ECE Technical Reports

Electrical and Computer Engineering

12-1-1995

An Evaluation of Implementations of the CMB Parallel Simulation Algorithm on Distributed Memory Multicomputers

José Miguel
Purdue University School of Electrical and Computer Engineering

Agustin Arruabarrena
UPV/EHU Dep. de Arquitectura y Tecnologia de Cornputadores

Ramon Beivide
Universidad de Cantabria Departamento de Electr6nica

José A..B. Fortes
Purdue University School of Electrical and Computer Engineering

Miguel, José; Arruabarrena, Agustin; Beivide, Ramon; and Fortes, José A..B., "An Evaluation of Implementations of the CMB Parallel Simulation Algorithm on Distributed Memory Multicomputers" (1995). *ECE Technical Reports.* Paper 160.
http://docs.lib.purdue.edu/ecetr/160

AN EVALUATION OF IMPLEMENTATIONS OF THE CMB PARALLEL SIMULATION ALGORITHM ON DISTRIBUTED MEMORY MULTICOMPUTERS

JOSÉ MIGUEL
AGUSTÍN ARRUABARRENA
RAMÓN BEIVIDE
JOSÉ A. B. FORTES

TR-ECE 95-28
DECEMBER 1995

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

An Evaluation of Implementations of the CMB Parallel Simulation Algorithm on Distributed Memory Multicomputers

José Miguel[1,2], Agustín Arruabarrena[2], Ramón Beivide[3] and José A.B. Fortes[1]

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

[2]Dep. de Arquitectura y
Tecnologia de Cornputadores
UPV/EHU
Apdo. 649
20080 San Sebastián
Spain

[3]Departamento de Electrónica
Universidad de Cantabria
Av. de los Castros s/n
39005 Santander
Spain

acpmialj@si.ehu.es, acparfra@si.ehu.es, beivide@ccucvx.unican.es, fortes@ecn.purdue.edu

This research has been done under the partial support of the CICYT, Spain, under contract TIC95-0378, and the National Science Foundation, USA, under grants MIP-9500673 and CDA-9015696. The first author is currently a Visiting Scholar at the School of Electrical and Computer Engineering, Purdue University.

Table of contents

Abstract

A model of a message-passing network is used to analyze the behavior of three implementations of the Chandy-Misra-Bryant parallel simulation algorithm. The characteristics of the model, the organization of the logical processes that constitute the simulator and the characteristics of the host parallel computer have a definite influence on the achieved performance, measured in terms of speedup. Large, loaded models help CMB to synchronize with a minimum overhead, efficiently exploiting the available parallelism. Mapping several LPs onto each processor achieves a better use of the available processing power, because while a LP is blocked (synchronizing) others can use the CPU. However, it is not convenient to map too many LPs onto each processor because the synchronization cost would be too high. The communication demands of CMB reduce its efficiency in environments where the cost of passing messages is too high: the performance of CMB running on a network of workstations is quite poor; in contrast, good speedups can be achieved using commercial multicomputers.

Keywords

Parallel discrete event simulation, conservative synchronization, multicomputer programming, performance evaluation, message passing networks.

Introduction

During the last years a substantial effort has been devoted to the parallel implementation of discrete event simulators. The objectives included. (1) to exploit the parallelism available in current multicomputers and multiprocessors and, mainly, (2) to accelerate simulation runs.

For some simulation studies it is necessary to run the simulator many times, in order to study the influence of a certain set of parameters on the system under study. In these cases, the most convenient way of accelerating the study is simply to run as many simulations as processors are available, each one with a different set of input parameters. This technique is called ***replication.*** The efficiency achieved is very good, because the simulations are completely independent and, therefore, communication and synchronization among the processors are not needed.

Unfortunately, it is not always possible to replicate the simulator. For some studies it is necessary to have the results of one simulation before starting the next one; this is the case when the aim is to ***tune*** a set of parameters. It may also happen that the memory available in each processor is not large enough to keep a complete copy of the simulator. These limitations of the replication approach justify the need for ways to parallelize a ***single*** simulation run.

Sometimes replication is feasible, because each one of the processors available to perform the study is powerful enough to run an individual simulation, but it happens that there are more processors than experiments to perform. Under these circumstances, replication would not fully utilize the available resources; it would be more cost-efficient to use many processors for each experiment, combining replication with other forms of parallel simulation.

The most promising set of parallel simulation techniques use the ***model decomposition*** principle: the system being simulated is decomposed into several sub-systems, and each sub-system is assigned to a logical process (LP). The collection of LPs can run concurrently, each one simulating its part of the whole. However, in order to maintain the causal relationships among the events during the simulation, a synchronization mechanism is needed. Three broad families of algorithms based on model decomposition can be found in the literature. They differ in the way LPs synchronize [FT94]:

- Parallel simulation may be ***synchronous.*** This means that all the LPs which form the simulator share the same vision of time, as if they had a global clock.

Events are simulated in the same order as in a sequential simulator. The only events that are simulated in parallel are the ones that are scheduled for the same time.

- Parallel simulation may also be ***asynchronous.*** In this case each LP has its own, local view of time. In order to perform a simulation globally correct, each LP needs to obey the following rule: execute all the incoming events in non-decreasing timestamp order. This rule is not easy to follow because, after executing a sequence of events, a new one might be triggered by another LP, with a timestamp smaller than that of the last executed event, thus impeding the receiver LP from obeying the stated rule.
  - ***A conservative*** simulator never allows these situations to happen. To do so, LPs block before executing events, until it is totally safe to proceed.
  - ***An optimistic*** simulator allows erroneous situations to arise, but those are detected and a ***rollback*** is performed, i.e. a jump back to an error-free point in the (simulated) past.

This work focuses on one particular asynchronous, conservative algorithm, known as CMB (Chandy-Misra-Bryant) [Brya77, CM79]. One problem of conservative algorithms is that the blocking synchronization mechanism can lead to deadlocks which prevent the simulator from advancing. For this reason, CMB includes a deadlock avoidance mechanism based on the interchange of ***null messages.*** CMB is described in detail in [Misr86, Wagn89].

The motivation behind our interest in parallel simulation algorithms is twofold. First, we use simulation to evaluate our architectural proposals for the design of new massively parallel computers, and we would like to have a fast tool to perform these simulations. Second, we would like to broaden the spectrum of applications that can be efficiently executed on current multicomputers (i.e., to perform something more than "number crunching").

For this study we have had access to three parallel computing systems. All of them use message passing for communication and synchronization, although with different features. These systems are:

- A transputer-based Supernode [Inmo89, Pars89], with 34 processing elements. The programming environment is the Inmos C Toolset [Inmo90].

- **An** Intel Paragon [Inte93], with 140 processing elements (i860). The NX library provides an comprehensive set of functions to develop parallel programs.
- A small network of 4 Sun SPARCstation 5, connected via an Ethernet local area network. A MPI library [MPI94] provides the support for programming parallel applications.

The objective of this study is to evaluate the influence that the following factors have on the performance of CMB running on a distributed memory parallel computer:

The parameters of the model being simulated.
The architectural characteristics of the host multicomputer.
- The organization of the simulator.

The rest of the paper is organized as follows. We start in Section **2** with a description of the model used in the simulation experiments: a network of message routers. In Section 3 three implementations of CMB are described: one for each one of the available parallel computing environments. Section 4 presents the results obtained after running a collection of experiments using the three CMB implementations. Finally, the conclusions of this work are summarized in Section 5.

2 The model under study

In this section we describe the model used to perform an exhaustive study of our three implementations of CMB. We selected this particular model because its behavior was already known [Arrua93] and, therefore, a reference point to confirm the correctness of the parallel simulations was available.

2.1 Description of the model

We study a network of message routers designed to be used as the communication infrastructure of a multicomputer system. Similar models have been identified as suitable candidates for parallel simulation; see for example [BH95, KY91, LPD95].

Each node of the network is composed of a processor and a router joined by a message interface. Processors are the source as well as the final destination of messages. Routers actually move messages from source to destination.. It is assumed that message length is fixed; it is measured in flits (flow control digits, [Dall86]). Each message has a 1-flit header, which contains the necessary information to make routing decisions. Figure 1 shows a sketch of a message router. Its main components are:

- 4 input ports, used to receive messages from the neighboring routers, through the corresponding input links. Each port is capable of storing one flit.

  4 output ports, used to send messages to the neighbors through the corresponding output links. The storage capacity is also one flit. A FIFO transit *queue* is associated with each output port, that stores messages temporarily when the corresponding link is busy. Each queue has 10 buffers, each one with capacity for a full message.
- An injection port, used to receive messages from the local processor, and a consumption port, used to send messages to the local processor. These ports connect to the message interface. A small FIFO injection *queue* inside the message interface has enough space to temporarily store up to 4 local messages, when the corresponding output queues are full.
- A routing automaton, which decides through which output port a message will be sent.

The network of routers works synchronously: in 1 cycle, a flit is moved from port to port, from queue to port, or from port to queue.

In the literature about message passing networks, many alternatives for topology, message flow-control and routing strategy can be found. In order to restrict the number of experiments to perform, we have decided to simulate only torus networks with cut-through flow-control and oblivious routing. The reasons behind this choice can be found in [Arru93, ABIM93].

It is assumed that processors immediately consume received messages, so they never force a message to stay in a router wasting resources. Processors generate messages following a given traffic pattern. The patterns most commonly used are random, hot-spot, local traffic and several permutations (perfect shuffle, bit reversal, matrix transpose, etc.). In this work we have only considered the random pattern, i.e., each node can generate messages for any other node in the network, with equal probability. When a new message is generated at a node and it cannot be injected in the routing network because the corresponding output port, the transit queue and the injection queue are full, the message is rejected (i.e., it is lost); if this situation arises, the network is saturated.

There are three parameters of the model whose values can be changed to assess their influence on the performance of the simulator. These are:

- Network size (D). We consider square torus networks of $D \times D$ nodes.
  Message length (M), measured in flits.
  Network load (L), measured as a percentage of the bandwidth of the network bisection, for a random traffic pattern.

These three parameters are needed to compute the time interval between the generation of two consecutive messages at a given node. The length of this interval is exponentially distributed, with a mean directly proportional to D and M, and inversely proportional to L. The actual expression for the mean is $(12.5 \times D \times M)/L$.

26 Types of events

Once we have the general description of the model, it is essential to determine the data structures that will represent the elements of a router, and the events that would be able to modify those elements. That is, we need to express our model in a way that can be simulated by an event-driven simulator. The definition should also be independent of the simulators used. However, the class of parallel simulators under consideration (based on the distribution of simulation tasks among a set of logical processes), and the available computing systems (distributed memory multicomputers) prevents the use of any kind of shared data structure, because the LPs constituting a simulator may be distributed among different processors. The design of the set of events needs to take this restriction into account.

Each router of the simulated network is represented in the obvious way: a record with fields representing ports, queues, etc., plus some additional fields for statistics gathering.

The events that represent in the simulator the evolution of the system are as follows:

INJECTION: the local processor generates a new message for another node.

STEP: the router tries to send the header of a message from an output port to an input port in a neighboring router.

PERMISSION: the neighboring router accepts the message.

ADVANCE: after the computation of the routing function, a header flit of a message is advanced from an input port to an output port or, if busy, to an output queue inside the router.

FREE–INP: an input port has been freed, so new messages can be accepted.

CONSUMPTION: a message has reached its destination.

FREE–OUT: the last flit of a message abandons an output port.

FREE–QUEUE: the last flit of a message abandons an output queue.

In the CMB implementations all the events except STEP and PERMISSION are always internal (i.e., scheduled to be consumed in the same LP that generates them); therefore, they never need to be encapsulated into messages sent to other LPs. In contrast, these two events can be either internal or external, depending on whether the involved routers are being simulated in the same or in different LPs. We will explain the mapping of routers onto LPs in the next section.

In a sequential simulator, PERMISSION events are not needed, because it is possible to directly check the availability of space in a neighboring router simply by accessing the data structure that represents it. In fact, they are included in the parallel version precisely because global information is not available and all the interactions must be done via messages.

23 Output data

The description of the model is detailed enough to allow ample insight into the behavior of the network of routers, such as message latencies, queue sizes, number of consumed messages, etc. This information is very valuable because it has already been gathered in previous studies, and it allows a validation of the correctness of our simulations. However, as the focus of this work is more on the behavior of the simulators than on the model, no output data will be shown.

3 The simulators

In this section we present the details of three implementations of CMB, one per available parallel computer. These implementations of CMB use the description of the algorithm given in [Wagn89]. A sequential event-driven simulator that can run on any of the three parallel system has also been implemented. For a given machine, the execution times of the sequential simulator running the optimizedl version of the model (i.e., the one that uses global information) is taken as the reference point to compute the speedups of the parallel version.

All the simulators share as much code as possible, in order to be fair when making comparisons and to reduce development effort. In particular, :in all the cases an efficient set of functions based on a heap data structure have been used to manipulate event lists, following the recommendations in [CSR93].

3.1 Input parameters for the simulators

In addition to selecting the parameters of the simulated model (size D, load L and message length M), a user running the simulators has to facilitate a series of additional parameters. These are enumerated in Table 1. The first two parameters (cycles, seed) are needed for all the simulators, sequential and parallel. The number of processing elements must be given for CMB. A mapping of the simulated network of routers onto the physical network of processing elements (PEs) in the target multicomputer (or network of workstations) must be done. The number of PEs is always a square of $P \times P$ elements, where P must be a perfect divisor of D (the number of routers per dimension in the simulated network). Under this condition, the partition of the model and its mapping onto the network of PEs is simplified (a square of size *DIP* routers is simulated in each PE) and perfectly balanced (all the PEs have the same load).

When more than one router is assigned to each PE (and this is always the case for the experiments we have performed), there are several possible organizations for the simulator. A CMB simulator always consists of a collection of collaborating LPs, where each LP is a Unix process (in the Paragon or in MPI) or a transputer process (in the Supernode). Mapping the model onto the host computer requires two steps: mapping routers onto LPs, and mapping LPs onto PEs. There are two trivial possibilities:

1 Map each router onto a single LP, and then map groups of $(D/P)^2$ LPs onto each PE. We say that the grain size of the LP is minimum. A large amount of interprocess communication is needed, because all the STEP and PERMISSION events are external (i.e., they need to be sent as messages, although they do not necessarily need to go from one PE to another).

2 Map $(D/P)^2$ routers onto one LP, and then each LP onto a different PE. We say that the grain size of the LP is maximum. In this case, many of the STEP and PERMISSION events are internal and the interprocess communication is significantly reduced.

Note that other grain size alternatives are possible. For example, Figure 2, shows a case where D/P = 4. Figures 2a and 2c represents the mappings for maximum and minimum grain sizes respectively, and Figure 2b represents a. mapping for intermediate grain size. If the mappings are always square, there can exist either none, one (like in the example) or several cases of intermediate grain size.

The last parameter in Table 1 indicates whether or not the LPs try to extract lookahead from the model.[Fuji89]. To improve the performance of CMB, it is highly recommended to analyze the simulated model to determine if some lookahead can be extracted, and to tailor the simulator to use this lookahead. If this could be effectively done, timestamps of null messages would have higher values and the overall number of required null messages would be reduced, allowing a faster clock advance of the LPs.

If an LP simulates only one router (i.e., the grain size is minimum), then the behavior of the simulation is quite predictable, and lookahead can be easily extracted. Let us suppose that, at time t, a router has sent a message header through an output port; the LP can guess that no new header will be sent through the same port at least until $t+M$, where M is the message length, because messages advance a flit per cycle. The difference between the current value of the LP's clock and t+Mis the lookahead. In contrast, when the grain size is not minimum, the cost of computing lookahead is high, and the obtained values are low. In fact, in most cases the obtained value is one, a minimum that can be assumed without any computation. For this reason, we decided to ignore the lookahead ability of the model, except for minimum grain size configurations.

3.2 Supernode implementation

The Parsys Supernode SN-1000 [Pars89] is a multicomputer with Inmos transputers [Inmo89] as processing elements. Each transputer includes, in addition to a CPU and some local memory, four serial communication links of 10 Mb/sec each. In the Supernode, the transputer links are all connected to a. collection of programmable switches, in such a way that the user can specify the topology of the interconnection network.

Each transputer can house many concurrent processes, which are efficiently managed by a built-in scheduler. Two transputer processes communicate via unidirectional channels. If the communicating processes run on the same transputer the channel is internal, just a word in memory. It is also possible to map a channel onto a communication link, if the communicating processes run on neighboring transputers. A maximum of two channels can be mapped onto one link, one in each direction. Communication via shared memory is possible, for processes running on the same transputer.

In order to run a CMB simulator on the Supernode, a group of transputers is arranged to form a torus network of ***worker*** transputers with a ***monitor*** transputer inserted in one of the wrap-around links. The number of processes assigned to each worker transputer depends on the grain size of the LPs. The LPs in one transputer can be joined directly via internal channels, while the external links are needed only if two logical neighbors are mapped onto different transputers. Since several logical neighbors need to communicate through a single external link, link sharing is needed. Some ***multiplexer*** processes perform this function. Obviously, if maximum grain size is used (only one LP per transputer), multiplexers are not needed; therefore, they are not used, in order to reduce overheads. Figure 3 shows the arrangement of processes in two neighboring worker transputers.

A monitor process, placed in a separate transputer outside the network of workers, collects statistics (output data of the simulation, see $2.3) and summarizes them. Additionally, due to its location in the network, it has to act as a bypass: every message received from the east/west has to be sent to the west/east. Thus, the workers do not perceive its presence in the network.

The set of LPs constitutes the core of the simulator. All the other components (monitor, multiplexers) are only needed to build a working system. Internally, an LP is composed of three communicating processes (Figure 4):

- An input process, which manages the input queues and the internal event calendar of the LP. It receives messages (events) from the neighboring LPs, and inserts them into the appropriate input queue. It also updates important information as the channel clocks (each channel clock stores the timestamp of the last message received through the channel) and the message-acceptance horizon (the reference point that allows to determine what messages can be consumed and what others must still wait).
- A simulator process, which consumes the events. It interacts with the input process, using channel pet to request messages, which are received through channel sig. When an event is consumed, new events might be scheduled. Those that will be consumed in the same LP are sent to the input process using channel *loc*. Events for other LPs are sent to the output process through *s2o*. When the simulator reaches the end–of–simulation time, a block with statistics is generated and sent to the output process.
- An output process, which manages messages to be sent to other LPs or to the monitor. It implements minimal routing mechanisms, for the rnanagement of statistics blocks. An input process may receive statistics blocks from other LPs; the output process forwards these blocks towards the monitor. These two processes communicate via a shared queue.

The division of the LP into three different sub-processes allows the decoupling of the event consumption and message interchange activities. If the design of the LPs were monolithic, communication deadlocks could easily happen. This is due to the way communication is accomplished in the transputer: simultaneous send() and receive() operations are needed to complete a message interchange. If a monolithic LP A wanted to send a message to a busy neighbor B, A would block (wasting time) until *B* invokes the peer operation. A deadlock would immediately happen if *B* was also blocked while trying to send a message to A. More complex deadlock scenarios, involving more than two LPs, are also possible.

With the proposed design the simulator process never blocks in a send(), because the corresponding output process is always ready to respond to it. The simulator might block in a receive(), but only if no suitable event is ready to be consumed (as the CMB algorithm requires). Meanwhile, all the incoming messages can be received and stored by the input process, which is a greedy receiver. This design avoids communication deadlocks, while allowing events to be managed as early as possible.

3.2.1 Simulator process

The basic scheme of the simulator process is as follows:

```
process simulator:
repeat {
     send(pet, " ");
     receive(sig, m);
     clock = m.timestamp;
     if (m.type == WILL-BLOCK) send-nulls();
     else consume(m);
     if (clock > end-of-simulation)
          build-and-send-statisticso;
}
```

The simulator process runs concurrently with the input and output processes. The consume0 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 which, in turn, forwards it to the monitor. This operation is done only once, although the simulator goes on working. The monitor process is responsible for stopping the simulation.

A CMB LP must block when no message is able to be consumed. This happens when no stored message has a timestamp below the message-acceptance horizon. In this situation, the input process sends a WILL_BLOCK 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 be consumed is requested no one will be given— so the simulator will block. The simulator, as always, updates its clock, and then sends null messages to its four neighbors. The timestamp of those null messages is computed as the value of the local clock plus one, unless special effort is devoted to extract lookahead. A null message is not sent if it does not produce an increment of the receiver's linkclock.

Once the null messages have been managed, 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, as a consequence of the (possible) increment they produce on the message-acceptance horizon of the LP. This is a task for the input process.

3.2.2 Input process

The input process manages all the messages that will be consumed in an 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 neighboring 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();
     }
}

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, WILL-BLOCK-message);
else blocked-simulator = TRUE;
```

Function deliver–message() computes the message-acceptance horizon (the minimum among the LP's channel clocks), 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 expected behavior of a CMB LP.

When no message is ready to be consumed a WILL_BLOCK message is sent to the simulator, in order to increment its clock and allow the other LPs to advance. Next time the simulator asks for a message, no one will be delivered, which will force the simulator to 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 need not be stored, because their only purpose is the advance they produce in the linkclocks. A linkclock advance (due to a null or a useful message) may increase the message-acceptance horizon and, therefore, may allow an awaiting message to be consumed. For this reason, each time a new message is received, function check_blocked_simulator() is invoked. This function, which is similar to deliver–message(), eventually unblocks the blocked simulator process, and allows the consumption of a message. When the acceptance horizon computed by check–blocked–simulator() surpasses the local clock, the simulator is awaked by means of a WILL–BLOCK message. This way no useful message is consumed, but the clock is advanced and this advance can be communicated to other LPs.

3.2.3 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 labeled with aport number that clearly states which channel the messages have to be sent through.

3.3 Paragon implementation

The Intel Paragon [Inte93] is a multicomputer organized as a rectangular mesh of nodes. Each node has one i860 to perform computation, another i860 to manage message passing, and an interface to a high-speed interconnection network. This network, composed of custom-designed routing chips, is able to carry messages between any two nodes at up to 200 MB/sec.

Parallel programs can be developed using Intel's proprietary NX library, which provides a comprehensive set of functions to manage processes and passing messages among them. Although in theory it is possible to run many concurrent processes on each node, in practice it is not efficient to do so, because a process does not relinquish the CPU when it is awaiting to complete a communication operation and, therefore, it is not possible to overlap one process' computation with another's communication.

Porting the CMB implementation from the Supernode to the Paragon required basically to cope with two differences between the programming environments of these systems: synchronization among processes and node multiprocessing ability. The Supernode implementation exploits the efficient multiprocessing abilities of the transputer: each transputer contains several processes (LPs and multiplexers) which, in turn, are structured in several sub-processes. Two main reasons justify this design. Firstly, the synchronous nature of message passing functions requires a decoupling of message management (reception, storage, sending) and message consumption tasks, in order to avoid communication deadlocks. The division of an LP into input, simulator and output processes provides this decoupling. Secondly, it is very efficient to share a transputer among several LPs: one LP can be working while others are blocked awaiting for communication[1]. The transputer built-in scheduler relinquishes the CPU from a process as soon as it blocks for communication.

None of these considerations apply in the case of the Paragon and, for this reason, a re-design of the simulator was necessary. The main differences between the Supernode and the Paragon versions of CMB are:

- The multiplexer processes and of the bypass functions of the monitor have been removed. These simplifications have been possible because, in the Paragon, messages can be interchanged between any pair of processes, independently of their position. Routing, link sharing and other functions related to message passing are managed by a separate network of message routers (very much like the object of our study), without interfering with the activity of the processes. An application controlling process on a separate PE acts as the monitor. Its purpose is to launch the LPs and to gather statistics at the end of the simulation run.
- The input, output and simulation processes (the three sub-processes which form an LP in the transputer) have been combined into a single process. This new design is ***possible*** because of the buffered nature of the communication in the Paragon, and ***necessary*** because of the poor multiprocessing abilities of the Paragon.
- Minimum and medium grain sizes alternatives for the LPs have been eliminated, that is, only maximum grain size is considered. This means than only one LP runs on each PE, again because of the inefficient scheduling policy of the Paragon.

The algorithm of the LP is as follows:

[1] **This will be seen when analyzing the performance of the Supernode implementation.**

```
process LP:
repeat {
     h = acceptance-horizon();
     ts = minimum-timestamp0;
     if (ts ◄ h) {
          m = message_with_minimum_timestamp();
          clock = m.timestamp;
          consume(m);
          if (clock > end-of-simulation) build-and-send-statisticso;
     3
     else {
          clock = h;
          send-nulls();
          m = receive(); /* Blocking */
          insert_local_calendar(m);
     }
}
```

Note that messages from other LPs are received only when the LP needs to increment its channel clocks in order to advance. While an LP is busy, messages can arrive from the neighbors, and they are stored in system buffers until the LP decides to actually receive them. This buffering provides enough decoupling to allow a collection of LPs to progress without communication deadlock.

Except for changes in the communication functions, the rest of the simulator code is the same as the Supernode's.

3.4 MPI Implementation

The third CMB implementation was designed to run on a small network of 4 Sun SPARCstation 5, connected via an Ethernet local area network. An MPI (Message Passing Interface, [MPI94]) library provides the support for programming parallel applications.

The MPI implementation of CMB is very similar to that for the Paragon. The only relevant difference is the substitution of the NX functions by their MPI counterparts. The change was straightforward, because these two libraries (NX and MPI) are semantically very close; furthermore, only a limited number of communication functions were used in the programs.

4 Experiments

In this section we present the results obtained after running a collection of experiments using the three implementations of CMB. We start with the Supernode version, then the Paragon version and, finally, the MPI version running on a network of workstations. We made many experiments with different model parameters (network size, message length and load) and simulator parameters (number of PEs, grain size of the LPs, use of lookahead information). Most of the times identical sets of parameters have been used with the three simulators. However, some experiments have been specifically designed to stress some particular characteristics of a given simulator.

The obtained results are displayed as a collection of speedup curves. The speedups have been calculated after running an optimized sequential, event-driven simulator with the same set of model parameters.

4.1 Description and results on the Supernode

In this section we present the results of seven experiments performed with the Supernode version of CMB. An analysis of these results is done in the next section. Tables 2, 3, 4 and 5 summarize the parameters used in the experiments. The first three parameters of Table 2 (network size, message length and load')are related to the model, while the next three (number of processing elements, grain size and use of lookahead information) are related to organization of the simulator. All the experiments ran for 4000 simulated cycles. The obtained results are plotted in Figures 5 (experiments 1S to 6S) and 6 (experiment 7S).

Experiments 1S—4S deal with a relatively small model of 16x16 routers, with message length 4 or 32, and running in 4 or 16 transputers. With these experiments we wanted to test the following hypothesis:

1 Results are better with 4-flit messages than with 32-flit messages. For the same network load, 4-flit messages means at least 8 times more simulation events than 32-flit messages. Execution times should be longer for both the sequential and the CMB simulators, but the latter will have more opportunities to self-synchronize without null messages, therefore reducing the synchronization overhead. This should be confirmed by comparing 1S with 3S and 25 with 4S.

2 Performance improves when the load increases. Again, the higher the load the larger the number of events managed by the simulator, therefore allowing the system to self-synchronize. This should be confirmed by all the experiments.

3 The simulator scales with the number of processors, i.e., the speedup obtained with 16 transputers is better than the speedup obtained with 4 transputers. This should be confirmed by comparing **1S** with **2S**, and **3S** with **4S**. Another experiment, **7S**, considers this aspect in more detail.

Additionally, we wanted to test the impact that the different grain sizes have on the performance, as well as the effect of extracting lookahead information from the model.

Experiments **5S** and **6S** work with a model four times larger than that used in the previous set. We wanted to confirm that the simulator performance improves with larger workloads. More work can be assigned to an LP by mapping onto it a larger number of routers.

Experiment **7S** is a scalability test. The same model is run on 4, 9, 16 and 25 transputers. As we will see, the simulator performance strongly depends on the grain size of the LPs. Sometimes a wide range of grain sizes are possible, while in others cases there are only two: maximum and minimum. In Figure 6 we only show the curves for the grain size alternative which gives better results, which is always an intermediate value. An exception is the 25-transputer case, where! the model has been slightly increased to allow a balanced partitioning, and no intermediate grain size is possible—so the maximum has been used.

46 Analysis of the results on the Supernode

After showing the experimental results, we proceed to analyze the effect that each parameter of the model or of the simulator has on the execution time. When convenient, several parameters are grouped and studied together.

4.2.1 Network size, message length and load

It is clear, from any of the speedup curves, that the following parameters of the model have a significant impact on the execution time of the simulation: load, message length and network size. The best situation arises with large and highly loaded systems that interchange short messages. Interestingly enough, this is the worst possible scenario for the sequential simulator. When the model has the opposite characteristics, the performance is not impressive, but the actual execution time is not very long.

The reason for this behavior can be found in the way LPs synchronize in a CMB simulator. All the mentioned parameters affect the number of "useful" messages (i.e., non-null messages) managed by the simulator. An increment in the number of these messages means that the LPs have more opportunities to synchronize, while doing useful computation. Null messages are needed less often, because LPs do not block frequently. We can say that there is a high degree of "natural" synchronization. When there are only a few useful messages to process, LPs block often, and null messages are needed to maintain the LPs' clocks updated. Consequently, LPs spend most of their (real) time blocked or processing null messages (i.e., synchronizing, instead of making progress).

4.2.2 Grain size

Looking at the results of the first four experiments and comparing maximum vs. minimum grain size, it is evident that coarse grain simulation is more effective than fine grain simulation for low and medium loads. This because a small number of LPs synchronizing with null messages results in lower overhead.

If now we compare the results for minimum and maximum grain size with those for intermediate grain size, it is clear that the latter are the best for intermediate and high loads. Only for very low loaded systems maximum grain size gives, sometimes, better performance than intermediate values. An analyis of the time used for synchronization in the simulator can explain this behavior:

- With maximum grain size, an LP can advance autonomously most of the time, due to the large number of interactions between the routers assigned to it. Nevertheless, sometimes the LP has to co-ordinate with the others, which causes the LP to send null messages and then block. This behavior is extremely inefficient because the LP is the only user of the transputer and, if it blocks, the freed CPU power is wasted.
- With minimum grain size, there is no problem if an LP blocks: plenty of others are awaiting to use the CPU. Here the problem is that, the larger the number of LPs, the larger the number of null messages needed to keep the system synchronized. Furthermore, since fine-grain LPs have very little work to do, they block very often—making things even worse.

- With intermediate values of grain size it is possible to find a balance: there are few LPs, each one with enough work to do, so null messages are not needed very often. In addition to that, as several LPs share a transputer, the probability of wasting CPU time decreases.

In conclusion, we can state that for the transputer, where communication operations are blocking and context switches are quite fast, coarse grain simulations are faster than fine-grain simulations. However, it is even more efficient to use intermediate grain sizes, with several LPs sharing a processor, in order to avoid idle processors and make the maximum usage of the available CPU power.

4.2.3 Lookahead

From the description of how lookahead is computed (see end of §3.1), it should be clear that an LP might obtain large lookahead values when (1) messages are long and (2) the LP has recently interacted with its neighbors.

The first situation can be clearly observed by comparing experiments **1S** and **3S** (Figure 5). For 4-flit messages curves 4min and 4minL (minimum grain size, with and without using lookahead information, respectively) give nearly the same results; hovever, for 32-flit messages 4minL is clearly better than 4min. The second situation can be seen in experiments **3S** and **4S**: in highly loaded systems, the LPs interact often and, if they have to block, the computation of the timestamp of the null messages can take advantage of the knowledge of which ports have recently sent header flits. At any rate, as lookahead can only be effectively exploited for minimum grain size, and this is not the best situation for this implementation of CMB, we do not see any advantage of using the lookahead information provided by this particular model.

4.2.4 Number of processing elements

From the complete set of experiments, and especially from experiment **7S**, it can be seen that the performance of the parallel simulator scales well with the number of processors, although not linearly. There are, though, some factors that can help to understand why the speedup curve in Figure 6 is not perfectly linear:

- Our way of distributing the workload among processors and processes, using squares, does not always allow to find the optimum grain size value. For example, the 25x25 network of experiment **7S**, when simulated over 5x5 processors, only allows maximum and minimum grain sizes; each processor must simulate 5x5 routers, and 5 is a prime number, which means that no intermediate alternatives are possible.

- The processors used in our experiments were not all identical. In most cases, T805 transputers running at 30 MHz were used but, as only 14 processors of this kind were available, in those experiments involving 16 or 25 processors some 20 MHz T800 were used. Those processors are a bottleneck, imposing their rate of execution on the others. For this reason the resulting speedups are not as good as they should.
- Since the problem size does not increase with the number of resources, the processors are not fully utilized. When the number of processors (and of LPs) increases, the probability of having a blocked LP increases too, so some processing power is wasted and more null messages are needed. To confirm this assertion, we have represented in Figure 7 the ratio of the number of useful messages to total number of messages managed by the simulator in experiment **7S**. Note how this ratio reduces when the number of processors is increased.

Note also that the chosen set of model parameters (D, M and L) for experiment **7S** constitutes neither the best possible scenario for the parallel simulator, nor the worst for the sequential one. With a higher load the results would be more favorable to the parallel simulator.

4.3 Description and results on the Paragon

The experiments performed with the Paragon implementation of CMB are summarized in Table 6. The corresponding results are shown in Figures 8 and 9. Experiments **1P** through 7P are basically the same as those run on the Supernode (**1S—7S**). In **7P**, it has been possible to run the 24x24 model using up to 64 Paragon PEs. Experiment **8P** is another scalability test specially re-designed to make a better use of the larger number of PEs available in the Paragon: a large model of 90x90 routers is simulated, using 4, 9, 25, 36, 81 and 100 processing elements.

As previously mentioned, the Paragon implementation of CMB only allows one mapping of routers onto LPs: maximum grain size. This means that only one LP (which simulates a group of routers) is assigned to each Paragon PE. As a consequence of this restriction, no effort is done to exploit the lookahead potential of the model.

For experiments 1P through 7P simulations ran for 4000 cycles. For experiment 8P, this number was reduced to 1000. It should be noticed that some of these experiments are not well dimensioned for the Paragon. Execution times are very short (less than 3 seconds in some cases) and, therefore, a minimum variation in the time measurement provided by the system can result in a significant variation in speedup, so trends are more significant than actual values.

Most of the conclusions drawn from the Supernode experiments apply to the Paragon too, except for the discussions about grain size and the use of lookahead information, which cannot be applied here. In summary, the best performance is obtained with large, highly loaded models managing short messages. This scenario is a challenge for sequential simulators, while it allows CMB to minimize the synchronization effort.

Speedups are better for the Paragon than they are for the Supernode. For example, for experiments 5S and 5P (which are identical for both machines) the peak speedup on the Supernode is slightly less than 8, while on the Paragon it is very close to 12. Scalability tests 7S and 7P are also equal for 4, 9 and 16 processors; in the latter case, the Supernode implementation reaches a speedup of 4.55, while the Paragon reaches 10.

The reason of this performance difference can be found in the architectural dissimilarities between the two machines and in the differences in the LP's design. In the Paragon, all message manipulations are done by a network of hardware message routers plus a second processor in each node (a message co-processor). This frees the computing node of most of the overheads of message handling. In the Supernode, however, there is neither routing hardware, nor message co-processors, and, therefore, each transputer has to divide its time between computation and message handling.

Another architectural difference comes from the fact that in the Paragon all the PEs are identical, while in the Supernode a mixture of 20 MHz and 30 MHz transputers are used in some experiments. Since the reference point (the execution time of the sequential simulator) was taken with respect to a 30 MHz transputer, the reported speedups for the Supernode are not as good as they should be.

Regarding the design of the LPs, in the Paragon case they have a monolithic structure, while in the Supernode they are divided into three sub-processes. This arrangement in the Supernode was necessary to avoid communication deadlocks. However, this does not come without an added cost. These sub-processes communicate mostly via internal channels; therefore, messages are copied from memory to memory several times in their life cycle. A message generated at a simulator process in one transputer that needs to be consumed at another simulator in a neigbouring transputer needs four internal copies (messages between processes in the same transputer) plus one external copy (message that actually traverses a link). In the case of the Paragon, due to the monolithic design of the LPs, no internal copies are necessary.

Figure 9 shows the results for experiments **7P** and **8P**, the scalability tests. Note that CMB scales fairly well: the curves are nearly straight lines. The workload is high enough to keep all the PEs busy most of the time, even for a large number of PEs. The efficiency achieved in experiment **8P** is higher than that of **7P**. For example, in the case of 36 processors the speedups are 24.3 in **8P** and 21 in 7P. The larger workload assigned to the LPs in **8P** makes them achieve higher performance. Figure 10 characterize one of the behavioral differences between these two experiments: the number of null messages needed to keep the simulator synchronized. In **8P** the proportion of null messages is very low, less than 1% for the case of 4, 9 and 25 PEs, about 3% for 81 PEs and less than 4% for 100 PEs. In contrast, this proportion rises to nearly a 40% for 7P in 64 processors.

The number of null messages is directly related to the required synchronization effort. However, null messages are not the only source of overhead. If many of these messages are being sent, this is because LPs are blocking very often, spending time awaiting for incoming messages that eventually will increase the acceptance horizon and allow the simulation to advance. Figure 11 has been obtained by running experiments **7P** and **8P** with an instrumented version of CMB that monitors the way PEs use their time. Total execution time has been divided into three components:

Tsim: time spent executing events, inserting messages in the local event list and in the input queues, and sending messages to other LPs. This is the time the LP devotes to advancing the simulation.

Trec: time spent receiving messages from other LPs. This includes receiving useful as well as null messages. As the receive() operation is blocking, this time can be considered mainly synchronization effort.

Thn: time spent performing other synchronization tasks, namely: (1) sending null messages (this includes computing whether a null message is necessary or not) and (2) computing the message acceptance horizon at each step of the simulator main loop.

From Figure 11 it is clear that if the workload assigned to an LP is low, the corresponding PE spends too much time synchronizing instead of executing events. In the case of **7P** on 64 PEs, each PE simulates 3x3 routers and the achieved proportion of effective work is 55%. In contrast, for **8P** on 100 PEs, each PE simulating 9x9 routers (9 times larger that in **7P**), CMB achieves a 75% efficiency. The 62% effective simulation time in **7P** with 36 PEs versus the 83% in **8P** with the same number of processors justifies the previously observed speedup difference.

4.5 Description and results on the network of workstations

The number of Sun workstations available for the experiment with the MPI implementation of CMB was limited to four. From the experience gained with the Supernode and the Paragon, speedups over 2 should be expected. The experiments described in Table 7 were set up to confirm this hypothesis. Note that experiment **8M** is no longer a scalability test, because of the limited number of available workstations. The other experiments are similar to their counterparts in the Supernode and the Paragon. As in the case of the Paragon, the grain size of the LPs is always maximum, and the lookahead ability of the model is not exploited. The resulting speedup curves are in Figure 12.

It is easy to observe how the performance improves when:

- Problem size is increased: compare **1M** (16×16), **5M** (32x32) and **8M** (90x90).
- Message length is reduced: compare **1M** (4 flits) with **3M** (32 flits), or **5M** with **6M**.
- The load is increased. All curves show this.

This behavior is the same observed in the other two implementations. The outstanding point is that the overall results are really poor. Since the programs are the same we used in the Paragon (except for minor details), we must find the reasons behind this poor performance in the characteristics of the computing system used in the experiments.

The main difference between the Paragon or the Supernode and the network of workstations is the way interprocess communication is performed. Even though in the three cases a message passing mechanism is used for synchronization and communication, the Supernode and the Paragon use high-speed, special purpose interconnection networks, while the workstations are connected via a general purpose Ethernet local area network, with the TCP/IP protocols over it. This means that communication in this environment is relatively slow, because:

The peak data rate of Ethernet is 10 Mb/s. In the Supernode each transputer provides 10 Mb/s per **each** of its 4 links, while in the Paragon the interface between a node and the communication network allows a. processor to send/receive information at 1400 Mb/s.

Ethernet allows all the devices connected to the network to share the available bandwidth, whether they are part of the simulation or not. In the Supernode the channels are used exclusively by the transputers that work in a simulation. In the Paragon the interconnection network is shared among all the simultaneous users, but as the network is able to move information at 1600 Mb/s (i.e., faster than the generation rate of the nodes) and the! users work in clusters which do not overlap, the sharing effect is barely noticeable.

The use of several layers of protocols (Ethernet, IP, TCP, MPI) imposes a significant overhead. The communication protocols used inside a multicomputer are much simpler; in particular, there are fewer layers. Because layering means encapsulation (i.e., addition of control information) its effects are worse for short messages than for long messages[2].

In order to compute the communication capabilities of the three parallel systems, (instead of using the raw data offered by the manufacturers) we ran a test where four processors are arranged in a logical unidirectional ring (in the case of the Transputer, the ring is also physical). The first processor in the ring sends messages of various sizes to the next one, which simply executes a store-and-forward procedure to send the received messages to the next processor in the ring. When a message arrives back to the first processor, it computes the real time that it took to complete the ring. Using this time and the message size, the data rate is computed. Message sizes varying from 1 to 2^{18} have been tested; the achieved data rates for those message sizes are plotted in Figure **13.** In all the cases it can be seen how the data rate increases with the message size, until it stabilizes at a point not far from the theoretical maximum.

[2] In this context we are speaking about real messages interchanged between processing elements, not about simulated messages.

Unfortunately, the actual messages managed by our simulators are very short: about 32 bytes. For this size, the achieved data rate is far from the maximum in the case of the Paragon and the network of workstations (see the vertical line cutting the curves in Figure 13). The Supernode, however, achieves a data rate close to its peak value even for short messages.

In conclusion, the computation to communication ratio of the network of workstations is not as balanced as in the other two systems. This would not be true if the CPUs of the workstations were proportionally slower, but this iis not the case. A comparison of the raw computing capabilities of each system has been done taking into consideration experiment 5 (5S, 5P and 5M), which has been performed for the 3 systems using the sequential as well as the parallel simulators. Table 8 summarizes the execution times of the sequential simulator running this experiment with load 90. A Sun SPARCstation 5 is slightly faster than a Paragon processing element, and about 7 times faster than a 30MHz T805 transputer (for the kind of computation we are doing). In contrast, the data rate achieved using MPI over Ethernet is more than 66 times smaller than that achieved using the Paragon interconnection network, or a set of interconnected transputers.

Our conclusion is that the communication demands of CMB (in particular, the need of a frequent interchange of short messages) make it unsuitable for this kind of parallel computing platform. In other words, CMB works specially well in fine-grain parallel computers, while a network of workstations might be used efficiently only for coarse grain problems.

5 Conclusions

In this paper we have presented, analyzed and coompared our experiences implementing and using CMB on three different multicomputing environments. The CMB simulators have been used to study a model of a message passing network designed to be used as the communication infrastructure of a multicomputer.

The characteristics of the simulated model have a definite influence on the achieved performance. In order to take advantage of CMB, a model with a high degree of internal communication is needed, which allows processes to remain synchronized without needing null messages. For the model used in this study, this happens when network size is large, load is high and messages are short. These three parameters have the strongest influence on the performance of'the simulator. It is interesting to observe that, conveniently, the best scenario for the CMB simulator is the worst for the sequential event-driven simulator: speedups are best precisely when they are most needed, and are poor only in cases where simulation runs are very short with the sequential but also with the parallel version.

If CMB is running on a set of processes statically assigned to a set of processors, it is important to use coarse grain processes, that is, to assign a significant amount of work to each process. This way, less processes are used to run the :model, and the synchronization overhead is reduced. This idea should not lead us to the extreme of assigning only one process to a processor, because if the process blocks, the processor stays idle. If the host parallel computer allows it, more than one process should be mapped onto each processor. For example, in the Supernode the best performance was achieved using intermediate grain sizes.

The knowledge of the behavior of the model may allow CMB to exploit some lookahead information, which helps maintain a good performance when the workload does not allow the simulator to self-synchronize. Unfortunately, the lookahead ability of our network of message routers is, in general, poor, particularly when an LP simulates a set of routers instead of only one (that is, when the grain size is not minimum). It is more advantageous to use intermediate or maximum grain sizes, even if that means renouncing to exploit the lookahead of'the model.

The communication demands of CMB are very strong, making it; unsuitable for environments such as a network of workstations, were communication costs are very high compared to computation costs. In contrast, the performance achieved in two commercial multicomputers, Supernode and Paragon, are reasonably good—better in the latter than in the former.

References

[ABIM93] A. Arruabarrena, R. Beivide, C. Izu and J. Miguel. "A performance evaluation of adaptive routing in bi-dimensional cut-through networks". *Parallel Processing Letters* Vol. 3 No. 4, 1993, 469—484.

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

[BH95] C. Benveniste and P. Heidelberger. *Parallel simulation ojfthe IBM SP-2 interconnection network*. IBM Research Report RC 20161 (8/15/95). To appear in the proceedings of the 1995 Winter Simulation Conference.

[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.

[CSR93] K. Chung, J. Sang and V. Rego. "A performance comparison of event calendar algorithms: an empirical approach". *Software—Practice and Experience*, Vol. 23(10), Oct. 1993, 1107—1138.

[Dall86] W.J. Dally. *A VLSI architecture for concurrent data structures*. Ph.D. dissertation, California Institute of Technology, 1986.

[FT94] A. Ferscha and S.K. Tripathi. *Parallel and distributed simulation of discrete event systems*. CS-TR-3336 Dept. of Computer Science, University of Maryland, Aug. 1994.

[Fuji89] R.M. Fujimoto. "Performance measurements of distributed simulation strategies". *Trans. of The Society for Comp. Simulation*, Vol. 6, No. 2, 1989, 89—132.

[Inmo89] *The Transputer Databook*. Inmos Databook Series, Inmos Ltd., Bristol, U.K., 1989.

[Inmo90] *ANSI C toolset*. Inmos Ltd., Bristol, U.K., 1990.

[Inte93] *Paragon user's guide*. Intel Corporation, 1993.

[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.

[LPD95] W. Liu, G. Petit and E. Dirkx. "Performance assessment of a large ATM switching network with parallel simulation tool". *Proc. 1995 Int. Conf. on Parallel Processing,* Vol. III, 142—145.

[MAB95] J. Miguel, A. Arruabarrena, R. Beivide. "Conservative parallel simulation of a message-passing network: a performance study". *Proceedings Summer Computer Simulation Conference SCSC'95*. Ottawa, Canada, July 1995,825—830.

[MABF96] J. Miguel, A.Arruabarrena, R. Beivide and J. Fortes. "An empirical evaluation of techniques for parallel discrete-event simulation of interconnection networks". *Proc. 4rd. Euromicro Workshop on Parallel and Distributed Processing PDP'96*. Braga, Portugal, Jan. 1996.

[MAIB95] J. Miguel, A.Arruabarrena, C. Izu and R. Beivide. "Parallel simulation of message routing networks". *Proc. 3rd. Euromicro Workshop on Parallel and Distributed Processing PDP'95*. San Remo, Italy, Jan. 1995.138—145.

[Misr86] J. Misra. "Distributed discrete-event simulation". *Computer Surveys,* Vol. 18, No. 1, March 1986, 39—65.

[MPI94] Message Passing Interface Forum. "MPI: a message-passing interface standard. *International Journal of Supercomputer Applications,* 8(3/4), 1994. Available at **http://www.mcs.anl.gov/mpi.**

[Pars89] *Idris and Supernode SN1000 series manuals.* Parsys Ltd..U.K., 1989

[Wagn89] D.B. Wagner. *Conservative parallel discrete-event simulation: principles and practice.* Ph.D. dissertation, Department of Computer Science and Engineering, Univ. of Washington, 1989.

Tables

Parameter	Meaning
Cycles	Duration of the simulation, in terms of cycles.
Seed	Seed for the random number generators.
Number of PEs	Number of processing elements used in the simulation.
Grain size	Number of routers assigned to each logical process of the parallel simulator.
Lookahead	A boolean value, indicating whether or not special effort must be done to extract lookahead from the model.

Table 1. Parameters of the simulators.

	1S	2S	3S	4S	5S	6S	7S
Network size	16×16	16×16	16×16	16×16	32×32	32×32	24×24 (25×25)
Message length	4	4	32	32	4	32	4
Load	5—90	5-90	5--90	5-90	5—90	5-90	50
Number of PEs	4	16	4	16	16	16	4— 25
Grain size, lookahead	Table 3	Table 4	Table 3	Table 4	Table 5	Table 5	Interm., no

Table 2. Experiments performed with CMB in the Supernode.

	LPs per Transp.	Routers per LP	lookahead
4Max	1	64	no
4ih	4	16	no
4il	16	4	no
4min	64	1	no
4minL	64	1	yes

Table 3. Values of the parameters grain size and lookahead for experiments **1S** and **3S**.

	LPs per Transp.	Routers per LP	lookahead
16Max	1	16	no
16i	4	4	no
16min	16	1	no
16minL	16	1	yes

Table 4. Values of the parameter grain size for experiments **2S** and **4S**.

	LPs per Transp.	Routers per LP	lookahead
ih	4	16	no
il	16	4	no

Table 5. Values of the parameter grain size for experiments **5S** and **6S**.

	1P	2P	3P	4P	5P	6P	7P	8P
Network size	16x16	16x16	16x16	16x16	32x32	32x32	24×24	90×90
Message length	4	4	32	32	4	32	4	4
Load	5-90	5-90	5-90	5-90	5-90	5—90	50	50
Number of PEs	4	16	4	16	16	16	4-44	4--100

Table 6. Experiments performed with CMB in the Paragon.

	1M	3M	5M	6M	8M
Network size	16×16	16x16	32x32	32x32	90×90
Message length	4	32	4	32	4
Load	5—90	5–90	5—90	5—90	5—90

Table 7. Experiments performed with MPI in a network of workstations.

Computing system	Data rate (Mb/s)	Execution time
Supernode	4	5597
Paragon	4	904
NOW with MPI	0.06	817

Table 8. Communication and computation abilities of the multicomputers used in the experiments. The data rate is for 32-byte messages. The execution times are those of the sequential simulator running experiment 5 at load 90.

Figures

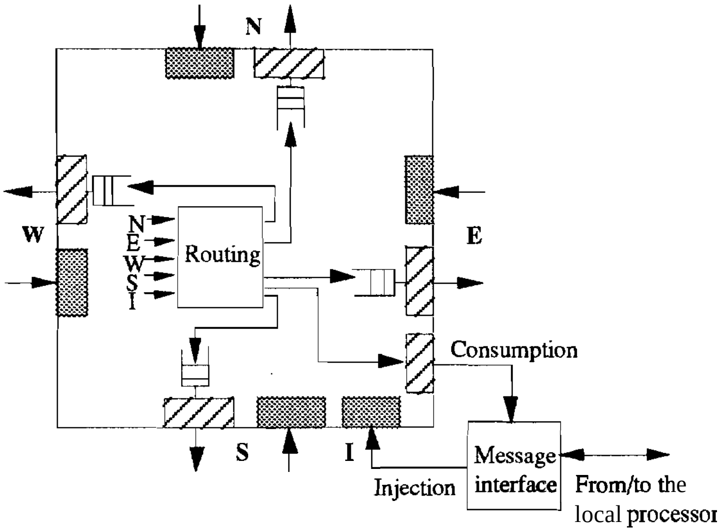


Figure 1. Model of message router.

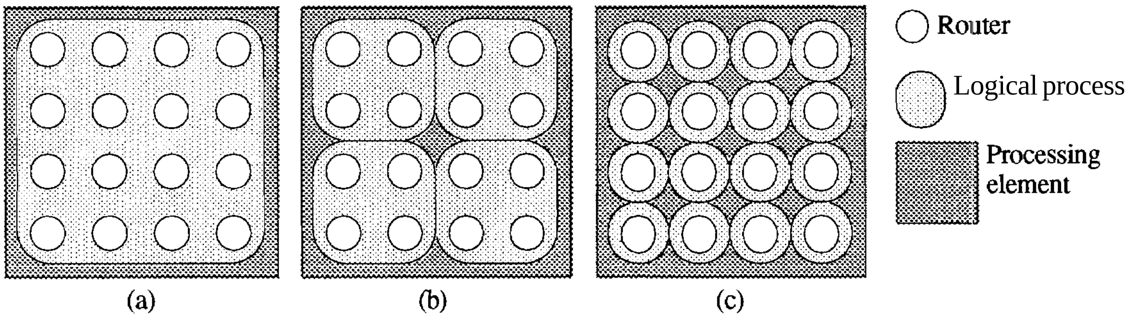


Figure 2. Mapping a network of 4x4 routers onto a PE. (a) Maximum grain size. (b) Intermediate grain size. (c) Minimum grain size.

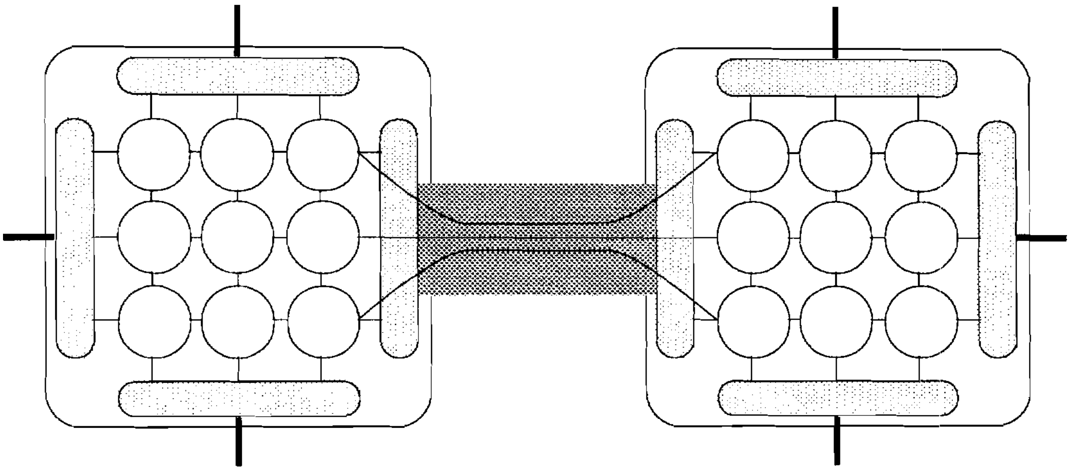


Figure 3. Organization of processes in a worker transputer. Circles represent LPs. Multiplexers are needed to share links.

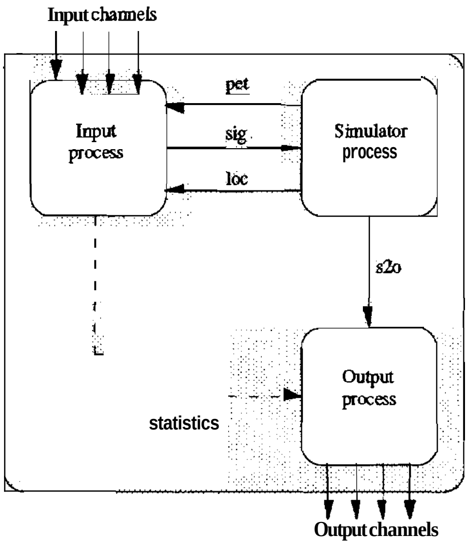
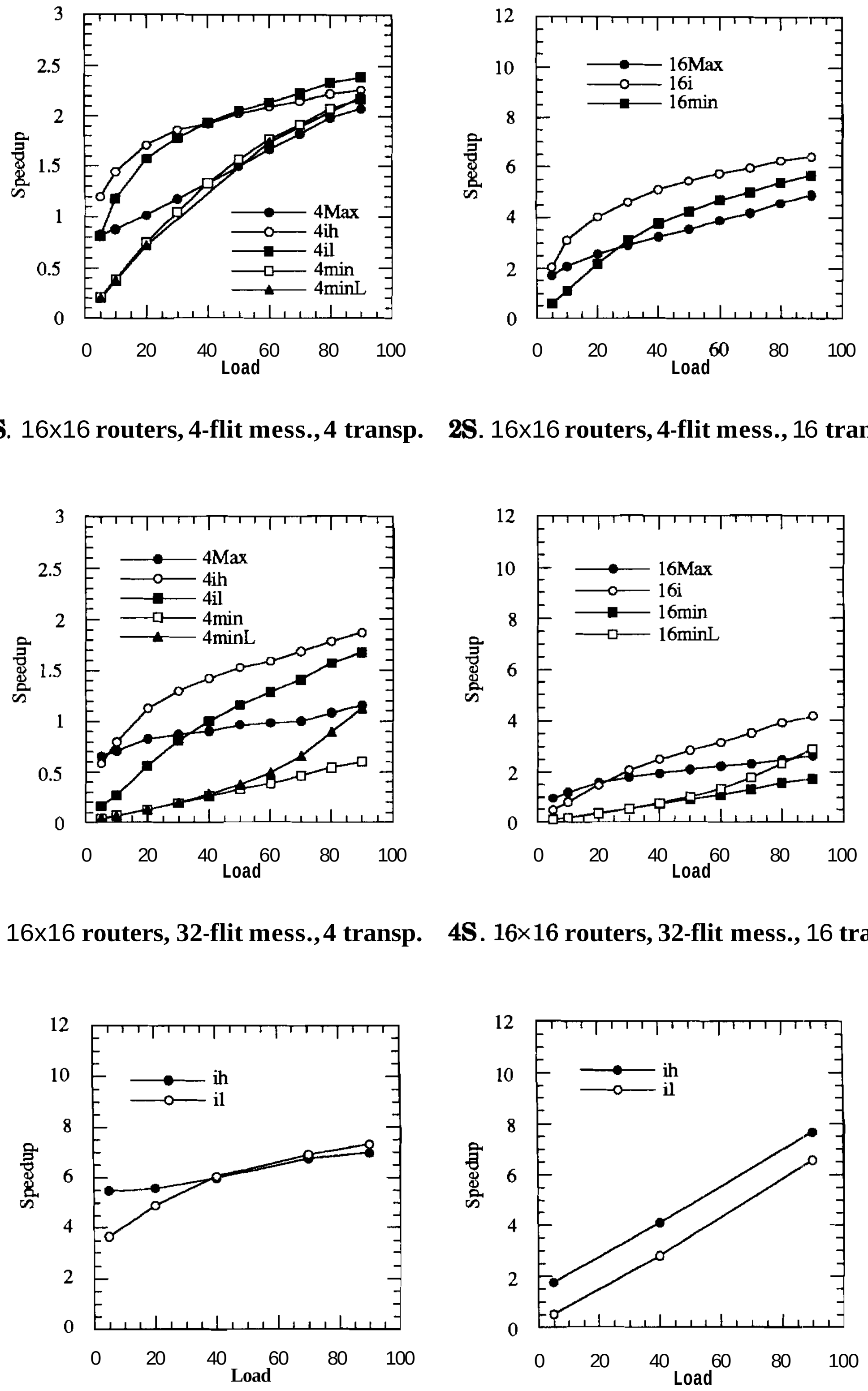


Figure 4. Structure of a logical process in the Supernode implementation of CMB.



1S. 16x16 **routers, 4-flit mess., 4 transp.** **2S**. 16x16 **routers, 4-flit mess.,** 16 **transp.**

3S. 16x16 **routers, 32-flit mess., 4 transp.** **4S**. **16×16 routers, 32-flit mess.,** 16 **transp.**

5S. 32x32 routers, 4-flit mess., 16 **transp.** **6S. 32x32 routers, 32-flit mess.,** 16 **transp.**

Figure 5. Results of experiments 1S—6S.

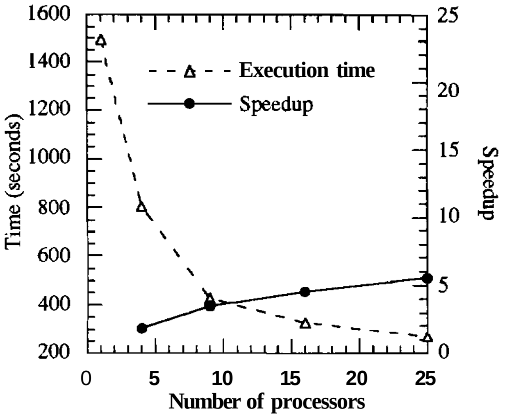


Figure 6. Results of experiment 7S. Simulation, using 4—25 transputers, of a network of 24x24 routers (25x25 for the case of 25 transputers) with 4-flit messages at.load 50.

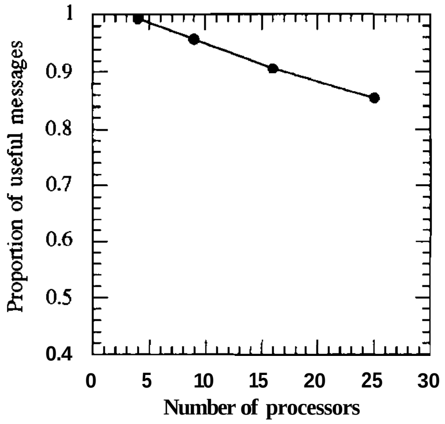
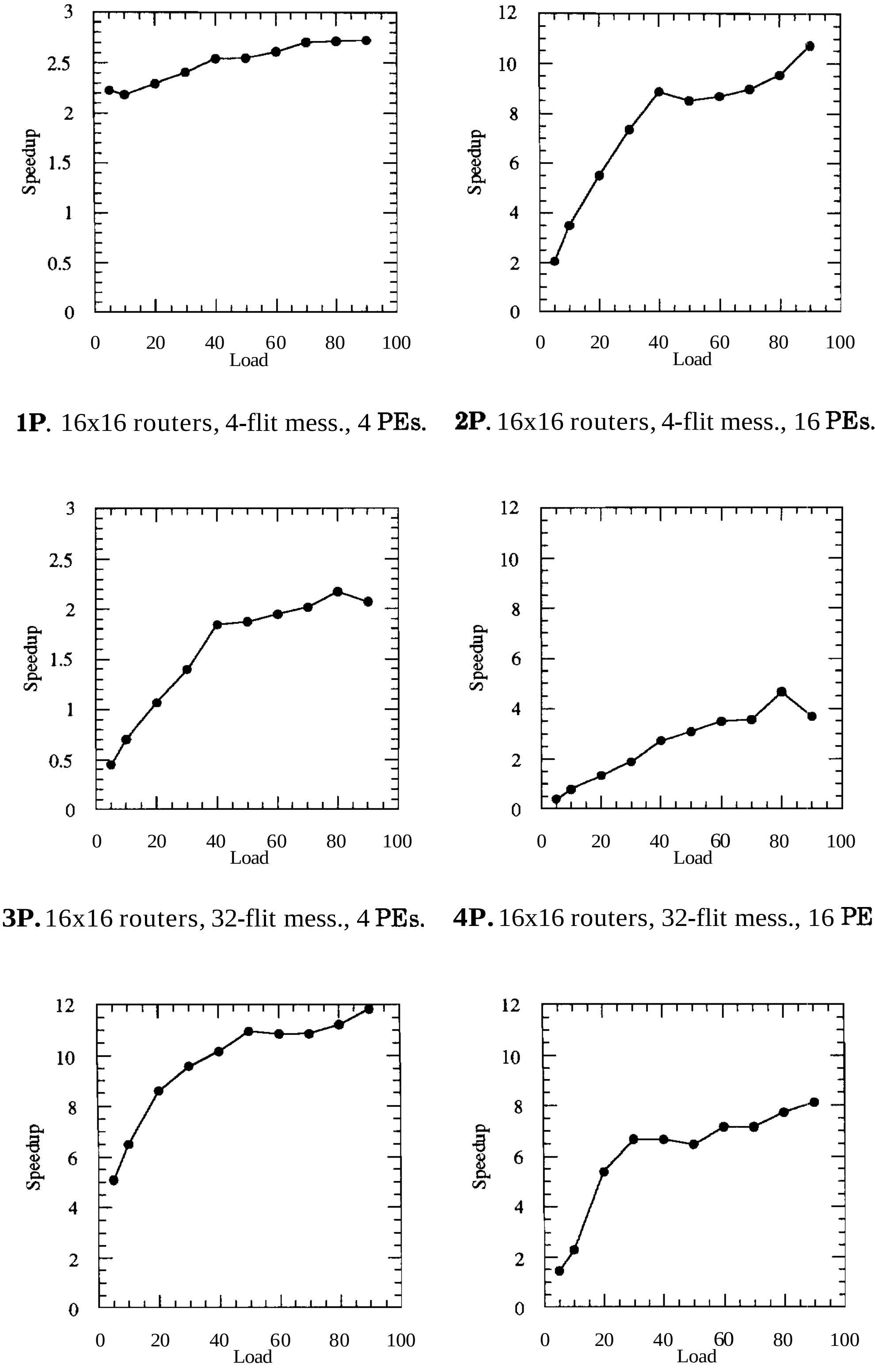


Figure 7. Ratio of useful to total messages for experiment 7S.



1P. 16x16 routers, 4-flit mess., 4 **PEs**.

2P. 16x16 routers, 4-flit mess., 16 **PEs**.

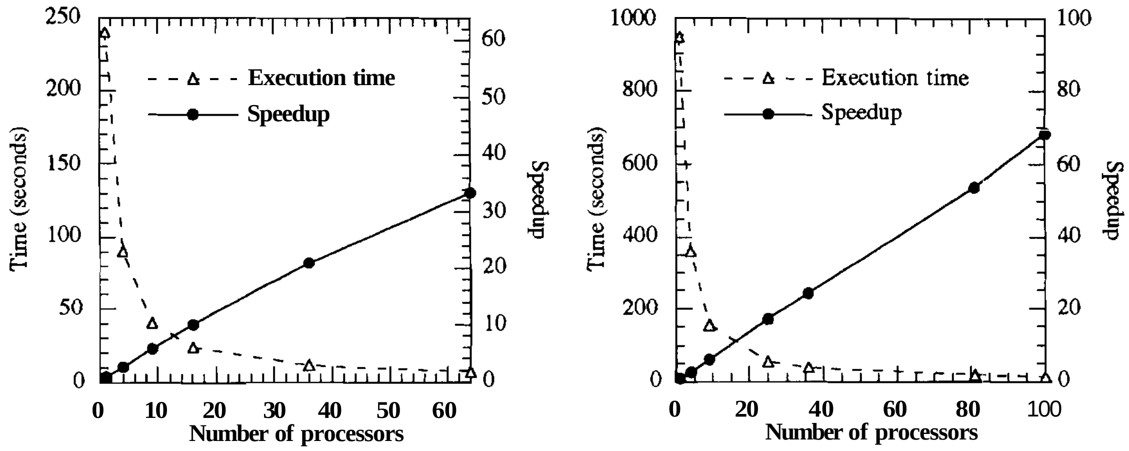
3P. 16x16 routers, 32-flit mess., 4 **PEs**.

4P. 16x16 routers, 32-flit mess., 16 **PEs**.

5P. 32x32 routers, 4-flit mess., 16 **PEs**.

6P. 32x32 routers, 32-flit mess., 16 **PEs**.

Figure 8. Results of experiments **1P—6P**.



7P.24x24 routers, 4--64 PEs. **8P. 90x90 routers, 4--100 PEs.**

Figure 9. Results of experiments 7P and 8P, the scalability tests.

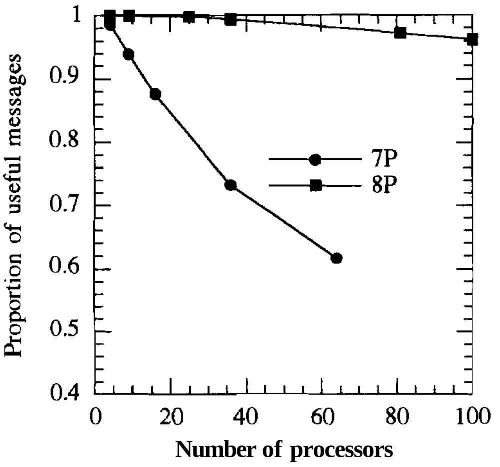
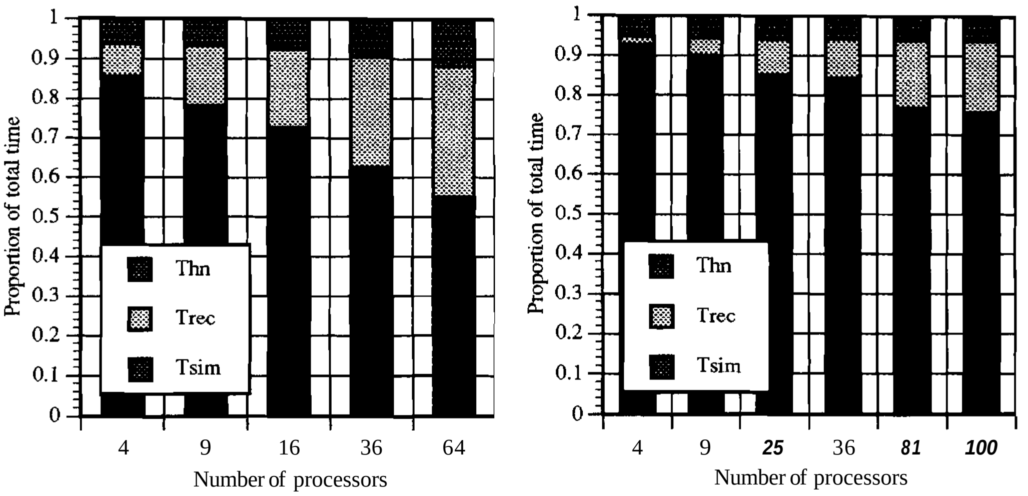


Figure 10. Ratio of useful to total messages for experiments 7P and 8P



7P. 24x24 routers, 4--64 PEs.

8P. 90x90 routers, 4--100 PEs.

Figure 11. Distribution of total execution time among simulation and synchronization for experiments 7P and 8P. Thn = time spent computing the acceptance horizon and sending null messages; Trec = time spent receiving messages (null messages included); Tsim = time spent executing events and sending non-null messages.

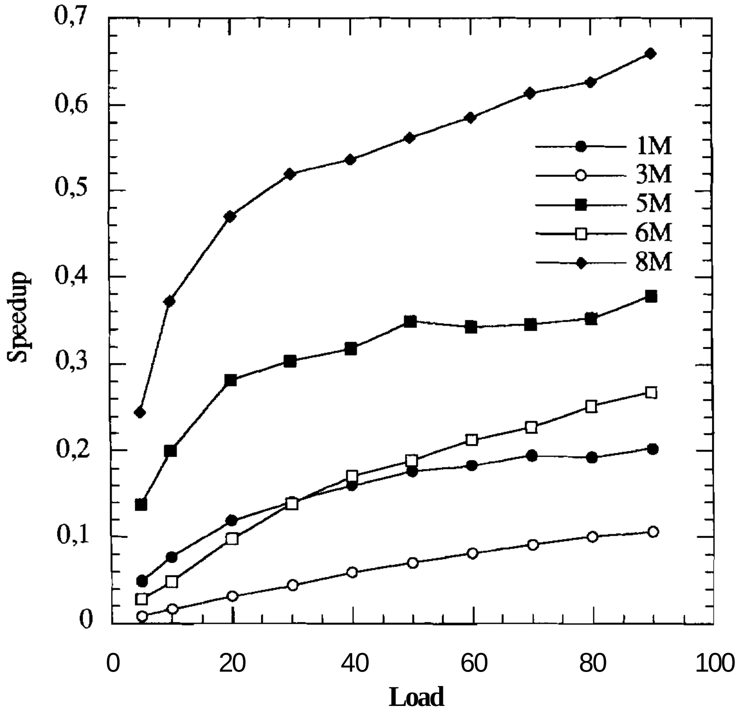


Figure 12. Results of the experiments 1M—8M.

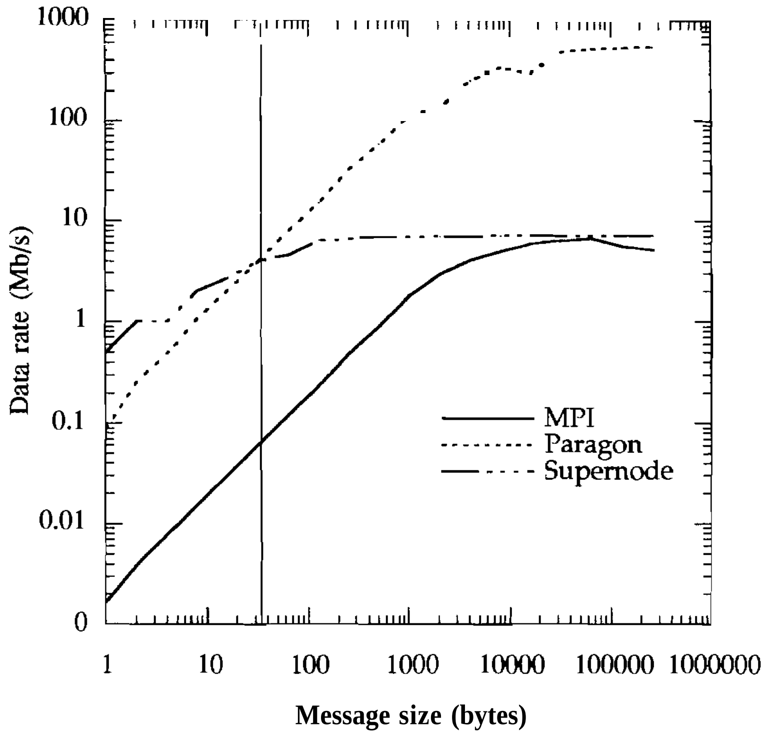


Figure 13. Achieved data rates as a function of the message size in the three multicomputer systems used in the experiments.