DAA/LANGLEY

NAG1-613

Report No. UIUCDCS-R-86-1303

UILU-ENG-86-1768

IN-60
64473 CR
P 19

Benchmarking Hypercube Hardware and Software

by

Dirk C. Grunwald and Daniel A. Reed

November - 1986

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

(NASA-CR-180620) BENCHMARKING HYPERCUBE HARDWARE AND SOFTWARE (Illinois Univ.) 19 p Avail: NTIS HC A02/MF A01 CSCL 09B

N87-26511

Unclas
G3/60 0064473

REPORT NO. UIUCDCS-R-86-1303

BENCHMARKING HYPERCUBE HARDWARE AND SOFTWARE

by
Dirk C. Grunwald
Daniel A. Reed

November 1986

DEPARTMENT OF COMPUTER SCIENCE
1304 W. SPRINGFIELD AVE.
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, IL 61801

This work was supported inpart by NSF Grant Number DCR 84-17948, NASA Contract Number NAG 1-613 and by the Jet Propulsion Laboratory.

Benchmarking Hypercube Hardware and Software

DIRK C. GRUNWALD[*] AND DANIEL A. REED[*]

The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.

Charles Babbage (1837)

Abstract. It has long been a truism in computer systems design that *balanced* systems achieve the best performance. Message passing parallel processors are no different. To quantify the balance of a hypercube design, we have developed an experimental methodology and applied the associated suite of benchmarks to several existing hypercubes. The benchmark suite includes tests of both processor speed in the absence of internode communication and message transmission speed as a function of communication patterns.

Introduction

The appearance of a new computer system always raises many questions about its performance, both in absolute terms and in comparison to other machines of its class. In addition, repeated studies have shown that a system's performance is maximized when the components are balanced (i.e., there is no single system bottleneck) [DeBu78]. Message passing parallel processors are no different; optimizing performance requires a judicious combination of node computation speed, message transmission latency, and operating system software. For example, high speed processors connected by high latency communication links restrict the classes of algorithms that can be efficiently supported. Although the interaction of communication and computation can be examined analytically [ReSc83], time varying behavior and the idiosyncrasies of system software can only be captured by observation and measurement. Consequently, we began a benchmark study of hypercubes, with three primary goals.

Because the performance of any system does depend on a combination of hardware and software, our first and primary goal was determining the performance of both the underlying hardware and the fraction of that performance lost due to poor compilers and operating system overhead. Second, we wished to characterize the balance of processing power and communication speed. With these parameters, algorithms can be developed that are best suited to the machine [SaNN86]. Finally, we are developing a high-performance, portable operating system for hypercubes, called Picasso, that provides dynamic task migration to balance workloads and adaptive routing of data to avoid congested portions of the network. To meet these goals, it must be possible to rapidly transmit small status messages. Thus, we sought performance data to tune Picasso's algorithms for each hypercube.

Overview

Because any hypercube computation combines both communication and computation [Heat86, ReFu86], a single number (e.g., MIPS, MFLOPS, or bits/sec) will not accurately reflect

[*]Department of Computer Science, University of Illinois, Urbana, Illinois 61801. This work was supported in part by NSF Grant Number DCR 84-17948, NASA Contract Number NAG-1-613 and by the Jet Propulsion Laboratory.

the interplay between communication and computation. Thus, computation speeds and message transmission rates both have hardware and software components (i.e., a fast processor can be coupled with a poor compiler, or slow communication hardware can be coupled with efficient communication software). To isolate the effects of hardware and software, and to explore their interaction, we have developed a hypercube benchmark set and associated methodology. This benchmark set includes four components:

- simple processor benchmarks,
- synthetic processor benchmarks,
- simple communication benchmarks, and
- synthetic communication benchmarks.

The simple processor benchmarks are, as the name implies, simple enough to highlight the interaction between processor and compiler, including the quality of generated code. In turn, the synthetic processor benchmarks reflect the typical behavior of computations and provide a ready comparison with similar benchmarks on sequential machines.

Communication performance is closely tied to system software. Some hypercubes support only synchronous communication between directly connected nodes; others provide asynchronous transmission with routing. In both cases we use the simple communication benchmarks to measure both the message latency as a function of message size and the number of links on which each node can simultaneously send or receive. For systems that support routing and asynchronous transmission, we have developed synthetic communication benchmarks that reflect communication patterns in both time *and* space.

The remainder of the paper examines the processor and communication performance of two commercial hypercubes, the Intel iPSC [Ratt85] and Ametek S/14, and the Mark-III prototype developed by the NASA Jet Propulsion Laboratory (JPL). We begin with a brief comparison of these machines.

Hypercube Comparisons

Three of the hypercubes tested are based on the 16-bit Intel 80286 microprocessor. The fourth uses the 32-bit Motorola 68020 microprocessor. Table 1 shows the salient features of each system.

JPL Mark-III

This machine is the latest in a series of hypercubes developed by the California Institute of Technology and JPL. During our tests, a four-node prototype of the Mark-III, running the CrOS-III operating system, was used. CrOS-III provides synchronous, single-hop communication primitives. The communication primitives used during testing included: `cwrite` to send a message to adjacent nodes, `cread` to read a message from an adjacent node, `exchange` to implement an indivisible read and write operation pair, and `broadcast` to send a message to all other nodes. All test programs were compiled using the Motorola 68020 C compiler provided on the CounterPoint host processor.

Intel iPSC

For all tests, we used a 32-node iPSC system running NX Beta Version 3.0, the latest version of the operating system. This iPSC operating system is derived from the Cosmic Environment [Seit85] and is functionally equivalent to that system. The communication primitives used during testing were: `send` to asynchronously send a message to any node while program execution continues, `sendw` to send a message to any node and wait for message transmission to

Table 1 Hypercube Hardware Characteristics

	Mark–III	Intel iPSC	Ametek S/14	Ametek S/14 β
Processor	16 MHz 68020	8 MHz 80286	8 MHz 80286	6 MHz 80286
Floating Point	16 MHz 68881	6 MHz 80287	8 MHz 20287	6 MHz 80287
I/O Processor	16 MHz 68020	none	10 MHz 80186	8 MHz 80186
Minimum Memory	4 Mbytes	0.5 Mbytes	1 Mbytes	1 Mbytes
Maximum Memory	4 Mbytes	4.5 Mbytes	1 Mbytes	1 Mbytes
Channels per Node	8	7	8	8
Peak Bandwidth	13.5 Mbits/s	10 Mbits/s	3 Mbits/s	3 Mbits/s

complete, `recv` to asynchronously receive a message while program execution continues, `recvw` to receive a message and wait for it to arrive. Operations equivalent to `exchange` can be composed from `send` and `recv` operations. The broadcast operation is performed by specifying a special destination address. Programs were compiled using the L–model of the Microsoft C compiler provided with Xenix, the operating system on the hypercube host.

Ametek S/14

A 32–node S/14 system running XOS Version D [Amet86] was used. XOS has capabilities similar to those of CrOS–III. The message primitives used during testing were: `wtELT` to send a message to an adjacent node, `rdELT` to receive a message from an adjacent node, `exchELT` to indivisibly exchange messages between two nodes, and `brELT` to broadcast a message to all other nodes in the system. All programs were compiled using the L–model of the Lattice C V3.1 compiler.

Ametek S/14 β

A beta–test version of the S/14 system was used. This machine is functionally equivalent to the S/14, although the processor clocks are slower, and XOS Version 1.1 was used. Programs were compiled using the D–model of Lattice C V2.14 compiler.

In contrast to the Intel iPSC operating system, both CrOS–III and XOS are derived from the Crystalline Operating System developed by the High Energy Physics group [FoOt84] at the California Institute of Technology, and do not provide any message routing or multi–hop capabilities.

Processor Benchmarks

The simple processor benchmarks were based on programs used to evaluate other microprocessors [PaSe82]. The synthetic benchmarks, Dhrystone [Weic84] and Whetstone [CuWi76], were designed to represent typical systems programs and test floating point speeds, respectively. All the benchmarks were written in the C programming language, the only language available on all the machines studied. Unfortunately, this precluded using such standard numerical benchmarks as Linpack [DoBM79].[1] Table 2 describes the tests, their characteristics, and the features they are each designed to test. Table 3, in turn, shows the results of the benchmarks.

For both variations of the *Dhrystone* benchmark, the results are given in *Dhrystones*, a normalized measure of Dhrystone performance. Likewise, the *Whetstone* benchmark results are

[1]With the availability of Fortran compilers for most hypercubes, we are now augmenting the benchmark set.

Table 2 Hypercube Processor Benchmarks

Simple Benchmarks	
Loops	10 repetitions of a 1,000,000 iteration null loop. Tests loop overhead
Sieve	100 repetitions of finding the primes from 1 to 8190. Tests loops, integer comparison, assignment.
Fibonacci	20 repetitions of finding Fibonacci(24). Tests recursion, integer addition, parameter return.
Hanoi	Solve the Towers of Hanoi problem with 18 disks. Tests recursion, integer comparison.
Sort	14 repetitions of quicksorting a 1000-element array of random elements. Tests recursion, comparisons, array references, multiplication and modulus.
Puzzle (subscript)	10 repetitions of Baskett's Puzzle program with subscripts. Tests explicit array subscript calculations and procedure calls.
Puzzle (pointer)	10 repetitions of Baskett's Puzzle program with pointers. Tests use of pointer vs. explicit array subscripts.
Synthetic Benchmarks	
Whetstone	The Whetstone synthetic benchmark. General test of floating point performance, including trigonometric functions, multiplications and divisions.
Dhrystone (no registers)	The Dhrystone synthetic benchmark without register optimization. Tests general integer scalar performance with 'typical' instruction mix.
Dhrystone (registers)	The Dhrystone synthetic benchmark with register optimization. Tests availability and effects of register variables

given in *Whetstones*, the normalized measure of Whetstone performance. All other results are in seconds. In all cases, the 95 percent confidence interval was less than 5 percent of the stated mean.

For those benchmarks involving many procedure calls, in particular the *Fibonacci*, *Sieve* and *Hanoi* tests, the Ametek S/14 provides the best performance of the Intel 80286-based hypercubes. However, all other benchmarks, even the highly recursive *Sort* test, favor the Intel iPSC. Most notably, the Intel iPSC is 4.4 times faster on the Dhrystone synthetic benchmark.

Because the processor architectures for the Intel iPSC and the Ametek S/14 are identical, these dramatic differences must be due to software. To investigate these differences, the *Loops* program was compiled using both the Microsoft and Lattice C compilers and then disassembled. The Intel 80286 provides a 16-bit ALU, while the innermost loop bound in *Loops* requires a 32-bit integer. The Lattice C V2.14 compiler used with the Ametek S/14 β machine invokes a subroutine for 32-bit comparison on each loop iteration. In contrast, the Lattice C V3.1 compiler for the Ametek S/14 expanded the procedure call in-line, providing better performance. Finally, the Microsoft C compiler for the Intel iPSC transformed the innermost loop into two loops, each of which tested a 16-bit integer. The outer loop tested the high-order 16-bits of the loop

Table 3 Hypercube Processor Comparison

	Mark–III	Intel iPSC	Ametek S/14	Ametek S/14 β
Loops	8.8	65.6	164.3	263.8
Fibonacci	10.0	39.2	21.3	32.7
Sieve	6.4	21.5	19.3	35.0
Sort	11.8	42.3	83.0	⊥
Hanoi	4.4	12.5	6.9	⊥
Puzzle (subscript)	45.5	87.6	97.9	164.3
Puzzle (pointer)	20.2	112.5	491.7	792.4
Whetstone‡	684,463	102,637	185,874	7367
Dhrystone (registers)‡	3472	724	165	107
Dhrystone (no registers)‡	3322	717	167	108

⊥These tests could not be run due to stack size limitations.
‡Performance figures in Whetstones or Dhrystones.

counter while the inner loop tested the low-order 16–bits.

To further corroborate our view that compiler technology was dominant performance determinant, the *Dhrystone* benchmark was compiled and disassembled using both the C compiler provided with the Ametek S/14 β machine and an Intel 80286 compiler developed by AT&T. The resulting lines of disassembled instructions, shown in Table 4, support the theory that quality of generated code is the major contributing factor to the performance differences observed.

Additional conclusions can be drawn from Table 3. The *Puzzle* benchmark exists in two variations. The first variation uses arrays; the second was "optimized" by using explicit pointers and pointer arithmetic rather than implicit array subscript calculations. On machines with 32–bit integers and pointers, the pointer version typically executes faster because the compiler can more easily optimize the code. On the JPL Mark-III this is the case. With the Intel 80286-based systems, pointer arithmetic is more expensive that array subscript calculations because the former requires 32-bit operations. In contrast, array indexing normally requires only 16–bit quantities.

Similarly, the two variations of the *Dhrystone* benchmark show that, for the 80286-based systems, "optimizing" the program by using register variables does not improve performance.

Table 4
Intel 80286 Compiler Comparison (*Dhrystone* Benchmark)

Compiler	*Lines of Assembly Code*
Lattice C V2.14	751
AT&T SysV C (unoptimized)	467
AT&T SysV C (optimized)	360

Precisely the opposite statements can be made for the JPL Mark-III.

The floating point performance of the Ametek S/14 is greater than that of the Intel iPSC, primarily due to the faster floating point unit. The Lattice C compiler for the Ametek S/14 β machine uses procedures to implement floating point operations, hence the dramatic performance reduction.

Finally, Table 3 shows that the JPL Mark-III is clearly superior to all machines based on the Intel 80286 microprocessor.[2] Based on the *Dhrystone* tests, a Mark-III node is approximately five times faster than a node based on the Intel 80286. Because of the large performance differential between the Mark-III and the Intel 80286-based hypercubes, it is difficult to compare them. In addition, while identical benchmark programs were used on all machines, the benchmarks are *not* identical. The Mark-III, based on the Motorola 68020, always manipulates 32-bit quantities, while the Intel 80286 operates on both 16-bit and 32-bit items. Thus, those benchmarks that can be executed using only 16-bit quantities are unfairly biased toward the Intel 80286-based hypercubes. Similarly, some of the C compilers used during the tests convert 32-bit floating point numbers to 64-bit double floats before performing floating operations.

Simple Communication Benchmarks

To provide a small set of benchmarks that test all link-level characteristics, we have derived five operations that reflect a broad range of common single-link communications. In all the simple communication benchmarks, the programs were structured to provide maximal communication concurrency, using the appropriate idiom in each hypercube operating system. In particular, the Intel iPSC versions use the asynchronous `send` and `recv` operations where possible, and the CrOS-III and XOS implementations use such special operations as `exchange` or `exchELT`. As with the processor benchmarks, all results are the means of 95 percent confidence intervals, and all intervals are less than 5 percent of the corresponding mean. Figure 1 illustrates the communication patterns tested by each of the simple communication benchmarks.

Simple Transfer

The first test measures transmission speeds across a single link between two processors. Figure 2 compares the four different hypercubes. As with the simple processor benchmarks, the JPL Mark-III is significantly faster than any of the systems based on the Intel 80286. In addition, the Intel iPSC transmission time for small messages is much smaller than that for earlier versions of the iPSC operating system [SaNN86].

If the messages transmitted between nodes are not broken into packets, message transmission time t can be modeled as

$$t = t_l + Nt_c, \qquad (1)$$

where t_l is the communication latency (i.e., the startup time for a transmission), t_c is the transmission time per byte, and N is the number of bytes in the message. Statistically, the linear model in (1) is a good fit to experimental data obtained for single link transmissions on all hypercubes we tested. Table 5 shows the result of a least-squares fit of the data to this linear model. Although the measured bandwidth for the JPL Mark-III is higher than the rated peak bandwidth (see Table 1), this anomaly is likely due to the use of a prototype system. The Ametek S/14 has a lower latency than the Intel iPSC, but the higher bandwidth of the Intel iPSC

[2]Informal benchmarks suggest that a single Mark-III node is comparable to a Sun-3/52 workstation.

Table 5 Hypercube Communication Comparison

	Mark–III	Intel iPSC	Ametek S/14	Ametek S/14 β
Latency (seconds)	9.5×10^{-5}	1.7×10^{-3}	5.5×10^{-4}	1.3×10^{-3}
Transmission time (sec/byte)	5.63×10^{-7}	2.83×10^{-6}	9.53×10^{-6}	1.17×10^{-5}
Bandwidth (Mbits/s)	14.3	2.8	0.84	0.69

yielded smaller total communication time for messages of more than 150 bytes.

We emphasize that extrapolating general communication performance from single link tests is fallacious. Such simple tests highlight only link-level characteristics. More general benchmarks are needed to investigate routing and buffering behavior; this is the motivation for the synthetic communication benchmarks presented in the next section.

Send-and-Reply and *Exchange*

In the second test, a message was sent, and the sender awaited a reply, analogous to a remote procedure call. This was designed to test the minimum response time of a node as a function of message size.

The third test was an *exchange* of data values, similar to that in many numerical computations, notably iterative solution of partial differential equations. If full-duplex transmission links were available, this test would measure their effective use.

Comparing the first test, Figure 2, with the second test, Figure 3, shows that, for both the JPL Mark-III and Ametek S/14, the time to complete the send-and-reply is roughly twice that for the simple send operation. However, for the Intel iPSC, the time for the send-with-reply is less than twice the cost of a simple send. Because all four hypercubes provide only simplex communication channels, this differential cannot be attributed to communication overlap.

In the first test, the Intel iPSC benchmark executes a `(sendw)/(recvw)` communication sequence (i.e., one processor executes a `sendw` and the other executes a `recvw` primitive). In the send-and-reply test, the call sequence is `(send;recvw)/(recvw;sendw)`. Here, the processor executing the send has completed most of the overhead associated with a `recvw` call by the time the second processor has initiated the `sendw` operation, allowing the operating system to write the message directly into the user's buffer rather than a system buffer.

This is more evident in the exchange operation in the third test; see Figure 4. Here, the Intel iPSC executes a `(send;recvw)/(recv;sendw)` sequence. The overlap afforded by the asynchronous `send` and `recv` calls allows the processor to establish where a message is to be placed before it arrives, eliminating buffer copies. Also, the processor is able to execute the setup code for the next message transmission while the communication is in progress. When the data for the third test on the Intel iPSC is fitted to (1), and the results are divided by two to compare them to a simple transfer, the latency drops to 1.4 milliseconds, a 20 percent improvement. In contrast, the JPL Mark-III provides no equivalent improvement because the CrOS-III operating system permits no asynchrony and executes little code during communication operations.

N-way Broadcast

The fourth test is an N–way broadcast, used when every processor must disseminate a message to all other processors. Because multiple communication links connected to each node can potentially be simultaneously active, this benchmark tests link simultaneity.

The intuitive implementation of this benchmark would use N broadcast operations. However, a more efficient approach, for existing hypercubes, uses a ***ring broadcast*** where node i receives messages from node $i-1$ and sends messages to node $i+1$. This requires only $O(N)$ time.

With simplex communication links, the time for a ring broadcast should be approximately six times that for a transmission across a single link. As Figure 5 shows, both the JPL Mark–III and Ametek S/14 exhibit this behavior. In contrast, the performance of the Intel iPSC is non–intuitive. We have not yet determined the reasons for the behavior of the Intel iPSC; one tenable explanation is that the increased interrupt processing required to service two channels causes a significant amount of time to be spent context switching. Because an Intel iPSC node does not have an I/O processor, the CPU must context switch to prepare an in–coming message, switch back to the user program, and switch twice more when the message has completely arrived. For small messages, context switching overhead is a large proportion of the total transmission time. As the message size increases, this cost is amortized over each byte of the message, and the Intel iPSC broadcast time is smaller than that for the Ametek S/14 when the message length becomes large.

Single Node Broadcast

The fifth and final test, shown in Figure 6, exercises the broadcast mechanism provided by all the operating systems. Since only a four node version of the Mark–III was available at the time of this study, the test is restricted to a four–node cube in all cases. As can be seen, the Intel iPSC performance is significantly better, relative to the other hypercubes, than for the N–way broadcast.

Synthetic Communication Benchmarks

Application programs can be used to study hypercube communication performance, but because each program occupies only a small portion of the space of potential communication behaviors, a large number of application programs are needed to achieve adequate coverage. Moreover, it is difficult to separate computation from communication. To study hypercube performance under a variety of communication traffic patterns, we have developed a model of communication behavior. Each hypercube node executes a copy of the model, generating network traffic that reflects some pattern of both ***temporal*** and ***spatial*** locality. Intuitively, temporal locality defines the pattern of internode communication in time. An application with high temporal locality would exhibit communication affinity among a subset of network nodes. However, these nodes need not be near one another in the network. Physical proximity is determined by spatial locality.

Temporal Locality

Our implementation of ***temporal*** locality is based on a Least Recently Used Stack Model (LRUSM), originally used to study management schemes for paged virtual memory [Denn80]. When used in the memory management context, the ***stack*** after memory reference $r(t)$ (i.e., the reference at time t) contains the n most recently accessed pages, ordered by decreasing recency of reference. The stack distance $d(t)$ associated with reference $r(t)$ is the position of $r(t)$ in the stack defined just after memory reference $r(t-1)$ occurs. These distances are assumed to be independent random variables such that $Probability[d(t) = i] = b_i$ for all t. For example, if $b_1 = 0.5$, there is a 50 percent chance that the next page referenced will be the same as the one just referenced.

In our adaptation of LRUSM to network traffic, each node has its own stack containing the n nodes that were most recently sent messages. In other words, destination nodes in the network are analogous to pages in address space of a process for the LRUSM model of memory reference patterns. Parameters to the model include the stack reference probabilities b_i and the stack size n.

We emphasize that the sum of the probabilities, $\sum_{i=0}^{n} b_i$ is normally less than 1. Consequently, it is possible for the condition $d(t) > n$ to occur. In this case, a stack *miss* occurs. This corresponds to a page fault in the virtual memory context. In our model, a stack miss means that a new node not currently in the stack is chosen as the destination of the next transmission. This selection is based on a *spatial* locality model.

Spatial Locality

Currently, our model provides three types of spatial locality:

- *Uniform.* Any network node can be chosen as the destination with equal probability. The uniform routing distribution is appealing because it makes *no* assumptions about the type of computation generating the messages; this is also its largest liability. However, because most computations should exhibit some measure of communication locality, it provides what is likely to be an upper bound on the mean internode message distance.
- *Sphere of Locality.* Each node is considered to be the center of a sphere of radius L, measured in hops. A node sends messages to the other nodes inside its sphere of locality with some (usually high) probability ϕ, and to nodes outside the sphere with probability $1 - \phi$. This model reflects the communication locality typical of many programs (e.g., the nearest neighbor communication typical of iterative partial differential equations solvers coupled with global communication for convergence checking).
- *Decreasing Probability.* The probability of sending a message to a node decreases as the distance from the source to the node increases. Specifically, the probability of sending a message l hops is

$$\Phi(l) = Decay(d,\ lmax) \cdot d^l, \qquad 0 < d < 1$$

  where $lmax$ is the diameter of the network, d is a selected decay factor, and $Decay(d,\ lmax)$ is a normalizing constant chosen such that

$$Decay(d,\ lmax) \cdot \sum_{i=1}^{lmax} d^l = 1.$$

  This model reflects the diffusion of work from areas of high utilization to areas of lower utilization.

More detailed descriptions of sphere of locality and decreasing probability routing can be found in [ReFu86]. By pairing parameters for temporal locality with parameters for spatial locality, a wide variety of traffic distributions can easily be generated. By flushing the temporal stack periodically, a new set of destinations is selected; this can be used to simulate multiprogramming at individual nodes. Finally, additional input parameters control the frequency of message generation and the distribution of message lengths.

Experimental Results

Figure 7 shows the mean time for a 16 node Intel iPSC to transmit 3000 messages whose length was drawn from a negative exponential distribution with a mean of 512 bytes. The horizontal line denotes the uniform message routing distribution. This provides a point of reference for the decreasing probability and sphere of locality routing distributions.

In Figure 7, $\phi = 1$ for the sphere of locality distribution. This means that all messages generated will be sent to destinations within the radius. Thus, the expected execution time approaches that of the uniform routing distribution as the radius approaches the network diameter. With the decreasing probability distribution, increasing d means that a larger fraction of all messages are sent to distant nodes. For large enough values of d, the decreasing probability distribution is *anti-local.* Specifically, the mean internode distance is larger than that of the uniform routing distribution.

Figure 8 shows the effect of varying the temporal locality of the three spatial locality distributions, using the same number and type of messages as in Figure 7. In the figure, the temporal stack has depth one. This means that the number of messages sent to the node in the stack is the mean of a binomial distribution with parameter p, the probability of referencing the stack. Intuitively, there are "runs" of consecutive messages sent to a single destination node. Figure 9 illustrates these runs for one of the spatial distributions.

The most striking feature of Figure 8 is the small variation in time to complete the suite of message transmissions. One would expect the temporal locality and its associated message runs to induce transient queues of outstanding messages on one link of each hypercube node. This phenomenon should manifest itself as increased delays as the probability of a stack reference increases. However, the variations shown in Figure 8 are not statistically significant. Why? Suppose the rate each node generated messages were perfectly balanced with the transmission capacity of each of the node's communications links. In this case, no message queues would form. Now suppose temporal locality were introduced. The presence of stack runs would induce an imbalance, a queue would form for at least one link, and messages would be delayed. This phenomenon does *not* occur in Figure 8, due to an imbalance of computation and communication speeds on the Intel iPSC. Nodes can generate messages faster than they can be transmitted by the communication links. Thus, queues develop for *all* links, independent of the presence of temporal locality, and these queues mask its effects.

Observations

Computation speeds, communication capacity, and communication patterns, both in time and space, interact in subtle ways. By using a synthetic benchmark that provides a broad spectrum of communication patterns, one can systematically and formally explore these interactions. This knowledge can be used to guide system design.

Summary

We have defined a collection of benchmarks designed to test both the computation and communication components of hypercubes. These benchmarks include simple processor tests that permit comparison of compilers and synthetic processor benchmarks for comparison with sequential machines. In addition, we defined a series of simple communication benchmarks and used them to determine the transmission latency and communication bandwidth of isolated links.

Finally, we outlined a synthetic communication benchmark for testing the communication network when presented with asynchronous, multihop messages.

Current Work

We are currently expanding the processor benchmark set to include the linear algebra benchmarks included in Linpack [DoBM79]. We believe this will provide a more realistic measure of processor floating point performance.

In addition, we are augmenting the benchmark set with an elliptic partial differential equation solver, based on successive over–relaxation. Because the (square) domain of the test problems can be divided into either a small number of large partitions or a large number of small partitions, the ratio of computation to communication at each hypercube node, and the corresponding frequency of internode communication, can be varied widely. This permits determination of the optimal balance of computation to nearest neighbor communication for different hypercubes. Finally, this partial differential equation solver implements three different convergence checking schemes: synchronous, where a global checker receives convergence information from all nodes each N iterations; asynchronous, where convergence checking is overlapped with computation; and statistical, where the rate of change in the error between iterations is used to predict the next iteration for convergence checking. These convergence checking schemes will allow us to test communication efficiency when messages must be routed through intermediate nodes. Finally, we will be able to test the effective overlap of computation and communication on those machines containing communication co–processors.

Acknowledgments

The Concurrent Computation Project at the Jet Propulsion Laboratory provided access to the JPL Mark–III. In addition, Jack Dongarra and Argonne National Laboratory provided access to the Intel iPSC. Their support and help made the benchmarking study possible.

We think there remains much detail to be worked out, and possibly some further invention needed, before the design can be brought into a state in which it would be possible to judge whether it would really so work.

Committee analyzing Babbage's Analytical Engine

References

[Amet86] Ametek Computer Research Division, "Ametek System 14 User's Guide: C Edition," Version 2.0, May 1986.

[CuWi76] H. J. Curnow and B. A. Wichman, "A Synthetic Benchmark," *Computer Journal,* Vol. 19, No. 1, February 1976.

[Denn80] P. J. Denning, "Working Sets Past and Present," *IEEE Transaction on Software Engineering,* January 1980.

[DeBu78] P. J. Denning and J. P. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys,* Vol. 10, No. 3, pp. 225-262, September 1978.

[DoBM79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, "Linpack Users' Guide," *SIAM Publications,* 1979.

[FoOt84] G. C. Fox and S. W. Otto, "Algorithms for Concurrent Processors," *Physics Today,* Vol. 37, pp. 50-59, May 1984.

[Heat86] M. T. Heath, *Hypercube Multiprocessors 1986,* Society for Industrial and Applied Mathematics, Philadelphia, 1986.

[PaSe82] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer,* Vol. 15, No. 9, pp. 8-22, September 1982.

[Ratt85] J. Rattner, "Concurrent Processing: A New Direction in Scientific Computing," *Conference Proceedings of the 1985 National Computer Conference,* AFIPS Press, Vol. 54, pp. 157-166, 1985.

[ReSc83] D. A. Reed and H. D. Schwetman, "Cost-Performance Bounds on Multimicrocomputer Networks," *IEEE Transactions on Computers,* Vol. C-32, No. 1, pp. 85-93, January 1983.

[ReFu86] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message Based Parallel Processing,* monograph submitted to *MIT Press.*

[SaNN86] J. H. Saltz, V. K. Naik, and D. M. Nicol, "Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures," *ICASE Report 86-4,* NASA Langley Research Center, to appear in the *SIAM Journal of Scientific and Statistical Computing.*

[Seit86] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM,* Vol. 28, No. 1, pp. 22-33, January 1985.

[Weic84] R. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM,* Vol. 27, No. 10, pp. 1013-1030, October 1984.

Figure 1 Simple Communication Benchmarks

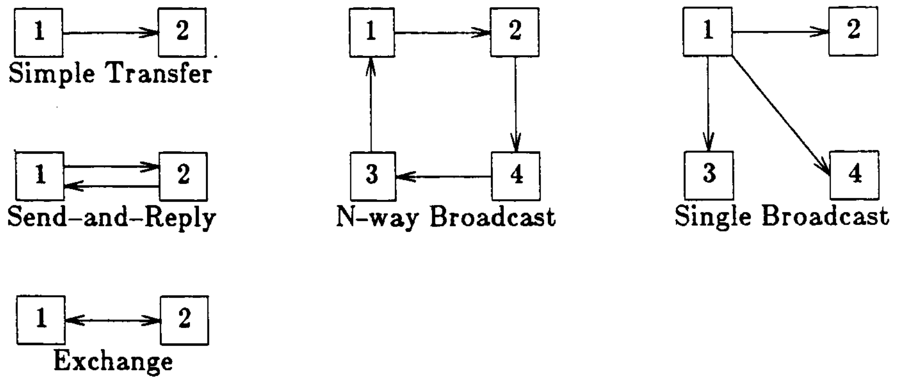


Figure 2 Simple Transfer

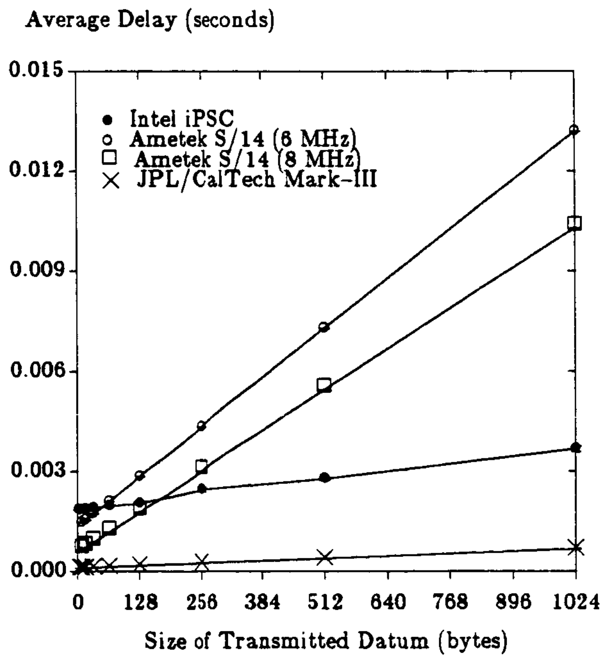


Figure 3 Send-and-Reply

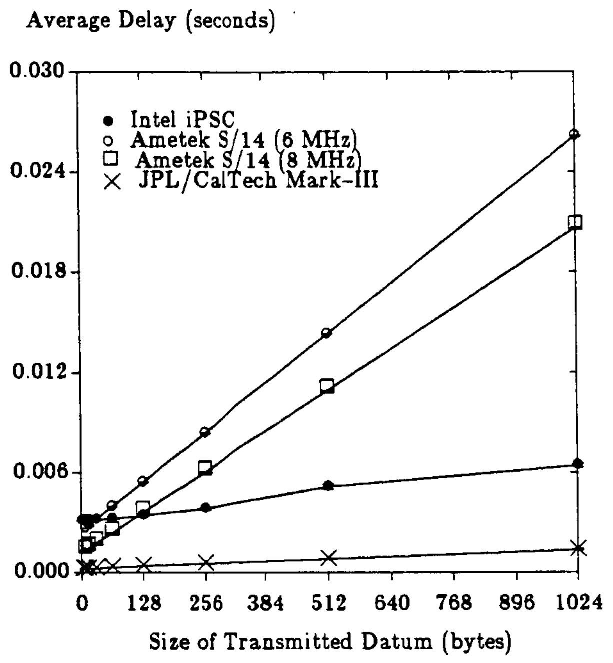


Figure 4 Exchange

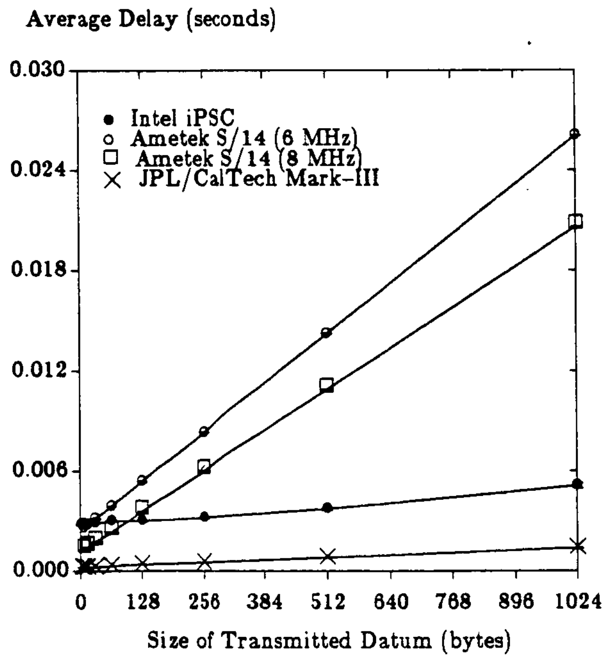


Figure 5 N-way Broadcast

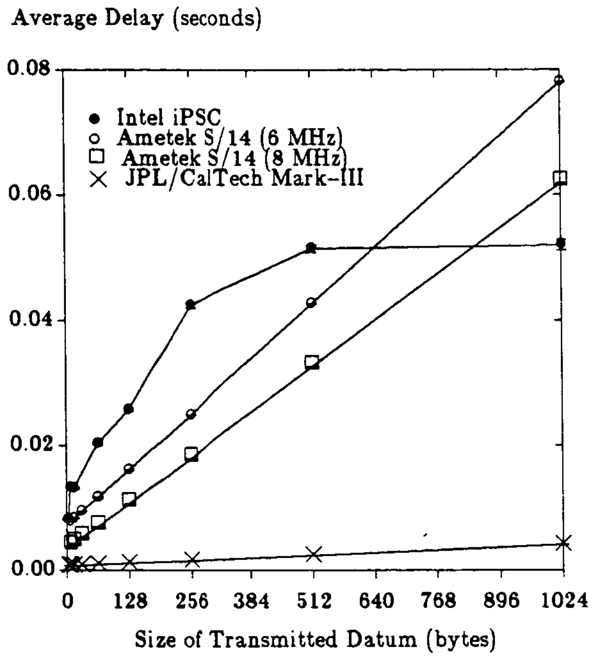


Figure 6 Single Broadcast

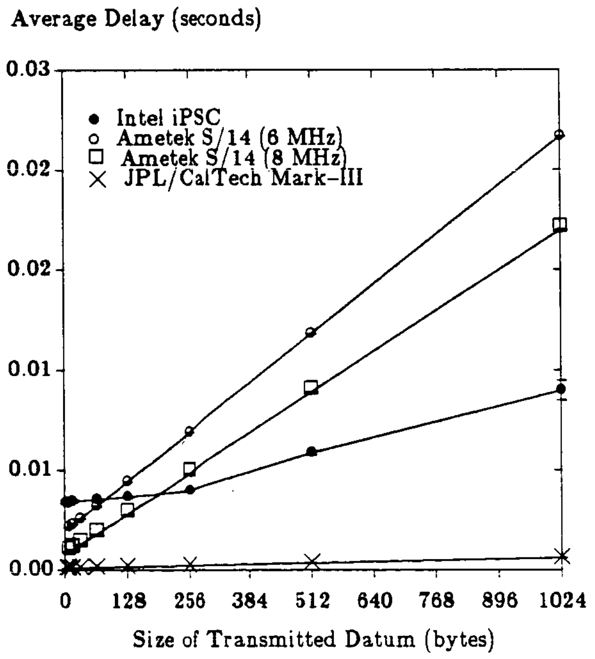


Figure 7 Spatial Communication Locality

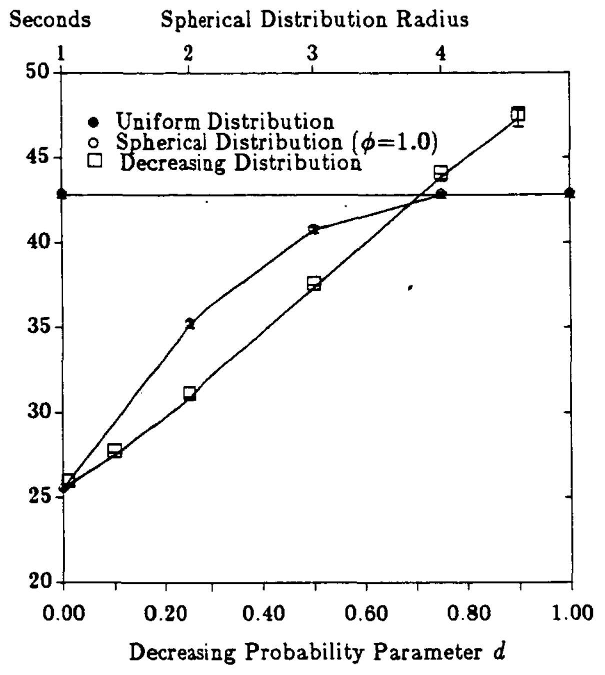


Figure 8 Temporal Communication Locality

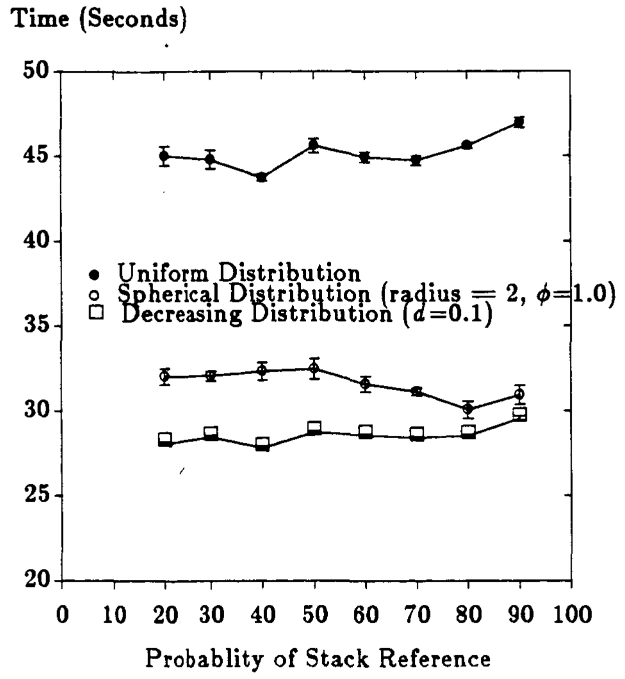
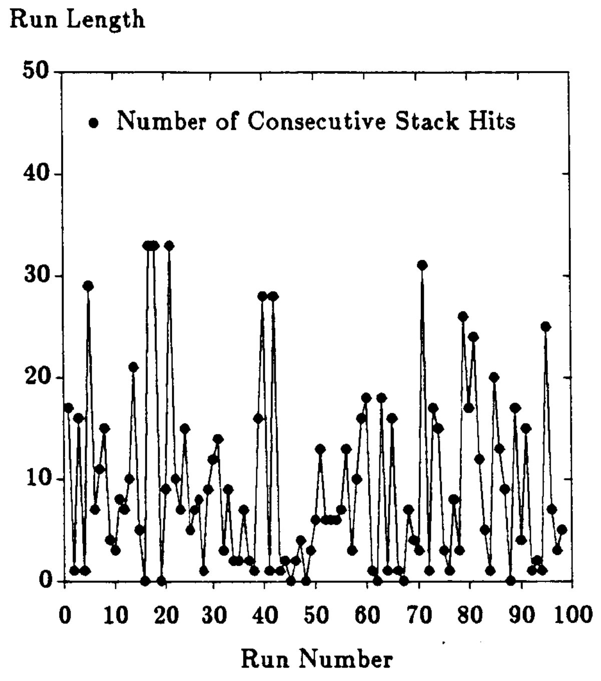


Figure 9 Temporal Locality Run Lengths
Decreasing Probability Spatial Locality ($d = 0.10$)
Single Stack Entry with Stack Hit Probability 0.90



BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-86-1303	2.	3. Recipient's Accession No.
4. Title and Subtitle Benchmarking Hypercube Hardware and Software			5. Report Date November 1986
			6.
7. Author(s) Dirk C. Grunwald and Daniel A. Reed			8. Performing Organization Rept. No. R-86-1303
9. Performing Organization Name and Address University of Illinois Department of Computer Science 1304 W. Springfield, 240 Digital Computer Lab Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NSF DCR84-17948 NASA NAG 1-613
12. Sponsoring Organization Name and Address a.National Science Foundation 1800 G Street, NW Washington, D. C. 20550	b.NASA Langley Research Center Hampton, Virginia 23665 c.Jet Propulsion Laboratory Pasadena, California		13. Type of Report & Period Covered
			14.
15. Supplementary Notes			

16. Abstracts

It has long been a truism in computer systems design that balanced systems achieve the best performance. Message passing parallel processors are no different. To quantify the balance of a hypercube design, we have developed an experimental methodology and applied the associated suite of benchmarks to several existing hypercubes. The benchmark suite includes tests of both processor speed in the absence of internode communication and message tansmission speed as a function of communication patterns.

17. Key Words and Document Analysis. 17a. Descriptors

Hypercubes
Benchmarking
Communication models
Software comparison

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 20
	20. Security Class (This Page UNCLASSIFIED	22. Price