion in the proceedings, and will personally present the paper at the workshop.