Part IV

Software for Parallel Programming

Chapter 10

Parallel Models, Languages, and Compilers

Chapter 11

Parallel Program Development and Environments

Summary

Part IV discusses software and programming requirements of parallel/vector computers. We begin with a characterization of parallel programming models: shared-variable, message-passing, data-parallel, object-oriented, functional, logic, and heterogeneous. Then we evaluate parallel languages and compiler technologies for parallel programming. This includes the study of language features, programming environments, compilers for parallelization and vectorization, and performance tuning. We will describe locks, semaphores, monitors, synchronization, multitasking, and various program decomposition techniques.

10

Parallel Models, Languages, and Compilers

This chapter is devoted to programming and compiler aspects of parallel and vector computers. To study beyond architectural capabilities, we must learn about the basic models for parallel programming and how to design optimizing compilers for parallelism. Models studied include those for shared-variable, message-passing, object-oriented, data-parallel, functional, and logic programming. We examine language extensions, parallelizing, vectorizing, and trace-driven compilers designed to support parallel programming.

10.1 PARALLEL PROGRAMMING MODELS

A programming model is a collection of program abstractions providing a programmer a simplified and transparent view of the computer hardware/software system. Parallel programming models are specifically designed for multiprocessors, multicomputers, or vector/SIMD computers. Five models are characterized below for these computers that exploit parallelism with different execution paradigms.

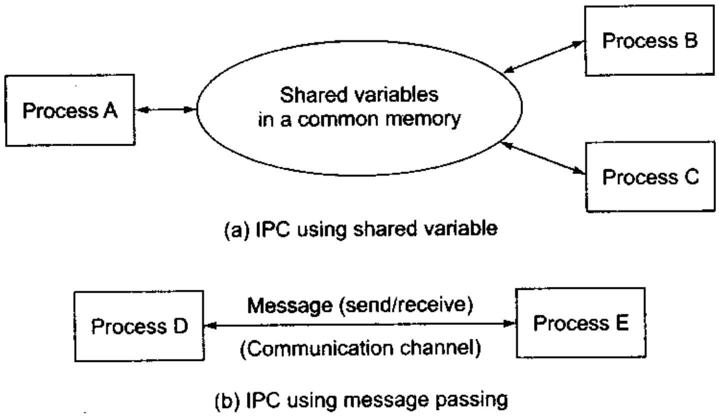
10.1.1 Shared-Variable Model

In all programming systems, we consider processors active resources and memory and I/O devices passive resources. The basic computational units in a parallel program are *processes* corresponding to operations performed by related code segments. The granularity of a process may vary in different programming models and applications.

A *program* is a collection of processes. Parallelism depends on how interprocess communication (IPC) is implemented. Fundamental issues in parallel programming are centered around the *specification*, *creation*, *suspension*, *reactivation*, *migration*, *termination*, and *synchronization* of concurrent processes residing in the same or different processors.

By limiting the scope and access rights, the process address space may be shared or restricted. To ensure orderly IPC, a mutual exclusion property requires the exclusive access of a shared object by one process at a time. We address these issues and explore their solutions below.

Shared-Variable Communication Multiprocessor programming is based on the use of shared variables in a common memory for IPC. As depicted in Fig. 10.1a, shared-variable IPC demands the use of shared memory and mutual exclusion among multiple processes accessing the same set of variables.



(a) IPC using shared variable

(b) IPC using message passing

Fig. 10.1 Two basic mechanisms for interprocess communication (IPC).

Fine-grain MIMD parallelism is exploited in tightly coupled multiprocessors. Interprocessor synchronization can be implemented either unconditionally or conditionally, depending on the mechanisms used.

The main issues in using this model include protected access of critical sections, memory consistency, atomicity of memory operations, fast synchronization, shared data structures, and fast data movement techniques, to be studied in Section 10.2.

Critical Section A *critical section* (CS) is a code segment accessing shared variables, which must be executed by only one process at a time and which, once started, must be completed without interruption. In other words, a CS operation is indivisible and satisfies the following requirements:

- *Mutual exclusion*—At most one process executing the CS at a time.
- *No deadlock in waiting*—No circular wait by two or more processes trying to enter the CS; at least one will succeed.
- *Nonpreemption*—No interrupt until completion, once entered the CS.
- *Eventual entry*—A process attempting to enter its CS will eventually succeed.

Protected Access The main problem associated with the use of a CS is avoiding race conditions where concurrent processes executing in different orders produce different results. The granularity of a CS affects the performance. If the boundary of a CS is too large, it may limit parallelism due to excessive waiting by competing processes.

When the CS is too small, it may add unnecessary code complexity or software overhead. The trick is to shorten a heavy-duty CS or to use conditional CSs to maintain a balanced performance.

In Chapter 11, we will study shared variables in the form of *locks* for implementing mutual exclusion in CSs. *Binary* and *counting semaphores* are used to implement CSs and to avoid system deadlocks. *Monitors* are suitable for structured programming.

Shared-variable programming requires special atomic operations for IPC, new language constructs for expressing parallelism, compilation support for exploiting parallelism, and OS support for scheduling parallel events and avoiding resource conflicts. Of course, all of these depend on the memory consistency model used.

Shared-memory multiprocessors use shared variables for interprocessor communications. Multiprocessing takes various forms, depending on the number of users and the granularity of divided computations. Four operational modes used in programming multiprocessor systems are specified below:

Multiprogramming Traditionally, *multiprogramming* is defined as multiple independent programs running on a single processor or on a multiprocessor by time-sharing use of the system resources. A multiprocessor can be used in solving a single large problem or in running multiple programs across the processors.

A multiprogrammed multiprocessor allows multiple programs to run concurrently through time-sharing of all the processors in the system. Multiple programs are interleaved in their CPU and I/O activities. When a program enters I/O mode, the processor switches to another program. Therefore, multiprogramming is not restricted to a multiprocessor. Even on a single processor, multiprogramming is usually implemented.

Multiprocessing When multiprogramming is implemented at the process level on a multiprocessor, it is called *multiprocessing*. Two types of multiprocessing are specified below. If interprocessor communications are handled at the instruction level, the multiprocessor operates in MIMD mode. If interprocessor communications are handled at the program, subroutine, or procedural level, the machine operates in MPMD (*multiple programs over multiple data streams*) mode.

In other words, we define MIMD multiprocessing with fine-grain instruction-level parallelism. MPMD multiprocessing exploits coarse-grain procedure-level parallelism. In both multiprocessing modes, shared variables are used to achieve interprocessor communication. This is quite different from the operations implemented on a message-passing system.

Multitasking A single program can be partitioned into multiple interrelated tasks concurrently executed on a multiprocessor. This has been implemented as *multitasking* on Cray multiprocessors. Thus multitasking provides the parallel execution of two or more parts of a single program. A job efficiently multitasked requires less execution time. Multitasking is achieved with added codes in the original program in order to provide proper linkage and synchronization of divided tasks.

Tradeoffs do exist between multitasking and not multitasking. Only when overhead is short should multitasking be practiced. Sometimes, not all parts of a program can be divided into parallel tasks. Therefore, multitasking tradeoffs must be analyzed before implementation. Section 11.2 will treat this issue.

Multithreading The traditional UNIX/OS has a single-threaded kernel in which only one process can receive OS kernel service at a time. In a multiprocessor as studied in Chapter 9, we want to extend the single kernel to be multithreaded. The purpose is to allow multiple *threads* of lightweight processes to share the same address space and to be executed by the same or different processors simultaneously.

The concept of *multithreading* is an extension of the concepts of multitasking and multiprocessing. The purpose is to exploit fine-grain parallelism in modern multiprocessors built with multiple-context processors or superscalar processors with multiple-instruction issues. Each thread will use a separate program counter. Resource conflicts are the major problem to be resolved in a multithreaded architecture.

The levels of sophistication in securing data coherence and in preserving event order increase from monoprogramming to multitasking, to multiprogramming, to multiprocessing, and to multithreading in that order. Memory management and special protection mechanisms must be developed to ensure correctness and data integrity in parallel thread operations.

Partitioning and Replication The goal of parallel processing is to exploit parallelism as much as possible with the lowest overhead. *Program partitioning* is a technique for decomposing a large program and data set into many small pieces for parallel execution by multiple processors.

Program partitioning involves both programmers and the compiler. Parallelism detection by users is often explicitly expressed with parallel language constructs. Program restructuring techniques can be used to transform sequential programs into a parallel form more suitable for multiprocessors. Ideally, this transformation should be carried out automatically by a compiler.

Program replication refers to duplication of the same program code for parallel execution on multiple processors over different data sets. Partitioning is often practiced on a shared-memory multiprocessor system, while replication is more suitable for distributed-memory message-passing multicomputers.

So far, only special program constructs, such as independent loops and independent scalar operations, have been successfully parallelized. Clustering of independent scalar operations into vector or VLIW instructions is another approach toward this end.

Scheduling and Synchronization Scheduling of divided program modules on parallel processors is much more complicated than scheduling of sequential programs on a uniprocessor. *Static scheduling* is conducted at post-compile time. Its advantage is low overhead but the shortcoming is a possible mismatch with the run-time profile of each task and therefore potentially poor resource utilization.

Dynamic scheduling catches the run-time conditions. However, dynamic scheduling requires fast context switching, preemption, and much more OS support. The advantages of dynamic scheduling include better resource utilization at the expense of higher scheduling overhead. Static and dynamic methods can be jointly used in a sophisticated multiprocessor system demanding higher efficiency.

In a conventional UNIX system, *interprocessor communication* (IPC) is conducted at the process level. Processes can be created by any processor. All processes asynchronously accessing the shared data must be protected so that only one is allowed to access the shared writable data at a time. This *mutual exclusion* property is enforced with the use of locks, semaphores, and monitors to be described in Chapter 11.

At the control level, virtual program counters can be assigned to different processes or threads. Counting semaphores or barrier counters can be used to indicate the completion of parallel branch activities. One can also use atomic memory operations such as *Test&Set* and *Fetch&Add* to achieve synchronization. Software-implemented synchronization may require longer overhead. Hardware barriers or combining networks can be used to reduce the synchronization time.

Cache Coherence and Protection Besides maintaining data coherence in a memory hierarchy, multiprocessors must assume data consistency between private caches and the shared memory. The multicache coherence problem demands an invalidation or update after each write operation. These coherence control operations require special bus or network protocols for implementation as noted in previous chapters. A memory system is said to be *coherent* if the value returned on a read instruction is always the value written by the latest write instruction on the same memory location. The access order to the caches and to the main memory makes a big difference in computational results.

The shared memory of a multiprocessor can be used in various consistency models as discussed in Chapters 4 and 9. Sequential consistency demands that all memory accesses be strongly ordered on a global basis. A processor cannot issue an access until the most recently shared writable memory access has been globally performed. A weak consistency model enforces ordering and coherence at explicit synchronization points only. Programming with the processor consistency or release consistency may be more restricted, but memory performance is expected to improve.

10.1.2 Message-Passing Model

Multicomputer programming is depicted in Fig. 10.1b. Two processes D and E residing at different processor nodes may communicate with each other by passing messages through a direct or indirect network. The messages may be instructions, data, synchronization, or interrupt signals, etc. The communication delay caused by message passing is much longer than that caused by accessing shared variables in a common memory. Multicomputers are considered loosely coupled multiprocessors. Two message-passing programming models are introduced below. Techniques for message-passing programming are treated in Sections 11.4 and 11.5. Message Passing Interface (MPI) is discussed in Chapter 13.

Synchronous Message Passing Since there is no shared memory, there is no need for mutual exclusion. *Synchronous message* passing must synchronize the sender process and the receiver process in time and space, just like a telephone call using circuit-switched lines. In general, no buffers are used in the communication channels. That is why synchronous communication can be blocked by channels being busy or in error since only one message is allowed to be transmittted via a channel at a time.

In a *synchronous* paradigm, the passing of a message must synchronize the sending process and the receiving process in time and space. Besides having a time connection, the sender and receiver must also be linked by physical communication channels in space. A path of channels must be ready to enable the message passing between them.

In other words, the sender and receiver must be coupled in both time and space synchronously. If one process is ready to communicate and the other is not, the one that is ready must be blocked (or wait). In this sense, synchronous communication has been also called a blocking communication scheme.

Asynchronous Message Passing Asynchronous communication does not require that message sending and receiving be synchronized in time and space. Buffers are often used in channels, which results in nonblocking in message passing provided sufficiently large buffers are used or the network traffic is not saturated.

However, arbitrary communication delays may be experienced because the sender may not know if and when the message has been received until acknowledgment is received from the receiver. This scheme is like a postal service using mailboxes (channel buffers) with no synchronization between senders and receivers.

Nonblocking can be achieved by *asynchronous message passing* in which two processes do not have to be synchronized either in time or in space. The sender is allowed to send a message without blocking, regardless of whether the receiver is ready or not.

Asynchronous communication requires the use of buffers to hold the messages along the path of the connecting channels. Since channel buffers are finite, the sender will eventually be blocked. In a synchronous multicomputer, buffers are not needed because only one message is allowed to pass through a channel at a time.

The critical issue in programming this model is how to distribute or duplicate the program codes and data sets over the processing nodes. Tradeoffs between computation time and communication overhead must be considered.

As explained in Chapter 9, fine-grain concurrent programming with global naming was aimed at merging the shared-variable and message-passing mechanisms for heterogeneous processing.

Distributing the Computations Program replication and data distribution are used in multicomputers. The processors in a multicomputer (or a NORMA machine) are loosely coupled in the sense that they do not share memory. Message passing in a multicomputer is handled at the subprogram level rather than at the instructional or fine-grain process level as in a tightly coupled multiprocessor. That is why explicit parallelism is more attractive for multicomputers.

Example 10.1 A concurrent program for distributed computing on a multicomputer (Justin Rattner, Intel Scientific Computers, 1990)

The computation involved is the evaluation of π as the area under the curve $f(x)$ between 0 and 1 as shown in Fig. 10.2. Using a rectangle rule, we write the integral in discrete form:

$$\pi = \int_0^1 f(x)\,dx = \int_0^1 \frac{4}{1+x^2}\,dx \doteq h\sum_{i=1}^{n} f(x_i)$$

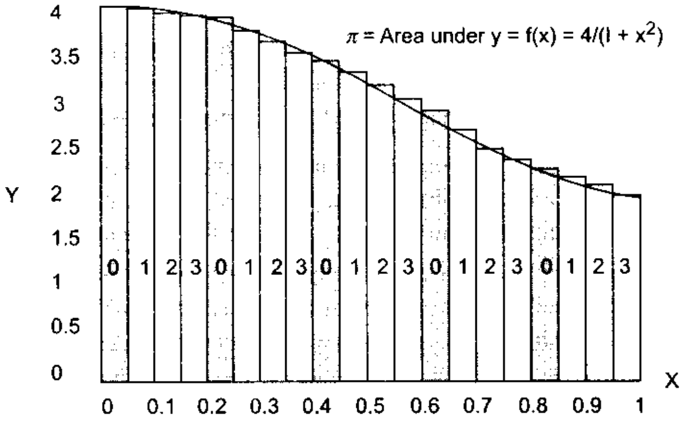


Fig. 10.2 Domain decomposition for concurrent programming on a multicomputer with four processors

where $h = 1/n$ is the panel width, $x_i = h(i - 0.5)$ are the midpoints, and n is the number of panels (rectangles) to be computed.

Assume a four-node multicomputer with four processors labeled 0, 1, 2, and 3. The rectangle rule decomposition is shown with $n = 20$ and $h = 1/20 = 0.05$. Each processor node is assigned to compute the areas of five rectangular panels. Therefore, the computational load of all four nodes is balanced.

```
Host program             Node program
input(n)                 p = numnodes()
send( n,allnodes)        me = mynode()
recv(Pi)                 recv(n)
output(Pi)               h = 1.0/n
                         sum = 0
                         Do i = me + 1, n, p
                             x = h × (i – 0.5)
                             sum = sum + f(x)
                         End Do
                         pi = h × sum
                         gop('+', Pi, host)
```

Each node executes a separate copy of the node program. Several system calls are used to achieve message passing between the host and the nodes. The host program *sends* the number of panels *n* as a message to all the nodes, which receive it accordingly in the node program. The commands *numnodes* and *mynode* specify how big the system is and which node it is, respectively.

The software for the iPSC system offers a global summing operation *gop*('+', pi, host) which iteratively pairs nodes that exchange their current partial sums. Each partial sum received from another node is added to the sum at the receiving node, and the new sum is sent out in the next round of message exchange.

Eventually, all the nodes accumulate the global sum multiplied by the height (pi = h × sum) which will be returned to the host for printout. Not all pairs of node communications need to be carried out. Only $\log_2 N$ rounds of message exchanges are required to compute the adder-tree operations, where N is the number of nodes in the system.This point will be further elaborated in Chapter 13.

10.1.3 Data-Parallel Model

With the lockstep operations in SIMD computers, the data-parallel code is easier to write and to debug because parallelism is explicitly handled by hardware synchronization and flow control. Data-parallel languages are modified directly from standard serial programming languages. For example, Fortran 90 is specially tailored for data parallelism. Thinking Machines' C* was specially designed for programming the erstwhile Connection Machines.

Data-parallel programs require the use of pre-distributed data sets. Thus the choice of parallel data structures makes a big difference in data-parallel programming. Interconnected data structures are also needed to facilitate data exchange operations. In summary, data-parallel programming emphasizes local computations and data routing operations (such as permutation, replication, reduction, and parallel prefix). It is applied to fine-grain problems using regular grids, stencils, and multidimensional signal/image data sets.

Data parallelism can be implemented either on SIMD computers or on SPMD multicomputers, depending on the grain size and operation mode adopted. In this section, we consider mainly parallel programming on SIMD computers that emphasize fine-grain data parallelism under synchronous control. Data parallelism often leads to a high degree of parallelism involving thousands of data operations concurrently. This is rather different from control parallelism which offers a much lower degree of parallelism at the instruction level.

Synchronization of data-parallel operations is done at compile time rather than at run time. Hardware synchronization is enforced by the control unit to carry out the lockstep execution of SIMD programs. We address below instruction/data broadcast, masking, and data-routing operations separately. Languages, compilers, and the conversion of SIMD programs to run on MIMD multicomputers are also discussed.

Data Parallelism Ever since the introduction of the Illiac IV computer, programming SIMD array processors has been a challenge for computational scientists. The main difficulty in using the Illiac IV had been to match the problem size with the fixed machine size. In other words, large arrays or matrices had to be partitioned into 64-element segments before they could be effectively processed by the 64 processing elements (PEs) in the Illiac IV machine.

A latter SIMD computer, the Connection Machine CM-2, offered bit-slice fine-grain data parallelism using 16,384 PEs concurrently in a single-array configuration. This demanded a lower degree of array segmentation and thus offered higher flexibility in programming.

Synchronous SIMD programming differs from asynchronous MIMD programming in that all PEs in an SIMD computer operate in a lockstep fashion, whereas all processors in an MIMD computer execute different instructions asynchronously. As a result, SIMD computers do not have the mutual exclusion or synchronization problems associated with multiprocessors or multicomputers.

Instead, inter-PE communications are directly controlled by hardware. Besides lockstep in computing operations among all PEs, inter-PE data communication is also carried out in lockstep. These synchronized instruction executions and data-routing operations make SIMD computers rather efficient in exploring spatial parallelism in large arrays, grids, or meshes of data.

In an SIMD program, scalar instructions are directly executed by the control unit. Vector instructions are broadcast to all processing elements. Vector operands are loaded into the PEs from local memories simultaneously using a global address with different offsets in local index registers. Vector stores can be executed in a similar manner. Constant data can be broadcast to all PEs simultaneously.

A masking pattern (binary vector) can be set under program control so that PEs can be enabled or disabled dynamically in any instruction cycle. Masking instructions are directly supported by hardware. Data-routing vector operations are supported by an inter-PE routing network, which is also under program control on a dynamic basis.

Array Language Extensions Array extensions in data-parallel languages are represented by high-level data types. We will specify Fortran 90 array notations in Section 10.2.2. The array syntax enables the removal of some nested loops in the code and should reflect the architecture of the array processor.

Examples of array processing languages are CFD for the Illiac IV, DAP Fortran for the AMT/Distributed Array Processor, C* for the TMC/Connection Machine, and MPF for the MasPar family of massively parallel computers.

An SIMD programming language should have a global address space, which obviates the need for explicit data routing between PEs. The array extensions should have the ability to make the number of PEs a function of the problem size rather than a function of the target machine.

Connection Machine C* language satisfied these requirements nicely. A Pascal-based language, *Actus*, was developed by R.H. Perrott for problem-oriented SIMD programming. *Actus* offered hardware transparency, application flexibility, and explicit control structures in both program structuring and data typing operations.

Compiler Support To support data-parallel programming, the array language expressions and their optimizing compilers must be embedded in familiar standards such as Fortran 77, Fortran 90, and C. The idea is to unify the program execution model, facilitate precise control of massively parallel hardware, and enable incremental migration to data-parallel execution.

Compiler-optimized control of SIMD machine hardware allows the programmer to drive the PE array transparently. The compiler must separate the program into scalar and parallel components and integrate with the OS environment.

The compiler technology must allow array extensions to optimize data placement, minimize data movement, and virtualize the dimensions of the PE array. The compiler generates data-parallel machine code to perform operations on arrays.

Array sectioning allows a programmer to reference a section or a region of a multidimensional array. Array sections are designated by specifying a start index, a bound, and a stride. *Vector-valued subscripts* are often used to construct arrays from arbitrary permutations of another array. These expressions are vectors that map the desired elements into the target array. They facilitate the implementation of *gather* and *scatter* operations on a vector of indices.

SIMD programs can in theory be recompiled for MIMD architecture. The idea is to develop a source-to-source precompiler to convert, for example, from Connection Machine C* programs to C programs running on an nCUBE message-passing multicomputer in SPMD mode.

In fact, SPMD programs are a special class of SIMD programs which emphasize medium-grain parallelism and synchronization at the subprogram level rather than at the instruction level. In this sense, the data-parallel programming model applies to both synchronous SIMD and loosely coupled MIMD computers. Program conversion between different machine architectures is needed to broaden software portability. The parallel programming paradigm based on openMP standard is described in Chapter 13.

10.1.4 Object-Oriented Model

If one considers special language features and their implications, additional models for parallel programming can be introduced. An object-oriented programming model is characterized below.

In this model, *objects* are dynamically created and manipulated. Processing is performed by sending and receiving messages among objects. Concurrent programming models are built up from low-level objects such as processes, queues, and semaphores into high-level objects like monitors and program modules.

Concurrent OOP The popularity of *object-oriented programming* (OOP) is attributed to three application demands: First, there is increased use of interacting processes by individual users, such as the use of multiple windows. Second, workstation networks have become a cost-effective mechanism for resource sharing and distributed problem solving. Third, multiprocessor technology in several variants has advanced to the point of providing supercomputing power at a fraction of the traditional cost.

As a matter of fact, program abstraction leads to program modularity and software reusability as is commonly exprienced with OOP. Other areas that have encouraged the growth of OOP include the development of CAD (computer-aided design) tools and other sophisticated applications with graphics capabilities.

Objects are program entities which encapsulate data and operations into single computational units. It turns out that concurrency is a natural consequence of the concept of objects. In fact, the concurrent use of coroutines in conventional programming is very similar to the concurrent manipulation of objects in OOP.

The development of *concurrent object-oriented programming* (COOP) provides an alternative model for concurrent computing on multiprocessors or on multicomputers. Various object models differ in the internal behavior of objects and in how they interact with each other.

An Actor Model COOP must support patterns of reuse and classification, for example, through the use of inheritance which allows all instances of a particular class to share the same property. An *actor model* developed at MIT is presented as one framework for COOP.

Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. In an actor model, message passing is attached with semantics. Basic actor primitives include:

(1) *Create*: Creating an actor from a behavior description and a set of parameters.

(2) *Send-to*: Sending a message to another actor.

(3) *Become*: An actor replacing its own behavior by a new behavior.

State changes are specified by behavior replacement. The replacement mechanism allows one to aggregate changes and to avoid unnecessary control-flow dependences. Concurrent computations are visualized in terms of concurrent actor creations, simultaneous communication events, and behavior replacements. Each message may cause an object (actor) to modify its state, create new objects, and send new messages.

Concurrency control structures represent particular patterns of message passing. The actor primitives provide a low-level description of concurrent systems. High-level constructs are also needed for raising the granularity of descriptions and for encapsulating faults. The actor model is particularly suitable for multicomputer implementations.

Parallelism in COOP Three common patterns of parallelism have been found in the practice of COOP. First, *pipeline concurrency* involves the overlapped enumeration of successive solutions and concurrent testing of the solutions as they emerge from an evaluation pipeline.

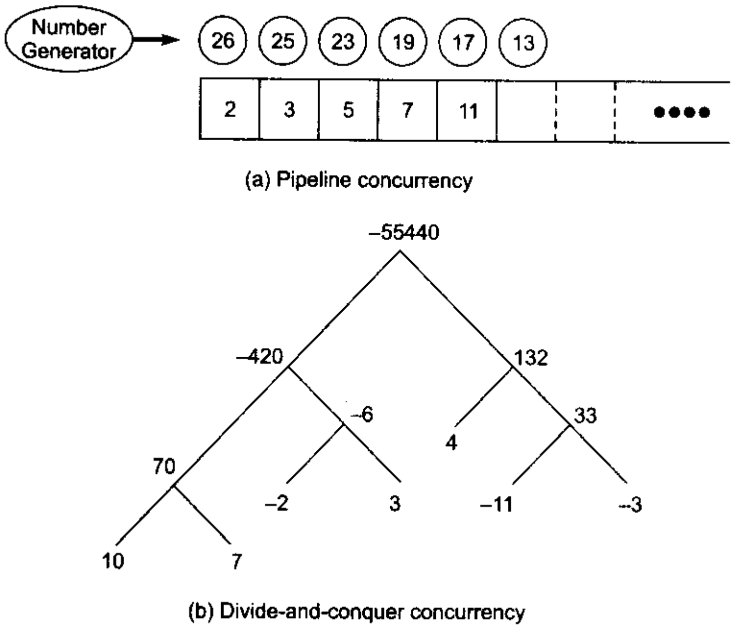
Second, *divide-and-conquer concurrency* involves the concurrent elaboration of different subprograms and the combining of their solutions to produce a solution to the overall problem. In this case, there is no interaction between the procedures solving the subproblems. These two patterns are illustrated by the following examples taken from the paper by Agha (1990).

Example 10.2 Concurrency in object-oriented programming (Gul Agha, 1990)

A prime-number generation pipeline is shown Fig. 10.3a. Integer numbers are generated and successively tested for divisibility by previously generated primes in a linear pipeline of primes. The circled numbers represent those being generated.

A number enters the pipeline from the left end and is eliminated if it is divisible by the prime number tested at a pipeline stage. All the numbers being forwarded to the right of a pipeline stage are those indivisible by all the prime numbers tested on the left of that stage.

Figure 10.3b shows the multiplication of a list of numbers (10, 7, –2, 3, 4, –11, –3) using a divide-and-conquer approach. The numbers are represented as leaves of a tree. The problem can be recursively subdivided into subproblems of multiplying two sublists, each of which is concurrently evaluated and the results multiplied at the upper node.



(a) Pipeline concurrency

(b) Divide-and-conquer concurrency

Fig. 10.3 Two concurrency types in object-oriented programming (Courtesy of G. Agha, *Commun. ACM*, September 1990)

A third pattern is called *cooperative problem solving*. A simple example is the dynamic path evaluation (computational objects) of many physical bodies (objects) under the mutual influence of gravitational fields. In this case, all objects must interact with each other; intermediate results are stored in objects and shared by passing messages between them. Interested readers may refer to the book on actors by Agha (1986).

Today companies such as IBM and Cray produce supercomputers with thousands of processors interconnected over high performance networks. At the same time, object-oriented programming and the message-passing model of inter-process communication have become established as standard paradigms of program design and development. Consider, for example, IBM's powerful Blue Gene line of supercomputers; the standard method of communication amongst node processes in these supercomputers is the Message-Passing Interface (MPI), customized for the architecture as needed. The Blue Gene line of supercomputers and MPI will both be discussed in Chapter 13.

10.1.5 Functional and Logic Models

Two language-oriented programming models for parallel processing are described below. The first model is based on using functional programming languages such as pure *Lisp*, *SISAL*, and *Strand* 88. The second model is based on logic programming languages such as *Concurrent Prolog* and *Parlog*. We reveal opportunities for parallelism in these two models and discuss their potential in AI applications.

Functional Programming Model A functional programming language emphasizes the *functionality* of a program and should not produce side effects after execution. There is no concept of storage, assignment, and branching in functional programs. In other words, the history of any computation performed prior to the evaluation of a functional expression should be irrelevant to the meaning of the expression.

The lack of side effects opens up much more opportunity for parallelism. Precedence restrictions occur only as a result of function application. The evaluation of a function produces the same value regardless of the order in which its arguments are evaluated. This implies that all arguments in a dynamically created structure of a functional program can be evaluated in parallel. All *single-assignment* and *dataflow languages* are functional in nature. This implies that functional programming models can be easily applied to data-driven multiprocessors. The functional model emphasizes fine-grain MIMD parallelism and is referentially transparent.

The majority of parallel computers designed to support the functional model were oriented toward Lisp, such as Multilisp developed at MIT. Other dataflow computers have been used to execute functional programs, including SISAL used in the Manchester dataflow machine.

Logic Programming Model Based on predicate logic, *logic programming* is suitable for knowledge processing dealing with large databases. This model adopts an implicit search strategy and supports parallelism in the logic inference process. A question is answered if the matching facts are found in the database. Two facts match if their predicates and associated arguments are the same. The process of matching and unification can be parallelized under certain conditions. Clauses in logic programming can be transformed into dataflow graphs. Parallel unification has been attempted on some dataflow computers built in Japan.

Concurrent Prolog, developed by Shapiro (1986), and *Parlog,* introduced by Clark (1987), are two parallel logic programming languages. Both languages can implement relational language features such as AND-parallel execution of conjunctive goals, IPC by shared variables, and OR-parallel reduction.

In *Parlog,* the resolution tree has one chain at AND levels, and OR levels are partially or fully generated. In *Concurrent Prolog,* the search strategy follows multiple paths or depth first. Stream parallelism is also possible in these logic programming systems.

Both functional and logic programming models have been used in artificial intelligence applications where parallel processing is very much in demand. Japan's *Fifth-Generation Computing System* (FGCS) project attempted to develop parallel logic systems for problem solving, machine inference, and intelligent human-machine interfacing.

In many ways, the FGCS project was a marriage of parallel processing hardware and AI software. The *Parallel Inference Machine* (PIM-I) in this project was designed to perform 10 *million logic inferences per second* (MLIPS). However, more recent AI applications tend to be based on other techniques, such as Bayesian inference.

10.2 PARALLEL LANGUAGES AND COMPILERS

The environment for parallel computers is much more demanding than that for sequential computers. A programming environment is a collection of software tools and system software support. Users should not have to spend a lot of time programming hardware details; they should focus instead on program parallelism using high-level abstractions. To break this hardware/software barrier, we need a parallel software environment which provides better tools for users to implement parallelism and to debug programs.

10.2.1 Language Features for Parallelism

Chang and Smith (1990) classified the language features for parallel programming into six categories according to functionality. These features are idealized for general-purpose applications. In practice, the real languages developed or accepted by the user community might have some or no features in some of the categories. Some of the features are identified with existing language/compiler development. The listed features set guidelines for developing a user-friendly programming environment.

Optimization Features These features are used for program restructuring and compilation directives in converting sequentially coded programs into parallel forms. The purpose is to match the software parallelism with the hardware parallelism in the target machine.

- Automated parallelizer—Examples are: Express C automated parallelizer and the Alliant FX Fortran compiler.
- Semiautomated parallelizer—Needs compiler directives or programmer's interaction, such as DINO.
- Interactive restructure support—Static analyzer, run-time statistics, dataflow graph, and code translator for restructuring Fortran code, such as the MIMDizer from Pacific Sierra.

Availability Features These are features that enhance the user-friendliness, make the language portable to a large class of parallel computers, and expand the applicability of software libraries.

- Scalability—The language is scalable to the number of processors available and independent of hardware topology.
- Compatibility—The language is compatible with an established sequential language.
- Portability—The language is portable to shared-memory multiprocessors, message-passing multicomputers, or both.

Synchronization/Communication Features Listed below are desirable language features for synchronization or for communication purposes:

- Single-assignment languages
- Shared variables (locks) for IPC
- Logically shared memory such as the tuple space in Linda
- Send/receive for message passing
- Rendezvous in Ada
- Remote procedure call
- Dataflow languages such as Id
- Barriers, mailbox, semaphores, monitors

Control of Parallelism Listed below are features involving control constructs for specifying parallelism in various forms:

- Coarse, medium, or fine grain

- Explicit versus implicit parallelism
- Global parallelism in the entire program
- Loop parallelism in iterations
- Task-split parallelism
- Shared task queue
- Divide-and-conquer paradigm
- Shared abstract data types
- Task dependency specification

Data Parallelism Features Data parallelism is used to specify how data are accessed and distributed in either SIMD or MIMD computers.

- Run-time automatic decomposition—Data are automatically distributed with no user intervention, as in Express.
- Mapping specification—Provides a facility for users to specify communication patterns or how data and processes are mapped onto the hardware, as in DINO.
- Virtual processor support—The compiler maps the virtual processors dynamically or statically onto the physical processors, as in PISCES 2 and DINO.
- Direct access to shared data—Shared data can be directly accessed without monitor control, as in Linda.
- SPMD (single program multiple data) support—SPMD programming, as in DINO and Hypertasking.

Process Management Features These features are needed to support the efficient creation of parallel processes, implementation of multithreading or multitasking, program partitioning and replication, and dynamic load balancing at run time.

- Dynamic process creation at run time
- Lightweight processes (threads)—Compared to UNIX (heavyweight) processes
- Replicated workers—Same program on every node with different data (SPMD mode)
- Partitioned networks—Each processor node might have more than one process and all processor nodes might run different processes
- Automatic load balancing—The workload is dynamically migrated among busy and idle nodes to achieve the same amount of work at various processor nodes

The above language features cannot be implemented without compiler support, operating system assistance, and integration with an existing environment. Software assets based on conventional languages form the basis for building an efficient parallel programming environment.

The optimization features emphasize code parallelization and vectorization at compile time. The availability features widen the application domains and make the languages machine-independent.

The synchronization features must be supported by efficient hardware and software mechanisms for their implementation. The control features often depend on tradeoffs among grain size, memory demand, and communication and scheduling overhead. Data parallelism exploits fine-grain computations on SIMD machines and medium-grain computations on MIMD computers.

The process management features are closely tied to the OS functions provided. Therefore, the languages, compilers, and OS must be developed jointly in an integrated fashion.

10.2.2 Parallel Language Constructs

Special language constructs and data array expressions are presented below for exploiting parallelism in programs. We first specify Fortran 90 array notations. Then we describe commonly used parallel constructs for program flow control.

Fortran 90 Array Notations A multidimensional data array is represented by an array name indexed by a sequence of subscript triplets, one for each dimension. Triplets for different dimensions are separated by commas. Examples are:

$$\begin{array}{r} e_1 : e_2 : e_3 \\ e_1 : e_2 \\ e_1 : * : e_3 \\ e_1 : * \\ e_1 \\ * \end{array} \tag{10.1}$$

where each e_i is an arithmetic expression that must produce a scalar integer value. The first expression e_1 is a *lower bound*, the second e_2 an *upper bound*, and the third e_3 an *increment* (*stride*). For example, $B(1:4:3, 6:8:2,3)$ represents four elements $B(l, 6, 3)$, $B(4, 6, 3)$, $B(l, 8, 3)$, and $B(4, 8, 3)$ of a three-dimensional array.

When the third expression in a triplet is missing, a unit stride is assumed. The * notation in the second expression indicates all elements in that dimension starting from e_1, or the entire dimension if e_1 is also omitted. When both e_2 and e_3 are omitted, the e_1 alone represents a single element in that dimension. For example, $A(5)$ represents the fifth element in the array $A(3 : 7 : 2)$. This notation allows us to select array sections or particular array elements.

Array assignments are permitted under the following constraints: The array expression on the right must have the same shape and the same number of elements as the array on the left. For example, the assignment $A(2:4, 5:8) = A(3:5, 1:4)$ is valid, but the assignment $A(1:4, 1:3) = A(1:2, 1:6)$ is not valid, even though each side has 12 elements. When a scalar is assigned to an array, the value of the scalar is assigned to every element of the array. For instance, the statement $B(3:4, 5) = 0$ sets $B(3, 5)$ and $B(4, 5)$ to 0.

Parallel Flow Control The conventional Fortran Do loop declares that all scalar instructions within the (**Do, Enddo**) pair are executed sequentially, and so are the successive iterations. To declare parallel activities, we use the (**Doall, Endall**) pair. All iterations in the Doall loop are totally independent of each other. This implies that they can be executed in parallel if there are sufficient processors to handle different iterations. However, the computations within each iteration are still executed serially in program order.

When the successive iterations of a loop depend on each other, we use the (**Doacross, Endacross**) pair to declare parallelism with loop-carried dependences. Synchronizations must be performed between the iterations that depend on each other. For example, dependence along the J-dimension exists in the following program. We use **Doacross** to declare parallelism along the I-dimension, but synchronization between iterations is required. The (**Forall, Endall**) and (**Pardo, Parend**) commands can be interpreted either as a Doall loop or as a Doacross loop.

```
Doacross I = 2, N
     Do J = 2, N
S_1:      A(I, J) = (A(I,(J - 1)) + A(I, J + 1))/2
     Enddo
Endacross
```

Another program construct is the (**Cobegin, Coend**) pair. All computations specified within the block could be executed in parallel. But parallel processes may be created with a slight time difference in real implementations. This is quite different from the semantics of the Doall loop or Doacross loop structures. Synchronizations among concurrent processes created within the pair are implied. Formally, the command

```
Cobegin
  P_1
  P_2
  ⋮
  P_n
Coend
```

causes processes $P_1, P_2, \ldots, P_n$ to start simultaneously and to proceed concurrently until they have all ended. The command (**Parbegin, Parend**) has equivalent meaning.

Finally, we introduce the **Fork** and **Join** commands in the following example. During the execution of a process P, we can use a **Fork** Q command to spawn a new process Q:

```
Process P          Process Q
    ⋮                  ⋮
 Fork Q                ⋮
    ⋮                 End
 Join Q
```

The **Join** Q command recombines the two processes into one process. Execution of Q is initialized when the **Fork** Q statement in P is executed. Programs P and Q are executed concurrently until either P executes the **Join** Q statement or Q terminates. Whichever one finishes first must wait for the other to complete execution, before they can be rejoined.

In a UNIX or LINUX environment, the **Fork-Join** statements provide a direct mechanism for dynamic process creation including multiple activations of the same process. The **Cobegin-Coend** statements provide a structured single-entry, single-exit control command which is not as dynamic as the **Fork-Join**. The (**Parbegin, Parend**) command is equivalent to the (**Cobegin, Coend**) command.

10.2.3 Optimizing Compilers for Parallelism

Because high-level languages are used almost exclusively to write programs today, compilers have become a necessity in modern computers. The role of a compiler is to remove the burden of program optimization and code generation from the programmer. A parallelizing compiler consists of the following three major phases: *flow analysis*, *optimizations*, and *code generation*, as depicted in Fig. 10.4.

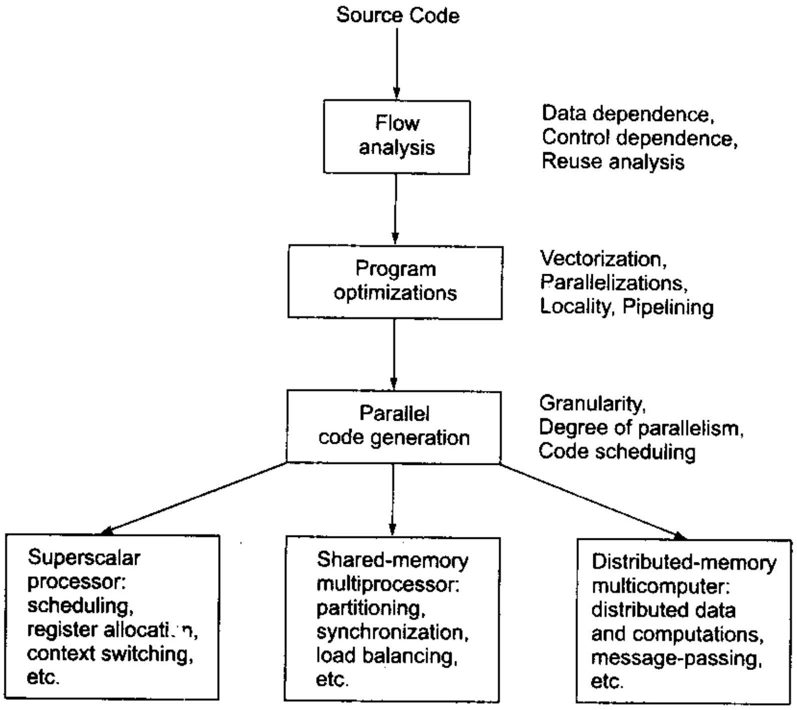

Source Code
Flow analysis
Data dependence, Control dependence, Reuse analysis
Program optimizations
Vectorization, Parallelizations, Locality, Pipelining
Parallel code generation
Granularity, Degree of parallelism, Code scheduling
Superscalar processor: scheduling, register allocati.n, context switching, etc.
Shared-memory multiprocessor: partitioning, synchronization, load balancing, etc.
Distributed-memory multicomputer: distributed data and computations, message-passing, etc.

Fig. 10.4 Compilation phases in parallel code generation

Flow Analysis This phase reveals the program flow patterns in order to determine data and control dependences in the source code. We have discussed data dependence relations among scalar-type instructions in previous chapters. Scalar dependence analysis is extended below to structured data arrays or matrices. Depending on the machine structure, the granularities of parallelism to be exploited are quite different. Thus the flow analysis is conducted at different execution levels on different parallel computers.

Generally speaking, instruction-level parallelism is exploited in superscalar or VLSI processors; loop level in SIMD, vector, or systolic computers; and task level in multiprocessors, multicomputers, or a network of workstations. Of course, exceptions do exist. For example, fine-grain parallelism can in theory be pushed down to multicomputers with a globally shared address space. The flow analysis must also reveal code/data reuse and memory-access patterns.

Program Optimizations This refers to the transformation of user programs in order to explore the hardware capabilities as much as possible. Transformation can be conducted at the loop level, locality level, or prefetching level with the ultimate goal of reaching global optimization. The optimization often transforms a code into an equivalent but "better" form in the same representation language. These transformations should be machine-independent.

In reality, most transformations are constrained by the machine architecture. This is the main reason why many such compilers are machine-dependent. At the least, we want to design a compiler which can run on most machines with only minor modifications. One can also conduct certain transformations preceding the global optimization. This may require a source-to-source optimization (sometimes carried out by a *precompiler*), which transforms the program from one high-level language to another before using a dedicated compiler for the second language on a target machine.

The ultimate goal of program optimization is to maximize the speed of code execution. This involves the minimization of code length and of memory accesses and the exploitation of parallelism in programs. The optimization techniques include vectorization using pipelined hardware and parallelization using multiple processors simultaneously. The compiler should be designed to reduce the running time with minimum resource binding. Other optimizations demand the expansion of routines or procedure integration with inlining. Both local and global optimizations are needed in most programs. Sometimes the optimization should be conducted at the algorithmic level and must involve the programmer.

Machine-dependent transformations are meant to achieve more efficient allocation of machine resources, such as processors, memory, registers, and functional units. Replacement of complex operations by cheaper ones is often practiced. Other optimizations include elimination of unnecessary branches or common expressions. Instruction scheduling can be used to eliminate pipeline or memory delays in executing consecutive instructions.

Parallel Code Generation Code generation usually involves transformation from one representation to another, called an *intermediate form*. A code model must be chosen as an intermediate form. Parallel code is even more demanding because parallel constructs must be included. Code generation is closely tied to the instruction scheduling policies used. Basic blocks linked by control-flow commands are often optimized to encourage a high degree of parallelism. Special data structures are needed to represent instruction blocks.

Parallel code generation is very different for different computer classes. For example, a superscalar processor may be software-scheduled or hardware-scheduled. How to optimize the register allocation on a RISC or superscalar processor, how to reduce the synchronization overhead when codes are partitioned for multiprocessor execution, and how to implement message-passing commands when codes/data are distributed (or replicated) on a multicomputer are added difficulties in parallel code generation. Compiler directives can be used to help generate parallel code when automated code generation cannot be implemented easily.

Two well-known exploratory optimizing compilers were developed over mid-1980: one was Parafrase at the University of Illinois, and the other was the PFC (Parallel Fortran Converter) at Rice University. These systems are briefly introduced below.

Parafrase and Parafrase 2 This system, developed by David Kuck and coworkers at Illinois, is a source-to-source program restructurer (or compiler preprocessor) which transforms sequential Fortran 77 programs into forms suitable for vectorization or parallelization. Parafrase contains more than 100 program transformations which are encoded as *passes*. A *pass list* is used to identify the particular sequence of transformations needed for restructuring a given sequential program. The output of Parafrase is the converted concurrent program.

Different programs use different pass list and thus go through different sequences of transformations. The pass lists can be optimized for specific machine architectures and specific program constructs. Parafrase 2 was developed for handling programs written in C and Pascal, in addition to converting Fortran codes. Information on Parafrase can be found in [Kuck84] and on Parafrase 2 in [Polychronopoulos89].

Parafrase is retargetable to produce code for different classes of parallel/vector computers. The program transformed by Parafrase still needs a conventional optimizing compiler to produce the object code for the target machine. The Parafrase technology was later transferred to implement the KAP vectorizer by Kuck and Associates, Inc.

The PFC and ParaScope Ken Kennedy and his associates at Rice University developed PFC as an automatic source-to-source vectorizer. It translated Fortran 77 code into Fortran 90 code. A categorized dependence testing scheme was developed in PFC for revealing opportunities for loop vectorization. The PFC package was also extended to PFC+ for parallel code generation on shared-memory multiprocessors. PFC and PFC+ also supported the ParaScope programming environment.

PFC (Allen and Kennedy, 1984) performed syntax analysis, including the following four steps:

(1) Interprocedural flow analysis using call graphs.

(2) Standard transformations such as Do-loop normalization, subscript categorization, deletion of dead codes, etc.

(3) Dependence analysis which applied the separability, GCD, and Banerjee tests jointly.

(4) Vector code generation. PFC+ further implemented a parallel code generation algorithm (Callahan et al, 1988).

Commercial Compilers Optimizing compilers have also been developed in a number of commercial parallel/vector computers, including the Alliant FX/F Fortran compiler, the Convex parallelizing/vectorizing compiler, the Cray CFT compiler, the IBM vectorizing Fortran compiler, the VAST vectorizer by Pacific Sierria, Inc., and Intel iPSC-VX compiler. IBM also developed a PTRAN (Parallel Fortran) system based on control dependence with interprocedural analysis.

10.3 DEPENDENCE ANALYSIS OF DATA ARRAYS

Dependence testing of successive iterations in multidimensional data arrays is described in this section. This provides a theoretical foundation for the development of vectorizing or parallelizing compilers.

10.3.1 Iteration Space and Dependence Analysis

Flow dependence, antidependence, and output dependence were defined for scalar data in Section 2.1.2. They can be summarized by the existence of dynamic references of R_1 and R_2, if and only if either R_1 or R_2 is a write operation, R_1 executes before R_2, or R_1 and R_2 both write the same variable. When the referenced object is a data array indexed by a multidimensional subscript, the dependence becomes very difficult to determine at compile time, since subscript values are not in general available.

Precise and efficient dependence tests are essential to the effectiveness of a parallelizing compiler. The process of computing all the data dependences in a program is called *dependence analysis*. The testing scheme presented below is based on the work of Goff, Kennedy, and Tseng (1991). These dependence tests were implemented at Rice University in PFC with the parallel ParaScope programming environment.

Dependence Testing Calculating data dependence for arrays is complicated by the fact that two array references may not access the same memory location. Dependence testing is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest. For the purpose of this explication, we ignore any control flow except for the loops themselves. Suppose we wish to test whether or not there exists a dependence from statement S_1 to S_2 in the following model loop nest of n levels, represented by n integer indices $i_1, i_2, \ldots, i_n$.

```
     Do $i_1 = L_1, U_1$
         Do $i_2 = L_2, U_2$
         ...
             Do $i_n = L_n, U_n$
$S_1$:           $A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \cdots$
$S_2$:           $\cdots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))$
             Enddo
             ...
         Enddo
     Enddo
```

Iteration Space The n-dimensional discrete Cartesian space for n-deep loops is called an *iteration space*. The *iteration* is represented as coordinates in the iteration space. The following example clarifies the concept of *lexicographic order* for the successive iterations in a loop nest.

Example 10.3 Lexicographic order for sequential execution of successive iterations in a loop structure (Monica Lam, 1992)

Consider a two-dimensional iteration space (Fig. 10.5) representing the following two-level loop nest in unit-increment steps:

```
Do $i = 0,5$
    Do $j = i, 7$
        $f(i,j) = \cdots$
    Enddo
Enddo
```

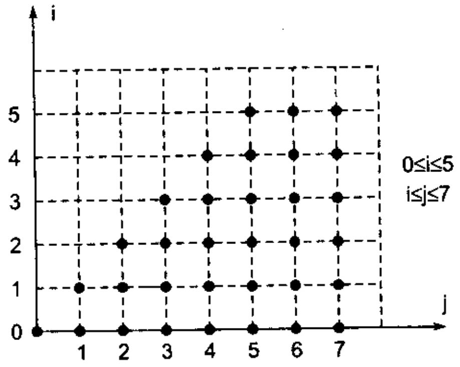


Fig. 10.5 A two-dimensional iteration space for the loop nest in Example 10.3

The following sequential order of iteration is a lexicographic order:

$$
\begin{array}{r}
(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7) \\
(1,1), (1, 2), (1, 3), (1, 4), (1,5), (1,6), (1, 7) \\
(2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7) \\
(3, 3), (3, 4), (3, 5), (3, 6), (3, 7) \\
(4, 4), (4, 5), (4, 6), (4, 7) \\
(5, 5), (5, 6), (5, 7)
\end{array}
$$

The lexicographic order is important to performing matrix transformation, which can be applied for loop optimization. We will apply lexicographic orders for loop parallelization in Section 10.5.

Dependence Equations Let α and β be vectors of n integer indices within the ranges of the upper and lower bounds of the n loops. There is a *dependence* from S_1 to S_2 if and only if there exist α and β such that α is lexicographically less than or equal to β and the following system of *dependence equations* is satisfied:

$$f_i(\alpha) = g_i(\beta) \qquad \forall i,\ 1 \le i \le m \tag{10.2}$$

Otherwise the two references are *independent*.

The dependence equations in Eq. 10.2 are linear expressions of the loop index variables. Dependence testing is thus equivalent to the problem of linear Diophantine equations, which is an NP-complete problem. *Exact tests* are dependence tests that will detect dependences if and only if they exist. In practice, exact tests are not performed due to the excessive overhead involved. Only approximate solutions (which are efficient to implement) are sought.

Parallelizing compilers have traditionally relied on two dependence tests to detect data dependences between pairs of array references: *Banerjee's inequalities* (Banerjee, 1988) and *GCD* tests (Wolfe, 1989). However, these tests are usually more general than necessary.

In Section 10.3.2, we present a practical testing algorithm developed by Rice University researchers led by Ken Kennedy. The test algorithm is based on partitioning the subscripts in a pair of array references. A suite of simple tests is developed to reduce the cost of performing dependence analysis, making it more practical for most compilers.

Distance and Direction Vectors Suppose there exists a data dependence for $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$. Then the *distance vector* $\mathbf{D} = (D_1, \ldots, D_n)$ is defined as $\beta - \alpha$. The *direction vector* $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ of the dependence is defined by

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases} \tag{10.3}$$

The elements are always displayed in order from left to right and from the outermost to the innermost loop in the nest.

For example, consider the following loop nest:

```
Do $i = L_1, U_1$
    Do $j = L_2, U_2$
```

```
        Do $k = L_3, U_3$
            $A(i + 1, j, k - 1) = A(i, j, k) + C$
        Enddo
    Enddo
Enddo
```

The distance and direction vectors for the dependence between iterations along three dimensions of the array A are $(1, 0, -1)$ and $(<, =, >)$, respectively. Since several different values of α and β may satisfy the dependence equations, a set of distance and direction vectors may be needed to completely describe the dependence.

Direction vectors are useful for calculating the level of *loop-carried* dependences. A dependence is *carried* by the outermost loop for which the direction in the direction vector is not "=". For instance, the direction vector $(<, =, >)$ for the dependence above shows the dependence is carried on the i-loop.

Carried dependences are important because they determine which loops cannot be executed in parallel without synchronization. Direction vectors are also useful in determining whether loop interchange is legal and profitable. Distance vectors are more precise versions of direction vectors that specify the actual distance in loop iterations between two accesses to the same memory location. They may be used to guide optimizations to exploit parallelism or the memory hierarchy.

10.3.2 Subscript Separability and Partitioning

Dependence testing thus has two goals. It tries to disprove the dependence between pairs of subscripted references to the same array variable. If dependences exist, it tries to characterize them in some manner, usually as a minimum complete set of distance and direction vectors. Dependence testing must also be *conservative* and *assume the existence of any dependence it cannot disprove*. Otherwise the validity of any optimizations based on dependence information is not guaranteed.

Subscript Categories The term *subscript* refers to one of the subscripted positions in a pair of array references, i.e. the pair of subscripts in some dimension of the two array references. When testing for dependence, we classify subscript positions by the total number of distinct loop indices they contain.

A subscript is said to be *zero index variable* (ZIV) if the subscript position contains no index in either reference. A subscript is said to be *single index variable* (SIV) if only one index occurs in that position. Any subscript with more than one index is said to be *multiple index variable* (MIV).

Example 10.4 Subscript types in a loop computation

Consider the following loop nest of three levels, identified by indices i, j, and k.

```
Do $i_1 = L_1, U_1$
    Do $j = L_2, U_2$
        Do $k = L_n, U_n$
```

```
            $A(5, i + 1, j) = A(N, i, k) + C$
        Enddo
    Enddo
Enddo
```

When testing for a flow dependence between the two references to A in the code, the first subscript is ZIV because 5 and N are both constants, the second is SIV because only index i appears in this dimension, and the third is MIV because both indices j and k appear in the third dimension. For simplicity, we have ignored the output dependence in this example.

Subscript Separability When testing multidimensional arrays, we say that a subscript position is *separable* if its indices do not occur in the other subscripts. If two different subscripts contain the same index, we say they are *coupled*. Separability is important because multidimensional array references can cause imprecision in dependence testing.

If all the subscripts are separable, we may compute the direction vector for each subscript independently and merge the direction vectors on a positional basis with full precision. The following examples clarify these concepts.

Example 10.5 From separability to direction vector and distance vector

Consider the following loop nest:

```
Do $i_1 = L_1, U_1$
    Do $j = L_2, U_2$
        Do $k = L_3, L_3$
            $A(i, j, j) = A(i, j, k) + C$
        Enddo
    Enddo
Enddo
```

The first subscript is separable because index i does not appear in the other dimensions, but the second and third are coupled because they both contain the index j. ZIV subscripts are separable because they contain no indices.

Consider another loop nest:

```
Do $i_1 = L_1, U_1$
    Do $j = L_2, U_2$
        Do $k = L_n, U_n$
            $A(i + 1, j, k - 1) = A(i, j, k) + C$
        Enddo
    Enddo
Enddo
```

The leftmost direction in the direction vector is determined by testing the first subscript, the middle direction by testing the second subscript, and the rightmost direction by testing the third subscript.

The resulting direction vector $(<, =, >)$ is precise. The same approach applied to distances allows us to calculate the exact distance vector $(1, 0, -1)$.

Subscript Partitioning We need to classify all the subscripts in a pair of array references as separable or as part of some minimal coupled group. A coupled group is *minimal* if it cannot be partitioned into two nonempty subgroups with distinct sets of indices. Once a partition is achieved, each separable subscript and each coupled group has completely disjoint sets of indices.

Each partition may then be tested in isolation and the resulting distance or direction vectors merged without any loss of precision. Since each variable and coupled subscript group contains a unique subset of indices, a merge may be thought of as a Cartesian product.

In the following loop nest, the first subscript yields the direction vector $(<)$ for the i-loop. The second subscript yields the direction vector $(=)$ for the j-loop. The resulting Cartesian product is the single vector $(<, =)$.

```
Do $i = L_1, U_1$
    Do $j = L_2, U_2$
        $A(i + 1, j) = A(i, j) + C$
    Enddo
Enddo
```

Consider another loop nest where the first subscript yields the direction vector $(<)$ for the i-loop.

```
Do $i = L_1, U_1$
    Do $j = L_2, U_2$
        $A(i+1, 5) = A(i, N) + C$
    Enddo
Enddo
```

Since j does not appear in any subscript, we must assume the full set of direction vectors for the j-loop: $\{(<), (=), (>)\}$. Thus a merge yields the following set of direction vectors for both dimensions:

$$\{(<, <), (<, =), (<, >)\}$$

10.3.3 Categorized Dependence Tests

The goal of dependence testing is to construct the complete set of distance and direction vectors representing potential dependences between an arbitrary pair of subscripted references to the same array variable. Since distance vectors may be treated as precise direction vectors, we will simply refer to direction vectors.

The Testing Algorithm The following procedure is for dependence testing based on a partitioning approach, which can isolate unrelated indices and localize the computation involved and thus is easier to implement.

(1) Partition the subscripts into separable and minimal coupled groups using the following algorithm:

Subscript Partitioning Algorithm (Goff, Kennedy, and Tseng, 1991)

```
Input:   A pair of $m$-dimensional array references
         containing subscripts $S_1 \ldots S_m$ enclosed in $n$ loops
         with indices $I_1 \ldots I_n$.
Output:  A set of partitions $P_1 \ldots P_{n'}$, $n' \le n$, each
         containing a separable or minimal coupled group.
For each $i$, $1 \le i \le n$ Do
    $P_i \leftarrow \{S_i\}$
Endfor
For each index $I_i$, $1 \le i \le n$ Do
    $k \leftarrow \{none\}$
    For each remaining partition $P_j$ Do
        if $\exists S_1 \in P_j$ such that $S_1$ contains $I_i$, then
            if $k = \{none\}$ then
                $k \leftarrow j$
            else
                $P_k \leftarrow P_k \cup P_j$
                Discard $P_j$
            Endif
        Endif
    Endfor
Endfor
```

(2) Label each subscript as ZIV, SIY, or MIV.

(3) For each separable subscript, apply the appropriate single subscript test (ZIV, SIY, MIV) based on the complexity of the subscript. This will produce independence or direction vectors for the indices occurring in that subscript.

(4) For each coupled group, apply a multiple subscript test to produce a set of direction vectors for the indices occurring within that group.

(5) If any test yields independence, no dependences exist.

(6) Otherwise merge all the direction vectors computed in the previous steps into a single set of direction vectors for the two references.

Test Categories Dependence test results for ZIV subscripts are treated specially. If a ZIV subscript proves independence, the dependence test algorithm halts immediately. If independence is not proved, the ZIV test does not produce direction vectors, and so no merge is necessary. For the implementation of the above algorithm, we have specified how to perform the single subscript tests (ZIV, SIY, MIV) separately. We consider below the trivial case of ZIV first, then SIY, and finally MIV which is more involved.

We first consider dependence tests for single separable subscripts. All tests presented assume that the subscript being tested contains expressions that are linear in the loop index variables. A subscript expression is linear if it has the form $a_1 i_1 + a_2 i_2 + \ldots + a_n i_n + e$, where i_k is the index for the loop at nesting level k; all a_k, $1 \le k \le n$, are integer constants; and e is an expression possibly containing loop-invariant symbolic expressions.

The ZIV Test The ZIV test is a dependence test performed on two loop-invariant expressions. If the system determines that the two expressions cannot be equal, it has proved independence. Otherwise the subscript does not contribute any direction vectors and may be ignored. The ZIV test can be easily extended for symbolic expressions. Simply form the expression representing the difference between the two subscript expressions. If the difference simplifies to a nonzero constant, we have proved independence.

The SIV Test An SIV subscript for index i is said to be *strong* if it has the form $(ai + c_1, ai' + c_2)$, i.e. if it is linear and the coefficients of the two occurrences of the index i are constant and equal. For strong SIV subscripts, define the *dependence distance* as

$$d = i' - i = \frac{c_1 - c_2}{a} \tag{10.4}$$

A dependence exists if and only if d is an integer and $|d| \le U - L$, where U and L are the loop upper and lower bounds. For dependences that do exist, the *dependence direction* is given by

$$\text{Direction} = \begin{cases} < & \text{if } d > 0 \\ = & \text{if } d = 0 \\ > & \text{if } d < 0 \end{cases} \tag{10.5}$$

The strong SIV test is thus an exact test that can be implemented very efficiently in a few operations. A bounded iteration space is shown in Fig. 10.6a. The case of a strong SIV test is shown in Fig. 10.6b.

Another advantage of the strong SIV test is that it can be easily extended to handle loop-invariant symbolic expressions. The trick is to first evaluate the dependence distance d symbolically. If the result is a constant, then the test may be performed as above. Otherwise calculate the difference between the loop bounds and compare the result with d symbolically.

A *weak* SIV subscript has the form $(a_1\, i - c_1, a_2\, i' + c_2)$, where the coefficients of the two occurrences of index i have different constant values. As stated previously, weak SIV subscripts may be solved using the single-index exact test. However, we also find it helpful to view the problem geometrically, where the dependence equation

$$a_1\, i + c_1 = a_2\, i' + c_2$$

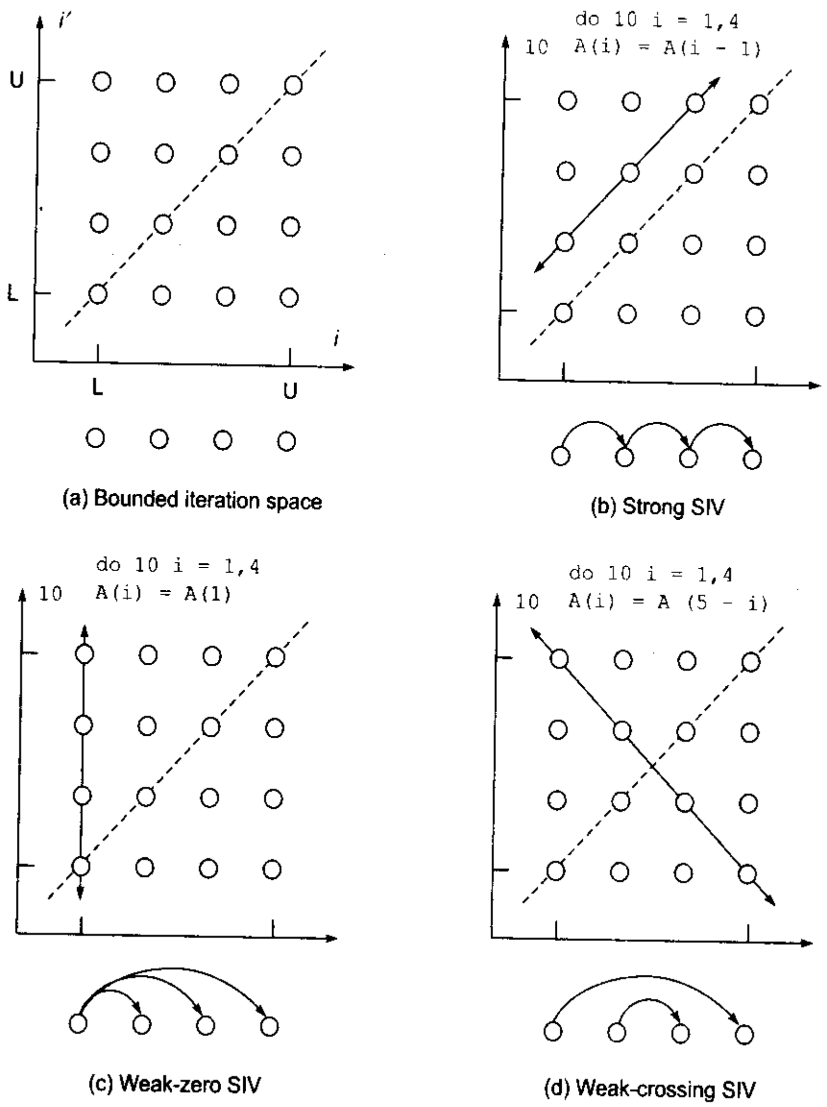
describes a line in the two-dimensional plane with i and i' as axes.

The weak SIV test can then be formulated as determining whether the line derived from the dependence equation intersects with any integer points in the space bounded by the loop upper and lower bounds, as shown in Fig. 10.5.

Two special cases should be studied separately.

Weak-Zero SIV Test The case in which $a_1 = 0$ or $a_2 = 0$ is called a *weak-zero* SIV subscript, as illustrated in Fig. 10.6c. If $a_2 = 0$, the dependence equation reduces to

$$i = \frac{c_2 - c_1}{a_1}$$



(a) Bounded iteration space

(b) Strong SIV

(c) Weak-zero SIV

(d) Weak-crossing SIV

Fig. 10.6 Geometric view of SIV tests in four cases (Courtesy of Goff et al, 1991; reprinted from ACM SIGPLAN *Conf. Programming Language Design and Implementation*, Toronto, Canada, 1991)

It is only necessary to check that the resulting value for i is an integer and within the loop bounds. A similar check applies when $a_1 = 0$.

The weak-zero SIV test finds dependences caused by a particular iteration i. In scientific codes, i is usually the first or last iteration of the loop, eliminating one possible direction vector for the dependence.

Consider the following loop for a strong SIV test:

```
Do i = 1, N
    A(i + 2N) = A(i + N)
Enddo
```

The strong SIV test can evaluate the dependence distance d as $2N - N$, which simplifies to N. This is compared with the loop bounds symbolically, proving independence since $N > N - 1$.

Consider the following simplified loop in the program *tomcatv* from the SPEC benchmark suite (Uniejewski, 1989):

```
Do i = 1, N
    Y(i, N) = Y(i, N) + Y(N, N)
Enddo
```

The weak-zero SIV test can determine that the use of Y(1, N) causes a loop-carried true dependence from the first iteration to all the other iterations. Similarly, with aid from symbolic analysis, the weak-zero SIV test can discover that the use of Y(N, N) causes a loop-carried antidependence from all iterations to the last iteration. By identifying the first and last iterations as the only cause of dependences, the weak-zero SIV test advises the user or compiler to peel the first and last iterations of the loop, resulting in the following parallel loop:

```
Y(1, N) = Y(1, N) + Y(N, N)
Do i = 2, N - 1
    Y(i, N) = Y(i, N) + Y(N, N)
Enddo
Y(N, N) = Y(N, N) + Y(N, N)
```

Weak-Crossing SIV Test All subscripts where $a_2 = -a_1$ are *weak-crossing SIV*. These subscripts typically occur as part of Cholesky decomposition, also illustrated in Fig. 10.6d. In these cases we set $i = i'$ and derive the dependence equation

$$i = \frac{c_2 - c_1}{2a_1}$$

This corresponds to the intersection of the dependence equation with the line $i = i'$. To determine whether dependences exist, we simply need to check that the resulting value i is within the loop bounds and is either an integer or has a noninteger part equal to 1/2.

Weak-crossing SIV subscripts cause *crossing* dependences, loop-carried dependences whose end points all cross iteration i. These dependences may be eliminated using a *loop-splitting* transformation (Kennedy et al, 1991) as described below.

Consider the following loop from the Callahan-Dongarra-Levine vector test (Callahan et al, 1988):

```
Do i = 1, N
    A(i) = A(N - i + 1) + C
Enddo
```

The weak-crossing SIV test determines that dependences exist between the definition and use of A and that they all cross iteration $(N + 1)/2$. Splitting the loop at that iteration results in two parallel loops:

```
Do i = 1, (N + 1)/2
    A(i) = A(N – i + 1) + C
Enddo
Do i = (N + 1)/2 + 1, N
    A(i) = A(N – i + 1) + C
Enddo
```

The MIV Tests SIV tests can be extended to handle complex iteration spaces where loop bounds may be functions of other loop indices, e.g. triangular or trapezoidal loops. We need to compute the minimum and maximum loop bounds for each loop index.

Starting at the outermost loop nest and working inward, we replace each index in a loop upper bound with its maximum value (or minimal if it is a negative term). We do the opposite in the lower bound, replacing each index with its minimal value (or maximal if it is a negative term).

We evaluate the resulting expressions to calculate the minimal and maximal values for the loop index and then repeat for the next inner loop. This algorithm returns the maximal range for each index, all that is needed for SIV tests.

The Banerjee-GCD test may be employed to construct all legal direction vectors for linear subscripts containing multiple indices. In most cases the test can also determine the minimal dependence distance for the carrier loop.

A special case of MIV subscripts, called RDIV (restricted double-index variable) subscripts, have the form $(a_1 i + c_1, a_2 j + c_2)$. They are similar to SIV subscripts except that *i* and *j* are distinct indices. By observing different loop bounds for *i* and *j*, SIV tests may also be extended to test RDIV subscripts exactly.

A large body of work was performed in the field of dependence testing at Rice University, the University of Illinois, and Oregon Graduate Institute. What was described above is only one of the many dependence testing algorithms proposed. Experimental results are reported from these research centers. Readers are advised to read published material on Banerjee's test and the GCD test, which provide other inexact and conservative solutions to the problem.

The development of a parallelizing compiler is limited by the difficulty of having to deal with many nonperfectly nested loops. The lack of dataflow information is often the ultimate limit on automatic compilation of parallel code.

10.4 CODE OPTIMIZATION AND SCHEDULING

In this section, we describe the roles of compilers in code optimization and code generation for parallel computers. In no case can one expect production of a true optimal code which matches the hardware behavior perfectly. Compilation is a software technique which transforms the source program to generate better object code, which can reduce the running time and memory requirement. On a parallel computer, program optimization often demands an effort from both the programmer and the compiler.

10.4.1 Scalar Optimization with Basic Blocks

Instruction scheduling is often supported by both compiler techniques and dynamic scheduling hardware. In order to exploit *instruction-level parallelism* (ILP), we need to optimize the code generation and scheduling process under both machine and program constraints. Machine constraints are caused by mutually exclusive use of functional units, registers, data paths, and memory. Program constraints are caused by data and control dependences. Some processors, like those with VLIW architecture, explicitly specify ILP in their instructions. Others may use hardware interlock, out-of-order execution, or speculative execution. Even machines with dynamic scheduling hardware can benefit from compiler scheduling techniques.

There are two alternative approaches to supporting instruction scheduling. One is to provide an additional set of nontrapping instructions so that the compiler can perform aggressive *static instruction scheduling*. This approach requires an extension of the instruction set of existing processors. The second approach is to support out-of-order execution in the micro-architecture so that the hardware can perform aggressive *dynamic instruction scheduling*. This approach usually does not require the instruction set to be modified but requires complex hardware support.

In general, instruction scheduling methods ensure that control dependences, data dependences, and resource limitations are properly handled during concurrent execution. The goal is to produce a schedule that minimizes the execution time or the memory demand, in addition to enforcing correctness of execution. Static scheduling at compile time requires intelligent compilation support, whereas dynamic scheduling at run time requires sophisticated hardware support. In practice, dynamic scheduling can be assisted by static scheduling in improving performance.

Precedence Constraints Speculative execution requires the use of program profiling to estimate effectiveness. Speculative exceptions must not terminate execution. In other words, precise exception handling is desired to alleviate the control dependence problem. The data dependence problem involves instruction ordering and register allocation issues.

If a flow dependence is detected, the *write* must proceed ahead of the *read* operation involved. Similarly, output dependence produces different results if two *writes* to the same location are executed in a different order. Antidependence enforces a *read* to be ahead of the *write* operation involved. We need to analyze the memory variables. Scalar data dependence is much easier to detect. Dependence among arrays of data elements is much more involved, as shown in Section 10.3. Other difficulties lie in interprocedural analysis, pointer analysis, and register allocations interacting with code scheduling.

Basic Block Scheduling A *basic block* (or just a *block*) is a sequence of statements satisfying two properties: (1) No statement but the first can be reached from outside the block; i.e. there are no branches into the middle of the block. (2) All statements are executed consecutively if the first one is. Therefore, no branches out or halts are allowed until the end of the block. All blocks are required to be *maximal* in the sense that they cannot be extended up or down without violating these properties.

For local optimization only, an *extended basic block* is defined as a sequence of statements in which the first statement is the only entry point. Thus an extended block may have branches out in the middle of the code but no branches into it. The basic steps for constructing basic blocks are summarized below:

(1) Find the *leaders*, which are the first statements in a block. Leaders are identified as being one or more of the following:
   (a) The first statement of the code.
   (b) The target of a conditional or unconditional branch.
   (c) A statement following a conditional branch.
(2) For a leader, a basic block consists of the leader and all statements following up to but excluding the next leader. Note that the beginning of inaccessible code (dead code) is not considered a leader. In fact, dead code should be eliminated.

Example 10.6 Basic block construction in a bubble sort program (S. Graham, J. L. Hennessy, and J. D. Ullman, 1992)

A bubble sort program sorts an array $A[j]$ with statically allocated storage. Each element of A requires 4 bytes of byte-addressable memory. The elements of $A[j]$ are numbered $j = 1, 2, \ldots, n$, where n is a variable. To be specific, $A[j]$ is stored in location $addr(A) + 4 * (j - 1)$, where $addr(A)$ produces the starting address of the array A. The following source code is for bubble sort:

```
For i : = n – 1 downto 1 do
    For j := 1 to i do
        If A[j] > A[j + 1] then
            Begin
                temp := A[j]
                A[j] := A[j + 1]
                A[j + 1] := temp
            End
    End of j-loop
End of i-loop
```

If a three-address machine is assumed, the above code is translated into the following assembly language code. Variable names on the right of := stand for values, and on the left for addresses.

```
      i := n – 1
s5:   if i < 1 goto sl
      j := 1
s4:   if j > i goto s2
      tl := j – 1
      t2 := 4 * tl
      t3 := A[t2]              /A[j]/
      t4 := j + 1
      t5 := t4 – 1
      t6 := 4 * t5
      t7 := A[t6]              /A[j+1]/
      if t3 < = t7 goto s3     /if A[j] > A[j+1] then begin .../
      t8 := j – 1
      t9 := 4 * t8
      temp := A[t9]            / temp := A[j]/
      t10 := j + 1
      t11 := tl0 – 1
```

```
     t12 := 4 * t11
     t13 := A[t12]           /A[j+1]/
     t14 := j – 1
     t15 := 4 * t14
     A[t15] := t13           /A[j] := A[j+1]/
     t16 := j + 1
     t17 := t16 – 1
     t18 := 4 * t17
     A[t18] := temp          /A[j+1] := temp/
s3:  j := j + 1
     goto s4
s2:  i := i – 1
     goto s5
s1:  halt
```

The above 31 statements are divided into 8 basic blocks as shown in Fig. 10.7. A program *flow graph* is drawn to show the precedence relationship among the basic blocks. Each node in the flow graph corresponding to one basic block may contain different numbers of statements. The entry node is B1, and the exit node is B2.

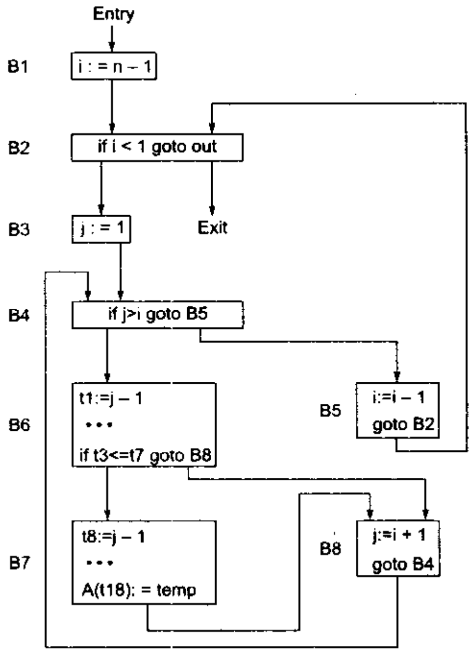


Fig. 10.7 A flow graph showing the precedence relationship among basic blocks in the bubble sort program (Courtesy of S. Graham, J. L. Hennessy, and J. D. Ullman. *Course on Code Optimization and Code Generation*, Western Institute of Computer Science, Stanford University, 1992)

While a program is being compiled, basic block records should keep pointers to predecessors and successors. Storage reclamation techniques or linked list structures can be used to represent blocks. Sources of program optimization include algebraic optimization to eliminate redundant operations, and other optimizations conducted within the basic blocks locally or on a global basis.

10.4.2 Local and Global Optimizations

We first describe local code optimization within basic blocks. Then we study global optimizations among basic blocks. Both intraprocedural and interprocedural optimizations are discussed. Finally, we identify some machine-dependent optimizations. Readers will realize the limitations and potentials of these code optimization methods.

Local Optimizations

These are code optimizations performed only within basic blocks. The information needed for optimization is gathered entirely from a single basic block, not from an extended basic block. No control-flow information between blocks is considered. Listed below are some local optimizations often performed:

(1) Local Common Subexpression Elimination If a subexpression is to be evaluated more than once within a single block, it can be replaced by a single evaluation. For Example 10.6, in block B7, t9 and t15 each compute 4 * (j – 1), and t12 and t18 each compute 4 * j. Replacing t15 by t9, and t18 by t12, we obtain the following revised code for B7, which is shorter to execute.

```
t8 := j – 1
t9 := 4 * t8
temp := A[t9]
t12 := 4 * j
t13 := A[t12]
A[t9] := t13
A[t12] := temp
```

(2) Local Constant Folding or Propagation Sometimes some constants used in instructions can be computed at compile time. This often takes place in the initialization blocks. The compile-time generated constants are then folded to eliminate unnecessary calculations at run time. In other cases, a local copy may be propagated to eliminate unnecessary calculations.

(3) Algebraic Optimization to Simplify Expressions For example, one can replace the *identity statement* $A := B + 0$ or $A := B * 1$ by $A := B$ and later even replace references to this A by references to B. Or one can use the *commutative law* to combine expressions $C := A + B$ and $D := B + A$. The *associative* and *distributive laws* can also be applied on equal-priority operators, such as replacing $(a - b) + c$ by $a - (b - c)$ if $(b - c)$ has already been evaluated earlier.

(4) Instruction Reordering Code reordering is often practiced to maximize the pipeline utilization or to enable overlapped memory accesses. Some orders yield better code than others. Reordered instructions lead to better scheduling, preventing pipeline or memory delays. In the following example, instruction I3 may be delayed in memory accesses:

```
I1:     Load    R1, A
I2:     Load    R2, B
I3:     Add     R2, R1, R2 – delayed
I4:     Load    R3, C
```

With reordering, the instruction I3 may experience no delay:

```
I1:     Load    R1, A
I2:     Load    R2, B
I4:     Load    R3, C
I3:     Add     R2, R1, R2 – not delayed
```

(5) Elimination of Dead Code or Unary Operators Code segments or even basic blocks which are not accessible or will never be referenced can be eliminated to save compile time, run time, and space requirements.

Unary operators, such as arithmetic negation and logical complement, can often be eliminated by applying algebraic laws, such as $x + (-y) = x - y$, $-(x - y) = y - x$, $(-x) * (-y) = x * y$, Not(Not A) = A, etc. Boolean expression evaluation can often be optimized after some form of minimization.

Global Optimizations These are code optimizations performed across basic block boundaries. Control-flow information among basic blocks is needed. John Hennessy (1992) has classified intraprocedural global optimizations into three types:

(1) Global Versions of Local Optimizations These include global common subexpression elimination, global constant propagation, dead code elimination, etc. The following example further optimizes the code in Example 10.6 if some global optimizations are performed.

Example 10.7 Global optimizations in the bubble sort program (S. Graham, J. L. Hennessy, and J. D. Ullman, 1992)

In Example 10.6, block B7 needs to compute *A*[t9] = A[4 * (j –1)], which was computed in block B6. To reach B7, B6 must be executed first and the value of j never changes between the two nodes. Thus the first three statements of B7 can be replaced by temp := t3. Similarly, t9 computes the same value as t2, t12 computes the same value as t6, and t13 computes the same value as t7. The entire block B7 may be replaced by

```
temp := t3
A[t2] := t7
A[t6] := temp
```

and, substituting for temp by

```
A[t2] := t7
A[t6] := t3
```

The revised program, after both local and global optimizations, is obtained as follows:

```
B1:  i := n – 1
B2:  If i < 1 goto out
B3:  j := 1
B4:  If j > i goto B5
B6:  t1:= j – 1
     t2 := 4 * t1
     t3 := A[t2]                /A[j]/
     t6 := 4 * j
     t7 := A[t6]                /A[j+1]/
     If t3 <= t7 goto B8
B7:  A[t2] := t7
     A[t6] := t3
B8:  j := j + 1
     goto B4
B5:  i := i – 1
     goto B2
out:
```

(2) Loop Optimizations These include various loop transformations to be described in subsequent sections for the purpose of vectorization, parallelization, or both. Somtimes *code motion* and *induction variable elimination* can simplify loop structures. For example, one can replace the calculation of an induction variable involving a multiplication by an addition to its former value. The addition takes less time to perform and thus results in a shorter execution time.

In other cases, loop-invariant variables or codes can be moved out of the loop to simplify the loop nest. One can also lower the loop control overhead using loop unrolling to reduce iteration or loop fusion to merge loops. Loops can be exchanged to facilitate pipelining of long vectors. Many loop optimization examples will be given in subsequent sections.

(3) Control-flow Optimization These are other global optimizations dealing with control structure but not directly with loops. A good example is *code hoisting*, which eliminates copies of identical code on parallel paths in a flow graph. This can save space significantly, but would have no impact on execution time.

Interprocedural global optimizations are much more difficult to perform due to sensitivity and global dependence relationships. Sometimes, procedure integration can be performed to replace a call to a procedure by an instantiation of the procedure body. This may also trigger the discovery of other optimization opportunities. Interprocedure dependence analysis must be performed in order to reveal these opportunities.

Machine-Dependent Optimizations With a finite number of registers, memory cells, and functional units in a machine, the efficient allocation of machine resources affects both space and time optimization of programs. For example, *strength reduction* replaces complex operations by cheaper operations, such as replacing $2a$ by $a + a$, a^2 by $a * a$, and $length(S1 + S2)$ by $length(S1) + length(S2)$. We will address register and memory allocation problems in code generation methods in the next section.

Other flow control optimizations can be conducted at the machine level. Good examples include the elimination of unnecessary branches, the replacement of instruction sequences by simpler equivalent code sequences, instruction reordering and multiple instruction issues in superscalar processors. Machine-level parallelism is usually exploited at the fine-grain instruction level. System-level parallelism is usually exploited in coarse-grain computations.

10.4.3 Vectorization and Parallelization Methods

Besides scalar optimizations, we need to perform vector and/or parallel optimizations. The purpose is to improve the performance of programs that manipulate large data arrays or can be partitioned for parallel execution. *Vectorization* is the process of converting scalar looping operations into equivalent vector instruction execution. *Parallelization* aims at converting sequential code into parallel form, which can enable parallel execution by multiple processors.

An optimizing compiler that does vectorization automatically or semiautomatically with directives from programmers is called a *vectorizing compiler* or simply a *vectorizer*. Similarly, a *parallelizing compiler* should be designed to generate parallel code from sequential code automatically or semiautomatically. We introduce below various methods suggested for vectorization and parallelization. Vector hardware must be provided to speed up vector operations. Multiprocessors or multicomputers must be used to execute parallelized codes. Inhibitors of vectorization and parallelization are also identified in some program constructs in order to avoid unrewarding attempts.

Vectorization Methods We describe below several basic methods for vectorization. Many other methods can be found in the extensive literature available on the subject. We use Fortran 90 notation; for example, successive iterations in the following loop are totally independent:

```
      Do 20 I = 8, 120, 2
20       A(I) = B(I+3) + C(I+1)
```

This scalar loop can be converted into one *vector-add* instruction defined by the following array assignment:

```
A(8:120:2) = B(11:123:2) + C(9:121:2)
```

(1) Use of Temporary Storage Consider the following Do loop:

```
      Do 20 I = 1, N
         A(I) = B(I) + C(I)
20       B(I) = 2 * A(I+1)
```

This loop represents the following sequence of scalar operations:

```
A(1) = B(1) + C(1)
B(1) = 2 * A(2)
A(2) = B(2) + C(2)
B(2) = 2 * A(3)
        ⋮
```

In order to enable pipelined execution by vector hardware, we need to introduce a temporary array TEMP(1:N) to produce the following vector code:

```
TEMP(1:N) = A(2:N+1)
A(1:N) = B(1:N) + C(1:N)
B(1:N) = 2 * TEMP(1:N)
```

Without the TEMP array, the second array assignment B(l:N) may use the modified A(1:N) array which was not intended in the original code.

(2) *Loop Interchanging* Vectorization is often performed in the inner loop rather than in the outer loop. Sometimes we interchange the loops to enable the vectorization. The general rules for loop interchanges are to make the most profitable vectorizable loop the innermost loop, to make the most profitable parallelizable loop the outermost loop, to enable memory accesses to consecutive elements in arrays, and to bring a loop with longer vector length (iteration count) to the innermost loop for vectorization. The profitability is defined by improvement in execution time. Consider the following loop nest with two levels:

```
      Do 20 I = 2, N
         Do 10 J = 2, N
S1:         A(I, J) = (A(I, J - 1) + A(I, J + 1)/2            (10.6)
10       Continue
20    Continue
```

The statement S_1 is both flow-dependent and antidependent on itself with the direction vectors (=, <) and (=, >), or more precisely the distance vectors (0, –1) and (0, 1), respectively. This implies that the loop cannot be vectorized along the J-dimension. Therefore, we have to interchange the two loops:

```
    Do 20 J = 2, N
       Do 20 I = 2, N
          A(I, J) = (A(I, J - 1) + A(I, J + 1))/2
20  Continue
```

Now, the inner loop (I-dimension) can be vectorized with the zero distance vector. The vectorized code is

```
    Do 20 J = 2, N
       A(2:N, J) = (A(2:N, J - 1) + A(2:N, J + 1))/2
20  Continue
```

In general, an innermost loop cannot be vectorized if forward dependence direction (<) and backward dependence direction (>) coexist. The (=) direction does not prevent vectorization.

(3) *Loop Distribution* Nested loops can be vectorized by distributing the outermost loop and vectorizing each of the resulting loops or loop nests.

```
      Do 10 I = 1, N
         B(I, 1) = 0
         Do 20 J = 1, M
            A(I) = A(I) + B(I, J) * C(I, J)
20       Continue
         D(I) = E(I) + A(I)
10    Continue
```

The I-loop is distributed to three copies, separated by the nested J-loop from the assignment to array B and D, and vectorized as follows:

```
       B(1:N, 1) = 0 (a zero vector)
       Do 30 I = 1, N
          A(I) = A(I) + B(I, 1:M) * C(I, 1:M)
   30  Continue
       D(1:N) = E(1:N) + A(1:N)
```

(4) Vector Reduction We have defined vector reduction instructions in Eqs. 8.6 and 8.7. A number of reductions are defined in Fortran 90. In general, a vector reduction produces a scalar value from one or two data arrays. Examples include the *sum*, *product*, *maximum*, and *minimum* of all the elements in a single array. The *dot product* produces a scalar $S = \Sigma_{i=1}^{n} A_i \times B_i$ from two arrays A(1:n) and B(1:n). The loop

```
       Do 40 I = 1, N
S1:       A(I) = B(I) + C(I)
S2:       S = S + A(I)
S3:       AMAX = MAX(AMAX, A(I))
40     Continue
```

has the following dependence relations: $S_1(=)S_2$, $S_1(=)S_3$, $S_2(<)S_2$, $S_3 < S_3$. Although statements S_2 and S_3 each form a dependence cycle, each statement is recognized as a reduction operation and can be vectorized as follows:

```
S1:  A(1:N) = B(1:N) + C(1:N)
S2:  S = S + SUM(A(1:N))
S3:  AMAX = MAX(AMAX, MAXVAL(A(1:N)))
```

where SUM, MAX, and MAXVAL are all vector operations.

(5) Node Splitting The data dependence cycle can sometimes be broken by node splitting. Consider the following loop:

```
       Do 50 I = 2, N
S1:       T(I) = A(I – 1) + A(I + 1)
S2:       A(I) = B(I) + C(I)
50     Continue
```

Now we have the dependence cycle $S_1(<)S_2$ and $S_1(>)S_2$, which seems to prevent vectorization. However, we can split statement S_1 into two parts and apply statement reordering:

```
       Do 50 I=2, N
S1a:      X(I) = A(I + 1)
S2:       A(I) = B(I) + C(I)
S1b:      T(I) = A(I – 1) + X(I)
50     Continue
```

The new loop structure has no dependence cycle and thus can be vectorized:

```
S1a:  X(2:N) = A(3:N + 1)
S2:   A(2:N) = B(2:N) + C(2:N)
S1b:  T(2:N) = A(1:N − 1) + X(2:N)
```

It should be noted that node splitting cannot resolve all dependence cycles.

(6) Other Vector Optimizations There are many other methods for vectorization, and we do not intend to discuss them all. For example, *scalar variables* in a loop can sometimes be expanded into dimensional arrays to enable vectorization. *Subexpressions* in a complex expression can be vectorized separately. Loops can be *peeled*, *unrolled*, *rerolled*, or *tiled* (blocking) for vectorization.

Some machine-dependent optimizations can also be performed, such as *strip mining* (loop sectioning) and *pipeline chaining*, introduced in Chapter 8. Sometimes a vector register can be used as an accumulator, making it possible for the compiler to move loads and stores of the register outside the vector loop. The movement of an operation out of a loop to a basic block preceding the loop is called *hoisting*, and the inverse is called *sinking*. Vector loads and stores can be hoisted or sunk only when the array reference and assignment have the same subscripts and all the subscripts are the induction variable of a vectorized loop or loop constants.

Vectorization Inhibitors Listed below are some conditions inhibiting or preventing vectorization:

(1) Computed conditional statements such as IF statements which depend on runtime conditions.

(2) Multiple loop entries or exits (not basic blocks).

(3) Function or subroutine calls.

(4) Input/output statements.

(5) Recurrences and their variations.

A *recurrence* exists when a value calculated in one iteration of a loop might be referenced in another iteration. This occurs when dependence cycles exist between loop iterations. In other words, there must be at least one *loop-carried dependence* for a recurrence to exist. Any number of *loop-independent dependences* can occur in a loop, but a recurrence does not exist unless that loop contains at least one loop-carried dependence.

Code Parallelization Parallel code optimization spreads a single program into many threads for parallel execution by multiple processors. The purpose is to reduce the total execution time. Each *thread* is a sequence of instructions that must execute on a single processor. The most often parallelized code structure is performed over the outermost loop if dependence can be properly controlled and synchronized.

Consider the two-deep loop nest in Eq. 10.6. Because all the dependence relations have a "=" direction in the *I*-loop, this outer loop can be parallelized with no need for synchronization between the loop iterations. The parallelized code is as follows:

```
      Doall I = 2, N
         Do J = 2, N
S1 :        A(I, J) = (A(I, J − 1) + A(I, J + 1)) / 2
         Enddo
      Endall
```

Each of the $N-1$ iterations in the outer loop can be scheduled for a single processor to execute. Each newly created thread consists of one entire J-loop with a constant index value for I. If dependence does exist between the iterations, the Doacross construct can be used with proper synchronization among the iterations. The following example shows five execution modes of a serial loop execution to various combinations of parallelization and vectorization of the same program.

Example 10.8 Five execution modes of a FX/Fortran loop on the Alliant FX/80 multiprocessor (Alliant Computer Systems Corporation, 1989)

FX/Fortran generates code to execute a simple Do loop in *scalar, vector, scalar-concurrent, vector-concurrent,* and *concurrent outer/vector inner* (COVI) modes. The computations involved are performed either over a one-dimensional data array A(1:2048) or over a two-dimensional data array B(1:256, 1:8), where A(K)=B(I, J) for K=8(I – l)+J.

By using array A, the computations involved are expressed by a pure scalar loop:

```
Do K = 1, 2048
   A(K) = A(K) + S
Enddo
```

where S is a scalar constant. Figure 10.8a shows the scalar (serial) execution on a single processor in 30,616 clock cycles.

The same code can be vectorized into eight vector instructions and executed serially on a single processor equipped with vector hardware. Each vector instruction works on 256 iterations of the following loop:

```
A(1:2048:256) = A(1:2048:256) + S
```

The total execution is reduced to 6048 clock cycles as shown in Fig. 10.8b.

The scalar-concurrent mode is shown in Fig. 10.8c. Eight processors are used in parallel, performing the following scalar computations:

```
Doall J = 1, 8
   Do I = 1, 256
      B(I, J) = B(I, J) + S
   Enddo
Endall
```

Now, the total execution time is further reduced to 3992 clock cycles.

Figure 10.8d shows the vector-concurrent mode on eight processors, all equipped with vector hardware. The following vector codes are executed in parallel:

```
Doall J = 1, 8
   A(K:2040+K:8) = A(K:2040+K:8) + S
Endall
```

This vectorized execution by eight processors results in a total time of 960 clock cycles.

Finally, the same program can be executed in COVI mode. The inner loop executes in vector mode, and the outer loop is executed in parallel mode. In Fortran 90 notation, we have:

B(1:8, 1:256) = B(1:8, 1:256) + S

The total execution time is now reduced to 756 clock cycles, the shortest among the five execution modes.

A(1) = A(1) + S, A(2) = A(2) + S, ..., A(2048) = A(2048) + S

(a) Scalar execution on one processor in 30,616 cycles

A(1: 256) = A(1: 256) + S, A(257: 512) = A(257 : 512) + S, ...
A(1793 : 2048) = A(1793 : 2048) + S

(b) Vector execution on one processor sequentially in 6048 cycles

P_1: B(1, 1) = B(1, 1) + S, B(1, 2) = B(1, 2) + S, ..., B(1,256) = B(1, 256) + S
P_2: B(2, 1) = B(2, 1) + S, B(2, 2) = B(2, 2) + S, ..., B(2, 256) = B(2, 256) + S
⋮
P_8: B(8, 1) = B(8, 1) + S, B(8, 2) = B(8, 2) + S, ..., B(8,256) = B(8, 256) + S

(c) Scalar-concurrent execution on eight processors in 3992 cycles

P_1: A(1: 2041 : 8) = A(1 : 2041 : 8) + S
P_2: A(2: 2042 : 8) = A(2 : 2042 : 8) + S
⋮
P_8: A(8 : 2048 : 8) = A(8 : 2048 : 8) + S

(d) Vector-concurrent execution on eight processors in 960 cycles

P_1: B(1, 1 : 256) = B(1, 1 : 256) + S
P_2: B(2, 1 : 256) = B(2, 1 : 256) + S
⋮
P_8: B(8, 1 : 256) = B(8, 1 : 256) + S

(e) COVI execution on eight processors in 756 cycles

Fig. 10.8 Five execution modes of a FX/Fortran loop on the Alliant Multiprocessor (Courtesy of Alliant Computer Systems Corporation, 1989)

Inhibitors of Parallelization Most inhibitors of vectorization also prevent parallelization. Listed below are some inhibitors of parallelization:

(1) Multiple entries or exits.

(2) Function or subroutine calls.

(3) Input/output statements.

(4) Nondeterminism of parallel execution.

(5) Loop-carried dependences.

While only backward dependences interfere with vectorization, forward and backward dependences both affect parallelization. The overhead of synchronization code can outweigh performance gains from parallelization. We will illustrate this tradeoff analysis for multitasking on the Cray X-MP in Chapter 11. Most code parallelization is conducted at the loop level. To reduce or increase grain size, one must consider the tradeoffs between computations and communication. This is a difficult problem, and none of the existing compilers for parallelism has this capability. In most cases, the tradeoff studies are done by programmers. However, compiler directives can be used to guide the code optimization process.

10.4.4 Code Generation and Scheduling

Issues involved in code generation include order of execution, instruction selection, register allocation, branch handling, post-optimizations, etc. We describe the concepts of basic blocks and instruction scheduling schemes for basic blocks. Then we consider register allocation, pattern matching, and other table-driven methods for advanced code generation. How to expand code generation methods for multiple processors systematically is still a wide-open research area.

Directed Acyclic Graphs Because instructions within each basic block are sequenced without any backtracks, computations performed can thus be represented by a *directed acyclic graph* (DAG). A DAG can be built in one pass through a basic block. The nodes in a DAG represent *values*. Each interior node is labeled by the operator that produces its value. Edges on the DAG show the data dependence constraints. The children of a node are the nodes producing the operand values. The leaf nodes carry the initial values or constants existing on entry to a basic block.

DAG construction repeats the following steps from node to node. Consider the statement $A := B + C$ in a basic block. We first find nodes representing the values of B and C. If B and C are not computed in the block, they must be retrieved from leaf nodes. Otherwise, B and C should come from interior nodes of the DAG. Then we create a node labeled "+". Children of this node are the nodes for values of B and C. If there is already an identical node (same label and same child nodes), node creation can be skipped. The node for "+" becomes the current node for A. In the case of a data transfer operation $A := B$, find the node representing the value of B. Then the node representing B becomes the current node for A. Exceptions do exist. A procedure call must assume all variable values have changed. If a variable could possibly point to another variable, then that variable could now have a new value. Assignment to elements of an array must be assumed to alter the entire array.

Example 10.9 Construction of a DAG for the inner loop kernel of the bubble sort program (S. Graham, J. L. Hennessy, and J. D. Ullman, 1992)

Listed below are the statements contained in the basic block B7 of the bubble sort program in Fig. 10.7.

```
t8       :=  j - 1    }
t9       :=  4 * t8   } temp := A[j]
temp     :=  A[t9]    }

t10      :=  j + 1    }
t11      :=  t10 - 1  } A[j + 1]
t12      :=  4 * t11  }
t13      :=  A[t12]   }

t14      :=  j - 1    }
t15      :=  4 * t14  } A[j] := A[j + 1]
A[t15]   :=  t13      }

t16      :=  j + 1    }
t17      :=  t16 - 1  } A[j + 1] := temp
t18      :=  4 * t17  }
A [t18]  :=  temp     }
```

The corresponding DAG representation of block B7 is shown in Fig. 10.9. For nodes with the same operator, one or more names are labeled provided they consume the same operands (although they may use different values at different times). The initial value of any variable x is denoted by x_0, such as the A_0, j_0, and $temp_0$ at the leaf nodes.

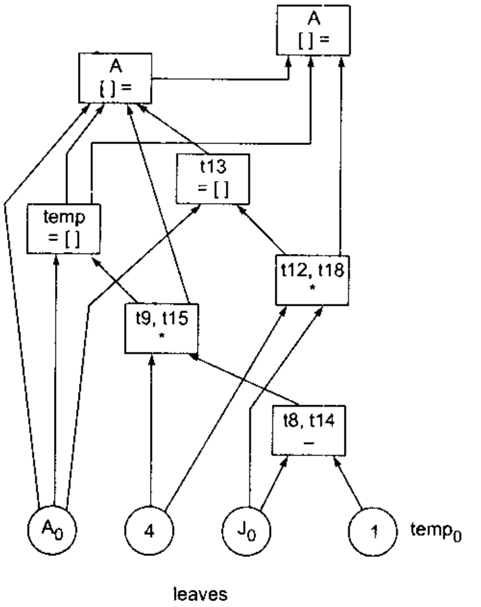


Fig. 10.9 Directed acyclic graph representation of the basic block B 7, the inner loop of the bubble sort program in Examples 10.6 and 10.7 (Courtesy of S. Graham, J. L. Hennessy, and J. D. Ullman, *Course on Code Optimization and Code Generation*, Western Institute of Computer Science, Stanford University, 1992)

In order to construct a DAG systematically, an auxiliary table can be used to keep track of variables and temporaries. The DAG construction process automatically detects common subexpressions and eliminates them accordingly. *Copy propagation* can be used to compute only one of the variables in each class. The construction process can easily discover the variables used or assigned and the nodes whose values can be computed at compile time. Any node whose children are constants is itself a constant. One can also label the edges in a DAG with the delays.

List Scheduling A DAG represents the flow of instructions in a basic block. A *topological sort* can be used to schedule the operations. Let READY be a buffer holding all nodes which are ready to execute. Initially, the READY buffer holds all leaf nodes with zero predecessors. Schedule each node in READY as early as possible, until it becomes empty. After all the predecessor (children) nodes are scheduled, the successor (parent) node should be immediately inserted into the READY buffer.

With list scheduling, each interior node is scheduled after its children. Additional ordering constraints are needed for a procedure call or assignment through a pointer. When the root nodes are reached, the schedule is produced. The length of the schedule equals the critical path on the DAG. To determine the critical path, both edge delays and nodal delays must be counted.

Some priority scheme can be used in selecting instructions from the READY buffer for scheduling. For example, the seven interior nodes of the DAG in Fig. 10.9 can be scheduled as follows, based on the topological order. In the case of two temporaries using the same node, we select the lower-numbered one. The following sequential code results:

```
t12 := 4 * j
t8 := j – 1
t13 := A[t12]
t9 := 4 * t8
temp := A[t9]
A[t9] := t13
A[t12] := temp
```

List scheduling schedules operations in topological order. There are no backtracks in the schedule. It is considered the best among critical-path, branch-and-bound for microinstruction scheduling (Joseph Fisher, 1979). Variations of topological list scheduling do exist such as introducing a *priority junction* for ready nodes, using *top-down* versus *bottom-up* direction, and using cycle scheduling as explained below. Whenever possible, parallel scheduling of nodes should be exploited, of course, subject to data, control, and resource dependence constraints.

Cycle Scheduling List scheduling is operation-based, which has the advantage that the highest-priority operation is scheduled first. Another scheduling method for instructions in basic blocks is based on a *cycle scheduling* concept in which "cycles" rather "operations" are scheduled in order. Let READY be a buffer holding nodes with zero unscheduled predecessors ready to execute in a current cycle. Let LEADER be a buffer holding nodes with zero unscheduled predecessors but not ready in a current cycle (e.g. due to some latency unfulfilled). The following cycle scheduling algorithm is modified from the list scheduling algorithm:

```
Current-cycle = 0
```

```
Loop until READY and LEADER are empty
    For each node n in READY (in decreasing priority order)
        Try to schedule n in current cycle
        If successful, update READY and LEADER
    Increment Current-cycle by 1
end of loop
```

The advantages of cycle scheduling include simplicity in implementation for single-cycle resources, such as in a superscalar processor. There is no need to keep records of source usage and it is also easier to keep track of register lifetimes. It can be considered an improvement over the list scheduling scheme, which may result in more idle cycles. LEADER provides another level of buffering. Nodes in LEADER that have become ready should be immediately loaded into the READY queue.

Register Allocation Traditional instruction scheduling methods minimize the number of registers used, which also reduces the degree of parallelism exploited. To optimize the code generated from a DAG, one can convert it to a sequence of *expression trees* and then study optimization for the trees. The registers can be allocated with instructions in the scheduling scheme.

In general, more registers would allow more parallelism. The above bottom-up scheduling methods shorten register lifetimes for expression trees. A round-robin scheme can be used to allocate registers while the schedule is being generated. Or one can assume an infinite number of registers to produce a schedule first and then allocate registers and add spill code later. Another approach is to integrate register allocation with scheduling by keeping track of the liveness of registers. When the remaining registers are greater than a given threshold, one should maximize parallelism. Otherwise, one should reduce the number of registers allocated.

Register allocation can be optimized by register descriptors (or tags) to distinguish among constant, variable, indexed variable, frame pointer, etc. This tagged register may enable some additional local or global code optimizations. Another advanced feature is special branch handling, such as delayed branches or using shorter delay slots if possible.

Code generation can be improved with a better instruction selection scheme. We can first generate code for expression trees needing no register spills. One can also select instructions by recursively matching templates to parts of expression trees. A match causes code to be generated and the subtree to be rewritten with a subtree for the result. The process ends when the subtree is reduced to a single node. When template matching fails, heuristics can be used to generate subgoals for another matching. The key ideas of instruction selection by pattern matching include:

(1) Convert code generation to primarily a systematic process.

(2) Use tree-structured patterns describing instructions and use a tree-structured intermediate form.

(3) Select instructions by covering input to instruction patterns.

Major issues in pattern-based instruction selection include development of pattern matching algorithms, design of intermediate form and target machine descriptions, and interaction with low-level optimization. Extensive table descriptions are needed. Therefore table compression techniques are needed to handle large numbers of patterns to be matched.

Advanced code generation needs to be directed toward exploitation of parallelism. Therefore special compilation support is needed for superscalar and multithreaded processors. These are still open research problems. Partial solutions to these problems can be found in subsequent sections. There is still a long way to go in developing really "intelligent" compilers for parallel computers.

10.4.5 Trace Scheduling Compilation

Branch prediction has been used in a software scheduling technique called *trace scheduling*. The idea was originally developed for scheduling and packing operations into *horizontal microinstructions*. Trace scheduling was proposed for use in VLIW architecture designed for scientific computation without vectorization.

The concept of trace scheduling is illustrated in Fig. 10.10. A *trace* is formed by a sequence of basic blocks, separated by assuming a particular outcome for every branch encountered in the sequence. The code example shows the first trace involving three basic blocks (A, B, and C). There are many traces for different combinations of branch outcomes. The second trace corresponds to another branch combination. Each trace is scheduled for parallel execution by a VLIW processor.

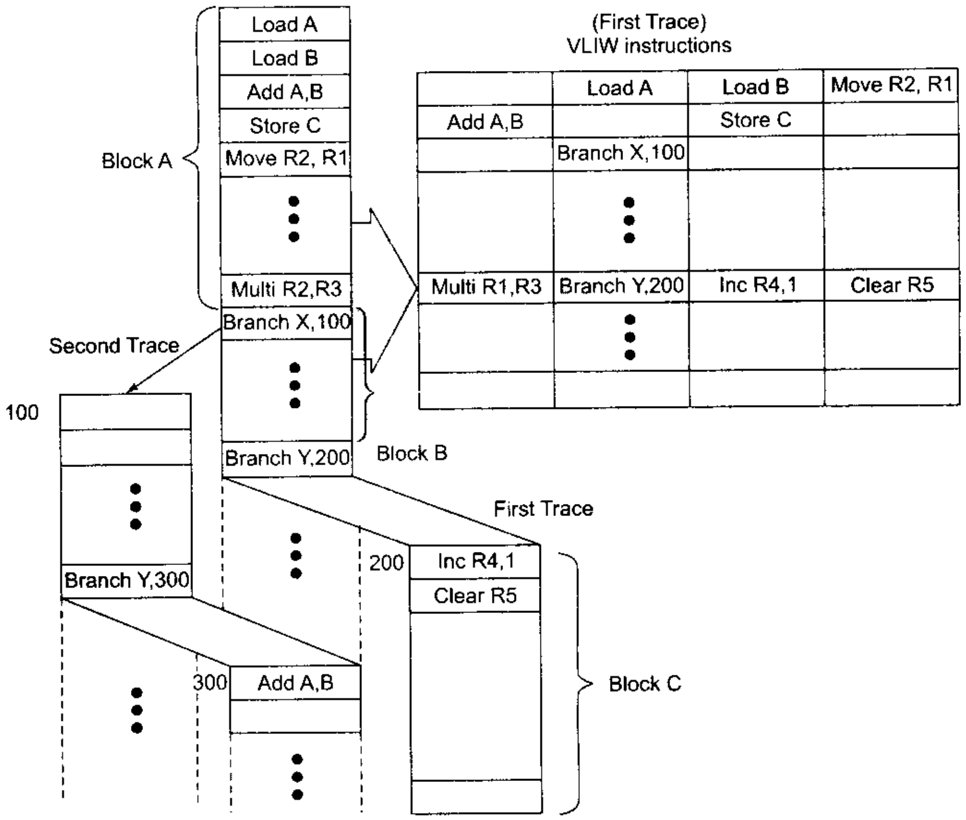


Fig. 10.10 Code compaction for VLIW processor based on trace scheduling developed by Joshep Fisher (1981)

Code Compaction Independent instructions in the trace are compacted into VLIW instructions. Each VLIW word can be packed with multiple short instructions which can be independently executed in parallel.

The decoding of these independent instructions is carried out simultaneously. Multiple function units (such as the memory-access unit, arithmetic unit, branch unit, etc.) are employed to carry out the parallel execution. Only independent operations are packed into a VLIW instruction.

Branch prediction is based on software heuristics or on using profiles of previous program executions. Each trace should correspond to the most likely execution path. The first trace should be the most likely one, the second trace should be the second most likely one, and so on. In fact, reduction in the execution time of an earlier trace is obtained at the expense of that of later traces. In other words, the execution time of likely traces is reduced at the expense of that of unlikely traces.

Compensation Code The effectiveness of trace scheduling depends on correct predictions at successive branches in a program. To cope with the problem of code movement followed by incorrect prediction, compensation codes are added to off-trace paths to provide correct linkage with the rest of program. Because code compaction may move short instructions up or down in the program, different compensation codes must be inserted to restore the original code distribution in the code blocks involved.

Example 10.10 Trace scheduling with code compaction and compensation

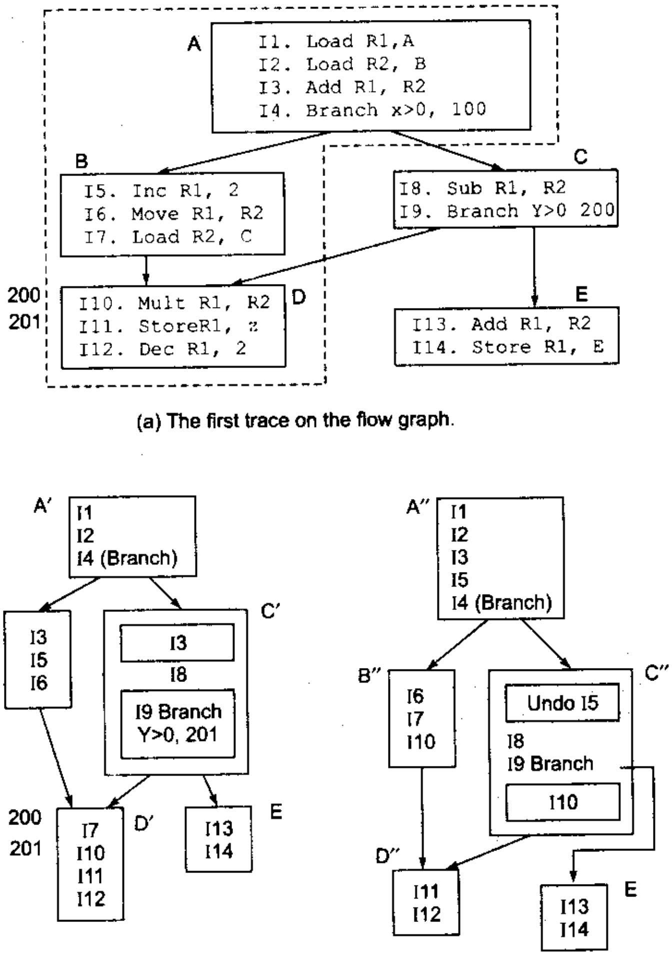
Consider an example program consisting of five basic blocks in Fig. 10.11a. The initial trace contains blocks A, B, and D, after it is predicted that execution of I4 and I9 will lead to the left path. In Fig. 10.11b, instruction I3 has been moved from block A to block B, and I7 moved to block D, to form the new blocks A′, B′ and D′.

Therefore, we need to insert instruction I3 in block C′ also and modify the target address to 201 in branch instruction I9. Similarly, some instructions have been moved up to preceding blocks (Fig. 10.11c). Compensation codes, Undo I5 and I10, must be inserted in block C′ to restore the original program semantics.

Compensation codes (in shaded boxes in Fig. 10.11) are needed on off-trace paths.This will make the second trace correctly executed, without being affected by the first trace due to code movement. The compensation code added should be a small portion of the code blocks. Sometimes the Undo operation cannot be used due to the lack of an inverse operation in the instruction set. By adding compensation code, software is performing the function of the branch history buffer described in Chapter 6.

The efficiency of trace scheduling depends on the degree of correct prediction of branches. With accurate branch predictions, the compensation code added may not be executed at all. The fewer the number of most likely traces to be executed, the better the performance will be.

Trace scheduling was mainly designed for VLIW processors. For superscalar processors, similar techniques can exploit parallelism in a program whose branch behavior is relatively easier to predict, such as in some scientific applications.



(a) The first trace on the flow graph.

(b) Downward code motion.

(c) Upward code motion

Fig. 10.11 Code motions for trace scheduling compacting in Example 10.10

10.5 LOOP PARALLELIZATION AND PIPELINING

This section describes the theory and application of loop transformations for vectorization or parallelization purposes. At the end, we address software pipelining techniques.

10.5.1 Loop Transformation Theory

Parallelizing loop nests is one of the most fundamental program optimization techniques demanded in a vectorizing and parallelizing compiler. In this section, we study a loop transformation theory and loop transformation algorithms derived from this theory. The bulk of the material is based on the work by Wolf and Lam (1991).

The theory unifies all combinations of loop interchange, skewing, reversal, and tilting as *unimodular transformations*. The goal is to maximize the degree of parallelism or data locality in a loop nest. These transformations also support efficient use of the memory hierarchy on a parallel machine.

Elementary Transformations A loop transformation rearranges the execution order of the iterations in a loop nest. Three elementary loop transformations are introduced below.

(1) *Permutation*—A permutation δ on a loop nest transforms iteration $(p_1, \ldots, p_n)$ to $(p\delta_1, \ldots, p\delta_n)$. This transformation can be expressed in matrix form as I_δ, the $n \times n$ identity matrix with rows permuted by δ. For $n = 2$, a loop interchange transformation maps iteration (i, j) to iteration (j, i). In matrix notation, we can write this as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix}$$

The following two-deep loop nest is being transformed using the permutation matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ as depicted in Fig. 10.12a.

```
Do i = 1, N                        Do j = 1, N
   Do j = 1, N                        Do i = 1, N
      A(j) = A(j) + C(i, j)   ⇒          A(j) = A(j) + C(i, j)
   Enddo                              Enddo
Enddo                              Enddo
```

(2) *Reversal*—Reversal of the *i*th loop is represented by the identity matrix with the *i*th element on the diagonal equal to -1.

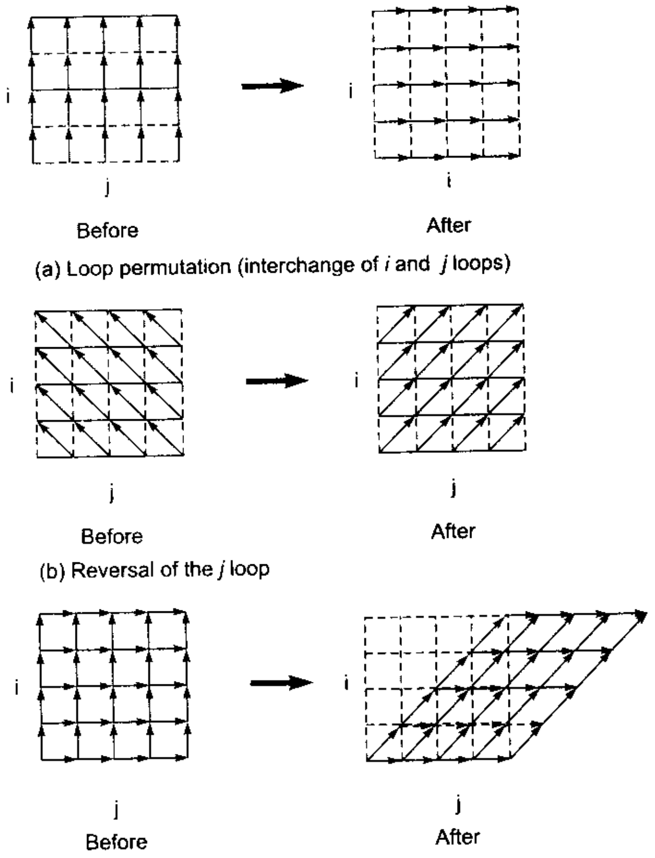
The following loop nest is being reversed using the transformation matrix $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ as depicted in Fig. 10.12b.

```
Do i = 1, N                           Do i = 1, N
   Do j = 1, N                           Do j = -N, -1
      A(i, j) = A(i -1, j + 1)   ⇒          A(i, -j) = A(i -1, -j + 1)
   Enddo                                 Enddo
Enddo                                 Enddo
```

(3) *Skewing*—Skewing loop I_j by an integer factor f with respect to loop I_i maps iteration $(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_n)$ to $(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j + fp_i, p_{j+1}, \ldots, p_n)$.

In the following loop nest, the transformation performed is a skew of the inner loop with respect to the outer loop by a factor of 1, represented by the transformation matrix $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ as depicted in Figure 10.12c.

```
Do i = 1, N                          Do j = 1, N
   Do j = 1, N                          Do j = 1, N
      A(i, j) = A(i, j - 1) +   ⇒          A(i, j - i) = A(i, j - i - 1) +
      A(i - 1, j)                          A(i - 1, j - i)
   Enddo                                Enddo
Enddo                                Enddo
```



(a) Loop permutation (interchange of *i* and *j* loops)

(b) Reversal of the *j* loop

(c) Skewing of the inner loop by a factor of 1

Fig. 10.12 Loop transformations performed in Example 10.9 (Courtesy of Monica Lam, *WISC Tutorial Nores*, Standford University, 1992)

Various combinations of the above elementary loop transformations can be defined. Wolf and Lam have called these *unimodular transformations*. The optimization problem is thus to find the unimodular transformation that maximizes an objective function given a set of schedule constraints.

Transformation Matrices Unimodular transformations are defined by *unimodular matrices*. A unimodular matrix has three important properties. First, it is square, meaning that it maps an n-dimensional iteration space into an n-dimensional iteration space. Second, it has all integer components, so it maps integer vectors to integer vectors. Third, the absolute value of its determinant is 1.

Because of these properties, the product of two unimodular matrices is unimodular, and the inverse of a unimodular matrix is unimodular, so that combinations of unimodular loop transformations and the inverse of unimodular loop transformations are also unimodular loop transformations. A loop transformation is said to be *legal* if the transformed dependence vectors are all lexicographically positive.

A compound loop transformation can be synthesized from a sequence of primitive transformations, and the effect of the loop transformation is represented by the products of the various transformation matrices for each primitive transformation. The major issues for loop transformation include how to apply a transform, correctness or locality, and desirability or advantages of applying a transform. Wolf and Lam (1991) have stated the following conditions for unimodular transformations:

(1) Let D be the set of distance vectors of a loop nest. A unimodular transformation (matrix) T is *legal*, if and only if $\forall d \in D$,

$$T \cdot d \geq 0 \tag{10.7}$$

(2) Loops i through j of a nested computation with dependence vectors D are *fully permutable*, if $\forall d \in D$,

$$((d_{1,} d_2, \ldots, d_{i\;\;1}) > 0 \quad \text{or} \quad (\forall i \leq k \leq j : d_k \geq 0) \tag{10.8}$$

Proofs of these two conditions are left as exercises for the reader. The following example shows how to determine the legality of unimodular transformations.

```
Do i = 1, N
    Do j = 1, N
        A(i,j) = f(A(i,j), A(i + 1, j - 1))
    Enddo
Enddo
```

This code has the dependence vector $d = (1, -1)$. The loop interchange transformation is represented by the matrix

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The transformation is illegal since $T \cdot d = (-1, 1)$ is lexicographically negative. However, compounding the interchange with a reversal represented by the transformation matrix

$$T' = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

is legal because $T' \cdot d = (1, 1)$ is lexicographically positive.

10.5.2 Parallelization and Wavefronting

The theory of loop transformation can be applied to execute loop iterations in parallel. In this section, we describe loop parallelization procedures. A wavefronting approach is presented for fine-grain parallelization. Tiling is applied to reduce synchronization costs in coarse-grain computations.

Parallelization Conditions The purpose of loop parallelization is to maximize the number of parallelizable loops. For n-deep loops whose dependences can be represented with distance vectors, at least $(n - 1)$ degrees of parallelism can be exploited in both fine-grain and coarse-grain computations.

The algorithm for loop parallelization consists of two steps: It first transforms the original loop nest into a canonical form, namely, a *fully permutable* loop nest. It then transforms the fully permutable loop nest to exploit coarse- and/or fine-grain parallelism according to the target architecture.

(1) *Canonical form*. Loops with distance vectors have the special property that they can always be transformed into a fully permutable nest via skewing. It is easy to determine how much to skew an inner loop with respect to an outer one to make these loops fully permutable. For example, if a doubly nested loop has dependences $\{(0, 1), (1, -2), (1, -1)\}$, then skewing the inner loop by a factor of 2 with respect to the outer loop produces $\{(0, 1), (1, 0), (1, 1)\}$.

(2) *Parallelization process*. Iterations of a loop can execute in parallel if and only if no dependences are carried by that loop. Such a loop is called a Doall loop. To maximize the degree of parallelism is to transform the loop nest to maximize the number of Doall loops.

Let $(I_1, \ldots, I_n)$ be a loop nest with lexicographically positive dependences $d \in D$. I_i is parallelizable if and only if $\forall d \in D$, $(d_1, \ldots, d_{i-1}) > (0, \ldots, 0)$, the zero vector, or $d_i = 0$. Once the loops are made fully permutable, the steps to generate Doall parallelism are simple. In the following discussion, we show that the loops in canonical form can be trivially transformed to produce both fine- and coarse-grain parallelism.

Fine-Grain Wavefronting A nest of n fully permutable loops can be transformed into code containing at least $(n - 1)$ degrees of parallelism. In the degenerate case where no dependences are carried by these n loops, the degree of parallelism is n. Otherwise, $(n - 1)$ parallel loops can be obtained by skewing the innermost loop in the fully permutable nest by each of the other loops and moving the innermost loop to the outermost position.

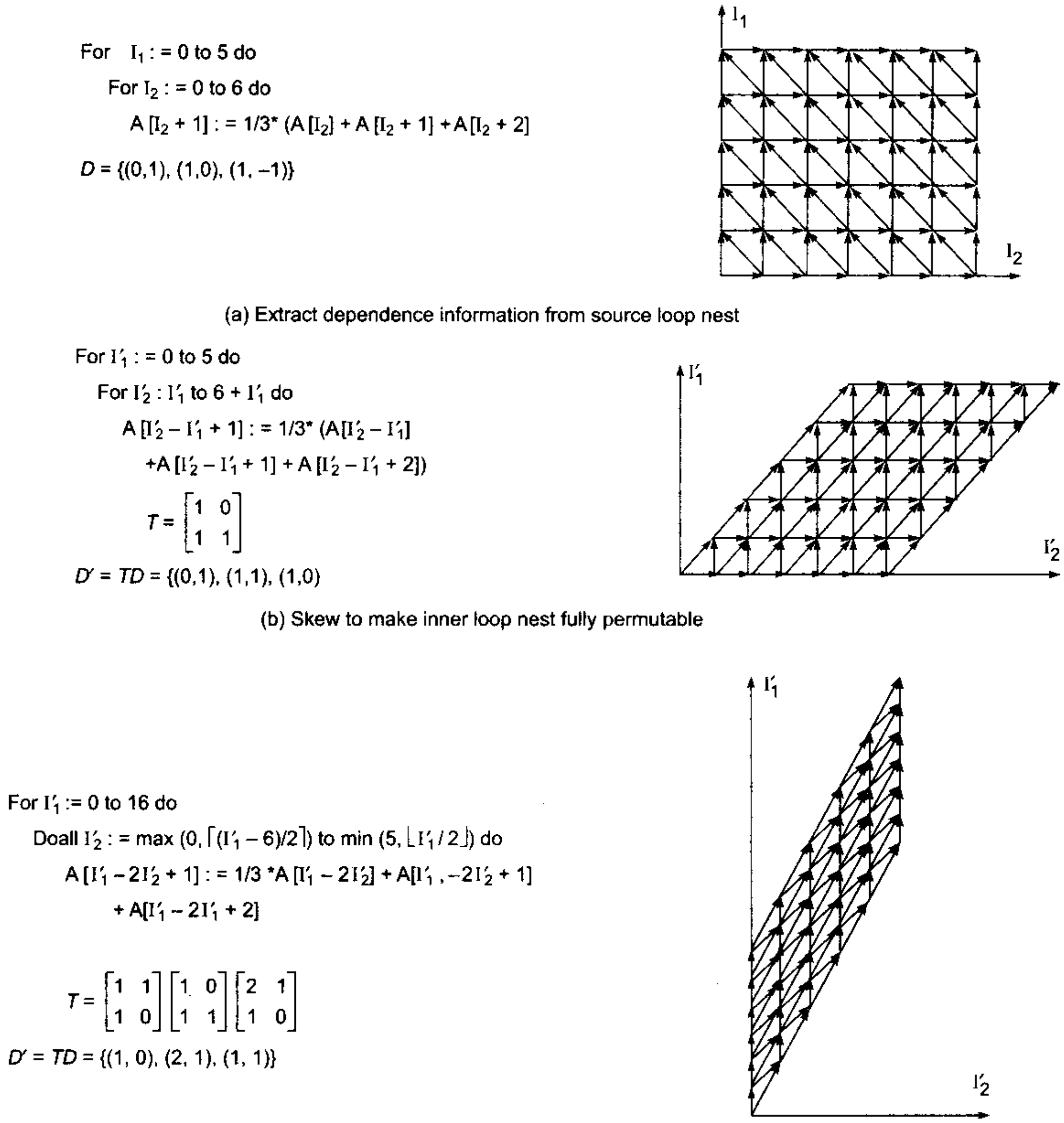
This transformation, called *wavefront transformation*, is represented by the following matrix:

$$T = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{10.9}$$

Fine-grain parallelism is exploited on vector machines, superscalar processors, and systolic arrays. The following example shows the entire process of loop parallelization exploiting fine-grain parallelism. The process includes skewing and wavefront transformation.

Example 10.11 Loop skewing and wavefront transformation (Michael Wolf and Monica Lam, 1991)

Figure 10.13a shows the iteration space and dependence of a source loop nest. The skewed loop nest is shown in Fig. 10.13b after applying the matrix $T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. Figure 10.13c shows the result of applying wavefront transformation to the skewed loop code, which is a result of first skewing the innermost loop to make the two-dimensional loop nest fully permutable and then applying the wavefront transformation to create one degree of parallelism.



```
For $I_1$ : = 0 to 5 do
    For $I_2$ : = 0 to 6 do
        A [$I_2$ + 1] : = 1/3* (A [$I_2$] + A [$I_2$ + 1] + A[$I_2$ + 2]
$D = \{(0,1), (1,0), (1, -1)\}$
```

(a) Extract dependence information from source loop nest

```
For $I'_1$ : = 0 to 5 do
    For $I'_2$ : $I'_1$ to 6 + $I'_1$ do
        A [$I'_2 - I'_1$ + 1] : = 1/3* (A[$I'_2 - I'_1$]
            +A [$I'_2 - I'_1$ + 1] + A [$I'_2 - I'_1$ + 2])
```

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$D' = TD = \{(0,1), (1,1), (1,0)$

(b) Skew to make inner loop nest fully permutable

```
For $I'_1$ := 0 to 16 do
    Doall $I'_2$ : = max (0, $\lceil (I'_1 - 6)/2 \rceil$) to min (5, $\lfloor I'_1 / 2 \rfloor$) do
        A [$I'_1 - 2I'_2$ + 1] : = 1/3 *A [$I'_1 - 2I'_2$] + A[$I'_1$ , $-2I'_2$ + 1]
            + A[$I'_1 - 2I'_1$ + 2]
```

$$T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

$D' = TD = \{(1, 0), (2, 1), (1, 1)\}$

(c) Wavefront transformation on the skewed loop nest

Fig. 10.13 Fine-grain parallelization by loop skewing and wavefront transformation in Example 10.11 (Courtesy of Wolf and Lam; reprinted from *IEEE Trans. Parallel Distributed Systems*, 1991)

There are no dependences between iterations within the innermost loop nest. The transform is a wavefront transformation because it causes iterations along the diagonal of the original loop nest to execute in parallel.

This wavefront transformation automatically places the maximum Doall loops in the innermost loops, maximizing fine-grain parallelism. This is the appropriate transformation for superscalar or VLIW machines. Although these machines have a low degree of parallelism, finding multiple parallelizable loops is still useful. Coalescing multiple Doall loops prevents the pitfall of parallelizing only a loop with a small iteration count.

Coarse-Grain Parallelism For MIMD coarse-grain multiprocessors, having as many outermost Doall statements as possible reduces the synchronization overhead. A wavefront transformation produces the maximum degree of parallelism but makes the outermost loop sequential if any are. For example, consider the following loop nest:

```
Do i = 1, N
   Do j = 1, N
      A(i,j) = f(A(i-1,j-1))
   Enddo
Enddo
```

This loop nest has the dependence (1, 1), and so the outermost loop is sequential and the innermost loop is a Doall. The wavefront transformation $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ does not change this. In contrast, the unimodular transformation $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$ transforms the dependence to (0, 1), making the outer loop a Doall and the inner loop sequential.

In this example, the dimensionality of the iteration space is two, but the dimensionality of the space spanned by the dependence vectors is only one. When the dependence vectors do not span the entire iteration space, it is possible to perform a transformation that makes outermost Doall loops.

A heuristic though nonoptimal approach for making loops Doall is simply to identify loops I_i such that all d_i are zero. Those loops can be made outermost Doall. The remaining loops in the tile can be wavefronted to obtain the remaining parallelism.

Loop parallelization can be achieved through unimodular transformations as well as tiling. For loops with distance vectors, n-deep loops have at least $(n-1)$ degrees of parallelism. The loop parallelization algorithm has a common step for fine- and coarse-grain parallelism in creating an n-deep fully permutable loop nest by skewing. The algorithm can be tailored for different machines based on the following guidelines:

- Move Doall loop innermost (if one exists) for fine-grain machines. Apply a wavefront transformation to create up to $(n-1)$ Doall loops.
- Create outermost Doall loops for coarse-grain machines. Apply tiling to a fully permutable loop nest.
- Use tiling to create loops for both fine- and coarse-grain machines.

10.5.3 Tiling and Localization

Tiling and locality optimization techniques are studied in this section. The ultimate purpose is to reduce synchronization overhead and to enhance multiprocessor efficiency when loops are distributed for parallel execution.

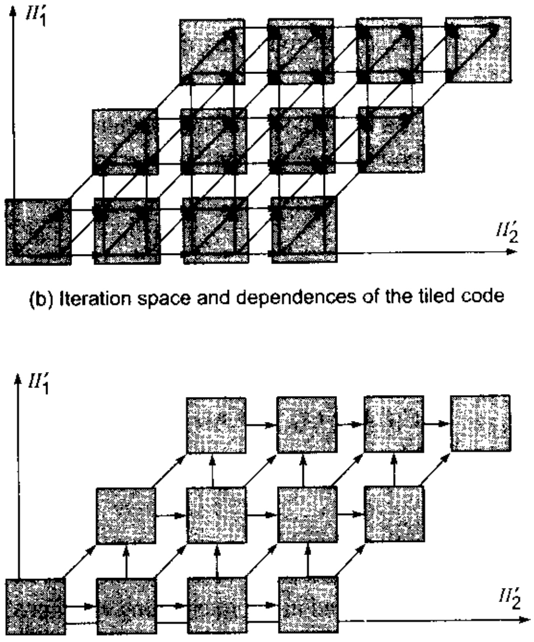
Tiling to Reduce Synchronization It is possible to reduce the synchronization cost and improve the data locality of parallelized loops via an optimization known as *tiling* (Wolfe, 1989). Tiling is not a unimodular transformation. In general, tiling maps an n-deep loop nest into a $2n$-deep loop nest where the inner n loops include only a small fixed number of iterations. Figure 10.14a shows the code after tiling the example in Fig. 10.13b using a tile size of 2 × 2. The two innermost loops execute the iterations within each tile, represented as 2 × 2 squares. The two outer loops, represented by the two axes, execute the 3 × 4 tiles. The outer loops of the tiled code control the execution of the tiles.

The same property that supports parallelization, full permutability, is also the key to tiling; Loops I_i through I_j of a legal computation can be tiled if they are fully permutable.

Thus, loops in the canonical form of the parallelization algorithm can also be tiled. Moreover, the characteristics of the controlling loops resemble those of the original set of loops. An abstract view giving the dependences of the tiles is shown in Fig. 10.14b. These controlling loops are themselves permutable and so are easily parallelizable. However, each iteration of the outer n loops is a tile of iterations instead of an individual iteration. Parallel execution of the tiled loops is shown in Fig. 10.14c.

```
For $II'_1$ : = 0 to 5 by 2 do
    For $II'_2$ : = 0 to 11 by 2 do
        For $I'_1$ : = $II'_1$ to min ($I'_1$ + 1, 5) do
            For $I'_2$ : = max ($II'_1$, $II'_2$) to min (6 + $I'_1$, $II'_2$ + 1) do
                A [$I'_2$] : = 1/3* (A[$I'_2$ – 1] + A [$I'_2$] + A [$I'_2$ + 1])
```

(a) Tiled code from the skewed code in Fig. 10.13b



(b) Iteration space and dependences of the tiled code

(c) Parallel execution of the tiled loops.

Fig. 10.14 Tiling of the skewed loops for parallel execution on a coarse-grain multiprocessor (Courtesy of Wolf and Lam; reprinted from *IEEE Trans. Parallel Distributed Systems*, 1991)

Tiling can therefore increase the granularity of synchronization and data are often reused within a tile. Without tiling, when a Doall loop is nested within a non-Doall loop, all processors must be synchronized with a barrier at the end of each Doall loop.

Using tiling, we can reduce the synchronization cost in the following two ways. First, instead of applying wavefront transformation to the loops in canonical form, we first tile the loops and then apply a wavefront transformation to the controlling loops of the tiles. In this way, the synchronization cost is reduced by the size of the tile. Certain loops cannot be represented as distances. Direction vectors can be used to represent these loops. The idea is to represent direction vectors as an infinite set of distance vectors.

Locality optimization in user programs is meant to reduce memory-access penalties. Software pipelining can reduce the execution time. Both are desired improvements in the performance of parallel computers. Program locality can be enhanced with loop interchange, reversal, tiling, and prefetching techniques. The effort requires a reuse analysis of a "localized" iteration space. Software pipelining relies heavily on sufficient support from a compiler working effectively with the scheduler.

The fetch of successive elements of a data array is pipelined for an interleaved memory. In order to reduce the access latency, the loop nest can interchange its indices so that a long vector is moved into the innermost loop, which can be more effective with pipelined loads. Loop transformations are performed to reuse the data as soon as possible or to improve the effectiveness of data caches.

Prefetching is often practiced to hide the latency of memory operations. This includes the insertion of special instructions to prefetch data from memory to cache. This will enhance the cache hit ratio and reduce the register occupation rate with a small prefetch overhead. In scientific codes, data prefetching is rather important. Instruction prefetching, as studied in Chapter 6, is often practiced in modern processors using prefetch buffers and an instruction cache. In the following discussion, we concentrate on data prefetching techniques.

Tiling for Locality Blocking or tiling is a well-known technique that improves the data locality of numerical algorithms. Tiling can be used for different levels of memory hierarchy such as physical memory, caches, and registers; multilevel tiling can be used to achieve locality at multiple levels of the memory hierarchy simultaneously.

To illustrate the importance of tiling, consider the example of matrix multiplication:

```
Do $i = 1, N$
    Do $j = 1, N$
        Do $k = 1, N$
            $C(i, k) = C(i, k) + A(i, j) \times B(j, k)$
        Enddo
    Enddo
Enddo
```

In this code, although the same rows of C and B are reused in the next iteration of the middle and outer loops, respectively, the large volume of data used in the intervening iterations may replace the data from the register file or the cache before they can be reused. Tiling reorders the execution sequence such that iterations from loops of the outer dimensions are executed before all the iterations of the inner loop are completed. The tiled matrix multiplication is:

```
Do $\ell = 1, N, s$
    Do $m = 1, N, s$
        Do $i = 1, N$
            Do $j = \ell, \min(\ell + s - 1, N)$
                Do $k = m, \min(m + s - 1, N)$
                    $C(i, k) = C(i, k) + A(i, j) \times B(j, k)$
                Enddo
            Enddo
        Enddo
    Enddo
Enddo
```

Tiling reduces the number of intervening iterations and thus data fetched between data reuses. This allows reused data to still be in the cache or register file and hence reduces memory accesses. The tile size s can be chosen to allow the maximum reuse for a specific level of memory hierarchy. For example, the tile size is relevant to the cache size used or the register file size used.

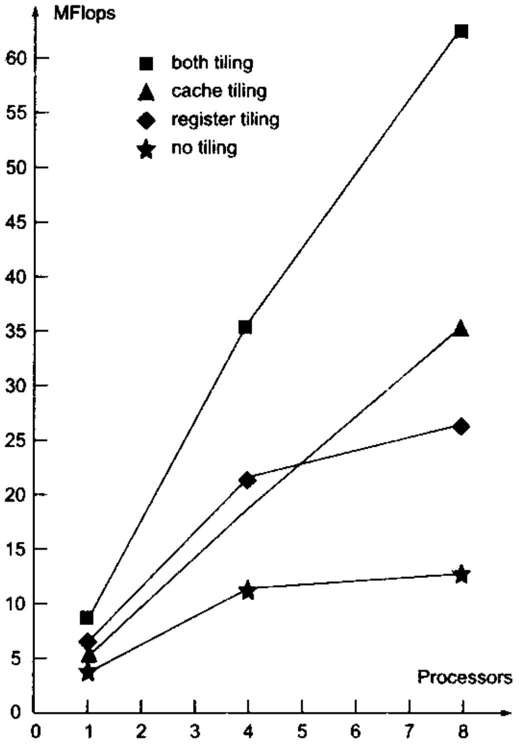


Fig. 10.15 Performance of a 500 × 500 double precision matrix multiplication on the SGI 4D/380. Cache tiles are 64 × 64 iterations and register tiles are 4 × 2 (Courtesy of Wolf and Lam; reprinted from ACM SIGPLAN *Conf. Programming Language Design and Implementation*, Toronto, Canada, 1991)

The improvement obtained from tiling can be far greater than that obtained from traditional compiler optimizations. Figure 10.15 shows the performance of 500 × 500 matrix multiplication on an SGI 4D/380 machine consisting of eight MIPS R3000 processors running at 33 MHz. Each processor has a 64-Kbyte direct-mapped first-level cache and a 256-Kbyte direct-mapped second-level cache. Results from four different experiments are reported: without tiling, tiling to reuse data in caches, tiling to reuse data in registers, and tiling for both register and caches. For cache tiling, the data are copied into consecutive locations to avoid cache interference.

Tiling improves the performance on a single processor by a factor of 2.75. The effect of tiling on multiple processors is even more significant since it reduces not only the average data-access latency but also the required memory bandwidth. Without cache tiling, contention over the memory bus limits the speedup to about 4.5 times. Cache tiling permits speedups of over 7 for eight processors, achieving an overall speed of 64 Mflops when combined with register tiling.

Localized Iteration Space Reusing vector spacing offers opportunities for locality optimization. However, reuse does not imply locality. For example, if reuse does not occur soon enough, it may miss the temporal locality. Therefore, the idea is to make reuse happen soon enough. A *localized iteration space* contains the iterations that can exploit reuse. In fact, tiling increases the number of dimensions in which reuse can be exploited.

Consider the following two-deep nest. Reuse is exploited for loops *i* and *j* only, which form the localized iteration space:

```
Do $i = 1$, N
    Do $j = 1$, N
        $B(i, j) = f(A(i), A(j))$
    Enddo
Enddo
```

Reference A(*j*) touches different data within the inner loop but reuses the same elements across the outer loop. More precisely, the same data A(*j*) is used in iterations (i, j), $1 \le i \le N$. There is reuse, but the reuse is separated by accesses to $N - 1$ other data. When N is large, the data is removed from the cache before it can be reused, and there is no locality. Therefore, a reuse does not guarantee locality. The tiled code is shown below:

```
Do $\ell = 1$, N, s
    Do $i = 1$, N
        Do $j = \ell$, $\max(\ell + s - 1, N)$
            $B(i, j) = f(A(i), A(j))$
        Enddo
    Enddo
Enddo
```

We choose the tile size such that the data used within the tile can be held within the cache. For this example, as long as s is smaller than the cache size, A(*j*) will still be present in the cache when it is reused. Thus, reuse is exploited for loops *i* and *j* only, and so the localized iteration space includes only these loops.

In general, if *n* is the first loop with a large bound, counting from innermost to outermost, then reuse occurring within the inner *n* loops can be exploited. Therefore the localized vector space of a tiled loop is simply that of the innermost tile, whether the tile is coalesced or not.

Obviously, memory optimizations are important. Locality can be upheld by intersecting the localized vector space with the reuse vector space. In other words, reuse directs the search for unimodular and tiling transformations. One should use locality information to eliminate unnecessary prefetches.

10.5.4 Software Pipelining

This refers to the pipelining of successive iterations of a loop in the source programs. The advantage of software pipelining is to reduce the execution time with compact object code. The idea was validated by implementation of a compiler for Warp, a systolic array of 10 processors built at CMU (Lam, 1988). Obviously, software pipelining is more effective for deep hardware pipelines. The concept is illustrated with an example taken from Lam's Tutorial Notes on Compilers for Parallel Machines (1992).

Pipelining of Loop Iterations Successive iterations of the following loop nest are to be executed on a two-issue processor first without software pipelining and then with pipelining.

```
Do I = 1, N
    A(I) = A(I) × B + C
Enddo
```

This is an example of Doall loops in which all iterations are independent. It is assumed that each memory access (*Read* or *Write*) takes one cycle and each arithmetic operation (*Mul* and *Add*) requires two cycles. Without pipelining, one iteration requires six cycles to execute as listed below:

Cycle	Instruction	Comment
1	Read	/Fetch A[I]/
2	Mul	/Multiply by B/
3		
4	Add	/Add to C/
5		
6	Write	/Store A[I]/

Therefore, N iterations require 6N cycles to complete, ignoring the loop control overhead. Listed below is the execution of the same code on an 8-deep instruction pipeline:

Cycle	Iteration			
	1	2	3	4
1	Read			
2	Mul			
3		Read		
4		Mul		
5	Add		Read	
6			Mul	

7		Add		Read
8	Write			Mul
9			Add	
10		Write		
11				Add
12			Write	
13				
14				Write

Four iterations of the software-pipelined code are shown. Although each iteration requires 8 cycles to flow through the pipeline, the four overlapped iterations require only 14 clock cycles to execute. Compared with the nonpipelined execution, a speedup factor of 24/14 = 1.7 is achieved with the pipelining of four iterations.

N iterations require $2N + 6$ cycles to execute with the pipeline. Thus, a speedup factor of $6N/(2N + 6)$ is achieved. As N approaches infinity, a speedup factor of 3 is expected. This shows the advantage of software pipelining, if other overhead is ignored.

Doacross Loops Unlike unrolling, software pipelining can give optimal results. Locally compacted code may not be globally optimal. The Doall loops can fill arbitrarily long pipelines with infinite iterations. In the following Doacross loop with dependence between iterations, software pipelining can still be done but is harder to implement.

```
Doacross I = 1, N
         A(I) = A(I) × B
         Sum = Sum + A(I);
Enddo
```

The software-pipelined code is shown below:

1	Read	
2	Mul	
3	Add	Read
4	Write	Mul
5		Add
6		Write

It is assumed that one memory access and one arithmetic operation can be concurrently executed on the two-issue superscalar processor in each cycle. Thus recurrences can also be parallelized with software pipelining.

As in the hardware pipeline scheduling in Chapter 6, the objective of software pipelining is to minimize the interval at which iterations are initiated; i.e. the *initiation latency* determines the throughput for the loop. The basic units of scheduling are minimally indivisible sequences of microinstructions. In the above Doall loop example, the initiation latency is two cycles per iteration, and the Doacross loop is also pipelined with an initiation latency of two cycles.

To summarize, software pipelining demands maximizing the throughput by reducing the initiation latency, as well as by producing small, compacted code size. The trick is to find an identical schedule for every iteration with a constant initiation latency. The scheduling problem is tractable, if every operation takes unit execution time and no cyclic dependences exist in the loop.

Summary

A programming model is a collection of program abstractions which present the programmer with a well-defined view of the software and hardware system. Parallel programming models are defined for the various types of parallel architectures which we have studied in the earlier chapters of the book. We started this chapter with a study of the parallel programming models which have become well-established, namely: shared variable model, message-passing model, data parallel model, object-oriented model and functional and logic models.

For any given model for parallel programming, the user needs to be provided with a parallel programming environment, which consists of parallel languages, compilers, support tools for program development, and runtime support. The programming environment must provide specific features for parallelism aimed at: optimization, availability, synchronization and communication, control of parallelism, data parallelism, and process management. Parallel language constructs are needed in the programming language, of which a few examples have been presented in this chapter. And the compiler must be capable of optimizing the machine code generated for the type of parallelism available in hardware.

Vectorizing or parallelizing compilers can, in theory, detect and exploit the potential parallelism which is present in a sequential program. In this process, dependence analysis of data arrays can reveal the presence or absence of dependences between successive references to array elements in a loop, or in nested loops. In general, two operations can be carried out in parallel only if there is no data or control dependence between them. We reviewed some specific techniques for dependence analysis of data arrays, such as iteration space analysis, subscript separability and partitioning, and categorized dependence tests.

Optimization of the machine code generated by the compiler, and cycle-by-cycle scheduling of machine instructions for execution on the processor, are both critical to achieving high performance computing. Local optimization can be carried out within basic blocks, but in general both local and global optimizations are required. We studied several vectorization and parallelization methods, such as the use of temporary storage, loop interchanging, loop distribution, vector reduction, and node splitting.

Code generation and scheduling make use of directed acyclic graphs of operations within basic blocks, and should utilize a register allocation strategy which does not inhibit parallel execution of instructions. Trace-scheduling compilation makes use of program traces obtained from multiple previous executions of the same program.

Loop transformations may in general be required prior to parallelization and/or vectorization of program code. Permutation, reversal, skewing, and transformation matrices are some of the specific techniques which can be applied. Wavefronting can be useful in exploiting fine-grain parallelism, while tiling can help achieve locality and reduce synchronization costs in coarse-grain computations. Software pipelining of loop iterations is another possible technique to parallelize a sequential program.

Exercises

Problem 10.1 Explain the following terms associated with message-passing programming of multicomputers:

(a) Synchronous versus asynchronous message-passing schemes.

(b) Blocking versus nonblocking communications.

(c) The *rendezvous* concept introduced in the Ada programming system.

(d) Name-addressing versus channel-addressing schemes for message passing.

(e) Uncoupling between sender and receiver using buffers or mailboxes.

(f) Lost-message handling and interrupt-message handling.

Problem 10.2 Concurrent object-oriented programming was introduced in Section 10.1.4. Chain multiplication of a list of numbers was illustrated in Example 10.2 based on a *divide-and-conquer* strategy. A fragment of a Lisp-like code for multiplying the sequence of numbers is given below:

```
(define tree-product
    (lambda [tree]
        (if (number ? tree)
            tree
            (*(tree-product(left-tree) tree))
            (*(tree-product(right-tree) tree)))))
```

In this code, a *tree* is passed to *tree-product*, which tests to see if the *tree* is a number (i.e. a singleton at the leaf node). If so, it returns the *tree*; otherwise it subdivides the problem into two recursive calls. The *left-tree* and *right-tree* are functions which pick off the left and right branches of the tree. Note that the argument to * may be evaluated concurrently.

Write a Lisp code to implement this divide-and-conquer algorithm for chain multiplication on a multiprocessor or a multicomputer system. Compare the execution time by running the same program sequentially on a uniprocessor system. The chain should be sufficiently long to see the difference.

Problem 10.3 *Gaussian elimination* with partial pivoting was implemented by [Quinn90] and Hatcher in C* code on the Connection Machine, as well as in concurrent C code on an nCUBE 3200 multicomputer.

(a) Discuss the translation/compiler effort from C* to C on the two machines after a careful reading of the paper by Quinn and Hatcher.

(b) Comment on SPMD (single program and multiple data streams) programming style as opposed to SIMD programming style, in terms of synchronization implementation and related performance issues.

(c) Repeat the program conversion experiments for a *fast Fourier transform* (FFT) algorithm. Perform the program conversion manually at the algorithm level using pseudo-codes with parallel constructs.

Problem 10.4 Explain the following terms related to shared-variable programming on multiprocessors.

(a) Multiprogramming.

(b) Multiprocessing in MIMD mode.

(c) Multiprocessing in MPMD mode.

(d) Multitasking.

(e) Multithreading.

(f) Program partitioning.

Problem 10.5 The following arrays are declared in Fortran 90.

```
REAL A(10, 10, 5)
REAL B(9, 9)
REAL C(3, 4, 5)
```

(a) List array elements specified by the following array expressions: A(5, 8:*, *), B(3:*:3, 5:8), and C(*, 3, 4).

(b) Can you make the following array assignments?

A(3:5, 7, 4:6) = C(*, 3, 3:5),
B(1:2, 7:9) = B(7:9,4:6),
C(*, 4, 4:5) = B(7:9, 8:9), and
A(5, 9:10, 2:4) = A(7, 3:4, 3:*) + C(2, 4:5, 1:3).

Problem 10.6 Determine the dependence relations among the three statements in the following loop nest. The direction vector and distance vector should be specified in all dependence relations.

```
         Do I = 1, N
             Do J = 2, N
S1:              A(I, J) = A(I, J-1) + B(I, J)
S2:              C(I, J) = A(I, J) + D(I+1, J)
S3:              D(I, J) = 0.1
             Enddo
         Enddo
```

Problem 10.7 Consider the following loop nest:

```
         Do I = 1, N
S1:          A(I) = B(I)
S2:          C(I) = A(I) + B(I)
S3:          E(I) = C(I+1)
         Enddo
```

(a) Determine the dependence relations among the three statements.

(b) Show how to vectorize the code with Fortran 90 statements.

Problem 10.8 Consider the following loop nest:

```
         Do I = 1, N
             Do J = 2, N
S1:              A(I, J) = B(I, J) + C(I, J)
S2:              C(I, J) = D(I, J)/2
S3:              E(I, J) = A(I, J-1)**2 + E(I, J-1)
             Enddo
         Enddo
```

(a) Show the data dependences among the statements.

(b) Show how to parallelize the loop, scheduling the parallelizable iterations to concurrent processors.

Problem 10.9 Consider the following loop nest:

```
         Do J = 1, N
             Do I = 1, N
S1:              A(I, J+1) = B(I, J) + C(I, J)
S2:              D(I, J) = A(I, J)/2
             Enddo
         Enddo
```

(a) Show how to compile the code for vectorization in the I-loop, assuming Fortran column-major storage order.

(b) Show how to compile the code for parallelization in the J-loop using the **Doacross** and **Endacross** commands. You can use a conditional statement or *Signal*(J) and *Wait*(J–1) for synchronization in the concurrent loop.

(c) Show how to compile the loop to perform the J-loop in vector mode, while using the **Doall** and **Endall** commands for the outer I-loop.

Problem 10.10 Explain the following loop transformations and discuss how to apply them for loop vectorization or parallelization:

(a) Loop permutation.

(b) Loop reversal.

(c) Loop skewing.

(d) Loop tiling.

(e) Wavefront transformation.

(f) Locality optimization.

(g) Software pipelining.

Problem 10.11 Loop-carried dependence (LCD) exists in the following loops:

(a) Consider the forward LCD in the following loop:

```
Do I = 1, N
    A(I) = A(I+1) + 3.14159
Enddo
```

Explain why a forward LCD does not prevent vectorization of a loop.

(b) The following loop contains backward LCDs:

```
Do I = 1, N – 1
      A(I) = B(I) + C(I)
      B(I+1) = D(I) * 3.14159
Enddo
```

Show that the loop can be vectorized by statement reordering.

Problem 10.12 Vectorize or parallelize the following loops if possible. Otherwise, explain why it is not possible.

(a)

```
Do I = 1, N
      A(I+1) = A(I) + 3.14159
Enddo
```

(b)

```
Do I = 1, N
      If (A (I) .LE. 0.0) then
            S = S + B(I) * C(I)
            X = B(I)
      Endif
Enddo
```

Problem 10.13 Tanenbaum and associates have suggested a hybrid parallel programming paradigm using shared objects and broadcasting. Study the paper that appeared in *IEEE Computer* (August 1992) and explain how to apply the software paradigm for either multiprocessors or multicomputers.