Problems

P5.4.1 Modify FFTRecur so that it does not involve redundant weight computation.

P5.4.2 Modify **strass** so that it can handle general n. Hint: You will have to figure out how to partition the matrix multiplication if n is odd.

P5.4.3 Let $f(q,r)$ be the number of flops required by **strass** if $n = 2^q$ and $n_{min} = 2^r$. Assume that $q \geq r$. Develop an analytical expression for $f(q,r)$. For $q = 1{:}20$, compare $\min_{r \leq s} f(q,r)$ and $2 \cdot (2^q)^3$.

5.5 Distributed Memory Matrix Multiplication

In a shared memory machine, individual processors are able to read and write data to a (typically large) global memory. A snapshot of what it is like to compute in a shared memory environment is given at the end of Chapter 4, where we used the quadrature problem as an example.

In contrast to the shared memory paradigm is the *distributed memory* paradigm. In a distributed memory computer, there is an *interconnection network* that links the processors and is the basis for communication. In this section we discuss the parallel implementation of matrix multiplication, a computation that is highly parallelizable and provides a nice opportunity to contrast the shared and distributed memory approaches.[1]

5.5.1 Networks and Communication

We start by discussing the general set-up in a distributed memory environment. Popular interconnection networks include the ring and the mesh. See Figs. 5.3 and 5.4.

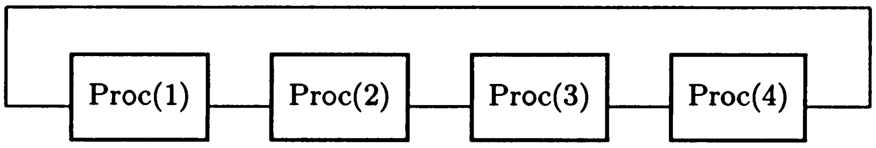


FIGURE 5.3 A ring multiprocessor

The individual processors come equipped with their own processing units, local memory, and input/output ports. The act of designing a parallel algorithm for a distributed memory system is the act of designing a *node program* for each of the participating processors. As we shall see, these programs look like ordinary programs with occasional **send** and **recv** commands that are used for the sending and receiving of messages. For us, a message is a matrix. Since vectors and scalars are special matrices, a message may consist of a vector or a scalar.

In the following we suppress very important details such as (1) how data and programs are downloaded into the nodes, (2) the subscript computations associated with local array access,

[1]The distinction between shared memory multiprocessors and distributed memory multiprocessors is fuzzy. A shared memory can be physically distributed. In such a case, the programmer has all the convenience of being able to write node programs that read and write directly into the shared memory. However, the physical distribution of the memory means that either the programmer or the compiler must strive to access "nearby" memory as much as possible.

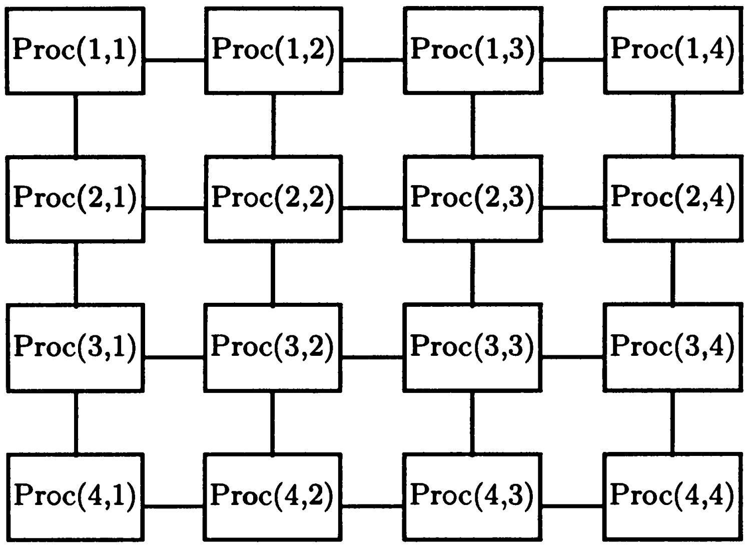


FIGURE 5.4 A mesh multiprocessor

and (3) the formatting of messages. We designate the μth processor by Proc(μ). The μth node program is usually a function of μ.

The distributed memory paradigm is quite general. At the one extreme we can have networks of workstations. On the other hand, the processors may be housed on small boards that are stacked in a single cabinet that sits on a desk.

The kinds of algorithms that can be efficiently solved on a distributed memory multiprocessor are defined by a few key properties:

- *Local memory size.* An interesting aspect of distributed memory computing is that the amount of data for a problem may be so large that it cannot fit inside a single processor. For example, a 10,000-by-10,000 matrix of floating point numbers requires almost a thousand megabytes of storage. The storage of such an array may require the local memories from hundreds of processors. Local memory often has a hierarchy, as does the memory of conventional computers (e.g., disks $\rightarrow$ slow random access memory $\rightarrow$ fast random access memory, etc.)

- *Processor speed.* The speed at which the individual processing units execute is of obvious importance. In sophisticated systems, the individual nodes may have vector capabilities, meaning that the extraction of peak performance from the system requires algorithms that vectorize at the node level. Another complicating factor may be that system is made up of different kinds of processors. For example, maybe every fourth node has a vector processing accelerator.

- *Message passing overhead.* The time it takes for one processor to send another processor a message determines how often a node program will want to break away from productive calculation to receive and send data. It is typical to model the time it takes to send or receive an n-byte message by

$$T(n) = \alpha + \beta n \tag{5.5.1}$$

Here, α is the *latency* and β is the *bandwidth*. The former measures how long it takes to "get ready" for a send/receive while the latter is a reflection of the "size" of the wires that connect the nodes. This model of communication provides some insight into performance, but it is seriously flawed in at least two regards. First, the proximity of the receiver to the sender is usually an issue. Clearly, if the sender and receiver are neighbors in the network, then the system software that routes the message will not have so much to do. Second, the message passing software or hardware may require the breaking up of large messages into smaller packets. This takes time and magnifies the effect of latency.

The examples that follow will clarify some of these issues.

5.5.2 A Two-processor Matrix Multiplication

Consider the matrix-matrix multiplication problem

$$C \leftarrow C + AB$$

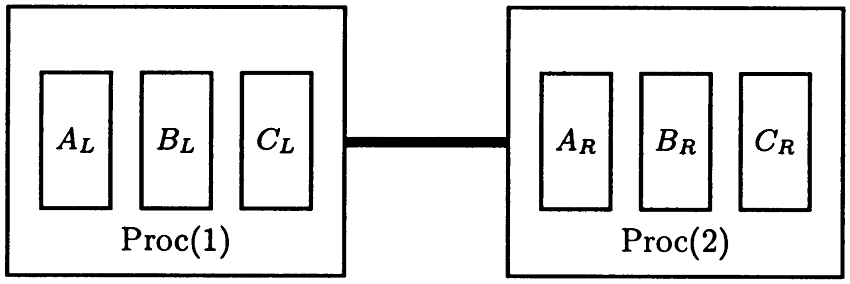
where the three matrices $A, B, C \in \mathbb{R}^{n\times n}$ are distributed in a two-processor distributed memory network. To be specific, assume that $n = 2m$ and that Proc(1) houses the "left halves"

$$A_L = A(:, 1{:}m) \quad B_L = B(:, 1{:}m) \quad C_L = C(:, 1{:}m)$$

and that Proc(2) houses the "right halves"

$$A_R = A(:, m+1{:}n) \quad B_R = B(:, m+1{:}n) \quad C_R = C(:, m+1{:}n)$$

Pictorially we have



We assign to Proc(1) and Proc(2) the computation of the new C_L and C_R, respectively. Let's see where the data for these calculations come from. We start with a complete specification of the overall computation:

```
for j=1:n
   C(:,j) = C(:,j) + A*B(:,j);
end
```

Note that the jth column of the updated C is the jth column of the original C plus A times the jth column of B. The matrix-vector product `A*B(:,j)` can be expressed as a linear combination of A-columns:

$$\mathtt{A} * \mathtt{B}(:,\mathtt{j}) = \mathtt{A}(:,1) * \mathtt{B}(1,\mathtt{j}) + \mathtt{A}(:,2) * \mathtt{B}(2,\mathtt{j}) + \cdots + \mathtt{A}(:,\mathtt{n}) * \mathtt{B}(\mathtt{n},\mathtt{j}).$$

Thus the preceding fragment expands to

```
for j=1:n
   for k=1:n
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

Note that Proc(1) is in charge of

```
for j=1:m
   for k=1:n
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

while Proc(2) must carry out

```
for j=m+1:n
   for k=1:n
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

From the standpoint of communication, there is both good news and bad news. The good news is that the B-data and C-data required are local. Proc(1) requires (and has) the left portions of C and B. Likewise, Proc(2) requires (and has) the right portions of these same matrices. The bad news concerns A. Both processors must "see" all of A during the course of execution. Thus, for each processor exactly one half of the A-columns are nonlocal, and these columns must somehow be acquired during the calculation. To highlight the local and nonlocal A-data, we pair off the left and the right A-columns:

```
Proc(1) does this:                                    Proc(2) does this:

for j=1:m                                             for j=m+1:n
   for k=1:m                                             for k=1:m
      C(:,j) = C(:,j) + A(:,k)*B(k,j);                      C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
      C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);                  C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end                                                   end
end                                                   end
```

In each case, the second update of `C(:,j)` requires a nonlocal A-column. Somehow, Proc(1) has to get hold of $A(:,k+m)$ from Proc(2). Likewise, Proc(2) must get hold of $A(:,k)$ from Proc(1).

The primitives **send** and **recv** are to be used for this purpose. They have the following syntax:

send({*matrix*} , {*destination node*}) **recv**({*matrix*} , {*source node*})

If a processor invokes a **send**, then we assume that execution resumes immediately after the message is sent. If a **recv** is encountered, then we assume that execution of the node program is suspended until the requested message arrives. We also assume that messages arrive in the same order that they are sent.[2]

With **send** and **recv**, we can solve the nonlocal data problem in our two-processor matrix multiply:

```
Proc(1) does this:

for j=1:m
    for k=1:m
        send(A(:,k),2);
        C(:,j) = C(:,j) + A(:,k)*B(k,j);
        recv(v,2);
        C(:,j) = C(:,j) + v*B(k+m,j);
    end
end
```

```
Proc(2) does this:

for j=m+1:n
    for k=1:m
        send(A(:,k+m),1);
        C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
        recv(v,1);
        C(:,j) = C(:,j) + v*B(k,j);
    end
end
```

Each processor has a local work vector v that is used to hold an A-column solicited from its neighbor. The correctness of the overall process follows from the assumption that the messages arrive in the order that they are sent. This ensures that each processor "knows" the index of the incoming columns. Of course, this is crucial because incoming A-columns have to scaled by the appropriate B entries.

Although the algorithm is correct and involves the expected amount of floating point arithmetic, it is inefficient from the standpoint of communication. Each processor sends a given local A-column to its neighbor m times. A better plan is to use each incoming A-column in all the $C(:,j)$ updates "at once." To do this, we merely reorganize the order in which things are done in the node programs:

```
Proc(1) does this:

for k=1:m
    send(A(:,k),2);
    for j=1:m
        C(:,j) = C(:,j) + A(:,k)*B(k,j);
    end;
    recv(v,2);
    for j=1:m
        C(:,j) = C(:,j) + v*B(k+m,j);
    end
end
```

```
Proc(2) does this:

for k=1:m
    send(A(:,k+m),1);
    for j=m+1:n
        C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
    end
    recv(v,1);
    for j=m+1:n
        C(:,j) = C(:,j) + v*B(k,j);
    end
end
```

Although there is "perfect symmetry" between the two node programs, we cannot assume that they proceed in lock-step fashion. However, the program is *load balanced* because each processor has roughly the same amount of arithmetic and communication.

[2]This need not be true in practice. However, a system of tagging messages can be incorporated that can be used to remove ambiguities.

In the preceding, the matrices A, B, and C are accessed as if they were situated in a single memory. Of course, this will not be the case in practice. For example, Proc(2) will have a local n-by-m array to house its portion of C. Let's call this array `C.loc` and assume that `C.loc(i,j)` houses matrix element $C(i, j + m)$. Similarly, there will be local array `A.loc` and `B.loc` that house their share of A and B, respectively. If we rewrite Proc(2)'s node program in the true "language" of its local arrays, then it becomes

```
for k=1:m
   send(A.loc(:,k),1);
   for j=1:m
      C.loc(:,j) = C.loc(:,j) + A.loc(:,k)*B.loc(k,j);
   end
   recv(v,1);
   for j=1:m
      C.loc(:,j) = C(:,j) + v*B.loc(k+m,j);
   end
end
```

We merely mention this to stress that what may be called "subscript reasoning" undergoes a change when working in distributed memory environments.

5.5.3 Performance Analysis

Let's try to anticipate the time required to execute the communication-efficient version of the two-processor matrix multiply. There are two aspects to consider: computation and communication. With respect to computation, we note first that overall calculation involves $2n^3$ flops. This is because there are n^2 entries in C to update and each update requires the execution of a length n inner product (e.g., $C(i,j) = C(i,j) + A(i,:) * B(:,j)$). Since an inner product of that length involves n adds and n multiplies, $n^2(2n)$ flops are required in total. These flops are distributed equally between the two processors. If computation proceeds at a uniform rate of R flops per second, then

$$T_{comp} = n^3/R$$

seconds are required to take care of the arithmetic.[3]

With respect to communication costs, we use the model (5.5.1) and conclude that each processor spends

$$T_{comm} = n(\alpha + 8\beta n)$$

seconds sending and receiving messages. (We assume that each floating point number has an 8-byte representation.) Note that this is not the whole communication overhead story because we are not taking into account the idle wait times associated with the receives. (The required vector may not have arrived at time of the `recv`.) Another factor that complicates performance evaluation is that each node may have a special input/output processor that more or less handles communication making it possible to overlap computation and communication.

[3]Recall the earlier observation that with many advanced architectures, the execution rate varies with the operation performed and whether or not it is vectorized.

Ignoring these possibly significant details leads us to predict an overall execution time of $T = T_{comp} + T_{comm}$ seconds. It is instructive to compare this time with what would be required by a single-processor program. Look at the ratio

$$S = \frac{2n^3/R}{(n^3/R) + n(\alpha + 8\beta n)} = \frac{2}{1 + R\left(\frac{\alpha}{n^2} + \frac{8\beta}{n}\right)}$$

S represents the *speed-up* of the parallel program. We make two observations: (1) Communication overheads are suppressed as n increases, and (2) if α and β are fixed, then speed-up decreases as the rate of computation R improves.

In general, the speed-up of a parallel program executing on p processors is a ratio:

$$\text{Speed-up} = \frac{\text{Time required by the best single-processor program}}{\text{Time required for the } p\text{-processor implementation}}$$

In this definition we do not just set the numerator to be the $p = 1$ version of the parallel code because the best uniprocessor algorithm may not parallelize. Ideally, one would like the speed-up for an algorithm to equal p.

Problems

P5.5.1 Assume that $n = 3m$ and that the n-by-n matrices A, B, C are distributed around a three-processor ring. In particular, assume that processors 1, 2, and 3 house the left third, middle third, and right third of these matrices. For example, $B(:, m+1{:}2m)$ would be housed in Proc(2). Write a parallel program for the computation $C \leftarrow C + AB$.

P5.5.2 Suppose we have a two-processor distributed memory system in which floating point arithmetic proceeds at R flops per second. Assume that when one processor sends or receives a message of k floating point numbers, then $\alpha + \beta k$ seconds are required. Proc(1) houses an n-by-n matrix A, and each processor houses a copy of an n-vector x. The goal is to store the vector $y = Ax$ in Proc(1)'s local memory. You may assume that n is even. **(1)** How long would this take if Proc(1) handles the entire computation itself? **(2)** Describe how the two processors can share the computation. Indicate the data that must flow between the two processors and what they must each calculate. You do not have to write formal node programs. Clear concise English will do. **(3)** Does it follow that if n is large enough, then it is more efficient to distribute the computation? Justify your answer.

M-Files and References

Script Files

`CircBench`	Benchmarks Circulant1 and Circulant2.
`MatBench`	Benchmarks various matrix-multiply methods.
`AveNorms`	Compares various norms on random vectors.
`ProdBound`	Examines the error in three-digit matrix multiplication.
`ShowContour`	Displays various contour plots of SampleF.
`Show2dQuad`	Illustrates CompQNC2D.
`FFTflops`	Compares FFT and DFT flops.
`StrassFlops`	Examines Strassen multiply.