Java is an object-oriented programming language, this much is true. It’s also an Internet programming language. These two features of Java are enough to make it an exciting and compelling programming language in its own right. However, Java is successful to a large extent because it is the quintessential concurrent and distributed programming language.

Concurrency is a far-reaching topic. This book is primarily focused on the use of Java for programming the “server side” (symmetric multiprocessor systems) and clusters of workstations. This chapter provides an overview of the key principles behind concurrent and parallel systems with discussions of architecture, performance analysis, challenges and complications arising in concurrent systems, and a quick survey of some programming techniques. These discussions are intentionally generic in nature. If you absolutely must “take a sip” of Java, please proceed directly to Chapter 2. We strongly recommend, however, that you read this chapter throughout your travels with this book.
The von Neumann Machine

A discussion of parallel computing must begin with a discussion of sequential computing and the von Neumann machine—our sequential computer. The von Neumann machine is one of the computer designs of John von Neumann. A processor fetches instructions and data from a memory, operates on the data, and writes the results back into memory. Computation is accomplished by making small incremental changes in the global memory.

The problem with the von Neumann machine is that the design relies on making a sequence of small changes, a highly sequential process. Note that current programming languages are designed assuming that the von Neumann machine will be used. The assignment statement fetches data from memory on the right hand side, performs computations, and writes results back into the memory for the left-hand-side variable. The statements are executed sequentially, with control accomplished by branches. In the language, the branches are given the syntactic sugar of if statements, while statements, and so on.

There is a problem in trying to speed up von Neumann machines. They are inherently sequential in principle. Attempts may be made to execute several instructions at once (super-scalar execution), but that gives only a few times the speedup. Similarly, it is difficult to gain high speedup from a program written in a von Neumann language without doing an extensive rewrite of the program.

Flynn’s Taxonomy

Flynn produced a taxonomy of parallel machines that is widely used. He classified computers with respect to how many different instruction streams they are fetching and executing at the same time, and by how many data sets (data streams) they are fetching and processing. His taxonomy is as follows:

- **SISD**—single instruction stream–single data stream: the familiar von Neumann machine. Fetching one sequence of instructions and fetching the data and the instructions address from one memory.
- **MIMD**—(pronounced “mim-dee”) multiple instruction–multiple data stream: a multiprocessor or multicomputer (and the subject of this book). Here, several processors are fetching their own instructions and operating on the data those instructions specify. To gain speedup on individual programs, these processors must synchronize and communicate with each other.
- **SIMD**—(pronounced “sim-dee”) single instruction stream–multiple data stream: These machines typically are used to process arrays. A single processor fetches instructions and broadcasts those instructions to a number of data units. Those data units fetch data and perform operations on them. The appropriate programming language for such machines has a single flow of control (like a Von Neumann language), but has operations that can operate on entire arrays, rather than on individual array
elements. The hardware needs ways in which data units will not execute some operations based on tests of their own data (e.g., so that some units can turn off for the then and others for the else parts of if-then-else statements), and it needs to let the control unit read the and or the or of the results of data tests at the data units (e.g. to know when all units have finished executing a while loop).

- MISD—multiple instruction stream–single data stream: It’s not totally clear what machines fit into this category. One kind of MISD machine would be designed for fail-safe operation; several processors perform the same operations on the same data and check each other to be sure that any failure will be caught.

Another proposed MISD machine is a systolic array processor. Streams of data are fetched from memory and passed through arrays of processors. The individual processors perform their own operations on the streams of data passing through them, but they have no control over where to fetch data from.

MIMD machines are divided into two varieties: shared memory and distributed memory. Shared-memory machines have several processors accessing a common memory. Unless the machine is for a special purpose, the processors will be accessing the shared memory through some sort of address-mapping hardware. To be used for parallel processing, the software must let the processors actually share that memory in their address spaces. Shared-memory machines have significant advantages for programming. All of the processors working on a common problem can share the large data structures (e.g., large arrays) and cooperate by working on parts of the data structure, while other processors work on other parts.

The problems with programming shared-memory machines have to do with synchronizing the processors. Since the processors work by fetching and storing small data elements, a processor updating a large data structure cannot read or write it in one instant. This means that if one processor is reading and another writing the same data structure at the same time, the reader may be getting some old components and some new ones. The state will not be consistent, and the computation will therefore become confused (“confused,” meaning there is an inconsistent state.) Similarly, if two processors are trying to write the same structure at the same time, parts of the two writes will be confused. Therefore, the software for shared-memory parallel machines must provide facilities for coordinating processors. The problem with programming is to make sure the coordination is done correctly.

There is another problem with shared-memory machines: It’s hard to build them to be large. The switch between the processors and memory becomes a bottleneck, limiting the traffic between processors and memory, or it tends to become expensive or slow. This is particularly the problem with UMA machines (Uniform Memory Access). UMA machines take the same amount of time to get to all memory locations. As the UMA machines get larger, physical packaging alone dictates that some memory will get further
from some processors than it will for smaller versions. When the problems of switching more processors to more memory chips are added, UMAs can be expected to get slower still.

An alternative to UMA is NUMA (nonuniform memory access) machines. Typically, NUMA machines have some memory attached to each processor, which the processor can access quickly. To get to the memory attached to another processor, the processor must go through some sort of switch, which slows down the access. By careful placement and replication of some data and subroutines, NUMA machines can have many of the programming conveniences of UMA machines, but with cheaper hardware, larger numbers of processors, and reasonable speed. However, programmers tend to discover that they can gain even better performance by copying entire data structures into local memory, operating on them locally, and writing them back to local memory. At this point, their code becomes more complex and less portable. Distributed-memory MIMD machines (MIMD-DM) are much easier to build, but much harder to program. The MIMD-DM machine is basically a collection of computers, called nodes, connected through a high-performance network. The major programming problem is that the individual machines must coordinate and communicate data by message passing; it often requires entire redesigns of programs to port them to MIMD-DM machines. The only way to access data on another node is to have that node send it to you. For that to happen, the other node must know that you need the data—or you must send it a request message, and it must be programmed to reply.

It is arguable that the change from shared-memory to distributed-memory machines is a radical shift; it goes to the root of how one thinks about computations. On the von Neumann machine, the most important entity is the process, the program in execution. The process fetches instructions and manipulates data, thereby embodying both control and data. On a shared-memory machine, the process is still the most important entity, although there are now multiple threads of execution within the data space. But at the global level, on a distributed-memory machine, messages convey data across the machine and the arrival of messages enables computations. At the global level of distributed-memory machines, it is the messages that embody control and data. Hence, the messages are more important than the processes that are running on the nodes.

Control–Memory Taxonomy
Flynn’s taxonomy is usually applied to machines with von Neumann processors. Further insight may be gained by considering other control mechanisms. The von Neumann machines may be characterized as “control driven”; it is the flow of control represented by the program counter that schedules operations for execution.

“Data-driven” processors schedule operations for execution when their operands become available. In the paradigmatic variety of data-driven machines, scalar data values flow in
tokens over an interconnection network to the instructions that work upon them (hence the term “data flow”). When a token arrives, the hardware checks that all operands are present and, if they are, schedules the instruction for execution. Data-flow machines are easily built distributed memory.

It is also possible to store the data in shared memory and signal each instruction whose operand is now available. Similarly, there is a technique for handling large data structures. An entire data structure cannot be passed in a token, so the structure is stored in a memory where components of the structure arrive as they are computed. Fetches of the elements arrive in tokens and wait for the value of the element to arrive, whereupon the value is sent on in a token to where the fetch specifies.

A “demand-driven” processor performs computations when values are demanded. For example, when the value of a binary operator is demanded, the operator, in turn, demands the values of its operands. A common implementation of demand-driven processors is based on “reductions,” which occur when a functional program is repeatedly rewritten until the solution is computed. The rewritings include replacing an operator applied to data values with its result and replacing a function call with the function body, with the actual parameters substituted for the formal. Reductions are performed on an internal representation of the program. Two common representations are graphs and strings. Graphs consist of nodes linked together with pointers and hence work best with shared memory. Strings can be spread across a chain of processors so that an individual processor can reduce sub-expressions contained entirely in its memory and neighboring processors can shift expressions falling across their boundary into one of them. “Pattern-driven” computation is typically done without specialized hardware and is implemented atop von Neumann machines. Shared-memory, pattern-driven programming usually means parallel-logic programming. Distributed-memory, pattern-driven programming is represented in this book by Active Messages and Concurrent Aggregates.

**Speedup and Efficiency**

We want to use parallelism to compute answers more quickly. How much more quickly? We define speedup as

\[ S = \frac{T_1}{T_n} \]

where \( T_1 \) is defined as the execution time of the best sequential algorithm for the problem on a single processor, and \( T_n \) is the execution time of the parallel algorithm on \( n \) processors. Notice several things:
High-Performance Java Platform Computing

1. \( T_n \) should be smaller than \( T_1 \), since the parallel algorithm should run faster than the sequential algorithm.

2. The larger the value of \( S \), the better. This is coherent with a cultural metaphor of bigger is better\(^1\) (even though we want the smallest run time possible).

3. \( T_1 \) is supposed to be the run time of the best possible sequential algorithm, but in general, the best possible algorithm is an unknown quantity. Thus, it is often the case that \( T_1 \) is simply a version of the parallel program that is run sequentially.

4. We define linear speedup as

\[
S = \frac{T_1}{T_n} = n
\]

We would expect that speedup cannot be better (larger) than linear, and indeed should be smaller. If the entire work of the sequential program could be evenly divided among the \( n \) processors, they could all complete in \( 1/n \) the time. But it is unlikely that the work could be divided evenly; programs tend to have a sequential part, such as initialization or reading data from or writing results to sequential files. If the sequential part can only be done on a single machine, then only the rest can be run in parallel. We will examine this in more detail when we discuss Amdahl’s law. Even if the program could be evenly divided among \( n \) processors, the processors would probably have to coordinate their work with each other, which would require extra instruction executions beyond the sequential program. Therefore, \( T_n \) may be \( 1/n \) of a larger value than \( T_1 \).

Moreover, \( T_1 \) is supposed to be the best known sequential algorithm.\(^2\) If the parallel algorithm runs faster on a single machine, it would be a better sequential algorithm, and therefore, you’d use it. So you can expect the algorithm \( T_1 \) to be at least as good as the algorithm for \( T_n \). You cannot expect any help from differences in the algorithms in achieving even linear speedup.

However, in practice, super-linear speedup (\( S > n \)) is sometimes observed. There are several reasons for this:

- The hardware is different. The parallel machine has more processors, and hence more cache memory, for one thing. Better locality and pipelining can also play a role.

\(^1\) In our lab, we also are known to say that “working software is even better.”

\(^2\) Another way to think of this is to rewrite the program such that it has nothing concurrent about it. It is unlikely that the time of the concurrent program would beat the nonconcurrent program time, simply because of the overheads involved to run a concurrent program.
The algorithm is different. For example, a depth-first search on a sequential machine might be translated into a collection of depth-first searches on the nodes of a parallel computer, but the parallel depth-first searches would have an element of a breadth-first search. A single depth-first search might spend a large amount of time on one fruitless branch, whereas with several searches, it is likely that another path might find the solution more quickly.

Efficiency is defined as

$$ E = S = \frac{T_1}{nT_n} = \frac{T_1/n}{T_n} $$

The formula shows two ways to think about efficiency. Suppose you were to run the parallel program on a serial machine. The serial machine would have to execute all the parallel processes. If there are \( n \) processes, then the serial execution shouldn’t take more than about \( nT_n \) (assuming that the time to swap the processor from one process to another is negligible). Efficiency, then, would measure the ratio of the actual sequential time to the worst expected time to execute the \( n \) processes sequentially.

Suppose that, on the other hand, you calculate how long you would expect it to take to run the sequential algorithm on \( n \) processors, assuming linear speedup. That gives you \( T_1/n \). The efficiency would be the ratio of execution time with linear speedup to observed execution time. If speedup is no greater than linear, efficiency will be less than or equal to 1.

**Amdahl’s Law**

Amdahl’s law\(^3\) does not really deserve the title of law. It is merely a back-of-the-envelope attempt (or conjecture) to prove that there are severe limits to the speedup that can be achieved by a parallel program. Amdahl’s law asserts that there is a serial part of any parallel program that must be executed sequentially, and the time required for this part will be a lower limit on the time the program takes to execute. Consider a serial program that executes in time \( T \). Let’s calculate the best speedup we could achieve if a fraction \( f \) of the exe-

---

\(^3\)Amdahl is the name of computer scientist–engineer, Gene Amdahl, who is one of the pioneers in the field of parallel-computing architecture. The Amdahl corporation is named after him.
High-Performance Java Platform Computing

cution time is taken up by sequential execution. If you divide the parallel execution time into the serial and parallel parts, you get speedup with an upper bound of

\[ S = \frac{T}{fT + \frac{(1-f)T}{n}} \]

We get this equation by taking the definition of speedup and breaking down \( T_n \) into the time taken by the serial fraction \((fT)\) and the time taken by the parallel fraction \([(1-f)T]\). We divide the parallel fraction by \(n\) to calculate the best we could expect from a linear speedup.

\( T \) appears as a factor in both the numerator and the denominator. Thus, it can be removed, which leads to an equation not involving \( T \), or

\[ S = \frac{1}{f + \frac{(1-f)}{n}} \]

As \( n \) approaches infinity (i.e., the number of processors is increased), we arrive at the following limit:

\[ \lim_{n \to \infty} S = \lim_{n \to \infty} \frac{1}{f + \frac{(1-f)}{n}} = \frac{1}{f} \]

So, no matter how many processors we use, we would not expect to gain any more speedup than the reciprocal of the serial fraction. If five percent of the program must run sequentially, speedup will never be better than 20.

**Scalability**

A flaw in the reasoning behind Amdahl’s law is that it deals with fixed-sized problems and questions how much faster they can be run. This is not, however, the way massively parallel processors are used. Take the example of weather forecasting. The calculations are made by superimposing a mesh onto the atmosphere and calculating pressure, temperature, humidity, etc., at each mesh point, repeatedly using the values at the surrounding points at small time intervals. The more numerous the mesh points and the smaller the time intervals, the better is the forecast. But the more calculations that are required, the
slower the program runs. And for weather forecasting, if the calculation takes too long, it loses all value. When presented with a faster machine, weather forecasters will use more grid points and a smaller step size. They increase the problem size to the largest possible value that allows the answer to be reached in the same amount of time.

Let’s rephrase the calculation, starting with a parallel program with serial fraction $g$ that runs in time $R$ on $n$ processors. If we ran the calculation on a single processor, how long would it take? The answer is

$$T = gR + n(1 - g)R$$

This equation follows, since the serial fraction will still take the same time ($g \times R$) and the $n$ parts of the parallel fraction $[(1 - g) \times R]$ would have to be interleaved. This results in the speedup calculation

$$S = \frac{gR + n(1 - g)R}{R} = g + n(1 - g)$$

a linear speedup with slope ($1 \times g$). The efficiency is

$$E = 1 - g \frac{n - 1}{n}$$

which approaches the parallel fraction as the number of processors increases. In this formulation, there is no theoretical limit on speedup. As long as we scale the problem size to the size of the machine, we will not run into limits.

Another aspect of this argument against Amdahl’s law is that, as the problem size increases, the serial fraction may decrease. Consider a program that reads in two $N$-by-$N$ matrices, multiplies them, and writes out the result. The serial I/O time grows as $N^2$, while the multiplication, which is highly parallelizable, grows as $N^3$.  

Problems of Parallelism

Grain Size

Grain size loosely refers to the amount of computation that is done between communications or synchronizations. Too large a grain size can result in an unbalanced load. Too small a grain size can waste too much time on system overhead. Consider eight processors that are to execute 10 independent tasks, each of which takes $t$ time units. Suppose the system takes $s$ time units to run a task. The schedule looks like this:

- Six processors execute one task completing in time $t + s$.
- Two processors execute two tasks completing in time $2t + 2s$.
- The overall completion time is the maximum of any processor’s completion time: $2t + 2s$.

Suppose we divide each task into 10 independent tasks, giving us 100 tasks for the entire job. Each task now will take $t/10$ time units. The schedule now looks like this:

- Four processors execute 12 tasks completing at time $12t/10 + 12s$.
- Four processors execute 13 tasks completing at time $13t/10 + 13s$.
- The overall completion time is the maximum of any processor’s completion time: $13t/10 + 13s$.

How do these compare? If $s$ is negligible compared to $t$, then schedule (1) will complete in $2t$, and schedule (2) in $1.3t$. However, $13s$ is significantly larger than $2s$, so system overhead $s$, being even a small fraction of grain size $t$, might destroy all of the advantages of load balancing. What is the cutover point? That is, at what fraction of $t$ does $s$ cause schedule (2) to take as long as schedule (1)? The answer is

$$2t + 2s = 1.3t + 13s$$
$$s = (0.7/11)t = 0.064t$$

So, if $s$ is even seven percent of $t$, the version with 100 tasks will be as slow as the version with 10. So how do you choose a good grain size? Folklore suggests that one millisecond of execution between communications is a reasonable amount. Other folklore suggests that processing 300–400 array elements between communications is good on some systems. What you will probably have to do is experiment for yourself to find a good grain size. By parameterizing your actual code, you can enable the possibility to experiment.
Starvation

Starvation results when some user computations do not get adequate processor time. Here’s an example of starvation on a distributed-memory machine: For some distributed computations, it is difficult to determine if they are finished. There are some algorithms that send system probe messages around to inquire about the state of the computation. Starvation can result if the probe messages use up significant processor time, making processor time unavailable to the user computations. On shared-memory machines, processors lock and unlock resources. When a resource is unlocked, one of the processors waiting for it (if any) is allowed to proceed. If the resource allocation mechanism is unfair, some waiting processes may be long delayed, while other processes acquire the resource repeatedly.

Deadlock

A set of processes is deadlocked if each process is waiting for resources that other processes in the set hold and none will release until the processes have been granted the other resources that they are waiting for. There are four conditions required for deadlock:

- Mutual Exclusion: Only a process in possession of a resource may proceed.
- Hold and Wait: Processes will hold resources and wait for others.
- No Preemption: A resource may not be removed from one process to give to another.
- Circular Wait: There exists a cycle of processes holding resources and waiting for resources the next process in the cycle holds.

There are three things you can try to do about deadlock:

- You can try to detect when deadlock has occurred and then try to do something about it. For example, you may cancel one or more of the processes involved to free the resources they hold. Usually, this requires the presence of a monitor process that effectively acts as a proxy for any resource request.
- You can try to avoid creating a deadlock by checking before each resource allocation to determine whether the allocation might result in a deadlock and then allowing processes to proceed only if it is safe to do so.
- You can try to make it impossible for deadlock to occur. The easiest prevention is to eliminate circular waits by numbering the resources and requesting resources in ascending numeric order. That is, never request a resource if you already possess one with a higher number.

Flooding and Throttling

Strangely, one of the problems with parallelism is having too much rather than too little. For many parallel algorithms (especially divide and conquer and combinatoric search), a problem is repeatedly broken into smaller parts that can be run in parallel. Once the number of parallel parts significantly exceeds the number of processors available, it is sometimes detrimental to create more parallelism: All processors will be kept busy anyway, the
time to create more parallel tasks will be wasted, and the storage for those task descriptions will tax the parallel machine’s memory.

Preventing a flood of parallelism typically requires extra programming: The algorithm must be broken down into the code that is executed before enough parallel tasks are created, which creates more tasks, and the code that is executed after sufficient tasks are available, which does its work within a single task.

Choice of when to switch from creating more tasks to executing within tasks can be made statically, before the algorithm runs, or dynamically, in response to the system load. Dynamic switching requires additional information about the current state of the system, which is oftentimes not available or is highly imprecise.

**Layout**

The layout of a data structure on a distributed-memory machine can make a significant difference in performance. There are two interacting concerns. First, it is important to balance the load so that all nodes have approximately the same amount of work to do. Secondly, it helps to have most communication between neighboring nodes; there won’t be as many queueing delays as messages contend for communication edges along longer paths.

Consider, though, a simulation of the cosmos: If you divide space into equally sized cubes and assign one cube to each node of a multicomputer, then communication of gravitation and movements of mass can be done between neighboring nodes on a mesh-connected computer. Unfortunately, there will be vast regions of nearly empty space mapped to some regions of nodes, while those parts of space with clusters of galaxies will be mapped into other nodes; that is, the load will be horribly imbalanced. A way to balance the load is to divide space into a larger number of regions and randomize their mapping onto the nodes, say, by hashing their coordinates to give the node number. Then you can count on the law of large numbers to balance the load, but communication between neighboring regions is no longer between neighboring nodes.

Suppose we have $N$ rows of an array that must be processed. How can we divide them evenly among $P$ nodes?

1. We could give floor $\lfloor N/P \rfloor$ rows to each of the first $P-1$ nodes and the remaining rows to the last node. If $N = 15$ and $P = 4$, nodes 0, 1, and 2 get three rows, and node 3 gets six. The load is imbalanced, and the completion time will be dominated by the last node.

2. We could give ceiling $\lceil N/P \rceil$ rows to each of the first $P-1$ nodes and the remaining rows to the last. If $N = 21$ and $P = 5$, we would assign five rows to each of the first four nodes and one row to the last. The last is underutilized, but it’s not as severe as case (1), where the last node was the bottleneck.
3. We could try to assign the rows so that no node has more than one row more than any other node. An easy way to do this is to assign node $i$ all rows $j$ such that $j \mod P = i$, assuming rows are numbered zero through $N-1$ and nodes are numbered zero through $P-1$. Node $i$ will contain rows $i, i + P, i + 2P, i + 3P, \ldots$.

We can assign blocks of rows to nodes, as in (1) and (2), but guarantee that no node has more than one more row than any other node, as in (3). Assign node $i$ the rows in the range $L_i$ to $U_i$ inclusive, where

$$L_i = \text{floor}(N/P) + \min(i, N \mod P)$$
$$U_i = L_i + 1 - 1$$

Some algorithms divide arrays into regions that communicate along their edges, so messages will be shorter and faster if the perimeters of regions are smaller: Square regions tend to be better than long, rectangular regions.

**Latency**

As machines get larger, physical packaging itself requires that components get further apart. Therefore, it will take longer for information to flow between some components rather than others. This implies that the larger, shared-memory machines will be NUMA (nonuniform memory access). Data layout becomes increasingly important. Algorithms may benefit by being rewritten to fetch remote objects, operate on them locally, and then write them back, rather than just manipulating them in place.

Latency is also one of the considerations in laying out tasks and data on distributed-memory machines. On distributed-memory machines, one has the extra option of using asynchronous message passing to allow other computations to be performed while messages are being passed.

**Scheduling**

Scheduling assigns tasks to processors to be run in a particular order or at particular times. There is a large amount of literature devoted to process scheduling, the major import of which is this: Almost any scheduling of activities on processors is an $NP$-hard problem. For practical purposes, the meaning of $NP$-hard is this: The worst-case time to run an $NP$-hard algorithm grows so rapidly (e.g., doubling when you add one element to the input), that you may not get a solution for a modestly sized problem before the sun burns out. Do not seek perfect solutions to $NP$-hard problems. Instead, look for ways to quickly get solutions that are reasonably good most of the time. Static scheduling of tasks on processors would be done before the tasks are run. Dynamic scheduling assigns tasks during execution. Self-scheduling is a form of dynamic scheduling in which the processors themselves select which task to execute next.
The techniques for partitioning rows among nodes that we saw in the discussion of layout are also applicable to processor scheduling on shared-memory machines. For technique (3), an easy way to assign process $i$ all rows $j$ such that $j \mod P = i$ is to handle the rows in a loop:

```
for (j = my_id; j<n; j += P) {
    process row j
}
```

where the rows are numbered 0 through $N-1$

$my_id$ is the node number in the range $0..P-1$

Rather than assign entire rows or columns to processors, better load balancing can sometimes be accomplished by assigning groups of elements. If there are $K$ total elements in an array, we can assign them numbers 0 through $K-1$, assign ranges of those numbers to processors, and convert from the element number to the array indices when necessary. For an $M \times N$ matrix $A$ with zero origin addressing, element $A[i,j]$ would be given the number $i*N+j$ in row major order. Similarly, the element with number $q$ would correspond to $A[i,j]$, where

```
i = \text{floor}(q/N)
j = q \mod N
```

A simple form of self-scheduling for $K$ elements is to keep the index $C$ of the next element to be processed and to allocate items by incrementing or decrementing $C$. The following code is illustrative:

```
initially, $C = K-1$
i = 0;
while ($i>=0$) {
    lock C_lock;
i = C;
    C = C-1;
    unlock C_lock;
    if ($i >= 0$) process item $i$;
}
```

However, if the processing of a single element takes little time, the grain size is too small. Of course the processor could allocate some constant number of elements greater than one at a time. This is clearly better for grain size, but may still have load balance problems. An improved self-scheduling algorithm has each of the $P$ processors allocate ceiling $(C/P)$ elements (i.e., allocate $1/P$ of the remaining elements):

---

4This technique is shown in the examples throughout the book and in Chapter 12.
Foundations

1. Programming Techniques

There are a number of techniques for programming MIMD parallel processors. There are two general techniques based on SIMD and MIMD computers. Data-parallel computation is based on SIMD. This involves a common data structure which is divided up among the processors so that the processors perform the same operations at the same time, but on different data elements. Control-parallel computation is based on MIMD. Here, the processors execute different code at the same time. There are a number of ways to divide up the computations. In the job-jar (or processor farm) organization, the program keeps a list of jobs (or tasks, or chores, or whatever you wish to call them); we call this list the job jar. When a processor becomes free, it pulls a job out of the job jar and executes it. The job may place more jobs in the job jar. In the pipelined or stream organization, a stream of data passes down a chain of processes, each of which reads, transforms, and writes the stream.

A reactive object organization is based on message passing. All computation is performed by objects responding to messages. Similar to the reactive object system is the client–server organization: Client processes send requests to servers, which perform the services requested and send responses back.

Macro dataflow is based on the data-driven computation model. The presence of its operands triggers a computation that uses them. The “macro” refers to the operands being larger than individual scalars (e.g., sections of arrays).

Chapter Wrap-up

The intention of this chapter has been to provide a “crash-course” style introduction to parallel computing. The word “Java” has not been mentioned in this chapter, since this discussion is intended to be brief and to provide an essential overview. The remainder of this book is dedicated to putting the ideas into practice. This chapter gives you the information

Initially, \( C = K \)
\( \text{low} = 0; \)
while (\( \text{low} \geq 0 \)) {
  lock \( C \_\text{lock}; \)
  \( t = \text{ceiling}(C/P); \)
  if (\( t = 0 \))
    \( \text{low} = -1; \)
  else {
    \( \text{high} = C-1; \)
    \( \text{low} = C = C-t; \)
  }
  unlock \( C \_\text{lock}; \)
  if (\( \text{low} \geq 0 \))
    process items \( \text{low} \) through \( \text{high} \) inclusive;
}
you’ll need over the long haul to give a “report card” to Java as a programming language. Both authors believe that Java has the potential to be a great language for parallel computing. There are many issues that must be overcome in order for Java to be taken seriously as a high-performance computing language. We will not dedicate bandwidth in this book to explaining all of the issues. We have provided some performance data for selected algorithms to illustrate the good, the bad, and the ugly of Java performance. For an informed discussion of the issues, Java must address to become the grande language for high-performance computing, an excellent source of information is the Java Grande Forum, of which one of the authors has served as Secretary General and contributor to the Java Grande Report. More information is available at http://www.javagrande.org on the Internet.

**Exercises**

1. Networks of workstations and rack-mounted CPUs connected with fast Ethernet are the dominant platform for parallel and distributed computing today. Why has this occurred? Into which classification would you place such an architecture, using Flynn’s taxonomy?

2. Why is linear speedup seldom achieved in practice?

3. Another interesting law is known as Moore’s law, which essentially states that processor speed is doubling every year. In practical terms, your algorithm will run twice as fast as it does today—if you are willing to wait a year. Some have used Moore’s law to suggest that parallel computing is dead and no longer has a place in the world. (We are inclined to disagree, since Prentice Hall published this book.) What is the potential flaw in the reasoning behind the law? Show an example of a problem that requires parallel computing to have any hope of being solved in a reasonable amount of time. Why will waiting a year not be acceptable for this problem?

4. Explain the flaw in the counterargument to Amdahl’s law, which essentially showed that speedup and efficiency can be obtained simply by increasing the problem size?

5. Explain how processor pipelining and locality can play a role in the super-linear speedup effect that was discussed.

6. This problem requires you to study the manual pages of your operating system. How long does it take to acquire a lock in your operating system? How many multiplications (floating point or integer) of two scalar values can be done in the same period of time. Can a ratio be obtained? What are the implications on grain size when locking is involved?
7. Latency is a major problem in NUMA systems, as well as in ordinary networking. How long does it take for a zero-byte TCP/IP message to be transmitted from one computer to another on your local area network (Ethernet)? Using the same methodology as in the previous question, what are the implications on grain size when network latency is involved?

8. Flynn’s taxonomy presents the entire gamut of known (and even unknown) computer architectures for high-performance computing. What architectures are in use today? Which ones are not in use? Are there any architectures that have been out of use that are now coming back into use? Is there anything that Flynn’s architecture has left out? Which architecture makes the most economical sense?

9. In the 1990s and the rolling 0s (2000s), the field of parallel computing could more aptly be described as clustered computing. What is a cluster? What kinds of machines make up a cluster? What kind of network is needed to support a cluster adequately? Is “cluster” just a euphemism for commodity parallel machine?

10. You don’t need to read this book to determine that Java programs run slower than hand-coded FORTRAN and C (or C++). Considering the discussion in this chapter about $T_1$, the serial performance, pick one of your favorite codes (say, matrix multiplication of integers/floats). Write the program in C and again in Java, making every effort to make best use of the language. How does the performance compare? This would appear to be a major strike against Java. How would you justify the lower performance to someone antagonistic to the Java cause? Will waiting a year (assuming processor speed doubles) make the Java version faster than the FORTRAN- or C-equivalent’s performance (using today’s performance numbers)?