Performance Study on the Connection Machine

Min-You Wu and Wei Shu

YALEU/DCS/TR-717
June 1989

This work appeared in the proceeding of the International Conference on Parallel Processing, 1989.

PERFORMANCE STUDY ON THE CONNECTION MACHINE

Min-You Wu and Wei Shu
Department of Computer Science
Yale University
New Haven, CT 06520
wu-min-you@cs.yale.edu shu.wennie.wei@cs.yale.edu

Abstract. In this paper, we present a performance study on the Connection Machine. Our study shows that communication overhead is so large that it dominates program performance. The study is broken down into five different groups of tests. In the first group, processor performance is measured. The second and third groups measure the performance of communication. NEWS grid communication is tested in the fourth group. In the fifth group, we test performance for virtual processors. We also present a method to estimate program performance based on this study. A programming guide suggests many hints to program the Connection Machine for better performance.

1. INTRODUCTION

Advances in VLSI technology have made it possible to make many reliable, inexpensive processors in a single chip. The small power dissipation and size of these chips facilitates the development of computers containing many processors working in parallel. The Connection Machine is a fine-grained, massively parallel machine produced by Thinking Machines Corp. This single instruction multiple data (SIMD) machine explores massively data parallelism by its 64K data processors. See [Hil85], [TR88], [SS88], [McB88] for more information on the architecture of the Connection Machine, benchmarking, and results for PDE problems.

Many researchers have concentrated on implementing various applications on the Connection Machine. However, not many individuals have studied the performance of these programs. In this paper, we present a performance study. The overall performance depends on individual computation and communication operations, as well as interactions between them. To achieve better performance, it is necessary to understand the impact of communication on program performance, and be able to predict the approximate behavior of the program. The performance measurement provides such possibilities. After briefly introducing the background of the Connection Machine, five groups of tests are presented in Section 3. These tests measure the performance of processors, communication between the front-end and processors, communication among processors, NEWS grid communication, and virtual processors. In section 4, a guideline is presented to program the Connection Machine for better performance. In section 5, the measurement results are then used to estimate program performance on the Connection Machine.

2. BACKGROUND

The architecture of the Connection Machine is shown in Figure 1. The front-end is a conventional computer, such as a DEC VAX 8000 series or a Symbolics 3600 system. As an SIMD machine, all instructions are issued from the front-end. All the other processors, called data processors, carry out the same operation synchronously. Each processor can access data from its 8K bytes memory at a rate of 5 megabits per second. There is a Weitek floating-point processor chip for every 32 single-bit processors. A fully configured CM-2 has 2K Weitek chips and 64K single-bit data processors, that are packaged 16 per chip. These processor chips are connected in a hypercube network.

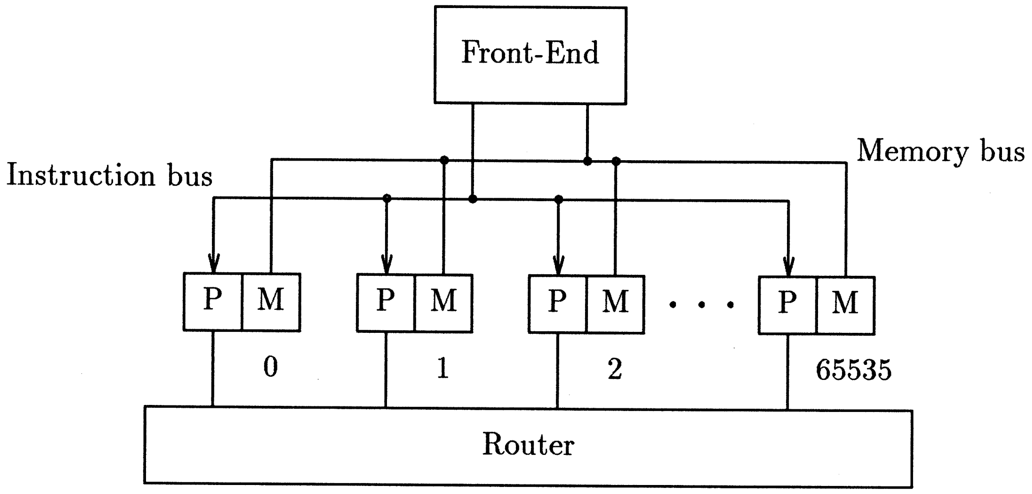

Figure 1: The Connection Machine Architecture.

All instructions reside in the memory of the front-end. The serial instructions are executed on the front-end. The parallel instructions are broadcasted through the *instruction bus* to data processors. This bus may also broadcast data to processors. Another bus, the *memory bus*, allows the front-end to read or write the memories of processors, one word at a time. General communication between processors is handled by a *router*, so that all processors can simultaneously store data into the local memories of other processors. On the same hand, processors may also fetch data from the local memories of other processors.

In the Connection Machine, a processor is assigned to every data item. If the number of data items exceeds the number of physical processors, virtual processors may be used. In the virtual processor scheme, the system logically divides the memory of each processor into n portions, and processes each directive n times, once for each portion, thereby timeslicing the physical processor n-fold [Thi87a]. The ratio of the number of virtual processors to physical processors is referred to as the *VP ratio.* The execution rate is higher than $\frac{1}{VP\ ratio}$ the rate of the physical processors because of the instruction sharing by virtual processors.

The programming languages currently supported for the Connection Machine system include C* and *Lisp. Both have new data types extended and are pretty close to their corresponding serial language specifications [Thi87a],[Thi88]. These languages can be compiled into a parallel instruction set, called *Paris*, which is used by the front-end computer to direct the actions of data processors.

3. PERFORMANCE MEASUREMENT

In this section, we present a performance study of the Connection Machine. The study is broken down into five different groups of tests. In the first group, processor performance is measured for integer and floating-point operations. The second group measures the performance of communication between the front-end and processors. The third group tests processor-to-processor communication. In the fourth group, NEWS grid communication is tested. For these four groups, the VP ratio of 1 is used. In the fifth group, we test the performance of processors and communication for several different VP ratios.

It is important to note, that in each group of tests, we give both *Real time* and *CM time*. Real time is the elapsed time of the front-end's wallclock. CM time is the time it takes to execute parallel instructions on the sequencer. The test is carried out on the CM-2 with 8K processors and operating under version 5.0 at a clock speed of 6.7 MHz. The front-end is a Symbolics 3600 and the programming language is *LISP. If not mentioned elsewhere, the length of all data is 32-bits.

Processors

One of the design principles of SIMD machines is that a large number of low-cost processors work synchronously to exploit massively data parallelism. In the Connection Machine, there are up to 64k data processors, each containing a simple one-bit ALU. Thus, all integer arithmetic and logic operations are carried out in a bit-serial fashion [Thi87b]. In Figure 2, we show the completion time of integer Add and Multiply operations with the different lengths of words (CM time only). The time to finish an operation increases with the number of bits in a word. However, Multiply grows much faster than Add. In the old version of the Connection Machine, floating-point operations were also performed in a bit-serial fashion. Currently, Weitek floating-point chips exist in the Connection Machine, and all floating-point operations taking place here are much faster than before. Each Weitek chip is responsible for computing data from 32 data processors. Table 1 shows the result of processor performance. Here, the integer number is 32-bits long and the floating-point operation is with single precision. Note that an integer number is converted into the floating-point format first, then a floating-point Divide operation is performed in the Weitek chip to avoid extremely slow bit-serial Divide operation.

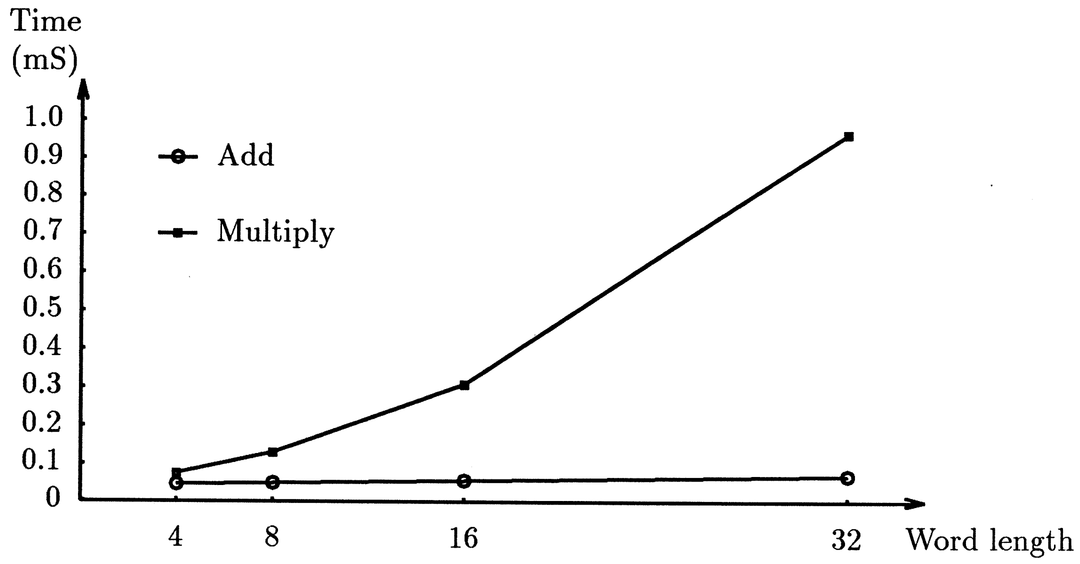

Figure 2: Completion Time for Different Word Lengths.

Table 1: Processor Operations (mS)

	Integer		Floating-point	
	Real time	CM time	Real time	CM time
add	0.139	0.068	0.171	0.067
multiply	0.966	0.965	0.177	0.064
divide	0.722	0.582	0.301	0.154

Communication between the front-end and processors

Table 2 shows three different kinds of communication between the front-end and processors: write, read, and broadcast. For communication between the front-end and one specific processor, data is transferred through the memory bus, one word at a time. Note that either *read* or *write* is one-to-one communication and a processor address always needs to be specified. For one-to-many communications, the front-end can *broadcast* data to all selected processors through the instruction bus without specifying the processor's address. With many-to-one communications, the contents of a parallel variable in all selected processors can be reduced into a value on the front-end, such as reduction with *sum, max, min, and, or, logand*, and *logior*. The completion time of reduction varies with different data types and operations. The results in Table 3 are for 32-bits integer numbers, except that *and* and *or* are for boolean variables. Reduction with *or* is extremely fast because it is carried out by using a global-or wire directly.

Table 2: Communication between the Front-End and Processors (mS)

	Real time	CM time
write	0.181	0.045
read	0.256	0.060
broadcasting	0.478	0.043

Table 3: Reduction from Processors to the Front-End (mS)

	Real time	CM time
sum	0.803	0.613
max,min	1.106	0.212
logand	0.596	0.133
logior	0.261	0.066
and	0.177	0.021
or	0.059	0.005

Communication among processors

Single data transfer, transferring a data item from one processor to another, can prove to be faster than a multiple data transfer. The time for a single data transfer depends on the distance of the two processors. The results of the single data transfer in Table 4 are the average time of different distances. Multiple data transfer is where data items are simultaneously transferred between pairs of processors. Performance here depends on the communication patterns. In Table 4, we use a uniform distribution pattern, in which all processors issue the same amount of communication operations. Here it can be seen that the store operation proves to be faster than the fetch operation.

Many parallel variables can be reduced into several values, named multiple reductions. The *reduction ratio* is the number of parallel variables reduced into a value. Table 5 shows the measured results of *min* reduction of 32-bits integers. The behavior of multiple reduction depends on the physical location of processors in the group, resulting in different communication patterns, such as clustering and distributing. In the clustering pattern, destination processors of reduction are allocated close to each other. In the distributing pattern, destination processors are scattered over the entire network. Figure 3 shows the comparison of multiple reductions for different communication patterns (CM time only). The complete time for the clustering pattern is longer than that for the distributing pattern because the former has more network contention.

Table 4: Simple Communication between Processors (mS)

	Store		Fetch	
	Real time	CM time	Real time	CM time
single data transfer	1.09	0.21	3.04	1.22
multiple data transfer	1.37	0.70	5.25	3.33

Multiple broadcasting means that every group of processors fetchs one variable in a single processor at the same time. Many processors accessing the same place result in *collisions*. There are three different modes to handle such collisions: *collisions-allowed*, *many-collisions* and *backward routing*. With the *collisions-allowed* mode, at most 32 processors can access a single processor. The time required to complete this operation is proportional to *collision ratio*, that is, the number of processors accessing a single processor. This mode outperforms the *many-collisions* mode for a small number of collisions (less

Table 5: Multiple Reduction between Processors (mS)

Reduction ratio	Clustering		Distributing	
	Real time	CM time	Real time	CM time
3	4.80	4.55	1.77	1.46
10	7.79	7.53	1.89	1.58
30	10.38	9.84	2.46	2.15
100	10.82	10.33	2.58	2.27
300	9.83	9.32	2.48	2.16
1000	4.36	4.11	2.46	2.15

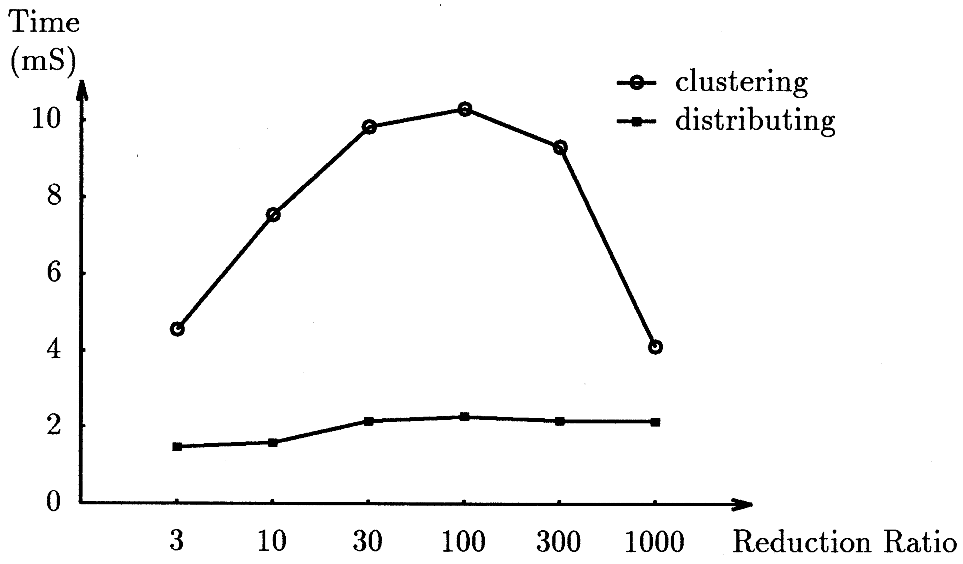

Figure 3: Multiple Reduction between Processors.

Table 6: Multiple Broadcasting between Processors (mS)

(*collisions-allowed*)

Collision ratio	Clustering		Distributing	
	Real time	CM time	Real time	CM time
3	12.0	8.77	5.52	2.37
10	38.9	33.0	13.3	7.17
30	131.0	118.0	37.7	24.2

Table 7: Multiple Broadcasting between Processors (mS)

(*many-collisions*)

Collision ratio	Clustering		Distributing	
	Real time	CM time	Real time	CM time
3	44.5	28.2	45.1	27.5
10	43.8	26.9	45.9	27.5
30	42.6	25.1	45.6	27.2
100	41.4	24.0	43.4	25.3
300	40.2	22.6	43.2	25.1
1000	39.9	22.1	41.8	23.0

Table 8: Multiple Broadcasting between Processors (mS)

(*backward routing*)

Collision ratio	Clustering		Distributing	
	Real time	CM time	Real time	CM time
3	12.8	11.6	4.58	3.31
10	20.3	19.0	4.89	3.61
30	27.0	25.8	6.45	5.16
100	28.3	27.1	6.74	5.47
300	24.7	23.1	6.47	5.47
1000	11.7	10.4	6.43	5.16

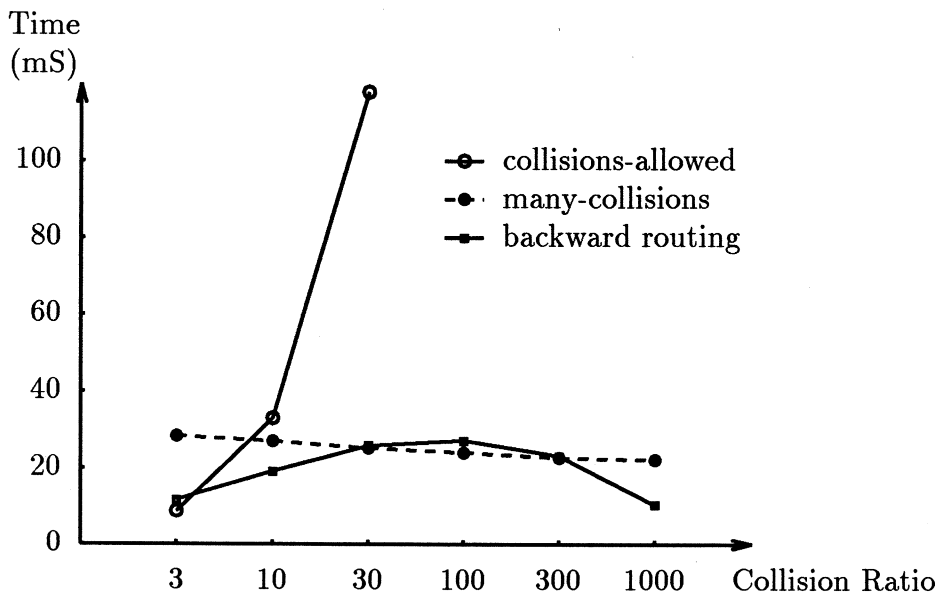

Figure 4: Multiple Broadcasting between Processors (clustering).

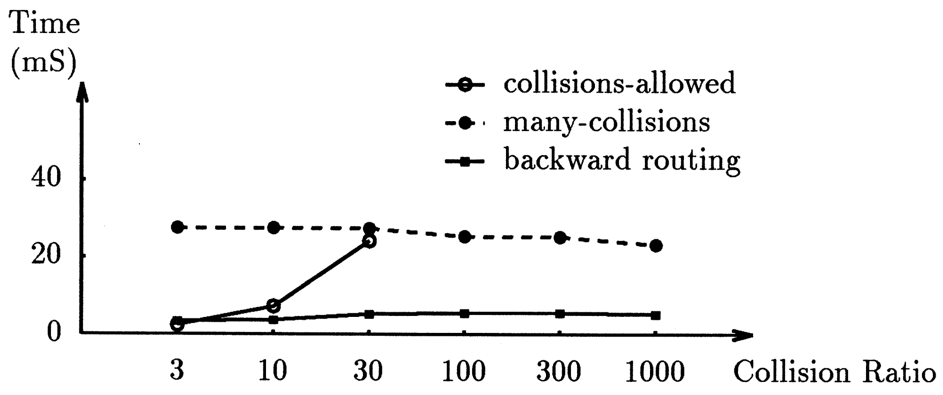

Figure 5: Multiple Broadcasting between Processors (distributing).

than 10 for clustering pattern and less than 30 for distributing pattern). When many processors access a single processor, we prefer the *many-collisions* mode, which almost takes constant time regardless of the collision ratio. *Backward routing* uses a large amount of memory space to trade in the time to deal with collisions. It is faster than the other two modes. However, *backward routing* can be used in limited cases because of memory space demand. Results for the three different modes are shown in Tables 6, 7, and 8. They are measured with the clustering and distributing communication patterns. Processors to be accessed are allocated close to each other in the clustering pattern. And in the distributing pattern, processors to be accessed are scattered over the entire network. The *collision-allowed* mode is sensitive to different communication patterns, whereas, the completion time for the *many-collisions* mode is almost invariant with the communication patterns. Performance of *backward routing* for the clustering pattern is much worse than that for the distributing pattern. Figures 4 and 5 show the comparison of multiple broadcasting for different collision modes and communication patterns (CM time only).

NEWS grid communication

The Connection Machine supports the NEWS (north, east, west, and south) grid system. Grid communication is faster than general interprocessor communication because of regularly structured communication patterns. Version 5.0 supports n-dimensional NEWS grids instead of only two-dimensional grids, where n is any positive integer less than 32. Tables 9 and 10 show the performance of NEWS store and fetch operations for a two-dimensional grid. Figure 6 is the comparison of NEWS store and fetch operations (CM time only). The completion time is increased with the number of communication hops. For example, CM time for NEWS fetch operations can be formulated as follows:

$$\begin{array}{ll} \textit{single transfer:} & 0.140 + 0.063 * h\ (mS) \\ \textit{multiple transfer:} & 0.147 + 0.063 * h\ (mS) \end{array}$$

where h is the number of hops. The completion time for the multiple data transfer is almost the same as that of the single data transfer because communication is well arranged to reduce or eliminate network contention. Notice that the NEWS store operation is unreasonably slow here. It is caused by some implementation abnormality.

Multiple broadcasting and reduction can be carried out on a NEWS grid also. In Table 11, data are broadcasted along the first dimension. Time increases with the number of processors in this dimension. Table 12 shows the completion time for multiple reductions with 32-bits integers. With multiple broadcasting and reduction available, the NEWS grid can be used for those applications which also have some kinds of global communications.

Table 9: NEWS Store Operation (mS)

	Single transfer		Multiple transfer	
Hops	Real time	CM time	Real time	CM time
1	5.564	0.620	5.814	0.639
2	5.811	0.829	6.040	0.850
3	6.017	1.040	6.303	1.060
4	6.345	1.251	6.568	1.271
5	6.609	1.461	6.834	1.481

Virtual Processors

The Connection Machine supports the virtual processor scheme. When the desired number of processors exceeds the number of physical processors, the memory in each physical processor can be logically divided into n portions, and the system processes each directive

Table 10: NEWS Fetch Operation (mS)

Hops	Single transfer		Multiple transfer	
	Real time	CM time	Real time	CM time
1	0.459	0.203	0.679	0.210
2	0.577	0.366	0.780	0.373
3	0.677	0.530	0.877	0.536
4	0.777	0.692	0.976	0.699
5	0.876	0.855	1.094	0.862

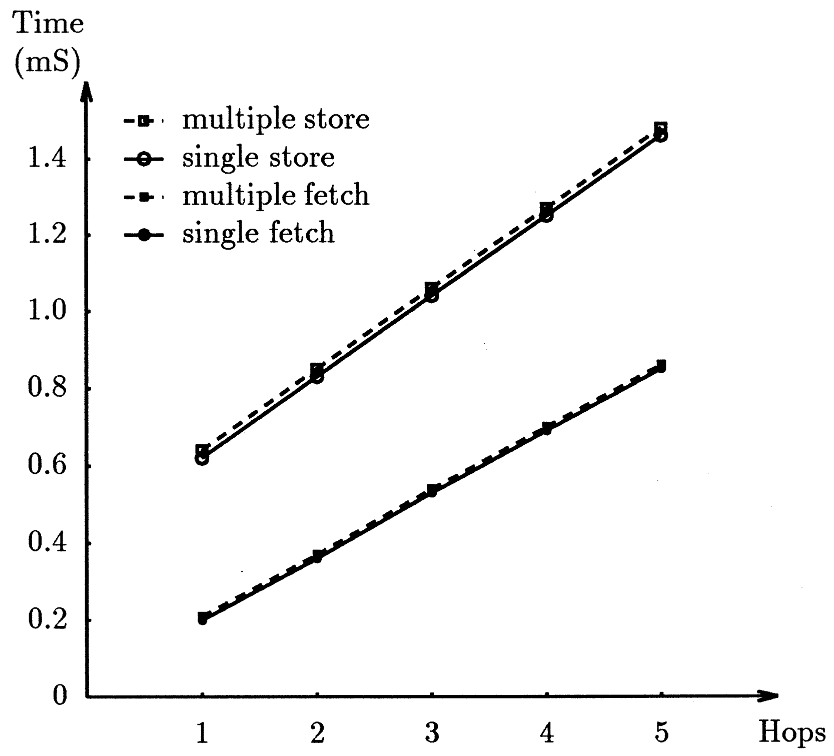

Figure 6: Communications between Processors through NEWS.

Table 11: Multiple Broadcasting on NEWS Grid (mS)

NEWS grid	Real time	CM time
(8,512)	1.866	0.779
(64,64)	2.236	0.897
(512,8)	2.426	0.960
(4096,1)	2.476	1.070

Table 12: Scan Operation on NEWS Grid (mS)

	Real time	CM time
copy	3.13	2.86
sum	1.68	1.62
max,min	1.54	1.49
logand	1.28	1.15
logior	1.19	1.10
and	1.18	0.29
or	1.15	0.29

n times, where n is the VP ratio. For such time-slicing, T_n, the execution time with VP ratio of n should be n times T_1, the execution time with VP ratio of 1. However, T_n is usually shorter than nT_1 because the n portions share a single instruction. Figures 7, 8, and 10 show this fact. However, in Figure 9, T_n is longer than nT_1 because of collisions.

In Figure 7, the completion time of an integer Add operation for different VP ratios is given. As the VP ratio increases, Real time increases slowly. When the VP ratio changes from 1 to 4, Real time increases from 139 microseconds to 191 microseconds. This is due to the fact that the front-end is not fast enough for the low VP ratio. After the VP ratio of 4, Real time and CM time increases linearly with VP ratios. The *sum* reduction from processors to the front-end shown in Figure 8 exhibits different behaviors. Real time and CM time increase slowly when the VP ratio becomes higher. Multiple broadcasting with the clustering pattern and the *collision-allowed* mode for different VP ratios is shown in Figure 9. The network contention becomes serious in this case. Real time and CM time increases fast because of very heavy contention to one physical processor at a high VP ratio. Figure 10 is for NEWS multiple fetch operations (1 hop), and shows the similar characteristics of Figure 7.

Now we compare the Connection Machine to another parallel system, the Intel's iPSC/2 hypercube machine. We have mentioned before that the Connection Machine is an SIMD machine, whereas the iPSC/2 is a multiple instruction multiple data (MIMD) machine. The advantage of the Connection Machine is ease to write a code for application problems. However, the Connection Machine is not so efficient as the iPSC/2. To illustrate that, the Gaussian elimination algorithm is implemented on the Connection Machines in *LISP, and on iPSC/2 in the C programming language. Table 13 compares the performance. From this example, it can be seen that the 8K CM-2 is slower than a 16 processors iPSC/2 machine.

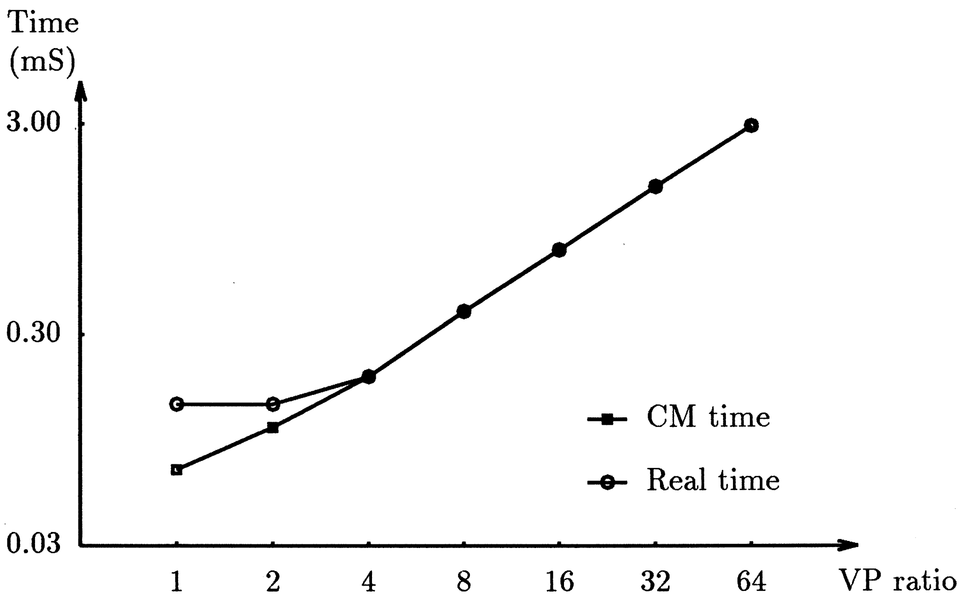

Figure 7: Integer Add for Different VP Ratios.

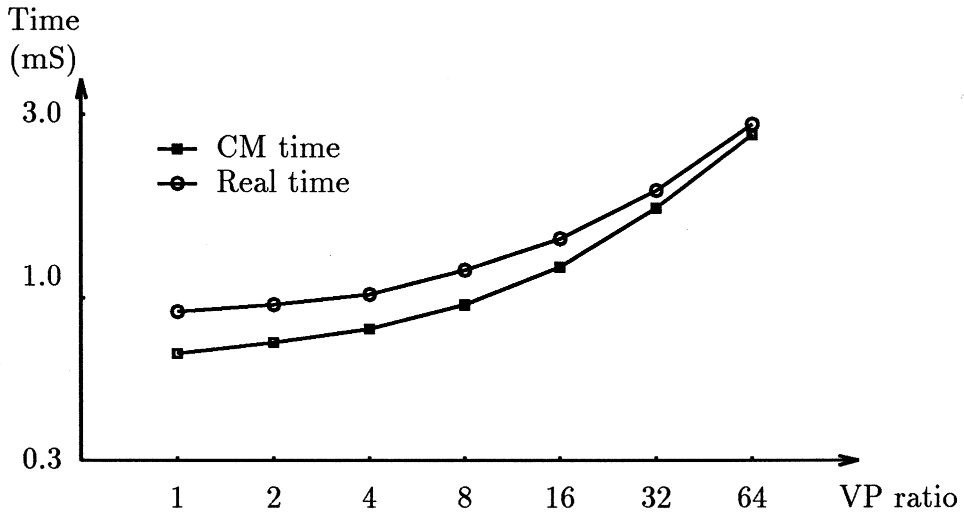

Figure 8: Reduction for Different VP Ratios.

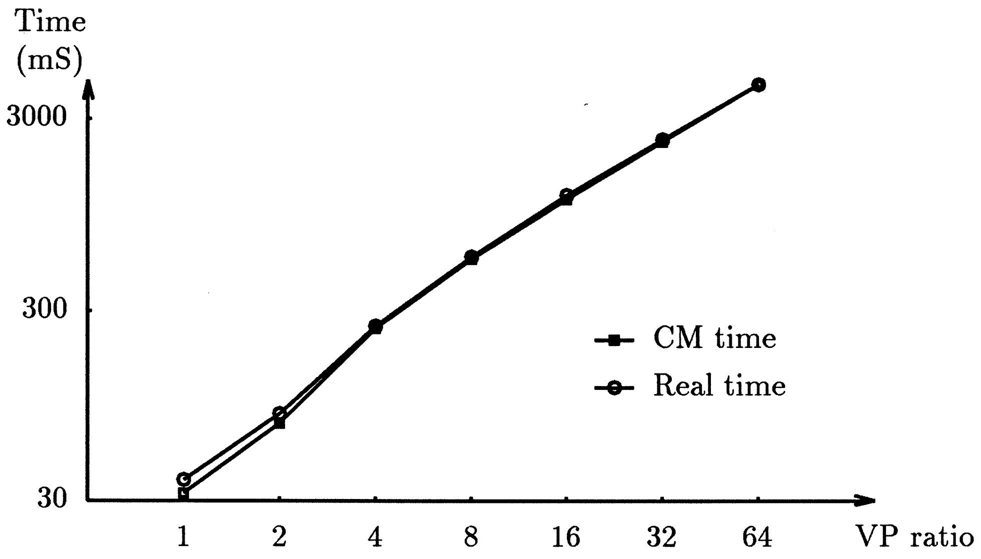

Figure 9: Multiple Broadcasting for Different VP Ratios.

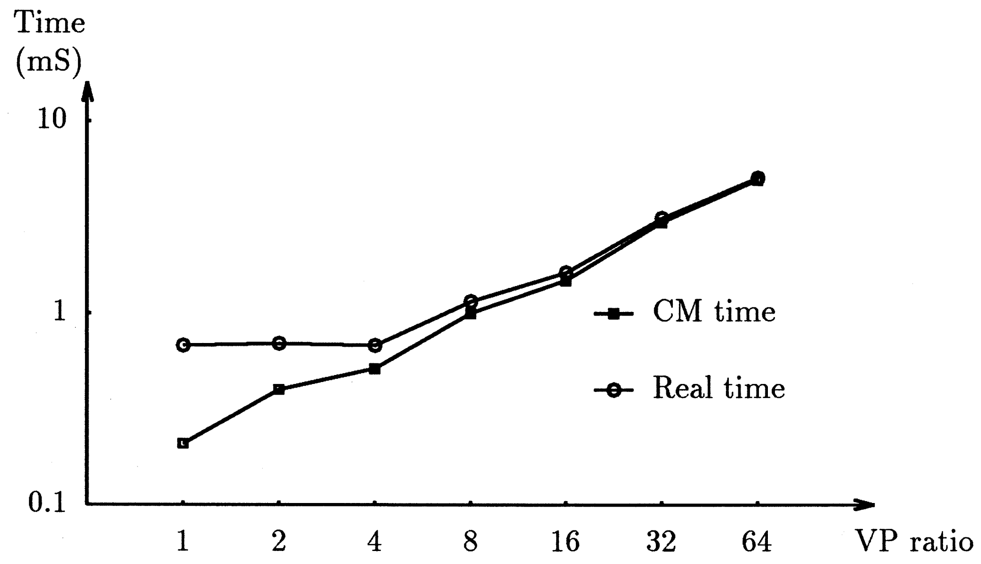

Figure 10: NEWS Multiple Fetch for Different VP Ratios.

We have also implemented the same algorithm in C* on CM-2, and the performance is not as good as *LISP.

Table 13: Comparison of CM-2 and iPSC/2 (Gaussian Elimination)

(*Seconds*)

Matrix size	CM-2 (8K processors)	iPSC/2 (16 processors)
63*64	0.98	0.39
127*128	2.58	1.11
255*256	4.86	4.15

4. PERFORMANCE ANALYSIS

Programming on the Connection Machine involves computation and communication operations. These individual operations, as well as interactions between them, determine the overall performance of the Connection Machine. It will be helpful if we can understand computation and communication properties, and predict the approximate behavior of the program. The performance measurement conducted in the last section provides such possibilities. Based on this measurement, we now suggest some guidelines for programming the Connection Machine.

(1) Communication carried out by busses is fast. Whenever a single element needs to be sent from one processor to another, it is better to let the front-end read the data element from the processor and then write it to the other processor. Directly sending the data element through the router spends more time. The following two *LISP statements can be mutually substituted:

▹ *single fetch:*

```
(*when (=!! (self-address!!) (!! d_addr))
        (*set d_pvar!! (pref!! s_pvar!! (!! s_addr))))
```

▹ *read & write:*

```
(*setf (pref d_pvar!! d_addr) (pref s_pvar!! s_addr))
```

where *d_addr* and *s_addr* are source and destination processor addresses, respectively. Either statement results in transferring the content of the parallel variable *s_pvar*!! from the source processor to *d_pvar*!! at the destination processor. Here, *read & write* could be 6 times faster than *single fetch*.

(2) For general communication, use the store operation instead of the fetch operation. In the next two statements, each processor shifts the parallel variable *pvar*!! towards its 10th succeeding processor.

▹ *fetch:*

```
(*set pvar!! (pref!! pvar!! (-!! (self-address!!) (!! 10))))
```

▹ *store:*

```
(*setf (pref!! pvar!! (+!! (self-address!!) (!! 10))) pvar!!)
```

Notice that if the distance between the source and destination processors is small, NEWS grid communication is preferable.

(3) Use multiple broadcasting very carefully, and avoid to use it whenever possible since it may lead to many collisions. If the memory space is not tight, *backward routing* could be attempted. Otherwise, we can choose the other two modes. When the number of collisions is small, it may take less time for multiple broadcasting by using the *collisions-allowed* mode. However, in many cases, the number of collisions depends on the input data, and cannot be determined before execution. The *many-collisions* mode has to be used for these cases. Also, try to distribute collisions when the *backward routing* or *collisions-allowed* mode is used.

An alternative of multiple broadcasting is to use NEWS *spread!!*, if we can possibly organize processors in an n-dimensional grid and broadcast data along some dimensions. In the following example, we have a two-dimensional array and try to broadcast the parallel variable *common_pvar!!* of each column's first processor down to its corresponding processors at the same column:

▷ *multiple broadcasting*

$$(*set\ \ pvar!!\ \ (pref!!\ \ common\_pvar!!\ \ colNum!!))$$

▷ *NEWS spread*

$$(*set\ \ pvar!!\ \ (spread!!\ \ common\_pvar!!\ \ 0\ \ 0))$$

where for a multiple broadcasting case, each processor fetches *common_pvar!!* according to its own *colNum!!*. With NEWS grid, the first row of the matrix is simply spread along its column direction. It could be about 10 times faster than the previous case.

(4) NEWS communication is efficient because it reduces network traffic and contention. However, it can only be used for regular computation structure and regular communication pattern. It also should not be used if the number of hops traveled is too large. In that case,

general communication must be applied. In the following, we give an example to show that NEWS communication is faster. The first statement with NEWS fetch operation is more than 3 times faster than the second with general fetch operation.

▷ *NEWS fetch*

```
(*set pvar!! (news!! pvar!! 0 1))
```

▷ *multiple fetch*

```
(*set pvar!! (pref!! pvar!! (+!! (self-address!!) (!! rowSize))))
```

In general, following these guidelines will allow the Connection Machine to be used more efficiently. Next, we illustrate how to estimate the performance of a program by using measurement results mentioned in the previous section.

5. PERFORMANCE ESTIMATION

In this section, a method of program performance estimation for the Connection Machine is introdued by an example, Gaussian elimination. The *LISP code in Figure 11 is the major segment of a Gaussian elimination program with partial pivoting. This code involves floating-point arithmetic operations, reduction from processors to the front-end, spread operation on NEWS grid, and some processor selection statements. Table 14 is a portion of the data from our performance measurement, which are collected for estimating this algorithm particularly. The number marked at the end of lines in Figure 11 stands for the corresponding operation in Table 14. Note that not all statements are counted for this estimation because of their relative small impact. The Real time and CM time of one iteration is calculated and listed in Table 14. The execution time of one iteration must be multiplied by the number of iterations to obtain the estimated time. There are 62

```
; N is the matrix dimension
; x!!   holds the value of an N x (N+1) matrix
; factor!!,tmp_pvar!!  are defined as parallel floating-point variables
; rowNum!!,colNum!!    are defined as parallel integer variables
; isActive!!           is  defined as parallel a boolean variable
; pivot_value, pivot_index  are defined as variables at the front-end
   (*set rowNum!!   (self-address-grid!! (!! 0)))
   (*set colNum!!   (self-address-grid!! (!! 1)))
   (*set isActive!! (and!! (<!! rowNum!! (!! N))
                           (<=!! colNum!! (!! N))))
   (dotimes (i (1- N))
       (*when isActive!!                                             (6)
           (*when (=!! colNum!! (!! i))                              (6)
               (setq pivot_value (*max (abs!! x!!)))                 (4)
               (*when (=!! (abs!! x!!) (!! pivot_value))             (6)
                   (setq pivot_index (*min rowNum!!))))              (4)
           (*set tmp_pvar!! (spread!! x!! 0 pivot_index))            (5)
           (*when (=!! colNum!! (!! i))                              (6)
               (*set factor!! (/!! x!! tmp_pvar!!)))                 (3)
           (*set factor!! (spread!! factor!! 1 i))                   (5)
           (*when (=!! rowNum!! (!! pivot_index))                    (6)
               (*set isActive!! nil!!))
           (*when isActive!!                                         (6)
               (*set x!! (-!! x!!                                    (1)
                             (*!! factor!! tmp_pvar!!))))            (2)
       ) ;endof *when
   ) ;endof dotimes
```

Figure 11: Gaussian Elimination Code for Program Estimation.

iterations for matrix size of 63*64, 126 for matrix size of 127*128, and 254 for matrix size of 255*256. This code is running on a CM-2 with 8K processors, and the measured results are compared to the estimated results in Table 15. It is proven that the estimated and real results are fairly close. Taking the advantages of such predictable characteristics, we can be benefited when choosing algorithms and coding programs.

Table 14: Performance Estimation of Gaussian Elimination Code (mS)

		Matrix size	63*64		127*128		255*256	
		VP ratio	1		2		8	
No.	Operation	Ct.	Real time	CM time	Real time	CM time	Real time	CM time
(1)	-!!	1	0.171	0.067	0.166	0.104	0.321	0.320
(2)	*!!	1	0.177	0.064	0.165	0.095	0.284	0.283
(3)	/!!	1	0.031	0.154	0.284	0.266	1.021	1.020
(4)	*max, *min	2	2x 0.384	2x 0.313	2x 0.552	2x 0.491	2x 1.524	2x 1.467
(5)	spread!!	2	2x 2.244	2x 0.897	2x 2.639	2x 1.317	2x 3.379	2x 2.238
(6)	select	6	6x 1.765	6x 0.113	6x 1.765	6x 0.180	6x 1.765	6x 0.650
Time for each iteration			16.50	3.383	17.59	5.161	22.02	12.93

Table 15: Estimated and Measured Results for Gaussian Elimination Code (*Seconds*)

Matrix size	63*64		127*128		255*256	
	Real time	CM time	Real time	CM time	Real time	CM time
Estimated time	1.02	0.21	2.22	0.65	5.59	3.29
Measured time	0.98	0.20	2.58	0.88	4.86	2.66

6. CONCLUSION

The main problem in multiprocessing is not only how to build a system, but also how to use it. That requires development of parallel algorithms and programs that can be executed efficiently. The Connection Machine is a massively parallel system, which potentially delivers high performance. However, it should be used in a prudent way to obtain good performance.

In this paper, we have studied the performance of processors and communication on the Connection Machine. We have shown that communication takes much more time than computation. It has been shown that communication overhead can be reduced by following several programming guidelines. Especially, NEWS grid communication which is efficient should be used whenever possible. Our measurement results are useful to estimate the performance of a program on the Connection Machine.

Acknowledgments

The authors wish to thank Paul E. Oppenheimer of the Thinking Machines Corp. for technical support on the Connection Machine. We also thank Eileen Daily for her editorial efforts, and Michel Jacquemin and Hong Tam for their comments. The generous support of the office of Naval Research under Contract N00014-86-K-0310 is gratefully acknowledged.

References

[Hil85] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[McB88] O. A. McBryan. "The Connection Machine: PDE solution on 65536 processors". *Parallel Computing*, 9(1):1–24, December 1988.

[SS88] R. K. Sato and P. N. Swarztrauber. "Benchmarking the Connection Machine 2". In *Proceedings Supercomputing '88*, pages 304–309, November 1988.

[Thi87a] *C* Reference Manual.* Thinking Machines Corp., version 4.0 edition, August 1987.

[Thi87b] *Connection Machine Model CM-2 Technical Summary.* Technical Report HA87-4, Thinking Machines Corp., April 1987.

[Thi88] **Lisp Reference Manual.* Thinking Machines Corp., version 5.0 edition, September 1988.

[TR88] L. W. Tucker and G. G. Robertson. "Architecture and applications of the Connection Machine". *IEEE Computer*, 21(8):26–38, August 1988.