Euskal Herriko Unibertsitatea
Universidad del País Vasco

Konputagailuen Arkitektura eta Teknologia Saila
Dep. de Arquitectura y Tecnología de Computadores

eman ta zabal zazu

universidad del país vasco
euskal herriko unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMATICA

An empirical evaluation of techniques for parallel simulation of message passing networks

Memoria que para optar al grado de Doctor en Informática
presenta
José Miguel Alonso

Dirigida por Ramón Beivide Palacio

San Sebastián, Octubre de 1995

Este trabajo ha sido parcialmente subvencionado por la Comisión Interministerial de Ciencia y Tecnología, bajo contrato TIC95-0378

Extracto - resumen

En el ámbito del diseño de computadores, la simulación es una herramienta imprescindible para la validación y evaluación de cualquier propuesta arquitectónica. Las ténicas convencionales de simulación, diseñadas para su utilización en computadores secuenciales, son demasiado lentas si el sistema a simular es grande o complejo. El objetivo de esta tesis es buscar técnicas para acelerar estas simulaciones, aprovechando el paralelismo disponible en multicomputadores comerciales, y usar esas técnicas para el estudio de un modelo de encaminador de mensajes. Este encaminador está diseñado para formar infraestructura de comunicaciones de un hipotético computador masivamente paralelo.

En este trabajo se consideran tres técnicas de simulación paralela: síncrona, asíncrona-conservadora y asíncrona-optimista. Estos algoritmos se han implementado en tres multicomputadores: un Supernode basado en Transputers, un Intel Paragon y una red de estaciones de trabajo. Se caracteriza la influencia que tienen en las prestaciones de los simuladores aspectos tales como los parámetros del modelo simulado, la organización del simulador y las características del multicomputador utilizado.

Se concluye que las técnicas de simulación paralela optimista no resultan adecuadas para trabajar con el modelo considerado, aunque pueden ofrecer un buen rendimiento en otros entornos. La red de estaciones de trabajo no resulta una plataforma apropiada para estas simulaciones, ya que una red local no reúne condiciones para la ejecución de aplicaciones paralelas de grano fino. Las técnicas de simulación paralela síncrona y conservadora dan muy buenos resultados en el Supernode y en el Paragon, especialmente si el modelo a simular es complejo o grande—precisamente el peor caso para los algoritmos secuenciales. De esta forma, estudios previamente considerados inviables, por ser demasiado costosos computacionalmente, pueden realizarse en tiempos razonables. Además, se amplía el espectro de posibilidades de los multicomputadores, utilizándolos para algo más que aplicaciones numéricas.

Abstract

In the field of computer design, simulation is an essential tool to validate and evaluate architectural proposals. Conventional simulation techniques, designed for their use in sequential computers, are too slow if the system to simulate is large or complex. The aim of this work is to search for techniques to accelerate simulations exploiting the parallelism available in current, commercial multicomputers, and to use these techniques to study a model of a message router. This router has been designed to constitute the communication infrastructure of a (hypothetical) massively parallel computer.

Three parallel simulation techniques have been considered: synchronous, asynchronous-conservative and asynchronous-optimistic. These algorithms have been implemented in three multicomputers: a transputer-based Supernode, an Intel Paragon and a network of workstations. The influence that factors such as the characteristics of the simulated models, the organization of the simulators and the characteristics of the target multicomputers have in the performance of the simulations has been measured and characterized.

It is concluded that optimistic parallel simulation techniques are not suitable for the considered kind of models, although they may provide good performance in other environments. A network of workstations is not the right platform for our experiments, because the communication demands of the parallel simulators surpass the abilities of local area networks—the granularity is too fine. Synchronous and conservative parallel simulation techniques perform very well in the Supernode and in the Paragon, specially if the model to simulate is complex or large—precisely the worst case for traditional, sequential simulators. This way, studies previously considered as unrealizable, due to their exceedingly high computational cost, can be performed in reasonable times. Additionally, the spectrum of possibilities of using multicomputers can be broadened to execute more than numeric applications.

Acknowledgments

There are so many people that helped the author in making possible this enterprise to arrive, that an acknowledgment list would be nearly endless. For this reason, I'll be as concise as possible:

> To all my colleagues, faculty and staff, at the Facultad de Informática, Universidad del País Vasco, with special mention to Josu, Juan and Pepe.
>
> To the personnel (again, faculty and staff) at the School of Electrical and Computer Engineering, Purdue University, in general, and the research group leaded by Prof. Fortes in particular, with special mention to Zina.
>
> To the members of my research group: Cruz, Agus and, of course, Ramón.

In a more technical side, I want to thank the staff at the computing centers of the University of the Basque Country and Purdue University (PUCC, ECN) for providing all the support that the author needed to complete this work.

A mis padres, hermano y hermanas

Table of contents

Chapter 1

Introduction

1.1 Motivation of this work

The objective of the work presented in this dissertation is to propose efficient, parallel ways to simulate message passing networks. The interest of the work is, from our point of view, twofold:

- Most of the effort of our research group is focused on the study of the architecture of parallel systems. As part of this research, we perform a significant amount of simulation work. These simulations are very time-consuming, hence we would like to find a way of accelerating them.
- We are also interested in the efficient use of currently available and future multicomputers; in this context, we see simulation as an interesting application to parallelize.

Next we elaborate these two ideas.

1.1.1 Analysis of real systems

During the last few years our research group has been working in the analysis and design of message routers, with the goal of proposing innovative and efficient architectural alternatives for the communication infrastructure of a distributed memory multicomputer. The effectiveness of our proposals has to be validated somehow, and these are the tools considered for this purpose: analytical models, actual implementations and computer simulation. This situation is common to many fields of science and engineering, so from now on we will widen our focus and speak, in general, about the evaluation of any kind of real (or imaginary) system.

Analytical tools can be described as cheap and fast to use. Of course the development of an analytical model of a system can be very complex, but once a set of equations has been developed, it is easy to extract information from it. Unfortunately, this approach commonly requires the assumption of simplifications in some (or many) of the characteristics of the system, for the researcher to be able to tackle the analytical problem. These simplifications can lead to a model whose behavior is (or may be) far from the behavior of the real system: the results might not be accurate if some of the simplifying assumptions are not realistic.

The second approach can be described as slow and expensive. The system has to be built, which usually means a good deal of prototyping and development of auxiliary systems, and the price to pay is high, generally speaking. Another problem of this approach is that it is very inflexible: once a system has been built, it is very difficult to introduce improvements, replace parts of the system, etc. It is evident, though, that the information obtained from the study of the real system (after an appropriate instrumentation) is very accurate.

Computer simulation offers an interesting compromise: the system can be described somehow (using a simulation language, in most cases) and then simulated using a computer. The description could include simplifying assumptions, like the analytical model, and then the simulation time would be short. In contrast, the description could be very detailed, containing as many elements as the real system, and then a highly accurate insight into the behavior of the system would be obtained. Of course, the accuracy comes at a price: long simulation times.

Computer simulation is extremely flexible. It can be used in many contexts, including the following ones:

- To validate an analytical model.
- To see how an existing system works, when it cannot be easily instrumented.
- To study a non existing system, without building it. There are many reasons not to build a system: it might be very expensive, or it might be simply impossible.
- To analyze the effect of different design parameters, in an existing or a non-existing system.

It has been mentioned already that the accuracy of the description of a system has an important influence on the execution time of the simulations. However, not only the degree of detail of a system description influences the execution times, but also its size. Those using simulators know that analyzing large and/or detailed systems can be desperately slow. In this context, any possibility of increasing the execution speed of the simulators is very welcome. The speed increments due to the advances in VLSI technology have been significant, but it always exists demand for more. The introduction in the market of reasonably priced parallel systems has allowed researchers to accelerate many computations, and it seems logical to think that simulation may also benefit from this technology.

1.1.2 Parallel implementation of applications in distributed memory parallel computers

As researchers in the broad field of parallel computing, we are interested in architectural proposals for building the next generation of Massively Parallel Processors, and also in making a good use of currently available parallel computers. During this research we have had access to three different parallel computing systems: a Supernode at the UPV/EHU, a Paragon at Purdue University and a network of workstations with MPI libraries also at Purdue. It was our interest to see how well traditionally sequential applications could be adapted to run in these machines. A good deal of work can be found in the literature reporting parallel algorithms to solve many problems, mainly in scientific and engineering fields, but most of those problems have some characteristics that make them, to some extent, easy to parallelize: big, partitionable data structures, simple communication patterns, reduced data dependencies, etc. There are many, general-purpose applications that have not received such attention: compilers, editors, document preparation programs, etc. In many cases, these applications do not have the properties that make the scientific code able to be successfully parallelized.

Simulation can be included in this last group. While it is a tool commonly used by scientists and engineers, simulation algorithms exhibit a behavior that makes them difficult to parallelize: data structures are not always regular, communication among the parts of the model may follow arbitrary patterns, there are very strong data dependencies, and so on. But difficult does not mean impossible. As it will be explained in Chapter 2, simulations can be parallelized, provided that new algorithms are developed, instead of simply trying to make in parallel some of the operations of the sequential programs.

Being successful in the search of parallel simulators is very important, because it widens the spectrum of applications that can be run in an available parallel computer, increasing the usefulness of the (in general expensive) investment, and demonstrating that parallel computers are useful for more things than “number crunching”.

Obviously, we are not the first research group working in this field, as many work has been done in the last 10—15 years. Most of the work discussed in the literature about parallel discrete event simulation has been done using shared memory multiprocessors. This is perfectly reasonable, for two reasons: (1) many multiprocessors have been built and sold, so it makes sense to use them, and (2) they allow a more optimized implementation of many algorithms (compared to distributed memory systems), including parallel simulation. However, it is currently assumed that future massively parallel systems will be distributed memory systems [Zorp92]. Many currently available machines, including the Intel Paragon multicomputer and systems based on transputers, are built this way, using messages for synchronization and communication. The use of a network of workstations as a fully distributed parallel system is also becoming very popular, because it is a very cost/effective alternative to a full-blown multicomputer. The most popular model of communication for these systems is also message passing, and this is the approach we follow in this work.

1.2 The tool: parallel simulation

In the last years a considerable effort has been devoted to the parallel implementation of discrete event simulators. The objectives were (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 several simulations 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 job is simply running as many simulations as processors are available, each one with different input parameters. This technique is called *replication*. The achieved efficiency is very good, because the simulations are completely independent, and therefore there is no need of communication or synchronization among the involved processors.

However, it is not always possible to replicate the simulator. In some studies it is necessary to have the results of one simulation before starting with the next one; this is the case when the aim is to tune a set of parameters. It is also possible that the memory available at each processor is not large enough to keep a complete copy of the simulator. These limitations of the replication approach justify the need of ways to parallelize a single simulation run. Even if the processing elements are powerful enough to run an individual simulation, it is possible to have more processors than experiments to perform. Under these circumstances, it would be interesting to use many processors for each experiment, combining replication with other forms of parallel simulation.

The most promising set of techniques to perform parallel simulation uses the *model decomposition* principle: the system to simulate is decomposed into several subsystems, and each subsystem 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 in the simulation, a synchronization mechanism is needed.

In this work we analyze three different synchronization mechanisms: synchronous, conservative and optimistic. A brief description follows.

- Parallel simulation can 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 a sequential simulator would choose, simulating in parallel only those events 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 arrive from other LP, with a timestamp smaller than that of the last executed event, thus impeding the receiver LP to obey the stated rule.
    - A *conservative* simulator never allows these situations to happen. To do so, LPs block before executing an event, until it is totally safe to proceed.
    - An *optimistic* simulator allows erroneous situations to arise, but those are detected and a *rollback* is done to jump to an error-free point in the (simulated) past.

A more comprehensive description of these techniques will be given in Chapter 2.

1.3 The objective: analysis of message passing networks

Previous research in parallel simulation shows that its efficiency is highly dependent on the characteristics of the system under study. For this reason it is not feasible to characterize the performance of the different parallel simulation algorithms in a general context. It is possible, however, to select a set of related models and extract conclusions about how a given algorithm performs with that set of models. Our research will focus on models of message passing networks for multicomputers.

In the quest for the TeraFLOP (and even the PetaFLOP) it seems that massively parallel computers are the most reasonable way to go. These architectures consist of a large collection of processors which might be able to collaborate to solve problems with fine-grain parallelism. The programming model might be based either on message passing or in the existence of a single memory space, although physically the memory banks are distributed among the computer nodes. In either case, a mechanism is needed to provide communication among the nodes, with low latency and high message throughput.

The communication subsystem is one of the key points in the design of these new architectures. Different designs and implementations, as those in the Cray T3D, the Thinking Machines CM-5 and the Intel Paragon, are good examples of current parallel computers where interconnection subsystems play fundamental roles.

Previous research of our group has been centered in this topic. In particular, several interconnection topologies, routing strategies and message flow control techniques have been studied with the aim of maximizing throughput and minimizing latency within a reasonable cost [BHBL87, BHBA91, Migu91, IBJA93, Arru93, Izu94]. These studies concluded that a bi-dimensional torus network with static routing in order of dimension and cut-through flow control is a good candidate for a high performance message passing network. The work presented in this dissertation has been focused on the simulation of networks with these characteristics. A detailed description of the model can be found in Chapter 5.

1.4 The means: programming in multicomputer environments

A parallel computer might provide one or more of these programming paradigms: *SIMD* (Single Instruction Multiple Data), meaning that all the processes run the same program, instruction by instruction, at the same time, and *MIMD* (Multiple Instruction Multiple Data), meaning that each process might run a completely different program. A particular, more restrictive case of MIMD is *SPMD* (Single Program Multiple Data), where all the parallel processes run the same program. The parallel simulation algorithms briefly introduced in the Section 1.2 can be easily implemented as SPMD programs.

The processes running in parallel need a means of communication and synchronization. The most common alternatives are shared memory and message passing. In the first case all the processes share the same vision of the memory space; communication is provided by means of shared data structures, while synchronization might be provided by mechanisms such as semaphores. In the second case each process has its own memory space; both communication and synchronization are provided by means of messages sent between processes.

Our work has been developed in message passing environments. There are several reasons for making this decision. Firstly, message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory; in particular, the Supernode, the Paragon and the network of workstations (*NOW*) with MPI used for this work provide only message passing for communication among processes. Secondly, parallel simulation algorithms based on model decomposition are described by means of a set of processes which interchange messages.

There is not a single paradigm for message passing, though. Communication in the Supernode is done using the concept of *channel*, and send/receive operations are both blocking: an operation on a channel keeps a process blocked until the peer operation is done by another process on the same channel. In the Paragon and MPI processes have identifiers (addresses), and messages are delivered using the destination address. The send operation is (usually) non-blocking, while a receive operation keeps a process blocked until a message is received.

The hardware mechanism used for message passing should not be a concern for the programmer, but nevertheless it has to be taken into account, because it greatly influences the performance of the communication functions and, thus, of the applications. Slow interconnection systems favor the development of solutions with relatively infrequent communication, while a fast system allows the development of fine-grain applications, where communication is very frequent. The characteristics of the three different systems used in this research are quite different:

- In the network of workstations, the MPI message passing library uses TCP/IP over an Ethernet local area network. This medium provides a raw 10 Mb/s data rate, which must be shared among all the stations attached to it. The communication effort is performed by the workstations CPUs, with the aid of the Ethernet cards for accessing the medium.
- In the Supernode, each node (a transputer) has 4 bi-directional serial links with a maximum data rate of 10 Mb/s each, so the aggregate data rate is higher than in

the previous case. The communication effort is done inside the transputer: there are hardware controllers for the links which can operate in parallel with the CPU, but some functions (link sharing, routing) do not have any hardware support.

- In the Paragon there is a good deal of hardware support for message passing: communication is separated of computation, by means of a collection of hardware message routers organized in a mesh topology. The communication links which join routers can move up to 1600 Mb/s. Additionally, each node has a second processor specialized in communication, leaving the main processor free to spend its time performing computation.

In Chapter 3 we will give a deeper insight into the programming models provided by parallel systems, as well as particular descriptions of the environments used in this research. The influence that the communication ability of the systems have on the simulation performance is further considered in Chapter 6.

1.5 Major contributions

The major contributions of this work include the implementation of a model of message passing network which can be simulated using a discrete event simulator, along with six simulation engines able to work with that model: one is sequential, and the other five are parallel, testing three different synchronization mechanism in three different multicomputing environments. We have characterized:

- Which synchronization mechanisms provide an adequate tool for our studies, and which others are not so good. The synchronous and conservative approaches perform well, while the optimistic approach does not seem to be the right one for our purposes.
- The influence of the target multicomputer on the efficiency of the simulation. Our simulators exploit a fine-grain parallelism which requires a fast message passing infrastructure. A network of workstations is not efficient, because communication is too costly compared to computation.
- The effect of the parameters of the model on the performance of the simulators. Models of large size whose components interact frequently constitute a challenge

for sequential simulators, but simplify the synchronization tasks of conservative and synchronous parallel simulators, offering good levels of performance.

- The effect that the organization of a simulator has in its performance. The partition of a system into a set of subsystems might be done in several ways, leading to fine, medium or large grain LPs. In the case of large grain LPs, each processor executes just one LP. In the case of medium and fine grain LPs each processor executes several (or many) LPs. Large grain LPs provide better performance than fine grain LPs but, if the processors and the operating system provide appropriate support for concurrency, it is advantageous to assign several medium grain LPs to each processor.

1.6 Overview of this dissertation

The previous sections have given an introduction to the work presented in this dissertation. Now we summarize how it is organized.

In Chapter 2 we provide a survey of parallel simulation techniques. We start with a description of the general process of studying a system by means or simulation, identifying the execution of a simulation program in a computer as one (but not the only one) step in this process. After introducing the main discrete event simulation concepts, two common sequential simulation techniques are presented: time-driven and event-driven. Then, a description of the ways of exploiting the parallelism available in current multiprocessors is given, focusing on those techniques based on model decomposition. After that, the synchronous, conservative and optimistic synchronization mechanisms are presented, along with a series of improvements and optimizations to the basic algorithms.

Chapter 3 gives an introduction to the programming environments available for parallel systems, with special attention to the three used in this research: a Supernode, an Intel Paragon and a network of Sun workstations with a MPI (Message Passing Interface) library. All three have in common that they are distributed memory systems, providing a message passing mechanism for synchronization and communication.

Chapter 4 presents the results of a preliminary study of simulation techniques performed in a transputer-based system. A simplistic, synthetic model is used to test how four different simulators (sequential time-driven, sequential event-driven, parallel conservative and parallel optimistic) behave under different parameters of the model, using different number of processors. Although the model is not realistic, it provides a simplification of the actual models we are interested in. It is observed that the characteristics of the simulated model have a great impact on the achieved performance, for the case of the parallel simulators. Additionally, the performance of the conservative simulator clearly exceeds that of the optimistic. The conclusions of this preliminary study were tested (and corroborated) by later studies with a more complex model.

Chapter 5 describes the experimental environment developed for this work. It includes a detailed description of the kind of interconnection network we are interested in: torus networks of pairs <processor, router>. The router is the key element of the study, because it is the network of routers what constitutes the communication infrastructure of a multicomputer. After the description of the model we present five parallel simulators which have been developed and studied as part of this work: conservative for the Supernode; conservative, optimistic and synchronous for the Paragon, and conservative for a network of workstations with MPI. Some portability issues are also discussed: one of the algorithms has been implemented in three computer systems, needing specific tailoring to adapt it to the different programming models.

Chapter 6 describes the experiments performed with the five parallel simulators. Each one is studied separately, presenting in first place the experiments, followed by the results and a series of partial conclusions. As the conservative algorithm has been implemented in three different systems, the influence of the host computer on the achieved performance is discussed. The availability of three different algorithms for one of the host computers (the Paragon) also allows for a characterization of the obtained performance as a function of the synchronization technique.

Finally, Chapter 7 summarizes the contributions of this work, suggesting lines for further research.

Chapter 2

Parallel discrete event simulation

This chapter surveys the literature about parallel discrete event simulation, with the purpose of introducing the terminology and algorithms used in the remainder of this dissertation. After a general introduction to discrete event simulation, it is shown how the concept of causal order is the key element which allows the parallelization of simulations, when used instead of temporal order. The three parallel simulation algorithms used in this research are then introduced: synchronous (SPED), conservative (CMB) and optimistic (TW).

2.1 Introduction

As already discussed in Chapter 1, most of the fields of science and technology require the modeling and analysis of the behavior of dynamic systems. Common to realistic models of time dynamic systems is their complexity, very often prohibiting numerical or analytical evaluation. Prototyping is a complementary tool to use, but in many cases it is very expensive and in others it is absolutely unfeasible. For those cases, simulation remains the only tractable methodology.

As computer simulation in general is such a broad field, in this chapter we will narrow our domain of study by focusing on simulation of discrete event systems. After doing so, the classical, sequential approaches to this class of simulation problems are described, and then we make a survey of the currently available techniques to realize parallel discrete event simulations (PDES). The methods and terminology described in this chapter will be repeatedly used in the remainder of this dissertation.

The chapter is structured as follows. We start considering the general topic of analyzing systems by means of computer simulation (§2.2), and then the sequential simulation algorithms for discrete event systems are introduced (§2.3). As these algorithms become very expensive in memory demands and execution time when the problem being simulated grows, the need of parallel computers to perform the simulations becomes evident. Several approaches to the parallelization of discrete event simulators are presented, with special attention to event-driven simulations based on model decomposition (§2.4). Sections 2.5, 2.6 and 2.7 are devoted to the description of three important families of parallel simulators: synchronous, asynchronous conservative and asynchronous optimistic. The basic algorithms are introduced, with their most important variations or optimizations. Finally, section 2.8 gives some conclusions and some directions for those interested in a deeper insight into this field.

2.2 Modeling and simulation

We will start defining the problem to solve, i.e., the simulation of real systems using computers [Zeig76, Bank84, Misr86]. This activity needs, at least, these four steps:

1 Study of the real system in order to understand its characteristics.
2 Modeling the system. A key issue in this step is to discriminate between relevant and irrelevant characteristics of the system. The irrelevant ones are not included in the model.
3 Simulation of the model, using computers.
4 Analysis of the simulator's output, to understand and predict the behavior of the real system.

We are mostly interested in the third step, i.e., the simulation of the model. The observation of systems, the modeling process and the evaluation of results strongly depend on the actual system under study, but most simulation algorithms are independent of the model characteristics.

The systems to study can be separated into two categories: discrete or continuous. A system is *discrete* when its *state* changes only at discrete times, whereas a system is *continuous* when its state varies continuously in time.

A model being simulated can be classified as static or dynamic, deterministic or stochastic, and discrete or continuous. A model is *static* when it tries to capture a snapshot of a system, at a particular instant of time, and it is *dynamic* if it tries to represent the evolution of the system along a certain interval of time. A model is *deterministic* when it generates, for a given set of input values, a single set of output values; it is *stochastic* when random variables are part of the input and, therefore, the output can only be considered as an estimate of the actual behavior of the system. Regarding the division of models among *discrete* and *continuous*, the same definitions used to characterize systems are valid here too. It should be stressed that it is not necessary to use a model of a given class to describe a system of the same class. It is possible, for example, to use a discrete model to describe a continuous system. The selected model depends on the nature of the system and also on the set of expected results.

Taking into account the nature of the systems we are interested in, we will concentrate on discrete, dynamic and stochastic simulation models. Given one of these models, we will introduce different ways of performing a computer simulation. Next we will characterize the models object of our study. Some terminology used when describing the different simulation techniques will also be introduced.

We consider that a real system, or *physical system*, is modeled as a set of one or many *physical processes* (PPs) which evolve in time. It is assumed that all the PPs share a common view of time, i.e., that a global *clock* exists which can be used as a reference of the advance of time in the system. Each PP evolves autonomously, except to interact with other PPs in the physical system. These interactions, called *events*, occur at discrete times, and are modeled as interchanges of *messages* between pairs of PPs. Messages contain two fields of information: the event they represent, and the time where that event should happen (its *timestamp*). Most of the time we will use the terms "event" and "message" interchangeably, unless the distinction is important.

We assume that PPs have a certain ability to predict the events that will occur in a next future. When a PP_i knows that, at time t ($t \geq clock$), it will interact with another PP_j, and that the interaction will be of type e, PP_i *schedules* an event for PP_j. This scheduling action is modeled as PP_i sending a message $\langle e, t\rangle$ to PP_j (see Figure 2.1). The restriction of t belonging to the future is self-explanatory: the past cannot be affected by a present event. The content of the message depends on the characteristics of the sender PP_i (initial state, behavioral rules) and on the messages PP_i received previously. The message $\langle e, t\rangle$ will be consumed in PP_j when the clock reaches the value t. As a result, PP_j's state will change accordingly to the class of interaction modeled by e. This state change can trigger the scheduling of new events for the future and, therefore, the emission of new messages.

Sometimes a message previously scheduled for the future needs to be canceled before it actually happens (i.e., before the clock reaches the event's timestamp). The timestamp of a message says when the event *should* happen. An already sent message (scheduled event) for time t can be canceled by means of another message timestamped less than t.

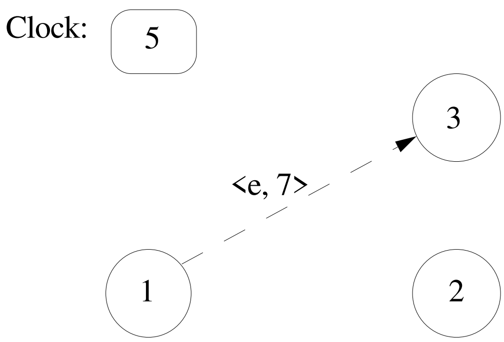


Figure 2.1. A group of 3 physical processes which form part of a physical system. The (global) clock indicates that current time is 5.
At this time, PP_1 schedules an event e to happen at time 7 in PP_3. To do so, a message <e, 7> is sent from PP_1 to PP_3.

Once we have a system modeled the way just described, and expressed in an executable way using either a simulation language or a general-purpose programming language, the model can be simulated in a computer. There are two basic approaches to do the simulation: it can be either time-driven or event-driven. Next we will give a sequential algorithm (i.e., designed to run in one processing element) to realize a *time-driven* simulation. The main elements (data structures) managed by this kind of simulator are:

- A *clock*, which represents the advance of time in the physical system. In each step of the algorithm this clock advances one unit of time. The value of the clock represents the time up to which the system has been simulated.
- A set of variables which represent the *state* of the set of PPs. This information must include the messages sent between the PPs, which represent future events that will occur in the system.
- Additionally, a set of variables for *statistics* gathering are usually required.

As already mentioned, in each step of the algorithm the clock advances one time unit. After doing so, all the state variables are examined, to check which events must occur at that particular time: those whose timestamp equals the value of the clock. Then, those events are *consumed*. Consuming an event in a particular PP produces the following effects:

- A change in the state of the PP, i.e., in its state variables.
- New events might be scheduled for the future, i.e., some messages might be sent to other PPs.
- Some previously scheduled events might be canceled.

Because consuming a message means to execute the simulation code associated with the corresponding event, we will sometimes use "to execute an event" meaning "to consume an event".

These two steps (clock advance, message consumption) are repeated until the simulation finishes. Usually this happens when the clock reaches a given *end_of_simulation* value, or when the system reaches a particular state.

As it can be seen from the description, the advance of the clock determines the advance in the simulation, and in each step exactly one time unit is simulated. However, in many systems events occur with a large time difference between each other, in such a way that, in most of the iterations of the algorithm, there are none (or just a few) events to consume. In these cases we have a low *event density*, where the event density is defined as the (average) number of events consumed per unit of simulated time.

Low event density scenarios led to an alternative simulation strategy, known as *event-driven*, where the clock can advance faster than it does in a time-driven simulator. The main elements of an event-driven simulator are, as in the previous case, a clock and a set of state variables, but a new data structure is added: an *event calendar* (also called *event list*, or *event queue*) which stores all the messages <*event, timestamp*> scheduled for the future, along with an identification of the target (destination) PP. Each message stored in the event calendar will be consumed when the clock reaches its timestamp, unless it is previously canceled. The event calendar keeps the messages in such a way that they can be retrieved in increasing timestamp order.

In each step of the algorithm the first message of the event calendar, i.e., the one with minimum timestamp, is removed, and the clock is advanced to reach that timestamp. Then the event is consumed at the destination PP, with the effects already described (the state of the PP might change, new messages might be sent, old messages might be canceled). This process is depicted in Figure 2.2. The way of advancing the simulation clock determines the difference between time-driven and event-driven simulation: in the last case, after consuming an event, the clock advances to reach the value of the next message's timestamp, with time jumps which might be larger than a time unit.

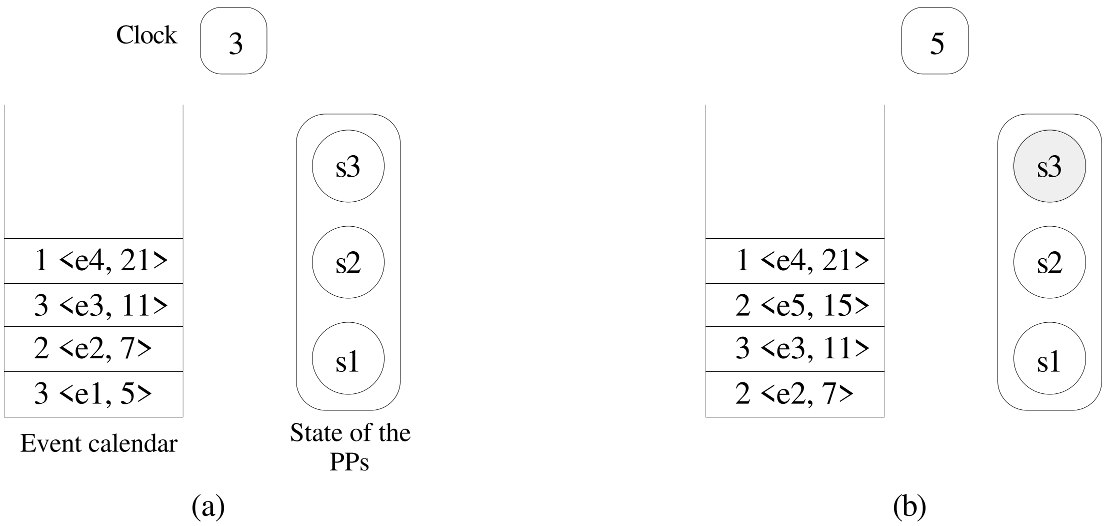


Figure 2.2. (a) State of the event-driven simulator before executing e_1, scheduled for PP_3 and time 5. (b) State after executing e_1: the clock has advanced 2 units, the state of PP_3 has been changed and event e_5 has been scheduled for PP_2 and time 15. Event e_5 has been inserted in the appropriate position in the event calendar, using the timestamp as the ordering criterion.

Figure 2.3 shows how an actual simulation of a physical system consists of two separate parts. One part is model dependent, an executable description of the model to simulate written in a general-purpose high-level language or in a simulation language (Simula, SimScript, etc.). This description includes the set of state variables of the model, the set of events that affects the system and the actions to execute when an event is consumed. The other part, or simulation engine, is model independent. It deals mainly with the management of the events: insertion of events in the calendar and delivery of stored events in timestamp order [DMS93].

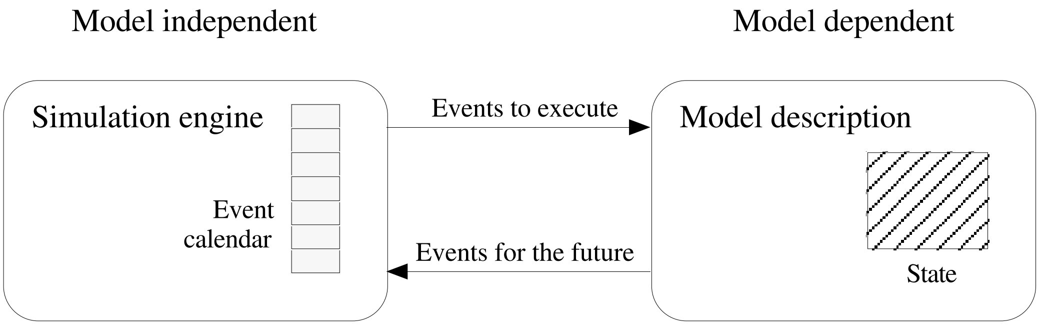


Figure 2.3. Sketch of a sequential simulator, where the description of the model is separated from the event management functions.

It is very important to realize a good implementation of the simulation engine, specially in the event-driven case, because the investment will be worthwhile, as it will be useful for many simulations. A key piece to consider in this part of the design is the implementation of the event calendar. This data structure is the core of the simulation, and a bad implementation can introduce an unacceptable complexity in the simulator. Many algorithms can be found in the literature about this topic, where highly optimized data types are proposed to implement efficient insertions/extractions of messages in/from the event calendar. A good survey can be found in [CSR93].

2.3 Parallel simulation alternatives

As computational demands grow and parallel computers become more popular, it is clear the need to adapt the classic sequential simulators to the new architectures. In the literature, several proposals can be found that take advantage of parallel systems, and some of those are reviewed here.

An immediate way of using the parallelism offered by a multiprocessor or a multicomputer is simply to simultaneously run as many simulations as processing elements are available [Fuji90b, RW89]. Usually, the study of a system needs multiple executions of the simulator, in order to analyze the system under different sets of input parameters or to reach a given confidence interval in the results. In these cases, *replication* of simulations is very efficient. However, the analysis of many system requires a search for a particular set of input parameters, and the output of a simulation run is needed precisely to refine the selection of parameter for the next run. Obviously, in this case the replication approach is not valid. An additional problem of replication comes from memory restrictions: many processing elements might be available, but often they do not have enough memory to keep a complete description of the system and work with it, so replication is simply impossible. In [Vaki92] a proposal of massively parallel simulation of independent sets of parameters is given, specifically tailored for SIMD architectures like the CM-2.

It should be apparent the need to parallelize *each* simulation run, to reduce the execution time and, at the same time, to distribute the data structures of the simulator along the available memory modules. One approach is to modify the sequential algorithms, making a *task distribution* among different functional units (specialized processing elements) [RW89]. As examples of functional units, the following can be mentioned: random number generation, event calendar management, statistics collection, input/output, file manipulation, graphics generation, overall supervision, etc. The degree of speed improvement using task distribution is quite limited, because the number of functional units that can be identified and put to work is very reduced, and so this technique is not viable for its implementation in massively parallel systems (although it can be effective if a specific-purpose machine is built).

Another approach is to parallelize the execution of each event, by dividing it into several sub-events which can be executed concurrently. This *event-parallel* approach is only useful when the simulation of the actions associated to an event leads to the execution of a highly parallelizable algorithm, and we cannot say that this is the most common situation.

The most promising PDES techniques realize a *model decomposition*, i.e., a division of the model into different sub-models, assigning those sub-models to processes which will be executed in the available processing elements. Clearly, the objective is to concurrently execute events corresponding to different sub-models. However, a mechanism must be provided to ensure that events are executed in the correct order. There are strong cause/effect relationships between events which prevent the simultaneous consumption of an arbitrary set of events. Let us consider the following scenario, where the existence of a single event calendar for all the simulation is assumed. The two first messages of the event calendar are $m_1 = <e_1, t_1>$ and $m_2 = <e_2, t_2>$, where $t_1 < t_2$. The sequential algorithm first removes m_1 from the calendar, and consumes it. The effects of executing e_1 include, among other things, message $m_3 = <e_3, t_3>$ being scheduled, where $t_3 < t_2$. In the next iteration the selected message is m_3, and its effect is to cancel m_2. If we try to simultaneously remove and consume m_1 and m_2, later on m_3 will try to cancel m_2, but it will be too late. Obviously, this situation cannot be allowed.

One approach to cope with this problem is to realize a *synchronous* PDES, where all the processes which constitute the parallel simulator have the same (or very close) view of time. If a time-driven approach is used, the simulation would proceed as follows: all the processes increment the clock in one time unit, then the events whose timestamp equals the value of the clock are executed and finally the processes synchronize (typically using a barrier operation) to prepare for the next iteration. Incorrect message consumption situations, like the one described above, cannot happen, because the simulator behaves exactly like a sequential one. If, after finishing an iteration, the synchronous parallel simulator performs (in a centralized or a decentralized way) a computation of the minimum timestamp among all the pending messages, and the clock is advanced to reach that point, the simulator is still synchronous, but event-driven. A method like this is described in [KY91].

A synchronous parallel simulator can be successful in its attempt to accelerate simulations only if the event density is high, and events are distributed evenly among all the processors. If this is not the case, in many iterations only a few (if any) processors would have work to perform. Consequently, in order to make a better use of the available parallelism, a method to simultaneously consume events with *different* timestamps is needed. Techniques that work this way are *asynchronous*, because the different elements of the parallel simulator might have a different perspective of time. Needless to say, mechanisms to perform some synchronization must be included in the simulators, to prevent the consumption of events in incorrect order.

Several techniques to perform asynchronous PDES have been proposed in the literature. They can be divided into two main groups: conservative [Brya77, CM79] and optimistic [Jeff85], with several variations. These techniques have many points in common, mainly that they are event-driven and asynchronous, but differ in the way the synchronization among the parallel processes which form the simulator is achieved. In the following sections each technique will be described, along with the most important variations or evolutions product of other researchers' work.

2.4 Basic PDES concepts

In the previous section it was mentioned that the main challenge of PDES techniques is to guarantee that the causal dependencies among events are respected. The simulation of an event cannot be allowed to affect other previously simulated events, because those situations would lead to an incorrect simulation. In this context, it can be proven that the already described sequential event-driven simulator is correct: events are processed in the right order, because in each iteration the event with minimum timestamp is selected, and this choice guarantees that the event dependencies are observed (a formal proof can be found in [Misr86]). Fortunately, this is not the only way of guaranteeing correctness in a simulation.

2.4.1 Event dependencies

It this section we will formally define the classes of event dependencies that must be observed in any event-driven simulation, sequential or parallel [Wagn89].

Definition 1: we say that event e_i *affects* the execution of e_j if at least one of these situations arise:

- The execution of e_i creates or cancels e_j.
- The execution of e_j reads or updates state information that was created or altered by the execution of e_i.

In any case, it is assumed that the timestamp of e_i is strictly less than the timestamp of e_j, because in a real system an event cannot influence past events.

Definition 2: we say that event *a causally affects* event *b* (or that *b causally depends* on *a*) if there is some chain of events $a = e_0, e_1, e_2, \dots, e_n = b$ such that, for each pair e_i and e_{i+1}, the execution of e_i affects the execution of e_{i+1}.

Definition 3: given two events *a* and *b*, if neither *a* causally affects *b* nor *b* causally affects *a*, then we say that *a* and *b* are *causally independent*. In particular, note that any two events with exactly the same timestamp are causally independent by assumption. The "causally affects" relation defines a partial order on the events in a simulation.

In a sequential simulator events are executed in non-decreasing timestamp order. It is not strictly increasing order, because several events might have the same timestamp, and those can be consumed in any order, even concurrently. This gives us the idea that *some* actions can be parallelized in the simulator. However, if it is not common to have equally timestamped events, no parallelism is available.

The objective is, then, to concurrently execute events with different timestamp. To do so we need to relax the requirement of executing events in temporal order, using instead the defined causal order. Given a traditional sequential event-driven simulator, and considering the previous definitions, a parallel simulator that executes all the pairs of causally dependent events in causal order satisfies these properties:

- Exactly the same events are executed in the parallel simulator and in the sequential one.
- When a given event is executed, the portion of the state of the system that affects the simulation of that event is exactly the same in the parallel simulator and in the sequential one.

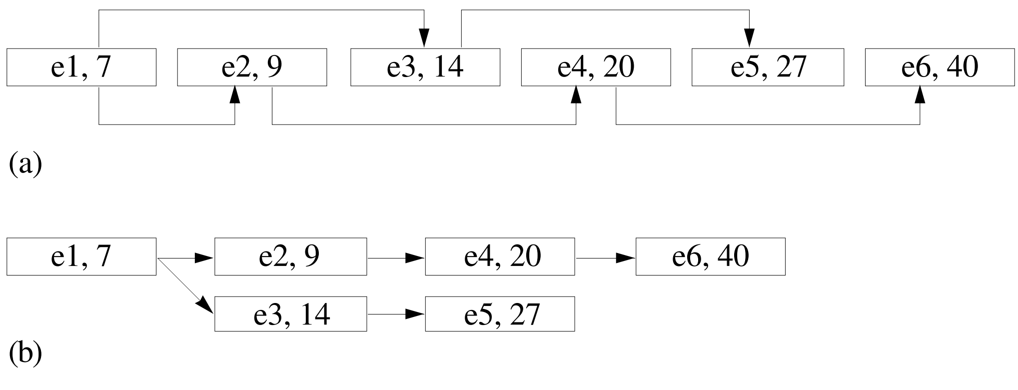


Figure 2.4. (a) A list of scheduled events, in timestamp order. The arrows indicate the causal relationship among them. (b) The same sequence, ordered by causal dependencies. In this case, the sequences (e_2, e_4, e_6) and (e_3, e_5) can be executed concurrently.

In other words, a simulation that executes events in any order consistent with the causal order is indistinguishable from a simulation that executes events in temporal order. Obviously, the temporal order imposed by a sequential simulator is consistent with the causal order, but the opposite is not always true. For this reason, imposing a temporal order in unnecessarily restrictive. The most important asynchronous PDES methods precisely try to take advantage of the more relaxed causal ordering to simultaneously execute events with (potentially) different timestamps. Figure 2.4 depicts an example of restrictions imposed by causal dependencies, and shows how the sequence of events (e_2, e_4, e_6) can be executed in parallel with the sequence (e_3, e_5). However, if any event were simulated in parallel with e_1, the causal dependencies would be violated.

2.4.2 Model decomposition

Now we will describe a set of common characteristics of the most important families of model decomposition-based PDES techniques. We consider, as defined at the beginning of the chapter, that the physical system to be simulated is composed of a set of physical processes which only interact at discrete times by means of messages. The message has two fields: the event to occur and the timestamp or time when the event should occur.

A parallel simulator is arranged as a collection of logical processes (LPs). For the sake of simplicity, we will consider that there is one LP per PP, although it does not necessarily have to be this way. The LPs do not share any kind of information among them, and the only allowed means of synchronization and information interchange is message passing. Each LP has its own local *clock*, which indicates up to what point in simulated time the evolution of the corresponding PP has been simulated.

The interaction between two PPs, modeled as one PP scheduling an event for a certain time, is simulated by means of a message interchange between the corresponding LPs. The timestamp of an event scheduled by a LP must be greater than the local clock of the LP; this is essential to maintain the causal relationships in the system.

Each LP has one or several *input queues*, where incoming messages with events awaiting to be executed are stored. Some PDES techniques use only one input queue, where all the received messages are kept in timestamp order. Alternatively, a queue can be kept for each possible source of incoming messages, without mixing messages generated in different LPs. In either case, the LP selects, as the candidate to be executed, the event with minimum timestamp among those awaiting in the input queues. As it happens in the sequential simulator, the effect of executing an event includes the advance of the clock (in this case, the local clock) to reach the timestamp of the event. Additionally, the state of the PP (and thus, of the LP) might be changed and new messages can be sent to other LPs.

It is important to remark that there is not any global information shared by the set of LPs. In particular, there is not a global clock but a collection of local clocks, which might not have the same value at a given instant of real time; similarly, there is not a central event calendar, but a collection of input queues which play the same role. In Figure 2.5 we show a physical system composed of 3 PPs simulated by a set of 3 LPs. In this case each LP keeps a separate input queue per message source, so messages received from different LPs are never mixed.

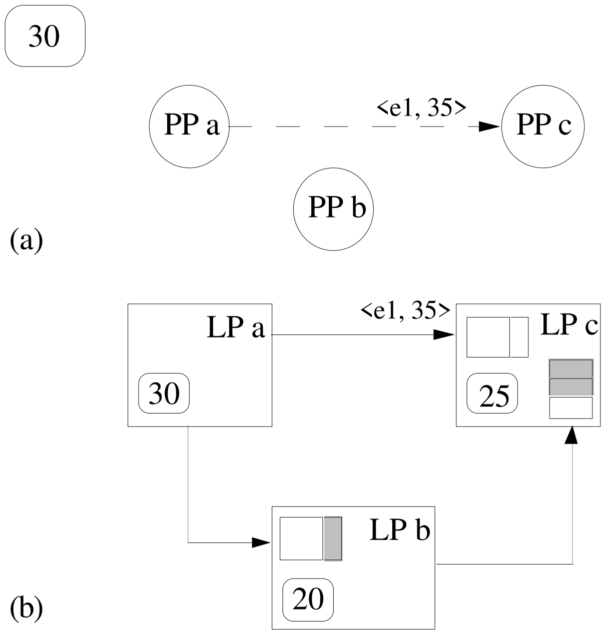


Figure 2.5. (a) In the physical system, at time 30 PP_a schedules event <e_1, 35> for PP_c. (b) Simulation of the previous situation: when the clock at LP_a is 30, message <e_1, 35> is sent to PP_c, which saves it in an input queue where only messages from PP_a are stored. The message will await there until it is time for it to be executed.

In a sequential simulation with only one event calendar it is easy to make the decision about what is the next event to execute: the one with minimum timestamp (the head of the list). However, when there are several event calendars (input queues) distributed throughout the collection of LPs, the selection of the next event is not trivial. Of course, if all the input queues of the system were examined and the message with minimum timestamp in the whole system were selected and executed by its LP, while all the remaining LPs await, the temporal order (and, therefore, the causal order) would be observed, but no parallelism would be exploited. Fortunately, this selection criterion in unnecessarily restrictive. In order to have a correct simulation, it is sufficient (although not necessary) to obey what it is know as the *local causality constraint* [Wagn89, Misr86]:

If each LP consumes messages in non-decreasing timestamp order, then the execution of the simulation is correct.

Remember that *correct* means that there are no causal errors in the simulation of events.

It is possible, while obeying this constraint, to have the collection of LPs selecting and executing events in parallel. However, this constraint is not as easy to observe as it can seem. Let us imagine the following scenario: a LP executes events $<e_1, t_1>$, $<e_2, t_2>$ and $<e_3, t_3>$, stored in its input queues, in non-decreasing timestamp order; after executing $<e_3, t_3>$ a new message $<e_4, t_4>$ is received, where $t_4 < t_3$. Clearly, the local causality constraint will be violated, because e_3 has been consumed before e_4. There is not an easy way to know what the other LPs are doing, so it is clear that if no mechanism is added to allow the LPs to interchange information about its progress, the simulation will not work.

Two different strategies can be found in the literature to ensure that the local causality constraint is kept while allowing the simultaneous execution of events with different timestamp. *Conservative* simulators guarantee that the constraint is *always* obeyed, stopping a LP when it does not have enough information from the other LPs to continue safely. *Optimistic* simulators allows an aggressive execution of events, with the effect that situations may arise where the constraint is violated in some LPs, but these situations are detected and then the affected LPs *roll back* to the past, undoing the erroneous computation, to reach a point where all the events were consumed in a correct causal order.

Any parallel computer that provides the SPMD or the MIMD models of computing allows the implementation of a parallel simulator with the described characteristics. If the communication model is message passing, as happens with the machines used in this research, the interchange of messages among LPs is implemented in the obvious way. If the system provides communication via shared memory, a library of functions to emulate message passing can be easily built. The communication infrastructure of the parallel computer must be able to support the interconnection topology of the LPs. In general, it is assumed that the communication is reliable: no message is lost, modified, duplicated, or delivered out of order.

2.5 Synchronous event-driven simulation

In this section we will describe a possible design for a synchronous, parallel event-driven (SPED) simulator, assessing its correctness and its performance potential. The description is aligned with the definitions given in the previous section, although different algorithms could be given using different assumptions. For example, we assume that the model is distributed among a collection of LPs, and simulation events are also stored in a distributed fashion along the LPs. Alternative designs could be based on a central event list, but they are not considered because this central element limits the scalabiliy of the design. Proposals like the one presented here can be found in [KY91, Soul92].

Each LP of a SPED simulator keeps the same data structures of a single, sequential event-driven simulator: clock, state variables, statistics and event calendar. Following the terminology of the previous section, we will use the term “input queue” instead of “event calendar”. The clock of all the LPs always keep the same value, so it can be said that the LPs share a common clock. The rest of the data structures are private. Only an input queue per LP is needed, where all the received messages are stored in timestamp order.

Each LP performs the following algorithm:

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

In the first step each LP obtains the timestamp of the first message of its input queue. Then, a global operation is performed to compute the minimum among those timestamps. This value is assigned to the clock of all the LPs. In the third step each LP consumes all the events whose timestamp equals the new value of the clock. The last step is needed to make the LPs start the next iteration at the same time. This synchronization must be done after all the messages generated in the previous step have been delivered and safely stored in the corresponding input queues.

From the previous description, it is clear that simulations performed by a SPED simulator are correct. Events are consumed in timestamp order. Only those with the same timestamp are executed concurrently, and they are (by definition) causally independent. The design of the LPs and the barrier synchronization ensures that the local causality constraint is always obeyed.

Regarding the performance of this simulator, it is guaranteed that at least one LP will consume one event in each iteration: the one that was used to compute the new clock; but this step might be void in other LPs, specially if the event density is very low or the events are not evenly distributed among the LPs. In the worst case, this simulator behaves exactly like a sequential one. However, in a well balanced scenario, it efficiently exploits the available parallelism, with a moderate synchronization cost. Two positive aspects can be found in this method: the simplicity of the design (which makes the simulator easy to build and to maintain) and the possibility of an efficient implementation in SIMD systems, while other approaches to model-distribution simulation are best suited for SPMD or MIMD systems.

2.6 Conservative synchronization

In works by Bryant [Brya77] and, independently, by Chandy and Misra [CM79], a method called *conservative* is proposed to realize parallel discrete event simulation. The name conservative comes from the way the local causality constraint is enforced: an LP must await, before consuming an event, until it is absolutely sure that no new message will arrive with smaller timestamp. To behave this way, some restrictions are imposed to the LPs:

- Each LP maintains one input queue for each possible source of messages. The interconnection topology of the LPs must be static, and known since the beginning of the simulation.
- Each LP must send messages through each of its input channels in a non-decreasing timestamp order.

The first restriction means that all the LPs which constitute the simulator must exist since the beginning, and that all the communication channels must also be defined since the beginning. No dynamic LP creation is allowed, and a LP cannot communicate with another one unless this circumstance has been foreseen.

The second restriction means that a LP_i cannot send a message *m* through the channel that communicates it with LP_j unless it is totally sure that no new output messages timestamped less than *m* will be generated. This restriction is very important to ensure that the input queues of all the LPs receive and keep incoming messages in timestamp order. It is being assumed, as already mentioned, that the message passing system reliably delivers messages in the same order they are emitted.

Once all the LPs of the simulator are defined, along with all the communication channels, the simulation can start. All the LPs execute, independently, the same algorithm. A particular LP_j keeps the following variables:

C_j: Local clock of LP_j. Indicates up to what point, in simulated time, the simulation of PP_j has been completed. It also gives a lower bound of the timestamps of the messages that LP_j may generate: it is only allowed to schedule events for the future.

cc_{ij}: Collection of channel clocks. For each communication that has LP_j as destination and LP_i as source, a channel clock is maintained in LP_j, which stores the timestamp of the last message received through that channel.

m_j: Auxiliary variable to store the next message to consume. Contains the event to execute and the timestamp of that event.

H_j: Message acceptance horizon of LP_j. Provides an upper bound on the timestamp of messages that LP_j is allowed to consume. In other words, a message cannot be consumed unless its timestamp is less or equal than H_j. This value is computed as the minimum among the channel clocks of LP_j ($\min_i\{cc_{ij}\}$).

The algorithm is as follows [Wagn89]:

```
Cj = 0;
for (each i) ccij = 0;
while (not finished) {
  while (input queues are empty) await message arrival;
  mj = message with minimum timestamp;
  Hj = mini{ccij};
  while (mj.timestamp > Hj) {
    await message arrival;
```

```
        mj = message with minimum timestamp;
        Hj = mini{ccij};
      }
      remove(mj);
      Cj = mj.timestamp;
      execute(mj.event);
    }
```

Although in the algorithm it is not explicitly indicated, it is assumed that each LP must manage its input queues. When a message is received, the LP stores it in the appropriate input queue (the selection depends on the source LP) and updates the corresponding channel clock.

A LP starts initializing its local clock and channel clocks. After that, it awaits to have some message in its queues, i.e., to have some work to do. Once one or more messages are available in the input queues, the one with minimum timestamp among them is selected as a candidate to be consumed. Before proceeding to do so, the message acceptance horizon must be computed: if the selected message falls below the acceptance horizon, it can be safely consumed; otherwise, the LP must block, awaiting for new messages.

The technique takes no risk when executing events (so its conservative character). Let us see a situation where $H_j = cc_{ij}$, i.e., the channel clock that communicates LP_i with LP_j has the minimum value among all the channel clocks and, therefore, gives the value of the acceptance horizon. The message with smaller timestamp in LP_j is m_{min}, with timestamp t_{min}. If $t_{min} > cc_{ij}$ then LP_j *blocks*, and the channel that communicates LP_i with LP_j becomes a *blocking channel*. A situation like this is depicted in Figure 2.6. It is easy to see the reason LP_j blocks: as cc_{ij} is smaller than t_{min}, it is perfectly possible to receive from LP_i a message with timestamp larger than cc_{ij} but smaller than t_{min}. If m_{min} were consumed in the current circumstances, it would be at the risk of violating the local causality constraint. The design of LP_j prevents it to do that, but forces it to be blocked for a while, maybe unnecessarily. LP_j can resume its work only if new messages are received through the blocking channel: the new message from LP_i will either be consumed, if its timestamp is smaller than t_{min}, or cause an increment in H_j, allowing m_{min} to be consumed.

It is important to notice how message arrival is the "fuel" that makes the simulation progress. An incoming message gives a LP work to do, and also triggers an increment in the receiving channel clock; this means a possibility of increasing the message acceptance horizon and, in turn, an increment in the number of stored messages that become candidates to be consumed. Note that a channel from which messages are rarely received becomes soon a bottleneck, because most of the time it will be the blocking channel, forcing the LP to stay blocked for long periods.

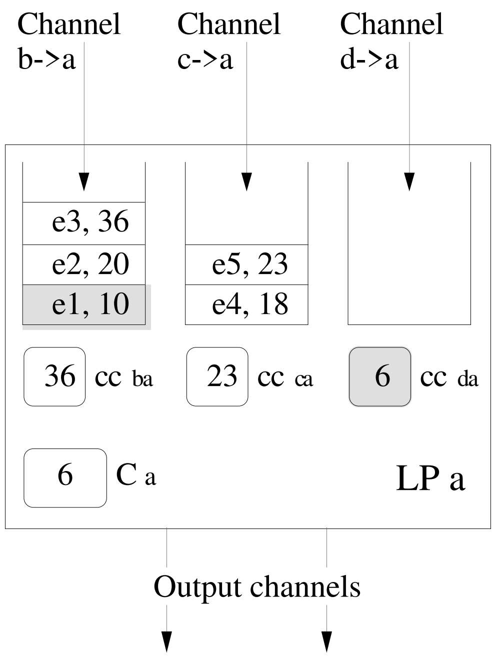


Figure 2.6. LP_a can receive messages from LP_b, LP_c and LP_d, and a separate input queue is reserved for each message source. The acceptance horizon is currently 6, the timestamp of the last message received from LP_d, which has been already executed (so the clock is also 6). Channel d->a is now blocking LP_a. Message $<e_1, 10>$ is the provisional candidate for next consumption, but it must wait: a message timestamped less than 10 might arrive from LP_d.

2.6.1 The deadlock problem

The previous conservative algorithm has a pitfall which prevents it to be really useful: situations of *deadlock*, where a set of LPs await without any opportunity to resume their job, may easily arise. To determine how these situations may appear, let us define five possible states of a LP which participates in a conservative simulation [Wagn89]:

- A LP is *halted* if its state satisfies the termination condition for the simulation, and therefore it will produce no further messages.
- A LP that is not halted is *idle* if no message is stored in any of its input queues.

- A LP that is not halted or idle is *blocked* if its message acceptance horizon is smaller than the minimum timestamp among all the pending messages.
- A LP is *deadlocked* if either (a) it is idle and all its message sources are either halted or deadlocked, or (b) it is blocked and the sources of all its blocking channels are either deadlocked or halted.
- A LP is *active* if it is not halted, idle, blocked or deadlocked.

The simulation as a whole is deadlocked if at least one LP is deadlocked and all other LPs are either halted or deadlocked.

From the definitions, two sources of deadlock can be identified. The simplest case of deadlock is the one where a LP cannot progress because it is awaiting messages from another LP that has already finished its work (it is halted) and will not generate new messages. The second case is more complex: deadlock happens because a cycle of LPs appears where each of the LPs in the cycle is blocked waiting for new messages from LPs that are in the same cycle. Figure 2.7 shows an example of a group of deadlocked LPs.

Clearly, deadlocks are undesirable because they prevent advance in the simulation. A mechanism is needed to cope with deadlocks, allowing the simulation to progress without imposing too much overhead.

The first source of deadlock is easy to avoid if each LP sends, just before halting, a special *end_of_simulation* message through all its output channels, timestamped ∞ (infinity, a value larger than any possible timestamp). This way all channels with that LP as origin will never become blocking channels.

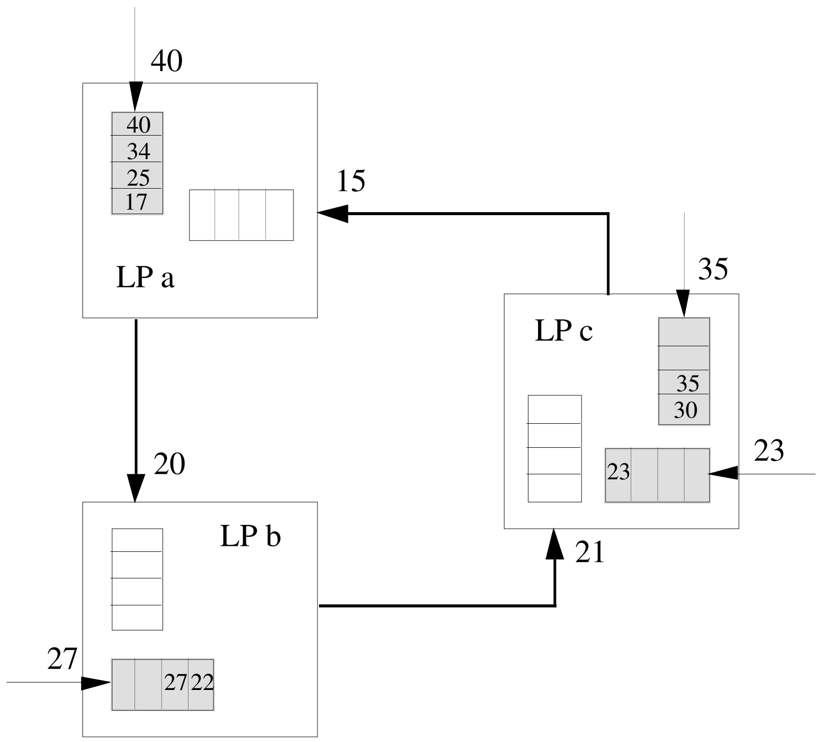


Figure 2.7. Three deadlocked LPs. The channel clocks of LP_c are 35, 23 and 21, thus H_c = 21. Channel b->c is blocking the progress of LP_c. LP_a and LP_b are in a similar condition. Note the cycle of blocking channels, where all the corresponding input queues are empty.

However, a more sophisticated mechanism is needed to cope with the second source of deadlocks. Two main alternatives can be found in the literature: *deadlock avoidance*, where deadlocks never occur, and *deadlock detection and recovery*, where deadlocks may occur they are detected and, after a recovery procedure, the simulation is allowed to resume.

2.6.1.1 Deadlock avoidance

The original works in conservative PDES propose an avoidance mechanism to cope with deadlocks. We will call this method CMB-DA (Chandy-Misra-Bryant, with deadlock avoidance). To implement this mechanism, a special class of messages are used in the simulator: *null messages*. These null messages are not directly related to the model being simulated, in contrast with the normal (useful) messages, and they do not carry an event field. However, null messages are, like useful messages, timestamped, so they affect the channel clocks and may, therefore, help advance the acceptance horizon of the receiving LPs.

To explain how this technique works, let us see how the use of null messages could allow LPs of Figure 2.7 to progress without deadlock. In LP_c the value of cc_{bc} is 21.

This means that a message timestamped 21 was received through channel *b->c*. As the associated input queue is now empty, that message has already been consumed and, therefore, the local clock is, at least, 21. LP_c is not allowed to consume any further message but it *might* know that, even if a new message timestamped 21 arrives, no output message timestamped less than 22 will be generated. To infer this, the LP *must* have some knowledge of the characteristics of the PP it is simulating.

LP_c may then send a message <null, 22> through channel c->a, meaning “I don’t know when I’ll send a useful message through this channel, but it won’t be before time 22”. The practical effect is that the reception of the null message in LP_a allows cc_{ca} to be incremented from 15 to 22, and then channel c->a is not longer blocking and the message timestamped 17 in LP_a becomes a candidate to be safely consumed.

It should be clear that a wise policy for sending null messages can effectively avoid deadlocks. A common one is to make a LP send null messages through all the output channels when it detects it is going to block (i.e., to become idle or blocked). This circumstance can easily be detected: if no message is available, or there is at least one but it cannot be consumed because it does not fall below the acceptance horizon, then the LP will block. Sending null messages will not prevent the blocking, but will allow the neighbors to progress. A null message <null, *t*> is a promise not to send a useful message timestamped less than t. The greater the value of t, the bigger the potential advance which is allowed in the receiver.

The CMB-DA approach to PDES presents two serious drawbacks. The first one is that many experiments report that the amount of null messages managed in the simulation is huge, imposing a terrible overhead in the simulation and minimizing achieved speedups (we will present some evaluations in Chapter 4). The second is more important, and even compromises the usefulness of the algorithm for some kind of models. In the previous presentation of how null messages prevent the appearance of deadlocks, a message <null, 22> was sent when the local clock of the sending LP was 21. We justified this timestamp value saying that the LP can guarantee, using information about the behavior of the model, that no message timestamped less than 22 will be sent. But, what if a LP does not have any prediction ability? If a group of LPs suffer from this absence of insight into the future, a cycle might appear with all the LPs in the cycle sending null messages to the others, without any increment in the timestamp of those messages. The simulation is not deadlocked, because null messages are consumed, but no effective progress is achieved. In Section 2.6.2 the *lookahead* concept will be introduced, which is tightly related to the prediction ability of a LP, and the relationship between this concept and the efficiency of the CMB-DA algorithm will be shown.

2.6.1.2 Deadlock detection and recovery

The fact that CMB-DA cannot be used for some kind of models led Chandy and Misra to the development of a variant of the conservative algorithm where no null messages are needed: the simulation simply progresses until it eventually deadlocks. After detecting this situation, a recovery algorithm is executed and the simulation can resume [CM81]. We will call this simulator CM-DDR (from deadlock detection and recovery). The simulation consists of two alternating phases: computation and deadlock resolution. The computation phase is done in parallel, i.e., several LPs can be consuming events at the same time, and progress until reaching a deadlock. The deadlock resolution phase can be done in centralized or in a distributed fashion [CMH83]. The easiest way of resolving a deadlock is to find the message with minimum timestamp in the whole the set of LPs, then putting the corresponding LP to run simulating this event.

During the deadlock resolution phase no progress is done, so this phase must be considered as an overhead in the simulation. This overhead is the main source of criticisms to this method: if deadlocks appear frequently, and solving them is costly (comparing to the actual work done in the computation phases), the method is not efficient.

2.6.2 The lookahead concept

When a conservative method is being used for parallel simulation, the simulator's knowledge of some characteristics of the model being simulated can be very useful to achieve good performance. If the LPs are able to predict some aspects of the future behavior of the model, they will be able to exploit this *look ahead* ability to reduce synchronization overheads [Fuji88].

In a conservative simulation, when a LP_i schedules an event e to occur in LP_j at time t, it sends the message $m = \langle e, t \rangle$. That message must be kept in LP_i until it is absolutely certain that no new message timestamped less than t will be generated. This is needed to maintain the restriction of sending all the messages in timestamp order. Due to this way of working, a certain amount of time will pass since e is scheduled until m is actually sent. The length of the time interval depends on the ability of the LPs to predict its future (look ahead). We can informally define the term *lookahead* as a measurement of the prediction ability of a LP. If the lookahead is large, the message can be sent soon, and this will be good for the receiver, as it will be allowed to increment the corresponding channel clock—hence opening opportunities for a faster advance. The bad news is that, in general, the value of the lookahead is a complex function which varies with time, and it is absolutely dependent on the details of the model being simulated.

The importance of lookahead is specially clear in the CMB-DA algorithm. When LP_i is next to block, it sends null messages to all its neighbors, with a timestamp calculated as $t = C_i + L_i$ (with $L_i > 0$), i.e., the value of the local clock plus a certain increment L_i. This increment is precisely the lookahead: LP_i is predicting that no useful messages will be sent through the output channels until time t. All the output pending messages timestamped less than t can safely be sent, and the receivers of the null messages can increment the corresponding channel clocks to reach t.

It is obvious that, the larger the lookahead, the larger the timestamp of the null messages, and better opportunities are given to the neighbors to advance faster: a faster increment in channel clocks means a faster increment in the acceptance horizon and, in turn, a larger amount of messages that can be safely consumed without fear of violating the local causality constraint. In other words, when null messages are used to make the simulation advance without deadlock, the level of advance is proportional to the lookahead of the LPs. An increase in the lookahead of the LPs reduces the number of null messages (an important overhead) and accelerates the simulation.

After this description of the conservative PDES technique, Figure 2.8 shows a graphical representation of a conservative LP. Again, the model description (model dependent part) can be separated from the simulation engine. It is important to note that the model description to use in a CMB simulator can be the same one used in a sequential simulator, provided that some restrictions are obeyed (e.g., that no global data structures are accessed). The simulation engine now is *not* totally model independent: in order to achieve a good performance, the lookahead of the model must be exploited, so somehow this information has to be obtained.

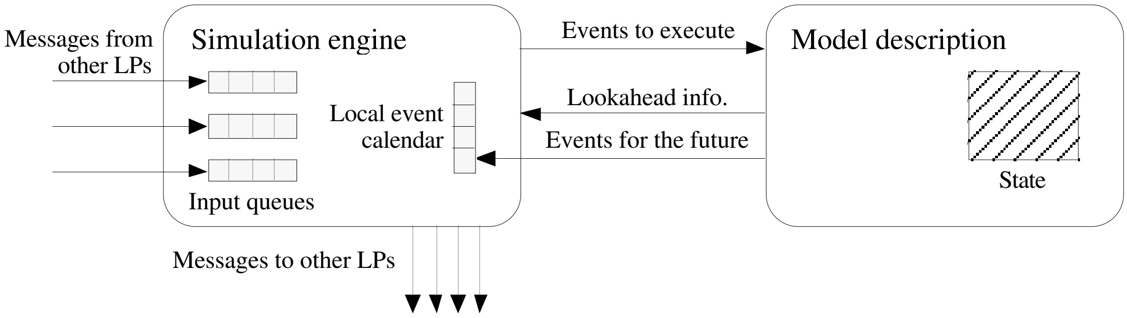


Figure 2.8. A LP in a CMB simulator. The simulation engine deals with the messages (events) scheduled for/by the LP, including null messages. The other part includes the description of the PP simulated by the LP.

2.7 Optimistic synchronization

Due, among other things, to some undesirable characteristics of the conservative PDES methods, an alternative proposal for LP synchronization was proposed by Jefferson, under the name of *Time Warp* [Jeff85]. This and similar approaches are know as *optimistic*, because they try to exploit parallelism in a most aggressive way. The two main criticism to conservative methods that led to the development of TW were:

- The need of an static LP interconnection topology. This restriction of conservative methods is too strong for the study of models where objects are dynamically created or destroyed. The only way to cope with this kind of models in a conservative simulator is to create, since the beginning of the simulation, all the LPs that are foreseen to be needed, plus all the necessary interconnections between them. If the number of LPs is large and many of them are not active all the time, the imposed overhead may be huge, in space and in synchronization effort.
- After performing some evaluations, it was seen that conservative simulators do not perform well when working with models that, in theory, are highly parallelizable. Conservative methods guarantee the absence of causality errors, but with the side effect of introducing deadlock risks, and the price to pay to solve this problem (in synchronization terms) is too big.

Conservative and optimistic methods differ in the way they cope with causality errors. Conservative algorithms totally avoid the errors, while optimistic algorithms *allow* errors to happen, i.e., they do not ensure that the local causality constraint is always obeyed. Each LP consumes messages as fast as it can, without worrying about causal problems. It may happen, however, that a new message arrives to a LP with a timestamp smaller than the local clock. Obviously, the local causality constraint has not been obeyed, and the simulation correctness is at risk. Messages that allow the detection of causal errors are known as *stragglers*.

Once the straggler has been received, the LP must continue its work, but from a point in the (simulated) past when there was no causal error. To do so, a *rollback* mechanism is implemented, which undoes part of the simulation. The LP jumps to the past, to a time equal or less than the timestamp of the straggler, consumes the straggler (now in the right order) and then resumes its work, consuming the messages stored in the input queue in timestamp order.

During the period of time while the temporal order was not obeyed (that between the timestamp of the straggler and the value of the clock when the straggler was received), surely the LP has modified its state variables and has sent messages to other LPs. The rollback procedure must jump back to an error-free situation, which means (1) recover an error-free collection of state variables and (2) cancel all the messages sent during the erroneous computation. To be able to do so, the LP must (1) periodically save copies of its state, just in case they are needed, and (2) maintain a log of all the generated messages. The last part is done keeping a list of *antimessages*, negative versions of normal (positive) messages. A message and its corresponding antimessage have the ability to *annihilate* each other.

During a rollback, a LP sends an antimessage per each erroneously sent message. When a LP receives an antimessage, it searches in its (single) input queue to see if the corresponding positive message has been already consumed. If not, then both messages are destroyed and no additional action is needed. However, if the positive message has already been consumed, an error arises, and the receiving LP must rollback to recover a point in the simulation before the positive message was consumed, and then annihilate it. This rollback might cause, in turn, new antimessages to be sent. After a sequence of rollbacks, the LPs will eventually reach an error-free state.

Regarding the limitation that conservative methods impose in the interconnection topology of LPs, TW is free of such a restriction. Only one input queue is kept in each LP, where all the received messages are merged and stored in timestamp order, independently of its source. It is assumed that the computer system that gives support to the simulation is able to provide communication among any pair of LPs, without needing to make a previous reservation of resources.

2.7.1 The basic Time Warp algorithm

The following data structures are needed at each LP to execute the Time Warp algorithm:

- One *input queue*, where all the received messages are stored. This queue is kept in timestamp order. All the messages must be stored: those awaiting to be consumed, and those already consumed. The next message to consume, thus, need not be the first on the queue. A variable called *next_event* points to the first non-executed message, i.e., the next candidate to be consumed.
- A *state queue*, which keeps past copies of the state of the LP. The simplest saving strategy is to make a copy of the state and store it in the state queue immediately before the consumption of each message. There exist more optimized state-saving policies, though.
- An *output queue*, where an antimessage is stored per each generated (positive) message.
- The local *clock*, which indicates up to what point the LP has progressed in the simulation of its part of the physical system. This is the *virtual time* of the LP.

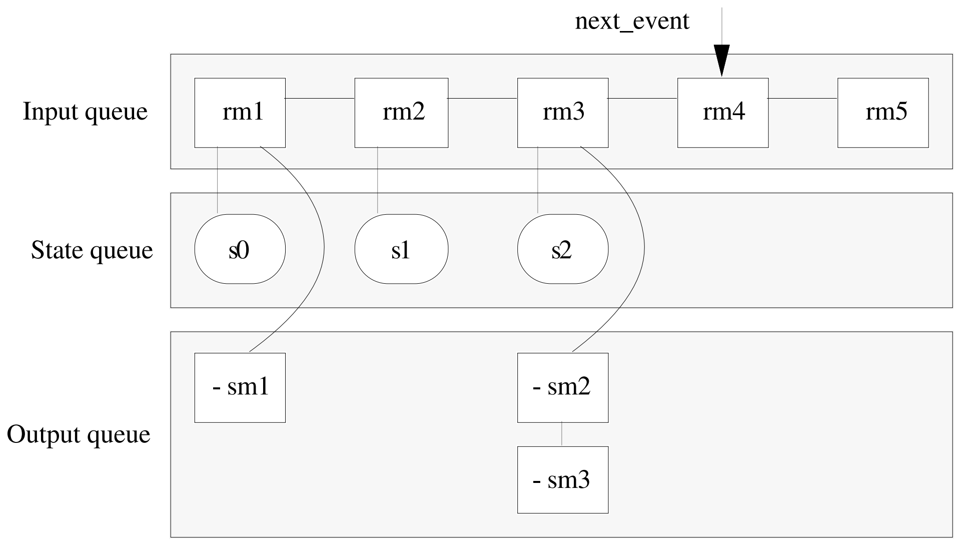


Figure 2.9. Data structures managed by a LP in TW. Before rm_3 was consumed, a copy of the state was saved in the state queue (s_2). After executing rm_3, messages sm_2 and sm_3

were sent to other LPs, so antimessages -sm_2 and -sm_3 have been stored in the output queue. Everything is now ready to execute rm_4.

Figure 2.9 sketches these data structures, along with the relationships between them. The sequence of actions that each LP executes is as follows:

1 If no unprocessed message is awaiting in the input queue, wait for new arrivals and then go to the next step.
2 Make a copy of the current state and save it in the state queue.
3 Consume the message pointed by *next_event*, i.e., advance the local clock, change the status according to the class of event, and send new messages to other LPs.
4 Add an antimessage to the output queue per each message sent in the previous step.
5 Advance the *next_event* pointer. Go to step 1.

This algorithm can be interrupted each time a message arrives. The received message can be positive or negative, and can belong to the past (if its timestamp is smaller than the local virtual time) or to the future (if its timestamp is larger than the local virtual time). Depending upon the circumstances, one of these four actions must be taken:

Positive message for the future. This is the common case in any event-driven simulator. The message simply carries an event scheduled for the LP's future. It is stored in the input queue, in the right position according to its timestamp.

Antimessage for the future. This is a kind of cancellation. The corresponding positive message is searched for and located in the input queue (it must be there, if the communication system delivers messages in order), and both messages (positive and negative) are annihilated.

Positive message for the past. This is a straggler. The local causality constraint has not been obeyed, so a rollback is needed. All the effects of simulating messages timestamped more than the straggler must be undone, to be re-executed after consuming the straggler. The straggler is inserted in the input queue. The state is restored to the copy saved just before consuming the message that now follows the straggler in the input queue. All the copies of the state following the restored one are destroyed. All the antimessages generated during the erroneous computation are sent. The *next_event* pointer is set to point the straggler. After all these steps, normal simulation can resume.

Antimessage for the past. The corresponding positive message (already consumed) is located in the input queue, and both messages are annihilated. A rollback must be done, recovering the state associated to the destroyed positive message, destroying other copies of the state and sending the necessary antimessages. The *next_event* pointer is set to point to the message just after the annihilated one. Normal computation can resume. Figure 2.10 depicts this situation.

It is to see how, when a LP receives a straggler, the rollback effects rarely are confined into that LP. Instead, a kind of "chain reaction" is started: the LP rolls back, which means that some antimessages are generated (as described in case 3 before) that, in turn, will probably cause rollbacks in other LPs (as described in case 4 before). The erroneous computation will progressively be undone, and eventually a correct state will be reached in all the LPs.

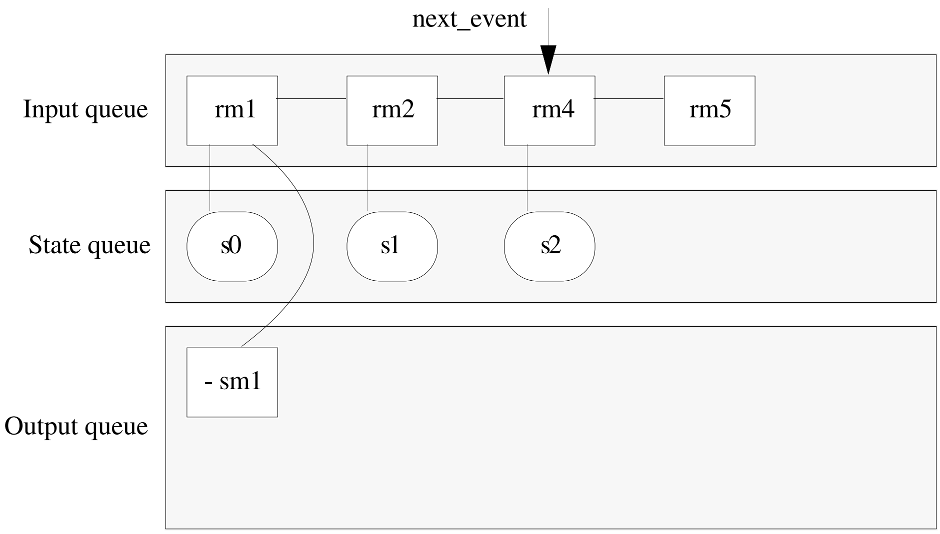


Figure 2.10. Effect of receiving antimessage -rm3 before consuming rm_4 (see Figure 2.9). State s_2 has been recovered, which corresponds to the situation of the LP just before executing rm_3. This message is annihilated by $-rm_3$. As part of the rollback, antimessages $-sm_2$ and $-sm_3$ have been sent. The rollback has finished; simulation can resume, executing rm_4.

Figure 2.11 represents an optimistic LP. Again, the model description is the only part dependent on the model, while the simulation engine is totally independent, valid for any simulation. However, a TW simulation engine must be able to access the state of the PP, to make copies of it or to restore it to a previously stored value.

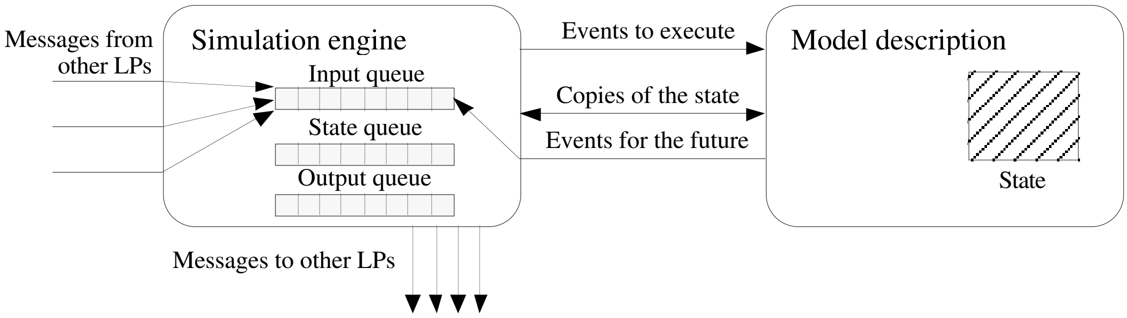


Figure 2.11. A LP in a TW simulator. The simulation engine deals with messages (events). The other part includes the description of the PP simulated by the LP.

2.7.2 Global control

Although most of the operations of the TW algorithm are done in a distributed fashion, with the LPs evolving autonomously, the system cannot work unless a series of global operations are done, satisfying these requirements:

- Guarantee that simulation advances, even taking rollbacks into account. The local clock at a LP is not an accurate estimation of the actual situation of the simulation: an unexpected straggler might arrive, making the LP jump back to the past. A mechanism is needed to establish a fixed point in time, in such a way that no jumps before that time will ever happen.
- Detect the end of the simulation. When a LP reaches the *end_of_simulation* time, it does not mean that it can finish: again, the possibility of a rollback exists, and some work might need to be re-done.
- Realize input/output operations (usually, read/write operations into files and terminals) without risk: if a LP needs to do an I/O operation at a given time, it must be sure that the operation will not need to be canceled due to a rollback. In general, I/O operations are considered as non-cancelable, therefore they cannot be committed until it is safe to do so.
- The last and more important problem to solve is memory management. This is probably the most complex part of TW. From the descriptions of the data structures managed by the LPs, it can be deduced that those structures grow unboundedly while simulation advances: all the messages are stored, a copy of the state is done before each message execution and an antimessage is stored for each sent message. All this information is stored because it might be needed to realize a

rollback. However, the amount of memory available to the LP is finite (sometimes it is quite small), and this limits the growth of the data structures.

To help solving all these problems a TW simulator needs, in addition to the collection of LPs, a *global control* mechanism whose purpose is to keep an up-to-date measurement of the *global virtual time* (GVT). This global time indicates up to what point (of simulated time) the simulation has been done, with a global rather than a local point of view. It is computed as the minimum among the timestamps of all non-executed messages in the simulator (see Figure 2.12).

Taking as a restriction that the consumption of a message can *never* affect the past, it can be guaranteed that it is not possible to do a rollback to a time before the GVT. Therefore, all the memory space associated with events timestamped less than the GVT can be safely retrieved, because it will not be needed. This includes past messages, the copies of the state stored before the execution of those messages and the antimessages stored as an effect of the execution of those messages. This process of retrieving memory space is known as *fossil collection*. Additionally, non-cancelable operations (such as I/O operations) can be safely committed when the GVT reaches the time when they must be done. The problem of signaling the end of simulation can also be solved the same way: simulation ends when the GVT reaches the *end_of_simulation* value.

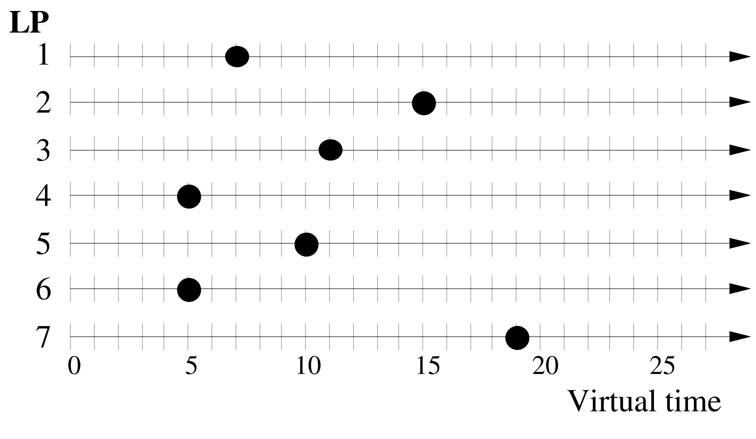


Figure 2.12. A snapshot of the local clocks of a collection of LPs in a TW simulator. The GVT is 5. All the computation done by the LPs with timestamp smaller than 5 is consolidated, because it will never be undone. The remaining computation is speculative, and may be undone by a rollback [FK91].

The complexity of the memory management in the LPs, and the need of a global control, makes implementations of TW quite tricky. In comparison, SPED and CMB algorithms are much simpler. Additionally, TW needs much more memory space to work properly. Although some researchers demonstrated that a TW simulator can work with a very reduced memory space, this does not mean that it will work efficiently. On the other hand, TW does not require the LPs to have a knowledge of the model being simulated to work properly (as it was the case with conservative simulators). A TW simulation engine may be designed and implemented just once, and then be used to simulate many different models without any special tailoring. This approach has been adopted by Jefferson's team, and led to the development of the Time Warp Operating System [Jeff87], which they use to realize their experiments.

2.7.3 Variations on the basic TW

In [Fuji89b], Fujimoto characterizes four sources of overhead which appear when TW is used to do parallel simulations, in comparison with an equivalent sequential simulation. Those are:

- Keeping a log of the history of the LPs. That is, keeping the input queue, the state queue and the output queue.
- Message passing. This is common to all the PDES techniques based on a distribution of the model among a collection of LPs. The overhead is not only the effort of passing messages, which can be very costly depending on the computer and the message passing software being used, but also the time to prepare them and extract information from them.
- Cancellations, rollbacks. One rollback does not impose a big overhead, but in general rollbacks do not appear alone: one straggler might cause an avalanche of rollbacks, and this in turn means the movement of an important number of antimessages.
- Erroneous computations. All the (real) time that a LP devotes to execute events whose effects are undone afterwards is lost time.

Once the problems have been characterized, solutions might be searched. In the literature, several proposals can be found which try to improve TW by reducing its sources of overhead [SSH89, LP91, GT93].

2.7.3.1 Lazy cancellation

In the TW algorithm previously described, which will be denoted as *basic TW*, during a rollback a set of antimessages is immediately sent, one per positive message generated during the erroneous computation. This policy of sending antimessages is known as *aggressive cancellation*. An alternative to aggressive cancellation has been proposed, known as *lazy cancellation* [Fuji90a, Fuji90b]. This approach tries to minimize the overhead imposed by the treatment of antimessages and, at the same time, to reduce the chain reaction effect of the rollbacks.

The optimization is based on temporally holding the antimessages to be sent as a consequence of the rollback. Instead of sending them immediately, the LP monitors the positive messages it sends during the normal advance phase which follows the rollback. If it sees that a newly generated message is identical to another generated during the erroneous phase, then the first message can be considered as correct, the antimessage need not be sent and the new positive message can be destroyed. If the described situation is common, i.e., many of the messages generated by a LP are correct even when the LP is violating the local causality constraint, the advantages of lazy cancellation are obvious: less antimessages are sent, and less rollbacks are triggered. However, in some cases lazy cancellation can be worse than aggressive cancellation. It requires additional overhead, and may allow erroneous computations to spread further than they would under aggressive cancellation [Fuji90b].

2.7.3.2 Lazy re-evaluation

Basic TW also performs *aggressive re-evaluation*, which means that past copies of the LP state are immediately removed during the rollback procedure. A lazy re-evaluation [Fuji90a, Fuji90b] approach also exists; in this case, copies of the state are not destroyed so promptly. After the straggler has been executed, the LP compares the copies of the state before and after that execution. If they are identical, then no further action is needed (no antimessages need to be sent, no copies of the state need to be removed), because the re-evaluation will produce exactly the same result as the original evaluation. Thus, simulation may resume at the point where it was before the reception of the straggler, without any re-evaluation of events. This is true unless new stragglers are received. The advantages of this technique are evident, provided that stragglers that do not modify the state are a majority. If this is not the case, the overhead imposed by state comparisons does not compensate the possible advantages.

Both lazy cancellation and lazy re-evaluation have an additional negative effect: antimessages or state copies are retained longer than in basic TW. On average, the data structures kept for logging purposes are longer than they would under the aggressive alternatives, so a larger amount of memory is needed to store them.

2.7.3.3 Conservative time windows

In many TW simulations it has been observed that, when a LP runs its part of the simulation faster than the others (because it runs in a faster processing element or because is less loaded), it produces the apparition of cascades of rollbacks: some straggler can roll back the fast processor, which has generated many messages which are now canceled. While the slower LPs are busy annihilating message/antimessage pairs, some of them rolling back and generating additional antimessages, the fast LP may progress forward again [NF94].

To avoid this scenario, the optimism of the LPs must be somehow controlled. A usual way of doing so is the imposition of *time windows*. For example, if the GVT is t, a LP is allowed to advance optimistically until time $t+\Delta t$. If all the messages in this window are consumed, and the remaining ones are timestamped more than $t+\Delta t$, the LP must block and wait until the window is advanced [SSH89].

The size of the window may be fixed, but then is a parameter difficult to tune: if the window is too wide, it is not effective; if it is too narrow, no optimism is allowed, and a synchronous simulation is performed. Instead of using a fixed window size, it is possible to tune it dynamically, i.e., to use an initial value and then make it vary according to the behavior of the LP [MAB94, Pala94]. The common approach is to increase the current window size if the LP is mainly doing useful work (i.e., if there is a significant advance without many rollbacks) and to narrow the window if the LP is rolling back too often. This approach is known as *adaptive time windows.*

2.7.3.4 Periodic and incremental state saving

Basic TW saves a copy of the state of the LP just before the execution of each message. This usually means that a huge amount of memory is consumed, specially if the size of the state to save is large. Using an optimization called *periodic state saving*, copies of the state are saved every N message executions, instead of after every message execution. This way memory demands are reduced considerably. However, if this optimization is included, the rollback procedure is more complex: the LP must recover a copy of the state saved before the one actually needed, and the right state must be re-constructed by means of a re-execution of already executed messages (this is called the *coast-forward* phase of the rollback). During this phase no messages are sent to other LPs. The practical effect is that less memory is needed, but more CPU time is consumed, compared to basic TW. However, as state saving is also a time consuming operation, its reduction can compensate the cost of the coast-forward phases.

Experience seems to demonstrate that this optimization actually improves the performance of the simulator, reducing execution time and memory demands [LPLL93, Pala94].

An alternative, but similar approach to periodic state saving is *incremental state saving*. With this optimization the complete state of the LP is again saved every N message executions. In the rest of the cases only incremental changes in state are saved. The coast-forward phase is then simpler: it is enough to find a full, old copy of the state and then update it by applying a sequence of increments to re-construct the required state value (Figure 2.13). It seems that, in general, this approach is more efficient than the previous one, specially when the cost of executing events is high and the amount of memory needed for an incremental state saving is low [PW93].

In either technique, we find again the problem of tuning the value of a parameter, in this case the interval between two full state copies. If this interval is too wide, the time spent saving copies of the state is reduced, but the coast-forward phase is very costly; if it is too narrow, no advantage is obtained over basic TW. As happened with the conservative time window optimization, this interval can be dynamically tuned to optimize its width, and the same tuning procedure can be used: reduce the interval if the LP is suffering from too many rollbacks, extend it otherwise.

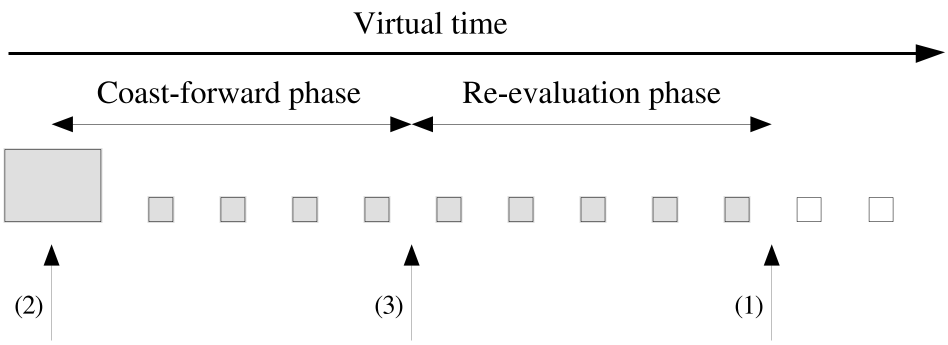


Figure 2.13. A graphical representation of incremental state saving. A big rectangle means a complete state copy, while a small rectangle means a state update. When a straggler is received (1) the LP rolls back until it finds a full copy of the state (2). Then it applies a sequence of updates until the state of the LP has the value that corresponds to a time equal to the timestamp of the straggler (3).

2.8 Conclusions

In this chapter we have introduced a series of basic ideas about simulation of discrete event systems, including two sequential algorithms to realize this kind of simulation: a time-driven and an event-driven one. It has been shown how it is not trivial to implement a parallel simulation by simply modifying a sequential one, so new approaches to the problem have been developed, based on the model decomposition concept. The simulation of a physical system is distributed among a set of cooperating logical processes, which execute the events that affect its part of the system. The collection of LP must be synchronized somehow, in order to prevent the violation of the cause-effect relationships among events.

The synchronous approach consists of making all the LPs progress at the same pace, executing in parallel only those events with the same timestamp. The asynchronous approach tries to go further, by executing in parallel events with different timestamp; a synchronization mechanism must be included, though, to avoid any violation of causal restrictions. Two asynchronous strategies have been presented: conservative and optimistic. The former totally avoids the violation of causal restrictions. The latter allows errors to happen, but recovers from them by means of a rollback procedure. Both kinds of synchronization have been studied, and some modifications which can be done to improve their performance have been also presented.

For the interested readers, many additional surveys about PDES can be found in the literature. Some of those concentrate on a particular technique, and many others try to cover a complete range of alternatives. Two main sources of information about conservative algorithms are, in addition to the seminal works [Brya77, CM79], a survey by Misra [Misr86] and Wagner's Ph.D. dissertation [Wagn89]. For optimistic methods, the work by Fujimoto [Fuji89b] is an excellent complement to the work by the author of Time Warp [Jeff85]. In the group of general surveys, recommended readings are [RW89, Fuji90b, FT94, MB95]. A recent survey by Nicol and Fujimoto [NF94] summarizes the current situation and the new trends on PDES research. In the survey by Ferscha & Tripathi [FT94] an interesting qualitative comparison of conservative and optimistic methods is done; we reproduce it in Table 2.1. As it will be seen in the rest of this dissertation, our experiences with both classes of simulators are quite in agreement with the information in this table. We will give some actual performance figures in Chapters 4 and 6.

	CMB	TW
Operational principle	Local causality constraint violation strictly avoided.	Lets local causality violations occur, but recovers when detected.
Synchro-nization	Achieved by blocking LPs while awaiting for safe messages. Deadlocks may appear, and must be avoided (CMB-DA) or detected and broken (CM-DDR).	Achieved by doing rollbacks, which may appear in cascades, imposing serious overheads in computation and in communication.
Parallelism	Model parallelism cannot be fully exploited: if causalities are probable but seldom, protocol behaves overly pessimistic.	Model parallelism is fully exploitable: if causalities are probable and frequent, it can gain most of the time.
Lookahead	Necessary to make CMB-DA operative, essential for performance	Not necessary to make TW work, but it can be used to improve performance
Balance	CMB performs well if all channels are equally utilized (there are not pathological blocking channels). Large dispersion of events in space and time is not bothersome.	TW performs well if average LVT progression is balanced along the LPs. Otherwise performance can be seriously degraded.
GVT	Implicitly executes along the GVT bound. No explicit computation is needed.	Relies on global control to compute GVT. Centralized GVT computation is bottleneck-prone and not scalable. Distributed GVT computation imposes high communication overhead.
States	Conservative memory utilization, copes with simulation models with arbitrarily large state spaces.	Performs best when state space and storage requirements are small.
Memory	Conservative memory consumption.	Aggressive memory consumption. State saving overhead. Fossil collection requires efficient and frequent GVT computation to be effective. Complex memory management schemes.
Communi-cation	Messages must arrive in timestamp order. Separation of input channels is required. Static LP interconnection topology.	Messages can arrive out of order (but executed in timestamp order). One input queue. Topology can be dynamic.
Implemen-tation	Straightforward. Simple control and data structures.	Hard to implement and debug. Complex data manipulations. Tricky implementations of control flow and memory organization. Performance very dependent on the implementation.
Performance tuning	Mainly relies on the deadlock management strategy. Computational and communication overhead per event is small, on average. Protocol in favor of fine grain simulation models. Overall performance highly dependent on the characteristics of the model.	Mainly relies on controlling the optimism and the memory consumption. Computational and communication overhead per event is high, on average. Protocol in favor of large grain simulation models. Overall performance highly dependent on the implementation and on the characteristics of the model.

Table 2.1. Qualitative comparison of CMB and TW [FT94].

It is interesting to note that, after more than fifteen years of research in PDES, with successful applications in many fields (some of those will be described in other chapters of this dissertation), the general simulation community has not embraced these techniques yet. A series of articles appeared in ORSA Journal on Computing in 1993 [Fuji93a, Abra93, Bagr93, Lin93, Reyn93, UC93, Fuji93b] considers this problem, and the main conclusion is that a big effort has been devoted to study the PDES algorithms, analyzing its behavior and proposing improvements, but still much work must be done to simplify the development of models, i.e., the work of researchers that use simulation as a tool, not as a research object. In this direction, future research lines are identified, including the following ones: application specific library packages, new simulation languages, support for shared memory, and automatic parallelization of models.

Chapter 3

Environments for parallel computing

In this chapter we introduce a series of concepts related to the architecture of parallel systems, and to the different programming models which can be used to develop applications in those systems. The objective is to stress the differential characteristics of the three parallel systems used in this research, from a hardware but also from a software point of view, and to justify our interest in the study of interconnection networks and message routers.

3.1 Introduction

In Chapter 1 we stated that our interest in multicomputers is twofold: from a hardware point of view, it is our intention to make architectural proposals for efficient multicomputer design. From the software point of view, we want to make an efficient use of currently available (and future) multicomputers, broadening the spectrum of applications that can use these architectures.

In this chapter we will study parallel computers in general, and multicomputers in particular, as platforms for the design and execution of parallel applications. While paradigms for sequential programs are well studied and understood, there is not a clear model of how parallel applications should be. In fact, careful decisions must be made to select the appropriate parallel programming model before starting with the design and implementation of an application. In many cases, however, the available computer and programming tools impose a given model, reducing the spectrum of design choices.

In Section 3.2 we will make an introduction to parallel programming from a software point of view, i.e., how a programmer perceives and uses a parallel computer. Some of the issues discussed in this chapter are purely software, independent of the underlying architecture, while some others are highly dependent on the architectural design of the parallel computer. After discussing a series of concept as MIMD vs. SIMD, shared memory vs. messages, blocking vs. nonblocking communication modes, and partner addressing techniques, the particular characteristics of the three environments used in this work are summarized. A thorough discussion of these three systems can be found in Appendix A.

In Section 3.3 we will give a brief introduction to some hardware issues involved in parallel computer design, again focusing on multicomputers. Some of the design choices have a clear impact on the performance achieved when running parallel applications. In this context, the model of message router used in Chapters 5 and 6 to test the efficiency of PDES is a case study on multicomputer design.

The chapter finishes with a series of conclusions in Section 3.4.

3.2 Parallel programming environments

One of the difficult aspects of parallel programming is that there is not a clear parallel programming paradigm efficient and easy to use for any kind of application and any kind of target computer. Some applications can be naturally described using a model of communicating processes, while in many others a data parallelism approach is more expressive. If programmers are provided with flexible tools that allow many ways of expressing and coding problems, they must make decisions about which ones are appropriate for their problems. In contrast, if the available tools provide just a paradigm of parallel programming, difficulties might arise when trying to implement applications that do not easily fit in that paradigm. Now we will discuss some of the options programmers must face when designing parallel programs.

3.2.1 MIMD vs. SIMD

Although it might be considered more a hardware than a software issue, the organization of a parallel computer very often have a definite impact on the way applications are programmed. From a software point of view, a *MIMD* (Multiple Instruction Multiple Data) system allows a set of processes to execute *separate* streams of instructions, each one on its own data. The memory space might be shared among all the processes, or might be separate for each process. In contrast, a *SIMD* (Single Instruction Multiple Data) system allows a collection of processes to execute *the same* instruction stream, each process working on a different piece of data. This second model of parallelism is appropriate for specialized applications characterized by a high degree of regularity, while MIMD might work for both regular and irregular applications.

Somewhere in between MIMD and SIMD, applications might follow a *SPMD* (Single Program Multiple Data) paradigm, which means that all the processes run exactly the same program, although not necessarily the same instruction at the same time, on separate data. SPMD is, in fact, a restricted class of MIMD.

In this research we only consider MIMD (or SPMD) applications. This restriction comes from the higher flexibility of this paradigm, and from the programming tools we have available. We consider a parallel application as a set of concurrent communication processes. A pair of those processes might run in parallel, if assigned to different processors of a physical computer, or might time-share one processing element. The term *process* is deliberately fuzzy: it might mean a Unix process, an Ada task or a POSIX thread. Each process runs a sequential flow of instructions and is able to communicate with other processes.

3.2.2 Shared memory vs. messages

Communication and synchronization are two operations needed in any concurrent programming environment, parallel or not. Two concurrent processes, even being totally unrelated, might need to compete for a shared resource, and they must synchronize before accessing that resource in order to guarantee that one waits while the other uses the resource without interferences. If the processes are cooperating to perform a common task, they might need to interchange information (communicate) in addition to synchronize. There are two basic paradigms for communication and synchronization among concurrent processes: shared memory and message passing. We will consider them separately.

If two or more processes share a common memory space, one easy way to communicate is by means of a shared variable: one process writes the variable while others can read it. Communication is achieved in a fast and efficient way. However, problems might arise when more than one process try to update a variable without any kind of synchronization. *Race conditions* are a fundamental issue when working with this paradigm. The variable used for communication has to be considered as a shared resource, and accesses to it must be somehow restricted to avoid inconsistent updates. Processes must synchronize to access that resource. Many synchronization mechanisms for shared memory environments might be found in the literature; two common ones are *test & set* locks and *semaphores*.

An alternate paradigm is message passing. In this case each process might have a separate memory space. Explicit communication functions are provided to copy one set of data (a message) from a sender process to a receiver process. Both the sender and the receiver must collaborate to actually perform the data movement: the sender performs a *send* (also called *write*) operation and the receiver performs a *receive* (also called *read*) operation. Send and receive operations might also provide synchronization capabilities, depending on its actual semantics. This topic will be further discussed later in this chapter.

The research presented in this dissertation has been done using a message passing paradigm. There are several reasons to justify this choice:

1 Message passing is a paradigm widely used in certain classes of parallel machines, specially those with distributed memory. Although there exist many variations on message passing, some of those discussed in this chapter, the basic concept of processes communicating through messages is well understood. Additionally, a message passing system might be efficiently and portably implemented in most parallel environments [MPI94].

2 The parallel simulation algorithms used in this research, based on model decomposition, are described by means of message interchange. The implementation is more direct this way.

3 The three parallel systems that have been available to perform this research (Supernode, Paragon, MPI) provide only message passing for communication among processors. Even if it was possible to select between shared memory and message passing in any of these machines, message passing would be the choice for portability reasons: porting an application from one machine to another is easier if both use the same communication paradigm.

Therefore, from now on we will focus on message passing based communication.

In some cases, it is possible to mix both paradigms in the same application. A common approach is to allow shared memory communication between processes running in the same processor (or, in general, multicomputer node) while messages are required if processes are in different nodes.

3.2.3 Semantics of send/receive operations

If the basic concepts of message passing are clear, the actual semantics of the send (write) and receive (read) operations offered by a parallel programming environment might be very different. We will try to organize those differences. Most of the following descriptions will be based on MPI [MPI94], an intended standard for message passing.

Communication primitives may be blocking or nonblocking. In the *blocking communication model*, read and write operations are done in one step. Returning from a read means that a message has been received; returning from a write means that the message has been sent, but not necessarily that it has been received. The blocking nature of receive adds a degree of synchronization to the communication primitives.

In the *nonblocking communication model* operations are done in two steps: first the read or receive primitive is *posted*, meaning that the corresponding function returns without completing the operation. A separate call must be done to actually complete the communication. In this case, it is assumed that there is a separate synchronization mechanism to inform a process when an operation finishes. For example, this might be done by the process polling for operation completion, or using signals.

In addition to the communication models, a message passing system might provide one or several of the following communication *modes*: basic, buffered and synchronous. The receive operation is the same for all these models, but the behavior of send is different, depending on what happens with the message and how long the calling process must wait for the operation to complete:

Basic send: completes when the message has been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Buffered send: completes immediately after storing the message in a local buffer. Its completion never depends on the occurrence of a matching receive.

Synchronous send: completes when a matching receive has been posted and the message interchange has been completed.

These differences in semantics make it difficult to port a message passing based application from one environment to another providing different semantics.

3.2.4 Channels vs. addresses

In a message passing system, a mechanism must be provided to allow the sender of a message to identify the intended destination, and the receiver of a message to identify its sender. Two common paradigms can be found:

- *Channels*. The collection of processes collaborating in a parallel application can be depicted as a graph, where nodes represent processes and arcs represent communication *channels*. A channel, then, connects two processes. Send and receive operations are done on channels, and the identification of the interlocutor is implicit. No communication is possible between two processes not connected by a channel. The process-channel graph can be static or dynamic.
- *Addresses*. Each process in the parallel application is identified, usually from 0 to N-1, where N is the number of processes. Communication is allowed between any pair of processes. Send operations include the address of the receiver. A receive

operation may include the address of a sender, meaning that it will complete only if a message from that particular sender is received. It is also possible to use receive with a wild card as the sender specification, meaning that messages from any sender are acceptable.

Using the channel paradigm, a pair of communicating processes may be joined by more than one channel, in such a way that different flows of information may be interchanged without being mixed. When addresses are used, it is common to have a way to add a *tag* to the messages, with the same purpose. The sender gives a tag, in addition to the destination address, and the receiver might choose to receive only those messages that have a particular tag. Another possibility is to allow a process to have more than one address.

3.2.5 Parallel programming languages & tools

In order to implement a parallel application, a programmer needs a language able to express the parallelism. Alternatively, a programmer might design a sequential application and let the compiler automatically extract as much parallelism as it can. We will not consider this second option, focusing on the design of applications where parallelism is explicit. We can identify at least three alternatives to do so: (1) parallel programming languages; (2) conventional programming languages enhanced with extensions to express parallelism, and (3) conventional programming languages with libraries of functions to deal with parallel operations.

In the first group we can find languages like Occam [Inmo89a], developed by Inmos as the preferred programming language for the transputer family of processors. A collection of processes run in parallel (or concurrently, if several of those are mapped onto the same processor) and communicate interchanging messages through channels, using a blocking, synchronous communication model. The PAR construct allows to explicitly express operations performed in parallel. The main disadvantage of this approach is that the programmer needs to learn a completely new programming language. On the other hand, the resulting executable code after compilation might be very efficient, specially if the hardware provides a good language support (as it is the case of transputers and Occam). Languages such as Ada might also be included in this group, because the language provides support for concurrency. An Ada program might consist of several *tasks* which communicate in a RPC-like fashion (RPC stands for

Remote Procedure Call), invoking special functions called *entries* in a blocking, synchronous way.

In the second group we can find tools as CC++ and Fortran M [Fost95]. CC++ is an extension of C++, where six new keywords have been added. Those keywords allow to express concurrency, communicate via shared memory, synchronize access to shared data, copy data from a process to another, etc. Globally, it is a powerful tool which allows the programmer to use many paradigms of concurrency and communication. Fortran M is a small set of extensions to Fortran which provides language constructs for creating concurrent processes and communication channels, and for sending and receiving messages. Communication is blocking and buffered.

In the third group we find libraries of functions which allow conventional languages like C or Fortran to work in a parallel environment, but without modifying the language itself. Many alternatives can be found, some of them commercial, tailored for a specific environment, and some others in the public domain with implementations for many host computers. The advantage of this approach is the use of a familiar programming language along with an available compiler—i.e., the investment is minimum. There is a price to pay in terms of efficiency, but only if the implementation of the library is not optimized for a given environment. In this group we can find the set of libraries which form part of the Inmos ANSI C Toolset [Inmo90], for transputer-based environments, the NX library provided to program the Paragon [Inte93] and several publicly available implementations of MPI (Message Passing Interface), able to work in many environments (multicomputers as IBM SP1 and SP2, Paragon, IPSC860, Touchstone Delta, Ncube2, Meiko CS-2, KSR-1, KSR-2, Convex Exemplar; SGI and Sun multiprocessors; networks of workstations from Sun, HP, DEC, IBM, SGI; networks of personal computers with LINUX or FreeBSD) [MPI94, Brid95].

In this research we have used the third alternative. The decision has been firstly forced by the available tools; secondly, using the same programming language (C, in this case) eases portability among platforms.

3.2.6 Programming environments used in this work

In this work we have used three different multicomputers, each one with a different programming environment. To program the Supernode we have used the Inmos C toolset [Inmo90]. The Paragon has been programmed using C and the Intel NX library [Inte93]. To implement parallel programs in a NOW (Network Of Workstations) we used C plus a MPI (Message Passing Interface) library [MPI94]. These environments have some characteristics in common, mainly the possibility of designing MIMD programs with message passing communication. Note that the selected programming language has been C for the three platforms; it has been this way to ease program portability. Inmos offers Occam as an alternative to C, while the NX and MPI libraries can be accessed from Fortran programs. The characteristics of each system are summarized in Table 3.1.

The programming models of the Paragon (NX) and MPI are quite similar, being the main differences the names of the communication functions and the parameters to pass. The Inmos C toolset, specifically tailored to exploit the characteristics of the transputer, is quite different: the blocking, synchronous nature of the message passing functions, plus the use of channels instead of addresses, require a higher complexity in the programs. Additionally, the Supernode is much more flexible than the other machines (many aspects are user-configurable), and the practical result is that the programmer must be concerned about much more things than simply coding the application. Details about these three machines and their programming environments can be found in Appendix A.

	Supernode	**Paragon**	**NOW**
Programming tool	ANSI C toolset, with libraries for parallel programming specifically tailored for transputer environments.	ANSI C with the NX library from Intel.	ANSI C with MPI library.
Model of parallelism	MIMD.	MIMD (SPMD preferred).	MIMD (SPMD preferred).
Communication paradigm	Message passing. Restricted forms of shared memory possible.	Message passing. Restricted forms of shared memory possible.	Message passing.
Communication models	Blocking.	Blocking, nonblocking.	Blocking, nonblocking.
Communication modes	Synchronous.	Basic.	Basic, buffered, synchronous.
Partner identification	Implicit (channels).	Explicit (addresses).	Explicit (addresses).

Table 3.1. Summary of characteristics of the programming environments used in this work.

3.3 Parallel computer design

Speaking in very general terms, a parallel computer consists of a set of processing elements interconnected by means of a communication network. Two broad groups of parallel systems might be characterized: *multiprocessors* and *multicomputers*.

The term multiprocessor is used to refer to a parallel system with shared memory, where p processors are connected to m memory modules by means of an interconnection network (Figure 3.1). The design of the network is a critical issue, because memory access times should be minimized. The most common class of networks used in these designs are buses and multistage interconnection networks (MIN), as Butterfly, shuffle-exchange, Omega and de Bruijkin networks. Another important issue in the design of multiprocessors is the cache memory: a local cache is needed at each process, to obtain a reasonable performance, and some cache coherency mechanism must be added, because a memory word might be simultaneously in several local caches. This issue, among other things, limits the scalability of multiprocessors.

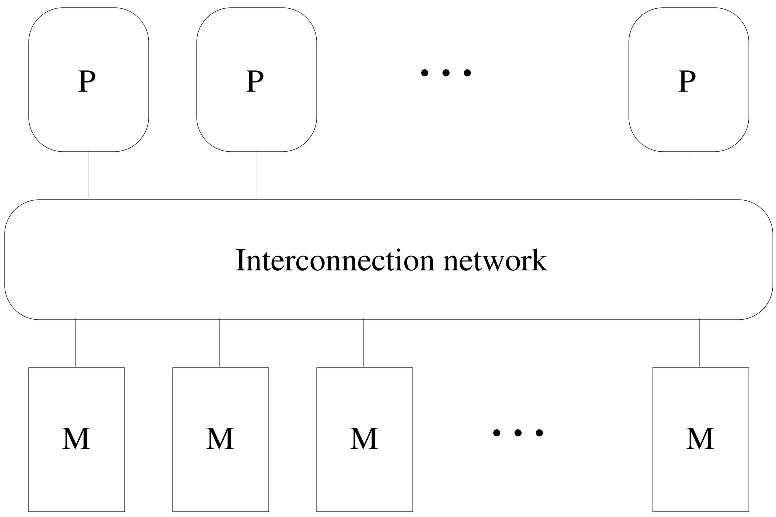


Figure 3.1. Model of a multiprocessor.

The term multicomputer applies to a set of nodes <processor, memory> interconnected by a network. Each node works as a conventional computer, and the network provides the infrastructure for communication. The communication and synchronization mechanisms are implemented by means of message interchanges through the network. The main issues in multicomputer design are the structure of the node as well as the organization of the interconnection network. We discuss these topics in the following subsections.

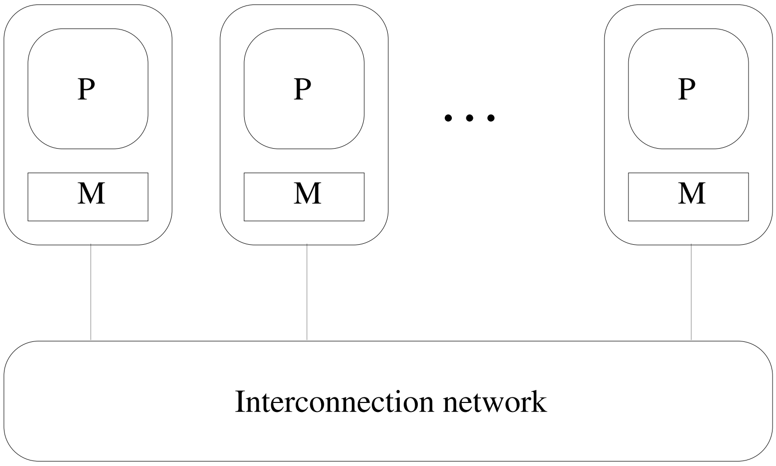


Figure 3.2. Model of a multicomputer.

The preferred communication paradigm in multiprocessor environments is shared memory, while in multicomputer environments is message passing. It is possible, however, to have the memory modules physically distributed along the nodes of a multicomputer while the programmer sees a shared memory space; a good deal of hardware/software support is needed to achieve this. In the same context, it is possible to simulate message passing over the shared memory space provided by multiprocessors.

3.3.1 Design of the node

Typically, each node of a multicomputer consists of a classical von Neumann machine: a CPU plus a certain amount of memory. Some multicomputer manufacturers use custom designs for the CPU, although in most cases general purpose microprocessors, like those used in workstations, are utilized. In some cases, the nodes of a multicomputer are actually small multiprocessors, with several CPUs and memory modules constituting a computing cluster; the interconnection network communicates clusters, instead of individual CPUs.

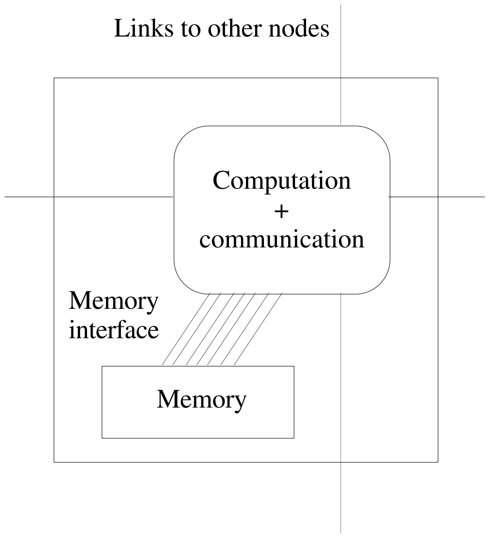


Figure 3.3. A multicomputer node where communication and computation functions are integrated in the same unit. Message relaying functions are implemented in software.

In addition to processing tasks, a node must provide some communication management functions. The kind of networks typically used in multicomputers are direct networks like hypercubes and meshes. In those networks, each node must perform certain message relay functions to allow a message to flow from its origin to its destination, traversing intermediate nodes if necessary.

First generation multicomputers, as the Cosmic Cube and systems based on transputers (as our Supernode), bundle computation and communication in a single element (Figure 3.3). The processor has to divide its time among computation tasks and relaying functions, which are implemented in software.

Current multicomputers, such as the CM-5, the Cray T3D and the Paragon, separate computation and communication tasks, providing hardware support to implement message passing functions, in such a way that these functions are assigned to a collection of hardware *routers*, while the CPUs can concentrate on computation tasks (Figure 3.4).

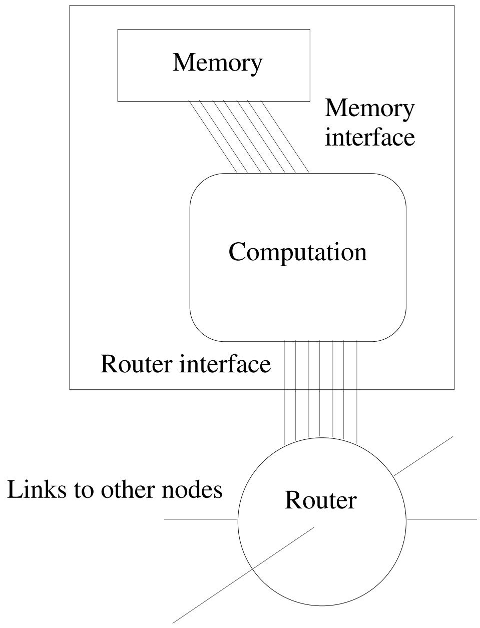


Figure 3.4. A multicomputer node where message relaying is separated from computation. A hardware router performs the message passing tasks.

A NOW can be considered an special case of multicomputer, where each node is a complete workstation and the interconnection network is typically a LAN (Local Area Network). Each workstation has a LAN interface which performs the network access functions.

In any of the three mentioned cases (first generation multicomputer, current multicomputer, NOW) the CPUs have to devote a certain amount of time to perform communication functions. Message passing has a series of overheads which might be reduced with appropriate hardware support, but which are very difficult to eliminate. Sending a message from one node to another requires a series of operations, summarized in Table 3.2 [Blum95].

The operations in the center of Table 3.2 might be efficiently performed in hardware, but there is still a good deal of software overhead. This can be very costly in systems like NOWs, where message passing functions are not implemented directly over the LAN hardware, but pass over several layers of protocols. As an example, the message passing system of the NOW used in this work requires messages to pass through three high level protocol layers, in addition to the LAN layer (in this case, an Ethernet): the MPI library, TCP and IP. This software overhead can be minimized if messages are long, but this is not a common situation when the objective is to achieve massive, fine-grain parallelism. In some machines (like the Paragon) each node consists of two CPUs, one for computation and another one devoted to perform all the message passing processing functions. The software overhead is greatly minimized, but with a significant cost increase.

	Sources of overhead
At the sender CPU	- Send system call - Argument processing - Allocate buffer - Prepare message - Initiation of send
At the network of routers (software or hardware)	- Transfer message via network interface, at origin - Transfer message over the network - Transfer message via network interface, at destination
At the receiver CPU	- Interrupt service - Buffer management - Message dispatch - Copy data to user space - Receive system call

Table 3.2. Overheads involved in a pair of send / receive operations.

3.3.2 Design of the interconnection network

The interconnection network in a multicomputer must provide a message passing support with:

1 low *latency*: messages must traverse the network as fast as possible, and
2 high *throughput*: the network must not be a bottleneck; it must be able to manage all the messages generated by the computing elements.

Other desirable characteristics are low cost, fault tolerance, expandability, and symmetry, not necessarily in this order. There are many issues to consider in order to design a network with these desirable characteristics. Some of those are:

1 *Topology* or shape of the network. Common topologies are: bus, ring, hypercube, mesh (2D and 3D), torus (2D and 3D) and fat-tree. Except for the bus, these are direct networks where communication between two parties might require the collaboration of intermediate nodes.
2 *Switching technique*: circuit switching or packet switching. Circuit switching means that a physical *connection* (reservation of links in the nodes along the path)

must be done before communication is possible. A packet switching network does not require reservation of links: each message is divided (if necessary) into a set of smaller packets, and each packet traverses the network separately, using resources like buffers and links only when they are needed. Packet switching provides a better way of sharing network resources, but suffers from some negative effects, including non-constant packet delays.

3 *Message flow control*: store-and-forward, wormhole, cut-through. Using store-and-forward, a packet must be completely copied from a node to the next in the path before it can start advancing towards the following node. With wormhole and cut-through, a packet is divided in smaller units called *flits* (flow control digits) and the forwarding process is done in a flit-by-flit basis. This means that a packet might be, at a given time, dispersed along several routers. The pipelining effect of these techniques makes the delay much less dependent on the distance between source and destination. When contention arises, cut-through is able to collapse the packet again, freeing network resources, making this technique more efficient than wormhole.

4 *Routing strategy*: static, adaptive, with many alternatives for both cases. Static means that messages traveling from a given source to a given destination always follow the same path. A common choice is to route packets in order of dimension; for example, in a 2D mesh, a packet moves first horizontally and then, when the column of the destination has been reached, vertically. Most of the above mentioned topologies provide more than one route between a given pair of nodes. Adaptive techniques try to exploit the availability of alternate routes, in order to make a better use of the network resources, or to provide fault tolerance. There are many adaptive routing techniques; some are partially adaptive and use only minimum length paths, while others are fully adaptive.

5 *Deadlock management*: necessary for some combinations of topology and routing strategy. For example, a torus network with static routing is deadlock prone. Adaptive routing techniques usually make a network deadlock prone, independently of its topology. The common approach to deal with deadlocks is to implement a deadlock avoidance technique. Such techniques are based on imposing restrictions on the use of resources, in such a way that deadlocks cannot appear. The cost to pay is a higher complexity in the router and a lower resource usage.

A detailed description of these terms is out of the scope of this dissertation. Those readers interested in these topics may check [Arru93, Izu94] and the references listed there.

	Supernode	Paragon	NOW
Node	Transputer T800	Intel i860 (two)	Sun Sparcstation 5
Implementation of message relay functions	Software	Hardware	-
Network topology	User defined	2D Mesh	Bus (Ethernet)
Switching technique	Packet switching	Packet switching	Packet switching
Message flow control	Store-and-forward	Wormhole	-
Routing	User defined	Static	-
Deadlock management	User defined	-	-

Table 3.3. Summary of the hardware characteristics of the multicomputers used in this work.

Commercially available machines offer many combinations of these parameters. Table 3.3 summarizes the characteristics of the multicomputers used in this research. Previous research performed by this group suggested that a torus network with packet switching, cut-through flow control, static routing and a deadlock avoidance technique described in [Arru93, Izu94], with all these functions performed by a hardware router, might provide a reasonable option for a multicomputer interconnection network, in terms of cost/performance. A network with these characteristics is the object of the simulation study presented in Chapters 5 and 6.

3.4 Conclusions

In this chapter we have reviewed a series of concepts related to the view a programmer has of a parallel programming system, and to the different architectural organizations that can be used to actually build such a system. The presentation has been purposefully biased towards multicomputer systems, where a set of computing nodes, comprising a CPU and a certain amount of local memory, are connected by means of a message passing network. This decision has been motivated by the computing systems available for this research, and also by the model we chose to use in our experiments.

The description of the hardware/software issues involved in parallel programming has served to introduce the main characteristics of the three computers systems used in this research: a transputer-based Supernode, an Intel Paragon and a network of Sun workstations with a MPI library. Much more detailed information can be found in Appendix A. This chapter has also served to introduce the reasons of our interest in interconnection networks and, particularly, in the design of hardware routers for building those networks. Some general ideas have been given here, while in Chapter 5 the reader will find a detailed description of a possible design for a router.

Chapter 4

An evaluation of simulation techniques

In this chapter we summarize the experimental work carried out in the Supernode multicomputer with the purpose of evaluating the performance of two sequential simulators (time-driven and event-driven) and two parallel simulators (CMB-DA and TW). A toy model consisting of a toroidal network of first-come, first-served queues was used to test the simulators. It was our aim to determine the influence that the parameters of the simulated model and the characteristics of the host multicomputer have on the performance of the simulators.

4.1 Introduction

In this chapter we present a first collection of experiments done to evaluate and compare different simulation techniques when implemented in distributed memory parallel computers. A simple model was chosen and four ad-hoc simulators were developed and tested:

- sequential time-driven,
- sequential event-driven,
- parallel conservative event-driven and
- parallel optimistic event-driven.

The objective of the study presented in this chapter was to select an appropriate parallel simulation algorithm for the analysis of the kind of models our group is interested in, i.e., interconnection networks for multicomputer systems.

For this study the Supernode system described in Chapter 3 and Appendix A was used. Up to 17 raw transputers were required for some of the experiments. Part of the results shown in this chapter can also be found in [MG93, MAB93, MAB94 and MB95]. The chapter is structured as follows. In §4.2 a review of the literature is done, summarizing the main results of studies similar to this one. Section 4.3 presents the model used in the experiments. In §4.4 a description of the implemented simulators is done. In §4.5 the behavior of two sequential simulators is studied. Section 4.6 is devoted to an exhaustive analysis of a parallel conservative simulator, while §4.7 compares this simulator with an optimistic one. Finally, some conclusions are summarized in §4.8.

Throughout this chapter, we will use TW to refer to Jefferson's Time Warp algorithm and CMB to refer to the Chandy-Misra-Bryant conservative algorithms. When necessary to make a distinction, the two main variants of the conservative approach will be denoted CMB-DA (deadlock avoidance via null messages) and CM-DDR (deadlock detection & recovery).

4.2 Related work

Several works can be found in the literature making studies very much like this one, although not identical. In this section some of the most exhaustive ones will be presented. Unfortunately, all of them have been done using shared memory parallel computers, and in most cases optimizations to exploit this kind of architecture have been used. As our research is focused on distributed memory systems like those described in the previous chapter, none of the reported optimizations could be used, so an specific analysis was needed.

It should be clear that there are many other studies of parallel simulation algorithms; the four presented here have been selected because of their generality or their similarity to our work. Some other works, focused on very specific models, will be summarized in the next chapter.

4.2.1 Reed, Malony & McCredie

This group analyzed in [RMM88] the performance of CMB algorithms when simulating several queuing networks in a Sequent Balance 21000 with 20 processors, a shared memory multiprocessor. The test included both CMB-DA and CM-DDR variants of the conservative algorithm.

Their conclusions were devastating for CM-DDR, and not much better for CMB-DA, even considering that speedups were computed taking as a reference the execution time of the parallel simulator running in one processor, instead of using a good implementation of a sequential simulator. The best figure they present is a speedup of about 7.5 using 18 processors, but in general speedups are much poorer.

Fortunately, many other works (some of them mentioned here) were able to detect a major pitfall in this study: no effort was made to exploit the lookahead of the studied models, mainly networks of FCFS (firs-come, first served) queues. After this work many other researchers concentrated on methods to exploit the lookahead of this and other networks disciplines, allowing quite impressive performance for many cases [Nico88, Fuji88, Wagn89].

4.2.2 Wagner & Lazowska

These two researchers analyzed in [Wagn89, WL87] several aspects of the implementation and usefulness of CMB algorithms for speeding up simulations. They developed the two variants of these algorithms in a Sequent Symmetry with 20 processors. This shared memory architecture allowed the introduction of some optimizations to the basic algorithm. For example, when using CMB-DA, no null messages are needed, because a LP can directly access another LP's channel clocks.

The main contribution of their work is a collection of methods to effectively exploit the lookahead of queuing network models, obtaining much better results than those presented by Reed et al. Continuing with the work by Nicol [Nico88] to accelerate the CMB simulation of FCFS servers, similar techniques are presented for other queuing disciplines such as multiclass servers with priorities, last-come first-served and processor sharing.

They test their proposals using several queuing network models, obtaining in most of the cases speedups about one half of the number of processors used in the simulations.

4.2.3 Konas & Yew

The work by Konas and Yew [KY91] analyzes three different parallel simulators: an implementation of CMB-DA, an implementation of TW and a system called Parsim, which is basically a synchronous parallel simulator. Parsim has a (single) central event list and a global clock managed by a simulation kernel. The simulation is time-driven, because the clock advances step by step. In each step, all the events with the same timestamp are processed concurrently.

In their experiments they use an Alliant FX/80 shared memory multiprocessor, with 8 processing elements. The centralized memory of the machine allows the introduction of several optimizations into the algorithms. To test the simulators they use two models:

- A *synchronous* systems consisting of a set of 16 processing elements and 16 memory modules connected via an Omega network.
- An *asynchronous* system consisting of a 4×4 torus network of FCFS servers, very much like the one we will introduce in §4.3.

For the synchronous case, Parsim clearly surpasses TW and CMB-DA. These two simulators show a similar behavior, in most cases, being TW slightly superior.

However, for the asynchronous system, Parsim is not able to extract any parallelism in the simulation, while TW and CMB-DA perform reasonably well, again exhibiting similar speedups (about 4 with 8 processors, with a significant variance depending on several parameters of the simulated model).

They conclude that the CMB method is inappropriate for the simulation of synchronous models. The TW approach can achieve good results with synchronous or asynchronous models, provided that the ratio (state saving overhead/performed computation) is low. Parsim performs well in the simulation of synchronous models, but the achieved speedup is very poor when dealing with asynchronous systems when there is not much parallelism available at each time step.

4.2.4 Fujimoto

This researcher is one of the most active workers in the PDES field. From its vast collection of papers, these four are specially interesting for our purposes: [Fuji88, Fuji89a, Fuji89b, Fuji90b]. The first two concentrate on the evaluation of CMB, while the third one analyzes the TW approach. The fourth one is an excellent survey of PDES techniques, and summarizes most of the results previously presented by this author. For this reason, we will present some results that appear in this last paper. In the experiments, a BBN Butterfly multiprocessor with 16 processing elements is used.

In the case of the conservative simulators, Fujimoto repeats some of the experiments of Reed et al., with improvements to exploit the lookahead of the models. For a queuing network model where a speedup less than one was reported, he obtains speedups about three (comparing with a sequential simulator). His main conclusion is that very good performances can be obtained with conservative methods if a careful analysis of the lookahead of the model is done. This is, however, the main criticism to this class of algorithms: an extra effort (compared to studies done with classical, sequential simulators) is required of the designer of a model in order to achieve good performances. Unfortunately, not all the models have good lookahead ability, or they have it but the way of exploiting it is not obvious.

In his last works Fujimoto has paid much more attention to optimistic methods, proposing optimizations to improve their performance when implemented in a shared memory system, being the main one the ability to annihilate previously sent messages directly accessing the memory space of the receiving LP, instead of using anti-messages.

Simulating with TW a 256-node hypercube of FCFS servers, nearly linear speedups are reported. If the queue discipline is changed to consider priorities and preemption, results are slightly worse. Not all is good news, however; he also reports that, when the size of the state to save increases, performance reduces dramatically.

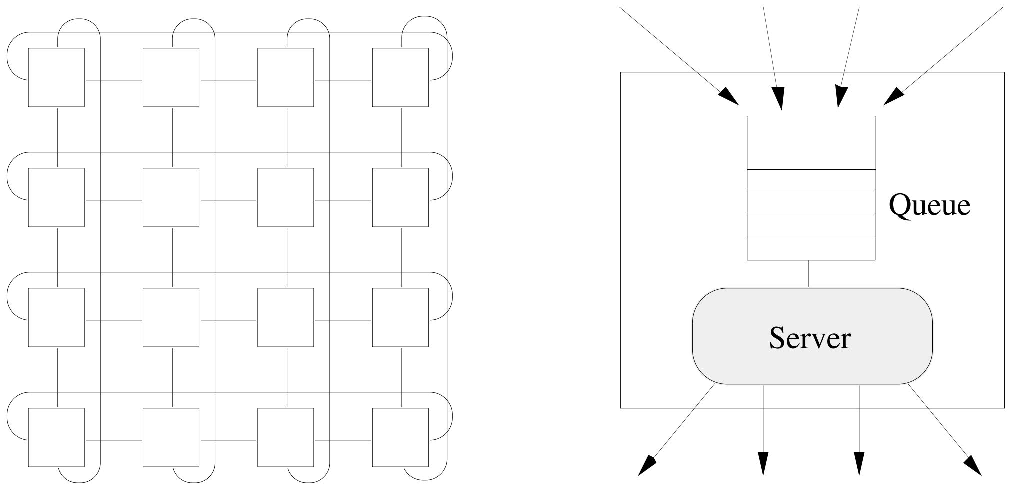
The main conclusions are that TW seems to outperform the conservative methods for most of the cases, with the additional advantage of not requiring any special effort to search for a way of exploiting the lookahead of the model under study, although the state saving and recovering mechanism can minimize the potential advantages. To reduce this effect, he even considers the possibility of building special-purpose hardware to perform this task.

4.3 The model under study

Several works of our research group have been devoted to the evaluation of toroidal networks of message routers. Some of our studies showed the advantages of using tori instead of meshes when message latency and network throughput are considered [ABIM93, Arrua93, Izu94]. Most of this research was carried out using sequential time-driven simulators. It was precisely the high cost in CPU time of these simulations the reason that conduced us to the research in the field of parallel simulation.

In this section we present a model of a network of FCFS servers which can be seen as a very simplified version of the kind of models we are interested in. We consider a network of nodes which communicate via bi-directional links arranged as shown in Figure 4.1a, i.e., forming a bi-dimensional torus. Each node of the network has, as shown in Figure 4.1b, a service center (a server) and an associated queue. The queue discipline is strictly FCFS. The network is traversed by a fixed number of jobs which visit the servers to be somehow processed. Each server removes a job from its queue (if there is any), processes it for a time (the service time) and, when finished, sends it to one of the four neighbor's queue. While a server is busy, queued jobs have to wait and incoming jobs are stored in the queue.

This model has been used as a testbed of simulation techniques in [Fuji89a] and in [KY91]. Similar models have been used in other works. For example, in [Fuji89b] a hypercube of FCFS servers was studied, while in [Wagn89] a torus network with mixed FCFS and PS (processor sharing) servers was simulated.



(a) (b)

Figure 4.1. Model used in the experiments presented in this chapter. (a) Torus network of FCFS servers. (b) Detail of a node.

The simulator might measure interesting characteristics of the model, to help to characterize its behavior: mean queue size, occupation of the server (busy time/total time), number of processed jobs per (simulated) time interval, mean queue time, mean waiting time, etc. However, we are interested in measuring the performance of our simulators, so the basic measurement we consider is the execution time of simulation runs, for a fixed amount of simulated time. This figure is taken as the main point of comparison for the different simulation approaches.

Some parameters of the model can be varied to evaluate their effects in the performance of the simulators:

- *Size* of the network, i.e., number of nodes.
- Initial *load* (C) of the network. The amount of jobs in the network is established at the beginning by inserting a number of jobs in each node's queue. A value of load C means that when the simulation starts each queue has C jobs waiting to be served. The total number of jobs in the network can be calculated as C times the size of the network.
- Distribution of the *service time*. A biased distribution has been considered, with mean value M and minimum value L. For this model, the minimum service time L can be taken as a lower bound of the lookahead of a process.
- For the parallel simulators, the number of *processors*—transputers—used.

Using this model, an important aspect to be taken into account is that the simulation of a job consumes a negligible amount of CPU time: only some increments of counters or movements of small amounts of data are needed. For this reason, when a simulator is running this model, the execution time is mainly due to the simulation algorithm itself (management of events, synchronization, message passing, etc.), and not to the simulated model (actual simulation of events). Another point to be considered is the high communication time needed to pass a message between two transputers, because the switching mechanism is store-and-forward and message management is done in software [IAB91, IAB92]. These considerations are specially relevant when evaluating the parallel simulators.

4.4 The simulators

The first considered simulator is sequential, time-driven. The network of queue-server pairs is represented as a bi-dimensional data structure. The simulator consists of a loop which traverses all the elements of this structure in each iteration. When a node is visited, a time advance of one cycle is simulated: a job is put into service and removed from the queue if necessary, a job is inserted in other node's queue, etc.

The second simulator uses the sequential event-driven algorithm. The network of nodes is also represented as a bi-dimensional data structure. Additionally, an event list has been implemented using a linked list. Some other implementation of event lists could have been better, but this one was chosen for the sake of simplicity. The simulator is also a loop; in each iteration the first event of the event queue is removed and the associated action is simulated: arrival of a job to a node, end of service in a server or end of simulation.

The third simulator is an implementation of the parallel CMB-DA [CM79, Brya77] algorithm as described in [Wagn89]. Each node of the network is simulated by a logical process, running in parallel with other LPs.

The fourth simulator is an implementation of TW [Jeff85] as described in [Fuji90b]. Again, a LP is devoted to the simulation of each node of the model. No particular performance improvement technique has been implemented, and a token-passing scheme is used to compute the global virtual time.

Once we have the set of LPs which will form the simulator, those processes must be mapped onto a set of processors (transputers, in this case). Figure 4.2 shows the mapping of a 8×8 network of LPs onto a 2×2 network of transputers. The logical channels which allow the communication between LPs have to be mapped onto either transputers' internal channels or external links. Unfortunately, the transputer does not allow the simple mapping show in Figure 4.2: it is not allowed to map more than one bi-directional channel onto one physical link. For this reason, and to provide a way of monitoring the evolution of the simulation, a set of other elements have been added to the parallel simulator to make the whole system work.

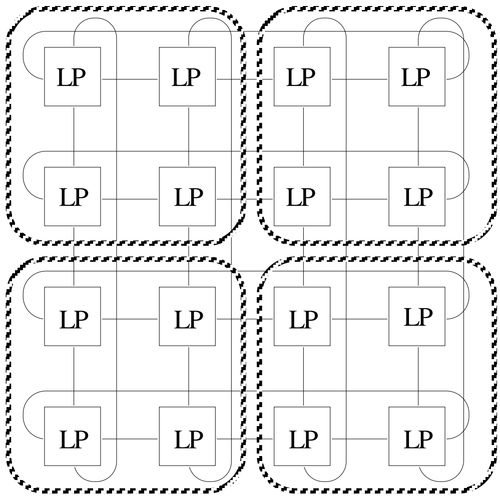


Figure 4.2. Mapping a set of LPs onto a set of transputers. Each transputer (dashed-line square) holds the same number of LPs, 4 in this example.

To run a simulator over N transputers (where $N = n \times n$), we arrange them to form a torus network, and then an additional transputer is inserted in one of the wrap-around links. A monitor process, whose purpose will be described later, runs in this last transputer, the only one directly connected to the Idris front-end (Figure 4.3a). This structure has been suggested in several works by Izu [Izu94] and by other researchers working with transputer systems. It is important to notice that the topology of the network of transputers used by the simulator *need not* be the same as the topology of the simulated network. As in the Supernode the network of raw transputers can be arbitrarily built, we selected this one because we considered it was reasonably good for our purposes.

As mentioned before, we have as many LPs as nodes in the simulated network. These LPs are evenly distributed among the available worker transputers, using the trivial mapping shown in Figure 4.2. To allow all the LPs to communicate, in each transputer there is a *router* process which manages all the message interchange between LPs, inside or outside the transputer (Figure 4.3b). When a LP wants to send a message to one of its neighbors, it transfers instead the message to its local router. If the destination process is in the same transputer, the router sends the message directly to the target process; otherwise, the message is sent to the appropriate transputer, traversing as many intermediate transputers as necessary, in a store-and-forward fashion and, finally, delivered to the destination LP by its local router. All the routing is done in software, which means that a considerable time is devoted to manage message movements.

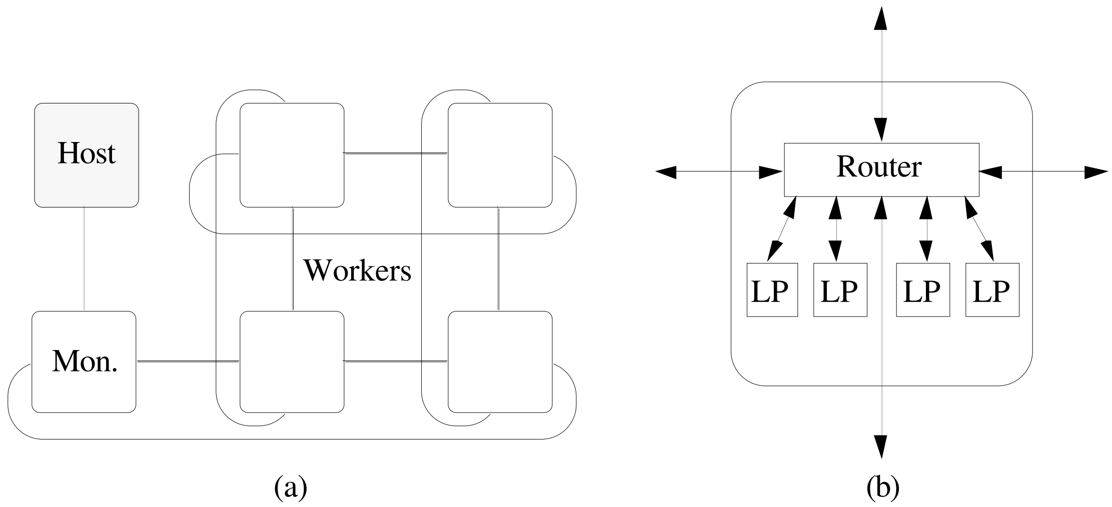


Figure 4.3. Structure of the parallel simulator. (a) Interconnection of a network of transputers to run the simulators. (b) Processes running in a worker transputer.

The *monitor* is a special process which connects the network of LPs with the external world. If a LP has a problem, a message is sent to the monitor, which prints a diagnostic on the screen. The monitor also collects all the information sent by the LPs when their part of the simulation finishes, processes it and prints a summary of the results on the screen. This is the only process with abilities to access the file system and the terminal.

4.5 Results of the experiments with the sequential simulators

This first set of experiments was done to evaluate and compare the performance of the two sequential simulators: time-driven and event-driven. We selected a set of parameters and simulated the corresponding model using one raw transputer of our Supernode.

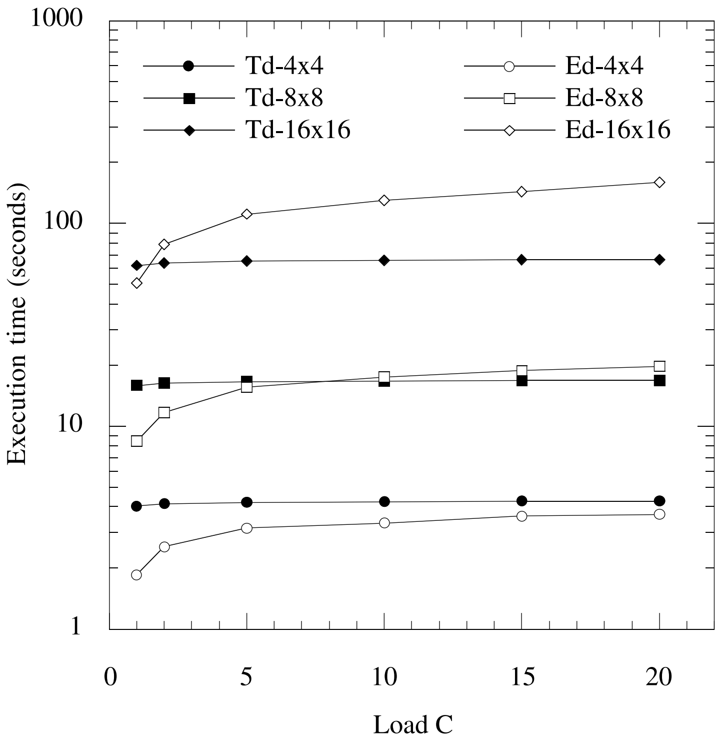


Figure 4.4. Execution times of the sequential simulators for different values of load and size of the simulated network. Td = time-driven. Ed = event-driven.

Figure 4.4 shows the execution time, measured in seconds, for the sequential simulators and for different values of load (C=1, 2, 5, 10, 15 and 20) and sizes of the simulated network (4×4, 8×8 and 16×16 nodes). Other parameters have been fixed to the following values: 10000 simulation cycles; time of service with mean M=9.5 and minimum L=1. Other values of L have also been considered, but its effect on the performance of the sequential simulators is negligible.

From Figure 4.4 some conclusions can be drawn:

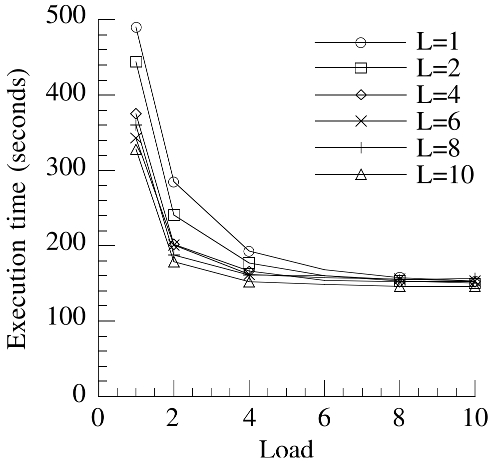
- The time-driven simulator is faster than the event-driven only when the number of events managed by the simulator in each cycle (of simulated time) is large. We will call this number the *density of events* in the simulation. Several parameters of the simulator can increment the density of events: the load C, the size of the network and the service time distribution.
- The performance of the event-driven simulator is much more sensitive to the density of events than the time-driven one, mainly due to a non-optimized management of the event queue. As the load and the size of the network increase,

the number of insertions and deletions in the event list also increases, and the same happens with the size of the list. For this reason the management of the event list becomes a bottleneck.

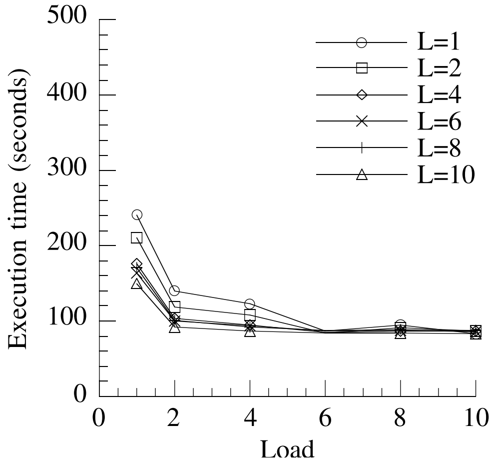
These conclusions should be taken into account when simulating a model, in order to select the right simulator. If in each cycle of simulated time many events are processed, then the time-driven simulator will perform well. On the other hand, if the time interval between events is large, then the event-driven simulator will be a better option.

4.6 Results of the experiments with the CMB-DA simulator

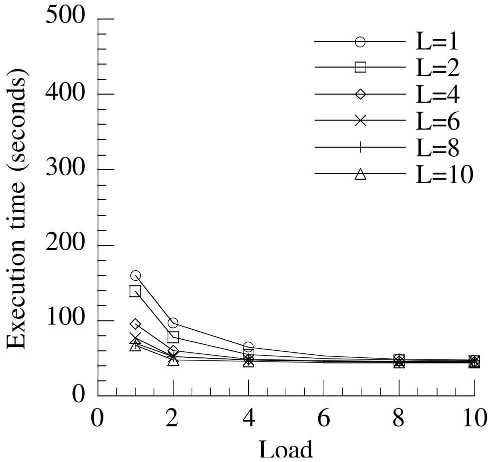
The next set of experiments has been designed to evaluate the conservative PDES with deadlock avoidance (CMB-DA) and to determine the effects that the different parameters of the model have in the performance of the simulator. Simulations with the following parameters have been executed: 10000 simulation cycles; simulated network of 12×12 nodes; distribution of the service time: M=9.5, Lookahead L=(1, 2, 4, 6, 8 and 10); load C=(1, 2, 4, 6, 8 and 10); number of transputers: 4, 9 and 16. In order to evaluate the performance of the parallel simulator, similar simulations have been done with the sequential event-driven program.



(a)



(b)



(c)

Figure 4.5. Execution time as a function of the network load for different lookahead values. Simulators with (a) 4 transputers (b) 9 transputers (c) 16 transputers.

Figure 4.5 shows the behavior of CMB-DA when the load, the lookahead and the number of processor are varied. From the figures, some conclusions can be drawn. First, it is clearly noticeable that the performance of CMB-DA is better when higher is the load of the system. When the number of jobs is large, the number of interactions between the LPs is also large, so the simulation can progress naturally: the LPs do not block often and few null messages are needed. Another characteristic of the model that plays a relevant role in obtaining good performance is the lookahead, mainly when the load level is low. Large lookaheads allow faster advance of the LPs' clocks when receiving null messages, so the synchronization effort is smaller. Finally, in relation to the obtained degree of parallelism, it clearly appears that, when a bigger set of transputers is used, a real gain in execution speed is obtained—i.e., the simulator scales well. For high loads, execution times are around 145 seconds in the 4-transputer case, 85 for 9 transputers and 45 for 16 transputers.

Figure 4.6 shows the speedups obtained when increasing the number of transputers. We use the sequential event-driven simulator as a reference point because it is not realistic to calculate speedups from the execution times of CMB-DA running in one transputer. In these figures, results with only one processor correspond to the sequential event-driven simulator, while results for 4, 9 and 16 processors correspond to CMB-DA.

It can be seen that achieved speedups are really poor in some cases (e.g., when the load is 1), even smaller than 1, but simulation accelerates more than three times when 16 transputers are used. Speedup increases as load increases. Lookahead is also important to improve the results, but its influence is clearly less important than the load's; it is relevant when the load is low and then synchronization messages (null messages) are sent much more often. It must be said that no special effort has been done in the simulator to aggressively exploit the lookahead of the model (using techniques such as those by Nicol [Nico88]), so these results could be unnecessarily pessimistic.

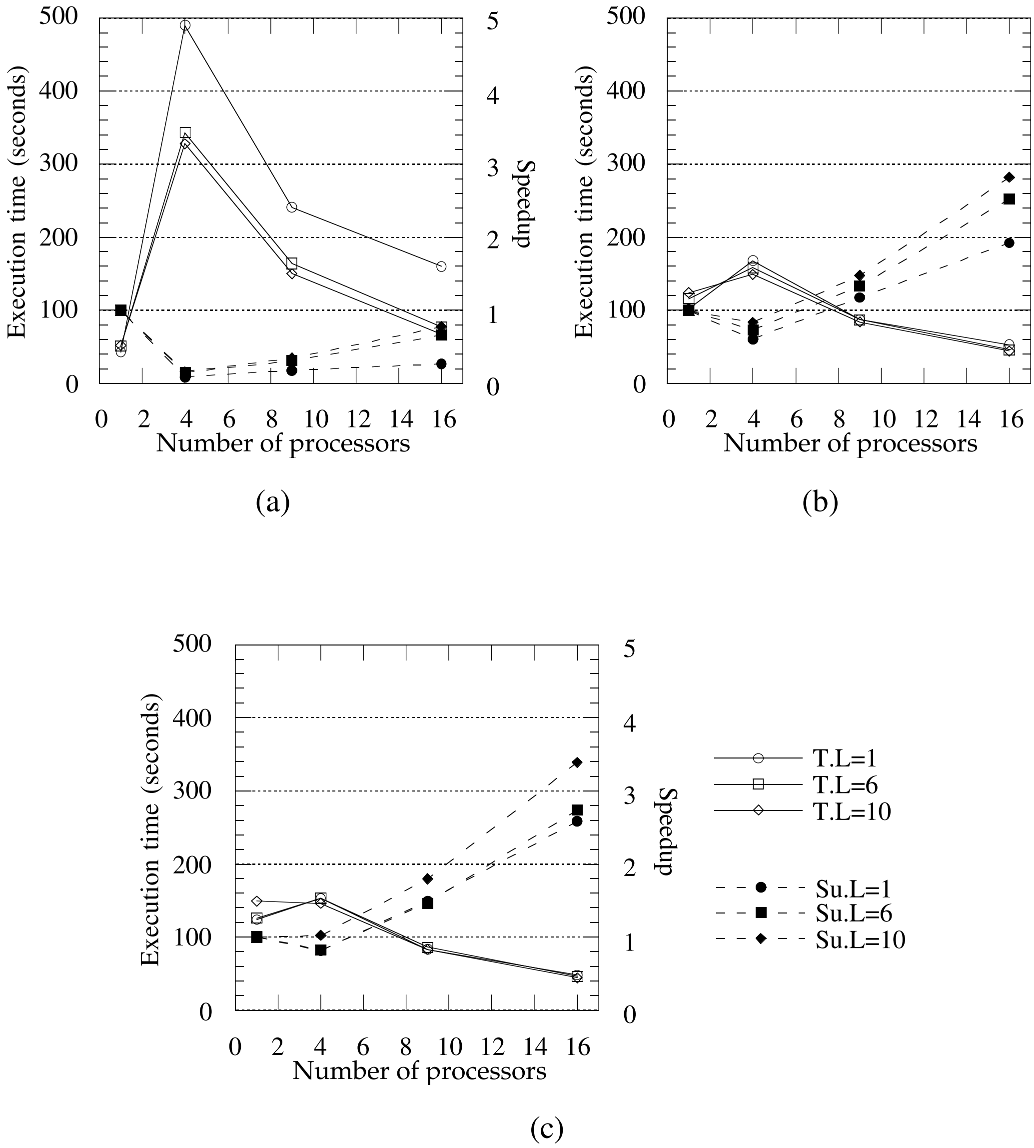


Figure 4.6. Execution time (in seconds) and speedup for different values of the load C (a) C=1 (b) C=6 (c) C=10

The obtained speedups are not really bad, specially if the consideration mentioned in §4.3 about the cost of "housekeeping" functions in the simulator, and the effort devoted to message interchange, are taken into account. In fact, most of the execution time is due to the cost of the simulation algorithm, mainly to the synchronization of the LPs, and to the interprocess communication, whereas the time to simulate an event is negligible.

We tested what happens when the actual cost of processing an event is high. To do so, we made some experiments artificially increasing this cost. A loop in the form “**for** i=1 **to** *W* **do** nothing” was inserted in the code of the simulator at the point where an entry of a job in the service center is simulated. The parameter *W* is a form of *synthetic workload*, which can be varied to evaluate its effect on the execution time of the simulator.

Experiments with significant synthetic workload exhibit very satisfactory speedups. The parallel simulator is able to distribute it among the processors, and thus available parallelism is conveniently exploited. Additionally, the communication/computation ratio becomes more balanced. Figure 4.7 shows how this workload can dramatically improve the obtained speedups. Figure 4.7a represents the execution times for two simulations (the first one with *C*=1 and *L*=1; the second one with *C*=6 and *L*=1) with the sequential event-driven simulator and a 16-transputer CMB-DA simulator. Figure 4.7b represents the achieved speedups. The addition of this workload significantly improves the achieved speedup, even when the other parameters of the model are not specially well suited for the parallel simulation.

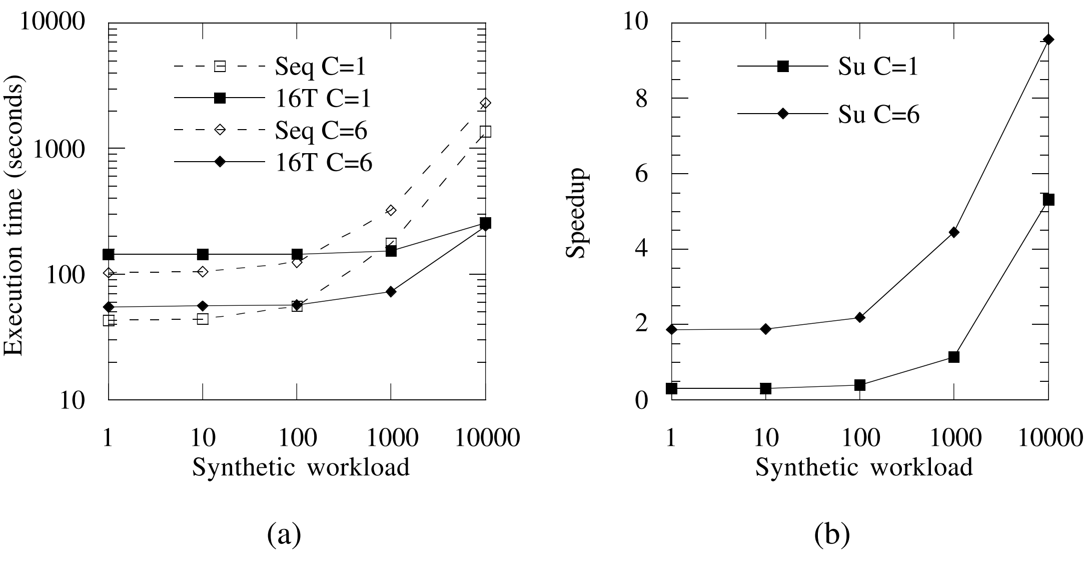


Figure 4.7. Influence of a synthetic workload in the simulation.
(a) Execution times for the sequential and CMB-DA (16 transputers) simulators, for *C*=1 and *C*=6. (b) Achieved speedups.

A series of considerations can be done about the use of a transputer-based machine as a workbench. The first one is that communication between processes *has to* be done using messages. Other implementations of CMB-DA in shared memory multiprocessors make optimizations when sending null messages: those messages are not actually sent, but the affected channel clocks are directly updated, accessing other LP's memory spaces. Additionally, software routers have to be implemented to convey messages between transputers, and those routers consume both memory and CPU cycles. Messages have to be sent from the sender LP to the router (internal communication), from one router to another (external communication, maybe several times) and from one router to the receiver LP (internal communication). The effort devoted to communication means a real burden, and has to be done independently of the type of message (null or real).

To summarize the results of all these experiments, it can be concluded that the conservative approach to parallel simulation exhibits a poor performance when simulating the type of physical model previously described. Nonetheless, other models could successfully be simulated using CMB-DA, if they belong to any (or even better, several) of these groups:

- Models that synchronize in a natural way by means of useful messages. In this case, the need of null messages is low, which is specially convenient when a message passing architecture is used.
- Models with high levels of lookahead, which allows progress in the simulation when the LPs do not have events to process. If it is not possible to process useful messages, at least null messages are generated less often and with larger timestamps, and the channel clocks (and then the acceptance horizons, LPs' clocks and simulation) advance faster.
- Models where the simulation of an event requires a high computation effort, which is distributed among the processors.

In all these cases, the ratio "useful computation/synchronization effort" is quite favorable. In general, this is the key parameter to optimize in order to get the best performance from any parallel system.

4.7 Results of the experiments with TW

In the previous section a quite exhaustive evaluation of the CMB-DA algorithm has been presented. In this section we will focus on our implementation of TW. Our aim is to discover under which circumstances TW performs better than CMB. To do so, instead of presenting a collection of figures with the same sets of parameters used before, we have selected five significant experiments, using them for our comparisons.

The parameters of the five selected experiments (**A** through **E**) are summarized in Table 4.1. The first three are cases without synthetic workload, and represent a bad, an average and a good case for the CMB-DA algorithm. The last two experiments consider the use of a high synthetic workload, where the CMB-DA algorithm performs well or even very well. Table 4.2 presents the figures for the conservative algorithm, extracted from the previous section, to be taken as a reference point. It should be remarked that results with 1 processor correspond to simulation runs with a sequential simulator, instead of using CMB-DA in only one processor. For each experiment, the left-hand column shows the execution time, and the right-hand column the obtained speedup, relative to the execution time of the sequential simulator.

Experiment	***C***	***L***	***W***
A	1	1	0
B	6	6	0
C	10	10	0
D	1	1	10000
E	6	1	10000

Table 4.1. Input parameters for the experiments reported in Tables 4.2 and 4.3.

	A		B		C		D		E	
Proc.	**Time**	**SU**	**Time**	**SU**	**Time**	**SU**	**Time**	**SU**	**Time**	**SU**
16	2’30”	0.29	0’46”	2.52	0’44”	3.39	4’16”	5.32	4’02”	9.56
9	3’32”	0.20	1’03”	1.84	0’58”	2.57	-	-	-	-
4	8’10”	0.09	2’38”	0.73	2’26”	1.02	-	-	-	-
1	0’43”	1	1’56”	1	2’29”	1	22’41”	1	38’33”	1

Table 4.2. Execution times and speedups for a series of experiments done with a conservative parallel event driven simulator.

Once we started experimenting with the TW simulator, we found that it was terribly greedy in memory demands. Many of our experiments were aborted because the memory assigned to a LP was exhausted: the LPs data structures grew too fast, and the fossil collection mechanism (made possible by means of a global control mechanism to calculate and distribute the GVT) did not free much memory to be reused. In addition to that, the message passing subsystem also failed because of a lack of space to store messages in their journey from a LP to another. After analyzing the evolution of the programs, we discovered two reasons for this behavior:

- The characteristics of the model. The toroidal network used in the experiments exhibits a high degree of local communication between neighbors. The result is that an erroneous computation in a LP spreads quickly to the neighboring LPs, making them advance in an erroneous way. What is worse, a feedback effect appears when, after erroneously processing a job, it is sent to another LP which also processes it and sends it back to the original LP, where the chain starts again. In the proposed model, this situation is highly probable. The result is that, firstly, a big deal of memory space is required to store anti-messages and copies of the state and, secondly, whenever a causal error is detected, an avalanche of anti-messages is sent, giving rise to an overflow in the message passing system.
- The transputer built-in scheduler. Many studies of optimistic simulators have been done using ad-hoc microkernels to schedule the execution of a given number of LPs in a smaller number of processors. Those schedulers prioritize the LPs with minimum local time [Fuji89b, Fuji90a]. Using this approach, the greediness of the optimistic LPs is strongly limited: erroneous computations do not go too far and, consequently, anti-messages are sent less often and in smaller number. In our simulator, instead of designing a new scheduler, we used the transputer built-in one, whose main characteristic is its attempt to be fair and distribute the CPU evenly among all the LPs.

To partially solve these problems we adopted a proposal of Sokol et al. [SSH89]: we added a certain degree of "conservatism" into the LPs. With a purely optimistic scheme, the LPs advance unboundedly (in fact, as just mentioned, they advance too fast). The conservatism is imposed by limiting the ability to go into the future: LPs are allowed to advance to a certain degree, calculated as the value of the GVT plus a *time window size*. If it tries to go beyond, it is (temporally) blocked. The window size is dynamically changed: if the LP is successful in its advance (i.e., if it does not force rollbacks), the window size is increased. Otherwise, the window is reduced. A minimum window size is guaranteed, to always have a degree of optimism.

We did the previously introduced series of experiments to evaluate our TW implementation with the time window mechanism. The results are compiled in Table 4.3. As many authors reported that TW's performance is much better than CMB's

[KY91, Fuji90b], we were quite disappointed: in most of the cases, the conservative simulator ran faster than the optimistic.

	A		B		C		D		E	
Proc.	**Time**	**SU**	**Time**	**SU**	**Time**	**SU**	**Time**	**SU**	**Time**	**SU**
16	1’48”	0.40	2’50”	0.68	2’40”	0.93	6’00”	3.78	11’51”	3.25
9	2’12”	0.32	4’00”	0.48	3’22”	0.74	-	-	-	-
4	3’40”	0.20	5’43”	0.34	5’02”	0.49	-	-	-	-
1	0’43”	1	1’56”	1	2’29”	1	22’41”	1	38’33”	1

Table 4.3. Execution times and speedups for a series of experiments done with an optimistic parallel event driven simulator.

Only in the worst case for CMB-DA (low load and low lookahead) the optimistic simulator was faster. We studied the reasons of this behavior, and we found (as other researchers did before [TV91]) that the memory management in the LPs is absolutely critical, up to the point that, in most cases, the overhead of managing the data structures is higher than the effort of synchronizing processes by means of blocking and null messages. Additionally, although null messages are not needed, anti-messages are, and its number is not negligible, especially when the optimistic advance of the LPs is not bounded. The global time computation mechanism also adds a serious burden to the simulator, because it needs a continuous interchange of messages among LPs to avoid an exhaustion of the memory space.

An important characteristic of the TW simulator is that execution times are less sensitive to parameters as load and, particularly, lookahead. This means that the user does not have to worry about the best way to exploit the lookahead of the model. The offered level of abstraction is higher. While a conservative simulator has to be done ad-hoc for a class of models, if a good performance is desired, the optimistic one is more general, and can be applied (with similar success) to a variety of models [Wiel89, WJ89].

From the results shown in Table 4.3, it can also be concluded that the speedup scales with the number of processors. Regarding the effect of the synthetic workload on the performance (experiments **D** and **E**), a large value of W has an important effect on the speedup, but this effect is not as dramatic as it was in the conservative case. The reason is that, in the conservative simulator, the busy-wait loop which emulates a high workload is done just whenever it is needed, and never has to be undone. On the other hand, with the optimistic simulator, many jobs are processed in a speculative way, and its effect have to be undone later. This means that, when the workload is big, the effect of erroneous computations is also a serious drawback: it could be better to wait and consume a message just when it has to be consumed, than to consume it and later undo its effects. As in experiment **E** the simulator has more jobs to process, the probability of making a mistake is bigger, so it takes more time than experiment **D**.

To summarize our experience with TW, we only can say that we were quite disappointed. It required much more programming effort than CMB-DA, because the management of the data structures of TW (i.e., the memory management) is anything but trivial. Additionally, it was much harder to debug. Due to the bad performance results, and to the limited number of experiments carried out, we are not able to state a definite set of characteristics of the simulated models that can help in obtaining a good performance from TW.

4.8 Conclusions

The current simulation techniques are fundamentally sequential, optimized to get the maximum performance from a single processor. Unfortunately, these methods cannot be easily adapted for its use in multicomputer systems. For this reason, new, parallel simulation methods have been developed. Among those, two promising ones are CMB and TW.

Our experiments with a conservative simulator confirms the results of other authors: high values of load, lookahead and synthetic workload make possible good performance. Nevertheless, the simulator has to be tailored to exploit the characteristics of the model. The optimistic simulator is more general, offering a higher level interface to the user, but its performance is, in our experiments, much less brilliant than expected. The memory management tasks involved in the Time Warp mechanism wastes most of the CPU time. The selected model, the computer system used as workbench, and our particular implementation allies to make the performance even worse.

Chapter 5

Simulation of message passing networks

This chapter is devoted to the use of PDES to study real world problems. We are interested in the study of a particular class of systems: message passing networks for multicomputers. We present a detailed model of one of these networks, along with the way this model can be described for its simulation. Six simulators have been developed to study this model. One is sequential, while the other five are parallel. Three parallel simulation strategies (SPED, CMB-DA, TW) have been implemented in three multicomputer environments (Supernode, Paragon, NOW).

5.1 Introduction

In the previous chapter we have presented our preliminary implementation and performance evaluation of several sequential and parallel simulation techniques in a multicomputer environment. To do that evaluation we designed a toy model, quite simple, but with a series of characteristics which were relevant to the kind of models we are interested in. The experience was useful, because it allowed us to learn about the implementation of the simulators, and to assess some of their properties. In this chapter we will try to advance further, abandoning toy models and concentrating our study on real world applications where PDES has been used with different degrees of success.

We will start by introducing some other researchers' accomplishments. The application domains considered by those authors are quite diverse: Petri nets, queuing systems, computer networks, war scenarios, logic circuits, etc. Following that, we will present our work in a particular field: the simulation of a message passing network, designed to be used as the communication infrastructure of a multicomputer. Five parallel simulators have been implemented to test three synchronization mechanisms (synchronous, conservative, optimistic) in three different multicomputer environments (Supernode, Paragon, network of workstations with MPI). These simulators are described in detail in this chapter, while the obtained performance results are presented in Chapter 6.

5.2 Related work

In this section we will review the literature to introduce significant domains of application of PDES techniques. Some of the domains are very specific: logic simulation, communication networks, computing systems, etc. Some others refer to the simulation of models described using a kind of "specification language", such as queuing networks or Petri nets, while the model itself can be anything from a supermarket to a factory.

5.2.1 Digital logic simulation

The verification of VLSI logic circuits is one of the areas where computer simulation is essential to check the circuit behavior, and has to be extensively performed before actually manufacturing a chip. As the complexity of VLSI circuits increases (and the trend is clearly in this direction), the execution time of the simulation also increases. It is clear, therefore, that any acceleration in the simulation process would be greatly welcome.

An interesting study in this field has been done by Soulé and Gupta at Stanford [SG89, SG91, Soul92]. They use parallel simulators to analyze several non-trivial logic circuits, including some circuits of the MIPS R6000 processor, the vector control unit for Ardent's Titan machine, and the cache coherence directory controller for the Stanford DASH multiprocessor. They identify the number of available tasks that can be performed concurrently by a parallel simulator, ranging from 70 to 250. This number gives a clue about the potential performance of a parallel simulator. However, it does not translate directly into speedups because of overheads such as communication between processors and load unbalance. The experimental work is done in an ideal 64-node multiprocessor, a 16-node Encore Multimax and a 16-node Stanford DASH. Three parallel simulators are implemented and tested: SPED, CMB-DA and CM-DDR. They conclude that both variants of CMB obtain better self-relative speedups (i.e., they are more scalable) than the SPED algorithm. Nevertheless, for the amount of available processors, SPED performs better. Achieved speedups using SPED range from 10 to 18 using 64 processors, while CMB only ranges from 1 to 6. Therefore, CMB is not considered a viable approach to digital logic simulation.

The work by Su [Su90] at Caltech is devoted to the development of applications for fine-grain multicomputers, using a set of CMB-DA simulations of small size logic circuits (around 1000 gates) as a testbed. Programs were run in up to 128 nodes of a Symult 2010. Su concludes that overheads of the CMB-DA algorithm are quite large if small numbers of processors are used. However, self-relative scalability of the CMB-DA algorithm is very good—almost linear, which makes CMB-DA a viable approach to simulation on fine-grain, massively parallel computers.

A research group at the University of Cincinnati studies in [Chaw94, Pala94, MW95] the applicability of Time Warp to perform logic simulations, using VHDL as the model description language and a network of workstations as the hardware platform. Most of this work is devoted to the design and implementation of improvements for TW, to reduce execution time. No speedups (relative to a sequential simulator) are given.

For those interested in the field of logic simulation, other experiences are reported in [ML93, SB93, CH94, CGFO94].

5.2.2 Communication networks

One of the revolutions in the past few years in the information technology field has been the increasing popularity of computer networks. They are becoming ubiquitous, and the technology in this field is advancing at a very fast pace. Given the tremendous number of options a network manager has to consider in order to update an existing network, or to build a new one, the availability of tools to analyze an existing or projected network (local, metropolitan or even global) is a must. As the size and the complexity of the simulation can be quite large, PDES is a natural tool to look at.

In [TV91], Time Warp is used to analyze a 5-layer, Ethernet-based communication architecture in a network of transputers, with disappointing results. These bad performance results are mainly due to the huge number of rollbacks and the costly memory management.

In [EGLW93], a new parallel simulation technique, named *synchronous relaxation*, is used to simulate a large scale circuit-switching network. This technique is specially appropriate for its use in SIMD machines, while most of the techniques described in this dissertation are specifically designed for MIMD systems.

In Canada, a team leaded by Unger [ACGW95, GOR95, Gome95] is developing ATM-TN, a set of tools with a modular architecture which supports modeling, simulation and analysis of ATM networks. A key piece in ATM-TN is an implementation of TW, optimized to run in shared memory environments. Previously, this team reported their experiences using TW to study a Canadian switched telephone network [Clea94].

5.2.3 Parallel computing

A growing number of papers can be found where experiences using PDES to analyze different aspects of parallel computing are reported. Some common topics are: performance of message passing networks (our work could be included here), performance of cache memory schemes, and behavior of parallel programs.

In this field, one interesting work is the Wisconsin Wind Tunnel [FW94, BW95], defined by the authors as “a parallel simulator for cache-coherent shared memory machines”. The main purpose of this system is to analyze the behavior of parallel programs running in multiprocessors, and the effect of different cache memory schemes in the execution efficiency of those programs. It uses a conservative algorithm for synchronization, where the duration of the interprocess communication primitives serves to compute the lookahead. The host machine for the simulations is a CM-5.

After some experiences with the WWT, the researchers become aware of an important pitfall in the simulator: the communication delay between (simulated) processing elements was supposed to be a constant, which is perfect to compute lookahead, but rather unusual in actual parallel systems. Contention in the interconnection network may produce arbitrary delays, which were not taken into account. Some modifications in WWT allow the detailed simulation of an interconnection network, using a model of router based in the Torus Routing Chip [Dall86]. This detailed simulation is done using a centralized (instead of a fully distributed) approach. Although the obtained data are much more accurate, there is a price to pay: the execution time of the simulator is about 10 times slower.

A similar work is being done at the University of California at Santa Barbara, under the name of Maya [ACLS94]. It is defined as “a simulation platform for evaluating the performance of parallel programs on parallel architectures”. Its purpose is “to allow the rapid prototyping of memory protocols with varying degrees of coherence and to facilitate the study of the impact of these protocols on application programs”. It has been developed to work in the Paragon and in networks of workstations. Again, a conservative synchronization scheme is used, with a mixture of deadlock avoidance and deadlock detection and recovery strategies. The communication network is not simulated in detail: the communication delay is supposed to depend on the distance between the communicating processing elements, plus a certain factor due to network contention. A major problem of this simulator is that a central *network manager* takes care of all the message passing among the LPs (to be able to compute communication delays with a reasonable accuracy), and this manager becomes a bottleneck when the number of LPs increases.

A different aspect of parallel computing led to the development of LAPSE [DHN94], defined as a system that “allows one to use a small number of parallel processors to simulate the behavior of a message passing code running on a large number of processors, for the purpose of scalability studies and performance tuning”. It has been implemented in the Paragon. Again, the synchronization mechanism is conservative. The authors report very good performance figures: a simulation of the execution of a 64-PE application in 16 real PEs is between 5.4 times (for computationally intensive applications) and 45 times (for communication intensive applications) slower than the actual execution on 64 actual PEs.

Closer to our work, Heidelberger et al. present in [ITH89, GHTY90, BH95] some simulation studies of multistage interconnection networks, using synchronous parallel simulation algorithms. They report speedups over 6 with 12 processors when simulating a detailed model of the IBM SP-2 interconnection network.

Other interesting work in the field is [NGL94], where a technique for massively parallel, trace-driven simulation of cache memory systems is presented.

5.2.4 Petri nets

Petri nets are widely used to model asynchronous concurrent systems that have parallelism, synchronization and resource sharing [CF93]. Analytical evaluation and discrete-event simulation of Petri net models allow researchers to perform qualitative as well as quantitative analysis of the modeled systems.

Ammar and Deng propose in [AD91] a method to simulate stochastic Petri nets using the Time Warp mechanism. They test it using an Encore Multimax with 18 processors. No speedup figures are given.

Chiola and Ferscha [CF93] present a way of simulating timed Petri nets using conservative and optimistic PDES. Tests are done using a Sequent Balance, an Intel iPSC/860 and a T805-based multicomputer. They state that efficient distributed simulations of timed Petri nets can be done, but real speedups can only be obtained after identifying the model's intrinsic parallelism and causality, and using this information to optimize the logical processes. Communication overheads seem to be the main obstacle to achieve good performance, so some methods to reduce the number of interchanged messages are proposed. A typical advise in this direction is to make a LP have a load big enough to keep the computation/communication ratio properly balanced.

5.2.5 Queuing systems

A significant effort has been devoted to efficiently simulate queuing networks systems, as many real world applications can be modeled using this approach. Maybe the most interesting works are those by Wagner and Lazowska, already described in Chapter 4, and those by Nicol [Nico88, Nico92]. These works offer methods to effectively exploit the lookahead of queuing systems to achieve good speedups when the CMB-DA method is used. The techniques to exploit the lookahead are different for each queue discipline. Considered disciplines are FCFS (First Come First Serve), PS (Processor Sharing) and RR (Round Robin), with or without priorities, with or without preemption.

Several papers by Rego et al. at Purdue University [Sang94, SCR94a, SCR94b] also study this field, although in a different context. The authors develop and analyze a parallel simulator based on process migration, instead of message interchange. They also propose mechanisms for efficient simulation of round-robin and processor sharing queue disciplines, useful in any kind of discrete event simulator (i.e., not specifically tailored for PDES).

Other interesting work in the field is [MR94], where a workbench for queuing systems simulation over a network of transputers using the CMB-DA algorithm is presented.

5.2.6 Other fields

There are many other fields where PDES has been successfully used. For example, in [TB93] a simulation study of communicating finite state machines is done. Wieland et al. analyze in [Wiel89, WJ89] the performance of TW simulating combat scenarios. Wagner considers in [Wagn89] a model of a health care system, and a study of road traffic is performed in [MRR90].

5.3 The model under study

In this section we describe the model we use to make an exhaustive study of parallel simulation techniques. This particular model has been selected because our research group has a good deal of experience working with it, mainly using sequential time-driven simulations. Working with a well-known model, we can be sure that our simulators are working properly.

For the interested reader, a much more detailed description of this model, and of most the concepts introduced in this section, can be found in [Arru93], with complementary information in [Izu94].

5.3.1 Description of the model

We study a network of message routers designed as a communication infrastructure which may be used for building multicomputer systems. Each node of the network is a pair <processor, router>, with the two components joined by a message interface. Processors are the source of messages, as well as the final destination of them. Routers actually move messages from their source nodes to their destinations. 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 conveys the necessary information to make routing decisions. Figure 5.1 shows a sketch of a message router. The main components are:

- 4 *input ports*. They are gates to receive messages from the neighboring routers, through the corresponding *input links*. Each port is capable of storing one flit.
- 4 *output ports*. They are gates to send messages to the neighbors through the corresponding *output links*. Again, the storage capacity is one flit.
- An *injection port*, where messages from the local source of messages (processor) are received.
- A *consumption port*, where messages for the local sink of messages (processor) are sent.
- 4 FIFO *transit queues*, one per output port, to temporary store messages when the corresponding output link is busy. Each queue has 10 buffers, each one with capacity for a full message. A buffer cannot be shared among two or more messages.
- A *routing automaton*, which decides through which output port a message will be sent.
- A small FIFO *injection queue*, located in the message interface, where up to 4 local messages can be temporally stored if the corresponding output queues are full.

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

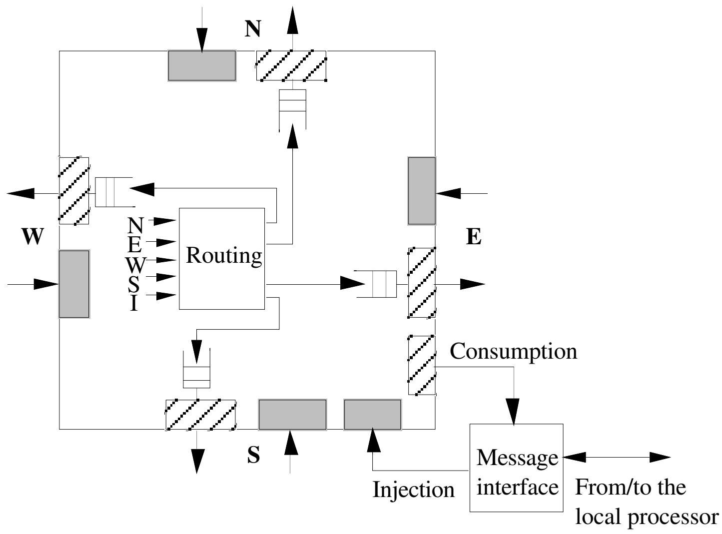


Figure 5.1. Model of message router.

In the literature about message passing networks, several strategies are proposed to control the flow of a message from its source to its destination, traversing several routers along its journey. The most common ones are *store-and-forward*, *wormhole* and *cut-through*. The model presented here, which includes internal buffering, is specially suited for cut-through, which is the most efficient of all these strategies.

The routing strategy, i.e., the selection of the route that a message follows to reach its destination, could be any feasible one, but in our studies only dimension order routing (DOR) has been considered. Other strategies can easily be integrated into the model.

It is assumed that processors immediately consume received messages, so they never force a message to stay in a router occupying resources. Processors generate messages following a certain traffic pattern. The most common ones for the analysis of this kind of models 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 to any other node of the network, with the same probability. The length of the time interval between two message generations at the same node is given by an exponentially distributed random variable; the method of computing the mean for this distribution will be explained later. 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 even the injection queue are full, the message is rejected (it is lost). This is considered as the saturation point of the message passing network.

This model of router can be used to form any kind of network of degree 4. Studies of our group considered mainly bi-dimensional meshes and tori. Here we have focused on tori, because they exhibit better performance characteristics. If the topology is a torus, deadlocks among messages competing for resources in the network might appear. Our model of router uses a deadlock-avoidance strategy based on imposing some restrictions to the injection of new messages and the change of dimension[1].

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

- Size of the network. We consider torus networks of $D \times D$ nodes (always a square). Typical sizes are 16×16 and 32×32.
- Message length (M). The values used in the experiments are 4 and 32 flits.
- Load of the network (L). It represents a percentage of the maximum theoretical bandwidth of the network, for a random traffic pattern. A 100% load means that all the bandwidth of the network bisection is used.

These three parameters are needed to compute the time interval between the generation of two consecutive messages at a given node. As said before, this time separation is computed as an exponentially distributed random number. The rationale to calculate the mean of that distribution is as follows.

Let us divide the network, a bi-directional 2D torus, from north to south into two parts of the same size. If we concentrate on the left part, under the random traffic assumption half of the messages there generated will have their destinations in the same part, while the other half should cross the bisection to be consumed in the right part. The symmetric reasoning can be applied to the right part. At each cycle, up to $4 \times D$ flits can traverse the bisection ($2 \times D$ in each direction). This imposes a limit to the number of messages generated in the network in one cycle. No more than $2 \times (4 \times D)$ flits, or $(8 \times D)/M$ messages, can be processed by the routing network in one cycle. At this point all the bisection links are fully occupied. Distributing these messages evenly among all the nodes, this results in $(8 \times D/M)/(D \times D) = 8/(D \times M)$ messages per node and per cycle. This situation corresponds to a load $L = 100$.

For a different value of load L, ranging from 0 to 100, the number of messages generated per node and per cycle is $(8 \times L)/(100 \times D \times M) = L/(12.5 \times D \times M)$. In other words,

---

1 The concept of deadlock in the model does *not* have any relationship with the deadlock in a CMB simulator.

the time interval between two message generations at a node can be computed as $(12.5\times D\times M)/L$. It should be clear that the same value of load can mean that the network manages many short messages or few long messages.

5.3.2 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 able to be simulated by a generic event-driven simulator. The definition should be independent of the simulators to be used. However, the class of parallel simulators we are working with (based on the distribution of simulation tasks among a set of logical processes) impedes the use of any sort of shared data structure, because the LPs constituting a simulator may be distributed among different processors in an actual multicomputer. 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 (struct) with elements representing ports, queues, etc., plus some additional elements for statistics gathering.

The events which represent the evolution of the system are as follows:

INJECTION: the local processor generates a new message to be routed to another node.

STEP: the router tries to send the header flit of a message from an output port to an input port in the 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 the output port that it occupied.

FREE_QUEUE: the last flit of a message abandons the position it occupied in an output queue.

In the parallel simulators all the events but STEP and PERMISSION are always internal, i.e., scheduled to be consumed in the same LP that generates them, so 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 the sequential simulator, PERMISSION events are not needed (and they are not used), 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 were included in the parallel version precisely because global state information is not available and all the interactions must be done by means of messages.

5.3.3 Output data

The description of the model is detailed enough to allow to obtain a good deal of insight into the behavior of the routers. The simulators can measure and give information about:

- Maximum and average message latency. The latency of a message is defined as the number of cycles from the instant a message is generated until it is consumed.
- Maximum and average message delay. The delay is the difference between the latency and the time a message would need to travel through the network if it were the only one using it.
- Maximum and average size of the transit and injection queues.
- Number of generated and consumed messages.
- Other data.

This information is very valuable to us, primarily because it has already been gathered in previous studies, and allows us to validate the correctness of our simulations.

As it was done in the previous chapter, we will not present any of the output data about the behavior of this model that the simulators generate. A reader interested in the model itself should consider the reference [Arru93].

5.4 The simulators

In this section we present a general description of the simulators used in this study. A detailed description of each simulator is given in the following sections. Six different simulators have been implemented and tested:

- SED (sequential event-driven), able to run in any of the parallel systems described in Chapter 3.
- CMB-DA (Chandy-Misra-Bryant with deadlock avoidance), with versions for the Supernode, the Paragon and MPI.
- TW (Time Warp), only for the Paragon.
- SPED (synchronous parallel event-driven), only for the Paragon.

All the parallel simulators work with the same description of the model. As mentioned before, SED works with a slightly different description of the same model, optimized to reduce the number of processed events. For a given machine and parallel simulator, the main performance figure to be considered is the speedup, taking as a reference the execution time of SED running in that particular machine.

All the simulators share as much code as possible, to be fair when making comparisons and, obviously, to reduce development effort. In particular, in all the cases a new set of functions to manipulate event lists has been used. These functions are based on a heap data structure, following the recommendations in [CSR93]. In the previous experiments (Chapter 4), events were stored in a linked linear list. For small, simple models where the event calendar does not grow too much, this data structure works well enough. However, for detailed, big models the complexity of the insertion function in a linear list is too high: O(n), where n is the length of the list. The extraction/deletion is trivial, O(1). Using a heap, both insertion and extraction/deletion operations cost O(log n).

The change from linear lists to heaps in the simulation engines resulted in an extraordinary performance improvement, specially for the sequential simulator, where all the events are stored in the same calendar. The difference was less noticeable in the parallel simulators, because events are distributed among all the LPs and, therefore, lists are shorter.

5.4.1 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 5.1.

The first two parameters are needed for all the simulators, sequential and parallel, and they do not need any further explanation.

The number of processing elements must be given for any parallel simulator. A mapping of the simulated network of routers onto the actual network of 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). This way, the partition of the model and its mapping onto the network of PEs is trivial (a square of size (D/P) routers is simulated in each PE) and perfectly balanced (all the PEs have the same load).

Parameter	Meaning
Cycles	Simulated amount of time while the behavior of the network of routers is studied.
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 exploit the lookahead ability of the model.

Table 5.1. Parameters of the simulators.

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 alternatives for organizing a parallel simulator. A parallel simulator always consists of a collection of collaborating LP (logical processes), where each LP is, in fact, a Unix process (in the Paragon or in MPI) or a transputer process (in the Supernode). Mapping the model onto the host computer requires 2 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 good deal of interprocess communication is needed, because all the STEP and PERMISSION events are external, i.e., they need to be sent as messages (although 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.

With the toy model of the previous chapter, we learned that it is good to have models with large grain size, to balance the computation/communication ratio of the PEs. For this reason, we redesigned the LPs to allow the simulation of a square of routers (even of size 1), while the version previously used was only designed for minimum grain size. It is interesting to notice that, in the case of maximum grain size, any of our parallel simulators running onto one PE behaves exactly like the sequential simulator.

Note that more alternatives of grain size are possible. See, for example, Figure 5.2, a case where $D/P = 4$. Figure 5.2a represents the mapping for maximum grain size, Figure 5.2c represents the mapping for minimum grain size and Figure 5.2b represents a mapping for *intermediate* grain size. If the mappings are always done using squares, there can exist either none, one (like in the example) or several cases of intermediate grain size.

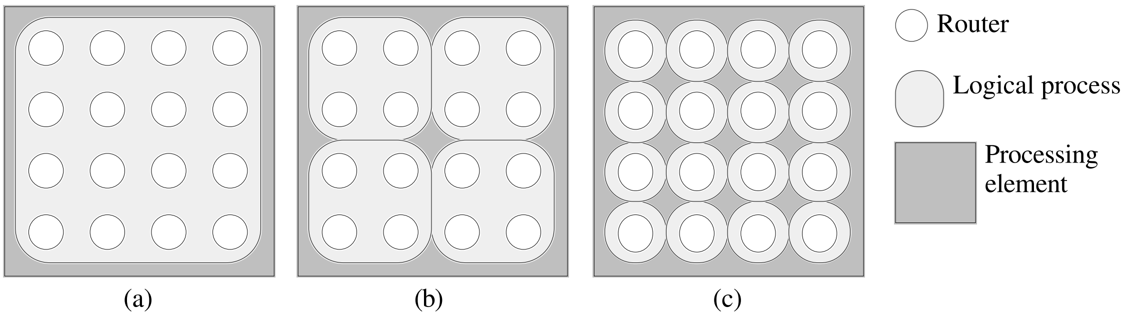


Figure 5.2. Mapping a network of 4×4 routers onto a PE. (a) Maximum grain size. (b) Intermediate grain size. (c) Minimum grain size.

The last parameter in Table 5.1 is the lookahead. When CMB-DA is being used, it is highly recommended to analyze the simulated model to see where some lookahead can be extracted, and to tailor the simulator to exploit that lookahead. If this can be effectively done, timestamps of null messages will have higher values and the overall number of required null messages will be reduced, while a faster clock advance of the LPs will be allowed.

If a LP simulates only one router (i.e., the grain size is minimum), then the behavior of the simulation is quite predictable, and some lookahead can be extracted. Let us suppose that, at time t, a router has sent a message header through an output port; the simulator can be instructed to know 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 $t+M$ is the lookahead.

The actual lookahead computation proceeds as follows. When a LP is going to block, it first sends null messages to each of its neighbors. For our models, there are always four neighbors: north, east, west and south. Each of those channels of the LP are directly related to the input/output ports of the simulated router. In order to compute the timestamp of the null message that will be sent through channel i, the following information has to be kept:

$lps[i]$: time when the LP sent the last PERMISSION message through output channel i, i.e., when the simulated router authorized the reception of a message through input port i.
$lss[i]$: time when the last STEP was sent through channel i, i.e., when the router tried to send a message through output port i.
$lpr[i]$: time when the last PERMISSION was received through i, i.e., when the router was authorized to send a message through output port i.
$lsr[i]$: time when the last STEP was received through i, i.e., when the router was asked to allow the reception of a message through input port i.

Let us make a the minimum timestamp for the next STEP to send, b the minimum timestamp for the next PERMISSION to send and ts the actual value to use as timestamp for the null message sent through channel i. If *clock* is the current value of the LP's clock, a, b, and ts are computed as follows:

```
if (lpr[i] > lss[i]) a = lpr[i] + M - 1;
else a = lss[i] + M;
if (lsr[i] > lps[i]) b = lsr[i] + 1;
else b = lps[i] + M;
ts = min(a, b);
if (ts <= clock) ts = clock + 1
```

The minimum timestamp for an eventual next STEP depends on whether the PERMISSION for the previous STEP has been received or not. If it has been received, another STEP can be sent M-1 cycles later. Otherwise, the next STEP will have to wait at least M cycles.

If a STEP has recently been received, the next PERMISSION should be granted as soon as possible, i.e., in the next cycle, unless the corresponding output queue is full. If no STEP has been received, the LP knows that the minimum separation between two PERMISSION messages is M.

The value of *ts*, a lower bound on the timestamp of the next message to generate (STEP or PERMISSION), is the minimum of the two previously calculated values. If *ts* falls below the current clock value, then the lookahead computation has been useless, and the minimum value *clock*+1 is used for *ts*.

Unfortunately, when a LP simulates an aggregate of routers the behavior of the aggregate is not easily predictable. The computation of the lookahead is too expensive for the achieved result: most of the times the obtained value is one, a minimum which can be used without any computation. For this reason, we have only measured the effect of the lookahead for the case of minimum grain size.

5.4.2 Output results

The output of the simulator consists of a set of statistics about the model, as already described, plus a set of measurements about the behavior of the simulator itself. The most important of those measurements is the execution time of the simulation, because it is the basis for computing speedup values.

Some other interesting figures are also obtained, for example:

- Number of null and useful messages in the CMB-DA simulator.
- Number of positive and negative messages in the TW simulator, plus number of rollbacks.
- Number of messages and barrier invocations in the SPED simulator.

These additional data allow us to reason about the performance of the different parallel simulators.

5.5 Supernode implementation of CMB-DA

In this and the next sections we will give additional details about how each simulator is implemented in each host multicomputer. Special attention will be paid to the Supernode implementation, because it was the origin of the others. For the Paragon and MPI versions we will pay attention to the details only where they differ from the Supernode implementation.

To run a CMB-DA simulator in the Supernode, a network of raw transputers is arranged, with the same structure explained in the previous chapter (a torus network of worker transputers with a monitor transputer inserted in one of the wrap-around links). The number of processes assigned to a worker transputer depends on the grain size of the LPs. The general organization of the processes in a worker transputer is shown in Figure 5.3. After the experience with the toy model, we observed that it is neither necessary to implement a full-blown router in each transputer, nor to manage all the messages through the routers. The LPs in one transputer can be directly joined via internal channels, while the external links are needed only if two logical neighbors are mapped onto different transputers. Even in this second case, a router is too sophisticated, because the interchange of messages always occurs between neighbors. However, channel sharing is still needed, and must be provided somehow.

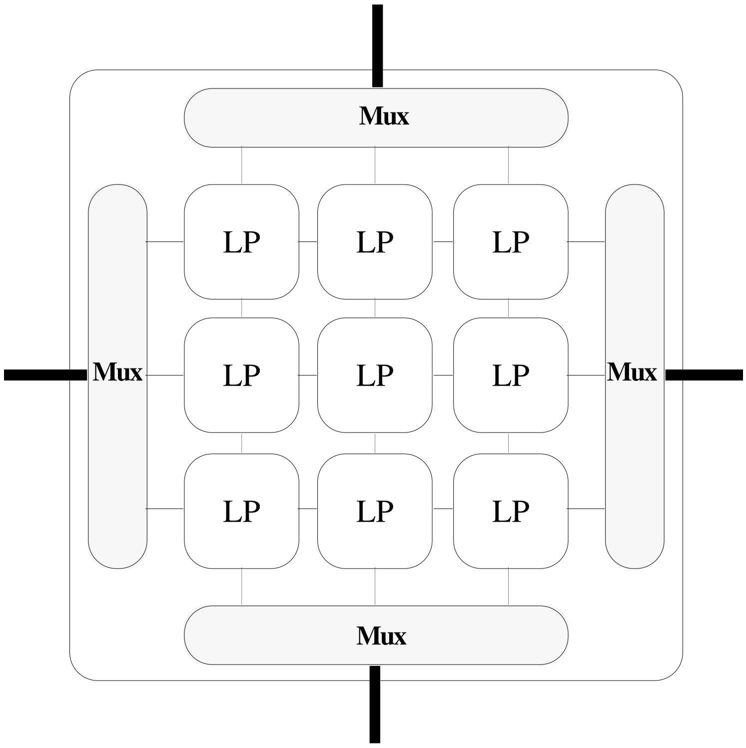


Figure 5.3. Configuration of each worker transputer.

The new approach is to simplify the design of the simulator, removing the routers and substituting them by low-overhead channel multiplexers. Communication through multiplexers is depicted on Figure 5.4. They are able to maintain several bi-directional logical channels over just one physical link. Obviously, if maximum grain size is used (only one LP per transputer), multiplexers are not needed, so they are not used and a source of overhead is removed.

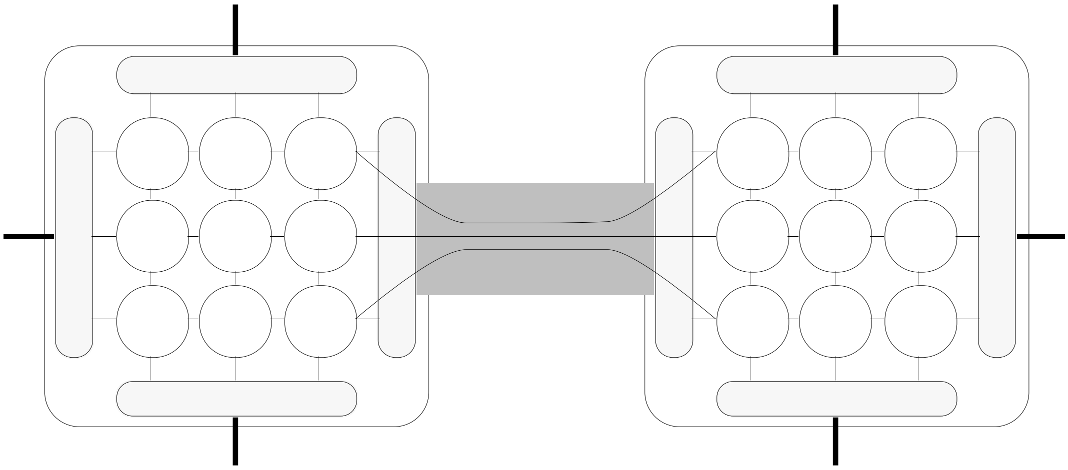


Figure 5.4. Multiplexing several logical channels over a single physical channel.

5.5.1 Components of the simulator

Each LP is a transputer H-process (see Appendix A), compiled and linked separately. Internally, the LPs are composed of three program-created L-processes. The multiplexers as well as the monitor are also H-processes with some internal concurrency. Next we describe how these processes work.

5.5.1.1 The monitor

The monitor, placed in a separate transputer outside the network of workers, collects statistics, summarizes them and sends the final result to the host, to be shown on the screen or saved in a file. Additionally, due to its location in the network, it has to act as a bypass: every message (but the statistics blocks) received from the east/west has to be sent to the west/east. This way the workers do not notice its presence in the network. The structure of the monitor is shown in Figure 5.5.

The monitor knows the number of LPs in the simulator. When a statistics block is received, always by the east channel, it is not bypassed; instead, it is stored in an internal array. When the number of statistics blocks received equals the number of LPs a summary is computed and sent to the host, which then signals the end of the simulation.

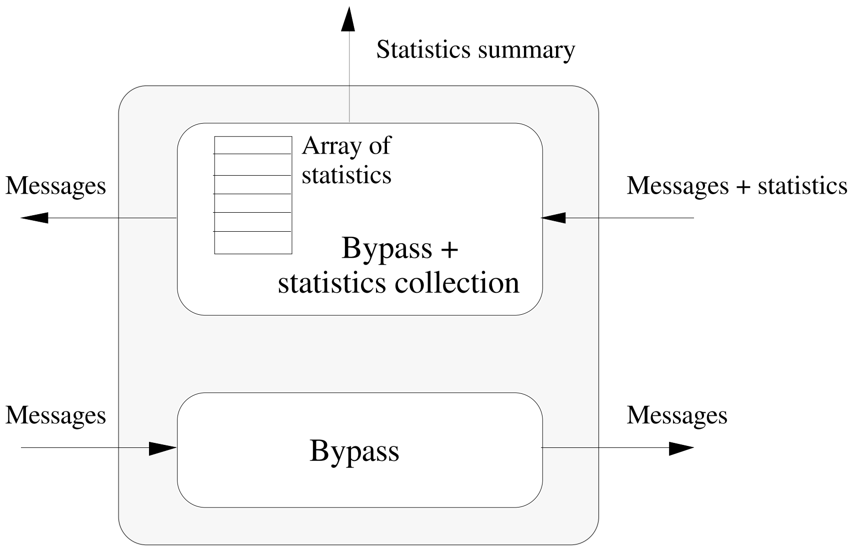


Figure 5.5. Structure of the monitor process.

5.5.1.2 The multiplexers

A multiplexer has to provide simultaneous communication between pairs of neighboring LPs that have been placed in different (but adjacent) transputers. As mentioned before, communication is always between immediate neighbors, so a more general software router is not needed. A pair of multiplexers exist in each side of a physical link.

As shown in Figure 5.4, only peripheral LPs are connected to a multiplexer. A LP does not worry about whether it is connected to a multiplexer or not: all the channels are equivalent. To keep this illusion, two communicating multiplexers maintain a very simple protocol:

- The internal channels connected to a multiplexer are numbered from 0 to N-1.
- When a message m is received from internal channel x, a message (m, x) is sent through the external link. In other words, a tag with the channel number is attached to the message.
- When a message (m, x) is received from the external link, m is sent through internal channel x.

This way, messages from different pairs of LPs are never mixed. The structure of a multiplexer is shown in Figure 5.6.

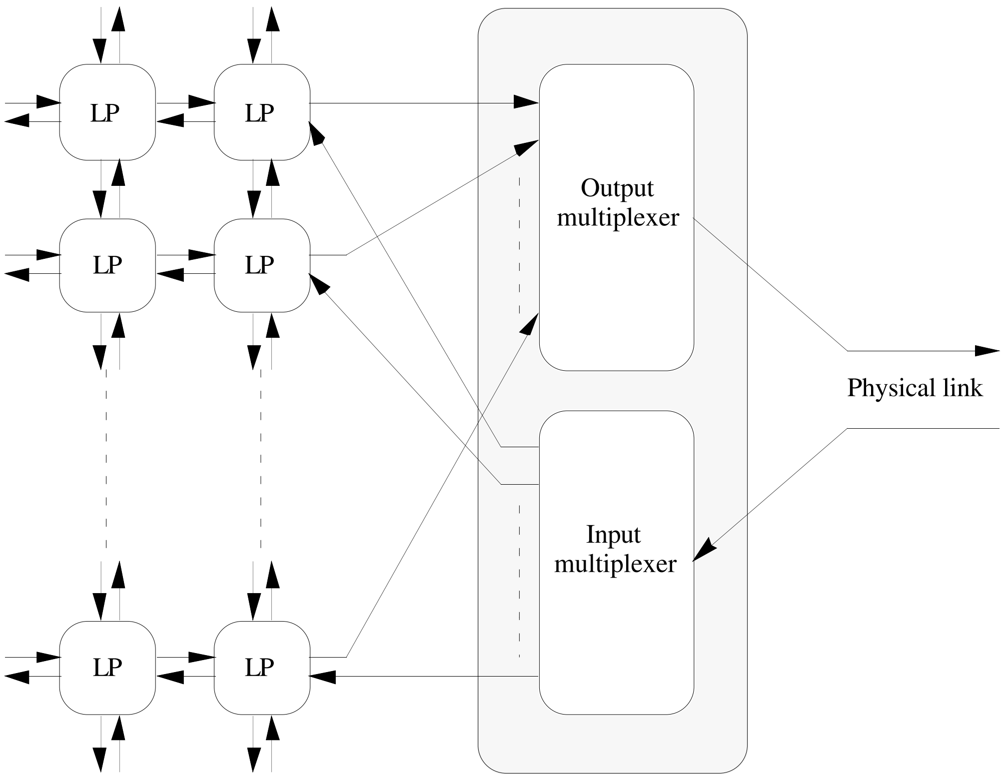


Figure 5.6. Structure of a multiplexer.

5.5.1.3 The logical processes (LPs)

The set of LPs constitutes the core of the simulator. All the other components are only needed to build a working system. Internally, a LP is composed of three processes (Figure 5.7):

- 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, updating important information as the channel clocks and the message-acceptance horizon.
- A *simulator process*, which consumes the events. It interacts with the input process, using channel *pet* to demand messages, which are received through *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*. Those 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 these messages to be sent to other LPs or to the monitor. It implements minimal routing mechanisms, for the management of statistics blocks. An input process may receive statistics blocks from other LPs, which the output process forwards towards the monitor. The communication between these two processes is done by means of a shared data structure.

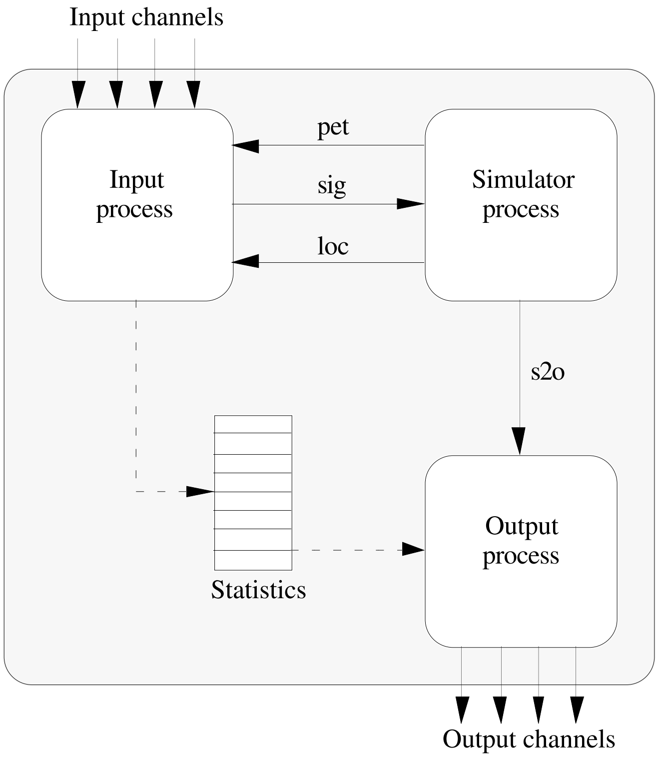


Figure 5.7. Structure of a logical process.

The division of the LP into three different L-processes allows to decouple the activities of event consumption and message interchange. If the design of the LPs were monolithic, communication deadlocks could easily appear[2], due to the way communication is accomplished in the transputer: simultaneous send() and receive() (ChanIn() and ChanOut()) 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 appear if B were also blocked trying to send a message to A. More complex deadlock scenarios, involving more than two LP, are also possible.

2 This is yet another class of deadlock, neither related to the model nor to the conservative simulation algorithm. These deadlocks might appear due to a bad design of the simulator program.

With the proposed design the simulator process never blocks in a send(), because the corresponding output process is always ready to attend it. The simulator might block in a receive(), but only if no suitable event is ready to be consumed (as the CMB-DA 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 soon as possible.

5.5.2 Algorithms

In this section we describe the algorithms of the main elements of a LP: the input, simulator and output processes, paying special attention to the input process, which determines the conservative character of the simulation. A C-like language is used to express the algorithms.

5.5.2.1 Simulator process

The basic scheme of the simulator process is as follows:

```
process simulator:
repeat {
   send(pet, " "); /* empty message through channel pet */
   receive(sig, m);
   clock = m.timestamp;
   if (m.type == WILL_BLOCK) send_nulls(); /* Risk of blocking */
   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) network of routers. 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 target router of an event 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 other LP, so a message is sent through *s2o* for the output process to manage it.

The main loop never stops. When the clock reaches the *end_of_simulation* value, the simulator collects a block of statistics, which 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 has to block when no message is available for consumption. This happens when no stored message has a timestamp below the message-acceptance horizon. When this situation arises, 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, next time an event to consume 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 calculated as the local clock plus one, unless special effort is devoted to exploit the lookahead of the simulated model.

To reduce the number of null messages, which impose an important overhead in the simulation, a null message is not sent unless it performs a useful task. Every LP keeps, in addition to 4 channel clocks (one per input channel), 4 *output* channel clocks, which store the timestamp of the last message sent through the associated output channels. Therefore, an output channel clock kept by one LP has the same value as the channel clock kept by the neighboring LP for that channel. Before sending a null message through a channel, the LP checks if it will increase the output channel clock. If this is the case, the null message will be useful for the receiver. Otherwise, it won't perform any task, so the LP does not need to send it. This saves communication time as well as processing time (in the receiver).

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

5.5.2.2 Input process

It has been mentioned already that an input process manages all the messages that will be consumed in its LP. To do so, it maintains four input queues plus a local event calendar for self-scheduled events. Requests for messages to be consumed are received from *pet*, messages for the local calendar are received from *loc* and messages from the neighboring LPs are received from the four input channels.

```
process input:
```

```
in = wait_for_input(); /* "in" is a channel ready for an input */
if (in == pet) {
  receive(pet, c); /* "c" is just a synchronization signal */
  deliver_message();
}
else {
  receive(in, m); /* "m" is either a null or a useful message */
  if (in == loc) insert_local_calendar(m);
  else {
    /* the message has been received from channel in */
    insert_input_queue(in, m);
    check_blocked_simulator();
  }
}
```

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. Only if this timestamp falls below the acceptance horizon the corresponding message can be safely removed from its queue and consumed (sent by *sig*). This is the behavior that makes the simulator conservative.

When no message is ready to be consumed, a WILL_BLOCK 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 process asks for a message, no one will be delivered (unless meanwhile a new arrival modifies the acceptance horizon) so the simulator will block. The input process activates the flag *blocked_simulator* to signal this condition.

```
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;
```

If a new message arrives from an input channel then the corresponding channel clock is advanced and, if it is not a null message, it is inserted in the associated input queue. Null messages need not be stored, because their only interest is the advance they produce in the channel clocks. This 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 function check_blocked_simulator() is invoked, which eventually may unblock a blocked simulator process and allow the consumption of a message.

```
function check_blocked_simulator:
if (blocked_simulator) {
  h = acceptance_horizon();
  ts = minimum_timestamp();
  if (ts <= h) {
    s = message_with_minimum_timestamp();
    send(sig, s);
    blocked_simulator = FALSE;
  }
  else if (h >= clock) {
    send(sig, WILL_BLOCK_message);
    blocked_simulator = FALSE;
  }
}
```

Clearly, this function is very much like deliver_message(). Note that, when the recently calculated horizon surpasses the local clock, the simulator is awaked by means of a WILL_BLOCK message. This way no useful message is consumed, but at least the clock is advanced and this advance can be communicated to other LPs, using null messages.

5.5.2.3 Output process

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

5.5.2.4 Statistics processing

In the previous algorithms few references to the statistics blocks have been done, to simplify the descriptions. We only said that, when a simulator finishes its period of simulation, a statistics block is prepared and given to the output process.

The output process routes it to the monitor process (which is placed in the monitor transputer), with the following strategy: if the LP is in the bottom row, send it to the west; otherwise, send it to the south. An input process, therefore, can receive (near the end of the simulation) statistics blocks mixed with normal messages. Those blocks must not be inserted in the input queues, but forwarded until they reach the monitor. The input process transfers these blocks to the output process using a shared queue (not a channel) for communication. The statistics queue has capacity for several blocks, because an avalanche of blocks can be received from the top and/or the east (specially at the bottom row of LPs) by the time the LPs finish their work.

In addition to its usual work, the output process has to examine the statistics queue and, if it is not empty, to forward the received blocks towards their destination. A mutual exclusion semaphore is used to avoid conflicts between the input and output processes when accessing this queue.

5.6 Paragon implementation of CMB-DA, TW and SPED

In Chapter 3 and Appendix A we give a description of the Supernode and the Paragon as development environments for parallel applications, and some important differences among these two platforms are identified. Table 5.2 summarizes the most significant ones.

	Supernode (Inmos C)	**Paragon (NX)**
Libraries	channel.h, process.h, misc.h, etc.	nx.h.
Network topology	Reconfigurable. User responsibility.	Static (bi-dimensional mesh).
Addressing	Communication through channels, which must be statically defined in the configuration file.	To send a message, the address (node, process) of the destination process must be given.
Message routing	Hand-made (software) message passing mechanisms to share links or to communicate two processes in non-adjacent transputers.	Fast, hardware message routing mechanisms.
Communication model	Blocking and synchronous. No buffering provided.	Blocking (or nonblocking) and basic. Buffering provided by the system.
Support for global operations	None.	Excellent.
Node multiprocessing ability	Excellent. Internal RR scheduling. A process is descheduled if awaiting for communication completion.	Poor. RR scheduling. A process always finishes its quantum, even if it is blocked for communication.
Parameters	In the configuration file. Reconfiguration is needed to re-run a program if any parameter is changed.	In the command line. No additional step is needed after compilation.

Table 5.2. Main differences between the Supernode and the Paragon.

The bottom line is: life is much more easier for a programmer in the Paragon. The preparation/execution of programs requires less steps and, in general, programs are less deadlock-prone, due to the buffering provided by the message passing system.

In the next subsection we will explain the changes in the CMB-DA simulator that were required when porting it to the Paragon. After that, some insights into the implementation of the TW and SPED simulators will be given.

5.6.1 CMB-DA

The adaptation of the CMB-DA simulator to run in the Paragon required basically to cope with two of the differences stated in Table 5.2: synchronization among processes and node multiprocessing ability. The change from channels to addresses was not difficult, although the idea of channel was still used, because it is a key element of a CMB-DA simulator. The other differences only resulted in a simplification of the programs.

The implementation of CMB-DA in the Supernode fully exploits the good multiprocessing abilities of the transputer: each transputer contains several H-processes (LPs and multiplexers) which, in turn, are structured in several L-processes. There were several reasons to justify this design:

- The synchronous nature of message passing functions requires a decoupling of message management (reception, storage, sending) and message consumption tasks, to avoid communication deadlocks. The division of a LP in input, simulator and output processes provides this decoupling.
- As we will see when analyzing the performance of the simulations, it is very efficient to share a transputer among several LPs: one LP can be working while others are blocked awaiting for communication. 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 needed. The main differences between the Supernode and the Paragon versions of CMB-DA are:

- Elimination of the multiplexer processes, and of the bypass functions of the monitor. 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 with message passing are managed by a separate network of message routers (very much like the one we are using as the model to analyze), without interfering with the activity of the processes. The application controlling process acts as the monitor, and its purpose is to launch the LPs and to gather statistics at the end of the simulation run.
- Integration of the input, output and simulation processes (the three L-processes which form a LP in the transputer) into a single process. This new design has been
  - *possible* because of the buffered nature of the communication in the Paragon, and
  - *forced* by the poor multiprocessing abilities of the Paragon.
- Elimination of the minimum and medium grain sizes alternatives for the LPs: only maximum grain size is considered. This means than only one LP run in each PE. The reason to make this decision has been, again, the inefficient scheduling policy of Paragon's PEs.

The algorithm of the LP is as follows:

```
process LP:
repeat {
```

```
        h = acceptance_horizon();
        ts = minimum_timestamp();
        if (ts <= h) {
          m = message_with_minimum_timestamp();
          clock = m.timestamp;
          consume(m);
          if (clock > end_of_simulation) build_and_send_statistics();
        }
        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 a LP is busy, messages can arrive from the neighbors, and they are stored in system buffers, awaiting until the LP decides to actually receive them. This buffering provides enough decoupling to allow a collection of LPs to progress without communication deadlock.

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

5.6.2 TW

Our implementation of TW follows, except for some minor details, the description given in Chapter 2. Among all the possible optimizations, these two were implemented:

- conservative time windows, with dynamic window size, and
- incremental state saving, dynamically adjusting the interval between full copies of the state.

Additionally, input queues were implemented as a combination of two data structures: past events are stored in a linked list, with the components of the output queue (antimessages) and the state queue (past copies of LP's state) attached to the already consumed events, while future events are stored in a heap, to optimize insertions.

The first optimization was included after the experience gained with the experiments reported in Chapter 4. Without it, LPs tend to eagerly consume all the allocated memory. Preliminary tests of TW with time windows gave us very poor performance figures, much worse than expected. We found out that the large size of the state to save per executed event was partially responsible for the poor results, so we added incremental state saving.

The algorithm of a TW logical process is as follows:

```
process LP:
repeat {
   if (message received) {
      m = receive();
      if (m.type = TOKEN_GVT) {
         manage_gvt();
         if (gvt > end_of_simulation) build_and_send_statistics();
      }
      else insert_input_queue(m);
   }
   if (next_timestamp() <= gvt + time_window) {
      m = next_event();
      clock = m.timestamp;
      consume(m);
      save_state();
   }
}
```

Global virtual time (GVT) computation is performed in a fully distributed fashion, using a token-based mechanism [Fuji90a]. LPs are numbered from 0 to N-1, and organized into a logical, unidirectional ring. A special message, GVT_TOKEN, is passed around this ring. The token contains three fields:

- *owner*, which identifies the LP owning the token,
- *gvt*, the last computed GVT, and
- *min*, a minimum timestamp value, which is used to compute the next GVT.

Each LP keeps track of its smallest local virtual time since it last received the token, *min_lvt*. When the token is received, the LP compares this value with the *min* field of the token. If the LP's *min_lvt* is smaller, it becomes the owner by writing its identification and local minimum into the token (fields *owner* and *min*, respectively). The LP then passes the token to the next LP in the ring. If the token returns to its owner, then it has traversed the ring without encountering a smaller local minimum, so the *min* value of the token is actually the computed GVT value. This value is written into the *gvt* field of the token. The owner then restarts the GVT computation by placing its local clock value into the *min* field of the token, and transmitting it to the next LP in the ring.

Each time a LP receives the GVT_TOKEN, an updated GVT value is obtained in the *gvt* field. This allows the LP to perform the fossil collection procedure, retrieving memory from the input, state and output queues. The updated GVT value may also allow a LP to progress when it is stopped because no event inside the time window is available. An additional operation is performed when the GVT_TOKEN is received: the adjustment of the time window size and the state saving interval [Pala94].

The LP keeps an activity counter, which is incremented each time an event is consumed, and decremented each time an event has to be undone. When the GVT computation is performed, the real-time interval between sending the token and receiving it again is computed. Then a *performance index* can be computed as the ratio of effectively consumed events to elapsed real time. If the newly computed performance index is equal or bigger than the previous value, then the LP is working well, and the reward is an increment in the time window and in the state saving interval. However, if the performance index is lower than before, it probably means that too much work is being undone, so the LP reduces its degree of optimism by reducing the time window and the state saving interval.

A second key piece of the LP is the insert_input_queue() function. It is invoked each time a message is received, to store it in the input queue. The received message may be an event scheduled for the future, which is simply inserted in the right position of the input queue (in fact, in the heap part of the input queue). It may also be a straggler, which causes a rollback before its insertion in the queue. Finally, it can be an antimessage, which requires a positive message to be annihilated, either without triggering a rollback (if the positive version has not been executed yet), or after a rollback (if the positive version has been consumed already). The algorithm is as follows:

```
function insert_input_queue (message m):
if (m.timestamp > clock) {
   if (m.type != ANTIMESSAGE) insert_future(m);
   else  annihilate_future(m);
}
else if (m.timestamp < clock) {
   if (m.type != ANTIMESSAGE) annihilate_past(m);
   else insert_past(m);
}
else /* m.timestamp == clock */ {
   if (m.type != ANTIMESSAGE) insert_future(m);
   else {
      if (already_consumed(m)) annihilate_past(m);
      else annihilate_future(m);
   }
}
```

Functions insert_past() and annihilate_past() include both the execution of a rollback procedure. Insert_past() is executed whenever a straggler arrives. It performs the following steps:

- Search, in the input queue, the location where the straggler has to be inserted.
- Adjust the local clock to match the straggler's timestamp. Recover the state of the LP at that time. This can be found in the state queue, in the position just before the straggler, or may require and additional search in the past plus a coast-forwarding phase.
- Clear the state and output queues, i.e., eliminate all the elements whose timestamp is larger than the straggler's.
- Execute the straggler.
- Save the new state.

Annihilate_past() is executed when a negative version of an already consumed message is received. This function is very similar to the previous one:

- Search the location in the input queue were the positive message is stored. Remove (annihilate) it.

- Recover the state of the LP saved just before the corresponding positive message was consumed. As before, this can be found in the state queue, in the position just before the annihilated event, or may require and additional search in the past and a coast-forwarding phase.
- Clear the state and output queue, i.e., eliminate all the elements whose timestamp is larger than the straggler's.

From these explanations, it is clear that our TW implementation performs aggressive cancellation and aggressive re-evaluation.

5.6.3 SPED

The experience by Soulé et al. with a synchronous parallel event-driven (SPED) simulator (reported at the beginning of this chapter) encouraged us to implement and test it in the Paragon. As the NX library offers an excellent support for global operations, it is easy to redesign the LPs to work synchronously (sharing the simulation clock), instead of working asynchronously (as it is the case for CMB-DA and TW). The basic design of a LP in our implementation of the SPED simulator follows, in broad lines, the description given in Chapter 2, with some modifications that take advantage of the set of communication operations offered by the NX library.

Each LP_i in the SPED simulator executes a loop of 4 basic operations:

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

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

3 Message distribution. LP_i sends the external events generated in the previous step, using messages. This is done in two phases:

a) every neighbor is informed about how many messages will be sent to it and

b) messages are actually sent.

4 Message gathering. LP_i reads all the external events, generated in other LPs, that have been sent to it. Again, this is done in two phases:
   a) gathering from the neighbors the number of messages to read and
   b) actually reading the messages.

Message distribution and gathering phases are designed in such a way that all the messages generated in one iteration are safely received and stored in the same iteration, without interfering with the next one.

The resulting algorithm for a LP in a SPED simulator is as follows:

```
process PE:
while (clock <= end_of_simulation) {
  ts = minimum_timestamp();
  clock = global_min(ts); /* Minimum among all the LPs */
  while (next_event_time() == clock) {
    m = next_event();
    consume(m);
  }
  send_messages();
  receive_messages();
}
build_and_send_statistics();
```

5.7 MPI Implementation of CMB-DA

The MPI implementation of CMB-DA is very similar to the Paragon version. There are only two relevant differences:

- Functions belonging to the Paragon's NX library (declared in <nx.h>) have been substituted by their equivalents in the MPI library (declared in <mpi.h>). The change was straightforward, because these two libraries are very close in semantics, and only a limited number of communication functions were used in the programs.

- In MPI the concept of controlling process does not exist. The two activities performed by the monitor in the Paragon are (1) launching the application and (2) gathering statistics. In MPI the first activity is performed from the Unix command line, and the second one, as it is performed after the simulation finishes, can be assigned to any LP without interfering with the simulation. This approach could have been used in the Paragon, too.

5.8 Conclusions

In this chapter we have shown how PDES has been successfully used to study a variety of real-world systems, including the study of different aspects of parallel computing. Our contribution in this field is the use of PDES to analyze the behavior of a network of routers designed to perform the message passing functions in a (hypothetical) multicomputer.

Five different parallel simulators have been implemented, testing three different synchronization mechanisms (synchronous, conservative, optimistic) and three parallel computing environments (Supernode, Paragon, MPI on a network of workstations). This has not been just an exercise on parallel simulation, but also an exercise on parallel programming, because the characteristics of the different computing systems have had an important influence in the way simulators have been programmed. The differences between the parallel programming environment used in the Supernode and the one provided with the Paragon are big enough to justify major design changes. In contrast, porting from the Paragon to MPI has been straightforward, because the semantics of their communication functions are very close.

In terms of programming effort, working with the Supernode has been harder than working with the Paragon or MPI. It is not only that the system is less user friendly, but also that most of the programming effort was in first place done for the Supernode versions and then ported to the other platforms.

In terms of difficulty of implementation, we can say that the synchronous simulator has been the simplest to program and debug. The conservative one needed considerably more development time, but it was done in first place, so it served to acquire most of the experience used with the others. The optimistic has been the most difficult to program, and the hardest to debug.

Chapter 6

Performance results

In the previous chapter we have presented a model of message router, along with five parallel simulators able to work with it. This chapter presents the experiments that have been executed with those simulators, with the aim of characterizing how the parameters of the model, the synchronization strategy, the ways of organizing the simulator and the characteristics of the target multicomputer influence the achieved performance. An analysis of the results allows us to select suitable combinations of algorithms and machines to efficiently carry out simulations of message passing networks.

6.1 Introduction

In the previous chapter, after introducing some work by other researchers using PDES as a system evaluation tool, we thoroughly described a model of a message passing network that is of interest for our research in massively parallel computers. Then we outlined the five simulators we implemented to simulate our model in three different multicomputers. These simulators are:

- CMB-DA in the Supernode, the Paragon and a NOW with MPI.
- TW in the Paragon.
- SPED in the Paragon.

A sequential, event-driven simulator capable to run in the three multicomputers has also been implemented. Its execution times will serve as the reference point for the computation of speedup values.

In this chapter we present the results obtained after performing a set of experiments with the parallel simulators, whose purpose was to get an insight into their behavior under different conditions: ways of organizing the LPs, parameters of the model and characteristics of the host multicomputer. As mentioned in Chapter 4, we are more interested in the behavior of the simulators than in getting useful information about the model. The reason in this case is that we already have an extensive set of data about the model, obtained with a sequential simulator and compiled in [Arru93]. What we want now is to show how some approaches to PDES are effectively useful for this particular domain of application: the evaluation of interconnection networks for parallel systems.

The presentation of the experiments is organized according to the synchronization mechanism: first CMB-DA, then TW and, finally, SPED. An analysis of the performance of each simulator is done after showing the obtained results. A final section is devoted to a series of overall conclusions.

6.2 Experiments with CMB-DA

In this section we present the results obtained after running a collection of experiments using the three implementations of CMB-DA. We start with the Supernode version, then the Paragon version and, finally, the MPI version running on a NOW. We made different experiments varying parameters of the model (network size, message length and load) as well as parameters of the simulator (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, but some experiments have been specifically designed to stress some characteristics of one particular simulator.

The obtained results are represented in the form of collections of speedup curves. Note that speedups have been calculated after running a sequential, event-driven simulator with the same set of model parameters.

The ideal of obtaining a speedup figure identical to the number of PEs used in the simulator can be considered, from the beginning, impossible to achieve. There are some overheads in any CMB-DA simulation which prevents things to be that way. A characterization of the sources of overhead follows.

Let us suppose that the sequential simulation of a certain model, with a certain set of input parameters, requires E events. Let us further suppose that the cost (measured in terms of execution time) of simulating each event, considering event consumption plus event manipulation (including calendar insertion and extraction) is a fixed value Te (i.e., it does not depend on the class of event). The cost of the sequential simulator is, thus, $Ts = E \times Te$. Even when not all the events have the same cost, the value Ts could be computed as the total execution time of the sequential simulator.

Now we compute the cost of CMB-DA running on P processing elements:

- Performing a CMB-DA simulation, the same set of E events is executed, with the same individual cost Te. Note that this is a simplifying assumption. The total number of events could be bigger in CMB-DA, as it is in the case we are considering: PERMISSION events are required in the parallel simulators, but not in the sequential simulator. On the other hand, some management costs might be reduced in CMB-DA, because event calendars are shorter. This difference in management costs can be minimized using a reasonable data structure (such as a heap) to implement event calendars. At any rate, let us consider that the cost of simulating the set of events is still Ts.

- In any parallel simulator messages are needed to carry events scheduled at one LP for another; a certain amount of time is needed to encapsulate an event into a message, send the message, receive it and extract the event from the message. Let us make Tc the total time spent by the LPs in message manipulation.
- A CMB-DA simulator needs to interchange null messages to avoid deadlocks. Let us say that the time required for null message manipulation is Tn.
- Finally, LPs in CMB-DA spend a certain amount of time blocked, awaiting increments in the acceptance horizon which might allow some messages to be consumed. In some cases this time is negligible, because as soon as a LP blocks the CPU can be assigned to another LP; this is the case when the grain size of the LPs is not maximum. However, even in this case there is a cost in the form of context switching. Being one way or the other, let us say that the time spent by LPs being blocked is Tb.

Thus, assuming a perfect load balance, the time required for a CMB-DA simulation is $Tp = ((Ts+Tc+Tn+Tb)/P)$. The speedup can now be computed as (Ts/Tp), which is always less than P.

While Ts and Tc are fixed costs, present in any parallel simulation, Tn and Tb are specific of CMB-DA, and can vary in different ways, depending on many factors. In the following sections we will try to characterize those factors. Note that these two overheads are tightly interrelated. Since null messages are sent just before LPs block, the higher the number of blockings, the higher the number of required null messages.

6.2.1 Description and results in the Supernode

In this section we present the results of seven experiments performed with the Supernode version of CMB-DA. An analysis of the results is done in the next section. Tables 6.1 to 6.4 summarize the parameters used in the experiments. The first three parameters of Table 6.1 (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 the way the simulator is organized. The obtained results, in the form of speedup curves, are plotted in Figures 6.1, 6.2 and 6.3.

	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	Max., int., min. (see Table 6.2)	Max., int., min. (see Table 6.3)	Max., int., min. (see Table 6.2)	Max., int., min. (see Table 6.3)	Int. (see Table 6.4)	Int. (see Table 6.4)	Int.
Lookahead	See Table 6.2	No	See Table 6.2	Yes (16minL)	No	No	No
Results	Fig. 6.1	Fig. 6.1	Fig. 6.1	Fig. 6.1	Fig. 6.2	Fig. 6.2	Fig. 6.3

Table 6.1. Experiments performed with CMB-DA 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 6.2. Values of the parameters grain size and lookahead for experiments **1S** and **3S**.

	LPs per Transp.	**Routers per LP**
16Max	1	16
16i	4	4
16min	16	1

Table 6.3. Values of the parameter grain size for experiments **2S** and **4S**.

	LPs per Transp.	**Routers per LP**
ih	4	16
il	16	4

Table 6.4. Values of the parameter grain size for experiments **5S** and **6S**.

Before analyzing the results, let us describe the purpose of these experiments. Experiments **1S—4S** deal with a relatively small model of 16×16 routers, with message length 4 or 32, and running in 4 or 16 transputers. With these experiments we wanted to confirm the results summarized in Chapter 4, obtained with a toy model. In particular, it was our intention to test the following hypothesis:

1 The simulator scales well, i.e., the speedup with 16 transputers is better than the speedup with 4 transputers. This should be confirmed by comparing **1S** with **2S**, and **3S** with **4S**.

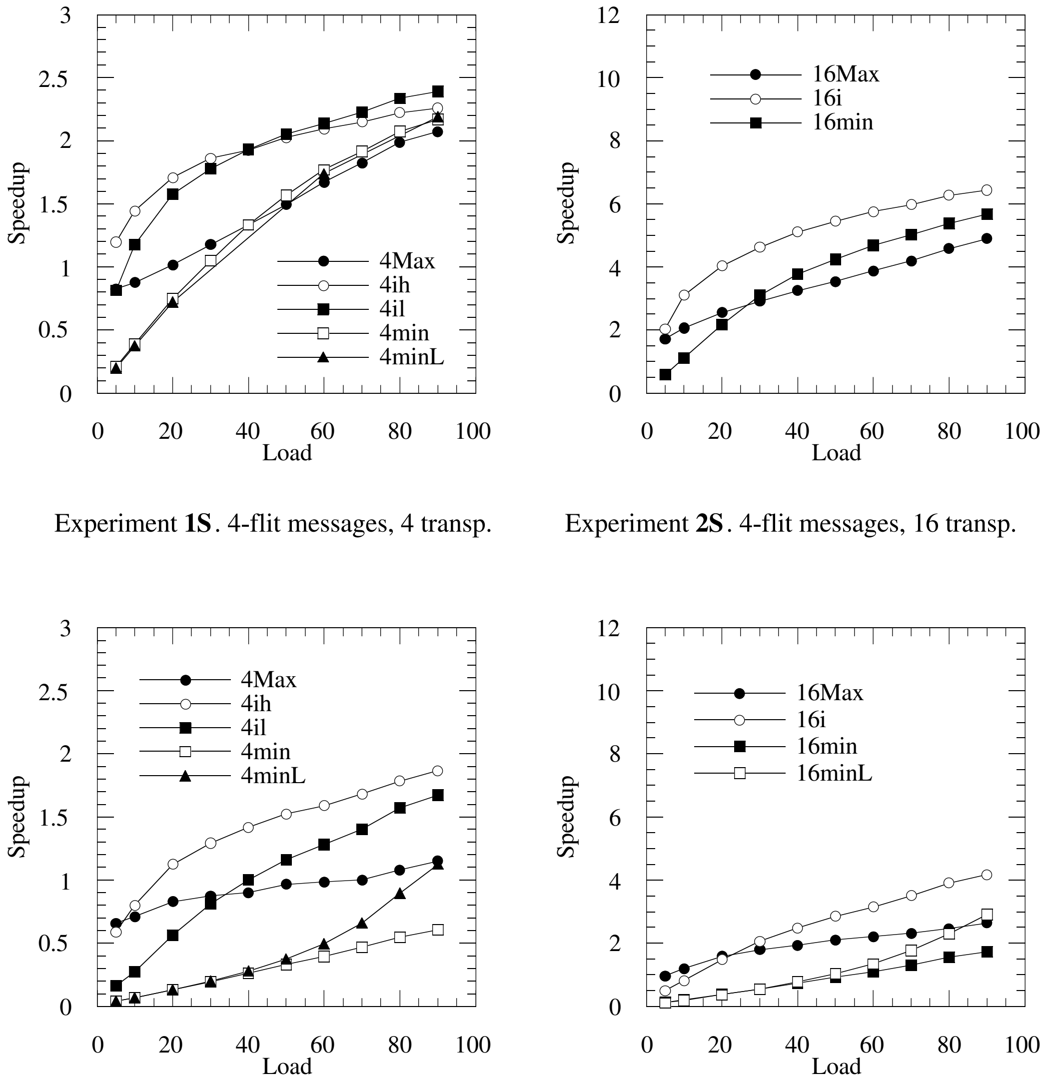
2 Results are better with 4-flit messages than with 32-flit messages. With the same level of load, 4-flit messages means al least 8 times more simulation events than 32-flit messages. Execution times should be longer for both the sequential and the CMB-DA simulators, but the latter will have more opportunities to self-synchronize without null messages, reducing the synchronization overhead. This should be confirmed by comparing **1S** with **3S** and **2S** with **4S**.

3 Performance improves when the load increases. Again, the higher the load the larger the number of events managed by the simulator, giving more synchronization opportunities. This should be confirmed by all the experiments.

Additionally, we wanted to test the impact that the different alternatives of grain size have on the performance, and 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 the imposed workload. In this case we do not use a synthetic workload: the way of giving more work to a LP is mapping onto it a larger number of routers.

Experiment **7S** is a scalability test. The same model is run in 4, 9, 16 and 25 transputers. As we will see, simulator performance strongly depends on the grain size of the LPs. Sometimes there exist a wide range of grain size alternatives, while in others cases there are only two: maximum and minimum. In Figure 6.3 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.



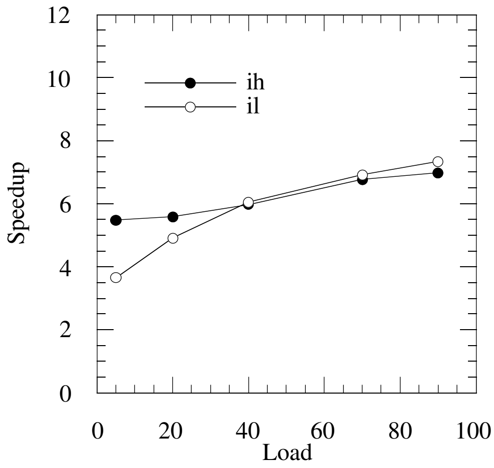
Experiment **1S**. 4-flit messages, 4 transp.

Experiment **2S**. 4-flit messages, 16 transp.

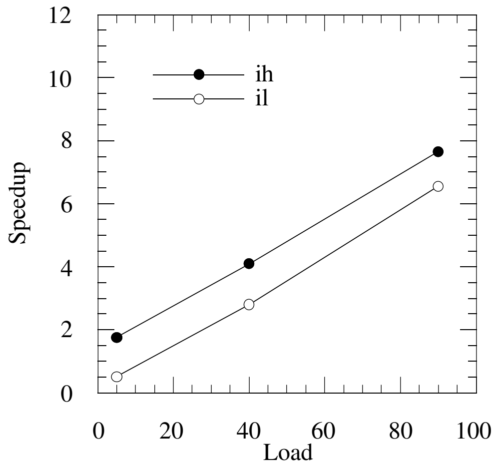
Experiment **3S**. 32-flit messages, 4 transp.

Experiment **4S**. 32-flit messages, 16 transp.

Figure 6.1. Results of experiments **1S—4S**. Simulation of a network of 16×16 routers.



Experiment **5S**. 4-flit messages.



Experiment **6S**. 32-flit messages.

Figure 6.2. Results of experiments **5S** and **6S**. Simulation of a network of 32×32 routers, using 16 transputers.

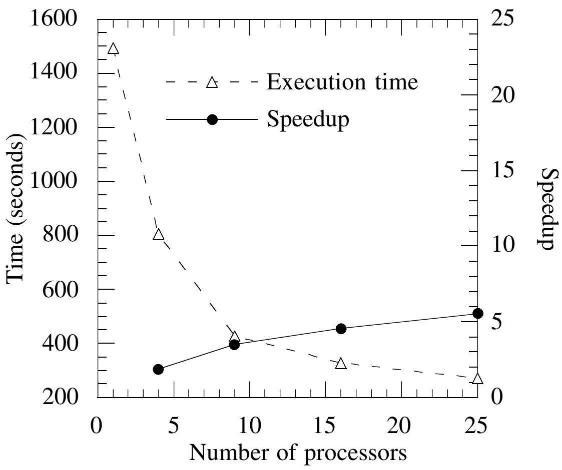


Figure 6.3. Results of experiment **7S**. Simulation, using 4, 9, 16 and 25 transputers, of a network of 24×24 routers (25×25 for the case of 25 transputers) with 4-flit messages at load 50.

6.2.2 Analysis of the results in the Supernode

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

6.2.2.1 Network size, message length and load

Just looking at the set of figures as a whole, it is clear 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 (for CMB-DA) 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 (speedup) is not brilliant, 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-DA 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, as 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. Then LPs spend most of their (real) time blocked or processing null messages, i.e., synchronizing, instead of making progress.

6.2.2.2 Grain size

The experiments described in Chapter 4 suggested that the performance of CMB-DA running on a network of transputers, a system with high communication overheads [Izu94], would increase with the grain size of the LPs. This way, the computation/communication ratio would be more balanced. If, looking at the results of the first four experiments, we compare maximum vs. minimum grain size, we can see that coarse grain simulation is more effective than fine grain simulation for low and medium loads, because a small number of LPs synchronizing with null messages results in lower overhead.

If now we compare with the intermediate grain size alternatives, it is clear that these are the best for intermediate and high loads. Only for very low loaded systems maximum grain size gives, sometimes, better performance than intermediate values. This behavior can be explained analyzing the time used for synchronization in the simulator:

- With maximum grain size, a LP can evolve autonomously most of the time, 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 this leads the LP to send null messages and then block. This behavior 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, there is no problem 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 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, they block very often.
- 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.

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

6.2.2.3 Lookahead

From the description of how lookahead is computed, it should be clear that a LP might obtain large lookahead values when:

- messages are long and
- the LP has recently interacted with its neighbors.

The first point can be clearly observed comparing experiments **1S** and **3S**: while for 4-flit messages curves 4min and 4minL (minimum grain size, with and without using lookahead information, respectively) give nearly the same results, when the length is 32 flits 4minL is clearly better than 4min. The second point 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 only can be effectively exploited for minimum grain size, and this is not the best situation for this implementation of CMB-DA, we do not see any advantage of using the lookahead information provided by this particular model.

6.2.2.4 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 gain 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 25×25 network of experiment **7S**, when simulated over 5×5 processors, only allows for maximum or minimum grain size; each processor must simulate 5×5 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 rhythm to the others; for this reason the resulting speedups are not as good as they should be.
- As 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 6.4 the ratio of useful messages to total 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 parameters for experiment **7S** does not constitute the best possible scenario for the parallel simulator, neither the worst for the sequential one. With a higher load the results would be more favorable to the parallel simulator.

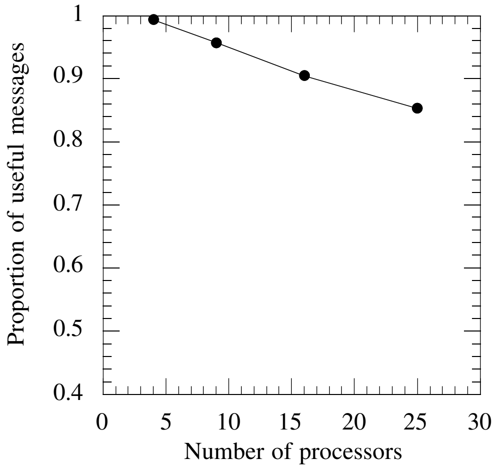


Figure 6.4. Ratio of useful messages for the different numbers of transputers used in experiment **7S**.

6.2.3 Description and results in the Paragon

In this section we will use the Paragon implementations of CMB-DA to evaluate the behavior of this algorithm in this particular machine. The performed experiments are summarized in Table 6.5, while the results are shown in Figures 6.5, 6.6 and 6.7. Experiments **1P** through **7P** are basically the same that those run in the Supernode (**1S—7S**). In **7P** it has been possible to run the 24×24 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 processing elements available in the Paragon: a large model of 90×90 routers is simulated, using 4, 9, 25, 36, 81 and 100 processing elements.

As previously mentioned, the Paragon implementation of CMB-DA only allows a class of mapping of routers onto LPs: maximum grain size. This means that only one LP runs in each Paragon PE, simulating a group of routers. A consequence of this restriction is that 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 of speedup, so trends are considered more significant than actual values.

	1P	2P	3P	4P	5P	6P	7P	8P
Network size	16×16	16×16	16×16	16×16	32×32	32×32	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—64	4—100
Grain size	Max.	Max.	Max.	Max.	Max.	Max.	Max.	Max.
Look-ahead	No	No	No	No	No	No	No	No
Results	Fig. 6.5	Fig. 6.5	Fig. 6.5	Fig. 6.5	Fig. 6.6	Fig. 6.6	Fig. 6.7	Fig. 6.7

Table 6.5. Experiments performed with CMB-DA in the Paragon.

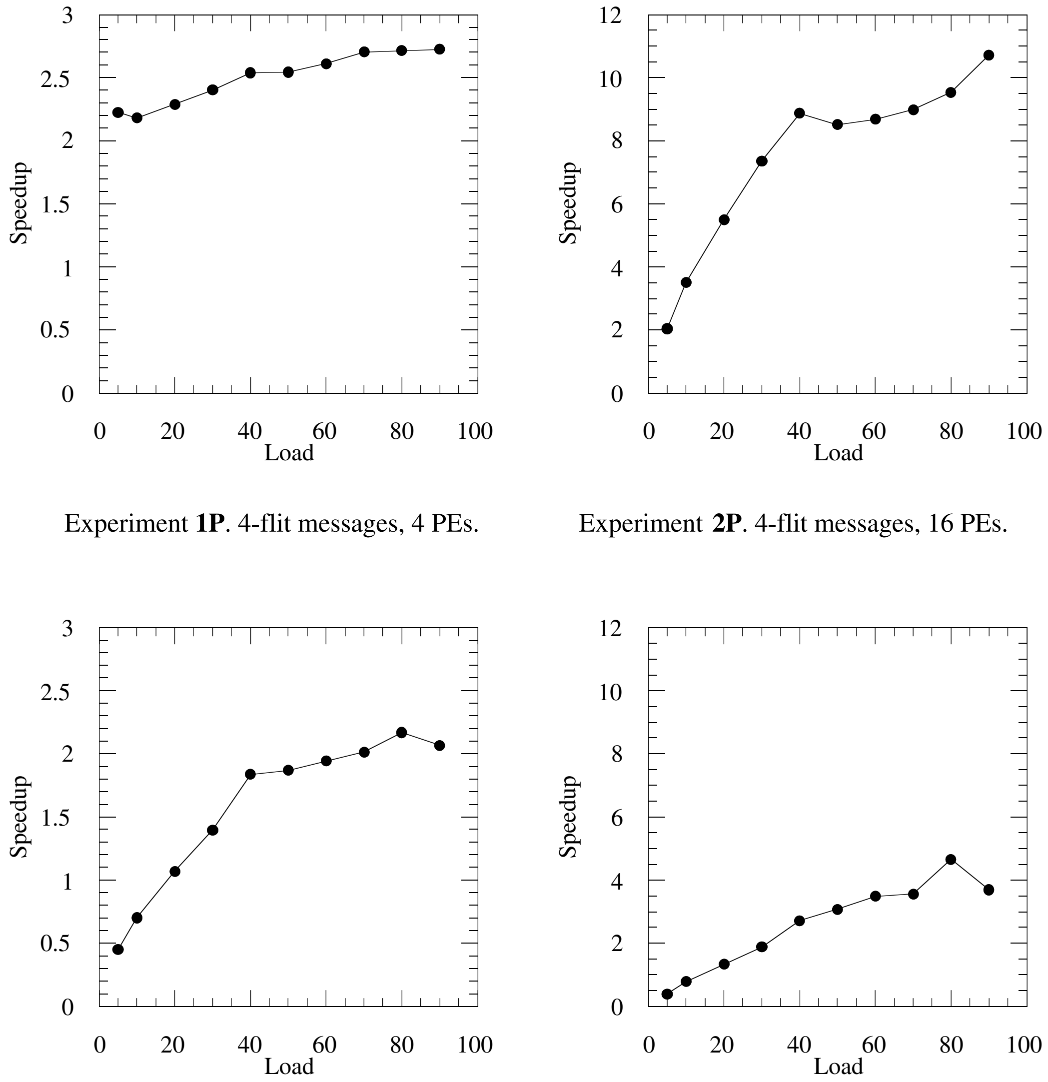
6.2.4 Analysis of the results in the Paragon

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. Summarizing, it can be said that the best performance is obtained with large, highly loaded models managing short messages. These scenarios are a challenge for sequential simulators, while CMB-DA distributes the load quite evenly. According to the characterization of the overheads that we did when introducing the experiments, the obtained speedups cannot be equal to the number of used processors, but they are not too far.

Comparing Paragon and Supernode results, speedup figures are better for the Paragon. Compare, for example, experiments **5S** and **5P**, which are equal for both machines; the peak speedup in the Supernode is slightly less than 8, while in the Paragon is very close to 12. Scalability tests **7S** and **7P** are also equal for 4, 9 and 16 processors; in the last case, the Supernode implementation obtains a speedup of 4.55, while the Paragon reaches 10.

The reason of this performance difference can be found in:

- the architectural dissimilarities between both machines, and
- the differences in the design of the LPs.



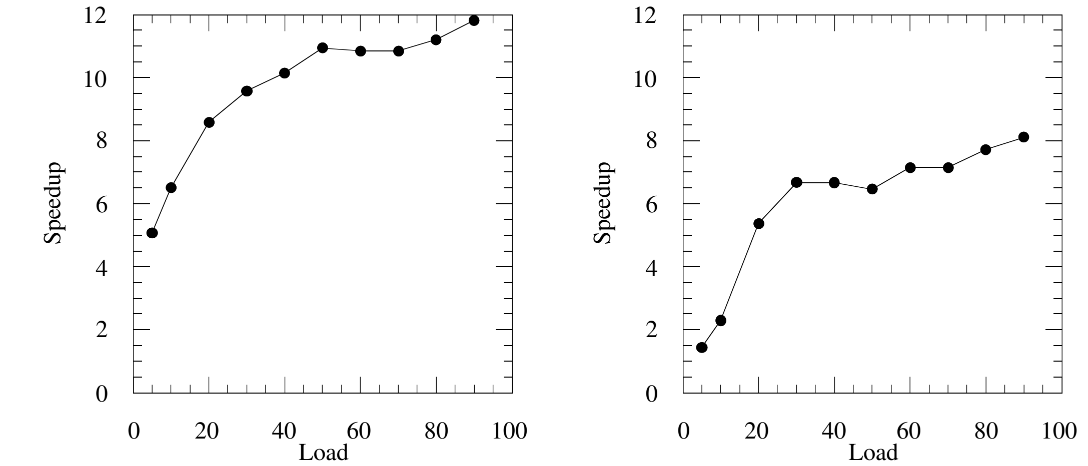
Experiment **1P**. 4-flit messages, 4 PEs.

Experiment **2P**. 4-flit messages, 16 PEs.

Experiment **3P**. 32-flit messages, 4 PEs.

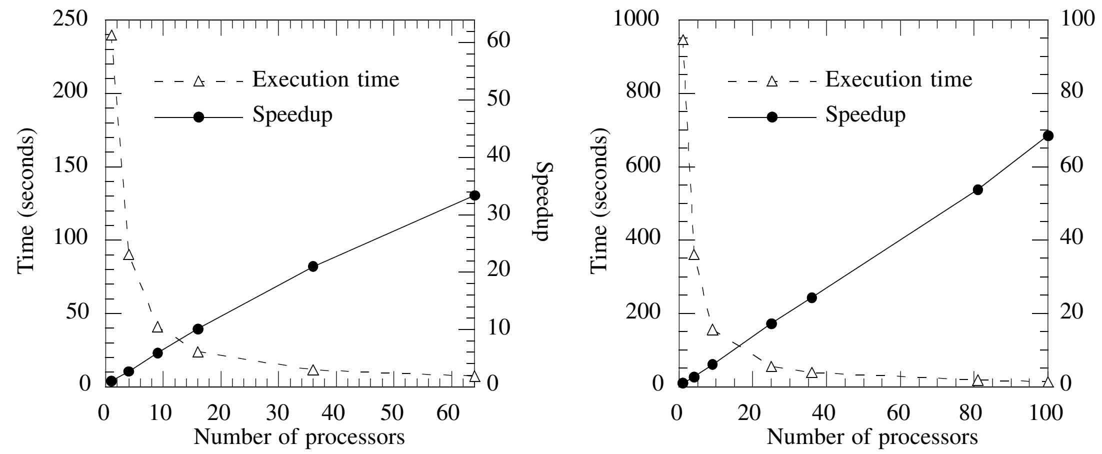
Experiment **4P**. 32-flit messages, 16 PEs.

Figure 6.5. Results of experiments **1P—4P**. Simulation of a network of 16×16 routers.



Experiment **5P**. 4-flit messages. Experiment **6P**. 32-flit messages.

Figure 6.6. Results of experiments **5P** and **6P**. Simulation of a network of 32×32 routers, using 16 Paragon PEs.



Experiment **7P**. 24×24 routers, 4—64 PEs. Experiment **8P**. 90×90 routers, 4—100 PEs.

Figure 6.7. Results of experiments **7P** and **8P**, the scalability tests, with CMB-DA.

We should remember that in the Paragon all message manipulations are done by a network of hardware message routers plus a second processor in each node (the message co-processor), freeing the computing node of most of the overheads of message handling. In the Supernode, however, there is neither hardware routing, nor message co-processors, and each transputer has to divide its time between computation and (software) 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 using 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 L-processes. This arrangement in the Supernode was necessary to avoid communication deadlocks, but there is a price to pay. Those L-processes communicate mostly via internal channels. Therefore, messages are copied from memory to memory several times in their life cycle. Let us count how many times a message is copied from process to process since the moment it is generated at a LP until it is consumed at another (in a different transputer):

- A message is generated at the simulator process of the source LP.
- The simulator process passes the message to the output process (internal copy).
- The output process passes the message to a local channel multiplexer (internal copy).
- The channel multiplexer passes the message to its peer at a neighboring transputer (external copy).
- The channel multiplexer at the destination transputer passes the message to the input process in the destination LP (internal copy).
- The input process will eventually pass the message to the simulator process, when it is its turn to be consumed (internal copy).

All these steps (4 internal copies plus one external copy) are managed by the PEs, i.e., the transputers. In the case of the Paragon, due to the monolithic design of the LPs, no internal copies are necessary and, as just said, hardware support exists for making external copies.

Regarding experiments **7P** and **8P**, the scalability tests, Figure 6.7 shows that CMB-DA scales well, the plots being nearly straight lines. The event density 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**. As an example, for the case of 36 processors the speedups are 24.3 and 21 respectively. The larger workload assigned to the LPs in **8P** makes them achieve higher performance. Figure 6.8 characterize one of the behavioral differences for both experiments; the number of null messages needed to keep the simulator synchronized in **8P** is very low, less than a 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, but null messages are not the only source of overhead; if many null messages are being sent, this is because LPs are blocking very often, spending time awaiting for incoming messages to increase the acceptance horizon (to allow simulation to progress). Figure 6.9 has been obtained after running both experiments with an instrumented version of CMB-DA which allows to monitor the way PEs use their time. Total execution time has been divided into three components:

Tsim: time spent executing events, inserting messages in the event calendar and in the input queues, and sending messages to other LPs. This is the time the LP devotes to advance in 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, i.e., prevents the LP to advance until a message is received, this time can be considered mainly synchronization effort.

Thn: time spent performing other synchronization tasks, namely:

- Sending null messages. This includes computing if a null message is necessary or not, plus the actual action of sending a necessary null message.
- Computing, at each step of the simulator main loop, the message acceptance horizon.

The conclusion drawn from Figure 6.9 is clear: if the workload assigned to a LP (and, thus, to a PE) is low, the PE spends too much time synchronizing instead of executing events. In the case of **7P** in 64 PEs, each PE simulates 3×3 routers and the achieved proportion of effective simulation time is 55%. In contrast, for **8P** in 100 PEs, each PE simulating 9×9 routers (9 times larger that in **7P**), CMB-DA 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 remarked speedup difference.

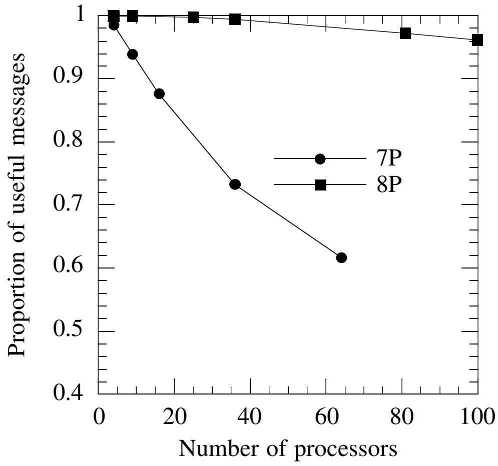
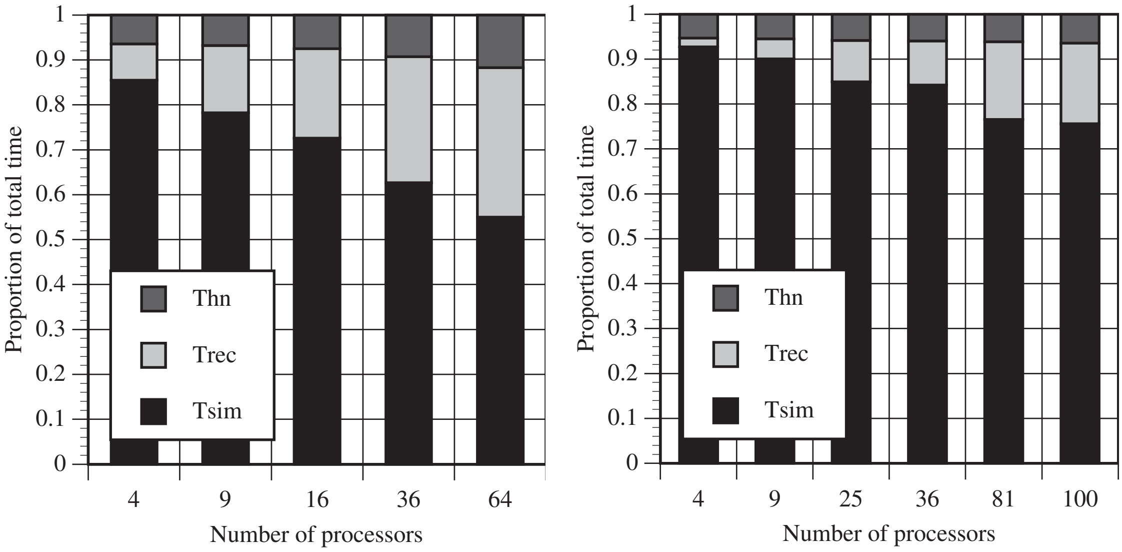


Figure 6.8. Proportion useful/total messages for experiments **7P** and **8P** with CMB-DA.



Experiment **7P**. 24×24 routers, 4—64 PEs.

Experiment **8P**. 90×90 routers, 4—100 PEs.

Figure 6.9. Distribution of total execution time among simulation and synchronization for experiments **7P** (left) and **8P** (right) using CMB-DA.
Thn = time spent computing the acceptance horizon and sending null messages; Trec = time spent receiving messages (including null messages);
Tsim = time spent executing events and sending non-null messages.

6.2.5 Description and results in the NOW

The number of Sun workstations available to experiment with the MPICH implementation of MPI was very limited, just 5, but from the experience gained with the Supernode and the Paragon, speedups over 2 might be expected with the CMB-DA simulator running in 4 workstations. We made the experiments described in Table 6.6 to confirm this hypothesis.

	1M	**3M**	**5M**	**6M**	**8M**
Network size	16×16	16×16	32×32	32×32	90×90
Message length	4	32	4	32	4
Load	5—90	5—90	5—90	5—90	5—90
Number of PEs	4	4	4	4	4
Grain size	Max.	Max.	Max.	Max.	Max.
Lookahead	No	No	No	No	No
Results	Fig. 6.10	Fig. 6.10	Fig. 6.10	Fig. 6.10	Fig. 6.10

Table 6.6. Experiments performed with MPI in a network of workstations.

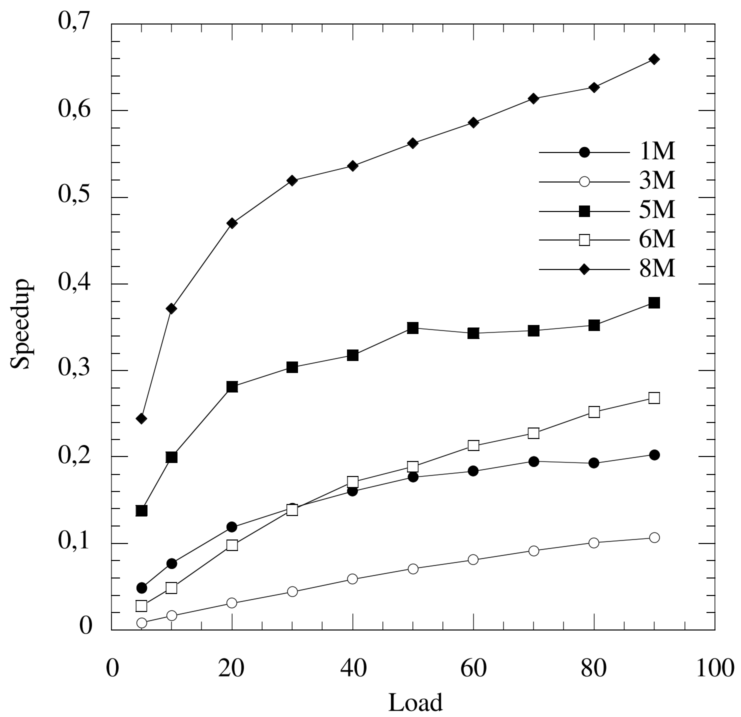


Figure 6.10. Results of the experiments with SPED on a network of workstations with MPI.

Note that experiment **8M** is no longer a scalability test, because of the few available workstations. The other experiments are similar to their counterparts in the Supernode and the Paragon, to ease comparisons. The results (speedup curves) are all plotted in Figure 6.10.

It is easy to observe how the performance improves when:

- Problem size is increased: compare **1M** (16×16), **5M** (32×32) and **8M** (90×90).
- 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 comes at not surprise after the experience with the Supernode and Paragon implementations. The outstanding point is that the overall results are really poor. As the programs are the same we used in the Paragon (except for minor details), we must find the explanation of these figures in the characteristics of the computing system used in the experiments.

The main difference between the Paragon or the Supernode and the NOW is the way interprocess communication is achieved. In the three cases a message passing mechanism is used for synchronization and communication, but in the Supernode and the Paragon a high-speed special purpose interconnection network is used, while in the network of workstations a general purpose Ethernet network, with the TCP/IP protocols over it, provides the necessary connectivity. This means that communication in this environment is relatively slow, because of:

- The peak data rate of Ethernet: 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.
- The shared nature of Ethernet. The available data rate must be shared among all the devices connected to the network, being or not part of the simulation environment. 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 software overhead imposed by the use of several layers of protocols (Ethernet, IP, TCP, MPI). The communication protocols used inside a multicomputer are much simpler; in particular, there are not as many layers. As layering means encapsulation, i.e., addition of control information, its effects are worse for short messages than for long messages[3].

To compute with more accuracy the communication capabilities of these three 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 following one, which simply executes a store-and-forward procedure, sending 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, and 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 6.11. 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.

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 NOW, although the Supernode offers a good rate. The achieved data rate for 32-byte messages has been indicated in Figure 6.11 by means of a line cutting the curves at the appropriate points. Table 6.7 also summarizes this information.

---

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

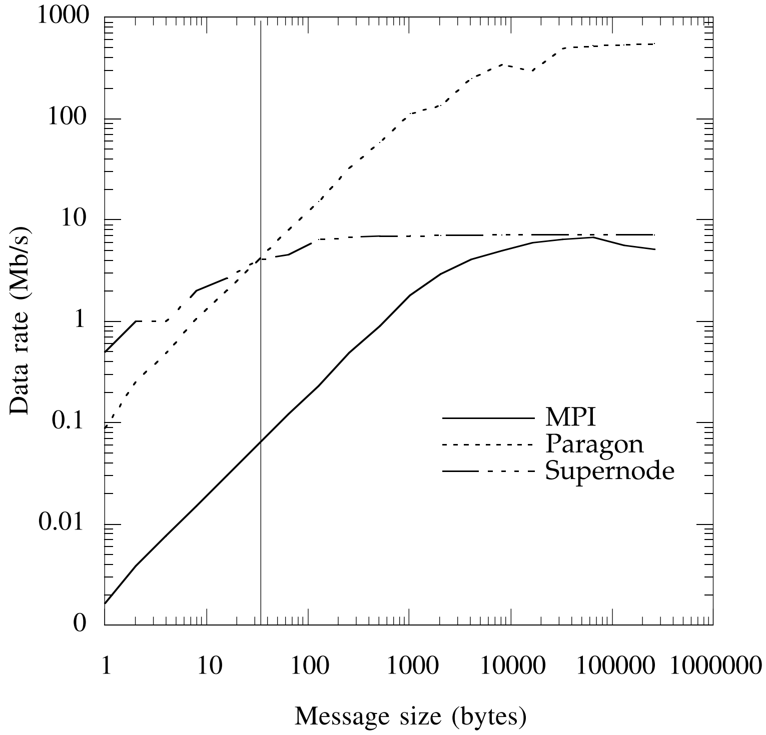


Figure 6.11. Achieved data rates as a function of the message size in the three multicomputer systems used in the experiments.

The conclusion is that the computation/communication ratio of the MPI system is not as good as in the other two systems, unless the CPUs of the workstations were proportionally slower, and this is 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 6.7 summarizes the execution times of the sequential simulator running this experiment at load 90. A Sun SPARCstation 5 is slightly *faster* than a Paragon processing element, and about 7 times faster than a T800 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. It is clear that the computation/communication ratio is too biased towards the computation power.

Computing system	Data rate (Mb/s) for 32-byte messages	Execution time of the sequential simulator running experiment 5 at load 90
Supernode	4	5597
Paragon	4	904
NOW with MPI	0.06	817

Table 6.7. Communication and computation abilities of the multicomputers used in the experiments.

Our conclusion is that the communication demands of CMB-DA (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-DA works specially well in fine-grain parallel computers, while a network of workstations might be used efficiently only for coarse grain problems.

6.3 Experiments with TW

In this section we discuss our experience with a TW implementation in the Paragon. Preliminary tests done with TW with conservative time windows showed very poor performance figures, much worse than expected after the experience reported in Chapter 4. We verified that the big size of the state to save before executing each event was partially responsible for the poor results: the state of a LP is the combination of all the state variables which represent the collection of routers assigned to that LP, plus a small number of variables for statistics gathering. Obviously this is excessive, because an event only modifies the state of one router, plus the statistics. This fact was what led us to implement the incremental state saving optimization: every *N* events the full state is saved, and in the remaining cases only the state of the affected router is saved. After including this optimization the execution speed was doubled, but still very far from that of the CMB-DA algorithm. In fact, for all the experiments, even the sequential simulator was always *much faster* than TW.

From our experience, we consider that TW is not a suitable approach for the kind of experiments we perform. The density of events is, in general, very high and, for this reason, the probability of a straggler to appear is very high too. This leads to continuous rollbacks in the LPs. It has been monitored that almost all the events executed in a speculative way have to be undone. The result is that the simulator looses most of its time doing the following “management” activities:

- Storing state copies and antimessages.
- Traversing the input queue to locate events to annihilate, or to find the right place to insert a straggler.
- Coast-forwarding to construct appropriate state versions.
- Sending and managing antimessages.
- Computing the global virtual time.

In the list of sources of overhead we do not include the insertion of future events in the input queue, or the extraction of the next event to simulate, because those operation must be done in any simulator. However, as many of those insertion/extraction operations are speculative and have to be undone and done again, they also contribute to the overall overhead.

In is interesting to observe that, despite the good deal of effort devoted to the implementation of the TW simulator in a distributed memory environment, its performance remains discouragingly low. This seems to be an exception to the observations by Fujimoto in [Fuji90a], one of the most comprehensive evaluations of TW: "*thus far, the experience of researchers (with TW) has been that (situations where most of the time is spent executing incorrect computations and rolling them back) are seldom encountered in practice, and, when discovered, usually points to a correctable flaw in the implementation rather than any fundamental flaw in the algorithm*". It should be mentioned that these observations were made in the context of a shared memory implementation of TW.

As also pointed out in [Fuji90a], state saving overhead seems to be in the origin of this poor performance. Fujimoto concludes that "*State-saving overhead limits the effectiveness of TW to applications where the amount of computation required to process an event can be made significantly larger than the cost of saving a state vector.*" This is not the case for this particular application: the cost of executing an event is very low, while the state size is huge. Even when incremental state saving is used the amount of data moved for state saving is several times larger than the data moved to execute an event.

It can be concluded that TW is not a viable approach for the parallel computer and the domain of models presented in this study. This must be said without implying that TW is not viable in any context: many references can be found in the literature (some of those are mentioned in Chapters 2, 4 and 5) that report successful application of this synchronization method, in different hardware platforms and with different problem domains.

6.4 Experiments with SPED

After performing a series of experiments with the SPED simulator in the Paragon, it was observed that it behaves very much like CMB-DA, with only a significant difference: SPED degrades sooner when the load of the LPs is reduced. Figure 6.12 (left) shows the execution time of these two simulators when running experiment **7P** (see Table 6.5). The scale is logarithmic to better appreciate the differences. Figure 6.12 (right) shows the speedups achieved by both simulators as a function of the number of PEs. In the same way, results for experiment **8P** are summarized in Figure 6.13.

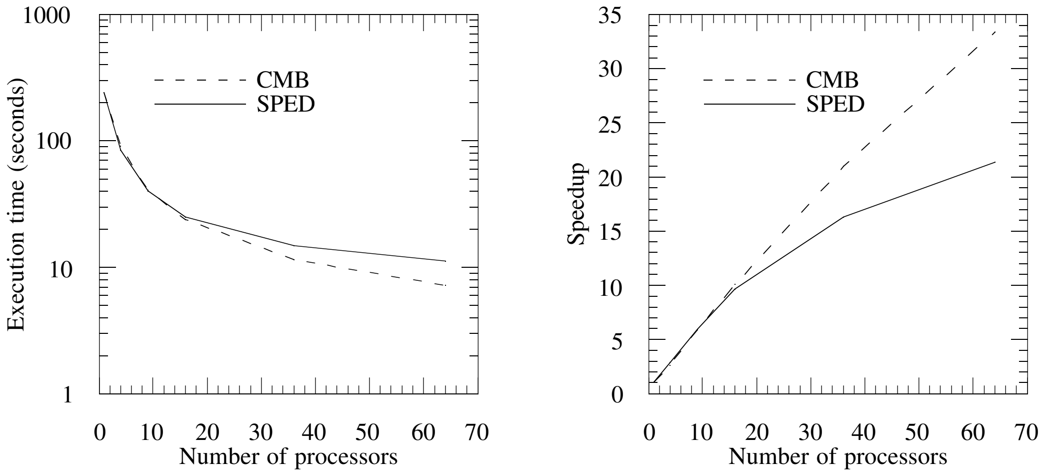


Figure 6.12. Execution times and speedups of SPED and CMB-DA running experiment **7P**.

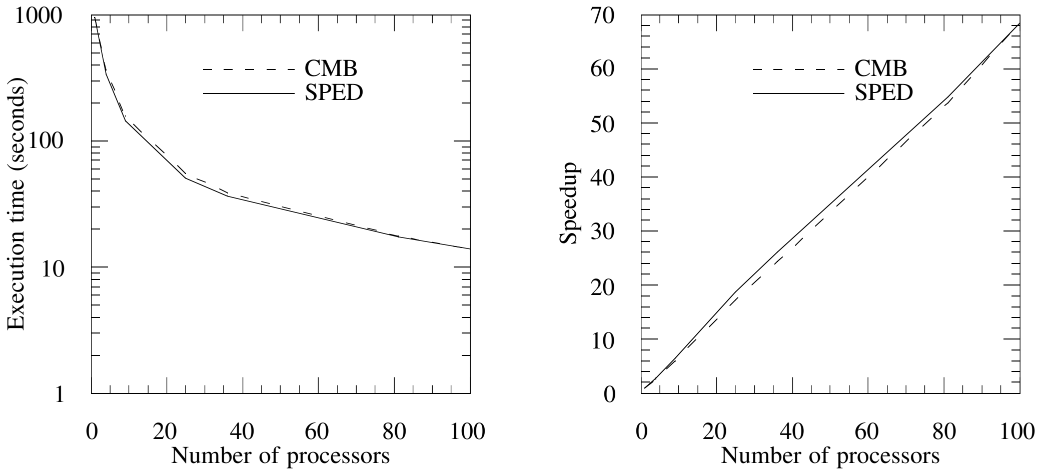
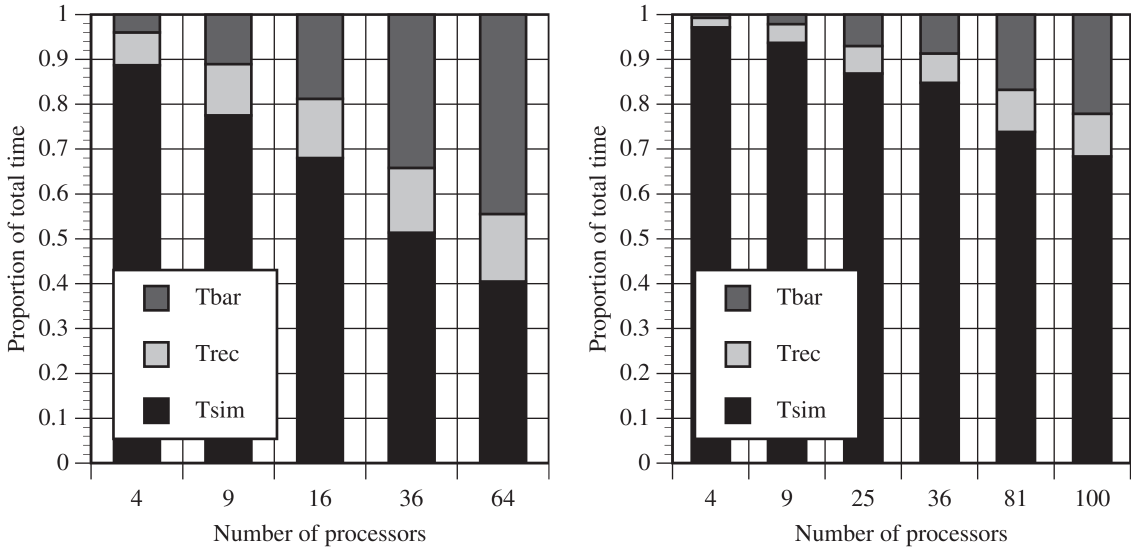


Figure 6.13. Execution times and speedups of SPED and CMB-DA running experiment **8P**.

Starting with **7P**, we can see that SPED performs slightly better than CMB-DA for small numbers of PEs. However, when the number of PEs increases (and, therefore, the workload of each LP decreases) CMB-DA performs better than SPED. Figure 6.14 (left) shows the contribution of computation and synchronization to LP execution time when running this experiment. **Tsim** is the time devoted to simulate events and send messages. **Trec** is the time spent awaiting and receiving messages. **Tbar** is the time spent barrier-synchronizing. The Figure is self-explanatory: the problem size of experiment **7P** is large enough to keep 4 PEs busy most of the time but, when distributed among 64 nodes, each node needs most of its time for synchronization (60%), performing useful computation only a 40% of the time.

Experiment **8P** works with a much larger simulated network (90×90 routers), which imposes a higher workload to the LPs. The results shown in Figure 6.13 indicate that, in this case, SPED performs slightly better than CMB-DA, even using 100 Paragon PEs. Besides, both simulators scale equally well. For this size of model the workload is large enough to keep all the LPs busy, performing useful computation most of the time and minimizing the (relative) effort devoted to synchronization. This can be confirmed in Figure 6.14 (right); see how LPs spend most of their time performing effective simulation, even when the simulator is distributed among 100 PEs.

Although the effective simulation time of SPED executing **8P** on 100 PEs is slightly below 70%, this figure is over 75% for CMB-DA (see Figure 6.9). CMB-DA should, therefore, perform better than SPED. However, this difference is only due to the way both programs are instrumented. The synchronous nature of SPED allows for a more precise measure of the time spent in the different activities a LP performs at each iteration. In CMB-DA the measurements are taken in a event-by-event basis, which means a significantly higher number of calls to the function that gives the real time clock value, and this affects the achieved accuracy.



Experiment **7P**. 24×24 routers, 4—64 PEs. Experiment **8P**. 90×90 routers, 4—100 PEs.

Figure 6.14. Distribution of total execution time among simulation and synchronization for experiments **7P** and **8P** using SPED.
Tbar = time spent barrier-synchronizing; Trec = time spent receiving messages;
Tsim = time spent executing events and sending messages.

The number of barriers executed by the LPs was observed to be 4000 (1000) in all the cases of **7P** (**8P**), i.e., as many barriers as simulation cycles. The reason is simple: the density of events is big enough to always have at least one event per cycle. Only for some particular cases with small models and very low loads (less than 5) the number of barriers was found to be lower than the number of cycles. Thus, we can conclude that, in general, SPED behaves as a time-driven algorithm. In fact, some experiments have been done with a parallel time driven algorithm, and the execution times were the same, except for the mentioned very low-loaded cases.

A good characteristic of this simulator is that it obtains reasonable speedups using a very simple algorithm, without the complicated memory management strategies of TW, and without some of the restrictions of CMB (no cycles with zero timestamp increment).

6.5 Conclusions

In this chapter we have presented our experiences using three parallel discrete event simulation strategies to study a model of a message passing network designed for its use as the communication infrastructure of a multicomputer. The knowledge obtained from previous experiments with a very simplistic model has been applied, when possible, to the simulation of this, more realistic, model.

In general, we have confirmed our preliminary ideas about which characteristics of the simulated model help to improve the performance of the conservative parallel simulator. In order to take advantage of using CMB-DA, we need a model with a high degree of internal communication, which allows the 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 the best scenario for the CMB-DA simulator is the worst for the sequential event-driven simulator, a very convenient situation because speedups are best precisely when they are most needed, and are poor only in cases where simulation runs are very short in both the sequential and the parallel version.

If CMB-DA is running in 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 effort is reduced. 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, if the host parallel computer allows it, more than one process should be mapped onto each processor. We have seen how, in the Supernode, using intermediate grain sizes always led to the best performance.

The knowledge of the behavior of the model may allow CMB-DA to exploit some lookahead information, which helps maintaining a good performance when the simulator has not useful work to do. Unfortunately, the lookahead ability of our network of message routers is, in general, poor, particularly when a LP simulates a set of routers instead of only one (that is, when the grain size is not minimum). It has shown more advantageous to use intermediate or maximum grain sizes, even if that means to renounce to exploit the lookahead of the model.

It has been noticed that the communication demands of CMB-DA are very strong, making it unsuitable for environments such as a network of workstations, were communication costs are very high compared to computation costs. While the performance achieved in two multicomputers, Supernode and Paragon, are reasonably good (better in the latter than in the former), a collection of workstations in an Ethernet network does not perform so well.

The experience with optimistic synchronization has been quite discouraging. The performance of the tested TW implementation is very poor, even after introducing a series of optimizations. The conclusion is that TW is not the right tool for the kind of study presented here.

Finally, the experience with SPED, the synchronous parallel simulator, has been very positive. It performs worse than CMB-DA only for very lowly loaded systems, being as efficient as CMB-DA (or even more) for medium and highly loaded systems. It has the additional advantage of a simpler implementation, and can be used in environments were CMB-DA is not applicable because of the presence of loops with zero timestamp increment.

Chapter 7

Conclusions and future work

Throughout this dissertation we have made a study of techniques for parallel simulation of discrete event systems, with special attention to the simulation of a particular kind of systems: models of message routers for massively parallel computers. We started reasoning the interest of this study: the need to accelerate simulations, and the consideration of simulators as interesting applications for its implementation in parallel computers.

The identification of causal dependencies among events allowed the relaxation of some constraints that sequential simulators impose on the order events are simulated, in such a way that several events can be processed in parallel, after splitting the simulator into a collection of collaborating logical processes. However, a synchronization mechanism must be added to guarantee that the collection of logical processes progresses, as a whole, in an consistent way.

We have presented three different synchronization alternatives: *synchronous*, *conservative* and *optimistic*. SPED, CMB-DA and TW are particular realizations of those alternatives, which have been implemented and tested, first to study a toy model designed to analyze the general characteristics of each simulator, and then using a model of a message router to test how well these simulators work with a real world application.

Implementations of the three simulators have been performed in three different multicomputing environments: a transputer based multicomputer, a Paragon and a network of workstations. As the programming environments available for these multicomputers are not identical, a study of their characteristics has been done, identifying the differences that have an important impact on the way algorithms can be implemented.

Given a model, three algorithms and three multicomputers, an exhaustive set of experiments has been done, allowing us to discover how the execution times of simulations depend on different characteristics of the synchronization algorithm, the model under study and the target multicomputer.

The contributions of this work can be summarized in the following points:

1 A description of a collection of alternatives for parallel simulation of discrete event systems, with an evaluation of them using a toy model designed as a stress test. Similar evaluations can be found in the literature, most of them using shared memory multiprocessors, which allowed the implementation of a variety of optimizations. However, our work has been developed in distributed memory systems where message passing is the only means of communication and synchronization.

2 A description of three different environments for parallel programming, identifying those differences among them that have an impact on the implementation of parallel algorithms. In this context, parallel simulation algorithms can be considered a case study on parallel programming.

3 A detailed description of a model of message router, along with the way it can be simulated using an event-driven approach. A description of this model using the C programming language has been done, able to be used by any of the simulation engines implemented as part of this work.

4 The implementation and analysis of five parallel simulators, using three different synchronization mechanisms and three host multicomputing systems:

- CMB-DA in a transputer-based Supernode multicomputer
- CMB-DA in an Intel Paragon multicomputer
- CMB-DA in a network of Sun workstations with a MPI library
- TW in a Paragon
- SPED in a Paragon

Additionally, a sequential simulator has been implemented and tested in the three machines, to use the obtained results as a reference point to compute speedups.

5 The design of an optimization on CMB-DA that allows a reduction on the number of null messages used for synchronization: null messages are sent only when a logical process is going to block and *only if they have a positive impact on the receiving LP*. Additionally, null messages are not stored in the receiving LPs: the only effect of the reception of a null message is an increment in the receiving channel's clock. With this optimization, the synchronization effort of the CMB-DA simulator is notably reduced.

6 The introduction of the concept of grain size of LPs in a CMB-DA simulator, when running in a distributed memory parallel system. The synchronization mechanism of CMB-DA requires LPs to block frequently, while they await until it is sure that advance is possible without causal risks. If only one LP is assigned to each processing element, CPU power is wasted while the LP is blocked.

Assigning several LPs of smaller size to each processing element (which means that each LP simulates a smaller part of the system under study), the CPU can be relinquished from a blocked LP and assigned to another, active one. However, LP's grain size should not be too small, because in that case the overall number of LPs would be notably increased, and much more null messages would be needed to keep the simulator synchronized. Unfortunately, certain operating system support is needed to take advantage of this multi-LP per PE approach, in the way of a scheduling mechanism that switches contexts as soon as a LP blocks, and some parallel systems do not provide it.

7 The characterization of the event density of the simulated systems as an important parameter to achieve good performance in CMB-DA and SPED simulators. In CMB-DA, models with high event density achieve an optimum performance, because the message interchange for event scheduling provide the LPs with the necessary synchronization, and a negligible number of null messages are required. In the same conditions, SPED performs equally well, because all the LPs have a similar work to do between barriers, exploiting parallelism efficiently. If the message density reduces, the performance also reduces, being SPED more sensitive to this change.

8 The way of exploiting the lookahead ability of the model of message router used for our experiments. Good levels of lookahead can be obtained in highly loaded systems where messages are long, because the lookahead is proportional to the message length and the frequency of interaction between routers. A minimum of one cycle is always guaranteed. Unfortunately, when a LP simulates an aggregate of routers instead of only one (and this is the case for grain size alternatives different to minimum) the obtained lookahead values are too low for the complexity of the involved computation. After a set of evaluations, it was shown that it is more advantageous to use medium or maximum grain size LPs without lookahead ability instead of minimum grain size LPs with lookahead ability.

9 A characterization of CMB-DA and SPED as highly scalable algorithms. Provided a large enough model, an increasing number of PEs results in an increase of performance. A problem size fitting the characteristics of the multicomputer is needed, though, in order to keep the computation/communication ratio properly balanced. The two multicomputers used in this work (Supernode and Paragon) provide a fast enough message passing mechanism to exploit the parallelism available in the simulators. In contrast, the network of workstations does not perform so well, because the communication infrastructure (based on TCP/IP over Ethernet) is not fast enough.

10 The conclusion that TW is not a suitable approach to the parallel simulation of the kind of models we are working with. The fine grain nature of the model, along with the large size of the data structures that represent the state of a LP, make it the worst possible scenario for TW. The implementation of some optimizations, like conservative time windows and incremental state saving, improves the performance of TW, but not enough to make this approach viable. We do not try, however, to invalidate this synchronization mechanism: experiences by other researchers confirm that, in different scenarios, it can be very effective.

11 Finally, we conclude that CMB-DA and SPED can be considered as suitable approaches for the parallel simulation of message passing networks. The main requirement is the availability a fast message passing mechanism, like those offered by current multicomputers. The more demanding the model to study, the higher the possibility of obtaining a good level of acceleration. Using 100 processors to simulate a considerably big model (8100 routers) speedups up to 70 have been obtained.

There are many ways to further extend the work presented here. The most appealing one, given the interests of our research group, is to build an analysis tool for message passing networks, based on a conservative or a synchronous parallel simulation engine. The tool should be able to allow a researcher to describe the model under study, given a collection of parameters such as network size, network topology, routing strategy, flow control mechanism, message length, link width, etc. A description language or a graphic environment should be designed to offer a way of providing this information in a user friendly way. This tool would be even more useful if the simulation could be done at several levels of detail:

- In a *high-level* simulation each node of the network could be considered as a simple queue-server element. Simulations would be fast, and measurements like overall network throughput could be obtained.

- A *medium-level* simulation could be like that presented in Chapter 5: the main elements which constitute a node could be identified and studied separately. The simulation effort would be increased, but measurements like the usage of internal queues could be provided by the simulator.
- A *low-level* simulation could be done at the gate level, considering the actual implementation of the elements which constitute a node. With this level of detail not only the design, but also the implementation of the nodes could be tested, at the cost of a slow simulation. For this case the availability of a parallel simulation engine would be specially convenient.

In relation to the SPED simulator, the current implementation follows a SPMD programming model. It would be very interesting to implement a version for SIMD machines (as the CM-2 or the MasPar) in order to test its behavior using several thousands of processing elements, instead of a few tens.

This study has, fundamentally, an empirical basis. An analytical study of the way the different characteristics of models, simulators and target multicomputers affect the execution time of simulators should be of great interest. The difficult part is that the spectrum of parameters to consider is quite large; several studies have been done in this direction, but most of them make a series of simplifying assumptions which limit its applicability.

Appendix A

Parallel systems used in this work

A.1 Introduction

In this appendix we present the three computer systems used for our research in parallel simulation. Our first work was made using a Supernode, a transputer-based multicomputer at the Facultad de Informática of the Universidad del País Vasco / Euskal Herriko Unibertsitatea. This machine allowed us to practice with parallel programming, and to develop the core of most of the simulators. Then the research continued in another multicomputer, an Intel Paragon system at the Purdue University Computing Center. This machine is bigger, faster and easier to deal with, compared to the Supernode. Finally, as the use of clusters of workstations is becoming more and more popular as a platform for parallel programming, we also made some experiments using an implementation of MPI over a collection of Sun SPARCstations connected to the Engineering Computer Network, also at Purdue University.

A separate section is devoted to each system: Supernode, Paragon and MPI in a network of workstations. We give a description of the configuration of each system, the programming environment available at each platform and the libraries used to build the actual programs. Our intention is neither to give a tutorial on programming in each environment (although some of the information could be helpful for a researcher interested in using any of these systems), nor to evaluate one system against the others. However some differences between the systems are remarked, specially when they have an impact on the way parallel algorithms are implemented.

To finish this introduction, the author wants to acknowledge all the support received by the personnel at these locations:

- *Centro de Cálculo de la Facultad de Informática de la UPV/EHU*, for their help installing and maintaining the Supernode and the other computers at this location, and for providing the means to access these systems even from the other side of the Atlantic Ocean.
- *Purdue University Computing Center* (Research Computing Division) for their invaluable help using the Paragon. Most of the information about this machine has been obtained through the personnel at the Advanced Applications Group of this division of the PUCC.

- *Parallel Processing Laboratory* of the School of Electrical and Computer Engineering, at Purdue University, for allowing us to use the network of workstations described in this appendix.
- *Engineering Computer Network* at Purdue University, for providing our everyday computing support at Purdue, including the means to remain connected to the UPV/EHU and to the rest of the world.

A.2 The Supernode system

In many of the experiments presented throughout this dissertation we have used a Parsys Supernode SN-100, a multicomputer with 34 Inmos transputers of the T800 family as processing elements [Pars89, Inmo89b]. The interconnection network of the transputers in the Supernode is not static: one of the most salient characteristics of this machine is that all the transputers are connected to a collection of switches which allow a dynamic configuration of the interconnection network in a nearly arbitrary way, obeying these two limitations:

- Each transputer has 4 links: 0 or North, 1 or East, 2 or West and 3 or South (NEWS ordering).
- The switches only allow connections N-S and E-W.

The T800 transputer is a microcomputer in one chip, composed of the following elements: a 32 bit RISC processor, 4 KB of local memory, a floating-point unit, an external memory interface and a communication subsystem. All these elements are connected by means of an internal 32-bit wide bus. The communication subsystem controls 4 serial links which allow a transputer to communicate with the external world or with other transputers. The data rate of the links can be selected among various values, being 10 Mb/s the one used in our machine.

Our Supernode also has an Ethernet adapter to connect it to SPRINet (the corporate network of the UPV/EHU, which is part of the Internet) and a SCSI adapter controlling two storage devices: a hard-disk and an Hexabyte tape. A personal computer is connected to the Supernode for booting and diagnosis purposes.

The operating system that controls the Supernode is Idris, the Parsys version of Unix. There are two alternatives to design and run parallel applications in the Supernode:

- The user can decide to design parallel applications using the traditional Unix interfaces for distributed programming: sockets or RPCs. To operate this way, each transputer has to house a complete copy of Idris, and a set of links connecting the transputers has to be defined and kept fixed, because the definition of that topology is needed by Idris to route the messages from a process to another. In addition to the AF_UNIX and the AF_INET address domains, an AF_TLINK domain is provided, to use a simplified protocol (instead of TCP/IP) over the transputer links to communicate pairs of processes. All the transputers have access to the I/O capabilities through simplified versions of the Internet protocols, adapted to run over the AF_TLINK domain. In particular, a network file system is implemented, FPS (derived from NFS), to allow file system access to all the transputers without the SCSI adapter. An advantage of this way of working is that most applications designed run in a network of workstations can be easily adapted to run in a network of Idris transputers.

- The user can decide to ignore the inter-process communication provided by Idris, using raw transputers (that is, with no O.S. support) and a separate development tool, like Inmos C, Inmos Occam or 3L Parallel C. At least one node has to run Idris to manage I/O. This way of working is much more efficient: more memory is available (a copy of Idris uses more than 2 MB of memory) and the communication capacity provided by the transputer links can be used without all the overhead that the Idris communication impose. However, the development environment is much more limited, and the application is not portable outside the transputer environment. An additional disadvantage is that a raw processor cannot be used by more than one user at a time, making the system of poor utility in a multi-user environment.

The usual approach is to find a balance between the extremes: some of the transputers are arranged in an Idris network, while the others form a pool of raw transputers. Our particular arrangement is shown in Figure A.1. There are 5 Idris transputers, named 1 to 5, connected in a ring fashion. These are T805 transputers, with a 30 MHz clock and the amount of external memory indicated in the Figure. The first one has a built-in SCSI adapter, and its four links are connected to the boot PC, an Ethernet board and transputers 2 and 3. Note that all the links of transputer 1 are used, while the other four transputers have free links.

The 29 remaining transputers form the raw transputer pool. There are 14 T805 at 30 MHz with 4 MB of external memory, and 15 T800 at 20 MHz with 2 MB of external memory. The peak performance for the 30 MHz T805 is 30 Mips and 4.3 Mflops, while for the 20 MHz T800 is 20 Mips and 2.2 Mflops.

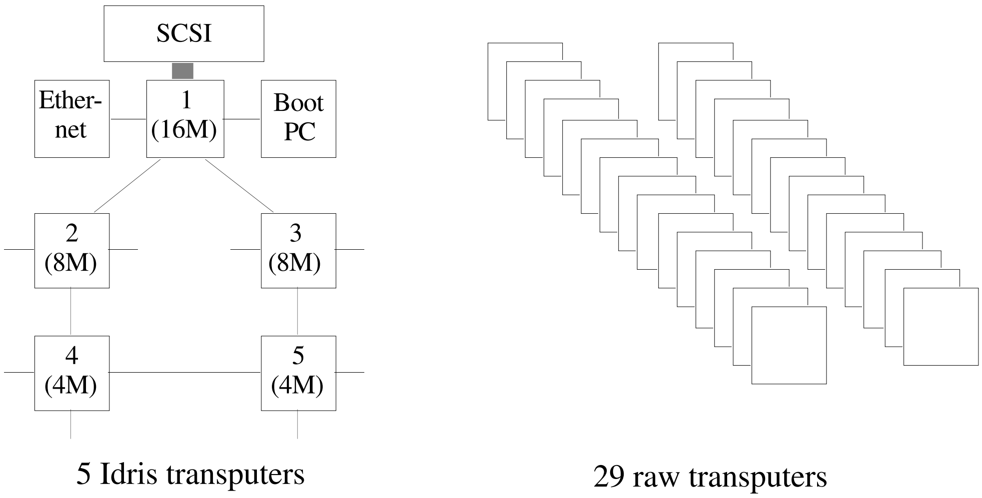


Figure A.1. Classes of processors in the Supernode system.

The external view of the system is as follows: a user can log in any of the Idris transputers (each one has a different IP address) and program and run Unix applications using, if desired, the AF_TLINK domain for interprocess communication. Alternatively, one of the 4 Idris nodes with free links can be used as a front-end to prepare and run programs in a network of raw transputers. To do so, a special tool is provided: nm. Nm is able to:

1 manage domains of raw transputers, that is, groups of transputers assigned to a user, and

2 program the Supernode switches to arrange the links of the transputers in the domain in the way the user wants (obeying the limitations stated before). One link of one raw processor must be connected to the Idris front-end. The directly-connected transputer is the *root transputer*, and the link is the *boot link*.

Figure A.2 shows a feasible scenario: one user is working in the Idris node 2, using it as a front-end to execute raw-transputer applications. Using nm, the user has gotten a domain (number 10) with 9 transputers, and has arranged them as a 3x3 mesh. Simultaneously, another user working at Idris node 5 has 8 transputers arranged as a ring in domain 14. As several raw transputers remain in the pool, and two Idris nodes are still available, up to two more users could comfortably work with networks of raw transputers. Note that other users could be using the Idris processors to run Unix applications, in one or several nodes.

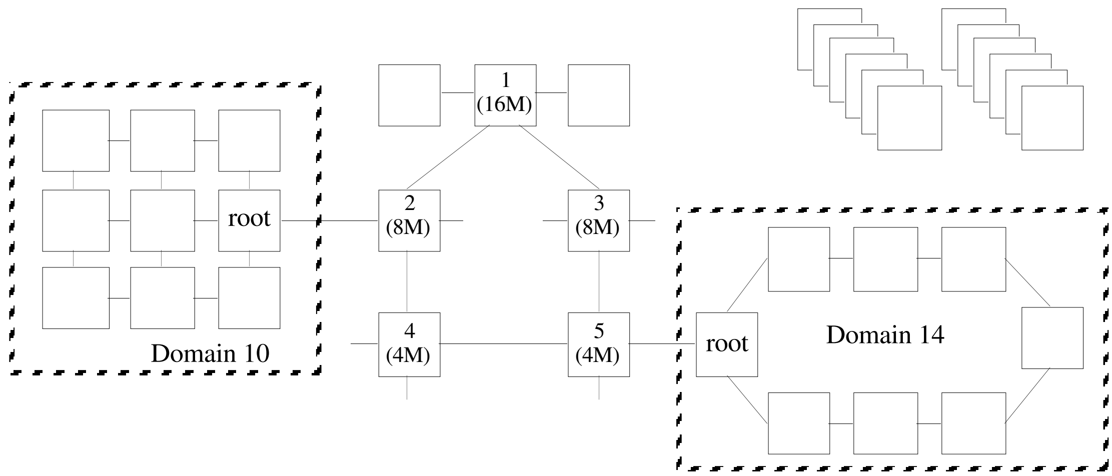


Figure A.2. Two domains of raw transputers arranged to run two separate applications.

The root transputer in a domain is the only one (unless special software arrangements are done) with access to the I/O capabilities of the Supernode. While an application is running in the raw transputer network, a program called iserver runs in the front-end Idris processor, and offers a complete set of I/O services through a special protocol over the boot link.

Programs are edited in the Idris front-end, where a set of tools are used to compile and execute applications. Those tools run in the root transputer: the Idris processor boots the tools into the root transputer, sends the input files through the boot link and stores in the file system the output files received through the link. For each tool, a script is provided so the user does not have to worry about where the tool runs, once a domain with a transputer connected to the front-end is set up.

As mentioned before, there are several alternatives for programming networks of raw transputers. We have used the Inmos C toolset, a collection of tools which provide an ANSI C development environment, enriched with libraries to efficiently program the transputers [Inmo90]. The main tools are listed in Table A.1, while the file extensions used by those tools are enumerated in Table A.2.

Tool	Description
icc	ANSI C compiler with concurrency support. Generates object code for many specific transputer targets.
icconf	Configurer. Creates configuration binary files from configuration descriptions.
icollect	Code collector. Creates bootable files from linked units or configuration binary files.
idebug	Network debugger. Provides post-mortem and interactive debugging of transputer programs.
idump	Memory dumper. Saves the image of a program onto disk, to allow post-mortem debugging.
ilink	Linker. Prepares linked units from sets of object files and libraries.

Table A.1. Main tools of the Inmos C Toolset.

Extension	File contents
.c	C source file.
.h	C header file.
.cfs	Configuration description source file.
.tco	Object file. Created by icc.
.cfb	Configuration binary file. Created by icconf.
.lku	Linked unit. Created by ilink.
.lib	Library file.
.lnk	Linker indirect file. Command file for ilink.
.btl	Bootable code file.

Table A.2. File extensions used by the Inmos C Toolset.

In addition to the Inmos C toolset, the programmer has most of the Unix application development tools available. Those include several editors, grep, make, etc., which run in the front-end Idris processor.

A.2.1 Preparing a parallel application to run in a network of raw transputers

The programmer has to prepare a collection of .c and .h files and a .cfs file. The .c and .h files contain the code of the application. The .cfs file is a description of the hardware/software that constitute the application. The steps that have to be done to obtain an executable application are depicted in Figure A.3.

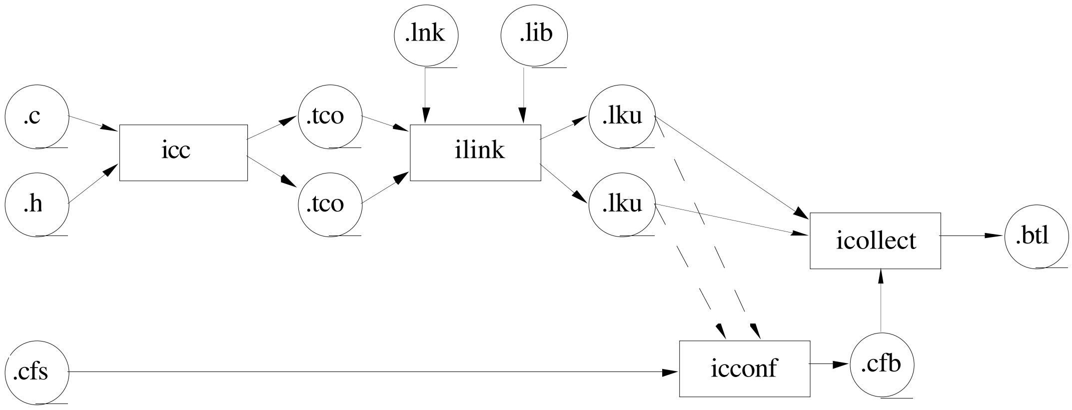


Figure A.3. Preparing an executable application in the Supernode.

A parallel application consists of a series of processes which run in one or more transputers. From the point of view of the programmer, two different classes of processes can be used: heavy-weight processes (H-processes) and light-weight processes (L-processes). From the point of view of the transputer, there is not such a difference. A H-process is a C program (with a main() function) which has to be compiled and linked to form a linked unit (.lku). All the H-processes constituting a parallel application are created when a .btl file is booted onto a network of transputers. A L-process has to be created at run time and put to run by a H-process, or by another L-process. Two H-processes can communicate only passing messages through channels, and can be mapped onto the same or different transputers. A L-process can communicate with its parent process (H or L) or with sibling processes using channels, and also by means of shared data structures, and runs in the same node where its parent is. If channels are used, they must be explicitly created. If shared memory is used, semaphores are usually needed to avoid race conditions.

When a programmer designs an application as a set of processes, a careful decision has to be made in order to determine which ones will be H-processes and which others will be L-processes. In general, when it is sure that two processes will always be allocated in the same node, L-processes can be used. H-processes allows an application to be designed to run in an arbitrary network of transputers, without any recompilation. The processes will be mapped onto the same or different transputers, and connected trough internal channels or external links, at the configuration step. Therefore, a design based on H-processes is more flexible, although it prevents any kind of shared memory-based communication.

The configuration file is a key piece of the application. It is a complete description of (1) the hardware where the application is going to run, (2) the software modules which constitute the application and (3) the mapping or projection of the software onto the hardware.

The hardware description includes the number of transputers, class (T212, T414, T800, etc.) and memory size of each one, and external link connections. At this point, it should be noted that it is only a description: the user is the final responsible for building the described network, using the Supernode nm tool.

The software description includes all the H-processes which form the application, with their input parameters. If a parameter has to be changed, the application needs to be reconfigured (but not recompiled). This description also includes all the channel connections between H-processes.

The final part tells in which .lku files the H-processes can be found, and how to map those processes onto the available transputers. It is also possible to indicate how to map logical channels onto physical links, but it is normally safer to let the configurer make the right decisions. Note that, although the H-processes may use many logical channels, when they have to be mapped onto the raw transputer network some restrictions have to be obeyed: (1) physical links can be simultaneously assigned to a maximum of two logical channels, one in each direction; (2) direct communication is only possible between H-processes in the same transputer (through an internal channel) or in neighboring transputers (through an external link); and (3) there are only 4 external links per transputer. The point is that it is not always possible to map a given software description onto a given hardware. Sometimes the programmer has to develop additional H-process to implement software multiplexers (to share channels among several processes), relays (to allow the communication between two processes allocated onto two non-neighbor transputers) or, in the most general case, software routers. It should be clear that those elements only appear to circumvent the limitations of the hardware/software platform that is being used, without being part of the application itself, and they are not required when the application is mapped onto a single transputer.

Figures A.4 and A.5 illustrate the previous concepts. Figure A.4a shows an application structured in a master/slave fashion. There is a master process (*m*) which sends tasks to any of its four slaves (*s1*, *s2*, *s3* and *s4*). The master is the only process connected to iserver (*isv*), so the slaves cannot use file or terminal I/O; this is a very common arrangement. Communication channels are unidirectional. For this reason, two logical channels are commonly set up to provide full communication between a pair of processes, one in each direction. Arrow heads are not shown for the sake of clarity in the drawings.

Figure A.4b shows a hardware network of 4 transputers: an Idris transputer which acts as the front-end (*fe*), a root transputer connected to the front-end (*r*) and two worker transputers (*w1* and *w2*). This network has to be constructed using the nm tool to get a domain, include 3 transputers from the pool into the domain and arrange the connections the way depicted. The hardware description part of the configuration source file has to be aligned with the constructed network; otherwise the application will not be able to be booted and run.

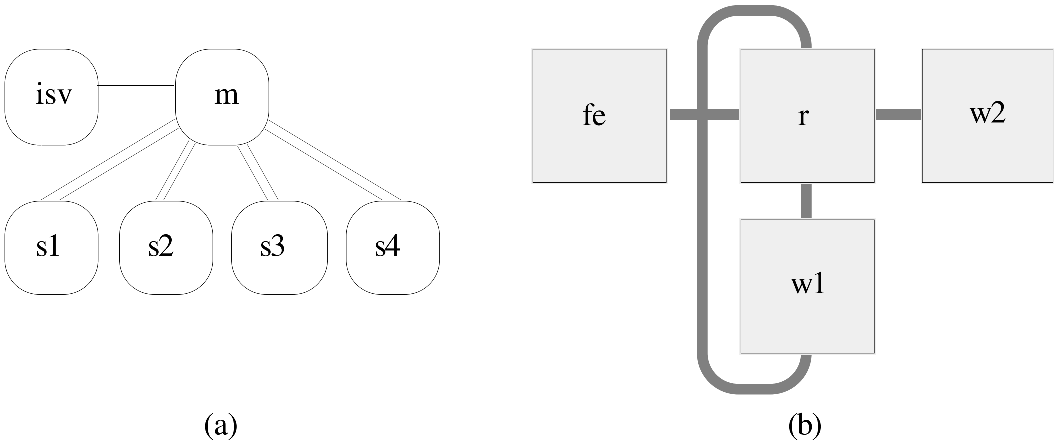


Figure A.4. Mapping an application onto a network of transputers.
(a) Example application. (b) Example network of transputers.

Next, processes and channels must be mapped onto the available transputers and links. Clearly, iserver will be running in the front-end transputer. The master will be mapped onto the root transputer, and the slaves will be distributed between the two worker transputers: *s1* and *s2* onto *w1*, and *s3* and *s4* onto *w2*.

Finally, channels connecting processes have to be mapped onto internal channels or external links. Each external link can support two channels, one in each direction. The communication between the front-end and the master presents no problem. Channels $m \leftrightarrow s1$ and $m \leftrightarrow s2$ are easy to map too, because there are two physical links connecting transputers *r* and *w1*. However, only one link joins transputers *r* and *w2*, while two pairs of channels are needed. To circumvent this limitation, the programmer has to develop a pair of multiplexer processes, *x1* and *x2*. These multiplexers allow several logical channels to share the physical link, of course without mixing up information belonging to different logical channels. The final mapping is shown in Figure A.5.

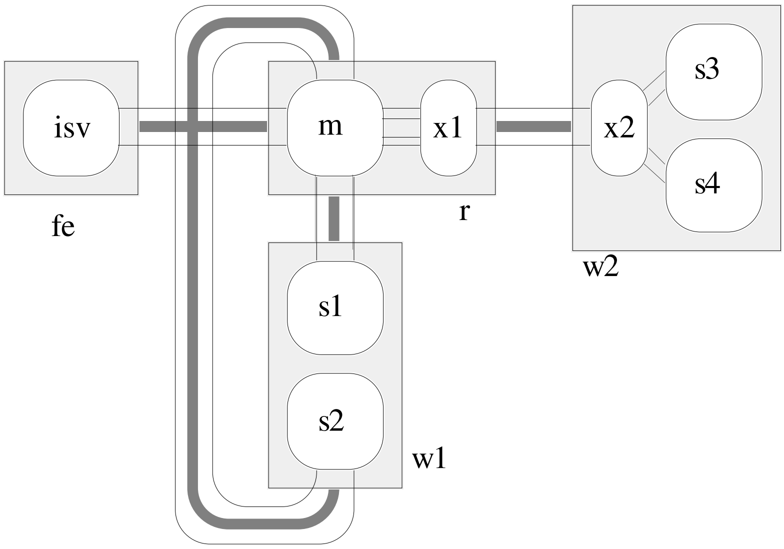


Figure A.5. A mapping of the application of Figure A.4a onto the network of Figure A.4b.

A.2.2 Running an application

When a .btl file containing an application ready to run has been obtained following the steps previously described, the actual way of executing the application is quite simple. The first step is to obtain a raw-transputer domain with the needed number of transputers, configuring the links in the way needed by the application; this has to be done using nm. The network has to be connected to an Idris transputer, through a boot link. Then the application can be put to run simply using the iserver tool, giving the name of the application file as a parameter. Iserver runs in the Idris processor, and boots the application through the boot link. This application is able to distribute itself along the available links, to put to work each H-process in its right place. Once the application has been booted, iserver keeps in its place, ready to satisfy I/O commands.

It should be mentioned that all the Inmos C tools, except iserver, are effectively .btl files. For example, when the programmer invokes icc in the Idris transputer, this command actually invokes a script that executes iserver, giving icc.btl as the application to boot. The compiler executes in a raw transputer, so before using any tool a domain with at least one raw transputer connected to the Idris front-end must be set up.

A.2.3 Libraries

In this section we describe the libraries that are part of the Inmos C Toolset, which allow to program parallel applications for networks of raw transputers using the C programming language.

A.2.3.1 The channel library (<channel.h>)

The channel library provides the basic mechanisms for interprocess communication. Processes can communicate by means of an interchange of messages through logical, unidirectional channels. The configurer has the ability to map a logical channel that joins two H-processes onto an external transputer link, or onto an internal channel (a word in the transputer memory). Additionally, it is also possible to create (ChanAlloc()), initialize (ChanInit()) and then use internal channels for communication between processes with a common ancestor.

The most important characteristics of channel communication functions is that they are blocking and synchronous: the collaboration of two symmetric parts is essential to succeed. When a process executes ChanIn(), it blocks until another process executes ChanOut() on the same channel. The two involved processes are allowed to continue only after the message is copied from the writer to the reader. Similar timing is applicable if first a process uses ChanOut() and later another process uses ChanIn().

This way of working makes applications extremely deadlock-prone. The programmer has to be very careful when organizing input and output parts of the program, and very often auxiliary processes and user-managed buffer space are needed to keep things working.

```
Channel *ChanAlloc (void);
void ChanInit (Channel *c);
void ChanIn (Channel *c, void *cp, int count);
void ChanOut (Channel *c, void *cp, int count);
```

A.2.3.2 The process library (<process.h>)

This library is a collection of functions to manage processes in one transputer. The transputer is especially designed to run several processes concurrently. It includes a built-in scheduler, able to manage two levels or priority (high and low); for each level a separate list of active processes is kept. High priority processes are scheduled in FIFO order, and run until completion or until they block for communication. Low priority processes are scheduled only when the high priority ready queue is empty, in a round-robin fashion, until termination, communication block or end of quantum. Context switches are very efficient, because processes are descheduled only at certain points where minimum state saving is necessary.

A process can create new L-processes and put them to run. Firstly, a Process data structure has to be allocated and initialized (ProcAlloc()), to do things as important as assigning a stack to the new process (space is taken from the heap), selecting the function that the new process will execute when started and passing a set of parameters to this function.

Once the Process structure is initialized, there are several functions to put a process to run. The simplest one is ProcRun(), which spawns a new processes to run concurrently with the calling process, whereas ProcPar() spawns a collection of processes which will run concurrently but blocks the calling process (the parent) until all the newly created processes (the children) terminate. In both cases, the priority of the children is the same as that of the parent. Other functions are available to explicitly indicate the priority of the new processes. A process can discover its own priority with ProcGetPriority().

```
Process *ProcAlloc (void (*func)(), int sp, int nparam, …);
void ProcPar (Process *p1, …);
void ProcPriPar (Process *phigh, Process *plow);
void ProcRun (Process *p);
void ProcRunHigh (Process *p);
void ProcRunLow (Process *p);
int ProcGetPriority (void);
```

It is also possible for a process to block for a given time with ProcWait() or until a specified time is reached with ProcAfter(). These two functions are related to the transputer built-in timer, whose value can be obtained with ProcTime().

```
void ProcAfter (int time);
int ProcWait (int time);
int ProcTime (void);
```

There is in this library a set of functions related more to communication than to process management. Those are the ProcAlt(), ProcSkipAlt() and ProcTimerAlt() functions. ProcAlt() allows a process to block until input is ready, but checking several channels simultaneously. The result is an index indicating a ready channel: a ChanIn() operation can be immediately done on that channel without blocking.

If it is not desirable for the process to block until any of the channels have information to read, ProcSkipAlt() allows for a test on all the channels, but without blocking: this function immediately returns the value (-1) if no channel is ready for input.

The last function, ProcTimerAlt() falls in the middle: it blocks the process but only for a maximum time, specified as a parameter. This function can return the index of a ready channel before the timer expires, or the value (-1) after timing out.

```
int ProcAlt (Channel *c1, …);
int ProcSkipAlt (Channel *c1, …);
int ProcTimerAlt (int time, Channel *c1, …);
```

A.2.3.3 The semaphore library (<semaphor.h>)

When two or more processes are running in the same transputer, and all of them have a common ancestor (which must be a H-process), they share the same memory space, including the heap, keeping private only their stacks. This shared memory space can be efficiently used for interprocess communication, as an alternative to channels. Nevertheless, a mechanism has to be used to avoid race conditions when accessing shared data structures. The semaphore library provides a set of functions for working with semaphores, which allow the implementation of mutual exclusion sections, as well as other ways of process synchronization.

The key data structure is the Semaphore. A Semaphore can be statically allocated in the global memory (or in the stack, but this is not common) or dynamically allocated in the heap (SemAlloc()) and then initialized (SemInit()) with the appropriate value, typically 1 for mutual exclusion.

```
Semaphore *SemAlloc(int value);
void SemInit (Semaphore *sem, int value);
void SemSignal (Semaphore *sem);
void SemWait (Semaphore *sem);
```

A.2.3.4 The input/output libraries (<stdio.h>)

When a program is booted from the front-end Idris transputer, that transputer runs a program called iserver, acting as an input/output provider. Many functions in <stdio.h> are implemented by means of a dialog with iserver. As only one channel is available to communicate with this server, only one H-process (and its children) can execute I/O functions. Nevertheless, it is possible to use a software multiplexer in such a way that several H-processes are connected (trough internal or external channels) to one multiplexer which, in turn, is connected to iserver. One of those H-processes can be another multiplexer connected to more H-processes. The conclusion is that it is quite simple to use I/O in one process running in the root transputer, but it is not so simple when processes in other transputers need to access files or interact with a terminal.

Not all the functions in <stdio.h> need to contact with iserver. In fact, there are two versions of the I/O library: complete and reduced. The programmer has to be careful when linking programs: if I/O functions are used, the complete library has to be linked. Otherwise, the reduced will be enough. If the complete version is used, the programmer has to define two logical channels (one for input and one for output) connecting the process directly to iserver or to a I/O multiplexer.

A.2.3.5 Other libraries and functions

Programmers can use all the standard C libraries: <string.h>, <stdlib.h>, <time.h>, <math.h>, <ctype.h>, etc. An additional library, defined in <misc.h> provide several hard to classify functions.

The most commonly used function in <misc.h> is get_param(). The mechanism that a H-process has to use to obtain the parameters assigned during the configuration is not that of the Unix operating system: there are not variables such as "int argc" and "char **argv". The process needs to invoke get_param() to read each parameter, making the appropriate type cast (because, unlike Unix, parameters need not be strings).

```
void *get_param (int n);
```

The rest of the functions in this library deal with several ways of stopping programs, or are designed to allow an interaction between a process and the debugger.

A.3 The Paragon

In this section we provide an introduction to the Purdue Paragon. After a brief description of the Paragon hardware, we focus on the way parallel applications are programmed and run. The information in this section has been obtained mainly from [PUCC95, Inte93].

A.3.1 Paragon XP/S 10 configuration and hardware description

The Purdue Paragon XP/S Model 10 (galaxy.cc.purdue.edu) is an Intel parallel supercomputer with 140 compute nodes, 1 boot and service node, 4 service nodes, 13 I/O nodes, 3 HiPPI/network nodes, and 1 Ethernet node. Compute nodes, service nodes, two of the three HiPPI nodes, and the boot node have 32 MB of DRAM each. The 13 I/O nodes, one HiPPI node, and the Ethernet node have 16 MB each. Five additional 128 MB compute nodes are also installed, but by the time of this writing they were not available for general use.

The Paragon is organized as a rectangular mesh of 12x16 node slots, not all of them used. Figure A.6 shows how the service and I/O nodes use the peripheral columns, plus some slots of the first two rows. The 140 compute nodes are all contiguous, structured as a 10x14 rectangle surrounded (except for the bottom) by the other classes of nodes or by empty slots.

```
    +-------------------------+
  0 | - - - - - - - - - - - B |
 12 | H - - - H - - - - - - E |
 24 | I * * * * * * * * * * S |
 36 | - * * * * * * * * * * S |
 48 | I * * * * * * * * * * I |    B: Boot and service
 60 | - * * * * * * * * * * S |    H: HiPPI
 72 | I * * * * * * * * * * I |    E: Ethernet
 84 | - * * * * * * * * * * S |    I: SCSI I/O
 96 | I * * * * * * * * * * I |    S: Service
108 | C * * * * * * * * * * C |    *: Compute
120 | I * * * * * * * * * * I |    C: 128 MB node
132 | C * * * * * * * * * * - |    -: Empty slot
144 | I * * * * * * * * * * I |
156 | C * * * * * * * * * * H |
168 | I * * * * * * * * * * - |
180 | C * * * * * * * * * * I |
    +-------------------------+
```

Figure A.6. Configuration of the Purdue Paragon.

Each Paragon node is powered by an Intel i860 XP microprocessor operating at 50 MHz, providing a theoretical peak floating-point double-precision performance of 75 Mflops per node. Single precision performance is 100 Mflops per node. This i860 XP microprocessor is called the application processor. Each i860 XP includes a 16 KB instruction cache and a 16 KB data cache. Peak memory-to-processor bandwidth is 400 MB/second; peak cache-to-processor bandwidth is 1.2 GB/second.

There is no shared memory in the Paragon XP/S. Each node's memory is local and data is shared between nodes by passing messages. At each compute node, 7 MB are required for the operating system, OSF/1, and 1 MB (the default value) is reserved for inter-node message buffers, leaving 24 MB for the application program. If desired, the message buffer size can be changed by the user.

As shown in the previous Figure, the nodes of the Paragon are connected in a two-dimensional rectangular mesh with each node connected only to its nearest neighbors on the mesh. The transmission of messages is carried out by an independent routing system of custom-designed Mesh Routing Chips (MRCs), one for each node. These chips route messages between any two nodes on the mesh at up to 200 MB/second, full duplex. Completing the interconnect is the Network Interface Controller (NIC), a custom VLSI chip which provides a full-bandwidth, pipelined interface between each node's MRC and memory. Each NIC permits simultaneous in-bound and out-bound communication at 175 MB/second.

Each node contains a second i860 XP, called the message co-processor, which is dedicated to handling message passing operations: it is not available for user applications. This processor handles message protocol processing for the application program, freeing the application processor to continue with numeric computation while messages are transmitted and received. The message co-processor is also used to implement efficient global operations such as synchronization, broadcasting, and global reduction calculations (e.g., global sum).

Paragon disk storage is provided by 14 RAID (Redundant Array of Inexpensive Disks) units connected to 13 I/O nodes and the boot and service node. Each unit consists of five disks and is connected to a node via an 8-bit wide SCSI-1 interface capable of transferring 5 MB/second. Each RAID unit has a formatted storage capacity of 4.8 GB.

The HiPPI and Ethernet nodes provide connectivity with other machines of the Purdue University Computing Center.

A.3.2 Distributed OSF/1 operating system

The Paragon operating system is a distributed version of the Open Software Foundation's OSF/1. It complies with the POSIX, AT&T System V.3, Berkeley 4.3bsd, and X/Open's XPG3 standards. NX/2 libraries provide compatibility with Intel's iPSC family of parallel processors. The operating system is fully and transparently distributed across the system's nodes. A MACH 3.0-based microkernel resides in each node.

A.3.3 Partitions

The nodes of the Paragon are organized in groups, called partitions. The naming system for partitions parallels the standard Unix file tree naming system. A period ('.') is used as the separator of partition name components. Just as '/' is the root of a Unix file system, '.' is the root of the partition tree.

A.3.3.1 The Service partition

The service partition of nodes is the partition in which the user's shell, OSF/1 commands, and non-parallel programs run. The name of the service partition is ".service".

When a user logs into the Paragon, a shell is started in a node in the service partition; when he or she executes a command, the command runs in a node in the service partition. Note that the command does not necessarily run in the same node as the shell; the system starts each new process in the least busy service node.

A.3.3.2 The Compute partition

The compute partition is the partition in which parallel applications run. The name of the compute partition is ".compute". When a user executes a parallel application, one process, called the controlling or proxy process, runs in the service partition; the other processes of the application run in sub-partitions of the compute partition.

The compute partition in the Purdue Paragon is logically sub-divided into two sub-partitions, one for interactive, development work and the other for batch production runs (submitted via NQS).

A.3.4 Program development

In this section the available tools (and machines) for programming Paragon applications are presented.

A.3.4.1 The cross-development facility

While it is possible to do all phases of program development in the Paragon, the Computing Center strongly discourages this style of use. It is far from the most efficient way to use the Paragon, because it adds unnecessary overhead to the Paragon's nodes and distracts them from their primary computational duties. Instead, the Paragon users are encouraged to take advantage of a Sun support host, named helios.cc.purdue.edu (a four CPU Sun S690MP, running Solaris 2.3), where the compilers, linkers, and libraries for Paragon OSF/1 are available. It is possible to read Paragon manual pages, edit files, compile and link Paragon programs in helios.cc—freeing the Paragon's nodes for computational tasks. The helios.cc file systems are mounted via NFS on the Paragon, allowing the Paragon to access programs developed in helios.cc.

A.3.4.2 Application development tools

C and Fortran cross-compilers and an i860 XP cross-assembler are available in helios.cc. The names and descriptions of these program development tools are listed in Table A.3.

Tool	Description
ar860	manages object code libraries
as860	assembles i860 source code
cpp860	preprocesses C programs
dump860	dumps object files
icc	compiles C programs
if77	compiles Fortran programs
ld860	links object files
nm860	displays symbol table (name list) information
size860	displays section sizes of object files
strip860	strips symbol information from object files

Table A.3. Development tools available in the Paragon and in the support host.

All these tools are also available at the Paragon, plus ipd, an interactive parallel debugger which also provides post-mortem debugging capabilities.

In this work we have used only the C programming language, so no further reference to Fortran will be made.

A.3.4.3 Creating a parallel program

The C header file <nx.h> must be included in all Paragon C programs. This file contains definitions and declarations needed by Paragon OSF/1 C system calls, including those of the NX parallel programming library. The main functions defined in this file will be described later in this chapter.

To compile a parallel application that runs in a compute sub-partition, the icc compiler is used, including the “-nx” switch in the compile command. For example, the command “icc -nx -o myprog myprog.c” uses the C cross-compiler to compile “myprog.c” into a parallel program called *myprog*. When *myprog* is put to run, it uses one or more nodes in a compute sub-partition. In contrast, the command “icc -o myprog myprog.c” creates a serial program that uses a single node in the service partition. Programs compiled with the “-nx” switch are classified as parallel applications and run in a compute sub-partition, even if they use only one node. Programs compiled without “-nx” are classified as single node applications and run in a single node in the service partition. Other compiler switches, such as “-I” (add directory to include file search path), and “-l” (search archived library), work as expected in OSF/1—i.e., like in other Unix implementations.

If the programmer wants a higher degree of control over the application, it is possible to create a controlling program for a parallel application, and then specify many aspects of the application (programs to run in each node, priority, etc.) from inside the controlling process, instead of from the command line. In this case, the programmer has to provide a controlling program, overriding the one automatically included when using the “-nx” switch. To do so, the “-lnx” switch has to be used instead.

A.3.4.4 Running parallel programs

During development time a large development sub-partition of the compute partition, named “.compute.OPEN” is available on a first-come first-served basis. The user must acquire a sub-partition of “.compute.OPEN” using a tool called psh. Once a partition has been set up, when the user wants to run a parallel application in, say, four Paragon nodes, the following command must be typed from a galaxy.cc window: “myprog -sz 4 [<input] [>output] [&]”.

The Paragon operating system uses the first four available nodes in the development sub-partition, moving across the columns of a 2-dimensional array. If “-sz” switch is not used, then by default the program runs in the number of nodes specified in the user’s NX_DFLT_SIZE environment variable. If the number of requested nodes is not available, an error message is displayed.

As mentioned before, when an application is successfully started, a controlling process is created in one of the service nodes, while the application itself runs in a set of compute nodes. The controlling process is able to make a series of start-up operations, which the user selects with command-line switches. Some interesting ones are:

-sz *size* or **-sz** *hXw*: the first possibility specifies the number of nodes needed for the application. The second one specifies a rectangular, contiguous node set in which to run the application (height × width).

-on *nodespec*: usually an application is loaded into all the nodes. With this option, it is possible to load it only into certain nodes. This is useful when an application consists of several different programs.

-pkt *packet_size*: sets the maximum packet size for message passing operations. Messages longer than the packet size are segmented when sent across the network.

-mbf *memory_buffer*: sets the total memory (in bytes) allocated for message buffers in each process.

A.3.4.5 Process management commands

OSF/1 contains the usual process management commands. Two particularly interesting ones are ps and kill, which work like in any Unix system. When a user invokes ps, a list of running processes is displayed. Controlling processes of running parallel applications are included in the list, but the application processes themselves are not. The kill command can be used to terminate an application controlling process, terminating at the same time all the processes belonging to the application.

The command pspart displays a list of the applications running in the compute partition, giving, among other information, the number of nodes used by each application. This command can be used to see the status of a user's application, or simply to check what is going in the Paragon. Another useful command is showpart, which produces a graphic representation of a partition, indicating (if the appropriate option is selected) the status (free, busy) of each node.

A.3.4.6 Production jobs in the Paragon. NQS

Some of the Purdue Paragon nodes are reserved for batch jobs, which are managed by the Intel NQS queuing system. Jobs for running under NQS can be submitted at any time, using the command qsub. The user must specify the queue (-q queue), a maximum wall clock running time (optional, -lT HH:MM:SS) and the program to submit, typically a shell script. Other available NQS commands are qstat, which displays the status of the NQS queues, and qdel, which deletes jobs from NQS queues.

A.3.5 The NX library <nx.h>

The NX library offers a wide set of functions to program parallel applications in the Paragon. Functions can be divided into the following groups: process characteristics, sending and receiving messages, global operations, control of application execution, partition management, floating-point control, Touchstone Delta compatibility, selection of I/O modes, file manipulation, extended arithmetic, Pthreads, mutexes, condition variables and others.

As this library contains many functions (about 200), we will concentrate on those somehow related to this work.

A.3.5.1 Process characteristics

Each Paragon process can be identified by a node number and a process type (ptype). These two numbers are needed for the message manipulation functions. Functions mynode() and myptype() return the node number and the ptype of the calling process respectively.

A controlling process does not have a known ptype, but it can be set up using setptype(). After that, messages can be sent to the controller, whose node number can be obtained with myhost().

An additional function of this group, numnodes(), returns the number of processors allocated to the application, not including the processor where the controlling process is running.

```
long mynode (void);
long numnodes (void);
long myptype (void);
long setptype (long ptype);
long myhost (void);
```

A.3.5.2 Synchronous send and receive

The two main functions for synchronously sending and receiving messages are csend() and crecv(). In this context, synchronous means that the process using one of these functions waits until the operation is finished. Note that the Intel definition of *synchronous* (*asynchronous*) corresponds to what we called *blocking* (*nonblocking*) in Chapter 3. After successful completion, the buffer containing the outgoing message can be safely reused (in the case of csend()), or a new incoming message is stored in the buffer (in the case of crecv()).

There is a clear difference in semantics between the transputer and the Paragon message passing functions: in the transputer a send and a receive are simultaneously needed to realize a communication. In the Paragon, crecv() effectively blocks a process until information is available, but csend() only blocks the calling process for a short period, while the message is sent to its destination or, most commonly, stored in a system buffer. Therefore, the end of a csend() does not mean that the message has already been received, it only means that the message buffer can be reused.

Another difference is that the communication is not done through channels, but addressing the destination process giving a <node, ptype> pair. Additionally, a message type has to be given when sending a message. This feature is commonly used to distinguish among different classes of information managed by the processes. A receiver can specify which message type is willing to accept, or just accept any type and, if desired, obtain the type (and other information) of a received message using the info…() functions, which will be described later.

While communication through channels only offers pair-wise information interchange, with this addressing scheme it is possible to realize multicast operations, using the gsendx() function.

```
void csend (long type, char *buf, long count, long node, long
ptype);
void crecv (long typesel, char *buf, long count);
```

```
void gsendx (long type, char *buf, long count, long nodes[], long
nodecount);
```

A.3.5.3 Asynchronous send and receive

Isend() and irecv() are asynchronous (meaning nonblocking) versions of csend() a crecv(). These functions request a send/receive operation and return control to the calling process immediately, before the operation is completed. A message identification is returned, which allows the process to ask about the status of the request.

Using a previously obtained message identification, msgdone() allows to determine whether the send or receive operations have completed—and then the allocated message buffers can be safely used. Msgwait() blocks the process until completion of the operation identified by the message identification given as its parameter. The effect is the same as performing a synchronous call, but some operations might have been done between the moment the operation was requested and the moment the process blocks for termination, allowing the user to overlap communication with computation.

```
long isend (long type, char *buf, long count, long node, long
ptype);
long irecv (long typesel, char *buf, long count);
long msgdone (long mid);
long msgwait (long mid);
```

A.3.5.4 Global operations

A set of g...() functions in the NX library perform operations that use data from every node in the application. Each node contributes to the operation with a piece of data, the operation is performed on the whole collection of data, and then a copy of the result of the operation is broadcasted to all the nodes.

These operations are synchronizing: if any node in an application makes one of these calls, it blocks until every other node in the application has made the same call. In the simplest case, gsync(), this synchronization is the only operation performed by the call. One process at each node in the application must make the call, and all the processes that make the call must have the same process type.

Global operations are implemented using dynamic algorithm selection for maximum performance. The system considers several ways of exchanging the needed information between the nodes, and selects the one that minimizes the time required to perform the global operation given the size and shape of the application.

Each global operation name begins with "g" and ends with the name of the operation. Some operations have several versions, which operate on different data types; for these calls, the data type is indicated by the second letter of the call's name ("l" for logical, "i" for integer, "s" for single-precision floating point, or "d" for double-precision floating point). For example, gdsum() performs a double-precision global sum and gilow() computes an integer global minimum. Most of the functions have three parameters: the data used for the computation, the size (number of components: the data can be a vector) and an auxiliary storage area. Gsync() does not need any parameter.

A.3.5.5 Other functions for managing messages

Cprobe() blocks the calling process until a message matching the specified message type arrives. After regaining control, the process can receive the message without blocking. Iprobe() simply checks if a message is available, that is, it is stored at the node, but it has not been read yet.

Immediately after a message has been received, or while it is pending awaiting to be read, the following functions can be used to obtain information about the message: infocount() to obtain its size in bytes; infonode() to obtain the node number of the sender; infoptype() to obtain the ptype of the sender; and infotype() to obtain the message type.

```
void cprobe (long typesel);
long iprobe (long typesel);
long infocount (void);
long infonode (void);
long infoptype (void);
long infotype (void);
```

A.3.5.6 Controlling application execution

As explained before, an application consisting of several processes, identical or different, can be loaded and run controlling all its characteristics from the command line. Additionally, it is also possible to write a controlling program that sets the application characteristics inside the program. To do so, the program has to be compiled with the -lnx option of icc, and needs to include a call to either nx_initve() or nx_initve_rect(). These two functions allocate a group of processors to run an application. The main difference between them is that nx_initve_rect() requests the allocation of a rectangular network of contiguous processors, while nx_initve() only asks for a given number of processors. Both fail if the request cannot be satisfied—that is, not enough nodes are free, or it is impossible to allocate the desired rectangle. The last two arguments of these functions allow the processing of the set-up arguments, which can be given when invoking the application from the shell (-sz, -on, -pkt or -mbf). The appropriate operations are done, and the arguments are removed from argv.

Once a group of nodes has been allocated, programs can be loaded. Nx_fork() makes copies of the controlling process in the specified nodes, assigning a specified ptype to all the newly created processes. Alternatively, nx_loadve() loads a program in the specified nodes, again assigning them a ptype. After the controller has put several processes to run, it can choose to block until all them finish, using nx_waitall().

Several processes can be loaded in the same processor, sharing it. The process scheduling policy is round robin, with a 10 milliseconds quantum. A process does not abandon the CPU when it is blocked awaiting a message operation to conclude: it simply stays idle until either completion of the operation or end of quantum. For this reason, allocating several processes to one processor with the purpose of having one using the CPU while others are blocked in input operations does not work in the Paragon. This behavior clearly differs from the way processes and communication are managed in the transputer.

```
long nx_initve (char *partition, long size, char *account, long
*argc, char *argv[]);
long nx_initve_rect (char *partition, long anchor, long rows, long
cols, char *account, long *argc, char *argv[]);
long nx_fork (long node_list[], long numnodes, long ptype, long
pid_list[]);
long nx_loadve (long node_list[], long numnodes, long ptype, long
pid_list[], char *pathname, char *argv[], char *envp[]);
long nx_waitall (void);
```

A.4 MPI in a network of workstations

MPI stands for Message Passing Interface. The goal of MPI, simply stated, is to develop a widely used standard for writing message passing programs. The interface should establish a practical, portable, efficient, and flexible standard for message passing [MPI94]. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. The purpose of MPI is to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

In designing MPI, the MPI Forum sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI has been strongly influenced by work at the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, and PARMACS. Other important contributions have come from Zipcode, Chimp, PVM, Chameleon, and PICL.

The main advantages of establishing a message passing standard are portability and ease-of-use. In a distributed memory communication environment, in which the higher level routines and/or abstractions are built upon lower level message passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines which they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The efforts of the MPI Forum have not stopped yet. MPI suffers from some serious limitations, which can make researchers reluctant to use it. Some of those limitations have been identified, and an effort to develop MPI2 is now in progress. The following items are considered as possible areas of expansion:

- I/O
- Active messages
- Process startup
- Dynamic process control
- Remote store/access
- Fortran 90 and C++ language bindings

- Graphics
- Real-time support
- Other enhancements

Some of the available implementations of MPI already include some of these enhancements, but clearly stating that they are not part of the current standard.

Several implementations of MPI have been made freely available. One of those is MPICH, developed by Argonne National Laboratory and Mississippi State University [Brid95]. MPICH is supported on a variety of parallel computers and workstation networks. Parallel computers that are supported include: IBM SP1, SP2 (using various communication options), TMC CM-5, Intel Paragon, IPSC860, Touchstone Delta, Ncube2, Meiko CS-2, Kendall Square KSR-1 and KSR-2, SGI and Sun Multiprocessors. Supported workstations include: Sun4 family running SunOS or Solaris, Hewlett-Packard, DEC 3000 and Alpha, IBM RS/6000 family, SGI, Intel 386- or 486-based PC clones running the LINUX or FreeBSD OS.

Other available implementations are:

- Edinburgh Parallel Computing Centre CHIMP implementation.
- Mississippi State University UNIFY implementation.
- Ohio Supercomputer Center LAM implementation.
- University of Nebraska at Omaha WinMPI implementation.

A.4.1 MPI programs

The current MPI specification includes bindings for the Fortran 77 and C programming languages. MPI2 will probably include bindings for Fortran 90 and C++. We will only discuss the C binding.

All MPI programs must include the header file “mpi.h”, where the definitions, macros and function prototypes necessary for compiling the programs can be found. Before using any other MPI function, a program must invoke MPI_Init(), to do all the necessary set-up operations. After finishing (but before exiting) the function MPI_Finalize() must be called for cleaning up.

```
int MPI_Init (int argc, char ***argv);
int MPI_Finalize ();
```

After the initialization, functions are available for point to point communication, collective communication and several environment management functions. In the following sections we give a description of the functions which are relevant for our work.

A.4.2 Point to point communication

MPI offers two communication models, *blocking* and *nonblocking*, and four communication modes: *basic*, *synchronous*, *buffered* and *ready*. In the blocking communication model, read and write operations are done in one step. Returning from a read means that a message has been received; returning from a write means that the message has been sent, but not necessarily that it has been received. In the nonblocking communication model operations are done in two steps: first the read or receive is *posted*, meaning that the corresponding function returns without completing the operation. A separate call must be done to actually complete the operation. With suitable hardware, this nonblocking model allows an overlapping of communication with computation, and may also avoid system buffering. Note that the NX library of the Paragon also offers these communication models, although the names used by Intel are different: synchronous for blocking, asynchronous for nonblocking.

For each communication model, four different send functions, one per communication mode, and one receive function are provided. The four blocking send operations are:

Basic send: MPI_Send() completes when the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. This semantics is close to that of the Paragon.

Buffered send: MPI_Bsend() completes immediately after storing the message in a local buffer, managed by the user. Its completion never depends on the occurrence of a matching receive.

Synchronous send: MPI_Ssend() only completes when a matching receive has been posted and the message interchange has been completed. This semantics is close to that of the Transputer.

Ready send: MPI_Rsend() can be started only if a matching receive has been already posted. Otherwise the operation is erroneous and its outcome is undefined.

All four blocking send operations take the same arguments. The prototype of MPI_Send() is as follows:

```
int MPI_Send (void *buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm);
```

The blocking receive operation can match any of the send modes, and returns only after the receive buffer contains the newly received message.

```
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int
src, int tag, MPI_Comm comm)
```

For the nonblocking model of communication, the functions MPI_Isend(), MPI_Ibsend(), MPI_Issend(), MPI_Irsend() and MPI_Irecv() are available to start an operation. Several forms of MPI_Wait() allow a process to block until a previously started operation (or set of operations) has been completed. Alternatively, a nonblocking test of completion can be done using MPI_Test(). A non-committed posted operation can be canceled with MPI_Cancel().

It is also possible to check if messages are pending to be received. MPI_Probe() is blocking, i.e., blocks the caller if no message is ready, until one is received. MPI_Iprobe() is just a nonblocking test. In either case, the message is not actually received until a receive operation is done.

A.4.3 Collective communication

Collective communication is defined as communication that involves a group of processes. All collective operations are global and blocking, that is, in order to perform an operation all the members of a group must call it, and control returns when it has been completed. In many cases a *root* process is mentioned. Any process in a group can be the root of an operation, but it is necessary that all the members agree in defining which process is the root. The collective communication functions provided by MPI are:

- Barrier synchronization across all group members: MPI_Barrier(). The call returns only after all group members have entered the call.

- Broadcast from one member to all members of a group. MPI_Bcast(). The process with rank root sends the information. All the processes in the group receive a copy (root included).
- Gather data from all group members to one member. MPI_Gather(). The root process receive a collection of messages (one per member of the group) and stores them in rank order.
- Scatter data from one member to all members of a group. MPI_Scatter(). Inverse of gather: the root process distributes a vector of values among all the members of the group (including itself), in rank order.
- A variation on Gather where all members of the group receive the result. MPI_Allgather(). All the members of the group receive a copy of the vector that results of the gather operation.
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all). MPI_Alltoall(). An extension of MPI_Allgather() to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j, and is placed in the ith block of the reception vector.
- Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members (MPI_Allreduce()) and a variation where the result is returned to only one member (MPI_Reduce()). The elements that are reduced may be vectors.
- A combined reduction and scatter operation. MPI_Reduce_scatter(). First a reduction is done, and then the result (a vector) is scattered along the processes in the group.
- Scan across all members of a group (also called prefix). MPI_Scan(). The operation returns, in the receive buffer of process i, the reduction of the values in the send buffer of processes $0..i$ (inclusive).

A.4.4 Communicators

The objective of communicators is to facilitate the construction of libraries easy to integrate in user programs. For example, if a library for matrix operations is done for a machine such as the Paragon with NX, that library would use a series of tags (message types, using Intel's notation) to label the messages. A programmer that wants to use the library must avoid the use of the same tags, because conflicts might appear. Obviously, this is not a desirable situation, and the introduction of the communicator concept helps to avoid it. Another situation where communicators help is the realization of global operations: those operations can be restricted to a subset of the processes, simply by defining a communicator that only include the appropriate processes.

An intra-communicator can be defined as a collection of processes that can send messages to each other and engage in collective operations. A particular intra-communicator, MPI_COMM_WORLD, includes all the processes in an application. Inter-communicators are used for sending messages between processes in different intra-communicators. A minimal intra-communicator is composed of a group and a context. A group is an ordered set of processes. Each process in a group is associated with an integer rank. Ranks are contiguous and start from zero. A context is a property of communicators that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-to-point operations, because they form part of different contexts.

MPI includes a series of operations for the management of communicators and groups. Contexts are not explicitly managed: they appear only as part of the realization of communicators. Two commonly used functions related to groups are MPI_Group_size(), which informs about the number of processes in a group, and MPI_Group_rank(), which tells a process about its rank in a group (provided that the process belongs to it). The discussion of the remaining functions falls out of the scope of this dissertation.

A.4.5 Data types

One of the parameters of send and receive operations is the type of the data units being transmitted (other being the number of units transmitted). Both sender and receiver must agree in the specified data type. A series of constants are defined to choose the appropriate data type. Table A.4 lists the data types available for the C binding (a different one is defined for the Fortran binding).

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short it
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Table A.4. MPI data types (C binding).

The last two types do not correspond to any C data type. A value of type MPI_BYTE consists of an octet, which is uninterpreted and different from a character. MPI_PACKED is used to interchange blocks of non-contiguous data items which have been packed into a contiguous buffer.

One of the goals of MPI is to support parallel computation across heterogeneous environments. This might require data conversions, to adjust between the different data representation formats of the communicating machines. In this context, the data type information is essential to perform the appropriate data conversions. The MPI_BYTE data type can be used to transfer data without any conversion.

MPI also offers mechanisms to send/receive data structures more complex than sequences of simple objects. Derived data types can be defined, and then objects of those types can be transmitted, performing the necessary data conversions. Again, explaining those mechanisms falls out of the scope of this work.

A.4.6 The MPICH implementation of MPI

MPICH 1.0.8 has been installed in a network of 5 Sun SPARCstation 5 workstations at the Parallel Processing Laboratory of the School of Electrical and Computer Engineering, Purdue University.

This implementation includes:

- A configuration program which sets up MPICH for running in a particular computing environment. In this case, a network of Sun workstations running

Solaris. Once the environment has been configured, a Makefile is provided to build the rest of the elements of this list.

- A library with all the MPI functions. In the case of our particular environment, this library is based on p4, which is also included in the MPICH installation, and uses TCP/IP as the communication means. Other alternatives are possible; for example, the MPI library can be built over Intel's NX library if the target machine is a Paragon.
- A library with extensions to MPI, not yet standardized. All the functions in this library carry the prefix MPE. The components of this library are:
    - A set of routines for creating log files, which can be used for program profiling; a tool called wrappergen and a collection of makefiles are available to help profiling programs.
    - A shared display parallel X graphics library. This library can be used to create real-time program animations.
    - Routines for sequentializing a section of code: processes execute that section in rank order, instead of doing it in parallel.
    - Debugger setup routines.
- Upshot, a program to analyze log files created with the MPE profiling functions.
- Mpirun, a tool for starting parallel programs in a way independent of the target machine. A series of switches determine some of the characteristics of the application to run, such as the number of processes that will be put to run (mpirun -np *number_of_processes* application).
- A collection of manuals and other documentation.

When launching an application in this particular environment, mpirun checks a configuration file called machines.solaris, where a list of available workstations can be found. It is assumed that all the machines of the list have the same view of the file system, which usually means that NFS is in use. Mpirun tries to launch, using the Unix tool rsh, as many processes as required, one per entry in the machines.solaris file. This means that it is possible to launch more processes than machines are available, simply by repeating machine names in the machines file.

The previous discussion about using mpirun works well for SPMD applications, and for homogeneous systems where all the workstations can find executable files in the same locations. However, it is possible to overcome these limitations using the -p4pg switch of mpirun. In this case, an application configuration file must be given as a parameter, being the format of its lines as follows:

hostname number_of_processes pahtname_of_executable login

Hostname is the name of a workstation to use. *Number_of_processes* indicates how many processes must be launched in that machine; this option is only valid when shared memory multicomputers are being used, otherwise the strategy of repeating machine names must be used. The *pahtname_of_executable* indicates where the executable of the process to run in the corresponding machine is. The last parameter, a *login* name, is optional, and can be used to comply with the security procedures of rsh.

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.

[Abra93] M. Abrans. “Parallel discrete event simulation: fact or fiction?” *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 231—233.

[ACGW95] M.F. Alitt, Y. Chen, R.J. Gurski and C.L. Williamson. “Traffic modeling in the ATM-TN Telesim project: design, implementation and performance evaluation”. *Proceedings Summer Computer Simulation Conference SCSC’95*. Ottawa, Canada, July 1995, 847—851.

[ACLS94] D. Agrawal, M. Choy, H.V. Leong and A.K. Singh. “Maya: a simulation platform for distributed shared memories”. *Proc. 8th Workshop on Parallel and Distributed Simulation PADS’94*, Edinburgh UK, July 1994, 151—155.

[AD91] H.H. Ammar and S. Deng. “Time Warp simulation of stochastic Petri nets”. *Proc. 4th Int. Workshop on Petri Nets and Performance Models PNPM´91*, Melbourne, Australia, Dec. 1991, 186—195.

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

[Bagr93] R. Bagrodia. “A survival guide for parallel simulation” *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 234—235.

[Bank84] J. Banks and J.S. Carson. *Discrete-Event System Simulation*. Prentice Hall, Inc. 1984. ISBN 0-13-215582-6.

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

[BHBA91] R. Beivide, E. Herrada, J.L. Balcázar and A. Arruabarrena. "Optimal distance networks of low degree for parallel computers". *IEEE Transactions on Computers*, Vol. 40, No. 10, Oct. 1991, 1109—1124.

[BHBL87] R. Beivide, E. Herrada, J.L. Balcázar and J. Labarta. "Optimized mesh-connected networks for SIMD and MIMD architectures". *Proc. 14th Int. Sym. on Comp. Architecture ISCA'87*, 163—170.

[Blum95] M.A. Blumrich et al. "Virtual-memory-mapped network interfaces". *IEEE Micro*, Vol. 15, No. 1, February 1995.

[Brid95] P. Bridges et al. *User's guide to mpich, a portable implementation of MPI*. Argonne National Laboratory, 1995

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

[BW95] D.C. Burger and D.A. Wood. "Accuracy vs. performance in parallel simulation of interconnection networks". *Proc. 9th Int. Parallel Processing Symp.*, Santa Barbara, California, April 1995, 22—31.

[CF93] G. Chiola and A. Ferscha. "Distributed simulation of Petri nets". *IEEE Parallel & Distributed Technology*. Vol. 1, No. 3, Aug. 93, 33—50.

[CGFO94] A.Costa, A. De Gloria, P. Faraboschi and M. Olivieri. "An evaluation system for distributed-time VHDL simulation". *Proc. 8th Workshop on Parallel and Distributed Simulation PADS'94*, Edinburgh UK, July 1994, 147—150.

[Clea94] J. Cleary et al. "Cost of state saving & rollback". *Proc. 8th Workshop on Parallel and Distributed Simulation PADS'94*, Edinburgh UK, July 1994, 94—100.

[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.

[CM81] K.M. Chandy and J. Misra. "Asynchronous distributed simulation via a sequence of parallel computations". *Comm. of the ACM*, Vol. 24, No. 11, April 1981, 198—205.

[CMH83] K.M. Chandy, J. Misra and L.M. Haas "Distributed deadlock detection". *ACM Transactions on Computer Systems*, Vol. 1, No. 2, May 1983, 144—156.

[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.

[CH94] R.G. Chamberlain and C.D. Henderson. "Evaluating the use of pre-simulation in VLSI circuit partitioning". *Proc. 8th Workshop on Parallel and Distributed Simulation PADS'94*, Edinburgh UK, July 1994, 139—146.

[Chaw94] P. Chawla. *Assignment strategies for parallel discrete event simulation of digital systems*. Ph.D. dissertation. Department of Electrical and Computer Engineering. University of Cincinnati, 1994.

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

[DHN94] P.M. Dickens, P. Heidelberger and D.M. Nicol. "A distributed memory LAPSE: parallel simulation of message-passing programs". *Proc. 8th Workshop on Parallel and Distributed Simulation PADS'94*, Edinburgh UK, July 1994, 32—38.

[DMS93] B. Dado, P. Menhart and J. Safarik. "Distributed simulation: a simulation system for discrete event systems". *Decentralized and Distributed Systems (A-39)*. M. Cosnard and R. Puigjaner (Eds.). Elsevier Science, 1993. 343—354.

[EGLW93] S.G. Eick, A.G. Greenberg, B.D. Lubachevsky and A. Weiss. "Synchronous relaxation for parallel simulation with applications to circuit-switched networks". *ACM Trans. on Modeling and Comp. Simulation*. Vol. 3, No. 4, Oct. 1993, 287—314.

[FK91] R.E. Felderman and L. Kleinrock. "Bounds and approximations for self-initiating distributed simulation without lookahead". *ACM Trans. on Modeling and Comp. Simulation*, Vol. 1, No. 4, 1991, 386—406.

[Fost95] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley, 1995. ISBN 0-201-57594-9.

[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.

[Fuji88] R.M. Fujimoto, "Lookahead in parallel discrete event simulation". *Proc. Int. Conf. Parallel Processing*, Aug. 1988, 34—41.

[Fuji89a] R.M. Fujimoto. "Performance measurements of distributed simulation strategies". *Trans. of The Society for Comp. Simulation*, Vol. 6, No. 2, 1989, 89—132.

[Fuji89b] R.M. Fujimoto, “Time warp on a shared memory multiprocessor”. *Proc. Int. Conf. Parallel Processing*, Vol. III, Aug. 1989, 242—249.

[Fuji90a] R.M. Fujimoto. “Optimistic approaches to parallel discrete event simulation”. *Trans. of The Society for Comp. Simulation*, Vol. 7, No. 2, 1990, 153—191.

[Fuji90b] R.M. Fujimoto. “Parallel discrete event simulation”. *Comm. of the ACM*, Vol. 33, No. 10, Oct. 1990, 30—53.

[Fuji93a] R.M. Fujimoto. “Parallel discrete event simulation: will the field survive?” *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 213—230.

[Fuji93b] R.M. Fujimoto. “Future directions in parallel simulation research” *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 245—248.

[FW94] B. Falsafi and D.A. Wood. “Cost/performance of a parallel computer simulator”. *Proc. 8th Workshop on Parallel and Distributed Simulation PADS’94*, Edinburgh UK, July 1994, 173—182.

[GHTY90] P.K. Goli, P. Heidelberger, D.F. Towsley and Q. Yu. “Processor assignment and synchronization in parallel simulation of multistage interconnection networks”. *Distributed Simulation 1990*, 181—187.

[Gome95] F. Gomes et al. “SimKit: a high performance logical process simulation class library in C++”. Submitted to the 1995 Winter Simulation Conference.

[GOR95] P. Gburzynski, T. Ono-Tesfaye and S. Ramaswamy. “Modeling ATM networks in a parallel simulation environment: a case study”. *Proceedings Summer Computer Simulation Conference SCSC’95*. Ottawa, Canada, July 1995, 869—874.

[GT93] D.W. Glazer and C. Tropper. “On process migration and load balancing in Time Warp”. *IEEE Trans. on Parallel and Dist. Systems*, Vol. 4, No. 3, March 1993, 318—327.

[IAB91] C.Izu, A. Arruabarrena and R. Beivide. “Communication capabilities of the T800 Transputer in a reduced size network”. *Proc. of the ISMM Int. Workshop on Parallel Computing*, Trani, Italy, Sept. 1991, 28—31.

[IAB92] C. Izu, A. Arruabarrena and R. Beivide. “Analysis and evaluation of message management functions for bi-dimensional transputer networks”. *Microprocessors and Microsystems*, Vol. 16, No. 6, 1992, 301—309.

[IBJA93] C.Izu, R. Beivide, C. Jesshope, A. Arruabarrena, “Experimental evaluation of Mad-Postman bi-dimensional routing networks”. *Microprocessing and Microprogramming* No. 38 (1993). 33—41.

[Inmo89a] *OCCAM 2 Reference manual*. Inmos Ltd., Bristol, U.K., 1989.

[Inmo89b] *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.

[ITH89] Q. Yu, D. Towsley and P. Heidelberger. "Time-driven parallel simulation of multistage interconnection networks". *Distributed Simulation 1989*, 191—196.

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

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

[Jeff87] D.R. Jefferson et al. "Distributed simulation and the Time Warp operating system". *Proc. 11 Annual ACM Symp. on Operating System Principles*, Austin, Texas, Nov. 1987, 77—93.

[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.

[Lin93] Y-B. Lin. "Will parallel simulation research survive?" *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 236—238.

[LP91] Y-B. Lin and B.R. Preiss. "Optimal memory management for Time Warp parallel simulation". *ACM Trans. on Modeling and Comp. Simulation*, Vol. 1, No. 4, Oct. 1991, 283—307.

[LPLL93] Y-B. Lin, B.R. Preiss, W.M. Loucks and E.D. Lazowska. "Selecting the checkpoint interval in Time Warp simulation". *Proc. 7th Workshop on Parallel and Distributed Simulation PADS'93*, San Diego CA, May 1993, 3—10.

[MAB93] J. Miguel, A. Arruabarrena and R. Beivide. "Conservative parallel discrete event simulation in a transputer-based multicomputer". *Transputer Applications and Systems'93*. R. Grebe et al. (Eds.). IOS Press, 1993. 636—650.

[MAB94] J. Miguel, A. Arruabarrena and R. Beivide. "Conservative and optimistic distributed simulation in massively parallel computers: a comparative study". *Proc. of the Conference on Massively Parallel Computing Systems MPCS'94*, Ischia, Italy, May 1994. 528—532.

[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.

[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.

[MB95] J. Miguel and R. Beivide. “Simulación paralela de sistemas de sucesos discretos”. *Informática y Automática*. To be published.

[MG93] J. Miguel and M. Graña. “Towards the distributed implementation of discrete event simulation languages”. *Decentralized and Distributed Systems (A-39)*. M. Cosnard and R. Puigjaner (Eds.). Elsevier Science, 1993. 355—366.

[Migu91] J. Miguel, C. Izu, A. Arruabarrena, J. García-Abajo and R. Beivide, “Toroidal networks for multicomputer systems”. *Proc. ISMM Int. Workshop on Parallel Computing*, Trani, Italy, Sept. 1991, 112—117.

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

[ML93] N. Manjikian and W.M. Loucks. “High performance parallel logic simulation on a network of workstations”. *Proc. 7th Workshop on Parallel and Distributed Simulation PADS’93*, San Diego CA, May 1993, 76—84.

[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.

[MR94] P. Mussi and H. Rakotoarisoa. “PARSEVAL: a workbench for queuing networks parallel simulation”. INRIA (France) Rapport de recherche Nº 2234, April 1994.

[MRR90] B.C. Merrifield, S.B. Richardson and J.B.G. Roberts. “Quantitative studies of discrete event simulation modelling of road traffic”. *Proc. SCS Multiconference on Distributed Simulation*, Vol. 22, SCS Simulation Series, Jan. 1990, 188—193

[MW95] T.J. McBrayer and P.A. Wilsey. “Process combination to increase event granularity in parallel logic simulation”. *Proc. 9th Int. Parallel Processing Symp.*, Santa Barbara CA, April 1995, 572—578.

[NF94] D. M. Nicol and R.M. Fujimoto. “Parallel simulation today”. To appear in *Annals of Operations Research*.

[NGL94] D.M. Nicol, A.G. Greenberg and B.D. Lubachevsky. "Massively parallel algorithms for trace-driven cache simulations". *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 8, Aug. 1994, 849—859.

[Nico88] D.M. Nicol. "Parallel discrete-event simulation of FCFS stochastic queueing networks". *Proc. ACM SIGPLAN Symp. Parallel Programming Experience in Applications, Languages and Syst*., New Haven CT, July 1988, 124—137.

[Nico92] D.M. Nicol. "Conservative parallel simulation of priority class queueing networks". *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 3, May 1992, 294—303.

[Pala94] A.C. Palaniswamy. *Dynamic parameter adjustment to speedup Time Warp simulation*. Ph.D. dissertation. Department of Electrical and Computer Engineering. University of Cincinnati, 1994.

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

[PUCC95] *Getting started on the Paragon*. Purdue University Computing Center, Advanced Applications Group of the Research Computing Division, 1995, http://www-rcd.cc.purdue.edu/Paragon/.

[PW93] A.C. Palaniswamy and P.A. Wilsey. "An analytical comparison of periodic checkpointing and incremental state saving". *Proc. 7th Workshop on Parallel and Distributed Simulation PADS'93*, San Diego CA, May 1993, 127—134.

[Reyn93] P.F. Reynolds. "The silver bullet" *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 239—241.

[RMM88] D.A. Reed, A.D. Malony and B.D. McCredie. "Parallel discrete event simulation using shared memory". *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, April 1988, 541—553.

[RW89] R. Righter and J.C. Walrand. "Distributed simulation of discrete event systems". *Proceedings of the IEEE*, Vol. 77, No. 1, Jan. 1989, 99—113.

[Sang94] J. Sang. *Multithreading in distributed-memory systems and simulation: design, implementation and experiments*. Ph.D. dissertation, Department of Computer Sciences, Purdue University, 1994.

[SB93] C. Sporrer and H. Bauer. "Corolla partitioning for distributed logic simulation of VLSI circuits". *Proc. 7th Workshop on Parallel and Distributed Simulation PADS'93*, San Diego CA, May 1993, 85—92.

[SCR94a] J. Sang, K. Chung and V. Rego. "Efficient algorithms for simulating service disciplines". *Simulation Practice and Theory,* No. 1, 1994, 223—244.

[SCR94b] J. Sang, K. Chung and V. Rego. “A simulation testbed based on lighweight processes”. *Software Practice & Experience*, Vol. 24, No. 5, May 1994, 485—505.

[SG89] L. Soulé and A. Gupta. *Analysis of parallelism and deadlocks in distributed-time logic simulation*. Stanford Univ. Technical Report CSL-TR-89-378, May 1989.

[SG91] L. Soulé and A. Gupta. “An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation”. *ACM Trans. on Modeling and Comp. Simulation*, Vol. 1, No. 4, Oct. 1991, 308—347.

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

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

[Su90] W-K. Su, *Reactive-process programming and distributed discrete event-simulation*. Ph.D. dissertation. Computer Science Department, California Institute of Technology, 1990. Caltech-CS-TR-89-11.

[TB93] C. Tropper and A. Boukerche. “Parallel simulation of communicating finite state machines”. *Proc. 7th Workshop on Parallel and Distributed Simulation PADS’93*, San Diego CA, May 1993, 143—150.

[TV91] F. Tallieu and F. Verboven. “Using Time Warp for computer network simulation on transputers”. *Proc. 24th Annual Simulation Symposium*, New Orleans, Luisiana, April 1991, 112—117.

[UC93] B.W. Unger and J.G. Cleary. “Practical parallel discrete event simulation” *ORSA Journal on Computing*, Vol. 5, No. 3, 1993, 242—244.

[Vaki92] P. Vakili. “Massively parallel and distributed simulation of a class of discrete event systems: a different perspective”. *ACM Trans. on Modeling and Comp. Simulation*, Vol. 2, No. 3, July 1992, 214—238

[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.

[Wiel89] F. Wieland et al. “The performance of a distributed combat simulation with the Time Warp Operating System”. *Concurrency: Practice and Experience*, Vol. 1, No. 1, Sep. 1989, 35—50.

[WJ89] F. Wieland and D. Jefferson. “Case studies in serial and parallel simulation”. *Proc. Int. Conf. Parallel Processing*, Vol. III, Aug.1989, 255—258.

[WL89] D.B. Wagner and E.D. Lazowska. “Parallel simulation of queueing networks: limitations and potentials”. *ACM Performance Evaluation Review*, Vol. 17, No. 1, May 1989, 146—155.

[Zeig76] B.P. Zeigler. *Theory of Modelling and Simulation*. Wiley-Interscience, 1976. ISBN 0-471-98142-4.

[Zorp92] G. Zorpette. “The power of parallelism”. *IEEE Spectrum*, Sept. 1992, 28—33.