PERFORMANCE MEASUREMENT AND TRACE DRIVEN SIMULATION OF PARALLEL CAD AND NUMERIC APPLICATIONS ON A HYPERCUBE MULTICOMPUTER *

Jiun-Ming Hsu and Prithviraj Banerjee

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

* This research was supported in part by the National Science Foundation Presidential Young Investigator Award under Grant NSF MIP 86-57563 PYI, and in part by an equipment grant NSF CCR 87-05240.

ABSTRACT

This paper presents the performance evaluation, workload characterization and trace driven simulation of a hypercube multicomputer running realistic workloads. Eleven representative parallel applications were selected as benchmarks. Software monitoring techniques were then used to collect execution traces. Based on the measurement results, we investigated both the computation and communication behavior of these parallel programs, including CPU utilization, computation task granularity, message interarrival distribution, the distribution of waiting times in receiving messages, and message length and destination distributions.

We modeled the various time interval distributions by statistical functions which were verified by nonlinear regression technique using the empirical data. The temporal and spatial localities of message destinations were also studied. A model for the temporal locality of message length has been introduced and used to analyze the communication traces.

A trace driven simulation environment, which uses the communication patterns of the parallel programs as inputs, was developed to study the behavior of the communication hardware under real workload. Simulation results on DMA and link utilizations are reported.

1. INTRODUCTION

Hypercube multicomputers have recently offered a cost-effective and feasible approach to supercomputing by connecting a large number of low-cost processors with direct links [1]. Each processor has its own local memory. Processes running on these processors communicate via message passing. This type of architecture is more readily scaled up to large numbers of processors than multiprocessor designs based on globally shared memory. Implementations of the hypercube architecture range from experimental prototype systems [2], to commercially available systems from Intel [3], Ametek, and NCUBE.

The evaluation of performance of parallel machines such as hypercubes is extremely important for exploring parallel program characteristics and parallel architecture behavior. One of the goals of our research was to implement a general performance evaluation environment for hypercubes based on software monitoring. In order to do an accurate evaluation, we decided to do some measurements of several realistic parallel applications (five in VLSI CAD and six in numerical domain) written for hypercubes. We used an Intel iPSC/2 hypercube as a testbed for our study. Some preliminary results of our study have been reported in [4].

On the basis of the measurements, we studied issues in both computational and communication behavior of realistic parallel programs on hypercubes. The computational characteristics that we measured were the statistics on CPU utilization, relative system time, user application time, idle time of various CPUs in the parallel system, statistics on task granularity between message generation, etc. The communication characteristics that we measured included the distribution of interarrival times of messages, the distribution of waiting times in receiving messages, and message length and destination distributions.

The motivation behind these measurements was to validate the assumptions typically made by researchers in the parallel computer systems modeling area. The application of queueing network models to computer network performance analysis typically assume some kind of message arrival interval, length, and destination distributions. The most common assumptions are Poisson message arrival, exponentially distributed message lengths, and evenly distributed message destinations [5], which make the problems more analytically tractable. Researchers in the load balancing area also make various assumptions about task granularity and intertask dependencies in computational task graphs that are mapped on to processors [6, 7]. We wanted to verify if indeed the above assumptions are valid, and if not, what distributions more accurately model the real world applications on hypercubes.

In experimental studies of hypercube performance [8, 9], simulations have been performed using various synthetic communication benchmarks, which assume different kinds of message interval, length and destination distributions. Again, based on the results of the measurement, we can verify whether these synthetic benchmarks are close to real applications by modeling the empirical data using statistical distributions [10, 11]. The resulting model will also help us in designing more realistic synthetic communication benchmarks.

The communication patterns of the parallel programs can critically affect the performance of the message passing machines, hence we were also interested in the localities of message destination and length. Models for both the spatial and temporal localities of message destinations have been proposed before by [12]. In this study we propose a model for the locality of message size based on the LRU stack model [13]. The accuracy of this model was verified by measurement results.

In distributed-memory multicomputers like hypercubes, synchronization and data sharing are achieved by explicit message passing. Hence, the speed and efficiency of communication are very important in the overall performance of such machines. In this study, we also investigate the performance of the communication channels in detail. The behavior of the communication hardware components (DMA channels and communication links) is not easily observable via software monitoring. Therefore a trace driven simulation environment was developed to model the hardware and software components of typical hypercube nodes. Experiments were then run using communication traces (message transmission patterns) collected from the earlier part of this work. We were therefore able to obtain reasonably accurate performance information of the communication channels.

The benchmarks used in the measurement are described in Section 2. Section 3 gives an overview of the measurement environment and monitoring facilities. Section 4 reports the results of the measurement and the characteristics of the communication workload. Section 5 describes the trace driven simulation environment. In Section 6, we present the simulation results.

2. BENCHMARKS

Eleven hypercube applications are used as benchmarks, which solve real-life VLSI CAD and numerical problems. These two types of programs are currently the most popular parallel processing applications. Different data partitioning schemes are employed in these benchmarks, and the communication patterns vary from ring, nearest neighbors, distance-two, to broadcast communication. The communication workload ranges from very light to very

heavy. Together, these applications represent a typical workload of hypercube multicomputers. The input data used are also realistic and of reasonable size, hence they are not toy benchmarks. Five parallel VLSI CAD programs are used, which are: standard cell placement, channel routing, VLSI circuit extractor, test pattern generation, and fault simulation. In addition, six numeric applications are also used as benchmarks, they are: QR factorization, Fast Fourier Transformation, Gaussian elimination, simplex algorithm, Navier-Stokes algorithm, and one program from the EISPACK library. In this section, we will briefly describe each of these hypercube applications.

2.1. Standard Cell Placement

The parallel cell placement program [14] is based on simulated annealing technique whose objective is to optimally place cells in a VLSI layout to minimize the total estimated wirelength. The basic idea used in this algorithm is to assign cells in a sub-area of the VLSI chip image to each processor in the hypercube, and to have parallel movements of cells between regions. Processors pair up among various dimensions of the hypercube to perform moves (displacement or exchange of cells), and the moves are evaluated and accepted/rejected based on the probabilistic annealing function. After a set of parallel moves, the processors synchronize with each other by using a ring-based broadcast mechanism to correct the errors in evaluating the cost functions.

2.2. Channel Routing

The parallel channel routing program [15] is also based on simulated annealing technique whose objective is to interconnect all the nets in a channel region so as to minimize the number of horizontal tracks. Initially nets are assigned to tracks randomly while allowing for both horizontal and vertical overlaps. Sets of adjacent tracks are assigned to processors arranged in a binary reflected gray code for linear arrays. The processors then pair up to move the nets between tracks in parallel, evaluate and accept the movements on the basis of a probabilistic annealing function, and broadcast information to other nodes to update their data structures using a ring-based communication strategy.

2.3. VLSI Circuit Extraction

The parallel circuit extraction program takes as input a set of rectangles at various mask layers and produces a circuit netlist in terms of transistors, resistances and capacitances [16]. It consists of two phases. The first phase, which is dominated by communication, is the distribution of data from the host to each of the node processors. The

second phase is the circuit extraction phase which is computationally intensive. In the first phase, the host reads the circuit in batches of rectangles, and distributes the rectangles to the nodes in a tree fashion. The nodes in the tree further divide and transmit the rectangles to their leaves while doing load balancing at the same time. Once all the rectangles have been read by the host, the nodes go through some post processing steps so that each of them gets all the rectangles that overlap or abut its assigned region. The objective is to give each processor the same number of rectangles. The next stage is the actual process of circuit extraction which identifies the devices and nets, and subsequently estimates the electrical parameters of devices and nets. In this step, a merge algorithm is used to perform a connected component labeling procedure which involves a number of steps of communication among processors in all the dimensions of the hypercube.

2.4. Test Pattern Generation

Given a logic circuit, the objective of a test generation algorithm is to derive a set input patterns that distinguish the fault-free circuit from a faulty one. The parallel test pattern generator [17] is based on a parallel branch and bound algorithm which proceeds in a depth first manner to implicitly enumerate the search space for a test vector. The parallel algorithm divides the search space so that intermediate choice points are skipped by one processor while at the same time the skipped choice points are evaluated by other processors. The scheduling and control of the parallel search is done by node zero, while other nodes do the searches in parallel.

2.5. Fault Simulation

In fault simulation the objective is to find out how many of a given set of faults are detected by a given set of input patterns. The parallel fault simulation program [18] partitions the faults among the processors initially and uses a dynamic load balancing scheme to distribute the faults to be processed at run time. The idle processors broadcast messages to request for more works from other busy processors.

2.6. Matrix QR Factorization

The QR factorization [19] of a matrix A involves the determination of an orthonormal matrix Q and an upper triangular matrix R such that $QA=R$, also written as $A=Q^TR$. The standard sequence for computing Q and R involves successive annihilation of the elements of A below the main diagonal in such a way that previously introduced zeros are not destroyed. Thus, Q is a product of simple orthonormal matrices (plane rotations) each chosen

to eliminate one element of A below the diagonal. The parallel QR factorization algorithm [20] applies more than one rotation at a time so as to eliminate matrix elements in parallel. The matrix rows $n-1$ to 0 are mapped, in that order, onto a ring of p processors by wrapping around. During the elimination of an element in row i of the matrix, the processor having row i obtains row $i-1$ from its successor, computes the rotation and updates row i. Concurrently, its successor receives row i, computes the rotation and updates row $i-1$.

2.7. Fast Fourier Transform

The Discrete Fourier Transform is one of the most important computation in digital signal processing. It is given by:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, w_N^{kn} \ , k = 0,1,..., N-1,$$

where $w_N = e^{-j(2\pi/N)}$ is the Nth root of unity. It is possible to rewrite this equation in an iterative form, which then forms the basis for the Fast Fourier Transform (FFT). Parallel implementation of the FFT on a hypercube [21] involves an even distribution of the input points among the processor nodes initially. The nodes subsequently perform the iterative sequential FFT algorithm on the points they have received, independently and in parallel. At the end of each such step, messages are exchanged across nodes to set up the points for the next iteration in the parallel scheme. The final results are sent back to the host.

2.8. Gaussian Elimination

A set of linear equations $Ax = b$ can be solved by performing Gaussian elimination to obtain an upper triangular matrix. On a hypercube, this algorithm is implemented by distributing complete rows of $Ax = b$ to each processor [22]. In each iteration of the computation, a pivoting row is selected by finding the global maximum. This row is then broadcasted to every node to eliminate one element in each of the remaining active rows, using the tree broadcasting algorithm. The node processors only communicate with nearest neighbors.

2.9. Simplex Algorithm

The simplex algorithm is a common method to solve linear optimization problems, in which some or all of the equations in $Ax = b$ are inequalities. The algorithm first transforms these inequalities into equalities while maintaining the $x \geq 0$ constraints. It then iterates systematically to minimize the objective (cost) function. In the parallel simplex algorithm [23], complete rows of $Ax = b$ are divided equally among the processors. It also involves finding

the global minimum and tree broadcasting.

2.10. Navier-Stokes Algorithm

The Navier-Stokes equations for fluid dynamics modeling can be solved numerically by an iterative relaxation scheme [24]. The parallel version of this algorithm partitions the grid points among the processors. After each iteration, the processors communicate to obtain the values of the points abut its region.

2.11. EISPACK

The last benchmark is the *tred2* routine in the EISPACK [25] matrix computation library. *Tred2* reduces a symmetric matrix to a symmetric tridiagonal matrix and is used in computing the eigenvalues of a matrix. The parallel version of this routine is generated by a semi-automatic parallelizing technique based on a column distribution scheme which exploits the maximum parallelism and ignores the communication cost. Therefore, this benchmark is very communication intensive.

2.12. Inputs to the Benchmarks

The input to the standard cell placement program is a 286 cells circuit. In the channel routing program, a circuit with 229 subnets is used. The input to the circuit extractor is a layout with about 66,000 rectangles. The test generator finds test patterns for a circuit with 3540 nets. The same circuit is used in the fault simulation program. The input matrices of QR factorization, Gaussian elimination, simplex algorithm and EISPACK are 128×128, 160×160, 469×220 and 64×64, respectively. In the FFT program, the number of points is 2^{12}. The total number of grid points in the Navier-Stokes program is 2^{10}.

Table 1 shows the speedup results of the application programs using the above input data. Note that the channel routing program, which uses a stochastic algorithm (simulated annealing), has super-linear speedups. The circuit extraction program also has super-linear speedups which is mainly due to the fact that the dynamic memory allocation takes more time on a single processor where there is not much memory left after all the data are loaded in [16]. The parallel EISPACK routine exploits all the available parallelism in the sequential program, hence it has slowdowns instead of speedups. We use it as an example to show the effect of running fine-grain parallel program on a medium-grain machine.

In each of the above applications, with the exception of the circuit extractor, only the main execution part is measured. The downloading of the data and the uploading of the results are not included because nothing interesting happens in these cases. For the extractor, both the downloading of the data and the execution part are traced, since there is an interesting dynamic load balancing that is performed during the reading of the input data.

3. MEASUREMENT FACILITIES

3.1. Hypercube System

The system on which we performed the measurement is a 4 dimensional Intel iPSC/2 hypercube [3], which contains 16 computing nodes (the node processors) and a front end processor called the *host*. The Intel 80386-based computing nodes are connected via bi-directional communication links to form a binary n-cube topology. The host is also an Intel 80386-based computer that serves as an interface between the user and the computing nodes. Only the host has secondary storage, hence the computing nodes access the file system through the host.

The host runs the UNIX System V operating system, and the computing node runs a proprietary operating system called the NX/2 [26] which supports, among other things, file access to the host and message passing between the nodes. The hypercube program usually consists of a host program running on the host and a node program running on every computing node. In most cases, the host program downloads the data to the node and uploads the results from the node, whereas the node program performs the real computation. Application programs access the message passing capabilities through two sets of system calls: 1) blocking communication calls, *csend()* and *crecv()*, and 2) non-blocking communication calls, *isend()* and *irecv()*.

Messages of 100 bytes or less (short messages) are unsolicited, i.e. there will always be buffers reserved for them in the destination nodes, so they can be send directly. However, messages longer than 100 bytes must request the receiving buffer from the destination node first before the transmission, so they follow a three-trip protocol. First, a control message (short message) is sent to the receiving node to make the buffer request. The receiving node will allocate the receiving buffer and send back an acknowledgment which is also a control message. Finally, the source node will send out the entire long message upon the receipt of the acknowledgment. Note that the request and acknowledgment are operating system control messages, hence they are not counted in the message statistics.

3.2. Measurement Facilities

We have implemented a software monitoring tool in the iPSC/2 processor nodes. The monitoring facility can be used to tune the software for better performance, to investigate resource utilization, and to determine the characteristics of computation and communication workload. Monitoring is done at the operating system level, i.e. the node operating system - NX/2 is instrumented to generate execution trace at run time [27]. The user program need not be modified except that two commands, *monitor_init()* and *monitor_end()*, are added to the node program to initiate and terminate the software monitoring facilities. The *monitor_init()* command also specifies the size of the trace buffer to be allocated by the operating system.

The instrumentation is done by inserting *probe points* at various places in the node operating system. An event is triggered during the execution of the application program when any of the probe points is hit. The monitoring facilities record each event onto a trace buffer in the memory together with the timestamp and an optional number of parameters of the particular event. At the end of the monitoring, when the *monitor_end()* command is executed, the trace buffer is sent to a background process in the host which stores the trace in the file system. One trace file is generate by each of the processor nodes. Hence, eight separate trace files will be generated if the application program is executed in an eight node configuration.

The places where we insert probe points are:

(1) entry and exit points of system calls (message and non-message),
(2) entry and exit points of interrupt service routines (message and non-message),
(3) points where the CPU enters and leaves idle state,
(4) points where the operating system performs message related activities, like start sending a message, set up receiving for a message, and
(5) entry and exit points of buffer operations, e.g. buffer allocation and buffer copying.

The first three types of probe points enable us to observe the CPU state and the specific operating system routine executed. The fourth one provides the information about communication activities, and the last type of probe points helps us determine the cost of various buffer operations.

Since the trace buffers are quite large, they are sent back to the host only after the monitoring is terminated in order not to interfere with the communication of the user program. The drawback of this approach is that the size of the trace from each node cannot exceed the buffer allocated, which is limited by the size of the main memory. In general, this means that the trace buffer available in each node is about two to three Megabytes depending on the memory requirement of the application program. One way to solve this problem is to stop the execution of the

application program once any of the trace buffers is filled up, and to resume execution only after all the trace buffers are sent back to the host. The second method is to sample the execution instead of monitoring the whole user program. The above methods were not adopted in this experiment because, for most of the programs we measured, the available buffer size is quite adequate in recording the entire program execution. In addition, the continuous trace allows us to do trace driven simulation which will be more accurate if an uninterrupted trace is available.

Once the traces are collected from every node, there are several post-processing programs available to interpret and summarize the results. The first program — *event_list* will print out the events one by one, together with their timestamps and optional parameters. The second one — *trace_summary* summarizes the trace file of a single node. The summary includes CPU utilization, message transmission statistics, and system call/interrupt activities. There are also several filters available to extract relevant information from the trace files for further analysis. The information that can be extracted includes message length and destination, message arrival interval, the computation time between communication, and the waiting time in receiving messages. Another post-processing program — *trace2sim*, which will be described in Section 5, can convert the traces to the input of the trace driven simulator HSIM.

Table 2 shows the input data used in the measurement, average number of messages sent and received, average message size, size of trace file, and number of trace events, of each node. As shown in the table, software monitoring slows down the execution of the benchmarks by 0.2 to 35 percent. Since the emphasis of the monitor is on communication and system activities, the percentage of the overhead is directly proportional to the rate of message and system calls. The channel routing program and the EISPACK routine have high message transmission rates, and the test pattern generation program makes frequent system calls, therefore their overhead percentages are very high. In all of the post processing programs, the delay caused by software monitoring is subtracted from the timing calculation in order to minimize the inaccuracy.

We were concerned about the fact that software monitoring might change the parallel program behavior. In order to understand the effect of monitoring, we compared the results and the message sequences of the benchmarks with and without monitoring. The results of all the benchmarks remained the same regardless of monitoring except the cell placement and channel routing programs, where the simulated annealing technique (a probabilistic method) is used and the result is different in each execution. Table 3 shows the percentage of out-of-order message arrivals in node 0 of an 8 node hypercube. In general, the out-of-order arrival rate is proportional to the monitoring

overhead. The fault simulation program uses dynamic load balancing technique, hence, after the first out-of-order message, the message sequence is totally different from the one without monitoring. By estimation, its out-of-order percentage is around one percent. Note that even without monitoring, the test pattern generator has 2.3 percent of the messages arrived out of order when comparing different executions, while it is 1.6 percent for the EISPACK routine. Because of the low out-of-order message arrival percentages, the errors in the measurement results on the CPU utilization and various timing distributions should be very small.

4. PERFORMANCE MEASUREMENT RESULTS

This section presents the results of the measurement of the eleven benchmarks. We show the characteristics of the computation and communication workload, and the utilization of the CPU. Models for the locality of message length and the distributions of various time intervals are also proposed.

4.1. Message Length and Destination

Figures 1 and 2 summarize the message destination and length distributions of node zero for four of the applications: placement, extraction, QR and FFT, running on a 16 nodes hypercube. The cell placement program communicates only with distance one and two nodes, with most of the messages going to one distance-one node. The message length varies from 0 to 4K bytes, but most are under 100 bytes. The circuit extractor sends almost all the messages to distance-one nodes, mostly with length less than 100 bytes or around 16K bytes. The nodes in the QR factorization program communicate in a ring fashion, sending 1028 byte messages to their successors and 1024 byte messages to their predecessors. In the FFT program, the nodes send equal amount of messages to all of the distance-one nodes, all with the same length.

The distributions for the channel router are similar to those of the placement program. The destination distributions of the programs that use tree-broadcast communication (fault simulation, Gaussian, simplex and EISPACK) are similar to that of the FFT.

Table 4 summarizes the spatial locality of message passing for the eleven benchmarks on a 16 node hypercube. Except for the test generation program, all the programs send most of the messages to the nearest neighbors. The communication patterns do not change for a 16 node configuration. This indicates that the message destination has spatial locality which favors nearest neighbors. Evenly distributed message destination is often assumed in both experimental [9] and analytical [5] approaches of computer network performance analysis. This assumption is not

true in most of the applications for hypercube because the programmers often try to avoid sending long distance and global messages.

In addition to the spatial locality, we were also interested in the temporal locality of the message destination, i.e. we wanted to know if the message destination repeats in time. We model the temporal locality using the *least recently used* (LRU) stack model [12, 13], in which the message destination is treated like memory address. In the LRU stack model of stack size n, each node has a stack which contains the n most recent message destinations. If the destination of the next message is in the stack, it is called an LRU stack hit, otherwise it counts as a miss. In Table 5, we show the average LRU stack hit rate of the eleven applications when stack size varies from one to four. Four out of the eleven benchmarks have high hit rates when the stack size is only one. They are the ones that do not require broadcast communication. All programs have high hit rates when the stack size is four, except the Navier-Stokes algorithm, where grid partition is used and some nodes have up to 8 neighbors. In the programs that use tree-broadcast communication, the hit rates are low when the stack size is less than four. This is because each node alternately sends messages to the four distance-one neighbors when doing broadcasting. However, the message destination is still highly predictable because the broadcasting is used throughout these programs.

If the message length is also treated like a memory address, then the sizes of messages also have spatial and temporal localities. From the distributions of the message length in Figure 2, it is clear that the lengths concentrate around only one or two peaks in every application, which shows that the lengths have high spatial locality. To investigate the temporal locality of the message length we propose a model called the **Most Recent Message Length** (MRML) stack model, which is based on the LRU stack model. In the MRML stack model of stack size m, each node has several stacks each corresponds to one possible message destination and contains the m most recent message sizes send to that node, with the most recent length in the first location. The message is considered a MRML stack hit if its length is smaller than or equal to the longest of the m last messages send to the same destination, i.e. the largest in the MRML stack. Note that this model is for the upper bound of the message length.

For example, if a node x sends messages to four other nodes, there will be four MRML stacks for node x each corresponds to one destination. Stack $x.y$ contains the sizes of the most recent 3 messages node x sent to y, assuming a stack size of 3. If the current content of $x.y$ is (6,10,5) and the next message is of size 7, then it is a MRML stack hit. The stack distance is two, since 7 is smaller than the second stack location but larger than the first one. The new $x.y$ stack will be (7,6,10). If the next message length is 12, then it counts as a miss.

Table 6 shows the average MRML stack hit rate of the eleven benchmarks when the stack size increases from one to four. The left hand side of this table reports the pure MRML stack hit rate without any limitation on minimum message length. Nine out of the eleven benchmarks have hit rates higher than 76 percent when the stack size is only one. When the stack size is two, all of the hit rates are greater than 88 percent except the circuit extractor. The hit rates of QR and FFT are almost 100 percent, since, for each destination node, the message length is constant.

In the right hand side of Table 6, following the message transmission protocol of iPSC/2, we assume that the minimum buffer size is 100 bytes, hence messages shorter than 100 bytes will be treated as 100 bytes. The hit rates for most of the programs improves substantially. The circuit extraction program often sends a short message and then a very long message alternately to each destination node, therefore the hit rate is very low when the stack size is one, and the minimum message size requirement does not make any difference.

We now show the predictability of the communication patterns by combining both of the models for the temporal localities of message destination and length. The message is considered a hit only when its destination is in the LRU stack *and* its length hits the MRML stack. Table 7 shows the average message hit rates of the benchmarks when the MRML stack size increases from one to two, and the destination LRU stack size varies from one to three. We also assume the minimum message size to be 100 bytes. Compared to the pure destination LRU stack hit rates shown in Table 5, the hit rates for most of the programs decline a little bit, while those of QR, FFT and EISPACK stay the same since the length prediction always hits. For the test generator, circuit extractor and Navier-Stokes the hit rates drop by a large amount, but recover substantially when the MRML stack size becomes two.

These measurements indicate that message length and destination have high spatial and temporal localities, hence the communication patterns are highly predictable. By utilizing these locality properties, the cost of message communication could be reduced by intelligent routing controllers and efficient buffer management strategies. In a circuit-switched message routing paradigm, the circuit can be kept connected even after the message is transmitted [28]. If the next message also goes to the same destination, this circuit can again be used, thereby reducing circuit set-up time. Software overhead can also be reduced by predicting the sizes of the incoming messages and pre-allocating buffers for them [29]. Broadcast and reduce (finding global minimum/maximum) operations are used substantially. If special hardware can be used to perform these operations, the communication speed will be greatly improved.

Note that the above measurement results are not particular to these benchmarks and the iPSC/2 machine, they can be generalized to other programs and other medium-grain message passing multiprocessors. The programmers try to localize the computation and communication in these machines in order to minimize the message latency and the contention in communication channels. They are more likely to use only a few types of messages, hence the message sizes will also be localized.

4.2. Communication and Computation Workload

In experimental and analytical studies of computer network performance, different kinds of message interarrival time distributions are often assumed, including exponential [5], normal distribution [9], and constant interval [8]. We wanted to verify these assumptions by analyzing the data collected from the measurements.

Figure 3 shows the distribution of the time interval between message sending of node zero for six of the benchmarks running on an eight node hypercube. The interarrival time between messages $i-1$ and i, t_i, can be broken down into three parts, which are the computation time, c_i, the waiting time for receiving messages, w_i, and the system time for processing messages, s_i. Hence, $t_i = c_i + w_i + s_i$. Figures 4 and 5 show the distributions of the computation time and waiting time interval of node zero. The system time for processing messages can be approximated by a linear function of the message size, $s_i = f(l)$, hence it is very predictable and will not be discussed here.

The message interarrival distribution of the placement program has two peaks, which correspond to the two types of messages it sends, ring broadcast and cell exchange. The distributions of the placement program are similar to those of the channel router, because they use similar algorithms and have similar message transmission patterns. The circuit extractor sends infrequent messages and, in the first phase, spends most of the time waiting for messages from the host. Its computing time interval is irregular. As expected, the distributions of the numeric algorithms, especially those of the FFT, are very regular. Each of the three distributions of the QR program has a single peak, and those of the FFT have two peaks. Most of the distributions for the rest of the numeric benchmarks are very regular, similar to those of the QR and FFT.

Due to the multiple peaks, the distributions clearly cannot be modeled by simple exponential, normal or gamma probability density functions (pdf). Therefore, we model the distributions by multi-stage pdf's [10], defined as:

$$f(t) = \sum_{i=1}^{n} a_i \; g_i(t),$$

where $a_i \le 1$ is the weight for each stage, $\sum_{i=1}^{n} a_i = 1$, and n is the number of stages. The probability density functions $g_i(t)$ can all be chosen from one of the following:

(1) Normal:

$$g_i(t) = \frac{1}{\sigma_i \sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{t-\mu_i}{\sigma_i}\right)^2\right], \quad t \ge 0,$$

where $\mu_i > 0$, and $\sigma_i > 0$.

(2) Shifted hypoexponential:

$$g_i(t) = \begin{cases} 0, & t \le s_i, \\ \dfrac{\lambda_{1i}\lambda_{2i}}{\lambda_{1i}-\lambda_{2i}} \left(e^{-\lambda_{1i}(t-s_i)} - e^{-\lambda_{2i}(t-s_i)}\right), & t > s_i, \end{cases}$$

where $\lambda_{1i} \ne \lambda_{2i}$, and $s_i > 0$ is the offset in time.

(3) Shifted gamma:

$$g_i(t) = \begin{cases} 0, & t \le s_i, \\ \dfrac{\lambda_i{}^{\alpha_i} \; (t-s_i)^{\alpha_i - 1} \; e^{-\lambda_i (t-s_i)}}{\Gamma(\alpha_i)}, & t > s_i, \end{cases}$$

where $\alpha_i > 0$, $\lambda_i > 0$ and $s_i > 0$. The gamma function is defined by the integral:

$$\Gamma(\alpha_i) = \int_0^{\infty} x^{\alpha_i - 1} e^{-x} \, dx, \quad \alpha_i > 0.$$

Thus, we have multi-stage normal, multi-stage shifted hypoexponential, and multi-stage shifted gamma pdf's. To determine the best model for the empirical distributions shown in Figures 3 through 5, we applied curve fitting technique (nonlinear regression) [30] to fit the distributions into the above models. Similar studies have been reported by [11] for file usage analysis and [10] for resource utilization. The criterion of the nonlinear regression procedure is to minimize the sum of square of errors (SSE) when estimating the parameters of a nonlinear model. Note that in the curve fitting process, we also used other types of multi-stage pdf models such as the multi-stage exponential distribution, but none of them converged as well as the above three models, hence only the results from these three are reported.

Table 8 shows the SSE of curve fitting for the message interval distribution of the placement program (shown in Figure 3.a). The models used are the three types of distributions with 1, 2 and 3 stages. The results indicate that two or three stage normal pdf models this distribution better than the other two types of pdf's. The same experiment repeated for the message interarrival time distribution of QR factorization program, shown in Table 9, also confirms this observation. By performing nonlinear regression on all the empirical distributions including the computation and waiting time intervals, we found that, while not the best model in every case, the multi-stage normal pdf has better overall performance than the other two models. A two-stage normal pdf models the distributions reasonably well, as can be seen from Tables 8 and 9, therefore it was chosen to model the empirical data. This confirms with the experience that the normal (or Gaussian) distribution models some real world problems well.

Table 10 shows the two-stage normal pdf parameters for the distributions of cell placement and QR factorization programs. For the placement program, the two model parameters, μ_1 and μ_2, correspond to the two peaks in its message interarrival distribution, as shown in Figure 3.a. The parameters of its computing time and waiting time distributions indicate that the task granularity is small and the waiting times for messages are also not long. The waiting time distribution of the QR program can be modeled by one normal pdf with a small variance. Since $t_i = c_i + w_i + s_i$, as previously stated, and the waiting time w_i and system time s_i are almost constant, the message interval and computing time distributions are very similar except that one is time shifted by about 1.4 mSec from the other, as can be seen from μ_1 and μ_2 of the two distributions.

In this subsection, we have shown that the distributions of message arrival, computation time and waiting time can be modeled reasonably well by the two-stage normal pdf models. The results of such a study can be useful in the generation of realistic synthetic benchmarks and accurate analytical models for the evaluation of hypercube applications and architectures.

We also note that the statistics on the computation time between messages relate to the computation task granularity, and the statistics on message length and destination distributions represent the inter-task dependencies in computation task graphs. Our results will therefore provide realistic models of task graphs for load balancing researchers.

4.3. CPU Utilization

The utilization of the CPU gives a good indication of the efficiency of the parallel algorithm. Table 11 shows the CPU utilization of the eleven applications running on an eight node hypercube. The total execution time of the CPU is divided into three parts, which are system, user and idle times. For each of the three parts, the average percentage is shown together with the maximum and minimum among the eight nodes.

The system time includes both the message-related and message-unrelated calls. In the benchmarks, most of the system calls are message-related, hence the system percentage is proportional to the rate of message sending and receiving. The variations in the user time percentages indicate the computation workload distribution of the applications. The CPU idle time occurs mostly in waiting for the completion of message sending and receiving. Lower idle time translates to more efficient use of system resources.

The channel router, QR, and EISPACK programs are communication intensive. In the test generation program, the communication and computation workloads are not balanced. This is because processor zero acts as the scheduler of the parallel branch and bound search, hence it has the lowest user time, highest system and idle time percentages. The circuit extraction and fault simulation programs, which do explicit load balancing, are well load-balanced. The first phase of the circuit extractor is I/O intensive, and the second phase is computation intensive. The fault simulation, FFT, and Navier-Stokes programs are very computation intensive. Most of the numeric applications are well load balanced.

Figure 6 shows the CPU utilizations of the cell placement and QR programs when the number of processors increases from 4 to 16. As expected, for both of the programs, the user time percentage goes down and the system time percentage goes up when the size of the hypercube grows. This is because the granularity of the computation task becomes smaller. Note that the user time percentage of the placement program declines faster than that of the QR program because the placement program sends more messages per node when the hypercube size grows. Figure 7 shows the CPU utilizations of the QR program when the data size increases. It again confirms that when the task granularity becomes larger, the user CPU utilization also increases.

5. TRACE DRIVEN SIMULATION ENVIRONMENT

A trace driven simulator for hypercube — HSIM has been developed to study the behavior of the communication hardware under real workload. By using real communication trace as input, this simulator is shown to be both

realistic and reasonably accurate.

HSIM, written in C++, is based on CSIM [31], which is a process oriented simulation language. In HSIM, concurrent hardware devices such as the CPU and DMA are modeled by CSIM processes. The emphasis of HSIM is in communication activities, therefore the operations of the DMA, routing elements and communication links are modeled in detail. The computation time of CPU between message sending and receiving is derived directly from the traces.

The inputs to HSIM are the *communication traces* which are derived from the execution traces of real applications by the post-processing program — *trace*2*sim*. Figure 8 shows a sample input to HSIM for a two node hypercube. Here the two processors exchange two messages while performing some computation in between. The *csend()/crecv()* instructions and the computation time are derived from measurement, while the software and hardware latencies of communication are modeled by the HSIM. Since HSIM actually executes *csend()* and *crecv()* commands, it has to wait for the completion of message transmission before continuing execution. Hence, it is realistic in modeling parallel program execution behavior. Note that Figure 8 is just an example. The simulation studies presented in the following section used the traces of real application programs running on an 8-node hypercube. The characteristics of these traces were reported in Section 4; for example, Figures 3, 4, and 5 report the various workload distributions, while Table 10 shows the model parameters for the workload.

HSIM models the architecture and communication protocol of the iPSC/2 [3, 32], which uses circuit-switched message routing scheme. Each communication link consists of two bi-directional, bit-serial channels. Two DMA channels are used to transmit data, one for memory to routing logic transfers (the *outgoing DMA*), the other for transfers in the opposite direction (the *incoming DMA*).

HSIM has been tuned to match the performance of the real machine. Table 12 compares the results of measurement versus simulation for the QR factorization program running on a four node hypercube. The total system, user and idle times of each node are compared. The errors in the user time are negligible since the computing time is derived from the measurement. The errors of the system and idle times are within four percent, which indicate that the simulator models the communication mechanism quite accurately. This enables us to justifiably put reliance on the results of the trace driven studies of link and DMA channel behavior to be presented in the following section.

6. SIMULATION RESULTS

This section presents the results of the trace driven simulation on DMA and link utilizations. The inputs to the simulator were converted from the same set of traces that we characterized in Section 4. Table 13 shows the utilizations of the incoming and outgoing DMA channels, each with the minimum, maximum and average utilizations among the eight nodes, and the average service time for each message. Also shown are the overlap between the two channels, the overall DMA utilization and the CPU and DMA overlap percentage. Table 14 shows the link utilization and average bandwidth for each of the three outgoing links of node zero and four, and the average utilization and bandwidth of all the links.

The DMA utilization table indicates that there is not much overlap between CPU and DMA. Part of the reason is that the DMA utilizations are not high anyway. In addition, most of these benchmarks use blocking communication calls, therefore almost all of the overlap occurs between CPU and the incoming DMA channel. The execution time could be reduced by overlapping communication with computation using non-blocking communication calls.

The fault simulation program uses non-blocking communication extensively, therefore it has high CPU/DMA overlap rate relative to its DMA usage. The channel router and the EISPACK routine have high DMA and link utilizations, this confirms the measurement result that they are communication intensive applications. Also, both of their CPU/DMA and incoming/outgoing channel overlaps are high. This is not due to explicit non-blocking message calls, but rather, it is caused by the asynchronous communication pattern of the processors where messages arrive when they are doing computation or sending out messages.

The QR factorization program is also a communication intensive application, as can be seen from the high DMA and link utilizations. However, in contrast to the channel router, QR has low overlap between CPU/DMA and incoming/outgoing channels. This is because QR has a more regular communication pattern and its computation load is more balanced. It is interesting to note that the channel router has a higher link utilization rate than QR, but its average link bandwidth is almost half of that of QR. The reason is that most of the messages of the channel router are very short, while QR sends long messages of 1K bytes. Hence the data transmission rate of QR is much higher.

The circuit extraction and FFT programs have longer average DMA service times because their average message sizes are larger than the other programs. Except the FFT program, where every node communicates with all of

its distance-one neighbors, there exist high variations among the utilizations of the links of each node.

Table 15 gives the resource utilization summary of the cell placement program running on different hypercube sizes and with different input data. The CPU utilization part of the table again confirms the results of measurement, which were shown in Figure 6. As the cube size grows, the program becomes less computation intensive and more communication intensive. Therefore, the user time percentage goes down and the utilization of DMA increases. However, the average utilization and bandwidth decrease despite the fact that there are more messages. This is because the total number of links increases in a higher rate than that of node number. When data size decreases, the program also becomes less computation intensive.

For all the benchmarks except channel router, QR and EISPACK, the DMA and link utilizations are very low, and the average bandwidths are well under the 2.8 Megabytes per second maximum. The low utilization rate indicates that a mesh architecture could probably do as well as the hypercube for these programs. In addition, in the iPSC/2 system, the delays in messages are not due to congestion in the links. With the existing hardware, better performance could be achieved just by reducing software overhead. The low transmission rate of small messages also shows that the system is not suitable for exploring low level parallelism in the applications.

7. CONCLUSION

We have investigated the computation and communication behavior of eleven parallel programs on a message passing multicomputer. An LRU stack model was used to study the temporal locality of message destinations. We also proposed a model for the temporal locality of message length which is called the *Most Recent Message Length* (MRML) stack model. By using these models, we have demonstrated that both the message destination and length have high temporal and spatial localities. These measurements and models have shown that the communication cost can be reduced by intelligent routing controllers and efficient buffer management strategies.

Multi-stage probability density functions were used to model computation task granularity distribution, message interarrival distribution, and the distribution of waiting times in receiving messages. It has been found, by nonlinear regression technique, that two-stage normal pdf's are most suitable in modeling the above distributions. Note that these measurement results are not particular to these benchmarks and the iPSC/2 hypercube, they can be generalized to other medium-grain distributed memory multiprocessors.

We also described a trace driven simulator which was used to study the communication activities in detail. The results of simulation show that the utilizations of DMA and links are very low, which indicates that there are still rooms for improvement even with the existing hardware.

By modeling the computation and communication workloads of realistic parallel programs, this study provided us some very important information that will be useful in both analytical and experimental studies of hypercube communication networks. Based on the models for distributions and localities, representative synthetic benchmarks and accurate analytical models can be designed which will hopefully produce more precise results. System designers can also benefit from these information in developing more efficient communication hardware and software.

REFERENCES

[1] C. L. Seitz, ‘‘The Cosmic Cube,’’ *Communications of the ACM*, vol. 28, pp. 22-33, Jan. 1985.

[2] J. Tuazon, J. Peterson, and M. Pniel, ‘‘Mark IIIfp Hypercube Concurrent Processor Architecture,’’ in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 71-80, Jan. 1988.

[3] P. Close, ‘‘The iPSC/2 Node Architecture,’’ in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 43-50, Jan. 1988.

[4] J.-M. Hsu and P. Banerjee, ‘‘Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer,’’ in *Proc. 17th Int’l Symp. on Computer Architecture*, Seattle, WA, pp. 260-269, May 1990.

[5] P. Kermani and L. Kleinrock, ‘‘Virtual Cut-Through: A New Computer Communication Switching Technique,’’ *Computer Networks*, vol. 3, pp. 267-286, Sep. 1979.

[6] S. H. Bokhari, ‘‘Partitioning Problems in Parallel, Pipelined and Distributed Computing,’’ *IEEE Trans. on Computers*, vol. 37, No. 1, pp. 48-57, Jan. 1988.

[7] R. Mirchandaney, D. Towsley, and J. A. Stankovic, ‘‘Analysis of the Effects of Delays on Load Sharing,’’ *IEEE Trans. on Computers*, vol. 38, No. 11, pp. 1513-1525, Nov. 1989.

[8] D. C. Grunwald and D. A. Reed, ‘‘Networks for Parallel Processors: Measurements and Prognostications,’’ in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 610-619, Jan. 1988.

[9] G. Buzzard and T. Mudge, ‘‘High Performance Hypercube Communication,’’ in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 600-609, Jan. 1988.

[10] M. C. Hsueh, R. K. Iyer, and K. S. Trivedi, ‘‘Performability Modeling Based on Real Data: A Case Study,’’ *IEEE Trans. on Computers*, vol. 37, No. 4, pp. 478-484, April 1988.

[11] M. V.-S. Devarakonda, ‘‘File Usage Analysis and Resource Usage Prediction: a Measurement-Based Study,’’ Technical Report CSG-79, Coordinated Science Lab., Univ. of Illinois, Urbana, IL, Dec. 1987.

[12] D. A. Reed and R. M. Fujimoto, in *Multicomputer Networks: Message-Based Parallel Processing*. Cambridge, MA: MIT Press, 1987.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, ‘‘Evaluation Techniques for Storage Hierarchies,’’ *IBM Systems Journal*, vol. 9, 1970.

[14] J. Sargent and P. Banerjee, ‘‘A Parallel Row-Based Algorithm for Standard Cell Placement with Integrated Error Control,’’ in *Proc. 26th Design Automation Conf.*, Las Vegas, NV, Jun. 1989.

[15] R. Brouwer and P. Banerjee, ‘‘A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor,’’ in *Proc. Int. Conf. on Computer Design*, Rye Brook, NY, pp. 4-7, Oct. 1988.

[16] K. P. Belkhale and P. Banerjee, ‘‘PACE2: An Improved Parallel VLSI Extractor with Parametric Extraction,’’ in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1989.

[17] S. Patil and P. Banerjee, ‘‘A Parallel Branch and Bound Approach to Test Generation,’’ in *Proc. 26th Design Automation Conf.*, Las Vegas, NV, Jun. 1989.

[18] S. Patil and P. Banerjee, ‘‘Fault Partitioning Issues in an Integrated Parallel Test Generation/ Fault Simulation Environment,’’ in *Proc. 1989 Int’l Test Conf.*, Washington, DC, pp. 718-726, Aug. 1989.

[19] G. H. Golub and C. F. Van Loan, *Matrix Computations*. Baltimore: The John Hopkins University Press, 1983.

[20] A. Pothen, S. Jha, and U. Vemulapati, ‘‘Orthogonal Factorization on a Distributed Memory Multiprocessor,’’ *Proc. 2nd SIAM Conf. on Hypercube Computers and Applications*, pp. 587-596, 1987.

[21] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, in *Solving Problems on Concurrent Processors, Vol. I*. Prentice-Hall, 1988.

[22] G. A. Geist and M. T. Heath, ‘‘Matrix Factorization on a Hypercube Multiprocessor,’’ in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN, Aug. 1985.

[23] C. B. Stunkel, ‘‘Linear Optimization via Message-Based Parallel Processing,’’ *Proc. 1988 Int’l Conf. on Parallel Processing*, vol. 3, pp. 264-271, Aug. 1988.

[24] R. A. Fatoohi and C. E. Grosch, ‘‘Implementation and Analysis of a Navier-Stokes Algorithm on Parallel Computers,’’ *Proc. 1988 Int’l Conf. on Parallel Processing*, vol. 3, pp. 235-242, Aug. 1988.

[25] J. J. Dongarra and C. B. Moler, ‘‘EISPACK - A Package for Solving Matrix Eigenvalue Problems,’’ in *Sources and Production of Mathematical Software*, ed., W. Cowell. Prentice Hall, pp. 68-87, 1984.

[26] P. Pierce, ‘‘The NX/2 Operating System,’’ in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 384-390, Jan. 1988.

[27] D. C. Rudolph, ‘‘Performance Instrumentation for the Intel IPSC/2,’’ Technical Report UIUCDCS-R-89-1524, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, July, 1989.

[28] J.-M. Hsu and P. Banerjee, ‘‘Hardware Support for Message Routing in a Distributed Memory Multicomputer,’’ in *Proc. 1990 Int’l Conf. on Parallel Processing*, vol. 1, St. Charles, IL, pp. 508-515, Aug. 1990.

[29] J.-M. Hsu and P. Banerjee, ‘‘A Message Passing Coprocessor for Distributed Memory Multicomputers,’’ in *Proc. Supercomputing ’90 Conf.*, New York, NY, pp. 720-729, Nov. 1990.

[30] SAS Institute Inc., ‘‘SAS User’s Guide: Statistics,’’ Version 5, Cary, NC, 1985.

[31] H. Schwetman, *CSIM Reference Manual (Revision 13)*. Austin, TX: Microelectronics and Computer Tech. Corp., May, 1988.

[32] S. F. Nugent, ‘‘The iPSC/2 Direct-Connect Communications Technology,’’ in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988.

LIST OF FIGURE CAPTIONS

Figure 1. Message Destination Distribution of Node 0.

Figure 2. Message Length Distribution of Node 0.

Figure 3. Message Interarrival Time Distribution of Node 0.

Figure 4. Computation Time Distribution of Node 0.

Figure 5. Waiting Time Distribution of Node 0.

Figure 6. CPU Utilization as a Function of Hypercube Size.

Figure 7. CPU Utilization as a Function of Data Size for QR Program on 8 Processors.

Figure 8. HSIM Input for a Two Node Hypercube.

LIST OF TABLE CAPTIONS

Table 1. Speedup of the Application Programs Running on iPSC/2 Hypercube.

Table 2. Monitoring Statistics (on an 8 Node Hypercube).

Table 3. Out-of-Order Message Arrivals due to Monitoring (Node 0 of an 8 Node Hypercube).

Table 4. Average Message Distance Distribution (16 Node Hypercube).

Table 5. Message Destination Hit Rates as a Function of Destination LRU Stack Size (16 Node Hypercube).

Table 6. Message Length Hit Rates as a Function of MRML Stack Size (16 Node Hypercube).

Table 7. Message Destination and Length Hit Rate (16 Node Hypercube).

Table 8. SSE of Curve Fitting on Message Interval Distribution of Node 0 of the Cell Placement Program.

Table 9. SSE of Curve Fitting on Message Interval Distribution of Node 0 of the QR Factorization Program.

Table 10. Two-stage Normal pdf Model Parameters for Placement and QR Programs (Node 0).

Table 11. CPU Utilization (on an 8 Node Hypercube).

Table 12. Comparison of Measurement and Simulation Results (QR factorization of 128×128 matrix).

Table 13. DMA Utilization (8 Node Hypercube).

Table 14. Link Utilization (8 Node Hypercube).

Table 15. Placement Program on Different Number of Nodes and Data Size.

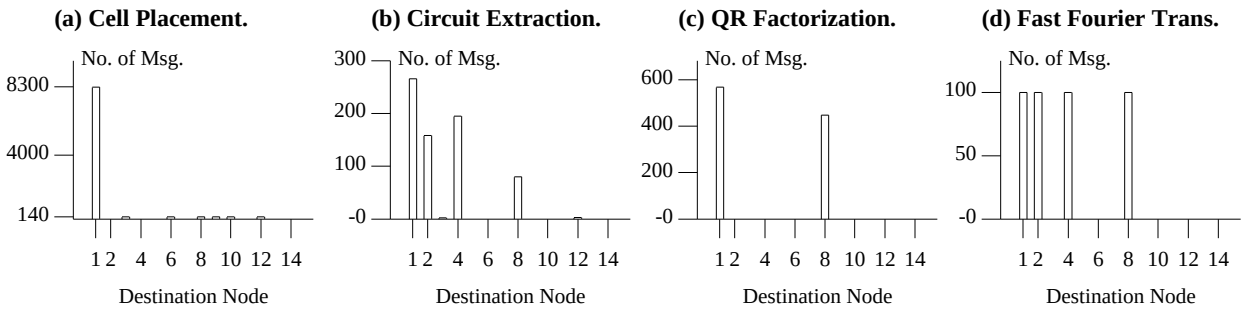

Figure 1. Message Destination Distribution of Node 0.

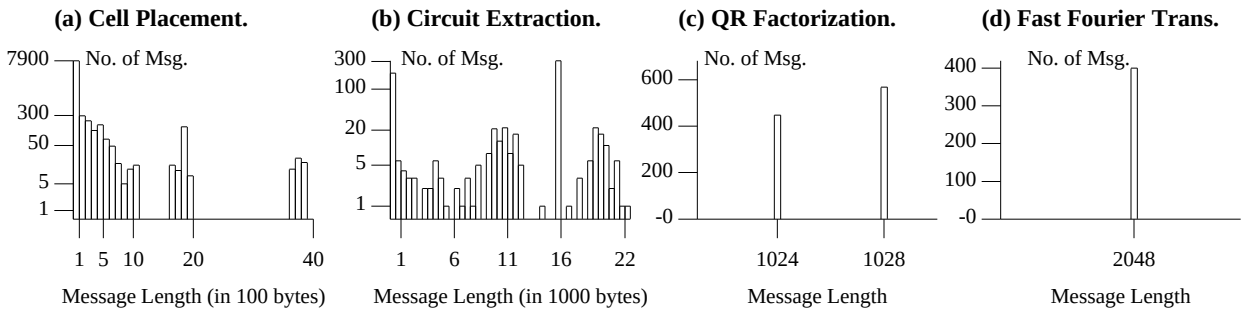

Figure 2. Message Length Distribution of Node 0.

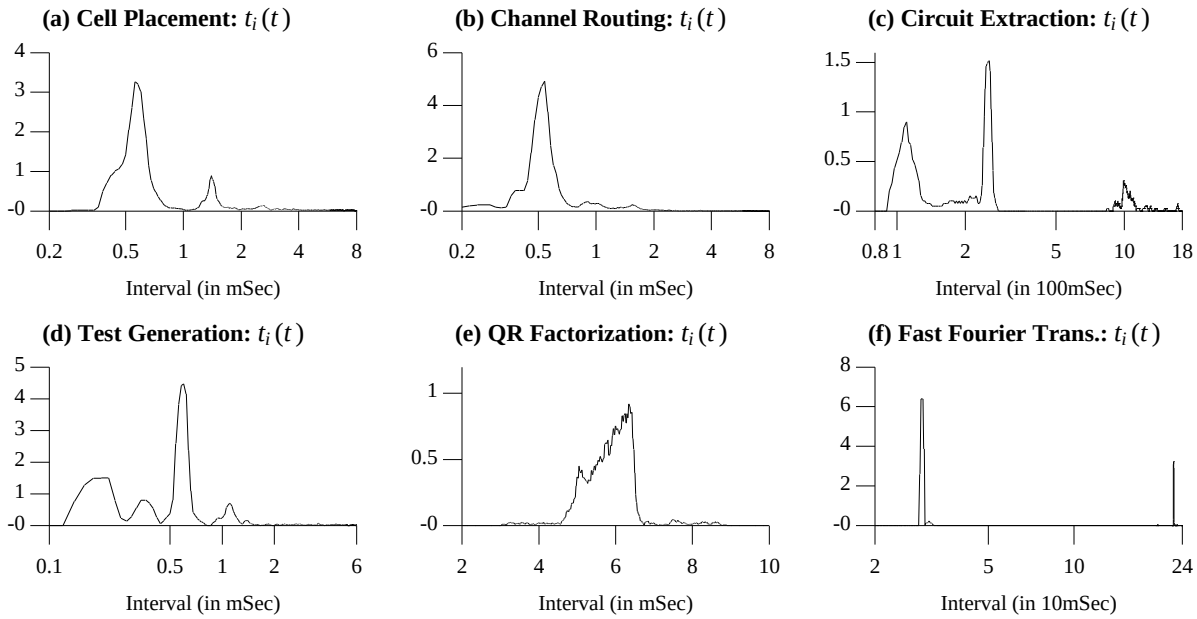

Figure 3. Message Interarrival Time Distribution of Node 0.

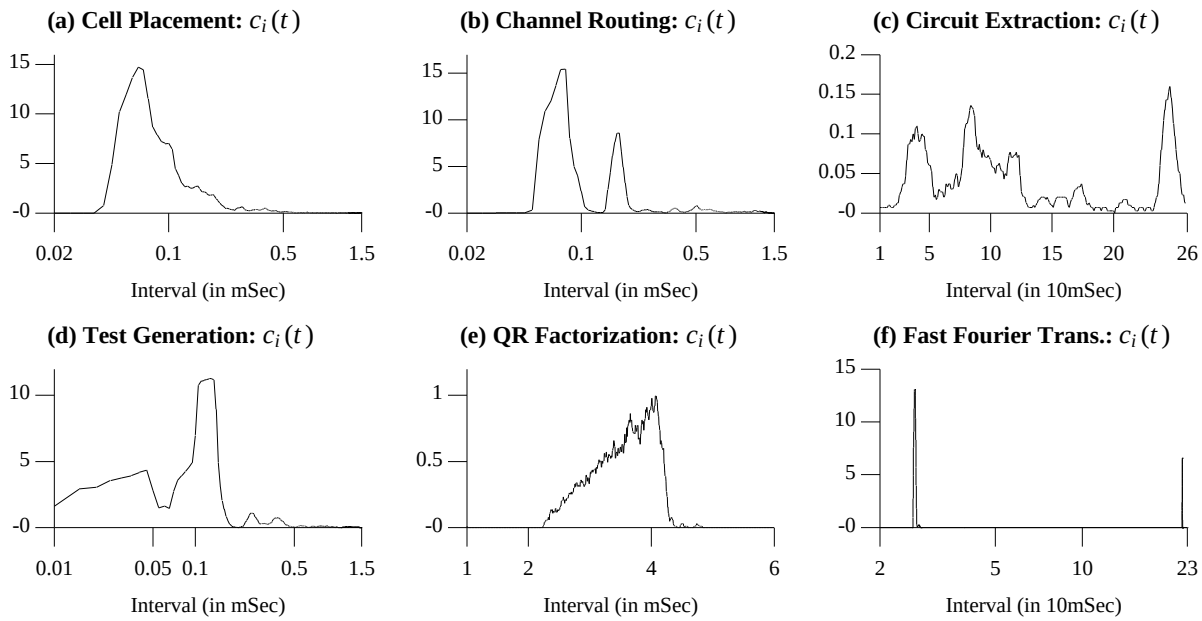

Figure 4. Computation Time Distribution of Node 0.

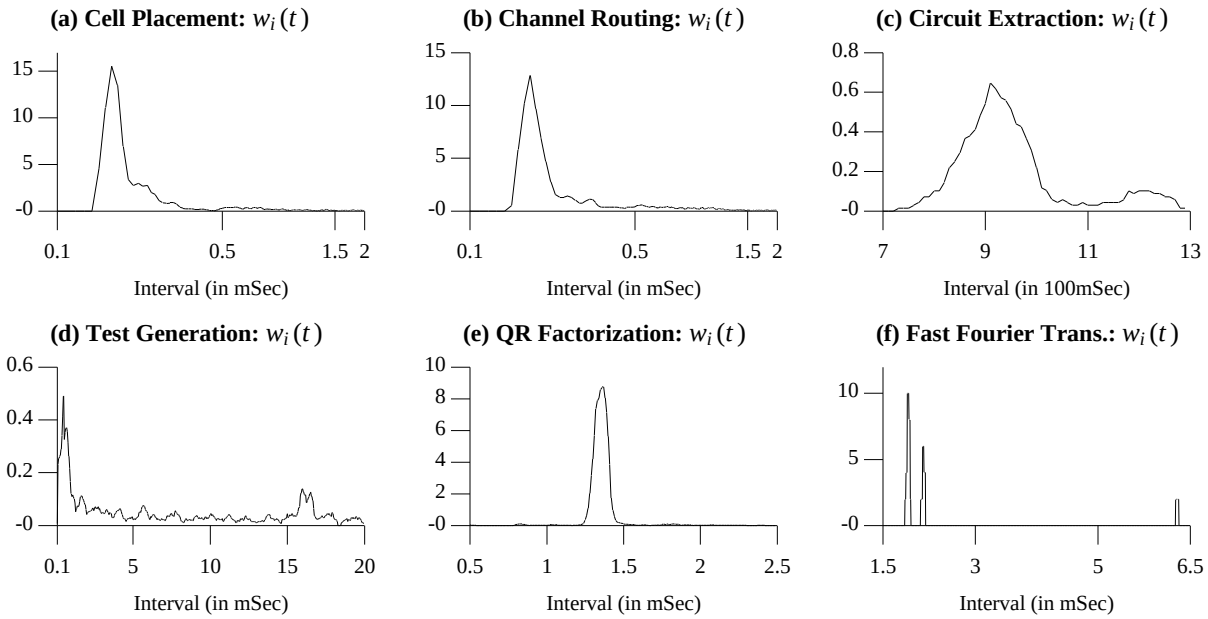

Figure 5. Waiting Time Distribution of Node 0.

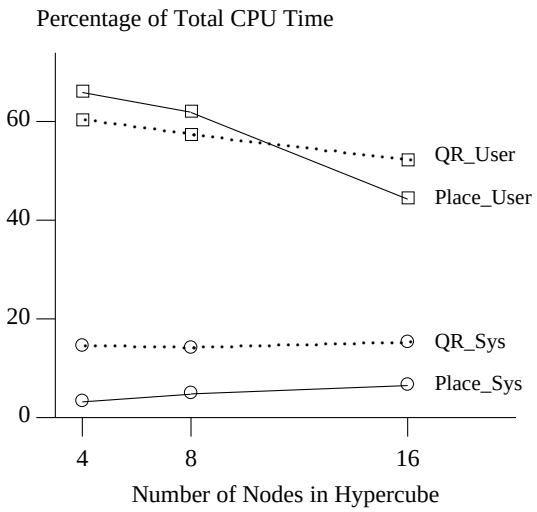

Figure 6. CPU Utilization as a Function of Hypercube Size.

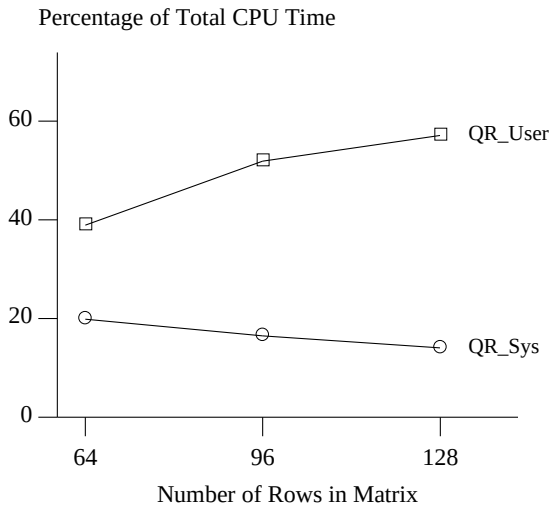

Figure 7. CPU Utilization as a Function of Data Size for QR Program on 8 Processors.

```
Node 0:                                   Node 1:
        Start node 0.                             Start node 1.
        Compute 350 µSec.                         Compute 500 µSec.
        Csend(type 1, length 100, to node 1).     Crecv(type 1, length 100).
        Compute 400 µSec.                         Compute 500 µSec.
        Crecv(type 2, length 500).                Csend(type 2, length 500, to node 0).
        Compute 450 µSec.                         Compute 400 µSec.
        End.                                      End.
```

Figure 8. HSIM Input for a Two Node Hypercube.

Application Program	Number of processors 4	8	16
Placement	3.9	4.7	7.9
Routing	5.9	10.2	17.7
Extraction	4.4	8.6	12.6
Test Gen.	3.2	7.2	13.7
Fault Sim.	3.1	5.4	8.8
QR	2.5	4.9	9.2
FFT	3.6	7.0	12.9
Gaussian	3.2	4.6	6.0
Simplex	3.8	6.2	10.3
Navier	3.7	6.4	11.4
EISPACK	0.9	0.7	0.5

Table 1. Speedup of the Application Programs Running on iPSC/2 Hypercube.

Application Program	Input Data	Monitoring Statistics (Average/Per Node)				Running Time		
		Messages Sent & Recv'd	Average Msg. Size (Bytes)	Trace File Size (Bytes)	No. of Events	Monitor Off (Sec)	Monitor On (Sec)	Overhead (%)
Placement	286 cells	12316	153.7	1036K	154K	89.84	93.82	4.4
Routing	229 nets	18780	17.1	1212K	159K	11.91	16.08	35.0
Extraction	66K rectangles	791	8519.0	122K	21K	181.76	182.14	0.2
Test Gen.	3540 nets	872	113.4	801K	158K	15.32	19.81	29.3
Fault Sim.	3540 nets	1457	293.0	410K	67K	114.03	115.66	1.4
QR	128×128 matrix	4128	1206	518K	71K	12.58	14.41	14.6
FFT	2^{12} points	600	4096	102K	16K	28.29	28.69	1.4
Gaussian	160×160 matrix	2572	328.5	83K	10K	4.44	4.71	6.2
Simplex	469×220 matrix	6642	500.1	294K	36K	24.52	25.49	3.9
Navier	2^{10} points	549	225.7	62K	8K	62.14	62.35	0.4
EISPACK	64×64 matrix	7719	174.0	681K	86K	8.81	11.09	25.8

Table 2. Monitoring Statistics (on an 8 Node Hypercube).

Application Program	Total Messages Received	Out of Order Arrival	Percentage
Extraction	329	1	0.3
Test Gen.	3342	164	4.91
QR	2033	9	0.44
FFT	300	0	0
Gaussian	757	10	1.32
Simplex	3183	35	1.1
Navier	201	0	0
EISPACK	4101	82	2.0

Table 3. Out-of-Order Message Arrivals due to Monitoring (Node 0 of an 8 Node Hypercube).

Application Program	Message Distance Distribution (%)			
	1 Hop	2 Hop	3 Hop	4 Hop
Placement	92.6	7.4	0	0
Routing	94.8	5.2	0	0
Extraction	99.4	0.6	0	0
Test Gen.	36.6	40.5	18.8	4.1
Fault Sim.	91.0	3.7	4.2	1.1
QR	100	0	0	0
FFT	100	0	0	0
Gaussian	100	0	0	0
Simplex	100	0	0	0
Navier	66.7	33.3	0	0
EISPACK	100	0	0	0

Table 4. Average Message Distance Distribution (16 Node Hypercube).

Application Program	Destination LRU Stack Hit Rate (%)			
	DSS=1	DSS=2	DSS=3	DSS=4
Placement	90.99	91.90	92.05	92.23
Routing	89.74	94.86	94.86	94.86
Extraction	17.77	45.41	70.39	98.33
Test Gen.	89.62	89.89	90.15	90.44
Fault Sim.	1.06	1.59	3.70	73.02
QR	76.81	99.81	99.81	99.81
FFT	0	0	0	99.00
Gaussian	19.27	52.13	73.63	99.38
Simplex	19.17	55.80	76.96	99.83
Navier	0.00	20.56	49.16	54.76
EISPACK	25.20	52.05	72.86	99.83

Table 5. Message Destination Hit Rates as a Function of Destination LRU Stack Size (16 Node Hypercube).

Application Program	MRML Stack Hit Rate (%)				MRML Stack Hit Rate (%) Min. Msg. Size = 100 Bytes			
	LSS=1	LSS=2	LSS=3	LSS=4	LSS=1	LSS=2	LSS=3	LSS=4
Placement	87.07	91.21	93.19	94.20	92.07	94.38	95.68	96.29
Routing	93.69	95.29	95.56	95.85	99.73	99.73	99.73	99.73
Extraction	47.40	72.96	83.69	85.74	48.96	72.96	83.69	85.74
Test Gen.	77.67	88.14	92.65	96.74	84.95	91.97	95.03	98.17
Fault Sim.	88.89	92.06	92.59	92.59	88.89	92.06	92.59	92.59
QR	99.81	99.81	99.81	99.81	99.81	99.81	99.81	99.81
FFT	99.00	99.00	99.00	99.00	99.00	99.00	99.00	99.00
Gaussian	81.38	88.30	92.20	95.15	81.46	88.38	92.20	95.15
Simplex	44.88	85.00	91.89	91.90	85.12	85.16	91.98	91.99
Navier	76.67	98.44	98.44	98.44	76.67	98.44	98.44	98.44
EISPACK	88.86	92.39	94.05	94.65	95.79	95.79	95.79	96.13

Table 6. Message Length Hit Rates as a Function of MRML Stack Size (16 Node Hypercube).

Application Program	Destination and Length Stack Hit Rate (%) Min. Msg. Size = 100 Bytes							
	MRML Stack Size = 1				MRML Stack Size = 2			
	DSS=1	DSS=2	DSS=3	DSS=4	DSS=1	DSS=2	DSS=3	DSS=4
Placement	87.79	88.59	88.64	88.72	88.84	89.65	89.72	89.80
Routing	89.60	94.71	94.71	94.71	89.60	94.71	94.71	94.71
Extraction	5.52	16.88	30.19	48.83	13.68	31.75	50.83	72.80
Test Gen.	75.01	75.28	75.54	75.83	82.03	82.30	82.56	82.85
Fault Sim.	0.53	0.53	2.65	71.43	1.06	1.59	3.70	73.02
QR	76.81	99.81	99.81	99.81	76.81	99.81	99.81	99.81
FFT	0	0	0	99.00	0	0	0	99.00
Gaussian	13.11	40.65	59.04	81.46	16.08	45.61	64.98	88.38
Simplex	13.99	46.30	64.94	85.12	14.01	46.33	64.98	85.16
Navier	0.00	13.21	36.92	36.92	0.00	20.56	49.16	54.76
EISPACK	24.65	50.81	70.62	95.69	24.65	50.81	70.62	95.69

Table 7. Message Destination and Length Hit Rate (16 Node Hypercube).

Number of Stages	Model		
	Normal	Hypoexp.	Gamma
1	63.1	46.6	50.1
2	6.5	11.1	12.6
3	1.7	7.5	4.1

Table 8. SSE of Curve Fitting on Message Interval Distribution of Node 0 of the Cell Placement Program.

Number of Stages	Model		
	Normal	Hypoexp.	Gamma
1	5.3	12.1	6.0
2	1.3	7.5	2.7

Table 9. SSE of Curve Fitting on Message Interval Distribution of Node 0 of the QR Factorization Program.

Application Program	Distributions	Two-stage Normal pdf Model Parameters					
		a_1	μ_1	σ_1	a_2	μ_2	σ_2
Placement	Msg. Interval	0.49	0.57	0.07	0.51	1.45	1.01
	Computing Time	0.52	0.07	0.02	0.48	0.14	0.07
	Waiting Time	0.54	0.18	0.02	0.46	0.47	0.42
QR	Msg. Interval	0.84	5.77	0.60	0.16	6.23	0.16
	Computing Time	0.71	3.38	0.50	0.29	3.98	0.19
	Waiting Time	1.00	1.35	0.05	-	-	-

(μ in mSec)

Table 10. Two-stage Normal pdf Model Parameters for Placement and QR Programs(Node 0).

Application Program	System Time (%)			User Time (%)			Idle Time (%)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Placement	4.80	4.88	4.82	50.40	78.44	61.86	16.68	44.79	33.32
Routing	30.42	31.86	30.93	37.01	60.85	44.66	8.73	31.78	24.41
Extraction	0.31	0.46	0.39	51.61	55.86	53.84	43.68	48.07	45.77
Test Gen.	3.36	8.63	5.91	1.55	61.07	45.01	30.70	91.26	49.08
Fault Sim.	0.92	2.01	1.45	97.76	98.83	98.26	0.23	0.55	0.29
QR	13.30	14.72	13.95	54.40	59.95	57.11	25.33	32.31	28.94
FFT	1.97	2.02	2.00	96.49	96.59	96.54	1.39	1.51	1.46
Gaussian	5.59	8.70	6.71	51.96	69.72	59.42	23.27	41.73	33.88
Simplex	3.27	6.42	4.45	61.78	75.94	68.57	17.70	33.51	26.98
Navier	0.40	0.56	0.48	83.84	99.35	92.46	0.09	15.76	7.06
EISPACK	25.40	29.36	26.86	13.75	15.79	14.73	54.84	60.85	58.42

Table 11. CPU Utilization (on an 8 Node Hypercube).

Node	System Time			User Time			Idle Time		
	Measure (Sec)	Simulate (Sec)	Error (%)	Measure (Sec)	Simulate (Sec)	Error (%)	Measure (Sec)	Simulate (Sec)	Error (%)
0	3.53	3.59	1.7	14.35	14.35	0	6.42	6.45	0.5
1	3.56	3.67	3.1	14.56	14.56	0	6.11	6.16	0.8
2	3.42	3.52	2.9	14.15	14.15	0	6.79	6.73	-0.9
3	3.48	3.59	3.2	14.36	14.36	0	6.46	6.44	-0.3

Table 12. Comparison of Measurement and Simulation Results.
(QR factorization of 128×128 matrix)

Application Program	Incoming DMA Channel				Outgoing DMA Channel				Avg. In/Out DMA Chan. Overlap (%)	Avg. DMA Util. (%)	Average CPU & DMA Overlap (%)
	Utilization (%)			Avg. Service Time (μSec)	Utilization (%)			Avg. Service Time (μSec)			
	Min.	Max.	Avg.		Min.	Max.	Avg.				
Placement	1.24	1.27	1.25	135.7	1.30	1.34	1.32	143.3	0.15	2.42	0.304
Routing	8.39	8.43	8.41	105.3	8.54	8.63	8.60	107.7	2.11	14.90	1.607
Extraction	1.50	2.74	2.43	1344.6	1.37	4.50	2.48	1331.1	0.01	4.90	1.127
Test Gen.	0.30	3.18	0.72	115.7	0.55	2.61	0.87	172.2	0.04	1.55	0.101
Fault Sim.	0.20	0.26	0.21	114.1	0.19	0.32	0.21	117.5	0.001	0.41	0.208
QR	8.19	9.01	8.60	179.3	8.28	9.10	8.67	180.7	0.02	17.26	0.210
FFT	1.71	1.71	1.71	544.6	1.80	1.82	1.81	576.8	0.29	3.22	0.253
Gaussian	2.77	3.55	3.06	159.0	2.82	3.67	3.13	162.3	0	6.19	0.177
Simplex	2.12	3.01	2.48	215.6	2.20	3.09	2.52	219.0	0	5.00	0.093
Navier	0.07	0.12	0.10	114.2	0.08	0.13	0.10	120.8	0.004	0.19	0.052
EISPACK	10.39	11.13	10.67	111.5	10.81	12.16	11.45	119.8	0.44	21.67	0.798

Table 13. DMA Utilization (8 Node Hypercube).

Application Program	Node 0						Node 4						Overall Avg.	
	Link 0		Link 1		Link 2		Link 0		Link 1		Link 2			
	Util.	BW	Util.	BW	Util.	BW	Util.	BW	Util.	BW	Util.	BW	Util.	BW
Placement	0.38	6.79	0.23	4.71	1.02	15.09	1.02	15.11	0.22	4.33	0.39	7.22	0.55	9.05
Routing	0.98	11.62	0.32	3.80	7.84	94.68	8.18	98.40	0.32	3.83	0.65	7.71	3.06	36.64
Extraction	1.31	35.69	0.20	4.37	0.02	0.24	1.27	35.00	0.26	5.43	1.01	27.68	0.83	22.20
Test Gen.	0.55	7.17	1.27	16.31	2.24	28.19	0.60	7.03	1.22	13.45	0.36	4.08	0.48	5.60
Fault Sim.	0.12	1.99	0.10	1.64	0.13	2.24	0.13	2.08	0.14	2.61	0.10	1.73	0.12	2.05
QR	4.57	110.36	0	0	4.09	99.14	4.14	100.79	0	0	4.08	98.83	2.87	69.84
FFT	0.60	15.27	0.60	15.27	0.60	15.27	0.60	15.27	0.60	15.27	0.60	15.27	0.60	15.27
Gaussian	0.81	12.63	1.11	20.20	1.78	36.31	0.34	7.90	0.69	15.81	1.76	36.01	1.04	21.09
Simplex	0.67	11.17	0.85	15.53	1.48	32.65	0.27	7.04	0.53	13.64	1.38	30.07	0.83	18.29
Navier	0.04	0.85	0.05	0.69	0	0	0.04	0.85	0.05	0.69	0	0	0.04	0.64
EISPACK	1.87	29.52	3.49	54.45	6.27	99.65	1.73	27.08	3.58	49.61	6.49	105.00	3.79	59.60

(Utilization in percentage, bandwidth in KBytes/Sec.)

Table 14. Link Utilization (8 Node Hypercube).

Data Size (Cells)	Number of Nodes	CPU			DMA		Link	
		Sys. (%)	User (%)	Idle (%)	In (%)	Out (%)	Util. (%)	BW (KB/Sec)
286	4	2.25	68.99	28.76	0.85	0.90	0.54	9.72
286	8	4.10	62.38	33.52	1.25	1.32	0.55	9.05
286	16	5.10	46.02	48.88	1.50	1.59	0.47	7.18
183	8	10.39	48.31	41.29	2.96	3.18	1.28	17.10

Table 15. Placement Program on Different Number of Nodes and Data Size.