CHAPTER 1

Fundamentals of Compiling for Parallel Machines

1.1 Introduction to Parallel Architectures

Computational speed has increased at an astounding rate over the past few decades. The first electronic digital computer, the Electronic Numerical Integrator and Computer (ENIAC) was developed in 1946, and was able to process roughly 5000 computations per second. Today, slightly less than 50 years later, computers capable of gigaflop (billions of floating point operations per second) performance are just becoming available. Single-processor workstations that can perform in excess of 25 million computations per second are considered routine. In other words, the speed at which computations can be done has grown by nearly four orders of magnitude in half a century.

This increase in computation power has been fueled primarily by changing processor technology. The ENIAC and its peers relied on electromechanical relays; this technology was followed by vacuum tubes, transistors, integrated circuits and microprocessors. Current supercomputers employ gallium arsenide; future supercomputers appear destined to use optical circuits.

While technology improvements have fueled a phenomenal growth in computing power, they have simultaneously delayed the introduction of some natural architectural schemes such as parallelism. Technology has improved so rapidly that the performance of a single processor of a new technology has been competitive with that of multiple processors of a slightly older technology. Since multiple processors have additional overhead in the form of communication hardware, parallelism has been fully exploited only at the high end of the processing spectrum, where the demand for computing cycles is never fully satisfied.

However, this situation appears to be changing. Processing technology seems to be approaching a plateau, with much less dramatic gains in speeds for single processors. As a result, parallelism has become a significant factor in the marketplace. Parallelism is no longer an element solely of the fastest supercomputers, but instead has become an integral part of minisupercomputers and workstations.

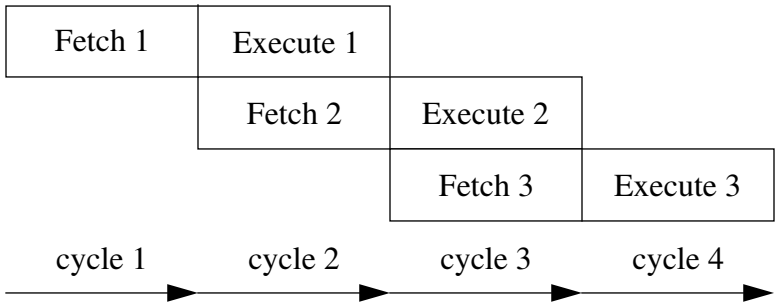
The key technical barrier to widespread acceptance of parallelism is the problem of programming parallel machines. Parallel architectures are harder to program because they are more complicated, with many components (such as processors, memory hierarchies and communications) that must be managed. Obviously, if some or all of these components can be managed automatically (say, via a compiler) rather than by a programmer, parallel architectures are much more likely to gain acceptance. In this section, we will survey the predominate forms of parallel architectures. This survey will include discussion of the complexities associated with parallel architectures, and the features required by a compiler if it is to be able to successfully manage these complexities.

1.1.1 Intra-functional unit parallelism

There are two principal ways by which parallelism has been exploited in hardware. The first is through the use of *pipelining*—subdividing a complex operation into a sequence of several independent segments. If the different segments all use different resources, they can be overlapped by

starting a second operation as soon as the first has completed the first segment.

FIGURE 1.1 Fetch-execute overlap on the IBM 7094.



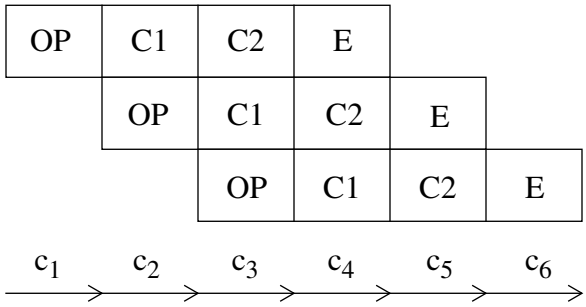
Consider as an example one of the earliest pipelines, the *instruction execution pipeline*. On the IBM 7094, a high-end scientific computer from the mid nineteen-sixties, each instruction was executed in two phases: the *instruction fetch* phase and the *execute* phase. Each phase typically required a memory access, since the execute phase almost always fetched a datum from memory to act as one of the operands. The innovation of the IBM 7094 over previous machine models was to exploit a memory organized into two banks. The system was designed so that the processor could fetch a single word from each bank on each machine cycle. If data and instructions were stored in different banks, it could overlap the execute phase of one instruction with the instruction-fetch phase of the next, as depicted in Figure 1.1.

The notion of an instruction execution pipeline was further refined in the 1970s by the introduction, on the IBM 370 series, of a pipelined execute phase consisting of four stages:

1. decode and compute effective address (OP),
2. compute physical address for cache access (C1)
3. fetch data from cache (C2) and
4. execute operation (E).

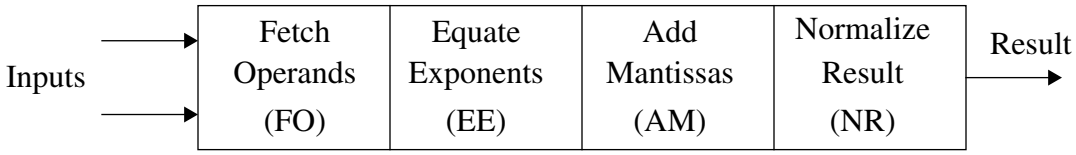
Since each of these phases makes use of different hardware resources, they can be pipelined as depicted in Figure 1.2. If no operation takes more than one cycle to execute, an instruction can be started every clock, reducing the cycle time to a fraction of that in simple pipelines such as the one on the IBM 7094.

FIGURE 1.2 IBM 370 Instruction Execution Pipeline.



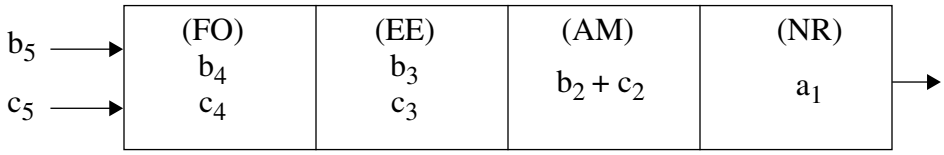
Unfortunately, many operations will take much longer than a single one of these shortened cycles. Most notable are the floating point operations, which may take four or more cycles to complete. Consider, for example, the steps in a typical floating point adder. Once both operands have been fetched from memory, they must be normalized to have the same exponent. Next, the two mantissas must be added together. Finally, the resulting addition may have to be renormalized before being stored into its destination. Figure 1.3 illustrates this process.

FIGURE 1.3 Typical floating-point adder.



Since each segment in the addition unit is independent of the others, there is no reason why each segment cannot be simultaneously operating on different operands. Thus, if it is necessary to compute the sum of several pairs of numbers, the computation can be accelerated by overlapping the phases of the addition unit, as illustrated in Figure 1.4.

FIGURE 1.4 Snapshot of a pipelined execution unit computing $a_i=b_i+c_i$.



If the time required by each segment to work on one pair of operands is one cycle, then the time required to perform *n* additions without pipelining is 4*n* cycles. If the segments are overlapped so that the computation is pipelined, the *n* additions require *n*+3 cycles, because once the pipe has been filled, the adder produces a result every cycle. Thus, for an infinite number of computations, the time to perform one addition is effectively reduced from 4 cycles to 1 cycle by pipelining.

A pipelined functional unit is effective only when the pipe is kept full; that is, only when there are operands available for operation on each segment clock cycle. Unfortunately, user computations rarely satisfy this requirement. As a result, effectively utilizing a pipelined functional unit requires that something (usually the compiler) reorder the user computation so that the necessary operands can be delivered fast enough to keep the pipe full.

1.1.2 Inter-functional unit parallelism

Pipelining speeds up the execution of a single function unit. Orthogonally, multiple processors or functional units can be applied to different parts of a computation to reduce the overall computation time. Thus, if four addition units are available, each taking four cycles, the computation described in the previous section can be performed in the same amount of time as with a pipelined unit, assuming that operations can be issued no faster than one per cycle (Figure 1.5).

There are three basic forms of parallelism that are commonly found in practice, listed below in order of decreasing simplicity.

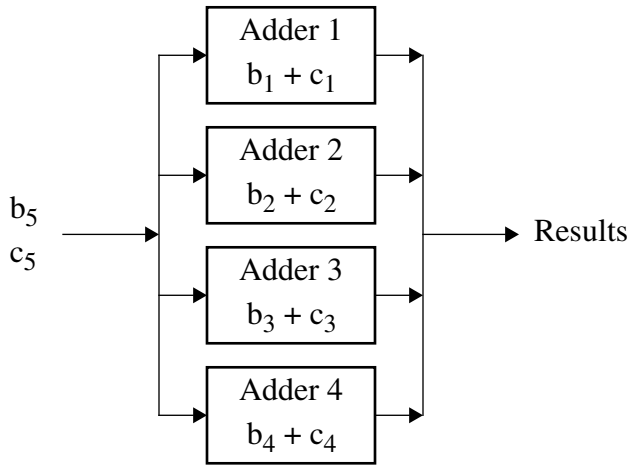
1. *Functional Unit Parallelism.* By replicating functional units, instructions can be issued to more than one functional unit, as depicted in Figure 1.5. When the execution unit issues an instruction, it sends it to a free functional unit if one exists. Otherwise it waits for a functional

unit to become free. If there are *n* functional units, each of which takes *m* cycles to complete its operation, the machine is capable of issuing *n*/*m* operations per cycle on average. This type of parallelism, also known as *fine-grained parallelism*, may appear more attractive than pipelining at first glance, since it permits a wider range of operational freedom. However, there are extra costs. First, a pipelined unit costs only a bit more than an nonpipelined unit[1], whereas multiple functional units cost much more than single functional units. Second, functional unit parallelism requires significantly more sophisticated execution logic than pipelined parallelism. Of course, it is possible to use both pipelining and multiple functional units.

2. *Synchronous Processor Parallelism.* Another approach to parallelism is to replicate whole processors, with each processor executing the same program on different portions of the data space. Examples of this type of parallel system are the Thinking Machines CM-2, the MasPar MP-2 and the AMT DAP. The main advantage of a synchronous system is that synchronization operations are cheap, because the instructions are executed in lock step. Thus these machines are capable of exploiting a much finer granularity of parallelism. On the other hand, synchronous machines are not very efficient at executing code with many branches because they must essentially execute both sides of every branch, with different processors disabled on each side.

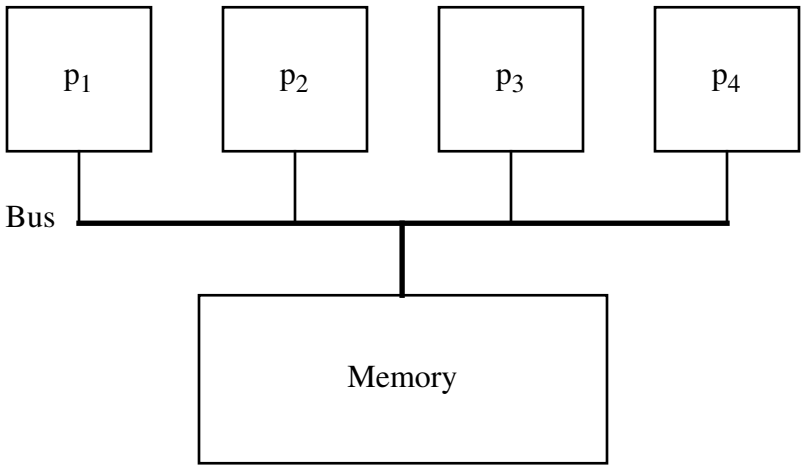
1. Seymour Cray discovered this in the 1960s between the design for the Control Data 6600 (circa 1965), which used multiple units and the Control Data 7600 (circa 1969), which used pipelining.

FIGURE 1.5 Multiple functional units.



3. *Asynchronous Processor Parallelism.* Another form of parallelism is to replicate whole processors, but allow each processor to execute different programs or different parts of the same program. A depiction of an asynchronous machine with a shared global memory is given in Figure 1.6. This design is typical of multiprocessors available from Convex, Cray, IBM, Sequent, Silicon Graphics and Sun Microsystems. If synchronization between processors is required on these machines, it must be explicitly specified. The problem with these machines lies in the cost of starting up a parallel execution. Processes must be spawned for each processor and the processors must be synchronized before accessing any shared data. Because of this, it is essential that parallel execution be used only if there is enough work to compensate for the overhead. When there is not enough work, the parallel version of a program can be much slower than the sequential version on a single processor.

FIGURE 1.6 Asynchronous shared-memory multiprocessor.



In spite of the advantages of parallelism, there are many problems that must be overcome. When a machine contains multiple processors, some mechanism is necessary to share the resources among the different processors. The processors must also be able to communicate with each other in order to coordinate the computations, which involves more complicated hardware. Finally, the software for parallel machines is generally much more complicated than that for scalar and vector computers. Because of these complexities, pipelined vector processors have achieved much wider acceptance than multiple-processor systems.

When pipelining and parallelism are both employed in a computer, impressive computation gains can be achieved. This fact can easily be illustrated by examining the time required by several different computers to compute the LINPACK benchmark, a program that solves a set of 100×100 linear equations. Table 1 lists the time required by several different machines to complete this benchmark[1].

As this example illustrates, parallel and vector hardware can achieve impressive computation rates when compared to single non-pipelined processors. However, fast hardware is only one part of the solution to

1. Source: "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment", Jack J. Dongarra, Technical Memorandum No. 23, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, April 4, 1986.

Until the table lists which machines are parallel and vector, this paragraph isn't going to make any sense.

obtaining fast programs. Unless software is able to take effective advantage of the parallelism present within a processor, parallel hardware simply becomes a wasted, expensive resource. Accordingly, increasing the effective computation rate for applications requires developing mechanism for exploiting parallelism at the language level.

.

Machine	Time in Seconds
NEC SX-2	0.015
Cray XMP-2 (1 processor)	0.028
CDC Cyber 205 (2 pipes)	0.039
Fujitsu VP-200	0.040
Hitachi S-810/20	0.042
IBM 3090 Model 200/VF	0.057

TABLE 1 LINPACK benchmark time

1.1.3 Memory Hierarchy

One complicating aspect of all modern architectures that is particularly important in parallelism is the memory hierarchy. Processor speeds continue to improve faster than memory speeds, causing the distance between main memory and processors (as measured in cycles for a register load) is increasing. This phenomenon is especially prominent on parallel machines, where complex interconnections are required to allow every processor access to all of memory. The ratio of processor speed to memory speed is unlikely to improve because total memory sizes are increasing as well, making a technology shift too expensive to consider seriously. While loads from memory rarely took more than 4 cycles two decades ago, load times in excess of 20 cycles are now very common. On the Cray 2, the first data element is delivered 50 cycles after the load is issued. On parallel processors, load times may be as high as several hundred machine cycles.

There are two measures of performance of a memory system:

1. *Latency* is the number of processor cycles required to deliver any single data element from memory and
2. *Bandwidth* is the number of data elements that can be delivered each cycle.

For the purposes of performance, both measures are important. Latency determines the time that the processor must wait for a value requested from main memory. Many processors stall until a load from main memory completes; on these processors, it is important to minimize the number of requests to memory. Other processors will continue work with an outstanding request to memory, but must stall when the results are required in an operation; on those processors, it is important to try to schedule enough operations between a load and the use of its results to keep the processor busy. Bandwidth determines how many memory operations can be supported each cycle; the higher the bandwidth, the more memory values that can be fetched at one time.

There are two ways to deal with processor latency: latency avoidance and latency tolerance. *Latency avoidance* involves strategies to reduce the typical latencies experienced in a computation. Memory hierarchies are the most common mechanisms for latency avoidance. If values that will be used multiple times are stored in an intermediate memory that is faster than main memory, then accesses after the first will be much cheaper. The earliest example of such an intermediate memory was the processor register set. On most machines built in the last thirty-five years, the processor has had a number of storage locations, called *registers*, which could hold intermediate quantities. Access to these registers is very fast—usually a cycle or less access to a quantity loaded into a register is essentially free. Another common form of intermediate storage is *cache* memory. Cache memory is a small memory (typically between 4K bytes and 4M bytes) built from a faster technology than main memory, and is usually only a few cycles away from the processor. Most cache memories are managed entirely by the hardware[1] and are "transparent" to the programmer. Many systems available today have more than one level of cache memory.

Latency tolerance means doing something else while data is being fetched. The use of explicit prefetching or nonblocking loads are two ways to hide latency. Another interesting latency tolerance mechanism was employed on the Denelcor HEP. The HEP provided a fast context switch operation that made it possible to switch to a new execution stream in only a cycle or two. If you have enough streams active, then if control is continually switched from one to another in a round-robin fashion, latencies will appear small to each stream.

1. The next section will show that this statement is not precisely true.

Although the intent of memory hierarchies is to transparently provide the performance of the fastest level of the hierarchy at roughly the cost of the slowest level, the success to which that goal is met is a factor of how well programs are structured to reuse memory values. Restructuring programs to provide more opportunity for reuse can often provide dramatic improvements in performance.

1.2 Compiler Control of High-Performance Hardware

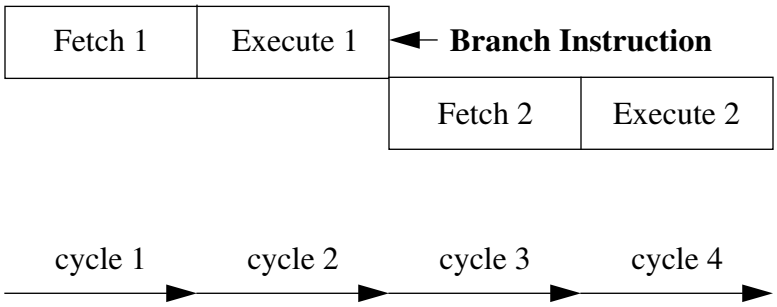
With each new machine architecture comes a new compiler challenge, because most architectural improvements are based on "program predictability"—that is, on the ability to predict the behavior of the instruction stream over a period of time. When that predictability is possible, architectural improvements permit more effective scheduling of work. For instance, registers and caches are effective only when a compiler can determine that values will be reused and should be kept in the very fastest levels of the memory hierarchy. Pipelined functional units are effective only when the compiler can predict and schedule instruction streams to utilize them. In other words, utilizing modern machine architectures requires uncovering large amounts of information about program behavior, a burden which falls primarily on the compiler and the programmer. Thus, the trend in modern machine architecture has been to increasingly shift the burden of achieving high performance to the compiler and the programmer. In this section we will give an overview of the types of compiler challenges that are presented by pipelined and parallel architectures.

1.2.1 Compiling for Pipelines.

In pipelined architectures, the key performance barrier has been *pipeline interlock*. Pipeline interlock occurs when a new set of inputs cannot be injected into the pipeline because they depend upon a previous computation stage that is still within the pipeline. This concept can be easily illustrated with examples for the various pipelines discussed earlier in this chapter.

On the simple instruction execution pipeline on the IBM 7094, pipeline interlock occurs whenever a branch instruction is issued, because the fetch of the next instruction cannot begin until the branch target is determined by the execute phase. This situation is depicted in Figure 1.7.

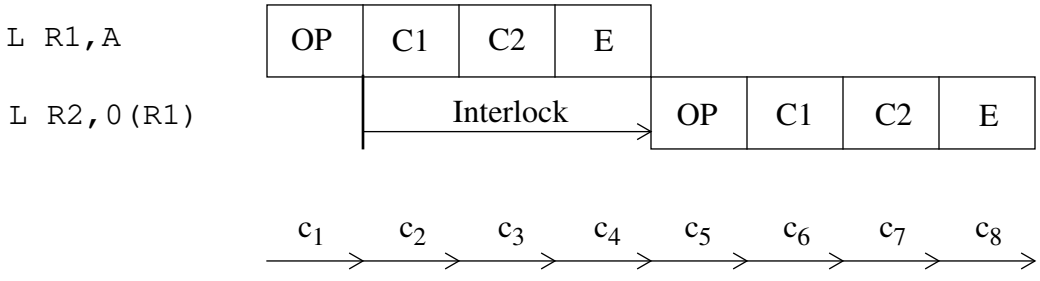
FIGURE 1.7 Branch Interlock on the IBM 7094.



On the 7094, there was no way to avoid this problem through programming. However, modern RISC architectures provide a "branch-and-execute instruction," which always executes the next instruction in sequence before performing the branch. This permits the compiler to rearrange instructions so that the last operation in the loop body, often a data store, is performed "under the branch."

A more complex form of interlock can occur on the IBM 370 class of architectures. Another kind of interlock other than branch interlock occurs when the result of one instruction is necessary for an address computation of a subsequent instruction. This interlock, called *address generation interlock,* is depicted in Figure 1.8. The way to avoid this interlock is to have the compiler arrange the order of instructions so that necessary results from previous instructions are out of the pipeline, if such an ordering is possible.

FIGURE 1.8 Address-generation Interlock on the IBM 370.



Pipelined functional units create a similar problem for the compiler: scheduling instructions to avoid delay because an input is still in the pipeline The following FORTRAN expression provides an example:

```
A*B*C*D
```

If the expression is evaluated in left-to-right order on a machine with a three stage floating point multiplication pipeline, each instruction after the first will have to wait two cycles for the preceding instruction. The total is nine cycles, which takes no advantage of the pipelining. If, on the other hand, the expression is reassociated as

```
(A*B)*(C*D)
```

the first two multiplications can be issued in the first two cycles, while the third can be issued after four cycles, for a total of seven cycles.

By definition, vector instructions are interlock-free. A vector instruction set typically includes loads into vector registers, stores from vector registers and register-to-register vector operations. The following sequence of vector operations adds two 64-element vectors[1] from A and B and stores the result into C.

```
VLOAD   V1,A
VLOAD   V2,B
VADD    V1,V2,V3
VSTORE  V3,C
```

Most vector operations interlock on load-store conflicts at registers, but not at memory (unlike non-vector instructions on machines like the IBM 370.

In order to illustrate the primary compiler problem for vector machines, it is first convenient to understand some of the details of FORTRAN 90 vector notation. Vector operations in FORTRAN 90 have semantics that reflect those of most vector instruction sets. As such, they provide an easier way of presenting compiler issues for vector machines. The previous vector add operation would be written in FORTRAN 90 as

```
C(1:64) = A(1:64) * B(1:64).
```

1. Throughout this book, when no specific hardware vector length is stated, 64 is assumed, since that is the most common vector length.

The semantics of FORTRAN 90 state that the statement must be implemented to behave as if every input on the right hand side of the assignment is loaded from memory before any datum is stored on the left hand side. In other words, all inputs mentioned on the right hand side refer to their values *before* the statement is executed. In the abstract, the operations on the right hand side occur simultaneously.

With these semantics in mind, how can compilers utilize vector units on source programs written in FORTRAN 77, which has no mechanism for specifying vector operations? The compiler will instead be presented with simple loops like

```
DO I = 1, 64
   C(I) = A(I) * B(I)
ENDDO
```

Before it can schedule this loop on vector hardware, the compiler must determine whether the loop is semantically equivalent to the FORTRAN 90 array assignment above. If so, the assignment inside the loop can be vectorized by transliterating the references to the loop induction variable into corresponding triplet notation. However, the question is not trivial, because the loop performs all the loads and stores corresponding to one iteration before beginning the next, while the array statement performs all loads before any stores. In this case, either execution order produces the same final results, so that the meaning is precisely the same. Hence the loop is referred to as *vectorizable*. However, a slightly different case shows a potential problem:

```
DO I = 1, 64
   A(I+1) = A(I) + B(I)
ENDDO
```

Here each iteration of the FORTRAN 77 loop uses a result from the previous iteration, unlike the transliterated FORTRAN 90 array statement:

```
A(2:65) = A(1:64) + B(1:64)
```

in which all inputs on the right are old values. Hence, this second example is not vectorizable. Distinguishing between these two cases is the fundamental problem of vectorization, and was the motivation behind the development of the theory of data dependence.

Given that the primary compiler problem for vector machines is uncovering vector operations, it may appear that using a language that contains explicit array operations (such as FORTRAN 90) would solve the prob-

lem of vectorization. Unfortunately (or fortunately, if you make your livelihood as a compiler writer!), explicit array operations have an analogous set of compiling problems. Consider the following vector statement:

```
A(2:129) = A(1:128) + B(1:128)
```

A naive code generator for a machine that has vector registers of length 64 (one of the most common vector lengths) might produce the following sequence of vector assembly language statements.

```
                VLOAD   V1,A(1:64)
                VLOAD   V2,B(1:64)
                VADD    V1,V2,V3
S1              VSTORE  V3,A(2:65)
S2              VLOAD   V1,A(65:128)
                VLOAD   V2,B(65:128)
                VADD    V1,V2,V3
                VSTORE  V3,A(66:129)
```

Given the limited vector length of 64, this code sequence loads, adds, and stores the first 64 operands, then loads, adds, and stores the last 64 operands. This schedule creates an interesting conflict between statements S_1 and S_2. The store in statement S_1 puts a value into `A(65)`; `A(65)` is also loaded by S_2. Depending on execution timing, the store might take place before or after the load of the same location (recall that vector machines typically do not enforce interlocks on memory). For correctness, the compiler is required to guarantee that the load takes place before the store. In this case, it is easy—statements S_1 and S_2 can simply be interchanged. However, in the general case it may not be known at compile time how much overlap there is between loaded and stored vectors:

```
A(NS:NS+1023) = A(1:1024) + B(1:1024)
```

If `NS` is large the overlap may be larger than all the vector registers. In this case, the only safe strategy is to compile the statement as if it were two statements:

```
T(1:1024) = A(1:128) + B(1:128)
A(NS:NS+127) = T(1:1024)
```

Here the overlap problem is used by computing the results into a temporary array and then copying them back into the target array. Although this ensures that the results are correct, it does so at significant execution costs—nearly double, not counting the time to allocate the array `T`.

Because of the execution inefficiency introduced by the temporary, the compiler must be able to determine when it may be safely omitted. The temporary is required because the vector registers present on most vector machines are fixed in length. This limitation demands that long vector operations be *strip mined* into a sequence of operations, each of whose size is less than or equal to the fixed vector register length. Strip mining translates an array statement of the form

```
A(1:N) = B(1:N) + C(1:N)
```

into a sequential loop containing vector operations of length equal to the vector register length (which we will call VL):

```
DO I = 1, N, VL
   A(I:I+VL-1) = B(I:I+VL-1) + C(I:I+VL-1)
ENDDO
```

Expressed this way, the problem can be illustrated with any vector length. A particularly interesting length is `VL = 1`, which is the length required to compile `FORTRAN 90` programs for scalar machines. After simplification, this case becomes:

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

In this case, the question of correctness of strip mining without a temporary reduces to the question: Does this "scalarized" loop mean the same thing as the original vector statement? In this case it does, but in the previous example (for which the scalarized loop would be)

```
DO I = 1, 1024
   A(I+1) = A(I) + B(I)
ENDDO
```

the scalarized loop is *not* equivalent to the vector statement.As it turns out, the key question in this example is the inverse of the vectorization question—the FORTRAN 90 array statement can be strip mined without a temporary whenever the scalarized version of that array statement is vectorizable. As a result, many of the problems encountered in vectorizing `FORTRAN 77` programs arise in a slightly different form when strip mining `FORTRAN 90` programs. These problems will be examined in more detail in a later chapter; the next section follows historical development by examining the problems of compiling sequential `FORTRAN 77` programs onto vector and parallel machines.

1.2.2 Compiling for Asynchronous Parallelism

Although it might at first seem that asynchronous parallelism should present roughly the same problems that vectorization presents, there are some additional complexities. First, execution schedules on parallel machines contain much more flexibility. Discussing this requires a FORTRAN notation for parallel loops.

The PARALLEL DO statement, which is modelled after the proposal by the Parallel Computer Forum, an informal standardization effort, specifies that there are no scheduling constraints among its iterations. As a result, these iterations may be executed in parallel on any schedule. The statement is an *assurance* from the programmer to the system that it should feel free to execute these iterations in parallel. Assuring that two iterations I_1 and I_2 can be safely executed in parallel is equivalent, according to Bernstein [Bernstein], to ensuring that

1. iteration I_1 must not write into a location that is read by iteration I_2.
2. iteration I_2 must not write into a location that is read by iteration I_1.
3. iteration I_1 must not write into a location that is also written into by iteration I_2.

To illustrate concretely, consider the following example that violates Bernstein's Conditions:

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

This simple case is analogous to the example used in vectorization. Here, the store into A(I+1) on one iteration modifies the same location that is loaded from A(I) on the next iteration. Because the schedule is unspecified, the result from the second of these iterations will differ, depending on the schedule—in one case it will use the new value of A(I) and get the same answer as the sequential loop; in the other case it will use an old value of A(I) and get a different answer.

A more subtle case is illustrated by the following loop:

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

In this case the store is into a location used on the *previous* iteration. On most vector architectures, this loop could be safely vectorized, but a par-

allelizing compiler must be more careful. Consider, for instance, the specific iterations `I=2` and `I=3`. In the sequential analog, the fetch of `A(2)` that occurs on iteration `2` always precedes the store to `A(2)` that occurs on iteration `3`. In the parallel analog, this fetch may come before or after the store, causing the loop to compute different values for `A`.

Finally, consider a loop that violates the third Bernstein Condition.

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

In the sequential analog after execution of the loop, `S` always contains the same value—that assigned on iteration `N`. In the parallel analog, any iteration may be the last to execute, giving `S` a value that is highly nondeterministic.

An correct parallelizing compiler must convert a sequential loop to a parallel one only when it is able to verify that none of Bernstein's conditions holds.

The second new problem introduced by asynchronous parallel machines is that of *granularity.* Because asynchronous parallel processes have large start-up and synchronization overhead, a parallel loop should not be initiated unless there is enough work to compensate for this added cost. Vector units have some overhead, but it is relatively small, permitting profitable vectorization of even single statement loops. Because of the costs involved in synchronizing processors, however, parallel loops must work to minimize synchronization points, which means increasing the granularity of the parallel regions. For the compiler, the implication is that outer loops must be parallelized rather than the inner loops (which is the choice in vectorization), so as to minimize the number of times processors must be synchronized. Similarly, a compiler must be able to parallelize loops with subroutine and function calls, because subroutine invocations are good sources of computation, and almost any loop with a lot of computation will contain some calls.

These considerations make the job of a parallelizing compiler much harder than the job of a vectorizing compiler, because they obligate it to study much larger regions. These regions often span several procedures, making the required analysis difficult within a paradigm of separate compilation.

The final problem encountered by compilers for asynchronous parallel machines has to do with memory access. Some of the most important parallel systems on the market (e.g., the Intel Paragon) do not have globally shared memory. Instead, each processor may access only the memory that is packaged with it. Seitz calls such machines *multicomputers*, to distinguish them from *multiprocssors* (which typically share memory). Parallelizing compilers for multicomputers must decide issues such as which processors' memories hold which variables and when communication primitives are required to move to a processor that doesn't own it. These issues are difficult to solve, and will be revisited in a later chapter.

1.2.3 Compiling for Memory Hierarchy

Although the intent of memory hierarchies are to transparently provide a memory with the performance of the fastest level and the cost of the cheapest level, they do not always succeed. When they fail, the result is performance equivalent to a slow level of memory hierarchy, which is typically *much* slower than a processor's capabilities. Consider, for instance, cache memories. A typical situation is that programs achieve very high performance on small test problems (which are those most commonly used to benchmark machines before purchase), but perform very poorly when the problem is increased to a realistic size. This usually means that the entire data set for the smaller problem could fit into cache, but the larger problem could not. As a result, a cache miss might occur on every data access for the large problem, but only once per array element on the smaller one. Consider a simple example.

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

Assume that this example is executed on a machine with one-word cache blocks and a cache that always replaces the least recently used (LRU) block when space is needed. Although the example code efficiently accesses the elements of array `A`, incurring only one miss per element, it will always miss on the access to `B(J)` if `M` is large enough. The breakover value of `M` occurs at the size of the cache in words, because an element of `B` is not reused until all of the `M-1` other elements of `B` are used. Because of this small amount of reuse, the needed element of `B` is guaranteed to have been pushed out of the cache by the LRU replacement policy.

One way to alleviate this problem is to strip mine the inner loop to a size that will fit in cache and interchanging the "by-strip" loop to the outermost position:

```
DO JJ = 1, M, L
   DO I = 1, N
      DO J = JJ, JJ+L-1
         A(I) = A(I) + B(J)
      ENDDO
   ENDDO
ENDDO
```

Here, `L` must be smaller than the number of words in the cache. This second example misses only once for each element of `B`, since an element of `B` stays in the cache until all its uses are finished. The cost is an increase in the number of misses on references to `A`. However the total number of misses on A is approximately $(NM)/L$ as opposed to NM misses on `B` in the original code.

While programmers can make changes such as this to their source programs, doing so leads to machine-specific source programs that must be retargeted by hand to each new machine. We believe this kind of machine-specific optimization should be the job of the optimizing compiler. Later chapters will show how compiler technology developed for vectorizing and parallelizing compilers can be adapted to optimizing use of memory hierarchy.

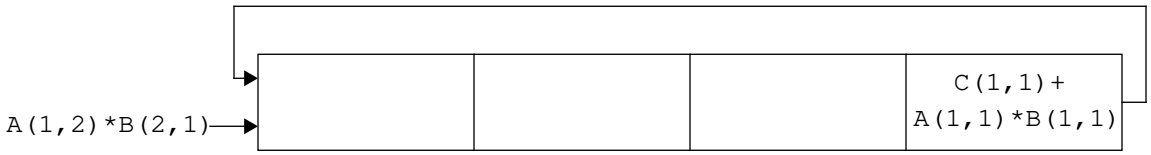
1.2.4 A Case Study: Matrix Multiplication

To illustrate these issues more fully, consider the problem of matrix multiplication—a computation that is at the heart of many important applications. Typical FORTRAN code to compute the product of two matrices `A` and `B` is shown below.

```
DO I = 1, N
   DO J = 1, N
      C(J,I) = 0.0
      DO K = 1, N
         C(J,I) = C(J,I) + A(J,K) * B(K,I)
      ENDDO
   ENDDO
ENDDO
```

This code fragment employs a straightforward approach to computing the product matrix `C`: the two outer loops select a specific element of the product matrix to compute and the inner loop computes that element by taking the inner product of the appropriate row of the first input and column of the second. On a scalar machine (that is, one containing no parallel or vector hardware), this code makes excellent use of the hardware. Since the inner loop accumulates products into `C(J,I)`, there is no need to access memory (either fetches or stores) for `C` during that loop. Instead, the intermediate results can be kept in a scalar register until the loop has completed, and then stored to memory. When presented with this fragment, a good optimizing compiler should generate code that comes close to the optimal performance possible on a scalar machine.

FIGURE 1.9 Matrix multiplication execution pipeline interlock.

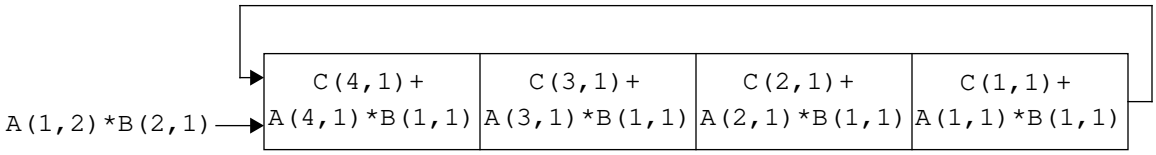


However, on a scalar machine with a pipelined floating-point unit, the same code is not likely to fare nearly so well. The code runs efficiently on a non-pipelined scalar machine because the result of one iteration of the innermost loop is used on the next, making it possible to reuse the result from a register. On a pipelined machine, each iteration must wait until the final addition on the previous iteration is available before it can begin its own final iteration (see Figure 1.9). One way to overcome this problem is to work on four different iterations of an outer loop at the same time, thus filling the four pipe stages with four independent computations, as shown in Figure 1.9. The FORTRAN code to accomplish this is given below (here `N` is assumed to be a multiple of 4).

```
DO I = 1, N,
   DO J = 1, N, 4
      C(J,I) = 0.0
      C(J+1,I) = 0.0
      C(J+2,I) = 0.0
      C(J+3,I) = 0.0
      DO K = 1, N
```

```
            C(J,I) = C(J,I) + A(J,K) * B(K,I)
            C(J+1,I) = C(J+1,I) + A(J+1,K) * B(K,I)
            C(J+2,I) = C(J+2,I) + A(J+2,K) * B(K,I)
            C(J+3,I) = C(J+3,I) + A(J+3,K) * B(K,I)
         ENDDO
      ENDDO
   ENDDO
```

FIGURE 1.10 Pipeline filling via outer loop unrolling.



On a vector machine, the inner loop of the scalar matrix multiplication cannot be vectorized because the computation is a recurrence—that is, the interleaved load-store order of the scalar code is required to maintain the meaning of the code (assuming that reassociation of the addition is prohibited because of precision requirements). On a vector machine like the Cray YMP, which has vector operations of length 64, we need to move 64-element vector operations into the inner loop so that the vector registers on the YMP can be reused on each iteration. Thus the best code for the YMP looks like this:

```
DO I = 1, N
   DO J = 1, N, 64
      C(J:J+63,I) = 0.0
      DO K = 1, N
         C(J:J+63,I) = C(J:J+63,I) +
                       A(J:J+63,K) * B(K,I)
      ENDDO
   ENDDO
ENDDO
```

This form of matrix multiplication extends the scalar register properties of the first code to the vector registers of the Cray. The `K` loop now performs all computations necessary for one vector section of the product

matrix; as a result, that section may be accumulated in a vector register just as the single element may be accumulated in the scalar form.

Achieving high performance on multiprocessors requires a different set of considerations than vector machines. Since vector loops are executed "synchronously" by hardware, they must be the innermost loops in a given nest. Parallel loops, on the other hand, are executed asynchronously and therefore it is desirable to move them to the outermost position to ensure that there is enough computation to compensate for the start-up and synchronization overhead. Thus, on a multiprocessor like the Sequent Symmetry, the optimal form of matrix multiplication would be:

```
PARALLEL DO I = 1, N
   DO J = 1, N
      C(J,I) = 0.0
      DO K = 1, N
         C(J,I) = C(J,I) + A(J,K) * B(K,I)
      ENDDO
   ENDDO
ENDDO
```

In this form, each processor can independently compute a column of the product matrix, with no need so synchronize with other processors until the product is complete. This form requires that all processors have access to the entire `A` matrix and one column of the `B` matrix, which is trivially true in the Sequent since its memory is shared among all its processors.

However, not all parallel processors have shared memory; the Intel Hypercube, for instance, has processors with local memory but no globally shared memory. It is possible that `N` in the above fragment may be so large that each processor cannot hold the entire `A` matrix. In such a case, it is necessary to parallelize the inner loop, not the outer.

```
DO I = 1, N
   DO PARALLEL J = 1, N
      C(J,I) = 0.0
      DO K = 1, N
         C(J,I) = C(J,I) + A(J,K) * B(K,I)
      ENDDO
   ENDDO
ENDDO
```

In this form, each processor need only hold one column of A and one element of B in order to compute a product element.

Of course, it is possibly to build machines with both vector instructions and multiple processors. A classic example is the Cray YMP, which fully configured contains eight vector register processors connected via a shared memory. From the previous discussion of vector processors and multiprocessors, it should be fairly obvious by now that the optimal form of matrix multiplication for this computer is

```
DO PARALLEL I = 1, N
   DO J = 1, N, 64
      C(J:J+63,I) = 0.0
      DO K = 1, N
         C(J:J+63,I) = C(J:J+63,I) +
                       A(J:J+63,K) * B(K,I)
      ENDDO
   ENDDO
ENDDO
```

In this form, each processor makes optimal use of its vector registers with a minimal amount of communication.

Two important lessons should be evident from this study of matrix multiplication:

1. Explicit representation of parallelism in a source language is not sufficient to guarantee optimal use of parallel hardware. Each of the six machine types presented required a different representation of parallelism—radically different in some cases. Furthermore, getting the best form for a specific machine required detailed knowledge of the machine's architecture. This observation suggests that explicitly parallel programs need to be tailored for individual architectures; they lose efficiency when ported from one machine to another.
2. While the optimal form of matrix multiplication differed for each kind of machine, each form could be obtained from the initial non-parallel source by relatively simple program transformations. Changing the order in which the loops are executed would permit most of the optimal forms to be found.

Given the increasing lifetime of software and the decreasing lifetime of hardware, these lessons suggest that tailoring code to the specific machine architecture is best left to the compiler. The next section intro-

duces the basic elements of a compiler technology designed to make this goal achievable.

1.3 Advanced Compiler Technology

Compilers transform computations from a representation that is easy for a human to understand down to a low level representation that a machine can execute. The high level of the human representation is rarely intended to accommodate details of machine architecture, and the translation process may introduce inefficiencies into the resulting machine-level program. The goal of optimization phases in compilers is to eliminate these inefficiencies, and to transform a computational representation into one that is well-tuned for a given architecture.

1.3.1 Automatic Parallelization

A natural extension of this philosophy is to have compilers automatically transform code to take advantage of parallelism. That is, the original program is viewed as a "specification", which sets requirements on the result the program is to compute. The compiler is free to transform code as desired as it lowers it down to executable level, so long as the transformed program computes the same results as the specification. While this extension seems natural enough, it is often viewed as unnatural, due primarily to the radical nature of the transformations required to exploit parallelism. For instance, matrix multiplication required loop interchange, loop splitting, loop distribution, vectorization, and parallelization to achieve optimal results across various parallel architectures. These transformations are not usually found in classical optimizers.

This book presents the theoretical and practical ideas behind a compiler approach based upon the paradigm of aggressively transforming from a sequential specification. Specifically, we present methods for uncovering parallelism in `FORTRAN 77` programs, since `FORTRAN` is currently the *lingua franca* of scientific computing. We will not argue the merits of this approach versus that of starting from a parallel computation; both approaches have strong arguments in their favor. However, we will note that starting from a sequential language has the very practical advantage that it targets the bulk of existing programs. Additionally (as will become evident later in this book), the same theory that is necessary to uncover parallelism within sequential programs can also be applied to optimizing parallel programs.

1.3.2 Dependence

When a programmer writes a program in a sequential programming language, the results that he expects the program to compute are those obtained by first executing the initial program statement, then the second, and so on, with appropriate exceptions for control flow constructs such as branches and loops. Essentially, the programmer has specified a total ordering on the operations he expects the computer to execute. Obviously, exploiting parallelism directly from such a program is impossible, because the programmer has specified exactly the order in which he expects operations to occur, and that order does not include parallel or vector execution. The fundamental challenge that an advanced compiler must meet is determining when an execution order (one that includes parallel or vector execution) that is different from that specified by the programmer will *always* compute the same results as the execution order the programmer specified. In other words, sequential languages introduce constraints that are not necessary to the results of a computation; the key to transforming such programs to parallel form is uncovering precisely those constraints which *must be* met in order for the program to compute the correct results. If these constraints can be precisely captured, then the compiler is free to reorder execution of the program as it wishes, so long as it does not change these essential constraints. This minimal set of constraints which must be observed is known as the *dependences* of the program. More precisely, dependence is a relation on the statements on the program. The pair $<S_1, S_2>$ is in the relation if S_2 is executed after S_1 in the sequential program, and *must* be executed after S_1 in any valid reordering of the program.

To illustrate the concept of dependence, consider the following simple code fragment:

```
S1  PI = 3.14159
S2  R = 5
S3  AREA = PI * R ** 2
```

The results of this fragment are defined as being those that occur when the execution order $<S_1, S_2, S_3>$ is taken. However, nothing in the code requires that S_2 be executed after S_1—in fact, the execution order $<S_2, S_1, S_3>$ produces exactly the same results (i.e. the same value of the variable AREA) as the original order. In contrast, the time at which S_3 is executed is critical; if it is executed before either S_1 or S_2 an incorrect value of AREA will be computed because the input operands will not have been set. In terms of dependence, the pairs $< S_1, S_3>$ and $<S_2, S_3>$

are in the dependence relation of the fragment, but the pair <S_1, S_2> is not.

In these terms, dependence is an easy concept to understand. Unfortunately, this simplistic viewpoint does not permit very effective utilization of parallelism. In order to efficaciously use vector hardware, it is necessary to perform the same operation on different elements of memory. In order to efficaciously use parallel hardware, it is necessary to have a number of regions that can be executed in "random" order. In either case, the key to speeding up a computation is speeding up the most frequently executed parts of the program. The implication is that a compiler must be able to successfully handle loops and arrays in order to uncover parallelism. Loops and arrays require more sophistication than the simple dependence theory presented so far. For instance,

```
    DO I = 1, N
S1    A(I) = B(I) + 1
S2    B(I+1) = A(I) - 5
    ENDDO
```

contains the dependence <S_1, S_2>, since on any loop iteration the computed value of A is immediately used, and also contains the dependence <S_2, S_1>, since every loop iteration other than the first uses a value of B computed on the previous iteration. Detecting these dependences is difficult enough, since different loop iterations touch different elements of the arrays. However, even that is only part of the problem; the cycle in the dependence graph, indicating the cyclic nature of the loop, complicates scheduling algorithms for utilizing parallelism.

Loops and arrays are only some of the obstacles that the compiler must overcome. Procedure calls are another: procedures intended to hide unnecessary details from human programmers also have the side effect of hiding important information from the compiler. IF statements are another problem. The fact that statements may be conditionally executed according to values available only at run-time is a distinct complication in the transformation process. For that matter, the dependences of a program may rely upon the values of certain key variables—information which again is available only at run-time.

1.3.3 Transformations

As described so far, dependence is only a passive guide for uncovering implicit parallelism within a program. However, it is far more than that.

Since a program's execution may be reordered in any fashion that preserves dependences, dependence can also form the basis of powerful transformations that enhance the parallelism present within a program. For instance, the simple exchange of two arrays

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

cannot be directly vectorized because the scalar temporary `T` creates a bottleneck to the vector computation—only one element can be transferred at a time in the program as specified. If the scalar temporary is expanded into a vector temporary

```
DO I = 1, N
   T(I) = A(I)
   A(I) = B(I)
   B(I) = T(I) 100
ENDDO
```

then the loop can be directly vectorized. Powerful transformations such as *scalar expansion* can be effectively applied based on the dependences of a program.

The dependence theory presented in this book is that developed in the Parallel Fortran Converter (also called PFC), a research project carried out at Rice University under the sponsorship of IBM corporation. This theory has built heavily on previous work at other institutions. In order to place things in historical perspective, the following section details a brief history of the development of dependence theory as presented in this book.

1.3.4 Historical Development

[Need to check with Presberg here for more historical details-- see if he can put us in touch with Lam-

Some of the earliest work on automatic vectorizers and parallelizers was done at Massachusetts Computer Associates, later known as Compass. In the early 1970's, Compass created a transformation tool for the Illiac IV known as the Parallelizer. This tool was based on the essential elements of dependence as described earlier. Statement dependences were determined using simple but effective data dependence tests, and dependence was represented in a form known as “distance vectors” and “direction vectors” (This form will be fully described in Chapter 2.) The important

aspect of direction vectors is that they permit optimization on a loop-by-loop basis. The criteria used to determine vector and parallel loops, while difficult to understand as originally presented, precisely partitions loops that can be correctly performed in vector from those that cannot. This criteria, called “tree inconsistency” in a technical report by Lamport, is equivalent to the notion of cycles in the dependence graph to be presented later. While the Parallelizer did many things well, it had some drawbacks also. The intermediate form used to hold a computation was very low level, causing many different types of dependence to be confused together. It is also unclear how much of the theory was actually implemented, and also whether the translator was fast enough to be useful. While the earliest version of the Parallelizer was not very successful, it evolved over time into VAST, a very successful vectorizer marketed by Pacific Sierra, Corporation. The key papers describing the Parallelizer include theoretical technical reports by Lamport [CACM paper, Compass report], a summary report by Presberg and Myszewski[Myszewski], and indirectly, a very elegant paper by Loveman espousing a philosophical approach toward optimization [Loveman].

Another very early vectorizer was written for the Texas Instruments Advanced Scientific Computer. While public reports on this compiler are somewhat sketchy, it is clear that the compiler was dependence-based. One of the important aspects of the compiler was the use of relatively sophisticated tests for determining dependences involving subscripted variables. Cohagen describes a Greatest Common Denominator dependence test based on the theory of Diophantine equations that could handle more complicated subscripts than the simple forms handled in the Parallelizer. Wedel describes several program transformations utilized in the compiler—loop reversal,...

While the Parallelizer and the ASC compiler were early commercial efforts, the true pioneering work into automatic program transformations was performed at the University of Illinois under the direction of David Kuck. Illinois had been involved with parallelism at its very early stages with the Illiac IV; the direction turned toward automatic program transformations in the mid 1970's with the inception of the Parafrase project. As can be surmised from its name, Parafrase attempts to restate sequential programs as parallel and vector equivalents. The basis of transformations in Parafrase is dependence in the form of direction vectors. However, the notion of dependence utilized in Parafrase is much cleaner than that used in the Parallelizer: data dependences are clearly classified into true dependences, antidependences, and output dependences, and extremely sophisticated tests are used to determine the presence or

absence of dependence. Because of its clear definition of dependence, Parafrase has a clean algorithm for detecting vector statements. In its terms, any statement can be executed in vector so long as it does not depend upon itself. As a result, a transitive close of the dependence graph is used to partition vectorizable and non-vectorizable statements. Many other important transformations that enhance parallelism, such as loop distribution, loop interchange, auxiliary induction variable substitution, loop partitioning, node splitting, scalar renaming. Among the many important papers produced in this project are papers on dependence testing [Banerjee, Kuhn, Wolfe], loop interchange [Wolf*2], auxiliary transformations [Wolfe, Leasure, Towle], low level parallelism [Muroka], different parallel constructs [Padua,?, Abu Sufah]. An outgrowth of the Parafrase project has been Kuck and Associates, Inc., a commercial enterprise which markets an advanced vectorizer known as KAP.

The Parallel Fortran Converter project was undertaken at Rice in the late 1970's under the direction of Ken Kennedy. Initially, this project started with a copy of Parafrase and worked on extending it in a number of ways. However, for a variety of reasons, the Parafrase version was dropped, and an entirely new effort was begun. The dependence representation in the new effort was "loop carried dependence"—a representation that is much cleaner than direction vectors for many types of transformations. As a result, clean efficient algorithms could be used to perform many program transformations. The advancements made in PFC include a distinct categorization of dependences that makes clear a number of areas to which dependence can be applied, efficient algorithms for a number of transformations—particularly vector code generation and loop interchange, methods for handling conditional statements within a program, and new transformations for enhancing parallelism. The PFC project itself served as a prototype for the IBM vectorizing compiler for the 3090; the theory utilized in PFC formed the basis of the Convex vectorizing compiler. The PFC project also spawned off other projects exploring the interactive display of dependence [PTOOL], and interactive transformation of programs [OASC proposal]. The theory employed in PFC is presented in the remainder of this book.

1.3.5 FORTRAN 90

In order to illustrate the concepts of program transformations, it is necessary to have a language in which examples can be presented. Since FORTRAN is by far and away the most heavily used language on parallel and vector computers today, the most logical choice of language is a version

of FORTRAN extended with vector and parallel operations. While there are a number of such dialects in existence today, the next standard of FORTRAN, currently called FORTRAN 90, contains such extensions. Since most vendors will probably move to FORTRAN 90 over the next few years, we have chosen vector and parallel syntax very similar to that in the FORTRAN 90 standard. This language is also the output language of PFC.

The remainder of this book will use FORTRAN 90 as the output language to illustrate parallelism. Appendix A on page 267 contains a complete introduction to the features of FORTRAN 90. For the purposes of this book, the key extensions of the language are array assignments statements and triplet notation (both of which specify vector statements with simultaneous semantics), the FORALL iterative construct (which was omitted from the standard but is useful nevertheless), and the WHERE statement.

1.4 Chapter Summary

This chapter has introduced the basic problems of utilizing parallel and vector hardware from a high level language. In order to utilize a vector architecture, a number of similar operations must be executed *simultaneously.* In order to utilize parallel architectures, the order in which operations are performed must not affect the final computed result. Unfortunately, most programming languages—in particular, FORTRAN—do not have constructs that utilize these kinds of hardware. Even those versions that do require the same basic theory developed in this book for efficient execution.

This book develops the theory and practice behind methods for automatically transforming sequential programs into forms able to take advantage of parallel and vector architectures. Compiling systems that perform such transformations work by determining those execution orders within the total ordering that *must* be observed in order to preserve the results of the computation. Those execution orders are captured in a relation known as dependence—statement S_2 is said to depend upon statement S_1 if S_2 follows S_1 in execution in the sequential program, and it *must* follow S_1 in any transformed program that computes the same results. With this definition, a compiler is free to reorder statements as it wishes so long as it does not violate any dependences.

The approach of transforming sequential programs to parallel programs has many advantages. The primary advantage is portability: since the compiler is the only part of the programming system that has detailed knowledge of the underlying machine architecture, it can do the tuning which often leads to non-portable programs. As a result, the programmer does not need to be aware of details of the parallel architecture—the compiler takes care of those. Additionally, old programs which were written for sequential machines can gain some advantage of parallel hardware. An equally valid approach is to take parallel programs and map them onto the parallelism of the existing hardware.