THE DESIGN AND APPLICATION OF PARSIM—A MESSAGE PASSING COMPUTER SIMULATOR

ANTHONY SYMONS AND V. LAKSHMI NARASIMHAN[1]

***Abstract*—Currently many interconnection networks and parallel algorithms exist for message passing computers. Users of these machines wish to determine which message passing computer is best for a given job, and how it will scale with the number of processors and algorithm size. This paper describes a general purpose simulator for message passing multiprocessors (Parsim), which facilitates system modelling. A structured method for simulator design has been used which gives Parsim the ability to easily simulate different topologies and algorithm combinations. This is illustrated by applying Parsim to a number of algorithms on a variety of topologies. Parsim is then used to predict the performance of the new IBM SP2 parallel computer, with topologies ranging up to 1024 processors.**

***Keywords*—Parallel Distributed Computing, Simulation, Hypercube, IBM SP2, Transport Optimisation, Transputer Mesh, Performance Parameters.**

***Source of Publication*—IEE Proceedings: Digital Technology, Vol. 144, No. 1, January 1997.**

1 INTRODUCTION

Recently we have seen the introduction of a number of large message passing computers, with the new IBM SP2 being able to scale to hundreds of processors. Examples of other message passing parallel computers are Transputer Hypercube [8], Transputer mesh and cluster of workstations connected via an ethernet.

One problem facing the user is whether or not a particular parallel processing computer will suit the application, i.e., what will the performance be, and would it be profitable to migrate the application to the new system.

To answer this question of predicting the performance of message passing computers, we see the need for a simulation tool with the following features:

- separation of the algorithm and hardware simulation
- easy re-configuration of the hardware
- ease of modifying various system parameters
- allow a study of various routing methods and interconnection networks
- provide interaction between communication processes and computation processes
- capability to simulate thousands of processors
- allow different processor speeds.

A number of simulation tools for parallel systems are available, such as Parallax [6], Pyrros [15] and Hypertool [14]. Hypertool is a tool for scheduling a program on a hypercube. The program to be analysed must firstly be written (in C) before analysis is performed. This tool does not allow a flexible topology and the need for having a working program is seen as a disadvantage as we do not want to spend time writing a program which may not be executed.

Parallax improves on these two points by extending the topology from a hypercube to a general interconnection network and the program to be scheduled is input as a task graph. However, Parallax was designed primarily for comparing differing scheduling methods and heuristics. In a similar manner, Pyrros is focused on providing more complex scheduling methods.

Whilst these tools do provide some of the features we desire in a simulation tool, not all of the desiderata are provided such as the interaction between communication and computation processes, different processor speeds and the ability to simulate thousands of processors. Therefore, to satisfy the needs of a simulation tool, in this paper, we will address the design and implementation issues of our message PAssing computeR SIMulator—Parsim.

1. Head, Information Management Group, Information Technology Division, DSTO—
Email: `Lakshmi.Narasimhan@dsto.defence.gov.au`

The rest of the paper is organized as follows: Section 2 outlines the goals and implementation of Parsim, sections 2.3 and 3 provides details of the topologies and algorithms simulated respectively, with the results discussed in section 4.

Section 5 provides pointers for future work in this area.

2 PARSIM DESIGN

In the design of a simulator, we have the choice between an event driven simulator or a time driven simulator. Event driven simulators order the execution of the events in the simulation in an a priori fashion. This has the advantage in that only events are simulated, i.e., the simulator can jump to the next event in the queue. However, in the message passing computer, there are interactions between computation and communication events which can affect both the duration and ordering of the events. To avoid this problem, we use a time driven simulator. A global clock is used to step through each event, analogous to the processor clock.

As Parsim is a time driven simulator, the objects in the simulation each have a number of associated states. While in a processing simulation state, depending on whether or not the processor has a separate communications processor, not all of the processor's power may be available for computation. For example, each active link on the transputer generates a 5% load on the processor, whereas for many workstations with no specific communications processor, the computation and communication cannot be overlapped.

To reflect these possibilities, each simulated processor has a value indicating the percentage of processing power that is available for that particular clock step, i.e., *proc_avail*. Each link attached to the processor may reduce this value if the link is involved in communication.

In designing a simulation tool, it has been proposed by Tanir in [13] that using a standardised approach for simulation models and specifications reduces the difficulties faced by the user in generating the models for simulation. Therefore, the design of Parsim follows the standard model for simulation environments introduced by Tanir which defines five levels shown in Table 1.

Table 1: Standard Model

Level 0	Host Language.
Level 1	Model Specification. The model abstraction.
Level 2	Knowledge Management. How are the models inter-connected?
Level 3	System Design. Data gathering.
Level 4	Application Layer. The application interface.

In the following five sections Parsim is developed corresponding to the levels of the standard model.

2.1 LEVEL 0: HOST LANGUAGE

The host language is the programming language in which the simulator is developed and the models for the simulator specified. The requirements for the host language are support of high level data structures, dynamic memory management and a familiar syntax. The language C++ is used to develop Parsim as it supports these requirements as well as providing object oriented capabilities with low execution overheads.

2.2 LEVEL 1: MODEL SPECIFICATION

Level 1 of the reference model is the model specification, which defines how the algorithms and applications are simulated and specified. Tanir abstracts the model specification as an object such as:

```
    class MODEL{
        operations
        data structures
    }
```

We generalise the algorithms to be simulated as being composed of a group of phases executed in sequential order. Each phase is subdivided into a set of like tasks. For example, in the one dimensional Fast Fourier Transform (FFT) [1], this algorithm consists of two phases namely the row FFTs and column FFTs. The first phase can be broken down into a number of single row FFT tasks, and similarly, the second phase can be broken down into a number of single column FFT tasks. The execution of the phases is based on the following pseudo code.:

```
for each phase{
                start up computation on host
                synchronise phase start
                for each task {
                        send communication
                        local processing
                        receive communication
                }
                synchronise phase end
                phase end computation on host
        }
```

In Parsim, we provide a C++ class, *specific*, which extends the MODEL class to allow the algorithms to be simulated. This class consists of the following elements:

- Computation
  - start-up computation
  - local processing
  - execution time for each task
  - end computation
- Communication
  - message size sent to the slave processors
  - message size sent to the host processors
  - receive communication
  - send communication
- Synchronisation
  - synchronisation between phases
- Algorithm Data Structures
  - number of tasks per phase
  - number of phases
  - specific data structure.

2.3 LEVEL 2: HARDWARE AND TOPOLOGY SPECIFICATION

Level 2 of the standard model provides the model interconnections. In this section, we will consider the interconnection of processors, communication nodes, links and the message routing. To allow flexibility, parameters specifying the hardware configuration to be simulated are read in via input files.The parameters required to model the hardware include:

1. Link latency and start up times.
2. Speeds of each CPU relative to a base CPU.
3. Topology interconnection and routing information.

The first two parameters can simply be read in as a file of floating point numbers which can be easily changed to reflect faster hardware and processing speed. However, more emphasis is necessary on the representation of the interconnection network and message routing information.

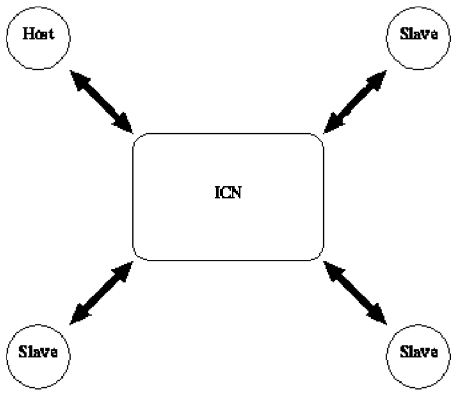


Figure 1: Message-passing Parallel Computer Configuration

The general processor interconnection is shown in Figure 1, where a number of processors are attached to an interconnection network. For multi-stage interconnection networks, such as the IBM SP2, processors are only connected to communication nodes (termed nodes). An example of these configurations is Figure 2a, where the ICN consists of nodes and links.

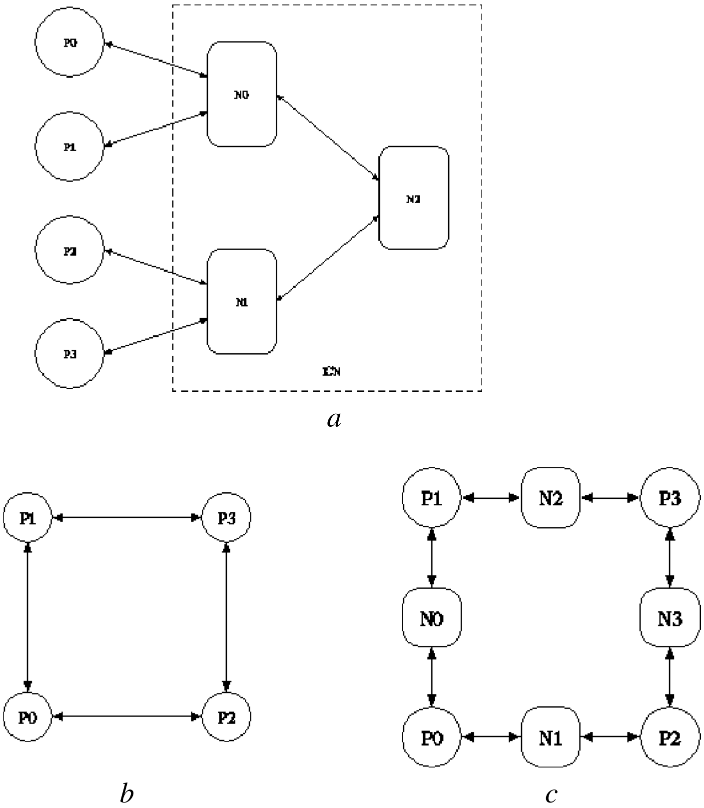


Figure 2: Processor–Processor Interconnection—*a* Multi-stage Interconnection; *b* Processor Only Interaction; *c* Added Notes

However, some topologies have processors only connected to other processors, such as the hypercube and mesh topologies. In this case, the ICN consists only of links to other processors, an example is shown in Figure 2b.

To provide homogeneity between these two interconnection methods, we transform the processor only topologies into processor node topologies by inserting a communication node between the processor-processor connection. This modification is shown in Figure 2c.

This model with inserted communication nodes has the following advantages over processor-processor only interconnection:

1. Allows a consistent model of processor - node connections.
2. Allows unidirectional and bi-directional communication to be regulated, e.g., using a node to provide the ethernet interconnection allows the constraint of only one processor sending a message to be applied.

The topology information is broken into two files—one for the processors and another for the communication nodes. If we have *P* processors and *N* nodes, then the processor information file will consist of *P* of the following entries:

```
<processor_id>
<number of send links connected>
<number of receive links connected>
<route information>
```

The node information file will consist of N entries of the following form:

```
<node_id>
<number of send links connected>
    <connected processor_id , processor link>
...

<number of receive links connected>
    <connected processor_id , processor link>
...

<route information>
```

As a node can be connected both to processors and other nodes, we represent the id for the processors as the numbers $0... P - 1$ and the node id as the numbers $- 1... - N$.

2.3.1 ROUTE INFORMATION

Each of the processor and node entries in the configuration input files contain route information necessary to route messages from one processor/node to the destination processor[1]. The routing for intermediate processors/nodes consists of selecting the correct send link, and then routing the message. The selection of the correct link to route the message can be achieved via a number of methods such as:

- dynamic route determination algorithm
- static route determination algorithm
- static route table.

Route determination algorithms required in the first two methods may be dependent on the topology simulated, and would need to be hand coded into the simulator, thus reducing Parsim's flexibility of simulating different routing strategies. To overcome this limitation, a static route table is used.

Each processor/node i has a route table which consists of a vector (P entries) of *link sets*. At position j in the vector, the link set is the set of links that i may use to communicate with processor j. The link sets are represented by an integer[2]. A link set is created by numbering the links *0...MAX LINKS* and setting bit $k = 1$ iff link k can be used for processor/node i to route messages to processor j.

For topologies such as the hypercube and mesh, each link set will have at most one non-zero bit. However, this is not the case in general. This implies that there can be a choice of links for routing purposes.

For large numbers of processors, it can be seen that it would be impractical to generate the processor and node information files manually. To aid in the development of new topologies, a number of tools are provided, namely, Linkup, Cube, Meshx, Meshy, Meshstep, Ethernet and Frame which help develop the node and processor configuration files.

- Linkup—Allows the user to specify directly the processor to processor connections and routings, thereby producing the processor and node information files.
- Cube—Hypercube topology of user specified size.
- Mesh[x,y,step]—Rectangular mesh topology with user specified number of rows and columns. The extension indicates the routing method used.
- Ethernet—Ethernet based cluster.
- Frame—BM SP2 MIN topology.

Note that a cluster of workstations connected via an ethernet can communicate directly with each other. However, a hypercube requires that messages be routed through intermediate processors. To achieve such a routing, *wormhole* communication [9] has been used.

---

1. As nodes do no computation, no messages can have a node as a destination.
2. The use of unsigned 32 bit integers allows 32 communication links per processor/node. If this is a limitation, then long integers or a specific data structure can be used. Note that a 32 dimensional binary hypercube would have about 10^9 processors.

2.4 DATA GATHERING

Level 3 in Tanir's model is important as it is in this level that the output data of the simulator is gathered. The data we desire to gather from the simulation are:

- The execution time of the algorithm—using this, speedups can be calculated and comparisons made between other algorithms and topologies.
- Processor utilisation—this indicates the efficiency of the parallel algorithm.
- Communication bottlenecks—the simulator can be used to pinpoint the presence of bottlenecks, and compare the effects of new routing methods or algorithms on the communication.

2.5 LEVEL 4: APPLICATION GUI

To aid the developer, a Graphical User Interface (GUI) based on Tcl/tk [10] has been added to the front end of Parsim. Screen dumps of Parsim are provided which outline the following functions.

- The algorithm to be simulated and the directories for the configuration files are selected in Figure 3.

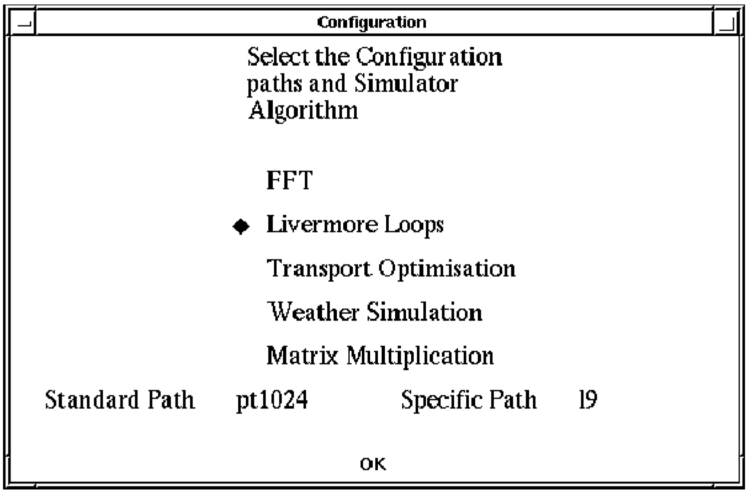


Figure 3: Configuration Menu for Parsim

- An example graphical output of Parsim is shown in Figure 4.

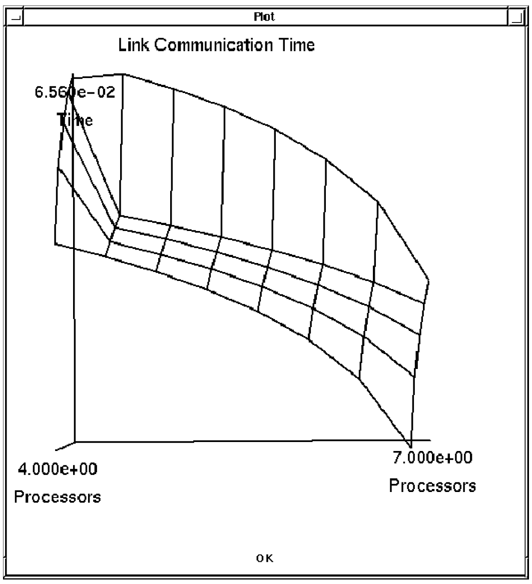


Figure 4: Graphical Output of Parsim

3 TEST ALGORITHMS ANALYSED

Three test algorithms are analysed using the simulator, a parallel FFT, a set of Livermore Loop Kernels and the Transport Optimisation Problem.

These algorithms can be parallelised in a variety of ways and whose suitability for message passing computers will also vary. Let us consider a typical algorithm on a message passing computer in order to determine features of the algorithm that are favourable for parallel execution.

Assumptions:

- The data size is represented by x and the number of processors[1] by P.
- The data originally resides on one processor, termed the host, and the results must also reside on this processor.
- We use an equal distribution of data over the processors, i.e., each processor receives data of size $\frac{x}{P}$.

Each processor will communicate a set of messages with the host and the total communication time of these sets of messages not overlapped by computation be as in equation 1, where C_0 is the fixed start up time and C is the communication time proportional to the data size.

$$\textit{Message Set Communication Time} = C_0 + C\frac{x}{P} \tag{1}$$

As there are $P - 1$ processors communicating with the host, the total communication time from 1 is therefore equation 2.

$$\textit{Total Communication} = (P - 1)\,(C_0 + C\,\frac{x}{P}) \tag{2}$$

Define the execution time as in equation 3.

$$\textit{Computation Time} = X(x) \tag{3}$$

As the parallel algorithm must do the work of the serial algorithm, the time to compute just the algorithm on P processors is therefore $\frac{X(x)}{P}$. Thus combining equations 2 and 3 gives the total execution time of the parallel algorithm $T(P)$ as in equation 4.

$$T(P) = (P\text{-}1)\,(C_0 + C\frac{x}{P}) + \frac{X(x)}{P} \tag{4}$$

With $X(x)\,(\frac{P - k}{kP})$ termed the scaled reduction in work, the following theorems have been proved in [12].

THEOREM 1

The scaled reduction in work using the parallel algorithm must be greater than the total communication time in order to obtain a speedup of k.

THEOREM 2

If the execution time of any parallel algorithm is denoted as the function $X(x)$, and the inverse of this function is denoted $X(x)$, then for large P, the minimum size of data x to achieve a speedup of k is given by:

$$x > X^{-1}\left(kPC_0 \times \frac{P - 1}{P - k}\right) \tag{5}$$

THEOREM 3

For small values of C_0, the minimum size of data x to achieve a speedup of k is given by $x > \sqrt[N-1]{\frac{kC}{X_N} \times \frac{P - 1}{P - k}}$ where the complexity of the algorithm can be given as $X(x) = X_N \times x^N$ and $N > 1$.

Let us also consider the special case of the algorithm complexity being linear with x, i.e., $X(x) = X'x$. Substitution into equation 4 gives equation 6.

$$x > \frac{kPC_0}{X' - kC\,\frac{P - 1}{P - k}} \times \frac{P - 1}{P - k},\quad X' - kC\,\frac{P - 1}{P - k} > 0 \tag{6}$$

1. Although x and P must be integers by definition, let us suppose that x and P are large enough to be considered a real number for this analysis.

3.1 FFT

The idea behind the Fast Fourier Transform (FFT) calculation may be attributed to [4], but the first widely known FFT algorithm is that of Cooley and Tukey [2]. It features in many applications ranging from image processing to speech recognition.

Real time processing, however, is severely constrained by the speed of general purpose uniprocessor systems and, as a consequence, proliferation of dedicated DSP hardware has now been observed. These dedicated systems still perform sequential calculations and this has caused the investigation of a number of parallel versions of the FFT.

In order to determine the performance of the FFT executing on various transputer topologies, a parallel FFT algorithm based on a two stage one dimensional transform ([11]) is simulated by Parsim. The input sequence is mapped onto a two dimensional matrix of R rows and C columns. Essentially the algorithm consists of two stages:

- Stage 1—Apply the FFT over the rows of the data.
- Stage 2—Apply the FFT over the columns of the data.

It is important to note that the time complexities of the row and column FFTs for equal distribution of data are $R(\frac{RC}{P} \log C)$ and $R(\frac{RC}{P} \log R)$ respectively.

3.2 LIVERMORE LOOP KERNELS

The Livermore loop kernels are a common way to measure the performance of parallel systems ([7]). Only a subset of the kernels which are applicable to the message passing networks have been simulated using Parsim. Each of the loops is blocked, i.e., the data is divided into contiguous blocks, which are distributed over the network so that each processor receives an equal amount. The block size is given by the number of iterations divided by the number of processors. These kernels are characterized for a 30Mhz T805 Transputer and a Sun Sparcstation-2 by the parameters shown in Table 2.

- Kernel number 1 is a fragment of hydrodynamics code.
- Kernel number 2 is a fragment of incomplete Cholesky-conjugate gradient.
- Kernel number 3 is the Inner Product.
- Kernel number 7 is an Equation of state fragment
- Kernel number 9 is a code fragment calculating the integrate predictors.
- Kernel number 12 calculates the first difference.

Table 2: System Parameters

Kernel Number	Flops Iteration	Mflop Single Transputer	Mflop Single Workstation	Iterations	Blocks Received	Blocks Sent
1	5	0.943×10^6	1.855×10^6	10,000	1	1
2	4	1.315×10^6	1.315×10^6	10,000	1	2
3	2	0.724×10^6	0.355×10^6	10,000	0	2
7	16	1.548×10^6	2.922×10^6	10,000	1	3
9	17	1.416×10^6	2.148×10^6	10,000	1	13
12	1	0.357×10^6	0.574×10^6	10,000	1	1

3.3 TRANSPORT OPTIMISATION

The problem of scheduling transportation routes and vehicles optimally is gaining more and more interest due to the large amounts of capital invested in the vehicles and the payroll of the drivers. To solve this transport optimisation problem, integer programming and dual simplex methods have been proposed. However, solutions to practical problems involving 1400 variables can take a few hours on a workstation.

Therefore much interest has been displayed in whether the optimisation problem can be parallelised effectively.

To answer this question, it is proposed that the optimisation algorithm be simulated using Parsim. The algorithm can be outlined as follows:

1. Chose a constraint branch from the initial basis.
2. Construct an infeasible basis for each of the 0 and 1 branches[1], termed decision nodes.
3. Apply Dual simplex to each node to generate two feasible bases.
4. Repeat until an integer solution is found.

Initially, there is one basis, but with each application of the Dual Simplex algorithm, two more nodes are produced in the decision tree.

The statistics collected for the execution of a Transportation Optimisation problem on a Sun Sparcstation are the following:

- The number of variables in the basis is 1421.
- The time to generate a successful sub-goal solution by the dual simplex iteration has been found to be a random variable with a range of 0 to 4 minutes.
- The number of decision nodes to be evaluated before reaching the optimal integer solution is 40.

4 SIMULATION RESULTS

The actual and simulated performance of the parallel FFT are shown in Figure 5. It is seen that the simulated performance is in close agreement with the actual performance. The variation is due to the simulator clock starting at zero, where as on the transputer, there is some bookkeeping overheads which meant that the clock does not start at time zero.

The simulated performance of the livermore loops on the transputer hypercube is shown in Figure 6. This graph shows that the transputer hypercube only provides an increase in performance for the Hydro Fragment (loop 1) and the Equation of state (loop 7). This can be explained for each individual Livermore Loops by corollary 1.

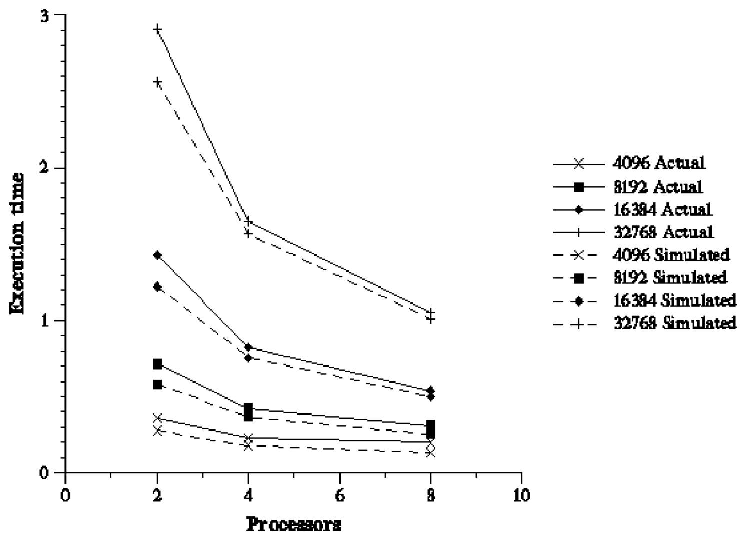


Figure 5: Comparison Between Actual FFT Performance and Simulated FFT Performance on a Transputer Hypercube

1. Any variable in the basis which is neither 0 nor 1 can be forced to a 0 or 1, thus moving the basis closer to an integer solution.

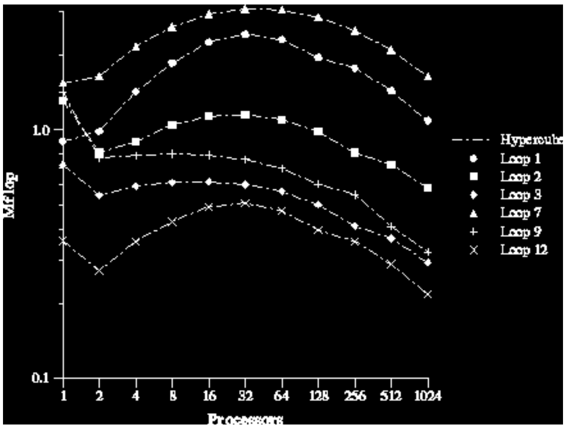


Figure 6: Simulated Livermore Loop Performance on a Transputer Hypercube—100,000 Iterations

COROLLARY 1

To achieve a speedup for any particular Livermore Loop, equation 7 must be satisfied where *Lk* is the transfer rate of communication links in floating point numbers per second, *Nl* is the number of communication links active per processor, *FI* is the floating point operations per iteration, *B* is the total number of blocks sent/received per processor and *MR* is the MFlop rating of a single processor.

$$\frac{Lk \times Nl}{MR} > \frac{B}{FI} \tag{7}$$

PROOF:

Using equation 6 and noting that the speedup factor k is 1 gives equation 8.

$$x > \frac{PC_0}{X' - C}, \quad X' - C > 0 \tag{8}$$

As we let $C_0 \rightarrow 0$, the necessary condition is $X' - C > 0$. X' is the rate of work of the Livermore Loop which can be calculated as $\frac{FI}{MR}$. Similarly, C is the communication rate, which can be calculated as the ratio of floating point numbers communicated to the speed of the communication links, i.e., $\frac{B}{Lk \times Nl}$. Substituting into equation 8 gives:

$$\frac{FI}{MR} - \frac{B}{Lk \times Nl} > 0$$
$$\frac{Lk \times Nl}{MR} > \frac{B}{FI}$$

□

This condition can be interpreted as to increase the overall speedup, the physical machine's ratio of transfer rate to processing rate must be greater than the algorithm's per iteration ratio of communication to computation.

The effect of this condition can be examined as follows. Let the number of active links be 1. The transputer's link transfer rate is 0.446×10^6 float/s, and the computation rate is approximately 1 Mflop. Therefore the ratio of Blocks sent/received to Floating point operations per iteration must be less than 0.446. From Table 2 it is seen that this is only true for loop numbers 1 and 7.

By increasing the transfer rate of the links ten-fold (which may reflect the use of faster communication processors), the ratio of Blocks sent/received to Floating point operations per iteration for a speedup with two processors is increased to 4.46. The simulation of the high performance links is shown in Figure 7. As can be seen, all loops now provide a speedup with two processors, as predicted by the higher communication computation ratio.

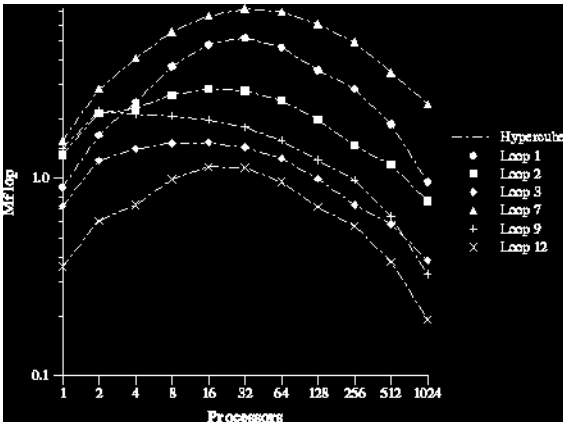
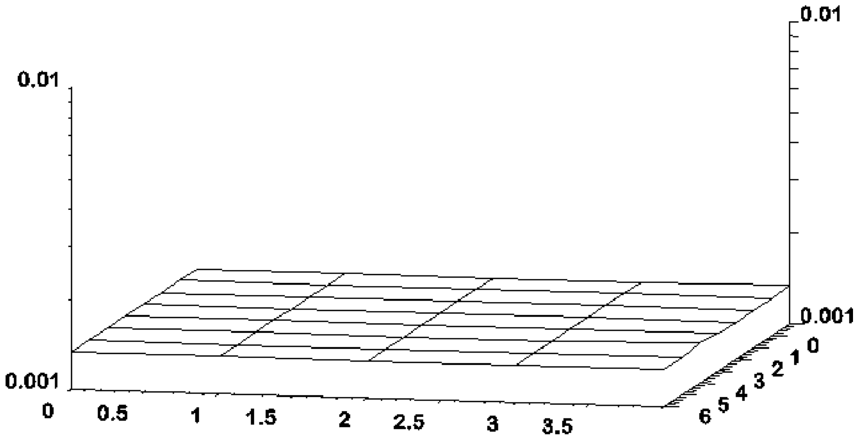
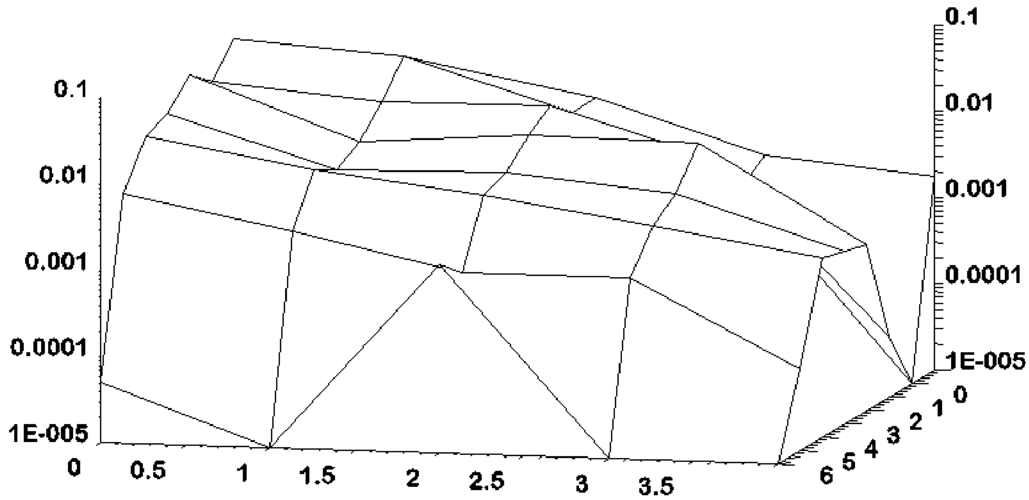


Figure 7: Simulated Livermore Loop Performance on a Transputer Hypercube—High Performance Links

The following five figures show the system performance of a 8 by 5 transputer mesh executing the first livermore loop. Figure 8a shows the total time spent in calculations by the processors. This shows that each processor performs the same amount of computation due to the equal distribution of data over the network. Figure 8b shows the amount of time spent by the links on each processor while being blocked. As there are no peaks and the middle is flat, it is concluded that no particular processor is blocked from communicating an excessive amount.



a



b

Figure 8: 8 × 5 Transputer Mesh Using Step Routing for the First Livermore Loop

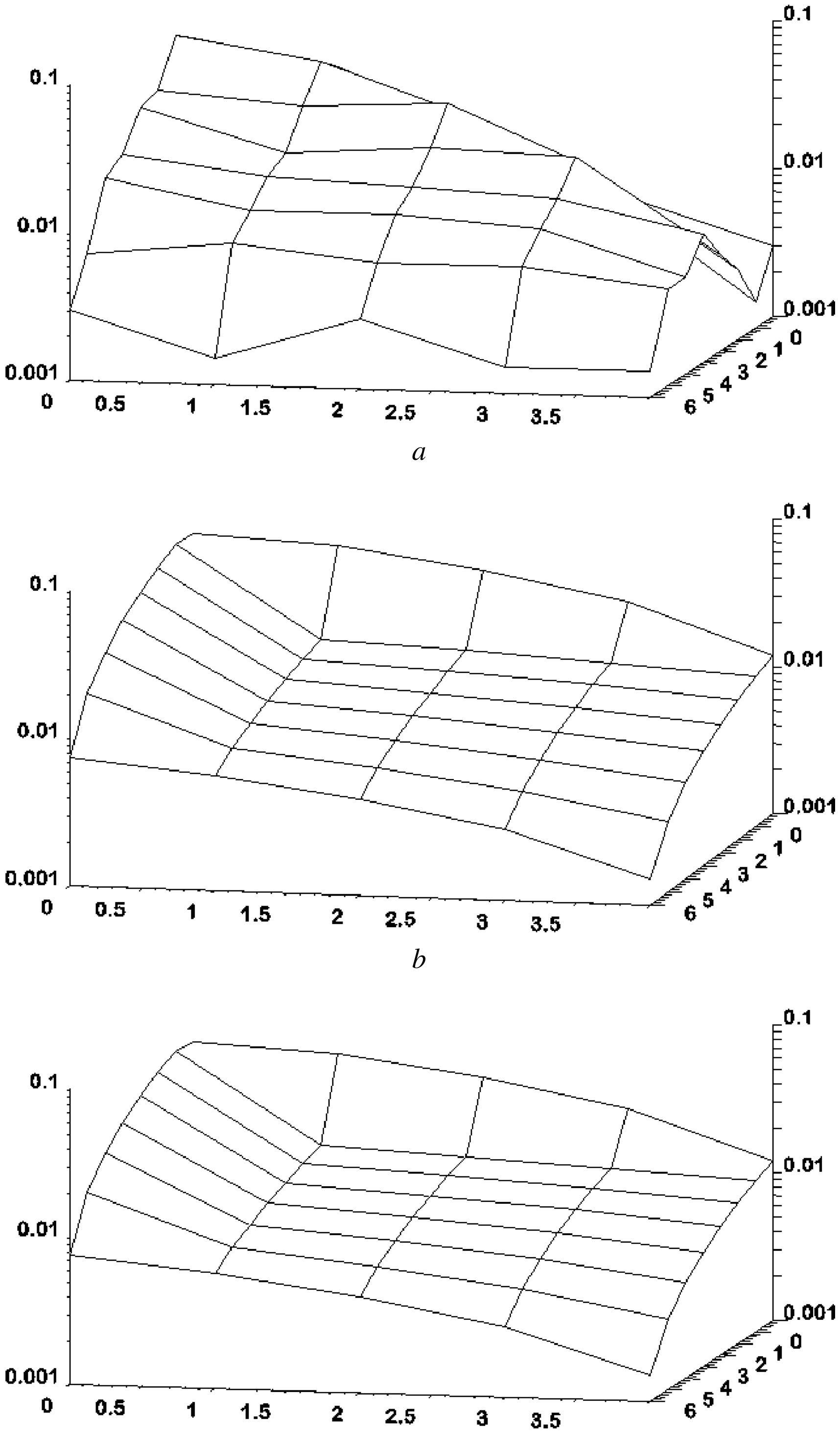
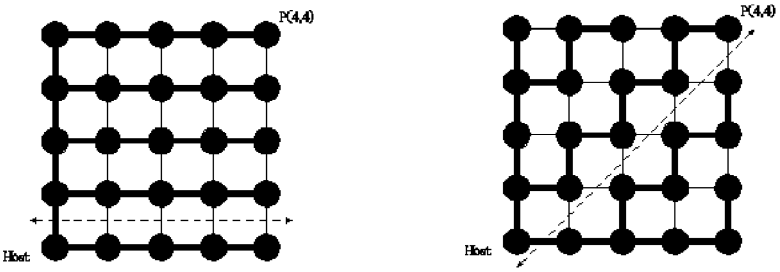


Figure 9: Link Communication Time of 8 × 5 Transputer Mesh for the First Livermore Loop—*a* Step Routing; *b* Row First (X) Routing; *c* Column First (Y) Routing.

Figure 9 (parts a, b and c) compares the link communication time for the three mesh routings. The step routing (Figure 9a) spreads the communication more evenly than the x routing (Figure 9b) or y routing (Figure 9c). In both the x and y routings, most communication is along the first row or column, thus creating a bottleneck.

To see why this is the case, consider the row routing shown in Figure 10a and the step routing shown in Figure 10b. It is seen that for the row route case, most of the processors communicate via the one link on the host processor. For the step routing the processors are divided approximately into half. In general, if the mesh is a square with $N \times N$ processors, then for the row routing at most $N \times (N - 1)$ processors are routed through one host link, whereas for the step routing, at most $\frac{N \times (N + 1)}{2}$ are routed through one host link.



a *b*

Figure 10: Routing on a 5 × 5 Transputer Mesh

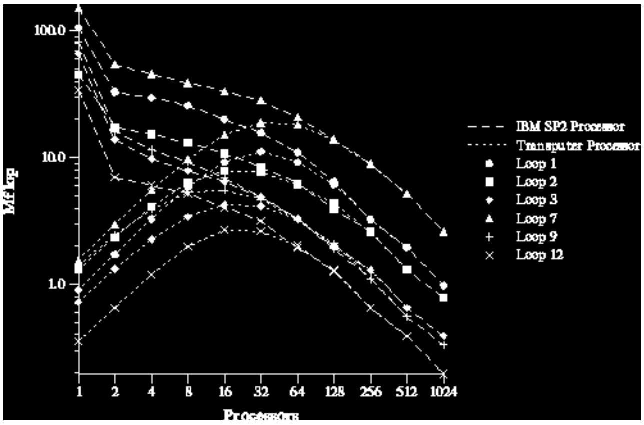


Figure 11: Simulated Livermore Loop Performance on an IBM HPS Network

4.1 MIN

Figure 11 shows the performance of the Livermore Loops executing on the MIN topology using the high performance switch (HPS) communication link values for the two cases: 1) IBM processing nodes, and 2) transputer nodes. This figure shows that for the case of the SP2 processing nodes, no performance improvement is gained by parallelising the Livermore Loops. This can be explained again by using corollary 1. The communication link transfer rate of the SP2 HPS is approximately 9×10^6 floating point numbers per second, and the computation rate of the SP2 node is about 100 Mflops. Therefore the ratio of link transfer rate to processing speed is approximately 0.1. Table 2 shows that the ratio of iteration blocks communicated to floating point operations per iteration for each loop is greater than 0.1, therefore from corollary 1 there will be no performance improvement through parallelisation of the Livermore Loops.

However, by replacing the SP2 nodes with the slower transputer nodes, a large performance improvement is gained for all loops. This is due to the ratio of link transfer rate to computation is now approximately 9 which can be satisfied by all of the loops in Table 2.

In addition, we see that as the number of processors increases (greater than 128 processors), the SP2 node performance and the transputer node performance converges. This indicates that for the total iteration size used (100,000 iterations), as the number of processors is increased, and therefore the block size per processor is reduced, the dominating term in the execution time is the communication overheads.

To allow analysis of a larger number of processors, the number of decision nodes to be evaluated before reaching the optimal solution is increased to 4000. The results are shown in Figure 12. The topologies simulated are the mesh, hypercube and SP2 MIN using transputer communication link values. This figure shows that there is little variation in the performances of each topology, with a maximum variation of 3%.

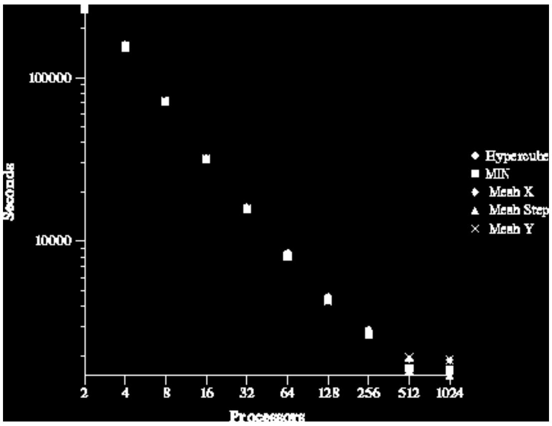


Figure 12: Topology Comparison for 4,00 Decision Nodes.

5 CONCLUSIONS

In this paper, we have introduced a tool to help predict the performance of message passing parallel systems. This simulator facilitates re-configuration in order to allow a variety of tasks and topologies to be simulated. A high degree of simulator accuracy is shown in the comparison between actual and simulated performance of a parallel FFT running on Transputer hypercube. The robustness of the simulator is shown in its ease of simulating 6 Livermore loop kernels on 4 topologies, as well as simulating the performance of an integer programming problem with random execution time on a cluster of workstations. The ability of Parsim to simulate new interconnection methods and variation in system parameters has been shown through the simulation of the unconventional step routing for meshes, large transputer hypercubes and transputer hypercubes using faster communication links. In the future, Parsim will be used to simulate various applications, such as weather modeling and shallow water temperature profiling, with large processor meshes of the order of 2000 processors.

REFERENCES

[1] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan, ‘A parallel FFT on a MIMD machine’, *Parallel Computing*, 15:61–74, 1990.

[2] J. W. Cooley and J. W. Tukey, ‘An algorithm for the machine calculation of complex Fourier series’, *Math. Comput.*, 19:297–301, April 1965.

[3] J. R. Gurd and C. Kirkham, ‘Dataflow: Achievements and prospects’, In *Inf Pross.*, pages 61–68. IFIP Conf, 1986.

[4] M. Heidemann, D. Johnson, and C. S. Burrus, ‘Gauss and the history of the FFT’, *IEEE Mag.*, 1, Oct 1984.

[5] H. Horikoshi and Y. Inagami, ‘Dataflow: From its practical viewpoints’, In *Inf Pross.*, pages 69–72. IFIP Conf, 1986.

[6] T. Lewis and H. El-Rewini, ‘Parallax: A tool for parallel program scheduling’, *IEEE Parallel and distributed technology*, 1(2):62–72, May 1993.

[7] F. H. McMahon, ‘The Livermore fortran kernels: A computer test of the numerical performance range’, *Lawrence Livermore National Laboratory*, UCRL-53745, December 1986.

[8] D. Mitchell et al, *Inside The Transputer*, Blackwell Scientific Publications, Melbourne, 1990.

[9] L. Ni et al, ‘A survey of wormhole routing techniques in direct networks’, *Computer*, pages 62–76, February 1993.

[10] J. K. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley Professional Computing Series, Sydney, 1994.

[11] A. Symons, V. L. Narasimhan, and K. Sterzl, 'Performance analysis of a parallel FFT algorithm on a transputer network', *Parallel Algorithms and Applications*, 4, 1994.

[12] A. Symons and V. Lakshmi Narasimhan, 'Parsim—message passing computer simulator', In *Proceedings of the First ICA^{3}PP-95 Conference*, pages 621–630, April 1995.

[13] O. Tanir and S. Sevinc, 'Defining requirements for a standard simulation environment', *IEEE Computer*, 27(2):28–34, February 1994.

[14] M. Y. Wu and D. D. Gajski, 'Hypertool: A programming aid for message-passing systems', *IEEE transactions on parallel and distributed systems*, 1(3):101–119, July 1990.

[15] T. Yang and A. Gersoulis, 'Pyrros: Static task scheduling and code generation for message-passing multiprocessors', In *Proceedings 6th ACM international conference on supercomputing*, pages 428–443, New York, 1992. ACM Press.