J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

PARALLELIZING AND VECTORIZING COMPILERS

Programming computers is a complex and constantly evolving activity. Technological innovations have consistently increased the speed and complexity of computers, making programming them more difficult. To ease the programming burden, computer *languages* have been developed. A programming language employs syntactic forms that express the concepts of the language. Programming language concepts have evolved over time as well. At first the concepts were centered on the computing machine being used. Gradually, the concepts of computer languages have evolved away from the machine, toward problem-solving abstractions. Computer languages require translators, called *compilers*, to translate programs into a form that the computing machine can use directly, called *machine language*.

The part of a computer that does arithmetical or logical operations is called the *processor*. Processors execute *instructions*, which determine the operation to perform. An instruction that does arithmetic on one or two numbers at a time is called a *scalar* instruction. An instruction that operates on a larger number of values at once (e.g. 32 or 64) is called a *vector* instruction. A processor that contains no vector instructions is called a *scalar processor*, and one that contains vector instructions is called a *vector processor*. If the machine has more than one processor of either type, it is called a *multiprocessor* or a *parallel computer*.

The first computers were actually programmed by connecting components with wires. This was a tedious task and very error-prone. If an error in the wiring was made, the programmer had to visually inspect the wiring to find the wrong connection.

The innovation of stored-program computers eliminated the physical wiring job. The stored-program computer contained a memory device that could store a series of binary digits (1s and 0s), a processing device that could carry out several operations, and devices for communicating with the outside world (input/output devices). An instruction for this kind of computer consisted of a numeric code indicating which operation to carry out, as well as an indication of the operands to which the operation was applied. Debugging such a machine-language program involved inspecting memory and finding the binary digits that were loaded incorrectly. This could be tedious and time-consuming.

Then another innovation was introduced—symbolic *assembly language* and an *assembler* program to translate assembly-language programs into machine language. Programs could be written using names for memory locations and registers. This eased the programming burden considerably, but people still had to manage a group of registers, check status bits within the processor, and keep track of other machine-related hardware. Though the problems changed somewhat, debugging was still tedious and time-consuming.

A new pair of innovations was required to lessen the onerous attention to machine details—a new language (Fortran), closer to the domain of the problems being solved, and a program for translating Fortran programs into machine code: a Fortran compiler. The 1954 Preliminary Report on Fortran stated that "... Fortran should virtually eliminate coding and debugging ...".

That prediction, unfortunately, did not come true, but the Fortran compiler was an undeniable step forward, in that it eliminated much complexity from the programming process. The programmer was freed from managing the way machine registers were employed and many other details of the machine architecture.

The authors of the first Fortran compiler recognized that in order for Fortran to be accepted by programmers, the code generated by the Fortran compiler had to be nearly as efficient as code written in assembly language. This required sophisticated analysis of the program, and *optimizations* to avoid unnecessary operations.

Programs written for execution by a single processor are referred to as *serial* or *sequential* programs. When the quest for increased speed produced computers with vector instructions and multiprocessors, compilers were created to convert serial programs for use with these machines. Such compilers, called vectorizing and parallelizing compilers, attempt to relieve the programmer of dealing with the machine details. They allow the programmer to concentrate on solving the object problem, leaving it to the compiler to concern itself with the complexities of the machine. Much more sophisticated analysis is required of the compiler to generate efficient machine code for these types of machines.

Vector processors provide instructions that load a series of numbers for each operand of a given operation, then perform the operation on the whole series. This can be done in *pipelined* fashion, similar to operations done by an assembly line, which is faster than doing the operation on each item separately. Parallel processors offer the opportunity to do multiple operations at the same time on the different processors.

This article will attempt to give the reader an overview of the vast field of vectorizing and parallelizing compilers. In the next section, we will review the architecture of high-performance computers. Thereafter we will cover the principal ways in which high-performance machines are programmed. Then we will delve into the analysis techniques that help parallelizing and vectorizing compilers optimize programs. Then we discuss techniques that transform program code in ways that can enable improved vectorization or parallelization. Two sections discuss the generation of vector instructions and parallel regions, respectively, and the issues surrounding them. The final section discusses a number of important compiler-internal issues.

Parallel Machines

Classifying Machines. Many different terms have been devised for classifying high-performance machines. In a 1966 paper, M. J. Flynn divided computers into four classifications, based on the instruction and data streams used in the machine. These classifications have proven useful, and are still in use today:

(1) *Single Instruction Stream, Single Data Stream (SISD)* These are single-processor machines.
(2) *Single Instruction Stream, Multiple Data Streams (SIMD)* These machines have two or more processors that all execute the same instruction at the same time, but on separate data. Typically, a SIMD machine uses a SISD machine as a *host*, to broadcast the instructions to be executed.
(3) *Multiple Instruction Streams, Single Data Stream (MISD)* No whole machine of this type has ever been built, but if it were, it would have multiple processors, all operating on the same data. This is similar to the idea of pipelining, where different pipeline stages operate in sequence on a single data stream.
(4) *Multiple Instruction Streams, Multiple Data Streams (MIMD)* These machines have two or more processors that can all execute different programs and operate on their own data.

Another way to classify multiprocessor computers is according to how the programmer can think of the memory system. *Shared-memory multiprocessors* (*SMP*s) are machines in which any processor can access the contents of any memory location by simply issuing its memory address. Shared-memory machines can be thought of as having a shared memory unit accessible to every processor. The memory unit can be connected to the machine through a *bus* (a set of wires and a control unit that allows only a single device to connect to a single processor at one time), or an interconnection network (a collection of wires and control units, allowing multiple data transfers at one time). If the hardware allows nearly equal access time to all of memory for each processor, these machines can be called *uniform memory access* (*UMA*) computers.

Distributed-memory multiprocessors (*DMP*s) use processors that each have their own local memory, inaccessible to other processors. To move data from one processor to another, a *message* containing the data must be sent between the processors. Distributed-memory machines have frequently been called *multicomputers*.

Distributed shared memory (*DSM*) machines use a combined model, in which each processor has a separate memory, but special hardware and/or software is used to retrieve data from the memory of another processor. Since in these machines it is faster for a processor to access data in its own memory than to access data in another processor's memory, these machines are frequently called *nonuniform memory access* (*NUMA*) computers. NUMA machines may be further divided into two categories—those in which cache coherence is maintained between processors (cc-*NUMA*) and those in which cache coherence is not maintained (nc-*NUMA*).

Parallel Computer Architectures. People have experimented with quite a few types of architectures for high-performance computers. There has been a constantly readjusting balance between ease of implementation and high performance.

Single-Instruction-Stream Multiple-Data-Stream Machines. The earliest parallel machines were SIMD machines. SIMD machines have a large number of very simple *slave* processors controlled by a sequential host or *master* processor. The slave processors each contain a portion of the data for the program. The master processor executes a user's program until it encounters a parallel instruction. At that time, the master processor broadcasts the instruction to all the slave processors, which then execute the instruction on their data. The master processor typically applies a bit mask to the slave processors. If the bit-mask entry for a particular slave processor is 0, then that processor does not execute the instruction on its data. The set of slave processors is also called an attached array processor, because it can be built into a single unit and *attached* as a performance upgrade to a uniprocessor.

An early example of a SIMD machine was the Illiac IV, built at the University of Illinois during the late 1960s and early 1970s. The final configuration had 64 processors, one-quarter of the 256 originally planned. It was the world's fastest computer throughout its lifetime, from 1975 to 1981. Examples of SIMD machines from the 1980s were the Connection Machine from Thinking Machines Corporation, introduced in 1985, and its follow-on, the CM-2, which contained 64K processors, introduced in 1987.

Vector Machines. A vector machine has a specialized instruction set with vector operations and usually a set of vector registers, each of which can contain a large number of floating-point values (up to 128). With a single instruction, it applies an operation to all the floating-point numbers in a vector register. The processor of a vector machine is typically pipelined, so that the different stages of applying the operation to the vector of values overlap. This also avoids the overheads associated with loop constructs. A scalar processor would have to apply the operation to each data value in a loop.

The first vector machines were the Control Data Corporation (*CDC*) Star-100 and the Texas Instruments ASC, built in the early 1970s. These machines did not have vector registers, but rather loaded data directly from memory to the processor. The first commercially successful vector machine was the Cray Research Cray-1. It used vector registers and paired a fast vector unit with a fast scalar unit. In the 1980s, CDC built the Cyber 205 as a follow-on to the Star-100, and three Japanese companies, NEC, Hitachi, and Fujitsu built vector machines. These three companies continued manufacturing vector machines through the 1990s.

Shared-Memory Machines. In a SMP, each processor can access the value of any shared address by simply issuing the address. Two principal hardware schemes have been used to implement this. In the first (called *centralized shared memory*), the processors are connected to the shared memory via either a system bus or an interconnection network. The memory bus is the cheapest way to connect processors to make a shared-memory system. However, the bus becomes a bottleneck, since only one device may use it at a time. An interconnection network has more inherent parallelism, but involves more expensive hardware. In the second (called *distributed shared memory*), each processor has a local memory, and whenever a processor issues the address of a memory location not in its local memory, special hardware is activated to fetch the value from the remote memory that contains it.

The Sperry Rand 1108 was an early centralized shared-memory computer, built in the mid 1960s. It could be configured with up to three processors plus two input/output controller processors. In the 1970s, Carnegie Mellon University built the C.mmp as a research machine, connecting 16 minicomputers (DEC PDP-11s) to 16 memory units through a crossbar interconnection network. Several companies built bus-based centralized shared-memory computers during the 1980s, including Alliant, Convex, Sequent, and Encore. The 1990s saw fewer machines of this type introduced. A prominent manufacturer was Silicon Graphics Inc. (*SGI*), which produced the Challenge and Power Challenge systems in that period.

During the 1980s and 1990s, several research machines explored the DSM architecture. The Cedar machine built at the University of Illinois in the late 1980s connected a number of bus-based multiprocessors (called *clusters*) with an interconnection network to a global memory. The global memory modules contained special synchronization processors that allowed clusters to synchronize. The Stanford DASH, built in the early 1990s, also employed a two-level architecture, but added a cache-coherence mechanism. One node of the DASH was a bus-based multiprocessor with a local memory. A collection of these nodes were connected together in a mesh. When a processor referred to a memory location not contained within the local node, the node's directory was consulted to determine the remote location. The MIT Alewife project also produced a directory-based machine. A prominent directory-based commercial machine was the Origin 2000 from SGI.

Distributed-Memory Multiprocessors. In a DMP, each processor has access to its local memory only. It can only access values from a different memory by receiving them in a message from the processor whose memory contains them. The other processor must be programmed to send the value at the right time.

People started extensive experimentation with DMPs in the 1980s. They were searching for ways to construct computers with large numbers of processors cheaply. Bus-based machines were easy to build, but suffered from a limitation on bus communication bandwidth, and were hampered by the serialization required to use the bus. Machines built using interconnection networks or crossbar switches had increased bandwidth and communication parallelism, but were expensive to build. Multicomputers simply connected processor/memory nodes with communication lines in a number of configurations, from meshes to toroids to hypercubes. These turned out to be cheap to build and (usually) had sufficient memory bandwidth. The principle drawback of these machines was the difficulty of writing programs for them.

One example of a DMP in the 1980s was a hypercube research machine built at the California Institute of Technology, which had processors connected in a hypercube topology by communication links. The nCUBE company and Intel Scientific Computers (*ISC*) were founded to build similar machines. nCUBE built the nCUBE/1 and nCUBE/2, while ISC built the iPSC/1 and iPSC/2. In the 1990s, nCUBE followed with the nCUBE/2S, and ISC built the iPSC/860, the iWarp for Carnegie Mellon University, and the Paragon.

Cache-Only Memory Architecture. A machine that uses all of its memory as a cache is called a *cache-only memory architecture* (*COMA*). Typically in these machines, each processor has a local memory, and data are allowed to move from one processor's memory to another during the run of a program. The term *attraction memory* has been used to describe the tendency of data to migrate toward the processor that uses it the most. Theoretically, this can minimize the latency to access data, since latency increases as the data get further from the processor.

The COMA idea was introduced by a team at the Swedish Institute of Computer Science, working on the Data Diffusion Machine. The idea was commercialized by Kendall Square Research (*KSR*), which built the KSR1 in the early 1990s.

Multithreaded Machines. A multithreaded machine attempts to hide latency to memory by overlapping it with computation. As soon as the processor is forced to wait for a data access, it switches to another thread to do more computation. If there are enough threads to keep the processor busy until each datum arrives, then the processor is never idle.

The Denelcor HEP machine was the first multithreaded processor in the early 1980s. In the 1990s, the Alewife machine used multithreading in the processor to help hide some of the latency of memory accesses. Also, in the 1990s, the Tera Computer Company developed the MTA machine, which expanded on many of the ideas used in the HEP.

Clusters of Shared-Memory Multiprocessors. Another approach to building a multiprocessor is to use a small number of commodity microprocessors to make centralized shared-memory *clusters*, then connect large numbers of these together. The number of microprocessors to use to make a single cluster is determined by the number that would saturate the bus (keep the bus constantly busy). Such machines are called *clusters of SMPs*. Clusters of SMPs have the advantage of being cheap to build. During the second half of the 1990s, people began building clusters out of low-cost components: Pentium processors, a fast network (such as Ethernet or Myrinet), the Linux operating system, and the message-passing interface (*MPI*) library. These machines are sometimes referred to as *Beowulf clusters*.

Programming Parallel Machines

From a user's point of view there are three different ways of creating a parallel program:

(1) Writing a serial program and compiling it with a parallelizing compiler
(2) Composing a program from modules that have already been implemented as parallel programs
(3) Writing a program that expresses parallel activities explicitly

Option 1 above is obviously the easiest for the programmer. It is easier to write a serial program than it is to write a parallel program. The programmer would write the program in one of the languages for which a parallelizing compiler is available (Fortran, C, and C++), then employ the compiler. The technology that supports this scenario is the main focus of this article.

Option 2 above can be easy as well, because the user does not need to deal with explicit parallelism. For many problems and computer systems there exist libraries that perform common operations in parallel. Among them, mathematical libraries for manipulating matrices are best known. One difficulty for users is that one must make sure that a large fraction of the program execution is spent inside such libraries. Otherwise, the serial part of the program may dominate the execution time when running the application on many processors.

Option 3 above is the most difficult for programmers, but gives them direct control over the performance of the parallel execution. Explicit parallel languages are also important as a target for parallelizing compilers. Many parallelizers act as source-to-source restructurers, translating the original, serial program into parallel form. The actual generation of parallel code is then performed by a *backend compiler* from this parallel language form. The remainder of this section discusses this option in more detail.

Expressing Parallel Programs. Syntactically, parallel programs can be expressed in various ways. A large number of languages offer parallel programming constructs. Examples are Prolog, Haskell, Sisal, Multilisp, Concurrent Pascal, and Occam. Compared to standard, sequential languages, they tend to be more complex, available on less machines, and lack good debugging tools, which contributes to the difficulty facing the user of option 3 above.

Parallelism can also be expressed in the form of *directives*, which are pseudocomments with semantics understood by the compiler. Many manufacturers have devised their own set of such directives (Cray, SGI, Convex, etc.), but during the 1990s the OpenMP standard emerged. OpenMP describes a common set of directives for implementing various types of parallel execution and synchronization. One advantage of the OpenMP directives is that they are designed to be added to a working serial code. If the compiler is told to ignore the directives, the serial program will still execute correctly. Since the serial program is unchanged, such a parallel program may be easier to debug.

A third way of expressing parallelism is to use library calls within an otherwise sequential program. The libraries perform the task of creating and terminating parallel activities, scheduling them, and supporting communication and synchronization. Examples of libraries that support this method are the POSIX threads package, which is supported by many operating systems, and the MPI libraries, which have become a standard for expressing message-passing parallel applications.

Parallel Programming Models.

Programming Vector Machines. Vector parallelism typically exploits operations that are performed on array data structures. This can be expressed using vector constructs that have been added to standard languages. For instance, Fortran90 uses constructs such as

$$A(1{:}n) = B(1{:}n) + C(1{:}n)$$

For a vector machine, this could cause a vector loop to be produced, which performs a vector add between chunks of arrays B and C, then places a vector copy of the result in a chunk of array A. The size of a chunk would be determined by the number of elements that fit into a vector register in the machine.

Loop Parallelism. Loops express repetitive execution patterns, which is where most of a program's work is performed. Parallelism is exploited by identifying loops that have independent iterations. That is, all iterations access separate data. Loop parallelism is often expressed through directives, which are placed before the first statement of the loop. OpenMP is an important example of a loop-oriented directive language. Typically, a single processor executes code between loops, but activates (*forks*) a set of processors to cooperate in executing the parallel loop. Every processor will execute a share of the loop iterations. A synchronization point (or barrier) is typically placed after the loop. When all processors arrive at the barrier, only the master processor continues. This is called a *join point* for the loop. Thus the term *fork–join parallelism* is used for loop parallelism.

Determining which processor executes which iteration of the loop is called *scheduling*. Loops may be scheduled *statically*, which means that the assignment of processors to loop iterations is fully determined prior to the execution of the loop. Loops may also be *self-scheduled*, which means that whenever a given processor is ready to execute a loop iteration, it takes the next available iteration. Other scheduling techniques will be discussed in the subsection "scheduling."

Parallel-Threads Model. If the parallel activities in a program can be packaged well in the form of subroutines that can execute independently of each other, then the *threads* model is adequate. Threads are parallel activities that are created and terminated explicitly by the program. The code executed by a thread is a specified subroutine, and the data accessed can either be private to a thread or shared with other threads. Various synchronization constructs are usually supported for coordinating parallel threads. Using the threads model, users can implement highly dynamic and flexible parallel execution schemes. The POSIX threads package is one example of a well-known library that supports this model.

The SPMD Model. Distributed-memory parallel machines are typically programmed by using the SPMD execution model. SPMD stands for "single program, multiple data." This refers to the fact that each processor executes an identical program, but on different data. One processor cannot directly access the data of another processor, but a message containing those data can be passed from one processor to the other. The MPI standard defines an important form for passing such messages. In a DMP, a processor's access to its own data is much faster than access to data of another processor through a message, so programmers typically write SPMD programs that avoid access to the data of other processors. Programs written for a DMP can be more difficult to write than programs written for an SMP, because the programmer must be much more careful about how the data are accessed.

Program Analysis

Program analysis is crucial for any optimizing compiler. The compiler writer must determine the analysis techniques to use in the compiler, according to the target machine and the type of optimization desired. For parallelization and vectorization, the compiler typically takes as input the serial form of a program, then determines which parts of the program can be transformed into parallel or vector form. The key constraint is that the results of each section of code must be the same as those of the serial program. Sometimes the compiler can parallelize a section of code in such a way that the order of operations is different than that in the serial program, causing a slightly different result. The difference may be so small as to be unimportant, or may actually alter the results in an important way. In these cases, the programmer must agree to let the compiler parallelize the code in this manner.

Some of the analysis techniques used by parallelizing compilers are also used by optimizing compilers compiling for serial machines. In this section we will generally ignore such techniques, and focus on the techniques that are unique to parallelizing and vectorizing compilers.

Dependence Analysis. A *data dependence* between two sections of a program indicates that during execution of the optimized program, those two sections of code must be run in the order indicated by the dependence. Data dependences between two sections of code that access the same memory location are classified by the type of the access (read or write) and the order, so there are four classifications:

Input Dependence	READ before READ
Antidependence	READ before WRITE
Flow Dependence	WRITE before READ
Output Dependence	WRITE before WRITE

Flow dependences are also referred to as *true* dependences. If an input dependence occurs between two sections of a program, it does not prevent the sections from running at the same time (in parallel). However, the existence of any of the other types of dependences *would* prevent the sections from running in parallel, because the results may be different from those of the serial code. Techniques have been developed for changing the original program in many situations where dependences exist, so that the sections can run in parallel. Some of them will be described later in this article.

A loop is parallelized by running its iterations in parallel, so the question must be asked whether the same storage location would be accessed in different iterations of a loop, and whether one of the accesses is a write. If so, then a data dependence exists within the loop. Data dependence within a loop is typically determined by equating the subscript expressions of each pair of references to a given array, and attempting to solve the equation (called the *dependence equation*), subject to constraints imposed by the loop bounds. For a multidimensional array, there is one dependence equation for each dimension. The dependence equations form a system of equations, the *dependence system*, which is solved simultaneously. If the compiler can find a solution to the system, or if it cannot prove that there is no solution, then it must conservatively assume that there is a solution, which would mean that a dependence exists.

Mathematical methods for solving such systems are well known if the equations are linear. That means the form of the subscript expressions is as follows:

$$\sum_{j=1}^{k} a_j i_j + a_0$$

where k is the number of loops nested around an array reference, i_j is the loop index in the jth loop in the nest, and a_j is the coefficient of the jth loop index in the expression. The dependence equation is then of the form

$$\sum_{j=1}^{k} a_j i'_j + a_0 \quad = \quad \sum_{j=1}^{k} b_j i''_j + b_0 \tag{1}$$

or

$$\sum_{j=1}^{k} (a_j i'_j - b_j i''_j) = b_0 - a_0 \tag{2}$$

In these equations, i'_j and i''_j represent the values of the jth loop index of the two subscript expressions being equated.

For instance, consider the loop below:

```
DO i = 1, 100
    A(i) = B(i)
    C(i) = A(i-1)
ENDDO
```

There are two references to array A, so we equate the subscript expressions of the two references. The equation is

$$i'_1 \; = \; i''_1 - 1$$

subject to the constraints

$$\begin{aligned} i'_1 &< i''_1 \\ 1 \le i'_1 &\le 100 \\ 1 \le i''_1 &\le 100 \end{aligned}$$

The constraint $i'_1 < i''_1$ comes from the idea that only dependences across iterations are important. A dependence within the same iteration $i'_1 \equiv i''_1$ is never a problem, since each iteration executes on a single processor, so it can be ignored.

Of course, there are many solutions to this equation that satisfy the constraints: $\{i'_1 : 1, i''_1 : 2\}$ is one; $\{i'_1 : 2, i''_1 : 3\}$ is another. Therefore, the given loop contains a dependence.

A *dependence test* is an algorithm employed to determine if a dependence exists in a section of code. The problem of finding dependence in this way has been shown to be equivalent to the problem of finding solutions to a system of Diophantine equations, which is NP-complete, meaning that only extremely expensive algorithms can be found to solve the complete problem exactly. Therefore, a large number of dependence tests have been devised that solve the problem under simplifying conditions and in special situations.

Iteration Spaces. Looping statements with pre-evaluated loop bounds, such as the Fortran do loop, have a predetermined set of values for their loop indices. This set of loop-index values is the *iteration space* of the loop. k-nested loop statements have a k-dimensional iteration space. A specific iteration within that space may be named by a k-tuple of iteration values, called an *iteration vector*:

$$\text{A(1:n)} = \text{B(1:n)} + \text{C(1:n)}$$

in which i_1 represents the outermost loop, i_2 the next inner, and i_k the innermost.

Direction and Distance Vectors. When a dependence is found in a loop nest, it is sometimes useful to characterize it by indicating the iteration vectors of the iterations where the same location is referenced. For instance, consider the following loop:

$$\{i_1, i_2, \ldots, i_k\}$$

The dependence between statements S1 and S2 happens between iterations

```
          DO i = 2, 100
            DO j = 1, 100
S1:           A(i,j) = B(i)
S2:           C(i) = A(i-1,j+3)
            ENDDO
          ENDDO
```

Since {2, 5} happens before {3, 2} in the serial execution of the loop nest, we say that the *dependence source* is {2, 5} and the *dependence sink* is {3, 2}. The *dependence distance* for a particular dependence is defined as the difference between the iteration vectors, the sink minus the source:

$$\{2, 5\} \quad \text{and} \quad \{3, 2\}, \qquad \{2, 6\} \quad \text{and} \quad \{3, 3\}, \quad \text{etc.}$$

Notice that in this example the dependence distance is constant, but this may not always be the case.

The dependence *direction vector* is also useful information, though coarser than the dependence distance. There are three directions for a dependence: $\{<, =, >\}$. The $<$ direction corresponds to a positive dependence distance, the $=$ direction corresponds to a distance of zero, and the $>$ direction corresponds to a negative dependence distance. Therefore, the direction vector for the example above would be $\{ <, > \}$.

Distance and direction vectors are used within parallelizing compilers to help determine the legality of various transformations, and to improve the efficiency of the compiler. Loop transformations that reorder the iteration space or modify the subscripts of array references within loops cannot be applied for some configurations of direction vectors. In addition, in multiply nested loops that refer to multidimensional arrays, we can test hierarchically for dependence, guided by the direction vectors, and thereby make fewer dependence tests. Distance vectors can help partially parallelize loops, even in the presence of dependences.

Exact versus Inexact Tests. There are three possible answers that any dependence test can give:

(1) *No Dependence* The compiler can prove that no dependence exists.
(2) *Dependence* The compiler can prove that a dependence exists.
(3) *Not sure* The test could neither prove nor disprove dependences. To be safe, the compiler must assume a dependence in this case. This is the *conservative assumption* for dependence testing, necessary to guarantee correct execution of the parallel program.

We call a dependence test *exact* if it only reports answers 1 or 2. Otherwise, it is *inexact*.

Dependence Tests. The first of the dependence tests was the *GCD test*, an inexact test. The GCD test finds the greatest common divisor g of the coefficients of the left-hand side of the dependence equation [Eq. (2) above]. If g does not divide the right-hand-side value of Eq. (2), then there can be no dependence. Otherwise, a dependence is still a possibility. The GCD test is cheap compared to some other dependence tests. In practice, however, often the GCD g is 1, which will always divide the right-hand side, so the GCD test does not help in those cases.

The *extreme-value test,* also inexact in the general case, has proven to be one of the most useful dependence tests. It takes the dependence equation (2) and constructs both the minimum and the maximum possible values for the left-hand side. If it can show that the right-hand side is either greater than the maximum or less than the minimum, then we know for certain that no dependence exists. Otherwise, a dependence must be assumed. A combination of the extreme-value test and the GCD test has proved to be very valuable and fast, because they complement each other very well. The GCD test does not incorporate information about the loop bounds, which the extreme-value test provides. At the same time, the extreme value test does not concern itself with the structure of the subscript expressions, which the GCD test does.

The extreme-value test is exact under certain conditions. It is exact if any of the following are true:

- All loop index coefficients are ± 1 or 0.
- The coefficient of one index variable is ± 1, and the magnitudes of all other coefficients are less than the range of that index variable.
- The coefficient of one index variable is ± 1, and there exists a permutation of the remaining index variables such that the coefficient of each is less than the sum of the products of the coefficients and ranges for all the previous index variables.

Many other dependence tests have been devised over the years. Many deal with ways of solving the dependence system when it takes certain forms. For instance, the *two-variable exact test* can find an exact solution if the dependence system is a single equation of the form

$$ai + b_j = c$$

The most general dependence test would be to use integer programming to solve a linear system—a set of equations (the dependence system) and a set of inequalities (constraints on variables due to the structure of the program). Integer programming conducts a search for a set of integer values for the variables that satisfy the linear system. *Fourier–Motzkin elimination* is one algorithm that is used to conduct the search for solutions. Its complexity is very high (exponential), so until the advent of the omega test (discussed below), it was considered too expensive to use integer programming as a dependence test.

The *lambda test* is an increased-precision form of the extreme-value test. While the extreme-value test basically checks to see whether the hyperplane formed by any of the dependence equations falls completely outside the multidimensional volume delimited by the loop bounds of the loop nest in question, the lambda test checks for the situation in which each hyperplane intersects the volume, but the intersection of all hyperplanes falls outside the volume. It is especially useful for the situation in which a single loop index appears in the subscript expression for more than one dimension of an array reference (a reference referred to as having *coupled subscripts*). If the lambda test can find that the intersection of any two dependence-equation hyperplanes falls completely outside the volume, then it can declare that there is no solution to the dependence system.

The *I test* is a combination of the GCD and extreme-value tests, but is more precise than would be the application of the two tests individually.

The *generalized GCD test*, built on Gaussian elimination (adapted for integers), attempts to solve the system of dependence equations simultaneously. It forms a matrix representation of the dependence system and then, using elementary row operations, forms a solution of the dependence system, if one exists. The solution is parametrized so that all possible solutions could be generated. The dependence distance can also be determined by this method.

The *power test* first uses the generalized GCD test. If that produces a parametrized solution to the dependence system, then it uses constraints derived from the program to determine lower and upper bounds on the free variables of the parametrized solution. Fourier–Motzkin elimination is used to combine the constraints of the program for this purpose. These extra constraints can sometimes produce an impossible result, indicating that the original parametrized solution was actually empty, disproving the dependence. The power test can also be used to test for dependence for specific direction vectors.

All of the preceding dependence tests are applicable when all coefficients and loop bounds are integer constants and the subscript expressions are all affine functions. The power test is the only test mentioned up to this point that can make use of variables as coefficients or loop bounds. In it, a variable can simply be treated as an additional unknown. The value of the variable is then expressed in terms of the free variables of the solution; then Fourier–Motzkin elimination can incorporate any constraints on that variable into the constraints on the free variables of the solution. A small number of dependence tests have been devised that can make use of variables and nonaffine subscript expressions.

The *omega test* makes use of a fast algorithm for doing Fourier–Motzkin elimination. The original dependence problem that it tries to solve consists of a set of equalities (the dependence system) and a set of inequalities (the program constraints). First, it eliminates all equality constraints (as was done in the generalized GCD test) by using a specially designed function mod to reduce the magnitude of the coefficients, until at least one reaches ± 1, when it is possible to remove the equality. Then, the set of resulting inequalities is tested to determine whether any integer solutions can be found for them. It has been shown that, for most real program situations, the omega test gives an answer quickly (in polynomial time). In some cases, however, it cannot, and it resorts to an exponential-time search.

The *range test* extends the extreme-value test to symbolic and nonlinear subscript expressions. The ranges of array locations accessed by adjacent loop iterations are symbolically compared. The range test makes use of range information for variables within the program, obtained by symbolically analyzing the program. It is able to discern data dependences in a few important situations that other tests cannot handle.

The *access region test* makes use of a symbolic representation of the array elements accessed at separate sites within a loop. It uses an intersection operation to intersect two of these symbolic access regions. If the intersection can be proven empty, then the potential dependence is disproven. The access region test likewise can test dependence when nonaffine subscript expressions are used, because in some cases it can apply simplification operators to express the regions in affine terms.

An array-subscript-expression classification system can assist dependence testing. Subscript expressions may be classified according to their structure; then choice of the dependence solution technique may be based on how the subscript expressions involved are classified. A useful classification of the subscript expression pairs involved in a dependence problem is as follows:

ZIV (Zero Index Variable) The two subscript expressions contain no index variables at all, e.g., A(1) and A(2).

SIV (Single Index Variable) The two subscript expressions contain only one loop index variable, e.g. A(i) and A(i+2).

MIV (Multiple Index Variable) The two subscript expressions contain more than one loop index variable, e.g. A(i) and A(j) or A(i+j) and A(i).

The different classifications call for unique dependence-testing methods. The SIV class is further subdivided into various special cases, each enabling a special dependence test or loop transformation.

The *delta test* makes use of these subscript expression classes. It first classifies each dependence problem according to the above types. Then, it uses a specially targeted dependence test for each case. The main insight of the Delta test is that when two array references are being tested for dependence, information derived from solving the dependence equation for one dimension may be used in solving the dependence equation for another dimension. This allows the delta test to be useful even in the presence of coupled subscripts. The algorithm attends to the SIV and ZIV equations first, since they can be solved easily. Then, the information gained is used in the solution of the MIV equations. Since the delta test does not attempt to use a single general technique to determine dependence, but rather special tests for each special case, it is possible for the delta test to accommodate unknown variable values more easily.

Run-Time Dependence Testing. It is very common for programs to make heavy use of variables whose values are read from input files. Unfortunately, such variables often contain crucial information about the dependence pattern of the program. In this situation, a perfectly parallel loop might have to be run serially, simply because the compiler lacked information about the input variables. In these cases, it is sometimes possible for the compiler to compile a test into the program that will test for certain parallelism-enabling conditions, then choose between parallel and serial code according to the result of the test. This technique of parallelization is called *run-time dependence testing*.

The *inspector–executor* model of program execution allows a compiler to run some kind of analysis of the data values in the program (the inspector), which sets up the execution, and then to execute the code according to the analysis (the executor). The inspector can do anything from dependence testing to setting up a communication pattern to be carried out by the executor. The complexity of the test needed at run time varies with the details of the loop itself. Sometimes the test needed is very simple. For instance, in the loop

```
DO i = 1, 100
    A(i + m) = B(i)
    C(i)  =  A(i)
ENDDO
```

no dependence exists in the loop if $m > 99$. The compiler might generate code that executes a parallel version of the loop if $m > 99$, otherwise a serial version.

More complicated situations might call for a more sophisticated dependence test to be performed at run time. The compiler might be able to prove all conditions for independence except one. Proof of that condition might be attempted at run time. For example, the compiler might determine that a loop is parallel if only a given array was proven to contain no duplicate values (i.e. be a *permutation vector*). If the body of the loop is large enough, then the time savings of running the loop in parallel can be substantial. It could offset the expense of checking for the permutation-vector condition at run time. In this case, the compiler might generate such a test to choose between the serial and parallel versions of the loop.

Another technique that has been employed is to attempt to run a loop in parallel despite not knowing for sure that the loop is parallel. This is called *speculative parallelization*. The preloop values of the memory locations that will be modified by the loop must be saved, because it might be determined during execution that the loop contained a dependence, in which case the results of the parallel run must be discarded and the loop must be re-executed serially. During the parallel execution of the loop, extra code is executed that can be used to determine whether a dependence really did exist in the serial version. The LRPD test is one example of such a run-time dependence test.

Interprocedural Analysis. A loop containing one or more procedure calls presents a special challenge for parallelizing compilers. The chief problem is how to compare the memory activity in different execution contexts (subroutines), for the purpose of discovering data dependences. One possibility, called *subroutine inlining*, is to remove all subroutine calls by directly replacing all subroutine calls with the code from the called subroutine, then parallelizing the whole program as one large routine. This is sometimes feasible, but often causes an explosion in the amount of source code that the compiler must compile. Inlining also faces obstacles in trying to represent the formal parameters of the subroutine in the context of the calling routine, since in some languages (Fortran is one example) it is legal to declare formal parameter arrays with dimensionality different from that in the calling routine.

The alternative to inlining is to keep the subroutine call structure intact and simply represent the memory access activity caused by a subroutine in some way at the call site. One method of doing this is to represent memory activity symbolically with sets of constraints. For instance, at the site of a subroutine call, it might be noted that the locations written were

```
DO i = 1, 100
    A(i + m = B(i)
    C(i) = A(i)
ENDDO
```

The advantage of using this method is that one can use the sets of constraints directly with a Fourier–Motzkin-based dependence test.

Several other forms for representing memory accesses have been used in various compilers. Many are based on *triplet notation*, which represents a set of memory locations in the form

$$\{A(i) \mid 0 \leq i \leq 100\}$$

This form can represent many regular access patterns, but not all.

Another representational form consists of *regular section descriptors* (*RSD*s), which uses a simple form $(I + \alpha)$, where I is a loop index and α is a loop-invariant expression. At least three other forms based on RSDs have been used: restricted RSDs (which can express access on the diagonal of an array), bounded RSDs (which express triplet notation with full symbolic expressions), and guarded array regions (which are bounded RSDs qualified with a predicate guard).

An alternative for representing memory activity interprocedurally is to use a representational format whose dimensionality is not tied to the program-declared dimensionality of a given array. An example of this type is the linear memory access descriptors used in the access-region test. This form can represent most memory access patterns used in a program, and allows one to represent memory reference activity consistently across procedure boundaries.

Symbolic Analysis. Symbolic analysis refers to the use of symbolic terms within ordinary compiler analysis. The extent to which a compiler's analysis can handle expressions containing variables is a measure of how good a compiler's symbolic analysis is. Some compilers use an integrated, custom-built symbolic analysis package, which can apply algebraic simplifications to expressions. Others depend on integrated packages, such as the omega constraint solver, to do the symbolic manipulation that they need. Still others use links to external symbolic manipulation packages, such as Mathematica or Maple. Modern parallelizing compilers generally have sophisticated symbolic analysis.

Abstract Interpretation. When compilers need to know the result of executing a section of code, they often traverse the program in execution order, keeping track of the effect of each statement. This process is called *abstract interpretation*. Since the compiler generally will not have access to the run-time values of all the variables in the program, the effect of each statement will have to be computed symbolically. The effect of a loop is easily determined when there is a fixed number of iterations (such as in a Fortran do loop). For loops that do not explicitly state a number of iterations, the effect of the iteration may be determined by *widening*, in which the values changing due to the loop are made to change as though the loop had an infinite number of iterations, and then *narrowing*, in which an attempt is made to factor in the loop exit conditions, to limit the changes due to widening. Abstract interpretation follows all control flow paths in the program.

Range Analysis. Range analysis is an application of abstract interpretation. It gathers the range of values that each variable can assume at each point in the program. The ranges gathered have been used to support the range test, as mentioned in the subsection “Dependence Analysis” above.

Data Flow Analysis. Many analysis techniques need global information about the program being compiled. A general framework for gathering this information is called *data flow analysis*. To use data flow analysis, the compiler writer must set up and solve systems of *data flow equations* that relate information at various points in a program. The whole program is traversed, and information is gathered from each program node, then used in the data flow equations. The traversal of the program can be either forward (in the same direction as normal execution would proceed) or backward (in the opposite direction from normal execution). At join points in the program’s control flow graph, the information coming from the paths that join must be combined, so the rules that govern that combination must be specified.

The data flow process proceeds in the direction specified, gathering the information by solving the data flow equations, and combining information at control flow join points until a steady state is achieved—that is, until no more changes occur in the information being calculated. When steady state is achieved, the wanted information has been propagated to each point in the program.

An example of data flow analysis is *constant propagation*. By knowing the value of a variable at a certain point in the program, the precision of compiler analysis can be improved. Constant propagation is a forward data flow problem. A value is propagated for each variable. The value gets set whenever an assignment statement in the program assigns a constant to the variable. The value remains associated with the variable until another assignment statement assigns a value to the variable. At control flow join points in the program, if a value is associated with the variable on all incoming paths, and it is always the same value, then that value stays associated with the variable. Otherwise, the value is set to “unknown.”

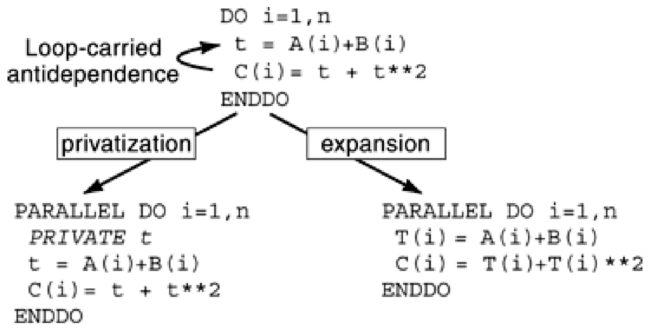
Data flow analysis can be used for many purposes within a compiler. It can be used for determining which variables are aliased to which other variables, for determining which variables are potentially modified by a given section of code, for determining which variables may be pointed to by a given pointer, and for many other purposes. Its use generally increases the precision of other compiler analyses.

Code Transformations to Aid Analysis. Sometimes program source code can be transformed in a way that encodes useful information about the program. The program can be translated into a restricted form that eliminates some of the complexities of the original program. Two examples of this are control-flow normalization and static single assignment (*SSA*) form.

Control-flow normalization is applied to a program to transform it to a form that is simpler to analyze than a program with arbitrary control flow. An example of this is the removal of GOTO statements from a Fortran program, replacing them with IF statements and looping constructs. This adds information to the program structure, which can be used by the compiler to possibly do a better job of optimizing the program.

Another example is to transform a program into SSA form. In SSA form, each variable is assigned exactly once and is only read thereafter. When a variable in the original program is assigned more than once, it is broken into multiple variables, each of which is assigned once. SSA form has the advantage that whenever a given variable is used, there is only one possible place where it was assigned, so more precise information about the value of the variable is encoded directly into the program form.

Gated SSA Form. Gated SSA form is a variant of SSA form that includes special conditional expressions (*gating expressions*) that make the representation more precise. Gated SSA form has been used for flow-sensitive array privatization analysis within a loop. Array privatization analysis requires that the compiler prove that writes to a portion of an array precede all reads to the same section of the array within the loop. When conditional branching happens within the loop, then the gating expressions can help to prove the privatization condition.



```
DO i=1,n
 t = A(i)+B(i)
 C(i)= t + t**2
ENDDO
```

```
PARALLEL DO i=1,n
 PRIVATE t
 t = A(i)+B(i)
 C(i)= t + t**2
ENDDO
```

```
PARALLEL DO i=1,n
 T(i) = A(i)+B(i)
 C(i) = T(i)+T(i)**2
ENDDO
```

Fig. 1. Data privatization and expansion.

Enabling Transformations

Like all optimizing compilers, parallelizers and vectorizers consist of a sequence of *compilation passes*. The program analysis techniques described so far are usually the first set of these passes. They gather information about the source program, which is then used by the transformation passes to make intelligent decisions. The compilers have to decide which transformations can legally and profitably be applied to which code sections and how to best orchestrate these transformations.

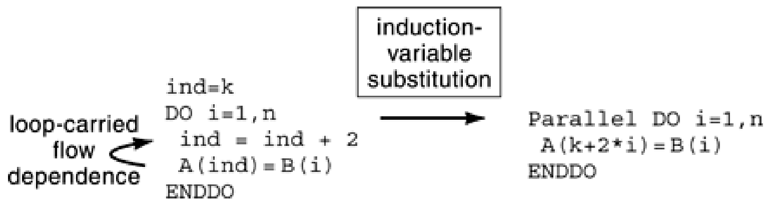
For the sake of this description we divide the transformations into two parts, those that enable other techniques and those that perform the actual vectorizing or parallelizing transformations. This division is useful for our presentation, but not strict. Some techniques will be mentioned in both places.

Dependence Elimination and Avoidance. An important class of enabling transformations deals with eliminating and avoiding data dependences. We will describe *data privatization, idiom recognition,* and *dependence-aware work partitioning* techniques.

Data Privatization and Expansion. Data privatization is one of the most important techniques, because it directly enables parallelization and it applies very widely. Data privatization can remove anti- and output dependences. These so-called storage-related or false dependences are not due to computation having to wait for data values produced by another computation. Instead, the computation must wait because it wants to assign a value to a variable that is still in use by a previous computation. The basic idea is to use a new storage location so that the new assignment does not overwrite the old value too soon. Data privatization does this as shown in Fig. 1.

In the original code, each loop iteration uses the variable t as a temporary storage. This represents a dependence, in that each iteration has to wait until the previous iteration is done using t. In the sequential execution of the program this order is guaranteed. However, in a parallel execution we would like to execute all iterations concurrently on different processors. The transformed code simply marks t as a *privatizable* variable. This instructs the code-generating compiler pass to place t into the private storage of each processor—essentially p times replicating the variable t, where p is the number of processors. *Data expansion* is an alternative implementation to privatization. Instead of marking t private, the compiler expands the scalar variable into an array and uses the loop variable as an array index.

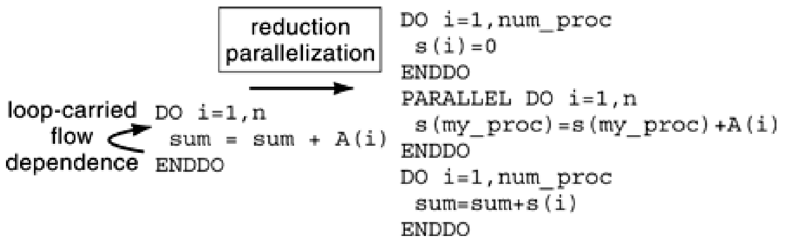
The main difficulty of data privatization and expansion is to recognize eligible variables. A variable is privatizable in a loop iteration if it is assigned before it is used. This is relatively simple to detect for scalar variables. However, the transformation is important for arrays as well. The analysis of array sections that are provably assigned before being used can be very involved and requires symbolic analysis, mentioned in the preceding section.



```
ind=k
DO i=1,n
 ind = ind + 2
 A(ind)=B(i)
ENDDO
```

```
Parallel DO i=1,n
 A(k+2*i)=B(i)
ENDDO
```

Fig. 2. Induction-variable substitution.



```
DO i=1,n
 sum = sum + A(i)
ENDDO
```

```
DO i=1,num_proc
 s(i)=0
ENDDO
PARALLEL DO i=1,n
 s(my_proc)=s(my_proc)+A(i)
ENDDO
DO i=1,num_proc
 sum=sum+s(i)
ENDDO
```

Fig. 3. Reduction parallelization.

Idiom Recognition: Reductions, Inductions, Recurrences. These transformations can remove true (i.e., flow) dependences. The elimination of situations where one computation has to wait for another to produce a needed data value is only possible if we can express the computation in a different way. Hence, the compiler recognizes certain idioms and rewrites them in a form that exhibits more parallelism.

Induction Variables. Induction variables are variables that are modified in each loop iteration in such a way that the assumed values can be expressed as a mathematical sequence. Most common are simple, additive induction variables. They get incremented in each loop iteration by a constant value, as shown in Fig. 2. In the transformed code the sequence is expressed in a closed form, in terms of the loop variable. The induction statement can then be eliminated, which removes the flow dependence.

More advanced forms of induction-variable substitution deal with multiply nested loops, coupled induction variables (which are incremented by other induction variables), and multiplicative induction variables. The identification of induction variables can be through pattern matching (e.g., the compiler finds statements that modify variables in the described way) or through abstract interpretation (identifying the sequence of values assumed by a variable in a loop).

Reduction Operations. Reduction operations abstract the values of an array into a form with lesser dimensionality. The typical example is an array being summed to a scalar variable. The parallelizing transformation is shown in Fig. 3.

The idea for enabling parallel execution of this computation exploits mathematical commutativity. We can split the array into p parts, sum them individually by different processors, and then combine the results. The transformed code has two additional loops, for initializing and combining the partial results. If the size of the main reduction loop (variable n) is large, then these loops are negligible. The main loop is fully parallel.

More advanced forms of this technique deal with array reductions, where the sum operation modifies several elements of an array instead of a scalar. Furthermore, the summation is not the only possible reduction operation. Another important reduction is finding the minimum or maximum value of an array.

Recurrences. Recurrences use the result of one or several previous loop iterations for computing the value of the next iteration. This usually forces a loop to be executed serially. However, for certain forms of linear recurrences, algorithms are known that can be parallelized. For example, in Fig. 4 the compiler has recognized a pattern of linear recurrences for which a parallel solver is known. The compiler then replaces this code by a

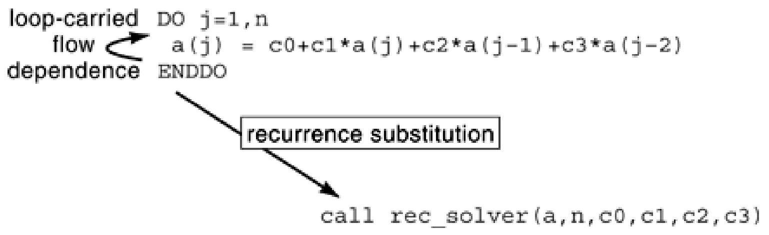

loop-carried
flow
dependence
DO j=1,n
a(j) = c0+c1*a(j)+c2*a(j-1)+c3*a(j-2)
ENDDO
recurrence substitution
call rec_solver(a,n,c0,c1,c2,c3)

Fig. 4. Recurrence substitution.

call to a mathematical library that contains the corresponding parallel solver algorithm. This substitution can pay if the array is large.

Many variants of linear recurrences are possible. A large number of library functions need to be made available to the compiler so that an effective substitution can be made in many situations.

Correctness and Performance of Idiom Recognition Techniques. Idiom recognition and substitution come at a cost. Induction-variable substitution replaces an operation with one of higher strength (usually, a multiplication replaces an addition). In fact, this is the reverse operation of strength reduction—an important technique in classical compilers. Parallel reduction operations introduce additional code. If the reduction size is small, the overhead may offset the benefit of parallel execution. This is true to a greater extent for parallel recurrence solvers. The overhead associated with parallel solvers is relatively high and can only be amortized if the recurrence is long. As a rule of thumb, induction and reduction substitution are usually beneficial, whereas the performance of a program with and without recurrence recognition should be more carefully evaluated.

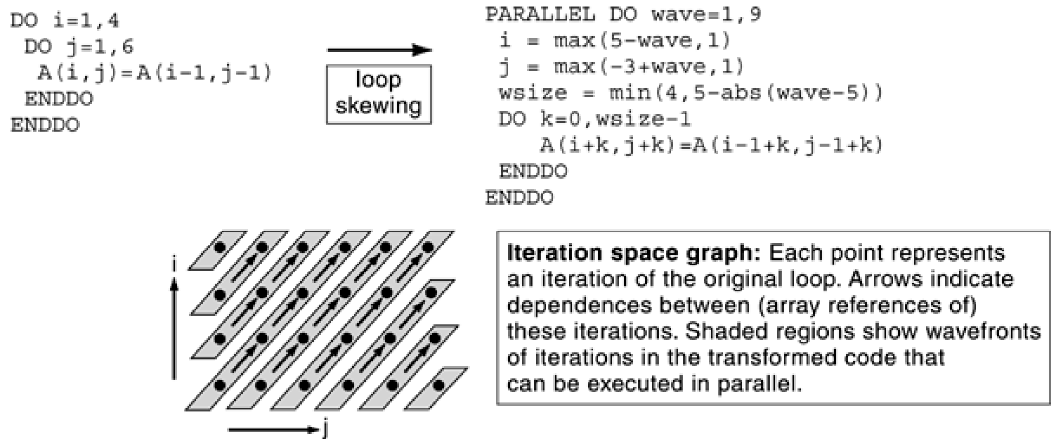
It is also important to note that, although parallel recurrence and reduction solvers perform mathematically correct transformations, there may be roundoff errors because of the limited computer representation of floating-point numbers. This may lead to inaccurate results in application programs that are numerically very sensitive to reordering operations. Compilers usually provide command-line options so that the user can control these transformations.

Dependence-Aware Work Partitioning: Skewing, Distribution, Uniformization. The above three techniques are able to remove data dependences and, in this way, generate fully parallel loops. If dependences cannot be removed, it may still be possible to generate parallel computation. The computation may be reordered so that expressions that are data-dependent on each other are executed by the same processor. Figure 5 shows an example loop and its iteration dependence graph.

By regrouping the iterations of the loop as indicated by the shaded *wavefronts* in the iteration-space graph, all dependences stay within the same processor, where they are enforced by the sequential execution of this processor. This technique is called *loop skewing*. The class of unimodular transformations contains more general techniques that can reorder loop iterations according to various criteria, such as dependence considerations and locality of reference (locality optimizations will be discussed in the subsection "Parallel-Loop Restructuring.")

Other techniques can find partial parallelism in loops that contain data dependences. For example, loop distribution may split a loop into two loops. One of them contains all dependent statements and must execute serially, while the other one is fully parallel. Another example is dependence uniformization, which tries to find minimum dependence distances. If all dependence distances are greater than a threshold *t*, then *t* consecutive iterations can be executed in parallel.

Enabling and Enhancing Other Transformations. Another class of enabling transformations contains prerequisite techniques for other transformations and techniques that allow others to be applied more effectively. Some transformations belong to both the enabling and enabled techniques. Because of this we will only give an overview. The following two sections will describe details of some of the techniques.



```
DO i=1,4
 DO j=1,6
  A(i,j)=A(i-1,j-1)
 ENDDO
ENDDO
```

```
PARALLEL DO wave=1,9
 i = max(5-wave,1)
 j = max(-3+wave,1)
 wsize = min(4,5-abs(wave-5))
 DO k=0,wsize-1
    A(i+k,j+k)=A(i-1+k,j-1+k)
 ENDDO
ENDDO
```

Iteration space graph: Each point represents an iteration of the original loop. Arrows indicate dependences between (array references of) these iterations. Shaded regions show wavefronts of iterations in the transformed code that can be executed in parallel.

Fig. 5. Partitioning the iteration space in the wavefront manner.

Various transformations require statements to be reordered. This can result in dependence distances getting shorter (the producing and consuming statements of a value are moved closer together), the points of use and reuse of a variable being moved closer together (which improves cache locality), or backwards dependences being turned into forward dependences.

Loop distribution splits loops into two or more loops that can be optimized individually. It also enables vectorization, discussed next. Interchanging two nested loops can help the vectorization techniques, which usually act on the innermost loop. It can also enhance parallelization, because moving a parallel loop to an outer position increases the amount of work inside the parallel region.

Splitting a single loop into a nest of two loops is called stripmining or loop blocking. It enables the exploitation of hierarchical parallelism (e.g., the inner loop may then be executed across a multiprocessor, while the outer loop gets executed by a cluster of such multiprocessors.) It is also an important cache optimization, as we will discuss.

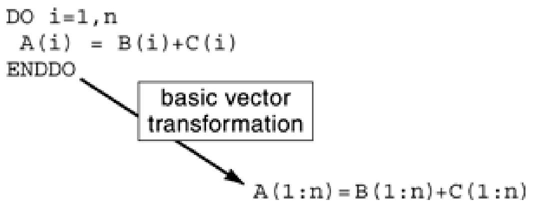
Vectorization: Exploiting Vector Architectures

Vectorizing compilers exploit vector architectures by generating code that performs operations on a number of data in a row. This was of great interest in classical supercomputers, which were built as vector architectures. In addition, vectorization has enjoyed renewed interest in modern microprocessors, which can accommodate several short data in one word. For example, a 64-bit word can accommodate a vector of 16 4-bit words. Instructions that operate on vectors of this kind are sometimes referred to as multimedia extensions (*MMX*s).

The objective of a vectorizing compiler is to identify and express such vector operations in a form that can then be easily mapped onto the vector instructions available in these architectures. A simple example is shown in Fig. 6. The following transformations aid vectorization in more complex program patterns.

Scalar Expansion. Private variables, introduced in the subsection "Dependence Elimination and Avoidance" above need to be expanded in order to allow vectorization. The following shows the privatization example of that subsection transformed into vector form:

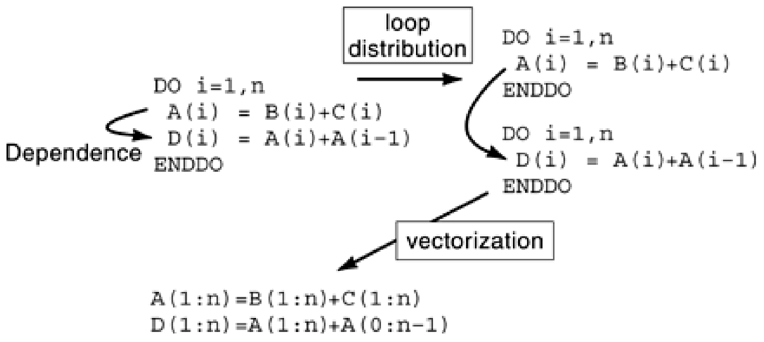
$$T(1:n) = A(1:n) + B(1:n)$$
$$C(1:n) = T(1:n) + T(1:n)^{**}2$$



```
DO i=1,n
 A(i) = B(i)+C(i)
ENDDO
```

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

Fig. 6. Basic vectorization.

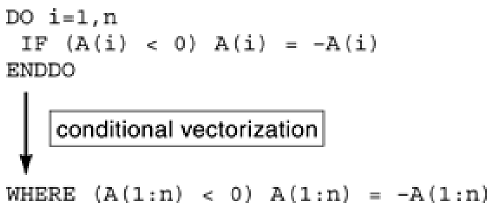


```
DO i=1,n
 A(i) = B(i)+C(i)
 D(i) = A(i)+A(i-1)
ENDDO
```

```
DO i=1,n
 A(i) = B(i)+C(i)
ENDDO
DO i=1,n
 D(i) = A(i)+A(i-1)
ENDDO
```

```
A(1:n)=B(1:n)+C(1:n)
D(1:n)=A(1:n)+A(0:n-1)
```

Fig. 7. Loop distribution enables vectorization.



```
DO i=1,n
 IF (A(i) < 0) A(i) = -A(i)
ENDDO
```

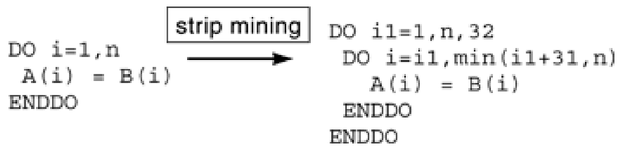
```
WHERE (A(1:n) < 0) A(1:n) = -A(1:n)
```

Fig. 8. Vectorization in the presence of conditionals.

Loop Distribution. A loop containing several statements must first be distributed into several loops before each one can be turned into a vector operation. Loop distribution (also called loop splitting or loop fission) is only possible if there is no dependence in a lexically backward direction. Statements can be reordered to avoid backwards dependences, unless there is a dependence cycle (a forward and a backward dependence that form a loop). Figure 7 shows a loop that is distributed and vectorized. The original loop contains a dependence in a lexically forward direction. Such a dependence does not prevent loop distribution. That is, the execution order of the two dependent statements is maintained in the vectorized code.

Handling Conditionals in a Loop. Conditional execution is an issue for vectorization because all elements in a vector are processed in the same way. Figure 8 shows how a conditional execution can be vectorized. The conditional is first evaluated for all vector elements, and a vector of true–false values is formed, called the *mask*. The actual operation is then executed conditionally, based on the value of the mask at each vector position.

Strip mining Vector Lengths. Vector instructions usually take operands of length 2^n—the size of the vector registers. The original loop must be divided into strips of this length. This is called *strip mining*. In Fig. 9, the iterations have been broken down into strips of length 32.



```
DO i=1,n
 A(i) = B(i)
ENDDO
```

```
DO i1=1,n,32
 DO i=i1,min(i1+31,n)
   A(i) = B(i)
 ENDDO
ENDDO
```

Fig. 9. Strip mining a loop into two nested loops.

Vector Code Generation. Finding vectorizable statements in a multiply nested loop that contains data dependences can be quite difficult in the general case. Algorithms are known that perform this operation in a recursive manner. They move from the outermost to the innermost loop level and test at each level for code sections that can be distributed (i.e., do not contain dependence cycles) and then vectorized, as described. Code sections with dependence cycles are inspected recursively at inner loop levels.

Parallelization: Exploiting Multiprocessors

Parallelizing compilers have been most successful on SMPs. Additional techniques are necessary for transforming programs onto DMPs. In this section we will first describe techniques that apply to both machine classes and then present techniques specific to DMPs.

Although very similar analysis techniques are used, parallelization differs substantially from vectorization. For example, data privatization is expressed by adding the variables to a *private list*, instead of applying scalar expansion. Loop distribution and stripmining are not a prerequisite, because the computation does not need to be reordered (although this can be done as an optimization, as will be discussed). Conditionals do not need special handling, because different processors can directly execute different code sections.

The most important sources of parallelism for multiprocessors are iterations of loops, such as do loops in Fortran programs and for loops in C programs. We will present techniques for detecting that loop iterations can correctly and effectively be executed in parallel. We will also briefly mention techniques for exploiting partial parallelism in loops and in nonloop program constructs.

All parallelizing compiler techniques have to deal with two general issues: (1) they must be provably correct and (2) they must improve the performance of the generated code, relative to a serial execution on one processor. The correctness of techniques is often stated by formally defining data-dependence patterns under which a given transformation is legal. While such correctness proofs exist for most of today's compiler capabilities, they often require the compiler to make conservative assumptions, as described above.

The second issue is no less complex. Assessing performance improvement involves the assumption of a machine model. For example, one must assume that a parallel loop will incur a start–terminate overhead. Hence, it will not execute n times faster on an n-processor machine than on one processor. Its parallel execution time is no less than $t_{1-\text{processor}}/n + t_{\text{overhead}}$. For small loops this can be more than the serial execution time. Unfortunately, even the most advanced compilers sometimes do not have enough information to make such performance predictions. This is because they do not have sufficient information about properties of the target machine and about the program's input data.

Parallelism Recognition.

Exploiting Fully Parallel Loops. Basic parallel-code generation for multiprocessors entails identifying loops that have no loop-carried dependences, and then marking these loops as parallelizable. Data-dependence analysis and all its enabling techniques for program analysis and dependence removal is most important in this process. Iterations of parallelizable loops are then assigned to the different processors for execution. This second step may be taken through various methods. The parallelizing compiler may be directly coupled with a code-generating compiler that issues the actual machine code. Alternatively, the parallelizer can be a preprocessor, outputting the source program annotated with information about which loops can be executed in parallel. A backend compiler then reads this program form and generates code according to the preprocessor's directives.

Exploiting Partial Loop Parallelism. Partial parallelism can also be exploited in loops with true dependences that cannot be removed. The basic idea is to enforce the original execution order of the dependent program statements. Parallelism is still exploited as described above, but each dependent statement now waits for a go-ahead signal telling it that the needed data value has been produced by a prior iteration. The successful implementation of this scheme relies on efficient hardware synchronization mechanisms.

Compilers can reduce the waiting time of dependent statements by moving the source and sink of a dependence closer to each other. Statement reordering techniques are important to achieve this effect. In addition, because every synchronization introduces overhead, reducing the number of synchronization points is important. This can be done by eliminating redundant synchronizations (i.e., synchronizations that are covered by other synchronizations) or by serializing a code section. Note that there are many tradeoffs for the compiler to make. For example, it has to decide when it is more profitable to serialize a code section than to execute it in parallel with many synchronizations.

Nonloop Parallelism. Loops are not the only source of parallelism. Straight-line code can be broken up into independent sections, which can then be executed in parallel. For building such parallel sections, a compiler can, for example, group all statements that are mutually data-dependent into one section. This results in several sections between which there are no data dependences. Applying this scheme at small basic code blocks is important for instruction-level parallelization, to be discussed later. At a larger scale, such parallel regions can include entire subroutines, which can be assigned to different processors for execution.

More complex is exploiting parallelism in the repetitive pattern of a recursion. Recursion splitting techniques can transform a recursive algorithm into a loop, which can then be analyzed with the already described means.

Nonloop parallelism is important for instruction-level parallelization. It is of lesser importance for multiprocessors, because the degree of parallelism in loops is usually much higher than in straight-line code sections. Furthermore, since the most prevalent parallelization technology is found in compilers for nonrecursive languages, such as Fortran, there has not been a pressing need to deal with recursive program patterns.

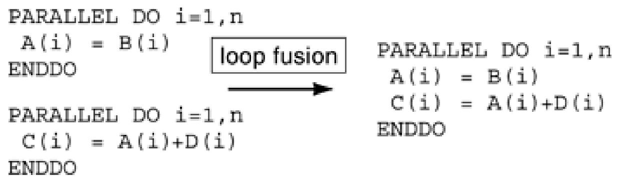
Parallel-Loop Restructuring. Once parallel loops are detected, there are several loop transformations that can optimize the program such that it (1) exploits the available resources in an optimal way and (2) minimizes overheads.

Increasing Granularity. A parallel computation usually incurs an overhead when starting and terminating. For example, starting and ending a parallel loop comes at a run-time cost, sometimes referred to as loop fork–join overhead. The larger the computation in the loop, the better this overhead can be amortized.

The techniques loop fusion, loop coalescing, and loop interchange can all increase the granularity of parallel loops by increasing the computation between the fork and join points. Each transformation comes with potential overhead, which must be considered in the profitability decision of the compiler.

Loop fusion combines two adjacent loops into a single loop. It is the reverse transformation of loop distribution and is subject to similar legality considerations. Fusion is straightforward if the loop bounds of the two candidates match, as shown in Fig. 10. Several techniques can adjust these bounds if necessary. The compiler may peel iterations (split off a number of iterations into a separate loop), reverse iterations (loop iterates from upper to lower bound), or normalize the loops (loop iterates from 0 to some new upper bound with a stride of one). These adjustments may cause overhead because they introduce new loops (loop peeling) or may lead to more complex subscript expressions.

Loop coalescing merges two nested loops into a single loop. Figure 11 shows an example. This transformation has additional benefits, such as increasing the number of iterations for better load balancing and exploiting two levels of loop parallelism even if the underlying machine supports only one level. Loop coalescing

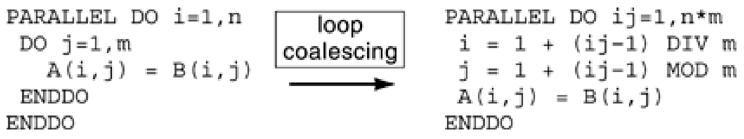


```
PARALLEL DO i=1,n
 A(i) = B(i)
ENDDO

PARALLEL DO i=1,n
 C(i) = A(i)+D(i)
ENDDO
```

```
PARALLEL DO i=1,n
 A(i) = B(i)
 C(i) = A(i)+D(i)
ENDDO
```

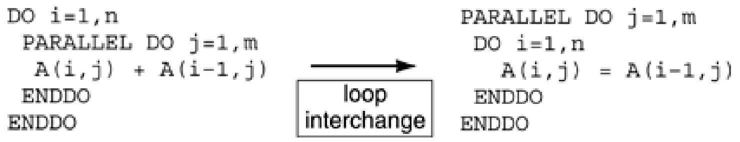
Fig. 10. Fusing two loops into one.



```
PARALLEL DO i=1,n
 DO j=1,m
   A(i,j) = B(i,j)
 ENDDO
ENDDO
```

```
PARALLEL DO ij=1,n*m
 i = 1 + (ij-1) DIV m
 j = 1 + (ij-1) MOD m
 A(i,j) = B(i,j)
ENDDO
```

Fig. 11. Coalescing two nested loops.



```
DO i=1,n
 PARALLEL DO j=1,m
  A(i,j) + A(i-1,j)
 ENDDO
ENDDO
```

```
PARALLEL DO j=1,m
 DO i=1,n
   A(i,j) = A(i-1,j)
 ENDDO
ENDDO
```

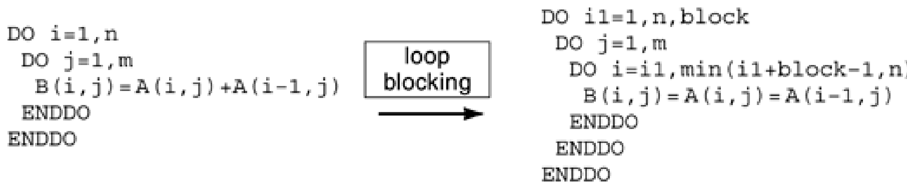
Fig. 12. Moving an inner parallel loop to an outer position.

introduces overhead, because it needs to introduce additional expressions for reconstructing the original loop index variables from the index of the combined loop. Again, benefits and overheads must be compared by the compiler.

Loop interchanging can increase the granularity significantly by moving an inner parallel loop to an outer position in a loop nest. This technique is shown in Fig. 12. As a result, the loop fork–join overhead is only incurred once overall instead of once per iteration of the outer loop. Loop interchanging is also subject to legality considerations, which are formulated as rules on data-dependence patterns that permit or disallow the transformation. For example, interchange is illegal if a dependence that is carried by the outer loop goes from a later iteration of the inner loop to an earlier iteration of the same loop. The interchanged loop would reverse the order of the two dependent iterations. In data-dependence terminology, one cannot interchange two loops if the dependence with respect to the outer loop has a forward direction ($<$) while the dependence with respect to the inner loop has a backward direction ($>$).

If the granularity of a parallel loop cannot be increased above the profitability threshold, it is better to execute the loop serially. Compile-time performance estimation capabilities are critical for this purpose. They rely on all the program analysis techniques that can determine the values assumed by certain variables, such as loop bounds. They also include machine parameters, for example the profitability threshold for parallel loops, of which the loop fork–join overhead is an important factor. In general, it is not possible to evaluate the profitability at compile time. One solution to this problem is that the compiler formulates conditional expressions that will be evaluated at run time and decide when the parallel execution is profitable. This can be implemented through two-version loops: the conditional expression forms the condition of an IF statement, which chooses between the parallel and serial versions of the loop.

Reducing Memory Latency. Techniques to reduce or hide memory access latencies are increasingly important, because the speed of computation in modern processors increases more rapidly than the speed of memory accesses. The primary hardware mechanism that supports latency reductions is the *cache*. It keeps



```
DO i=1,n
 DO j=1,m
  B(i,j)=A(i,j)+A(i-1,j)
 ENDDO
ENDDO
```

```
DO i1=1,n,block
 DO j=1,m
  DO i=i1,min(i1+block-1,n)
   B(i,j)=A(i,j)=A(i-1,j)
  ENDDO
 ENDDO
ENDDO
```

Fig. 13. Loop blocking to increase cache locality.

copies of memory cells in fast storage, close to the processor, so that repeated accesses incur much lower latencies (which is referred to as temporal locality). In addition, caches fetch multiple words from memory in one transfer, so that accesses to adjacent memory elements hit in cache as well (spatial locality). Compiler techniques try to reorder computation so that the temporal and spatial locality of the program is increased. While this is already important in compilers for single-processor machines, there are additional considerations in multiprocessors. This is because of the need to keep multiple caches coherent and because of the interaction between locality and parallelism.

Loop interchange is one of the most effective transformations for increasing spatial locality. It can change the order of the computation so that it performs stride-1 references. That is, adjacent iterations access adjacent memory cells. Note that this transformation may be different for different programming languages. For example, Fortran programs place the leftmost dimension of an array in contiguous memory (referred to as column major order), whereas C programs use row major order. Therefore, Fortran compilers will try to move the loop that accesses the leftmost dimension of an array to the innermost position in a loop nest. C compilers will do the same with the rightmost dimension. The loop interchange example shown above achieves this effect as well. However, note that the two goals of obtaining stride-1 references and increasing granularity may conflict. In this case, the compiler will have to estimate and compare the performance of the two program variants.

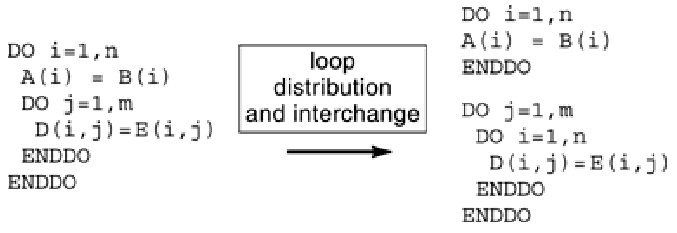
Another important cache optimization technique is *loop blocking*, which is basically the same as strip-mining, introduced above. By dividing a computation into several blocks, we can reorder it so that the use and reuse of a datum are moved closer to each other. It is then more likely that the datum is still in cache and has not been evicted by other computation before it is reused. Hence, loop blocking can increase temporal cache reuse. Figure 13 gives an example of this transformation.

Loop tiling is a more general form of reordering computation that can increase cache reuse. The iteration space of a multiply nested loop is divided into a number of tiles. Each tile is then executed by one processor. Tiling has several goals. In addition to increasing cache locality, it can partition the computation according to the dependence structure in order to identify parallel loops. This was described above as dependence-aware work partitioning.

Other transformations can influence cache locality. For example, *loop fusion* binds pairs of iterations from adjacent loops to each other. They are then guaranteed to execute on the same processor. This can increase cache reuse across the two loops.

Loop distribution can be an enabling transformation for cache optimization. For example, the code in Fig. 14 shows a non-perfectly-nested loop that is distributed into a single loop and a perfectly nested double loop. The loop nest is then interchanged to obtain stride-1 accesses.

There are more advanced techniques to reduce memory latency, which are less widely used and not generally supported by today's computer architectures. They include compiler cache management and prefetch operations. Software cache management controls cache operations explicitly. The compiler inserts instructions that flush the cache content, select cache strategies, and control which data sections are cached. Prefetch operations aim at transferring data into cache or into a dedicated prefetch buffer before the executing program requests it, so that the data are available in fast storage when needed.

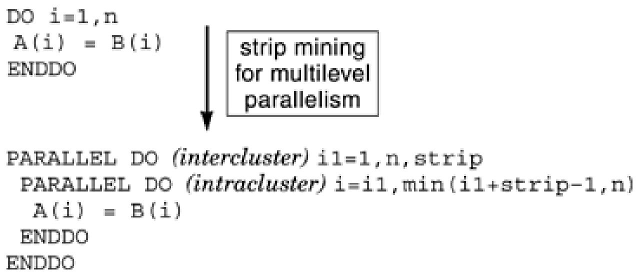


```
DO i=1,n
 A(i) = B(i)
 DO j=1,m
  D(i,j)=E(i,j)
 ENDDO
ENDDO
```

```
DO i=1,n
A(i) = B(i)
ENDDO

DO j=1,m
 DO i=1,n
  D(i,j)=E(i,j)
 ENDDO
ENDDO
```

Fig. 14. Loop distribution enables interchange.



```
DO i=1,n
 A(i) = B(i)
ENDDO
```

```
PARALLEL DO (intercluster) i1=1,n,strip
 PARALLEL DO (intracluster) i=i1,min(i1+strip-1,n)
  A(i) = B(i)
 ENDDO
ENDDO
```

Fig. 15. Strip mining enables two-level parallelism.

Multilevel Parallelization. Most multiprocessors today offer one-level parallelism. That means that, usually, a single loop out of a loop nest can be executed in parallel. Architectures that have a hierarchical structure can exploit two or more levels of parallelism. For example, a cluster of multiprocessors may be able to execute two nested parallel loops, the outer one across the clusters while the inner loop employs the processors within each cluster.

Program sections that contain singly nested loops can be turned into two-level parallelism by strip mining them into two nested loops as shown in Fig. 15.

Scheduling. After parallelism is detected and loops are restructured for optimal performance, there is still the issue of defining an execution order and assigning parallel activities to processors. This must be done in a way that (1) balances the load, (2) performs computation where the necessary resources are, and (3) considers the environment. Scheduling decisions can be done at compile time (statically) or at run time (dynamically). Both methods have advantages and disadvantages.

Load balancing is the primary reason for dynamic scheduling. Static scheduling methods typically split up the loop iterations into equal chunks and assign them to the different processors. This works well if the loop iterations are equal in size. However, that is not the case in loops that contain conditional statements or in an inner loop of a nest whose number of iterations depends on an outer loop variable. Dynamic scheduling methods assign loop iterations to processors, a chunk at a time. The chunk can contain one or more iterations. Of special interest also are scheduling schemes that vary the chunk size, such as trapezoidal scheduling and guided self-scheduling methods. Dynamic scheduling methods come with some run-time overhead for performing the scheduling action. However, this is often negligible compared to the gain from load balancing.

The goal of computing where the resources are, on the other hand, favors static scheduling methods. In heterogeneous systems it is mandatory to perform the computation where necessary processor capabilities or input/output devices are. In multiprocessors, data are critical resources, whose location may determine the best executing processor. For example, if a datum is known to be in the cache of a specific processor, then it is best to execute other computations accessing the same datum on the same processor as well. The compiler has knowledge of which computation accesses which data in the future. Hence, such scheduling decisions are good

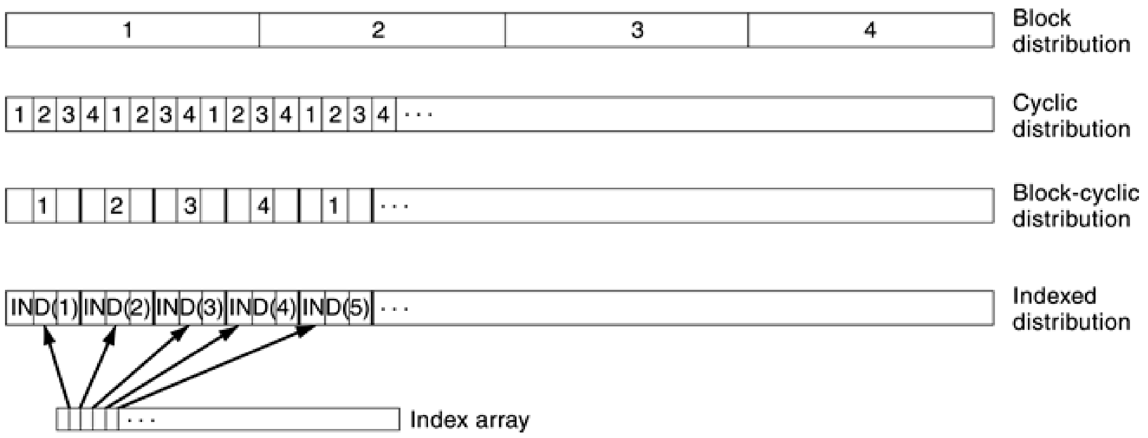


Fig. 16. Data distribution schemes. Numbers indicate the nodes of a four-processor DMP on which the array section is placed.

to make at compile time. In distributed-memory systems this situation is even more pronounced. Accessing a datum on a processor other than its owner involves communication with high latencies.

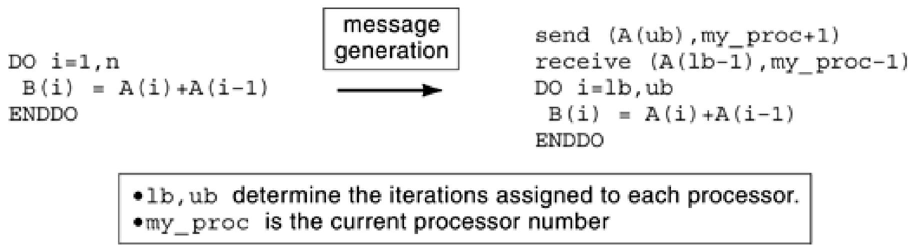
Scheduling decisions also depend on the environment of the machine. For example, in a single-user environment it may be best to statically schedule computation that is known to execute evenly. However, if the same program is executed in a multiuser environment, the load of the processors can be very uneven, making dynamic scheduling methods the better option.

Techniques Specific to Distributed-Memory Processors. DMPs do not provide a shared address space. Many of the techniques described so far assume that all computation can see the necessary data, no matter where it is performed. In DMPs this is no longer the case. The compiler needs to distribute the program's data onto several compute nodes, and data that are needed by nodes other than their home need to be communicated by sending and receiving messages. This creates two major new tasks for the compiler: (1) finding a good data partitioning and distribution scheme, and (2) orchestrating the communication between the nodes in an optimal way.

Data Partitioning and Distribution. The goal of data partitioning and distribution is to place each datum on the compute node that accesses it most frequently. Data partitioning and distribution are often performed as two or more steps in a compiler. For simplicity we describe it here as one data distribution step. Several issues need to be resolved. First, the proper units of data distribution need to be determined. Typically these are sections of arrays. Second, data access costs and frequencies need to be estimated to compare distribution alternatives. Third, if there are program sections with different access patterns, redistribution may need to be considered at run time.

The simplest data distribution scheme is block distribution, which divides an array into p sections, where p is the number of processors. Block distribution creates contiguous array sections on each processor. If adjacent processors access adjacent array elements, then cyclic distribution is appropriate. Block-cyclic distribution is a combination of both schemes. For irregular access patterns, indexed distribution may be most appropriate. Figure 16 illustrates these distribution schemes.

A major difficulty is that data distribution decisions affect all program sections accessing a given datum. Hence, global optimizations need to be performed, which can be algorithmically complex. Furthermore, compile-time information about array accesses is often incomplete. Better global optimization would require knowledge of the program input data. Also, distribution decisions cannot be made in isolation. They need to factor in available parallelism and the cost of messages, described next. Finally, indexed distribution, although most



```
DO i=1,n
 B(i) = A(i)+A(i-1)
ENDDO
```

```
send (A(ub),my_proc+1)
receive (A(lb-1),my_proc-1)
DO i=lb,ub
 B(i) = A(i)+A(i-1)
ENDDO
```

- `lb,ub` determine the iterations assigned to each processor.
- `my_proc` is the current processor number

Fig. 17. Generating messages for data exchange in a distributed-memory machine.

flexible, may incur additional overhead because the index array may need to be distributed itself, causing double latencies for each array access. Because of all this, developing compiler techniques for automatic data partitioning and distribution is still an active research topic. Many current parallel programming approaches assume that the user assists the compiler in this process.

Message Generation. Once the owner processor for each datum is identified, the compiler needs to determine which accesses are from remote processors and then insert communication to and from these processors. The most common form of communication is to send and receive messages, which can communicate one or several data between two specific processors. More complex communication primitives are also known, such as broadcasts (send to all processors) and reductions (receive from all processors and combine the results in some form).

The basic idea of message generation is simple. Each statement needs to communicate to/from remote processors the accessed data elements that are not local. Often, the *owner-computes* principle is assumed. For example, assignment statements are executed on the processor that owns the left-hand-side element. (Note that this execution scheme is different from the one assumed for SMPs, where an entire loop iteration is executed by one and the same processor.) Data-partitioning information hence supplies both the information about which processor executes which statement and that about which data are local and which remote. Figure 17 shows the basic messages generated. It assumes a block partitioning of both arrays, A and B.

Although generating messages in this scheme would lead to a functionally correct program, this program may be inefficient. To increase the efficiency, the compiler needs to aggregate communication. That is, messages generated for individual statements need to be combined into a larger message. Also, messages may be moved to an earlier point in the instruction stream, so that communication latencies can be overlapped with computation. Message aggregation for the general block-cyclic distribution is already rather complex. It is made even more difficult because message sizes may only be known in the form of symbolic expressions. For indexed distributions, support through “communication libraries for irregular computation” has been an active research topic. In general, compilers that deal with the issues of message generation for DMPs are highly sophisticated and complex.

Exploiting Parallelism at the Instruction Level

Instruction-level parallelism (*ILP*) refers to the processor’s ability to execute several instructions at the same time. Instruction-level parallelism can be exploited implicitly by the processor without the compiler issuing special directives or instructions to the hardware. Or the compiler can extract parallelism explicitly and express it in the generated code. Examples of the latter type of generated code are *very-long instruction word* (*VLIW*) and *explicitly parallel instruction computing* (*EPIC*) architectures. In addition

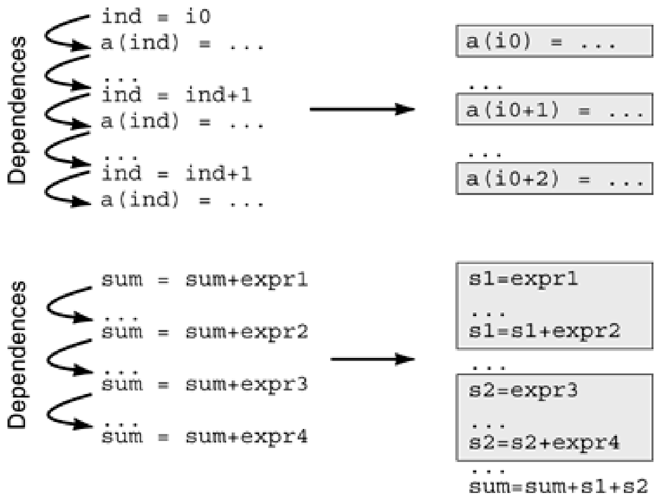


Fig. 18. Dependence-removing transformation for instruction-level parallelism. Shaded blocks of instructions are independent of each other and can be executed in parallel.

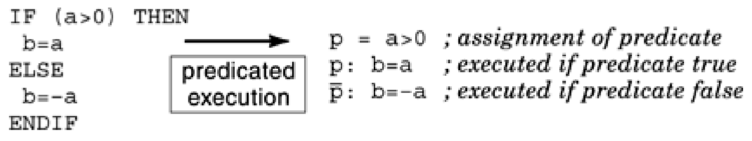
to the techniques presented here, all or most techniques known from classical compilers are important and are often applied as a first set of transformations and analysis passes in ILP compilers (see Program Compilers).

Implicit Instruction-Level Parallelism. For implicit instruction-level parallelism, the compiler-generated code can be the same as for single-issue machines. However, knowing the processor's ILP mechanisms, the compiler can change the code so that the processor can do a more effective job. Three categories of techniques are important for this objective: (1) scheduling instructions, (2) removing dependences, and (3) increasing the window size within which the processor can exploit parallel instructions.

Instruction Scheduling. Modern processors exploit ILP by starting a new instruction before an earlier instruction has completed. All instructions begin their execution in the order defined by the program. This order can have a significant effect on performance. Hence, an important task of the compiler is to specify a good order. It does this by moving instructions that incur long latencies prior to those with short latencies. Such instruction scheduling is subject to data-dependence constraints. For example, an instruction consuming a value in a register or memory must not be moved before an instruction producing that value. Instruction scheduling is a well-established technology, discussed in standard compiler textbooks. We refer the reader to such literature for further details.

Removing Dependences. The basic patterns for removing dependences are similar to the ones discussed for loop-level parallelism. Antidependences can be removed through variable-renaming techniques. In addition, register renaming becomes important. It avoids conflicts between potentially parallel sets of instructions that make use of the same register for storing temporary values. Such techniques can conflict with good register allocation in sequential instruction streams, where nonoverlapping lifetimes of different variables are assigned to the same register. Because of this, the compiler may rely on hardware register-renaming capabilities available in the processor.

Similarly to the induction-variable recognition technique discussed above, the compiler can replace incremental operations through operations that use operands available at the beginning of a parallel code section. Likewise, a sequence of sum operations may be replaced by sum operations into a temporary variable, followed by an update step at the end of the parallel region. Figure 18 illustrates these transformations.



```
IF (a>0) THEN
 b=a
ELSE
 b=-a
ENDIF
```

```
p = a>0   ; assignment of predicate
p: b=a    ; executed if predicate true
p̄: b=-a   ; executed if predicate false
```

Fig. 19. Generating predicated code.

Increasing the Window Size. A large window of instructions within which the processor can discover and exploit ILP is important for two reasons. First, it leads to more opportunities for parallelism, and second, it reduces the relative cost of starting the instruction pipeline, which usually happens at window boundaries.

Window boundaries are typically branch instructions. Instruction analysis and parallel execution cannot easily cross branch instructions, because the processor does not know what instructions will execute after the branch until it is reached. For example, an instruction may only execute on the true branch of a conditional jump. Hence, although the instruction could in that case be executed in parallel with another instruction before the branch, this is not feasible. If the false branch is taken, the instruction might have written an undesired value to memory. Even if all side effects of such instructions were kept in temporary storage, they might raise exceptions that could incorrectly abort program execution.

There are several techniques for increasing the window size. Code motion techniques can move instructions across branches under certain conditions. For example, if the same instruction appears on both branches, it can be moved before the branch, subject to data-dependence considerations. In this way, the basic block on the critical path can be increased, or another basic block can be completely removed.

Instruction *predication* can also remove a branch by assigning the branch condition to a mask, which then guards the merged statements of both branches. Figure 19 shows an example. Predicated execution needs hardware support, and the compiler must trade off the benefits of enhanced ILP with the overhead of more executed instructions.

Basically, ILP is exploited in straight-line code sections. Additional performance can be gained when exploiting parallelism across loop iterations. A straightforward way to achieve this is to unroll loop iterations, that is, to replicate the loop body by a factor n (and divide the number of iterations by the same factor). Many techniques that were discussed under loop optimizations for multiprocessors are applicable to this case as well.

Explicit Instruction-Level Parallelism. The basic techniques for removing dependences and increasing the window size are important for explicit ILP as well. However, the goal is no longer to expose more parallelism to the hardware detection mechanisms, but to make parallelism more analyzable by the compiler itself. As an example of explicit ILP we choose a VLIW architecture and a software pipelining technique. The goal is to find a repetitive instruction pattern in a loop that can be mapped efficiently to a sequence of VLIW instructions. It is called *pipelining* because the execution may “borrow” instructions from earlier or later iterations to fill an efficient schedule. Hence the execution of parts of different iterations overlap with each other. This is illustrated in Fig. 20. The reader may notice that there would be a conflict in register use between the overlapping loop iterations. For example, in the same VLIW instruction $s4$ uses $R0$’s value of one loop iteration, while $s2$ uses $R0$’s value belonging to the next iteration. Both software and hardware register renaming techniques are known to solve this problem.

Efficient parallelism at the instruction level depends on the compiler’s ability to identify code sequences that are executed with high probability. The instructions of these code sequences must then be reordered so that the processor’s functional units are exploited in an optimal way. The two tasks are sometimes called *trace selection* and *trace compaction*, respectively. Trace selection is a global optimization in that it looks for code sequences across multiple branches, factoring in estimated branch frequences. Trace compaction can move instructions across branches, whereby it may add bookkeeping code to ensure correct execution if a predicted branch is not taken. The two techniques together are referred to as *trace scheduling*.

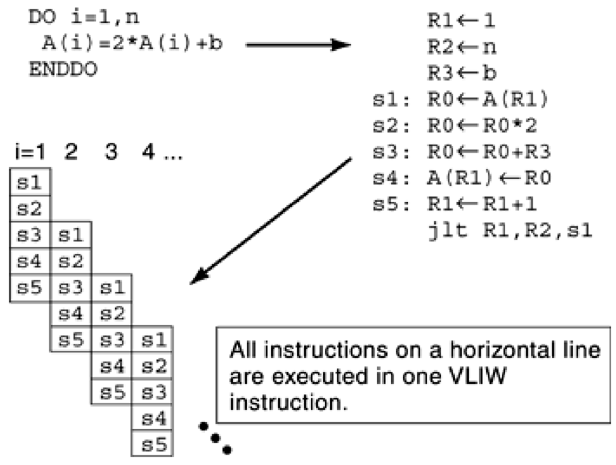


Fig. 20. Translating a loop in a software pipelining scheme.

Compiler-Internal Concerns

Compiler developers have to resolve a number of issues other than designing analysis and transformation techniques. These issues become important when creating a complete compiler implementation, in which the described techniques are integrated into a user-friendly tool. Several compiler research infrastructures have played pioneering roles in this regard. Among them are the Parafrase (1,2), PFC (3), PTRAN (4), ParaScope (5), Polaris (6), and SUIF (7) compilers. The following subsections describe a number of issues that have to be addressed by such infrastructures. An adequate compiler-internal representation of the program must be chosen, the large number of transformation passes need to be put in the proper order, and decisions have to be made about where to apply which transformation, so as to maximize the benefits but keep the compilation time within bounds. The user interface of the compiler is important as well. Optimizing compilers typically come with a large set of command-line flags, which should be presented in a form that makes them as easy to use as possible.

Internal Representation. A large variety of compiler-internal program representations (*IR*s) are in use. They differ with respect to the level of program translation and the type of program analysis information that is implicitly represented. Several IRs may be used for several phases of the compilation. The *syntax-tree* IR represents the program at a level that is close to the original program. At the other end of the spectrum are representations close to the generated machine code. An example of an IR in between these extremes is the *register transfer language*, which is used by the widely available GNU C compiler. Source-level transformations, such as loop analysis and transformations, are usually applied on an IR at the level of the syntax-tree, whereas instruction-level transformations are applied on an IR that is closer to the generated machine code. Examples of representations that include analysis information are the *static single assignment* (*SSA*) form and the *program dependence graph* (*PDG*). SSA was introduced in the section 4 "Program Analysis." The PDG includes information about both data dependences and control dependences in a common representation. It facilitates transformations that need to deal with both types of dependences at the same time.

Phase Ordering. Many compiler techniques are applied in an obvious order. Data-dependence analysis needs to come before parallel-loop recognition, and so do dependence-removing transformations. Other techniques mutually influence each other. We have introduced several techniques where this situation occurs. For example, loop blocking for cache locality also involved loop interchanging, and loop interchanging was made possible through loop distribution. There are many situations where the order of transformations is not easy to determine. One possible solution is for the compiler to generate internally a large number of program variants and then estimate their performance. We have already described the difficulty of performance estimation. In addition, generating a large number of program variants may get prohibitively expensive in terms of both compiler execution time and space need. Practical solutions to the phase-ordering problem are based on heuristics and ad hoc strategies. Finding better solutions is still a research issue. One approach is for the compiler to decide on the applicability of several transformations at once. This is the goal of *unimodular* transformations, which can determine a best combination of iteration-reordering techniques, subject to data dependence constraints.

Applying Transformations at the Right Place. One of the most difficult problems for compilers is to decide when and where to apply a specific technique. In addition to the phase-ordering problem, there is the issue that most transformations can degrade performance if applied to the wrong program section. For example, a very small loop may run slower in parallel than serially. Interchanging two loops may increase the parallel granularity but reduce data locality. Stripmining for multilevel parallelism may introduce more overhead than benefit if the loop has a small number of iterations. This difficulty is increased by the fact that machine architectures are getting more complex, requiring specialized compiler transformations in many situations. Furthermore, an increasing number of compiler techniques are being developed that apply to a specific program pattern, but not in general. For reasons discussed before, the compiler does not always have sufficient information about the program input data and machine parameters to make optimal decisions.

Speed versus Degree of Optimization. Ordinary compilers transform medium-size programs in a few seconds. This is not so for parallelizing compilers. Advanced program analysis methods, such as data-dependence analysis and symbolic-range analysis, may take significantly longer. In addition, as mentioned above, compilers may need to create several optimization variants of a program and then pick the one with the best estimated performance. This can further multiply the compilation time. It raises a new issue in that the compiler now needs to make decisions about which program sections to optimize to the fullest of its capabilities and where to save compilation time. One way of resolving this issue is to pass the decision on to the user, in the form of command-line flags.

Compiler Command-Line Flags. Ideally, from the user's point of view—and as an ultimate research goal—a compiler would not need any command-line flags. It would make all decisions about where to apply which optimization technique fully automatically. Today's compilers are far from this goal. Compiler flags can be seen as one way for the compiler to gather additional knowledge that is unavailable in the program source code. They may supply information that otherwise would come from program input data (e.g., the most frequently executed program sections), from the machine environment (e.g., the cache size), or from application needs (e.g., degree of permitted roundoff error). They may also express user preferences (e.g., compilation speed versus degree of optimization). A parallelizing compiler can include several tens of command-line options. Reducing this number can be seen as an important goal for the future generation of vectorizing and parallelizing compilers.

BIBLIOGRAPHY

1. D. J. Kuck *et al.* The structure of an advanced vectorizer for pipelined processors, *Proc. COMPSAC 80, The 4th Int. Computer Software Applications Conf.*, New York: IEEE Computer Society Press 1980, pp. 709–715.
2. C. Polychronopoulos *et al.* Parafrase-2: A new generation parallelizing compiler, *Proc. 1989 Int. Conf. Parallel Processing*, St. Charles, IL, 1989, Vol. II, pp. 39–48.
3. J. R. Allen K. Kennedy PFC: A program to convert Fortran to parallel form, in K. Hwang (ed.), *Supercomputers: Design and Applications*, IEEE Computer Society Press, 1985, pp. 186–205.
4. F. Allen *et al.* An overview of the PTRAN analysis system for multiprocessing, *Proc. Int. Conf. Supercomputing*, 1987, 194–211. Lecture Notes in Computer Science, 297, New York: Springer Verlag.

5. V. Balasundaram *et al.* The ParaScope editor: An interactive parallel programming tool, *Proc. Int. Conf. Supercomputing*, New York: ACM, 1989, pp. 540–550.
6. W. Blume *et al.* Parallel programming with Polaris, *IEEE Computer*, **29** (12): 78–82, 1996.
7. M. W. Hall *et al.* Maximizing multiprocessor performance with the SUIF compiler, *IEEE Computer*, **29** (12): 84–89, 1996.

READING LIST

G. S. Almasi A. Gottlieb *Highly Parallel Computing*, 2nd ed. Redwood City, CA: Benjamin/Cummings, 1994.

U. Banerjee *et al.* Automatic program parallelization, *Proc. IEEE*, **81** (2): 211–243, 1993.

U. Banerjee *Dependence Analysis*, Boston: Kluwer Academic, 1997.

D. A. Patterson J. L. Hennessy *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco: Morgan Kaufmann, 1996.

K. Kennedy *Advanced Compiling for High Performance*, San Francisco: Morgan Kaufmann, 2001.

D. J. Kuck *High Performance Computing, Challenges for Future Systems*, New York: Oxford Univ. Press, 1996.

B. R. Rau J. A. Fisher Instruction-level parallel processing: History, overview, and perspective, *J. Supercomput.*, **7**: 9–50, 1993.

M. Wolfe *High-Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

H. Zima B. Chapman *Supercompilers for Parallel and Vector Computers*, Reading, MA: Addison-Wesley, 1991.

RUDOLF EIGENMANN
Purdue University
JAY HOEFLINGER
Intel