O foco aqui é compreender as oportunidades e os desafios da computação paralela.

Chapter 3: Understanding Parallelism

Introduction

The advantages of parallelism have been understood since Babbage's attempts to build a mechanical computer. Almost from the beginning of electronic computation parallel hardware has been used in the implementation of sequential computers. Efforts to build true parallel computers began in the 1970's and have continued at an accelerating pace, driven by advances in silicon technology. Industrial and academic researchers have studied every imaginable aspect of parallel computation. There is much to learn, and it cannot all be presented in complete detail in a single chapter. So, we begin with an informal tour of almost the entire parallel landscape, knowing that many sights will demand further attention in later chapters. For now, it suffices to gain an appreciation of the opportunities and challenges of parallel computation.

We look at parallelism from different perspectives. The first is performance, since improving performance is the point of parallel computation. The second perspective concerns the structural features of an algorithm that contribute to or hinder performance. Finally, we discuss general parallel problem solving approaches.

Três aspectos abordados de forma superficial.

Opportunities For Performance Improvement

As the add-a-vector-of-numbers example of Chapter 1 indicates, programs can embody different amounts of parallelism despite requiring the same amount of work (in that case the same number of additions). The naïve summation loop produced a sequential specification, which if executed as specified, requires $O(n)$ time because no provision was made for other processes to contribute to the solution. The tree summation was described in a way that allows sub-computations to be performed simultaneously, which with sufficient processing capacity, would lead to an $O(log_2\ n)$ time execution. Is this the best solution available? What limitations might prevent the best performance? Are there opportunities that are not being exploited? We discuss such issues in this section.

exemplo discutido na sala com o Prof. João Costa.

Inherently Sequential. There are computations that are inherently sequential, meaning that all algorithms to solve them have limited parallelism. One such computation is the *circuit value problem*, which takes a circuit specification over logical operators OR, AND and NOT taking *m* inputs, and an *m*-length binary sequence, and evaluates the circuit on the input sequence.

Parallelism vs. Performance

Ideally, a problem that takes T time to execute on a single processor can be solved in T/P time if we can formulate a solution to the problem that exhibits P-fold parallelism. Thus, it is tempting to think that our goal is simply to maximize parallelism, but this is not true.

Consider again the summation of Chapter 1 chapter. For n values, we maximize parallelism by using $P=n/2$ processors, which allows us in each step to perform all pair-wise additions simultaneously. The total algorithm takes $O(log_2\ n)$ time using P processors.

Now consider a variant of the algorithm, which we call the *Schwartz algorithm*. It makes each processor responsible for $log_2\ n$ data items instead of 2 items. (In Figure 3.1, the leaves, which represent data stored on the parent processor, are a total of $log_2\ n$ items.) The idea is that because the height of the summation tree is $log_2\ n$, the tree height defines the computation time; by beginning with each processor finding the sum of $log_2\ n$ local elements, the execution time is only doubled over the naïve solution. That is, in essentially the same time a significantly larger problem can be solved.

Because we are looking at this idea somewhat "backwards," let's put it into numerical terms. Adding a 1000 items using the original tree-based summation takes 10 steps (log_2 1000) using 500 threads of concurrency. If each leaf, rather than being a singleton, were a sequence of 10 items, then a 10,000 item summation could be performed by the same number of threads in 28 steps (9 for each local sum, and 10 to combine them). Using the original summation solution would have required 5,000 threads of concurrency and completed the task in 14 steps. Often, the amount of available parallelism is very small compared to the amount of data, making the idea very attractive.

Schwartz's algorithm shows that trying to maximize parallelism is not always smart. In our original algorithm to process $nlog_2\ n$ data, we would use $P= (n\ log_2\ n)/2$ processors, and we would get a running time of $O(log_2\ (nlog_2\ n)) = O(log_2\ n + loglog_2\ n)$ time. In essence, we use a larger tree having greater depth with the original algorithm. Schwartz's algorithm is not only a simple way to see that maximizing parallelism is not always smart, but it is an excellent solution technique. We will apply it often in Chapter 4.

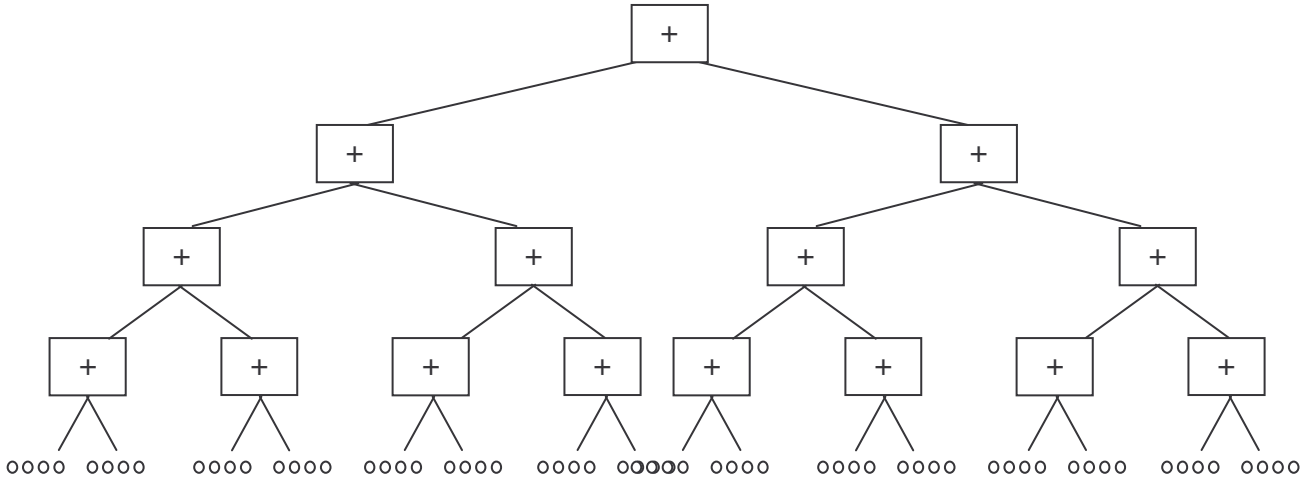


Figure 3.1. Schwartz's approach to the summation computation. Processing nodes are indicated by boxes; the leaves each represent O($log_2\ n$) items.

Our discussion of Schwartz's algorithm makes two points. First, parallelism alone is not the goal. Instead, we need to consider the resources used to exploit this parallelism. Second, when performance is the goal, we need to understand what performance means. The next two sections describe these two topics in turn.

Threads and Processes

To help us reason about the resources needed to exploit parallelism, we will use two common abstractions for encapsulating resources—threads and processes.

A *thread* refers to a thread of control, logically consisting of program code, a program counter, a call stack, and some modest amount of thread-specific data including a set of general purpose registers. Threads share access to the memory, so threads can communicate with other threads by reading from or writing to memory that is visible to them all. (Threads also share access to the file system.) Programming with threads is known as *thread-based parallel programming* or *shared memory parallel programming*.

A *process* is a thread of control that has its own private address space. When multiple processes execute concurrently, they require some mechanism for communicating with each other, since they do not share memory. One cumbersome mechanism might be to communicate through the file system, but a more direct approach is to send messages from one process to another. Parallel programming with processes is often referred to as *message passing parallel programming* or *non-shared memory parallel programming*. A key issue in message passing parallel programming is problem decomposition, since portions of the computation's data structures must be allocated to the separate process memories, that is, they usually cannot be wholly replicated within each process.

In addition to the obvious difference between threads and processes—the distinction between shared and separate memory spaces—there are also distinctions of "weight" and "agility." Threads are usually seen as "lighter weight," being created and completing dynamically throughout a computation. Processes, by contrast, are "heavier weight," taking more time to setup and tear down. Though created dynamically, usually in response to input conditions, they often persist throughout most or all of a computation. Processes can "come and go," but with the (memory) setup time being much greater, they tend to be longer lived.

Latency and Bandwidth

Since performance is the goal, it is important to agree upon what performance means. We often speak of speeding up a computation, but realize that there are two possible goals: latency and bandwidth.

Latency. Latency refers to the amount of time it takes to complete a given piece of work.

Bandwidth. Bandwidth instead refers to the amount of work that can be completed per unit time.

Thus, latency is measured in terms of time or some derivative of time, such as clock cycles. Bandwidth is measured in terms of work per unit time. The distinction between latency and bandwidth is important because they represent different issues with different solutions. For example, consider a web server that returns web pages. The web server's bandwidth can be increased by using multiple processors that allow multiple requests to be served simultaneously, but such parallelism does not reduce the latency of any

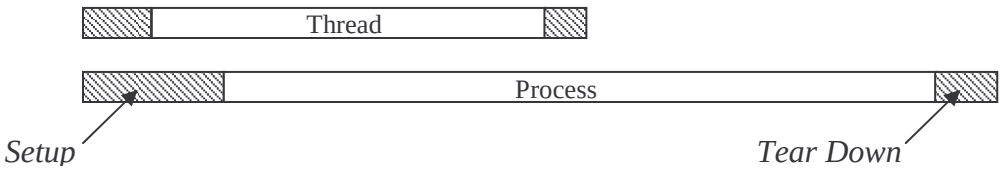
individual request. Alternatively, a web server could employ multiple physically distributed caches that can both decrease the latency of individual requests—for clients that are close to one of the caches—and increase the server's overall bandwidth. In many cases, latency can be reduced at the cost of increased bandwidth. For example, to hide long latencies to memory, modern microprocessors often perform data prefetching to speculatively bring data to caches, where its latency to processors is lower. However, because prefetching invariably brings in some data that is not used, it increases the demand for memory system bandwidth. This idea of trading bandwidth for latency is not new: The Multics operating system used the idea in the 1960s when it introduced the notion of context switching to hide the latency of expensive disk I/O.

The use of latency and bandwidth is common in some, but not all, parallel computation subcommunities, so our use of it throughout this book somewhat broadens its application. We will use latency to refer to the length of execution time or the duration of the computation, and bandwidth to refer to the capacity of a processor, its instruction execution rate. We have slightly expanded the scope of latency and bandwidth to unify terminology. There should be little confusion when encountering alternate terms in the literature.

Sources of Reduced Performance

While we ideally would hope that *P* processors could speed up a computation by a factor of *P*, there are many reasons why this might not be the case. We explore these factors in this section.

Overhead. Any cost that is incurred in the parallel solution but not in the serial solution is considered *overhead*. There is overhead in setting up threads and processes to execute concurrently and also some for tearing them down, as the following schematic indicates.



Because memory allocation and its initialization are expensive, processes incur greater setup overhead than threads. After the first process is set up, all subsequent thread and process setups incur overhead not present in a sequential computation. These costs must be charged against the benefits of parallelism; see the section, Measuring Performance below.

Communication. Communication among threads and processes is a major component of overhead. Since a sequential computation doesn't have to (cannot!) communicate, all communication is a charge against the benefits of parallelism. These costs have been described in detail in Chapter 2, and though they are different depending on the communication mechanism chosen—shared memory, 1-sided or message passing—they are all substantial compared to a local memory reference. To be clear, there is always a

communication charge unless the data is local; the components of the charge are given in Table 3.1.

Synchronization. Synchronization is a form of overhead that arises when one thread or process must wait for another. Synchronization is implicit in many forms of message passing, while synchronization is often explicit when programming with threads.

Table 3.1: Sources of communication overhead by communication mechanism.

Mechanism	Components of Communication Cost
Shared Memory	Transmission delay, coherency operations, reference protection, unavailability
1-sided	Transmission delay, reference protection, unavailability
Message Passing	Transmission delay, data marshalling, message formation, demarshalling, unavailability

***Contention*.** Contention is the degradation of system throughput caused by competition for a shared resource. For example, we saw in Chapter 1 how lock contention can reduce network throughput by creating excessive network traffic, and we saw how false sharing can degrade performance by causing data values to bounce back and forth among different caches.

Idle Time. When we conceptualize a parallel computation, we imagine that the processors are all working all of the time, but they might not be. The main reason is that a process or thread cannot proceed because there is no work to do or because the needed data is not yet available. As the next section on Dependences demonstrates, idle time manifests itself in many ways.

Load Imbalance. One common source of idle time is an uneven distribution of work to processors, which is known as load imbalance. For example, the Schwartz algorithm has an advantage over the standard prefix summation because the former keeps all processors busy with useful work much of the time, thereby allowing larger (by a factor of $log_2 n$) problems to be solved with the same number of processors.

Balancing load is straightforward for easy tasks like summation, but most computations are much more complex. We sometimes display the allocation of array computation, especially for the process model, by showing the array and its decomposition among processors; Figure 3.2 shows a schematic example for the LU Decomposition computation, a widely used algorithm for solving systems of linear equations. As shown in Figure 3.2(a) the LU computation builds a lower (black) and upper (white) triangle beginning at left; the area of the computation is shown in gray, and after every iteration of the computation one row and one column are added to the completed portion of the array. Figure 3.2(b) shows sixteen processors logically arranged as a grid, and (c) shows how the array might be allocated to processor memories in a process model of the computation. Though the allocation of data is balanced, i.e. each processor is assigned roughly the same number of array elements, the work is not balanced. For example, after

the first 25% of the rows and columns have been added to the result arrays, there is no more work to do for the seven processors on the left and top sides of the array. That is, nearly half of the processors will be idle after one quarter of the rows/columns have been processed. Though it is true that the amount of work per iteration diminishes as the active (gray) portion of the array shrinks, this allocation of work is still quite unbalanced. Indeed, the last 25% of the rows/columns are computed by processor P_F. Or putting it another way, the last 25% of the rows/columns are computed sequentially.

Redundant Computation. *P* processors will not speed up a sequential computation by a factor of *P* if the parallel version of the computation requires more instructions. But extra instructions are almost always required. For example, if the sequential computation requires the program to loop k times, and if the parallel computation also requires each process to loop k times, then the loop overhead instructions—initialization, incrementing, testing for termination—are not sped up by parallelism. As another example, recall the example of generating a random number from Chapter 2; although it was smart to repeat the computation to avoid non-local communication, having each process generate its own random number means there will be no parallel improvement of that portion of the computation. Of course, the programmer's goal is to make most of the computation non-redundant.

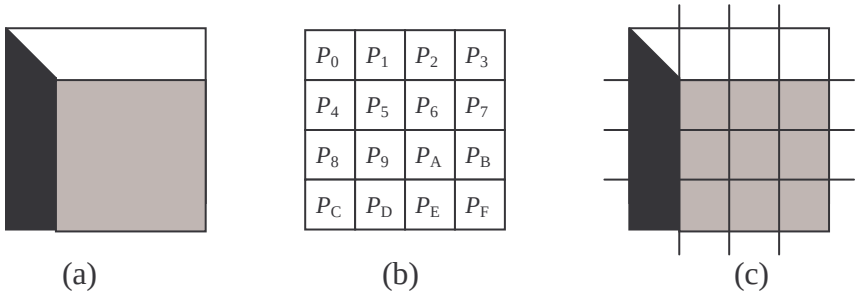


Figure 3.2: Schematic diagram of (a) the LU Decomposition algorithm, (b) sixteen processors (indexed in hexadecimal) arranged in a logical grid, and (c) the allocation of the array elements to the processors, e.g. processor P_0 is assigned that part of the array in the upper left that has completed.

Parallel Structure

By the end of the chapter we will conclude that the ideal parallel computation is one that has large blocks of independent computation that can be executed concurrently. With separate parts of the problem being performed on different processors, there will be little idle time and the solution will be found fast. To prepare to embrace "blocks of independent" computation, we must understand what "dependent" computation is. That is, our ideal case will be formed from normal computation in which we avoid certain performance limiting characteristics of programming. In this section we discuss such features in terms of the concept of dependences.

Dependences

A *dependence* is an ordering relationship between two computations. Dependences can arise in different ways in different contexts. For example, a dependence can occur

between two processes when one process waits for a message to arrive from another process. Dependences can also be defined in terms of read and write operations. Consider a program that requires that a particular memory location be read after an update (write) to the same memory location; as an example, recall the `count` variable in Figure 1.7. In this case, there is a dependence between the write operation and the read operation. If the order of the two operations is swapped, the value read would not reflect the update, so the dependence would be violated by the swap and the semantics of the program would be altered. Any execution ordering that obeys all dependences will produce the same result as the originally specified program. Thus, the notion of dependences allows us to distinguish those execution orderings that are necessary for preserving program correctness from those that are not.

Dependences provide a general way to describe limits to parallelism, so they are not only useful for reasoning about correctness, but they also provide a way to reason about potential sources of performance loss. For example, a data dependence that crosses a thread or process boundary creates a need to synchronize or communicate between the two threads or processes. By knowing the data dependence exists we can understand the consequences for parallelism even if we don't know what aspect of the computation caused the ordering relationship in the first place. To make this point more concrete, let us consider a specific type of dependence, known as data dependences.

Data dependence. A data dependence is an ordering on a pair of memory operations that must be preserved to maintain correctness. There are three kinds of data dependences:

- *Flow dependence:* read after write
- *Anti dependence:* write after read
- *Output dependence:* write after write

Flow dependences are also called *true dependences* because they represent fundamental orderings of memory operations. By contrast, anti and output dependences are collectively referred to as *false dependences* because they arise from the re-use of memory rather than from a fundamental ordering of the operations.

To understand the difference between true and false dependences, consider the following program sequence:

```
1. sum = a + 1;
2. first_term = sum * scale1;
3. sum = b + 1;
4. second_term = sum * scale2;
```

There are flow dependences (via `sum`) relating lines 1 and 2, and there are flow dependences relating lines 3 and 4. Further, there is an anti dependence on `sum` between line 2 and line 3. This anti dependence prevents the first pair of statements from executing concurrently with the second pair. But we see that by renaming `sum` in the first pair of statements as `first_sum` and by renaming the `sum` in the second pair of statements as `second_sum`,

```
1. first_sum = a + 1;
2. first_term = first_sum * scale1;
3. second_sum = b + 1;
4. second_term = second_sum * scale2;
```

the pairs can execute concurrently. Thus, at the cost of increasing the memory usage by a word, we have increased the program's concurrency. By contrast, flow dependences cannot be removed by renaming variables. It may appear that the flow dependences can be removed simply by substituting for sum in the second and fourth lines,

```
1. first_term = (a + 1) * scale1;
2. second_term = (b + 1) * scale2;
```

but this doesn't eliminate the dependence because no matter how it is expressed the addition must precede the multiplication for both terms. The flow—the write of the sum (possibly to an internal register) to the read as an operand (possibly from an internal register)—remains.

Dependences Limit Parallelism

To understand how dependences limit parallelism, recall the following code from Chapter 1, which specifies the summation of a set of *n* numbers:

```
sum = 0
for (i=0; i<n; i++) {
    sum += x[i];
}
```

This program, which we described as sequential, is abstracted in Figure 3.3(a); the more parallel tree solution is shown in 3.3(b). In the figure, an edge not involving a leaf represents a flow dependence, because the computation of the lower function will write into memory, and the upper function will read that memory. The key difference between the two algorithms is now evident. In Figure 3.3(a) the sequential solution defines a sequence of flow dependences; they are true dependences whose ordering must be respected. By contrast Figure 3.3(b) specifies shorter chains of flow dependences, imposing fewer ordering constraints and permitting more concurrency. In effect, when we gave the C specification for adding the numbers, we were specifying more than just which numbers to add. (We needed the extra fact of associativity of addition to know that the two solutions produce the same result.)

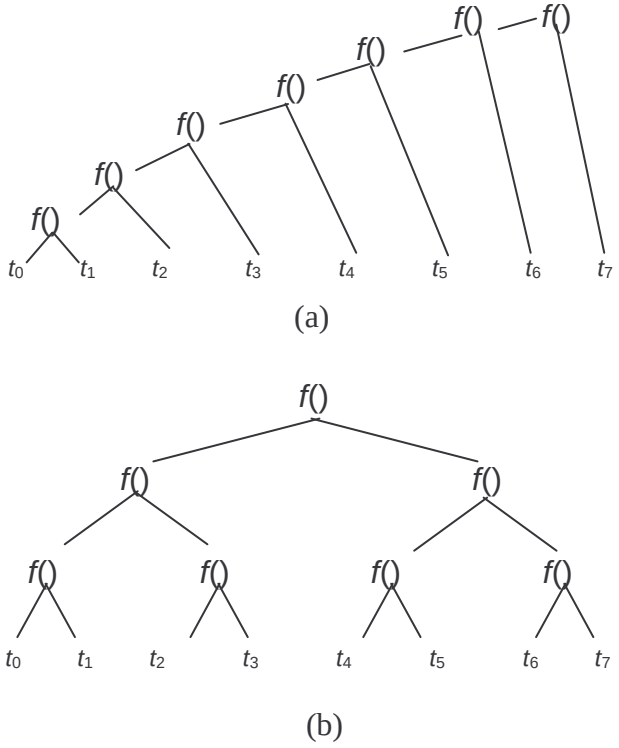


Figure 3.3. Schematic diagram of sequential and tree-based addition algorithms.

The point is that care must be exercised, when programming, to avoid introducing dependences that do not matter to the computation, because such dependences will unnecessarily limit parallelism. (Knowing that *f*() is addition allows powerful compiler techniques to transform this code into a more parallel form, but such technology has a limited scope of application.)

Granularity

A key concept for managing the constraints imposed by dependences is the notion of granularity. We identify and explain two closely related ways in which this term is used:

- Granularity of work
- Granularity of interactions

Notice that grain size is usually described using terms *coarse* and *fine*, though large and small are also used.

Granularity of Interaction. Interaction measures the frequency of dependences crossing the boundaries of threads or processes, where frequency is measured in number of useful instructions separating the interactions. Thus, coarse grain refers to threads and processes that only infrequently depend on data or events in other threads or processes, and conversely, fine grain interactions are those that occur often. As mentioned earlier dependences that cross thread or process boundaries introduce communication with its

associated overhead. Further, frequent interactions imply that waiting time can accumulate as threads and processes stall. For threads sharing through memory the cost for communicating is lower and the amount of work between interactions may be similar, suggesting that fine grain interactions may be worthwhile, especially if used in abundance. Because the overhead of message passing is typically large, processes work best with coarse grain interactions.

Granularity of Work. Work is usually measured by such things as number of instructions executed, or number of data values assigned to a thread or process. Accordingly, a coarse grain computation has a large time and/or memory footprint. Conversely, a fine grain computation has only few values processed locally and contributes mainly by being used in large quantity. Consistent with earlier points, threads often support fine grain parallelism and processes support course grained parallelism. Other semantic nuances include the sense that fine grain computations are more flexible, being available for smaller opportunities for parallelism. By contrast, coarse grain computations can provide better opportunities for amortizing overhead and hiding latency, as we discuss below.

Applying Granularity Concepts. The key point is that no fixed granularity is best for all situations. Instead, it is important to match the granularity of the computation with both the underlying hardware's available resources and the solution's particular needs. For example, the original prefix summation described in Chapter 1 was a fine grain computation involving a small amount of work and fine grain interactions with the adjacent threads. The Schwartz variant of the computation increased the grain size at the start of the computation, performing much more work before communicating. This larger granularity led to better performance. Notice that the fine grain interaction remains in the "accumulation" part of the Schwartz computation. To "coarsen" this part of the computation, the degree of the tree must be increased, where the degree, presently 2, is the number of children of each parent. For other problems a coarse granularity might lead to poor load balance.

In the limit the coarsest computations involve huge amounts of computation and no interaction. SETI@home is such an example. Subproblems are distributed to personal computers and solved entirely locally; the only communication comes at the end to report the results. In this setting the parallel computer can be an Internet-connected collection of PCs. Such super-coarse grain is essential because of the huge cost of communication.

At the other end of the spectrum are threads running on Chip Multiprocessors (CMPs) that provide low latency communication among processors that reside on the same chip, making fine grain threads practical.

Most parallel computation falls between these extremes.

Locality

A concept that is closely related to granularity is that of locality. Computations can exhibit both temporal locality—memory references that are clustered in time—and spatial locality—memory references that are clustered by address. Recall that locality is an

important phenomenon in computing, being the reason why caches work, so improving locality in a program is always a good thing. Of course, the processors of parallel machines also use caches, so all of the benefits of temporal and spatial locality are available. Keeping references local to a thread or process ensures that these benefits will be realized. Indeed, algorithms like the Schwartz approach that operate on blocks of data rather than single items, virtually always exploit spatial locality, and are preferred.

In the parallel context, locality has the added benefit of minimizing dependences among threads or processes, thereby reducing overhead and contention. As outlined above, non-local references imply some form of data communication, which is pure overhead that limits parallel performance. Furthermore, by making non-local references, the threads or processes will often contend with each other somewhere in the execution, either colliding on the shared variable in the case of threads or colliding in the interconnection network in the case of processes. Thus, non-locality has the potential of introducing two kinds of overhead.

A simple example makes both parts clear: Consider a set of threads Counting 3s in a large set of numbers using the scalable algorithm (Try 4 in Chapter 1); by working on a contiguous block of memory, a thread exploits spatial locality; by making the intermediate additions to a local accumulation variable, it benefits from temporal locality. Moreover, by combining with the global variable at the end rather than with each addition, it reduces the number of dependences among threads until the communication is absolutely necessary to achieve the final result. With the reduced number of dependences, locality is improved while overhead and contention are reduced. Note that this use of a local accumulation variable is another example of using a small amount of extra memory to break false dependences.

Forms of Parallelism

Though we have distinguished between thread-based parallelism and process-based parallelism, we have done so to focus on implementation differences, such as granularity and communication overhead. Now we are concerned with understanding where the parallelism can be found at the algorithmic level. We recognize three general types:

- data parallelism
- task parallelism
- pipelining

We now consider each, realizing that there is overlap among the categories.

Data Parallelism

Data parallelism refers to a broad category of parallelism in which the same computation is applied to multiple data items, so the amount of available parallelism is proportional to the input size, leading to tremendous amounts of potential parallelism. For example, the first chapter's "counting the 3s" computation is a data parallel computation: Each element must be tested equal to 3, which is a fully parallel operation. Once the individual outcomes are known, the number of "trues" can be accumulated using the tree summation technique. Notice that the tree add applies to all result elements only for its initial step

and has logarithmically diminishing parallelism thereafter. Still, the parallelism is generally proportional to the input size, so global sum is considered to be a data parallel operation.

As we observed in our discussion of locality and granularity above, the availability of full concurrency does not imply that the best algorithms will use it all. The Schwartz algorithm showed that foregoing concurrency to increase locality and reduce dependences with other threads produces a better result. Indeed, one of the best features of data parallelism is that it gives programmers flexibility in writing scalable parallel programs: The potential parallelism scales with the size of the input, and since, usually, $n >> P$, programs must be designed to process more data per processor than one item. That is, the program should be able to accommodate whatever parallelism is available. (It has been claimed that writing programs as if $n == P$ leads to effective programs because processors can be virtualized, i.e. the physical processors can simulate any number of logical processors, leading to code—it's claimed—that adapts well to any number of processors. This is not our experience. Virtualizing processors leads to extremely fine grain specifications that miss both the benefits of locality and the "economies of scale" of processing a batch of data. We prefer solutions like Schwartz's that explicitly handle batches of data.)

Task Parallelism

The broad classification of task parallelism applies to solutions where parallelism is organized around the functions to be performed rather than around the data. The term "task" in this case is not to be contrasted necessarily to "thread" as we normally do, because the emphasis is on the functional decomposition, which could be implemented with either tasks or threads.

For example, a client-server system employs task parallelism by assigning some tasks the job of making requests and others the job of servicing requests. As another example, the sub-expressions of a functional program can be evaluated in any order, so functional programs naturally exhibit large amounts of task parallelism. Though it is common for task parallel computations to apply an operation to similar data, as data parallel computations do, the task parallel approach becomes desirable when the context in which the data is evaluate matters significantly.

The challenges to task parallelism are to balance the work and to insure that all the work contributes to the result. In many cases, task parallelism does not scale as well as data parallelism.

Pipelining

Pipelined parallelism is a special form of task parallelism where a problem is divided into sub-problems, which can each be operated on independently, and where there are multiple problem instances to be solved. At any point in time, multiple processes can be busy, each working on a sub-problem of a different problem instance. As is familiar with bucket brigades, assembly lines, and pipelined processors, the solution is to run the operations concurrently, but on different problem instances. As the pipeline fills and

drains, there is less than full parallelism, as the opportunities for concurrency increase (fill) and then diminish (drain). A more crucial issue is the balancing of work of each operation. For pipelining to be maximally effective, the operations (stages) must complete in the same amount of time. Pipeline performance is determined—even for pipelines that are not clocked—by the longest running stage. Balancing the stages equals out the work, allowing all stages to process at the maximum prevailing rate.

Though pipelining is frequently thought of as a parallelism approach for cases defined by only a fixed length sequence of operations, it arises more generally. The number of (potential) stages is often determined by the input size. In such cases data dependences entail receiving input value(s) from one or more neighbors, computing, and then yielding the result(s) to opposite neighbor(s). The schematic in Figure 3.4 illustrates the idea. Clearly, in addition to maximizing the use of the processors, such computations are challenging in terms of avoiding stalls caused by fine grain interactions.

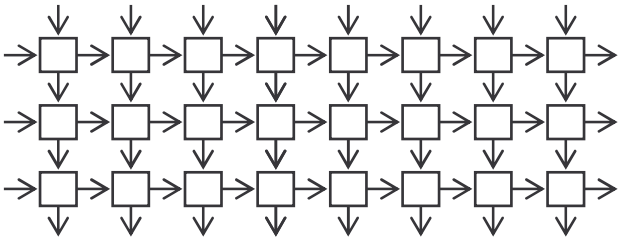


Figure 3.4. Schematic of a 2-dimensional pipelined computation, showing computation (boxes) and data flow (arrows). External data is presumed to be initially present; on the first step only the upper-left computation is enabled.

Summary

In this chapter we have introduced many concepts briefly. The goal has been to become aware of opportunities and challenges to parallel programming. Because the concepts interact in complex ways, it is not possible to understand them completely when treated in isolation. Rather, we have introduced them all in a quick, albeit limited, overview of the issues, and have prepared ourselves for the next chapter where we will develop algorithms and see first hand the consequences of these complexities.

Exercises

1. In transactional memory systems, a thread optimistically assumes that it makes no references to shared data. The transaction either *commits* successfully if there was no shared access detected, or the transaction *rolls back* if there was. Identify the sources of performance loss in a transactional memory system, classifying each as overhead, contention, or idle time.

2. Should contention be considered a special case of overhead? Can there be contention in a single-threaded program? Explain.

3. Should idle time be considered a special case of overhead? Can there be idle time in a single-threaded program? Explain.

4. Does a chess program provide data parallelism or task parallelism?

5. Does quicksort provide data parallelism or task parallelism?

6. Describe a program whose speedup does not increase with increasing problem size.

Bibliographic Notes

Schwartz's algorithm has been discovered later by theoreticians who have given it a different name. [Need to look up this name.] J.T. Schwartz, "Ultracomputers", ACM Transactions on Programming Languages and Systems, 2(4):484-521, 1980

Chapter 4: General Algorithmic Techniques

To become effective programmers, we need to learn a programming language and how to use it to express basic problem solving techniques like building data structures. We must learn how to analyze programs to determine their running time and memory usage. These will be topics for future chapters. For now, perhaps the most important understanding to acquire is the ability to "think about the computation 'right'." That is, we want to think about solving problems in a way that matches well the languages and computers available to us. In this chapter we learn the 'right' way to think about parallel computation.

What Is The Opposite of Sequential?

Many researchers have claimed that the best way to think about parallel computation is to think about the most parallel solution imaginable assuming an unlimited number of processors. They acknowledge that unlimited capacity is not realistic, but—their argument goes—it is possible to "scale back" parts of the computation to be sequential, and arrive at an ideal solution.

So, for example, return to the problem from Chapter 1 in which we want to count the number of 3s in an array `A`. Using the maximum parallelism approach, we expect a solution in which one processor initializes the count value

```
count = 0;
```
Performed by p_0

and then processor *i*, assigned to the i^{th} data element, performs the operations

```
if (A[i] == 3) count = count + 1;
```
Performed by p_i

Such a specification makes sense from an individual data element's point of view, perhaps, but not when viewed more globally, because processors can collide when referencing `count`. Though advocates of the unlimited parallelism approach have addressed the issue of collisions with everything from "It's an error" to "It's OK, thanks to special (Fetch & Add) hardware," we know from our discussion in Chapters 1 and 2 that there are difficulties that can arise with existing parallel computers:

- races can occur caused by the action of other processors changing `count` between the time processor p_i accesses it to get its value and the time p_i updates it
- the possibility of races implies that `count` must be protected by a lock
- the need for a lock implies the potential for lock contention when `A` contains many 3s and many processors attempt to update it simultaneously
- lock contention results in lock access being serialized
- serializing locks implies that for an array of mostly 3's the execution time is $O(n)$ regardless of the number of processors available.

There may be different "unlimited parallelism" solutions, but this is an obvious one; it does not lead to a very parallel result.

The great body of literature on unlimited parallelism comes from a study of parallel models of computation collectively known as PRAMs, acronymic for Parallel Random Access Machines, though many other unlimited parallelism approaches have been invented as well. The problem with these approaches is that finding parallelism is usually *not* the difficult aspect of parallel programming. Rather—and this is our motive for introducing the topic here—parallel programming is generally concerned with *the consequences of parallel threads interacting*, as the bulleted items just illustrated. These are the dependences discussed in Chapter 3. They arise when processors must access shared resources and when processors contend for, and therefore must wait on, shared resources. Thread interaction influences performance as much as the amount of concurrent work embodied in a problem, often more so. To be effective parallel programmers, we need to focus on the right part of the problem, and that is on the interactions between parallel threads.

Blocks of Independent Computation

If dependences between interacting threads are a significant problem, then the ideal parallel computation must be one composed of large blocks of independent computation with no interactions at all. Such computations exist: SETI@home, the Search for Extra Terrestrial Intelligence, is typical; independent computational tasks are downloaded to participants' idle PCs, computed, and the results returned to the server, which compiles the results. Other tasks from Monte Carlo simulations to integer factorization have these same features. They may be ideal, but they are not typical; nearly all parallel computations require that threads interact, and the amount of interaction is correlated with the amount of parallelism.

General parallel computations, though more complicated, still benefit whenever they can exploit the blocks-of-independent-computation strategy. Our Count 3s solution from Chapter 1 used this approach. The final solution (Try 4) partitioned the array among several threads, allocated a local variable `private_count` to each thread to record intermediate progress, and at the end combined the local results to compute the global result. The application of the principle of blocks-of-independent-computation is evident. Further, our initial tries at solving the problem were largely aimed at neutralizing the consequences of thread-to-thread dependences: races were avoided with locks, contention was removed with the private variables, false sharing was avoided with padding, etc. And as the experimental data showed, the program performed. This is one example of many that we will see of an important principle:

Guideline #1. Parallel programs are better designed when they emphasize (large) blocks of independent computation that minimize the interthread dependences (interactions).

Though our final Count 3s result was quite satisfactory, it was not actually *scalable*; that is, capable of executing well for any amount of parallelism. True, the number of threads was a parameter, so if the number of parallel processors P is greater than one, then the solution partitions the array into blocks, and P of these can execute concurrently. It is fully parallel during the scan of the data array. But there is potential for lock contention

during the final step of combining the `private_count` variables. If *P* is not likely to be a large number, then any serialization due to lock contention is not likely to be a serious problem; if *P* could be large, however, lock contention could harm performance. So, to make the solution more scalable, we combine the `private_count` variables pairwise in a tree, using the tree addition algorithm. This solution gets good performance using any number of processors, though when $P > n/\log n$, the final combining tree may be deeper than necessary, implying that using so many processors is not making the computation faster. (We discuss such tuning issues later in Chapter 5.)

Guideline #2. Just as algorithms are written to be independent of the number of input values, *n*, parallel algorithms should also be written to be independent of the number of parallel threads, *P*, and be capable of improved performance using additional processors.

Finally, reviewing this last version of Count 3s computation, notice that it is really just a simple variation of the Schwartz computation. Recall that the Schwartz algorithm was designed to add array elements, but testing and tallying those elements that equal 3 is a trivial variation. The Schwartz algorithm processes a block of elements locally, as our Count 3s program does. And to produce the final result the Schwartz algorithm uses a tree to combine the intermediate results, as our revised Count 3s solution does. Finally, the range of values over which *P* can vary is the same, as are the considerations of using more or fewer processors.

In summary, as we create parallel algorithms, we will attempt to formulate them as blocks of maximally independent computation, where "maximally independent" means that we try to reduce the interactions (dependences) among the threads. This is a challenging task, and we will often find that our best attempts do not attain our performance goals. Fortunately, there are many techniques like Schwartz's approach that give us direction and ideas for solving problems in parallel.

Assigning Work To Processors Statically

The basic way to assign work is to statically assign data to processors, and require each processor to compute on the data it "owns." This technique works for a wide variety of situations, and is the subject of this section. This is the data parallel approach, because we use the data as the basis for organizing the computation.

Basic Block Allocations

Since our goal is to exploit locality, it follows that contiguous portions of a data structure should be allocated together on the same processor. (The exceptions to this thinking are treated below.) Thus, 1-dimensional arrays are assigned to processors in blocks of consecutive indices. For 2-dimensional arrays, allocating by 2-dimensional blocks, that is, consecutive indices in both dimensions, generally leads to efficient solutions. The reason 2-dimensional blocks tend to make more sense than allocating, say, whole rows, is that blocks can often reduce communication. For example, for computations that rely on neighboring values, the so called *stencil computations* such as

```
B[i,j] = (A[i-1,j] + A[i,j+1] + A[i+1,j] + A[i, j-1])/ 4.0;
```

there is a surface-to-volume advantage, as can be seen in Figure 4.1. That is, a squarish block of array values has the property that the elements that must be referenced by other processors for the stencil computation are on the edge (surface), and as the size (volume) of the block increases, the number of edge elements grows much more slowly, reducing communication costs. This small example isn't very dramatic, but the difference for a 32x32 block is 128 versus 2048 values referenced by (communicated to) other processors. For higher *d*-dimensional arrays, allocating as *d*-dimensional blocks is frequently used for the surface to volume advantage, too, but almost as common is to allocate only two of the dimensions and keep the other dimension(s) allocated locally. The latter choice is often the result insufficiently many processors, or extreme aspect ratios.

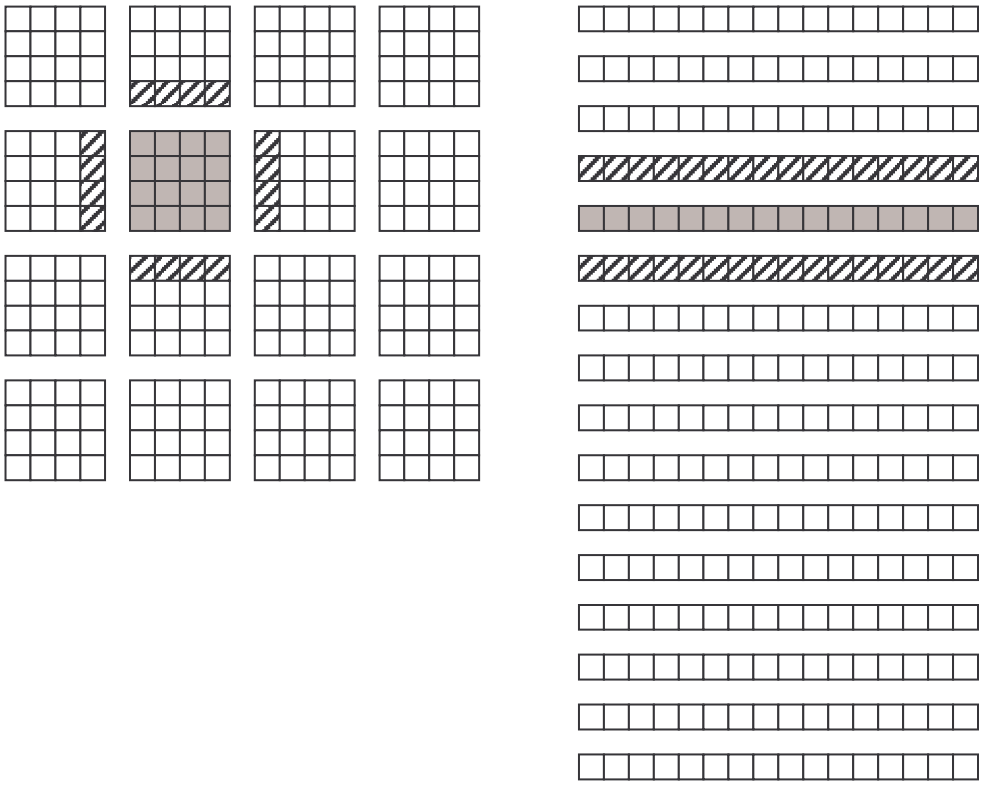


Figure 4.1: Two allocations of a 16x16 array to 16 processors: (a) 2-dimensional blocks and (b) rows. For the processor with shaded values to compute a 4-nearest neighbor computation requires communication with other processors to transmit the hatched values. The row allocation requires twice as many values to be transmitted, and because of the surface to volume advantage, the blocked allocation improves as the number of local items increases.

The Specifics of Block Layouts

Our goal, when allocating the array's blocks, is to balance the data assigned to each processor, because the work tends to be proportional to the number of data items. Occasionally, everything "divides perfectly," and each processor is assigned the identical amount of work; and sometimes partitioning can be simplified by making the array

dimensions multiples of the number of processors. But more often the problem size and therefore the size of the arrays, $r \times c$, is dictated by other considerations. In such cases there are various ways to allocate the arrays in blocks.

Assume $P = uv$, that u does not divide r, and that v does not divide c; that is, the divisions

$$r = a_1 u + e_1 \text{ and } e_1 > 0$$
$$c = a_2 v + e_2 \text{ and } e_2 > 0$$

have nonzero remainders. To discuss allocations, let

$$r = a_1'(u\text{-}1) + e_1' \text{ and } e_1' > 0$$
$$c = a_2'(v\text{-}1) + e_2' \text{ and } e_2' > 0$$

The two most obvious schemes can be called “Direct Division” and “Ceiling-Floor,” see Figure 4.2:

> **Direct Division:** Allocate blocks of $a_1' \times a_2'$ elements to $(u\text{-}1)(v\text{-}1)$ processors, allocate blocks of $e_1' \times a_2'$ to $(v\text{-}1)$ processors, allocate blocks of $a_1' \times e_2'$ to $(u\text{-}1)$ processors, and allocate a block of $e_1' \times e_2'$ to one processor.
> **Ceiling - Floor:** Allocate blocks of $a_1' \times a_2'$ elements to $e_1'e_2'$ processors, allocate blocks of $a_1' \times (a_2' - 1)$ elements to $(v - e_2')$ processors, allocate blocks of $(a_1' - 1) \times a_2'$ to $(u - e_1')$ processors, and allocate blocks $(a_1' - 1) \times (a_2' - 1)$ elements to the remaining $(u - e_1')(v - e_2')$ processors.

The allocations are the same in their most important respect, the size of the largest block, $a_1' \times a_2'$. This means that if a computation is strictly proportional to the number of local data items, both schemes have the same worst case. However, the Direct Division is likely to have $u+v$-1 processors that have significantly less work to do than the others, and so are more likely to be idle, waiting on others to complete. Such imbalance often wastes parallel resources. The Ceiling - Floor allocation has the advantage that the number of elements in each dimension differs by only one, making the quantity of data assigned more balanced compared to the Direct Division approach. The work is distributed somewhat better. Without additional information about the characteristics of the problem, the Ceiling - Floor is slightly better.

Sensitivity to Processor p_0: Independent of which allocation is chosen, it is sensible to assign the minimum allocation of data to processor p_0. This is because p_0 is often given additional tasks, such as managing I/O, serving as the root of a combining tree, etc. (These can generally be thought of as controller functions, relative to the CTA model of Chapter 2.) By allocating it the least work, p_0 is available to perform the additional duties.

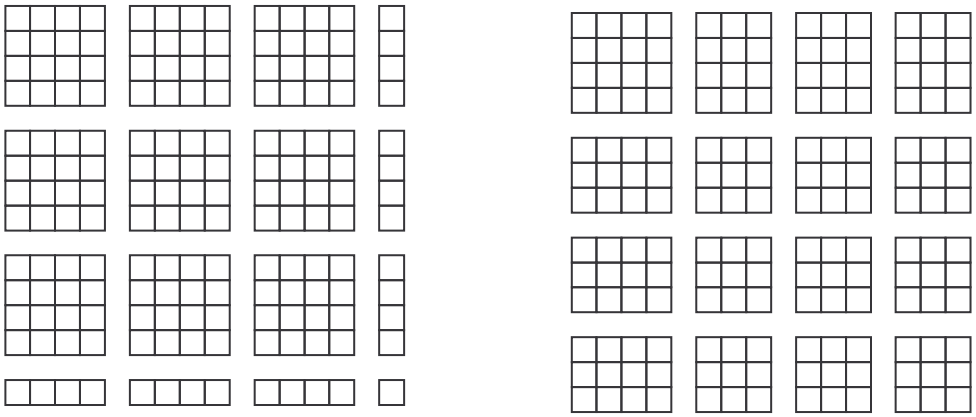


Figure 4.2: Two array-to-processor allocations for a 13×13 array on 16 processors; (a) Direct Division, (b) Ceiling - Floor.

Fluff or Halo Buffers: Computations like the 4-point nearest neighbor stencil

```
B[i,j] = (A[i-1,j] + A[i,j+1] + A[i+1,j] + A[i, j-1])/ 4.0;
```

require values stored on other processors. For the block allocations of the type being discussed Figure 4.1(a) shows that for `i` and `j` in the top row of the allocation, references to `A[i-1,j]` are off processor to the north, and similarly for the other edge elements of the block. Such nearest neighbor references, which are quite common in scientific computations, are best solved with the following approach:

- get the necessary values of the adjacent array blocks from the other processors
- store the values in in-position buffers arranged around the local data block
- perform the computation on the (now) entirely local data values

The buffers are known as *fluff* or *halo buffers*, and are allocated in their proper position relative to the other elements in the array block; see Figure 4.3.

Several advantages recommend this approach. First, once the fluff is filled, all references in the computation are local. This means that the many processor-dependences implied by a loop performing the stencil computation over an array block have been merged into *b* dependences, if the computation references *b* neighboring processors. The result is a large block of dependence-free code to execute. Further, the statement uses the same index calculations for all references to A; that is, they can be performed in a single loop having no special edge cases. Finally, moving non-local data to the local thread at one time offers the opportunity (generally available) to batch the data movement; that is, the whole row or column of an adjacent processor, perhaps stored in one or few cache lines, might be moved at once. This is a significant advantage, as noted in Chapter 2, because typically data transmission takes $t_o + dt_b$ seconds to transmit d bytes, where t_o is overhead and t_b is the time per byte. Batching communication saves the multiple overhead charges from multiple transmissions, and any additional waiting times caused by them.

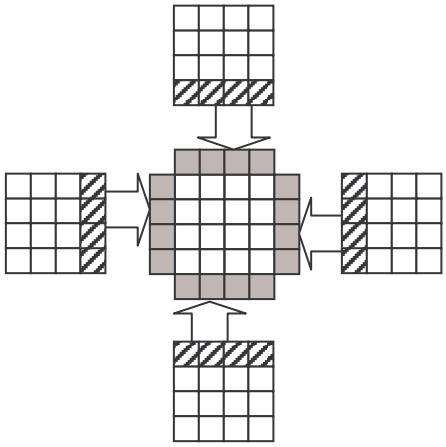


Figure 4.3: The fluff (shaded) for an array block showing the non-local values on adjacent processors that must be moved to fill the fluff; once the fluff is filled, the stencil computation is entirely local. (The "missing corners" are not used, but they are allocated to simplify array index calculations.)

Cyclic and Block Cyclic

As effective as block allocations are, they are not optimal for all algorithms, because of load balancing considerations. For example, as mentioned in Chapter 3, the well-known LU decomposition algorithm begins with all rows and columns participating in the initial step. As the computation proceeds, however, a row and column are completed with each iteration, leading to the schematic shown in Figure 4.4. When all of the rows and columns allocated to a processor are completed, it becomes idle. In the figure, 3 of the 4 processors are idle for half the computation. What should be done?

One solution is to reallocate the data periodically during the computation, but this requires moving nearly all of the active values to other processors, which is considered extremely expensive. The more practical solution is to use a cyclic or block cyclic distribution. The "cyclic" idea is to assign consecutive items to processors in a round-robin order, or as it's often described, as if dealing out cards. Thus a cyclic allocation of array elements proceeds through the array in, say, row-major order, allocating elements to processors. Because keeping track of individual array elements is burdensome, it is more common to "deal out" consecutive subarrays, a strategy called *block-cyclic* allocation.

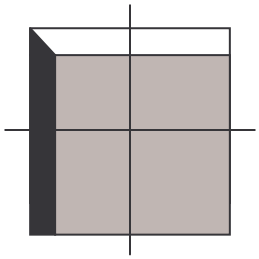


Figure 4.4: Schematic diagram of the LU Decomposition algorithm block-assigned to four processors; the final result is a lower (black) and an upper (white) triangular matrices; active computation is gray; a column and row are completed and added to each result matrix, respectively, in each iteration.

Figure 4.5 shows a block-cyclic allocation in which consecutive array blocks are assigned to separate processors cyclically. The figure shows several features of block-cyclic allocations: A block's dimension (called the *chunk size*) does not have to divide the array's dimension; the block is simply truncated. Each processor receives blocks from throughout the array, implying that as the computation proceeds, completed portions will be resident on each processor, as will not-yet-completed portions. Figure 6 shows the schematic allocation of Figure 5 as it would appear part way through an LU-type computation. Notice how well the remaining work is balanced across the processors.

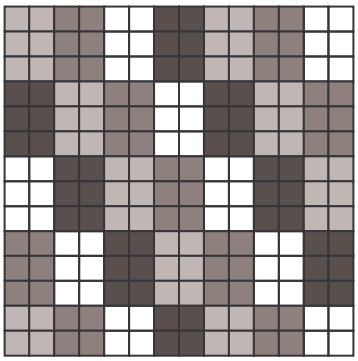


Figure 4.5: Block-cyclic allocation of 3×2 blocks to a 14×14 array distributed to four processors (colors).

The block-cyclic approach has the great advantage of balancing the work across the processors (see Figure 4.6), but this does not come without costs. The most obvious cost is the potential of block-cyclic allocation to complicate the algorithm. If the computation uses the spatial properties of an array—for example, rows—then because block-cyclic breaks some of these relationships, special cases may have to be added to the algorithm. This effect can sometimes be neutralized by picking chunk sizes to create easy-to-work-with patterns in the allocation. Another common recommendation is to allocate relatively large blocks, say 64×64, as a means of amortizing the overhead of the special checking. Of course, allowing the blocks to become too large probably means that the work will be less evenly allocated, and that the unbalanced case at the "end" of the computation will occur sooner and last longer. It is a delicate matter to balance the competing goals of a block-cyclic allocation.

Finally, notice that the block allocations discussed above and the block-cyclic allocations discussed here do not exploit locality equally well, even when block-cyclic uses large blocks. In general, for a given number of processors and array size, block-cyclic will use many smaller blocks whereas the block approach will use a single larger block per processor. This means for computations requiring nearest neighbor communication, e.g. stencils, the surface to volume advantage of blocks will result in much less communication. (The extremely small blocks of Figure 4.5 emphasize this point since every element is on the surface!) Of course, for computations compatible with a single allocation strategy, it is an easy matter to choose the right one. But, for cases where different phases of the computation would benefit from different allocations, it can be difficult to find the right compromise.

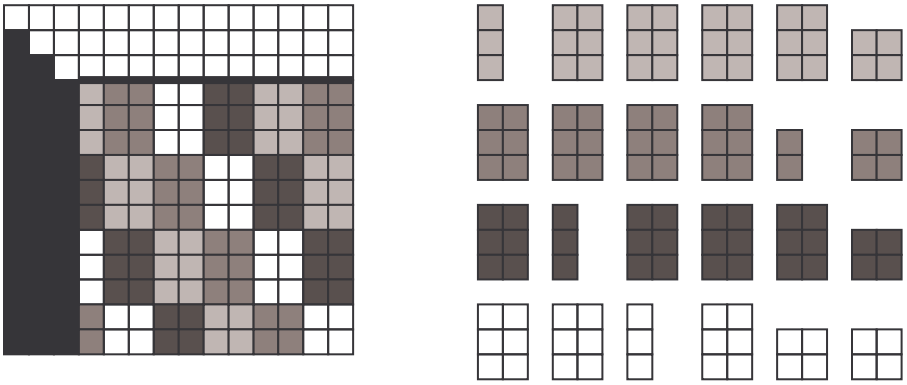


Figure 4.6: The block-cyclic allocation of Figure 4.5 midway through the computation; the blocks to the right summarize the active values for each processor.

Assigning Work to Processors Dynamically

In many cases it is not possible to adopt a fixed work assignment and stay with it. Examples include algorithms that create tasks as they proceed, algorithms whose tasks have highly variable execution times, adaptive algorithms that apply their computing power where the solution needs the greatest work, etc. In these and similar cases the work queue strategy may lead to the best assignment of computation.

Work Queue

A work queue is a first-in first-out list of task descriptors. If as a computation proceeds new work is generated, it is packaged into a task descriptor and is appended to the work queue; if as computation proceeds work is completed and a processor becomes available, it removes a task descriptor from the queue and begins work on it. The commonly used names for these two roles are *producer* and *consumer*.

As an example of a trivial task that can be expressed in work queue form, consider the $3n+1$ conjecture (or Collatz Conjecture), which proposes an affirmative answer to the question "For any positive integer a_0, does the process defined by

$$a_i = \begin{cases} 3a_{i-1} + 1 & a_{i-1} \text{ odd} \\ a_{i-1} / 2 & a_{i-1} \text{ even} \end{cases}$$

converge to 1?" (See http://mathworld.wolfram.com/CollatzProblem.html.) Though this conjecture is known to be true for all integers less than $3 \cdot 2^{53}$, we will program a search of the integers as an example, because it illustrates several aspects of work queue technique.

Our solution postulates a work queue containing the next integers to test. We initialize the queue to the first P positive integers:

```
void init(work_q: q) {     //setup globally-allocated queue array
      int i;
```

```
      for (i=0; i<P; i++) {
        q[i] = i+1;
      }
)
```

The integers are our task descriptors. As a general principle, it is wise to make the task descriptors as small as possible, while making them self-contained.

A worker thread, of which we assume there are *P*, will consume the first item from the work queue, add *P* to it, and append the result to the work queue. The rationale for adding *P* is that *P* threads will be checking integers at once, so advancing by *P* has the effect of skipping those that are (logically) being processed by other threads. (The other threads may not all be computing yet, but they will be.) The worker thread has the following logic:

```
int tester(int: limit) {    //test the conjecture
      int a;                //test number
      int n;                //count number of rounds of testing
      while (n < limit) {
        n = 0;              //initialize
        a = consume();      //remove the first element of work q
        produce(a+P);       //place new item on work q
        while (a != 1 && n < limit) {
           if (even(a))
               a = a / 2;
           else
               a = 3*a + 1;
        n++;
        }                   // exiting w/a==1 confirms converge
      }                     // exiting means limit was exceeded
      return a;             // tell what number is the culprit
)
```

The tester takes a `limit` as a parameter. The processing loop is controlled by `a != 1` and by `n < limit`. Because we think the conjecture is true, we expect the processing loop to exit with `a == 1`, but if there were an integer for which the conjecture is false, the loop will exceed the `limit`. If the `limit` is reached the worker will stop and return; otherwise it continues checking until it is eventually preempted by the parent routine. Of course, the parent, receiving a number back from a worker, cannot know if it is a counter-example to the conjecture, but it can test further to see if the number is actually convergent, but that the `limit` was set too low. This simply requires increasing the size of `limit` and rechecking.

Consider the behavior of the work queue. First, notice that in effect *P* subsequences are being checked simultaneously, but they will not remain in lock step because the amount of work required of each check is different. For example, with four processors the queue might transition through the states at the beginning

Work Queue	Active Processors [task]

1, 2, 3, 4	--
2, 3, 4, 5	$\boldsymbol{p_0}$**[1]**
3, 4, 5, 6	$\boldsymbol{p_0}$**[2]**
4, 5, 6, 7	p_0[2], $\boldsymbol{p_1}$**[3]**
5, 6, 7, 8	p_0[2], p_1[3], $\boldsymbol{p_2}$**[4]**
6, 7, 8, 9	$\boldsymbol{p_0}$**[5]**, p_1[3], p_2[4]
7, 8, 9, 10	p_0[5], p_1[3], p_2[4], $\boldsymbol{p_3}$**[6]**

illustrating that because processing 1 is trivial, p_0 might return to the queue to get the next value, 2, before any of the other processors start; processing 2 is also easy, so it is back again quickly. Further, notice that workers do not necessarily process the same subsequence. Indeed, if the timing works out all of the processors could be working on the same subsequence at once. Summarizing, although our work queue is regimented, the way its tasks are processed can accommodate any timing characteristics.

One important detail has been ignored in the worker code. It is the matter of races resulting from multiple threads referencing the work queue simultaneously. We have assumed that the two procedures, `consume()` and `produce()` contain the appropriate synchronization apparatus. Exact details of how to construct the appropriate protection are presented in the next chapter.

The Reduce & Scan Abstractions

The success of Schwartz's algorithm for addition and Count 3s computations is not accidental. Operations of adding and counting array elements are instances of a general form of computation known as *reduce* and *scan*. They are well understood, and so, efficient solutions are known. By recognizing such computations, we can avoid working out their details each time and simply apply standard solutions, such as Schwartz's algorithm. We can save our serious thinking for those parts of a computation that need brain power.

Reduce, which can be thought of as short for "reduce the operand data structure to a single value by combining with the given operator," has been a part of programming languages since the creation of the array language APL. The operators are often limited to the associative and commutative operations *add*, *multiply*, *and*, *or*, *max* and *min*, but many operations work. Though some languages have built-in reduce operations, many do not, and so we will create our own general routine based on the Schwartz approach. To simplify our discussion of reduce, we adopt the notation *operator* **//** *operand*, as in `+ // A`, to describe `A`'s elements being reduced using addition. We might write the Count 3s computation `+ // (A == 3)`, and expect to implement it by instantiating a general Schwartz solution with implementations of these operations.

Scan is a synonym for "parallel prefix," mentioned in Chapter 1. Scan is related to reduce in that scan is the reduce computation in which the intermediate values are saved, assuming a specific order of evaluation. Thus, for array `A` with values

```
4   2   5   6   1
```

the plus-scan of A is

```
4   6   11   17   18
```

Reduce is simply the final value produced in a scan. (We adopt the notation *operator* **** *operand*, as in `+ \\ A`.) As with reduce the associative and commutative operations *add*, *multiply*, *and*, *or*, *max* and *min* are common, but many other also computations work.

To illustrate using the reduce and scan abstractions to compute the bounding-box enclosing points in the plane. Let A be an array of n points of the form,

```
type planePt = record
        x : int;
        y : int;
     end;
```

then the sequential computation

```
maxX = A[0].x;
for (i = 1, i < n, i++)
{
     if (maxX < A[i].x)
         maxX = A[i].x;
}
maxY = A[0].y;
for (i = 1, i < n, i++)
{
     if (maxY < A[i].y)
         maxY = A[i].y;
}
minX = A[0].x;
for (i = 1, i < n, i++)
{
     if (minX > A[i].x)
         minX = A[i].x;
}
minY = A[0].y;
for (i = 1, i < n, i++)
{
     if (minY > A[i].y)
         minY = A[i].y;
}
```

for computing the four defining values for the bounding box is equivalent to four applications of reduce,

```
maxX = max//A.x
maxY = max//A.y
minX = min//A.x
minY = min//A.y
```

which as we saw in Chapter 1 can be efficiently implemented in parallel. Of course, these four can be merged into one implementing procedure, so that there is only one pass over the data and only one combining tree.

Generalized Reduce, Scan and Vector Operations

Because reduce and scan are such effective abstractions for thinking about parallel computation, we advocate using them, and developing tools for their convenient application. Because there are so many programming systems in use, we describe how to construct an implementation, rather than giving a specific one. We begin the section with reduce, and move on to scan. Finally, we observe that the concepts can be applied to general vector operations.

Structure for Generalized Reduce

To build a general reduce or scan implementation, visualize the Schwartz algorithm, as abstracted in Figure 4.7. Overall, the figure shows local computation performed at the leaves of a combining tree, which emits the reduction result at its root. Looking more closely at the diagram, we see that a data structure called the tally is used together with four functions, two applied on each processor:

> `init()` initializes the process on each processor, setting up the *tally* data structure recording the local result as the reduce performs the accumulation operation
> `accum(tal, val)` performs the actual accumulation by combining the running tally with the operand value

Once the local results are found, they must be combined to form the global result, using two more functions:

> `combine(left, right)` combines two tally values to create a new tally value
> `reduceGen(root)` takes the global tally value and outputs the correct result for reduce.

For example, to compute `+//A`, the `init()` routine would initialize a tally variable to 0; the `accum()` routine would add its tally to an operand value; the `combine()` would add two local tallies and `reduceGen()` is a noop that simply returns the result.

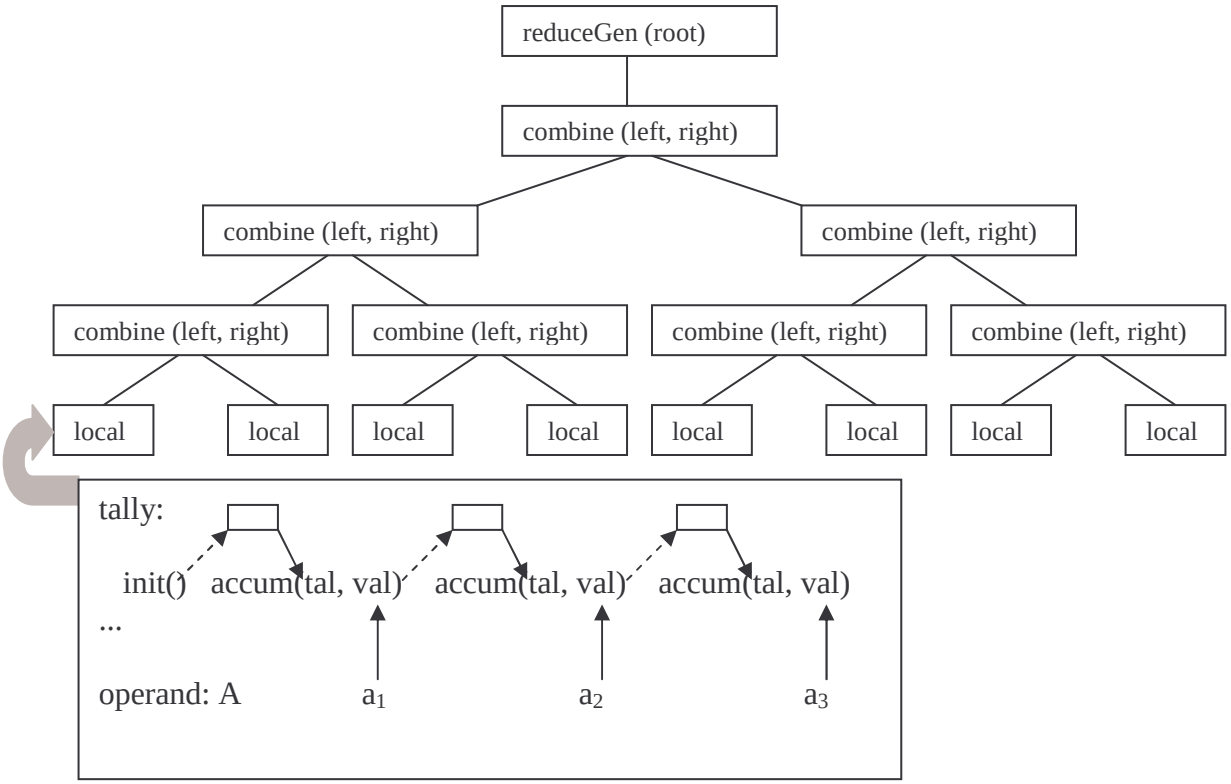


Figure 4.7: Schematic diagram of the Schwartz algorithm used to implement user-defined reduce. The local operations are abstracted in the box; function `init()` sets up the tally; `accum()`combines the tally and the operand data structure; outside the box, in the combining tree, `combine()` forms a new tally from two others, and `reduceGen()` produces the final answer from the global tally.

For the simple operation of `+//A`, the four-function formulation is excessively general, but this structure is essential in more complex (and more powerful) cases, as illustrated in a moment. The key to understanding the roles of the four reduce functions is first to recognize that the tally data structure need not have the same type as the operand data, and second that the four routines take different argument types. So, for example, imagine a user-defined `secondMin // A` that finds the second smallest element in an array, useful perhaps for an array of non-negative numbers with many 0s. In this case the tally data structure would have to be a two-element record or array storing the smallest and second smallest values. The `init()` function would set-up the tally, initialized to the + infinity for the operand data type; `accum()` would compare an operand value with the tally elements recording the two smallest; `combine()` would find the two smallest of its two tally arguments, and the `reduceGen()` would return the second smallest value as the result. When called at the right points in a parallel procedure implementing Schwartz's algorithm, they produce a parallel `secondMin()` solution. Figure 4.8 makes this logic precise.

```
type tally = record
       sm1 : float;                    //smallest element
       sm2 : float;                    //second smallest
     end;

void init(tally: tal) {       //setup globally-allocated tally
      tal.sm1 = MAX_FLOAT;
      tal sm2 = MAX_FLOAT;
)
void accum(float: elem, tally: tal) {   //local accumulation
      if (tal.sm1 > elem) {
          tal.sm2 = tal.sm1;
          tal.sm1 = elem;
      }
      elseif (tal.sm2 > elem) {
          tal.sm2 = elem;
      }
}
void combine(tally: left, tally: right) { //combine into "left"
      accum(right.sm1, left);             //by accumulating right
      accum(right.sm2, left);             //values one at a time
}
float reduceGen(tally: tal) {
      return tal.sm2;
}
```

Figure 4.8: The four user-defined reduce functions implementing `secondMin` reduce. The tally, globally defined on each processor, is a two-element record.

Structure for Generalized Scan

Generalized scan applies the same concepts as generalized reduce. The primary difference is that after the combining is complete the "parallel prefix" values must be passed back down the combining tree. That is, in order to complete the prefix computation on the local values, an intermediate value from the combining tree will be needed by each processor. (Refer to the parallel prefix discussion in Chapter 1 and review Figure 1.2.)

The generalized scan begins like the generalized reduce, and there is no conceptual difference in the three functions `init()`, `accum()` and `combine()` for the two algorithms. However, the scan is not finished when the global tally value has been computed. Rather, tally values must be propagated down the tree subject to the constraint that

> *for any node, the value it receives from its parent is the tally for the values that are left of its leftmost leaf*

which because we are computing on blocks, means the first item in the leftmost leaf's block. This causes us to call `init()` to create the value as the input from the logical parent of the root, because there is no tally for the items to the left of those covered by the root. Each node, receiving a value from the parent, relays it to its left child; for its right

child, it combines the value received from the left child on the upsweep with the value received from the parent, and sends the result to the right.

When the tally value is received at a leaf, it must be combined with the values stored in the operand array to compute the prefix totals, which are stored in the operand position. In Figure 4.9 these operands are shown schematically in the box at the bottom. Thus, the `scanGen()` procedure produces the final result.

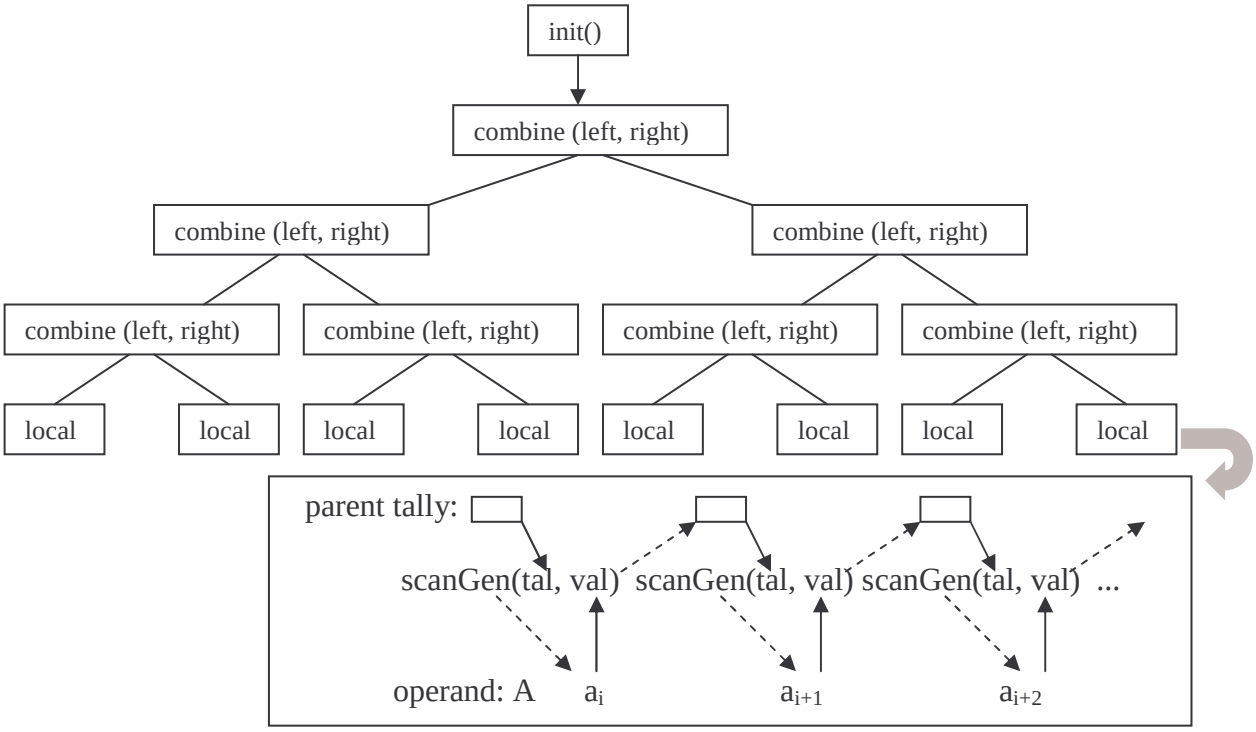


Figure 4.9: Schematic of the scan operation. The first part of the algortihm is simply the generalized reduce, schematized in Figure 4.7. Once the global tally is found, prefixes are propagated down the tree. When a prefix arrives at a leaf, the local operation applies the `scanGen()` function, and stores the result in the operand item.

To illustrate the operation of user-defined scan, imagine an array `A` of integers from the sequence 0, …, *k*-1. The scan `sameAs \\ A` records in position `A[i]` the number of elements in the first *i* matching `A[i]`. We use as a tally an array of *k* elements, which is initialized to 0s; the accumulate function increments the array item count for the operand value; combine function adds the two arrays; and the scan generator performs an accumulate (initialized this time by the prefix received from the parent), and stores the count for the item found. Figure 4.10 shows the functions that realize this result. (Notice that the tally array at the end is a histogram for the array.)

```
void init(tally: tal) {         //setup globally-allocated array
      for (i=0; i<k; i++) {
         tal[i]= 0;
      }
)
void accum(int: elem, tally: tal) {       //local accumulation
      tal[elem]++;
}
void combine(tally: left, tally: right) { //combine into "left"
      for (i=0; i<k; i++) {
           left[i] += right[i];
      }
}
int scanGen(int: elem, tally: tal) {      //finalizing scan
      accum(elem, tal);                   //accum w/parent tally
      return tal[elem];                   //store running count
}
```

Figure 4.10: User-defined scan functions to return the running count of *k* items; the tally is a globally allocated array of *k* elements.

Structure for Gneralized Vector Operations

The foregoing discussion shows that instantiating the basic structure of reduce and scan with custom functions can create efficient parallel solutions. But the idea is even more general. There is no need that the operations "accumulate from left to right;" computations that can be performed on blocks of data that can be merge to produce a larger solution are good candidates for applying this same structure. We illustrate the idea by computing the longest run of positive values stored an array. So, for the sequence

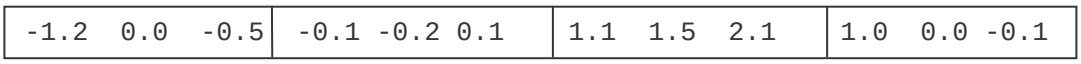
```
-1.2  0.0  0.5 -0.1 -0.2 0.1  1.1  1.5  2.1  1.0  0.0 -0.1
```

the computation would return 5.

To formulate a local block computation to find the longest run of positive values, observe that the run could straddle the boundary (or boundaries) between local blocks. For that reason, we select a tally that has three values

```
type runLen = record
       atstart : int;   // count of positives from left
       longest : int;   // longest (interior) run found so far
       current : int;   // length of current run
     end;
```

that will count the number of positive values beginning at the start of a block, `atstart`, if any, the longest run properly contained in the block, `longest`, and the length of the current run, `current`. This last variable will have the effect of recording the length of the positive run extending to the right end of the block, if any. Because the block will be processed left to right, it will be convenient to treat a sequence that completely spans a block as having an undefined `longest` and `current` values. So for example, dividing the foregoing example among four processors



results in the tallies

Notice, the third thread has the undefined values in its tally.

To build the four reduce functions, initialize the tally items to undefined (represented as -1), but to simplify the combining logic later, set `current` to 0.

```
atstart: -1
longest: -1
current:  0
```

Accumulating begins by counting positive items in `atstart` until the sequence is broken; thereafter, it counts positive items in `current`, and records the length of the runs in `longest`. Thus, `accum()` must separate the initial sequence from the others, and for that it requires a cascade of logic as shown in Figure 4.11.

Given two tallies, their combined tally must handle four cases. These are

- both blocks span only positive elements: the result spans positive elements, so add the right block's `atfirst` to the left block's `atfirst`, the blocks' `longest` and `current` are the same
- the left block spans only positive elements: the right block's `atfirst` adds to left's `atfirst`, and the right block's `longest` and `current` apply
- the right block spans only positive elements: add the right block's `atfirst` to the left block's `current`, and the left block's `atfirst` and `longest` apply
- both blocks have non-positives; the left block's `current` plus the right block's `atfirst` could be longer than either `longest`, the left block's `atfirst` and the right block's `current` apply

We use `longest == -1` as our test for a positive only block. This logic is implemented as a cascade of `if`-statements.

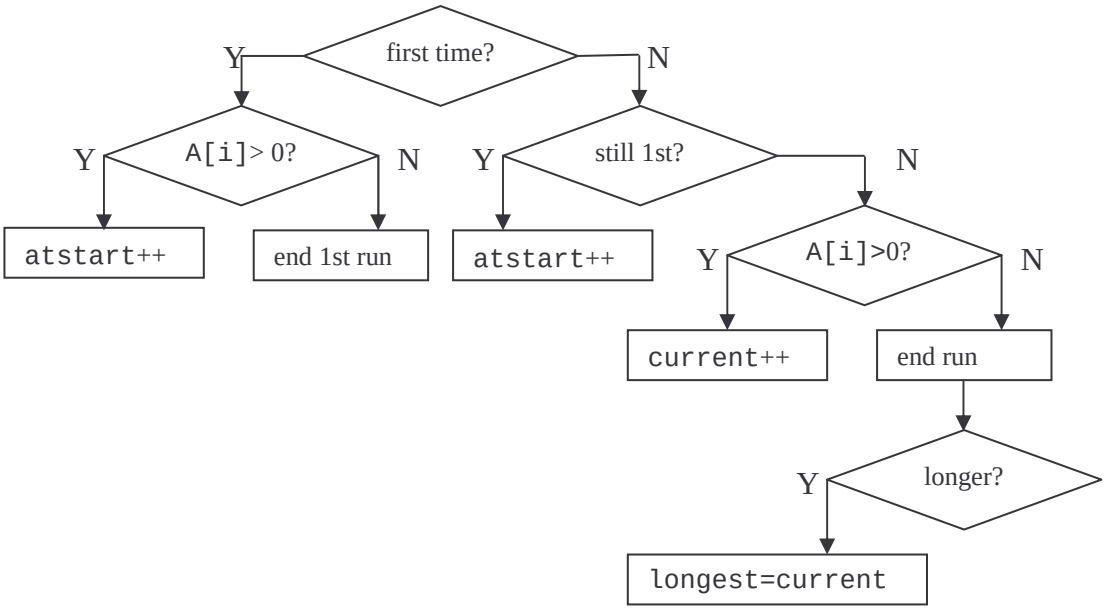


Figure 4.11: The accumulate logic. The "first time?" test is true when `atfirst == -1`; the "still 1st?" test is true when `longest == -1`.

Finally, the global result must pick the largest of its `atfirst`, `longest` and `current` values. If the longest sequence starts at the beginning, then the preceding logic ensures that `atfirst` will record its length; if the longest sequence extends to the end, the `current` will record the length. Otherwise the longest sequence is somewhere in the middle, and `longest` will record the value.

See Figure 4.12 for the exact logic.

These are powerful techniques that have efficient parallel implementations.

```
void init(runLen: zero) {     //setup globally-allocated record
      zero.atstart = -1;
      zero.longest = -1;
      zero.current = 0;
)
void accum(int: elem, runLen: z) {   //local accumulation
      if (z.atstart == -1) {         //first time?
         if (elem > 0)
             z.atstart = 1;
         else {
             z.atstart = 0;
             z.longest = 0;
         }
      }
      else {
         if (z.longest == -1) {      //still first time?
             if (elem > 0)
                  z.atstart++;
```

```
            else
                 z.longest = 0;
         }
         else {
            if (elem > 0)
               z.current++;
            else {
              if (z.longest > z.current)
                  z.longest = z.current;
              z.current = 0;
            }
         }
      }
}
void combine(runLen: left, runLen: right) { //combine into "left"
      if ((left.longest == -1) && (right.longest == -1)) //spans
          left.atstart = left.atstart + right.atstart;
      else {
         if (left.longest == -1) {
             left.atstart = left.atstart + right.atstart;
             left.longest = right.longest;
             left.current = right.current;
         }
         else {
            if (right.longest == -1)
                left.current = left.current + right.atfirst;
            else {
               left.longest = MAX(left.longest,
                                  left.current + right.atstart,
                                  right.longest);
               left.current = right.current;
            }
         }
      }
}

int reduceGen(runLen: z)   {
      if (z.longest < z.atfirst)
          z.longest = z.atfirst;
      if (z.longest < z.current)
          z.longest = z.current;
      return z.longest;
}
```

Figure 4.12: The four reduce/scan functions for the “longest positive run” computation.

Trees

After arrays, trees must be the most import way to represent a computation. They present challenges in parallel computation for several reasons. First, trees are usually constructed using pointers, and in many parallel computation situations, pointers are local only to one processor. Second, we typically use trees for their dynamic flexibility, but dynamic

behavior often implies performance-bashing communication. Third, trees complicate allocation-for-locality. But, challenging or not, trees are too useful to ignore.

Representation of Trees

Begin by noticing that we have already used trees in several computations to perform accumulation and parallel prefix operations. They were implicit in that they derive from the communication patterns used. So, in the reduce and scan primitives above, the `combine()` operations were performed pairwise, with the intermediate results also being combined pairwise, etc. inducing a tree, as shown in Figure 4.13. There are no pointers; processors simply perform the appropriate tree roles, and the result is achieved. By this technique we use trees to perform global operations even when they are not the base data structure of the computation.

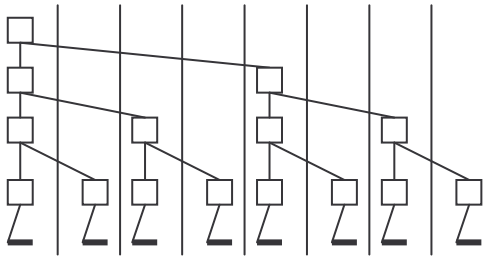


Figure 4.13: Induced tree. Each processor computes on a sequence of values (heavy lines), and then combines the results pairwise, inducing a tree; notice processor 0 participates at each level in the tree.

Our Guideline #1 rule, to maximize the number of large blocks of independent computation, motivates us to use the implicit tree idea even in cases where the base data structure is a tree. This means that we separate the local and the global paradigms: Locally, we may choose to use pointers in our implementation, but at the higher levels of the tree where the edges are non-local, we use the implicit solution – each node simply performs its proper tree role. Though it can be inconvenient to shift processing paradigms, the advantage may be that it allows us to write a single-threaded solution to the subproblem (it might already exist), and then incorporate those subproblems into the global solution.

Breadth First. Consider first trees that can naturally be enumerated breadth first, that is, all nodes of a level can be generated given their parent nodes. In this case we conceptually generate the complete tree down to the level, or pair of levels, having P nodes, one corresponding to each processor. So, in the binary tree case, if $P = 2^l$, generate to level l. For example, for $P = 8$, we generate a binary tree down to level 3, as shown in Figure 4.14(a). When P does not equal the number of nodes on a level, pick the greatest level less than P and then expand enough of the nodes to the next level to equal P, as shown in Figure 4.14(b). Then, assuming the tree extends much more deeply below each of these nodes, allocate to each processor corresponding to a node the entire subtree enumerated from the node. The computation is conceptually local for the whole subtree.

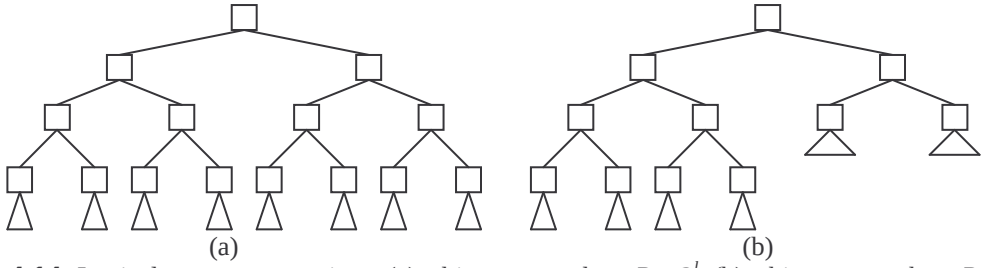


Figure 4.14: Logical tree representations: (a) a binary tree where $P = 2^l$; (b) a binary tree where $P = 6$.

Example. This technique works well for problems that can be recursively partitioned into subproblems. For example, suppose we are searching a game tree for Tic-Tac-Toe (Naughts and Crosses) games on $P = 4$ processors. When symmetries are considered, there are only three initial positions, and we expand one of these to fill out the 4 search tasks, see Figure 4.15. That is, each processor will search the game tree descendant from the indicated board position.

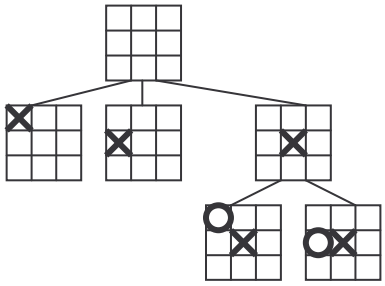


Figure 4.15: Enumerating the Tic-Tac-Toe game tree; a processor is assigned to search the games beginning with each of the four initial move sequences.

Depth First. Trees that should be enumerated by depth can be implemented in parallel using a work queue approach. The queue is initialized with the root; a processor removes a node and if that action leaves the queue empty, the processor expands the node, taking one descendant as its task and appending the others to the queue. Such an approach corresponds to standard iterative depth first traversal, and has a structure as shown in Figure 4.16.

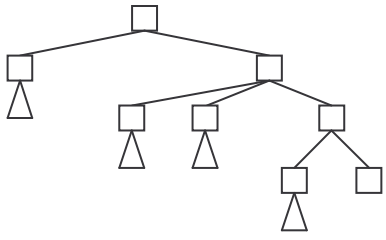


Figure 4.16: Basic depth allocation of a tree to P=4 processors, which are each responsible for the subtree rooted at the node; the right-most node remains in the queue.

Having assigned a subtree to each processor, consider the main aspect of depth first enumerations, the feedback from the early parts of the enumeration to the later parts. This

takes various forms including a list of nodes visited, alpha-beta limits on productive subtrees to consider, and other measures of "progress". A parallel algorithm will have to adapt such feedback to the processing of independent subtrees. For example, consider a packing algorithm trying to minimize the area occupied by the arrangement of several objects. A global variable recording the area of the best arrangement found so far can be referenced by all processors and so be used in each subtree to prune the enumeration.

Full Enumeration. Certain trees must be expanded in their entirety. A common example is the family of K-D Trees used in gravitational simulations and the closely related Barnes-Hut trees. Such trees are used for rapid lookup of related elements. In the case of gravitation simulation, 3D space is partitioned into octants, which are in turn partitioned, etc. until each region contains only one point. This allows the points physically near a given point to be quickly located by tree traversal. (Advanced algorithms allow groups of points acting at a distance to be approximated as a single meta-point.) Areas of high concentration can lead to locally deep trees.

Perhaps the best allocation for such a tree is the so-called "cap allocation," as shown in Figure 4.17. In the allocation the *P* nodes nearest the root, the cap, are redundantly allocated to each processor. Additionally, a processor is also allocated one of the subtrees rooted at the bottom of the cap. As the computation proceeds, the cap portion of the tree must be maintained coherently. That is, all processors must "see" the same state, and a locking protocol must be respected. Interaction among the subtrees can use a messaging protocol.

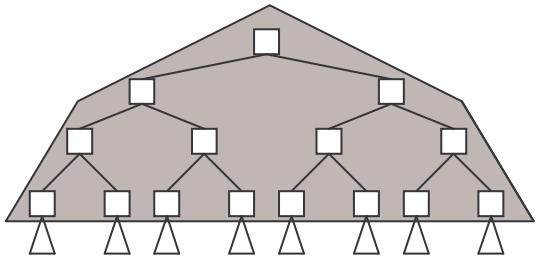


Figure 4.17. Cap allocation for a binary tree on *P*=8 processors; the cap (shaded) and one of the "leaf subtrees" are allocated to each processor.

The cap allocation is effective primarily because most of the activity takes place in the subtrees, and therefore is entirely local to a processor. Further, activity around the root is rare, so there is little likelihood for lock contention. Finally, the availability of the root means that interactions "crossing the root" can be navigated by percolating up in the local tree and crossing the cap locally, so as to identify the correct destination subtree. Navigating in the destination subtree is typically assigned as a task to the owner. As an additional bonus in the advanced algorithms for gravitational simulations in which large regions of the problem are aggregated into meta-points, the points around the root are all meta-points, and therefore are read-only, eliminating races and locking as issues.

Summary

Exercises

Exercise 0. Write a sequential program to perform the operations illustrated in Figure 3. That is, using an imagined library operation GetIt(<direction>, <buffer>) to transmit the data from the neighboring threads, compute a 4-point stencil computation on the interior data array A. Assume that GetIt blocks if the data on the other processor isn't ready; also, assume that buffer is a separate block of memory not related to A, i.e. you must include the fluff buffers in your portion of the data array and fill them manually.

Exercise 1. Generalize the longest positive run program to return a Boolean mask with a 1 in the element positions for every positive value in the longest run, and 0s everywhere else. This computation uses the existing functions slightly modified, but replaces the `reduceGen()` with `scanGen()` to produce the final values. There is also a revision required for the tally data structure. [Hint: Assume in the accumulate function the availability of a variable, called `index`, giving the index of the element being processed.]

Exercise 2. Revise the timing assumptions of the work queue example so that all processors are working on the subsequence: 4, 8, 12, 16. (There are multiple solutions.)

Ex. Revise the tester program so that it exploits the fact that if the threads have established that all number less than k converge, then no thread need check further when $a_i < k$.

Historical Context

Need in the historical section a bunch of things about the PRAM, Fetch and Add, and other one-point-per processor schemes, such as SIMD. Need to cite Anderson/Snyder to emphasize that there are more fundamental reasons to not like PRAMs than lock contention. Cite Blelloch. Ladner and Fischer, Deitz; who thought up block-cyclic—Lennert?