## 1.10 SUMMARY

For many years we've enjoyed the fruits of ever faster processors. However, because of physical limitations the rate of performance improvement in conventional processors is decreasing. In order to increase the power of processors, chipmakers have turned to **multicore** integrated circuits, that is, integrated circuits with multiple conventional processors on a single chip.

Ordinary **serial** programs, which are programs written for a conventional single-core processor, usually cannot exploit the presence of multiple cores, and it's unlikely that translation programs will be able to shoulder all the work of **parallelizing** serial programs, meaning converting them into **parallel programs**, which can make use of multiple cores. As software developers, we need to learn to write parallel programs.

When we write parallel programs, we usually need to **coordinate** the work of the cores. This can involve **communication** among the cores, **load balancing**, and **synchronization** of the cores.

In this book we'll be learning to program parallel systems so that we can maximize their performance. We'll be using the C language with either MPI, Pthreads, or OpenMP. MPI is used for programming **distributed-memory** systems, and Pthreads and OpenMP are used for programming **shared-memory** systems. In distributed-memory systems, the cores have their own private memories, while in shared-memory systems, it's possible, in principle, for each core to access each memory location.

**Concurrent** programs can have multiple tasks in progress at any instant. **Parallel** and **distributed** programs usually have tasks that execute simultaneously. There isn't a hard and fast distinction between parallel and distributed, although in parallel programs, the tasks are usually more tightly coupled.

Parallel programs are usually very complex. So it's even more important to use good program development techniques with parallel programs.

## 1.11 EXERCISES

**1.1** Devise formulas for the functions that calculate `my_first_i` and `my_last_i` in the global sum example. Remember that each core should be assigned roughly the same number of elements of computations in the loop. *Hint*: First consider the case when $n$ is evenly divisible by $p$.

**1.2** We've implicitly assumed that each call to `Compute_next_value` requires roughly the same amount of work as the other calls. How would you change your answer to the preceding question if call $i = k$ requires $k + 1$ times as much work as the call with $i = 0$? So if the first call ($i = 0$) requires 2 milliseconds, the second call ($i = 1$) requires 4, the third ($i = 2$) requires 6, and so on.

**1.3** Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, ...).

Some application programming interfaces or APIs for parallel systems define new programming languages. However, most extend existing languages, either through a library of functions—for example, functions to pass messages—or extensions to the compiler for the serial language. This latter approach will be the focus of this text. We'll be using parallel extensions to the C language.

When we want to be explicit about compiling and running programs, we'll use the command line of a Unix shell, the `gcc` compiler or some extension of it (e.g., `mpicc`), and we'll start programs from the command line. For example, if we wanted to show compilation and execution of the "hello, world" program from Kernighan and Ritchie [29], we might show something like this:

```
$ gcc -g -Wall -o hello hello.c
$ ./hello
hello, world
```

The $-sign is the prompt from the shell. We will usually use the following options for the compiler:

- `-g`. Create information that allows us to use a debugger
- `-Wall`. Issue lots of warnings
- `-o <outfile>`. Put the executable in the file named `outfile`
- When we're timing programs, we usually tell the compiler to optimize the code by using the `-O2` option.

In most systems, user directories or folders are not, by default, in the user's execution path, so we'll usually start jobs by giving the path to the executable by adding `./` to its name.

## 2.10 SUMMARY

There's a *lot* of material in this chapter, and a complete summary would run on for many pages, so we'll be very terse.

### 2.10.1 Serial systems

We started with a discussion of conventional serial hardware and software. The standard model of computer hardware has been the **von Neumann architecture**, which consists of a **central processing unit** that carries out the computations, and **main memory** that stores data and instructions. The separation of CPU and memory is often called the **von Neumann bottleneck** since it limits the rate at which instructions can be executed.

Perhaps the most important software on a computer is the **operating system**. It manages the computer's resources. Most modern operating systems are **multitasking**. Even if the hardware doesn't have multiple processors or cores, by rapidly switching among executing programs, the OS creates the illusion that multiple jobs are running simultaneously. A running program is called a **process**. Since a running

process is more or less autonomous, it has a lot of associated data and information. A **thread** is started by a process. A thread doesn't need as much associated data and information, and threads can be stopped and started much faster than processes.

In computer science, **caches** are memory locations that can be accessed faster than other memory locations. CPU caches are memory locations that are intermediate between the CPU registers and main memory. Their purpose is to reduce the delays associated with accessing main memory. Data are stored in cache using the principle of locality, that is, items that are close to recently accessed items are more likely to be accessed in the near future. Thus, contiguous **blocks** or **lines** of data and instructions are transferred between main memory and caches. When an instruction or data item is accessed and it's already in cache, it's called a cache **hit**. If the item isn't in cache, it's called a cache **miss**. Caches are directly managed by the computer hardware, so programmers can only indirectly control caching.

Main memory can also function as a cache for secondary storage. This is managed by the hardware and the operating system through **virtual memory**. Rather than storing all of a program's instructions and data in main memory, only the active parts are stored in main memory, and the remaining parts are stored in secondary storage called **swap space**. Like CPU caches, virtual memory operates on blocks of contiguous data and instructions, which in this setting are called **pages**. Note that instead of addressing the memory used by a program with physical addresses, virtual memory uses **virtual addresses**, which are independent of actual physical addresses. The correspondence between physical addresses and virtual addresses is stored in main memory in a **page table**. The combination of virtual addresses and a page table provides the system with the flexibility to store a program's data and instructions anywhere in memory. Thus, it won't matter if two different programs use the same virtual addresses. With the page table stored in main memory, it could happen that every time a program needed to access a main memory location it would need two memory accesses: one to get the appropriate page table entry so it could find the location in main memory, and one to actually access the desired memory. In order to avoid this problem, CPUs have a special page table cache called the **translation lookaside buffer**, which stores the most recently used page table entries.

**Instruction-level parallelism** (ILP) allows a single processor to execute multiple instructions simultaneously. There are two main types of ILP: **pipelining** and **multiple issue**. With pipelining, some of the functional units of a processor are ordered in sequence, with the output of one being the input to the next. Thus, while one piece of data is being processed by, say, the second functional unit, another piece of data can be processed by the first. With multiple issue, the functional units are replicated, and the processor attempts to simultaneously execute different instructions in a single program.

Rather than attempting to simultaneously execute individual instructions, **hardware multithreading** attempts to simultaneously execute different threads. There are several different approaches to implementing hardware multithreading. However, all of them attempt to keep the processor as busy as possible by rapidly switching

between threads. This is especially important when a thread **stalls** and has to wait (e.g., for a memory access to complete) before it can execute an instruction. In **simultaneous multithreading**, the different threads can simultaneously use the multiple functional units in a multiple issue processor. Since threads consist of multiple instructions, we sometimes say that **thread-level paralleism**, or TLP, is **coarser-grained** than ILP.

### 2.10.2 Parallel hardware

ILP and TLP provide parallelism at a very low level; they're typically controlled by the processor and the operating system, and their use isn't directly controlled by the programmer. For our purposes, hardware is parallel if the parallelism is visible to the programmer, and she can modify her source code to exploit it.

Parallel hardware is often classified using **Flynn's taxonomy**, which distinguishes between the number of instruction streams and the number of data streams a system can handle. A von Neumann system has a single instruction stream and a single data stream so it is classified as a **single instruction, single data**, or **SISD**, system.

A **single instruction, multiple data**, or **SIMD**, system executes a single instruction at a time, but the instruction can operate on multiple data items. These systems often execute their instructions in lockstep: the first instruction is applied to all of the data elements simultaneously, then the second is applied, and so on. This type of parallel system is usually employed in **data parallel** programs, programs in which the data are divided among the processors and each data item is subjected to more or less the same sequence of instructions. **Vector processors** and **graphics processing units** are often classified as SIMD systems, although current generation GPUs also have characteristics of multiple instruction, multiple data stream systems.

Branching in SIMD systems is handled by idling those processors that might operate on a data item to which the instruction doesn't apply. This behavior makes SIMD systems poorly suited for **task-parallelism**, in which each processor executes a different task, or even data-parallelism, with many conditional branches.

As the name suggests, **multiple instruction, multiple data**, or **MIMD**, systems execute multiple independent instruction streams, each of which can have its own data stream. In practice, MIMD systems are collections of autonomous processors that can execute at their own pace. The principal distinction between different MIMD systems is whether they are **shared-memory** or **distributed-memory** systems. In shared-memory systems, each processor or core can directly access every memory location, while in distributed-memory systems, each processor has its own private memory. Most of the larger MIMD systems are **hybrid** systems in which a number of relatively small shared-memory systems are connected by an interconnection network. In such systems, the individual shared-memory systems are sometimes called **nodes**. Some MIMD systems are **heterogeneous** systems, in which the processors have different capabilities. For example, a system with a conventional CPU and a GPU is a heterogeneous system. A system in which all the processors have the same architecture is **homogeneous**.

There are a number of different interconnects for joining processors to memory in shared-memory systems and for interconnecting the processors in distributed-memory or hybrid systems. The most commonly used interconnects for shared-memory are **buses** and **crossbars**. Distributed-memory systems sometimes use **direct** interconnects such as **toroidal meshes** and **hypercubes**, and they sometimes use **indirect** interconnects such as crossbars and **multistage** networks. Networks are often evaluated by examining the **bisection width** or the **bisection bandwidth** of the network. These give measures of how much simultaneous communication the network can support. For individual communications between nodes, authors often discuss the **latency** and **bandwidth** of the interconnect.

A potential problem with shared-memory systems is **cache coherence**. The same variable can be stored in the caches of two different cores, and if one core updates the value of the variable, the other core may be unaware of the change. There are two main methods for insuring cache coherence: **snooping** and the use of **directories**. Snooping relies on the capability of the interconnect to broadcast information from each cache controller to every other cache controller. Directories are special distributed data structures, which store information on each cache line. Cache coherence introduces another problem for shared-memory programming: **false sharing**. When one core updates a variable in one cache line, and another core wants to access *another* variable in the same cache line, it will have to access main memory, since the unit of cache coherence is the cache line. That is, the second core only "knows" that the line it wants to access has been updated. It doesn't know that the variable it wants to access hasn't been changed.

### 2.10.3 Parallel software

In this text we'll focus on developing software for homogeneous MIMD systems. Most programs for such systems consist of a single program that obtains parallelism by branching. Such programs are often called **single program, multiple data** or **SPMD** programs. In shared-memory programs we'll call the instances of running tasks threads; in distributed-memory programs we'll call them processes.

Unless our problem is **embarrassingly parallel**, the development of a parallel program needs at a minimum to address the issues of **load balance**, **communication**, and **synchronization** among the processes or threads.

In shared-memory programs, the individual threads can have **private** and **shared-memory**. Communication is usually done through **shared variables**. Any time the processors execute asynchronously, there is the potential for **nondeterminism**, that is, for a given input the behavior of the program can change from one run to the next. This can be a serious problem, especially in shared-memory programs. If the nondeterminism results from two threads' attempts to access the same resource, and it can result in an error, the program is said to have a **race condition**. The most common place for a race condition is a **critical section**, a block of code that can only be executed by one thread at a time. In most shared-memory APIs, **mutual exclusion**

in a critical section can be enforced with an object called a **mutual exclusion lock** or **mutex**. Critical sections should be made as small as possible, since a mutex will allow only one thread at a time to execute the code in the critical section, effectively making the code serial.

A second potential problem with shared-memory programs is **thread safety**. A block of code that functions correctly when it is run by multiple threads is said to be **thread safe**. Functions that were written for use in serial programs can make unwitting use of shared data—for example, static variables—and this use in a multithreaded program can cause errors. Such functions are not thread safe.

The most common API for programming distributed-memory systems is **message-passing**. In message-passing, there are (at least) two distinct functions: a **send** function and a **receive** function. When processes need to communicate, one calls the send and the other calls the receive. There are a variety of possible behaviors for these functions. For example, the send can **block** or wait until the matching receive has started, or the message-passing software can can copy the data for the message into its own storage, and the sending process can return before the matching receive has started. The most common behavior for receives is to block until the message has been received. The most commonly used message-passing system is called the **Message-Passing Interface** or MPI. It provides a great deal of functionality beyond simple sends and receives.

Distributed-memory systems can also be programmed using **one-sided communications**, which provide functions for accessing memory belonging to another process, and **partitioned global address space** languages, which provide some shared-memory functionality in distributed-memory systems.

### 2.10.4 Input and output

In general parallel systems, multiple cores can access multiple secondary storage devices. We won't attempt to write programs that make use of this functionality. Rather, we'll write programs in which one process or thread can access `stdin`, and all processes can access `stdout` and `stderr`. However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing `stdout`.

### 2.10.5 Performance

If we run a parallel program with $p$ processes or threads and no more than one process/thread per core, then our ideal would be for our parallel program to run $p$ times faster than the serial program. This is called **linear speedup**, but in practice it is rarely achieved. If we denote the run-time of the serial program by $T_{\text{serial}}$ and the parallel program's run-time by $T_{\text{parallel}}$, then the **speedup** $S$ and **efficiency** $E$ of the parallel program are given by the formulas

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \text{ and } E = \frac{T_{\text{serial}}}{pT_{\text{parallel}}},$$

respectively. So linear speedup has $S = p$ and $E = 1$. In practice, we will almost always have $S < p$ and $E < 1$. If we fix the problem size, $E$ usually decreases as we increase $p$, while if we fix the number of processes/threads, then $S$ and $E$ often increase as we increase the problem size.

**Amdahl's law** provides an upper bound on the speedup that can be obtained by a parallel program: if a fraction $r$ of the original, serial program isn't parallelized, then we can't possibly get a speedup better than $1/r$, regardless of how many processes/threads we use. In practice many parallel programs obtain excellent speedups. One possible reason for this apparent contradiction is that Amdahl's law doesn't take into consideration the fact that the unparallelized part often decreases in size relative to the parallelized part as the problem size increases.

**Scalability** is a term that has many interpretations. In general, a technology is scalable if it can handle ever-increasing problem sizes. Formally, a parallel program is scalable if there is a rate at which the problem size can be increased so that as the number of processes/threads is increased, the efficiency remains constant. A program is **strongly** scalable, if the problem size can remain fixed, and it's **weakly** scalable if the problem size needs to be increased at the same rate as the number of processes/threads.

In order to determine $T_{\text{serial}}$ and $T_{\text{parallel}}$, we usually need to include calls to a timer function in our source code. We want these timer functions to give **wall clock** time, not CPU time, since the program may be "active"—for example, waiting for a message—even when the CPU is idle. To take parallel times, we usually want to synchronize the processes/threads before starting the timer, and, after stopping the timer, find the maximum elapsed time among all the processes/threads. Because of system variability, we usually need to run a program several times with a given data set, and we usually take the minimum time from the multiple runs. To reduce variability and improve overall run-times, we usually run no more than one thread per core.

### 2.10.6 Parallel program design

**Foster's methodology** provides a sequence of steps that can be used to design parallel programs. The steps are **partitioning** the problem to identify tasks, identifying **communication** among the tasks, **agglomeration** or **aggregation** to group tasks, and **mapping** to assign aggregate tasks to processes/threads.

### 2.10.7 Assumptions

We'll be focusing on the development of parallel programs for both shared- and distributed-memory MIMD systems. We'll write SPMD programs that usually use **static** processes or threads—processes/threads that are created when the program begins execution, and are not shut down until the program terminates. We'll also assume that we run at most one process or thread on each core of the system.

```
void Merge_low(
      int  my_keys[],     /* in/out    */
      int  recv_keys[],   /* in        */
      int  temp_keys[],   /* scratch   */
      int  local_n        /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

**Program 3.16:** The `Merge_low` function in parallel odd-even transposition sort

## 3.8 SUMMARY

MPI, or the Message-Passing Interface, is a library of functions that can be called from C, C++, or Fortran programs. Many systems use `mpicc` to compile MPI programs and `mpiexec` to run them. C MPI programs should include the `mpi.h` header file to get function prototypes and macros defined by MPI.

`MPI_Init` does the setup needed to run MPI. It should be called before other MPI functions are called. When your program doesn't use `argc` and `argv`, `NULL` can be passed for both arguments.

In MPI a **communicator** is a collection of processes that can send messages to each other. After an MPI program is started, MPI always creates a communicator consisting of all the processes. It's called `MPI_COMM_WORLD`.

Many parallel programs use the **single program, multiple data,** or **SPMD,** approach, whereby running a single program obtains the effect of running multiple different programs by including branches on data such as the process rank. When you're done using MPI, you should call `MPI_Finalize`.

To send a message from one MPI process to another, you can use `MPI_Send`. To receive a message, you can use `MPI_Recv`. The arguments to `MPI_Send` describe the contents of the message and its destination. The arguments to `MPI_Recv` describe the storage that the message can be received into, and where the message should be received from. `MPI_Recv` is **blocking**, that is, a call to `MPI_Recv` won't return until the

message has been received (or an error has occurred). The behavior of `MPI_Send` is defined by the MPI implementation. It can either block, or it can **buffer** the message. When it blocks, it won't return until the matching receive has started. If the message is buffered, MPI will copy the message into its own private storage, and `MPI_Send` will return as soon as the message is copied.

When you're writing MPI programs, it's important to differentiate between **local** and **global** variables. Local variables have values that are specific to the process on which they're defined, while global variables are the same on all the processes. In the trapezoidal rule program, the total number of trapezoids $n$ was a global variable, while the left and right endpoints of each process' interval were local variables.

Most serial programs are **deterministic**, meaning if we run the same program with the same input we'll get the same output. Recall that parallel programs often don't possess this property—if multiple processes are operating more or less independently, the processes may reach various points at different times, depending on events outside the control of the process. Thus, parallel programs can be **nondeterministic**, that is, the same input can result in different outputs. If all the processes in an MPI program are printing output, the order in which the output appears may be different each time the program is run. For this reason, it's common in MPI programs to have a single process (e.g., process 0) handle all the output. This rule of thumb is usually ignored during debugging, when we allow each process to print debug information.

Most MPI implementations allow all the processes to print to `stdout` and `stderr`. However, every implementation we've encountered only allows at most one process (usually process 0 in `MPI_COMM_WORLD`) to read from `stdin`.

**Collective communications** involve all the processes in a communicator, so they're different from `MPI_Send` and `MPI_Recv`, which only involve two processes. To distinguish between the two types of communications, functions such as `MPI_Send` and `MPI_Recv` are often called **point-to-point** communications.

Two of the most commonly used collective communication functions are `MPI_Reduce` and `MPI_Allreduce`. `MPI_Reduce` stores the result of a global operation (e.g., a global sum) on a single designated process, while `MPI_Allreduce` stores the result on all the processes in the communicator.

In MPI functions such as `MPI_Reduce`, it may be tempting to pass the same actual argument to both the input and output buffers. This is called **argument aliasing**, and MPI explicitly prohibits aliasing an output argument with another argument.

We learned about a number of other important MPI collective communications:

- `MPI_Bcast` sends data from a single process to all the processes in a communicator. This is very useful if, for example, process 0 reads data from `stdin` and the data needs to be sent to all the processes.
- `MPI_Scatter` distributes the elements of an array among the processes. If the array to be distributed contains $n$ elements, and there are $p$ processes, then the first $n/p$ are sent to process 0, the next $n/p$ to process 1, and so on.
- `MPI_Gather` is the "inverse operation" to `MPI_Scatter`. If each process stores a subarray containing $m$ elements, `MPI_Gather` will collect all of the elements onto

a designated process, putting the elements from process 0 first, then the elements from process 1, and so on.

- `MPI_Allgather` is like `MPI_Gather` except that it collects all of the elements onto *all* the processes.
- `MPI_Barrier` approximately synchronizes the processes; no process can return from a call to `MPI_Barrier` until all the processes in the communicator have started the call.

In distributed-memory systems there is no globally shared-memory, so partitioning global data structures among the processes is a key issue in writing MPI programs. For ordinary vectors and arrays, we saw that we could use block partitioning, cyclic partitioning, or block-cyclic partitioning. If the global vector or array has $n$ components and there are $p$ processes, a **block partition** assigns the first $n/p$ to process 0, the next $n/p$ to process 1, and so on. A **cyclic partition** assigns the elements in a "round-robin" fashion: the first element goes to 0, the next to 1, ..., the $p$th to $p-1$. After assigning the first $p$ elements, we return to process 0, so the $(p+1)$st goes to process 0, the $(p+2)$nd to process 1, and so on. A **block-cyclic partition** assigns blocks of elements to the processes in a cyclic fashion.

Compared to operations involving only the CPU and main memory, sending messages is expensive. Furthermore, sending a given volume of data in fewer messages is usually less expensive than sending the same volume in more messages. Thus, it often makes sense to reduce the number of messages sent by combining the contents of multiple messages into a single message. MPI provides three methods for doing this: the `count` argument to communication functions, derived datatypes, and `MPI_Pack/Unpack`. Derived datatypes describe arbitrary collections of data by specifying the types of the data items and their relative positions in memory. In this chapter we took a brief look at the use of `MPI_Type_create_struct` to build a derived datatype. In the exercises, we'll explore some other methods, and we'll take a look at `MPI_Pack/Unpack`

When we time parallel programs, we're usually interested in elapsed time or "wall clock time," which is the total time taken by a block of code. It includes time in user code, time in library functions, time in operating system functions started by the user code, and idle time. We learned about two methods for finding wall clock time: `GET_TIME` and `MPI_Wtime`. `GET_TIME` is a macro defined in the file `timer.h` that can be downloaded from the book's website. It can be used in serial code as follows:

```
#include "timer.h"  // From the book's website
. . .
double start, finish, elapsed;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
elapsed = finish - start;
printf("Elapsed time = %e seconds\n", elapsed);
```

MPI provides a function, MPI_Wtime, that can be used instead of GET_TIME. In spite of this, timing parallel code is more complex, since ideally we'd like to synchronize the processes at the start of the code, and then report the time it took for the "slowest" process to complete the code. MPI_Barrier does a fairly good job of synchronizing the processes. A process that calls it will block until all the processes in the communicator have called it. We can use the following template for finding the run-time of MPI code:

```
double start, finish, loc_elapsed, elapsed;
. . .
MPI_Barrier(comm);
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
loc_elapsed = finish - start;
MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX,
      0, comm);
if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

A further problem with taking timings lies in the fact that there is ordinarily considerable variation if the same code is timed repeatedly. For example, the operating system may idle one or more of our processes so that other processes can run. Therefore, we typically take several timings and report their minimum.

After taking timings, we can use the **speedup** or the **efficiency** to evaluate the program performance. The speedup is the ratio of the serial run-time to the parallel run-time, and the efficiency is the speedup divided by the number of parallel processes. The ideal value for speedup is $p$, the number of processes, and the ideal value for the efficiency is 1. We rarely achieve these ideals, but it's not uncommon to see programs that get close to these values, especially when $p$ is small and $n$, the problem size, is large. **Parallel overhead** is the part of the parallel run-time that's due to any additional work that isn't done by the serial program. In MPI programs, parallel overhead will come from communication. When $p$ is large and $n$ is small, it's not unusual for parallel overhead to dominate the total run-time and speedups and efficiencies can be quite low. If it's possible to increase the problem size ($n$) so that the efficiency doesn't decrease as $p$ is increased, a parallel program is said to be **scalable**.

Recall that MPI_Send can either block or buffer its input. An MPI program is **unsafe** if its correct behavior depends on the fact that MPI_Send is buffering its input. This typically happens when multiple processes first call MPI_Send and then call MPI_Recv. If the calls to MPI_Send don't buffer the messages, then they'll block until the matching calls to MPI_Recv have started. However, this will never happen. For example, if both process 0 and process 1 want to send data to each other, and both send first and then receive, process 0 will wait forever for process 1 to call MPI_Recv, since process 1 is blocked in MPI_Send, and process 1 will wait forever for process 0.

That is, the processes will hang or **deadlock**—they'll block forever waiting for events that will never happen.

An MPI program can be checked for safety by replacing each call to `MPI_Send` with a call to `MPI_Ssend`. `MPI_Ssend` takes the same arguments as `MPI_Send`, but it always blocks until the matching receive has started. The extra "s" stands for *synchronous*. If the program completes correctly with `MPI_Ssend` for the desired inputs and communicator sizes, then the program is safe.

An unsafe MPI program can be made safe in several ways. The programmer can schedule the calls to `MPI_Send` and `MPI_Recv` so that some processes (e.g., even-ranked processes) first call `MPI_Send` while others (e.g., odd-ranked processes) first call `MPI_Recv`. Alternatively, we can use `MPI_Sendrecv` or `MPI_Sendrecv_replace`. These functions execute both a send and a receive, but they're guaranteed to schedule them so that the program won't hang or crash. `MPI_Sendrecv` uses different arguments for the send and the receive buffers, while `MPI_Sendrecv_replace` uses the same buffer for both.

## 3.9 EXERCISES

**3.1.** What happens in the greetings program if, instead of `strlen(greeting)+1`, we use `strlen(greeting)` for the length of the message being sent by processes $1, 2, \ldots,$ `comm_sz-1`? What happens if we use `MAX_STRING` instead of `strlen(greeting) + 1`? Can you explain these results?

**3.2.** Modify the trapezoidal rule so that it will correctly estimate the integral even if `comm_sz` doesn't evenly divide $n$. (You can still assume that $n \geq$ `comm_sz`.)

**3.3.** Determine which of the variables in the trapezoidal rule program are local and which are global.

**3.4.** Modify the program that just prints a line of output from each process (`mpi_output.c`) so that the output is printed in process rank order: process 0's output first, then process 1's, and so on.

**3.5.** In a binary tree, there is a unique shortest path from each node to the root. The length of this path is often called the **depth** of the node. A binary tree in which every nonleaf has two children is called a **full** binary tree, and a full binary tree in which every leaf has the same depth is sometimes called a **complete** binary tree. See Figure 3.14. Use the principle of mathematical induction to prove that if $T$ is a complete binary tree with $n$ leaves, then the depth of the leaves is $\log_2(n)$.

**3.6.** Suppose `comm_sz` $= 4$ and suppose that $\mathbf{x}$ is a vector with $n = 14$ components.

**a.** How would the components of $\mathbf{x}$ be distributed among the processes in a program that used a block distribution?