Distrib Comput (1995) 8: 181–190

DISTRIBUTED COMPUTING

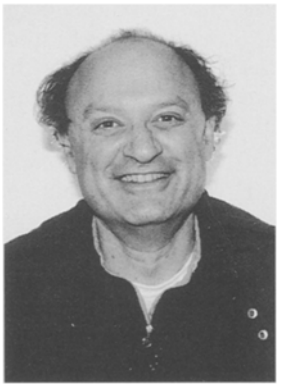
Parallel simulation on the hypercube multiprocessor

Azzedine Boukerche*, Carl Tropper**

School of Computer Science, McGill University, 3480 University Street, Montreal, Quebec H3A-2A7, Canada

Received: December 1990 / Accepted: December 1994

Azzedine Boukerche received the State Engineer degree in Software Engineering from Oran University, Oran, Algeria, and the M.Sc. degree in Computer Science from McGill University, Montreal, Canada. He is a Ph.D. candidate at the School of Computer Science, McGill University. During 1991–1992, he was a visiting doctoral student at the California Institute of Technology. He is employed as a Faculty Lecturer of Computer Science at McGill University since 1993. His research interests include parallel simulation, distributed algorithms, and system performance analysis. He is a student member of the IEEE and ACM.



Carl Tropper is an Associate Professor of Computer Science at McGill University. His primary area of research is parallel discrete event simulation. His general area of interest is in parallel computing and distributed algorithms in particular. Previously, he has done research in the performance modeling of computer networks, having written a book, *Local Computer Network Technologies*, while active in the area. Before coming to university life, he worked for the BBN Corporation and the Mitre Corporation, both located in the Boston area. He spent the 1991–92 academic year on a sabbatical leave at the Jet Propulsion Laboratories of the California Institute of Technology where he contributed to a project centered about the verification of flight control software. As part of this project he developed algorithms for the parallel simulation of communicating finite state machines. During winters he may be found hurtling down mountains on skis.

Summary. This paper focuses upon a particular conservative algorithm for parallel simulation, the Time of Next Event (TNE) suite of algorithms [13]. TNE relies upon a shortest path algorithm which is independently executed on each processor in order to unblock LPs in the processor and to increase the parallelism of the simulation. TNE differs fundamentally from other conservative approaches in that it takes advantage of having several LPs assigned to each processor, and does not rely upon message passing to provide lookahead. Instead, it relies upon a shortest path algorithm executed independently in each processor. A deadlock resolution algorithm is employed for interprocessor deadlocks. We describe an empirical investigation of the performance of TNE on the iPSC/i860 hypercube multiprocessor. Several factors which play an important role in TNE's behavior are identified, and the speedup relative to a fast uniprocessor-based event list algorithm is reported. Our results indicate that TNE yields good speedups and out-performs an optimized version of the Chandy&Misra-null message (CMB) algorithm. TNE was 2–5 times as fast as the CM approach for less than 10 processors (and 1.5–3 times as fast when more than 10 processors were used for the same population of processes.)

Key words: Conservative approach – Distributed deadlock – FCFS queueing network – Lookhead – Multiprocessor – Parallel simulation – Torus

1 Introduction

Two major classes of algorithms have been developed for the synchronization of parallel simulations – the optimistic approach (aka Time Warp) due to Jefferson [17] and the conservative algorithms inspired by Chandy and Misra [4] and Bryant [3]. Both approaches are based on a distributed model of computation in which processes communicate only via *message passing*. They differ, however, in the way they deal with causality errors. Excellent surveys of the area may be found in [9] and [27].

The algorithms which we examine in this paper are conservative. In this model a network of logical processes (LPs) is used to simulate the objects of a system, e.g.,

* This work has been completed while the author was a visiting doctoral student at the California Institute of Technology

** Was on sabbatical leave at the Jet Propulsion laboratories, California Institute of Technology

Correspondence to: C. Tropper

a computer network. Explicit links connect the LPs and messages are forwarded between LPs over the links. In the conservative paradigm an event cannot be simulated at an LP before it is certain that an event with a smaller time stamp cannot arrive. As a consequence of this blocking behavior, deadlocks can arise. Several solutions to this problem exist, each requiring a certain amount of overhead. A substantial amount of work has been done to evaluate the performance of these strategies [8, 19, 28, 29].

The objective of this paper is to assess the performance of a particular conservative algorithm, the Time of Next Event algorithm [13] and to compare its performance to an "optimized" version of the Chandy-Misra null message algorithm [4]. We also attempt to understand the factors which are basic to its performance. TNE (our abbreviation) employs a shortest path algorithm which computes the greatest lower bound on the arrival times of messages on all of the empty input links to an LP. This has the effect of speeding up a simulation by providing a "lookahead" ability to LPs. The algorithm can also be shown to break all intraprocessor deadlocks. When used on a distributed memory machine such as the Intel i860, inter-processor deadlocks can occur since TNE does not have access to global data structures. Deadlock breaking algorithms are described in [13] which deal with this eventuality. We note in passing that TNE is also described in PADS conference proceedings – in [10], [11] and [12].

The remaining sections of the paper are organized along the following lines. Section 2 compares TNE to other conservatives algorithms. Section 3 contains a description of the TNE algorithm, based upon its description in [13]. Section 4 contains a description of the parallel simulation which was built making use of TNE. Section 5 reports on the experiments which were carried out to assess TNE's performance relative to a fast uniprocessor simulator (i.e., the speedup) and to determine the factors which strongly influence its performance. The sources of overhead for TNE are also discussed. In Sect. 6, we describe the comparison of TNE with an optimized Chandy-Misra algorithm. Section 7 contains the conclusion.

2 Previous work

A number of conservative algorithms are related in spirit to TNE in that they attempt to unblock LPs by providing some form of estimate on the arrival time of a message to an empty queue. We describe several of these algorithms and then draw attention to the differences between these approaches and TNE.

We turn first to the original Chandy-Misra null-message algorithm [4]. In this algorithm, when an event message is sent on an output link, a null message bearing the same time-stamp is sent on all of the other out-bound links of the LP. This enables the recipient to simulate all events bearing a time-stamp *less than or equal* to the minimum timestamp of all of the null-messages. However, the arrival of a null-message at an LP can cause the recipient to generate yet another null-message, resulting in the potential generation of an overwhelming number of null messages. As a result, a number of attempts to optimize this basic scheme have appeared in the literature. For example, in [30] the authors refrain from sending null messages until such time as the LP becomes blocked. They refer to this approach as eager events, lazy null messages.

The use of demand-driven null messages is described in [22]. In this algorithm, an LP can query a predecessor (*via request messages*) in an attempt to obtain a new lower bound on the time stamps of messages which it can consume. One possible choice for when to make these queries is the time at which the LP becomes blocked. Another possibility is to wait until the processor becomes idle. However, no analysis of the performance of their algorithm is given. In the event that the predecessor cannot increase this lower bound, then it can query its own predecessors via request messages. This, however, can lead to deadlock [6, 22]. Consequently, a deadlock detection and breaking algorithm must also be employed [6].

Wagner et al. [30] describe a shared-memory version of this technique, which is called lazy blocking avoidance, since the algorithm waits until a processor is idle before it attempts to provide a new lower bound for blocked LPs. We note in passing that in a shared memory environment messages need not be sent within a processor – it suffices to queue or to de-queue events at the queues maintained by the individual LPs. When, for example, a message containing an event is to be "sent" to a neighboring LP, we need only attach the event to the queue associated with the LP. However, their algorithms are limited to a shared memory environment.

Nicol describes an appointment protocol [24] in which he makes use of a future events queue, containing pre-computed service times of jobs which are to arrive in an LPs' future. Making use of these service times, as well as pre-computed routing probabilities, the algorithm computes a lower bound on the arrival time of the next job at a successor LP. An LP requests an appointment (in the form of this lower bound) from a predecessor when it finds itself blocked.

Lubachevsky, in [20] describes the *bounded lag algorithm*. The bounded lag restriction with the parameter B allows events to be simulated concurrently if their time stamps lie in the small interval $[\mu, \mu + B]$, where μ is the current smallest timestamp in the simulated system and B is a known positive and finite constant. The efficiency of Lubachevsky's method depends upon the broadcasting algorithm and the values of B. If B is small, there might be little parallelism available in the interval $[\mu, \mu + B]$. On the other hand if B is larger the overhead of detecting an enabled event is larger as well.

All of the algorithms which we have described provide information about the future arrival of messages to an LP by making use of information obtained from neighboring LPs. In order to obtain this information, either actual messages have to be sent (between processors) or events have to be placed in queues. TNE differs fundamentally from this approach in that it relies upon the periodic execution of a shortest path algorithm, making use of information about all of the LPs in a processor. It calculates a greatest lower bound on the arrival time of messages to all empty input queues in a processor in one

execution of the algorithm. The input to the algorithm consists of the minimum time stamps at all non-empty input queues, the local simulation times at each LP, and the LP graph. TNE is executed when the processor is idle, i.e. when no useful computation is being performed.

The algorithms described above rely on the propagation of successive lower bounds between actual LPs, whereas TNE makes use of the information obtained from a snapshot of the state of all LPs within a processor. As we have indicated above, this lack of state information results in the possibility of interprocessor deadlocks.

As the complexity of a shortest path algorithm is $O(n \log n)$, the overhead of TNE when executed within the confines of one processor is modest. By making use of an heuristic (described in Sect. 4) which avoids calling a deadlock detection algorithm to detect inter-processor deadlocks unnecessarily, we create a global algorithm which has a low overhead. This is borne out by experimental results described later in the paper (Sect. 5.)

In order to examine the performance of TNE, we carried out two sets of experiments. One was aimed at determining the important factors affecting TNE's performance. The second set of experiments compared TNE's performance to that of an "optimized" version of the Chandy-Misra null message algorithm. In our version of the null message algorithm, a newly arrived null message at a queue will over-write any pre-existing message in the queue. This simple optimization saves both buffer space and execution time.

3 Overview of time next event (TNE) algorithm

In this section, we describe the intuition behind the TNE algorithm and present pseudo code for TNE. As a consequence of space limitations we do not provide a complete and detailed description. The reader should consult [13] for a detailed description.

We employ the example presented in Fig. 1 to informally described TNE.

Figure 1 shows three logical processes LP_x, LP_y and LP_2 connected by directed empty links. Let LST_x, LST_y and LST_z be the local simulation times, and $T_{\min_x}$, $T_{\min_y}$ and $T_{\min_z}$ be the smallest timestamp at LP_x, LP_y and LP_z respectively. $T_{\text{smin}_{i,j}}$ denotes the smallest time-stamp increment an event sent from LP_i to its neighbor LP_j. We define T_i to be the smallest timestamp which can be sent by LP_i and $T_{i,j}$ the TNE of the link (i,j).

Consider the empty link from LP_x to LP_y. LP_x cannot send an event message with a smaller timestamp than T_x, where $T_x = \max(LST_x, T_{\min_x})$. On its way[1], the message has to pass through LP_y as well. A service time has to be added to each output sent by LP_x. Therefore LP_y cannot expect a message with smaller timestamp than $T_{x,y}$, where $T_{x,y} = T_x + T_{\text{smin}_{x,y}}$. Thus a new T_y is computed as: $T_y = \min(T_y, T_{x,y})$. LP_z cannot expect a message with a smaller timestamp than $T_{y,z}$ from LP_y for the same reason.

Fig. 1. LPs connected by empty links

Based on these observations, a shortest path algorithm [5] may be employed to compute the TNE. The TNE algorithm explores the directed graph of LPs connected by empty links. It finds estimates for the next timestamps on all empty links that belong to the same processor. If the estimate(s) of the future timestamp at all of an LP's empty links are higher than the LP's smallest timestamps, then the event with the smallest timestamp can be processed, i.e. the LP is unblocked.

Pseudo code of the TNE is contained in Fig. 2 below.

We could improve the lower bound produced by the TNE by precomputing a portion of the computation for future events [24]. The priority queue (PQ) might be implemented as a heap or a splay tree. The TNE algorithm is not computationally very expensive. In [13], the authors demonstrated that TNE can be mapped to the shortest path problem, and therefore has the same complexity, i.e. $O(n \log(n))$ where n is the number of nodes in the graph. The TNE algorithm helps to unblock LPs and breaks all local deadlocks – see [13], p. 114 for an example. However, it doesnot have access to any global information, and this unfortunately makes *inter-processor deadlocks* possible. Thus another algorithm is needed to take care of this kind of deadlock.

3.1 Distributed deadlock detection and recovery

A necessary and sufficient condition for a deadlock is given in the following theorem.

Theorem 3.1. A deadlock *exists if and only if there is a cycle of empty links which all have the same time-of-next event, and LPs which all are blocked because of these links and only these links*, i.e. there exists a knot.

This theorem generalizes the theorem proved by Peacock et al. [25] which proved the same result for a cycle, as opposed to a knot.

Figure 3 shows LP_x, LP_y, LP_z and LP_w connected by directed empty links. The local simulation times are shown in parentheses. Suppose that the old TNEs are equal to 5. Since the TNE is executed in each processor, it does not have global information. Thus the TNE computed by the TNE algorithm would be equal to 11, which is in fact the highest LST[2] in the cycle. The propagation of the TNEs does not help to break the deadlock, since the smallest timestamp in the cycle is larger than the largest LST. As a consequence we have an inter-processor deadlock.

[1] We assume that it is safe for LP_x to process the event with the smallest time stamp T_x

[2] LST is the time when the last event was processed by an LP

Input: graph $G(V, E)$ and its subgraph $G_p(V_p, E_p)$ where V are LPs E are empty links, V_p is the set of *LPs* assigned to processor p, and E_p is the set of empty links belonging to processor p; LST_u, $T_{\min_u}$, $T_{\text{smin}_{u,v}}$, $T_{w,u}$ such that $u \in V_p$, $v \in V$ and $w \in E(V - V_p)$.
Output: time-of-next-event $T_{u,v}$ for all empty links (u, v), such that $u \in V_p$ and $v \in V$.
Temporary data structure: priority queue PQ.

```
begin
  PQ = empty;
  for all (u,w) s.t. u ∈ V_p and w ∈ (V − V_p) do        {* Initialization Step *}
    T_w = 0;
  endfor;
  for all v ∈ V_p do                                     {* Compute the smallest time stamp which can be sent by LP_v *}
    Temp = min_{w∈V−V_p}(T_{min_v}, T_{w,u});            {* T_{min_v} is the smallest time stamp at LP_v *}
    T_v = max(Temp, LST_v),
    insert (v, PQ);
  endfor;
  while not empty (PQ) do                                {* Compute TNEs from the smallest T in PQ *}
    select u ∈ PQ s.t. T_u = min_{v∈PQ}(T_v);
    delete(u, PQ);
    for all (u,v) s.t. v ∈ V_p do                        {* Compute TNEs for links (u,v) *}
      T_{u,v} = T_u + T_{smin_{u,v}};                    {* Re-compute T_v if necessary *}
      if (T_v > max(T_{u,v}, LST_v) then
        T_v = max(T_{u,v}, LST_v);
    endfor;
  endwhile;
end.
```

Fig. 2. TNE algorithm

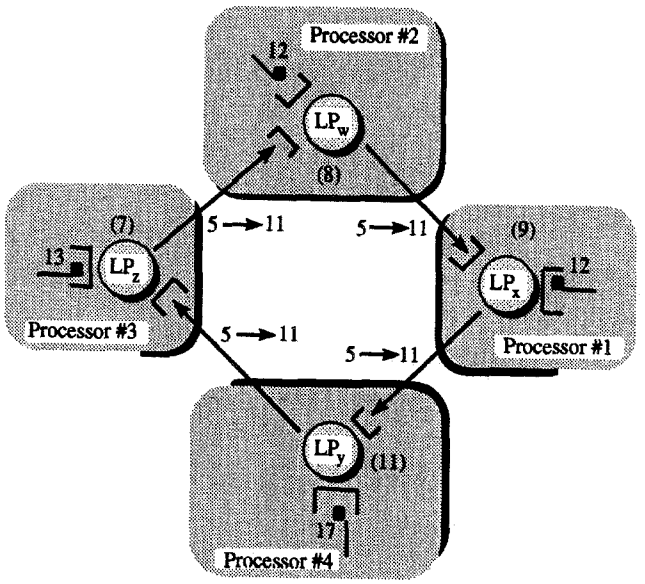


Fig. 3. Inter-processor deadlock

We now present a brief description of the inter-processor deadlock breaking algorithm, namely the Distributed Limited version of the TNE algorithm (DLTNE in short) [13]. DLTNE is able to break deadlocks between several processors while the remaining processors continue to process their part of the simulation. It is based on a distributed shortest path algorithm [5].

Its main idea is to follow empty input links in the opposite direction of the message flow. When the algorithm moves from one blocked LP_i to another blocked LP_j, it computes a quantity B called the *bias*, [13], defined below. When the algorithm reaches an unblocked LP, it stops. This is because that LP cannot send a message with a timestamp smaller than $\max(LST, T_{\text{smin}})$. LST is the local simulation time of the LP while T_{smin} is the smallest timestamp at the LP.

Let us assume that we want to compute the TNE for the link from LP_b to LP_a. For simplicity, we will assume that the service time is zero. (Non-zero service times can be used in the algorithm.) Groselj and Tropper define the *bias* for the LP_i as:

$$B_i = \max_{j \in P_{i,b}}(LST_j);$$

where $P_{i,b}$ is a sequence of LPs $(i, \ldots, b, a)$ such that there is an empty directed link between each LP and its successors. The bias is computed for each path of LPs reaching LP_i.

With this definition, (see [13] for further details) one can easily see that LP_a cannot expect a message with a smaller time stamp than B_i from LP_i along the path $P_{i,a} = (i, \ldots, b, a)$. Once the bias is found at each LP_i, it is also clear that LP_i can not send a message with a time stamp smaller than $\max(B_i, T_{\text{smin}_i})$ from LP_i along the path $P_{i,a}$. Consequently, the TNE, $T_{b,a}$, is computed as follows:

$$T_{b,a} = \min_{i \in V}(\max(B_i, T_{\text{smin}_i}));$$

where V is a set of LPs attached to each other by empty links and T_{smin_i} the smallest time stamp at LP_i. A priority queue Q is used to temporarily store the computed biases. At each step, the smallest bias is removed from the queue Q and used to compute the TNE.

Table 1. iPSC/i860 basic communication primitives

type	Primitives	Description
Blocking	csend	send a message and wait for completion
	crecv	receive a message and wait for completion
Non-blocking	isend	send a message
	irecv	receive a message
	msgwait	wait until completion for communication

A formal description of the DLTNE, a proof of its correctness and its complexity may be found in [13].

4 The parallel simulation

A parallel simulator testbed (PARADIS) [1] has been designed and implemented on the iPCS/i860 hypercube multiprocessor to evaluate the TNE/DLTNE suite of algorithms. The Intel iPSC[3]/i860 is a distributed memory multiprocessor, in which the processing elements are interconnected in a hypercube topology [16]. The host and node processors are 32 bit RISC processors running at 40 MHz, and each node has 8 MBytes of main memory. The node operating system supports asynchronous communication. In Table 1 we show the basic communication primitives provided by the communication system of the i860.

The scheduler program is executed on each processor. The events in each processor are processed in the order of increasing time stamps. The smallest time stamp in the processor is taken from the event list and code from the appropriate logical process is executed to simulate the event.

Several heuristics for the scheduling of TNE were investigated in the course of our experiments. We settled on the so-called lazy approach. (See [1] for details of the algorithms). In the lazy approach, TNE is called when a processor becomes idle, i.e. when it has no events to process. In the event that it does not unblock the processor a heuristic, the purpose of which is to avoid unnecessary calls to the deadlock detection algorithm, is called. Known also as an artificial deadlock avoidance algorithm [1], it examines empty links issuing from a neighboring processor through the simple expedient of back-tracking. Upon finding three successive empty links, a knot detection algorithm is called. In the event that a knot is discovered, DLTNE is called to break the knot. The algorithm, is an extension of [21] in that it permits transitions between blocked and unblocked processes. The artificial deadlock heuristic is quite successful at avoiding unnecessary calls to DLTNE, as we shall see.

[3] iPCS is a trademark of Intel Corporation

Table 2. Service time distribution used in experiments

Distribution	Expression to compute random value[4]	Lookahead[5]	LAR
Deterministic	1.0	1.0	1.0
Shifted Uniform	$0.1 + uni()$	0.1	6
Shifted Exponential	$0.1 - \log(uni())$	0.1	11.0
Bimodal	$0.95238\ uni() +$ *if* $uni() < 0.1$ *then* 9.5238 *else* 0	0.1	10

5 Simulation experiments

In our experiments we selected a queueing network model with an FCFS service discipline as a benchmark. Earlier simulation studies [9, 19, 26, 30] showed that the performance of a simulation strategy is sensitive to the topology of the simulated network. Several network topologies for inter-processor networks have been proposed in the literature [7]. We have selected the toroid network topology primarily because it provides a stress test for the algorithms. This is a result of the large number of cycles present in the torus, thereby increasing the number of possible deadlocks. Other justifications for the use of the torus queueing network topology is that it doesnot contain any inherent bottlenecks and is used in multiprocessor systems such as PACS [15] and MOPAC [31].

In our model, each network node is a server with infinite capacity input queues. A server, modeled by an *LP*, removes an event from one to the queues and starts service. The service time distribution has been selected from one of the following distributions (see Table 2): deterministic, shifted uniform, shifted exponential, and bimodal. The choice of these distributions is predicted upon their use in evaluating serial simulations and are the same ones as used in [8]. When service completes, the event is forwarded to one of its neighbors, selected with equal probability. As opposed to the previous studies [9, 13, 27, 30] in our experiments, the message population is not constant during the simulation. In our model, messages are deleted at the destination (determined randomly). We selected an arbitrary number of messages (three on each queue) with which to start the simulation in order to control the amount of available parallelism. The message population as well as the number of LPs determines the amount of parallel activity that can occur in the simulation.

The efficiency of a parallel simulation depends a great deal upon its ability to predict the next service time at each *LP*. TNE looks ahead into simulated future to compute the lower bound on the next event time for all empty links within a processor. This results in the scheduling of new events by unblocking some of the *LP*s. This notion is captured in the idea of the lookahead of a service distribution. Lookahead is defined to be the minimum service time

[4] uni( ) returns a random real number uniformly distributed between 0 and 1

[5] Lookahead is defined as the minimum value for the distribution

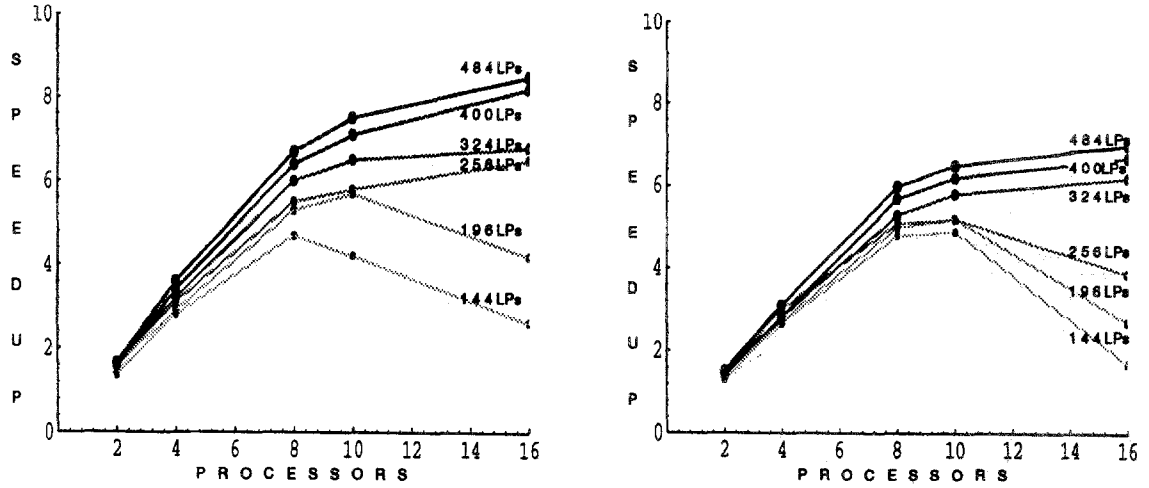


Fig. 4a, b. Speedup versus number of processors. **a** Deterministic distribution; **b** shifted uniform distribution

of the distribution, while the lookahead ratio (LAR) is the ratio of the mean of the service time of the distribution to the minimum service time [8]. Table 2 describes the distributions we use, in particular their lookahead and their lookahead ratio. As pointed by [8], the LAR plays a significant role in performance of conservative algorithms.

Nicol [24] proposed improving the lookahead ability of a process by precomputing service times of future events. In our model, for each process we precompute the next service time for each link subsequent to sending a message on that link.

A variety of work-loads were simulated, as characterized by service distributions, the number of processors and the lazy approach discussed earlier. In all cases, we varied the number of processors from 2 to 16 and the dimension of the torus from 12×12 to 22×22 (i.e., from 144 nodes to 484 nodes). The experimental data which follows was obtained by averaging several trial runs on the parallel simulation testbed PARADIS [1].

We divide our discussion of the experimental results into two sections. In the first section we assess the speedup obtained by TNE relative to a fast sequential simulator. The sequential simulator makes use of a splay tree, because it is among the fastest methods for implementing an event list [18]. (Implementation details are described in [1]). In the course of our discussion we point out significant factors affecting TNE's performance. Following this, we discuss the overhead implicit in the use of TNE, pointing out the importance of inter-processor deadlocks.

5.1 Speedup

We define the speedup $SP(n)$ to be the time ET_1 required for a the sequential simulator to perform the simulation on one processor divided by the time ET_n required for the parallel simulation to perform the same simulation on n processors, i.e. $SP(n) = ET_1/ET_n$.

Figure 4 portrays the speedups obtained for the deterministic and shifted uniform distributions, for 7 population levels ranging from 64 LPs to 484 LPs. As we can see, TNE yields good speedup curves for both distributions. This is largely due to the TNE's efficiency in unblocking LPs and breaking deadlocks, as will be demonstrated in Sect. 5.2. We also note that lookahead plays an important role in TNE's success as does the total number of LPs in the model. (The deterministic distribution has an LAR of 1 while the shifted uniform has an LAR of 6).

We observe significant speedups up to 10 processors for all population levels followed by a gradual flattening of the curves, the result of a decreasing number of LPs per processor. TNE is executed in each processor, without knowledge of the state of LPs in other processors. Hence it does not have knowledge of event timestamps or of local simulation times of LPs in other processors. Consequently, when the number of LPs per processor decreases, TNE can unblock fewer processes. The situation becomes more dramatic at a population level of 196 LPs, when we see a decreasing speedup after 10 processors. The number of LPs per processor becomes so low that TNE is no longer able to increase the parallelism of the simulation by unblocking LPs, coupled with the fact that the number of deadlock detection/breaking messages has increased.

Communication/execution time percentages are reported in Table 3. We compute this percentage by dividing the average time which an LP spends in sending messages to or receiving messages from neighboring processors divided by the execution time of the simulation. As can be seen from Table 3, the inter-processor communication plays an important role [14] in determining the speed of the simulation when we employ a large number of processors.

Table 3. Communication/Execution time percentage (deterministic distribution)

Number of processors	144 lps %	256 lps %	400 lps %	484 lps %
2	5.13	5.1	5.9	5.7
4	6.32	6.2	7.2	7.3
8	7.1	7.4	7.51	7.63
16	15.9	18.7	20.5	22.1

Recall that we make use of a simple static mapping strategy of LPs to processors, as in [8], in which the torus is subdivided into grids, and in which the LPs in the same grids are allocated to the same processor. This grid like structure does not exploit the positive characteristics of the hypercube machine, i.e. rich interconnectivity and powerful message passing with transparent routing. More sophisticated mapping strategies can yield better performance [2, 24]. Several load balancing strategies as well as the relationship between the execution time and the interprocessor communication time are under investigation [2].

When the lookahead properties of the distributions become worse, as is the case of the shifted exponential distribution or the bimodal distribution, speedups were not as dramatic. (Graphs may be found in [1]). This sort of behavior for conservative simulations has been pointed out in [8]. More generally, the ability of TNE to exploit the inherent parallelism of a model is constrained by the same factors which apply to any conservative simulation – the lookahead of the service time distribution, the number of LPs in the model, and the number of processors employed in the simulation.

Recall that we are subjecting TNE to a stress test in these experiments – the topology contained a large number of cycles and the message population LPs was low (3 per link). In this light, we can conclude that TNE performed quite well. The comparisons of TNE with an optimized Chandy-Misra algorithm in the next section further underscore this conclusion.

5.2 Overhead

Our first way of assessing the overhead engendered by TNE is by measuring the overhead of inter-processor deadlocks. We follow this with a description of a heuristic for avoiding calling the distributed knot detection algorithm too frequently. Our second measure of overhead is directed towards the fraction of time spent executing TNE. Finally, we examine the extent to which TNE is actually successful in unblocking LPs.

We define the inter-processor deadlock ratio (IPDR) as the number of messages involved in deadlock detection and breaking divided by the number of event messages. As we can see from Fig. 5, the behavior of the IPDR mirrors that of the speedup curves. We see a maximum IPDR of 10% for 4 processors for both distributions, followed by a gradual increase to maxima of 15% and 20% respectively at 10 processors, followed by a sharper ascent to maxima of 20% and 35% at 16 processors.

Recall that a heuristic algorithm was employed to avoid calling the deadlock detection algorithm unnecessarily. The heuristic examines empty links issuing from a neighboring processor. Upon finding 3 successive empty links, the deadlock detection algorithm is called. Otherwise, the simulation continues. Table 4 shows the success this heuristic had in avoiding unnecessary overhead. In fact, the heuristic (known as the artificial deadlock avoidance algorithm) is absolutely critical to the performance of TNE. Without it, the overhead generated by frequent calls to the deadlock detection algorithm would simply overwhelm TNE.

The second measure of the TNE overhead is the fraction of time spent executing the TNE algorithm, which we call the TNE time ratio. In the course of our experiments, we observe that the TNE time ratio tends to increase with an increasing number of processors. The TNE time ratio varied between 10% and 25% for the deterministic and the shifted uniform distributions. The interested reader might wish to consult [1] for further details and for the graphs.

We now turn to a discussion of the success TNE has in unblocking LPs. We employ two measures of efficiency in the course of our discussion, the TNE success ratio (TNESR) and TNE efficiency (TNEF).

We define the TNE Success Ratio, $TNESR(n)$, to be the percentage of calls to TNE which are successful (i.e., which unblock a blocked LP) on n processors. TNE is called successfully if it unblocks at least one LP when it is

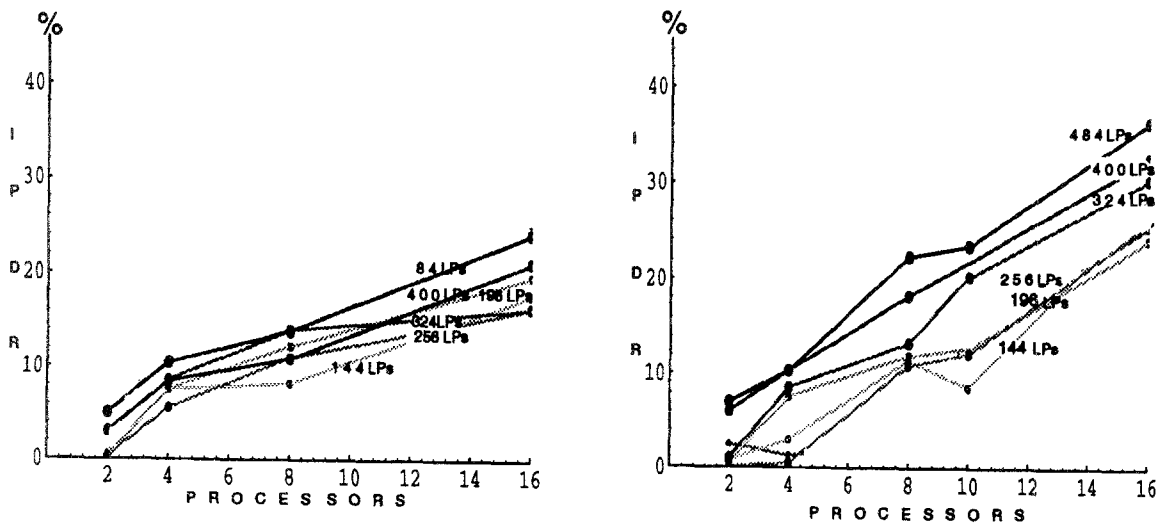


Fig. 5a, b. IPDR versus number of processors. **a** Deterministic distribution; **b** shifted uniform distribution

Table 4. Number of artificial deadlock detection calls compared to a number of knot detection calls. – Lazy approach

Service time Distribution	Nbr of processors	Nbr of processes	Nbr of calls of the artificial deadlock avoidance algorithm	Nbr of calls of the knot detection algorithm
	2	324	75	36
Deterministic	4	324	115	43
	8	324	185	60
	16	324	215	73
	2	196	76	28
Uniform	4	196	96	48
	8	196	112	65
	16	196	169	79

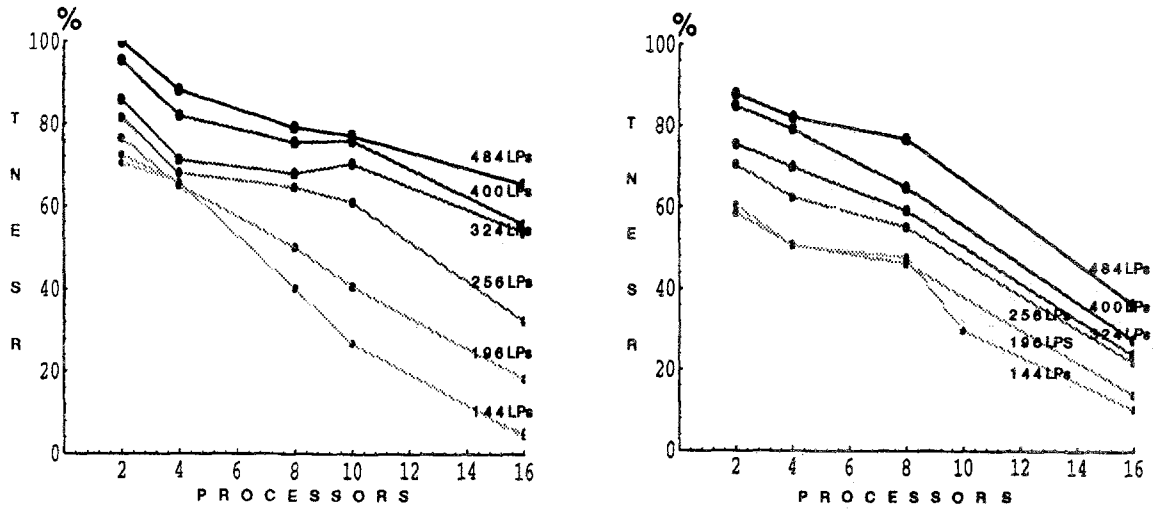


Fig. 6a, b. TNE success ratio versus number of processors. **a** Deterministic distribution; **b** shifted uniform distribution

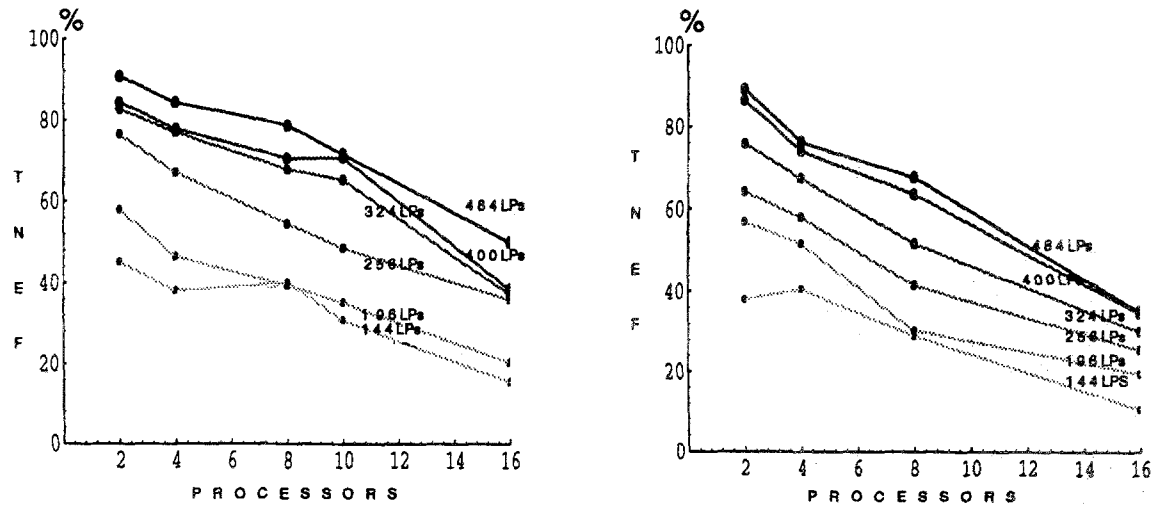


Fig. 7a, b. TNE efficiency versus number of processors. **a** Deterministic distribution; **b** shifted uniform distribution

called. Hence $TNESR(n) = \sum_1^n TNESR_i/n$; where n the number of processors and where $TNESR_i$ is defined as follows $TNESR_i = TNE_{\text{suc}_i}/TNE_{\text{tot}_i}$; TNE_{suc_i} is the number of times TNE is called successfully on processor (P_i); while TNE_{tot_i} is the number of times TNE is called.

Closely related to TNESR is the TNE efficiency $(TNEF(n))$, which measures the percentage of *LP*s that have been unblocked by a successful TNE algorithm executed on each processor during the simulation. This indicates how successful TNE was in increasing the parallelism, i.e. $TNEF(n) = \sum_1^n (TNEF_i)/n$; where n is the number of processors, and $TNEF_i$ is defined as the sum of the percentage of *LP*s unblocked by each successful TNE divided by the number of times TNE is called successfully.

We see in Figs. 6 and 7 that both measures decrease with an increase in the number of processors for all

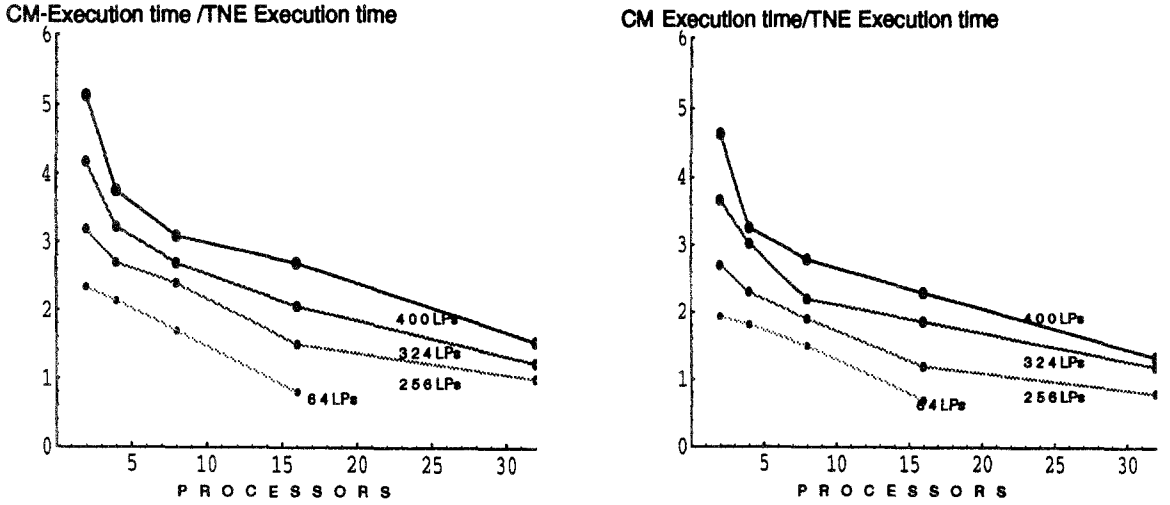


Fig. 8a, b. TNE compared to the Chandy-Misra approach. **a** Deterministic distribution; **b** shifted uniform distribution

population levels, for both distributions. (This applies to the bimodal and shifted exponential distributions as well [1]). For the higher levels of population, both measures of effectiveness are extremely high – e.g., a TNESR and a TNEF of 85% and 80% respectively for a shifted uniform distribution for 4 processors and a large number of processes. As TNE only makes use of information located within the confines of one processor, making use of more processors for a given population level simply decreases the ability of TNE to unblock an LP, as the information required to unblock the LP might be located in another processor. It is therefore tempting to speculate that scheduling TNE over a cluster of processors would improve TNE's efficiency when a large number of processors (and processes) are used.

6 TNE and Chandy-Misra, a comparison

In this section, we compare the execution times of the TNE algorithm to an optimized version of the Chandy-Misra (CM) algorithm. In [4], the authors employ null messages in order to avoid deadlocks and to increase the parallelism of the simulation. When an event is sent on an output link, a null message, bearing the same timestamp as the event message is sent on all other output links. As is well known, it is possible to generate an inordinate number of null messages under this scheme, nullifying any performance gain [9].

In order to increase the efficiency of this basic scheme, we employ the following approach. In the event that a null message is queued at an LP and a subsequent message (either null or event) arrives on the same channel, we overwrite the (old) null message with the new message. We associate one buffer with each input channel at an LP to store null messages, thereby saving space as well as the time required to perform the queueing and de-queueing operations associated with null messages.

Once again we make use of the torus queueing model described in Sect. 5 as an experimental test-bed. In Fig. 8 we present graphs of the ratio of time required for the CM algorithm to complete a simulation compared to the TNE algorithm. The comparison is presented for 4 population levels for the deterministic (Fig. 8a) and the shifted uniform distribution (Fig. 8b). We employed 4, 8, 16 and 32 processors in making this comparison.

Figure 8 makes clear how large an improvement TNE is over our optimized version of the CM algorithm. If we confine ourselves to less than 10 processors we see that TNE is from 2 to 5 times as fast as the CM algorithm, (with the exception of the population of 64 LPs). As we employ more processors, the ratio decreases to the 1.5–3 range. The decrease in TNE's effectiveness is due to an increasing number of processors for the same model. As we have already seen, the shortest path algorithm at the heart of TNE is executed in each individual processor and can therefore only make use of the information in the processor itself. By decreasing the number of LPs in each processor we decrease the information available to the algorithm. In addition to this, increasing the number of processors increases the number of executions of the deadlock detection and breaking algorithms. When 64 LPs are employed, TNE's advantage over the optimized version persists up to 8 processors.

7 Conclusion

This paper has centered about a performance study of the TNE algorithm as described by Groselj and Tropper in [13]. We computed the traditional speedup curves relative to an efficient uniprocessor simulator and identified important factors which influence TNE's behavior. We also compared its performance to an optimized version of the venerable Chandy-Misra algorithm [4].

TNE relies upon a shortest path algorithm which is independently executed in each processor in order to unblock LPs. A deadlock resolution algorithm is employed for inter-processor deadlocks. The factors which play a role in TNE's behavior influence any conservative

simulator. These include the lookahead of the service distribution, the number of LPs in the model, and the number of processors used.

The results obtained demonstrate conclusively that TNE is successful in speeding up a parallel simulation. In our experiments we observed a good speedup up to 10 processors (see Fig. 4), followed by a gradual flattening of the curve. We attribute this to the smaller amount of information supplied TNE when the model is spread out among more processors, as well as an increased number of calls to the deadlock resolution algorithm. Comparisons to the optimized version of the CM algorithm were very much in favor of TNE. For less than 10 processors, TNE was 2–5 times as fast as the CM algorithm and 1.5–3 as fast when more than 10 processors were employed for the same population of LPs. The only exception to this was when the LP population was too small (64) to supply adequate information to TNE.

We close the paper with the following observations. (1) An implementation on a shared memory machine would not require the use of a separate deadlock resolution scheme because TNE can break deadlocks within its purview, leading to a simpler implementation and better performance. (2) In a distributed memory multicomputer, it is possible to employ a shortest path algorithm [5] over several nodes of the multicomputer instead of the present approach of confining TNE to a single processor. This would be of particular interest when a large number of processors are employed. (3) Finally, inter-processor communication is a basic factor to contend with on any distributed memory machine [14]. We are evaluating its relationship with the execution time of the simulation in the quest for a load-balancing algorithm [2].

Acknowledgement. We would like to thank Richard Fagen, Director of the Campus Computing Organization at the California Institute of Technology, and his staff for providing us an encouraging and comfortable computing environment and making our stay enjoyable. Thanks are in order to Paul Messina, director of the CCSF at CalTech for providing us with super-computer access.

References

1. Boukerche A, Tropper C: Performance analysis of distributed simulation with clustered processes. Proc 1991, SCS Multiconference on Advances in Distributed Simulation, Anaheim, Calif. Jan. 1991, pp 112–120
2. Boukerche A, Tropper C: A static partitioning and mapping algorithm for conservative parallel simulations. In: Proc of the 1994 Parallel and Distributed Simulation, vol 24, no 1, July 1994, pp 164–172
3. Bryant RE: Simulation of packet communications architecture computer systems. MIT/LCS/TR-188, Masachusetts Institute of Technology, Cambridge, Mass, Nov 1977
4. Chandy KM, Misra J: Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans Softw Eng (SE-5) 440–452 (1979)
5. Chandy KM, Misra J: Distributed computation on graphs: shortest path algorithms. Commun ACM 25(11): 198–206 (1982)
6. Cote C, Tropper C: On distributed and pseudosimulation. Proc of the SCS Multiconf on Parallel and Distributed Simulation, SCS Simulation Serie, vol 24, no 3, Jan 1992, pp 97–106
7. Feng TY: A survey of interconnection networks. Computers, Special Issue on Interconnection Networks, Dec 1981, pp 12–27
8. Fujimoto RM: Performance measurements of distributed strategies. In: Proc 1988, SCS Multiconference on Distributed Simulation, vol 19, no 3, Feb 1988, pp 14–20
9. Fujimoto RM: Parallel discrete event simulation. In: Proc of the 1989 Winter Simulation Conference, 1989, pp 19–28
10. Groselj B, Tropper C: The time-of-next-event algorithm. In: Proc of the 1988 Distributed Simulation Conference, SCS Simulation Series, vol 19, No 3, 1988, pp 25–29
11. Groselj B, Tropper C: A deadlock resolution scheme for distributed simulation. Proc of the SCS Eastern Multiconf, SCS Simulation Series, vol 21, No 2, 1988, pp 108–112
12. Groselj B: CSP cocktail party and its performance on a network of workstations. In: Proc of the 1990 Distributed Simulation Conference, vol 22, no 1, 1990, pp 70–73
13. Groselj B, Tropper C: The distributed simulation of clustered processes. Distrib Comput 4: 111–121 (1991)
14. Gustafson JL, Monty GR, Benner RE: Development of parallel methods for 1024-processor hypercube. SIAM J Sci Stat Comput 9(4): 609–638 (1988)
15. Hoshino et al.: PACS: a parallel microprocessors array for scientific calculation. ACM Trans Comput Syst 1(3): 195–221 (1983)
16. Intel: iPSC/860 Users Guide. Intel Scientific Computers, Beaverton, 1990
17. Jefferson DR: Virtual time. ACM Trans Prog Lang Syst 77(3): 405–425 (1985)
18. Jones DW: An empirical comparison of priority-queue and event-set implementations. Commun ACM 29(4): 300–311 (1986)
19. Lin YB, Lazowska ED: Conservative parallel simulation for systems with no lookahead prediction. TR 89-07-07, Department of Comput Science and Engineering, University of Washington, 1989
20. Lubachevsky BD: Efficient distributed event-driven simulations of multiple loop networks. Commun ACM 32: 111–123 (1989)
21. Misra J, Chandy KM: Distributed graph algorithm: knot detection. ACM Trans Prog Lang Syst 4: 678–686 (1982)
22. Misra J: Distributed discrete-event simulation. ACM Comput Surv 18(1): 39–65 (1986)
23. Nandy B, Loucks WM: An algorithm for partitioning and mapping conservative parallel simulation onto multicomputer. Proc of the SCS Multiconf on Parallel and Distributed Simulation, SCS Simulation Serie, 1992, pp 139–146
24. Nicol DM: Parallel discrete-event simulation of FCFS stochastic queuing networks. In: Proc of the ACM SIGPLAN Symposium on Parallel Programming, Environments, Applications, and Languages, Yale University, July 1988
25. Peacock JK, Wong JW, Manning EG: Distributed simulation using a network of processors. Comput Networks 3(1): 44–56 (1979)
26. Reed DA, Malony A: Parallel discrete event simulation: the Chandy-Misra approach. In: Proc 1988 SCS Multiconf on Distributed Simulation, 1988, pp 8–13
27. Righter R, Walrand JC: Distributed simulation of event systems. Proc of the IEEE, vol 77, no 1, Jan 1989, pp 99–113
28. Seethalakshmi M: Performance analysis of distributed simulation. MSc Report, Computer Science Department, University of Texas, Austin, Texas, 1978
29. Su WK, Seitz CL: Variants of the Chandy-Misra-Bryant distributed discrete event simulation algorithm. In: Proc of the SCS Multiconference on Distributed Simulation, vol 21, no 2, March 1989
30. Wagner DB, Lazowska ED, Bershad B: Techniques for efficient shared memory parallel simulation. Proc 1989 SCS Multiconf on Distributed Simulation, 1989, pp 29–37
31. Wong-Hua L, Miroslaw M: MOPAC: A partitionable and reconfigurable multicomputer array. Proc of the 1983 International Conference on Parallel Processing, 1983, pp 506–510