NPAC Technical Report SCCS-662
To appear in: *Concurrency:Practice and Experience*

Benchmarking the Computation and Communication Performance of the CM-5 [1]

Kivanc Dincer Zeki Bozkus Sanjay Ranka Geoffrey Fox

Northeast Parallel Architectures Center
111 College Place, Room 3-217
Syracuse University
Syracuse, NY 13244-4100
{dincer, zbozkus, ranka, gcf}@npac.syr.edu

First Draft:September 1992
Revised: January 10, 1995

Abstract

Thinking Machines' CM-5 machine is a distributed-memory, message-passing computer. In this paper we devise a performance benchmark for the base and vector units and the data communication networks of the CM-5 machine. We model the communication characteristics such as communication latency and bandwidths of point-to-point and global communication primitives. We show, on a simple Gaussian elimination code, that an accurate static performance estimation of parallel algorithms is possible by using those basic machine properties connected with computation, vectorization, communication, and synchronization. Furthermore, we describe the embedding of meshes or hypercubes on the CM-5 fat-tree topology and illustrate the performance results of their basic communication primitives.

[1]This work was supported in part by NSF under CCR-9110812 and by DARPA under contract # DABT63-91-C-0028. This work was also supported in part by a grant of HPC time from the DoD HPC Shared Resource Center, Army High-Performance Computer Center at University of Minnesota CM-5 machine. The contents do not necessarily reflect the position or the policy of the U.S.Government, and no official endorsement should be inferred.

1 Introduction

The CM-5 is a parallel distributed-memory machine that can scale up to 16,384 processing nodes. Each node contains a SPARC microprocessor, a custom network interface, a local memory up to 128 MBytes, and either a memory controller or vector controller units. The processing nodes are connected by three networks: the *diagnostics network* which identifies and isolates errors throughout the system; the high speed *data network*, which communicates bulk data; and the *control network*, which is mainly responsible for the operations requiring the participation of all nodes simultaneously, such as broadcasting and synchronization. As data communication between two nodes can be performed by using either the data network or the control network, we restrict our analysis to these two.

In making this study we have two objectives. The first includes evaluating the computation and communication performance of the CM-5 and modeling the system parameters such as computational processing rate, communication start-up time, and the latency and data transfer bandwidth. The fundamental measurement made in our benchmark programs is the elapsed time for completing some specific tasks or for completing a communication operation. All other performance figures are derived from this basic timing measurement.

Second, we want to investigate the feasibility and efficiency of embedding other kinds of network topologies into the CM-5 fat-tree topology and to devise a benchmark for the basic communication primitives of those topologies on the CM-5. There is an enormous number of parallel algorithms for different types of network topologies in the literature [8, 17]. We address the problem of efficiently embedding meshes and hypercubes into the fat-tree topology, and we present timings for basic mesh and hypercube primitives. Our benchmarking study shows that these embeddings give efficient results and that many algorithms can be transported to the CM-5 with little or no change.

The results of our study make it possible to predict the performance of parallel algorithms without actually running them on the CM-5. We present a Gaussian elimination code and give the corresponding real and estimated execution times in order to show the accuracy of the estimated performance figures.

Related Work

There are numerous articles in the literature about benchmarking different aspects of recent parallel architectures or supercomputers [3, 4, 11, 12, 13, 14, 16]. There are also several benchmark suits specially developed to provide a common ground to test the performance of different high-performance computers [1, 2, 10, 15]. Some of them investigate the use of real application programs, while others employ short kernel codes to evaluate the performance, just as we do here.

Overview

The rest of this paper is organized as follows. Section 2 gives a brief description of the CM-5 architecture. Section 3 introduces the test configurations and the message-passing library that were used to perform our experiments. Section 4 gives the computational performance of the SPARC processor and the vector units. Section 5 presents the benchmarks to measure communication performance from one node to another. Section 6 addresses the global operations provided by the CM-5. Sections 7 shows how meshes and hypercubes can be simulated on the fat-tree network topology. Section 8 presents the estimation of the performance for a Gaussian elimination kernel code on the CM-5.

2 CM-5 System Overview

The CM-5 is a scalable distributed-memory computer system which can efficiently support up to 16,384 computation nodes. Each node contains a SPARC microprocessor and a portion of the global memory connected to the rest of the system through a network interface. Every node in the CM-5 is connected to two inter-processor communication networks, the *data network* and the *control network*. This section gives a brief overview of the CM-5 processing nodes, data, and control networks, which have a remarkable importance in our study.

2.1 Processing Nodes

Each CM-5 computation node consists of a SPARC microprocessor, a custom network interface that connects the node to the rest of the system through data and control networks, a local memory up to 128 Mbytes, and an associated memory controller unit (Figure 1-a.)

SPARC has a clock rate of 33 MHz. It has 64 KB cache that is used for both instructions and data. The SPARC is also responsible for managing the communication with other system components via the network interface.

Node memory is allocated as 8 MB chunks and controlled by a special memory controller. Optionally, this memory controller can be replaced by up to four vector units (Figure 1-b.) In this configuration, size of each memory unit may be either 8 or 32 MB. The scalar multiprocessor is able to issue vector instructions to any subset of vector units. Each vector unit has a vector instruction decoder, a pipelined ALU, and 64 64-bit registers like a conventional vector processor (Figure 2). The 16 MHz vector unit allows one memory operation and one arithmetic operation per clock cycle which gives 16 Mflops peak performance for single arithmetic operations like add or multiply. On the other hand, it can perform a multiply-and-add operation in only one cycle which increases the peak performance to 32 Mflops for this operation. To summarize, a node with four vector units has 256 64-bit data registers, 32 to 128 MB of DRAM memory, and 64 to 128 Mflops peak performance for floating-point arithmetic operations.

All the components inside a node are connected via a 64-bit bus. The bandwidth of the local memory can go up to 512 MBytes per second when vector units are attached.

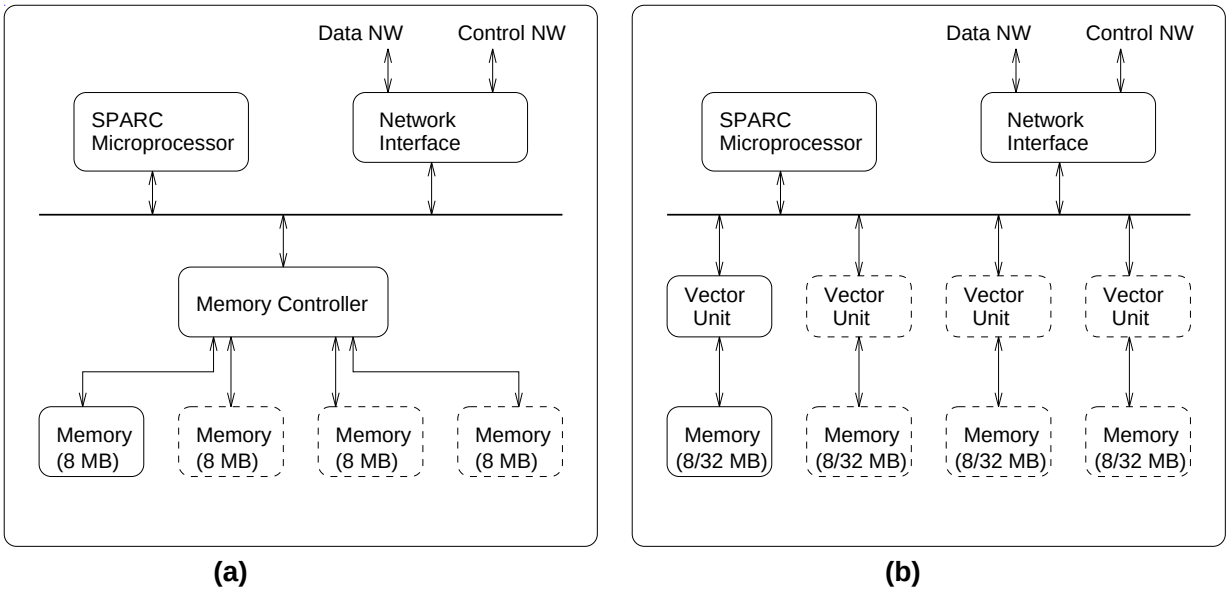

Figure 1: *CM-5 processing node (a) without and (b) with vector units. (The dashed lines indicate optional hardware.)*

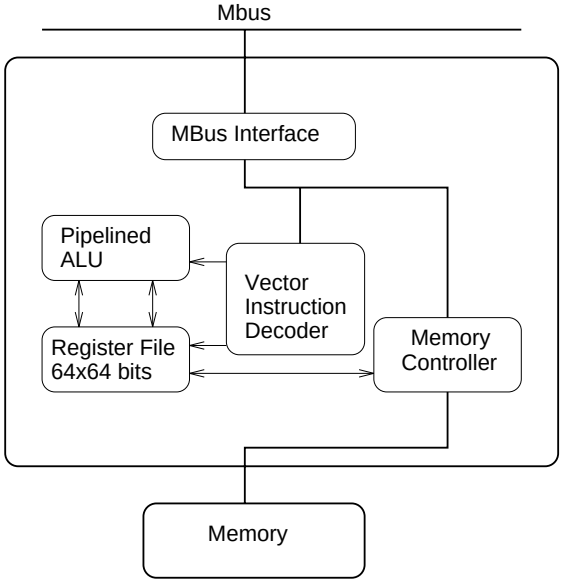

Figure 2: *Vector unit functional architecture.*

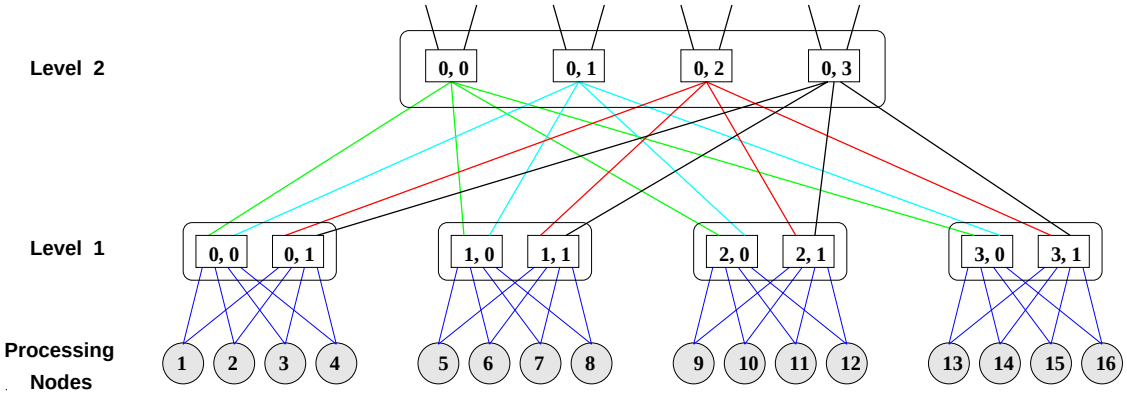

Figure 3: *CM-5 Data network's fat-tree topology with 16 nodes (including network switches.)*

2.2 The Control Network

The CM-5 control network provides high bandwidth and low latency for global operations, such as broadcast, reduction, parallel prefix and barrier synchronizations, where all the nodes are involved.

CM-5 control network has three subnetworks responsible for handling the global operations; a *broadcast subnetwork* which is responsible for broadcast operations, a *combining subnetwork* which supports global operations like reduction or parallel prefix, and a *global subnetwork* which takes care of the synchronization.

2.3 The Data Network

The data network is a high bandwidth network optimized for bulk transfers where each message has one source and one destination. It is a message-passing-based point-to-point routing network that guarantees delivery. In addition, it is deadlock free and has fair conflict arbitration.

The network architecture is based on fat-tree (quad-tree) topology with a network interface at all the leaf nodes. Each internal node of the fat-tree is implemented by a set of switches. The number of switches per node doubles for each higher layer until level 3, and from there on it quadruples. Figure 3 illustrates a data network having 16 nodes. The communication switches are labeled as `(i,j)`, where `i` shows the number of the child switch and `j` the number of the parent switch.

The CM-5 is designed to provide a point-to-point peak transfer bandwidth of 5 MBytes/sec between any two nodes in the system. However, if the destination node is within the same 4-node cluster or 16-node cluster, it can reach to a peak bandwidth of 20 MBytes/sec and 10 MBytes/sec, respectively.

3 Test System

Our experiments were performed on a 32-node CM-5 at the Northeast Parallel Architecture Center at Syracuse University and on a 864-node CM-5 (recently upgraded to 896 nodes) at the Army High Performance Research Center at the University of Minnesota. Both machines are timeshared and run under CMOST version 7.2. There were no one else using the systems while we were running our benchmarking programs.

The CM-5 processing nodes can be grouped into one or more logical partitions, each of which is controlled by a *partition manager*. Each partition uses separate processors and network resources and has equal access to the shared system resources. For example, Minnesota's 864-node CM-5 machine is divided into 32-, 64-, 256- and 512-node partitions.

Most of the values reported in this paper were measured by using a set of short benchmark codes written in C with calls to the CM message-passing library (CMMD Version 3.0 Final). The codes were compiled by using the Gnu C compiler with all the optimizations turned on in order to benefit the full potential of the hardware. The precision of the CM-5 clock is one microsecond. The timings were estimated by recording the CM node busy-time for an average of 100 repetitions of the experiment and dividing the total time by the number of repetitions. *CM node busy-time* is the duration in which the user code is executed on a certain node within its own operating system time-sharing slice. We used the CM Fortran language [5] (Version 2.1.1.2), which partitions and stores the vectors directly into the vector unit memories, to measure the vector unit performance.

As might be expected, testing the hardware system by using high-level software (e.g., CM Fortran or C compilers and CMMD message-passing software) influences the performance negatively. Performance is bounded by the software's ability to exploit the capabilities of the hardware.

3.1 CM-5 Message-Passing Library: CMMD

CMMD [6] provides facilities for cooperative message passing between processing nodes. We used the *nodeless model of programming*, where all the processing nodes execute the same SPMD (Single-Program Multiple-Data) program and the partition manager acts simply as an I/O server.

At the lowest layer, CMMD implements *active messages* [19], which provide fast packet-based communication and simple low-latency array transfer. When a message is to be sent across the data network, the data message is divided into a group of packets of size 20 bytes; 16 bytes of this packet is used for the user data, and the remaining 4 bytes contain control information such as the destination and the message size [7].

Operation	*Operator*	*short int*	*long int*	*single-precision*	*double-precision*
	add	0.23	0.24	0.24	0.24
	subtract	0.23	0.24	0.24	0.24
a[i] & s1 & s2	multiply	0.24	0.24	0.23	0.23
a[i][l] & s1 & s2	divide	0.24	0.24	0.24	0.24
	add & multiply	0.23	0.24	0.24	0.24
	add	0.36	0.37	0.43	0.52
	subtract	0.37	0.37	0.43	0.52
a[i] & b[j] & c[k]	multiply	0.91	0.92	0.43	0.55
a[i][l] & b[j] & c[k]	divide	1.76	1.77	0.94	1.37
	add & multiply	0.44	0.45	0.79	0.58
	add	0.31	0.31	0.36	0.41
	subtract	0.31	0.31	0.36	0.41
a[i] & b[i] & s	multiply	0.70	0.71	0.36	0.44
a[i][l] & b[j] & s	divide	1.56	1.56	0.90	1.03
	add & multiply	0.36	0.36	0.74	0.64

Table 1: *Execution times of various arithmetic operations on* SPARC *microprocessor. (Time is given in microseconds.)*

4 Computation Benchmarks

4.1 SPARC Performance

We run a set of benchmark programs to measure the computational speed of the SPARC microprocessor for basic integer and floating-point operations. Execution times for any of the basic arithmetic operations were the same when all the operands were stored in the registers. We obtained a peak performance of 22 Mips for integer add-multiply and 11 Mips for other integer operations. Floating-point performance was 22 Mflops for add-multiply and 11 Mflops for other operations.

When the operands are not in registers but available in the on-board cache, computation performance drops sharply because of the overhead of accessing the cache. The execution times for various arithmetic operations when the operands are initially stored in the cache are given in Table 1. In the "operation" column an entity like $x\&y\&z$ indicates any combination of these three operands in an arithmetic statement, e.g., $x = y \bullet z$, $y = x \bullet z$, and so on, where $\bullet$ indicates an arithmetic operator.

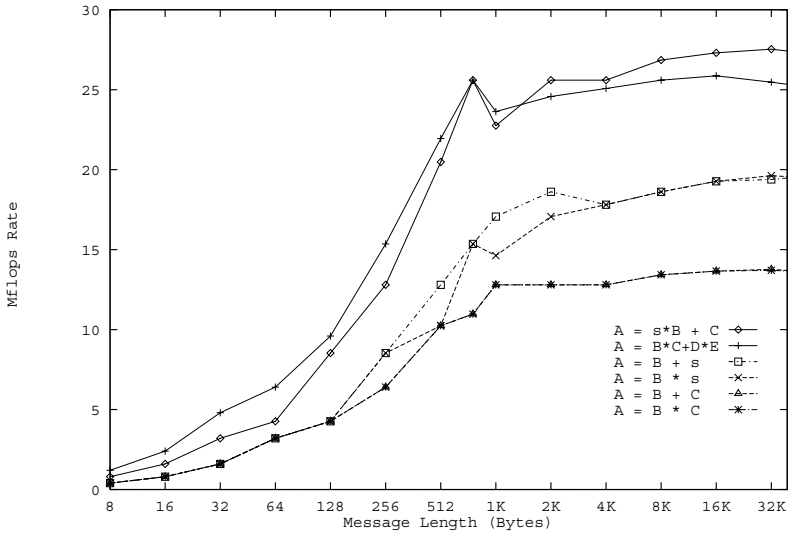

Figure 4: *Performance of vector units in one node for double-precision data.*

#	*Operation*	*One Node Performance*			*Peak Rate (GFLOPS)*		
		R_∞	$N_{1/2}$	N_v	*64-node*	*256-node*	*512-node*
1	A(I) = B(I) + s	19.51	327	22	1.26	4.77	9.25
2	A(I) = B(I) + C(I)	13.53	202	18	0.88	3.42	6.84
3	A(I) = B(I) × s	19.51	324	22	1.26	4.77	9.25
4	A(I) = B(I) × C(I)	13.49	200	16	0.88	3.42	6.84
5	A(I) = s × B(I) + C(I)	27.31	318	18	1.76	6.84	13.59
6	A(I) = B(I) × C(I) + D(I) × E(I)	25.23	190	16	1.64	6.55	12.84

Table 2: *Length-related measures of vector performance for double-precision data.*

4.2 Vector Performance

The performance of vector processing performance on the CM-5 can be characterized by three length-related parameters; R_∞, $N_{1/2}$, and N_v [9]. R_∞ is the asymptotic performance obtained as the vector length tends to infinity, $N_{1/2}$ corresponds to the vector length needed to reach one-half of the R_∞, and N_v is the vector length needed to make the vector mode faster than the scalar mode. The values of these three parameters will depend on the operations being performed.

To evaluate the performance of the CM-5 vector units, we first measured the execution times of some vector operations which are frequently used in scientific application codes. The execution rates for each operation is shown in Figure 4 for vector lengths of up to 32 KB. Then we derived the length-related performance parameters for each vector operation. The results for double-precision and single-precision data are illustrated in Tables 2 and 3,

#	*Operation*	*One Node Performance*			*Peak Rate (Gflops)*		
		R_∞	$N_{1/2}$	N_v	*64-node*	*256-node*	*512-node*
1	A(I) = B(I) + s	11.17	214	20	0.71	2.86	5.52
2	A(I) = B(I) + C(I)	9.10	171	18	0.57	2.28	4.56
3	A(I) = B(I) × s	11.15	212	22	0.71	2.86	5.52
4	A(I) = B(I) × C(I)	9.05	170	20	0.58	2.28	4.77
5	A(I) = s × B(I) + C(I)	18.20	168	28	1.15	4.56	9.53
6	A(I) = B(I) × C(I) + D(I) × E(I)	19.82	160	20	1.25	4.92	9.83

Table 3: *Length-related measures of vector performance for single-precision data.*

respectively.

R_∞ is important for estimating the peak performance. Double-precision operations are always faster than the single-precision ones, since vector unit registers are configured as 64-bit registers, and all the internal buses are of 64-bit. Manipulating a scalar operand (operations 1 and 3) is faster compared to manipulating a vector operand (operations 2 and 4). This is because the scalar operand comes free, while the vector operands in operations 2 and 4 require a memory or cache access to load the corresponding vector into the vector registers.

Additions and multiplications give us about the same timings. Although addition is expected to be faster, cycle time is stretched to handle one addition, one multiplication, or one add-multiply operation in a clock cycle. Therefore, a multiply-add operation gives twice the Mflops rate of a single add or multiply operation.

$N_{1/2}$ is a good measure of the impact of overhead. For finite vector lengths, a start-up time is associated with each vector operation. $N_{1/2}$ parameterizes this start-up time. The use of vector units for processing of vectors shorter than the $N_{1/2}$ will result in significant loss in performance. We obtained large values for $N_{1/2}$ which indicate that efficient use of vector units begins at large vector lengths on the CM-5. $N_{1/2}$ is longer for single-precision data than for double-precision data. This is, in fact, related to the higher Mflops rating of the double-precision data, as explained above.

N_v measures both the overhead and the speed of scalars relative to vectors. The node processor can manipulate vectors of up to about 20 data items faster than the vector units can.

Table 2 and 3 also show the achievable peak rate in Gigaflops when the vectors are distributed across all the vector units. Peak performance figures indicate that, even for 512 nodes, the peak performance is close to the multiplication of the number of processors with the peak speed of a single node. This is a good indication of the scalability of vector processing capability. For these kinds of simple loops there is an insignificant amount of overhead, but it should not be forgotten that the overhead penalties encountered in real case problems may be much larger.

5 Point-to-Point Communication Benchmarks

In distributed-memory machines like the CM-5, data items are physically distributed among the node memories. Thus the performance of the communication primitives used to access non-local data is crucial. Point-to-point communication benchmarks measure basic communication properties of the CM-5 data network by performing the *ping-pong* test between a pair of nodes. The transmission time is recorded as half of the time of a round-trip message in the *ping-pong* test.

We used blocking sends and receives that transfer varying sizes of data blocks between two nodes. Both the source and the destination nodes take active parts in this exchange process, and the receiving node waits until it receives the last data byte from the data network.

Regression analysis of the transmission time allows the calculation of the start-up time and the asymptotic bandwidth between a pair of nodes. The total transmission time T between two nodes can be formulated as

$$T(l) = t_{start-up} + l \times t_{send},$$

where l is the message length in bytes, $t_{start-up}$ is the time to set up the communication requirements, and t_{send} is the transfer time for one unit (byte) of data.

The asymptotic data transfer rate can be found approximately by taking the reciprocal of the transmission time (i.e., $1/t_{send}$.)

5.1 Nearest-Neighbor Communication

In this experiment we studied the communication time for sending a single message to another node in the same cluster of four nodes for different message sizes. This represents the shortest possible distance a message can travel. Figure 5 shows the communication time for messages of size 0-10 KB between two neighboring nodes on a 32-node CM-5. The communication time increases linearly with the increasing message size. To establish a communication link between two nodes, a preliminary handshake is required. This *start-up time* is observed to be 84.65 microseconds. Using a linear chi-square fit, we can model the communication time for aligned messages within a cluster of four processors as a function of message size:

$$T(l) = 84.65 + 0.117 \times l \ \ microseconds. \tag{1}$$

The thick appearance of the curve in Figure 5 is because of the *sawtooth effect* caused by data alignment patterns. Figure 6 shows a smaller section (for message sizes of 320–576 bytes) of the previous graph to magnify this sawtooth effect. As indicated by dips in the curve, when the message length is a multiple of the byte size, the communication time goes down to a local minimum. On the CM-5, the unaligned message transfer is more costly than aligned message transfers, but the communication time differences between byte-aligned, word-aligned, and double-word-aligned data are negligible. As stated earlier, each data packet contains 16 bytes of user data. Misalignment causes hardware complications since the memory is typically aligned on a word boundary. A misaligned memory access will

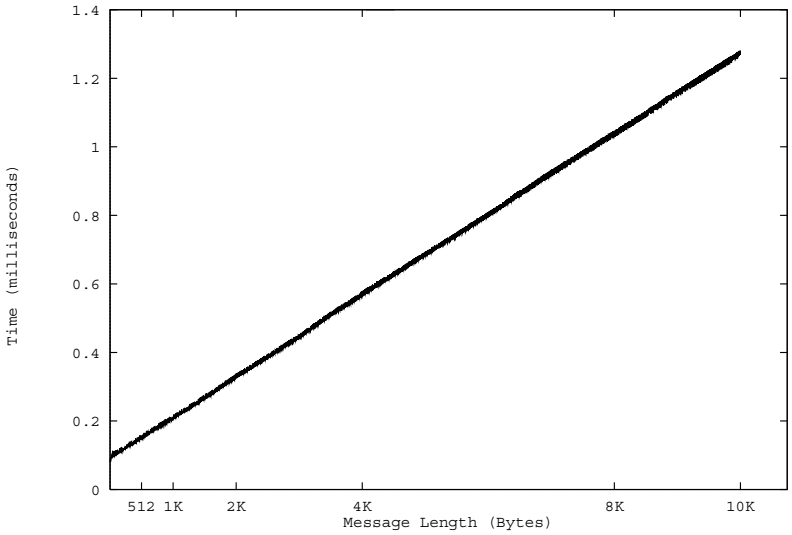

Figure 5: *Communication time between two nearest-neighbor nodes as a function of message size.*

be performed, therefore, by using several aligned memory accesses. In addition, since the network interface accepts only word and double-word writes, odd-sized buffers can not be efficiently moved into the data registers.

We studied the maximum bandwidth that can be sustained for a single message traveling to the shortest possible distance for message sizes up to 32 Kbytes. Figure 7 illustrates that the transfer rate (approximately $1/t_{send}$) for an aligned buffer is around 8.5 MB/sec. This bandwidth is significantly lower than the theoretical peak bandwidth of 20 MB/sec. In the current CMMD implementation, a node's ability to inject data into the network is much less than the network's capacity to accept the data [14]. Assembler codes can achieve close to 18 MB/sec moving data from one node's registers to another's [18]. However C codes with calls to the CMMD library tend to run slower, partly because the C compiler's output is never as efficient as a hand-crafted assembler code.

5.2 Effect of Distance on Communication

In this section we examine how the communication between any two nodes compares with the communication between two nearest neighbors. We measured the communication time from node 0 to every other node using the same strategy as in the previous section. Figures 8 and 9 show the effect of distance on the communication time on a 512-node CM-5 for message sizes of 16 bytes and 1 Kilobyte, respectively. If we ignore the spikes related to noise in the network, it can be observed that the communication time is not significantly affected by the inter-node distance. Each time another level of the fat-tree topology is traversed, there is a slight increase in time (about 1 microsecond.) This is due to the cost of traversing an extra switch in the data network, i.e., the cost of extra hop needed

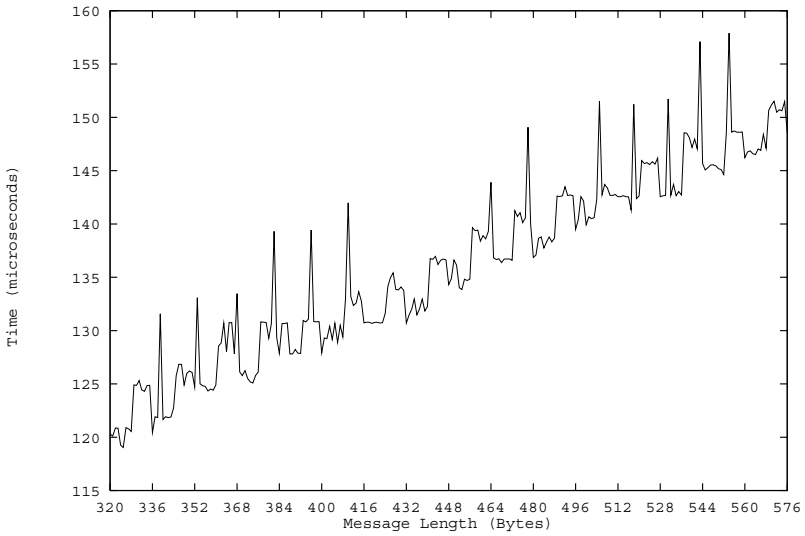

Figure 6: *Communication time between two nearest-neighbor nodes for message sizes of 320–576B.*

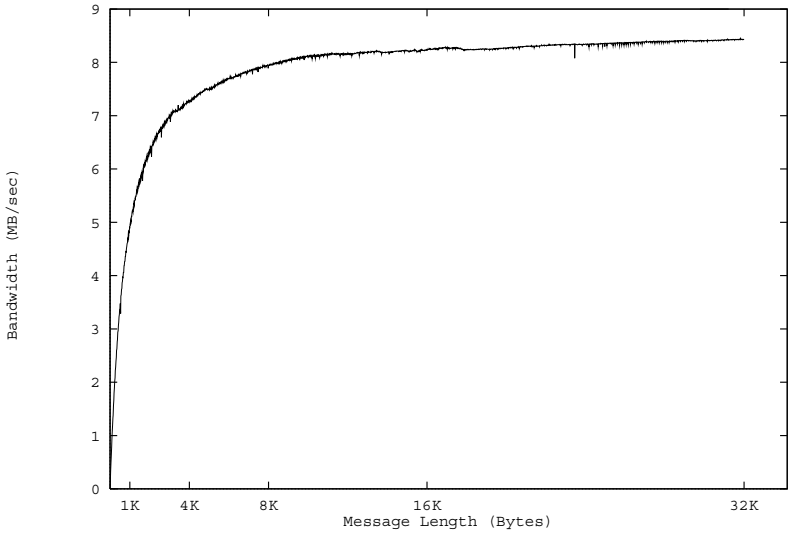

Figure 7: *Transfer rate between two nearest neighbors for word-aligned messages.*

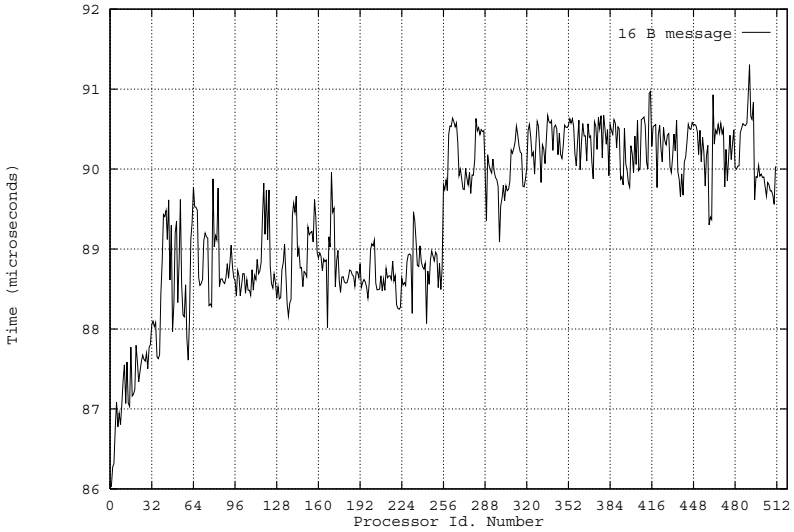

Figure 8: *Communication time between node 0 and other nodes on a 512-node CM-5 for a message of 16 bytes.*

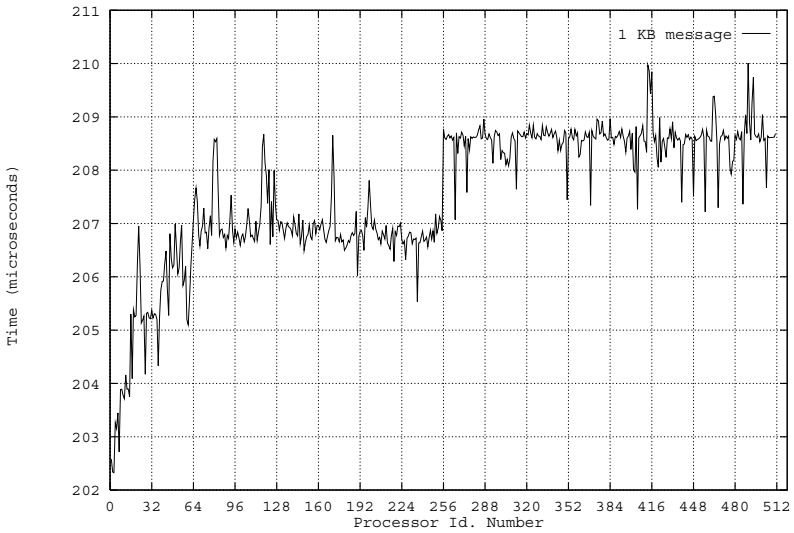

Figure 9: *Communication time between node 0 and other nodes on a 512-node CM-5 for a message of 1 Kilobyte.*

The transmission time difference between the nearest neighbor and the neighbor at the maximum distance is less than 5 microseconds on a 512-node CM-5. The results are consistent for both short (16 bytes) and long messages (1 Kilobyte.)

6 Global Communication Benchmarks

The CM-5 hardware supports a rich set of global (cooperative) operations. Global operations involve any data transfer among processors, possibly with an arithmetic or logical computation on the data while it is being transferred. Collective communication patterns, such as reduction, broadcast, concatenation or synchronization, are very important in the implementation of high-level language constructs for distributed-memory machines.

We measured the performance of the communication networks by using a set of benchmark programs employing the global operations provided by the CM-5 hardware.

6.1 Scans

A *scan* (parallel prefix) operation creates a running tally of results in each processor in the order of the processor identifier. Assuming that the `A[j]` represents the element `A` in the `j`th processor and `R[j]` represents the result `R` in the `j`th processor, an inclusive scan with a summation operator performs the following operation:

$$R[i] = \sum_{j=0}^{i} A[j], \quad 0 \leq i < Number\_of\_Processors - 1.$$

Table 4 summarizes the performance of scan operations using different data types on a 32-node CM-5. Integer scan operations take about 6 microseconds. On the other hand, the double-precision minimum/maximum scans and add scans are about 3 to 5 times slower than the integer scans.

In a *segmented scan*, independent scans are computed simultaneously on different subgroups (or segments) of the nodes. The beginning of segments are determined at run-time by an argument called the *segment-bit*. Table 4 shows the performance of the segmented scan operations on a 32-node CM-5, assuming the *segment-bit* of a processor is turned on with a probability of 10%. Computation of integer-segmented scans takes slightly longer than regular scans, primarily because of testing the extra condition at run-time. Timings for the double-precision maximum or minimum segmented scans are almost equal to those for regular scans, but the time for a double-precision segmented add scan operation is almost twice that of a corresponding regular scan operation.

The CM-5 control network has integer arithmetic hardware that can compute various forms of scan operations. Integer minimum, maximum, and logical segmented scans are also supported by the hardware. On the other hand, single- and double-precison floating-point scan operations are handled partially by software, which results in a much longer time. While the floating-point minimum and maximum scans take advantage of the hardware partially,

Operation	*type*	*add*	*max*	*min*	*ior*	*xor*	*and*
scan	integer	6.33	6.41	5.47	6.08	6.06	5.17
scan	unsigned int.	6.30	5.54	5.50	6.06	6.06	5.16
scan	double-precision	33.70	21.28	20.37	-	-	-
segmented scan	integer	6.95	6.80	6.13	6.77	6.77	5.89
segmented scan	unsigned int.	6.96	6.24	6.15	6.76	6.73	5.85
segmented scan	double-precision	57.35	19.93	20.31	-	-	-
reduction	integer	4.62	4.36	4.03	4.35	4.38	3.71
reduction	unsigned int.	4.61	3.98	4.00	4.33	4.34	3.70
reduction	double-precision	28.38	14.24	17.32	-	-	-

Table 4: *Execution times of global operations on a 32-node CM-5. Time is in microseconds. ('-' represents an undefined operation.)*

floating-point add scan is performed almost completely by the software. This is the reason *add* scans and segmented scans are so costly.

6.2 Reductions

A *reduction operation* takes an input value from each node, applies a global operation such as *summation*, *minimum* or bitwise *xor* on all the values, and returns the result to all other nodes.

We measured the speed of combining subnetworks for various types of reduction operations (Table 4). Double-precision reduction operations take 4 to 6 times longer than integer reductions. Again, this can be explained by the same reasons described above.

6.3 Concatenation

Some computations on distributed data structures require that each processor receive data from all the other processors. For example, in the classical N-body algorithm, every particle interacts with every other particle. *Concatenation* is a cumulative operation that appends a value from each processor to the values of all the preceding processors in processor identifier order.

Assume that there are `P` processors, and `B` = `N/P` data elements of a large vector are distributed among these processors so that processor `p` contains a vector V_p`[p.B···(p+1)B-1]`. The global concatenate operation stores the resultant vector `V[0 ··· N-1]` in every node.

We tested the effects of message size and number of processors on the concatenation operation execution time. Figure 10 shows the time required for the concatenation operation using 32-, 64-, 256-, and 512-node partitions. We can derive the following equation for the

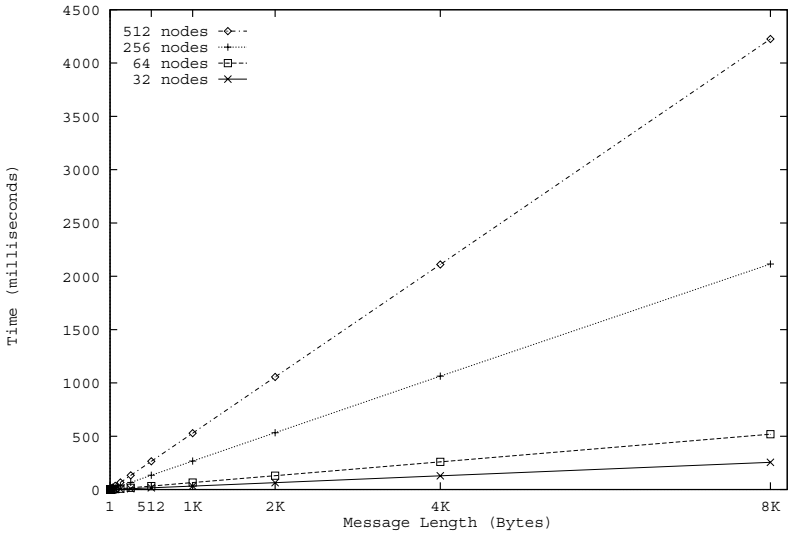

Figure 10: *Execution time for concatenation operation using 32, 64, 256 and 512 nodes.*

concatenation operation.

$$T(l, P) = 23.44 + 0.975 \times (P \times l) \quad microseconds,$$

where p is the number of processors in that partition and l is the size of the local portion of the data to be concatenated. Note that time for concatenation depends only on `P` for its contribution to the message size, and the the operation is itself independent of `P`.

From Figure 10 it is clear that the time for concatenation on 512 nodes is about 16 times larger than the time on 32 nodes, which may be surprising when compared to scan operations. The amount of data sent by each node is about `N` data items which leads to `N` $\times$ `P` data items in the network and may cause congestion in the network, especially for large messages. Therefore, as the message length and number of processes increase, the horizontal distance between the lines increases.

6.4 One-to-All Broadcast

When we use SPMD style programming, one of the basic types of communication is to broadcast a value from one node to all the other nodes. For example, spreading a row to all other rows is a common operation in LU Decomposition and many other linear algebra computations. On the CM-5 any node can broadcast a buffer of a specified length to all other nodes within the partition.

We measured the performance of the broadcast subnetwork using CMMD broadcast intrinsics. The results for 32-, 64-, 256- and 512-node partitions are shown in Figure 11.

We can derive Equations 3 and 4 for a 32- and a 512-node CM-5, respectively.

$$T(l) = 6.96 + 1.15 \times l \quad microseconds. \tag{2}$$

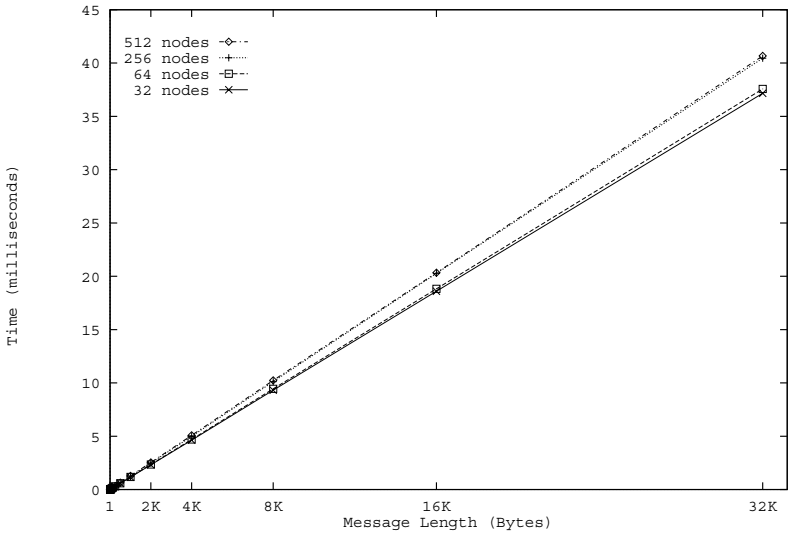

Figure 11: *One-to-all broadcast timings on 32-, 64-, 256- and 512-node partitions.*

$$T(l) = 7.40 + 1.24 \times l \quad microseconds. \tag{3}$$

The broadcast time is almost the same for 32- and 64-node partitions, and for 256- and 512-node partitions. Since the broadcast is implemented in the network in a spanning tree fashion, the number of hops (or switches traversed) slightly affects the timings. Since values can be reduced in 3 hops in 32- and 64-node partitions (which can communicate via the third level of the fat-tree), it is faster than using 256- and 512-node partitions, which require 4 and 5 hops, respectively. Moreover, the initial setup times for different sized partitions slightly differ, as seen in the above equations.

6.5 Synchronization

Synchronization is very important in MIMD machines since they are fundamentally asynchronous and must be synchronized prior to most communication steps. Many machines, also use the common communication network also for synchronization, causing significant performance degradation. The CM-5 uses a separate barrier synchronization network (the control network) to carry out synchronization efficiently. We measured the delay to do a global synchronization on CM-5and found that it takes 5 microseconds, independent of the number of nodes in the partition.

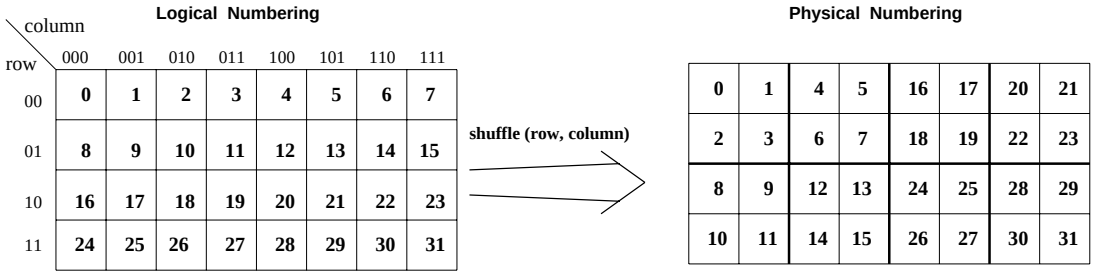

Figure 12: *A* 2×2 *locality-preserving mapping of a* 4×8 *mesh to a 32-node CM-5.*

```
Logical_ProcNum_TO_Coordinate(L_PNum, row, col)
{
   * row = L_PNum / NUM_COL;
   * col = L_PNum % NUM_COL;
}
```

```
Coordinate_TO_Physical_ProcNum(row, col, PNum)
{
   result = 0;
   for ( pos = intlog2(NUM_COL)-1; pos >= 0; pos--) {
     result = (result << 1) | getbit(row, pos);
     result = (result << 1) | getbit(col, pos);
   }
   *PNum = result;
}
```

Figure 13: *Two main functions used for address calculation for the mapping of a mesh onto the CM-5 fat-tree topology.*

7 Embedding of other topologies into CM-5 fat-tree

7.1 Embedding of a mesh into fat-tree

A wrap-around mesh (torus) can be embedded into the CM-5 fat-tree-based architecture by using the *shuffle row-major mapping* [17]. The physical node number corresponding to a logical mesh point is found by shuffling the row and column binary numbers of that point in the mesh topology. If a processor's location is `row`=*abcd* and `col`=*efgh*, then bitwise shuffling of `row` and `col` gives the bit string *aebfcgdh*. This kind of mapping preserves the locality of 2×2, 4×4, etc. submeshes. A representative example for this is illustrated in Figure 12.

`Logical_ProcNum_TO_Coordinate()` and `Coordinate_TO_Physical_ProcNum()` are two basic routines used for mapping a point on an $m \times n$ mesh to a node of the fat-tree. The former is used to calculate the coordinate location of a point on the mesh. It is also useful for determining the neighbors of a point on the mesh. The latter is used to transform a given location on the mesh to a physical node number on the fat-tree. `getbit()` returns the corresponding bit of the string at the specified position. These routines are listed in Figure 13 for reference.

Table 5 displays the timings for shift operations in a given direction which are very common in mesh topologies. We simulated 16×32, 8×64, 4×128, and 2×256 meshes mapped to the fat-tree topology on a 512-node CM-5.

Mesh Size	*Message Length*	*NORTH*		*EAST*		*WEST*		*SOUTH*	
		max	min	max	min	max	min	max	min
16x32	16 KB	3.83	3.58	4.21	4.01	3.86	3.62	3.84	3.56
16x32	32 KB	8.29	7.96	7.34	7.12	7.34	7.10	7.55	7.03
16x32	64 KB	16.55	15.99	16.16	15.56	17.47	16.78	16.24	15.56
8x64	16 KB	5.05	4.69	3.86	3.60	3.91	3.59	3.85	3.55
8x64	32 KB	7.87	7.53	7.24	7.00	7.26	6.98	7.31	6.92
8x64	64 KB	14.92	14.48	15.65	15.25	16.26	15.86	16.80	16.27
4x128	16 KB	3.92	3.73	4.74	4.50	4.54	3.58	3.92	3.71
4x128	32 KB	7.51	7.13	7.43	6.96	7.81	6.94	7.51	7.11
4x128	64 KB	15.69	13.60	14.18	13.63	15.74	15.17	16.48	16.01
2x256	16 KB	3.93	3.41	4.79	4.52	3.89	3.60	3.93	3.39
2x256	32 KB	8.53	7.63	7.45	6.97	9.26	6.92	7.60	7.38
2x256	64 KB	16.24	15.70	16.18	15.67	15.68	15.21	15.07	14.45

Table 5: *The timings for* 16×32, 8×64, 4×128 *and* 2×256 *mesh simulations on a 512-node CM-5 (time is in milliseconds).*

We can deduce from Table 5 that mesh bandwidths are at about 4 Mbytes per second, which is less than the expected 5 Mbytes/sec bandwidth between any arbitrary nodes. The main reason for that is the contention happening in the data network when all the nodes send long data messages at the same time.

7.2 Embedding of a hypercube into fat-tree

For many computations, the required communication pattern is similar to the connections of a hypercube architecture. These include bitonic sort, the Fast Fourier Transform, and many divide-and-conquer strategies [17]. This section discusses the time requirements for such types of communication patterns.

A d-dimensional hypercube network connects 2^d processing elements (PEs). Each PE has a unique index in the range of `[0,2`^{d}`-1]`. Let $(\mathtt{b}_{d-1}\mathtt{b}_{d-2}\ldots\mathtt{b}_0)$ be the binary representation of the PE index p and $\overline{b}_k$ be the complement of bit b_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit; i.e., processor $(\mathtt{b}_{d-1}\mathtt{b}_{d-2}\ldots\mathtt{b}_0)$ is connected to processors $(\mathtt{b}_{d-1}\ldots\overline{\mathtt{b}}_k\ldots\mathtt{b}_0)$, $\mathtt{0} \leq \mathtt{k} \leq \mathtt{d\text{-}1}$. We use the notation $p^{(k)}$ to represent the number that differs from p in exactly bit k.

Node p of a logical hypercube is mapped onto node p of the CM-5 (Figure 14). We consider communication patterns in which data may be transmitted from one processor to another if it is logically connected along one dimension. At a given time, data is transferred from PE p to PE $p^{(k)}$ and from PE $p^{(k)}$ to PE p.

The communication patterns performed for a logical hypercube on the CM-5 using this

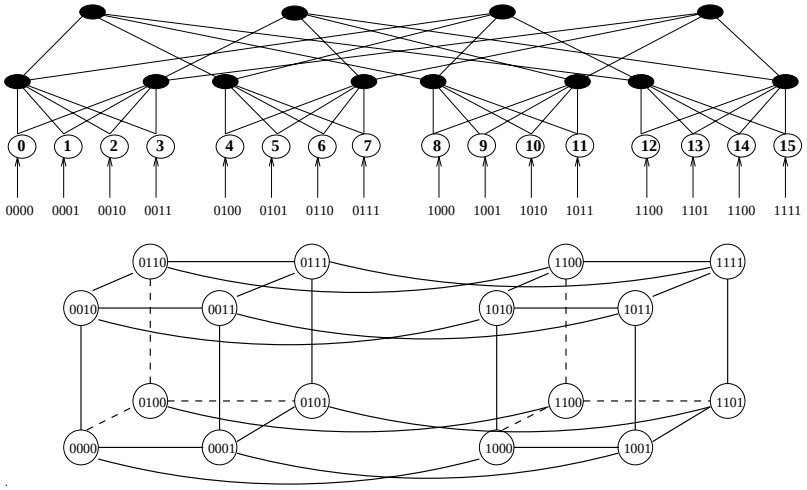

Figure 14: *Embedding of a 4-cube into a 16-node fat-tree.*

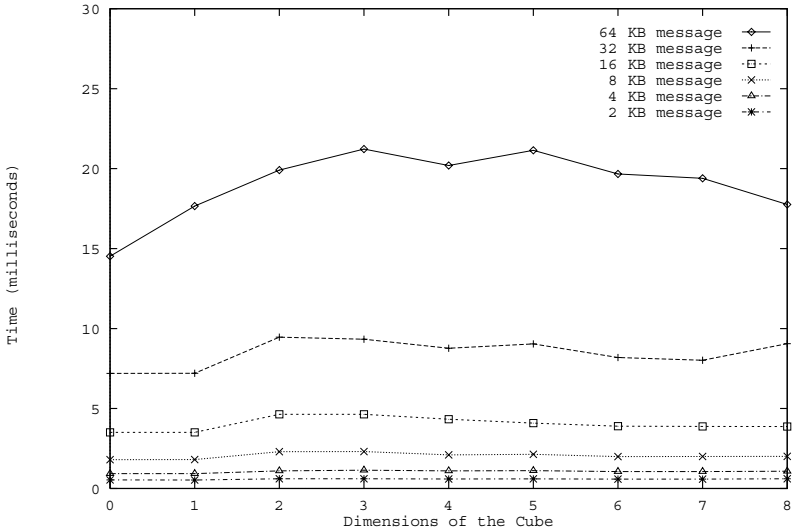

Figure 15: *Communication times for each level of the quad-tree (leaves are at level 0, root is at level 8.)*

mapping are shown in Figure 15. The first two dimensions of the cube require the first level of the fat-tree to be traced, and the 8th dimension needs five levels to be traced on a 512-node CM-5. We observe that all six plots are almost horizontal, from which we conclude that the time required for swapping data along different dimensions is approximately the same for all dimensions and that it scales linearly with the size of the message.

Having more switches at higher levels is one reason for being able to achieve this performance. More bandwidth can therefore be handled as we go up in the network connection tree. The rate of transfer is between 3.3 Mbytes/sec and 3.6 Mbytes/sec, respectively. This is close to the peak bandwidth for long-range communication on the CM-5.

8 Performance Estimation for Gaussian Elimination

Modeling of basic computation and communication primitives is often used in estimating the performance of a given program [20]. We illustrate how to estimate the performance of a program by using the results stated in the previous sections. A Gaussian elimination code that uses the *row-oriented algorithm with partial pivoting* algorithm [8] is given in Figure 16. Assuming that there are `P` nodes, the rows of the matrix `A[N][N]` are distributed using a block-mapping strategy, such that the first `N/P` rows are assigned to node 0, the second `N/P` rows are assigned to node 2, and so on. The code gives just the enough detail about the elimination phase, back-substitution phase is not shown here.

The elimination phase is performed column by column. The outer loop which iterates over pivots is executed in parallel by all processors. Within the loop body there are computational phases, separated by communication phases. Computational phases include finding the maximum value of the current column among the rows owned, computing the multipliers, updating the permutation vector in which the pivoting sequence is saved, and reducing the part of nonpivot rows. Communication phases include a reduction operation to determine the pivot value in a column, another reduction operation to find the maximum row number (pivot) in the case of a tie among the processors, and a broadcast operation to announce the pivot row to all nodes. This code uses collective communication primitives but does not attempt to overlap computation and communication.

The costs of the communication operations (as modeled by our benchmarking programs) required for the Gaussian elimination are given in Tables 6 and 7. We counted the number of arithmetic operations performed in the inner loop bodies to determine the computational time in one iteration. The execution time of each iteration is multiplied by the number of iterations to obtain the estimated time. There are `N` iterations for a matrix of size `N`$\times$`N`.

We counted the conditional expressions as one arithmetic operation (according to the type of test) as in the GENESIS benchmark suite [10]. The percentage of the time the conditional test evaluates to true depends on the specific values assigned to a specific processor at a given time. We assumed the condition yields a true value 50% of the time which is a very close approximation in average.

This code was executed on a 32-node CM-5. The measured results are compared to the estimated results in Table 7 and are found to be within 10% of the estimated results for

```
1.  v = 0;    done [0:BS] = FALSE;
2.  for  (j=0; j<N; j++)  {
3.     locPivotVal = MIN_VAL;
4.     locPivot = 0;
5.     for  (i=0; i<BS; i++)
6.        if  (A[i][j] > locPivotVal) {
7.           locPivotVal = A[i][j];
8.           locPivot = mypid*BS+i;
9.        }
10.    pivotval = Reduce_double(locPivotVal, MAX);
11.    if  (pivotval != locPivotVal)
12.       locPivot = -1;
13.    pivot = Reduce_int(locPivot, MAX);
14.    perm[v++] = pivot;
15.    done[pivot] = TRUE;
16.    for  (i=0; i<BS; i++)
17.       fac[i] = A[i][j] / pivotVal;
18.    if  (pivot == locPivot)
19.       Broadcast_src(A[pivot][0:N], (N+1)*sizeof(double));
20.    else
21.       Broadcast_dest(pivotRow, (N+1)*sizeof(double));
22.    for (i=0; i<BS; i++)
23.       if (!done[mypid*BS+i])
24.          for (k=j; k<N+1; k++)
25.             A[i][k] -= fac[i] * pivotrow[k];
26. }
```

Figure 16: *The Gaussian elimination SPMD node program for static execution time estimation.*

Operation	*Reference*	*Mesh Sizes*			
		64×65	128×129	256×257	512×513
Double reduction using maximum	Table 4	14.24	14.24	14.24	14.24
Integer reduction using maximum	Table 4	4.36	4.36	4.36	4.36
Broadcast double array from a node	Equation 2	604.96	1193.76	2371.36	4726.56
Computation	Table 1	31.39	117.18	451.96	1774.32
Time per iteration(microsec)		654.95	1329.54	2841.92	6519.48

Table 6: *Cost of required operations for Gaussian elimination on a 32-node CM-5 (time is in microseconds).*

Matrix Size	64×65	128×129	256×257	512×513
Estimated Time(msec)	41.92	170.18	727.53	3337.97
Measured Time(msec)	45.62	185.52	787.29	4598.69

Table 7: *Comparison of the estimated and measured times for Gaussian elimination code on a 32-node CM-5.*

matrices of size smaller than 512×512. For a 512×512 coefficient matrix, there is a bigger discrepancy since the matrix is too big to fit into cache, therefore extra memory overhead is incurred to fetch and bring the data into cache.

As seen, such modeling can be very useful in performance prediction for different algorithms on the CM-5. This information can be used to choose optimal algorithms and to optimize program codes and automate performance estimation at compile-time by using the cost function of each basic primitive.

9 Conclusions

In this paper we presented a benchmarking study of the computation and communication performance of the CM-5 multicomputer. We formulated the communication overhead in terms of message size and latency.

Using vector units become more efficient than using only the `SPARC` microprocessor, when the vector lengths go over twenty. We can get half the peak performance for vector lengths of 100–200 for single-precision numbers, and of 200–300 for double-precision numbers. Vector units give us up to 30 Mflops rate which results in about a 15 Gflops processing rate for a 512-node CM-5.

Communication benchmarks show that the data network has a start-up latency of 84 microseconds and a bandwidth of 8.5 MB/sec for unidirectional transfer between two nearest-neighbor nodes. Communication latencies for misaligned messages are longer than latencies for aligned-messages. Message transmission latencies and bandwidths are independent of partition size and vary only slightly with the number of network levels crossed.

There are several global operations that use the control network for communication. Concatenation operation requires time linearly proportional to the size of the resultant array. The reduction operators take about 5 microseconds for integers and 15–20 microseconds for floating-point numbers. Scans and segmented scans are quite fast and can be completed in 6–7 microseconds for integers.

We simulated basic communication primitives of mesh and hypercube topologies on the CM-5. The bandwidth for hypercube-type of communications was less than 4 MB/sec. This was also true in cases when all communication passed through the root of the CM-5 interconnection network. For mesh-type of communication patterns, the bandwidth was again about 4 MB/sec.

The CM-5 data and control networks were found to be highly scalable. The performance figures remained constant for most operations when we evaluated similar primitives from 32 to 512 nodes.

We used the timing results of the computation and communication primitives in estimating the execution time of a small program. We implemented the Gaussian elimination algorithm with partial pivoting on the CM-5. The real execution time of the algorithm was found to be close to the estimated time which shows that we can use the results of our study to do static performance estimation at compile-time before running a program.

Acknowledgments

We would like to thank Steve Swartz of the Thinking Machines Corporation for his many clarifications regarding the architecture of the CM-5. We would like to thank the Army High Performance Computing Center at the University of Minnesota for providing access to their CM-5. We would also like to thank Kubilay Cardakli, Betul Dincer, and Elaine Weinman for proofreading this manuscript and Nancy McCracken for her administrative support during the preparation of this manuscript.

References

[1] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, D.H. Simon, V. Venkatakrishnan, and W. Weeratunga. The NAS Parallel Benchmarks. *Int. J. of Supercomputer Applications*, **5** (3):63–73 (1991).

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int. J. of Supercomputer Applications*, **3** (3):5–40 (1989).

[3] L. Bomans and D. Roose. Benchmarking the IPSC/2 Hypercube Multiprocessor. *Concurrency: Practice and Experience*, **1** (1):3–18 (1989).

[4] Z. Bozkus, S. Ranka, and G.C. Fox. Benchmarking the CM-5 Multicomputer. In *Proc. of 4th Sym. on the Frontiers of Massively Parallel Computation*, 100–107 (1992).

[5] Thinking Machines Corporation. *CM Fortran Programming Guide* (1993).

[6] Thinking Machines Corporation. *CMMD Reference Manual, Version 3.0* (1993).

[7] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary* (1993).

[8] G.C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall (1988).

[9] J. L. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann (1990).

[10] A.J.G. Hey. The GENESIS Distributed-Memory Benchmarks. *Parallel Computing*, **17** (10-11):1275–1283 (1991).

[11] R.W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, **17** (10-11):1111–1130 (1991).

[12] T.T. Kwan, B.K. Totty, and D.A. Reed. Communication and Computation Performance of the CM-5. In *Proc. of Supercomputing 1993*, pages 192–201 (1993).

[13] C.E. Leiserson and et al. The Network Architecture of the Connection Machine CM-5. In *Proc. of Parallel Algorithms and Architectures Symposium*, 272–285 (1992).

[14] M. Lin, R. Tsang, D.H.C. Du, A. E. Klietz, and S. Saroff. Performance Evaluation of the CM-5 Interconnection Network. In *Proc. of Spring COMPCON 93* (1993).

[15] P. Messina, C. Baillie, E. Felten, P. Hipes, R. Williams, A. Alagar, A. Kamrath, R. Leary, W. Pfeiffer, J. Rogers, and D. Walker. Benchmarking Advanced Architecture Computers. *Concurrency: Practice and Experience*, **2** (3):195–256 (1990).

[16] R. Ponnusamy, R. Thakur, A. Choudhary, K. Velamakanni, Z. Bozkus, and G.C. Fox. Experimental Performance Evaluation of the CM-5. *J. of Parallel and Distributed Computing*, **19** (3):192–202 (1993).

[17] S. Ranka and S. Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition (Bilkent University Lecture Series).* Springer-Verlag (1990).

[18] S. Swartz. Thinking machines corporation. Personal communications (1992).

[19] T. Von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th International Symposium on Computer Architecture*, 256–266 (1992).

[20] W. Wu, M.-Y.and Shu. Performance Estimation of Gaussian-Elimination on the Connection Machine. *1989 Int. Conf. on Parallel Processing*, 181–184 (1989).