Durham E-Theses

The treatment of time in distributed simulation

Ryang, Nue Chille

How to cite:

Ryang, Nue Chille (1990) *The treatment of time in distributed simulation*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/6147/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

The Treatment Of Time In Distributed Simulation

Nue Chille Ryang

A thesis submitted for the degree of
Master of Science

School of Engineering and Applied Science
(Computer Science)
University of Durham

December 1990

25 JUN 1991

Abstract

Simulation is one of the most important tools to analyse, design, and operate complex processes and systems. Simulation allows us to make a 'trial and error' in order to understand a system and describe a problem. Therefore, it is of great interest to use simulation easily and practically. The advent of parallel processors and languages help simulation studies. A recent simulation trend is distributed simulation which may be called discrete-event simulation, because distributed simulation has a great potential for the speed-up. This thesis will survey discrete-event simulation and examine one particular algorithm. It will first survey simulation in general and secondly, distributed simulation. Distributed simulation has broadly two mechanisms: conservative and optimistic. The treatment of time in these mechanisms is different, we will look into both mechanisms. Finally, we will examine the conservative mechanism on a network of transputers using Occam. We will conclude with the result of the experiments and the perspective of distributed simulation.

Acknowledgement

I would like to thank Mr. A. J. Slade and Dr. J. Welford for their guidance and support in my course.

I would also like to thank Roy for his help with TDS, Mr. M. Pegman for spending time discussing the project with me, and Russell and Simon for their help with proof reading my thesis.

I would finally like to thank Yun Ji Oun, my uncle, for financing my study in Britain.

Contents

1 INTRODUCTION **1**

1.1 Motivation 1

1.2 Thesis Outline 3

2 SIMULATION **5**

2.1 The Concept of a System 5

2.2 What is Simulation? 6

2.3 Development of Simulation 8

2.4 Advantages and Disadvantages of Simulation 9

2.5 Classification 11

2.5.1 Discrete and Continuous Systems 11

2.5.2 Stochastic and Deterministic Systems 13

2.5.3 Open and Closed Systems 14

2.6 Simulation Techniques 14

2.6.1 Random-number Generators 14

2.6.2 Queuing Theory 16

2.7 Languages 19

2.7.1 Multipurpose Languages 19

2.7.2 GASP IV 21

2.7.3 SLAM (Simulation Language for Alternative Modeling) 22

2.7.4 SIMSCRIPT II.5 22

2.7.5 GPSS (General-Purpose Simulation System) 23

2.7.6 DYNAMO 23

3 DISTRIBUTED SIMULATION AND TIME TREATMENT 24

3.1 Distributed Simulation 24

3.2 Programming Support 26

3.3 Possible Approaches in Distributed Simulation 31

3.3.1 Parallelising Compilers 32

3.3.2 Distributed Experiments 32

3.3.3 Distributed Simulation Events 33

3.3.4 Language Function Distribution 34

3.3.5 Model Function Distribution 35

3.4 Distributed Simulation Models 36

3.4.1 Time-driven Simulation 36

3.4.2 Event-driven Simulation 38

3.5 Conservative Approach 40

3.5.1 Conservative Approach Algorithm 41

3.5.2 Waiting Rules For Logical Processors (lps) 42

3.5.3 System Deadlock 43

3.5.4 Deadlock Avoidance (Chandy and Misra Algorithm) 45

3.5.5 Deadlock Detection and Recovery 46

3.6 Optimistic Approach 47

3.7 Previous Performance Studies 51

4 STUDIES **53**

4.1 Motivation 53

4.2 The Simulation Algorithm 54

4.3 Experimental Environment 55

4.4 Experiment I 57

4.4.1 Topologies 57

4.4.2 Input and Output 59

4.4.3 Termination 59

4.5 Testing 60

4.6 Deadlock Avoidance 65

4.6.1 Deadlock Avoidance by Input Messages 65

4.6.2 Deadlock Avoidance by Buffers 67

4.7 Experiment II 69

4.7.1 Input and Output 69

4.7.2 Experiments 69

4.7.3 Results 72

4.8 Summary 74

5 CONCLUSIONS **75**

Bibliography **77**

Appendix Program Listings **83**

Chapter 1

INTRODUCTION

1.1 Motivation

Simulation is one of the most important and useful tools for analysing the design and operation of complex processes and systems. Simulation enables people either to understand the behaviour of a system or to evaluate the performance of a system. Simulation is suitable for many fields. Shannon (see [Shannon 75]) remarks that simulation receives its original impetus from the nuclear and aerospace programs. He indicates the broad field of present applications through books published on the use of simulation in business, economics, marketing, education, politics, social science, behavioural science, international relations, transportation, law enforcement, urban studies, and global systems, to name only a few.

Two separate classes of simulation exist: discrete and continuous. As the term suggests, in discrete simulation, a system instantaneously changes its state at discrete points in time, whereas in continuous simulation, changes occur smoothly and continuously over time. However, simulations are slow to develop and slow to run. A simulation model may become complicated, so it may be expensive in terms of manpower and computer power.

In order to be statistically significant, simulation has to generate a sufficient number of typical evolutions of the system.

Discrete-event simulation changes its state only at a countable number of points in time. In practice, discrete-event simulations can be characterised as being event-driven systems. The discrete-event simulation advances simulated time in irregular intervals defined by the time of occurrence of each simulated event. Typical sequential implementations maintain an 'event list' which is a list of expected future events ordered in increasing order of expected time occurrence. The event is removed from the event list and the simulation clock is advanced to the time of the event. Simulating an event may change the values of variables that describe the state of the system, and may cause events to be added to or deleted from the event list. Distributed simulation attempts to reduce the time needed to perform a simulation by spreading its execution over multiple processors. It is accomplished by exploiting the parallelism inherent in discrete-event simulation allowing the distributed processes to run asynchronously. The advent of inexpensive and increasingly powerful multi-computer systems could be used to create the parallelism.

Discrete-event simulation mechanisms fall broadly into two categories: conservative and optimistic. Conservative approaches were historically the first to appear. In conservative approaches, messages are transmitted strictly in chronological order. These approaches rely on some strategy to determine when it is 'safe' to process an event (in other words, they must determine when all events that could affect the event in question have been processed). One of these approaches uses *null* messages when a process is blocked (see [Chandy 81]). There are several other approaches. An alternative to the null message approach is to send *query* or *probe* when a process is blocked and needs an improved clock time (see [Peacock 79a], [Misra 86], [Misra 83], and [Bain 88]). In optimistic approaches, on the other hand, messages may, or may not, be transmitted chronologically. When a message with an earlier time stamp is transmitted, the process 'rolls back' to an earlier virtual time. Work has been done on evaluating the performance of various conservative approaches, see [Fujimoto 88], [Reed 85], [Reed 88a], and [Reed 88b]. The empirical evidence to support the optimistic approach is shown in [Lomow 88], [Mitra 84], and [Gilmer 88]. In distributed simulation, however, many approaches have been researched thoroughly, though more empirical work needs to be done. Since it may be difficult to define an approach for general distributed simulation, empirical work should be carried

out to determine which approach is appropriate to which circumstances.

In this thesis, we will examine deadlock avoidance using null messages. This will be evaluated using a transputer board and the Occam parallel programming language. Since distributed simulations are based on a process-interaction by passing messages, Occam is naturally suited to implementing this type of problem. It is reasonable to expect that process interaction models running in Occam on a transputer will perform well, because Occam is claimed to be an assembly language for the transputer. We will examine the performance of one transputer and a network of transputers. We will run the same program on both, and evaluate the effect of null messages, processing time, and the like. We will also observe the effect of the system structure, the process service time by various types of messages.

I hope that this thesis will help those computer scientists who survey approaches in distributed simulation.

1.2 Thesis Outline

The rest of this thesis is divided into four Chapters.

Chapter 2 surveys simulation in general. The terminology of simulation is introduced and a classification is presented. Some techniques and possible implementation languages are surveyed.

Chapter 3 focuses on distributed simulation. Five possible decomposition techniques are explained and examples given of each: parallelising compilers, distributed experiments, distributed simulation events, language function distribution, and model function distribution. Two major approaches, conservative and optimistic, are analysed.

Chapter 4 examines deadlock avoidance using null messages introduced in conservative approaches. A network of transputers and Occam are used. An attempt is made to determine the treatment of time, the effect of null messages, and the effect of the system structure, and the like.

Chapter 5 summarises this thesis and talks about further work in distributed simulation.

Appendix contains a list of a fork and merged network program on a network of transputers using the keyboard and the screen, and files, respectively.

Chapter 2

SIMULATION

2.1 The Concept of a System

Simulation mimics a real system to let people understand how the system works, how to solve the problem, how to operate the system, and the like. Therefore, central to simulation study is the idea of a system. To model a system, one must first understand what a system is. The term *system* is used in a broad range of contexts with various meanings. In simulation the term is normally used to designate a collection of objects with a well defined set of interactions between the objects (see [Adkins 77]). We could choose one of the definitions according to our purpose.

> A *system* is defined as an aggregation or assemblage of objects joined in some regular interaction or interdependence (see [Gordon 78]).

A system may take into consideration all external factors capable of causing a change in the system. These external factors form the *system environment.* A real-world object is called an *entity*, *attributes* are characteristics or properties of entities. The state of

a system is the minimal collection of information with which its future behaviour can be uniquely predicted in the absence of chance events. Since the inclusion of time in the consideration of a system implies that the state of a system changes, there must be an *activity*, either a process or event, which prompts this change. The system state may change in response to activities internal to the system or to activities external to the system. The term *endogenous* is used to describe activities occurring within the system and the term *exogenous* is used to describe activities in the environment that affect the system. Although it is convenient to distinguish between endogenous and exogenous activities, it is not always possible to do so. When one is defining a given system, it is not always apparent which factors are internal to the system and which are external. Furthermore, with a given system definition an exogenous activity may prompt a series of endogenous activities. The resulting system state may in turn trigger another exogenous activity. Thus in many cases very little distinction can be made between endogenous and exogenous activities.

2.2 What is Simulation?

Simulation is one of the most powerful techniques available for problem solving. It involves the construction of a replica or model of the problem, on which we experiment and test alternative courses of action. This gives us a greater insight into the problem and places us in a better position from which to seek a solution.

We conduct experiments in a systematic way until we either get a satisfactory answer or give up through lack of progress. The greater our understanding of the problem the quicker we are able to produce an answer. We start from the point of present understanding of the problem and proceed, according to ability and application, to search for the best possible solution in the time available. This means that simulation can be laborious and expensive, and does not necessarily produce an acceptable answer, much less the optimum answer.

Simulation forces us into observing and understanding the behaviour of the problem by identifying those factors which are important. This results in an appreciation of the

dynamics of the total system under study, and helps avoid bias towards solving special mathematical problems relating to one aspect of the system, which is a danger inherent in the analytical approach.

> *Simulation* is a 'trial and error' approach which allows us to describe a problem and gain understanding of the factors involved, by asking questions and observing the answers (see [Poole 77]).

Though the literature gives many definitions for simulation, this definition seems to encompass the more important aspect of this problem solving process (see [Graybeal 80]). Simulation is essentially an experimental problem-solving technique. Many simulation runs have to be made to understand the relationships involved in the system, so the use of simulation in a study must be planned as a series of experiments.

To simulate the system, the process of preparing a suitable model, *modeling*, is required. Modeling is the process of developing an internal representation and set of transformation rules which can be used to predict the behaviour and relationships between the set of entities composing the system. The internal representation requires identification of a sufficient set of *variables* to be used to describe *system state*; these variables are changed by the application of the transformational rules.

Compared with analytical solution of problems, the main drawback of simulation is that it gives specific solutions rather than general solutions. An analytical solution gives all the conditions that can cause events. Each execution of a simulation tells only whether a particular set of conditions did or did not cause events. To try to find all such conditions requires that the simulation be repeated under many different conditions. The step-by-step nature of the simulation technique means that the amount of computation increases very rapidly as the amount of detail increases. Coupled with the need to make runs to explore the range of conditions, the extra realism of simulation models can result in a very extensive amount of computing.

2.3 Development of Simulation

The development of simulation involves six major steps:

1. Preliminary analysis to determine if a simulation is worth developing.
2. Formulation of the problem.
3. Collection and analysis of pertinent information.
4. Model construction.
5. Computer programming.
6. Validation.

In most cases, the first two steps should be completed before anything else is done. Steps three through six are carried out in outlapping times, as dictated by the particular circumstances. A major factor in successful simulation development is the control of these overlapping activities to obtain a unified result.

In the first step, preliminary analysis, the analyst should think about (1) What is the system really like? and (2) How much do we already know about it? The next step requires rough estimates of the resources that will be needed to simulate the system and a description of the ways that the simulation will be used. This work gives the analyst the best possible guess at the cost and the benefits of the simulation study.

The essence of problem formulation is the detailed specification of the applications to be made of the simulation. A computer simulation must be designed to accommodate a set of specific applications. Since the simulation is usually intended to provide information for management, the best source for ideas of specific applications is management itself, at all levels. At this stage, the analyst must decide which problems the simulation will be able to help and which it will not. The analyst will also develop lists of the entities, attributes and activities to be included in the simulation. The choice of suitable measures of system effectiveness is very much a part of the formulation of the problem. Since measures often prove unsuitable or incomplete for a particular simulation development.

A model is validated by proving that the model is a correct representation of the real system. Validation should not be confused with verification. When a computer program is verified, for example, the program is checked to ensure that the logic does what it was intended to do. A verified computer program can in fact represent an invalid model. The program may do exactly what the programmer intended, but it may not represent the operation of the real system.

Validation of computer simulation is a difficult task.

It is improved by using models that are parametric in so far as possible. Parameters in a simulation are variables that denote the state of the environment and the underlying characteristics of a system. Use of parameters rather than constants wherever possible makes it easier to modify the system characteristics and the relation of the system to its environment and thus to increase the validity of the simulation during development.

The tasks involved in development of a simulation are summarised as follows. The first stage of development is the planning and preparation. It includes the initial encounter with the system, the problem to be solved, and the factors pertaining to the system and its environment that are likely to affect the system of the problem. The second stage is the modeling. In this stage the programmer constructs a system model, which is a representation of the real system. The last stage is the validation and application. Once the model has been properly validated, it can be applied to solving the problem at hand. However, the development of the simulation model may still not be complete, observation of the model is needed.

2.4 Advantages and Disadvantages of Simulation

Simulation has been applied to wide field of human activity. There are, however, many cases in which it does not apply; there may be easier and cheaper ways of solving the problem. There are distinct advantages and disadvantages to simulation.

Advantages

1. It permits controlled experimentation. A simulation experiment can be run a number of times with varying input parameters to test the behaviour of the system under a variety of simulations and conditions.

2. It permits time compression. Operation of the system over extended periods of time can be simulated in only minutes with high speed computers.

3. It permits sensitivity analysis by manipulation of input variables.

4. It does not disturb the real system. This is a great advantage, since most managers would be reluctant to try experimental strategies on an on-line system.

5. It is an effective training tool.

Disadvantages

1. A simulation model may become expensive in terms of manpower and computer time. This is not surprising if the magnitude of the problems being attempted is considered. For example, consider the simulation of message through a large-scale (1000-node) communication network. Just the book-keeping requirements for a problem of this magnitude are staggering. The cost of a simulation experiment can be minimised through in-depth understanding of the system being simulated before the model is developed and through careful design of the simulation experiment.

2. Extensive development time may be encounted. Most simulation models are quite large and, like any large programming project, take time. Strategies such as the chief programmer team, top-down design, and modular programming, which have been applied to other large programming projects, are likely to be useful in the development of system simulations and could reduce the development time.

3. Hidden critical assumptions may cause the model to diverge from reality. Idealy this phenomenon should be discovered in the validation phase of the simulation process, but it might go undetected, depending on the severity of the problem and the diligence with which the model is validated.

4. Model parameters may be difficult to initialise. These may require extensive time in collection, analysis, and interpretation.

2.5 Classification

2.5.1 Discrete and Continuous Systems

The terms continuous and discrete applied to a system refer to the nature or behaviour of changes with respect to time in the system state. A system whose changes in state occur continuously over time are *continuous* systems; systems whose changes occur in finite quanta, or jumps, are *discrete* systems. Some of the system state variables may vary continuously in response to events while others may vary discretely. Such systems can be called *hybrid* systems.

In continuous systems, the changes are predominantly smooth. The models of continuous systems generally consists of sets of differential equations; the description of a continuous system generally involves the specification of the rate at which certain attributes change. Examples include fluid moving through a conduit or pipe, aircraft in flight, a spacecraft in orbit about the earth, and electrical circuits.

In discrete systems, changes are predominantly discontinuous like the factory. Few systems are wholly continuous or discrete. An aircraft, for example, may make discrete adjustments to its trim as altitude changes, while, in the factory example, machining proceeds continuously, even though the start and finish of a job are discrete changes. The complete aircraft system might even be regarded as a discrete system. If the purpose of studying the aircraft were to follow its progress along its scheduled route, with a view, perhaps, to studying air traffic problems, there would be no point in following precisely how the aircraft turns. It would be sufficiently accurate to treat changes of heading at scheduled turning points as being made instantaneously, and so regard the system as being discrete. In addition, in the factory system, if the number of parts is sufficiently

large, there may be no point in treating the number as a discrete variable. Instead, the number of parts might be represented by a continuous variable with the machining activity controlling the rate at which parts flow from one state to another. This is, in fact, the approach of a modeling known as System Dynamics. However, in most systems one type of change predominates, so that systems can usually be classified as being continuous or discrete.

The state of a system, either continuous or discrete, is usually expressed as a function of time. The *simulation time* refers to the period of time simulated by the model whatever interval the researcher is interested in. This simulation time is set to 0 at the beginning of the simulation run and acts as a counter to the number of simulation time units.

Time management

Simulation models have been used to model both static (time-independent) and dynamic (time-dependent) situations (see [Graybeal 80]). A static model shows the relationships between entities and attributes when the system is in a state of equilibrium. Most simulation models are dynamic models because a simulation must generally include a means for depicting a time change in the system. This is the *time management.* Two common ways are periodic scan and event scan.

The *periodic scan*, or fixed-time increment, technique adjusts the simulation clock by one pre-determined uniform unit and then examines the system to determine whether any events occurred during that interval. If any occurred, the event or events are simulated; otherwise no action is taken. The simulation clock is then advanced another unit, and the process is repeated. An example of this time management is illustrated in Figure 2.1.

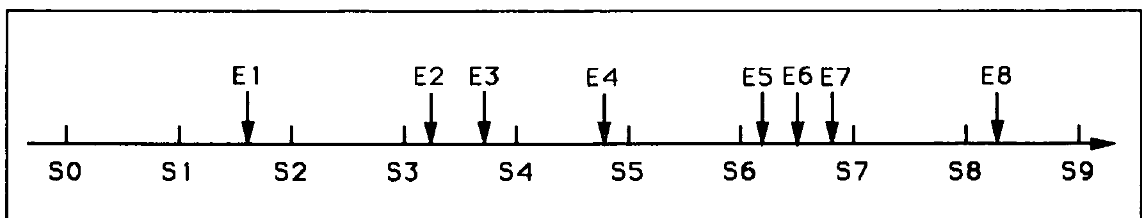

Figure 2.1:

In Figure 2.1 no event occurs in the first unit of simulated time, so the clock is immediately advanced and the system scanned. Then event E_1 occurs in the second time increment. This event would be simulated and the clock advanced again. Since there is no event to simulate during the third interval the clock is again advanced. During fourth interval two events are to be simulated, E_2 and E_3. Following their simulation the clock is again advanced. This process of advancing the clock, scanning the system, and simulating events if necessary is repeated until the duration of the simulation run is reached. With this method the exact time of the occurrence of particular events is largely ignored. All events that occur during a given interval are treated as if these events occurred at the end of that interval.

In the *event scan* approach the clock is advanced by the amount necessary to trigger the occurrence of the next, most imminent, event, not by some fixed, predetermined interval. Thus the time advance intervals are of variable lengths. This approach requires a scheme for determining when events are to occur. When events are discovered or generated, they are generally stacked in a list, or queue, in time order. The length of the required time advance interval can then be determined merely by scanning the event lists to determine the next earliest event. The simulation clock is then advanced to that time, and occurrence of the event is simulated.

2.5.2 Stochastic and Deterministic Systems

A system may be regarded either as deterministic or stochastic, depending upon the causal relationship between input and output. The output of a *deterministic* system can be predicted completely if the input and the initial state of the system are known. That is, for particular state of the system, a given input always leads to the same output. However, a *stochastic* system in a given state may respond to a given input with any one among a range or distribution of outputs. For a stochastic system, it is possible to predict only the range within which the output will fall and the frequency with which various particular outputs will be obtained over many repetition of the observation. It is impossible to predict the particular output of a single observation of the system.

The randomness of stochastic activity would seem to imply that the activity is part of the system environment since the exact outcome at any time is not known. However, the random output can often be measured and described in the form of a probability distribution. If, however, the occurrence of the activity is random, it will constitute part of the environment. For example, in the case of the factory, the time taken for machining would be considered to be an endogenous activity. On the other hand, there may be power failures at random intervals of time. These would be the result of an exogenous activity. If an activity is truly stochastic, there is no known explanation for its randomness. Sometimes, however, when it requires too much detail or is just too much trouble to describe an activity fully, the activity is represented as stochastic.

2.5.3 Open and Closed Systems

A *closed* system is a system in which all state changes are prompted by endogenous activities. In contrast, *open* systems are systems whose states change in response to both exogenous and endogenous activities. It is sometimes difficult to distinguish between endogenous and exogenous activities, and even if the distinction can be made, they are handled in the same way in most simulation studies. Therefore, it is also difficult to distinguish between open and closed systems.

2.6 Simulation Techniques

2.6.1 Random-number Generators

In many simulations events appear to occur at random or to involve attributes whose values must be assigned somewhat by chance. For instance, in many cases the duration of an event is known to fall within a certain range. Simulation of the event requires that a particular value be assigned. Consider the simulation of a general-purpose computer system. One event that must be modeled is the retrieval of a record from a direct-access

storage device. The duration of this event can be determined to fall within certain interval; the actual value, however, is influenced by chance variables such as the position of the record relative to the read head when the request is made. Another instance in which chance appears to play a part is in the widespread use of decision logic in simulation. For example, suppose that in the operation of a system, a given path is known to have taken a certain percentage of the time. Simulation of the system requires a method for selecting this path over others so that the long-run behaviour of the simulator is similar to that of the actual system. Since in most cases these decisions are nondeterministic, the choice is normally based on probabilistic relationships. For these reasons and others, almost any simulations model is required to generate random numbers.

A number of techniques have been applied to overcome the inherent non-reproducibility of random sequences. The first approach is to generate the sequence by some means and to store it, say on tape. This approach is generally unsatisfactory because of the time involved. Each time a random number is required, a read operation must be initiated, and this is a time-consuming operation. This technique also potentially suffers from a short repeatability cycle unless a large sequence is stored. The second approach is to generate a random sequence and hold it in memory. This approach would overcome the speed problem of the previous technique, however, to store a list large enough to satisfy the requirements of many simulation studies would require an inordinate amount of memory. The third and most common approach is to use a specified input value to generate a random number using some algorithm. This technique overcome the problems of speed and memory requirements but suffers from potential problems with independence and repeatability.

The use of an algorithm to generate random numbers seems to violate the basic principle of randomness. For this reason numbers generated by an algorithm are called synthetic or pseudo-random numbers. These numbers meet certain criteria for randomness but always begin with a certain initial value called the seed and proceed in a completely deterministic, repeatable fashion. Extreme care must be taken when using pseudo-random sequences to insure that a fair degree of randomness is present and that the repeatability cycle is long enough. Random numbers are so important to simulation studies that much work has been done in devising and testing algorithms that produce pseudo-random sequences of numbers.

2.6.2 Queuing Theory

Waiting lines, or queues, are encountered in nearly all aspects of life. Queues range from waiting lines at the barber shop, supermarket, or filling station to a backlog of messages at a communication centre, or jobs at a computer centre. The reason that waiting lines form is quite simple: there are simply not enough serving facilities (or servers) to satisfy all the customers simultaneously. The reason for an inadequate number of servers is simple economics. Customers seem to arrive at random; thus to guarantee that there will be no waiting lines, the service station manager would have to hire as many servers as there are customers. This is not economically feasible, hence a fixed number of servers are normally hired with the hope that the waiting lines do not become intolerably long. Should the customers become discouraged and leave before being served, the manager would want to hire more servers to avoid losing business. A queuing system is a system in which customers arrive, wait if that service is not immediately available, receive the necessary service, and then depart.

In the queuing model, the following items are concerned.

1. Queue length. Both the maximum and the average queue lengths are useful in characterising the behaviour of a system.
2. Time in the system. The expected length of time that a customer will spend in a system is of interest to the analyst as well as to the customer.
3. Idle and busy time of the server. Optimal utilisation of the service facility is one of the aims of a system designed.

Now let us see the queuing discipline that determines how the next entity is selected from a waiting line.

1. A First-In, First-Out discipline or, as it is commonly abbreviated, FIFO, occurs when the arriving entities assemble in the time order in which they arrive. Service is offered next to the entity that has waited longest.

2. A Last-In, First-Out discipline, usually abbreviated to LIFO, occurs when service is next offered to the entity that arrived most recently. This is approximately the discipline followed by passengers getting in and out at a crowded train and lift. It is the precise discipline for records stored on a magnetic tape that are read back without rewinding the tape.

3. A random discipline means that a random choice is made between all waiting entities at the time service is to be offered. Unless specified otherwise, the term random implies that all waiting entities have an equal opportunity of being selected.

D. G. Kendall (see [Kendall 53]) developed a widely accepted notation with this convention, a queuing system is described by a series of symbols separated by slashes. For example,

`A/B/C/D/E`

In this notation `A` represents the inter-arrival time distribution, `B` the service time distribution, `C` the number of parallel servers, `D` the system capacity, and `E` the queue discipline. Some of the common inter-arrival distributions are `M` (exponential), `D` (deterministic), E_k(Erlang type k), and `G` (general). For example, `M/M/1/`∞`/FIFO` indicates a single-server with infinite system capacity, exponentially distributed inter-arrival times, exponentially distributed service times, and a first-in, first-out queuing discipline.

Priority queuing systems

In all the previous queuing disciplines, the next customer selected for service in the system was the one at the head of the line. In the priority system, certain customers are given precedence over others. One reason to use priority queue is to reduce the average cost of the system. It may be more expensive to have certain customers wait in line rather than others. It would seem reasonable then to serve the high-cost customers first and thereby reduce the average total cost to the system. Another motivation might be to reduce the average number of customers in the system. The required service time for certain customers may be considerably shorter than for other customers in the system.

By giving priority to customers who require the least service, it is possible to reduce the average number of customers in this system. Two general classes of priority queuing disciplines must be examined.

Nonpreemptive. Once the service of a given customer has started, it cannot be interrupted. If there are customers in the queue with different priorities, the next customer selected for service is the one with the highest priority. If there are customers in the queue with the same priority, some alternative discipline, normally first-in, first-out, is used to determine which customer is served next.

Pre-emptive. In this scheme, if an arriving customer has a higher priority than the customer currently being served, service is interrupted for the current customer, and the higher priority customer gains control of the service facility. The interrupted customer rejoins the queue for service. The question is then, what happens when the interrupted customer is again selected for service? If the portion of service that the customer received is lost and service begins again, the discipline is known as a pre-emptive repeat discipline. If the service is resumed from the point of interruption, the discipline is known as a pre-emptive resume discipline.

Whether a given discipline is pre-emptive or non-pre-emptive does not determine the priority of customers in the queue. Some techniques for assigning priorities to customers in the queue are given below:

> *Shortest service first.* This scheme requires that the required service of each customer be known. Customer are then assigned priorities based on the required service, and the customer requiring the least amount of service is given the highest priority. This technique is generally used in non-pre-emptive disciplines.
>
> *Willingness to pay.* In some systems, customers are allowed to buy a higher priority. Rates are set for various levels of priority, and a customer is charged according to the level of priority desired. This technique is normally used in non-pre-emptive systems.
>
> *Round robin.* Each customer in the queue is given some interval (quanta) of

service before any customer receives a second interval. If the quanta is not sufficient to complete the service on a given customer, service is interrupted and the customer rejoins the queue in a cycle fashion. A number of techniques have been used in order to handle the customer who has received only part of the service. The customer can rejoin the original queue, for example, or join a second queue.

2.7 Languages

One of the most important decision a programmer must make in performing a simulation study is the choice of an implementation language. Many simulations perform similar functions.

Some of these functions are

1. Generating random variates
2. Managing simulation time
3. Handling routines to simulate event executions
4. Managing queues
5. Collecting data
6. Summarising and analysing data
7. Formulation and printing output

2.7.1 Multipurpose Languages

Many programmers tend to select multipurpose languages such as FORTRAN, ALGOL, and PL/I for use in simulation. One of the main reasons is the widespread availabil-

ity of these languages. Even a very small computer installation probably has a FORTRAN, ALGOL, or PL/I compiler. However, these languages have no capability to generate random variates. Many installations have among their library routines a function that generates standard uniform variates. If they use the standard functions to generate the uniform variates, they must code the routines to transform the standard uniform variates to normal or exponential distributions.

List programming in FORTRAN is weak and is usually implemented by arrays. This approach can cause problems, since the maximum size and dimension of the arrays must be determined and declared beforehand. Hence it is not really possible to simulate the operation, such as an $M/M/1/\infty/FIFO$ queuing system. Manipulation of pointers in FORTRAN is also inefficient, since pointers are normally included as a part of multidimensional array. Accessing a particular pointer can then become time-consuming. The actual penalty that results from FORTRAN's list-processing capability depends on the model.

When using multipurpose languages, the programmer must consider the formatting and printing of results. Unlike the specialised languages, multipurpose languages have no automatic output. Input and output routines that are part of the implementation of multipurpose languages provide for flexible formatting, under programmer control. Many installations provide plot routines that may be invoked to provide a visual presentation of the output. However, some effort is required on the part of the programmer to define, interface, and initialise the parameters needed by these routines.

Debugging aids in the multipurpose languages are somewhat limited. They all identify syntactic errors, such as the use of undefined variables and typing errors. Errors in logic, however, must be detected by the programmer. To do this, the programmer must have some idea of what results to expect from the model. The model is then run and its output compared with the expected results. Debugging in these languages is in many respects a trial-and-error process. The programmer finds one bug and eliminates it, only to expose another.

Nevertheless, there are probably more simulation models written in FORTRAN than in any other language. Advantages of using multipurpose languages are as follows.

1. Most Programmers already know a multipurpose language, but this is often not the case with a simulation language.
2. Multipurpose languages are available on virtually every computers, but a particular simulation language may not be accessible on the computer that the programmer wants to use.
3. An efficiently written FORTRAN program may require less execution time than the corresponding program written in a simulation language. This may be because of systems with one set of building blocks, but a FORTRAN program can be tailored to the particular application.
4. Multipurpose languages allow greater programming flexibility than certain simulation languages.

However, a number of programming languages have been produced to simplify the task of simulation programming. Let us see some common specialised languages.

2.7.2 GASP IV

GASP IV is an event-oriented simulation language for discrete-event simulation models consisting of more than thirty FORTRAN subroutines and functions, each of which performs a required simulation activity. Since GASP IV is written in FORTRAN, it is very easy to learn and is usable on almost any computer with a FORTRAN compiler. GASP IV views a system as consisting of entities, their associated attributes, and files which contain entities with common characteristics. All files are stored in one master array. The language provides the user with an executive routine called GASP, which automatically performs such activities as determining the next event from the event list and advancing the simulation clock. On the other hand, the user must write a main program, an initialisation subroutine INTLC (option), a subroutine EVENTS, the usual event routines, and a report subroutine OTPUT (option). GASP IV automatically provides the user with a standard output report at the end of the simulation. If the user wants to obtain additional output data, they can be obtained by writing a subroutine called OTPUT.

2.7.3 SLAM (Simulation Language for Alternative Modeling)

SLAM is an event-oriented or a process-oriented simulation language. The event orientation in SLAM is similar to that in GASP IV. In the process orientation in SLAM, a programmer combines a set of standard symbols, called nodes and branches, into an inter-connected network structure. It affords the diversity of modeling approaches. The programmer can build discrete-event modeling using either the event or the process orientation (or both), continuous models employing differential or difference equations, and combined models using all these elements.

2.7.4 SIMSCRIPT II.5

SIMSCRIPT II.5 is an event-oriented or a process-oriented simulation language considered by many to be the most powerful simulation language available today. General programming tasks can be done more efficiently than in FORTRAN, because of the power and diversity of the statements available in SIMSCRIPT. Furthermore, its English-like and free-form syntax makes SIMSCRIPT programs easy to read and almost self-documenting. SIMSCRIPT II.5 is the only major simulation language with a package for performing statistical analyses of simulation output data. Continuous and combined discrete-event simulation can be performed in SIMSCRIPT II.5. A SIMSCRIPT II.5 model views a system as consisting of entities, attributes, and sets. Entities are of two types, permanent and temporary. Permanent entities correspond to objects in a system, for example, servers in a queuing system, whose number remains fairly constant during the simulation. Conversely, temporary entities represent objects in a system, for example, customers arriving to a queuing system, whose number may vary considerably during the simulation. Attributes are data values which characterise either type of entities, and sets are collections of entities with a common property. To construct a discrete event simulation model in SIMSCRIPT II.5, the programmer must write a preamble, a main program, and the usual event routines.

2.7.5 GPSS (General-Purpose Simulation System)

GPSS is a process-oriented simulation language particularly well suited for queuing systems. The principal appeal of GPSS is the ease and speed with which simulation models can be built. Since many projects operate against tight time deadline, this programming power can be a very important consideration. GPSS offers less programming flexibility than GASP IV or SIMSCRIPT II.5. The programmer who wants to do complicated numerical calculations or obtain a special output report when using GASP IV will have to write a subroutine in, say, FORTRAN and interface it with his program by means of the GASP HELP statement. The GPSS consists of more than 40 standard statements, each of which has a corresponding pictorial representation (called a block) that is intended to be suggestive of the operation performed by a typical customer as it progresses through the system of interest.

2.7.6 DYNAMO

DYNAMO is a language specially developed for System Dynamics models. Variables in DYNAMO are represented by symbols from one to five characters, with some reserved names. The name TIME is reserved for reference to the time in the system model. The symbol DT designates the length of the constant interval which can be decided by the user. To simplify programming, DYNAMO defines a number of equation forms, each of which is a prototype. All equations must comply with these prototypes. The user selects the form of equation he desires to use, and completes the equation in accordance with the prototype structure, using the symbols of the particular variables to which the equation is applied. Each equation type defines a single variables on the left-hand side of the equation in terms of some combination of variables on the right-hand side.

Chapter 3

DISTRIBUTED SIMULATION AND TIME TREATMENT

3.1 Distributed Simulation

A system simulation has the following repetitions: Fetch one event from a data structure, carry out one step of simulation, and update the data structure. However, such simulation is practical only when the number of events being simulated is modest. The recent development of multiprocessors has resulted in demands for new tools for simulations; but simulation is proving to be inadequate for analysis because of the sheer magnitude of the problem. Highly detailed simulation models can be computationally taxing. Computer system simulations are particularly vexing because of the small time scale, milliseconds or microseconds. For instance, a telephone switch generates about 100 internal messages in completing a local call. Large telephone switches can handle 100 or more calls per second. Thus simulation of a telephone switch for 15 minutes of real time requires the simulation of nearly 10 million messages, which will require several hours on a very fast uniprocessor. And also simulation of complex (VLSI) digital circuits for logic verification and fault analysis, for example, can consume months of machine time but designers have

little choice; the alternative, untested designs are unacceptable. Moreover, simulation complexity continues to increase dramatically.

One alternative is to exploit the cost benefits of cheap micro/minicomputers and high-bandwidth lines by partitioning the simulation problem and executing the parts in parallel. Parallel processing is now an active area of research as interconnected arrays of microprocessors are becoming available both commercially and in research laboratories. Therefore distributed simulation may be used. It attempts to reduce the time needed to perform a simulation by spreading its execution over multiprocessors. This is accomplished exploiting the parallelism inherent in discrete-event simulation, allowing the distributed processes to run asynchronously.

Distributed simulation models can mimic a distributed system closely, so we should have a definition of the distributed system. The distributed computing system consists of multiple autonomous processors. They are best suited for computation-intensive numerical applications. Multiprocessors consist of several autonomous processors sharing a common primary memory. These are well suited for running different subtasks of the same program simultaneously. Multicomputers are similar to multiprocessors, except that the processors do not share memory, but rather communicate by sending messages over a communication network. In the distributed system, the processors do not share memory, but cooperate by sending messages over a communication network. Each processor executes its own instructions and uses its own local data, both shared in its local memory. Distributed systems can be contrasted with microprocessors, in which processors communicate through a shared memory.

There are two types of communication networks: *closely coupled* and *loosely coupled* distributed systems. Closely coupled distributed systems use a communication network consisting of fast, reliable point-to-point links, which connect each processor to some subset of the other processors; such as hypercubes and transputer networks. Communication costs for this type of systems used to be on the order of a millisecond, but are expected to drop to less than a microsecond in the near future. Loosely coupled systems are slow and suffer unreliable communication between processors that are physically dispersed; such as the local-area network (LAN). LAN allows direct communication between any two processors. Communication cost is typically on the order of milliseconds. In many LANs,

communication is not totally reliable. A message could be damaged, arrive out of order, or not arrive at its destination at all, so software protocols must be used to implement reliable communication.

Distributed simulation offers a radically different approach to simulation. Shared data objects of sequential simulation such as the clock and event list are discarded. In fact, there are no shared variables in this algorithm, so a speed-up of the entire simulation process is possible. It also offers another advantage in that it requires little additional memory compared to sequential simulation. There is little global control exercised by any machine. Simulation of a system can be adapted to the structure of the available hardware; for instance, if only a few machines are available for simulation, several physical processes may be simulated (sequentially) on one machine. Although, distributed simulation has several problems. It depends on the existence of multi-microcomputer networks. These networks are currently in the design stage and are not commercially available. And assignment of server/queue pairs to processors can be difficult. Only limited performance analyses have been taken, no comprehensive investigation of the expected performance gains has yet been done. Only a subset of all discrete event simulation models are amenable to distributed simulation. Events depending on the global system state are disallowed. Finally, deadlock can occur. Several distributed simulation algorithms have appeared in the literature. They all employ the same basic mechanism of encoding physical time as part of each message. Various distributed simulation algorithms differ in the way they resolve the deadlock issue. We will see them in the Section 3.4.

3.2 Programming Support

Since a distributed simulation could consist of more than one processor, it is possible to have more than one part of a program running at the same time. This is what we mean by parallelism. To obtain a faster simulation means to dedicate more resources to it. In particular, we may be able to speed up a simulation by using a multiprocessor system instead of a single processor. Since most simulations are of systems which consist of many components operating in parallel, it seems reasonable to suppose that the inherent

parallelism in the system can be exploited by the simulation. In addition, independent simulation runs may be required to obtain accurate performance measures of stochastic systems, and these can be done efficiently in parallel. Also, many simulation tasks, such as statistical collection and processing, can be done in parallel with the rest of the simulation.

In distributed simulation processors communicate with one another by message passing. Sending the message involves describing who sends it, what is sent, to whom is it sent, is it guaranteed to have arrive at the remote host, is it guaranteed to have been accepted by the remote process, is there a reply (or replies), and what happens if something goes wrong. Reception of the message involves examining for which process or processes on the host, if any, is the message intended; is a process to be created to handle this message; if the message is intended for an existing process, what happens if the process is busy — is the message queued or discarded; and if a receiving process has more than one outstanding message waiting to be serviced, can it choose the order in which it services messages — be it FIFO, by sender, by some message type or identifier, by the contents of the message, or according to the receiving process' internal state.

Many methods to send messages are introduced. The major design issue for a point-to-point message-passing system is the choice between *synchronous* and *asynchronous* message passing.

With synchronous message passing, the sender is blocked until the receiver has accepted the message. Thus, the sender and receiver not only exchange data, but they also synchronise. In the synchronous message passing, there can be only one pending message from any process S to a process R. Usually, no ordering relation is assumed between messages sent by different processes. Buffering problems are less severe in the synchronous model, as a receiver needs buffer at most one message from each sender, and additional flow control will not change the semantics of the primitive. On the other hand it does have disadvantages. Synchronous message passing is less flexible than asynchronous message passing, because the sender always has to wait for the receiver to accept the message, even if the receiver does not have to return an answer. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message. Conceptually, the sender continues immediately after sending the message. In the asynchronous message passing, there are some semantic difficulties to be dealt with. Since the sender S does

not wait for the receiver R to be ready, there may be several pending messages sent by S, but not yet accepted by R. If the message-passing primitive is order preserving, R will receive the messages in the order they were sent by S. The pending messages are buffered by the language run-time system or the operating system. The problem of a possible buffer overflow can be dealt with in one of two ways. Message transfers can simply fail whenever there is no more buffer space. Unfortunately, this makes message passing less reliable. The next option is to use flow control, which means the sender is blocked until the receiver accepts some messages. This introduces a synchronisation between the sender and receiver and may result in unexpected deadlocks.

Many interactions between processors, however, are essentially two-way in nature. Next, we will look at the two way point-to-point message passing. For example, in the client/server model the client requests a service from a server and then waits for the result returned by the server. This behaviour can be simulated using two point-to-point messages, but a single higher level construct is easier to use and more efficient to implement. Let us examine such a construct, *rendezvous*. With rendezvous, two processes come together, and pass information after which they proceed on their separate ways in parallel. No buffer is required to hold the information as it is passed directly between the processes. If there are no parameters it corresponds to the sending and receiving of a signal between two processes. Whichever process encounters its input or output command first must wait for the other process to reach its corresponding output or input command in which the waiting process is named. Only then both processes will execute their communication statement, that is, rendezvous takes place and the information is passed. The processes are first synchronised after which message can be passed. It is also possible for input parameters to be assigned directly to variables. We will show this example in the following statements using the Ada programming language:

For example, there are processes P and Q, and they can contain **accept** statements, which look like subroutines, with parameters and bodies of instructions.

P reaches the statement:

accept R(X: in; Y: out)

```
    do any process
end
```

It is waiting until Q reaches the statement:

```
R(Z, W)
```

Similarly, if Q reaches its calling statement first, it is waiting until P reaches a corresponding accept statement. If and when both are ready, a rendezvous occurs. First Q delivers its value Z to P's variable X, just like a parameters in a subroutine. Then P proceeds to carry out its body 'any process' which should eventually set Y to some value. Finally, P delivers the final value of its Y to Q. Q in turn receives it in W, much as some subroutines return values. The rendezvous then ends and both processes continue their normal execution.

Distributed simulation could be supported by languages (see [Bal 89]). It is difficult to determine exactly how many languages exist; such as Communicating Sequential Processes (CSP) (see [Hoare 78]), Occam (see [Inmos 88b], [Jones 88], [Galletly 90], and [Pountain 88]), Ada, Concurrent C, Concurrent PROLOG, and Linda, and the like. These languages are distinguished according to how that parallelism is expressed in the language and how parallel units are mapped onto processors, that is, the communication and synchronisation primitives. We have already seen some languages in the previous Chapter, so let us see only Occam in this Section, because we use this language later. Occam is modeled on Hoare's CSP and was designed for programming Inmos' transputers. Occam is essentially the assembly language of the transputer. This language does not have standard features in most modern programming languages, such as records, recursive procedures, and modules. There are three basic actions in Occam: assignment, input and output. The input and output operate via channels and provide inter-process communication between concurrent processes. A channel is a one-way communications link between two concurrent processes. The channel is used to pass data from one concurrent process to another. A channel is shared between only two communication processes — one process may output on the channel, the other may input.

The input process allows a value to be input from a channel and that value to be assigned to a named variable. The input process has the form

channel ? variable

where *channel* is an Occam channel identifier, and *variable* is an Occam variable which receives the named variable.

The output process outputs the value of an expression along a named channel. It has the form

channel ! expression

where *channel* is an Occam channel identifier, and *expression* is an Occam expression.

Suppose that the system consists of process 1 and 2. Process 1 has a procedure A ! 2, and process 2 has a procedure A ? B. This will be read as 'output 2 to A' and 'input from A to B'. Since process 1 and 2 are independent, they might well be executed at different times. The act of transferring a value from one end of the channel to the other can only happen when both processes are ready. In other words, if the output in process 1 is executed before the input in process 2 executes, process 1 will automatically wait for process 2 before sending a value, and vice versa, that is, the processes synchronise.

Both parallel and sequential execution of a group of processes must be explicitly stated, by heading the group with a `PAR` or `SEQ`, respectively. Occam provides a facility for assigning processes to processors. Parallel processes may be prioritised by prefixing the group with `PRI PAR`. The first process in the group is given highest priority; the second, second priority; and so on. Occam also provides an `ALT` construction to express nondeterminism. The constituents of this construction can be prioritised. If input is available more than one channel, the one with the highest priority will be accepted. In Occam, parallel processes communicate indirectly through channels. A channel is a one-way link between two processes. Channel communication is fully synchronous. Only one process may be

allowed to input from, and output to, a channel at a given time. Channels are typed, and their names can be passed as parameters to process, PROC. The current time can be read from an input-only channel declared as a TIMER. A delay until a certain time can be made with the 'WAIT AFTER *t*' construction. This can be used as a constituent of an ALT construction, for example, to prevent a process from waiting forever if no input is forthcoming.

It is worth mentioning some special languages briefly which are DEMOS, SAMOA and MAY (see [Misra 86]). DEMOS is a discrete-event modeling package implemented in SIMULA. It provides an extensive list of features for event scheduling, data collection, and report generation. SAMON uses Ada as the base language. MAY (see [Bagrodia 87]) provides a very small set of constructs for message communication: these features have been used to build an extensive library for simulations of computer and communication networks. It includes constructs to create and terminate processes, to send messages, and to wait for messages to arrive or for simulation time to elapse. The minimality of MAY makes it possible for it to be implemented even on personal computers.

3.3 Possible Approaches in Distributed Simulation

There are five possible approaches of simulations on multiple processors.

Parallelising Compilers can be used to compile a sequential simulation, written in a conventional sequential language to run on a sequential uni-processor, so that it will run on a multi processor hardware.

Distributed Experiments may be conducted by running separate simulations on separate processors in parallel.

Distributed Simulation Events uses a global event list, as in sequential simulation, to schedule available processors to process the next event on the list.

Language Function Distruibution involves different sub-routines or tasks of a simulation being placed on separate processors. For instance processors may be dedicated to random number generation, event list processing, statistics collection, etc.

Model Function Distribution involves decomposing the system model into loosely coupled components and simulating each with a process. One or more processes are then allocated to each processor.

We could have a sixth alternative by using some combination of the others. We show the merits and drawbacks of these approaches in turn.

3.3.1 Parallelising Compilers

A possible approach is to apply a parallelising compiler to a sequential simulation program. The compiler takes a conventional sequential high-level language as its input, and produces the object code to run on each of the multi processors as its output. Thus, the compiler has the responsibility to recognise sequences in the source code which can be executed in parallel and scheduled to run on separate processors. The advantage is that the approach is largely transparent to the user. A new parallel language does not have to be learned, the structure and existing sequential software may be ported. The disadvantage is that the problem has been coded in sequential form, thus ignoring any parallelism in the structure of the problem. This results in relatively small portions of the available parallelism in the problem being exploited and, hence, the speed-up in moving to a multi processor architecture is generally disappointing.

3.3.2 Distributed Experiments

An obvious approach for using N processors to do stochastic simulation is to do independent replications of the simulation on separate processors, and then take an average of the results at the end. This approach seems extremely efficient because no coordination

is required between processors, except for the averaging. Hence, with N processors, the speed up is virtually N. In general, if the run length is long or if the initial transient is weak, then replications will be statistically more efficient than distributed simulation in estimating steady state quantities. Heidelberger (see [Heidelberger 86]) gives conditions on the bias and variance of estimates of steady state performance measures obtained from simulations. This paper concludes that distributed simulation will be statistically more efficient than replications for short runs, for systems with a strong initial transient or if a large number of processors are available. It also says that distributed simulation has the potential for greater statistical speed-up. In general, distributing experiments will be more efficient if the system quickly reaches steady state, and if the simulation run times are long.

However, distributing experiments may not be possible because it requires that all processors have enough memory to contain the entire simulation program as well as the memory to run it. This may be a severe restriction since for many distributed message passing systems, the memory for each individual processor is small. Even in shared memory systems there may not be enough memory for each processor to run an independent simulation. Nevertheless, when these deficiencies are overcome, the distributed experiments approach will be efficient and can use existing sequential simulation programs.

3.3.3 Distributed Simulation Events

Another approach is to maintain a global event list, as in traditional sequential simulation. Protocols to preserve consistency are required, since the next event on the list may be affected by events currently being processed. This approach is particularly appropriate for shared memory systems, since the event list can be accessed by all processors. Jones (see [Jones 86]) uses 'limit times' of currently executing events to determine whether the next event can be safely executed. During the event scheduling phase of the simulation, the limit time is always the same as the simulated time of the most recently scheduled event. This event can be used as a record of the limit time. The distributed events approach may be reasonable when there are a small number of processes and when there is a large amount of global information used by components of the system.

3.3.4 Language Function Distribution

The next approach is to design the simulation support tasks for individual processors. For example, a processor or set of processors are used for random variable generation, event set processing, statistics collection, and the like. The advantage is that it avoids deadlock problems and is transparent to the user. Its disadvantage is that it does not exploit any of the parallelism in the system being modeled. Krishnamurthi et al. (see [Krishnamurthi 85]) describe an implementation using a Motorola 68000 based architecture, a master/slave type configuration, shared memory communication, and the GASP IV simulation language. They use a similar method to the one processed by Chandy and Misra (see the Section 3.5.4). The difference to the C-M algorithm is that the successor process maintains and updates the clock value of its predecessors while processing the messages. The advantage of this approach is that it avoids deadlock that arises due to total absence of messages along any link. When deadlock occurs, for example, the blocked node, say P_1 sends an awakening signal to one blocking node, say P_n to require P_n to update its clock. This awakening signal is propagated until it reaches the predecessor having the greater local clock value than P_1 does. If no such predecessor exists the signal is transmitted back to P_1 which detects deadlock and avoids by not considering the clock value of P_n in computing its forward simulation time. If such a predecessor, say P_k exists P_k sends its latest message to P_1 as a reaction to the awakening signal which then can process all messages with a time stamp less than or equal to the message from P_k. Since the clock values are not maintained for links all the similar messages from various predecessors are enqueued in a single buffer. Multiprocessor simulation systems based on distributing the simulation language functions are also considered by Comfort (see [Comfort 84] and [Comfort 88]). A master/slave approach is described in [Comfort 84] in which all non-event set processing is performed by the principle processor (called the host), event set processing is coordinated by a front-end processor (the master) and actually performed by several other functionally identical processors (the slaves). The results are that a considerable improvement in run time is obtained by using two or more slave processors; a speed up is between 1.2 to 1.3; more than three such processors produce no incremental benefit. In [Comfort 88], Comfort uses transputers and Occam to simulate an `M/M/k/c` queuing system. The maximal efficiency is about 60% on a two processor (host plus one network processor) system, with the network processor doing the random number generation. He

concludes that environment partitioned simulation promises an effective way to reduce the cost of performing large simulations. The transputer is applicable to distributed simulation but certain hardware and methodological limitations must be overcome before that promise is realised. It seems that Comfort's approach produces the significant speed-up when only a few processors are used but the marginal speed-up with additional processors drops off rapidly as processors are added.

3.3.5 Model Function Distribution

The final approach is to decompose the model into loosely coupled components and assign the simulation of each component to a process, where several processes could be run on the same processor. For example, a process might simulate a machine for a manufacturing application. Depending on the definition of the objects, an object-oriented program might use the distributed model approach for decomposing the simulation. Model function distribution is a promising approach for the system which does not require a lot of global information and control, since it has the ability of inherent parallelism. In distributed simulation, the processes communicate by message passing. Usually, messages include time stamps which represent the simulated time of an event. The system is usually modeled as a directed graph in which nodes represent processes and links represent possible interactions or message paths. For fixed topology systems, such as queuing network type systems, the most natural way to model is to assign a process to each station and let the messages represent the movement of customers. A message from one process to another would represent the arrival of a customer from the station, simulated by the first process, to the station, simulated by the second, and would probably include a time stamp representing the simulated time of arrival. An alternative approach is to have processes simulate the customers as well, and a message from a station process to a customer process might represent a change in status of the customer. For dynamic topology systems, such as battlefield scenarios or mobile radio systems, processes could represent the components that are interacting, such as tanks or cars. Messages between processes would represent the interactions between the components being simulated.

The model function distribution requires care in synchronisation. We show the syn-

chronisation for model function distribution in more detail in the following Section.

3.4 Distributed Simulation Models

The model function distribution requires explicit schemes for synchronisation and deadlock handling. How these issues are resolved, depends on whether the simulation time advances in fixed increments or is moved from one event time to the next, and whether simulation is synchronous or asynchronous. *Simulation time* is the abstraction of real time, in that the state of the real system at any real time will corresponding simulated time. The values assumed by simulation time must be discrete since we are interested in discrete simulations. If the simulation is synchronous there is a global clock and all processes must have the same simulation time. Contrary, in the asynchronous simulation each process has its own local clock and the simulated time for different processes may be different. There are number of aspects to be considered in the characterisation of a simulation methods. Each link is associated with a sequence of events which are in monotonic non-decreasing simulation time order. This means that the events arriving on input links are in the correct time sequence. It follows that nodes must generate events for their output links in non-decreasing simulation time order. Another aspect is the availability of one or more processors. The idea of mapping each component of a simulated system onto a separate processor has great intuitive appeal, since the processors are manipulated as the components of the real system, and parallelism inherent the simulation may be exploited. Also, the interactions between the components of the system must be reflected in the interactions between processors. We show synchronisation issues in more detail in turn.

3.4.1 Time-driven Simulation

In time-driven simulation, simulated time advances in fixed increments, called ticks, and each process simulates its component over each tick. The clock ticks must be short to guarantee accuracy, but shorter ticks imply longer simulation time, because it is more

likely that nothing will happen during a tick. The simulation may be synchronous or asynchronous. If it is synchronous all processes must finish simulating a tick before any can start simulating the next tick. In this case, the simulation of a tick typically proceeds in two phases, a simulation or computation phase, and a state update and communication phase. When clocks are local and simulation is asynchronous a process can begin simulating the next tick as soon as its predecessors have finished the last tick. Synchronisation for local clocks is implicitly provided by sending messages from a processor to its successors.

There are many possible implementations of a global clock. A centralised approach requires a dedicated process to act as synchroniser. In a message passing system a global clock could be implemented in a distributed fashion with an appropriate broadcast algorithm. A global clock could be implemented in a shared memory system in which a counter would be decrement by a process when it finishes simulating the current tick. There could also be a separate synchronisation bus to provide the global clock. Peacock et al. (see [Peacock 79a]) describe a possible implementation. They say that a method is *scaled* if it has the property that

```
s = int (k * r / q) * q,
```

where s is simulation time, r is real time, q is the quantum step size of simulation time, and k is the time scaling factor. With this property, observation of the dynamics of a system with a constant rate of simulation time passage relative to real time is possible. They call this method the *Scaled Real Time Method*, and it strongly resembles analogue computing. Its advantages are that arbitrary specification of functions is possible, and hence non-linear components are easily modelled; high accuracy is attainable; and digital electronics are exploited. One problem is that the simulation may be indeterminate, in other words, if the same simulation is run twice, the results may be different. Since each processor uses a different clock, and it is unlikely that these clocks are in complete synchronisation.

Asynchronous simulation seems to permit greater concurrency, but it may increase communication costs, depending on the application. In a message passing system with local clocks, messages must be sent on all paths for each tick. Since a processor cannot simulate the next tick until it knows its predecessors are finished simulating the last

tick, it must receive a message from each of its predecessors for each tick even if the predecessor does not change state or there is no interaction with the predecessor for that tick. On the other hand, if simulation is synchronous, once the global clock is synchronised, only messages signaling interactions or state updates need to be transmitted. In general synchronising the global clock can be done more efficiently than by sending messages on all paths. Hence, if state changes or interactions occur much less frequently than every tick, a global clock may be more efficient.

Time-driven simulation seems less efficient than event-driven simulation since there may be ticks, during which no events occur, that must still be simulated. However, time-driven simulation avoids the extra synchronisation overhead required for synchronous and asynchronous event-driven simulation. Thus, time-driven simulation may be particularly appropriate for dynamic topology systems, and for systems in which many things are happening at the same time. To improve the efficiency of time-driven simulation, it may be advantageous to vary the tick size so that large ticks would be used in simulating a time when there is little activity in the system, although this would increase the overhead.

3.4.2 Event-driven Simulation

In event-driven simulation simulated time is incremented from one event time to the next, where an event represents a change in state. Thus, event-driven simulation may have greater potential speedup than time-driven simulation. As with time-driven simulation, the computation may be either synchronous or asynchronous. If it is synchronous, the global clock is sent to the minimum time of next event for all processes, whereas if it is asynchronous, the local clock for each process is set to the minimum next event time for that process.

1. Synchronous event-driven simulation

As in time-driven simulation, the implementation of a global clock can either be centralised, with a dedicated process to act as synchroniser, or distributed. In the centralised

approach, the global clock is the minimum simulation time of the next event that involves an interaction between processes that are being executed on different processors. This scheme requires a control processor to maintain the global clock. Venkatesh et al. (see [Venkatesh 86]) propose centralised synchronous approach. They introduce the Accelerated Time Advancement algorithm. This algorithm does not require the channels to be FIFO and it also optimises the propagation of test messages (reducing overhead) by accumulating evaluation results at each process from an optimal set of predecessor processes before a new test result is generated and forwarded from that process. Each sender process retains minimal and precise information about the simulation times of the stimuli sent on every outgoing channel. This enables each process perform the evaluation only once in a test iteration. Distributed algorithms could be implemented by the hierarchical tree architecture. In the tree architecture, the lowest levels are processing elements that actually simulate the events, and the highest levels are coordinators used for synchronisation and message routing. Processors send their next event times to the coordinators at the level above them. Each coordinator determines the minimum next event time for the level below it and sends this time to the coordinator above it. The coordinator at the top or root of the tree determines the minimum next event time for all processing elements (i.e., the global clock) and propagates the time back down the tree.

Another possibility is to use a single one-bit synchronisation bus. Each processor computes the next event time for the processes. They then put the first, or leftmost, bit of that time on the bus. The output is the logical AND of all the inputs, in other words, the minimum. Those processors whose input matches the output, repeat the procedure with the next bit. The rest of the processors, those whose most significant bit was greater than the global clock, drop out until the next global clock update. The procedure is repeated for each bit, and the total output is the time of the next event for all processors, in other words, the global clock. This procedure is constant in the number of processors and can be implemented efficiently with a hard wired-logical AND gate. It is also possible to combine time-driven and event-driven simulations. The global clock is set to the next event time for all processes, as in event-driven simulation, but processes may simulate events within a time interval or tick of the global clock, as in time-driven simulation. The performance of algorithms for synchronous event-driven simulation has not been evaluated.

2. **Asynchronous event-driven simulation**

Asynchronous event-driven simulation has received the greatest attention due to its potentially high performance. Since processes spend less time waiting for other processes than that in time-driven simulation or synchronous event-driven simulation. Events that do not affect one another can be simulated simultaneously even if they occur at different simulated times. It is assumed that processes communicate by sending messages to each other with time stamps, (t, m), either using actual messages in a message passing system, or passing pointers to message queues in a shared memory system. Two implementations are proposed: conservative and optimistic. In conservative approaches a process' clock can never exceed the clocks of its incoming links, insuring correct chronology for all event processing. In the optimistic Time Warp approach, a process' clock may run ahead of the clocks of its incoming links and, if errors are made in the chronology, time must be 'rolled back' to correct them. Lamport (see [Lamport 78]) works on the definition of clocks in distributed simulation. He uses logical clock and defined that the concept of 'happening before'. He shows that the times '*happens before*' and '*happens after*' are operationally definable within a distributed system form only a partial order instead of a total order. And he further shows an algorithm for extending that partial ordering to a somewhat arbitrary total ordering. We show two approaches in detail in the following Sections.

3.5 Conservative Approach

Physical systems are simulated by partitioning them into physical processes (pps) that communicate by messages. The logical process (lp) depends only upon the pp that is simulated. There is a communication line from the ith lp to the jth lp in the logical system if and only if the ith pp sends messages to the jth pp, in the physical system. Hence, there is no central process, logical messages synchronise without a global clock. When the pp sends a message at time t, it cannot be influenced by messages transmitted to it after t. This is called *realisability.* Realisability says merely that a pp cannot guess any message it will receive in the future. *Predictability* is lookahead to be the amount of time that a pp can look into the future. For example, the local clock time is t, and

the process determine all messages it will send with time stamps less than $t + \epsilon$, $\epsilon > 0$. Predictability guarantees that the system is well defined, because the output of every pp up to any time t can be computed given the initial state of the system.

3.5.1 Conservative Approach Algorithm

In conservative approaches, messages from any process to any other process are transmitted in chronological order according to their time stamps. A message m from pp_i to pp_j at time t is simulated by lp_i sending lp_j a message (t, m). For example, the sequence of messages sent by lp_i to lp_j is (t_1, m_1), (t_2, m_2), $(t_3, m_3), \cdots$. It must follow that $0 \leq t_1 \leq t_2 \leq t_3 \cdots$ are monotonically increasing. It also implies that pp_i must have sent the message m_k to pp_j at time t_k, k=1, 2, 3,..., and pp_i must have sent no other messages to pp_j besides $m_1, m_2, ..., m_k, ...$ in other words, the sequence of messages sent by an lp must correspond exactly to the actual sequence of messages sent by the corresponding pp. During the simulation, if lp_i sends lp_j a message (t_k, m_k) it is implied that all messages from pp_i to pp_j have been simulated up to time t_k. A message is transmitted from lp_i to lp_j only if lp_i is waiting to send the message to lp_j and lp_j is waiting to receive a message from lp_i, in other words, the processes synchronise. If there is a non-zero size of buffers between lp_i and lp_j, then lp_i may transmit messages until the buffer is full.

An example of such a system is shown in Figure 3.1, it consists of a source, a sink, and two queues.

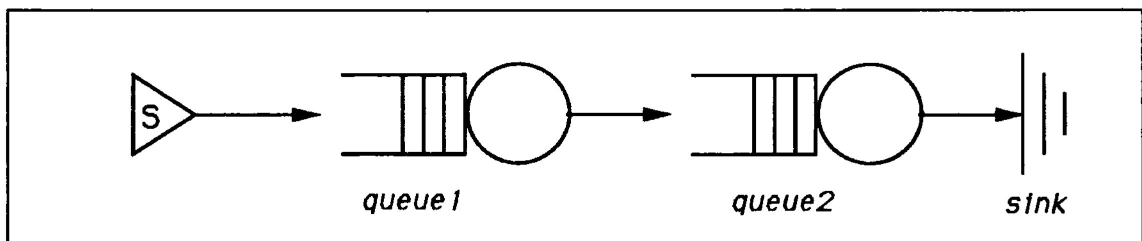

Figure 3.1:

Each queue is a First-Come First-Served discipline, all components are simulated by distinct lp's. Assume that queuel has service time (stl) 5, queue2 has service time (st2)

10. Now suppose that the source (LP0) generates the message (3, m_1), and sends it to queuel (LP1). Upon receipt of this message, LP1 can determine that this message will depart at time 8 (arrival time + service time). Therefore, LP1 can now send the message (8, m_1) to queue2 (LP2). Upon receipt of this message, LP2 can determine that this message will depart at time 18. It will send the message (18, m_1) to the sink simulated by LP3. Meanwhile, LP0 would have sent the message (5, m_2) to LP1. Since LP1 can process this message after sending the previous message (8, m_1), LP1 will send the message (13, m_2) to LP2 (where 13 = last departure + service time). Upon receipt of this message LP2 will send (28, m_2) to LP3. Note that all the LPs could work in parallel.

3.5.2 Waiting Rules For Logical Processors (lps)

The lp would determine which set of lines it should wait either to receive the message or to transmit the message according to following two wait rules; the first, an lp waits to receive messages on input lines whose clock values equal the lp clock value; the second, an lp waits on all output lines on which there is a message to be sent.

Suppose that there is a pp as shown in Figure 3.2. The pp consists of two input lines

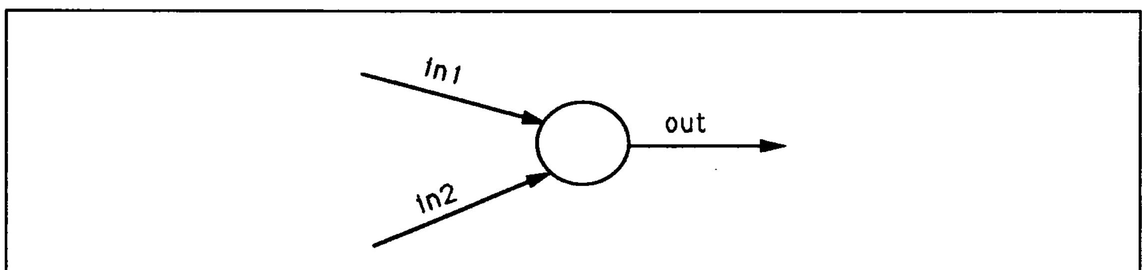

Figure 3.2:

(in_1 and in_2) and an output line (out), these lines are First-Come First-Served queues. Assume that the pp's service time is 5. Initially the clock values for all lines are 0 and lp clock value = 0. The lp is waiting for messages on in_1 and in_2 and is not waiting to output, since there is no message to be output. Now suppose that the lp receives the message (5, m_1) on in_1, but the lp cannot process this message. Since it is waiting for the message on in_2 according to the waiting rulel. Assume that (3, m_2) is received on

in_2. Then the lp can guarantee that no other message will arrive at pp before time 3 and the next output will occur at $3 + 5 = 8$ corresponding to m_2. The lp clock value is now 3. The lp waits to input on in_2, since the link time for in_2 = lp clock value and waits to output (8, m_2), since it has something to output.

3.5.3 System Deadlock

Deadlock can occur in a simulation. Consider the following example shown in Figure 3.3. The system consists of the source, sink, and 4 servers (P1 to P4). P1 is a fork

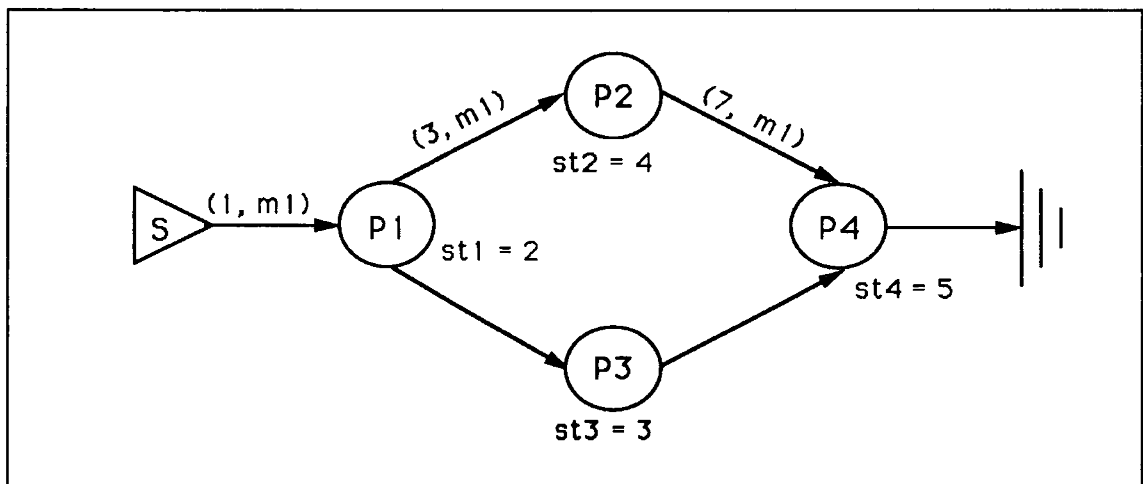

Figure 3.3:

consisting of a single input line and two output lines, and P4 is a merge consisting of two input lines and a single output line. Suppose that the servers P1 to P4 have service time 2, 4, 3, and 5, respectively. Now assume that the source sends the message (1, m_1) to P1. P1 would process it and send the message (3, m_1) to P2, P2 would process it and send the message (7, m_1) to P4, and P4 would receive it and then wait to receive the message on P3-P4 (which is a link between P3 and P4), according to the waiting rules. Now the P1 clock value is 1, the P2 clock value is 3, the both P3 and P4 clock values are 0. Meanwhile, the source would have generated the next message (5, m_2). P1 would process and send (7, m_2) to P2, P2 would process and try to send (11, m_2) to P4. However, P4 is not waiting for the message on P2-P4 but on P3-P4. P2 must wait because it has the message to be output. When the source generates the messages and P1 sends all to P2

(see Table 3.1), eventually it would happen that P4 waits to receive the message on P3-P4, P3 waits to receive the message on P1-P3, P2 waits to send the message on P2-P4, and P1 waits to send the message on P1-P2. In Table 3.1, the number is the time component of the message to be sent, W implies that the link between processes does not have any message to be sent, and the number with W implies that the process has a message to be sent on its output link but the destination process is not waiting for it, so the process must wait to send.

LINK	*JOB*			
	1	2	3	4
source → P1	1	5	15	20 W
P1 → P2	3	7	17 W	W
P1 → P3	W	W	W	W
P2 → P4	7	11 W	W	W
P3 → P4	W	W	W	W

Table 3.1:

In other words, P3 blocks P4, P1 blocks P3, P4 blocks P2, and P2 blocks P1, in other words, deadlock occurs. Deadlock occurs when there is a cycle of waiting (W) → not waiting (N) arcs that are assumed to go from W to N (see Figure 3.4).

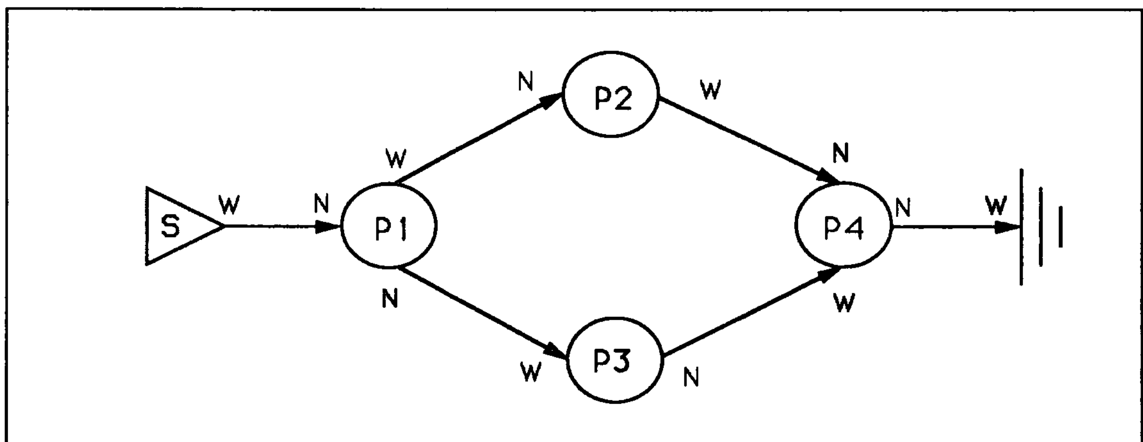

Figure 3.4:

3.5.4 Deadlock Avoidance (Chandy and Misra Algorithm)

One scheme for breaking deadlock is to send the null message (t, null) proposed by Chandy and Misra (see [Chandy 81]). The procedure that lp_i sends (t, null) to lp_j implies that pp_i does not have the message to be transmitted to pp_j in the time interval between the last message along line (i, j) and t. The null message does not correspond to any message in the physical system. Reception of a null message is treated in the same manner as the reception of any other message. Therefore, it causes the lp to update its internal state, including the clock value, and to send messages. We use the previous example to explain the deadlock avoidance by null messages.

In the previous example, P1 could send a message (t, null) along an outgoing line every time it sends a message (t, m) on the other outgoing line. This is shown in Table 3.2 in which numbers marked with a * are null messages. For instance, P1 sends (3, null) to P3,

LINK	*JOB*			
	1	2	3	4
source → P1	1	5	15	20
P1 → P2	3	7	17	22
P1 → P3	3*	7*	17*	22*
P2 → P4	7	11	21	26
P3 → P4	6*	10*	20*	25*

Table 3.2:

P3 can then predict that it does not have a message to send P4 until time 6 and hence it will send (6, null) to P4. Note that the merge P4 will output a stream of messages (6, null), (7, m_1), (10, null), (11, m_2), etc. Thus deadlock is avoided.

With the C-M algorithm deadlock cannot occur. Deadlock occurs if there is a cycle of blocked processes. Suppose that there is such a cycle, in which P1 is blocking P2, P2 is blocking P3, and so on to P_n and P_n is blocking P1. Assuming that t_i is the P_i clock value. Then if P_i is blocking P_j, P_j is waiting for the message from P_i and its clock value t_j must equal the link clock value $P_i - P_j$. However, if P_i is also blocked, it must have updated its output link clocks to be greater than or equal to P_i clock value, t_i. Hence $t_j \geq t_i$ and therefore, $t_1 \geq t_2 \geq \cdots \geq t_n \geq t_1$. By the predictability assumption, at least

one process in the cycle, say P_i has output link clocks that are strictly greater than its local clock, i.e., $t_i > t_{i+1}$. Hence this simulation has $t_1 > t_1$, a contradiction. Therefore, deadlock cannot occur. It is interesting to note that the simulator never deadlocks; even if the system being simulated does. During the simulation of the deadlock null messages will be only transmitted.

3.5.5 Deadlock Detection and Recovery

Another approach to break deadlock is to detect deadlock and recover it. Various methods are proposed. Chandy and Misra present (see [Chandy 81]) two phase schemes in which simulation proceeds until deadlock then deadlock is detected and corrected. Chandy and Misra use a '*controller*' (see [Chandy 81]) which monitors for deadlock and control deadlock recovery. Misra proposes the use of a '*marker*' (see [Misra 83]). The marker circulates among all processes to detect deadlock. The marker records the number of blocked processes and the minimum of next event times. When the marker detects deadlock, it knows the next event time and the lp at which this next event occurs.

Peacock et al. (see [Peacock 79a] and [Peacock 79b]) and Bain and Scott (see [Bain 88]) propose the use of *probe messages.* When a process is blocked it sends a probe message time-stamped with its local clock to some of its predecessors in order to obtain information on their clocks. A process that received this probe message will send its local clock value to the requesting process if it is later than the requesting processes local time. Otherwise it sends probes to its predecessors. This approach could require that the probe messages contain the path it has traversed and the local clocks of the processes in the path in order to detect and correct a deadlock. Therefore, messages grow in length as they are passed along. Bias and Scott use three types of probe messages, YES, NO, and RYES. RYES is reflected yes or conditional yes. This algorithm requires that each process keeps its probe messages it has received, and the path length is fixed. Therefore, the message includes a process identification from the originating process and the requested time.

Groselj and Tropper (see [Groselj 88]) propose an algorithm for computing the greater lower bound of the time stamps of the next events to arrive at all empty links of lps

located in one processor. This algorithm is based on the shortest path computation. The algorithm helps to unblock the blocked lps and therefore increases the parallelism of a simulation. However, this algorithm does not include the global deadlock problem so it must be used in conjunction with one of the other methods.

3.6 Optimistic Approach

For an optimistic approach, Time Warp was proposed by Jefferson (see [Jefferson 85]). Time Warp is a general lookahead-rollback algorithm. Whenever a conflict is discovered after the fact, the offending process(es) are rolled back to just before the conflict, no matter how far back that is, and then executed forward again along a revised path. All messages consist of four values: the name of the server, the virtual send time, the name of the receiver, and the virtual receive time. The virtual send time is the virtual time at the moment the message is sent, and likewise the virtual receive time is the virtual time at the moment the message is received. The virtual receive time is the same as the time stamp of the conservative approaches. The send time is used for implementation of the Time Warp algorithm. The Time Warp approach maintains two clocks; the local virtual clock (LVT) and the global virtual clock (GVT). The LVT of a process is set to the minimum receive time of unprocessed messages. Processes can execute events and proceed in local simulated time whenever they receive any event on any input link. This is in contrast to the conservative approach which requires the process to receive an input from all of its predecessors to execute. Therefore, the LVT of a process may be ahead of its predecessors' LVTs, and it may receive the message with a lower time stamp, in other words, before its LVT. If this happens, the process 'rolls back' to an earlier virtual time, cancelling all intermediate side effects, and then executes forward again, this time receiving the later message in its proper sequence. The process is constantly gambling that no message will arrive with a virtual receive time less than the one stamped on the message it is currently processing. As long as it wins this bet, execution proceeds smoothly, however, whenever the bet is lost the process pays a performance penalty by rolling back. GVT is the minimum of the LVT in all processes and the send times of all messages sent but unprocessed. Hence, GVT is the states of the each process since the last

'correct' time in order to enable rollback. Assuming infinite memory, and assuming the system being simulated does not deadlock, the Time Warp algorithm will not deadlock. This is because individual processes do not deadlock as long as they have some inputs and GVT can be shown to always eventually increase. GVT must be estimated so often during the execution. High frequency of estimation produces faster response time and better memory space utilisation, but it also uses processor time and network bandwidth, and thus slows progress. A disadvantage of Time Warp is that it requires a large amount of memory. One way to limit the amount of memory needed is to estimate GVT often, and to remove outdated messages from the input queues and outdated states from the state queue. In addition, states with time stamps greater than LVT, which arise because of rollback, may be discarded. The Time Warp approach has potentially greater speed-up than conservative approaches, but it requires the greater memory size. An advantage of Time Warp over conservative approaches is that the topology of possible interactions between processes need not be known. In addition, Time Warp does not require that messages be received in the order sent along links. To implement Time Warp each process must maintain the following: its process name; its LVT; its current state; a state queue containing copies of its previous states, with at least one state before GVT; an input queue containing all received messages with sent times greater than or equal to GVT, in receive time order; and an output queue containing copies of all messages sent with send times greater than or equal to GVT, in send time order. Messages sent forward in simulated time have a positive (+) sign; the copies that are kept in the sender's output queue for use in case the rollback are antimessages and have a negative (-) sign. Whenever a message and its antimessage are in the same queue they immediately 'annihilate' one another.

Let us see the example shown in Figure 3.5. Ordinarily, when the process receives a

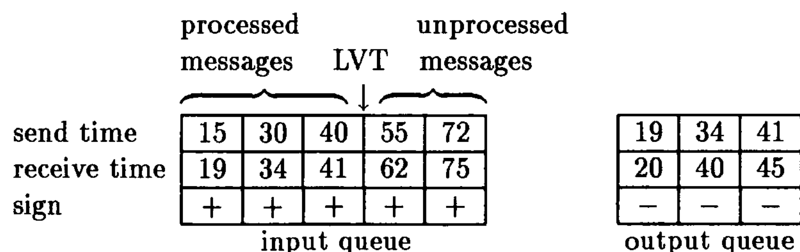

Figure 3.5:

message with the time stamp greater than its LVT, the message is simply enqueued, and

the running process continues. Suppose that the message is received at clock value 35 by the process, say P1, whose LVT is 62. P1 must roll back since it receives a message with a time stamp less than its LVT. Such a message is called a '*straggler*'. At first, P1 searches the state queue for the last state of P1 saved before the straggler, that is before 35, and then restores 35 as the value of its LVT. After this, all the states saved after the straggler on the state queue must be discovered and P1 starts executing forward again. However, the simulation is incorrect between 35 and 62, so P1 sends the antimessages to cancel the transmitted messages within that time. Figure 3.6 shows the rollback for the example in Figure 3.5.

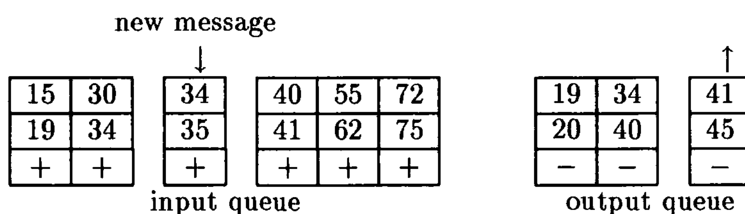

Figure 3.6:

When a process receives an antimessage several things may happen. If the positive message has arrived, but it has not processed yet, its receive time must be greater than the receiver's LVT. Then, the antimessage, having the same receiver time, will not cause a rollback, but will be enqueued. However, it will cause an annihilation. Therefore, both messages disappear from the input queue. If the positive message arrives first, its antimessage arrives next, and the LVT of the process is less than both receive times when the antimessage arrives (so the messages are in the simulated future of the process), then they will annihilate each other and the process will proceed. If the positive message arrives first, its antimessage arrives next, and the LVT of the process is greater than the antimessage's receive time when the antimessage arrives, then the process must roll back. It sets its current state to the last state saved with simulated time of the message. The positive message and its antimessage will annihilate each other, and the process will proceed. As a consequence of the rollback, more antimessages may be sent to other processes.

The procedure we have seen so far is called aggressive cancellation since when a process rolls back, it immediately sends antimessages cancelling messages sent at simulation times

later than its new LVT. An alternative is lazy cancellation, in which antimessages are not sent immediately after the rollback. Instead, the process resumes executing forward in simulation time from its new LVT. When the process produces a message it compares it with the messages in its output queue. Therefore, only positive messages that have not previously been sent are transmitted, and only antimessages that are produced in the forward computation are transmitted. Under aggressive cancellation a process may send a message to a successor, then send its antimessage, and then send the same message again. Under lazy cancellation the message would be sent just once. Thus, under lazy cancellation a rollback at the successor may be avoided. On the other hand, if messages are not reproduced, then rollbacks at successor processes will be required under both mechanisms, and they will occur sooner with aggressive cancellation. States may be saved less frequently, at the expense of greater overhead for rollback. However, lazy cancellation requires more memory than aggressive cancellation.

Lomow et al. (see [Lomow 88]) study the performance of Time Warp. They conclude that lazy cancellation can achieve better speed-ups compared to aggressive cancellation on a large number of processors, but when the number of processors approaches the number of processes the speed-up declines markedly. Poor assignment of processes to processors and feedback in the simulation model makes the simulation slow down.

The performance of Time Warp has been studied analytically by Mitra and Mitrani (see [Mitra 84]). They use two processors whose speeds are different. They find that it is sometimes advantageous to slow down the faster processor, but this analysis cannot be extended to more than two processes. The empirical work promises that Time Warp is an efficient approach to synchronisation in distributed simulation.

Gilmer (see [Gilmer 88]) obtains efficiencies as high as 90%. He finds that the number of processes per processor and the load balance has significant impact on performance. When there are 8 processes per processor for all 128 processors, an efficiency of 91% is achieved. However, below a ration of 8 processes per processor, the number of antimessages starts to increase rapidly because each roll back causes increased number of message cancellations.

3.7 Previous Performance Studies

Various methods have been proposed using the conservative approach in order to speed-up, resolve deadlock problems, distribute the processes on to processors, and the like. So far we have seen some proposed algorithms. We have also seen some empirical work. However, very little empirical work has been done to determine the performance of conservative schemes.

Fujimoto (see [Fujimoto 88]) simulates deadlock avoidance and deadlock detection and recovery, on a shared memory multiprocessor, the BBN Butterfly. He assumes a toroid system topology (a two-dimensional mesh with warp-around). He obtains efficiencies as high as 75% for the null message scheme, where the speed-up is computed relative to a single processor simulation using a single event as opposed to a single processor distributed simulation. He shows that the poor (or good) speed-up is caused by high (or low) overhead in the simulation strategy. He finds that a process with poor lookahead will lead to poor performance. Since in conservative approaches, messages are sent in non-decreasing order, if lookahead is poor, the process may not even be able to determine the proper time stamp of the message very far in advance. Therefore, poor lookahead causes a delay in the sending of messages, effectively decreasing the message population and the available parallelism. He also shows '*message avalanche*'. There exists a critical message population level, above that the performance improves dramatically, below that performance is poor and relatively constant. He concludes that conservative approaches can obtain significant speed-ups for some, but not all workloads carring moderate to high degrees of parallelism.

Reed (see [Reed 85]) simulates a central server closed queuing network on a uniprocessor. He finds that the queue management is the dominant overhead in distributed simulation and that hardware queue support could reduce simulation time. He also finds that the deadlock detection and recovery is superior to the deadlock avoidance, particularly when there is a feedback mechanism and small job populations in the networks.

Reed, Maloney, and McCredie (see [Reed 88a] and [Reed 88b]) simulate the conservative approach using shared memory on a Sequent Balance 21000 containing 20 processors. In a shared memory implementation, all node state information, including input message

queues, resides in shared memory. Each node communicates with each other node by messages via shared access to the message queues of each node. Each message queue is protected by a synchronisation lock to guarantee mutual exclusion. Before transmitting a message, a node must first acquire a free message from a shared free message list. A lock is necessary to prevent simultaneous access to the free message list. After retrieving the free message, the node time stamps it and writes it to the destination node's message queue. A message is returned to the free message list once it has been processed by the destination node. They obtain the maximal speed-up of about five relative to a uniprocessor running the distributed simulation. They find that a single processor implementation of the C-M algorithm is usually slower than the equivalent sequential, event-driven simulation. Networks with cycles require deadlock avoidance or recovery techniques, and the inability to lookahead limits parallelism. They conclude that the C-M algorithm is not a viable approach to parallel simulation of queuing network models, but may be appropriate for other models.

Chapter 4

STUDIES

4.1 Motivation

In this Chapter we study the conservative approach using null messages on transputers. The C-M algorithm is the so-called 'conservative' approach developed by Chandy and Misra (see [Chandy 81] and [Misra 86]). This algorithm is based on deadlock avoidance using null messages. As we have seen in the previous Chapter, some people have studied this algorithm by various methods. Although, there are no definite results as to whether this algorithm is reasonable, it may be evaluated under certain conditions; such as the system structure, the number of reasonable processors, the number of processes per processor, the data size, and the like. More empirical work is required for use of this algorithm, despite it having been thoroughly developed. It is therefore important to examine its applicability in distributed simulation.

Transputers were developed by Inmos (see [Inmos 88a]). Since 1985, the IMS T414 32-bit transputer has been introduced, as well as further developments which increase the memory, processing performance and communication performance. The floating point transputer, IMS T800, was first introduced in 1987. The transputer is a microprocessor

with its own local memory and with links for connecting one transputer to another. Initially, the transputer consists of memory, processor and communication systems connected via a 32-bit bus. The transputer uses point-to-point serial communication links for direct connection to other transputers. It supports the scheduler which in turn supports two priority levels-high priority and low priority. It can run several processes consisting of a sequence of instructions in parallel.

Occam has also been developed by Inmos (see the previous Chapter for the details of Occam). An entire system can be designed and programmed in Occam, from system configuration down to low level I/O and real time interrupt. It is also possible to program the transputer in several high level languages such as C, Fortran, Pascal and Ada for which compilers have been written, but Occam programs run a good deal faster than these because they are translated into machine language with greater ease. The transputer and Occam should be powerful enough to examine the C-M algorithm, and recent developments in user-friendly environments have caused more people to use them.

4.2 The Simulation Algorithm

For studies of the conservative approach we used an algorithm based on the C-M algorithm introduced in [Chandy 81] and [Misra 86]. We will examine deadlock avoidance using null messages. The C-M algorithm is totally asynchronous so each process maintains its own local clock; global synchronisation, such as a global clock, is not used. Every process communicates with each other by passing messages which keep strict chronological order and which are transmitted synchronously. Also, every process works in parallel with no shared variables (see details in the previous Chapter).

In the experiments we used the link time in order to send real messages and null messages to each node evenly. Consider the following example shown in Figure 4.1.

Suppose that P1's service time (st1) is 3, st2 is 4, st3 is 3 and st4 is 2. Assume that the **source** sends (1, m_1) to P1, P1 then sends (4, m_1) to P2 and (4, null) to P3. P2

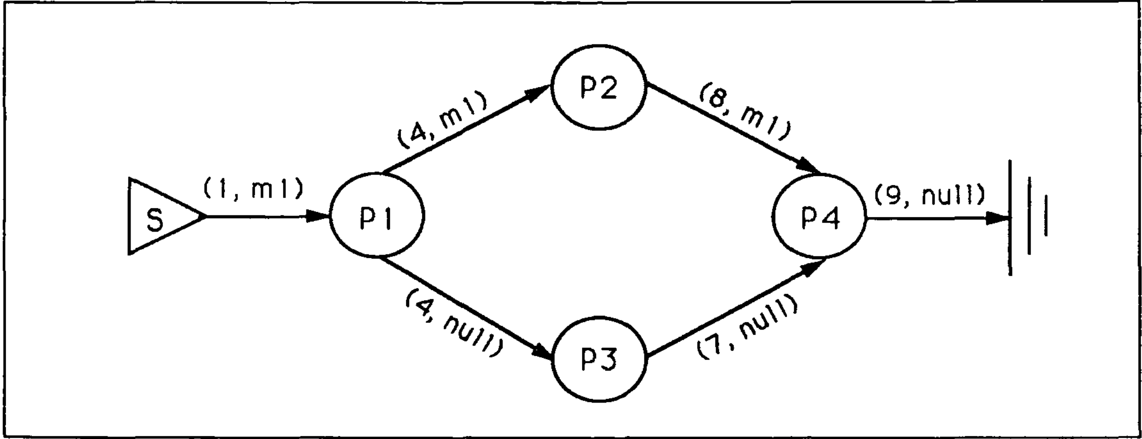

Figure 4.1:

and P3 receive messages and send their messages to P4, respectively. Now the link time between P1 to P2 (lt_{1-2}) is 4 and lt_{1-3} is 0 because the null message does not advance the link time. Meanwhile, the **source** generates the message (6, m_2) and sends it. P1 then predicts that the next message output is (9, m_2), checks lt, and sends the real message on the link having the smallest link time. In this example (9, m_2) is transmitted to P3 and (9, null) is transmitted to P2. Actually, in this example the **fork** has two branches so real messages are transmitted alternately. When the **merge** receives messages on all its input links, it processes the smallest time component message so that the chronological message passing is kept. This method is the same as one making use of waiting rules.

If we do not care about link time and send messages, all real messages could be transmitted to P2 and all null messages could be transmitted to P3. Deadlock may be avoided but this implies that during simulation P3 does not actually work, and only increases its clock value by receiving null messages, that is, P3 is idle. This system may be expensive. However, if the link time is used, it will increase the amount of parallelism.

4.3 Experimental Environment

We used the FAST4 board consisting of four IMS T800s, each with 1Mbyte of memory. The board is plugged into an IBM PC. The link layout on this board is shown in Figure 4.2.

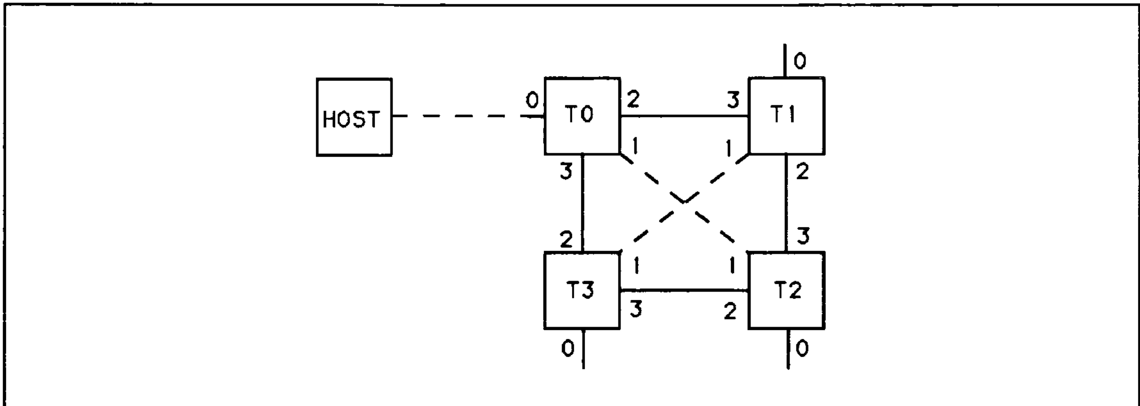

Figure 4.2:

Links 0 and 1 from each processor are taken to the rear connector. The remaining links are pre-wired into a square with link 2 of each processor connected to link 3 of the next processor. The FAST4 board is supplied with a pre-wired connector to attach the host link to link 0 of processor 0. For completion, it also connects the spare links 1 in a cross: T0 to T2 and T1 to T3.

Occam is the programming language used in the Transputer Development System (TDS). It was designed to provide both high- and low-level transputer facilities, and allows development of concurrent programs and distributed systems. The TDS was developed to support transputer networks in Occam. The TDS comprises integrated editor, file manager, compiler and debugging system. The TDS allows Occam programs to be written, compiled and then run from within it. Programs may also be configured to run on a target network of transputers. The TDS includes an interactive programming environment, compilation utilities and other programming tools, a number of libraries to support program development, and an extensive set of examples in source form. We will concentrate on examining the C-M algorithm using one transputer to one transputer physical links rather than logical links using the multiplexer, since we are examining the C-M algorithm and not the transputer itself. As we have seen in the previous Chapter the C-M algorithm may be affected by the number of branches. So to easily observe the `fork` and the `merge` it should be better to put the branches all together. Therefore, we will map some processes on a transputer keeping one input and one output physical links.

4.4 Experiment I

4.4.1 Topologies

We have a transputer board on which all the processor links are fixed, so the number of different topologies is somewhat limited. We use five node types: `source`, `sink`, `server`, `fork`, and `merge`. A `source` generates various messages except the null message. A `sink` accepts all incoming messages and does not send them anywhere. It does not make null messages. Therefore, the `source` and `sink` never cause deadlock in the simulation. A `fork` accepts messages from a single input and distributes messages across N outputs. Upon receiving a real or null message a `fork` transmits messages to the selected output node and creates N-1 null messages, each with the same time stamp as the message processed. One null message is transmitted to each destination node not selected. A `merge` accepts messages on N inputs and transmits them in chronological order to a single output. A `server` accepts messages on a single input and sends them to a single output. When the time of last message arrival is greater than the time of last message departure, and the nodes `fork`, `merge`, and `server` have no real messages to process, they produce a null message with a time stamp equal to the minimum time of the next message departure.

We have made three topologies: tandem network with 4 nodes, forked network, and fork and merged network. The tandem network shown in Figure 4.3 includes a `source`, `sink` and `servers`.

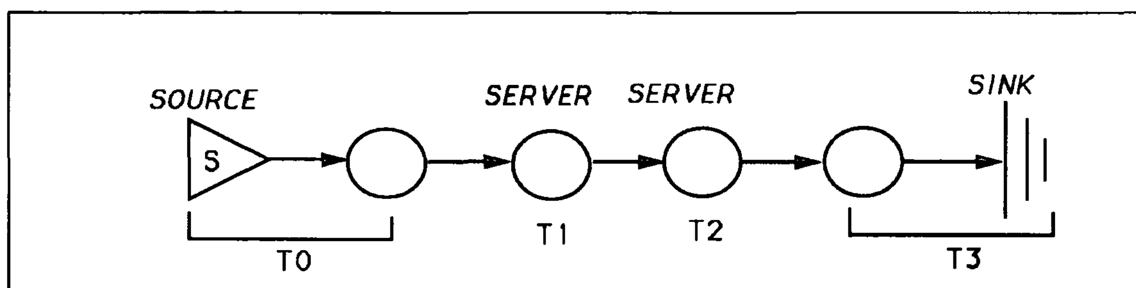

Figure 4.3:

The host transputer (T0), running the TDS, acts as a `source`, two transputers (T1 and T2) in the network act as `servers`, and another one (T3) in the network acts as a

sink.

The forked network shown in Figure 4.4 consists of a **source**, **fork** and two **servers**. T0 acts as a **source**, T1 acts as a **fork**, and T2 and T3 are attached to the branches of

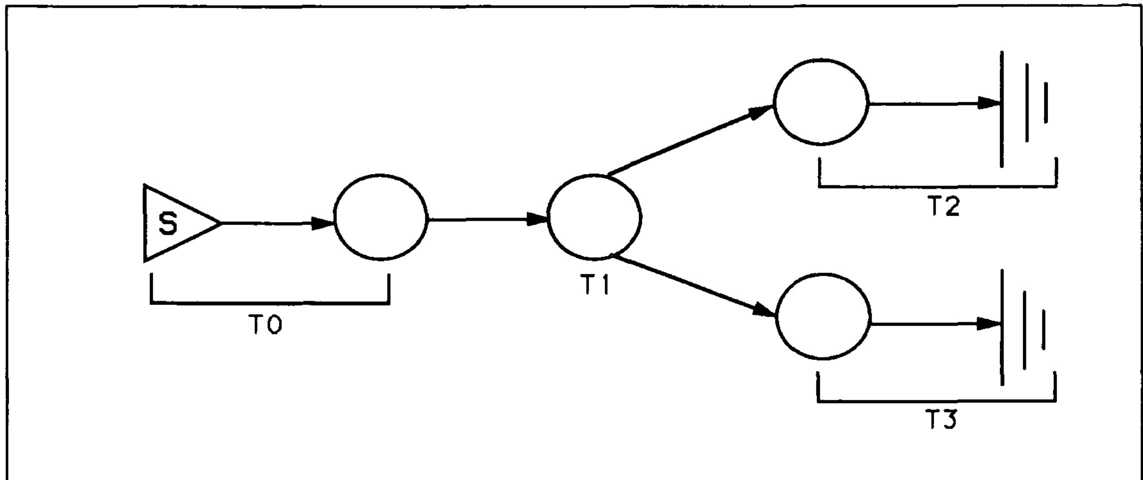

Figure 4.4:

the **fork** and act as both a **server** and **sink**, respectively.

The fork and merged network shown in Figure 4.5 consists of a **source**, **fork**, **merge**, **sink** and **servers**. T0 acts as both a **source** and **fork**, T1 and T3 act as **servers**,

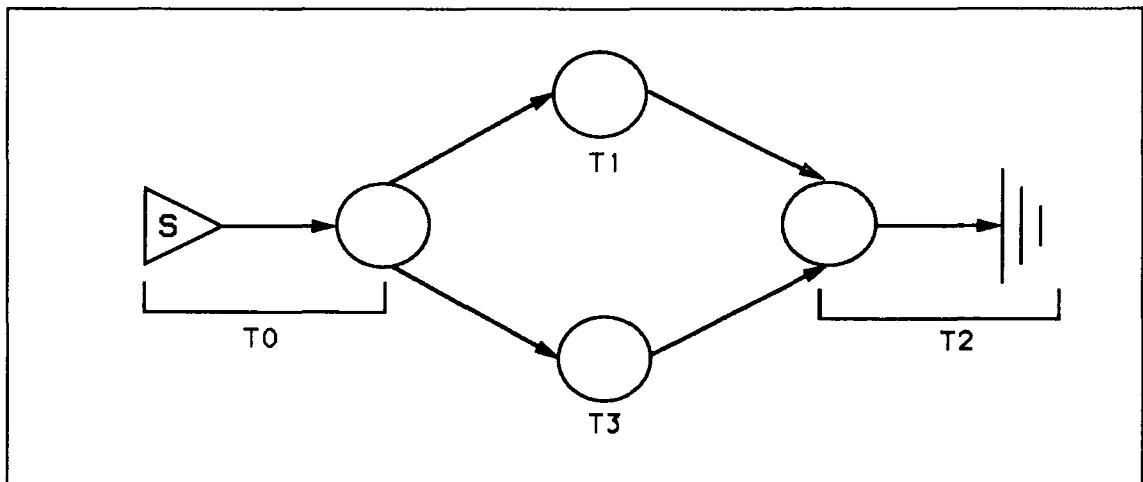

Figure 4.5:

respectively, and T2 acts as both a **sink** and **merge**.

4.4.2 Input and Output

First of all, input and output are considered. Messages are input from the keyboard and output on the screen. Whenever the user inputs the message, it is received by T0. T0 works as the interface between the host machine and the transputer network, and acts as a node, in parallel. Therefore, T0 does not miss any input message. The processed messages on each processor appear on the screen, so that the user can make sure how each processor works. In the C-M algorithm, the null message could be sent to announce absence of the real message. In order to decide when creation of null messages should take place we use the clock, TIMER provided in Occam. A node waits for the next message for 6 seconds, if there is no message it produces the null message. Each transputer works with its own clock value and all are working in parallel. Every transputer receives the message, processes and sends it to the destination node(s). In Occam, it is not allowed to use the screen in parallel, and in FAST4 only T0 can access to the screen. Therefore, all results must be sent back to T0. Hence, the result is sent both to the destination node and to T0. This restriction causes an inconvenience to the keyboard input. While the user types at the keyboard he/she cannot see the message until the terminator (<RETURN>) is sent, since there may be the result from any node to the screen. Hence, T0 controls the keyboard and screen, and processes messages in parallel. The remaining transputers in the network act as nodes and send results to T0.

4.4.3 Termination

It is necessary to decide when simulation terminates. When deadlock occurs, simulation is 'hung up' but does not terminate. Consider the example in Figure 4.1. Suppose that the `source` does not generate messages any more, simulation is finished. Although, P4 still has the message $(8, m_1)$ to send, an extra message with a t component exceeding 8 must be sent to 'flush out' this message. We will make this extra message (INFINIT, Z). When the message Z is sent from the `source` all nodes realise that the simulation has terminated. Every node then outputs its remaining messages. For correct output, INFINIT must be greater than any input message. Furthermore, the message (INFINIT, Z) must be the last output along all lines and after sending (INFINIT, Z) the clock value

on the every line is INFINIT.

4.5 Testing

We first examined the tandem network. The link topology of transputers' is shown in Figure 4.6, light arrows indicating the message flow and bold arrows indicating results sent back to T0.

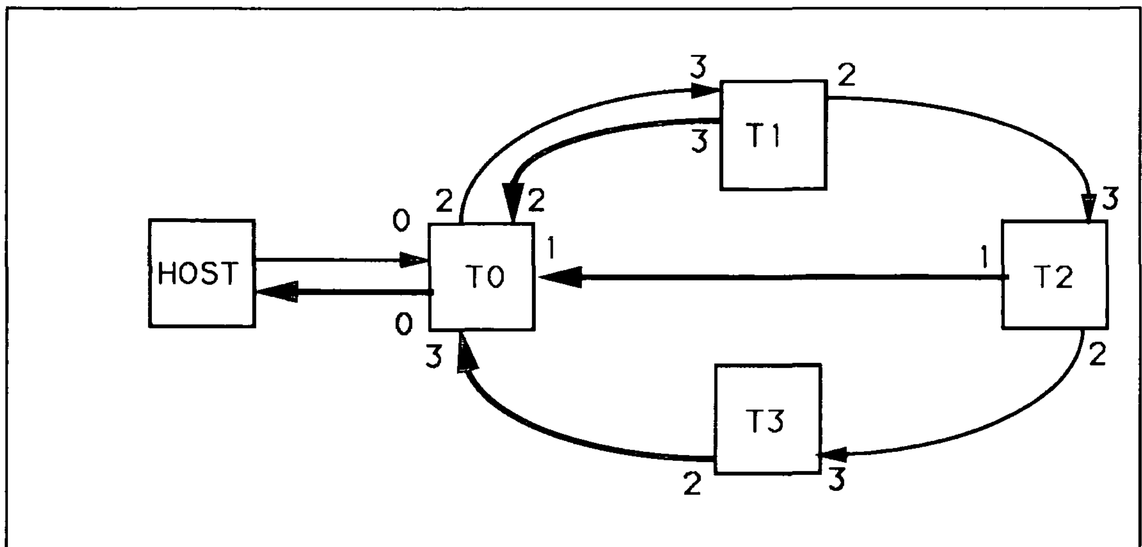

Figure 4.6:

We tested changing each node's service time and used real and null messages. The system did not deadlock and it was not necessary to consider the link time since it did not have a **fork** and **merge**. This particular topology is probably the best possible configuration for the distributed approach to be successful.

The next test included a **fork** and was carried out in the same way as the tandem network. The link topology is shown in Figure 4.7. In this topology, deadlock did not occur, and the link time algorithm was used at the **fork** node, which sent real messages alternately.

The last link topology example is shown in Figure 4.8, including a **fork** and **merge**.

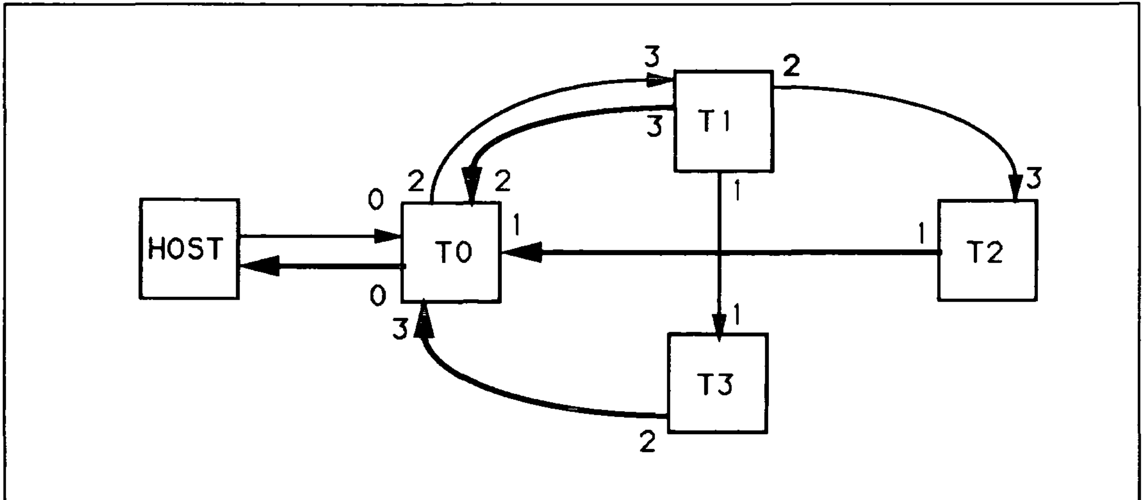

Figure 4.7:

In this topology the link time is fully used at the `fork`, and the `merge` processed messages

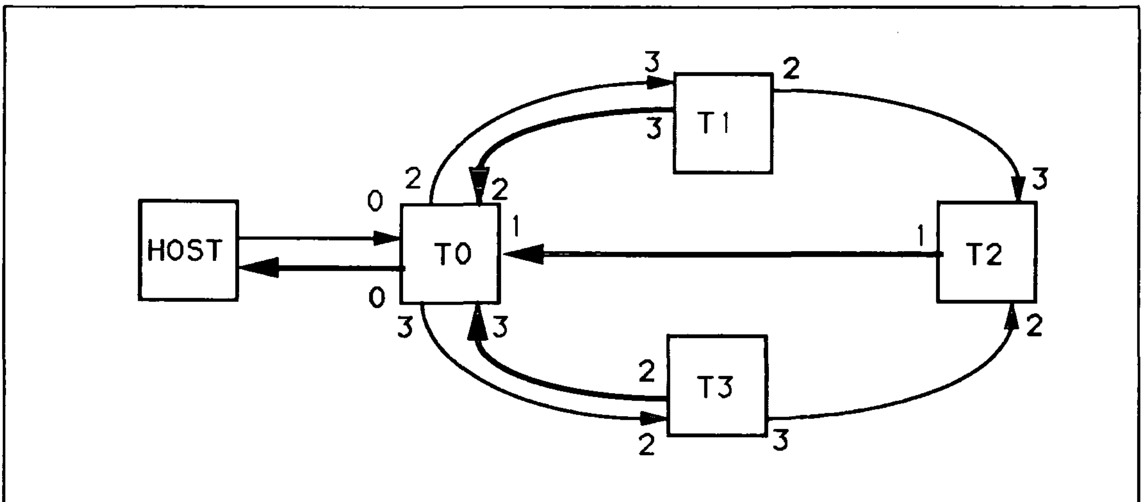

Figure 4.8:

in chronological order. It was supposed that the C-M algorithm could be implemented using a bounded buffer, even with buffer size 0 (see [Chandy 81]), so we did not use any buffer. Therefore, messages are transmitted synchronously, for instance, when T0 has a message to send to T1 and T1 is not ready to receive the message from T0, T0 has to wait until T1 is ready. In our last example with buffer size 0 and synchronisation, the `merge` node was blocked, that is, deadlock occurred. For simplicity, we suppose that the `fork` and `merge` do not have the service time (st0 = st2 = 0), so T0 receives the input message from the keyboard and simply branches them, and T2 receives messages from both T1 and

T3, in chronological order and does not send them to any node. Table 4.1 shows message transmissions in the simulation of the last example and deadlock, where each horizontal row is a time slice and each entry corresponds to a single activity of one of the processes.

It is evident that several activities may happen concurrently. Suppose that stl is 2, st3 is 10, and input messages are shown in Table 4.1. Messages marked with a † are those received and messages marked with a * are those waiting to be sent. Deadlock is caused by T2, since T2 cannot receive the message (33, m_4) from T3, T3 cannot receive the message (27, null) from T0, and T0 cannot receive the message (35, m_6) from the keyboard. In this example, deadlock occurs when all links are in the situation where a message is on the output line and the destination node also has a message to be output on the output line. We also observed the overhead of null messages. In Table 4.1 the `sink` receives 5 null messages and 4 real messages until deadlock occurs. A similar example, Table 4.2, shows the `sink` receiving 4 null messages and 4 real messages until deadlock.

Most of the null messages are created at `fork`. The `fork` creates a null message for a node whenever it sends a real message to another node. Therefore, the `merge` processes about twice the number of messages that the `source` produces. It seems that the number of null messages created is caused by the number of `forks` and by the number of branches on the `fork`. In [Chandy 81] the authors say that if there is a feedback path from the output of T2 to the input of T0, a large number of 'null job' will be created at T0 for every 'real job' entering T0. Every message entering T0 will cause a null message to be sent along at least one of the two outgoing edges although there is no mechanism to annihilate the null job.

step	key	T0			T1(2)		T3(10)		T2		
	send	rec key	send T1	send T3	rec T0	send T2	rec T0	send T2	rec T1	rec T3	send sink
1						2, null†		10, null†			
2	2, M_1	2, M_1							2, null†	10, null†	
3	9, M_2		2, M_1†	2, null†							2, null
4		9, M_2			2, M_1		2, null				
5			9, null†	9, M_2†		4, M_1†		12, null†			
6	20, M_3				9, null				4, M_1†		
7						11, null†					4, M_1
8									11, null†		
9		20, M_3					9, M_2			12, null†	10, null
10	23, M_4		20, M_3†	20, null†				19, M_2†			11, null
11					20, M_3						
12						22, M_3†					
13									22, M_3†		
14		23, M_4					20, null			19, M_2†	12, null
15	27, M_5		23, null†	23, M_4†				30, null†			19, M_2
16		27, M_5			23, null		23, M_4			30, null*	
17	35, M_6*		27, M_5†	27, null*		25, null†		33, M_4*			22, M_3
18					27, M_5				25, null†		
19						29, M_5†					25, null
20									29, M_5†		
21											29, M_5

Table 4.1:

step	key	T0			T1(2)		T3(10)		T2		
	send	rec key	send T1	send T3	rec T0	send T2	rec T0	send T2	rec T1	rec T3	send sink
1						2, null†		10, null†			
2	7, M_1	7, M_1							2, null†	10, null†	
3	11, M_2		7, M_1†	7, null†							2, null
4		11, M_2			7, M_1		7, null				
5			11, null†	11, M_2†		9, M_1 †		17, null†			
6	15, M_3				11, null				9, M_1†		
7						13, null†					9, M_1
8									13, null†		
9	19, M_4	15, M_3					11, M_2			17, null†	10, null
10			15, M_3†	15, null†				21, M_2†			13, null
11					15, M_3						
12						17, M_3†					
13	23, M_5								17, M_3†		
14		19, M_4					15, null			21, M_2†	17, M_3
15			19, null†	19, M_4†				25, null†			
16					19, null						
17	27, M_6					21, null†					
18									21, null†		
19		23, M_5					19, M_4			25, null†	21, M_2
20			23, M_5†	23, null†				31, M_4†			
21					23, M_5						
22	31, M_7*					25, M_5†					
23									25, M_5†		
24		27, M_6					23, null			31, M_4*	25, M_5
25			27, null†	27, M_6*				33, null*			
26					27, null						
27						29, null†					
28									29, null†		
29											29, null

Table 4.2:

4.6 Deadlock Avoidance

In previous Sections we performed experiments of deadlock avoidance using the C-M algorithm. In spite of this, deadlock still occurred. In this Section we will study deadlock avoidance once more. We made two hypotheses to avoid deadlock: one was that we may need buffers; the other was that we may have to control input messages. We used three methods for the treatment of time to observe the simulation more generally, and examined the fork and merged network. First, the `fork` checks the link time and sends the real message to the smaller link time's node used in the previous Section. Secondly, the `fork` sends all real messages to the node having the smaller service time. Finally, the `fork` sends all real messages to the node having the greater service time.

4.6.1 Deadlock Avoidance by Input Messages

We first examined deadlock avoidance by messages shown in Figure 4.9.

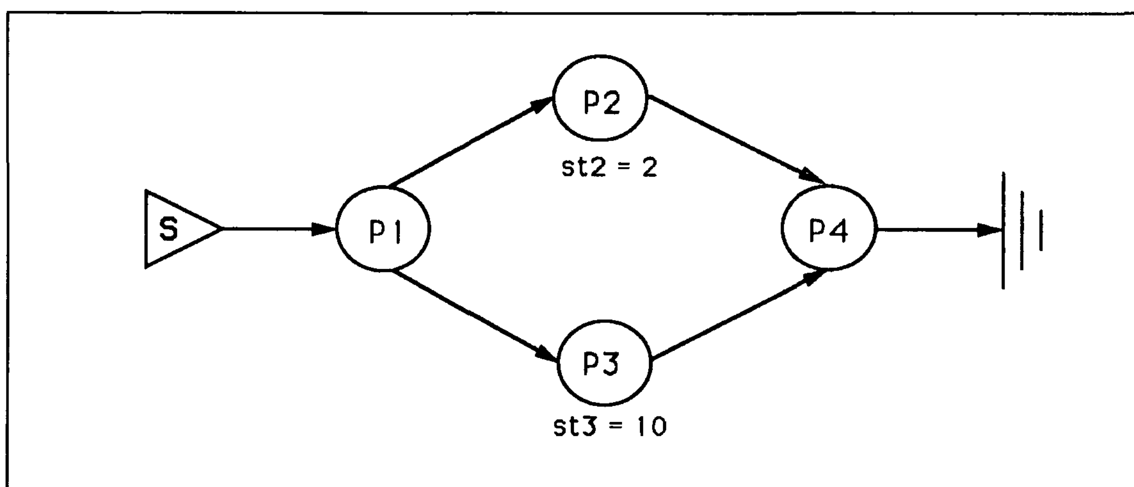

Figure 4.9:

We did not use buffers, instead we controlled input messages and made the `merge` receive all messages from each branch node, so that deadlock may be avoided. For simplicity, we supposed that the `fork` (P1) and `merge` (P4) did not have the service time (st1 = st4 = 0), st2 was 2 and st3 was 10. Deadlock seemed to occur because of the time gap

(gt) of messages and each node's service time. We examined the following three gts:

$$gt_1 > \max(\text{st2}, \text{st3}) : gt_1 = 11$$
$$gt_2 = \max(\text{st2}, \text{st3}) : gt_2 = 10$$
$$gt_3 < |st2 - st3| : gt_3 = 9$$

We did not set the start value. For example, messages may be generated (3, m_1), (14, m_2), (25 m_3) ... etc. for gt_1. In this experiment we only changed the `fork` program leaving others the same. We first examined all real messages that were received by the node (P2) having the smaller service time and all null messages that were received by the node (P3) having the greater service time. The second experiment was to use link time so that real messages were received alternately. In the first and second experiments deadlock did not occur for all gts. We finally examined all real messages that were received by P3 and all null messages that were received by P2. In the last experiment gt_1 and gt_2 were successful but deadlock occurred at gt_3.

That is the first of two cases which have the smaller likelihood of deadlock. The reason for deadlock is the different treatment of null messages from that of real messages. When a node receives a null message its local clock advances and the node sends a null message with that advanced time. However, if the node receives only null messages the output time is different to if it had processed a real message, since a null message does not change the output link time. In the last example of gt_3, P2 never waits to output a message but P3 waits for all output. P3 receives (3, m_1) and sends (13, m_1) (13 = arrival time + service time). When it receives the next message (12, m_2) it sends (23, m_2) but not (22, m_2) because the real message changes the last departure time i.e. 23 = 13 + 10. Therefore, some extra waiting time may be accumulated upon receiving the real message. In addition P3 has a greater service time so the `merge` could receive the smaller time component of the message from P2. This causes deadlock. Since there is no buffer, messages to be output are stored in variables, and each process has one variable to output. Therefore, if the number of `server` nodes between the `fork` and the `merge` (one in this example) is greater, deadlock could occur later. To avoid deadlock the input message should be greater than the maximum service time in the branches. Then no node accumulates waiting time. Although, this simulation is too restricted, the time component

of messages must be increased monotonically. However, it should not be increased as much as the maximum service time, and if the time gap is always greater than the maximum service time, one of the processors in the branches will be always idle. Therefore, this simulation will be expensive. We should think about other ways to avoid deadlock. Let us examine how to use a buffer so that all messages are transmitted immediately.

4.6.2 Deadlock Avoidance by Buffers

We put buffers into the system so that when a message is sent it is received immediately. Buffers may be put into either the `fork` or the `merge`. If put into the `fork`, any message from the `fork` will be placed into the buffer which will wait until the destination node is ready. If put into the `merge`, any message sent there will be stored in the buffer and not transmitted until the `merge` is ready. We decided to put buffers into the `merge` since when the system does not have a `merge` (for example in a forked network) their case is avoided. Hence, we altered the `merge` program.

The altered `merge` consists of three processes: receiver, event queue, and simulator. The receiver receives messages from all input links which are attached to the `merge`, and sends them to the event queue. The event queue has two processes: one is for the receiver, the other is for the simulator. Each input link has its own event queue. The event queue maintains all queues which are on a First-Come First-Served basis. It receives messages from the receiver, adds messages on the queue according to their input links, and increments the pointer which indicates the next space on the queue. It also communicates with the simulator, sends the message when the simulator requires, decrements the pointer, and shifts all messages on the queue forward. The event queue does not allow the simulator and the receiver access to the same queue at the same time, since we do not know whether it is after incrementing the pointer, or while adding a message (in other words, before incrementing the pointer), that the simulator accesses. In order to make sure we use the `ALT` construction. The simulator performs like the previous system. The difference is that it first sends the request signal to the event queue when it is ready, gets a message from the event queue, and processes it. All processes are running in parallel. In the program, the receiver has two `SEQ` constructions in parallel, each of which is used for a forthcoming

link. The event queue has two ALT constructions in parallel, and the simulator inputs, processes, and outputs sequentially. We examined deadlock avoidance with buffers by various messages using a smaller time gap than the maximum service time. Deadlock did not occur. When the last message (INFINIT, Z) is received, it flushes out all messages on the queue. Therefore, when simulation terminates the queue is empty.

It is important to decide on the number of buffer spaces because on the system memory is usually bounded. The system may have as many buffer spaces as the number of input messages. The null message (t, null) means that there is no message before time t, so the null message can be annihilated in the following rule. Any messages (both real and null) put in the buffer after a null message annihilate any null messages ahead of it still in the buffer, since messages must have greater time components. The null message does not affect the number of buffer spaces. Therefore, the number of buffer spaces is the number of real messages plus one (where the one is used to store the null message until it is annihilated). It was found in our system that if the `fork` has two branch nodes, the one having the greater service time needs the buffer whilst the other does not. When the number of buffer spaces is reduced to one, the event queue acts as a pipeline so that the messages are added on the queue and removed immediately. In fact the `merge` never reads messages continually from the greater service time node. When the number of buffer spaces is reduced to zero, processors communicate synchronously, that is the sender has to wait until the receiver is ready. This implies that the sender may spend a considerable amount of time in waiting. On the other hand, it was found in [Misra 86] that in the simulation of a certain class of queuing networks, performance improved rapidly until the number of buffer spaces on a channel approached ten, increased less rapidly until twenty, and remained essentially unchanged thereafter. These numbers may depend on the type of problem and the speeds of processors and communication lines.

Null messages are also annihilated at the `merge`. For example, when the `merge` receives messages (30, null) and (30, m_k), the `merge` does not have to process (30, null), but only process (30, m_k). The annihilation of null messages on the `merge` and the buffer depends on the system, so it is difficult to tell how many null messages can be annihilated. However, it may be true that the number of null messages transmitted to the `sink`, and the number of annihilations, is smaller than those created.

4.7 Experiment II

4.7.1 Input and Output

In this experiment, the keyboard input and screen output is discarded. Instead we used input and output from a file. Two distinct classes of file may be accessed on the TDS. Fold files which are part of the fold structure of the development system, and host files which are not. The TDS provides various user file interfaces according to whether the program is running inside the TDS, loaded directly by the TDS server or loaded by another server, either written by the user or, for example, the host file server supplied with the TDS. Files to be read may be created as folds using the editor, or by another program. New files may be created within the bundle and written into. Such files are readable by the user. All access to filed folds is sequential, and procedures are designed to facilitate the reading of existing files as if they were a `source` of characters like a keyboard, and the writing of new files as if they were a simple screen or printer.

When the file is used the results are saved so all messages are checked afterwards to see if they are correct. All messages are read continually so null messages are made once at the beginning at all nodes and only the `fork` continues to make null messages during simulation. The time-out to make a null message is still used and although messages are read before time-out (every six seconds), it does not make null messages. To measure the process time ticks are used. The result of the ticks is also saved to the file.

4.7.2 Experiments

Experiments were carried on using fork and merged network as in the previous Section. As before, the `fork` and the `merge` do not have any service time. The service time on one `server` is 2, the other 10. However, one more branch is added so that the `fork` has 3 `servers`, shown in Figure 4.10, and the new node's service time is 5.

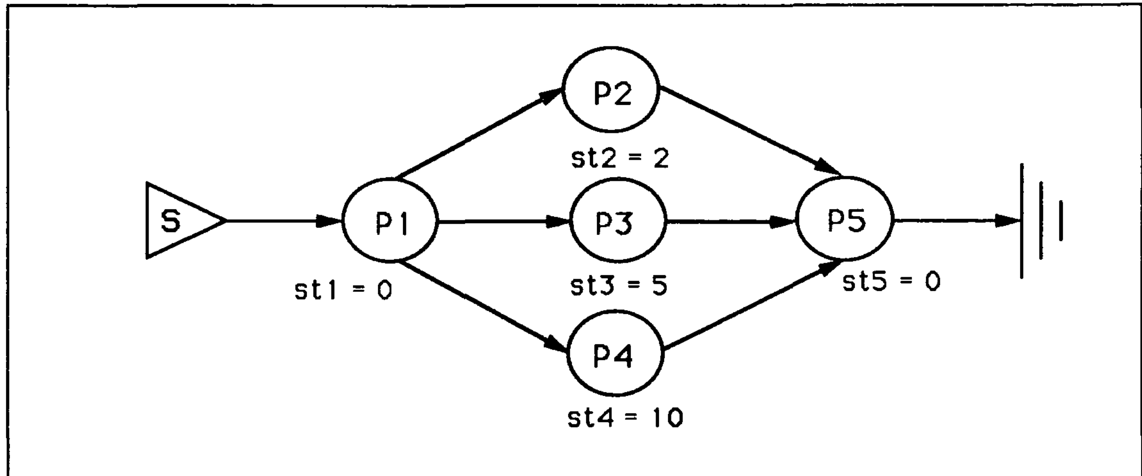

Figure 4.10:

This topology is mapped onto one transputer. To make a good comparison between 2 **servers** and 3 **servers**, we have transferred the 2 **servers**' program onto one transputer. The difference in programming between using a network of transputers and one transputer is that on the network hardware channels are used but on one transputer all channels are software channels. The software channels may be declared into an array and implemented by loops, but the hardware channels may not. In this experiment, in order to make an accurate comparison, we did not declare software channels in the array. Therefore, we have examined 3 topologies: two **servers** in a network, two **servers** and three **servers** on a transputer.

The program with three **servers** on one transputer is based on that of the two **servers** on one transputer. But we had to put more parameters and loops into some functions. It could be difficult to implement the three **servers** network on our transputer board because its hardware links are fixed, and one processor would have to have two nodes. To maintain the load balance, we must consider the structure of the system, the service time, and data flow. When the load balance is not maintained, the parallelism may be fulfilled less efficiently, may cause more null messages to be made and the process time may take longer. On the other hand, when all nodes are mapped onto one transputer the development time takes longer. The SC (a 'Separate Compilation' unit) may be used to develop the system on one transputer, although all SCs must be linked whenever the program changes. In the network, the programming of the host transputer is separated from that of the network. Therefore, when the program on the host transputer changes only the EXE

(an 'Executable' program) is compiled, and when the program on the network changes only the PROGRAM (a program intended to run on a network of transputers, including configuration information) is compiled.

All messages are first loaded into an array, processed, and saved into a file. After loading the file the tick is read, and before saving the tick is read again, the process time being measured by subtracting the first tick from the second tick. The loading and saving time depends on the empty spaces on the disk so it should not be included in the process time. The screen output may help the user observe how the system works during simulation although it makes an extra job in parallel and causes an extra interrupt. Therefore, it is not used in this experiment.

Three types of messages are used: gt_1 is constantly 9, gt_2 is random and smaller than the maximum service time, and gt_3 is random and greater than maximum service time. All input messages are shown in Table 4.3.

$gt_1 = 9$	$gt_2 <$ max st	$gt_3 \geq$ max st
3 a	5 a	1 a
12 b	7 b	11 b
21 c	13 c	23 c
30 d	18 d	35 d
39 e	21 e	45 e
48 f	25 f	56 f
57 g	30 g	70 g
66 h	33 h	81 h
75 i	35 i	93 i
84 j	36 j	105 j
93 k	41 k	116 k
102 l	47 l	127 l
111 m	52 m	140 m
120 n	57 n	150 n
129 o	60 o	162 o
138 p	66 p	173 p
147 q	73 q	188 q
156 r	77 r	199 r
165 s	80 s	210 s
174 t	84 t	220 t
INFINIT Z	INFINIT Z	INFINIT Z

Table 4.3:

We experimented on the following systems. On the network, the **fork** has two nodes and sends real messages; alternately to two nodes, all to the node having the smaller service time, and all to the node having the greater service time, both with and without buffers, that is six cases. On the transputer, with the two node **fork**, real messages are sent; alternately to two nodes, with and without buffers, all to the node which has the greater service time, and all to the node which has the smaller service time. The last two with buffers. Finally, on the transputer using the three node **fork**, real messages are sent; alternately to three nodes with and without buffers, all to the node which has the smaller service time, all to the node which has the middle service time, and all to the node which has the greater service time. The last three all using buffers. Therefore, there are nine cases on one transputer.

4.7.3 Results

The results are shown in Table 4.4 and Table 4.5.

NETWORK	gt_1		gt_2		gt_3	
	null	ticks	null	ticks	null	ticks
alter, no buf, 2, 10	23	123	DEADLOCK		23	122
small, no buf, 2, 10	23	123	DEADLOCK		23	122
great, no buf, 10, 2	DEADLOCK		DEADLOCK		23	116
alter, with buf, 2, 10	22	122	10	116	22	123
small, with buf, 2, 10	21	121	8	107	22	121
great, with buf, 10, 2	20	127	19	146	22	124

Table 4.4:

As before deadlock occurs when the system does not have a buffer, and input messages have a time gap smaller than the branch node's maximum service time. Therefore on the two **server** system, when no buffer with gt_2, deadlock will occur, and when the **fork** sends all real messages to the node having the greater service time with gt_1, deadlock will occur. When the system has buffers, deadlock may not occur unless the buffer space is too small.

ONE TRANSPUTER	gt_1		gt_2		gt_3	
	null	ticks	null	ticks	null	ticks
alter, no buf, 2, 10	22	224	DEADLOCK		22	224
alter, with buf, 2, 10	12	266	10	265	12	283
small, with buf, 2, 10	9	257	6	251	9	257
great, with buf, 10, 2	20	298	19	307	22	298
alter, no buf, 2, 5, 10	42	339	DEADLOCK		43	341
alter, with buf, 2, 5, 10	15	379	13	374	16	378
small, with buf, 2, 5, 10	4	378	5	379	5	379
middle, with buf, 5, 2, 10	5	382	5	382	6	383
great, with buf, 10, 2, 5	5	383	11	397	6	384

Table 4.5:

It seems that the buffered system has less chance to send null messages as they may be annihilated in the buffer. However, on the system in which the **fork** has two **servers** and sends all real messages to the node having the greater service time, the number of null messages can not be annihilated. Since all null messages are sent to the node having the smaller service time and this node has the size of one buffer space, most of the null messages are sent immediately after having been stored in the buffer. On the system in which the **fork** has three **servers**, all nodes have the size of ten buffer spaces, so a greater annihilation of null messages occurs. When one more branch is added on the unbuffered system, the number of null messages is increased to nearly as many as the number of input messages. It implies that the **fork** handles most of the null messages, and that the number of branches affects the overhead of null messages with the result that the process time is increased.

When the network of transputers is used, four transputers work in parallel. When all processes are mapped onto one transputer, all processes (in other words, **fork**, **merge**, **server**, main program inputting results assigning arrays) are working in the **PAR** construction. It would be expected that those two programs work exactly the same, although, the one transputer version takes more than twice the processing time. It may imply that when the transputer has many **PAR** constructions it does not work in parallel for all, and some inefficient parallel process causes the longer processing time. For example, when the **fork** alternately sends real messages to **servers** with buffers, on one transputer, the **merge** program has three **PAR** constructions; receiver, event queue, and simulation. In the event queue, the receiver has high priority access to the event queue, and while the receiver is

accessing the simulator is not allowed to access to the event queue. For this the ALT construction is used. The ALT performs the process associated with a guard which is ready. It is a first-past-the-post race between a guard of channels, with only the winner's process being executed. We expected that the receiver would access the event queue first and add one message. Then the simulator would alternately access the event queue and remove a message. It seems, however, that on the one transputer system the receiver and the event queue are working in parallel but the simulator is not, so that the simulator can access the event queue when the receiver finishes storing messages on the buffer. Therefore, the buffer is often filled with messages and null messages could be annihilated further. In fact, on the one transputer system, the processing time is longer but some null messages are annihilated in the buffer.

4.8 Summary

In this Chapter we have observed deadlock, the process time and the null message. Deadlock is avoided either by the control of input messages or by using buffers. The process time may be shorter when each processor has less work to do. The number of null messages is increased by the number of branches in the `fork`, and some null messages may be annihilated in the buffer and at `merge`. In addition all these results are related to each other. For example, on the buffered one transputer system, the processing time is longer but many null messages are annihilated in the buffer. In the network of transputers the system works a good deal faster but there is an overhead of null messages. The best implementation may depend on the system structure, the service time on each node, input messages, the size of buffer spaces, data flow, the number of processors, and the like. In our experiments, the buffered network of transputers system uses input messages. These are random and have a time gap smaller than the maximum service time. The result is that the system works a great deal faster and has little overhead of null messages.

Chapter 5

CONCLUSIONS

The design, planning, improvement, and operation of complex systems are each very difficult to perform since they are based on unrealistic assumptions and require many approximations. Hence, simulation is the viable means for obtaining accurate measurements of performance for very complicated systems. Currently, however, simulations are extremely slow. Distributed simulation seems to be a promising approach for speeding up simulations, although more work needs to be done to determine the extent of its promise. The advantage of distributed simulation will depend on the system being simulated, the available multiprocessor system, and the approach used. The type of application is also an important consideration in determining an appropriate approach to distributed simulation. For instance, time-driven simulation may be better than event-driven simulation if there are many interactions at the same time, or if the simulation is to be combined with on-line graphics. Synchronous simulation may be more appropriate than asynchronous simulation for more tightly coupled systems in which global information is used. Moreover if the system is very tightly coupled, distributing the model will look less attractive relative to other decomposition approaches. The architecture of the processing system, in particular the number of processors, the amount of memory, and whether the memory is shared or not, will also determine the approach to be used for distributed simulation.

We have studied the conservative approach and examined its viability in distributed simulation. This approach requires less memory than the optimistic approach, but when more memory is available for buffering messages, more null messages could be annihilated. In the conservative approach the simulation processes the events in time stamped order. Suppose that the simulation is distributed over several processors, it becomes possible for a processor to process an event which is not the earliest. Also, in processing this event the simulation may affect conditions for earlier, as yet un-simulated events. Thus the future is affecting the past. This is known as a *causality* error. The conservative approach avoids causality errors. Therefore, the implementation of the conservative approach may be simpler and easier. The assignment of processes to processors may degrade simulation performance. If the load is unbalanced some processors will have so much work to be done that the amount of parallelism would be decreased and the simulation time would be increased. Even if there is some information, the assignment problem would be complex, and the relative load would change during the simulation. To summarise, the conservative approach can achieve good performance with the system which has a rather simple system structure and with a certain amount of memory for buffering. Transputers work a good deal faster so they are suitable for implementing distributed simulation.

Many approaches are proposed in distributed simulation, although very little empirical work has been done. Empirical and analytical studies need to determine the relative advantages of different approaches and which are best under what circumstances. More tools are required to empirically measure the performance of a simulation. For example, the TDS which is used to develop a program using Occam is somewhat difficult to operate and it does not have utilities such as graphics. Neither the conservative nor the optimistic approach has yet adequately addressed real-time applications. There are some issues to be solved if distributed simulation is to be a viable alternative to uniprocessor simulation, but the approach is promising and impressive.

Bibliography

[Adkins 77] G. Adkins and W. Pooch, "Computer Simulation: a Tutorial", Computer, pp.12-17, April 1977

[Bagrodia 87] R. J. Bagrodia, K. M. Chandy, and J. Misra, "A Message-Based Approach to Discrete-Event Simulation", IEEE Transactions on Software Engineering, Vol. Se-13, No. 6, pp.654-665, June 1987

[Bain 88] W. L. Bain and D. S. Scott, "An Algorithm for Time Synchronization in Distributed Discrete Event Simulation", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.30-33, 1988

[Bal 89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems", Computing Surveys, Vol. 21, No. 3, pp.261-322, September 1989

[Brennan 88] A. M. Brennan, "The Use of a Multi-Transputer System and the Investigation of a Load-Balancing Harness", M.Sc Thesis, University of Durham, 1988

[Burns 88] A. Burns, "Programming in Occam 2", Addison-Wesley Publishing Company, 1988

[Chandy 79a] K. M. Chandy, V. Holmes, and J. Misra, "Distributed Simulation of Networks", Computer Networks 3, pp.105-113, 1979

[Chandy 79b] 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, pp.440-452, September 1979

[Chandy 81] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via Sequence of Parallel Computations", Communications of the ACM, Vol. 24, No. 11, pp.198-206, April 1981

[Chandy 83] K. M. Chandy and J. Misra, "Distributed Deadlock Detection", ACM Transactions on Computer Systems, Vol. 1, No. 2, pp.144-156, 1983

[Comfort 84] J. C. Comfort, "The Simulation of a Master-Slave Event Set Processor", Simulation, Vol. 42, No. 3, pp.117-124, March 1984

[Comfort 88] J. C. Comfort, "Environment Partitioned Distributed Simulation with Transputers", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.103-108, 1988

[Davis 90] N. J. Davis IV, D. L. Mannix, W. H. Shaw, and T. C. Hartrum, "Distributed Discrete-Event Simulation Using Null Message Algorithms on Hypercube Architectures", Journal of Parallel and Distributed Computing 8, pp.349-357, 1990

[Dowsing 88] R. D. Dowsing, "Introduction to Concurrency Using Occam", T. J. Press Ltd, 1988

[Franta 77] W. R. Franta, "The Process View of Simulation", North-Holland, 1977

[Fujimoto 88] R. Fujimoto, "Performance Measurements of Distributed Simulation Strategies", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.14-20, 1988

[Fujimoto 90] R. Fujimoto, "Parallel Discrete Event Simulation", Communications of the ACM, Vol. 33, No. 10, pp.31-53, October 1990

[Galletly 90] J. Galletly, "Occam 2", Pitman Publishing, 1990

[Gilmer 88] J. B. Gilmer Jr, "An Assessment of 'Time Warp' Parallel Discrete Event Simulation Algorithm Performance", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.45-49, 1988

[Gofni 88] A. Gofni, "Rollback Mechanism for Optimistic Distributed Simulation Systems", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.61-67, 1988

[Gordon 78] Geoffrey Gordon, "System simulation", Prentice-Hall, 1978

[Graybeal 80] Wayne T. Graybeal and Udo. W. Pooch, "Simulation: Principles and methods", Little Brown and Company, 1980

[Groselj 88] B. Groselj and C. Tropper, "The Time-of-next-event Algorithm", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.25-29, 1988

[Heidelberger 86] P. Heidelberger, "Statistical Analysis of Parallel Simulations", in Proceedings of the 1986 Winter Simulation Conference, pp.290-295, 1986

[Hind 90] A. Hind, "Parallel Processing Architectures for Performance Engineering of Telecommunication Networks", Research report, University of Durham, September 1990

[Hoare 78] C. A. R. Hoare, "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, pp.666-677, August 1978

[Homewood 87] M. Homewood, D. May, D. Shepherd, and R. Shepherd, "The IMS T800 Transputer", MICRO, pp.10-26, October 1987

[Hull 89] M. E. C. Hull and A. Zarea-Aliabadi, "Real-Time System Implementation -The Transputer and Occam Alternative", Microprocessing and microprogramming 26, pp.77-84, 1989

[Inmos 88a] Inmos, "Transputer Development System", Prentice Hall, 1988

[Inmos 88b] Inmos, "Occam 2 Reference Manual", Prentice Hall, 1988

[Inmos 89] Inmos, "The Transputer Databook", Redwood Burn, 1989

[Jefferson 85] D. R. Jefferson, "Virtual Time", ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pp.404-425, July 1985

[Jones 86] D. W. Jones, "Concurrent Simulation: An Alternative to Distributed Simulation", in Proceedings of the 1986 Winter Simulation Conference, pp.417-423, 1986

[Jones 88] G. Jones and M. Goldsmith, "Programming In Occam 2", Prentice Hall, 1988

[Kaudel 87] F. J. Kaudel, "A Literature Survey on Distributed Discrete Event Simulation", Simuletter, Vol. 18(2), pp.11-21, June 1987

[Kendall 53] D. G. Kendall, "Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of Imbedded Markov Chains", *Ann. Math. Statistics* 24, pp.338-354, 1953

[Kleinrock 85] L. Kleinrock, "Distributed Systems", Communications of the ACM, Vol. 28, No. 11, pp.1200-1213, November 1985

[Krishnamurthi 85] M. Krishnamurthi, U. Chandrasekaran, and S. Sheppard, "Two Approaches to the Implementation of a Distributed Simulation System", in Proceedings of the 1985 Winter Simulation Conference, pp.435-444, 1985

[Kumar 78] B. Kumar and E. S. Davidson, "Performance Evaluation of Highly Concurrent Computers by Deterministic Simulation", Communications of the ACM, Vol. 21, No. 11, pp.904-913, November 1978

[Kumar 86] D. Kumar, "A Novel Approach to Sequential Simulation", IEEE Software, pp.25-33, September 1986

[Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol. 21, No. 7, pp.558-565, July 1978

[Law 82] Averill M. Law and W. David Kelton, "Simulation modeling and analysis", McGraw-Hill, 1982

[Lomow 88] G. Lomow, J. Cleary, B. Unger, and D. West, "A Performance Study of Time Warp", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.50-55, 1988

[Maisel 72] Herbert Maisel and Giuliano Gnugnoli, "Simulation of Discrete Stochastic Systems", Science Research Associates, 1972

[Misra 83] J. Misra, "Detecting Termination of Distributed Computations Using Markers", in Proceedings of the 2nd ACM Conference on Principles of Distributed Computing, pp.290-293, 1983

[Misra 86] J. Misra, "Distributed Discrete-Event Simulation", Computing Surveys, Vol. 18, No. 1, pp.39-65, 1986

[Mitra 84] D. Mitra and I. Mitrani, "Analysis and Optimum Performance of two Message-Passing Parallel Processors Synchronized by Rollback", in Proceedings of the Computer Performance Modeling 10th International Symposium, pp.35-50, 1984

[Nance 81] R. E. Nance, "The Time and State Relationships in Simulation Modeling", Communications of the ACM, Vol. 24, No. 4, pp.173-179, April 1981

[Naylor 66] Thomas H. Naylor, Joseph L. Balintfy, Donald S. Burdick, and Kong Chu, "Computer Simulation Techniques", John Wiley & Sons, 1966

[Nevison] C. Nevison, "Discrete Event Simulation Using Occam", pp.222-230

[Nicoud 89a] J. Nicoud and A. M. Tyrrell, "The Transputer T414 Instruction Set", MICRO, pp.60-75, January 1989

[Nicoud 89b] J. Nicoud and A. M. Tyrrell, "Scheduling and Parallel Operations on the Transputer", Microprocessing and microprogramming 26, pp.175-185, 1989

[Payne 82] James A. Payne, "Introduction to simulation", McGraw-Hill, 1982

[Peacock 79a] J. K. Peacock, J. W. Wong, and E. G. Manning, "Distributed Simulation Using a Network of Processors", Computer Networks 3, pp.44-56, 1979

[Peacock 79b] J. K. Peacock, J. W. Wong, and E. G. Manning, "A Distributed Approach to Queuing Network Simulation", in Proceedings of the Winter Simulation Conference pp.399-407, December 1979

[Perrott 88] R. H. Perrott, "Parallel Programming", Addison-Wesly Publishing Company, 1988

[Poole 77] T. G. Poole and J. Z. Szymankiewicz, "Using simulation to solve problems", McGraw-Hill, 1977

[Pountain 88] D. Pountain and D. May, "A Tutorial Introduction to Occam 2", BSP Professional Books, March 1988

[Reed 85] D. A. Reed, "Parallel discrete Event Simulation: A Case study", in Proceeding of the 18th IEEE Annual Simulation Symposium, pp.95-107, 1985

[Reed 88a] 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, pp.541-553, April 1988

[Reed 88b] D. A. Reed and A. D. Malony, "Parallel Discrete Event Simulation: The Chandy-Misra Approach", in Proceedings of the SCS Multiconference on Distributed Simulation, pp.8-13, 1988

[Righter 89] R. Righter and J. C. Walrand, "Distributed Simulation of Discrete Event Systems", in Proceedings of the IEEE, Vol. 77, No. 1, pp.99-113, January 1989

[Shannon 75] R. E. Shannon, "Simulation: a Survey with Research Suggestions", AIIE Transactions, Vol. 7, No. 3, pp.289-301, September 1975

[Ulrich 78] E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events", Communications of the ACM, Vol. 21, No. 9, pp.777-785, September 1978

[Vaucher 75] J. G. Vaucher and P. Duval, "A Comparison of Simulation Event List Algorithms", Communications of the ACM, Vol. 18, No. 4, pp.223-230, April 1975

[Venkatesh 86] K. Venkatesh, T. Radhakrishnan, and H. F. Li, "Discrete Event Simulation in a Distributed System", IEEE COMPSAC, pp.123-129, 1986

[Wyman 75] F. P. Wyman, "Improved Event-Scanning Mechanism for Discrete Event Simulation", Communications of the ACM, Vol. 18, No. 6, pp.350-353, June 1975

Appendix

Program Listings

```
VAL string.length   IS 5:
PROTOCOL Message    IS INT; [string.length]BYTE:
PROTOCOL TMessage   IS INT; INT; [string.length]BYTE:
PROTOCOL NewMessage IS INT; INT; [string.length]BYTE; INT:

PROTOCOL letters
  CASE
    letter; INT
    end.of.letters
    terminate
:
VAL return          IS INT '*c':
VAL ft.del.chl      IS #CD:
VAL ft.del          IS #CE:
VAL tt.left         IS 8:
VAL tt.stop.ch      IS 256:
VAL space           IS INT '*s':

VAL Number.of.transputers IS 4:
VAL NULL            IS BYTE '0':
VAL EOF             IS BYTE 'Z':
VAL YES             IS 1:
VAL NO              IS 0:
VAL linkout0 IS 0:
VAL linkout1 IS 1:
VAL linkout2 IS 2:
```

```
VAL linkout3 IS 3:
VAL linkin0 IS 4:
VAL linkin1 IS 5:
VAL linkin2 IS 6:
VAL linkin3 IS 7:
```

```
-- This program is for a source and fork (root transputer)
-- The root receives the message from the keyboard,
-- sends it to the Transputer 1 and 3
-- receives the acknowledgement from T1, T2, T3, and
-- sends the ack to the screen

#USE cheader
#USE userio
#USE strings

VAL Branch   IS 2:

CHAN OF TMessage input0, input1, input2, result:
CHAN OF TMessage out1, out2:
[Number.of.transputers]INT time:
[Number.of.transputers]INT pro.time:
[Number.of.transputers][string.length]BYTE message:
INT char, x, y, up.time:
[Branch]INT out.time:

PLACE out1 AT linkout2: -- MES to Transputer 1
PLACE out2 AT linkout3: -- MES to Transputer 3
PLACE input0 AT linkin2: -- ACK from Transputer 1
PLACE input1 AT linkin1: -- ACK from Transputer 2
PLACE input2 AT linkin3: -- ACK from Transputer 3

#USE cheader

PROC keyboard.handler (INT st,
                       CHAN OF INT in,
                       INT in.time,
                       []BYTE message,
                       INT up.time)

  #USE userio
  #USE ioconv
  BOOL going, next:
  [string.length]BYTE string:
  INT char, length, pro.time:

  SEQ
    PAR i = 0 FOR string.length
      string[i] := '*s'
    going := TRUE
    next := TRUE
    length := 0
    string[length] := BYTE st
    length := length + 1
    WHILE going
      SEQ
        in ? char
        CASE char
          space
            BOOL err:
            SEQ
              IF
                next
```

```
                SEQ
                  STRINGTOINT(err, in.time, string)
                  IF
                    err
                      SEQ
                        PAR i = 0 FOR string.length
                          string[i] := '*s'
                    TRUE
                      SKIP
                  length := 0
              TRUE
                SKIP
        return
          SEQ
            IF
              (length <> 0) AND (in.time > 0)
                SEQ
                  IF
                    up.time >= in.time
                      in.time := up.time + 1
                    TRUE
                      SKIP
                  [message FROM 0 FOR string.length] := string
                  going := FALSE
                  up.time := in.time
              TRUE
                SKIP
        ft.del.chl
          IF
            length > 0
              SEQ
                length := length - 1
                string[length] := '*s'
            TRUE
              SKIP
        ELSE
          IF
            length < string.length
              SEQ
                string[length] := BYTE char
                length := length + 1
            TRUE
              SKIP
:

#USE cheader


PROC goto.screen(INT x, y, pro.time, time,
                 []BYTE message,
                 CHAN OF ANY out)

  #USE strings
  #USE userio

  SEQ
    goto.xy (out, 0, y)
```

```
    write.full.string(out,
    "                                                                              ")
    goto.xy (out, x, y)
    y := y + 1
    IF
      y >= 24
        SEQ
          y := 0
      TRUE
        SKIP
    write.int(out, pro.time, 5)
    right(out)
    write.int(out, time, 5)
    right(out)
    write.full.string(out, message)
:

#USE cheader

VAL Branch       IS 2:

-- CHAN OF TMessage input: input channel from former channel
-- CHAN OF TMessage output1, output2: next output channel
-- CHAN OF Message result: output channel to display the result
-- INT in.time: time component
-- receive the message from input, process, and
-- output it to otput1, output2, and result; 1 INPUT to 2 OUTPUT

PROC fork(CHAN OF TMessage output1, output2,
          []INT out.time,
          INT in.time,
          []BYTE strings)
  #USE userio
  #USE strings

  VAL INFINIT IS 10000:
  [Branch]INT time:
  [Branch][string.length]BYTE message:
  INT dep.time, dummy:
  INT mflag, min, bnumber:

  SEQ
    PAR i = 0 FOR Branch
      time[i] := 0
    PAR i = 0  FOR Branch
      PAR j = 0 FOR string.length
        message[i][j] := '*s'
    mflag := char.pos(EOF, strings)
    dummy := 0
    IF
      mflag < 0 -- not the last message
        SEQ
          dep.time := in.time
                -- since this node does not have its service time
          min := out.time[0]
          bnumber := 0
          SEQ i = 0 FOR Branch -- choose the smaller time component
```

```
            IF
              min > out.time[i]
                SEQ
                  min := out.time[i]
                  bnumber := i
              TRUE
                SKIP
          PAR i = 0 FOR Branch -- send the message to both
            IF
              (i = bnumber) AND (strings[0] <> NULL) -- real message
                SEQ
                  message[i] := strings
                  IF
                    out.time[i] > in.time
                      SEQ
                        time[i] := out.time[i]
                    TRUE
                      SEQ
                        time[i] := dep.time
                  out.time[i] := time[i]
              TRUE -- null message
                SEQ
                  message[i][0] := NULL
                  IF
                    out.time[i] > dep.time
                      SEQ
                        time[i] := out.time[i]
                    TRUE
                      SEQ
                        time[i] := dep.time
      TRUE -- terminate
        PAR i = 0 FOR Branch
          PAR
            time[i] := INFINIT
            message[i] := strings
    PAR -- output: process local time; time component; message component
      output1 ! dummy; time[0]; message[0]
      output2 ! dummy; time[1]; message[1]
      in.time := time[bnumber]
      [strings FROM 0 FOR string.length] := message[bnumber]
:

SEQ
  y := 0
  PAR i = 0 FOR Branch
    out.time[i] := 0
  PAR i = 0 FOR Number.of.transputers
    pro.time[i] := 0
  PAR
    INT flag:
    SEQ -- fork (source) process
      flag := -1
      up.time := 0
      WHILE flag < 0
        SEQ
          keyboard ? char -- input the message (t, m)
          keyboard.handler (char, keyboard, time[0], message[0], up.time)
```

```
        fork (out1, out2, out.time, time[0], message[0])
        flag := char.pos (EOF, message[0])
  INT flag:
  SEQ -- screen process
    flag := -1 -- each link receives: local clock; output time; output message
    WHILE flag < 0
      ALT
        input0 ? pro.time[1]; time[1]; message[1] -- Transputer 1
          SEQ
            IF
              pro.time[1] < 0 -- branch1 echo back input data
                x := 0
              TRUE
                x := 15
            goto.screen(x, y, pro.time[1], time[1], message[1], screen)
        input1 ? pro.time[2]; time[2]; message[2] -- Transputer 2
          SEQ
            x := 35
            goto.screen(x, y, pro.time[2], time[2], message[2], screen)
        input2 ? pro.time[3]; time[3]; message[3] -- Transputer 3
          SEQ
            IF
              pro.time[3] < 0 -- branch2 echo back input data
                x := 0
              TRUE
                SEQ
                  x := 50
                  flag := char.pos (EOF, message[3])
            goto.screen(x, y, pro.time[3], time[3], message[3], screen)

keyboard ? char
```

```
-- CHAN OF TMessage input1, input2: input channel from former channels
-- CHAN OF TMessage result: next output channel
-- VAL INT ser.time: process service time
-- This program is for a merge
-- The merge inputs on two stream of messages, compares them,
-- and outputs the smaller time message on one stream of messages
-- ; 2 INPUT to 1 OUTPUT

-- MODIFIED!: there are 3 processes running in parallel
-- Receiver: whenever the message comes it inputs all and sends them to the event queue
-- Event queue: inputs messages from the receiver, outputs them to the simulator
-- Simulator: processes messages and sends to the next processor

#USE cheader

PROC merge (CHAN OF TMessage input1, input2,
            CHAN OF TMessage result,
            VAL INT ser.time)
  #USE userio
  #USE strings

  PROC key.in (CHAN OF TMessage input,
               []BYTE message,
               INT dummy, in.time,
               VAL INT pro.time,
               BOOL nflag,
               INT now,
               VAL INT timeout,
               TIMER clock)
    SEQ
      PAR i = 0 FOR string.length
        message[i] := '*s'
      clock ? now
      ALT
        input ? dummy; in.time; message
          SEQ
            IF
              in.time < pro.time -- message error
                SEQ
                  -- STOP
                  in.time := pro.time
              TRUE
                SKIP
            nflag := FALSE
        NOT (nflag) & clock ? AFTER now PLUS timeout
          SEQ
            in.time := pro.time
            nflag := TRUE
            message[0] :=  NULL
  :

  -- make the next message
  -- INT time: process local time
  PROC process ([]BYTE message,
                INT pro.time, in.time, time,
                VAL INT ser.time,
                INT real, out.time)
```

```
  SEQ
    pro.time := in.time
    time := pro.time + ser.time
    real := char.pos(NULL, message)
    IF
      real < 0
        SEQ
          IF
            out.time > pro.time
              SEQ
                time := out.time + ser.time
            TRUE
              SKIP
          out.time := time
      TRUE
        IF
          time < out.time
            SEQ
              time := out.time
          TRUE
            SKIP
:

-- sort chronologically

PROC sort.queue([]INT pro, time,
                [][]BYTE message,
                INT pt)

  SEQ i = 0 FOR pt
    PAR
      pro[i] := pro[i + 1]
      time[i] := time[i + 1]
      message[i] := message[i + 1]
:

VAL INFINIT IS 10000:
VAL SEC     IS 100000:
VAL Merge   IS 2:
VAL Buf1    IS 1:
VAL Buf2    IS 10:

CHAN OF TMessage ch.in1, ch.in2:
CHAN OF TMessage ch.out1, ch.out2:
CHAN OF ANY      signal1, signal2:

PAR
  ------------------------------------------------------------------------
  [Merge]INT pro.time, in.time:
  [Merge][string.length]BYTE message:
  PAR -- receiver
    INT mflag:
    SEQ
      mflag := -1
      WHILE mflag < 0
        SEQ
```

```
        input1 ? pro.time[0]; in.time[0]; message[0] -- input from the link
        ch.in1 ! pro.time[0]; in.time[0]; message[0] -- output to the bufffer
        mflag := char.pos(EOF, message[0])

  INT mflag:
  SEQ
    mflag := -1
    WHILE mflag < 0
      SEQ
        input2 ? pro.time[1]; in.time[1]; message[1] -- input from the link
        ch.in2 ! pro.time[1]; in.time[1]; message[1] -- output to the buffer
        mflag := char.pos(EOF, message[1])

----------------------------------------------------------------------
PAR -- event queue
  INT mflag, pt, req: -- pt: queue pointer
  [Buf1]INT pro.buf, time.buf:
  [Buf1][string.length]BYTE mess.buf:
  SEQ
    mflag := -1
    pt := 0
    WHILE (mflag < 0) OR (pt > 0)
      -- run until the last message araival and the queue is empty
      PRI ALT
        (pt > 0) & signal1 ? req -- request from the simulator
          SEQ
            ch.out1 ! pro.buf[0]; time.buf[0]; mess.buf[0]
            pt := pt - 1
            IF -- sort the queue
              pt > 0
                SEQ
                  sort.queue(pro.buf, time.buf, mess.buf, pt)
              TRUE
                SKIP
        (pt < Buf1) & ch.in1 ? pro.buf[pt]; time.buf[pt]; mess.buf[pt]
          SEQ
            mflag := char.pos(EOF, mess.buf[pt])
            IF -- check if previous message is null
              (pt > 0) AND (mess.buf[pt - 1][0] = NULL)
                PAR -- ignore null messages
                  pro.buf[pt - 1] := pro.buf[pt]
                  time.buf[pt - 1] := time.buf[pt]
                  mess.buf[pt - 1] := mess.buf[pt]
              TRUE -- add the message on the queue
                pt := pt + 1

  INT mflag, pt, req: -- pt: queue pointer
  [Buf2]INT pro.buf, time.buf:
  [Buf2][string.length]BYTE mess.buf:
  SEQ
    mflag := -1
    pt := 0
    WHILE (mflag < 0) OR (pt > 0)
      -- run until the last message arrival and the queue is empty
      PRI ALT
        (pt > 0) & signal2 ? req -- the request from the simulator
          SEQ
```

```
          ch.out2 ! pro.buf[0]; time.buf[0]; mess.buf[0]
          pt := pt - 1
          IF -- sort the queue
            pt > 0
              SEQ
                sort.queue(pro.buf, time.buf, mess.buf, pt)
            TRUE
              SKIP
      (pt < Buf2) & ch.in2 ? pro.buf[pt]; time.buf[pt]; mess.buf[pt]
        SEQ
          mflag := char.pos(EOF, mess.buf[pt])
          IF -- check previous message is null
            (pt > 0) AND (mess.buf[pt - 1][0] = NULL)
              PAR
                pro.buf[pt - 1] := pro.buf[pt]
                time.buf[pt - 1] := time.buf[pt]
                mess.buf[pt - 1] := mess.buf[pt]
            TRUE -- add the message on the buffer
              pt := pt + 1

------------------------------------------------------------------------
[Merge]BOOL nflag:
INT pro.time, out.time, time, id:
INT mflag, timeout, real1, real2, min, mnumber:
[Merge]INT in.time, pre.pro, now:
[Merge][string.length]BYTE message:
[Merge]TIMER clock:
SEQ -- simulator
  pro.time := 0
  out.time := 0
  time := 0
  mflag := -1
  PAR i = 0 FOR Merge
    PAR
      in.time[i] := 0
      nflag[i] := FALSE
  timeout := 1
  WHILE mflag < 0
    SEQ
      PAR
        SEQ
          IF
            in.time[0] = pro.time
              SEQ
                signal1 ! YES
                key.in(ch.out1, message[0], pre.pro[0], in.time[0],
                       pro.time, nflag[0], now[0], timeout, clock[0])
            TRUE
              SKIP
        SEQ
          IF
            in.time[1] = pro.time
              SEQ
                signal2 ! YES
                key.in(ch.out2, message[1], pre.pro[1], in.time[1],
                       pro.time, nflag[1], now[1], timeout, clock[1])
            TRUE
```

```
        SKIP
timeout := SEC
min := INFINIT
SEQ i = 0 FOR Merge -- Find smaller service time
  IF
    min > in.time[i]
      SEQ
        min := in.time[i]
        mnumber := i
    TRUE
      SKIP
mflag := char.pos(EOF, message[mnumber])
IF
  mflag < 0 -- Process the messages
    SEQ
      id := NO
      process (message[mnumber], pro.time, in.time[mnumber], time,
                ser.time, real1, out.time)
      IF -- If the mesage is real it's sent
        real1 < 0 -- real message
          result ! pro.time; time; message[mnumber]
        TRUE -- null message
          SKIP
      -- Check whether time components are the same
      SEQ i = 0 FOR Merge
        IF
          (i <> mnumber) AND (in.time[i] = in.time[mnumber])
            SEQ
              process (message[i], pro.time, in.time[i], time,
                        ser.time, real2, out.time)
              IF -- If the mesage is real it's sent
                real2 < 0 -- real message
                  SEQ
                    id := YES
                    result ! pro.time; time; message[i]
                TRUE
                  SKIP
          TRUE
            SKIP
      -- If no equal time components and the message is null
      -- or all are null messages, one null message is sent
      IF
        (id = NO) AND (real1 >= 0)
          result ! pro.time; time; message[mnumber]
        TRUE
          SKIP
  TRUE -- terminate
    SEQ
      time := INFINIT
      pro.time := INFINIT
      result ! pro.time; time; message[mnumber]
:
```

```
-- CHAN OF TMessage input1: input channel from former channels
-- CHAN OF TMessage result: next output channel
-- VAL INT ser.time: process service time
-- This program is for a server.
-- The server inputs one stream of messages and
-- outputs one stream of messages; 1 INPUT to 1 INPUT

#USE cheader

PROC slave (CHAN OF TMessage input, output, result,
             VAL INT ser.time)
  #USE userio
  #USE strings

  PROC key.in (CHAN OF TMessage input, output,
               []BYTE message,
               INT pre.pro, in.time,
               VAL INT pro.time,
               BOOL nflag,
               INT now,
               VAL INT timeout,
               TIMER clock)
    SEQ
      PAR i = 0 FOR string.length
        message[i] := '*s'
      clock ? now
      ALT
        input ? pre.pro; in.time; message
          SEQ
            output ! -1; in.time; message
            IF
              in.time < pro.time -- message error
                SEQ
                  -- STOP
                  in.time := pro.time
              TRUE
                SKIP
            nflag := FALSE
        NOT (nflag) & clock ? AFTER now PLUS timeout
          SEQ
            in.time := pro.time
            nflag := TRUE
            message[0] :=  NULL
  :

  VAL INFINIT IS 10000:
  VAL SEC     IS 100000:
  BOOL nflag:
  INT pro.time, out.time, in.time, time, pre.pro:
  INT ch.pos, mflag, real, now, timeout:
  [string.length]BYTE message:
  TIMER clock:

  SEQ
    nflag := FALSE
    pro.time := 0
    out.time := 0
```

```
    time := 0
    in.time := 0
    mflag := -1
    real := -1
    timeout := 1
    WHILE mflag < 0
      SEQ
        key.in(input, result, message, pre.pro, in.time, pro.time, nflag, now,
               timeout, clock)
        timeout := SEC
        mflag := char.pos(EOF, message)
        IF
          mflag < 0
            SEQ
              pro.time := in.time
              time := pro.time + ser.time
              real := char.pos(NULL, message)
              IF
                real < 0 -- real message
                  SEQ
                    IF
                      out.time > pro.time
                        SEQ
                          time := out.time + ser.time
                      TRUE
                        SKIP
                    out.time := time
                TRUE -- null message
                  IF
                    time < out.time
                      SEQ
                        time := out.time
                    TRUE
                      SKIP
          TRUE -- terminate
            SEQ
              time := INFINIT
              pro.time := INFINIT
        output ! pro.time; time; message
        result ! pro.time; time; message
:
```

```
-- This is a configuration file.

VAL Se.time1     IS 2:
VAL Se.time2     IS 0:
VAL Se.time3     IS 10:

[Number.of.transputers]CHAN OF TMessage link:
[Number.of.transputers]CHAN OF TMessage result:

PLACED PAR
  PROCESSOR 1 T8
    PLACE link[0] AT linkin3:
    PLACE link[1] AT linkout2:
    PLACE result[0] AT linkout3:
    slave (link[0], link[1], result[0], Se.time1)


  PROCESSOR 2 T8
    PLACE link[1] AT linkin3:
    PLACE link[2] AT linkin2:
    PLACE result[1] AT linkout1:
    merge (link[1], link[2], result[1], Se.time2)

  PROCESSOR 3 T8
    PLACE link[3] AT linkin2:
    PLACE link[2] AT linkout3:
    PLACE result[2] AT linkout2:
    slave (link[3], link[2], result[2], Se.time3)
```

```
-- This program is for a source and fork(root transputer)
-- The root inputs the message from the file,
-- sends it to the Transputer 1 and 3
-- receive acknowledgement from T1, T2, T3, and
-- outputs results to the file.

#USE cheader
#USE userio
#USE interf
#USE strings

VAL Branch   IS 2:
PROTOCOL W.FILE IS INT; INT; [string.length]BYTE; INT:

CHAN OF TMessage input0, input1, input2:
CHAN OF TMessage out1, out2:
CHAN OF TMessage in.fork:
CHAN OF W.FILE   output:
[Number.of.transputers]INT time:
[Number.of.transputers]INT pro.time:
[Number.of.transputers][string.length]BYTE message:
INT char, tr.number:

PLACE out1 AT linkout2: -- MES to Transputer 1
PLACE out2 AT linkout3: -- MES to Transputer 3
PLACE input0 AT linkin2: -- ACK from Transputer 1
PLACE input1 AT linkin1: -- ACK from Transputer 2
PLACE input2 AT linkin3: -- ACK from Transputer 3

#USE cheader
PROTOCOL W.FILE IS INT; INT; [string.length]BYTE; INT:

PROC finput (CHAN OF TMessage f.data,
             CHAN OF ANY from.file, to.file)
  #USE uservals
  #USE userio
  #USE interf

  SEQ
    INT input.error:
    [string.length]BYTE message, string:
    SEQ
      CHAN OF INT filekeys:
      PAR
        ----------------------------------------------------------
        keystream.from.file (from.file, to.file,
                             filekeys, 1, input.error)
          -- check input.error when real screen accessible again
        -----------------------------------------------------------
        INT kchar:
        INT time, len:
        SEQ
          kchar := 0
          len := 1
          WHILE kchar <> ft.terminated
            SEQ
              read.char (filekeys, kchar)
```

```
              IF
                kchar < 0
                  SKIP
                TRUE
                  INT k:
                  SEQ
                    read.int (filekeys, time, kchar)
                    PAR i = 0 FOR string.length
                      message[i] := ' '
                    read.text.line(filekeys, len, string, kchar)
                    k := 0
                    WHILE string[k] <> '*c'
                      SEQ
                        message[k] := string[k]
                        k := k + 1
              IF
                kchar = ft.terminated
                  SKIP
                TRUE
                  SEQ
                    IF
                      kchar = ft.number.error
                        f.data ! -2; time; message
                      TRUE
                        SKIP
                    f.data ! -1; time; message
          IF
            (kchar >= 0) OR (kchar = ft.number.error)
              keystream.sink (filekeys)
                    -- consume the rest of the keyboard file
            TRUE
              SKIP  -- keyboard file has terminated or failed
        -------------------------------------------------------------
      IF
        input.error <> 0
          SEQ
            f.data ! -3; input.error; message
        TRUE
          SKIP
:

PROC foutput (CHAN OF W.FILE f.data,
              CHAN OF ANY from.file, to.file)

  #USE uservals
  #USE userio
  #USE interf

  INT st, et:
  TIMER clock:

  SEQ
    clock ? st

    CHAN OF ANY echo:
    PAR
      -------------------------------------------------------------
```

```
      INT time, pro.time, tr.number:
      [string.length]BYTE message:
      SEQ
        message[0] := ' '
        tr.number := 0
        WHILE (message[0] <> EOF) OR (tr.number <> (Number.of.transputers - 1))
          -- WHILE not the last message & not the last transputer
          SEQ
            f.data ? pro.time; time; message; tr.number
            SEQ i = 0 FOR (80 / Number.of.transputers) * tr.number
              write.char (echo, ' ')
            write.int (echo, pro.time, 0)
            write.char (echo, ' ')
            write.int (echo, time, 0)
            write.char (echo, ' ')
            write.full.string (echo, message)
            -- write.char (echo, ' ')
            -- write.int (echo, tr.number, 0)
            newline (echo)
        clock ? et
        write.full.string (echo, "run time ticks = ")
        write.int (echo, et - st, 0)
        write.endstream (echo) -- terminate scrstream.sink
      --------------------------------------------------------------
      INT fold.number, result:
      SEQ
        scrstream.to.file (echo, from.file, to.file,
                           "output data", fold.number, result)
                               -- write on the file

:

#USE cheader

VAL Branch       IS 2:

-- CHAN OF TMessage input: input channel form former channel
-- CHAN OF TMessage output1, output2: next output channel
-- CHAN OF Message result: output channel to display the result
-- VAL INT ser.time: process service time
-- receive the message from input, implement by slave time, and
-- output it to otput1, output2, and result; 1 INPUT to 2 OUTPUT

PROC fork(CHAN OF TMessage output1, output2,
          CHAN OF TMessage input)

  #USE userio
  #USE strings

  VAL INFINIT IS 10000:
  [Branch]INT time, out.time:
  [Branch][string.length]BYTE message:
  [string.length]BYTE strings:
  INT dep.time, pro.time, in.time:
  INT mflag, min, bnumber:

  SEQ
```

```
PAR i = 0 FOR Branch
  out.time[i] := 0
strings[0] := ' '

-- WHILE loop

WHILE strings[0] <> EOF
  SEQ
    input ? pro.time; in.time; strings
    PAR i = 0 FOR Branch
      time[i] := 0
    PAR i = 0  FOR Branch
      PAR j = 0 FOR string.length
        message[i][j] := '*s'
    pro.time := 0

    mflag := char.pos(EOF, strings)
    IF
      mflag < 0
        SEQ
          dep.time := in.time
          min := out.time[0]
          bnumber := 0
          SEQ i = 0 FOR Branch
            IF
              min > out.time[i]
                SEQ
                  min := out.time[i]
                  bnumber := i
              TRUE
                SKIP
          PAR i = 0 FOR Branch
            IF
              (i = bnumber) AND (strings[0] <> NULL) -- real message
                SEQ
                  message[i] := strings
                  IF
                    out.time[i] > in.time
                      SEQ
                        time[i] := out.time[i]
                    TRUE
                      SEQ
                        time[i] := dep.time
                  out.time[i] := time[i]
              TRUE -- null message
                SEQ
                  message[i][0] := NULL
                  IF
                    out.time[i] > dep.time
                      SEQ
                        time[i] := out.time[i]
                    TRUE
                      SEQ
                        time[i] := dep.time
      TRUE -- terminate
        PAR i = 0 FOR Branch
          PAR
```

```
              time[i] := INFINIT
              message[i] := strings
        PAR -- output messages to all branches
          output1 ! pro.time; time[0]; message[0]
          output2 ! pro.time; time[1]; message[1]

:

SEQ
  write.full.string (screen, "start")
  newline (screen)
  PAR
    ------------------------------------------------------------------
    finput (in.fork, from.user.filer[1], to.user.filer[1])
            -- input data from file
    ------------------------------------------------------------------
    foutput (output, from.user.filer[2], to.user.filer[2])
            -- output data to file
    ------------------------------------------------------------------
    fork (out1, out2, in.fork)
            -- distribute to branches
    --------------------------------------------------------------------
    INT flag:
    SEQ
      flag := -1
      WHILE flag < 0
        ALT
          input0 ? pro.time[0]; time[0]; message[0] -- Transputer 1
            SEQ
              IF
                pro.time[0] < 0
                  SEQ
                    tr.number := 0
                    write.full.string (screen, "transputer 0")
                    right (screen)
                    write.int (screen, time[0], 0)
                    right (screen)
                    write.full.string (screen, message[0])
                    newline (screen)
                TRUE
                  SEQ
                    tr.number := 1
                    write.full.string (screen, "transputer 1")
                    right (screen)
                    write.int (screen, time[0], 0)
                    right (screen)
                    write.full.string (screen, message[0])
                    newline (screen)
              output ! pro.time[0]; time[0]; message[0]; tr.number
          input1 ? pro.time[1]; time[1]; message[1] -- Transputer 2
            SEQ
              tr.number := 2
              write.full.string (screen, "transputer 2")
              right (screen)
              write.int (screen, time[1], 0)
              right (screen)
              write.full.string (screen, message[1])
```

```
        newline (screen)
        output ! pro.time[1]; time[1]; message[1]; tr.number
    input2 ? pro.time[2]; time[2]; message[2] -- Transputer 3
      SEQ
        IF
          pro.time[2] < 0
            SEQ
              tr.number := 0
              write.full.string (screen, "transputer 0")
              right (screen)
              write.int (screen, time[2], 0)
              right (screen)
              write.full.string (screen, message[2])
              newline (screen)
          TRUE
            SEQ
              tr.number := 3
              write.full.string (screen, "transputer 3")
              right (screen)
              write.int (screen, time[2], 0)
              right (screen)
              write.full.string (screen, message[2])
              newline (screen)
              flag := char.pos (EOF, message[2])
        output ! pro.time[2]; time[2]; message[2]; tr.number

write.full.string (screen, "process ends type ANY to return to TDS")
keyboard ? char
```