Chapter 5

Algorithms on Grids of Processors

5.1 Logical Two-Dimensional Grid Topologies

In Chapter 4 we have developed algorithms on a ring of processors. In this chapter we motivate and demonstrate the use of a more complex logical topology: a two-dimensional (2-D) grid, or grid for short. Figure 5.1(a) shows an example square grid with $p = q^2$ processors. Processors in the corners have only two neighbors, processors on the edges have three, and processors in the middle have four. Processors are indexed by their row and column in the grid, $P_{i,j}, 0 \leq i, j < p$. One popular variation of the grid topology is obtained by adding loopbacks from edge to edge, forming what is commonly called a 2-D torus, or torus for short, as shown in Figure 5.1(b). In this case every processor belongs to two different rings. As in the case of a ring topology, links can be either unidirectional or bidirectional. Choosing an appropriate flavor of the grid topology is up to the parallel algorithm developer, but we will see that a bidirectional torus proves very convenient and this is the topology we will use by default. For the sake of simplicity we will always assume that the number of processors is a perfect square, leading to square grids. The algorithms presented in this chapter can be adapted to rectangular grids, which often require only more cumbersome index calculations.

But as with a ring, an assumption here is that communication can occur in parallel on different links. So, using the grid in Figure 5.1(b), $P_{i,j}$ can send data to $P_{i,j}$ while $P_{i,j}$ sends data to $P_{i,j}$, and computing apply (see Section 3.3). For all the algorithms developed in this chapter, we will assume that each processor can be engaged in computing, sending, and receiving concurrently as long as there are no race conditions.

Two new issues arise with a bidirectional grid topology.

The first issue is that of a processor concurrently sending and receiving on the same bidirectional link. We were careful that such concurrent communication on a single link did not arise in our use of a bidirectional link for the stencil application described in Section 4.5.2. In this chapter, we will make the assumption that links are full-duplex, meaning that communications can flow both ways on each link without contention. In other terms, a processor can concurrently send and receive a given amount of data on a link in the same time as it can send (or receive) that amount of data. This assumption may or may not be realistic for the underlying physical platform. It is straightforward to modify the programs presented in this chapter and, more importantly, their performance analyses, in case the full-duplex assumption does not hold.

The second issue is that of the number of communications in which a single processor can be engaged simultaneously. With four bidirectional links, conceivably a processor can be involved in one send and one receive on all its network links, all concurrently. The assumption that such concurrent communications are allowed at each processor with no decrease in communication speed when compared to a single communication is termed the multi-port model. In the case of our grid topology, we talk of a 4-port model. If instead at most two concurrent communications are allowed, one of them being a send and one of them being a receive, then one talks of a 1-port model. Going back to our stencil algorithm on a bidirectional ring (Section 4.5.2), the reader will see that we had implicitly used the 1-port model. In this chapter, we will show performance analyses for both the 1-port and the 4-port model. Once again, it is typically straightforward to adapt these analyses to other assumptions regarding concurrency of communications.

As discussed at the end of Chapter 4, an important issue is impedance matching between logical and physical topologies. How do the grid and ring logical topologies compare in terms of realism for a given physical platform? It turns out that there are platforms whose physical topologies are or include grids and/or rings. A famous example of a supercomputer using a grid is IBM’s Blue Gene/L supercomputer and its three-
dimensional torus topology, which contains grids and rings. When both a ring and a grid map well to the physical platform, the grid is preferable for many algorithms. For a given number of processors \( p \), a torus topology uses \( 2p \) network links (and a grid topology \( 2(p - \sqrt{p}) \)) twice more than a ring topology, which uses only \( p \) network links. As a result, more communications can occur in parallel and there are more opportunities for developing parallel algorithms with lower communication costs. Interestingly, even on platforms whose physical topology does not contain a grid (e.g., on a platform using a switched interconnect), using a logical grid topology can allow for more concurrent communications. In this chapter, we will see that the opportunity for concurrent communications is the key advantage of logical grid topologies for implementing popular algorithms. But we will also see that even if the underlying platform offers no possibility of concurrent communications, writing some algorithms assuming a grid topology is inherently beneficial.

### 5.2 Communication on a Grid of Processors

In this section, we define communication primitives and convenient functions that we will use in the upcoming sections to write parallel algorithms on grids of processors. Processor \( P_{i,j} \), \( 0 \leq i, j < q \), is said to be in processor row \( i \) and to be in processor column \( j \). A processor can obtain the indices of its processor row and column via the two following functions:

\[
\text{\texttt{MyProcRow}}() \quad \text{and} \quad \text{\texttt{MyProcCol}}().
\]

A processor can determine the total number of processors, \( p = q^2 \), by calling function \( \text{\texttt{NumProc}}() \). Since we assume square grids, this function makes it possible to know the number of processors in each processor row or column. Two separate functions may be needed in the case of rectangular grids.

A processor can send a message of \( L \) data items stored at address \( \text{\texttt{srcaddr}} \) to one of its neighbor processors by calling the following primitive:

\[
\text{\texttt{Send}}(\text{\texttt{dest}}, \text{\texttt{addr}}, L),
\]

where \( \text{\texttt{dest}} \) has value \( \text{\texttt{North}}, \text{\texttt{South}}, \text{\texttt{West}}, \text{\texttt{East}} \). For a torus topology, which is the topology we will use for the majority of the algorithms in this chapter, the \( \text{\texttt{North}}, \text{\texttt{South}}, \text{\texttt{West}}, \text{\texttt{East}} \) neighbors of processor \( P_{i,j} \) are \( P_{i+1,j} \mod q, P_{i-1,j} \mod q, P_{i,j+1} \mod q, \text{and} P_{i,j-1} \mod q \), respectively. We often omit the modulo and assume that all processor indices are taken modulo \( q \) implicitly. If the topology is a grid, then some dest values are not allowed for some source processors. Each \( \text{\texttt{Send}} \) call has a matching \( \text{\texttt{Recv}} \) call:

\[
\text{\texttt{Recv}}(\text{\texttt{src}}, \text{\texttt{addr}}, L).
\]

Let us assume that the grid is square and contains \( p = q^2 \) processors, that matrices are also square of dimension \( n \times n \), and that \( q \) divides \( n \). In the case of a torus, we have the additional problem of identifying a starting point (or seed processor) for a broadcast. This problem is solved by using either the \( \text{\texttt{Send}} \) and \( \text{\texttt{Recv}} \) primitives or the two functions \( \text{\texttt{MyProcRow}}() \) and \( \text{\texttt{MyProcCol}}() \). In either case, we can use the following broadcast algorithm:

\[
\text{\texttt{Broadcast}}(\text{\texttt{src}}, \text{\texttt{srcaddr}}, L).
\]

where \( \text{\texttt{srcaddr}} \) is the address in the memory of processor \( P_{i,j} \) of a message of length \( L \) that is to be sent. Once received, the message is stored at address \( \text{\texttt{destaddr}} \) in the memory of all involved processors. In the same spirit as for the broadcast function described in Chapter 3, all processors in the processor row place the same call and the source processor sends a message to itself. We assume a similar function for broadcasting within processor column \( j \):

\[
\text{\texttt{BroadcastCol}}(\text{\texttt{src}}, \text{\texttt{srcaddr}}, \text{\texttt{destaddr}}, L).
\]

Note that in the case of a torus, each processor row and processor column is a ring embedded in the processor grid. Therefore, the above two functions can use the pipelined implementation of the broadcast on a ring developed in Section 3.3.4. If, in addition, links are bidirectional and one assumes a 4-port model (or in fact just a 2-port model in this case), then the broadcast can be done faster by sending data from the source processor in both directions simultaneously. We will see later in this chapter that this does not change the asymptotic performance of the broadcast. If the topology is not a torus but links are bidirectional, then this broadcast can be implemented by sending messages both ways from the source processor. If the topology is not a torus and links are unidirectional, then these functions cannot be implemented. We assume that a processor that calls these functions and is not in the relevant processor row or column returns immediately. This assumption will simplify the pseudo-code of our algorithms by removing the need for processor row and column indices before calling \( \text{\texttt{BroadcastRow}}() \) or \( \text{\texttt{BroadcastCol}}() \).

### 5.3 Matrix Multiplication on a Grid of Processors

In this section, we present four popular algorithms for computing the matrix product \( C = A \times B \) on a square grid of processors. The first question that arises is that of the distribution of the matrices among the processors. Let us assume that the grid is square and contains \( p = q^2 \) processors, that matrices are also square of dimension \( n \times n \), and that \( q \) divides \( n \). In the case of a torus, we have the additional problem of identifying a starting point (or seed processor) for a broadcast. This problem is solved by using either the \( \text{\texttt{Send}} \) and \( \text{\texttt{Recv}} \) primitives or the two functions \( \text{\texttt{MyProcRow}}() \) and \( \text{\texttt{MyProcCol}}() \). In either case, we can use the following broadcast algorithm:

\[
\text{\texttt{Broadcast}}(\text{\texttt{src}}, \text{\texttt{srcaddr}}, L).
\]

where \( \text{\texttt{srcaddr}} \) is the address in the memory of processor \( P_{i,j} \) of a message of length \( L \) that is to be sent. Once received, the message is stored at address \( \text{\texttt{destaddr}} \) in the memory of all involved processors. In the same spirit as for the broadcast function described in Chapter 3, all processors in the processor row place the same call and the source processor sends a message to itself. We assume a similar function for broadcasting within processor column \( j \):

\[
\text{\texttt{BroadcastCol}}(\text{\texttt{src}}, \text{\texttt{srcaddr}}, \text{\texttt{destaddr}}, L).
\]

Note that in the case of a torus, each processor row and processor column is a ring embedded in the processor grid. Therefore, the above two functions can use the pipelined implementation of the broadcast on a ring developed in Section 3.3.4. If, in addition, links are bidirectional and one assumes a 4-port model (or in fact just a 2-port model in this case), then the broadcast can be done faster by sending data from the source processor in both directions simultaneously. We will see later in this chapter that this does not change the asymptotic performance of the broadcast. If the topology is not a torus but links are bidirectional, then this broadcast can be implemented by sending messages both ways from the source processor. If the topology is not a torus and links are unidirectional, then these functions cannot be implemented. We assume that a processor that calls these functions and is not in the relevant processor row or column returns immediately. This assumption will simplify the pseudo-code of our algorithms by removing the need for processor row and column indices before calling \( \text{\texttt{BroadcastRow}}() \) or \( \text{\texttt{BroadcastCol}}() \).
of a one-dimensional (1-D) topology, i.e., a ring, we had used a natural 1-D data distribution. Here, our 2-D topology naturally induces a 2-D data distribution. We define \( m = n/p \). The standard approach is to assign an \( m \times m \) block of each matrix to each processor according to the grid topology. More precisely, processor \( P_{i,j} \) for \( 0 \leq i,j < q \), holds matrix elements \( A_{i,j} \), \( B_{i,j} \), and \( C_{i,j} \) with \( i,m \leq k < (i+1)m \) and \( j,m \leq l < (j+1)m \). We denote the three matrix blocks assigned to processor \( P_{i,j} \) by \( A_{i,j} \), \( B_{i,j} \), and \( C_{i,j} \), as depicted in Figure 5.2 for matrix \( A \). All algorithms hereafter use this distribution scheme.

5.3.1 The Outer-Product Algorithm

While the standard algorithm for matrix multiplication is often written as a sequence of inner product computations (as in Section 4.2), operations can be ordered in many different ways. One option is to write the algorithm as a series of outer products, which amounts to simply switching the order of the loops. Assuming that all elements of matrix \( C \) are initialized to zero, the sequential algorithm is written as

\[
\begin{align*}
&\text{for } k = 0 \text{ to } n - 1 \text{ do} \\
&\quad \text{for } i = 0 \text{ to } n - 1 \text{ do} \\
&\quad \quad \text{for } j = 0 \text{ to } n - 1 \text{ do} \\
&\quad \quad \quad C_{i,j} = C_{i,j} + A_{i,k} \times B_{k,j}
\end{align*}
\]

It turns out that this so-called “outer-product algorithm” \([1, 55, 76]\) leads to a particularly simple and elegant parallelization on a torus of processors. The algorithm proceeds in \( n \) steps, that is, \( n \) iterations of the outer loop. At each step \( k \), \( C_{i,j} \) is updated using \( A_{i,k} \) and \( B_{k,j} \). Recall that all three matrices are partitioned in \( p^2 \) blocks of size \( m \times m \), as on the right side of Figure 5.2.

![Figure 5.2: 2-D block distribution of an \( n \times n \) matrix (\( n = 24 \)) on a uni-directional grid/torus of \( p \) processors (\( p = 4^2 = 16 \)].

The algorithm above can be written in terms of matrix blocks and of matrix multiplications, and it proceeds in \( q \) steps as follows:

\[
\begin{align*}
&\text{for } k = 0 \text{ to } q - 1 \text{ do} \\
&\quad \text{for } i = 0 \text{ to } q - 1 \text{ do} \\
&\quad \quad \text{for } j = 0 \text{ to } q - 1 \text{ do} \\
&\quad \quad \quad C_{i,j} = C_{i,j} + A_{i,k} \times B_{k,j}
\end{align*}
\]

Now consider the execution of this algorithm on a torus of \( p = p_1^2 \) processors. Processor \( P_{i,j} \) holds block \( C_{i,j} \) and is responsible for updating this block at each step of the above algorithm. To perform this update at step \( k \), processor \( P_{i,j} \) needs blocks \( A_{i,k} \) and \( B_{i,k} \). At step \( k \), processor \( P_{i,j} \) already happens to hold \( A_{i,k} \). For all other steps, \( P_{i,j} \) must receive \( A_{i,k} \) from the processor that holds it, that is, \( P_{i,k} \). This is true for all \( P_{i,j} \) with \( j \neq k \). Therefore, at step \( k \), processor \( P_{i,k} \) must broadcast its block of matrix \( A \) to all processors \( P_{i,j} \) with \( j \neq k \), that is, all processors that are on processor \( P_{i,j} \)'s processor row. This is true for all \( i \). Similarly, blocks of matrix \( B \) must be broadcasted at step \( k \) by \( P_{i,j} \) to all processors on its processor column, for all \( j \). The resulting communication pattern is illustrated in Figure 5.3. The figure shows which blocks of matrices \( A \) and \( B \) are sent to which processors at step \( k = 1 \) of the algorithm in the case of a \( 4 \times 4 \) torus. For instance, block \( A_{2,2} \), which is held by processor \( P_{2,2} \), is sent to processors \( P_{2,0}, P_{2,1}, P_{2,3}, \) and \( P_{2,3} \).

![Figure 5.3: Communications of blocks of matrices \( A \) and \( B \) at step \( k = 1 \) of the outer-product matrix multiplication algorithm on a \( 4 \times 4 \) torus of processors.]

The outer-product algorithm on a torus of processors is given in Algorithm 5.1. Statement 1 in the algorithm declares the square blocks of the three matrices stored by each processor, assuming that elements of array \( C \) are initialized to zero, and that arrays \( A \) and \( B \) contain the blocks of matrices \( A \) and \( B \) according to the data distribution shown in Figure 5.2. Statement 2 declares two helper buffers that will be used by the processors to store received...
5.3. Matrix Multiplication on a Grid of Processors

The reader will remember that in Section 4.2 we have already given a distributed memory matrix multiplication algorithm with optimal asymptotic efficiency. Furthermore, that algorithm was for a ring of processors, which is a simpler topology. Therefore, one may wonder why there is any advantage to the grid topology. It is true that asymptotically there is no advantage, due to the fact that matrix multiplication has an \( \Omega(n^3) \) computational complexity and an \( O(n^2) \) data size. However, communication terms that become negligible asymptotically do matter for practical values of \( n \), and in fact many

---

**Algorithm 5.1:** The outer-product algorithm on a grid of processors.

```
1. var A, B, C: array[0...m-1, 0...m-1] of real
2. var bufferA, bufferB: array[0...m-1, 0...m-1] of real
3. q := irq (
4.     My_Proc_Col()
5. )
6. myrow := My_Proc_Row()
7. for k := 0 to q-1 do
8.     if (myrow = k) and (mycol = k) then
9.         MatrixMultiplyAdd(C, bufferA, bufferB, m)
10.     else if (myrow = k) then
11.         MatrixMultiplyAdd(C, bufferA, bufferB, m)
12.     else if (mycol = k) then
13.         MatrixMultiplyAdd(C, bufferA, bufferB, m)
```

blocks of matrices \( A \) and \( B \). In statement 3, each processor determines the square root of the number of processors. In statements 4 and 5, each processor obtains its location on the processor torus. The \( q \) steps of the algorithm are implemented by the loop in statement 6. At iteration \( k \), three things happen. First, in statements 7 and 8, the \( k \) processors in processor column \( k \) broadcast their block of \( A \) in their respective processor rows. Note that these statements are correct because we assume that processors that should not participate in a broadcast return immediately from the `BroadcastRow()` call. Then, statements 9 and 10 implement similar broadcasts for blocks of matrix \( B \) along processor columns. Once all these broadcasts have completed, each processor holds all the necessary blocks. Each processor multiplies a block of \( A \) by a block of \( B \) and adds the result to the block of \( C \) for which it is responsible. If the processor is both on processor row \( k \) and on processor column \( k \), then this processor can just multiply the two blocks of \( A \) and \( B \) it holds (statements 12 and 13). Note that for brevity we assume the existence of a function `MatrixMultiplyAdd(C, bufferA, bufferB, m)`, which multiplies two \( n \times n \) matrices stored in arrays \( A \) and \( B \) and adds the result to an \( n \times n \) matrix stored in array \( C \). If the processor is on processor row \( k \) and not on processor column \( k \), then it must multiply the block of \( A \) it has just received with the block of \( B \) it holds (statements 14 and 15). Similarly, a processor on processor column \( k \) but not on processor row \( k \) multiplies the block of \( A \) it holds with the block of \( B \) it has just received (statements 16 and 17). Finally, in statements 18 and 19, a processor that is neither on processor row \( k \) nor on processor column \( k \) multiplies the two blocks of \( A \) and \( B \) it has received. Note that it is possible to adapt this algorithm to the case of non-square matrices and/or non-square grids by allocating rectangular blocks of the matrix to processors, which we leave as an exercise for the reader.

Performance analysis of the algorithm is straightforward. At each step, each processor is involved in two broadcasts of messages containing \( m^3 \) matrix elements to \( q-1 \) processors. Using the pipelined broadcast implementation on a ring described in Section 3.3.4, the time for each broadcast is

\[
T_{broadcast} = \left( \sqrt{q^2 - 2L} + \sqrt{2q} \right)^2,
\]

where \( L \) is the communication startup cost, and \( q \) is the time to communicate a matrix element in steady state. After the first broadcast, each processor performs an \( n \times m \) matrix multiplication, which takes time \( m^3w \), where \( w \) is the computation time for a basic operation (multiplying two matrix elements and adding the result to another matrix element). After the communication phase in step 0, communication at step \( k \) can always occur in parallel with computation at step \( k-1 \). The communication phase in the last step does not occur in parallel with any computation. Since there are \( q \) steps, the overall execution time \( T(n, p) \) is

\[
T(n, q) = 2T_{broadcast} + (q-1) \max \{2T_{broadcast}, m^3w\} + m^3w. \tag{5.1}
\]

The above analysis is for the 4-port model, with the horizontal and the vertical broadcasts happening in sequence at each step. With the 4-port model, both broadcasts can occur concurrently, and the execution time of the algorithm is obtained by removing the factor 2 in front of each \( T_{broadcast} \) in the above equation. When \( n \) becomes large, \( T_{broadcast} \sim n^3L/p \), and thus \( T(n, q) \sim n^3w/p \), which shows that the algorithm achieves an asymptotic efficiency of 1. This algorithm is used by the ScaLAPACK [36] library, albeit often using a block-cyclic data distribution (see Section 5.4).

5.3.2 Grid vs. Ring?

The reader will remember that in Section 4.2 we have already given a distributed memory matrix multiplication algorithm with optimal asymptotic efficiency.
algorithm designers have striven to reduce communication costs for parallel matrix multiplication. We discuss below in what way a grid topology is advantageous compared to a ring topology.

In practice, for large values of $n$, up to a point, the algorithm’s execution time can be dominated by communication time. This happens, for instance, when the ratio $n/b$ is low. The communication time (which is then approximately equal to the execution time) on a grid, using the outer-product algorithm described in the previous section, would be $2c(b/n)^2 p$, assuming a 1-port model. The communication time of the matrix multiplication algorithm on a ring, developed and analyzed in Section 4.2, is $p n^2/n b = n b$. The time spent in communication when using a grid topology is thus a factor $4/p$ smaller than when using a ring topology. This is easily seen when examining the communication patterns of both algorithms. When using a ring topology, at each step the communication time is equal to that needed for sending $n^2/p$ matrix elements between neighbor processors, and there are $p$ such steps. By contrast, for the algorithm on a grid, the communication at each step involves two broadcasts of $n^2/p$ matrix elements, and there are $p$ such steps. With the pipelined implementation of the broadcast, broadcasting $n^2/p$ matrix elements in a processor row or column can be done in approximately the same time as sending $n^2/p$ matrix elements from one processor to another on the ring (provided that $n$ is not too small). Since there are two broadcasts, at each step the algorithm on the grid spends twice as much time communicating as on the ring. But it performs a factor $4/p$ fewer steps! So the algorithm on a grid spends a factor $4/p$ less time communicating than the algorithm on a ring. With a 4-port model, this factor is $2/p$.

The above advantage of the grid topology can be attributed to the presence of more network links and to the fact that many of these links can be used concurrently. In fact, for matrix multiplication, the 2-D data distribution induced by a grid topology is inherently better than the 1-D data distribution induced by a ring topology, regardless of the underlying physical topology! To see this, let us just compute the total amount of data that needs to be communicated in both versions of the algorithm.

The algorithm on a ring communicates $p$ matrix stripes, each containing $n^2/p$ elements at each step, for $p$ steps, amounting to a total of $p n^2$ matrix elements sent on the network. The algorithm on a grid proceeds in $n/p$ steps. At each step $2 \times \sqrt{p}$ blocks of $n^2/p$ elements are sent, each to $\sqrt{p} - 1$ processors, for a total of $2 \sqrt{p} \sqrt{n^2/p} \leq 2 n^2$ elements. Since there are $\sqrt{p}$ steps, the total number of matrix elements sent on the network is lower than $2 \sqrt{p} \times \sqrt{n^2} = \sqrt{2n^2}$, i.e., at least a factor $\sqrt{2} \sqrt{p}$ lower than in the case of the algorithm on a ring! We conclude that when using a 2-D data distribution one inherently sends less data then when using a 1-D distribution, by a factor that increases with the number of processors. Although we do not show it here formally, this result is general and holds for any (reasonable) matrix multiplication algorithm. The implication of this result is that, for the purpose of matrix multiplication, using a grid topology (and the induced 2-D data distribution) is at least as good as using a ring topology, and possibly better. For instance, when implementing a parallel matrix multiplication in a physical topology on which all communications are serialized (e.g., on a bus architecture like a non-switched Ethernet network), one should opt for a logical grid topology with a 2-D data distribution to reduce the amount of transferred data. Recall, however, that for $n$ sufficiently large, the two logical topologies become equivalent, with execution time dominated by computation time.

### 5.3.3 Three Matrix Multiplication Algorithms

In this section, we review three classic algorithms for multiplying square matrices, named after their designers: Cannon, Fox, and Snyder. These algorithms are more complex than the outer-product algorithm, but they are interesting to study and should be part of the culture of all parallel algorithm designers. We assume a $q \times q$ torus of $p = q^2$ processors as shown on the left side of Figure 5.2, with the corresponding 2-D data distribution for matrices $A$, $B$, and $C$ as shown on the right side of Figure 5.2. In this section, we write algorithms using high-level pseudo-code for the sake of simplicity. Indeed, although these algorithms are easy to understand intuitively and visually with examples, the codes for the full algorithms can be rather lengthy. At this point in the book the reader should be able to develop full-detailed implementations of these algorithms, and we leave such implementations as exercises.

#### The Cannon Algorithm

The Cannon algorithm [38] requires an initial re-distribution of matrices $A$ and $B$ as follows. Each block row of matrix $A$ is shifted (by zero, one, or more positions) so that each processor in the first processor column holds a diagonal block of the matrix. Similarly, each block column of matrix $B$ is shifted so that each processor in the first processor row holds a diagonal block of the matrix. After these shifts, block $A_{ij}$ of matrix $A$ is stored on processor $P_{(i-1)p+j}$ and block $B_{ij}$ of matrix $B$ is stored on processor $P_{(j-1)p+i}$. The resulting distributions are shown in Figure 5.4 for a $4 \times 4$ torus. At the end of the algorithm similar shifts are necessary to restore matrices $A$ and $B$ to their initial distributions. These initial and final steps are typically called preskewing and postskewing.

The Cannon algorithm only requires point-to-point communication between neighboring processors. Once the preskewing is done, the algorithm proceeds in $q$ steps. At each step, each processor computes the product of the blocks of $A$ and $B$ it holds and adds the result to its block of $C$ (whose elements were initialized to zero). Then, blocks of matrix $A$ are shifted by one position toward the left (horizontal shifts within processor rows) and blocks of matrix $B$ are shifted by one position toward the top (vertical shifts within processor columns). Blocks of matrix $C$ never move. Communication and
that eventually each processor computes all products, which block multiplications are performed by each processor. The symbol $P_{k,0}$ updates its block of matrix $C$, $C_{k,0}$, by adding to it the result of the $A_{k,1}B_{1,2}$ product, while processor $P_{k,2}$ adds $A_{k,2}B_{2,3}$ to $C_{k,2}$, and processor $P_{k,0}$ adds $A_{k,0}B_{0,1}$ to $C_{k,0}$. Intuitively one can see that eventually each processor $P_{k,j}$ will have computed all $A_{i,j}B_{j+1,i}$ products, $i = 0, \ldots, q - 1$, needed to obtain the final value of $C_{k,j}$.

**Algorithm 5.2: The Cannon algorithm.**

We depict the first two steps of the algorithm in Figure 5.6, which shows how block multiplications are performed by each processor. For instance, in the first step, processor $P_{1,0}$ updates its block of matrix $C$, $C_{1,0}$, by adding to it the result of the $A_{1,1}B_{1,2}$ product, while processor $P_{1,2}$ adds $A_{1,2}B_{2,3}$ to $C_{1,2}$, and processor $P_{1,0}$ adds $A_{1,0}B_{0,1}$ to $C_{1,0}$. In the second step, blocks of $A$ and $B$ have been shifted horizontally and vertically, as seen in the figure. So, during this step, processor $P_{2,0}$ adds $A_{2,0}B_{0,1}$ to $C_{2,0}$, while processor $P_{i,0}$ adds $A_{i,0}B_{0,1}$ to $C_{i,0}$, for $i = 1, 2, \ldots, q$. Intuitively one can see that eventually each processor $P_{k,j}$ will have computed all $A_{i,j}B_{j+1,i}$ products, $i = 0, \ldots, q - 1$, needed to obtain the final value of $C_{k,j}$.

**Algorithm 5.3: The Fox algorithm.**

As for the Cannon algorithm, we illustrate the first two steps of this algorithm in Figure 5.6. The figure shows how matrix block multiplications are performed by each processor at each step. During the first step relevant
blocks of $A$ are blocks $A_{i,i}$, $0 \leq i < q$, that is, blocks of the first block diagonal of matrix $A$. For instance, in the first step, processor $P_{1,2}$ updates its block of matrix $C$, $C_{1,2}$, by adding to it the results of the $A_{2,2} \times B_{2,2}$ product, while processor $P_{2,3}$ adds $A_{3,2} \times B_{3,2}$ to $C_{3,2}$. In the second step, relevant blocks of $A$ are blocks $A_{i,i+1 \mod q}$, that is, blocks on the second block diagonal. During this step, processor $P_{1,2}$ adds $A_{1,3} \times B_{1,3}$ to $C_{1,3}$ and processor $P_{2,3}$ adds $A_{2,3} \times B_{2,3}$ to $C_{2,3}$. Here again it is easy to see that eventually processor $P_{i,j}$ will have computed all block products necessary to obtain the final value of $C_{i,j}$.

The Snyder Algorithm

Our third algorithm, proposed by Snyder in [83], uses both a preskewing and postskewing phase, as well as vertical shifts of matrix $B$ and global sum operations. The preskewing step consists in simply transposing the blocks of matrix $B$ (so that block $B_{i,j}$ is stored on processor $P_{j,i}$). Like the previous ones, this algorithm proceeds in $q$ steps. At each step $k$, $1 \leq k < q$, each processor multiplies the blocks of matrices $A$ and $B$ it holds and performs a vertical shift of $B$ (by one position upward). Then, processor $P_{i,1+k-1 \mod q}$ receives all block products computed by the processors on its processor row, including itself, which are summed together and stored into the block of matrix $C$ this processor holds, i.e., $C_{i,1+k-1 \mod q}$. Note that these blocks of $C$ are those of the $k$-th block diagonal of matrix $C$. The Snyder algorithm, written in high-level pseudocode, is shown in Algorithm 5.4.

![Algorithm 5.4: The Snyder algorithm](image)

**FIGURE 5.6:** The first two steps of the Fox algorithm on a 4 x 4 grid of processors, where at each step each processor multiplies one block of $A$ and one block of $B$ and adds this product to a block of $C$.

**FIGURE 5.7:** The first two steps of the Snyder algorithm on a 4 x 4 grid of processors, where at each step each processor multiplies one block of $A$ and one block of $B$. All such products are added together and added to the blocks of $C$ shown in boldface, within each processor row.

Figure 5.7 shows the first two steps of the algorithm. The blocks of $C$ that are updated by the global sum operations are shown in boldface, along the first diagonal for the first step, and the second diagonal for the second step. In this sense the meaning of the $+$ sign in this figure is different from that in Figures 5.5 and 5.6. Indeed, only $q$ blocks of matrix $C$ are updated at each step, as opposed to $q^2$ blocks. But each block is updated only once during the execution of the algorithm. For instance, in the second step, processor $P_{2,3}$ updates block $C_{2,3}$ by adding to it the three products $A_{2,2} \times B_{2,3}$, $A_{2,1} \times B_{2,3}$, and $A_{2,2} \times B_{2,2}$ received from processors $P_{1,3}$, $P_{1,2}$, and $P_{2,2}$ respectively, and
5.3. Matrix Multiplication on a Grid of Processors

Matrix products can be preskewed in two ways: by preskewing each processor row and the shifts of blocks of matrices, and the sum of two terms, \( B \). The time necessary to multiply an element of matrix \( A \) takes time \( T^{\text{comp}} \), and adds the result to an element of matrix \( B \). The algorithm's execution time is \( T^{\text{exec}} \), which is the communication startup cost, \( T^{\text{comp}} \) is the time for preskewing and postskewing the locally computed product \( A_{1,3} \times B_{3,1} \).

### 5.3.4 Performance Analysis of the Three Algorithms

Table 5.1 summarizes the essential features of the three algorithms. In this section, we analyze their performance on a \( q \times q \) torus, both under the \( 1 \)-port assumption and the less stringent \( 4 \)-port assumption. In both cases we assume full-duplex bidirectional network links. For completeness, we also mention results obtained in the literature assuming that the underlying platform implements wormhole routing. Note that a typical approach is to consider a hypercube topology to develop global communication algorithms. One then "casts" these algorithms to a grid topology, which is straightforward as a grid topology is easily embeddable in a hypercube topology. Essentially, the wormhole routing assumption makes it possible to ignore issues of processor proximity when accounting for communication time. More details on the hypercubic, on the notion of topology embedding, and on wormhole routing are presented in Chapter 3.

We use the following notation, with which the reader should be familiar by now: \( m = n/q \). \( L \) is the communication startup cost. \( q \) is the time necessary to send a matrix element over a network link in steady state, and \( w \) is the time necessary to multiply an element of matrix \( A \) by an element of matrix \( B \) and add the result to an element of matrix \( C \).

#### Cannon Algorithm

Let us start with the 4-port model. The algorithm's execution time is the sum of two terms: \( T^{\text{exec}}_{\text{Cannon}} \), the time for preskewing and postskewing the matrices, and \( T^{\text{comp}}_{\text{Cannon}} \), the time to perform the computation ("4p" stands for 4-port).

For the preskewing and postskewing steps, one can limit the number of shifts of blocks of matrices \( A \) and \( B \) to \( q/2 \). To understand this, consider a processor row and the shifts of blocks of matrix \( A \) that must be performed by processors in that row. It should be clear to the reader that performing \( q - 1 \) left shifts is equivalent to performing one right shift. More generally, performing \( z \) left shifts is equivalent to performing \( q - z \) right shifts. Therefore, in the worst case, a processor row only needs to perform \( q/2 \) shifts, this maximum number of shifts being performed by the middle processor row(s). Therefore, the preskewing of matrix \( A \) takes time \( \lfloor q/2 \rfloor (L + w^2) \). The time to preskew matrix \( B \) is identical. In the 4-port model, horizontal and vertical communications can occur concurrently, and thus the total time for preskewing is simply \( \lfloor q/2 \rfloor (L + m) \). The time for postskewing is identical to the time for preskewing. Therefore, the total time spent for preskewing and postskewing is

\[
T^{\text{4p}}_{\text{Cannon}} = 2 \lfloor q/2 \rfloor (L + m + b)).
\]

At each step, each processor computes an \( m \times m \) matrix multiplication, which is done in time \( m^3 w \), and sends an \( m \times m \) block of \( A \) and an \( m \times m \) block of \( B \) to its neighbors, which both take time \( L + m + b \). These two communications can occur concurrently in the 4-port model because one is horizontal and the other is vertical. Assuming that computation can occur concurrently with communication, and accounting for the fact that there are \( q \) steps to the computation, we obtain the overall execution time as

\[
T^{\text{4p}}_{\text{Cannon}} = q \max (m^3 w, L + m + b). \]

Using the same basic algorithm with a 1-port model, the added constraint is that horizontal and vertical communications cannot happen concurrently. Consequently, preskewing (and postskewing) for matrices \( A \) and \( B \) must happen in sequence and we obtain \( T^{\text{1p}}_{\text{Cannon}} = 2 T^{\text{2p}}_{\text{Cannon}} \) ("1p" stands for 1-port). Similarly, at each step the communications of blocks of \( A \) and blocks of \( B \) cannot happen concurrently, leading to

\[
T^{\text{1p}}_{\text{Cannon}} = q \max (m^3 w, 2L + 2m + b). \]

In the end we obtain the overall execution times for both models:

\[
T^{\text{4p}}_{\text{Cannon}} = 2 \lfloor q/2 \rfloor (L + m + b) + q \max (m^3 w, L + m + b), \]

\[
T^{\text{1p}}_{\text{Cannon}} = 4 \lfloor q/2 \rfloor (L + m + b) + q \max (m^3 w, 2L + 2m + b). \]

#### Fox Algorithm

The Fox algorithm proceeds in \( q \) steps, with no preskewing or postskewing, and thus the execution time is simply the time taken by a single step multiplied by \( q \). The computation time at each step is \( m^3 w \), just like for the Cannon algorithm.

At each step, there are \( q \) concurrent broadcasts of blocks of matrix \( A \), one broadcast in each processor row. Using the pipelined broadcast presented in Section 3.3.4 with the optimal packet size, the time for the broadcast, \( T_{\text{bcast}} \), is

\[
T_{\text{bcast}} = \left( \sqrt{q} - \sqrt{2L} + \sqrt{2m + b} \right)^2. \]

Due to the fact that links are bidirectional, with the 4-port model the broadcast time above can be reduced by having the source processor simultaneously...
send data in both directions on the ring. With this technique, the execution time of the broadcast is obtained by replacing $q$ in the above equation by $[q/2]$. This is because the first packets sent in both directions go through at most $[q/2]$ hops. Note that the asymptotic performance of the broadcast when $q$ gets large is unchanged by this modification.

The shift of the blocks of matrix $B$ can be done in time $L + m^2 b$. In the 4-port model, this shift can occur concurrently with the broadcasts of the blocks of matrix $A$ at each step. As a result, the time to perform the shift is completely hidden ($T_{bcast} \geq L + m^2 b$, for $q > 1$). Computation at each step occurs concurrently with these communications. However, processors must all wait for the first broadcast to complete before proceeding. Then, at each step, the computation for that step, the shift for that step, and the broadcast for the next step can occur concurrently. In the last step, only the computation and the shift occur. We obtain the overall execution time in the 4-port model, $T^{4p}$, as

$$T^{4p} = [\frac{1}{2} \left( \sqrt{(q-2)L + \sqrt{m^4 b^2}} \right)^2 + \left( q - 1 \right) \max \left( \left\lfloor \frac{m}{4} \left( \sqrt{(q-2)L + \sqrt{m^4 b^2}} \right) \right\rfloor + L + m^2 b \right) + \max \left( m^4 b, L + m^2 b \right)$$

With the 1-port model, horizontal and vertical communications cannot occur concurrently and during the broadcast the source can only send data in one direction at a time. Therefore, the execution time is simply

$$T^{1p} = \left( \sqrt{(q-2)L + \sqrt{m^4 b^2}} \right)^2 + \left( q - 1 \right) \max \left( \left\lfloor \frac{m}{4} \left( \sqrt{(q-2)L + \sqrt{m^4 b^2}} \right) \right\rfloor + L + m^2 b \right) + \max \left( m^4 b, L + m^2 b \right)$$

If the underlying platform implements wormhole routing, the execution time on the broadcasts in the 1-port model can be reduced approximately to being only logarithmic in $q$ due to the use of a (binary) broadcast tree, while in the above equation it is linear in $q$. The broadcast starts with one processor sending its matrix block to another processor. Then, both these processors send their blocks to another two processors, and so on, for a total of $\log q$ steps. See Section 3.4.2 for a complete description of the broadcast algorithm with wormhole routing.

Snyder Algorithm

The major differences between the Snyder algorithm and the previous ones are the use of a pre- and post-transposition of matrix $B$ and the use of global sums to compute blocks of matrix $C$ by accumulation. Let us discuss both these operations.

5.3. Matrix Multiplication on a Grid of Processors

There are various ways to implement a matrix transposition on a grid of processors. A simple approach is discussed in Exercise 3.4. We discuss here another simple approach (note that we transpose the blocks, but not the content of the blocks). Consider a block of matrix $B$ held initially by processor $P_{i,j}$, where $i > j$ (i.e., a block in the lower left part of the matrix). One way for this block to reach its destination, processor $P_{j,i}$, is for it to travel to the right, all the way to processor $P_{j,q}$ (which is on the diagonal of the processor grid), and then upward all the way to processor $P_{q,i}$. Blocks in the upper right half of the matrix travel similarly in the other direction. In the 4-port model, all blocks can be communicated concurrently and the blocks that require the largest number of hops are the blocks owned by processors $P_{i+1}$ and $P_{q,i}$. These blocks are communicated between neighboring processors $2(q-1)$ times, giving us the time for the entire transposition as

$$T_{bcast-transp}^{4p} = 2(q - 1)(L + m^2 b)$$

If our logical topology is a torus, this time can be cut roughly in half. Indeed, with a torus, transposing the block initially held by processor $P_{i,j}$ takes only two communication steps, using the loopback links. The factor $2(q-1)$ above then becomes $2(q/2)$. The 4-port model precludes concurrent communications of the blocks originally in the upper half of the matrix and of the blocks originally in the lower half of the matrix. Therefore, the execution time is multiplied by a factor 2.

Another approach could consist in shifting the blocks in each row so that they are in their destination columns. As soon as a block has reached its destination column it can be shifted vertically to reach its destination row provided there is no link contention. While in a 4-port model this algorithm has the same execution time as the algorithm above, in the 1-port model it uses a smaller number of steps. Writing the algorithm to satisfy the "there is no link contentation" constraint requires care, and we leave it as an exercise for the reader. Note that enforcing this constraint is not necessary to have a correct algorithm, but it may be difficult to analyze its performance.

Finally, note that if wormhole routing is implemented on the underlying platform, then a clever recursive transposition algorithm described in [45] leads to a shorter execution time:

$$T_{bcast-transp}^{4p} = \left\lceil \log_2(q) \right\rceil (L + m^2 b)$$

Let us first consider the 4-port model when developing the performance analysis of the rest of the algorithm. The execution consists of a sequence of $q$ steps, where each step consists in a product of matrix blocks, a shift of blocks of matrix $B$ along processor columns, and a global sum of blocks of matrix $C$ along processor rows (see Algorithm 5.4). The first global sum can only be done after the matrix products have been performed. These matrix products take time $m^4 b$, and can be done concurrently with the shift of matrix $B$.
5.3. Matrix Multiplication on a Grid of Processors

which takes time $L + m^2b$. As usual, with appropriate use of buffers and pointers, all three operations can be done in parallel for $q - 1$ iterations of the for loop in the algorithm. By now, it should be clear to the reader that the shift of matrix $B$ is completely hidden by the global sum of matrix $C$ as it entails strictly fewer communication steps. The algorithm completes with a final global sum of matrix $C$ blocks. Using $\Gamma$ to denote the execution time of this global sum, which is to be determined, the execution time of the $q$ steps is

$$\max \left( m^3w, L + m^2b \right) + (q - 1) \max \left( m^3w, \Gamma \right) + \Gamma$$

The global sum of blocks of matrix $C$ can be implemented in various ways. A complex but likely efficient approach would be to have partial sums computed recursively and in parallel by increasingly large groups of processors within a row. Instead, let us design a simple algorithm. At each step, in each processor row, the two processors the farthest away from the destination processor (i.e., the one on the $k$-th diagonal of the processor grid at step $k = 1, \ldots, q$) send a matrix block toward this processor in opposite directions on the ring. Each processor receiving this block adds it to its own block and forwards the result toward the destination processor, and so on. The destination processor will receive two blocks, which will be added to its own block of matrix $C$. If $q$ is odd, then these two blocks will arrive at the same time and thus will be added one after the other. It follows that this algorithm requires $\left( \frac{q}{2} \right)$ communication steps and $\left( \frac{q + 1}{2} \right)$ computation steps, for an overall execution time of

$$\Gamma = \left( \frac{q}{2} \right) \left( L + m^2b \right) + \left( \frac{q + 1}{2} \right) m^3w.$$ 

Note our use of $w'$ in this equation to denote the time needed to add two matrix elements together, which is lower than $w$, the time needed to multiply two matrix elements together and add them to a third matrix element.

Alternatively, one can design a pipelined algorithm on a unidirectional ring in the same fashion as the pipelined broadcast developed in Section 3.3.4. Note that it is possible to implement a faster broadcast using the fact that our rings are bidirectional, as for the Fox algorithm. However, note that here we have both communications and computations (for the summing of blocks of $C$), so this technique is only useful if communication time is larger than computation time. We leave the development of a bidirectional global sum of blocks of $C$ as an exercise for the reader. We split the $m^2$ matrix elements to be sent and added by each processor into $r$ individual chunks. Let us consider a given processor row $i$. Without loss of generality, let us also assume that $P_{i-1}$ is the destination processor and that communication occurs in the direction of increasing processor column indices. In the first step of the algorithm, processor $P_0$ sends a chunk of $m^2b/r$ matrix elements to $P_1$. In the second step, processor $P_2$ adds these matrix elements to the corresponding matrix elements it owns (to perform the global sum) and receives the next chunk from $P_0$. In the third step, processor $P_1$ is engaged in three activities: receiving $m^2b/r$ elements, adding $m^2b/r$ elements, and sending $m^2b/r$ elements, which can all occur concurrently and take time $\max \left( L + m^3b/r, m^3w/r \right)$. With this scheme, processor $P_{i-1}$ finishes computing the first chunk of its block of matrix $C$ after $2(q - 1) - 2q - 2$ steps. Since the first step entails only communication and all the others entail both communication and computation, processor $P_{i-1}$ finishes computing its first chunk by time

$$L + m^3b/r + (2q - 3) \max \left( L + m^3b/r, m^3w/r \right).$$

Afterwards, a new chunk is computed at every step. Since there are $r - 1$ remaining chunks, the overall execution time for the global sum, $\Gamma(r)$, is

$$\Gamma(r) = L + m^3b/r + (2q - 4 + r) \max \left( L + m^3b/r, m^3w/r \right).$$

As in the case of the simple pipelined broadcast, one goal here is to find the optimal value for $r$. Fortunately, we can apply the same technique. For instance, assuming that $b \geq w'$, the maximum in the above equation becomes simply $L + m^3b/r$. The optimal execution time is obtained for $r = r_{opt}$, where

$$r_{opt} = \sqrt{\frac{(2q - 3)m^3b}{L}}.$$ 

Recall that the above can be obtained by directly applying the “goat in a pen" theorem, or by simply computing the derivative of $\Gamma(r)$ with respect to $r$ (see Sections 3.2.2 and 3.3.3). The optimal execution time for the global sum when $r = r_{opt}$, $\Gamma_{opt}$, is then

$$\Gamma_{opt} = (2q - 2)L + m^3b + 2m\sqrt{(2q - 3)L}.$$ 

The case $b < w'$ is more involved, and is left as an exercise for the interested reader. We finally have the overall execution time for the algorithm, $T_B$, in the case $b \geq w'$, as

$$T_B = 2T_{commute} + \max \left( m^3w, L + m^b \right) + (q - 1) \max \left( m^3w, (2q - 2)L + m^b + 2m\sqrt{(2q - 3)L} \right) + (2q - 2)L + m^b + 2m\sqrt{(2q - 3)L}.$$ 

With the 1-port model, the shifts of blocks of matrix $B$ cannot occur concurrently with the global sums of blocks of matrix $C$. We obtain the overall execution time as

$$T_B = 2T_{commute} + \max \left( m^3w, L + m^b \right) + (q - 1) \max \left( m^3w, L + m^b + (2q - 2)L + m^b + 2m\sqrt{(2q - 3)L} \right) + L + m^b + (2q - 2)L + m^b + 2m\sqrt{(2q - 3)L}.$$
5.4 Two-Dimensional Block Cyclic Data Distribution

If the underlying platform implements wormhole routing, the global sum can be implemented following an approach similar to the one for the broadcast in the Fox algorithm, that is, using a binary broadcast tree. The only difference is that some computation is involved at each step to compute the global sum. Essentially, one processor sends its block of matrix \( C \) to another processor, which adds the received block to the block it holds. Then, both these processors send their blocks to another two processors, and so on, for a total of \( \log q \) steps. Remembering that with wormhole routing the data transfer times do not depend on the distance between the two communicating processors (or only in a way that is typically negligible), the execution time of the global sum is then approximately equal to \( \Omega(\log q)(L + n^2w + m^2h) \).

Conclusion

When \( n \) gets large, all these algorithms achieve an asymptotic parallel efficiency of 1. By now it should be clear to the reader that this is not terribly difficult to achieve for matrix multiplication, given that the computation is \( O(n^3) \) and the communication is \( O(n^2) \). The above performance analyses make it possible to compare the three algorithms for particular values of \( n \) and \( q \) and of the characteristics of the platform. More importantly, the main merit of going through these admittedly lengthy performance analyses is to expose the reader to several typical algorithms such as matrix transposition or global sums, to put principles such as pipelining to use, and to better understand the impact of 1-port and 4-port models on algorithm design and performance.

5.4 Two-Dimensional Block Cyclic Data Distribution

On a 2-D grid we have used the natural 2-D data distribution of matrices by which square blocks of a matrix are assigned to processors. This distribution was well suited to, for instance, the multiplication of two matrices. In Chapter 4, we have seen applications that strongly benefit from cyclic 1-D data distributions, namely, stencil applications and the LU factorization. We demonstrated that cyclic distributions can reduce idle time via latency reduction, improved pipelining of communication and computation, and/or improved load balancing. It is therefore natural to combine the advantages of cyclic data distributions and of 2-D data distributions, in so-called 2-D block cyclic data distributions.

Figure 5.8 depicts an example 2-D block cyclic distribution of a square matrix on a square processor grid, in which processors are allocated 2 x 2 blocks of the matrix. While we have seen 1-D distributions that were cyclic across rows or across columns, this 2-D distribution is cyclic across both rows and columns. More formally, for a matrix \( A \) of dimension \( n \times n \) distributed in a 2-D block cyclic fashion on a processor grid with \( p = q^2 \) processors using \( b \times b \) blocks, block \( A_{i,j} \) is stored on processor \( P_{x \mod q, y \mod q} \). Note that for simplicity we assume that \( b \)-divides \( n \). Depending on the values of \( n, q \), and \( b \), some processors may hold more blocks of \( A \) than some others. For the example shown in Figure 5.8, processor \( P_{0,0} \) holds nine blocks of the matrix while processor \( P_{3,3} \) holds only four blocks.

All the algorithms we have seen in this chapter can be implemented using a 2-D block cyclic distribution. Another advantage of the 2-D block cyclic distribution is that block sizes can be chosen arbitrarily, rather than be limited to being \( n/q \times n/q \). This provides more flexibility for performance tuning and more opportunities to utilize each processor’s memory hierarchy to the best of its potential. Both the ScaLAPACK [36] and PLAPACK [116] libraries provide 2-D block cyclic implementations of linear algebra functions. Such implementations are usually not conceptually difficult, but require the implementation of a software infrastructure to support and implement the 2-D block cyclic abstraction. This is especially true when the algorithms cannot rely on simplifying assumptions and must instead handle, say, rectangular matrices, grids, or blocks, and prime matrix, grid, or block dimensions. Note also that the High-Performance Fortran (HPF) language supports 2-D block cyclic data distributions natively [75]. We do not develop algorithms that use a 2-D block cyclic data distribution in this book and refer the reader to the aforementioned libraries for example implementations.

Bibliographical Notes

In addition to the many references cited in this chapter, a valuable reference for examples of algorithms on 2-D processor grids is, once again, the book by Kumar et al. [76]. Of interest is also the book by Cosnard et al. [45]. Finally, both the ScaLAPACK [36] and PLAPACK [116] libraries contain many implementations of interesting algorithms on 2-D processor grids.