10.10 Write a parallel program implementing the Ising model described in Section 10.5.3. The objective is to find the energy level of a 100×100 system after 1,000,000 iterations. Let $J = 1$, $B = 0$, and $kT = 1$. Give the system a "cold start" by initializing every site σ_k to "up" in state x_0. Evaluate $\sum_{i,j} J\sigma_i\sigma_j$ for every pair of sites that are horizontally or vertically adjacent. Repeat the experiment 1,000 times.

10.11 Implement a parallel program solving the room assignment problem posed in Section 10.5.4. Assume $n = 20$ and $T = 1$. Use a random number generator to construct matrix D. Each entry should be a uniform random variable between 0 and 10. Each process should solve the problem for the same matrix D, but with different seeds for the random number generator.

10.12 Implement a parallel program solving the parking garage problem posed in Section 10.5.5. Assume $S = 80$, $A = 3$, and $M = 240$. Determine the average number of stalls occupied by cars, and the probability of a car being turned away because the garage is full, as $t \rightarrow \infty$, that is, in the steady state.

10.13 Implement a parallel program solving the traffic circle problem posed in Section 10.5.6. Use the program to answer these two questions:

a. For each of the four traffic circle entrances, what is the steady state probability that a car will have to wait before entering the circle?

b. For each of the four traffic circle entrances, what is the average length of the queue of vehicles waiting to enter the traffic circle, in the steady state?

C H A P T E R

11 Matrix Multiplication

We go on multiplying our conveniences only to multiply our cares. We increase our possessions only to the enlargement of our anxieties.

Anna C. Brackett, *The Technique of Rest*

11.1 INTRODUCTION

Considering how often the matrix multiplication algorithm is presented in computer science classes, it's ironic that few scientific and engineering problems require the multiplication of large matrices. Here are two domains in which matrix multiplication is used. Computational chemists represent some problems in terms of states of a chemical system. Each index corresponds to a different basis state, and the matrix approximates the Hamiltonian of the system. A change of basis is accomplished through matrix multiplication. As another example, some transforms used in signal processing rely on the multiplication of large matrices.

This chapter presents two sequential matrix multiplication algorithms and then explores two different approaches to parallel matrix multiplication. In Section 11.2 we review the standard sequential matrix multiplication algorithm. Charting the algorithm's performance as matrix sizes increase, we see how performance drops dramatically once the second factor matrix no longer fits inside cache memory. We then show how a recursive implementation of matrix multiplication that multiplies blocks of the original matrices can maintain a high cache hit rate.

In Section 11.3 we design a parallel algorithm based upon a rowwise block-striped decomposition of the matrices. We derive an expression for the expected computation time of this algorithm, and we analyze its isoefficiency. In Section 11.4 we go through the same design and analysis methodology for a parallel algorithm based on a checkerboard block decomposition of the matrices.

11.2 SEQUENTIAL MATRIX MULTIPLICATION

11.2.1 Iterative, Row-Oriented Algorithm

The product of an $l \times m$ matrix A and an $m \times n$ matrix B is an $l \times n$ matrix C whose elements are defined by

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

A sequential algorithm implementing matrix multiplication appears in Figure 11.1. The algorithm requires lmn additions and the same number of multiplications. Hence the time complexity of multiplying two $n \times n$ matrices using this sequential algorithm is $\Theta(n^3)$. Sequential matrix multiplication algorithms with a lower time complexity have been developed, such as Strassen's algorithm, but every algorithm developed in this chapter is a parallelization of the straightforward algorithm.

It's easy to implement this algorithm. We've benchmarked a C implementation of this matrix multiplication algorithm on a node of a Beowulf cluster: a Linux computer with a 933 MHz Pentium III CPU with a 233 Kilobyte level 2 cache. The results of the benchmarking appear in Figure 11.2. For smaller matrices the execution speed is about 220 megaflops, but for larger matrices the execution speed is about 80 megaflops. What accounts for this drop in performance?

Consider Figure 11.3. During each iteration of the outer i loop, every element of matrix B is read. If matrix B is too large for the cache, then later elements read into cache displace earlier elements read into cache, meaning that in the next iteration of the loop indexed by i, all of the elements of B will need to be read into cache again. Hence once the matrices reach a certain size, the cache hit rate falls dramatically, lowering the performance of the CPU.

The CPU we used for benchmarking has a 256 Kilobyte cache. We are multiplying double-precision floating-point numbers, meaning each matrix element fills eight bytes. Hence the cache can hold at most 32,768 matrix elements. The square root of 32,768 is about 181. The performance of the algorithm reflects that when $n \leq 150$, the cache hit rate is much higher than when $n \geq 200$.

```
Matrix Multiplication (row-oriented):

Input:
  a[0..l − 1, 0..m − 1]
  b[0..m − 1, 0..n − 1]

Output:
  c[0..l − 1, 0..n − 1]

for i ← 0 to l − 1
  for j ← 0 to n − 1
    c[i, j] ← 0
    for k ← 0 to m − 1
      c[i, j] ← c[i, j] + a[i, k] × b[k, j]
    endfor
  endfor
endfor
```

Figure 11.1 Iterative, row-oriented matrix multiplication algorithm.

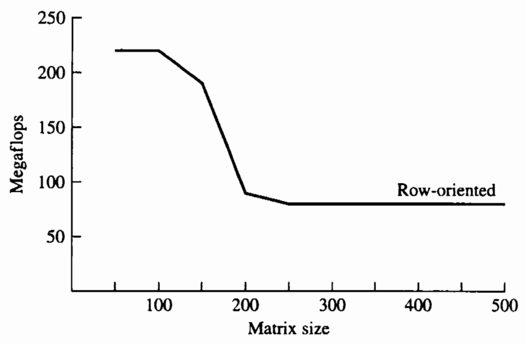


Figure 11.2 Performance of row-oriented matrix multiplication algorithm on a computer with a 933 MHz Pentium III CPU. When matrix B no longer fits in the cache, the performance of the row-oriented matrix multiplication algorithm drops sharply.

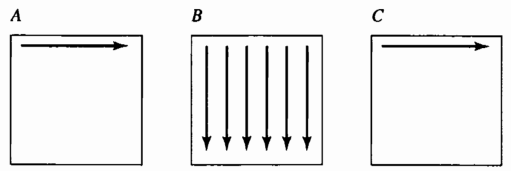


Figure 11.3 In a single iteration of the loop indexed by i, row i of matrix A and all of matrix B are read, while row i of C is written.

11.2.2 Recursive, Block-Oriented Algorithm

In order to perform the matrix multiplication AB, the number of columns of A must be equal to the number of rows of B.

```
double a[N][N], b[N][N], c[N][N];

void mm (int crow, int ccol, /* Corner of C block */
        int arow, int acol, /* Corner of A block */
        int brow, int bcol, /* Corner of B block */
        int l,              /* Block A is l x m */
        int m,              /* Block B is m x n */
        int n)              /* Block C is l x n */
{
   int lhalf[3], mhalf[3], nhalf[3]; /* Quadrant sizes */
   int i, j, k; double *aptr, *bptr, *cptr;

   if (m * n > THRESHOLD) {

      /* B doesn't fit in cache---multiply blocks of A, B */

      lhalf[0] = 0;  lhalf[1] = l/2;  lhalf[2] = l - l/2;
      mhalf[0] = 0;  mhalf[1] = m/2;  mhalf[2] = m - m/2;
      nhalf[0] = 0;  nhalf[1] = n/2;  nhalf[2] = n - n/2;
      for (i = 0; i < 2; i++)
         for (j = 0; j < 2; j++)
            for (k = 0; k < 2; k++)
               mm (crow+lhalf[i], ccol+mhalf[j],
                   arow+lhalf[i], acol+mhalf[k],
                   brow+mhalf[k], bcol+nhalf[j],
                   lhalf[i+1], mhalf[k+1], nhalf[j+1]);
   } else {

      /* B fits in cache --- do standard multiply */

      for (i = 0; i < l; i++)
         for (j = 0; j < n; j++) {
            cptr = &c[crow+i][ccol+j];
            aptr = &a[arow+i][acol];
            bptr = &b[brow][bcol+j];
            for (k = 0; k < m; k++) {
               *cptr += *(aptr++) * *bptr; bptr += N;
            }
         }
   }
}
```

Figure 11.4 C function implementing recursive, block-oriented matrix multiplication. The initial call to this function is `mm (0, 0, 0, 0, 0, 0, N, N, N)`.

Let's suppose A has l rows and m columns, while B has m rows and n columns. If we divide A into four smaller matrices

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

and divide B into four smaller matrices

$$B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

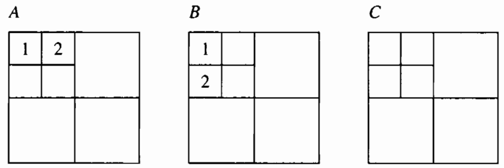


Figure 11.5 A recursive matrix multiplication algorithm breaks the matrices into smaller and smaller blocks until they can fit in cache. Here the algorithm has recursed twice before the blocks are small enough. Each block of C is the sum of the results of two block matrix multiplications.

such that the number of columns in A_{00} and A_{10} is equal to the number of rows in B_{00} and B_{01}, then the matrix product

$$C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{11} + A_{11}B_{11} \end{pmatrix}$$

where each $A_{ik}B_{kj}$ represents multiplication of the block matrices and each $+$ represents matrix addition.

Our goal is to compute the matrix product $C = AB$. If matrix B is too large to fit into cache, we can divide it into four pieces and use the idea of block matrix multiplication to compute C. If block B_{ij} is too large to fit into cache, we can apply this idea recursively until we have blocks that do fit in cache. A C implementation of the resulting recursive algorithm appears in Figure 11.4.

Figure 11.5 illustrates how the recursive matrix multiplication algorithm works. In this example, matrix B is too large for cache, so it is divided into four pieces. Each of the four pieces is still too large, so the algorithm recurses a second time.

We've benchmarked a C implementation of this recursive matrix multiplication algorithm on the same computer we used to measure the speed of the straightforward algorithm. The results of both benchmarking experiments appear in Figure 11.6. The recursive algorithm maintains high performance, even as the sizes of the matrices grow well beyond the cache capacity.

11.3 ROWWISE BLOCK-STRIPED PARALLEL ALGORITHM

In this section we develop a parallel matrix multiplication algorithm based upon a rowwise block-striped decomposition of the matrices.

11.3.1 Identifying Primitive Tasks

Each element of the product matrix C is a function of elements in A and B. Since A and B are not modified during the algorithm, it is possible to compute every

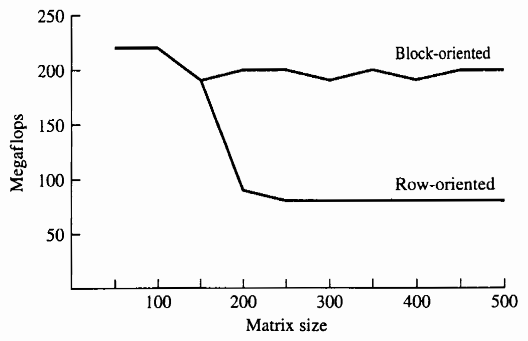


Figure 11.6 Performance of both sequential matrix multiplication algorithms on a computer with a 933 MHz Pentium III CPU. The block-oriented matrix multiplication algorithm keeps the cache hit rate high and achieves better performance than the row-oriented algorithm.

element of C simultaneously. As a first step in our parallel design, then, we can associate one primitive task with every element of C.

Precisely which elements does each of these tasks need? Computing element $c_{i,j}$ of the product matrix involves finding the inner product (dot product) of row i of A and column j of B.

11.3.2 Agglomeration

We can use this data dependence information to agglomerate tasks. It is natural to agglomerate tasks associated with either a row of C or a column of C, since they share a need for either a row of A or a column of B, respectively. Algorithms based on either of these design choices are quite similar. Let's choose to agglomerate tasks associated with a row of C.

It's simpler if we use the same agglomeration for all matrices. That way, the result of one matrix multiplication can be used as either factor matrix in another matrix multiplication. We assume, then, that each task is responsible for corresponding rows of A, B, and C.

Let's think about what task i can do with row i of A, row i of B, and row i of C. Recall that

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

With row i of matrices A and B, the task can compute $a_{i,i}b_{i,0}$, which is one of the terms of $c_{i,0}$. It can also compute $a_{i,i}b_{i,1}$, which is one of the terms of $c_{i,1}$, and so on. In other words, the task can perform n multiplications that represent partial sums for the n elements of row i of C.

Then what? We've already noted in an earlier section that row i of C is the product of row i of A and matrix B. So to complete its work each task must eventually access each row of B.

11.3.3 Communication and Further Agglomeration

If we organize the tasks as a ring, and each task passes its row of B to the next task in the ring, after a series of m iterations every task will have had possession of every row of B.

Figure 11.7 illustrates this process assuming there are four rows in C.

The number of processes on which we execute our parallel algorithm is probably much less than the number of rows in the product matrix, so we need to think of further agglomeration. Given the communication pattern we have developed, it makes sense to use a rowwise block-striped decomposition scheme.

11.3.4 Analysis

To simplify our analysis, we assume that A, B, and C are all $n \times n$ matrices. We also assume that n is a multiple of p, the number of active processes. Each process controls the same n/p rows of A and C throughout the algorithm. The contiguous groups of n/p rows of B are passed from process to process as illustrated in Figure 11.7.

When the algorithm begins, each process initializes its $(n/p) \times n$ portion of C to 0. During each iteration every process multiplies an $(n/p) \times n/p$ block of A by the $(n/p) \times n$ portion of B it currently possesses. It adds the resulting $(n/p) \times n$ matrix to its portion of C. If χ is the time needed for one of the add-multiply steps inside an inner product, the computational time of each iteration is

$$\chi(n/p)(n/p)n = \chi n^3/p^2$$

During every iteration, each process must also communicate its portion of B to the next process on the ring. If the communication is done after the computation, the time needed to send these elements would add $\lambda + (n/p)n/\beta$ to the execution time of each iteration. Receiving the next section of B from the predecessor ring process would occur at the same time.

The algorithm has p iterations. The total computation and communication time, then, is

$$p[\chi n^3/p^2 + \lambda + n^2/(p\beta)] = \chi n^3/p + p\lambda + n^2/\beta$$

Let's double-check on this expression. The sequential algorithm would have execution time χn^3. Since the computations are divided perfectly among the processes, it makes sense that the computational portion of the parallel algorithm has execution time $\chi n^3/p$.

Every process sends p messages, so the $p\lambda$ term also makes sense. Finally, every process handles all of B and sends all of it (one piece at a time) to its successor process, so the n^2/β term fits.

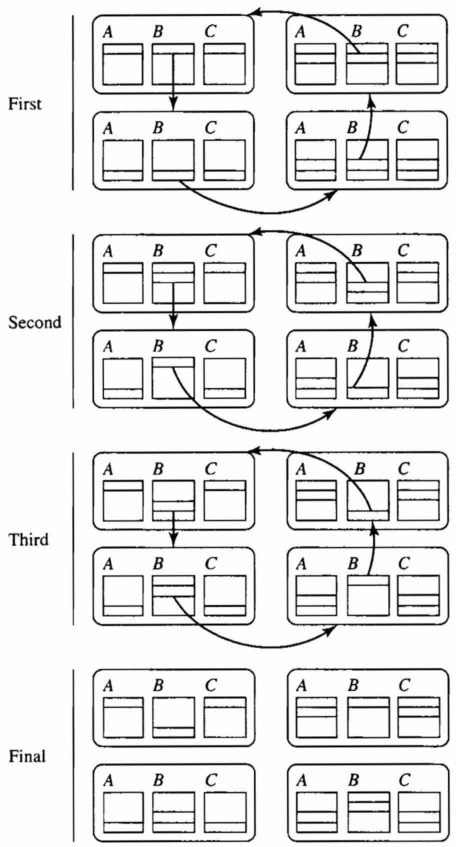


Figure 11.7 Communication of B in row-oriented parallel matrix multiplication algorithm. Each task is responsible for a row of A, a row of B, and a row of C. If B has m rows, then after $m-1$ communication steps each task has had access to every row of B.

Let's determine the isoefficiency of the rowwise block-striped matrix multiplication algorithm. The sequential algorithm has time complexity $\Theta(n^3)$. The communication complexity of the parallel algorithm is $\Theta(n^2)$. We multiply the community complexity by the number of processors p to get the overhead term: $T_o(n, p) = \Theta(pn^2)$. Hence the isoefficiency relation for the rowwise block-striped

matrix multiplication algorithm is

$$n^3 \geq Cpn^2 \Rightarrow n \geq Cp$$

We note that the memory utilization function $M(n) = n^2$. Let's determine how memory utilization per processor must increase in order to maintain a constant level of efficiency:

$$M(Cp)/p = C^2p^2/p = C^2p$$

This algorithm is not highly scalable.

Finally, this algorithm presents a good opportunity for overlapping communications with computation. Assuming there is enough memory to receive a new section of B while performing computations on the current section, each process could initiate its send/receive of B sections before it performed the matrix multiplication step. Since the communication complexity is $\Theta(n^2)$ and the computational complexity is $\Theta(n^3)$, the communication step can be almost completely overlapped with computations when the matrix sizes are large enough. (The time to initiate the communication cannot be overlapped with a computation.) When this happens, speedup can be very high.

Can we do better?

Let's consider the computation/communication ratio of the parallel row-oriented algorithm. When multiplying two $n \times n$ matrices on p processes, where n is a multiple of p, each process iterates through p iterations of a loop in which it multiplies an $(n/p) \times (n/p)$ submatrix of A with an $(n/p) \times n$ submatrix of B. Since the matrix multiplication steps are interleaved with communication steps in which elements of B are being passed from process to process, the ratio of computations per element of B is

$$\frac{2n^3/p^2}{n^2/p} = \frac{2n}{p}$$

The ratio is relatively low, because the submatrices of B have p times as many columns as rows.

In the next section we will develop an algorithm that improves the computation-to-communication ratio.

11.4 CANNON'S ALGORITHM

In this section we develop a parallel algorithm based upon a checkerboard block decomposition of the matrices. The algorithm is often referred to as **Cannon's algorithm** [14].

11.4.1 Agglomeration

The row-oriented parallel algorithm has a low ratio of computations per element of B because the blocks of B being manipulated are short and fat—having p times as many columns as rows.

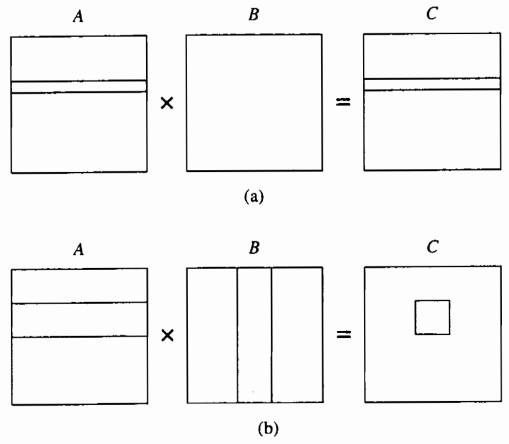


Figure 11.8 Comparison of number of elements of A and B needed to compute a process's portion of C in the two parallel matrix multiplication algorithms. (a) In the row-oriented algorithm, each process is responsible for computing n/p rows of C. It needs to reference n/p rows of A and every element of B. (b) In Cannon's algorithm, each process is responsible for computing an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of C. It needs to reference $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ columns of B.

The task responsible for computing element $c_{i,j}$ of the product matrix requires access to every element of row i of A and every element of column j of B. With a row-oriented agglomeration, every process is responsible for computing elements of entire rows of C, meaning it requires access to every element of B. (See Figure 11.8a.)

If, in contrast, we agglomerate tasks responsible for a square (or nearly square) block of C, the number of elements of B any process needs to access is dramatically reduced.

Let's figure out how much better this scheme is. To simplify the math, let's assume that matrices A, B, and C have dimensions $n \times n$, that p is a square number, and that n is a multiple of $\sqrt{p}$. Each process is responsible for computing an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of matrix C. To compute these elements, the process needs to reference $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ columns of B (See Figure 11.8b.)

Each process still performs an equal share of the computations—$2n^3/p$. The number of elements each process needs access to is $2n(n/\sqrt{p})$. The computation-to-communication ratio is

$$\frac{2n^3/p}{2n^2/\sqrt{p}} = \frac{n}{\sqrt{p}}$$

Let's determine when the computation-to-communication ratio for Cannon's algorithm is superior to the ratio for the "rowwise" algorithm:

$$\frac{n}{\sqrt{p}} > \frac{2n}{p} \Rightarrow \sqrt{p} > 2 \Rightarrow p > 4$$

Cannon's algorithm seems to hold more promise when the number of processes is greater than four.

11.4.2 Communication

Now that we've established the potential for an algorithm based on a checkerboard block decomposition, let's see if we can unlock that potential. First, let's take a look at how A and B are distributed among the processes in a checkerboard block decomposition (Figure 11.9a). Process $P_{i,j}$ contains blocks $A_{i,j}$ and $B_{i,j}$ and is responsible for computing block $C_{i,j}$. Except for the processes on the main diagonal, processes hold blocks of A and B that do not need to be multiplied.

We need to move the blocks around so that every process $P_{i,j}$ has a pair of blocks whose multiplication will contribute to the calculation of $C_{i,j}$. One way to do this is illustrated in Figure 11.9b. Each process in row i of the process mesh cycles its block of A to the process i places to its left. Each process in column j of the process mesh cycles its block of B to the process j places above it. Now

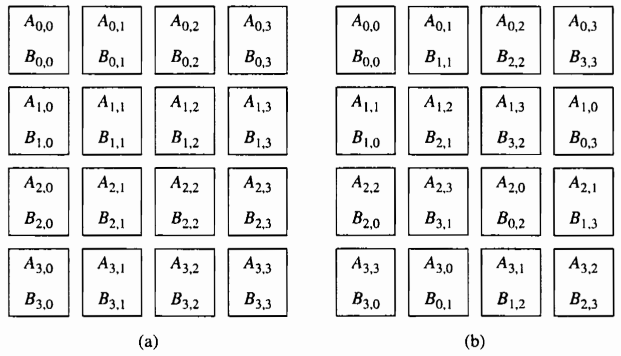


Figure 11.9 Alignment of blocks for matrix multiplication. (a) Initial distribution of blocks among processes. Process $P_{i,j}$ contains blocks $A_{i,j}$ and $B_{i,j}$. The block matrix multiplication algorithm multiplies all pairs $A_{i,k}B_{k,j}$. Note that in the original distribution only the processes on the main diagonal ($P_{0,0}$, $P_{1,1}$, $P_{2,2}$, and $P_{3,3}$) have such pairs. (b) The parallel algorithm cycles each row i of A to the left by i column positions. It cycles each column i of matrix B upward by i row positions. Now every processor $P_{i,j}$ has a pair of blocks to multiply.

we've satisfied our condition: each process can multiply the blocks of A and B it controls to produce a partial result for its block of C.

Recall the size of the process mesh is $\sqrt{p} \times \sqrt{p}$. After the initial step to rearrange the blocks of A and B, the parallel checkerboard matrix multiplication algorithm has $\sqrt{p}$ steps. Each process multiplies the blocks of A and B it controls, adding the result to its partial sum of C. It cycles its block of A to the process to its left, and it receives a new block of A from the process on its right. It cycles its block of B to the process above it, and it receives a new block of B from the process below it. Figure 11.10 illustrates this block-cycling activity from the point of view of process $P_{1,2}$ in a 4×4 process mesh.

11.4.3 Analysis

In this subsection we'll derive an expression for the expected execution time of Cannon's algorithm. To simplify our analysis, we assume that A, B, and C are all $n \times n$ matrices. We also assume that p is a square number and that n is a multiple of $\sqrt{p}$, the number of active processes. Each process is responsible for computing an $(n/\sqrt{p}) \times (n/\sqrt{p})$ portion of C.

First let's consider the computation time. When the algorithm begins, each process initializes its portion of C to 0. During each iteration, every process multiplies an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of A by an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of B and adds the result to its partial result for C. If χ is the time needed for one of the add-multiply steps inside an inner product, the computational time of each iteration is

$$\chi(n/\sqrt{p})^3 = \chi n^3/p^{3/2}$$

The algorithm has $\sqrt{p}$ iterations. Hence the total computation time is (as we should expect)

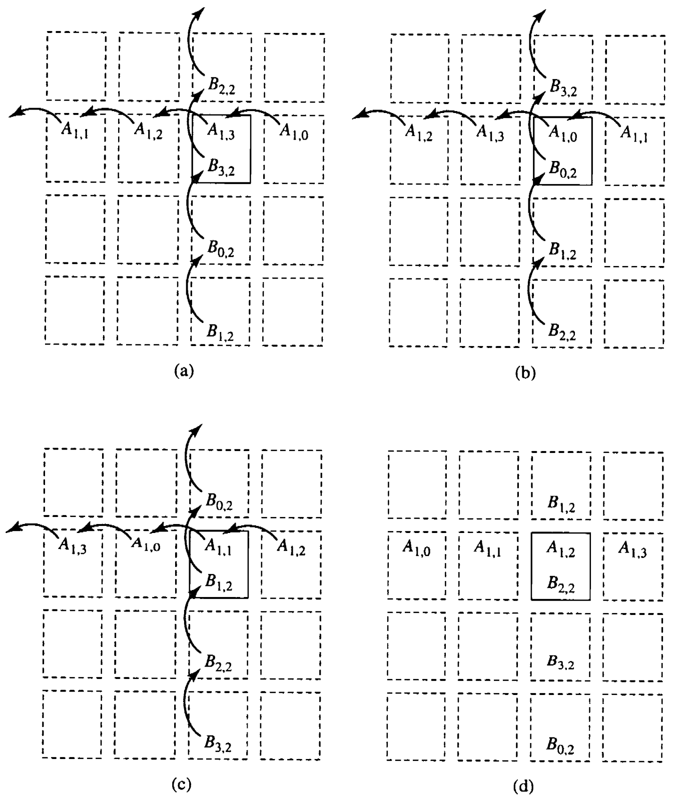
$$\sqrt{p}\chi n^3/p^{3/2} = \chi n^3/p$$

Now let's look at the communication requirements. Before the first iteration, each process must send its blocks of A and B to the appropriate destination processes and receive the blocks of A and B it needs for the first iteration. Our model assumes that messages may be sent and received concurrently, but it allows only a single message at a time to be sent or received. Let $1/\beta$ be the time needed to transmit a single matrix element. The time needed for the initial block distribution is

$$2\left(\lambda + \frac{n^2}{p\beta}\right)$$

During each of the $\sqrt{p}$ iterations, every process must pass along its A and B blocks and receive new blocks to multiply. The total time required for these steps is

$$2\sqrt{p}\left(\lambda + \frac{n^2}{p\beta}\right)$$



$B_{2,2}$
$A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{1,0}$
$B_{3,2}$
$B_{0,2}$
$B_{1,2}$
(a)
$B_{3,2}$
$A_{1,2}$ $A_{1,3}$ $A_{1,0}$ $A_{1,1}$
$B_{0,2}$
$B_{1,2}$
$B_{2,2}$
(b)
$B_{0,2}$
$A_{1,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$
$B_{1,2}$
$B_{2,2}$
$B_{3,2}$
(c)
$B_{1,2}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$B_{2,2}$
$B_{3,2}$
$B_{0,2}$
(d)

Figure 11.10 Cannon's matrix multiplication algorithm from the point of view of process $P_{1,2}$. Note that processes are organized into a 2-D mesh, and each process has already sent its blocks of A and B to the process that needs them for the first iteration. (a) First block multiplication step. After each block multiplication process $P_{1,2}$ sends its block of A to the process on its left and receives a new block of A from the process on its right. Similarly, it sends its block of B to the process above it and receives a new block of B from the process below it. (b) Second block matrix multiplication step. (c) Third block matrix multiplication step. (d) Final block matrix multiplication step. Summing the results of all block matrix multiplications yields $C_{1,2}$.

Adding these three terms, our expression for the expected overall execution time of Cannon's algorithm is

$$\chi n^3/p + 2(\sqrt{p}+1)\left(\lambda + \frac{n^2}{p\beta}\right)$$

What is the isoefficiency of Cannon's algorithm? The sequential algorithm has time complexity $\Theta(n^3)$. The communication complexity of the parallel algorithm is $\Theta(n^2/\sqrt{p})$. We multiply the community complexity by the number of processors p to get the overhead term: $T_o(n,p) = \Theta(\sqrt{p}n^2)$. Hence the isoefficiency relation for the rowwise block-striped matrix multiplication algorithm is

$$n^3 \geq C\sqrt{p}n^2 \Rightarrow n \geq C\sqrt{p}$$

Recall $M(n) = n^2$. Hence the scalability function is:

$$M(C\sqrt{p})/p = C^2p/p = C^2$$

Because constant memory utilization per processor is sufficient to maintain efficiency as processors are added, we conclude Cannon's algorithm is highly scalable.

As in the case of the row-oriented algorithm, Cannon's algorithm presents a good opportunity for overlapping communications with computation. If there is enough memory to buffer new A and B blocks while working on the current blocks, each process can initiate its sends and receives of A and B blocks before starting the matrix multiplication for that iteration. After the matrix multiplication step, the process can check for the completion of the message receives before starting the next iteration. Since the communication complexity is $\Theta(n^2)$ and the computational complexity is $\Theta(n^3)$, the communication step can be almost completely overlapped with computations when the matrix sizes are large enough.

11.5 SUMMARY

In this chapter we have developed two parallel algorithms for matrix multiplication. The first algorithm is based on a rowwise block-striped matrix decomposition, while the second (Cannon's algorithm) is based on a checkerboard block matrix decomposition. Both algorithms divide the computations evenly among the processes. Cannon's algorithm, however, requires less communication among processes. Isoefficiency analysis reveals that Cannon's algorithm is highly scalable, while the first is not. If sufficient memory is available, both algorithms can benefit from communication/computation overlapping.

We also explored performance issues related to sequential matrix multiplication. The straightforward algorithm has a memory reference pattern that results in a poor cache hit rate once the second factor matrix no longer fits in cache. We presented a recursive matrix multiplication algorithm that divides matrices into blocks when the matrices are too large to fit in cache. We showed how a program based on this algorithm maintains high CPU performance, even as the matrix sizes grow beyond the cache limits.

In order to achieve best performance, parallel programs performing matrix multiplication should rely upon a high-speed sequential matrix multiplication function, such as the recursive function presented in this chapter, when multiplying submatrices.

11.6 KEY TERMS

Cannon's algorithm

11.7 BIBLIOGRAPHIC NOTES

In this chapter we showed how a recursive matrix multiplication algorithm led to an improved cache hit rate. Recursion is often an effective variable blocking technique for dense linear algebra algorithms, as pointed out by Gustavson [48].

11.8 EXERCISES

11.1 Suppose $A = \begin{pmatrix} 1 & 2 & -3 & -2 \\ 4 & 1 & -1 & 3 \\ 3 & 2 & 1 & -4 \end{pmatrix}$ and $B = \begin{pmatrix} -2 & -3 \\ 3 & 2 \\ 4 & -1 \\ 1 & -4 \end{pmatrix}$

a. Compute $C = AB$.

b. Consider the submatrices

$A_{00} = \begin{pmatrix} 1 & 2 \end{pmatrix}$ $A_{01} = \begin{pmatrix} -3 & -2 \end{pmatrix}$

$A_{10} = \begin{pmatrix} 4 & 1 \\ 3 & 2 \end{pmatrix}$ $A_{11} = \begin{pmatrix} -1 & 3 \\ 1 & -4 \end{pmatrix}$

and

$B_{00} = \begin{pmatrix} -2 \\ 3 \end{pmatrix}$ $B_{01} = \begin{pmatrix} -3 \\ 2 \end{pmatrix}$

$B_{10} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$ $B_{11} = \begin{pmatrix} -1 \\ -4 \end{pmatrix}$

Compute $C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{11} + A_{11}B_{11} \end{pmatrix}$.

Show the result of each block matrix multiplication.

11.2 In the parallel matrix multiplication algorithm based upon a rowwise block-striped matrix decomposition, each process ends up multiplying its portion of A by the entire matrix B. If we replicated B across all processes, it would greatly simplify the algorithm. What is the fundamental problem with this approach?

11.3 Both the rowwise algorithm and Cannon's algorithm can call the recursive sequential matrix multiplication algorithm as a subroutine when multiplying their portions of A and B.

a. Why is Cannon's algorithm a better match for the recursive sequential matrix multiplication algorithm than the algorithm based on a rowwise striped decomposition?

b. Design a modification to the recursive sequential matrix multiplication algorithm that addresses the problem raised in part (a).

11.4 Consider the optimization of overlapping communication steps with computation steps in the two parallel matrix multiplication algorithms discussed in this chapter. Suppose $p = 16$, $\beta = 1.5 \times 10^6$/sec, $\lambda = 250$ μsec, and $\chi = 10$ nanosec.

a. For what values of n can we expect the communication time per iteration of the rowwise algorithm to be less than the computation time?

b. For what values of n can we expect the communication time per iteration of Cannon's algorithm to be less than the computation time?

11.5 Write a program implementing the parallel matrix multiplication algorithm described in Section 11.3. The program should read the factor matrices from files and write the product matrix to a file; the names of the files should be specified on the command line. Assume the matrices contain double-precision floating-point values. They should be stored in the files according to the protocol first described in Chapter 6: two integers m and n, indicating the number of matrix rows and columns, respectively, followed by mn double-precision floating-point values.

a. Benchmark your program on $1, 2, 3, 4, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, ignoring file I/O time. Plot the four speedup curves on a graph.

b. Benchmark your program on $1, 2, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, taking into account file I/O time. Plot the four speedup curves on a graph.

11.6 Write a program implementing Cannon's algorithm described in Section 11.4, assuming that the number of processes executing the program is a square number. The program should read the factor matrices from files and write the product matrix to a file; the names of the files should be specified on the command line. Assume the matrices contain double-precision floating-point values. They should be stored in the files according to the protocol first described in Chapter 6: two integers m and n, indicating the number of matrix rows and columns, respectively, followed by mn double-precision floating-point values.

a. Benchmark your program on $1, 4, 9, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, ignoring file I/O time. Plot the four speedup curves on a graph.

b. Benchmark your program on $1, 4, 9, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, taking into account file I/O time. Plot the four speedup curves on a graph.

11.7 Design a version of Cannon's algorithm that works when the number of processes is not a square number.

11.8 Write a parallel program based on Cannon's algorithm that takes advantage of all processes available, even when the number of processes is not a square number. The program should read the factor matrices from files and write the product matrix to a file; the names of the files should be specified on the command line. Assume the matrices contain double-precision floating-point values. They should be stored in the files according to the protocol first described in Chapter 6: two integers m and n, indicating the number of matrix rows and columns, respectively, followed by mn double-precision floating-point values.

a. Benchmark your program on $1, 2, 3, 4, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, ignoring file I/O time. Plot the four speedup curves on a graph.

b. Benchmark your program on $1, 2, 3, 4, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, taking into account file I/O time. Plot the four speedup curves on a graph.