PORTABILITY AND PARALLELISM WITH "LIGHTWEIGHT p4"

Adam Back and Stephen Turner,

Department of Computer Science,
University of Exeter,
Prince of Wales Road,
Exeter EX4 4PT England
Email: steve@dcs.exeter.ac.uk

November 29, 1993

Abstract

In this paper we discuss the use of cluster based computing as a basis for portable, heterogeneous parallel computing. We describe p4, a cluster based communications library which originates from Argonne National Laboratory, and our implementation of p4 on the transputer architecture. We present an extension to p4, "lightweight p4", which we designed for use in our research into automatic program parallelisation based on optimistic execution. We briefly discuss this research and the underlying techniques which are drawn from parallel discrete event simulation.

1 Introduction

The p4 parallel library [2, 3], which originates from Argonne National Laboratory, provides a portable programming model for a large set of parallel machines. The model combines message passing with shared memory to form a cluster based model of parallel computing. In this paper we discuss the suitability of p4 as a model for general purpose parallel computing and the design issues of our implementation of p4 on the transputer architecture.

We present an extension to p4, "lightweight p4", designed to allow programming using lightweight pre-emptive processes while retaining portability. These features are required for our research into the use of optimistic mechanisms, based on the virtual time paradigm [13], in general purpose parallel computing[1]. Our reasons for choosing p4 (which is related to PARMACS [9]) rather than other message passing systems such as PVM [18] were based on performance [8] and the suitability of the p4 programming model for our research. We are closely following the efforts to define a message passing interface standard [7], but this work is still at a draft stage.

[1] The "lightweight p4" library also forms the basis of other research projects at Exeter University.

1.1 Clusters

The cluster model in p4 groups together processes into clusters. All p4 processes can communicate via message passing, and p4 processes within the same cluster can share memory.

The p4 model of computation has been implemented by other researchers on Unix workstations, shared memory multiprocessors, distributed memory multicomputers and virtual shared memory parallel computers. The p4 library provides heterogeneous operation across these architectures, so that a single p4 program may be run on a combination of architectures simultaneously.

With standard p4 on Unix workstations, the p4 process corresponds to a Unix process, and a cluster is formed by a set of Unix processes sharing memory. Communication between processes in the same cluster is implemented using shared memory, and between processes in different clusters by using sockets.

With shared memory multiprocessors, the shared memory can be implemented in hardware between p4 processes which are running in parallel, as opposed to the time sharing of processes on a single processor Unix machine. On a distributed memory architecture, the p4 processes within a cluster will all reside on the same processor. Message passing is implemented using the underlying communications mechanism provided by the hardware.

With a virtual shared memory parallel computer, two options are possible:

- to have processes in the same cluster placed on the same processor, where virtual shared memory is used for message passing only;
- to make use of the virtual shared memory provided by the architecture to allow processes in the same cluster to be placed on different processors.

Thus with virtual shared memory, the choice of clusters could be based on the structure of the application, rather than on the characteristics of the hardware.

The p4 system includes monitors within the programming model to control shared memory. It provides synchronous and asynchronous message passing facilities between p4 processes. Messages may be typed: it is possible to request a message of a particular user defined type, and the p4 system will automatically buffer messages of different types until they are requested. Similarly, it is possible to request a message from a certain p4 process, and messages from other processes will be buffered until required.

Facilities are provided to define a set of machines and a description of how clusters map on to those machines. This configuration is pseudo-dynamic in that it can occur at run time, but it must remain fixed after the p4 library has been initialised. There is also provision for dynamic process creation, although such processes are not able to communicate via message passing.

A global barrier mechanism and a cluster barrier mechanism are provided by p4, implemented by message passing and semaphores respectively. With the cluster barrier mechanism, it is possible to wait for all, or a subset of the cluster processes to synchronize. With these facilities, it is possible to use p4 in a similar way to the Bulk Synchronous

Parallel model [20]. However, the p4 model emphasises locality based computation, whereas the BSP model assumes a global memory with uniform access. We feel that this assumption of the BSP model may not be realistic in heterogeneous parallel environments.

1.2 Heterogeneous operation

The p4 system provides a framework within which it is possible to have a combination of machines of different classes presenting a common model. It allows us to execute programs on combinations of parallel machines, workstations and multiprocessors. In this way we could develop an application which would run simultaneously on a set of p4 clusters. Some clusters could be located on shared memory multiprocessors (for example SGI multiprocessors), some on Unix workstations (for example SUN Sparcstations) and some on distributed memory parallel machines (such as transputers).

To allow messages to be passed between machines of different architectures, p4 uses the XDR (eXternal Data Representation) library. XDR [15] provides a standard representation for float, double, int, long into which messages must be translated on send and from which they must be translated on receive. In this way messages can be passed between little-endian and big-endian machines, and between machines which do and do not use the IEEE floating point format.

Our transputer implementation is thus able to operate in a heterogeneous manner with Unix workstations, and shared memory multiprocessors. Transparently to the p4 applications programmer, communication in our example network will be taking different paths depending on the location of the p4 processes involved:

- Transputer ↔ Transputer, via transputer hardware links,
- Transputer ↔ Unix machine, via transputer socket library and Unix sockets.
- Unix machine ↔ Unix machine, via Unix socket library.

2 The Transputer Implementation

The transputer implementation differs from the generic Unix implementation in two areas: in the way processes are created, and the message passing mechanism.

2.1 Fork

The Argonne implementation of p4 makes use of the standard Unix fork, which means that processes have their own copy of the data segment and of the heap segment.

The Unix fork does not fit well with the T805 transputer architecture as that architecture is unsegmented. It is possible to provide a fork using a compiler which generates extra code to emulate segmented memory, as is done in the Helios operating system [16]. However there are two costs incurred here: the overhead of the software emulated segments, and the memory and time overhead of copying forks. The T9000 series transputers

[14, 19] have a segmented architecture, and will remove the need for software segment emulation.

The code generation model the current Inmos C compiler provides allows us to construct a fork function which has a subset of the standard Unix fork functionality. We are able to copy data items with immediate values, but not pointer values or heap values. That is, after a fork call any pointers will still point to the parent process's data items, or heap space, while immediate data values are copied correctly. Also it is restricted to copying the data values within a compilation unit, external data items will still refer to the parent process's data values.

Our transputer fork function is not meant to be of use to the p4 applications programmer: its main purpose is in implementing p4 itself, where the p4 process's state information is stored in a structure `p4_local`. We use this fork on the compilation unit containing the `p4_local` definition. This allows us to have dynamically creatable processes, each with a different global pointer to their `p4_local` structure. This structure contains such information as the process's id, state information for its XDR buffers, etc.

In this way, we are able to retain the transputer's advantage of fast dynamic process creation, and we have minimised the memory overhead. The result of these implementation decisions is that the p4 model now has only a `p4_shmalloc()`: the function which allocates memory from the shared memory area. If an application uses `p4_malloc()`, it will have the same functionality as `p4_shmalloc()` and the memory it allocates will not be copied for p4 processes.

Another implication for p4 programs is that the data segment is no longer copied. These restrictions do not present a problem as far as our intended use of p4 is concerned. If it is desired to port other p4 applications onto the transputer, the restrictions can be partly overcome by using the preprocessor described in section 4.2 to implement separate data segments for each process. This preprocessor groups together all global and static data items into a structure, and modifies all accesses to these data items to go indirectly through a pointer to this structure. In this way, when a process is forked it is possible to take a copy of its data segment by copying the data segment structure.

2.2 Message Passing: Channels

The generic Unix implementation of p4 uses sockets for communication. Vendor specific communication libraries are used for communication on parallel machines with specific communication hardware.

Message passing in p4 provides point to point communication, whereas the transputer hardware assumes placed channel communications. On the T805 transputer, the capabilities of the hardware are limited to the placing of only one communications channel on each of the hardware links. Furthermore, the hardware provides no facilities to send a message which would have to traverse several hardware links.

The T9000 transputer/C104 router provides a hardware solution to these problems [10, 19, 14]. On the current generation of transputers, several software solutions exist: Tiny [5], the Southampton VCR (Virtual Channel Router) software [6] and the Inmos VCR [12]. We have implemented the p4 message passing calls in terms of the Inmos

VCR system, which provides virtual channels which can be placed between processes on any processor in a transputer network, the necessary through routing and multiplexing being performed by the VCR software.

In order to implement p4's point to point communication in terms of virtual channels we provide a configuration which links each processor to each other processor in the network. Then we provide a multiplexor and a demultiplexor process for each virtual channel. The multiplexor for a particular channel will forward all messages destined for the remote processor via the virtual channel.

The steps involved in a message passing asynchronously from process 1 on processor A to process 2 on processor B are as follows. There will be a virtual channel V_{AB} from processor A to processor B, and a multiplexor M_B on processor A; also a demultiplexor D_A on processor B (see figure 1).

- Process 1 inserts the message into multiplexor M_B's queue
- Process 1 resumes execution
- Multiplexor M_B forwards the message to demultiplexor D_A via virtual channel V_{AB}
- Demultiplexor D_A receives the message and inserts it in its message queue
- Process 2 removes the message from demultiplexor D_A's message queue

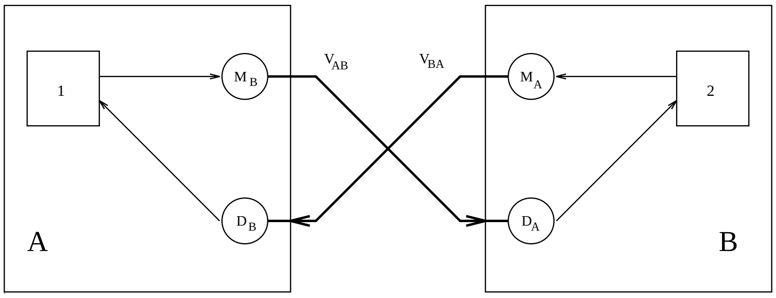

Figure 1: P4 communications

There is however a problem with this communication scheme as it stands: consider processes 1 and 3 on processor A; processes 2 and 4 on processor B. Process 1 sends a large message to process 2 which ties up the virtual channel for a significant amount of time. Now process 3 sends a small message to process 4, but the transmission of this message must wait until the large message has been delivered.

To alleviate this problem we place two virtual channels in each direction between each pair of processors: one for large messages and the other for small messages. Since there are now two paths a message can take depending on the message size, this could mean that messages no longer arrive in the same sequence as they were sent[2]. To avoid this problem we use information about the number of undelivered messages for a process on

[2]Consider processes 1 and 2 on processors A and B respectively, where process 1 sends two messages to process 2, a large message followed by a small message. It would be possible for the small message to be delivered to process 2 before the large message even though it was sent later.

the large and small message queues for each multiplexor. If there are any undelivered large messages then we delay sending a small message until the large message has been sent. Similarly if there are any undelivered small messages, we delay sending large messages.

3 Performance

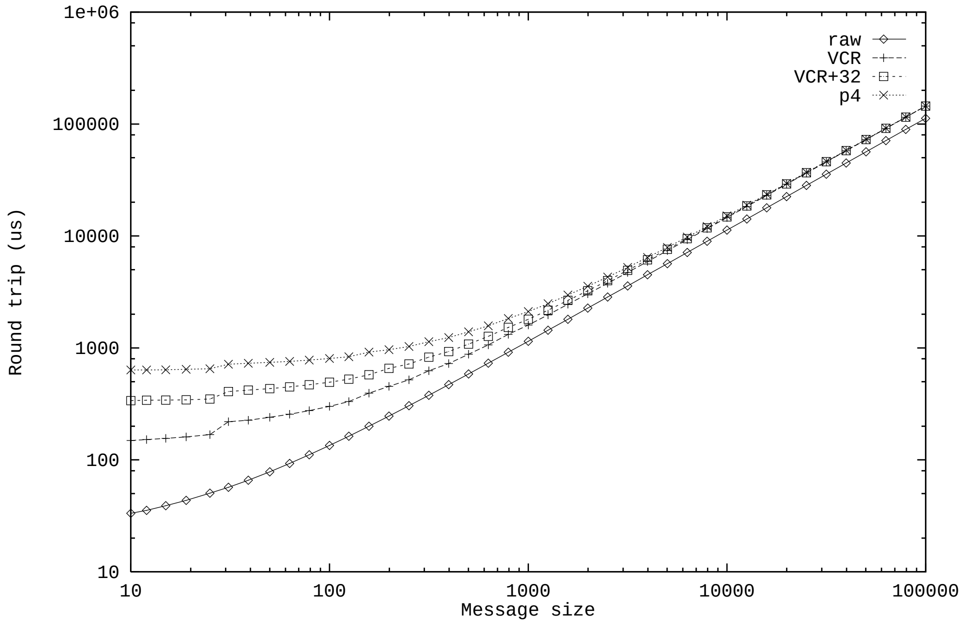

Figure 2: "Ping" 2 nodes: round trip time

Figure 2 demonstrates the efficiency of the p4 system as compared to the basic Inmos C communications library using virtual channels. These times are obtained with a simple "Ping" application where a message is sent from one transputer to an adjacent transputer and then back to the originator. This is repeated a large number of times in order to obtain accurate measurements. It can be seen that the communications overhead of p4 is reasonable, in view of the additional functionality over the virtual channel routing software that p4 provides.

The lines labelled "VCR" and "raw" show the performance of fixed sized messages with and without the Inmos VCR. The performance difference between raw and VCR messages is due to packetization and multiplexing of multiple "virtual" channels on a single hardware link. "VCR+32" shows the performance that would be achieved with variable sized messages where a 32 byte header is sent as a separate message before the actual message. This is the size of the header used by p4 and would contain in p4 information about the message size, type, sender's p4 process id, whether XDR is to be used etc. This is shown for comparison with the actual p4 communication performance labelled "p4" to distinguish the overhead p4 incurs for variable size packets from the extra overhead due to buffer management and the provision of typed messages.

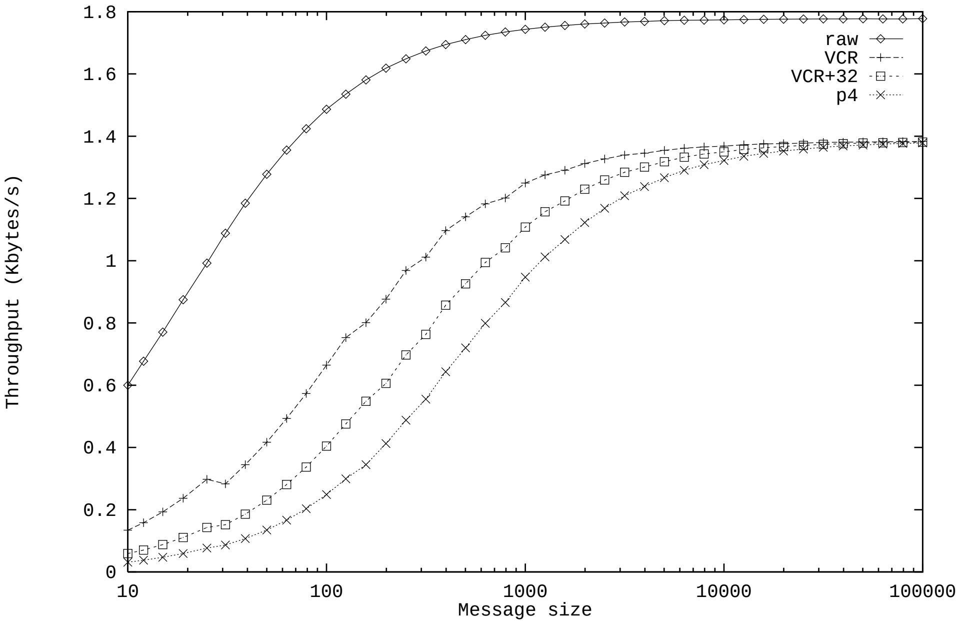

Figure 3: "Ping" 2 nodes: throughput

Figure 3 shows the corresponding bandwidth achieved for p4, VCR+32, VCR and raw communication, using the same application.

4 The "Lightweight p4" extension

This extension was designed to enable us to use p4 as a portable base for our research into the use of optimistic parallel simulation mechanisms in general purpose computing. This has the requirement of being able to control the scheduling policy at run time. We wished to be able to do this in a lightweight process environment, but at the same time we did not wish to lose the portability of our system. In an attempt to satisfy both of these goals we designed the following extension to p4.

We provide a lightweight process programming library where each process has a priority level which determines the scheduling. Processes with a priority level equal to that of the highest runable process are executed using a round robin policy. When processes at the current highest priority all become blocked or die, processes at the next lower priority run. If at any time a higher priority process becomes ready, the lower priority process is pre-emptively descheduled. The library is able to pre-emptively schedule and deschedule processes within two transputer time-slices. That is, if a higher priority process becomes ready while a lower priority process is running, the lower priority process will run for at most two time slices before being descheduled.

The library provides facilities to suspend, resume, and kill processes. It also allows the priority of each process to be altered at run time.

4.1 Scheduler: Transputer implementation

A multi-priority scheduler with the capability of pre-emptively descheduling processes was constructed based on the work of Shea *et. al.* [17]. A library was constructed which allows processes to be externally suspended, resumed and killed, and allows control over each process's priority.

Each process has an id and an entry in a process table containing information about the process's priority, workspace start and workspace size. The provision of a process id is used to externally control processes:

```
int GetId();                                /* current process id */
int ProcRunAt( Process*, int priority );    /* returns new process id */

void SetProcPriority( int priority, int id );
int GetProcPriority( int id );

SuspendProc( int id )
ResumeProc( int id );
KillProc( int id );

boolean Alive( int id );                    /* suspended or runable */
boolean Active( int id );                   /* runable */
```

4.2 Scheduler: Generic Unix implementation

On a Unix machine, we enable the generation of a shared memory lightweight process application by using a preprocessor. This preprocessor was developed using NewYacc [21] and takes C source code as input and outputs a modified program with all global and static variables grouped in a structure. This structure can then be placed in a shared memory segment which is shared by a set of Unix processes created by fork.

The standard Unix fork makes a duplicate copy of the data segment and heap segment of the forking process. This differs from the desired lightweight process creating fork in that a lightweight fork would copy neither the data segment nor the heap segment as both of these would be shared.

The copying of the heap is simple to avoid: we obtain a shared memory segment and use this for `mallocs`, then processes forked with the standard fork will not copy the shared heap segment. To avoid the copying of the data segment, the preprocessor groups together all data which would normally be placed in the data segment and creates a structure containing these data items. Next it declares a pointer to this structure and allocates this from the shared memory segment. This allows the the data segment area to be shared by the parent and child process after a standard Unix fork[3].

[3] Note that the same preprocessor can be used to the opposite effect on the transputer: the data segment structure can be copied when a process is forked if the assumptions of "lightweight p4" are found to be too restrictive.

In this way we can build a completely generic implementation of the lightweight process package, but there is scope for improved efficiency with some implementations of Unix by making use of vendor specific threads packages, or by writing the necessary context switching code for a given architecture.

5 General Purpose Optimistic Computing

As mentioned above, we are intending to use "lightweight p4" as a portable basis for research into the use of optimistic mechanisms in general purpose parallel computing. These techniques are based on ideas from parallel discrete event simulation.

5.1 Parallel Discrete Event Simulation

In discrete event simulation the physical system can be modelled in terms of events, each of which corresponds to a state transition of an object in the physical system. Simulation events each have a time-stamp which corresponds to the time at which the event would occur in the system being modelled. A sequential simulator proceeds by taking the event with the lowest time-stamp and simulating its effect: this may alter the state of the object being modelled, and also create further events which will be scheduled at some future simulation time. The simulation moves forwards in simulation time by jumping from the the time-stamp of one event to the next. This is in contrast to time-driven simulation methods where time moves forward uniformly. The simulation is complete when there are no more events to simulate.

In parallel discrete event simulation, the physical system is modelled by a set of processes which correspond to the interacting objects in the physical system. The interactions between the physical objects are modelled by the exchange of time-stamped event messages. Parallelism is achieved by placing these processes on the different nodes of a parallel computer.

Each process has a logical clock which denotes the simulation time of that process. A process's logical clock will be increased to the time stamp on each event message as it is processed. In this way the logical clock of each process will advance according to the time-stamps of events. The progress of the simulation is measured by the global virtual time (GVT), which is the minimum of the logical clocks (and time-stamps of messages in transit).

We must ensure that events are simulated in non-decreasing time-stamp order. To see why this is necessary, consider a process receiving a message which has an earlier time-stamp than the logical clock time. This means that the message was sent by an event which should have been executed earlier, and events have been executed past this which could have been affected by that event. We cannot process the late event as this would amount to the current event being affected by events in the future.

This is known as a causality error, from the cause and effect principle: the fact that events in the future cannot affect events in the past. There are two approaches to ensuring that causality is not violated in the simulation: conservative and optimistic.

5.2 Conservative vs Optimistic Simulation

Conservative approaches [4] avoid the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is safe to process an event. This will be when all events which could affect the event in question have been processed.

Optimistic methods [13] use a detection and recovery approach: causality errors are detected, and a roll-back mechanism is invoked to recover. A roll-back will be required when a causality violation is detected due to an event message arriving too late (this is known as a *straggler*). The roll-back must restore the state of the process in question to a point in time before the time-stamp of the straggler. To enable this to occur, processes must periodically save their states. After the roll-back, execution resumes from that point in time. There is a balance which must be achieved in deciding the frequency of state saving. Too often and state saving will become too much of an overhead, too infrequently and it is necessary to roll back further. In the course of rolling back a process, anti-messages are sent to "undo" the effect of previous event messages which should not have been sent. The anti-message will annihilate the real message and possibly cause the receiving process to roll back. This may in turn require the sending of more anti-messages.

The optimistic approach is less restrictive and so potentially allows for more parallelism in the execution of the simulation. This is because it allows events to be processed out of time-stamp order even if there is a possibility of one event affecting the other. Optimistic methods work well if the number of cases where a causal relation actually exists between events is comparatively infrequent. However, there are overheads incurred in maintaining the information necessary for roll-back, and the roll-back itself consumes execution time. We have to be careful that the roll-backs are not too frequent or we could lose more than we gain.

5.3 General Purpose Programming

In applying the optimistic technique to general purpose parallel computing, the program becomes the "physical" system and the optimistic simulation of this system the execution of the program. The aims are to make use of more of the available parallelism than is possible with automatic parallelisation schemes based on static program analysis. Some work has been done by Bacon [1] on the optimistic execution of CSP (Communicating Sequential Processes) [11], but the use of optimistic execution as a parallelisation tool has been largely unexplored.

The optimistic execution of a program can parallelise some code which static program analysis would indicate was sequential. This is because the optimistic parallel execution can presume that a particular event corresponding, say, to a branch in the program will not occur. A conservative approach would have to take the worst case: if it is possible for the branch to be taken it would need to assume that it is always taken. This means that the optimistic approach can execute more of the program in parallel, but it also means that it may do some work which it later decides is wrong because of a causality violation. The chances of this happening can be reduced by limiting how far ahead a process may get from the global virtual time. In this way we can reduce the amount of work wasted due to later roll-back.

Our research is currently addressing the problem of time-stamp generation. Optimistic execution requires events to be time-stamped, and we are investigating time-stamp generation schemes which are able to create time-stamps for unbounded loops. Such a scheme will form part of a parallelising compiler which converts conventional code into a form which has calls to an optimistic run time system built upon lightweight p4.

6 Conclusions

In conclusion, we believe that the cluster based model of computation, as provided by p4, is a useful model of parallel computation for a wide range of applications. It presents the programmer with a uniform model which enables the development of efficient, portable applications that can be run in a heterogeneous environment.

Our results suggest that it is possible to retain this efficiency even on platforms such as the transputer which support a fast process switching mechanism and high speed communication links. Since transputer applications often involve a relatively high communication to computation ratio (compared to applications written for a network of workstations), this means that the code in the transputer version of the communications library has to be carefully written to avoid degrading the communications performance. If this is done, then using the p4 library to obtain portability will not prove to be a serious overhead.

This paper has also highlighted some of the incompatibilities between the Unix concept of a process and that provided by the transputer architecture. We have suggested a method of resolving these incompatibilities without sacrificing efficiency. The "lightweight p4" extension that we have presented makes use of the fast process creation and switching mechanism of the transputer and provides a convenient framework for our research into optimistic execution schemes. At the same time, the portability of our approach and the ability to run in a heterogeneous environment has been retained.

References

[1] D.F. Bacon and R.E. Strom. Optimistic parallelization of communicating sequential processes. *ACM SIGPLAN Notices*, 26(7):155–166, 1991.

[2] R. Butler and E. Lusk. User's guide to the p4 parallel programming system. Technical report, ANL-92/17, Argonne National Laboratory, 1992.

[3] R. Butler and E. Lusk. Monitors, messages, and clusters: the p4 parallel programming system. Technical report, Argonne National Laboratory, 1993.

[4] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Engineering*, S.E.5(5):440–452, 1979.

[5] L. Clarke and G. Wilson. Tiny: an efficient routing harness for the Inmos transputer. *Concurrency: Practice and Experience*, 3(3):221–245, 1991.

[6] M. Debbage, M.B. Hill, D. Nicole, and A. Sturges. The virtual channel router. *Transputer Communications*, 1(1):1–16, 1993.

[7] J.J. Dongarra, R. Hempel, A.J.G. Hey, and D.W. Walker. A proposal for a user-level message passing interface in a distributed memory environment. Technical report, ORNL/TM-12231, Oak Ridge National Laboratory, 1993.

[8] C.C. Douglas, T.G. Mattson, and M.H. Schultz. Parallel programming systems for workstation clusters. Technical report, Yale University, 1993.

[9] R. Hempel. The ANL/GMD macros (PARMACS) in fortran for portable parallel programming using the message passing programming model – users' guide and reference manual. Technical report, GMD, Sankt Augustin, Germany, 1991.

[10] A. Hipperson. T9000 virtual link performance: A comparison of short message performance on T805s and T9000s. Technical report, PARSYS Limited, England, 1993.

[11] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[12] Inmos. *ANSI C Toolset User Guide*, 1992.

[13] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[14] M.D. May, P.W. Thompson, and P.H. Welch. *Networks, Routers and Transputers*. IOS Press, 1993.

[15] SUN Microsystems. *Network Programming Guide*, 1990.

[16] Perihelion. *The helios parallel operating system*. Prentice Hall, 1991.

[17] K.M. Shea, M.H. Cheung, and F.C.M. Lau. An efficient multi-priority scheduler for the transputer. In *Proc. 15th WoTUG Technical Meeting (Aberdeen)*, pages 139–153. IOS Press, 1992.

[18] V. Sunderam. PVM: a framework for parallel distributed computing. *Concurency: Practice and Experiencs*, 2(4):315–339, 1990.

[19] University of Kent. *T9000 Systems Workshop*, 1992.

[20] L.G. Valiant. A bridging model for parallel computation. *Comm. A.C.M.*, 33(8):103–111, 1990.

[21] E.L. White, J.R. Callahan, and J.M. Purtilo. *The NewYacc User's Manual*.