Chapter 2

Parallel Programming Considerations

Ken Kennedy, Jack Dongarra, Geoffrey Fox, Ian Foster, Dan Reed, and Andy White

2.1 Introduction

The principal goal of this chapter is to introduce the common issues that a programmer faces when implementing a parallel application. The treatment assumes that the reader is familiar with programming a uniprocessor using a conventional language, such as Fortran. The principal challenge of parallel programming is to decompose the program into subcomponents that can be run in parallel. However, to understand some of the low-level issues of decomposition, the programmer must have a simple view of parallel machine architecture. Thus we begin our treatment with a discussion of this topic. This discussion, found in Section 2.2, focuses on two main parallel machine organizations — shared memory and distributed memory — that characterize most current machines. The section also treats clusters, which are hybrids of the two main memory designs.

The standard parallel architectures support a variety of decomposition strategies, such as decomposition by task (task parallelism) and decomposition by data (data parallelism). Our introductory treatment will concentrate on data parallelism because it represents the most common strategy for scientific programs on parallel machines. In data parallelism, the application is decomposed by subdividing the data space over which it operates and assigning different processors to the work associated with different data subspaces. Typically this strategy involves some data sharing at the boundaries, and the programmer is responsible for ensuring that this data sharing is handled correctly — that is, data computed by one processor and used by another is correctly synchronized.

Once a specific decomposition strategy is chosen, it must be implemented. Here, the programmer must choose the programming model to use. The two most common models are:

- the *shared-memory model*, in which it is assumed that all data structures are allocated in a common space that is accessible from every processor, and
- the *message-passing model*, in which each processor is assumed to have its own private data space.

In the message-passing model, data is distributed across the processor memories; if a processor needs to use data that is not stored locally, the processor that owns that data must explicitly "send" the data to the processor that needs it. The latter must execute an explicit "receive" operation, which is synchronized with the send, before it can use the communicated data. These issues are discussed in Section 2.3.

To achieve high performance on parallel machines, the programmer must be concerned with *scalability* and *load balance*. Generally, an application is thought to be scalable if larger parallel configurations can solve proportionally larger problems in the same running time as smaller problems on smaller configurations. To understand this issue, we introduce in Section 2.3.1 a formula that defines parallel *speedup* and explore its implications. Load balance typically means that the processors have roughly the same amount of work, so that no one processor holds up the entire solution. To balance the computational load on a machine with processors of equal power, the programmer must divide the work and communications evenly. This can be challenging in applications applied to problems that are unknown in size until run time.

A particular bottleneck on most parallel machines is the performance of the memory hierarchy, both on a single node and across the entire machine. In Section 2.4, we discuss various strategies for enhancing the reuse of data by a single processor. These strategies typically involve some sort of loop "blocking" or "strip-mining," so that whole subcomputations fit into cache.

Irregular or adaptive problems present special challenges for parallel machines because it is difficult to maintain load balance when the size of subproblems is unknown until run time or if the problem size may change after execution begins. Special methods involving runtime reconfiguration of a computation are required to deal with these problems. These methods are discussed in Section 2.3.3.

Several aspects of programming parallel machines are much more complicated than their counterparts for sequential systems. Parallel debugging, for example, must deal with the possibilities of race conditions or out-of-order execution (see Section 2.5). Performance analysis and tuning must deal with the especially challenging problems of detecting load imbalances and communication bottlenecks (see Section 2.6). In addition it must present diagnostic information to the user in a format that is related to the program structure and programming model. Finally, input/output on parallel machines, particularly those with

distributed memory, presents problems of how to read files that are distributed across disks in a system into memories that are distributed with the processors (see Section 2.7).

These topics do not represent all the issues of parallel programming. We hope, however, that a discussion of them will convey some of the terminology and intuition of parallel programming. In so doing, it will set the stage for the remainder of this book.

2.2 Basic Parallel Architecture

The next chapter provides a detailed review of parallel computer architectures. For the purposes of this chapter we will provide a simple introduction to these topics that covers most of the important issues needed to understand parallel programming.

First, we observe that most of the modern parallel machines fall into two basic categories:

1. *Shared-memory machines*, which have a single shared address space that can be accessed by any processor, and
2. *distributed-memory machines*, in which the system memory is packaged with individual processors and some form of communication is required to provide data from the memory of one processor to a different processor.

Shared Memory The organization of a shared-memory machine is depicted in Figure 2.1, which shows a system with four processors, each with a private cache, interconnected to a global shared memory via a single system bus. This organization is used on modern multiprocessor workstations, such as those from Sun and Hewlett-Packard. Many simple desktop multiprocessors share this design.

In a shared-memory system, each processor can access every location in global memory via standard load operations. The hardware ensures that the caches are "coherent" by watching the system bus and invalidating cached copies of any block that is written into. This mechanism is generally invisible to the user, except when different processors are simultaneously attempting to write into the same cache line, which can lead to "thrashing". To avoid this problem, the programmer and programming system must be careful with shared data structures and non-shared data structures that can be located on the same cache block, a situation known as "false sharing." Synchronization of accesses to shared data structures is a major issue on shared-memory systems — it is up to the programmer to ensure that operations by different processors on a shared data structure leave that data structure in a consistent state.

The main problem with the uniform-access shared-memory system is that it is not scalable. Most bus-based systems are limited to 32 processors. If the bus is replaced by a crossbar switch, systems might well scale to as many as 128 processors although the cost of the switch goes up as the square of the number of processors.

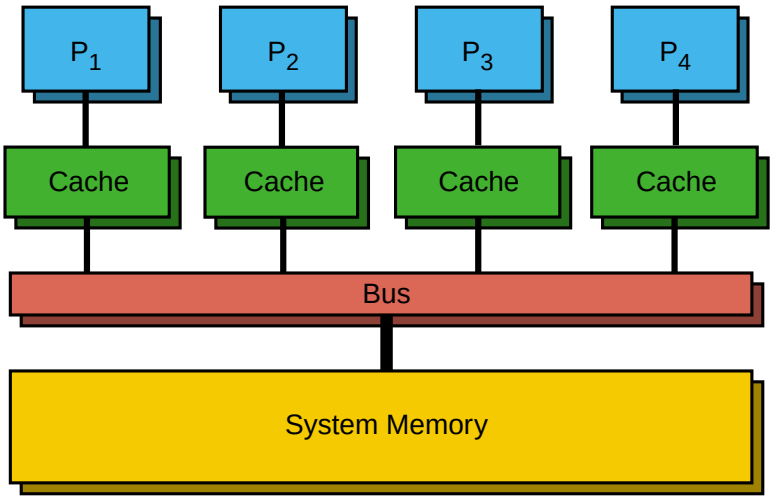

Figure 2.1: A Uniform-Access Shared-Memory Architecture

Distributed Memory The scalability limitations of shared memory have led designers to use distributed-memory organizations such as the one depicted in Figure 2.2. Here the global shared memory has been replaced by a smaller local memory attached to each processor. Communication among these configurations makes use of an interconnection network. Interconnection networks of this sort can employ a scalable design such as a hypercube.

The advantage of a distributed-memory design is that access to local data can be made quite fast. On the other hand, access to remote memories requires much more effort. On *message-passing systems*, the processor owning a needed datum must send it to the processor that needs it. These "send–receive" communication steps typically incur long start-up times, although the bandwidth after start-up can be high. Hence, it typically pays to send fewer, longer messages on message-passing systems.

Distributed Shared Memory Some distributed-memory machines allow a processor to directly access a datum in a remote memory. On these *distributed-shared-memory (DSM)* systems, the latency associated with a load varies with the distance to the remote memory. Cache coherency on DSM systems is a complex problem and is usually handled by a sophisticated network interface unit. As of this writing, the Silicon Graphics Origin and the HP/Convex Exemplar are the only commercial DSM systems.

The principal programming problem for distributed-memory machines is to manage communication of data between processors. On message-passing systems this amounts to minimizing the number of communications and attempting to overlap communication with computation.

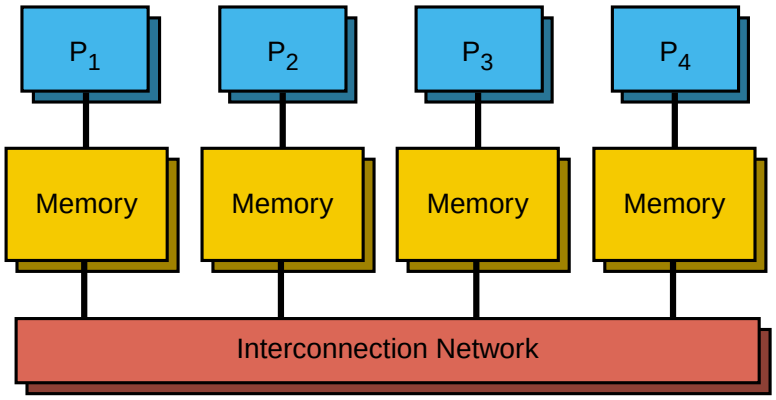

Figure 2.2: A Distributed-Memory Architecture

Clusters For very large parallel systems, a hybrid architecture called a *cluster* is becoming increasingly common. A cluster looks like a distributed-memory system in which each of the individual components is a shared-memory multiprocessor rather than a single processor node. This design permits high parallel efficiency within a multiprocessor node while permitting systems to scale to hundreds or even thousands of processors. Most manufacturers of high-end parallel systems are planning to offer some sort of cluster.

Memory Hierarchy The design of memory hierarchies is an integral part of the design of parallel computer systems because the memory hierarchy determines the performance of the individual nodes in the processor array of the machine. A typical memory hierarchy is depicted in Figure 2.3. Here the processor and a level-1 cache memory are found on chip and a larger level-2 cache lies between the chip and the memory.

When a processor executes a load instruction, the level-1 cache is first interrogated to determine if the desired datum is available. If it is, the datum can be delivered to the processor in 2–5 processor cycles. If the datum is not found in the L1 cache, the processor stalls while the L2 cache is interrogated. If the desired datum is found in L2, then the stall may last for only 10–20 cycles. If the datum is not found in either cache, a full *cache miss* is taken with a delay of possibly a hundred cycles or more. Whenever a miss occurs, the datum is saved in every cache in the hierarchy, if it is not already there. Note that on every modern machine, each cache has a minimum-size *cache block*, so that whenever a datum is loaded to that cache, the entire block containing that datum comes with it.

Generally the performance of the memory hierarchy is determined by two hardware parameters:

- *Latency,* which is the time required to fetch a desired datum from memory, and

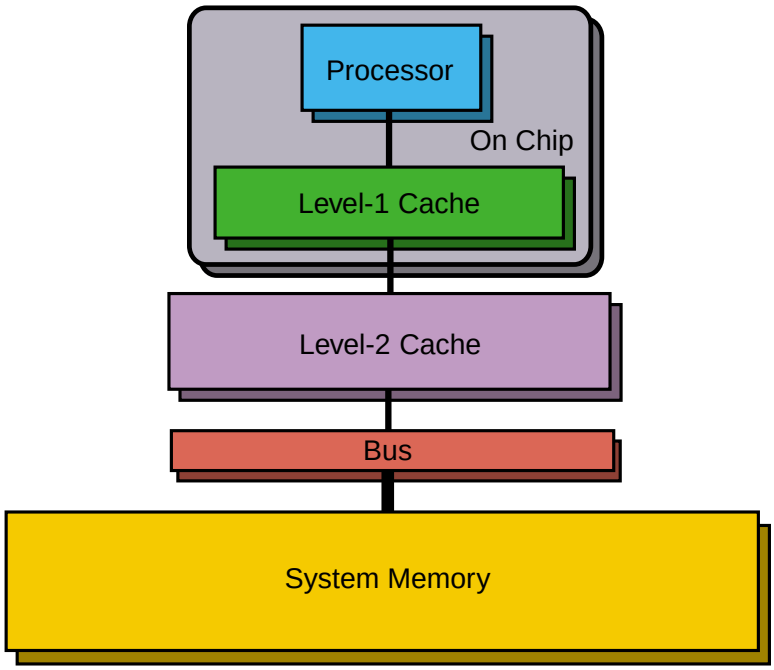

Figure 2.3: A Standard Uniprocessor Memory Hierarchy

- *Bandwidth*, which is the number of bytes per unit time that can be delivered from the memory at full speed.

Generally these two factors are complicated by the multilevel nature of memory hierarchies, because each level will have a different bandwidth and latency to the next level. For example, the SGI Origin can deliver about 4 bytes per machine cycle from L1 to the processor and 4 bytes per cycle from L2 to L1 but only about 0.8 bytes per cycle from memory to L1.

Another important parameter that affects memory performance on a uniprocessor is the length of the standard *cache block* (or *cache line*). Most cache systems will only transfer blocks of data between levels of the memory hierarchy. If all the data in a block that is actually transferred are used, then no bandwidth is wasted and the miss can be amortized over all the data in the block. If only one or two data items are used, then the average latency is much higher and the effective bandwidth much lower.

There are generally two kinds of strategies for overcoming latency problems. *Latency hiding* attempts to overlap the latency of a miss with computation. Prefetching of cache lines is a latency-hiding strategy. *Latency tolerance*, on the other hand, attempts to restructure a computation to make make it less subject to performance problems due to long latencies. Cache blocking, which we shall discuss shortly, is one such latency-tolerance technique.

Strategies that improve reuse in cache also improve effective bandwidth uti-

lization. Perhaps the most important way to ensure good bandwidth utilization is to organize data and computations to use all the items in a cache line whenever it is fetched from memory. Ensuring that computations access data arrays in strides of one is an example of how this might be done.

The memory hierarchies on parallel machines are more complicated because of the existence of multiple caches on shared-memory systems and the long latencies to remote memories on distributed-memory configurations. There may also be interference between data transfers between memories and from local memory to a processor.

2.3 Decomposing Programs for Parallelism

Identification of Parallelism Given that you have decided to implement a program for a parallel machine, the first task is to identify the portions of the code where there is parallelism to exploit. To do this we must address a fundamental question: *When can we run two different computations in parallel?* We cannot answer this question without thinking about what it means for two computations to run in parallel. Most programmers think of the meaning of a program to be defined by the sequential implementation. That is, for a parallel implementation to be correct, it must produce the same answers as the sequential version every time it is run. So the question becomes: *When can we run two computations from a given sequential program in parallel and expect that the answers will be the same as those produced by the sequential program?* By "running in parallel," we mean asynchronously, with synchronization at the end. Thus the parallel version of the program will fork, with each of the computations running until the end, and then they will synchronize.

The naive answer to the question is that we can run computations in parallel if they do not share data. However, we can refine this substantially. Certainly it does not cause a problem if two computations both read the same data from a shared-memory location. Therefore, for data sharing to cause a problem, one of them must write into a memory that the other accesses by either reading or writing. If this is the case, then the order of those memory operations is important. If the sequential program writes into a location in the first computation and then reads from the same location in the second computation, parallelizing the computation might cause the read to be executed first, leading to wrong answers. Such a situation is called a *data race*.

In the 1960s Bernstein formalized a set of three conditions capturing this notion [90]. For the purposes of parallelization these three conditions can be stated as follows: Two computations $\mathbf{C_1}$ and $\mathbf{C_2}$ can be run in parallel without synchronization if and only if none of the following holds:

1. $\mathbf{C_1}$ writes into a location that is later read by $\mathbf{C_2}$ — a *read-after-write* (RAW) race,

2. $\mathbf{C_1}$ reads from a location that is later written into by $\mathbf{C_2}$ — a *write-after-read* (WAR) race, and

3. $\mathbf{C_1}$ writes into a location that is later overwritten by $\mathbf{C_2}$ — a *write-after-write* (WAW) race.

We will see how these conditions can be applied in practice to common programming structures.

Decomposition Strategy Another important task in preparing a program for parallel execution is to choose a strategy for decomposing the program into pieces that can be run in parallel. Generally speaking, there are two ways to do this. First, you could identify the tasks (major phases) in the program and the dependences among them and then schedule those tasks that are not interdependent to run in parallel. In other words, different processors carry out different functions. This approach is known as *task parallelism*. For example, one processor might handle data input from secondary storage, while a second generates a grid based on input previously received.

A second strategy, called *data parallelism*, subdivides the data domain of a problem into multiple regions and assigns different processors to compute the results for each region. Thus, in a 2D simulation on a 1000 by 1000 grid, 100 processors could be effectively used by assigning each to a 100 by 100 subgrid. The processors would then be arranged arranged as a 10 by 10 processor array. Data parallelism is more commonly used in scientific problems because it can keep more processors busy — task parallelism is typically limited to small degrees of parallelism. In addition, data parallelism exhibits a natural form of scalability. If you have 10,000 processors to apply to the problem above, you could solve a problem on a 10,000 by 10,000 cell grid, with each processor still assigned a 100 by 100 subdomain. Since the computation per processor remains the same, the larger problem should take only modestly longer running time than the smaller problem on the smaller machine configuration.

As we shall see, task and data parallelism can be combined. The most common way to do this is to use pipelining, where each processor is assigned to a different stage of a multistep sequential computation. If many independent data sets are passed through the pipeline, each stage can be performing its computation on a different data set at the same time. For example, suppose that the pipeline has four stages. The fourth stage would be working on the first data set, while the third stage would be working on the second data set, and so on. If the steps are roughly equal in time, the pipelining into four stages provides an extra speedup by a factor of four over the time required to process a single data set, after the pipeline has been filled.

Programming Models A second consideration in forming a parallel program is which programming model to use. This decision will affect the choice of programming language system and library for implementation of the application. The two main choices were originally intended for use with the corresponding parallel architectures.

- In the *shared-memory* programming model, all the data accessed by the application occupies a global memory accessible from all parallel proces-

sors. This means that each processor can fetch and store data to any location in memory independently. Shared-memory parallel programming is characterized by the need for synchronization to preserve the integrity of shared data structures.

- In the *message-passing* model, data are viewed as being associated with particular processors, so communication is required to access a remote data location. Generally, to get a datum from a remote memory, the owning processor must *send* the datum and the requesting processor must *receive* it. In this model, send and receive primitives take the place of synchronization.

Although these two programming models arise from the corresponding parallel computer architectures, their use is not restricted. It is possible to implement the shared-memory model on a distributed-memory computer, either through hardware (distributed–shared memory) or software systems such as TreadMarks [23], which simulates a DSM. Symmetrically, message-passing can be made to work with reasonable efficiency on a shared-memory system. In each case there may be some loss of performance.

In the rest of this section, we emphasize the shared-memory model first, because it is easier to understand. However, we will also endeavor to show how the same problem could be coded in each of the two models.

Implementation We now turn to the issues related to the implementation of parallel programs. We begin with data parallelism, the most common form of parallelism in scientific codes. There are typically two sources of data parallelism: iterative loops and recursive traversal of tree-like data structures. We will discuss each of these in turn.

Loops represent the most important source of parallelism in scientific programs. The typical way to parallelize loops is to assign different iterations, or different blocks of iterations, to different processors. On shared-memory systems, this decomposition is usually coded as some kind of `PARALLEL DO` loop. According to Bernstein, we can do this without synchronization only if there are no RAW, WAR, or WAW races between iterations of the loop. Thus we must examine the loop carefully to see if there are places where data sharing of this sort occurs. In the literature on compiler construction, these kinds of races are identified as *dependences* [19]. These concepts can be illustrated by a simple example. Consider the loop:

```
DO I = 1, N
   A(I) = A(I) + C
ENDDO
```

Here each iteration of the loop accesses different locations in the array `A` so there is no data sharing. On the other hand, in the loop:

```
DO I = 1, N
   A(I) = A(I+1) + C
ENDDO
```

there would be a write-after-read race because the element of A being read on any given iteration is the same as the element of A that is written on the next iteration. If the iterations are run in parallel, the write might take place before the read, causing incorrect results.

Thus the main focus of loop parallelization is the discovery of loops that have no races. In some cases, it is possible to achieve significant parallelism in the presence of races. For example, consider:

```
SUM = 0.0
DO I = 1, N
   R = F(B(I),C(I)) ! an expensive computation
   SUM = SUM + R
ENDDO
```

There is a race in this loop involving the variable SUM, which is written and read on every iteration. If we assume that floating-point addition is commutative and associative (which it isn't on most machines) then the order in which results are added to SUM does not matter. Since we assume that the computation of function F is expensive, some gain can still be achieved if we compute the values of F in parallel and then update the SUM in the order in which those computations finish. The thing we must do to make this work is ensure that only one processor updates SUM at a time and each finishes before the next is allowed to begin. On shared-memory systems, *critical regions* are designed to do exactly this. Here is one possible realization of the parallel version:

```
SUM = 0.0
PARALLEL DO I = 1, N
   R = F(B(I),C(I)) ! an expensive computation
   BEGIN CRITICAL REGION
      SUM = SUM + R
   END CRITICAL REGION
ENDDO
```

The critical region ensures that SUM is updated by one processor at a time on a first-come, first-served basis. Because sum reductions of this sort are really important in parallel computation, most system offer a primitive procedure that computes such reductions using a scheme that takes time that is logarithmic in the number of processors.

A programmer who wishes to perform the calculation above on a message-passing system will usually need to rewrite the program in the *single-program, multiple-data (SPMD)* form [210, 451]. In an SPMD program, all of the processors execute the same code, but apply the code to different portions of the data. Scalar variables are typically replicated across the processors in SPMD programs. In addition, the programmer must insert explicit communication primitives in order to pass the shared data between processors. For the sum-reduction calculation above, the SPMD program might look something like this:

```
! This code is executed by all processors
```

```
! MYSUM is a private local variable
! GLOBALSUM is a global collective communication primitive
MYSUM = 0.0
DO I = MYFIRST, MYLAST
   R = F(B(I),C(I)) ! an expensive computation
   MYSUM = MYSUM + R
ENDDO
SUM = GLOBALSUM(SUM)
```

Here the communication is built into the function `GLOBALSUM`, which takes one value of its input parameter from each processor and computes the sum of all those inputs, storing the result into a variable that is replicated on each processor. The implementation of `GLOBALSUM` typically uses a logarithmic algorithm. Explicit communication primitives will be illustrated in the next section in conjunction with pipelined parallelism.

To handle recursive parallelism in a tree-like data structure, the programmer should typically fork a new process whenever it wishes to traverse two different paths down the tree in parallel. For example, a search for a particular value in an unordered tree would examine the root first. If the value were not found, it would fork a separate process to search the right subtree and then search the left subtree itself.

A Simple Example We conclude this section with a discussion of a simple problem that is intended to resemble a finite-difference calculation. We will show how this example might be implemented on both a shared-memory and a distributed-memory machine.

Assume we begin with a simple Fortran code that computes a new average value for each data point in array `A` using a two-point stencil and stores the average into array `ANEW`. The code might look like the following:

```
REAL A(100), ANEW(100)
. . .
DO I = 2, 99
   ANEW(I) = (A(I-1) + A(I+1)) * 0.5
ENDDO
```

Suppose that we wish to implement a parallel version of this code on a shared-memory machine with four processors. Using a parallel-loop dialect of Fortran, the code might look like:

```
REAL A(100), ANEW(100)
. . .
PARALLEL DO I = 2, 99
   ANEW(I) = (A(I-1) + A(I+1)) * 0.5
ENDDO
```

While this code will achieve the desired result, it may not have sufficient granularity to compensate for the overhead of dispatching parallel threads. In

most cases it is better to have each processor execute a block of iterations to achieve higher granularity. In our example, we cam ensure that each processor gets a block of either 24 or 25 iterations by substituting a strip-mined version with only the outer loop parallel:

```
REAL A(100), ANEW(100)
. . .
PARALLEL DO IB = 1, 100, 25
   PRIVATE I, myFirst, myLast
   myFirst = MAX(IB, 2)
   myLast =  MIN(IB + 24, 99)
   DO I = myFirst, myLast
      ANEW(I) = (A(I-1) + A(I+1)) * 0.5
   ENDDO
ENDDO
```

Here we have introduced a new language feature. The `PRIVATE` statement specifies that each iteration of the `IB`-loop has its own private value of each variable in the list. This permits each instance of the inner loop to execute independently without simultaneous updates of the variables that control the inner loop iteration. The example above ensures that iterations 2 through 25 are executed as a block on a single processor. Similarly, iterations 26 through 50, 51 through 75, and 76 through 99 are executed as blocks. This code has several advantages over the simpler version. The most important is that it should have reasonably good performance on a machine with distributed shared memory in which the arrays are stored 25 to a processor.

Finally, we turn to the message-passing version of the code. This code is written in SPMD style so the scalar variables `myP`, `myFirst`, and `myLast` are all automatically replicated on each processor — the equivalent of `PRIVATE` variables in shared memory. In the SPMD style, each global array is replaced by a collection of local arrays in each memory. Thus the 100-element global arrays `A` and `ANEW` become 25-element arrays on each processor named `Alocal` and `ANEWlocal` respectively. In addition, we will allocate two extra storage locations on each processor — `A(0)` and `A(26)` — to hold values communicated from neighboring processors. These cells are often referred to as *overlap areas*.

Now we are ready to present the message-passing version:

```
! This code is executed by all processors
! myP is a private local variable containing the processor number
!     myP runs from 0 to 3
! Alocal and ANEWlocal are local versions of arrays A and ANEW

IF (myP /= 1) send Alocal(1) to myP-1
IF (myP /= 4) send Alocal(25) to myP+1
IF (myP /= 1) receive Alocal(0) from myP-1
IF (myP /= 4) receive Alocal(26) from myP+1
```

```
myFirst = 1
myLast = 25
IF (myP == 1) myFirst = 2
IF (myP == 4) myLast = 24
DO I = myFirst, myLast
   ANEWlocal(I) = (Alocal(I-1) + Alocal(I+1)) * 0.5
ENDDO
```

Note that the computation loop is preceded by four communication steps in which values are sent to and received from neighboring processors. These values are stored into the overlap areas in each local array. Once this is done, the computation can proceed on each of the processors using the local versions of A and ANEW.

As we shall see later in the book, the performance can be improved by inserting a purely local computation between the sends and receives in the above example. This is an improvement because the communication is overlapped with the local computation to achieve better overall parallelism. The following code fragment inserts the computation on the interior of the region before the receive operations, which are only needed for computing the boundary values.

```
! This code is executed by all processors
! myP is a private local variable containing the processor number
!     myP runs from 0 to 3
! Alocal and ANEWlocal are local versions of arrays A and ANEW

IF (myP /= 1) send Alocal(1) to myP-1
IF (myP /= 4) send Alocal(25) to myP+1
DO I = 2, 24
   ANEWlocal(I) = (Alocal(I-1) + Alocal(I+1)) * 0.5
ENDDO
IF (myP /= 1) THEN
   receive Alocal(0) from myP-1
   ANEWlocal(1) = (Alocal(0) + Alocal(2)) * 0.5
ENDIF
IF (myP /= 4) THEN
   receive Alocal(26) from myP+1
   ANEWlocal(25) = (Alocal(24) + Alocal(26)) * 0.5
ENDIF
```

2.3.1 Scalability and Load Balance

The idealized goal of parallel computation is to have the running time of an application reduced by a factor that is inversely proportional to the number of processors used. That is, if a second processor is used, the running time should be half of what is required on one processor. If four processors are used, the running time should be a fourth. Any application that achieves this goal is said

to be *scalable*. Another way of stating the goal is in terms of *speedup*, which is defined to be the ratio of the running time on a single processor to the running time on the parallel configuration. That is,

$$Speedup(n) = T(1)/T(n) \tag{2.1}$$

An application is said to be scalable if the speedup on n processors is close to n. Scalability of this sort has its limits — at some point the amount of available parallelism in the application will be exhausted and adding further processors may even detract from performance.

This leads us to consider a second definition of scalability, called *scaled speedup* — an application will be said to be scalable if, when the number of processors and the problem size are increased by a factor of n, the running time remains the same [365]. This captures the notion that larger machine configurations make it possible to solve correspondingly larger scientific problems.

There are three principal reasons why scalability is not achieved in some applications. First, the application may have a large region that must be run sequentially. If we assume that T_S is the time required by this region and T_P is the time required by the parallel region, the speedup for this code is given by:

$$Speedup(n) = \frac{T_S + T_P}{T_S + \frac{T_P}{n}} \leq \frac{T(1)}{T_S} \tag{2.2}$$

This means that the total speedup is limited by the ratio of the sequential running time to the running time of the sequential region. Thus if 20 percent of the running time is sequential, the speedup cannot exceed 5. This observation is known as Amdahl's Law [22].

A second impediment to scalability is the requirement for a high degree of communication or coordination. In the global summation example above, if the computation of the function `F` is fast, then the cost of the computation is dominated by the time required to take the sum, which is logarithmic at best. This can be modeled to produce a revised speedup equation:

$$Speedup(n) = \frac{T(1)}{\frac{T(1)}{n} + \lg(n)} = \frac{n}{1 + \frac{\lg n}{T(1)}} \tag{2.3}$$

Eventually, the logarithmic factor in the denominator will grow to exceed the sequential running time and hence reduce scalability.

The third major impediment to scalability is poor *load balance*. If one of the processors takes half of the parallel work, speedup will be limited to a factor of two, no matter how many processors are involved. Thus a major goal of parallel programming is to ensure good load balance.

If all the iterations of a given loop execute for exactly the same amount of time, load balance can be achieved by giving each processor exactly the same amount of work to do. Thus the iterations could be divided into blocks of equal number, so that each processor gets roughly the same amount of work, as in the following example:

```
K = CEIL(N/P)
PARALLEL DO I = 1, N, K
   DO ii = I, MAX(I+K-1,N)
      A(I) = A(I+1) + C
   ENDDO
ENDDO
```

However, this strategy fails if the work on each iteration takes a variable amount of time. On shared-memory machines, this can be ameliorated by taking advantage of the way parallel loops are scheduled. On such machines, each processor goes back to the queue that hands out iterations when it has no work to do. Thus by reducing the amount of work on each iteration (while keeping it above threshold) and increasing the total number of iterations, we can ensure that other processors take up the slack for a processor that has a long iteration. Using the same example:

```
K = CEIL(N/(P*4))
PARALLEL DO I = 1, N, K
   DO ii = I, MAX(I+K-1,N)
      A(I) = A(I+1) + C
   ENDDO
ENDDO
```

On average, each processor should execute four iterations of the parallel loop. However, if one processor gets stuck, the others will take on more iterations to naturally balance the load.

In cases where neither of these strategies is appropriate, such as when the computer is a distributed-memory message-passing system or when load is not known until run time, a dynamic load-balancing scheme may be required, in which the assignment of work to processors is done at run time. Hopefully, such a load-balancing step will be done infrequently so that the cost is amortized over a number of computation steps.

2.3.2 Pipeline Parallelism

To this point we have been primarily dealing with parallelism that is *asynchronous* in the sense that no synchronization is needed during parallel execution. (The exception was the summation example, which required a critical section.) Ideally, we should always be able to find asynchronous parallelism, because this gives us the best chance for scalability. However, even when this is not possible, some parallelism may be achievable. To see this, consider the following variant of successive over-relaxation:

```
DO J = 2, N
   DO I = 2, N
      A(I,J) = (A(I-1,J) + A(I+1,J))*0.5
   ENDDO
ENDDO
```

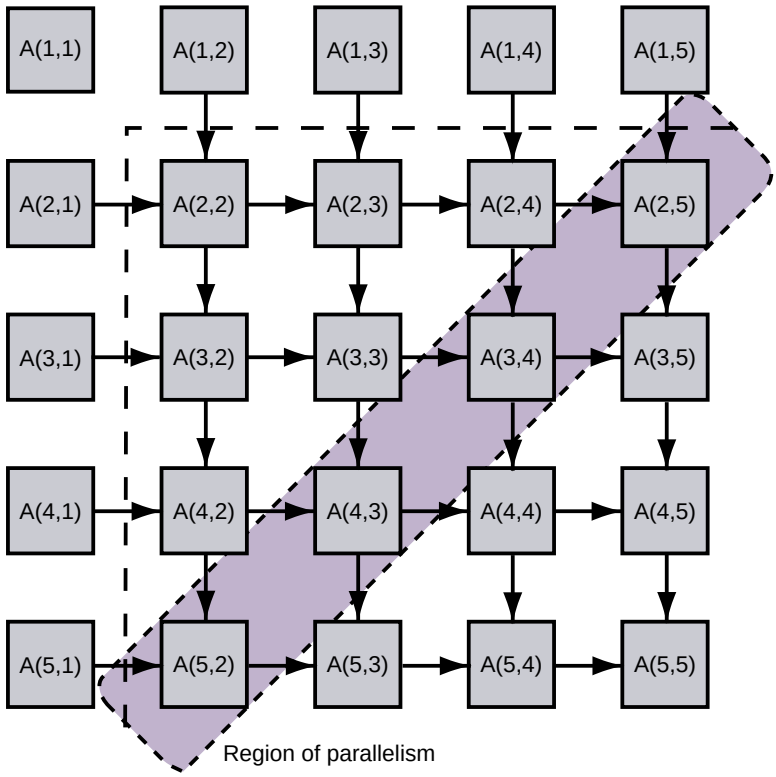

Figure 2.4: Wavefront Parallelism

Although neither of the loops can be run in parallel, there is some parallelism in this example, as is illustrated in Figure 2.4. All of the values on the shaded diagonal can be computed in parallel because there are no dependences.

Suppose, however, that we wish to compute all the elements in any column on the same processor, so that `A(*,I)` would be computed on the same processor for all values of `I`. If we compute the elements in any column in sequence, all of the dependences along that column are satisfied. However, we must still be concerned about the rows. To get the correct result, we must delay the computation on each row by enough to ensure that the corresponding array element on the previous row is completed before the element on the current row is computed. This strategy, known as *pipelining*, can be implemented via the use of events, as demonstrated in the following pseudocode:

```
EVENT READY(N,N) ! Initialized to false
PARALLEL DO I = 1, N
   POST(READY(I,1))
ENDDO
PARALLEL DO J = 2, N
```

```
      DO I = 2, N
         WAIT(READY(I-1,J))
         A(I,J) = (A(I-1,J) + A(I+1,J))*0.5
         POST(READY(I,J))
      ENDDO
   ENDDO
```

Initially all the events are false — a wait on a false event will suspend the executing thread until a post for the event is executed. All of the `READY` events for the first column are then posted so the computation can begin. The computation for the first computed column, `A(*,2)`, begins immediately. As each of the elements is computed, its `READY` event is posted so the next column can begin computation of the corresponding element. The timing of the computation is illustrated in Figure 2.5. Note that the event posting has aligned the region of parallelism so that all processors are simultaneously working on independent calculations.

2.3.3 Regular versus Irregular Problems

Most of the examples we have used in this chapter are regular problems — defined on a regular, fixed grid in some number of dimensions. Although a large fraction of scientific applications focus on regular problems, a growing fraction of applications address problems that are irregular in structure or use adaptive meshes to improve efficiency. This means that the structure of the underlying grid is usually not known until run time. Therefore, these applications present special difficulties for parallel implementation because static, compile-time methods cannot be used to perform load balancing and communication planning.

To illustrate these issues we will present a code fragment from a simple force calculation that might be part of a molecular dynamics code.

```
DO I = 1, NPAIRS
   F(L1(I) = F(L1(I)) + FORCE(X(L1(I)), X(L2(I)))
   F(L2(I) = F(L2(I)) + FORCE(X(L2(I)), X(L2(I)))
ENDDO
DO I = 1, NPART
   X(I) = MOVE(X(I), F(I))
ENNDO
```

Here the first loop is intended to traverse a list of particle pairs where the two particles in the pair are located at index `L1(I)` and index `L2(I)`, respectively. In molecular dynamics codes these pair lists are often constructed by taking every pair of particles that are within some cutoff distance of one another. For each pair, the force arising from the particle interaction (due to electromagnetic interaction) is calculated by function `FORCE` and added to an aggregate force for the particle. Finally the aggregate forces are used to calculate the new location for each particle, represented by `X(I)`.

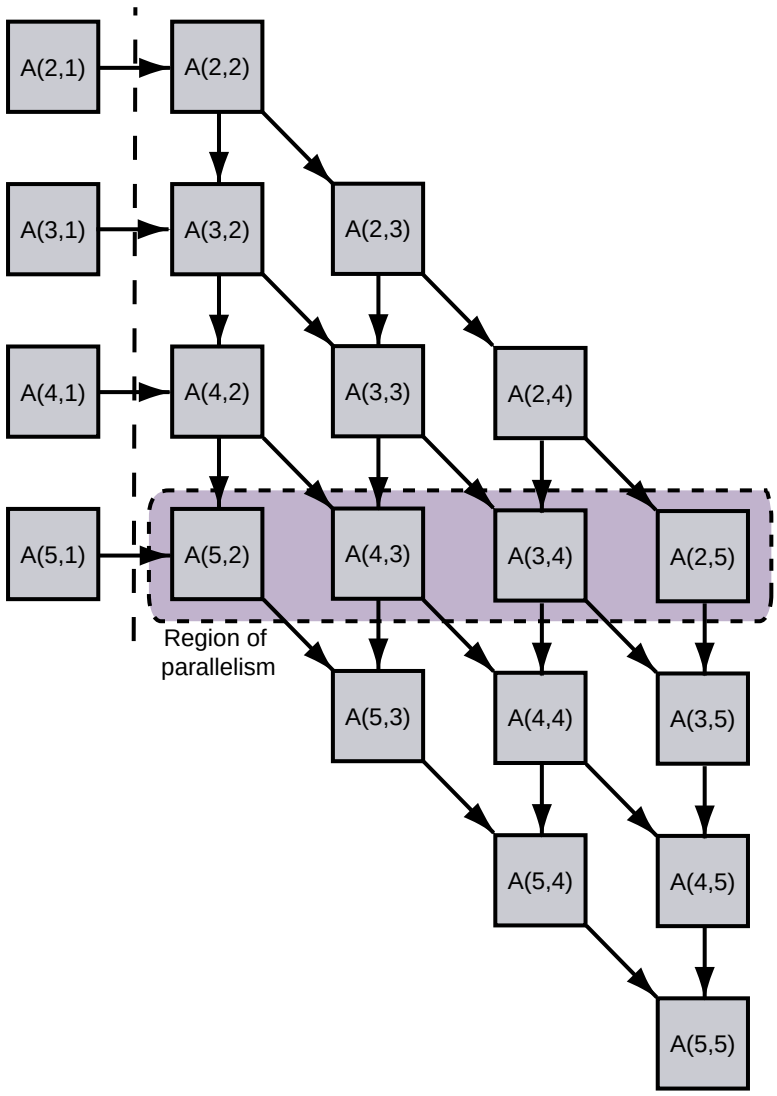

Figure 2.5: Wavefront Parallelism

This code illustrates a fundamental characteristic of irregular problems, namely the use of subscripted index variables in the index positions of the fundamental data quantities of the problem. These index arrays are computed at run time so the actual data locations of the quantities involved in calculations in the first loop are not known until the pair list is constructed at run time. Therefore, load balancing cannot take place until the pair list is constructed. Furthermore, if communication is required, the optimization of communication must be postponed until run time as well.

To address this issue, a standard approach is to perform three steps as execution begins:

1. Read in all the data.
2. Perform load balancing by distributing data to different processors and possibly reorganizing data within a processor.
3. Step through the computation loop (without performing a computation) to determine a communication schedule, if one is needed. Schedule, if required.

This last step is known as the *inspector* because its goal is to inspect the calculation to plan communication.

In the force calculation above, the goal of load balancing would be to organize the layout of particles and force pairs so that the maximum number of particle interactions are between particles on the same processor. There are many ways to approach this problem. As an example, we will describe a simple but fairly effective strategy that uses *Hilbert curves*, often called *space-filling curves*, to lay out the data. A Hilbert curve traces through 1D, 2D or 3D space in an order that ensures that particles that are close together on the curve are usually close together in space. The particles can then be ordered in memory by increasing position on the Hilbert curve, with an equal number of particles allocated to each processor. Pairs can be allocated to processors so that it is likely that one element of the pair will reside on the processor where the pair resides. In many cases this can be accomplished by some form of lexicographic sort applied to the pairs [576]. Finally, the pairs requiring communication can be determined by an inspector that steps through the pair list to see if one of the elements of the pair is on another processor. All of the communication from the same processor can then be grouped and the data delivered in a block from each processor at the beginning of every execution step.

Although this discussion is much oversimplified, it should give the flavor of the strategies used to parallelize irregular problems. There is one further complication worth mentioning, however. In the example above, the pair list may need to be reconstructed from time to time, as the particles move around and some drift out of the cutoff area while others drift in. When this happens, it may be necessary to reorganize the data, rebalance the load, and invoke the inspector once again. However, this should be done as seldom as possible to ensure that the potentially high cost of these steps is amortized over as many execution steps as possible.

2.4 Memory-Hierarchy Management

In this section we discuss programming strategies that can help make optimal use of the memory hierarchy of a modern parallel computer system. We begin with the strategies that improve the performance of a uniprocessor node within the memory and then proceed to the issues that are complicated by parallelism.

Uniprocessor Memory-Hierarchy Management The critical issue in getting good memory-hierarchy performance on a uniprocessor is achieving high degrees of reuse of data in both registers and cache memory. Many programmers are surprised to find that proper organization of their programs can dramatically affect the performance that they achieve.

There are three principal strategies available to programmers for improving the performance of memory hierarchy:

- *Stride-one access.* Most cache memories are organized into blocks of data much longer than a single data item. For example, the level-2 cache block on the SGI Origin can hold 16 double precision floating point numbers. On every machine, these numbers are at contiguous addresses in memory. If a program is arranged to iterate over successive items in memory, it can suffer at most one cache miss for every cache block. All successive data items in the cache block will be hits. Thus, programs in which the loops access contiguous data items are typically much more efficient than those that do not.

- *Blocking.* Program performance can also be improved by ensuring that data remains in cache between subsequent accesses to the same memory location. As an example, consider the following code:

```
DO I = 1, N
   DO J = 1, N
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

  Although this loop achieves a high degree of reuse for array A, missing only N/L times where L is the size of the cache block, it has a dismal performance on array B, on which it incurs N^2/L misses. The problem is that, even though access to B is stride-one, B(J) cannot remain in cache until its next use on the next iteration of the outer loop. Therefore, all N/L misses will be incurred on each iteration of the outer loop. If the loop on J is "blocked" to a size where all the elements of B that it touches can remain in cache, then the following loop results:

```
DO J = 1,N,S
   DO I = 1, N
      DO j = J, min(J+S,N)
         A(I) = A(I) + B(J)
      ENDDO
   ENDDO
ENDDO
```

where S is the maximum number of elements of B that can remain in cache between two iterations of the outer loop.

In this new organization, we suffer at most N/L misses for B, because each element of B is reused N times. On the other hand, we increase the number of misses on A to $N^2/(LS)$ because we must now miss for all the elements of A on each of the N/S iterations of the outer loop. In this example, we have strip-mined or blocked the inner loop to the size of the cache and moved the iterate-by-strip loop to the outermost loop position.

- *Data reorganization.* A third strategy for improving the behavior of a uniprocessor memory hierarchy is to reorganize the data structures so that data items that are used together are stored together in memory. For example, many older Fortran programs use multidimensional arrays to store complex data structures. In these programs one often sees an array declaration such as:

```
DOUBLE PRECISION PART(10000,5)
```

Here the second dimension is being used to store the fields of a data structure about one of the 10,000 particles in a simulation. If the fields describing a single particle are updated together, this is the wrong data organization because Fortran uses column-major order and the 5 fields are likely to appear on different cache lines. This organization can reduce the effective bandwidth in the program by up to a factor of 5. A much better organization for Fortran is to swap the data dimensions:

```
DOUBLE PRECISION PART(5,10000)
```

However, this requires rewriting every access to the array PART in order to ensure correctness. Thus this task is often best left to a tool, although the programmer should be aware of this problem while writing the program initially. Data reorganization is also very effective on irregular problems, even though the reorganization must take place at run time. In irregular particle codes, the cost of misses due to a bad data organization far outweighs the cost of rearranging the data at run time.

Judicious use of these strategies can improve performance by integer factors. However, they are tedious to apply by hand, so many of these strategies have been built into modern Fortran compilers.

Multiprocessor Memory Hierarchies Multiprocessors add a number of complexities to the problem of managing accesses to memory and improving reuse. In this section, we will concentrate on three of the most significant problems.

- *Synchronization.* In many parallel programs it is useful to have several different processors updating the same shared data structure. An example is a particle-in-cell code where the forces on a single particle are computed by a number of different processors, each of which must update the aggregate force acting on that particle. If two processors attempt two different updates simultaneously, incorrect results may occur. Thus it is essential to use some sort of locking mechanism, such as a semaphore or a critical region, to ensure that when one processor is performing such an update on a given particle, all other updates for the same particle are locked out. Typically, processors that are locked out execute some sort of busy–waiting loop until the lock is reset. Most machines are designed to ensure that these loops do not cause ping-ponging of the cache block containing the lock.

- *Elimination of False Sharing.* False sharing is a problem that arises when two different processors are accessing data that resides on the same cache block. On a shared-memory machine, if both processors attempt to write into the same block, the block can ping-pong back and forth between those processor caches. This phenomenon is known as false sharing because it has the effect of repeated access to a shared datum even though there is no sharing. False sharing is typically avoided by ensuring that data used by different processors resides on different cache blocks. This can be achieved by the programmer or a compiler through the use of *padding* in data structures. Padding is the process of inserting empty bytes in a data structure to ensure that different elements are in different cache blocks.

- *Communication Minimization and Placement.* Communication with a remote processor can have a number of negative effects on the performance of a computation node in a parallel machine. First, the communication itself can cause computation to wait. The typical strategy for addressing this problem is to move send and receive commands far enough apart so that the time spent on communication can be overlapped with computation. Alternatively, a program reorganization may reduce the frequency of communication, which not only reduces the number of start-up delays that must be incurred but also reduces the interference with local memory-hierarchy management. A form of blocking can be useful in this context — if large data structures are being cycled through all the processors of a distributed-memory computer, it pays to block the data so that all computations involving that data structure by one processor can be carried out at the same time so that each block has to be communicated to a given processor exactly once.

Once again, many of the useful strategies can be automated in a compiler.

2.5 Parallel Debugging

Parallel debugging is the process of ensuring that a parallel program produces correct answers. We will say that it produces correct answers if it satisfies two

criteria:

1. *Absence of Nondeterminism.* It always produces the same answers on the same inputs.

2. *Equivalence to the Sequential Version.* It produces the same answers as the sequential program on which it is based.

These criteria are based on two underlying assumptions. First, we assume that a parallel program will typically be developed by producing and debugging the sequential version and then converting it to use parallelism. In this model the sequential program becomes a specification for the desired answers.

Second, we assume that nondeterminism is not a desirable property. Although there is much discussion in the literature of using nondeterminism in programming, our experience is that most scientific users want repeatability in their codes (except of course for Monte Carlo codes and the like). Because the sequential program is almost always equivalent to the parallel program run on one processor, we will concentrate on the goal of eliminating nondeterminism.

In shared-memory programming models, the principal sources of nondeterminism are *data races*. A data race occurs when different iterations of a parallel loop share data, with one iteration writing to the shared location. As an example, consider the following loop.

```
PARALLEL DO I = 1, N
   A(I) = A(I+5) + B(I)
ENDDO
```

Even though this loop can be vectorized, it has a data race because if iteration 6 gets far enough ahead of iteration 1, it might store into `A(6)` before the value is loaded on iteration 1. This would produce wrong answers because when `I` = 1, the sequential version reads the value of `A(6)` as it is on loop entry.

Data races are often difficult to detect because they do not show up on every execution. Thus tools are typically required to detect them. One strategy that can be used to uncover many races is to run all the parallel loops in a program backward and forward sequentially and compare the answers. Although this is not guaranteed to find all the races, it can uncover the most common ones.

A number of sophisticated tools have been developed or proposed to detect data races. These generally fall into two classes:

1. *Static analysis tools*, which use the analysis of dependence from compiler parallelization to display potential data races in parallel loops. An example from CRPC is the ParaScope Editor [59].

2. *Dynamic analysis tools*, which use some sort of program replay with shadow variables to determine if a race might occur at run time.

In message-passing programs, the most common parallel bugs arise from messages that arrive out of order. When most message-passing programs execute receive operations from a given processor, the programmer expects that

the message would be one that came from a particular send. However, if more than one message is being sent between the same pair of processors, they might arrive out of order, leading to nondeterministic results. For this reason, many message-passing libraries use "tags" to ensure that send and receive pairs match. A tag can be thought of as specifying a specific channel on which messages are to be watched for. Since there can be more than one channel between the same pair of processors, this can be used to ensure that message-out-of-order bugs do not occur.

Another problem in message-passing programs arises because it is possible to execute a receive of a message that never arrives. This can happen, for example, if the receive is always executed but the send is executed only under certain conditions. A somewhat symmetric bug occurs when more messages are sent than are received, due to mismatching conditions. This can cause messages to never be received, with resulting wrong (or at least surprising) answers. Problems of this sort can usually be detected by analyzing traces of communication operations.

2.6 Performance Analysis and Tuning

Because the primary goal of parallel computing is to obtain higher performance than is possible via sequential computation, optimizing parallel application behavior is an integral part of the program development process. This optimization requires knowledge of the underlying architecture, the application code parallelization strategy, and the mapping of the application code and its programming model to the architecture.

The basic performance tuning cycle consists of four steps:

- *Automatic or manual instrumentation.* This instrumentation typically inserts measurement probes in application code and system software, perhaps with additional software measurement of hardware performance counters.

- *Execution of the instrumented application and performance data.* Such executions record hardware and software metrics for offline analysis. The recorded data may include profiles, event traces, hardware counter values, and elapsed times.

- *Analysis of the captured performance data.* Using recorded data, analysis, either manual or automatic, attempts to relate measurement data to hardware resources and application source code, identifying possible optimization points.

- *Modification of the application source code, recompilation with different optimization criteria, or modification of runtime system parameters.* The goal of these modifications is to better match application behavior to the hardware architecture and the programming idioms for higher performance.

As a concrete example, consider an explicitly parallel message-passing code, written in C or Fortran 77 and intended for execution on a distributed-memory parallel architecture (e.g., a Linux PC cluster with a 100 Mb/second Ethernet interconnect). A detailed performance instrumentation might include (a) use of a profiler to estimate procedure execution times, (b) recording of hardware instruction counters to identify operation mixes and memory-access costs, and (c) use of an instrumented version of MPI to measure message-passing overhead.

A profile, based on program-counter sampling to estimate execution times, typically identifies the procedures where the majority of time is spent. Examining the program's static call graph often suggests the invocation pattern responsible for the overhead (e.g., showing that inlining the body of a small procedure at the end of a call chain in a loop nest would reduce overhead).

If a procedure profile is insufficient, hardware counter measurements, when associated with loop nests, can identify the types and numbers of machine instructions associated with each loop nest. For example, seeing memory reference instruction stall counts may suggest that a loop transformation or reblocking would increase cache locality.

Finally, analysis and visualization of an MPI trace (e.g., via Jumpshot) may suggest that the computation is dominated by the latency associated with transmission of many small messages. Aggregating data and sending fewer, larger messages may lead to substantially higher performance.

The problems for data parallel or implicitly parallel programs are similar, yet different. The range of possible performance remedies differs, based on the programming model (e.g., modifying array distributions for better memory locality), but the instrumentation, execution, and analysis, and code optimization steps remain the same.

Perhaps most critically, the common theme is the need to intimately understand the relations among programming model, compiler optimizations, runtime system features and behavior, and architectural features. Because performance problems can arise at any point in the multilevel transformations of user-specified application code that precede execution, one cannot expect to obtain high performance without investing time to understand more than just application semantics.

2.7 Parallel Input/Output

Most parallel programs do more than just compute (and communicate): they must also access data on secondary storage systems, whether for input or output. And precisely because parallel computations can execute at high speeds, it is often the case that parallel programs need to access large amounts of data. High-performance I/O hence becomes a critical concern. On a parallel computer, that inevitably means *parallel* I/O; without parallelism we are reduced to reading and writing files from a single processor, which is almost always guaranteed to provide only low performance. That is, we require techniques that can allow many processors to perform I/O at the same time, with the goal of exploiting parallelism in the parallel computer's communication network and I/O system.

The parallel programmer can take two different approaches to achieving concurrency in I/O operations. One approach is for each process to perform read and write operations to a distinct file. While simple, this approach has significant disadvantages: programs cannot easily be restarted on different numbers of processors, the underlying file system has little information on which to base optimization decisions, and files are not easily shared with other programs. In general, it is preferable to perform true parallel I/O operations, which allow all processes to access a single shared file.

The parallel I/O problem is multifaceted and often quite complex. This is due to the need to deal with issues at multiple levels, including the I/O architecture of the parallel computer (e.g., each compute processor can have a local disk, or disks can be managed by distinct I/O processors), the file system that manages access to this I/O architecture, the high-level libraries that may be provided to map application-level I/O requests into file system operations, and the user-level API(s) used to access lower-level functions. Fortunately, after much research, the community has succeeded in developing a standard parallel I/O interface, namely the parallel I/O functions included in the MPI-2 standard. (These are sometimes also referred to as MPI-IO.) These functions are supported on most major parallel computer platforms in the form of a vendor-supported library and/or the public-domain ROMIO package developed at Argonne National Laboratory.

MPI-IO functions enable a set of processes to open, read, write, and eventually close a single shared file. The read and write functions and many of the read and write functions are collective, meaning that all processes call them together; in the case of read and write operations, each process contributes part of the data that is to be read (or written). This use of collective operations allows the underlying I/O library and file system to perform important optimizations: for example, they can reorganize data prior to writing it to disk.

The following example gives the flavor of the MPI-IO interface. This code fragment first opens a file for read-only access and then calls the collective I/O function `MPI_File_read_all`. Each calling process will obtain a piece of the file in its *local_array*.

```
MPI_File_open(MPI_COMM_WORLD, ''/pfs/datafile'', MPI_MODE_RDONLY,
       MPI_INFO_NULL, &fh);
...
MPI_File_read_all(fh, local_array, local_array_size,
     MPI_FLOAT, &status);
```

A detailed discussion of parallel I/O techniques and MPI-IO is provided in Chapter 11.

2.8 Chapter Summary

This chapter has presented an introductory treatment of a number of the strategies and issues that a new parallel programmer must deal with, including programming models and strategies, application partitioning, scalability, pipelin-

ing, memory hierarchy management, irregular versus regular parallelism, parallel debugging, performance tuning, and parallel I/O. These topics will be discussed in more detail in the remainder of this book. In addition they will be tied to specific application programming interfaces, such as languages and runtime systems.

2.9 Further Reading

â€¢ Parallel programming is more difficult than its sequential counterpart. â€¢ However we are reaching limitations in uniprocessor design. â€¢ Physical limitations to size and speed of a single chip â€¢ Developing new processor technology is very expensive â€¢ Some fundamental limits such as speed of light and size of atoms. â€¢ Parallelism is not a silver bullet. â€¢ There are many additional considerations â€¢ Careful thought is required to take advantage of parallel machines. Performance. â€¢ A key aim is to solve problems faster. parallel-processing dynamic-programming program-transformation. Share.Â We recently published a paper showing how to parallelize any d.p. on a shared memory multicore computer by means of a shared lock-free hash table: Stivala, A. and Stuckey, P. J. and Garcia de la Banda, M. and Hermenegildo, M. and Wirth, A. 2010 "Lock-free parallel dynamic programming" J. Parallel Distrib. Comput. 70:839-848 doi:10.1016/j.jpdc.2010.01.004. Introduction to parallel programming. The past few decades have seen large uctuations in the perceived value of parallel computing. At times, parallel computation has optimistically been viewed as the solution to all of our computational limitations.Â Many parallelizing compilers, including those named above, take an intermediate ap-proach in which programmers add directives to their codes to provide the compiler with information to aid it in the task of parallelizing the code. Parallel programming is a very wide topic. It depends on what you mean by parallel. So you want concurrent multi-threading on single core CPU, multi-threading on multi-core SMP CPU or across a server farm or on GPU?Â , Ph.D. Computer Science, Rensselaer Polytechnic Institute Â· Author has 3.5K answers and 14.8M answer views. Since hardware instructions are ultimately done in parallel, how does it work for programming languages to be sequential? First, a picture. I promise itâ€™s relevant.