3.3 Case Study: The Unidirectional Ring

We consider a computing platform that consists of \( p \) processors arranged in a logical unidirectional ring, as seen in Figure 3.7. The processors are denoted by \( P_k \) for \( k = 0, \ldots, p - 1 \). Each processor can determine its logical index by calling function \( \text{NodeIndex()} \). Note that we use the term “processor” to denote both a physical compute device (or node) and a running program on that device, i.e., a process.

There is no confusion because we always assume one process per device.

To access data that are not in their local memories, processors must communicate via explicit sending and receiving of messages. This mode of operation is common and termed “communication primitives.” A few important points must be noted:

- Calls to communication primitives must match: If processor \( P_i \) executes a \( \text{Send} \), then its predecessor (processor \( P_{i-1 \ mod \ p} \)) must execute a \( \text{Send} \). Otherwise, the program cannot terminate.
- Since each processor has a single outgoing communication link there is no need for specifying the destination processor: it is always the successor of the sending processor. Similarly, when executing a \( \text{Receive} \), the source of the incoming message is always the predecessor processor. For a more complex logical topology, one may have to specify the destination processor in the \( \text{Send} \) and the source processor in the \( \text{Receive} \).
- The address of the first data element in a message, which is passed to \( \text{Send} \), is an address in the local memory of the sender processor. Similarly, the address passed to \( \text{Receive} \) is in the local memory of the receiver processor. Therefore, even if the program uses the same variable for storing both addresses (remember that we are in an SPMD execution), the value of this variable will most likely be different in both processors.
- One can of course use two different variables, as shown below:

\[
\begin{align*}
\text{Send} & (\text{addr1}, m) \\
\text{Receive} & (\text{addr2}, m)
\end{align*}
\]

3.3. Case Study: The Unidirectional Ring

![Figure 3.7: A unidirectional ring with \( p \) processors.](image)

Each processor has its own local memory. All processors execute the same program, which operates on the data in their respective local memories. This mode of operation is common and termed \textit{single program multiple data} (SPMD).

To access data that are not in their local memories, processors must communicate via explicit sending and receiving of messages. This mode of operation is also common and is termed \textit{message passing}. Each processor can send a message to its successor on the ring by calling the following function:

\[
\text{Send}(\text{addr}, m),
\]

where \( \text{addr} \) is the address (in the memory of the sender processor) of the first data element to be sent and \( m \) is the message length expressed as a number of data elements. For simplicity we assume that the data elements of a message must be contiguous in memory, starting at base address \( \text{addr} \).

Note that message passing implementations, for instance, those of the MPI standard [109], typically provide sophisticated capabilities to communicate non-contiguous data with a single function call. To receive a message, a processor must call the following function:

\[
\text{Receive}(\text{addr}, m).
\]

Both functions above and others allowing processors to communicate are often termed “communication primitives.” A few important points must be noted:

- A rather restrictive assumption consists in assuming that each \( \text{Send} \) and \( \text{Receive} \) is blocking, i.e., the processor that calls one of these communication primitives can continue its execution only once the communication has completed. This completely synchronous message passing mode, which is also called “rendez-vous,” is typical of first generation parallel computing platforms.

So far we have not said anything regarding the semantics of the communication primitives. There are three standard assumptions:

- A rather restrictive assumption consists in assuming that each \( \text{Send} \) and \( \text{Receive} \) is blocking, i.e., the processor that calls one of these communication primitives can continue its execution only once the communication has completed. This completely synchronous message passing mode, which is also called “rendez-vous,” is typical of first generation parallel computing platforms.
A classical assumption is to keep the \texttt{RECEIVE} blocking but to assume that the \texttt{SEND} is non-blocking. This allows a processor to initiate a send but to continue execution while the data transfer takes place. Typically, this is implemented via two functions: one to initiate the communication, and the other to check whether the communication has completed. In this chapter, in order to not clutter the pseudo-code of our algorithms, we do not use the second function. Instead, in the description of our algorithms we mention which operations are blocking and which ones are non-blocking.

A more recently proposed assumption is that both communication primitives are non-blocking; a single processor can send data, receive data, and compute simultaneously. Of course the three should occur concurrently only if there is no race condition. Again, this is implemented via two functions for each communication primitive: initialization and completion check. It is then convenient to think of each program running on a processor as three logical threads of control, one for computing, one for sending data, and one for receiving data (even though the implementation may not be multi-threaded or at least may not appear multi-threaded to the programmer).

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.

### 3.3.1 Broadcast

For a given processor index \( k \), we wish to write a program by which processor \( P_k \) sends the same message, of length \( m \), to all other processors; this operation is called a broadcast. This is a fundamental collective communication primitive. For instance, the sender processor can be a "master" that broadcasts general information (e.g., problem size, input data) to all other "worker" processors.

The implications of these different assumptions will be clear in the course of writing our first parallel programs in the upcoming sections of this chapter and in the upcoming chapters. We almost always use the least restrictive third assumption above, both when writing the programs and when analyzing their performance. It is straightforward to adapt both the programs and the performance analyses to the first two more restrictive assumptions if they seem more appropriate for the underlying computing platform. Our goal here is to convey general principles of parallel algorithm design and of parallel algorithm performance analysis, and these principles hold regardless of the underlying assumptions.

As explained earlier, we use one of the simplest performance models. The time to send/receive a message of length \( m \) in \( L + b \) seconds, where \( L \) and \( b \) are two constants that depend on the platform. \( L \) is the startup cost, in seconds, due to the physical network latency and the software overhead involved in a network communication. \( b \) is the inverse of the data transfer rate, and measures the raw speed of the communication in steady-state.
ring topology in this chapter for two reasons. First, a linear topology such as a ring makes for straightforward collective communication algorithms while highlighting key general concepts such as communication pipelining. Second, we will see in Chapter 4 that when designing algorithms for a logical ring topology it is sometimes beneficial to implement collective communications “by hand.”

### 3.3.2 Scatter

We now turn to the scatter operation by which processor \( P_0 \) sends a different message to each processor. To simplify, we assume that all sent messages have the same length, \( m \). A scatter is useful, for instance, to distribute different data to worker processors, such as matrix blocks or parts of an image.

At the beginning of the execution, processor \( P_0 \) holds the message to be sent to processor \( P_k \) at address \( addr[q] \). To keep things uniform, we assume that there is a message at address \( addr[q] \) to be sent by processor \( P_k \) to itself. (Such a convention is used in standard communication libraries as it turns out to be convenient in practice.) At the end of the execution, each processor holds its own message at address \( msg[p] \).

The simple but key idea to implement the scatter operation is to pipeline message sending, starting with the message to be sent to the furthest processor, that is, processor \( P_{p−1} \). While this message is on its way, other messages to be sent to closer processors can be sent as well.

```
1 Scatter(k, msg, addr, m)
2 q = My_Rank()
3 p = Num_Peers()
4 if q = 0 then
5   for i = 1 to p − 1 do
6     Send(addr[k + p − i mod p], m)
7     msg[addr[i]]
8 else
9   Receive(tempR, m)
10  for i = 1 to p − 1 do
11     Send(msg[q] − i mod p, m)
12     tempS − tempR
13  Receive(tempS, m)
14  msg − tempR
```

**ALGORITHM 3.2:** Scatter on a ring of processors.

### 3.3.3 All-to-All

A third important collective communication primitive is the all-to-all exchange. For all \( k \) from 0 to \( p − 1 \), processor \( P_k \) wishes to send a message to all others, which amounts to \( p \) simultaneous broadcasts. Again, we assume that all messages have the same length, \( m \). At the beginning, each processor holds the message it wishes to send at address \( msg[p] \). At the end, each processor will hold an array \( addr[p] \) of messages, where \( addr[i] \) holds the message from processor \( P_i \). The algorithm is surprisingly straightforward and consists of \( p \) steps, with \( p \) messages on the network during each step, as shown in Algorithm 3.3.

```
1 AllToAll(msg_message, addr, m)
2 q = My_Rank()
3 p = Num_Peers()
4 addr[q] − msg_message
5 for i = 1 to p − 1 do
6   Send(addr[q] − i mod p, m)
7   Receive(addr[q] − i mod p, m)
```

**ALGORITHM 3.3:** All-to-all on a ring of processors.

Considerations regarding the semantics of the communication primitives are identical to those for the scatter operation in the previous section. And again, the execution time is \( (p − 1)(L + mb) \) as all communication links are used at each step.

There is a last classical collective communication operation called gossip.
in which each processor sends a different message to each processor. We leave the gossip algorithm as an exercise for the reader.

3.3.4 Pipelined Broadcast

Can a broadcast be executed faster? The implementation we have proposed runs in time \((p-1)(L+m/b)\), for broadcasting a message of length \(m\). The algorithm is simple, but it turns out to be rather inefficient for long messages. To improve execution time, one idea is to split the message into \(r\) pieces of the same length (assuming that \(r\) divides \(m\)). The sender processor sends the \(r\) message pieces in sequence, allowing the pieces to travel on the ring simultaneously. Such pipelining is key for reducing communication time. At the beginning of the execution, the \(r\) message pieces are stored at all processors. At each step, while a processor receives a message piece, it also sends the piece it has previously received, if any, to its successor. The program is shown in Algorithm 3.4.

![Algorithm 3.4: Pipelined broadcast on a ring of processors.](image)

To determine the execution time, we just need to determine when the last processor, \(P_{n-1}\), receives the last message piece. There must be \(p-1\) communication steps for the first message piece to arrive at \(P_{n-1}\), which amounts to time \((p-1)(L+m/b)\). Then, all remaining \(r-1\) message pieces arrive subsequently, in time \((r-1)(L+m/b)\). Therefore, the overall execution time is the sum of the two:

\[
(p + r - 2) \left( L + \frac{m}{r} \right).
\]

We seek the value of \(r\) that minimizes the previous expression. Using the "goat in a pen" theorem, we obtain \(r_{opt} = \sqrt{\frac{m}{p-1}}\) and the optimal execution time is:

\[
\sqrt{(p-2)L + \sqrt{mb}}.
\]  

(3.2)

In this expression, \(p\), \(L\), and \(b\) are all fixed. Therefore, for long messages, the expression tends to \(mb\), which does not depend on \(p\)! This is the same principle as the one used in IP networks, which split messages into many small packets to improve throughput over multi-hop paths thanks to pipelining of communications over multiple communication links.
Chapter 4

Algorithms on a Ring of Processors

When writing a parallel algorithm on a distributed-memory platform that consists of multiple processors (e.g., a cluster of workstations with some interconnect technology), it is convenient to abstract away the physical network topology of the platform and to design algorithms using a logical topology instead. This chapter presents several parallel algorithms that are designed for use with a logical ring topology. This topology is both simple, which makes it an ideal candidate for a first look at distributed-memory parallel algorithms, and popular. Indeed, a ring is a linear interconnection network: Each processor has a single predecessor and a single successor. We will see that linear networks make for a natural decomposition of regular data structures like arrays.

Our first algorithm is a simple matrix-vector multiplication (Section 4.1), followed by its extension to a matrix-matrix multiplication (Section 4.2). Next, we present a "stencil" algorithm such as those used, for instance, in many numerical methods (Section 4.3). We then provide an in-depth study of the famous LU factorization of a dense matrix (Section 4.4). Our last algorithm in this chapter is a stencil algorithm that motivates the use of a bidirectional ring (Section 4.5). We end the chapter with two sections about practical implementation considerations (Sections 4.6 and 4.7).

4.1 Matrix-Vector Multiplication

Let us write our first parallel algorithm on a unidirectional ring of processors: the multiplication \( y = Ax \) of a matrix \( A \) of dimension \( n \times n \) by a vector \( x \) of dimension \( n \) (we number indices from 0 to \( n - 1 \)). A matrix-vector multiplication is simply a sequence of \( n \) scalar products, as seen in the algorithm below:

\[
\begin{align*}
\text{for } i = 0 \text{ to } n - 1 & \text{ do} \\
& \quad y_i = 0 \\
\text{for } j = 0 \text{ to } n - 1 & \text{ do} \\
& \quad y_i = y_i + A_{ij} \times x_j
\end{align*}
\]

Each iteration of the outer loop (the \( i \) loop) computes the scalar product of one row of \( A \) by vector \( x \). Furthermore, all these scalar product computations are independent, in the sense that they can be performed in any order. Consequently, a natural way to parallelize a matrix-vector multiplication is to distribute the computations of these scalar products among the processors. Let us assume that \( n \) is divisible by \( p \), the number of processors, and let us define \( r = n/p \). Each processor will compute \( r \) scalar products to obtain \( r \) components of the result vector \( y \). To do this, each processor must store \( r \) rows of matrix \( A \). It is natural for a processor to store \( r \) contiguous such rows, which is often termed a block of rows, or simply a block row. For instance, one can allocate the first block row of matrix \( A \) to the first processor, the second block row to the second processor, and so on. The components of vector \( y \) (which are to be computed) are then distributed among the processors in a similar manner as the rows of matrix \( A \) (e.g., the first \( r \) components to the first processor, etc.). These kinds of data distributions are typically called "one-dimensional distributions," because arrays are partitioned along a single dimension. A one-dimensional (1-D) data distribution is a natural choice when developing an algorithm on a 1-D logical topology like a ring or processor.

If one assumes that vector \( x \) is fully duplicated across all processors, then the computations of the scalar products are completely independent since no input data are shared. But it is usual, in practice, to assume that vector \( y \) is distributed among the processors in the same manner as matrix \( A \) and vector \( x \). This is for reasons of modularity and consistency. Parallel programs often consist of sequences of parallel operations. Assuming that the input vector is distributed in the same manner as the matrix and the output vector makes it possible to perform a second matrix-vector multiplication \( z = By \), where the input vector is the output vector of the previous computation (assuming that matrix \( B \) is distributed similarly to matrix \( A \)). Furthermore, parallel algorithms are often easier to understand and modify when all data objects are distributed in a consistent manner.

Each processor holds \( r \) rows of matrix \( A \), stored in an array \( A \) of dimension \( r \times n \), and \( r \) elements of vectors \( x \) and \( y \), stored in two arrays \( x \) and \( y \) of dimension \( r \). More precisely, processor \( P_0 \) holds rows \( qr \) to \( (q + 1)r - 1 \) of matrix \( A \), and components \( qr \) to \( (q + 1)r - 1 \) of vectors \( x \) and \( y \). Thus, one needs the following declarations in our parallel program:

\[
\text{var } A_0[A_0..A_{n-1} \times r], x, y : \text{array}[0..r-1, 0..n-1] \times \text{real};
\]

Using these declarations, element \( A_0[0,0] \) on processor \( P_0 \) corresponds to element \( A_{qr} \) of the matrix, but element \( A_0[0,0] \) on processor \( P_1 \) corresponds to element \( A_{(q+1)r} \) of the matrix. Typically the indices of matrix elements, which we denote via subscripts, are called the global indices, while the indices of array elements, which we denote with square brackets, are called the local indices. The mapping between global and local indices is one of the technical difficul-
arrays. However, for our particular program in this section, this mapping is

\[ A_{i,j} \]

and can calculate the partial scalar product that corresponds to a diagonal

\[ x \]

at the beginning (step 0), each processor

\[ P_i \]

can compute

\[ y \]

During this time, each processor

\[ P_j \]

can compute

\[ x \]

Note that faster execution is not the only motivation for the parallelization

\[ P_{(i,j)} \]

of a sequential algorithm on a distributed memory platform. Parallelization

\[ P_{(i,j)} \]

also makes it possible to solve larger problems. For instance, in the case of

\[ P_{(i,j)} \]

matrix-vector multiplication, the matrix is distributed over the local mem-

\[ P_{(i,j)} \]

ory of \( p \) processors rather than being stored in a single local memory as in

\[ P_{(i,j)} \]

the sequential case. Therefore, the parallel algorithm can solve a problem

\[ P_{(i,j)} \]

(roughly) \( p \) times larger than its sequential counterpart.

The principle of the algorithm is depicted in Figure 4.1, which shows the

\[ P_{(i,j)} \]

initial distribution of matrix \( A \) and of vector \( x \), and in Figure 4.2, which

\[ P_{(i,j)} \]

shows the \( p \) steps of the algorithm. At each step, the processors compute a

\[ P_{(i,j)} \]

partial result, i.e., the product of an \( r \times r \) matrix by a vector of size \( r \).

\[ P_{(i,j)} \]

In the beginning (step 0), each processor \( P_i \) holds the \( q \)-th block of vector \( x \),

\[ P_{(i,j)} \]

and can calculate the partial scalar product that corresponds to a diagonal

\[ P_{(i,j)} \]

block of matrix \( A \). We use a block notation by which \( B_{i,j} \) is the \( q \)-th block of size \( r \times r \) of vector \( x \) (resp. \( y \)), and by which \( A_{i,j} \) is the \( r \times r \) block at the

\[ P_{(i,j)} \]

intersection of the \( q \)-th block row and the \( s \)-th block column of matrix

\[ P_{(i,j)} \]

\( A \). Using this notation, each processor \( P_i \) can compute \( y_i = x_i \times A_{i,j} \) during

\[ P_{(i,j)} \]

the first algorithm step. While this computation is taking place, one can do

\[ P_{(i,j)} \]

a circular block shift of vector \( x \) among the processors: processor \( P_i \) sends

\[ P_{(i,j)} \]

\( x_i \) to processor \( P_{i+1} \). Note that we assume that processor indices are taken

\[ P_{(i,j)} \]

modulo \( p \).

This is shown in Figure 4.2 with communicated blocks stored into buffer

\[ P_{(i,j)} \]

tempR. As mentioned earlier in Section 3.3, we assume that sending, re-

\[ P_{(i,j)} \]

ceiving, and computing can all occur concurrently at a processor as long as

\[ P_{(i,j)} \]

they are independent of each other. By the beginning of step 1, processor

\[ P_{(i,j)} \]

\( P_i \) has received \( x_{i-1} \). (Note that, like for processor indices, we implicitly

\[ P_{(i,j)} \]

assume that block indices are taken modulo \( p \).) Processor \( P_i \) can therefore

\[ P_{(i,j)} \]

compute \( y_i = y_i + A_{i,j-1} \times x_{i-1} \), and at the same time it can participate in

\[ P_{(i,j)} \]
than the communication component as the computation component in the equation above is asymptotically larger.

The reader will notice that the circular block shift of vector \( qr \), which we have used to hold the block rows of matrix \( A \), is identical to that of our parallel matrix-vector multiplication. Each processor holds the desired \( i \) components of the result:

\[
C_{i,j} = A_{i,j} Q_{j}\]

At the end of each step, processor \( P_q \) computes the block rows of matrix \( B \) for all \( i \) from \( 0 \) to \( n-1 \). These products are initialized to zero. While this computation is being performed, a circular shift of the block rows of matrix \( B \) among the processors also takes place: Each processor \( P_q \) sends blocks \( B_{q,l} \) to processor \( P_{q,l+1} \) for all \( l \). By the beginning of step \( q \), processor \( P_q \) has thus received blocks \( B_{q,l-1} \) for all \( l \) and can therefore compute all \( C_{i,j} = A_{i,j} Q_{j} + B_{q,j} \) products. Thus, one needs the following declarations in our parallel program:

\[
\text{var } A, B, C: \text{array}[0..r-1, 0..n-1] \text{ of real;}
\]

The principle of the algorithm is identical to that of our parallel matrix-vector multiplication and thus also proceeds in \( p \) steps. At each step, each processor computes \( p \) partial matrix multiplications, that is, \( p \) multiplications of an \( r \times r \) matrix by an \( r \times r \) matrix. Using the block notation again, at step 0 of the algorithm, each processor \( P_q \) holds blocks \( A_{i,j} \), \( E_{j,l} \), and \( C_{i,j} \) of matrices \( A, B \), and \( C \) for \( l = 0, \ldots, p-1 \). (Note that the blocks of matrix \( C \) are all initialized to zero.) Processor \( P_q \) computes the \( A_{i,j} \times B_{j,l} \) products for all \( l \). Each such product is added to block \( C_{i,j} \). While this computation is being performed, a circular shift of the block rows of matrix \( B \) among the processors also takes place: Each processor \( P_q \) sends blocks \( B_{q,l} \) to processor \( P_{q,l+1} \) for all \( l \). By the beginning of step \( q \), processor \( P_q \) has thus received blocks \( B_{q,l-1} \) for all \( l \) and can therefore compute all \( C_{i,j} = A_{i,j} Q_{j} + B_{q,j} \) products. As in the previous section, we assume that block indices are taken modulo \( p \). These products are added to the blocks of matrix \( C \) held by processor \( P_q \). After \( p \) such steps (and a final shift of the block rows of matrix \( B \) so that its distribution among the
Matrix-Matrix Multiplication

The algorithm we consider updates the value of cell $c_i$ as a function of its eight neighbor cells, as shown in Figure 4.3. We will see how this total amount of communication can be reduced.

In this section, we discuss the parallelization of a popular class of applications commonly called "stencil applications." These applications operate on a discrete domain, $D$, that consists of cells. Each cell holds some value(s) and has neighbor cells. The application uses an algorithm that applies pre-defined rules to update the value(s) of a cell using the values of its neighbor cells. The location of a cell's neighbors and the function used to update cell values form a "stencil" that is applied to all cells in the domain. Stencil applications arise in many areas of science and engineering. Examples include image processing algorithms in which pixels are updated using a local neighborhood of pixels; numerical methods used to compute approximate solutions of partial differential equations over a physical domain, in which case cells are contiguous small regions of the domain; and simulations of complex cellular automata, of which Conway's Game of Life is a well-known example. For some of these applications the domain is large enough that one may have to resort to a distributed-memory implementation, rather than using a simple shared-memory implementation. In this section, we investigate how a distributed-memory parallel stencil application can be implemented using a logical ring topology.

4.3 A Simple Sequential Stencil Algorithm

We consider a stencil application that operates on a two-dimensional (2-D) domain of size $n \times n$ for $n \geq 8$. Each cell has eight neighbor cells, as shown in Figure 4.3.

The algorithm we consider updates the value of cell $c$ as a function of its current value and of the already updated values of its West and North neighbors.

4.2 Matrix-Matrix Multiplication

| ALGORITHM 4.2: Matrix-matrix multiplication algorithm on a ring of processors.
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 var $A, B, C$: array[$0..r-1, 0..n-1$] of real</td>
</tr>
<tr>
<td>2 $q$ := Proc-&gt;$\text{Recv}$()</td>
</tr>
<tr>
<td>3 $p$ := Proc-&gt;$\text{Proc}[]$</td>
</tr>
<tr>
<td>4 $\text{tempS} := B$</td>
</tr>
<tr>
<td>5 for $\text{step} := 0$ to $p - 1$ do</td>
</tr>
<tr>
<td>6 $\text{Send}(\text{tempS}, r \times n)$</td>
</tr>
<tr>
<td>7 $\text{tempR} := C$</td>
</tr>
<tr>
<td>8 for $i := 0$ to $p - 1$ do</td>
</tr>
<tr>
<td>9 for $j := 0$ to $r - 1$ do</td>
</tr>
<tr>
<td>10 for $k := 0$ to $r - 1$ do</td>
</tr>
<tr>
<td>11 $C_{ij, br + j} := C_{ij, br + j} + A_{ij, r \times {q - \text{step} \mod p}} \times \text{tempS}[k, br + j]$</td>
</tr>
<tr>
<td>12 $\text{tempS} := \text{tempR}$</td>
</tr>
</tbody>
</table>

112 | Chapter 4. Algorithms on a Ring of Processors |

This notion of sending data in bulk is a well-known and general principle thereby reducing the overhead due to network startup cost $L$. This change does not reduce the asymptotic efficiency, but it can be significant in practice. This notion of sending data in bulk is a well-known and general principle used to reduce communication overhead in parallel algorithms, and we will see it again and again in this chapter. Furthermore, with modern processors, operating on blocks of data is often better for performance as it exploits a processor's memory hierarchy (registers, multiple levels of cache, main memory) to the best of its capabilities. As a result, operating on blocks of data improves both computation and communication performance. Note that in this algorithm the total number of data elements sent over the network is $(p \times n + r = p \times n^2$ (recall that $r = n/p$). We will see how this total amount of communication can be reduced.
4.3. A First Look at Stencil Applications

The stencil is depicted in Figure 4.4 and can be formalized as follows:

\[ c_{\text{new}} = \text{UPDATE}(c_{\text{old}}, W_{\text{new}}, N_{\text{new}}) \]

FIGURE 4.3: The eight neighbor cells of cell \( c \).

FIGURE 4.4: A simple stencil in which cell \( c \) is updated using the values of its West and North neighbors.

This stencil, albeit simple, is at the heart of important applications such as the Gauss-Seidel numerical method [97] and the Smith-Waterman biological string comparison algorithms [118]. Note that the stencil, as it is described above, cannot be applied to the cells in the topmost row or the leftmost column of the domain. Indeed, the former have no North neighbors and the latter have no West neighbors. This is handled by the UPDATE function, for instance, by using constant values for neighbors outside the domain or by using modified calculations when a cell has only one or even no neighbors as is the case for the top left cell. To indicate that no neighbor exists for a cell update, we pass a Nil argument to UPDATE. For instance, the call UPDATE(\( c \), \( W_{\text{new}} \), Nil) is used to update a cell on the upper edge of the domain, and the call UPDATE(\( c \), Nil, Nil) is used to update the top left cell.

4.3.2 Parallelizations of the Stencil Algorithm

Consider a ring of \( p \) processors \( P_0, \ldots, P_{p-1} \). We must distribute the cells in the domain among the processors. A natural solution is to allocate rows of cells to processors. The main challenge here is to come up with a distribution that balances the computational load among the processors without hindering performance due to overly expensive communications.

Greedy Algorithm

A first idea to parallelize our stencil algorithm is to use a greedy approach by which processors send the cells they compute to their neighbors as early as possible. Such a parallelization reduces the startup latencies (i.e., processors start computing as early as possible) and leads to a good load balance.

Let us assume for now that the number of processors \( p \) is equal to the domain’s dimension \( n \). In this case, we allocate row \( i \) of domain \( A \) to processor \( P_i \) and thus each processor has the following declaration:

```plaintext
var A: array[0..n-1] of real;
```

As soon as processor \( P_i \) has computed a cell value, it sends that value to processor \( P_{i+1} \) (0 \( \leq i < p-1 \)). As in Sections 4.1 and 4.2, we use subscripts for global indices and we denote by \( A_{i,j} \) the value of the cell on row \( i \) and column \( j \) of the domain (0 \( \leq i, j \leq n-1 \)).

Given the shape of our stencil and the fact that the already updated values of the West and North neighbor cells are needed, initially only \( A_{0,0} \) can be computed. Once \( A_{0,0} \) has been computed, then both \( A_{1,0} \) and \( A_{0,1} \) can be computed. It can be seen easily that the computation proceeds in steps: At step \( k \) all values of cells on the \( k \)-th anti-diagonal are computed. The corresponding program is shown in Algorithm 4.3. Figure 4.5 shows each cell labeled by the step at which its value can be computed.

At time \( i+j \), processor \( P_i \) performs the following operations:
- it receives \( A_{i-1,j} \) from \( P_{i-1} \);
- it computes \( A_{i,j} \);
- then it sends \( A_{i,j} \) to \( P_{i+1} \).

Note that the above is not technically true because \( P_0 \) does not need to receive cell values to update its cells and \( P_{p-1} \) does not need to send cell values after
A processor has the following declaration:

```
m: Row
```

If a processor is assigned to it, then it takes at least \( m \) steps before the value of the first cell of the last row assigned to \( \text{P}_i \) is computed. This cell value is necessary for procedure \( \text{P}_i \) to start computing the first cell of its block of rows. Therefore, \( \text{P}_1 \) (and all other processors) stay idle for at least the first \( n/p \) steps of the algorithm. In a view to allowing processors to start computing as early as possible, one can instead assign rows to processors in an interleaved, or cyclic, manner. Row \( j \) is assigned to processor \( \text{P}_{j \mod p} \). Such a cyclic distribution is a classic technique to achieve a good load balance across processors. Each processor has the following declaration:

```
var A: array[0..n-1] of real;
```

ALGORITHM 4.3: Greedy algorithm for the stencil application on a ring of processors.

```
var A: array[0..n-1] of real
q = MemSize()
p = NumProcs()
if q = 0 then
  A[0] := Update(A[0], Nil, Nil)
else
  while Recv(v, 1) do
    A[0] := Update(A[0], v)
  if q = 0 then
  else if q = p-1 then
  else
    Send(A[j-1, 1] || Recv(v, 1))
```

ALGORITHM 4.4: Algorithm for the stencil application on a ring of processors using a cyclic-data distribution.

```
var A: array[0..n/p-1,0..n-1] of real
q = MemSize()
p = NumProcs()
for i = 0 to n/p-1 do
  if q = 0 and i = 0 then
    A[i, 0] := Update(A[i, 0], Nil, Nil)
    Send(A[i, 0], 1)
  else
    Recv(v, 1)
    A[i, 0] := Update(A[i, 0], v)
  if q = 0 and i = n/p-1 then
    Send(v, 1)
    A[i, 0] := Update(A[i, 0], v)
  else if q = p-1 and i = n/p-1 then
    Recv(v, 1)
    A[i, 0] := Update(A[i, 0], v)
  else
    Send(A[i, 0], 1) || Recv(v, 1)
    A[i, 0] := Update(A[i, 0], v)
```

This is a contiguous array of rows that are non-contiguous in the domain. We can now modify our previous algorithm to implement a cyclic distribution of rows among processors, as shown in Algorithm 4.4.

Let us compute the execution time of this algorithm, \( T(n,p) \). We assume that receiving a message is a blocking operation, while sending a message is not. Therefore, the sending of a message at step \( k \) of the algorithm occurs in parallel with the reception of a message at step \( k+1 \). Consequently, the time needed to perform one algorithm step is \( w + k + L \), where \( w \) is the time needed to update a cell value, \( k \) is the rate at which cell values are communicated on a network link, and \( L \) is the communication startup cost. We now need to compute the number of such steps. The computation terminates when processor \( \text{P}_{n/p} \) finishes computing the rightmost cell value of its last row of cells. It takes \( p-1 \) algorithm steps before processor \( \text{P}_{n/p} \) can start doing its first computation. At this point, processor \( \text{P}_{n/p} \) computes one cell value at
Chapter 4. Algorithms on a Ring of Processors

3. A First Look at Stencil Applications

Each step. There are \( n^2 \) cells in the domain, and each processor holds \( n^2/p \) cells. Therefore, processor \( P_{i-1} \) is engaged in cell value computations for \( n^2/p \) steps, and the total number of steps is \( p - 1 + n^2/p \), which gives

\[
T(n, p) = \left( p - 1 + \frac{n^2}{p} \right) (w + b + \ell).
\]

This algorithm was designed so that the time between a cell value computation and its reception by the next processor is as short as possible. But a glaring problem is that the algorithm performs many communications of data items that may be small. In practice, the communication startup cost \( \ell \) can be orders of magnitude larger than \( b \) if the size of a cell value is small. In the case of stencil applications, the cell value is often as small as a single integer or a single floating-point number. Furthermore, \( w \) may not be large. It may be comparable to or smaller than \( L \). Indeed, cell value computations can be very simple and involve only a few operations. This is the case for many numerical methods that update cell values based on simple arithmetic formulas. Therefore, for many applications, a large fraction of the execution efficiency tends to be due to the communication overhead reduces parallel efficiency significantly. This can be seen more plainly by simply computing the parallel efficiency: The parallel efficiency of the algorithm is equal to the sequential execution time, \( n^2w/p \), divided by \( p \times T(n, p) \). When \( n \) gets large, the parallel efficiency tends to \( w/(w + b + \ell) \), which may be well below 1. In the next section we present two techniques for addressing this problem.

Augmenting the Granularity of the Algorithm

A simple technique to decrease communication costs, or, more precisely, communication overhead due to startup latencies, is to send fewer but larger messages. Using the same allocation of rows to processors as before, a processor now computes \( k \) contiguous cell values in each row at each step, where \( 1 \leq k \leq n \), as opposed to just one. As before, the algorithm proceeds in several steps. For simplicity we assume that \( k \) divides \( n \) so that each row contains \( n/k \) segments of \( k \) contiguous cells. If \( k \) does not divide \( n \), then the \( \left( \lfloor n/k \rfloor \right) \)-th segment of the first row stored by each processor spills over to the second row stored by that processor, and so on. The last segment of the last row stored by that processor may contain fewer than \( k \) cells, which does not modify the spirit of the algorithm. We make this assumption to simplify the performance analysis of the algorithm. The data distribution is depicted in Figure 4.6 for four processors, where each segment of \( k \) contiguous cells is labeled by the step at which it is computed.

With this algorithm the number of cell values communicated among processors is the same as with the greedy algorithm, but cell values are communicated in bulk, \( k \) at a time. Larger values of \( k \) lead to lower communication overhead due to network latencies. However, larger values of \( k \) also imply that the time between a cell value computation and its reception by the next processor is larger. This goes against the principle guiding our design of the algorithm in the previous section. In this algorithm processors will start computing cell values later, leading to more idle time.

\[
\begin{align*}
P_0 & \text{ First row of each processor} \\
P_1 & \text{ Second row}
\end{align*}
\]

FIGURE 4.6: Steps of the modified algorithm with \( k > 1 \) and \( p = 4 \) processors.

Another technique to reduce communication costs is to reduce the number of cell values that need to be communicated between the processors. This can be done by allocating blocks of \( r \) steps of \( k \) rows to processors \((r p \leq n)\), still in a cyclic fashion. At each step each processor now computes \( r \times k \) cell values. We assume that \( r \times p \) divide \( n \) for simpler performance analysis. For instance, with \( r = 3, n = 36, \) and \( p = 4 \), we have the following allocation of rows to processors:

\[
\begin{align*}
P_0 & 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \\
P_1 & 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23 \\
P_2 & 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35
\end{align*}
\]

More formally, processor \( P_i \) holds rows \( j \) such that \( i = \left\lfloor \frac{j}{r} \right\rfloor \mod p, 0 \leq j \leq n - 1 \). This notion of a block-cyclic allocation is classic and we will use it often.

The steps of the algorithm are similar to those shown in Figure 4.6, simply replacing individual rows by blocks of rows. We can easily obtain a sufficient condition for processors not to be idle. A processor, say \( P_0 \), computes all cell values in its first block of rows in \( n/k \) steps of the algorithm. Processor \( P_0 \) sends \( k \) cell values to processor \( P_1 \) after \( p \) algorithm steps. These values are necessary for \( P_1 \) to start computing its second block of rows. Therefore, if \( n \geq kp \), no processor stays idle, which is desirable, and which we will assume hereafter. Note that if \( n > kp \), then processors have to temporarily
store received cell values while finishing computing their block of rows for the previous step.

With this new distribution of rows to processors, the amount of data communicated between processors is \( r \) times smaller than with the previous algorithm. Indeed, the processors need to exchange data only at the boundaries between blocks of rows. So the larger \( r \), the lower the communication cost, but the larger \( r \), the larger the time between the computation of a cell value and its communication to the next processor. This is once again the same trade-off: lower communication costs or lower latency between a cell computation and its reception by the next processor.

One interesting question is, are there optimal values of \( k \) and \( r^* \)? We can answer this question via performance analysis of the algorithm. We assume that \( n \geq hp \) so that no processor stays idle and leave the (less relevant in practice) \( n < hp \) case as an exercise for the motivated reader. The algorithm proceeds in a sequence of steps. At each step at least one processor is involved in the following activities:

- it receives \( k \) cell values from its predecessor;
- it computes \( kr \) cell values;
- it sends \( k \) cell values to its successor.

The analysis is very similar to that for our simple greedy algorithm. Here again we assume that receiving a message is a blocking operation while sending a message is not. Consequently, the time needed to perform one step of the algorithm is \( T(n, p, r, k) \), whose parallel efficiency was \( \omega/(u + b + L) \). Dividing the sequential execution time \( n\omega \) by \( p \times T(n, p, r, k) \) we obtain an asymptotic efficiency of \( \omega/(u + b + L)/rk \). In essence, increasing \( r \) and \( k \) makes it possible to achieve significantly higher asymptotic efficiency by reducing communication costs.

While low asymptotic parallel efficiency is important, one may wonder what values of \( k \) and \( r \) should be used in practice. It turns out that, for \( n \) and \( p \) fixed, one can easily compute the optimal value of \( k \). Equating the derivative of \( T(n, p, r, k) \) to zero and solving for \( k \) one obtains the value \( k^*(r) \):

\[
k^*(r) = \frac{L}{p(p - 1)[(u + b) + b]}
\]

Since \( k \leq n/p \), we obtain \( k_{opt}(r) = \min(k^*(r), n/p) \). Finally, for given values of \( n, u, w \) and \( b \), one can compute \( k_{opt}(r) \) and inject it in the expression for \( T(n, p, r, k) \). One can then determine the best value for \( r \) numerically.

### 4.4 LU Factorization

In this section, we develop a parallel algorithm that performs the classic LU decomposition of a square matrix \( A \) using the Gaussian elimination method, which in turn allows for the straightforward solution of linear systems of the form \( Ax = b \). Readers familiar with the algorithm will notice that we make two simplifying assumptions as not to make matters overly complicated. First, we use the Gaussian elimination method without any pivoting. This assumption is rather inconsequential at least in the case of partial pivoting. Since the columns of \( A \) will be distributed among the processors, partial pivoting would not add any extra communication. Second, our algorithm eliminates columns of \( A \) one after another. This is not realistic for modern computers as their memory hierarchies would not be exploited to the best of their capability. Instead, the algorithm should really process several columns at a time (i.e., column blocks), exactly as in our matrix-matrix multiplication algorithm (Section 4.2). But from a purely algorithmic standpoint there is no conceptual difference between computing a single column or a column block, and the algorithm’s spirit is unchanged. Note, however, that when writing the corresponding programs in practice the utilization of column blocks requires many more lines of code. This is true even for sequential algorithms. For instance, see the sequential version of the Gaussian elimination algorithm in the publicly available LAPACK library [8].

#### 4.4.1 First Version

The sequential algorithm for a matrix \( A \) of dimension \( n \times n \) (with global indices from \( 0 \) to \( n - 1 \)) proceeds in \( n \) steps, as shown in Algorithm 4.5. Note that we show two inlined functions, \texttt{ Prep } and \texttt{ Update }. This is to identify and to refer to the two main computations performed by the algorithm conveniently, namely, the \texttt{ Prep } aration of the current column and the update of the bottom right sub-matrix of matrix \( A \). We depict the execution of the algorithm in Figure 4.7.
we store updated column elements. The use of such an array is often necessary.

Algorithm 4.6. LU factorization algorithm on a ring of processors.

\begin{verbatim}
for k = 0 to n - 2 do
    Prep(k):
        for i = k + 1 to n - 1 do
            A_{i,k} = A_{i,k} / A_{k,k}
        for j = k + 1 to n - 1 do
            A_{i,j} = A_{i,j} + A_{k,k} * A_{j,k}

    Update(k, j):
        for i = k + 1 to n - 1 do
            A_{i,j} = A_{i,j} + A_{i,k} / A_{k,k} * A_{k,j}

    Broadcast(Aloc(k), buffer, n - k - 1)

if Aloc(k) = q then
    Prep(k):
        for i = k + 1 to n - 1 do
            buffer[i-k-1] = A_{i,k} - A_{i,q} / A_{q,q}

    Broadcast(Aloc(k), buffer, n - k - 1)

for i = k + 1 to n - 1 do
    Update(k, j):
        for j = k + 1 to n - 1 do
            A_{i,j} = A_{i,j} + buffer[i-k-1] * A_{j,k}

\end{verbatim}

A point of detail for the interested reader: The \( A = LU \) decomposition is obtained after the above algorithm completes by defining \( L_{i,k} = -A_{i,k} \) for \( i > k \) (\( L \) is lower triangular with a unit diagonal) and \( U_{i,j} = A_{i,j} \) for \( j \leq j \) (\( U \) is upper triangular). This sign swapping for the \( L_{i,k} \) values is due to the fact that these values are in fact the coefficients used to eliminate columns when \( L \) and \( U \) is upper triangular \([63]\). Also, as stated above, adding pivoting in functions \texttt{Prep} and \texttt{Update} would be straightforward.

The question we wish to answer is, how can we parallelize this algorithm on a ring of \( p \) processors? Since the algorithm processes the matrix column-by-column, it is natural to distribute these columns among the processors. Rather than fixing the data distribution scheme at this point, let us just assume that we have defined a function \texttt{Aloc}, which, given a column index \( k \) between 0 and \( n-1 \), returns the index of the processor that stores column \( k \) in its memory. At step \( k \) of the algorithm, processor \texttt{Aloc(k)} broadcasts column \( k \) to all processors, allowing them to update their own columns of matrix \( A \). Using the \texttt{Broadcast} function (see Section 3.3), we obtain the parallel algorithm shown in Algorithm 4.6.

Note that in the algorithm above we use a helper array called \texttt{buffer} in which we store updated column elements. The use of such an array is often necessary due to the memory layout of 2-D arrays, in our case array \texttt{ROCS} for row-major storage of 2-D arrays (as in C) rather than column-major storage (as in FORTRAN). This assumption implies that elements of a row are contiguous in memory, while elements of a column are not. The reader will recall that in all the algorithms we have seen so far processors were communicating rows of 2-D arrays, rather than columns. Consequently, the processors were always sending array elements that were contiguous in memory, which was possible with single calls to \texttt{Send}. The LU decomposition algorithm, however, needs to broadcast columns of the array, that is, elements that are not contiguous in memory! Therefore, we place these elements into a helper array so that a single call to \texttt{Broadcast} can send matrix elements in bulk. The alternative would have been to place \( n \) individual calls to \texttt{Broadcast}, one for each column in the current column of matrix \( A \). This alternative leads to high overhead due to network latencies (typically much higher than the overhead of an extra memory copy). Understanding memory layout is always a good idea (for instance, to improve cache reuse) but is paramount when implementing distributed memory algorithms to ensure that data are communicated in bulk as much as possible.

One remaining issue is to map global indices to local indices for accessing elements of matrix \( A \). Defining \( r = n/p \), each processor stores \( r \) column of
Although one could think of doing the same thing with matrix columns here, this was not a di-
or, in other terms, which matrix columns are assigned to which processors?

indices to replace the matrix element specifications in the previous algorithm, unchanged at other processors. At step \( k = 1 \), processor \( \text{Alloc}(1) \) increments its own value of \( l \) after calling \( \text{Per}(1) \). Using array declarations and array indices to replace the matrix element specifications in the previous algorithm, we now obtain the full program shown in Algorithm 4.7.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{var} \( A \) : array\([0..n-1, 0..r-1]\) of real
\State \textbf{var} buffer: array\([0..n-1]\) of real
\State \textbf{var} \( q \) - \( \text{NewProc}(1) \)
\State \textbf{var} \( r \) - \( \text{NewProc}(1) \)
\State \textbf{var} \( j \) = 0
\For{\( l = 0 \) to \( n-2 \)}
\If{\( \text{Alloc}(l) = q \)}
\For{\( i = k + 1 \) to \( n-1 \)}
\State \textbf{for \( i = k + 1 \) to \( n-1 \)}
\State \textbf{for \( i = k + 1 \) to \( n-1 \)}
\EndFor
\EndIf
\EndFor
\EndFor
\EndFor
\EndFor
\EndFor
\State \textbf{end}
\end{algorithmic}
\end{algorithm}

4.4. LU Factorization

The only remaining thing that needs to be specified is the function \( \text{Alloc} \), or, in other terms, which matrix columns are assigned to which processors?

This was not a difficult issue for the matrix-vector or the matrix-matrix multi-

multiplication: Just allocate even blocks of consecutive matrix rows to processors.

Although one could think of doing the same thing with matrix columns here,

this simple solution is not adequate for two reasons:

- The amount of data to process varies throughout the algorithm's exe-
ucion. At the core of the algorithm is the update of column \( k+1 \) to \( n-1 \) at each step \( k \). Therefore, there are fewer and fewer columns to process as the algorithm makes progress, that is, when \( k \) increases.

- The amount of computation is not proportional to the amount of data!
Indeed, column \( k \) is updated \( k \) times. Therefore, from the perspective of a processor, holding column \( n-1 \) implies significantly more computation than holding, say, column \([n/2] \).

Given the above, we need to find an allocation that balances both the memory consumption (all processors must hold the same number of columns so that matrices as large as possible can be processed) and the computation (processors must perform similar numbers of operations to ensure that no processing power is wasted due to processors being idle). Furthermore, com-
putation must be balanced at every step of the algorithm since the amount and distribution of the computation vary throughout algorithm execution.

Perhaps expectedly, a cyclic allocation, which assigns column \( j \) to processor \( P_{jn/k} \), gets the job done. Unlike for the stencil application, in which a cyclic allocation leads to a perfect balance of computation across processors, here processor \( P_{jn/k} \) will have slightly more computations to perform than processor \( P_{j(n+1)} \), but it can be easily shown that the difference is asymptotically negligible. Indeed, consider a column \( j \) assigned to processor \( P_{jn/k} \). This column is updated at algorithm steps \( k = 0, \ldots, j-1 \). More precisely, at step \( k \), the elements of column \( j \) that are updated are the ones from position \( k+1 \) to position \( n-1 \). Therefore, at step \( k \), the number of basic update operations performed by the processor holding column \( j \) is equal to \( n-k \).

Consequently, the total number of elemental updates performed by the processor holding column \( j \) on this column is

\[ \text{Ops}(j) = \sum_{i=0}^{k-1} (n-k-i) = \frac{1}{2} k^2 + \left( n - \frac{k}{2} \right)j. \]

With a cyclic distribution, processor \( P_j \) holds columns \((p+i)\) for \( i = 0, \ldots, n/p - 1 \). Therefore, processor \( P_j \) performs a total of \( \text{Ops}(i) \) update operations, where

\[ \text{Ops}(i) = \sum_{j=p+i}^{n-1} \text{Ops}(p+i). \]

Replacing \( \text{Ops}(p+i) \) by its expression, separating terms, and using well-
known formulas for the sums of consecutive integers and sums of the squares of consecutive integers, one obtains that \( \text{Ops}(i) = \frac{i}{2} (n-1) + O(n^2) \). Therefore, asymptotically, \( \text{Ops}(i) \) does not depend on \( i \) and all processors perform the same amount of update operations. It is easy to show that asymptotically
sections we describe techniques to reduce communication costs. However, note that there are $O(n)$ times more update operations than column preparation operations, making the time spent doing column preparations asymptotically negligible. To modify our program to use a cyclic allocation of matrix columns to processors, one just has to replace the test $\text{Allc}(k) = q$ by $k = q \mod p$.

Let $p(k, P_k)$ and $w(k, j, P_k)$ denote the executions of functions $\text{Prep}(k)$ and $\text{Update}(k, j)$ on processors $P_k$, respectively. The execution time of the algorithm is that of the longest path of operations (also called the critical path), that is

$$p(0, P_0) \rightarrow \left( \sum_{j=1}^{n-1} w(0, 1, P_j), p(1, P_1) \right) \rightarrow \left( \sum_{j=2}^{n-1} w(1, j, P_j), p(2, P_2) \right) \rightarrow \ldots,$$

where each $\rightarrow$ symbol denotes a communication between neighbor processors. After expressing the individual function execution times as functions of the platform parameters, as we have done above for the number of column update operations, it is straightforward to obtain that the overall execution time is the sum of:

- a term $nL + \frac{3}{2}b + O(1)$ that accounts for the $n - 1$ communications;
- a term $\frac{n}{2}w + O(1)$ for the column preparations, and
- a term $\frac{3}{2}w + O(n^2)$ for the column updates.

Note that to be precise we use $w'$ to denote the time for a basic column preparation operation (one division and one negation), and $w$ to denote the time for a basic column update operation (one multiplication and one addition). As $n$ grows, the overall execution time becomes asymptotically equivalent to $\frac{3}{2}w$.

Note that the total number of update operations to be performed is given by

$$\sum_{i=1}^{n-1} \text{ops}(i),$$

which can be computed by replacing $\text{ops}(i)$ by its expression. Simple calculations show that the above sum is asymptotically equal to $\frac{3}{2}w$ when $n$ becomes large. The execution time of our algorithm is asymptotically $p$ times shorter than this amount, and consequently our algorithm has an asymptotic parallel efficiency of 1. In spite of this good asymptotic result, the communication term above can still be significant in practical situations. In the next two sections we describe techniques to reduce communication costs.

4.4. LU Factorization

all processors also perform the same number of column preparations. However, note that there are $O(n)$ times more update operations than column preparation operations, making the time spent doing column preparations asymptotically negligible. To modify our program to use a cyclic allocation of matrix columns to processors, one just has to replace the test $\text{Allc}(k) = q$ by $k = q \mod p$.

Let $p(k, P_k)$ and $w(k, j, P_k)$ denote the executions of functions $\text{Prep}(k)$ and $\text{Update}(k, j)$ on processors $P_k$, respectively. The execution time of the algorithm is that of the longest path of operations (also called the critical path), that is

$$p(0, P_0) \rightarrow \left( \sum_{j=1}^{n-1} w(0, 1, P_j), p(1, P_1) \right) \rightarrow \left( \sum_{j=2}^{n-1} w(1, j, P_j), p(2, P_2) \right) \rightarrow \ldots,$$

where each $\rightarrow$ symbol denotes a communication between neighbor processors. After expressing the individual function execution times as functions of the platform parameters, as we have done above for the number of column update operations, it is straightforward to obtain that the overall execution time is the sum of:

- a term $nL + \frac{3}{2}b + O(1)$ that accounts for the $n - 1$ communications;
- a term $\frac{n}{2}w + O(1)$ for the column preparations, and
- a term $\frac{3}{2}w + O(n^2)$ for the column updates.

Note that to be precise we use $w'$ to denote the time for a basic column preparation operation (one division and one negation), and $w$ to denote the time for a basic column update operation (one multiplication and one addition). As $n$ grows, the overall execution time becomes asymptotically equivalent to $\frac{3}{2}w$.

Note that the total number of update operations to be performed is given by

$$\sum_{i=1}^{n-1} \text{ops}(i),$$

which can be computed by replacing $\text{ops}(i)$ by its expression. Simple calculations show that the above sum is asymptotically equal to $\frac{3}{2}w$ when $n$ becomes large. The execution time of our algorithm is asymptotically $p$ times shorter than this amount, and consequently our algorithm has an asymptotic parallel efficiency of 1. In spite of this good asymptotic result, the communication term above can still be significant in practical situations. In the next two sections we describe techniques to reduce communication costs.

4.4. LU Factorization

4.4.2 Pipelining on the Ring

The algorithm we developed in the previous section does not rely on the fact that the processors are arranged in a logical ring. The algorithm just uses a broadcast primitive that could be implemented on any logical topology. This makes our algorithm portable, which is good, but, on the other hand, it does not exploit the ring topology to the best of its potential. In particular, the $n - 1$ broadcasts are not overlapped with any computations. On a ring topology and with a cyclic allocation of the matrix columns to the processors, realizing a good overlap is, however, natural. In fact, one can almost directly insert the source code of the broadcast algorithm into the LU algorithm!

We modify the algorithm so that as soon as a processor receives column $k$ (at step $k$), it sends it to its successor. The processor performs this communication before starting to update the columns it holds in memory. The algorithm is shown in Algorithm 4.8. An application execution is depicted in Figure 4.8, where each column corresponds to one of four processors and time progresses from top to bottom. For the sake of illustration, we depict column preparation, sending, receiving, and column updating as taking the same
FIGURE 4.8: Example execution of the pipelined LU algorithm on a ring.

amount of time. This figure makes the pessimistic assumption that communication cannot be overlapped with computation at a processor, even though the two may be independent from each other. Even with this assumption one can see that some communication phases take place concurrently with certain computation phases. While column $k$ travels on the ring, the first processors have started their column updates while the last processors are still in the process of receiving column $k$. As a result, processor $P_k$ is ahead of processor $P_{k+1}$, which makes it possible to pipeline the communication and computation phases as seen in the figure. After each $p$ steps there is a gap, also called a pipeline bubble. These bubbles happen when the processor that is the most behind holds the column that must be sent to all the other processors.

Assume now, like we have done throughout this chapter, that our processors can perform concurrent communication and computation. This assumption makes it possible for the column for step $k+1$ to be communicated along the ring completely asynchronously while the computations for step $k$ are taking place. In this case, communication is, for the most part, fully overlapped with computation and pipeline bubbles can be reduced. Application execution, with this assumption, is depicted in Figure 4.9. Note, however, that while the idle periods are somewhat reduced, they are not completely eliminated.

4.4.3 Look-Ahead Algorithm

Is it possible to improve on the previous algorithm? It turns out that it is, based on a rather clever way of interleaving communication and computation phases. Consider processor $P_{k+1}$.

1. At step $k - 0$, processor $P_0$ receives column 0 from processor $P_k$.
2. It immediately sends it to its successor, processor $P_1$.
3. Then, it updates its columns by executing $\text{UPDATE}(0,j)$ for all $j$ such that $j = 1 \mod p$.
4. It then moves on to step $k - 1$ and prepares column $k - 1$ by executing $\text{PREP}(1)$.

FIGURE 4.9: Example execution of the pipelined LU algorithm on a ring, with concurrent communication and computation at each processor.
Instead, two domains must be stored in memory: the original one, which we deployed in the Jacobi numerical method [97]:

values of a cell's West, East, North, and South neighbors are commonly employed in the Jacobi numerical method.

other stencils are possible. For instance, stencils that use the

dated using the

4.5 A Second Look at Stencil Applications

In Section 4.3 we studied a stencil application in which a cell value is updated using the updated value of its North and West neighbor cells. Many other stencils are possible. For instance, stencils that use the non-updated values of a cell's West, East, North, and South neighbors are commonly employed in the Jacobi numerical method [97]:

\[ c_{new} = \text{update}(c_{old}, W_{old}, E_{old}, N_{old}, S_{old}). \]

With this stencil, which is applied on a 2-D \( n \times n \) domain, it is not possible to update the domain in place without overwriting cell values prematurely. Instead, two domains must be stored in memory: the original one, which we call \( A \), and an empty one into which updated values will be stored, which we call \( B \). As a result, the execution of the algorithm is in fact simpler than that for the stencil in Section 4.3, and so is the data distribution.

4.5.1 Parallelization on a Unidirectional Ring

We consider our usual unidirectional ring of \( p \) processors. As done previously, we assume that \( p \) divides \( n \) to simplify the performance analysis. The most natural data distribution is to allocate \( r = n/p \) consecutive rows of the domain to each processor, exactly like we did for the matrix in Section 4.1. Therefore, each processor has the following declarations:

```plaintext
var A: array[0..r-1,0..n-1] of real;
```
where array $A$ contains initial cell values, and array $B$ is to be filled with updated cell values. Each processor can update rows 1 to $r - 2$ of $B$ using the values stored in $A$. However, the updating of the first row of array $B$ at processor $P_n$ requires values stored in the last row of array $A$ at processor $P_r$. Similarly, the updating of the last row of array $B$ at processor $P_n$ requires values stored in the first row of array $A$ at processor $P_1$. For the sake of simplicity, we assume that the domain “wrap around” in the sense that processors $P_0$ and $P_{n - 1}$ exchange rows. This assumption removes the need for the $q = 0$ or $q = p - 1$ tests that cluttered the algorithm in Section 4.3. Furthermore, this assumption does not decrease performance. Indeed, since the execution goes only as fast as the slowest processor, communication savings on only two out of $p$ processors does not reduce execution time. In practice, it is actually common to avoid special cases and have all the processors perform the exact same communications. This leads to simpler algorithms, as we will see, and data that have been communicated superfluously can be explicitly ignored by some processors when they perform their computations.

With the above assumption, each processor must send data to both its successor and its predecessor on the ring. Given that we use a unidirectional ring, sending data to a predecessor amounts to sending those data all around the ring with $p - 1$ hops, with intermediate processors merely relaying messages. Our parallel stencil algorithm on the unidirectional ring is shown in Algorithm 4.10.

![FIGURE 4.10: Example execution of the look-ahead LU algorithm on a ring, with concurrent communication and computation at each processor.](image-url)
This algorithm proceeds in two phases: a first phase from statements 4 to 20 and a second phase from statements 21 to 28. The first phase consists of two concurrent sub-phases. In the first sub-phase, statements 4 to 11, each processor communicates the top and bottom rows of array $A$ to its predecessor and its successor, respectively. In the second sub-phase, statements 13 to 20, each processor computes all rows of $B$ but for the top and bottom rows. As done previously in this chapter, we assume that processors can compute and communicate concurrently, and can send and receive concurrently. Removing these assumptions would simply mean removing the || sign in the program and changing our performance analysis. By the beginning of the second phase of the algorithm each processor has received the bottom row from its predecessor and the top row from its successor. Therefore, in the second phase, each processor can finally compute the top and bottom rows of array $B$. We detail the critical steps of this algorithm below.

Statement 2 in the program declares array fromP for storing the bottom row sent by the predecessor processor, and array fromS for storing the top row sent by the successor processor. Statement 5 sets the pointer tempS to the beginning of the first row of array $A$, this processor’s top row. Note our use of a function ADDR to access the address of an array element (equivalent to the & operator in C, for instance), and our reliance on the fact that array $A$ is stored in row-major fashion as explained in Section 4.4.1. At the first iteration of the for loop at statement 6 the processor sends its bottom row to its successor, and receives its predecessor’s top row. At every following iteration the processor forwards what it receives from its predecessor to its successor. After $p - 2$ iterations, the final step in statement 9 consists in doing a last communication phase rather cumbersome.

The second phase of the algorithm consists in computing two rows, and thus takes time $2wv$. The overall execution time of the algorithm, $T(n, p)$, is

$$
T(n, p) = \max \{ pL + ph, (n^2/p - 2n)v \} + 2wv.
$$

When $n$ becomes large, $T(n, p) \sim wn^2/p$. Since the sequential execution time is $wn^2$, the parallel algorithm’s asymptotic efficiency is 1.

### 4.5.2 Parallelization on a Bidirectional Ring

The previous algorithm is overly complicated. Indeed, because we used a unidirectional ring, we had to send messages from a processor to its predecessor around the ring with $p - 1$ hops. In spite of its optimal asymptotic parallel efficiency, in practice the first term of the maximum in Equation (4.1) may be large (for instance, if $w$ and $n$ are relatively small). Furthermore, and perhaps more importantly, using a unidirectional ring makes the code for the communication phase rather cumbersome.

Since as parallel algorithm designers we choose which logical topology to use, we can just decide to use a bidirectional ring if we feel it is more appropriate. Let us then redefine the $\text{SEND}$ and $\text{RECV}$ functions to allow direct communication to both the predecessor and the successor processor. We just add an argument, with value $\text{pred}$ or $\text{succ}$, to both calls to specify the direction of the communication. For instance, $\text{SEND}(\text{pred, addr, 1})$ is used by a processor to send one data element stored at address $\text{addr}$ to its predecessor. The predecessor must place a matching call such as $\text{RECV}(\text{succ, addr, 1})$. This new logical topology, the communication phase of the algorithm is rewritten as

\[
\text{SEND}(\text{pred, addr, 0, 0, n}) || \text{RECV}(\text{succ, fromS}, n) \\
\text{SEND}(\text{succ, addr, 0, 1, 0, n}) || \text{RECV}(\text{pred, fromP}, n)
\]

With this new communication phase, the algorithm’s execution time becomes

$$
T(n, p) = \max \{ 2(L + nb), (n^2/p - 2n)v \} + 2wv.
$$
4.6 Implementing Logical Topologies

In this chapter, we have seen our first example of changing a logical topology to better fit the requirements of an algorithm. We even made the statement that as designers of parallel algorithms we can choose the logical topology. Some readers may wonder how this can be done in practice. It turns out that message passing libraries, for instance, ones implemented according to the MPI standard [209], assign an integer identifier to each processor and allow communications between any pair of processors. This is done via the Send andRecv functions that take processor identifiers as arguments for specifying communication sources and destinations. As a result, the developer has complete control over which logical communication paths are used by the algorithm. Using a logical topology restricts communications to only a few paths, which makes algorithm design simpler (provided an appropriate logical topology is chosen). When defining a logical topology, one just arbitrarily defines for each processor which processors are its neighbors. One then decides which way communication can flow on each logical link between two neighboring processors. The logical topology is then implemented via a set of functions by which processors identify their neighbors. For instance, one could implement a function \( \text{LeftNeighbor}(q) \) that, given a processor’s identifier \( q \), returns the identifier of the processor that is the left neighbor of processor \( q \) in the logical topology, for some definition of “left.” A call such as \( \text{Send}(\text{LeftNeighbor}(q), \text{Recv}(\text{LeftNeighbor}(q)), \text{add}) \) then implements communication on the logical topology. While in this chapter we have only used a ring topology, many others are used in practice. In the next chapter, for instance, we will see logical 2-D grid and torus topologies. Other common topologies include single-level and multi-level trees, or topologies that combine any of the above.

A difficult question is that of matching the logical topology to the physical topology. Indeed, it is often the case that an algorithm can be implemented using different logical topologies. The SCALAPACK library [36], for instance, allows the user to choose among several logical topologies for executing parallel linear algebra algorithms. The common wisdom is that a logical topology that resembles the underlying physical topology should lead to good performance. If the physical network topology is, say, a tree, implementing a logical ring topology would cause several performance difficulties. For instance, the communication times between neighboring processors on the logical ring would vary among pairs of processors, which could lead to unexpected idle time in the execution of a parallel algorithm. Paradoxically, the point of using a logical topology is often to hide the complexity of the underlying physical topology. Choosing a logical topology that closely resembles the physical topology could make algorithm design very difficult. The goal is then to choose a logical topology that at the same time makes algorithm design natural and is reasonably compatible with the underlying physical topology. Achieving this goal can be straightforward or challenging, depending on the application and on the platform. Note that some modern supercomputers provide multiple physical networks to better support various communication patterns and thus various logical topologies. Also, platforms such as commodity clusters often use switched networks that are akin to fully connected (but possibly hierarchical) topologies. Such physical topologies often support various logical topologies reasonably well, but often extensive benchmarking is required to determine the best logical topology for a given algorithm on a given platform.

In this book we do not study this question of matching logical and physical topology in depth because it depends on the physical characteristics of the parallel platforms at hand, which vary across platforms and across platform generations. However, the logical topologies in this chapter and the ones we study in the next chapter are known to be useful in the majority of practical scenarios.

4.7 Distributed vs. Centralized Implementations

When designing the parallel algorithms in this chapter we have always assumed that data (i.e., arrays) were already distributed among processors at the onset of the algorithm execution. For instance, in our matrix-vector multiplication algorithm in Section 4.1, we assumed that at the onset of the parallel execution each processor magically holds a particular subset of the row or column of matrix \( A \). The reader may wonder how this distribution of data occurs in the first place and whether the algorithm should distribute the data directly. One can distinguish two approaches: distributed and centralized. In this book we have opted for always using the distributed approach. As a result, our algorithm implementations do not perform any data distribution and we assume that the data are already distributed. In the centralized approach one assumes instead that data reside at a single “master” location (a processor or, if the data are large, a file on disk). The algorithm must then distribute the data among available processors, apply the desired parallel algorithm, and “un-distribute” (gather) the results of the computation back to the master location. While data distribution has to occur one way or another, the...
question of which approach is best arises when developing a general-purpose library of routines that implement parallel, distributed-memory algorithms. In other terms, should the library user or the library itself be responsible for distributing data?

One advantage of the centralized approach is that each library routine can enforce whichever data distribution is deemed best by the library developer. As a result, the library user does not have to worry about data distribution, which is another advantage. Unfortunately, for best performance, the choice of data distribution for a given algorithm must account for the particular underlying physical topology. Since this topology is not known at library development time, the library developer may wish to implement multiple versions of each routine for different possible data distributions. The user then would have to choose which routine to call depending on the underlying platform. As discussed in Section 4.6, this choice may be difficult without extensive benchmarking. The main drawback of the centralized approach, however, emerges when the user calls multiple library routines in sequence with the same data. For instance, imagine that a user wishes to compute a matrix product $C = A \cdot B$, followed by an LU factorization of matrix $C$, followed by a sequence of $\text{(LU)}x_i = b_i$ linear system resolutions. Data will be distributed and redistributed at each library call, which typically leads to prohibitive overhead. Unfortunately, such use of a library is common and, as a result, most library developers opt for a distributed implementation.

For the same reasons as those given in the case of a centralized approach, routines in a distributed library implementation should support multiple data distribution schemes for best performance. This leads to an interesting trade-off. Indeed, the user may have called a library routine and used a given distribution for a given matrix. When calling another routine for which a different distribution would be best, a trade-off arises. Is it better to incur the overhead of re-distributing the matrix from the old distribution to the new distribution, which can be significant, or is it better to call the most routine with a sub-optimal data distribution? This trade-off is difficult to resolve, especially at library development time. Consequently, the typical approach is to provide routines that can execute the necessary algorithms using a variety of data distribution schemes, and routines that can redistribute data from any distribution scheme to any other. The user is then given maximum flexibility for data distributions and data re-distributions. By contrast, a centralized implementation offers at best marginal flexibility. Some users argue that this flexibility only leads to endless dilemmas, but it is the price to pay for the hope of achieving best performance.

4.8 Summary of Algorithmic Principles

Throughout this chapter we have highlighted several principles that are commonly employed when designing and implementing distributed memory algorithms to reduce processor idle time and/or to reduce communication overhead. While our focus was the ring topology, these principles are generally applicable. Unfortunately, as seen in previous sections, these principles often conflict with each other and force the algorithm developer to understand and choose among multiple trade-offs. We summarize the principles below:

- **Sending data in bulk** - Aggregating communication operations reduces communication overhead due to network latencies.
- **Sending data early** - Sending data as early as possible allows processors to start computing as early as possible, which reduces processor idle time.
- **Overlapping communication and computation** - It is always a good idea to overlap communication and computation in order to hide communication time.
- **Block data distribution** - Using a block distribution by which processors are assigned blocks of contiguous data elements reduces the amount of communication (and can often improve cache reuse).
- **Cyclic data distribution** - Using a cyclic distribution by which data elements are interleaved among processors makes it possible to achieve better load balancing between processors, which reduces processor idle time.

Bibliographical Notes

In terms of algorithms, a seminal reference is the book by Kumar et al. [76]. The content of this chapter’s section on matrix-vector and matrix-matrix multiplications belongs to parallel computing knowledge. The discussion and performance modeling of stencil applications are inspired by the articles by Miguet and Robert [90, 91]. Finally, the parallel Gaussian elimination algorithm used in our LU factorization is a classic (see [101] and other cited references).