IN-64-CR

217924

Parallel Matrix Multiplication on the Connection Machine

Walter F. Tichy

November, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.41

NASA Cooperative Agreement Number NCC 2-387

(NASA-CR-185423) PARALLEL MATRIX MULTIPLICATION ON THE CONNECTION MACHINE (Research Inst. for Advanced Computer Science) 42 p CSCL 12A

N89-26627

Unclas
G3/64 0217924

RIACS

Research Institute for Advanced Computer Science

Parallel Matrix Multiplication on the Connection Machine

*Walter F. Tichy**

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.41
November, 1988

Matrix multiplication is a computation and communication intensive problem. Six parallel algorithms for matrix multiplication on the Connection Machine are presented and compared with respect to their performance and processor usage. For n by n matrices, the algorithms have theoretical running times of $0(n^2 \log n)$, $0(n \log n)$, $0(n)$, and $0(\log n)$, and require n, n^2, n^2, and n^3 processors, respectively. With careful attention to communication patterns, the theoretically predicted runtimes can indeed be achieved in practice. The parallel algorithms illustrate the tradeoffs between performance, communication cost, and processor usage.

This paper will appear in the *International Journal of High Speed Computing*, and the *Scientific Applications of the Connection Machine*, World Scientific Publishing, June 1989

Work herein is supported by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

*Walter F. Tichy is located at the University of Karlsruhe, FRG. Work performed while visiting at RIACS and performed at the University of Karlsruhe, FRG.

1 Introduction

Matrix multiplication is a conceptually simple problem that is computation and communication intensive. It is computation intensive, because multiplying two n by n matrices takes $O(n^3)$ arithmetic operations. It is communication intensive, because $O(n^3)$ different pairs of operands have to be brought together during the computation. Communication time is not an issue for a single processor that fetches one operand at a time from memory. On multiprocessors, however, data and intermediate results are distributed, and the communication delays for bringing them together for processing are not negligible. Studying parallel matrix multiplication has the advantage of exposing the tradeoffs among various degrees of parallelism, performance, and communication patterns on a simple and easily understood problem.

The Connection Machine (CM) is well suited for experimenting with large-scale parallelism. The CM model 2 is an SIMD computer with up to 65,536 (2^{16}) processors connected in a 16-dimensional hypercube network. Each processor has 8 kilobytes of local memory; the entire primary memory of the CM comprises half a gigabyte. A parallel I/O system matches the speed of the processors. The CM is the first, and so far only, computer that provides enough processors to credibly run large-scale parallel programs. Although the processors are bit-serial and therefore quite weak, their large total number permits realistic experimentation with massive parallelism.

The high connectivity of the hypercube network also contributes significantly to experimenting with parallel algorithms, because the hypercube can efficiently implement the routing patterns of many important regular and irregular communication topologies. Regular topologies that can be embedded efficiently include rings, multi-dimensional grids and tori, multigrids, trees, and the perfect shuffle. A total of 4048 routing processors (1 routing processor is shared by 16 of the regular processors)

handle irregular communication patterns. These special-purpose processors compute communication paths, store and forward messages, and manage contention. They make parallel communication among processors as easy to program as array indexing.

The simultaneous availability of many communication patterns is important, since many algorithms change communication patterns while running. Our matrix multiplication algorithms use trees, forests, grids, tori, and the perfect shuffle. Half of our algorithms actually need at least two patterns out of this set. The high connectivity of the hypercube allows efficient implementations of all relevant algorithms, and thus allows a fair comparison.

An important aspect that simplifies parallel programming is the concept of virtual processors. Each of the 2^{16} physical processors can simulate a number of virtual processors. The maximum number of virtual processors is limited by the available memory, because the virtual processors need to share it. The firmware of the CM efficiently implements the time-multiplexing of the physical processors, in a manner that is transparent to the programmer. Using virtual processors means that programs are independent of the physical processors available. Transparency is important for scalability: in order to solve a larger problem, one simply changes the relevant constants and reruns the program. For more information on the CM, see references [1,2].

The following section describes the algorithms. Section 3 presents the performance results.

2 Algorithms

The algorithms for parallel matrix multiplication are interesting in their own right, but the reader less interested in the details can skim this section.

2.1 Ground rules

There are a number of parameters that affect the performance of the parallel algorithms. In order to arrive at meaningful comparisons, some of these parameters must be kept constant.

All algorithms are for dense, rectangular arrays:

$$C^{l,n} = A^{l,m} \times B^{m,n}$$

The usual specification of matrix multiplication is:

$$\forall i,j \mid 0 <= i < l,\ 0 <= j < n : C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$

Our goal is to compare algorithms that work for arrays of any size. This means that the algorithms do not take advantage of square matrices, nor are they limited to preferred array sizes, such as powers of 2. The algorithms do not pad the matrices with dummy elements to make them square or to adapt them to preferred sizes. The only padding we allow in some of the algorithms is to reserve the same amount of space, namely $\max(l,m) \times \max(m,n)$ for all arrays. The space wasted by this type of padding cannot be put to other uses anyway, given the architecture of the CM. For instance, by attempting to save this padding, the maximum size of arrays that can be run on the CM would not increase. Furthermore, the algorithms do not introduce dummy processors except proportional to the padding just discussed. Thus, the algorithms waste neither space nor processors frivolously.

By leaving out special-casing for array sizes, we exclude some of the faster algorithms. For example, a fast $O(n)$ systolic algorithm requires square matrices. However, by insisting on generality we obtain programs that run efficiently and without waste, independent of whether the matrices are square, skinny, or fat.

For brevity, however, section headings quote performance characteristics for square arrays of size $n \times n$; we leave it as an exercise to the reader to derive the precise formulas for general, rectangular arrays.

A further aspect regarding generality is that all input and output matrices for a given algorithm are allocated in the same manner, for example row-major. We do *not* assume that matrices A or B are transposed beforehand to speed up communication. If a matrix must be transposed or brought into some other special alignment, then our algorithms include the necessary steps, and our measurements include the time consumed by these steps.

All algorithms use double precision floating point.

All algorithms are implemented in C*, except the sequential one, which is written in C. Reprogramming the C* programs in *Lisp would probably make them run faster.

All algorithms use the general CM communication; they do not take advantage of the grid communication or reduction operations in CM microcode. Most algorithms would run faster by using these features. Programming with some of these features is difficult in C*.

2.2 The $O(n^3)$ (sequential) algorithm

The sequential algorithm is the standard textbook version with three nested loops, highly optimized. In particular, code motion and strength reduction (replacing multiplication with addition) make address computations fast. Registers hold indices, offsets, and temporary results. This algorithm does not use the CM at all, because it runs completely on the frontend. It is included for comparison purposes.

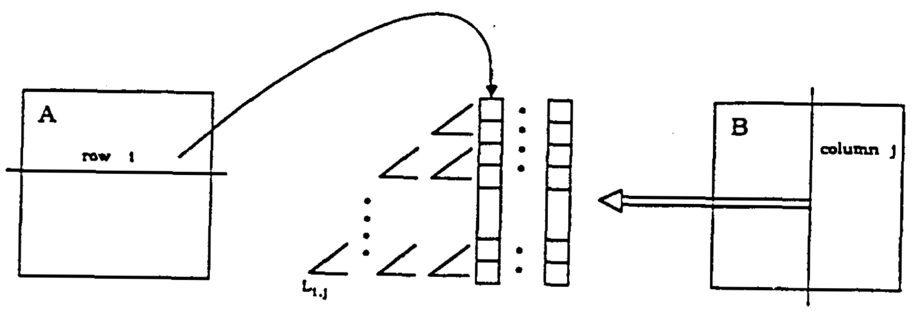

Figure 1: The $O(n^2 \log n)$ algorithm

2.3 The $O(n^2 \log n)$ algorithm

In this algorithm, each processor contains a row of each of the matrices A, B, and C. Thus, we need $\max(l, m)$ processors. The algorithm computes $l \times n$ inner products in sequence, one for each element of C. An inner product executes m multiplications in parallel in constant time, and then a parallel sum reduction to produce the sum in $O(\log m)$ time. For details regarding parallel sum reduction see reference [3].

The communication costs of this algorithm are as follows. Since arrays are stored one row per processor, a single column of B is spread over m processors. For the parallel multiplication in the inner product, each row of A must be spread over the m processors containing the columns. The spreading of a row takes $O(m)$ communication steps, since a processor can send only one element at a time. However, a given row of A must be spread only once for computing n inner products. The multiplications require no communication. The logarithmic sum reduction uses the router with the reduction into the target coefficient of C. This step takes $O(\log m)$ communication steps. See Fig. 1 for an illustration.

2.4 The slow $O(n \log n)$ algorithm

This algorithm is a straight-forward generalization of the previous one. Instead of using m processors to perform a single, parallel inner product step at a time, we use $l \times m$ processors to produce an entire column of C at once.

Each processor contains one element of A, B, and C. Assume that matrix elements are assigned in row-major order to the processors (see Fig 2).

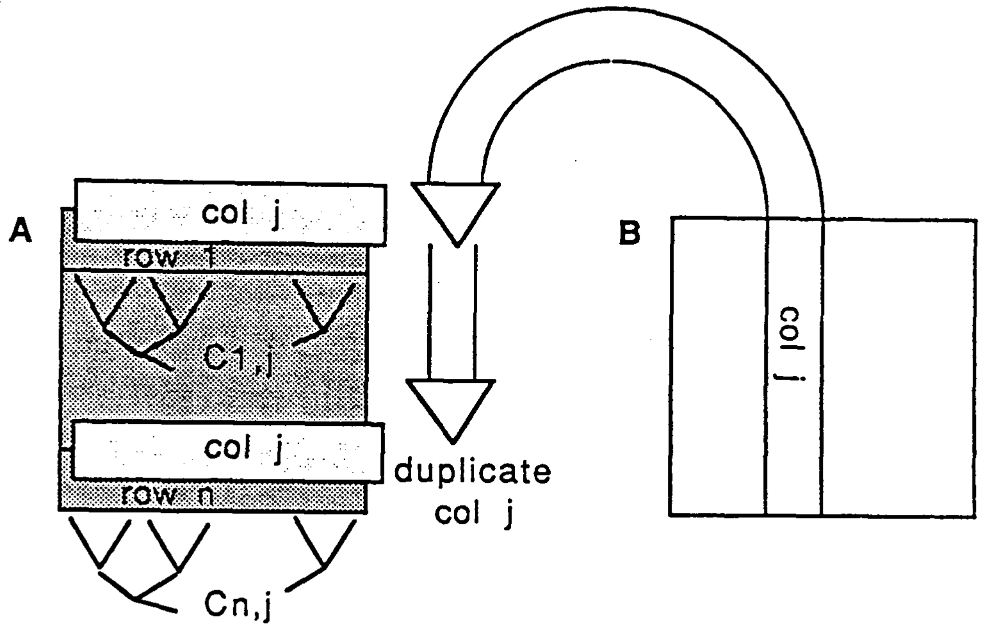

Figure 2: The slow $O(n \log n)$ algorithm

For computing a column of C at once, we first need to broadcast the relevant column of B to all rows of A, then execute $l \times m$ multiplications in parallel, and then perform l sum reductions in parallel. The multiplications take constant time and the sum reductions $O(\log m)$ time, including communication. The broadcast of columns of B must be arranged carefully as follows.

There are several alternatives for implementing the broadcast on the CM. First, the frontend computer is highly efficient at broadcast: It could first retrieve a column element from B and then pass it simultaneously to all elements of A. However, the frontend can only broadcast one element at a time, so the process would take $O(m)$ steps, resulting in quadratic runtime overall. The second alternative is to exploit the router for performing an implicit broadcast: Each element of A could simply retrieve the required coefficient from B directly. Unfortunately, this would again result in linear broadcast time, since each element of B's column would be requested by l rows of A simultaneously, and each processor can only honor one request at a time. Instead, we must program a fanout tree for each column element. A fanout tree has the same structure as a tree for reduction, except that data flows from the root to the leaves instead of vice versa. Since we need a tree for each column element, we actually need to construct a parallel fanout forest. The fanout forest broadcasts B's column in logarithmic time.

In detail, the fanout forest operates as follows. First, we "seed" the column of B into the first row of A, in parallel for all elements. Next, we instruct the first row of A to duplicate the seeded coefficients one row down, also in parallel. Next, the first 2 rows of A to duplicate their elements 2 rows down, then the first 4 rows duplicate 4 rows down, and so on. In each step, the number of copies of the column of B doubles, until the entire matrix A is filled. This process takes $O(\log l)$ communication steps. We programmed the broadcast explicitly, although the CM actually provides a primitive for it. Using the primitive instead would significantly speed up that portion of the program.

Another detail concerns the sum-reductions. The l sum-reductions run along the rows of A, orthogonal to the broadcast. Again, we programmed this process directly rather than using the corresponding CM primitive. Using this primitive also speeds up the program.

2.5 The fast $O(n \log n)$ algorithm

When analyzing the broadcast operation in the previous algorithm, one notices that it uses processors and communication bandwidths poorly. In the first step of the broadcast, only m of the $l \times m$ processors operate, and in the last one, barely half of the processors operate. The algorithm in this section eliminates the slow broadcast altogether.

As before, we lay out matrices one element per processor, in row-major order. Figure 3 illustrates. First, we transpose matrix B and overlay it onto A; the router performs this operation in constant time. The transposed overlay has the effect that row i of A is lined up with column i of B. Next, we perform l parallel inner product steps, producing the main diagonal of C in $O(\log m)$ time. As the next step, we rotate the transposed matrix B up one row, with the topmost row reentering at the bottom. Now row i of A is lined up with column $(i+1) \bmod n$, and we compute the upper main diagonal of C, along with element $C_{l-1,0}$. After n steps of inner product computation and rotation, C is complete.

This algorithm uses the bandwidth of the communication network and the processors effectively. Again, all communication is implemented directly, rather than using CM primitives.

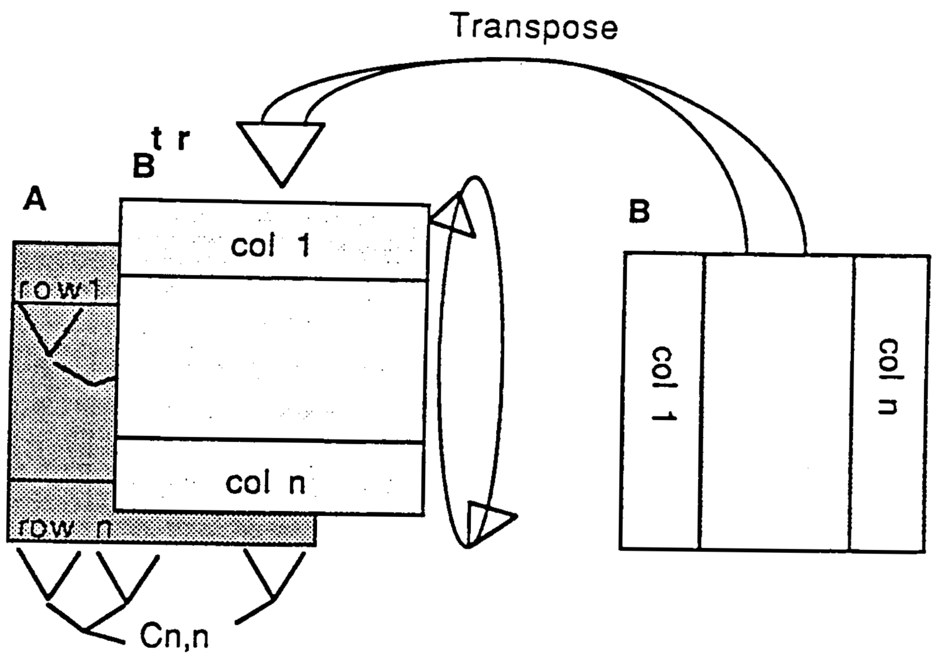

Figure 3: The fast $O(n \log n)$ algorithm

A detail involves the relative sizes of A and B: In general, the number of rows of A does not match the number of columns of B, so the transposition and rotations must be done with care. Our approach was to keep the smaller of the two arrays in place and rotate the larger, although the opposite might improve performance somewhat.

2.6 The slow $O(n)$ algorithm

A problem with the previous algorithms is that they all perform logarithmic sum reduction. The algorithm in this section avoids the corresponding factor of $O(\log n)$ by distributing the cost of the addition over the communication, and thus achieves linear runtime.

The algorithm is called "systolic", because it alternates between two distinct phases, a communication phase and a computation phase. Assume we have $l \times n$ processors, each computing one element of the result matrix C. The initialization sets C to zero. Rows of A enter C from the left and shift horizontally through C. Similarly, columns of B enter C from the top

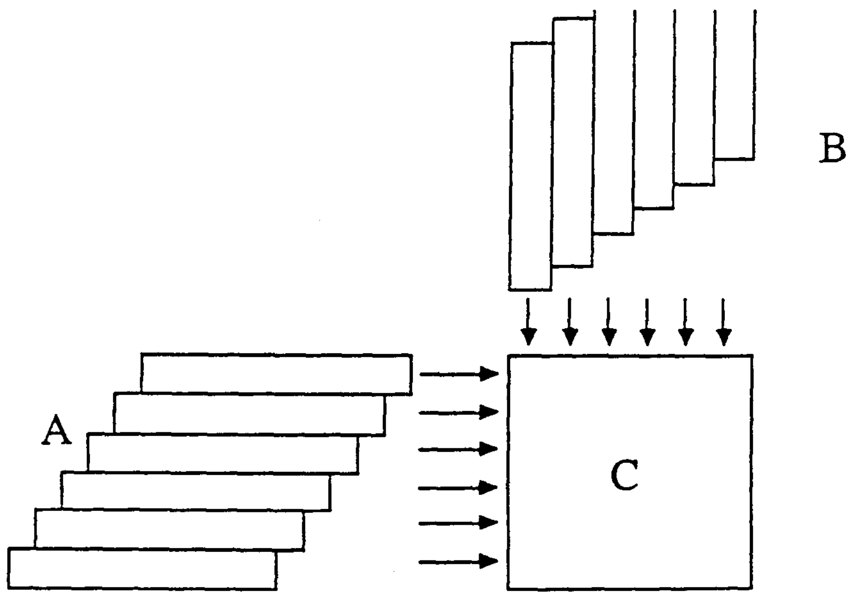

Figure 4: The slow $O(n)$ algorithm

and shift down (compare Figure 4). The shifts transport a new column and row coefficient to elements of C in each step. The computation phase multiplies these coefficients and adds them to $C_{i,j}$. For the coefficients to line up in the right order, the rows and columns enter C in skewed order, i.e., they are delayed from entering by one step per row or column. Processors without coefficients in a particular step are simply disabled. Figure 5 shows the resulting configuration after 3 shifts. For more details on the systolic algorithms, see reference [4], Chapter 8 or [5].

The algorithm operates for a total of $O(l + m + n)$ steps, until the last row and last column of A and B have shifted through C. Each step takes constant time.

We experimented with two variants of the systolic algorithm. The first variant treats the first row and column of C as a special case: Elements in these positions retrieve the relevant coefficients from A and B directly, rather than moving A and B into place. The remaining elements of C retrieve their coefficients from their north and west neighbors. Thus, the first row and column of C inject the coefficients. Unfortunately, this approach does not

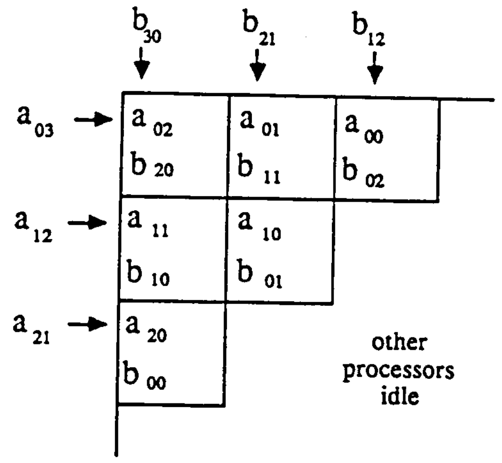

Figure 5: Configuration of C in Figure 4 after 3 shifts

lead to the most efficient implementation. Since the Connection Machine is SIMD, the injection and the shifts cannot occur simultaneously: First, the interior elements retrieve from the north and west, and then the leading row and column of C send for their next elements. Thus, the communication phase has two subphases, each of which idles a significant portion of the processors.

We therefore modified the algorithm, realizing that the shifts are only an artifact of the topology of systolic processor arrays. With general communication on the CM, any processor can retrieve data from any other processor in essentially the same time. Thus, during each communication phase, every element of C retrieves the required coefficients from A and B directly. The address computation is identical for all elements and splitting the communication phase into subphases is avoided. Since no two elements of C retrieve the same coefficients, there are no problems with fanout as in Section 2.4. In section 3, we report only on the faster of the two variants.

2.7 The fast $O(n)$ algorithm

A flaw of the previous algorithm is that even the faster variant underutilizes the processors. The activation of processors spreads from the north-west corner towards the south-east corner, with never more than half the processors busy. On average, processor utilization is only 1/3. How can we keep

$K^{(1)}_{i,j}$

0	1	2	3
1	2	3	0
2	3	0	1
3	0	1	2

Figure 6: Initial indices for the fast $O(n)$ algorithm

all processors busy all the time?

The answer to this question derives from the fact that the inner products for the elements of C need not be computed in the same order. Consider

$$C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$

Because of the commutativity of the addition, there is no need to accumulate the sum starting with $k = 0$. Instead, we could start with any $k_{i,j}$ in $[0 \ldots m-1]$, sum to $m-1$, then "wrap around" and add the the terms from 0 to $k_{i,j} - 1$. Observe furthermore, that in each of the $l \times n$ multiplications we must make sure that no two use the same coefficients from A or B, because we would have a slowdown caused by fanout otherwise. We can exploit the commutativity of the addition to achieve this separation. One way to use different coefficients everywhere is to let $k_{i,j} = (i + j) \bmod m$. In other words, the starting index for building the inner product is skewed for each row of A and column of B, guaranteeing that no element of A or B is used twice in a single inner product step. Figure 6 illustrates the starting assignment of $k_{i,j}$ in a 4×4 array.

For square matrices, this approach is equivalent to overlaying A, B, and C, skewing the rows of A and the columns of B with wrap-around, and then rotating the rows of A and the columns of B during each step. The rotation does not work well for non-square matrices, and since we did not use the general grid addressing on the CM, rotation has no advantage over

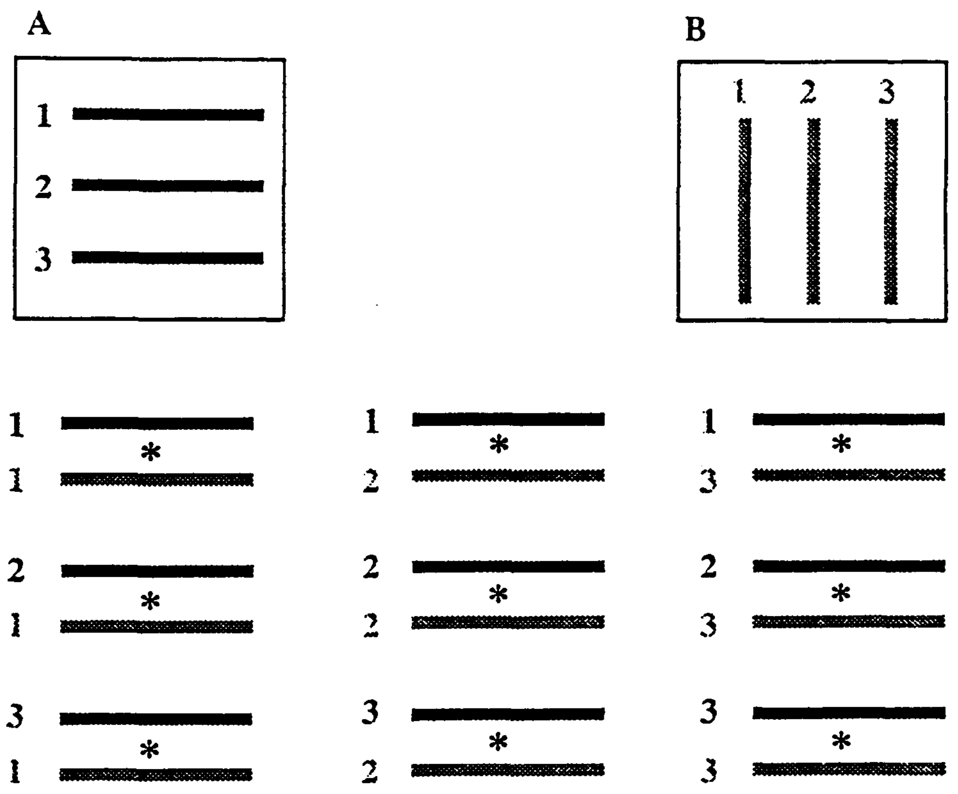

Figure 7: The $O(\log n)$ algorithm

general communication. We therefore simply used general communication to retrieve the required coefficients directly from A and B. The resulting program is actually the simplest of all those considered here. It consists of a loop over m, with two statements in its body: one for the inner product step, and one for incrementing $k_{i,j}$ modulo m.

2.8 The $O(\log n)$ algorithm

The fastest algorithm is one that uses n^3 processors to compute all n^3 products simultaneously, and then performs n^2 sum reductions in parallel to produce C. The multiplication takes constant and the sum reduction logarithmic time. We also must take into account the duplication and alignment of data prior to the multiplication. Each row of A and each column of B must be duplicated n and l times, respectively, and paired properly one with the other. Two fanout forests, one for A and one for B, broadcast and align the data in logarithmic time. Figure 7 illustrates the pairings of rows and columns for 3×3 matrices.

3 Performance Results

We implemented all algorithms in C* and timed them on a Connection Machine model 2 with 32K processors (system version 5.0, field test), without floating point chips. As stated, all arithmetic (except address calculations) used *double-precision floating point*. The frontend controlling the CM was a DEC VAX under the ULTRIX operating system.

Since the speed of the VAX is often no match for the CM, we report timings measured only on the CM. Although this may be unrealistic for the given configuration, we believe this choice is justified for the following reasons. When the virtual to physical processor ratio is 1, the frontend time is typically more than twice as high as the CM time, with a CM utilization of less than 50 per cent. This indicates that the VAX cannot keep up with the CM, mainly because of its raw MIPS rating, but also because the frontend is timeshared among other users. When the virtual to physical processor ratio is high, such as 8, then the times on the frontend and the CM are nearly identical (for both elapsed times and the combined system and user times). This is an indication that the simulation of virtual processors is slowing the CM enough to match the speed of the VAX.[1] We are therefore convinced that in this situation, the timings on the CM are more accurate. For a well-matched, faster frontend, such as a Symbolics Lisp machine or a SUN/4, timings would best be taken on the frontend.

For simplicity, all measurements were run with square matrices. The results are summarized in Figures 8 and 9. Figure 8 shows all 7 curves for array sizes of up to 250×250. We shall discuss the curves clockwise, starting form the top left. The leftmost curve represents the $O(n^2 \log n)$ algorithm, using n processors. This algorithm appears slower than even the sequential, $O(n^3)$ algorithm (second curve). This is not surprising, since the difference between $\log n$ and n is not enough to offset the difference between a 1-bit processor and a 32-bit processor for the small values of n shown. Note, however, that the two curves will eventually cross, since the first is of a lower order. We estimate that the crossover point is $n \approx 450$.

[1]Note that the virtual processor simulation is performed by the microcode of the CM: Each instruction issued by the frontend is repeated implicitly by the CM's instruction decoder for the number of virtual processors assigned to the physical processors.

ORIGINAL PAGE IS
OF POOR QUALITY

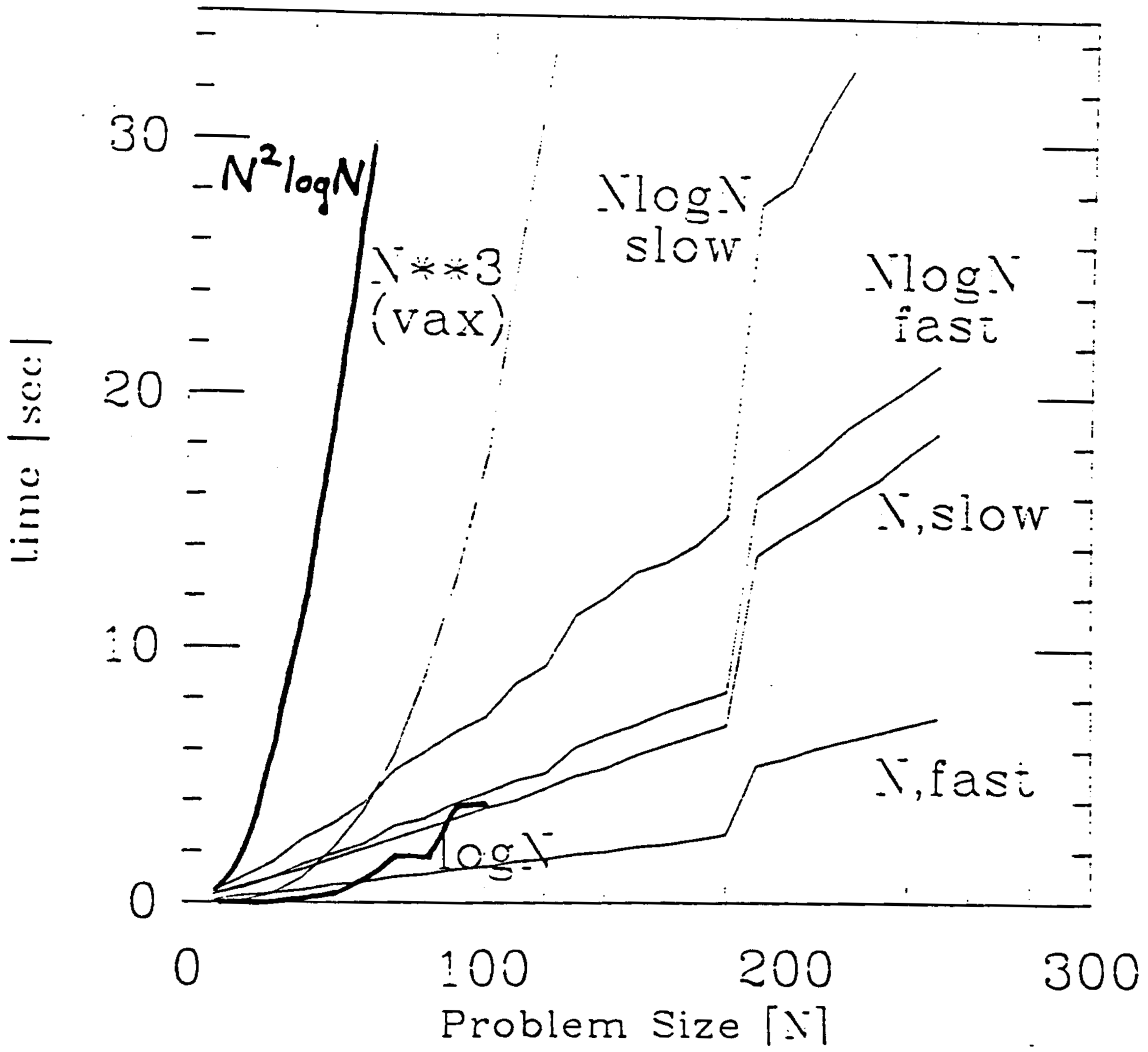

Figure 8: CM performance for moderate array sizes

Using n^2 processors boosts performance far beyond that of the sequential processor. The crossover points occur early, clearly demonstrating that slow, but numerous processors can outperform a single, fast sequential processor. The fast $O(n \log n)$ algorithm is almost twice as fast as the slower variant, demonstrating the significance of communication costs. The faster algorithm is almost as fast as the slow linear algorithm. The second linear algorithm is another factor of 2 faster, demonstrating the effect of full utilization of processor and communication bandwidth.

A curious effect is the jump in these 4 curves, occurring for a problem size of about 180. Note that at this point, 32,400 processors are in use, which is the capacity of the CM available (32,768 processors). An increase of the problem size beyond 181 requires a virtual to physical processor ratio of 2. Thus, execution times double, and the gradient of the curve doubles also. The next such doubling would occur for $n > 256$, and then again for $n > 362$. On a full connection machine with 65,536 processors, the first jump would not occur until $n > 256$.

By doubling the virtual processor ratio, the times do not quite double, since communication between virtual processors simulated on the same physical processor is more efficient than communication among separate physical processors. However, the savings observable are minor. In the case of the fastest linear algorithm, increasing the virtual to physical processor ratio from 1 to 2 increased the time by a factor of about 1.93 rather than 2. Thus, for matrix multiplication, the savings are only 7 per cent.

The last curve is for the $O(\log n)$ algorithm, which requires n^3 processors. This algorithm is the fastest for small problem sizes, but requires a large number of processors. At problem size 32, all processors of the available CM are in use. For this size, the algorithm simply brings all the available hardware to bear on the problem. For a problem size of 60, the overhead of virtualization is so high as to slow the program down below the fast linear algorithm. (Note that the virtualization overhead grows proportional to $n^3/2^{15}$.) For a problem size of 100, each real processor must simulate 16 virtual processors. At this point, all the available Connection Machine memory of 256 Mbytes is used up by replicated array data and virtual processor stacks. Larger problems simply do not fit the capacity of the 32K processor CM.

The effect of the virtual to physical processor ratio for some of the less processor-intensive problems is shown in Figure 9. For larger problem sizes, the cubic behavior of the matrix multiplication cannot be denied. The performance of the "linear" algorithm is still cubic, once the number of processors is exhausted. Essentially, the performance curve is a cubic parabola, divided by the large constant factor of 32,000. The straight line at the bottom is the virtual time of each processor. This is the time we would see if we had as many processors as memory words.

Note that the absolute performance achieved is hardly overwhelming. Counting only double-precision floating point additions and multiplications, the fast sequential algorithm achieves only about 4 Mflops for 180×180 arrays; this number would double for a full (64K processor) CM-2. Including arithmetic instructions for address calculations, we come up with about 20

Mips, or 40Mips for a full CM-2. (This number excludes the instructions for stack manipulation and communication). Thus, we achieved only about 1/60 of the "typical application performance" quoted by Thinking Machines Corporation for general computing. Apparently, our algorithms are communication bound, and using the special features of the communication network would pay significant dividends. The numbers also demonstrate how difficult it is with the present programming languages to harness the power of the CM.

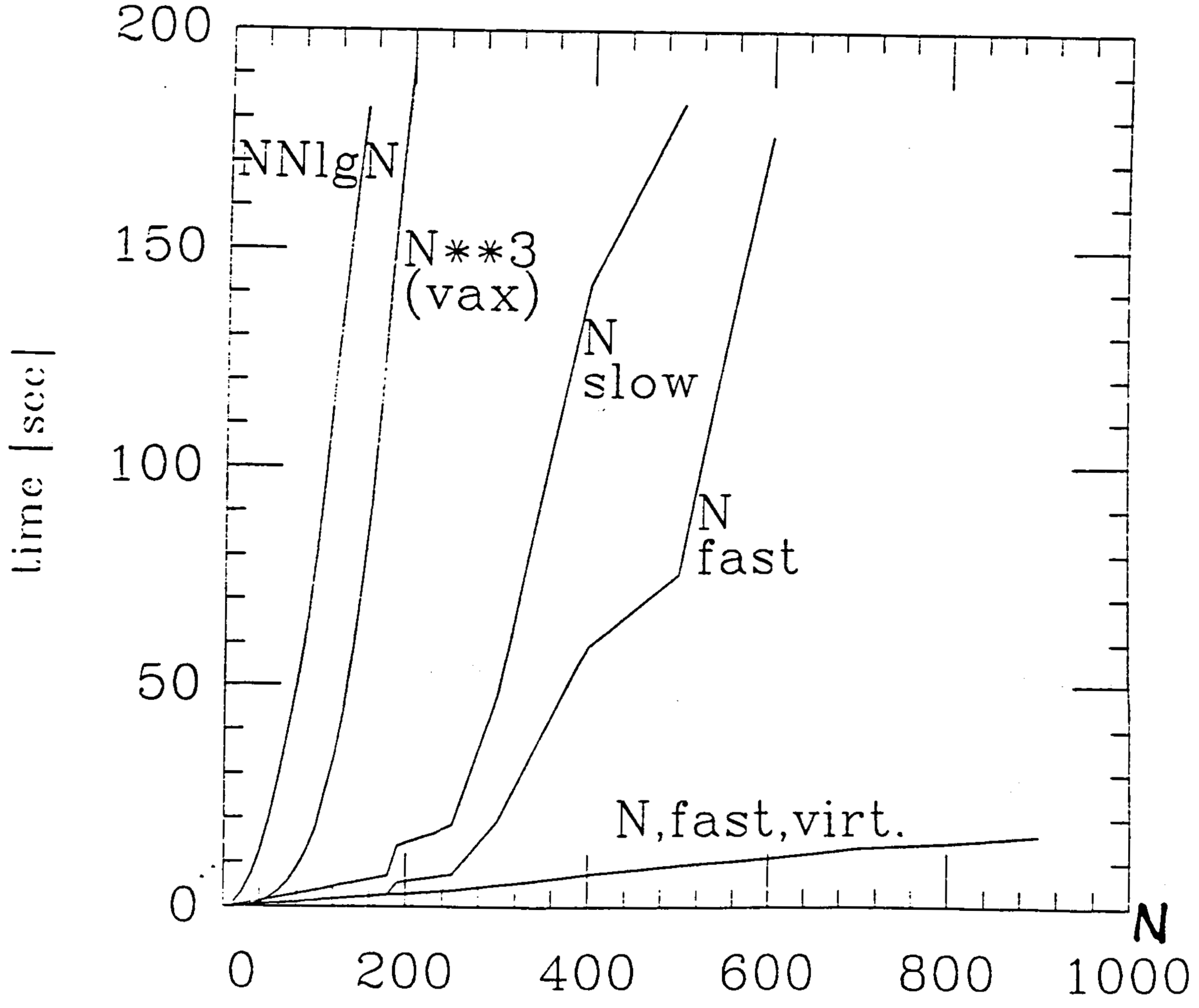

Figure 9: CM performance for large array sizes

The measurements in Figure 8 were taken in increments of 10, so the curves are quite accurate. Each individual point was computed by repeating the same problem long enough to have a cumulative runtime of between 2 and 4 minutes, and then dividing by the number of actual runs. The variance in time for the individual runs was so small as to be invisible on the diagram. For Figure 9, we took measurements in increments of 50, with increments of 10 around the points were virtual to physical processor ratios change. Some of the higher points in this diagram represent the average of only a few runs.

4 Conclusion

A number of conclusions can be drawn from this case study. First, even for problems as simple as matrix multiplication, a surprisingly varied number of different algorithms exists, and the tradeoffs among speed, communication patterns, and processor usage are interesting and non-trivial. It appears that with large-scale parallelism, all of our sequential algorithms must be rethought. CM programmers have already discovered some new and interesting, totally parallel solutions for many problems, from multi-grid methods to document retrieval to ray tracing. Furthermore, we predict that many of our sequential algorithms will turn out to be special cases of parallel ones.

A second important insight is that with the right choice of algorithm and communication pattern, the speedup attainable is indeed proportional to the number of processors used. With few exceptions, all previous experiments with multiprocessors showed a point of diminishing and even reversing returns, when the addition of processors did not speed up a program proportionally or even slowed it down. At no time did we observe these effects on the CM; performance was always within a constant factor of the theoretically predicted, asymptotic performance. We suspect that earlier multiprocessors simply had insufficient communication bandwidth and high synchronization overhead. Because of the SIMD nature of the CM, there is no synchronization overhead, and the bandwidth of the hypercube is well matched to the demands that the processors can generate.

We can also confirm that the concept of the virtual processor is a great simplification for parallel programming. Not having to write twisted code for mapping a given problem onto a particular set of processors makes for easily written, easily understood, and easily ported programs. Further study is required to make this concept applicable when programs need to change the number of virtual processors dynamically.

There are also a number of negative conclusions. First, using a superlinear polynomial of processors severely limits the problem size, and the resulting program may not run efficiently because of the overhead of virtualization. In our example, using one processor per data element yielded the best overall performance. However, for small problem sizes, a superlinear number of processors is the best way to bring the entire available hardware to bear on a problem.

Second, it became quite clear that automatically transforming "dusty deck" sequential problems to large-scale parallel ones is a pipedream. Considering matrix multiplication, it is easy to see how a compiler would detect the inner loop of the sequential program and transform it into a vector operation. However, we severly doubt whether a general compiler could be built that could generate all six variants we discussed from a single, sequential program. If automatic transformation can be done at all, it would have to start with the problem specification and not with a sequential implementation. In a sequential program, too many opportunities for parallelism have been hidden or eliminated.

A number of further studies should be done to get a better grasp of the idiosyncrasies of the Connection Machine. First, all programs should be rewritten in *Lisp, to compare the quality of the two two language implementations and the effect of the frontend. Second, to quantify the potential gains from the special features of the router, all programs should be modified to use them. Preliminary experiments have shown that by using just the reduction operators, the $O(n \log n)$ algorithms run almost as fast as the corresponding linear algorithms. Of course, the linear algorithms could also be improved by using grid addressing. Finally, the ratio of communication time to computation time should be determined by simply leaving out the floating point operations. It appears that all our implementations are communication bound and that floating point operations actually consume a negligible percentage of the time. Matrix multiplication shares this property with many other problems. Perhaps communication cost will turn out to be the dominant cost for all large-scale parallel algorithms.

References

[1] L. W. Trucker and G. B. Robertson, "Architecture and applications of the connection machine," *IEEE Computer*, vol. 21, pp. 26–38, August 1988.

[2] W. D. Hillis, *The Connection Machine.* Cambridge, Mass: The MIT Press, 1985.

[3] W. D. Hillis and G. L. Steele Jr., "Data parallel algorithms," *Communications of the ACM*, vol. 29, pp. 1170–1183, December 1986.

[4] C. Mead and L. Conway, *Introduction to VLSI systems.* Addison-Wesley Publishing Company, 1980.

[5] S. Kung, "VLSI array processors," *IEEE ASSP Magazine*, pp. 4–22, July 1985.

A General remarks about the programs

All programs have a macro called DEBUG. When this macro is defined, either in the program directly, or via the -D option on the cc or cs command line, then detailed tracing information about the matrices will be printed.

With the exception of the sequential algorithm, array dimensions are compiled into the programs. By using constants rather than variables, the programs run about 10 percent faster on the CM. There is no noticeable difference for the sequential algorithm.

For CM programs, the macros L, M, and N determine the dimensions of the arrays as follows:

Matrix A: L by M
Matrix B: M by N
Matrix C: L by N

If the macro DEBUG is defined, L, M, and N are already predefined (to 3, 4, and 5, respectively). Otherwise, the macros L, M, and N must be either defined in the program itself, or on the command line. To compile a CM program in file f.cs, one would use the following commands:

For tracing: `cs f.cs -DDEBUG -o f`

For timing: `cs f.cs -DL=10 -DM=20 -DN=30 -O -o f`

When running a CM program, the first argument specifies the number of times the matrix multiplication is to be performed. For accuracy, a high enough number of runs should be chosen, such that the total time is above 60 sec.

For the sequential algorithm, L, M, and N are variables. The first argument specifies L, M, and N simultaneously, and the second the number of runs. If no argument is given, L, M, and N are set to defaults, and the number of runs to 1.

B The $O(n^3)$ (sequential) algorithm

```
/* This is a sequential C-program for matrix multiplication.
 * It takes 0, 1 or 2 numeric arguments:
 * 0 arguments: (for debugging) array dimensions are fixed.
 *              The multiplication will be executed once.
 * 1 argument:  The argument gives the dimensions of all arrays;
 *              The multiplication will be executed once.
 * 2 arguments: The first argument gives the dimensions of all
 *              arrays; The second argument specifies the number of
 *              times to run the multiplication.
 */

#include <stdio.h>

/*#define DEBUG /* for debugging purposes */

#define T double
#define MAXSIZE 300

T A[MAXSIZE*MAXSIZE]; /* multiplicand matrix; dimensions L,M */
T B[MAXSIZE*MAXSIZE]; /* multiplicand matrix; dimensions M,N */
T C[MAXSIZE*MAXSIZE]; /* multiplicand matrix; dimensions L,N */

extern int atoi();

main(argc,argv)
int argc; char * argv[];
{
register int i,j,k;
register T inner_prod;
register int istarM, istarN, kstarN;
int run, num_of_runs;
int L,M,N;


    switch (argc) {
    case 1: num_of_runs=1;
            L=3; M=4; N=5;
            break;
    case 2: num_of_runs=1;
            L=M=N=atoi(argv[1]);
            break;
    case 3: L=M=N=atoi(argv[1]);
            num_of_runs=atoi(argv[2]);
            break;
    }
    if ((L>M?L:M)*(M>N?M:N) > MAXSIZE*MAXSIZE) {
      printf("Array dimensions exceed %d\n",MAXSIZE);
      exit(0);
    }

#   ifdef DEBUG
    /* initialize A */
    for (i=0;i<L;i++)
      for (j=0; j<M; j++)
        A[i*M+j]= i*j;
    /* intialize B */
    for (i=0;i<M;i++)
      for (j=0; j<N; j++)
        B[i*N+j]= i*j*2;
    fputs("\nMatrix A:\n",stdout);print_Tarray(A,L,M);
    fputs("\nMatrix B:\n",stdout);print_Tarray(B,M,N);
#   endif

    /* this is the loop for timing */
    for (run=0; run<num_of_runs; run++) {

      /* This is the matrix multiply, with strength reduction */
      istarM=0;istarN=0;
      for (i=0; i<L; i++) {
        for (j=0; j<N; j++) {
          inner_prod=0;
          kstarN=0;
          for (k=0; k<M; k++) {
            /*inner_prod=inner_prod+A[i*M   +k]*B[k*N   +j];*/
            inner_prod = inner_prod+A[istarM+k]*B[kstarN+j];
            kstarN=kstarN+N;
          }
          /*C[i*N +j]=inner_prod;*/
          C[istarN+j]=inner_prod;
        }
        istarM=istarM+M; istarN=istarN+N;
      }
#     ifdef DEBUG
      fputs("\nResult of Multiplying A and B:\n",stdout);
      print_Tarray(C,L,N);
#     endif
    }
    printf("Number of runs: %d; Number of processors used: %d\n",
            num_of_runs,1);
    printf("Array dimensions: (%d*%d) and (%d*%d)\n",L,M,M,N);
}
```

C The $O(n^2 \log n)$ algorithm

```
/* This program performs parallel matrix multiplication in n*n*logn steps.
 * The matrices are allocated such that each processor
 * has one row of each matrix.
 * The algorithm performs n*n inner products in sequence.
 * All communication is done by the router.
 * number of processors: n; performance: n*n*logn
 */

#include <stdio.hs>
#include <cm/cmtimer.hs>

/*#define DEBUG     /* prints out matrices for debugging */

#ifdef DEBUG
#define L 3
#define M 4
#define N 5
#endif /* otherwise, define with cs -DL=... */

#define TOTAL_SIZE (L>M?L:M)  /* must be max of L and M */
#define T   double

domain  arrays {T    poly A[M];  /* multiplicand matrix;  dimensions L,M */
                T    poly B[N];  /* multiplicator matrix; dimensions M,N */
                T    poly C[N];  /* destination matrix;   dimensions L,N */
               } data[TOTAL_SIZE];

extern void arrays::print_Tarray(T arrays:: * mono Tarray,
                                 int mono rows, int mono cols);
extern int  atoi(char * s);
extern unsigned CM_virtual_to_physical_processor_ratio; /* (v*w)/(p*q) */


void main(int argc, char *argv[]) {
  register int mono i,j,k;
  register int mono run, num_of_runs;
  CM_timeval_t * mono timer_results;

  num_of_runs=(argc==1)?1:atoi(argv[1]);
  [domain arrays].{
    T   poly Arow;        /* for holding a row of A */
    T   poly temp;        /* for holding products */
#   ifdef DEBUG
    /* initialize A */
    if ((&data[0] <= this) && (this < &data[L]))
```

```
        for (j=0; j<M; j++)
            A[j] = (this-&data[0])*j;

    /* initialize B */
    if ((&data[0] <= this) && (this < &data[M]))
        for (j=0; j<N; j++)
            B[j] = (this-&data[0])*j * 2;

    fputs("\nMatrix A:\n",stdout); print_Tarray(A,L,M);
    fputs("\nMatrix B:\n",stdout); print_Tarray(B,M,N);
#   endif

    CM_start_timer(1);
    for (run=0; run<num_of_runs; run++) { /* this is the loop for timing */
        if (this<&data[M]) {
            for (i=0; i<L; i++) {
                /* line up row i of A with column i of B; could do this  */
                /* with the router (and collisions) or a front-end loop. */
                for (k=0; k<M; k++)
                    data[k].Arow = data[i].A[k];
                /* compute inner product */
                for (j=0; j<N; j++) {
                    temp=Arow*B[j];
                    data[i].C[j] = (+= temp);
                    /* elliminating temp causes a collision bug*/
                }
            }
        }
#       ifdef DEBUG
        fputs("\nResult of multiplying A with B:\n",stdout);
        print_Tarray(C,L,N);
#       endif
    }/* end for (run) */
    timer_results=CM_stop_timer(1);
    printf("Number of runs: %d; Number of processors used: %d\n",
                num_of_runs,TOTAL_SIZE);
    printf("Array dimensions: (%d*%d) and (%d*%d); VP ratio: %d\n",
           L, M, M, N, CM_virtual_to_physical_processor_ratio);
    printf("Real CM time per run: %g\n",timer_results->cmtv_cm/num_of_runs);
    printf("Virtual CM time per run: %g\n",
            timer_results->cmtv_cm/num_of_runs/
                CM_virtual_to_physical_processor_ratio);
  } /* end domain arrays */
} /* end main */
```

D The slow $O(n \log n)$ algorithm

```
/* This program performs parallel matrix multiplication in nlogn steps.
 * The matrices are allocated such that each processor has one elment of
 * of each matrix. Each column of the second matrix is broadcast over the
 * rows of the first matrix, then the products are all formed in parallel,
 * and the rows are summed in parallel. This is repeated for every column
 * of the second matrix. All communication is done by the router.
 * Number of processors: n**2; performance: nlogn
 */

#include <stdio.hs>
#include <cm/cmtimer.hs>

/*#define DEBUG     /* prints out matrices for debugging */

#ifdef DEBUG
#define L 3
#define M 4
#define N 5
#endif /* otherwise, define with cs -DL=...  */

#define TOTAL_SIZE ((L>M?L:M)*(M>N?M:N))
/* must be he max of L*M, M*N, L*N */
#define T double

domain  arrays {T  A;  /* multiplicand matrix;  dimensions L,M */
                T  B;  /* multiplicator matrix; dimensions M,N */
                T  C;  /* destination matrix;   dimensions L,N */
                } data[TOTAL_SIZE];

#define THIS_ROW(columns) ((this-&data[0]) / columns)
#define THIS_COL(columns) ((this-&data[0]) % columns)

extern void arrays::print_Tarray(T arrays::iarray,
                                 int mono rows, int mono cols);
extern int  atoi(char * s);
extern unsigned CM_virtual_to_physical_processor_ratio; /* (v*w)/(p*q) */


void main(int argc, char *argv[]) {
  register int mono stride;
  register int mono B_col; /* runs through column numbers of B */
  register int mono run, num_of_runs;
  CM_timeval_t * mono timer_results;

  num_of_runs=(argc==1)?1:atoi(argv[1]);
  [domain arrays].{
    int poly this_A_row;  /* y-coordinate of each element of A */
    int poly this_A_col;  /* x-coordinate of each element of A */
    T   poly temp;        /* temporary array for broadcasting cols of B.*/
                          /* multiplying with A, and sum reduction; */
                          /* dimensions of temp: L*M*/
    /* initialize A */
    if ((&data[0] <= this) && (this < &data[L*M]))
      A = THIS_ROW(M) * THIS_COL(M);

    /* initialize B */
    if ((&data[0] <= this) && (this < &data[M*N]))
      B = THIS_ROW(N) * THIS_COL(N) * 2;

#   ifdef DEBUG
    fputs("\nMatrix A:\n",stdout);
    print_Tarray(A,L,M);
    fputs("\nMatrix B:\n",stdout);
    print_Tarray(B,M,N);
#   endif

    CM_start_timer(1);
    for (run=0; run<num_of_runs; run++) { /* this is the loop for timing */
        if (this < &data[L*M]) {  /* select whole array A */

          /* compute row and column numbers for each element of A)*/
          this_A_row = THIS_ROW(M);
          this_A_col = THIS_COL(M);

          /* for each column vector of B, multiply it into A.*/
          /* put result vector into corresponding column of C */
          for (B_col = 0; B_col < N; B_col++) {
#           ifdef DEBUG
            printf("\nMultiplication with column %d of B",B_col);
#           endif

            /* Step 1.1: Seed column elements of B */
            /*           into first row of temp for broadcast */
            if (this < &data[M]) /* restrict to first row*/
                temp = data[this_A_col*N+B_col].B;

            /* Step 1.2: Distribute elements down columns of temp.*/
            /*                  (recursive doubling */
            for (stride=M; stride < L*M; stride <<= 1) {
                if ((this+stride)< &data[L*M])
                   (this+stride)->temp = temp;
            }
#           ifdef DEBUG
            printf("\nTemp after distribution of column %d of B:\n",B_col);
            print_Tarray(temp,L,M);
#           endif

            /* Step 2: multiply into temp */
            temp = temp*A;
#           ifdef DEBUG
            fputs("Temp after multiplication with A:\n",stdout);
            print_Tarray(temp,L,M);
#           endif

            /* Step 3: sum scan in parallel, for all rows */
            /* This is a segmented sum scan; segments of equal length */
            for (stride=1; stride < M; stride <<= 1) {
                if ((this_A_col + stride) < M)
                    temp = (this+stride)->temp + temp;
            }
#           ifdef DEBUG
            fputs("Temp after row-wise sum reduction:\n",stdout);
            print_Tarray(temp,L,M);
#           endif

            /* Step 4: copy out of temp into result matrix C*/
            if (this_A_col==0) /*Select first column of A to send values */
                data[this_A_row*N+B_col].C = temp;
            /*Could save this last assignment by letting last iteration */
            /*of step 3 compute the result into C instead of temp.      */
          } /* end for (B_col) */
        } /* end A selection */
#       ifdef DEBUG
        fputs("\nResult of multiplying A with B:\n", stdout);
        print_Tarray(C,L,N);
#           endif
    } /* end for (run) */

    timer_results=CM_stop_timer(1);
    printf("Number of runs: %d; Number of processors used: %d\n",
            num_of_runs,TOTAL_SIZE);
    printf("Array dimensions: (%d*%d) and (%d*%d); VP ratio: %d\n",
          L, M, M, N, CM_virtual_to_physical_processor_ratio);
    printf("Real CM time per run: %g\n",timer_results->cmtv_cm/num_of_runs);
    printf("Virtual CM time per run: %g\n",
            timer_results->cmtv_cm/num_of_runs/
                  CM_virtual_to_physical_processor_ratio);
  } /* end domain arrays */
} /* end main */
```

E The fast $O(n \log n)$ algorithm

```
/* This program performs parallel matrix multiplication in nlogn steps.
 * The matrices are allocated such that each processor has one element
 * of each matrix. In this method, the second matrix is first transposed
 * over the first, and then rotated up row by row.
 * A full parallel multiplication and sum reduction is done for
 * each rotation. All communication is done by the router.
 * number of processors: n**2; performance: n*logn
 */

#include <stdio.hs>
#include <cm/cmtimer.hs>

/*#define DEBUG     /* prints out matrices for debugging */
#ifdef DEBUG
#define L 3
#define M 4
#define N 5
#endif /* otherwise, define with cs -DL=... */

#define LNmin (L<N?L:N)
#define LNmax (L>N?L:N)
#define TOTAL_SIZE ((L>M?L:M)*(M>N?M:N))
/* must be max of L*M, M*N, and L*N */
#define T   double

domain  arrays {T    poly A;  /* multiplicand matrix;  dimensions L,M */
                T    poly B;  /* multiplicator matrix; dimensions M,N */
                T    poly C;  /* destination matrix;   dimensions L,N */
               } data[TOTAL_SIZE];

#define THIS_ROW(columns) ((this-&data[0]) / columns)
#define THIS_COL(columns) ((this-&data[0]) % columns)
extern void arrays::print_Tarray(T arrays::iarray,
                                 int mono rows, int mono cols);
extern int  atoi(char * s);
extern unsigned CM_virtual_to_physical_processor_ratio;   /* (v*w)/(p*q) */


void main(int argc, char *argv[]) {
  register int mono stride;
  register int mono rotation_count; /* counts upward rotations of big*/
  register int mono run, num_of_runs;
  CM_timeval_t * mono timer_results;

  num_of_runs=(argc==1)?1:atoi(argv[1]);
  [domain arrays].{
    T   poly Btr;         /* transpose of B */
    T   poly small, big; /* hold matrices A and Btr */
    T   poly temp;        /* temporary variable for multiply, sum reduce */
    int poly rowB, colB; /* row and column numbers of each element of B */
    int poly rowA, colA; /* row and column numbers of each element of
                             arrays A, temp, big, small*/
    /* initialize A */
    if ((&data[0] <= this) && (this < &data[L*M]))
      A = THIS_ROW(M) * THIS_COL(M);

    /* initialize B */
    if ((&data[0] <= this) && (this < &data[M*N]))
      B = THIS_ROW(N) * THIS_COL(N) * 2;
#   ifdef DEBUG
    fputs("\nMatrix A:\n",stdout); print_Tarray(A,L,M);
    fputs("\nMatrix B:\n",stdout); print_Tarray(B,M,N);
#   endif

    CM_start_timer(1);
    for (run=0; run<num_of_runs; run++) { /* this is the loop for timing */
            rowA = THIS_ROW(M); colA = THIS_COL(M);
            rowB = THIS_ROW(N); colB = THIS_COL(N);

            /* Step 1: transpose B into Btr */
            if ((rowB<M) && (colB<N))
                data[colB*M+rowB].Btr = B;

            /* Step 2: multiply, sum reduce, then rotate rows of Btr up.
             * A and Btr have the same number of columns, but may have
             * differing number of rows. Rotate the larger one; keep the
             * smaller one in place, because this is easier to program.
             * The smaller one goes into array small, the larger one into
             * array big. Can only do this for commutative operators.
             * (rotating the smaller array would mean less communication.)
             */
            if (L>N) { big=A;   small=Btr;}
            else     { big=Btr; small=A; }
#           ifdef DEBUG
            printf("\nMatrix small:\n");print_Tarray(small,LNmin,M);
#           endif
            for (rotation_count=0; 1;/*end with break*/ rotation_count++) {
#               ifdef DEBUG
                printf("\nMatrix big after rotation %d:\n",rotation_count);
                print_Tarray(big,LNmax,M);
#               endif

                if (rowA < LNmin) {
                    temp = small*big; /* commutativity enters here */
#                   ifdef DEBUG
                    fputs("\nMatrix temp after mult. of small and big:\n",
                          stdout); print_Tarray(temp,LNmin,M);
#                   endif

                    /*Sum reduction in parallel, for all rows of temp */
                    /*This is a segmented sum scan */
                    for (stride=1; stride < M; stride <<= 1) {
                        if (((colA%(stride<<1))==0) && ((colA+stride)<M))
                            temp = (this+stride)->temp + temp;
                    }
#                   ifdef DEBUG
                    fputs("\nMatrix temp after sum reduction:\n",stdout);
                    print_Tarray(temp,LNmin,M);
#                   endif

                    /* copy temp into result matrix C*/
                    if (colA ==0) /* select column 0 for send */
                        data[rowA*N + ((rowA+rotation_count)%N)].C = temp;
                }
                if (rotation_count>=(LNmax-1)) break; /*exit form middle*/

                /*now rotate all rows of big up one row--could use grid*/
                if ((rowA<LNmax) && (colA<M))
                   big = data[((rowA+1)%LNmax)*M+colA].big;

            } /* end for (rotation_count) */
#           ifdef DEBUG
            fputs("\nResult of multiplying A with B:\n", stdout);
            print_Tarray(C,L,N);
#           endif
      }/* end for (run) */
      timer_results=CM_stop_timer(1);
      printf("Number of runs: %d; Number of processors used: %d\n",
                  num_of_runs,TOTAL_SIZE);
      printf("Array dimensions: (%d*%d) and (%d*%d); VP ratio: %d\n",
              L, M, M, N, CM_virtual_to_physical_processor_ratio);
      printf("Real CM time per run: %g\n",
                  timer_results->cmtv_cm/num_of_runs);
      printf("Virtual CM time per run: %g\n",
               timer_results->cmtv_cm/num_of_runs/
                  CM_virtual_to_physical_processor_ratio);
   } /* end domain arrays */
} /* end main */
```

F The slow $O(n)$ algorithm

```
/* This program performs parallel matrix multiplication.
 * The matrices are allocated such that each processor
 * has an element of each matrix.
 * This is a pipelined algorithm: The rows of the first matrix
 * and the columns of the second matrix are pumped into
 * the result matrix from the West and North, resp.
 * Each element of the result matrix retrieves coefficients
 * from the West and North, multiplies them, and adds them to the
 * running total. All communication is done with router.
 * number of processors: n**2; performance: n
 */


#include <stdio.hs>
#include <cm/cmtimer.hs>

/*#define DEBUG     /* prints out matrices for debugging */

#ifdef DEBUG
#define L 3
#define M 4
#define N 5
#endif  /* otherwise, define with cs -DM=... etc. */

#define TOTAL_SIZE ((L>M?L:M)*(M>N?M:N))
/* must be the max of L*M, M*N, L*N */

#define T double

domain  arrays {T  poly A;   /* multiplicand matrix;  dimensions L,M */
                T  poly B;   /* multiplicator matrix; dimensions M,N */
                T  poly C;   /* destination matrix;   dimensions L,N */
               } data[TOTAL_SIZE];

#define THIS_ROW(columns) ((this-&data[0]) / columns)
#define THIS_COL(columns) ((this-&data[0]) % columns)

extern void arrays::print_Tarray(T arrays::iarray,
                                  int mono rows, int mono cols);
extern int  atoi(char * s);
extern unsigned CM_virtual_to_physical_processor_ratio; /* (v*w)/(p*q) */


void main(int argc, char *argv[]) {
```

```
register int mono run, num_of_runs;
register int mono r; /* pipelining counter */
CM_timeval_t * mono timer_results;

num_of_runs=(argc==1)?1:atoi(argv[1]);
[domain arrays].{
  int poly i, j;       /* row and column numbers of each element of C */
  int poly r_i_j;         /* common subexpression */
  T   poly north, west;/* for pumping columns of B and rows of A */

  /* initialize A */
  if ((&data[0] <= this) && (this < &data[L*M]))
    A = THIS_ROW(M) * THIS_COL(M);

  /* initialize B */
  if ((&data[0] <= this) && (this < &data[M*N]))
    B = THIS_ROW(N) * THIS_COL(N) * 2;

#  ifdef DEBUG
  fputs("\nMatrix A:\n",stdout);
  print_Tarray(A,L,M);
  fputs("\nMatrix B:\n",stdout);
  print_Tarray(B,M,N);
#  endif


  CM_start_timer(1);
  for (run=0; run<num_of_runs; run++) { /* this is the loop for timing */

      if ((&data[0] <= this) && (this < &data[L*N])) {
        /* select whole array C */

        C=0.0;  /* initialize C */
        /* compute row and column numbers for each element of C.*/
        i = THIS_ROW(N);  j = THIS_COL(N);

        for (r=0; r<M+N+L-2; r++) {
          r_i_j = r-i-j; /* common subexpression */
          if ((0 <= r_i_j) && (r_i_j < M)) {
              west = data[i*M + r_i_j].A;
              north= data[r_i_j*N + j].B;
              /* This code actually does not do any systolic pipelining.
               * Instead, coefficients are retrieved directly from
               * A and B, with general communication.
               * The pipelining code is below, but it is slower, because
               * it does more communication (even with grid addressing).
               *if (i!=0) /* not first row -- get from north */
               *  north=data[(i-1)*N + j].north; /*cou'd use grid here */
```

```
                    *else /* first row -- get data from B */
                    *  north=data[(r_i_j)*N + j].B;
                    *if (j!=0) /* not first column -- get from west */
                    *  west=data[i*N + (j-1)].west;  /*could use grid here */
                    *else /* first column -- get data from A */
                    *  west=data[i*M + r_i_j].A;
                    */
                   C = C + west*north;

                }
#               ifdef DEBUG
                printf("\nPhase %d:\n",r);
                printf("North elements:\n"); print_Tarray(north,L,N);
                printf("West elements:\n"); print_Tarray(west,L,N);
                printf("Matrix C:\n"); print_Tarray(C,L,N);
#               endif
              } /* end for r */
            } /* end A selection */
    } /* end for run */
    timer_results=CM_stop_timer(1);
    printf("Number of runs: %d; Number of processors used: %d\n",
              num_of_runs,TOTAL_SIZE);
    printf("Array dimensions: (%d*%d) and (%d*%d); VP ratio: %d\n",
           L, M, M, N, CM_virtual_to_physical_processor_ratio);
    printf("Real CM time per run: %g\n",timer_results->cmtv_cm/num_of_runs);
    printf("Virtual CM time per run: %g\n",
                timer_results->cmtv_cm/num_of_runs/
                     CM_virtual_to_physical_processor_ratio);
  } /* end domain arrays */
} /* end main */
```

G The fast $O(n)$ algorithm

```
/* This program performs parallel matrix multiplication.
 * The matrices are allocated such that each processor
 * has an element of each matrix.
 * This is a systolic algorithm: All elements
 * of the result matrix perform a step of the inner product
 * during each iteration. The ordering of forming the inner
 * products is skewed, such that each elmemt of A and B
 * is needed exactly once in each iteration.
 * All communication is done by the router.
 * number of processors: n**2; performance: n
 * With grid communication, this program could be speeded up
 * considerably by rotating the rows and columns of A and B into place.
 */


#include <stdio.hs>
#include <cm/cmtimer.hs>

/*#define DEBUG     /* prints out matrices for debugging */

#ifdef DEBUG
#define L 3
#define M 4
#define N 5
#endif  /* otherwise, define with cs -DM=... etc. */

#define TOTAL_SIZE ((L>M?L:M)*(M>N?M:N))
/* must be the max of L*M, M*N, L*N */

#define T double

domain  arrays {T  poly A;   /* multiplicand matrix;  dimensions L,M */
                T  poly B;   /* multiplicator matrix; dimensions M,N */
                T  poly C;   /* destination matrix;   dimensions L,N */
               } data[TOTAL_SIZE];

#define THIS_ROW(columns) ((this-&data[0]) / columns)
#define THIS_COL(columns) ((this-&data[0]) % columns)

extern void arrays::print_Tarray(T arrays::iarray,
                                 int mono rows, int mono cols);
extern int  atoi(char * s);
extern unsigned CM_virtual_to_physical_processor_ratio; /* (v*w)/(p*q) */
```

```
void main(int argc, char *argv[]) {
  register int mono run, num_of_runs;
  register int mono r; /* inner product step counter */
  CM_timeval_t * mono timer_results;

  num_of_runs=(argc==1)?1:atoi(argv[1]);
  [domain arrays].{
    int poly i, j;       /* row and column numbers of each element of C */
    int poly k;          /* index for inner product - avoids contention */
    /* initialize A */
    if ((&data[0] <= this) && (this < &data[L*M]))
      A = THIS_ROW(M) * THIS_COL(M);

    /* initialize B */
    if ((&data[0] <= this) && (this < &data[M*N]))
      B = THIS_ROW(N) * THIS_COL(N) * 2;

#   ifdef DEBUG
    fputs("\nMatrix A:\n",stdout);
    print_Tarray(A,L,M);
    fputs("\nMatrix B:\n",stdout);
    print_Tarray(B,M,N);
#   endif


    CM_start_timer(1);
    for (run=0; run<num_of_runs; run++) { /* this is the loop for timing */

        /* select whole array C */
        if ((&data[0] <= this) && (this < &data[L*N])) {
            /* initialize C */
            C=0.0;
            /* compute row and column numbers for each element of C.*/
            i = THIS_ROW(N);  j = THIS_COL(N);
            /* k is initialized such that access to A and B is skewed */
            k = (i+j)%M;

            for (r=0; ; ) { /* exit from middle */

#               ifdef DEBUG
                printf("\nPhase %d:\n",r);
                printf("k:\n"); print_Tarray(k,L,N);
#               endif DEBUG
                /* This code actually does not do any systolic
                 * pipelining. Instead, coefficients are retrieved
                 * directly from A and B, with general communication. */
                C = C + data[i*M + k].A * data[k*N + j].B;
#               ifdef DEBUG
                printf("Matrix C:\n"); print_Tarray(C,L,N);
#               endif
                if (r >= (M-1)) break;
                k = (k+1)%M;
                r++;
            } /* end for r */
        } /* end C selection */
    } /* end for run */
    timer_results=CM_stop_timer(1);
    printf("Number of runs: %d; Number of processors used: %d\n",
                num_of_runs,TOTAL_SIZE);
    printf("Array dimensions: (%d*%d) and (%d*%d); VP ratio: %d\n",
           L, M, M, N, CM_virtual_to_physical_processor_ratio);
    printf("Real CM time per run: %g\n",timer_results->cmtv_cm/num_of_runs);
    printf("Virtual CM time per run: %g\n",
            timer_results->cmtv_cm/num_of_runs/
                  CM_virtual_to_physical_processor_ratio);
  } /* end domain arrays */
} /* end main */
```

H The $O(\log n)$ algorithm

```
/* This program performs parallel matrix multiplication in log n steps.
 * The matrices are allocated such that each processor
 * has one element of each matrix.
 * The algorithm replicates the arrays such that all n**3 multiplication
 * can be done in parallel, followed by the parallel sum-reduction for
 * the n**2 inner products.
 * All communication is done by the router.
 * Number of processors: n**3; performance: log n.
 */


#include <stdio.hs>
#include <cm/cmtimer.hs>

/*#define DEBUG     /* prints out matrices for debugging */

#ifdef DEBUG
#define L 3
#define M 4
#define N 5
#endif  /* otherwise, define with cs -DL=... */
/* for full processor utilization,
 *L*M*N should equal the number of processors
 * 2**15 = 2**5 * 2**5 * 2**5, or approx. 25**3.
 * 2**16 = 2**5 * 2**6 * 2**5, or approx. 40**3  */

#define TOTAL_SIZE (L*M*N)

#define T double

domain  arrays {T  A;  /* multiplicand matrix;  dimensions L,M */
                T  B;  /* multiplicator matrix; dimensions M,N */
                T  C;  /* result matrix;        dimensions L,N */
               } data[TOTAL_SIZE];


#define THIS_ROW(columns) (proc_number / (columns))
#define THIS_COL(columns) (proc_number % (columns))
/* requires proc_number to be initialized with (this-&data[0]) */

extern void arrays::print_Tarray(T arrays::iarray,
                                 int mono rows, int mono cols);
extern int atoi(char * s);
extern unsigned CM_virtual_to_physical_processor_ratio; /* (v*w)/(p*q) */
```

```
void main(int argc, char *argv[]) {
  register int stride;
  register int mono run, num_of_runs;
  CM_timeval_t * mono timer_results;

  num_of_runs=(argc==1)?1:atoi(argv[1]);

  [domain arrays].{

    int poly proc_number;       /* processor number */
    T   poly Aspread;           /* A spread out */
    T   poly Bspread;           /* B transposed and spread out */
    int poly index;             /* temporary for column and row indices */

    proc_number=this-&data[0]; /* set processor number */

    /* initialize A */
    if ((&data[0] <= this) && (this < &data[L*N]))
        A = THIS_ROW(M) * THIS_COL(M);

    /* initialize B */
    if ((&data[0] <= this) && (this < &data[M*N]))
        B = THIS_ROW(N) * THIS_COL(N) *2;
#   ifdef DEBUG
    fputs("\nMatrix A:\n",stdout);
    print_Tarray(A,L,M);
    fputs("\nMatrix B:\n",stdout);
    print_Tarray(B,M,N);
#   endif


    /* Algorithm: the data of the two matrices A and B is replicated
     * and aligned such that all multiplications can be done in parallel.
     * Sum reductions are also done in parallel. This is the layout:

   A: |row0|row0|...|row0  |row2|row2|...|row2  |  .... |rowL-1|...|rowL-1|

   B: |col0|col1|...|colN-1|col0|col1|...|colN-1|  .... |col0  |...|colN-1|
     */

    CM_start_timer(1);
    for (run=0; run<num_of_runs; run++) { /* this is the loop for timing */

        proc_number=this-&data[0]; /* set processor number */
        /* Step 1: replicate each row of A N times into Aspread */
        /* Step 1.1: First, place initial rows */
        if (proc_number<L*M)
            data[THIS_ROW(M)*M*N+THIS_COL(M)].Aspread=A; /* send */
        /* Step 1.2: Duplicate each row N times with recursive doubling */
        /* View Aspread as an array of L rows with M*N columns. */
        /* The first M columns have to be spread right */
        index = THIS_COL(M*N); /* column index in Aspread(L,M*N) */
        for (stride=M; stride < M*N; stride <<= 1) {
          if (index+stride < M*N)
            (this+stride)->Aspread=Aspread; /* this is a send */
        }
#       ifdef DEBUG
        printf("\nAspread (A's rows replicated %d times end to end):\n",N);
        print_Tarray(Aspread,L,M*N);
#       endif

        /* Step 2: Replicate the entire data of B L times into Bspread */
        /* Step 2.1: First, transpose B into Bspread */
        if (proc_number<M*N)
            data[THIS_COL(N)*M+THIS_ROW(N)].Bspread=B;
        /* Step 2.2: replicate the first M*N elements of Bspread L times */
        /* View Bspread as an L*(MN) array; spread rows down */
        for (stride=N*M; stride < L*M*N; stride <<= 1) {
            if ((this+stride) < &data[L*M*N])
                (this+stride)->Bspread=Bspread; /* this is a send */
        }
#       ifdef DEBUG
        printf("\nBspread (B's rows replicated %d times:\n",L);
        print_Tarray(Bspread,L,M*N);
#       endif


        /* Step 3: Multiply in parallel */
        Bspread=Aspread*Bspread;
#       ifdef DEBUG
        printf("\nBspread (elmentwise product of Aspread and Bspread:)\n");
        print_Tarray(Bspread,L,M*N);
#       endif

        /* Step 4: Sum scan (Could be done with a segmented sum scan) */
        /* View Bspread as an array of L*N vectors of length M. */
        /* Sum the vectors in parallel; scan faster than reduction */
        index=THIS_COL(M); /* column index in Bspread(L*N,M) */
        for (stride=1; stride<M; stride <<= 1) {
            if ((index+stride) < M)
               Bspread=Bspread+(this+stride)->Bspread;  /* this is a get */
        }
#       ifdef DEBUG
        printf("\nBspread (add reduction on subvectors of length %d):\n",M);
        print_Tarray(Bspread,L,M*N);
#       endif

        /* Step 5:Gather results into C */
        if ((0<=proc_number) && (proc_number< L*N))
            C=data[proc_number*M].Bspread;

#       ifdef DEBUG
        fputs("\nResult of multiplying A with B:\n", stdout);
        print_Tarray(C,L,N);
#       endif
    } /* end for (run) */
    timer_results=CM_stop_timer(1);
    printf("Number of runs: %d; Number of processors used: %d\n",
                num_of_runs,TOTAL_SIZE);
    printf("Array dimensions: (%d*%d) and (%d*%d); VP ratio: %d\n",
           L, M, M, N, CM_virtual_to_physical_processor_ratio);
    printf("Real CM time per run: %g\n",timer_results->cmtv_cm/num_of_runs);
    printf("Virtual CM time per run: %g\n",
            timer_results->cmtv_cm/num_of_runs/
                  CM_virtual_to_physical_processor_ratio);
  }/* end domain arrays */
}/* end main */
```

I The print routines for the CM programs

The following two routines are needed for the DEBUG option, to print out matrices. These routines work for all CM programs, but not for the sequential matrix multiply.

```
void arrays::print_Trow(Tarray, row, cols)
/* print a row of length cols from an array in CM-memory */
    T   arrays:: Tarray;  /* array */
    int mono     row;     /* row number */
    int mono     cols;    /* row length */
{   int mono col;
    for (col=0; col<cols;) {
        printf("%4g  ",data[row*cols + col].Tarray);
        if ((cols <= 14) || (col != 8))
            col++; /* go on to next element */
        else { /* skip some elements */
            printf("...  "); col=cols-4;
        }
    }
}

void arrays::print_Tarray(Tarray,rows,cols)
/* print array Tarray */
    T   arrays::Tarray; /* array */
    int mono    rows;   /* number of rows */
    int mono    cols;   /* number of columns */
{   int mono row;       /* row counter */
    for (row=0; row<rows;) {
        print_Trow(Tarray,row,cols);
        putc('\n',stdout);
        if ((rows <= 14) || (row != 8))
            row++; /* go on to next row */
        else { /* skip some rows */
            fputs("........\n",stdout);
            row=rows-4;
        }
    }
}
```