

Chapter 11

Optimizing for Parallelism and Locality

This chapter shows how a compiler can enhance parallelism and locality in computationally intensive programs involving arrays to speed up target programs running on multiprocessor systems. Many scientific, engineering, and commercial applications have an insatiable need for computational cycles. Examples include weather prediction, protein-folding for designing drugs, fluid-dynamics for designing aeropropulsion systems, and quantum chromodynamics for studying the strong interactions in high-energy physics.

One way to speed up a computation is to use parallelism. Unfortunately, it is not easy to develop software that can take advantage of parallel machines. Dividing the computation into units that can execute on different processors in parallel is already hard enough; yet that by itself does not guarantee a speedup. We must also minimize interprocessor communication, because communication overhead can easily make the parallel code run even slower than the sequential execution!

Minimizing communication can be thought of as a special case of improving a program's *data locality*. In general, we say that a program has good data locality if a processor often accesses the same data it has used recently. Surely if a processor on a parallel machine has good locality, it does not need to communicate with other processors frequently. Thus, parallelism and data locality need to be considered hand-in-hand. Data locality, by itself, is also important for the performance of individual processors. Modern processors have one or more level of caches in the memory hierarchy; a memory access can take tens of machine cycles whereas a cache hit would only take a few cycles. If a program does not have good data locality and misses in the cache often, its performance will suffer.

Another reason why parallelism and locality are treated together in this same chapter is that they share the same theory. If we know how to optimize for data locality, we know where the parallelism is. You will see in this chapter that the

program model we used for data-flow analysis in Chapter 9 is inadequate for parallelization and locality optimization. The reason is that work on data-flow analysis assumes we don't distinguish among the ways a given statement is reached, and in fact these Chapter 9 techniques take advantage of the fact that we don't distinguish among different executions of the same statement, e.g., in a loop. To parallelize a code, we need to reason about the dependences among different dynamic executions of the same statement to determine if they can be executed on different processors simultaneously.

This chapter focuses on techniques for optimizing the class of numerical applications that use arrays as data structures and access them with simple regular patterns. More specifically, we study programs that have *affine* array accesses with respect to surrounding loop indexes. For example, if i and j are the index variables of surrounding loops, then $Z[i][j]$ and $Z[i][i + j]$ are affine accesses. A function of one or more variables, x_1, x_2, \dots, x_n is *affine* if it can be expressed as a sum of a constant, plus constant multiples of the variables, i.e., $c_0 + c_1x_1 + c_2x_2 + \dots + c_nx_n$, where c_0, c_1, \dots, c_n are constants. Affine functions are usually known as linear functions, although strictly speaking, linear functions do not have the c_0 term.

Here is a simple example of a loop in this domain:

```
for (i = 0; i < 10; i++) {
    Z[i] = 0;
}
```

Because iterations of the loop write to different locations, different processors can execute different iterations concurrently. On the other hand, if there is another statement $Z[j] = 1$ being executed, we need to worry about whether i could ever be the same as j , and if so, in which order do we execute those instances of the two statements that share a common value of the array index.

Knowing which iterations can refer to the same memory location is important. This knowledge lets us specify the data dependences that must be honored when scheduling code for both uniprocessors and multiprocessors. Our objective is to find a schedule that honors all the data dependences such that operations that access the same location and cache lines are performed close together if possible, and on the same processor in the case of multiprocessors.

The theory we present in this chapter is grounded in linear algebra and integer programming techniques. We model iterations in an n -deep loop nest as an n -dimensional polyhedron, whose boundaries are specified by the bounds of the loops in the code. Affine functions map each iteration to the array locations it accesses. We can use integer linear programming to determine if there exist two iterations that can refer to the same location.

The set of code transformations we discuss here fall into two categories: *affine partitioning* and *blocking*. Affine partitioning splits up the polyhedra of iterations into components, to be executed either on different machines or one-by-one sequentially. On the other hand, blocking creates a hierarchy of iterations. Suppose we are given a loop that sweeps through an array row-by-

row. We may instead subdivide the array into blocks and visit all elements in a block before moving to the next. The resulting code will consist of outer loops traversing the blocks, and then inner loops to sweep the elements within each block. Linear algebra techniques are used to determine both the best affine partitions and the best blocking schemes.

In the following, we first start with an overview of the concepts in parallel computation and locality optimization in Section 11.1. Then, Section 11.2 is an extended concrete example — matrix multiplication — that shows how *loop transformations* that reorder the computation inside a loop can improve both locality and the effectiveness of parallelization.

Sections 11.3 to Sections 11.6 present the preliminary information necessary for loop transformations. Section 11.3 shows how we model the individual iterations in a loop nest; Section 11.4 shows how we model array index functions that map each loop iteration to the array locations accessed by the iteration; Section 11.5 shows how to determine which iterations in a loop refer to the same array location or the same cache line using standard linear algebra algorithms; and Section 11.6 shows how to find all the data dependences among array references in a program.

The rest of the chapter applies these preliminaries in coming up with the optimizations. Section 11.7 first looks at the simpler problem of finding parallelism that requires no synchronization. To find the best affine partitioning, we simply find the solution to the constraint that operations that share a data dependence must be assigned to the same processor.

Well, not too many programs can be parallelized without requiring any synchronization. Thus, in Sections 11.8 through 11.9.9, we consider the general case of finding parallelism that requires synchronization. We introduce the concept of pipelining, show how to find the affine partitioning that maximizes the degree of pipelining allowed by a program. We show how to optimize for locality in Section 11.10. Finally, we discuss how affine transforms are useful for optimizing for other forms of parallelism.

11.1 Basic Concepts

This section introduces the basic concepts related to parallelization and locality optimization. If operations can be executed in parallel, they also can be reordered for other goals such as locality. Conversely, if data dependences in a program dictate that instructions in a program must execute serially, there is obviously no parallelism, nor is there any opportunity to reorder instructions to improve locality. Thus parallelization analysis also finds the available opportunities for code motion to improve data locality.

To minimize communication in parallel code, we group together all related operations and assign them to the same processor. The resulting code must therefore have data locality. One crude approach to getting good data locality on a uniprocessor is to have the processor execute the code assigned to each

processor in succession.

In this introduction, we start by presenting an overview of parallel computer architectures. We then show the basic concepts in parallelization, the kind of transformations that can make a big difference, as well as the concepts useful for parallelization. We then discuss how similar considerations can be used to optimize locality. Finally, we introduce informally the mathematical concepts used in this chapter.

11.1.1 Multiprocessors

The most popular parallel machine architecture is the symmetric multiprocessor (SMP). High-performance personal computers often have two processors, and many server machines have four, eight, and some even tens of processors. Moreover, as it has become feasible for several high-performance processors to fit on a single chip, multiprocessors have become even more widely used.

Processors on a symmetric multiprocessor share the same address space. To communicate, a processor can simply write to a memory location, which is then read by any other processor. Symmetric multiprocessors are so named because all processors can access all of the memory in the system with a uniform access time. Fig. 11.1 shows the high-level architecture of a multiprocessor. The processors may have their own first-level, second-level, and in some cases, even a third-level cache. The highest-level caches are connected to physical memory through typically a shared bus.

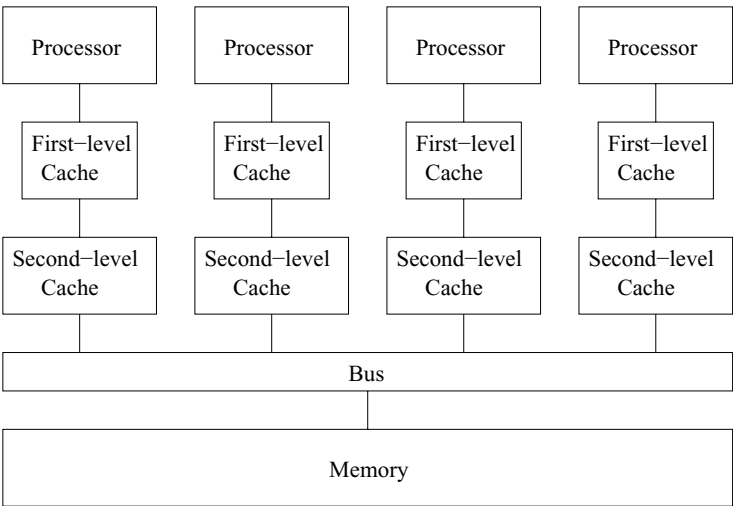


Figure 11.1: The symmetric multi-processor architecture

Symmetric multiprocessors use a *coherent cache protocol* to hide the presence of caches from the programmer. Under such a protocol, several processors are

allowed to keep copies of the same cache line¹ at the same time, provided that they are only reading the data. When a processor wishes to write to a cache line, copies from all other caches are removed. When a processor requests data not found in its cache, the request goes out on the shared bus, and the data will be fetched either from memory or from the cache of another processor.

The time taken for one processor to communicate with another is about twice the cost of a memory access. The data, in units of cache lines, must first be written from the first processor's cache to memory, and then fetched from the memory to the cache of the second processor. You may think that interprocessor communication is relatively cheap, since it is only about twice as slow as a memory access. However, you must remember that memory accesses are very expensive when compared to cache hits—they can be a hundred times slower. This analysis brings home the similarity between efficient parallelization and locality analysis. For a processor to perform well, either on its own or in the context of a multiprocessor, it must find most of the data it operates on in its cache.

In the early 2000's, the design of symmetric multiprocessors no longer scaled beyond tens of processors, because the shared bus, or any other kind of interconnect for that matter, could not operate at speed with the increasing number of processors. To make processor designs scalable, architects introduced yet another level in the memory hierarchy. Instead of having memory that is equally far away for each processor, they distributed the memories so that each processor could access its local memory quickly as shown in Fig. 11.2. Remote memories thus constituted the next level of the memory hierarchy; they are collectively bigger but also take longer to access. Analogous to the principle in memory-hierarchy design that fast stores are necessarily small, machines that support fast interprocessor communication necessarily have a small number of processors.

There are two variants of a parallel machine with distributed memories: NUMA (nonuniform memory access) machines and message-passing machines. NUMA architectures provide a shared address space to the software, allowing processors to communicate by reading and writing shared memory. On message-passing machines, however, processors have disjoint address spaces, and processors communicate by sending messages to each other. Note that even though it is simpler to write code for shared memory machines, the software must have good locality for either type of machine to perform well.

11.1.2 Parallelism in Applications

We use two high-level metrics to estimate how well a parallel application will perform: *parallelism coverage* which is the percentage of the computation that runs in parallel, *granularity of parallelism*, which is the amount of computation that each processor can execute without synchronizing or communicating with others. One particularly attractive target of parallelization is loops: a loop may

¹You may wish to review the discussion of caches and cache lines in Section 7.4.

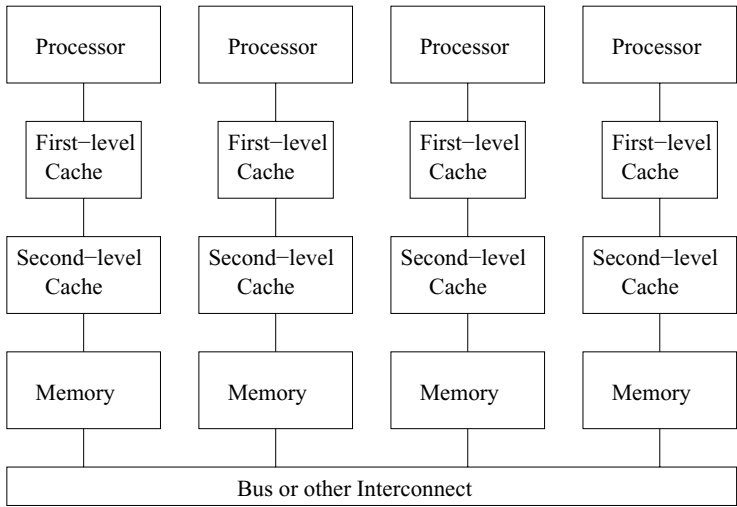


Figure 11.2: Distributed memory machines

have many iterations, and if they are independent of each other, we have found a great source of parallelism.

Amdahl's Law

The significance of parallelism coverage is succinctly captured by Amdahl's Law. *Amdahl's Law* states that, if f is the fraction of the code parallelized, and if the parallelized version runs on a p -processor machine with no communication or parallelization overhead, the speedup is

$$\frac{1}{(1 - f) + (f/p)}.$$

For example, if half of the computation remains sequential, the computation can only double in speed, regardless of how many processors we use. The speedup achievable is a factor of 1.6 if we have 4 processors. Even if the parallelism coverage is 90%, we get at most a factor of 3 speed up on 4 processors, and a factor of 10 on an unlimited number of processors.

Granularity of Parallelism

It is ideal if the entire computation of an application can be partitioned into many independent coarse-grain tasks because we can simply assign the different tasks to different processors. One such example is the SETI (Search for Extra-Terrestrial Intelligence) project, which is an experiment that uses home computers connected over the Internet to analyze different portions of radio telescope data in parallel. Each unit of work, requiring only a small amount

of input and generating a small amount of output, can be performed independently of all others. As a result, such a computation runs well on machines over the Internet, which has relatively high communication latency (delay) and low bandwidth.

Most applications require more communication and interaction between processors, yet still allow coarse-grained parallelism. Consider, for example, the web server responsible for serving a large number of mostly independent requests out of a common database. We can run the application on a multiprocessor, with a thread implementing the database and a number of other threads servicing user requests. Other examples include drug design or airfoil simulation, where the results of many different parameters can be evaluated independently. Sometimes the evaluation of even just one set of parameters in a simulation takes so long that it is desirable to speed it up with parallelization. As the granularity of available parallelism in an application decreases, better interprocessor communication support and more programming effort are needed.

Many long-running scientific and engineering applications, with their simple control structures and large data sets, can be more readily parallelized at a finer grain than the applications mentioned above. Thus, this chapter is devoted primarily to techniques that apply to numerical applications, and in particular, to programs that spend most of their time manipulating data in multidimensional arrays. We shall examine this class of programs next.

11.1.3 Loop-Level Parallelism

Loops are the main target for parallelization, especially in applications using arrays. Long running applications tend to have large arrays, which lead to loops that have many iterations, one for each element in the array. It is not uncommon to find loops whose iterations are independent of one another. We can divide the large number of iterations of such loops among the processors. If the amount of work performed in each iteration is roughly the same, simply dividing the iterations evenly across processors will achieve maximum parallelism. Example 11.1 is an extremely simple example showing how we can take advantage of loop-level parallelism.

Example 11.1: The loop

```
for (i = 0; i < n; i++) {  
    Z[i] = X[i] - Y[i];  
    Z[i] = Z[i] * Z[i];  
}
```

computes the square of differences between elements in vectors X and Y and stores it into Z . The loop is parallelizable because each iteration accesses a different set of data. We can execute the loop on a computer with M processors by giving each processor a unique ID $p = 0, 1, \dots, M - 1$ and having each processor execute the same code:

Task-Level Parallelism

It is possible to find parallelism outside of iterations in a loop. For example, we can assign two different function invocations, or two independent loops, to two processors. This form of parallelism is known as *task parallelism*. The task level is not as attractive a source of parallelism as is the loop level. The reason is that the number of independent tasks is a constant for each program and does not scale with the size of the data, as does the number of iterations of a typical loop. Moreover, the tasks generally are not of equal size, so it is hard to keep all the processors busy all the time.

```
b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

We divide the iterations in the loop evenly among the processors; the p th processor is given the p th swath of iterations to execute. Note that the number of iterations may not be divisible by M , so we assure that the last processor does not execute past the bound of the original loop by introducing a minimum operation. \square

The parallel code shown in Example 11.1 is an SPMD (Single Program Multiple Data) program. The same code is executed by all processors, but it is parameterized by an identifier unique to each processor, so different processors can take different actions. Typically one processor, known as the *master*, executes all the serial part of the computation. The master processor, upon reaching a parallelized section of the code, wakes up all the *slave* processors. All the processors execute the parallelized regions of the code. At the end of each parallelized region of code, all the processors participate in a *barrier synchronization*. Any operation executed before a processor enters a synchronization barrier is guaranteed to be completed before any other processors are allowed to leave the barrier and execute operations that come after the barrier.

If we parallelize only little loops like those in Example 11.1, then the resulting code is likely to have low parallelism coverage and relatively fine-grain parallelism. We prefer to parallelize the outermost loops in a program, as that yields the coarsest granularity of parallelism. Consider, for example, the application of a two-dimensional FFT transformation that operates on an $n \times n$ data set. Such a program performs n FFT's on the rows of the data, then another n FFT's on the columns. It is preferable to assign each of the n independent FFT's to one processor each, rather than trying to use several processors to collaborate on one FFT. The code is easier to write, the parallelism coverage

for the algorithm is 100%, and the code has good data locality as it requires no communication at all while computing an FFT.

Many applications do not have large outermost loops that are parallelizable. The execution time of these applications, however, is often dominated by time-consuming *kernels*, which may have hundreds of lines of code consisting of loops with different nesting levels. It is sometimes possible to take the kernel, reorganize its computation and partition it into mostly independent units by focusing on its locality.

11.1.4 Data Locality

There are two somewhat different notions of data locality that need to be considered when parallelizing programs. *Temporal* locality occurs when the same data is used several times within a short time period. *Spatial* locality occurs when different data elements that are located near to each other are used within a short period of time. An important form of spatial locality occurs when all the elements that appear on one cache line are used together. The reason is that as soon as one element from a cache line is needed, all the elements in the same line are brought to the cache and will probably still be there if they are used soon. The effect of this spatial locality is that cache misses are minimized, with a resulting important speedup of the program.

Kernels can often be written in many semantically equivalent ways but with widely varying data localities and performances. Example 11.2 shows an alternative way of expressing the computation in Example 11.1.

Example 11.2: Like Example 11.1 the following also finds the squares of differences between elements in vectors X and Y .

```
for (i = 0; i < n; i++)
    Z[i] = X[i] - Y[i];
for (i = 0; i < n; i++)
    Z[i] = Z[i] * Z[i];
```

The first loop finds the differences, the second finds the squares. Code like this appears often in real programs, because that is how we can optimize a program for *vector machines*, which are supercomputers which have instructions that perform simple arithmetic operations on vectors at a time. We see that the bodies of the two loops here are *fused* as one in Example 11.1.

Given that the two programs perform the same computation, which performs better? The fused loop in Example 11.1 has better performance because it has better data locality. Each difference is squared immediately, as soon as it is produced; in fact, we can hold the difference in a register, square it, and write the result just once into the memory location $Z[i]$. In contrast, the code in this example writes $Z[i]$ long before it uses that value. If the size of the array is larger than the cache, $Z[i]$ needs to be refetched from memory the second time it is used in this example. Thus, this code can run significantly slower. \square

```

for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        Z[i,j] = 0;

```

(a) Zeroing an array column-by-column.

```

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;

```

(b) Zeroing an array row-by-row.

```

b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;

```

(c) Zeroing an array row-by-row in parallel.

Figure 11.3: Sequential and parallel code for zeroing an array

Example 11.3: Suppose we want to set array Z , stored in row-major order (recall Section 6.4.3), to all zeros. Fig. 11.3(a) and (b) sweeps through the array column-by-column and row-by-row, respectively. We can transpose the loops in Fig. 11.3(a) to arrive at Fig. 11.3(b). In terms of spatial locality, it is preferable to zero out the array row-by-row since all the words in a cache line are zeroed consecutively. In the column-by-column approach, even though each cache line is reused by consecutive iterations of the outer loop, cache lines will be thrown out before reuse if the size of a column is greater than the size of the cache. For best performance, we parallelize the outer loop of Fig. 11.3(b) in a manner similar to that used in Example 11.1 [see Fig. 11.3(c)]. \square

The two examples above illustrate several important characteristics associated with numeric applications operating on arrays:

- Array code often has many parallelizable loops.
- When loops have parallelism, their iterations can be executed in arbitrary order; they can be reordered to improve data locality drastically.
- As we create large units of parallel computation that are independent of each other, executing these serially tends to produce good data locality.

11.1.5 Introduction to Affine Transform Theory

Writing correct and efficient sequential programs is difficult; writing parallel programs that are correct and efficient is even harder. The level of difficulty

increases as the granularity of parallelism exploited decreases. As we see above, programmers must pay attention to data locality to get high performance. Furthermore, the task of taking an existing sequential program and parallelizing it is extremely hard. It is hard to catch all the dependences in the program, especially if it is not a program with which we are familiar. Debugging a parallel program is harder yet, because errors can be nondeterministic.

Ideally, a parallelizing compiler automatically translates ordinary sequential programs into efficient parallel programs and optimizes the locality of these programs. Unfortunately, compilers without high-level knowledge about the application, can only preserve the semantics of the original algorithm, which may not be amenable to parallelization. Furthermore, programmers may have made arbitrary choices that limit the program's parallelism.

Successes in parallelization and locality optimizations have been demonstrated for Fortran numeric applications that operate on arrays with affine accesses. Without pointers and pointer arithmetic, Fortran is easier to analyze. Note that not all applications have affine accesses; most notably, many numeric applications operate on sparse matrices whose elements are accessed indirectly through another array. This chapter focuses on the parallelization and optimizations of kernels, consisting of mostly tens of lines.

As illustrated by Examples 11.2 and 11.3, parallelization and locality optimization require that we reason about the different instances of a loop and their relations with each other. This situation is very different from data-flow analysis, where we combine information associated with all instances together.

For the problem of optimizing loops with array accesses, we use three kinds of spaces. Each space can be thought of as points on a grid of one or more dimensions.

1. The *iteration space* is the set of the dynamic execution instances in a computation, that is, the set of combinations of values taken on by the loop indexes.
2. The *data space* is the set of array elements accessed.
3. The *processor space* is the set of processors in the system. Normally, these processors are assigned integer numbers or vectors of integers to distinguish among them.

Given as input are a sequential order in which the iterations are executed and affine array-access functions (e.g., $X[i, j + 1]$) that specify which instances in the iteration space access which elements in the data space.

The output of the optimization, again represented as affine functions, defines what each processor does and when. To specify what each processor does, we use an affine function to assign instances in the original iteration space to processors. To specify when, we use an affine function to map instances in the iteration space to a new ordering. The schedule is derived by analyzing the array-access functions for data dependences and reuse patterns.

The following example will illustrate the three spaces — iteration, data, and processor. It will also introduce informally the important concepts and issues that need to be addressed in using these spaces to parallelize code. The concepts each will be covered in detail in later sections.

Example 11.4: Figure 11.4 illustrates the different spaces and their relations used in the following program:

```
float Z[100];
for (i = 0; i < 10; i++)
    Z[i+10] = Z[i];
```

The three spaces and the mappings among them are as follows:

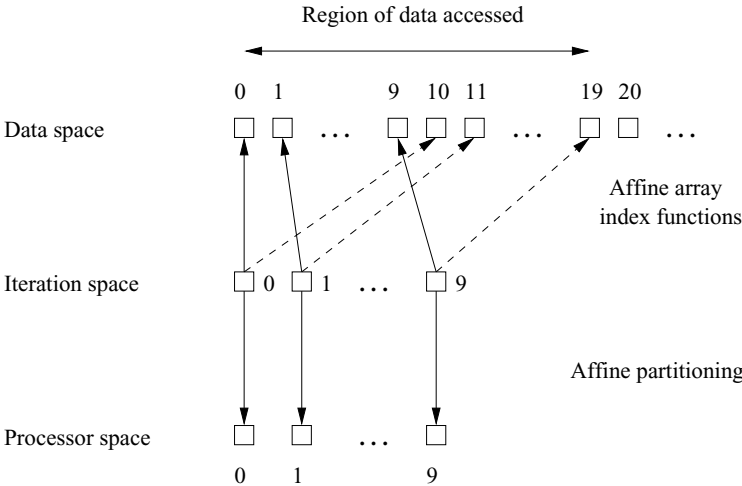


Figure 11.4: Iteration, data, and processor space for Example 11.4

1. *Iteration Space:* The iteration space is the set of iterations, whose ID's are given by the values held by the loop index variables. A d -deep *loop nest* (i.e., d nested loops) has d index variables, and is thus modeled by a d -dimensional space. The space of iterations is bounded by the lower and upper bounds of the loop indexes. The loop of this example defines a one-dimensional space of 10 iterations, labeled by the loop index values: $i = 0, 1, \dots, 9$.
2. *Data Space:* The data space is given directly by the array declarations. In this example, elements in the array are indexed by $a = 0, 1, \dots, 99$. Even though all arrays are linearized in a program's address space, we treat n -dimensional arrays as n -dimensional spaces, and assume that the individual indexes stay within their bounds. In this example, the array is one-dimensional anyway.

3. *Processor Space*: We pretend that there are an unbounded number of virtual processors in the system as our initial parallelization target. The processors are organized in a multidimensional space, one dimension for each loop in the nest we wish to parallelize. After parallelization, if we have fewer physical processors than virtual processors, we divide the virtual processors into even blocks, and assign a block each to a processor. In this example, we need only ten processors, one for each iteration of the loop. We assume in Fig. 11.4 that processors are organized in a one-dimensional space and numbered $0, 1, \dots, 9$, with loop iteration i assigned to processor i . If there were, say, only five processors, we could assign iterations 0 and 1 to processor 0, iterations 2 and 3 to processor 1, and so on. Since iterations are independent, it doesn't matter how we do the assignment, as long as each of the five processors gets two iterations.
4. *Affine Array-Index Function*: Each array access in the code specifies a mapping from an iteration in the iteration space to an array element in the data space. The access function is affine if it involves multiplying the loop index variables by constants and adding constants. Both the array index functions $i + 10$, and i are affine. From the access function, we can tell the *dimension* of the data accessed. In this case, since each index function has one loop variable, the space of accessed array elements is one dimensional.
5. *Affine Partitioning*: We parallelize a loop by using an affine function to assign iterations in an iteration space to processors in the processor space. In our example, we simply assign iteration i to processor i . We can also specify a new execution order with affine functions. If we wish to execute the loop above sequentially, but in reverse, we can specify the ordering function succinctly with an affine expression $10 - i$. Thus, iteration 9 is the 1st iteration to execute and so on.
6. *Region of Data Accessed*: To find the best affine partitioning, it useful to know the region of data accessed by an iteration. We can get the region of data accessed by combining the iteration space information with the array index function. In this case, the array access $Z[i + 10]$ touches the region $\{a \mid 10 \leq a < 20\}$ and the access $Z[i]$ touches the region $\{a \mid 0 \leq a < 10\}$.
7. *Data Dependence*: To determine if the loop is parallelizable, we ask if there is a data dependence that crosses the boundary of each iteration. For this example, we first consider the dependences of the write accesses in the loop. Since the access function $Z[i + 10]$ maps different iterations to different array locations, there are no dependences regarding the order in which the various iterations write values to the array. Is there a dependence between the read and write accesses? Since only $Z[10], Z[11], \dots, Z[19]$ are written (by the access $Z[i + 10]$), and only $Z[0], Z[1], \dots, Z[9]$ are read (by the access $Z[i]$), there can be no dependencies regarding the relative order of a read and a write. Therefore, this loop is parallelizable. That

is, each iteration of the loop is independent of all other iterations, and we can execute the iterations in parallel, or in any order we choose. Notice, however, that if we made a small change, say by increasing the upper limit on loop index i to 10 or more, then there would be dependencies, as some elements of array Z would be written on one iteration and then read 10 iterations later. In that case, the loop could not be parallelized completely, and we would have to think carefully about how iterations were partitioned among processors and how we ordered iterations.

□

Formulating the problem in terms of multidimensional spaces and affine mappings between these spaces lets us use standard mathematical techniques to solve the parallelization and locality optimization problem generally. For example, the region of data accessed can be found by the elimination of variables using the Fourier-Motzkin elimination algorithm. Data dependence is shown to be equivalent to the problem of integer linear programming. Finally, finding the affine partitioning corresponds to solving a set of linear constraints. Don't worry if you are not familiar with these concepts, as they will be explained starting in Section 11.3.

11.2 Matrix Multiply: An In-Depth Example

We shall introduce many of the techniques used by parallel compilers in an extended example. In this section, we explore the familiar matrix-multiplication algorithm to show that it is nontrivial to optimize even a simple and easily parallelizable program. We shall see how rewriting the code can improve data locality; that is, processors are able to do their work with far less communication (with global memory or with other processors, depending on the architecture) than if the straightforward program is chosen. We shall also discuss how cognizance of the existence of cache lines that hold several consecutive data elements can improve the running time of programs such as matrix multiplication.

11.2.1 The Matrix-Multiplication Algorithm

In Fig. 11.5 we see a typical matrix-multiplication program.² It takes two $n \times n$ matrices, X and Y , and produces their product in a third $n \times n$ matrix Z . Recall that Z_{ij} — the element of matrix Z in row i and column j — must become $\sum_{k=1}^n X_{ik}Y_{kj}$.

The code of Fig. 11.5 generates n^2 results, each of which is an inner product between one row and one column of the two matrix operands. Clearly, the

²In programs of this chapter, we shall generally use C syntax, but to make multidimensional array accesses — the central issue for most of the chapter — easier to read, we shall use Fortran-style array references, that is, $Z[i,j]$ instead of $Z[i][j]$.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    Z[i,j] = 0.0;
    for (k = 0; k < n; k++)
      Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
  }
```

Figure 11.5: The basic matrix-multiplication algorithm

calculations of each of the elements of Z are independent and can be executed in parallel.

The larger n is, the more times the algorithm touches each element. That is, there are $3n^2$ locations among the three matrices, but the algorithm performs n^3 operations, each of which multiplies an element of X by an element of Y and adds the product to an element of Z . Thus, the algorithm is computation-intensive and memory accesses should not, in principle, constitute a bottleneck.

Serial Execution of the Matrix Multiplication

Let us first consider how this program behaves when run sequentially on a uniprocessor. The innermost loop reads and writes the same element of Z , and uses a row of X and a column of Y . $Z[i,j]$ can easily be stored in a register and requires no memory accesses. Assume, without loss of generality, that the matrix is laid out in row-major order, and that c is the number of array elements in a cache line.

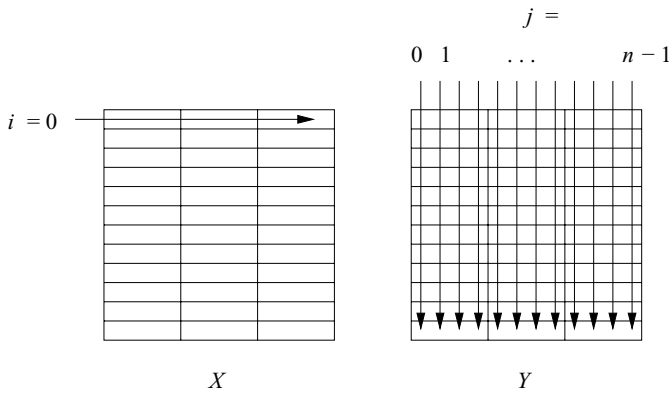


Figure 11.6: The data access pattern in matrix multiply

Figure 11.6 suggests the access pattern as we execute one iteration of the outer loop of Fig. 11.5. In particular, the picture shows the first iteration, with $i = 0$. Each time we move from one element of the first row of X to the next,

we visit each element in a single column of Y . We see in Fig. 11.6 the assumed organization of the matrices into cache lines. That is, each small rectangle represents a cache line holding four array elements (i.e., $c = 4$ and $n = 12$ in the picture).

Accessing X puts little burden on the cache. One row of X is spread among only n/c cache lines. Assuming these all fit in the cache, only n/c cache misses occur for a fixed value of index i , and the total number of misses for all of X is n^2/c , the minimum possible (we assume n is divisible by c , for convenience).

However, while using one row of X , the matrix-multiplication algorithm accesses all the elements of Y , column by column. That is, when $j = 0$, the inner loop brings to the cache the entire first column of Y . Notice that the elements of that column are stored among n different cache lines. If the cache is big enough (or n small enough) to hold n cache lines, and no other uses of the cache force some of these cache lines to be expelled, then the column for $j = 0$ will still be in the cache when we need the second column of Y . In that case, there will not be another n cache misses reading Y , until $j = c$, at which time we need to bring into the cache an entirely different set of cache lines for Y . Thus, to complete the first iteration of the outer loop (with $i = 0$) requires between n^2/c and n^2 cache misses, depending on whether columns of cache lines can survive from one iteration of the second loop to the next.

Moreover, as we complete the outer loop, for $i = 1, 2$, and so on, we may have many additional cache misses as we read Y , or none at all. If the cache is big enough that all n^2/c cache lines holding Y can reside together in the cache, then we need no more cache misses. The total number of cache misses is thus $2n^2/c$, half for X and half for Y . However, if the cache can hold one column of Y but not all of Y , then we need to bring all of Y into cache again, each time we perform an iteration of the outer loop. That is, the number of cache misses is $n^2/c + n^3/c$; the first term is for X and the second is for Y . Worst, if we cannot even hold one column of Y in the cache, then we have n^2 cache misses per iteration of the outer loop and a total of $n^2/c + n^3$ cache misses.

Row-by-Row Parallelization

Now, let us consider how we could use some number of processors, say p processors, to speed up the execution of Fig. 11.5. An obvious approach to parallelizing matrix multiplication is to assign different rows of Z to different processors. A processor is responsible for n/p consecutive rows (we assume n is divisible by p , for convenience). With this division of labor, each processor needs to access n/p rows of matrices X and Z , but the entire Y matrix. One processor will compute n^2/p elements of Z , performing n^3/p multiply-and-add operations to do so.

While the computation time thus decreases in proportion to p , the communication cost actually rises in proportion to p . That is, each of p processors has to read n^2/p elements of X , but all n^2 elements of Y . The total number of cache lines that must be delivered to the caches of the p processors is at least

$n^2/c + pn^2/c$; the two terms are for delivering X and copies of Y , respectively. As p approaches n , the computation time becomes $O(n^2)$ while the communication cost is $O(n^3)$. That is, the bus on which data is moved between memory and the processors' caches becomes the bottleneck. Thus, with the proposed data layout, using a large number of processors to share the computation can actually slow down the computation, rather than speed it up.

11.2.2 Optimizations

The matrix-multiplication algorithm of Fig. 11.5 shows that even though an algorithm may *reuse* the same data, it may have poor data locality. A reuse of data results in a cache hit only if the reuse happens soon enough, before the data is displaced from the cache. In this case, n^2 multiply-add operations separate the reuse of the same data element in matrix Y , so locality is poor. In fact, n operations separate the reuse of the same cache line in Y . In addition, on a multiprocessor, reuse may result in a cache hit only if the data is reused by the same processor. When we considered a parallel implementation in Section 11.2.1, we saw that elements of Y had to be used by every processor. Thus, the reuse of Y is not turned into locality.

Changing Data Layout

One way to improve the locality of a program is to change the layout of its data structures. For example, storing Y in column-major order would have improved the reuse of cache lines for matrix Y . The applicability of this approach is limited, because the same matrix normally is used in different operations. If Y played the role of X in another matrix multiplication, then it would suffer from being stored in column-major order, since the first matrix in a multiplication is better stored in row-major order.

Blocking

It is sometimes possible to change the execution order of the instructions to improve data locality. The technique of interchanging loops, however, does not improve the matrix-multiplication routine. Suppose the routine were written to generate a column of matrix Z at a time, instead of a row at a time. That is, make the j -loop the outer loop and the i -loop the second loop. Assuming matrices are still stored in row-major order, matrix Y enjoys better spatial and temporal locality, but only at the expense of matrix X .

Blocking is another way of reordering iterations in a loop that can greatly improve the locality of a program. Instead of computing the result a row or a column at a time, we divide the matrix up into submatrices, or *blocks*, as suggested by Fig. 11.7, and we order operations so an entire block is used over a short period of time. Typically, the blocks are squares with a side of length B . If B evenly divides n , then all the blocks are square. If B does not evenly

divide n , then the blocks on the lower and right edges will have one or both sides of length less than B .

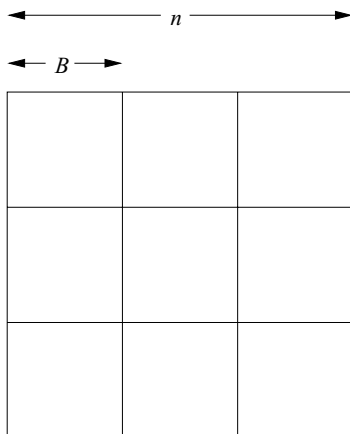


Figure 11.7: A matrix divided into blocks of side B

Figure 11.8 shows a version of the basic matrix-multiplication algorithm where all three matrices have been blocked into squares of side B . As in Fig. 11.5, Z is assumed to have been initialized to all 0's. We assume that B divides n ; if not, then we need to modify line (4) so the upper limit is $\min(ii + B, n)$, and similarly for lines (5) and (6).

```

1)  for (ii = 0; ii < n; ii = ii+B)
2)      for (jj = 0; jj < n; jj = jj+B)
3)          for (kk = 0; kk < n; kk = kk+B)
4)              for (i = ii; i < ii+B; i++)
5)                  for (j = jj; j < jj+B; j++)
6)                      for (k = kk; k < kk+B; k++)
7)                          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];

```

Figure 11.8: Matrix multiplication with blocking

The outer three loops, lines (1) through (3), use indexes ii , jj , and kk , which are always incremented by B , and therefore always mark the left or upper edge of some blocks. With fixed values of ii , jj , and kk , lines (4) through (7) enable the blocks with upper-left corners $X[ii, kk]$ and $Y[kk, jj]$ to make all possible contributions to the block with upper-left corner $Z[ii, jj]$.

If we pick B properly, we can significantly decrease the number of cache misses, compared with the basic algorithm, when all of X , Y , or Z cannot fit in the cache. Choose B such that it is possible to fit one block from each of the matrices in the cache. Because of the order of the loops, we actually need each

Another View of Block-Based Matrix Multiplication

We can imagine that the matrices X , Y , and Z of Fig. 11.8 are not $n \times n$ matrices of floating-point numbers, but rather $(n/B) \times (n/B)$ matrices whose elements are themselves $B \times B$ matrices of floating-point numbers. Lines (1) through (3) of Fig. 11.8 are then like the three loops of the basic algorithm in Fig. 11.5, but with n/B as the size of the matrices, rather than n . We can then think of lines (4) through (7) of Fig. 11.8 as implementing a single multiply-and-add operation of Fig. 11.5. Notice that in this operation, the single multiply step is a matrix-multiply step, and it uses the basic algorithm of Fig. 11.5 on the floating-point numbers that are elements of the two matrices involved. The matrix addition is element-wise addition of floating-point numbers.

block of Z in cache only once, so (as in the analysis of the basic algorithm in Section 11.2.1) we shall not count the cache misses due to Z .

To bring a block of X or Y to the cache takes B^2/c cache misses; recall c is the number of elements in a cache line. However, with fixed blocks from X and Y , we perform B^3 multiply-and-add operations in lines (4) through (7) of Fig. 11.8. Since the entire matrix-multiplication requires n^3 multiply-and-add operations, the number of times we need to bring a pair of blocks to the cache is n^3/B^3 . As we require $2B^2/c$ cache misses each time we do, the total number of cache misses is $2n^3/Bc$.

It is interesting to compare this figure $2n^3/Bc$ with the estimates given in Section 11.2.1. There, we said that if entire matrices can fit in the cache, then $O(n^2/c)$ cache misses suffice. However, in that case, we can pick $B = n$, i.e., make each matrix be a single block. We again get $O(n^2/c)$ as our estimate of cache misses. On the other hand, we observed that if entire matrices will not fit in cache, we require $O(n^3/c)$ cache misses, or even $O(n^3)$ cache misses. In that case, assuming that we can still pick a significantly large B (e.g., B could be 200, and we could still fit three blocks of 8-byte numbers in a one-megabyte cache), there is a great advantage to using blocking in matrix multiplication.

The blocking technique can be reapplied for each level of the memory hierarchy. For example, we may wish to optimize register usage by holding the operands of a 2×2 matrix multiplication in registers. We choose successively bigger block sizes for the different levels of caches and physical memory.

Similarly, we can distribute blocks between processors to minimize data traffic. Experiments showed that such optimizations can improve the performance of a uniprocessor by a factor of 3, and the speed up on a multiprocessor is close to linear with respect to the number of processors used.

11.2.3 Cache Interference

Unfortunately, there is somewhat more to the story of cache utilization. Most caches are not fully associative (see Section 7.4.2). In a direct-mapped cache, if n is a multiple of the cache size, then all the elements in the same row of an $n \times n$ array will be competing for the same cache location. In that case, bringing in the second element of a column will throw away the cache line of the first, even though the cache has the capacity to keep both of these lines at the same time. This situation is referred to as *cache interference*.

There are various solutions to this problem. The first is to rearrange the data once and for all so that the data accessed is laid out in consecutive data locations. The second is to embed the $n \times n$ array in a larger $m \times n$ array where m is chosen to minimize the interference problem. Third, in some cases we can choose a block size that is guaranteed to avoid interference.

11.2.4 Exercises for Section 11.2

Exercise 11.2.1: The block-based matrix-multiplication algorithm of Fig. 11.8 does not have the initialization of the matrix Z to zero, as the code of Fig. 11.5 does. Add the steps that initialize Z to all zeros in Fig. 11.8.

11.3 Iteration Spaces

The motivation for this study is to exploit the techniques that, in simple settings like matrix multiplication as in Section 11.2, were quite straightforward. In the more general setting, the same techniques apply, but they are far less intuitive. But by applying some linear algebra, we can make everything work in the general setting.

As discussed in Section 11.1.5, there are three kinds of spaces in our transformation model: iteration space, data space, and processor space. Here we start with the iteration space. The iteration space of a loop nest is defined to be all the combinations of loop-index values in the nest.

Often, the iteration space is rectangular, as in the matrix-multiplication example of Fig. 11.5. There, each of the nested loops had a lower bound of 0 and an upper bound of $n - 1$. However, in more complicated, but still quite realistic, loop nests, the upper and/or lower bounds on one loop index can depend on the values of the indexes of the outer loops. We shall see an example shortly.

11.3.1 Constructing Iteration Spaces from Loop Nests

To begin, let us describe the sort of loop nests that can be handled by the techniques to be developed. Each loop has a single loop index, which we assume is incremented by 1 at each iteration. That assumption is without loss of generality, since if the incrementation is by integer $c > 1$, we can always replace

uses of the index i by uses of $ci + a$ for some positive or negative constant a , and then increment i by 1 in the loop. The bounds of the loop should be written as affine expressions of outer loop indices.

Example 11.5: Consider the loop

```
for (i = 2; i <= 100; i = i+3)
    Z[i] = 0;
```

which increments i by 3 each time around the loop. The effect is to set to 0 each of the elements $Z[2], Z[5], Z[8], \dots, Z[98]$. We can get the same effect with:

```
for (j = 0; j <= 32; j++)
    Z[3*j+2] = 0;
```

That is, we substitute $3j + 2$ for i . The lower limit $i = 2$ becomes $j = 0$ (just solve $3j + 2 = 2$ for j), and the upper limit $i \leq 100$ becomes $j \leq 32$ (simplify $3j + 2 \leq 100$ to get $j \leq 32.67$ and round down because j has to be an integer). \square

Typically, we shall use for-loops in loop nests. A while-loop or repeat-loop can be replaced by a for-loop if there is an index and upper and lower bounds for the index, as would be the case in something like the loop of Fig. 11.9(a). A for-loop like `for (i=0; i<100; i++)` serves exactly the same purpose.

However, some while- or repeat-loops have no obvious limit. For example, Fig. 11.9(b) may or may not terminate, but there is no way to tell what condition on i in the unseen body of the loop causes the loop to break. Figure 11.9(c) is another problem case. Variable n might be a parameter of a function, for example. We know the loop iterates n times, but we don't know what n is at compile time, and in fact we may expect that different executions of the loop will execute different numbers of times. In cases like (b) and (c), we must treat the upper limit on i as infinity.

A d -deep loop nest can be modeled by a d -dimensional space. The dimensions are ordered, with the k th dimension representing the k th nested loop, counting from the outermost loop, inward. A point (x_1, x_2, \dots, x_d) in this space represents values for all the loop indexes; the outermost loop index has value x_1 , the second loop index has value x_2 , and so on. The innermost loop index has value x_d .

But not all points in this space represent combinations of indexes that actually occur during execution of the loop nest. As an affine function of outer loop indices, each lower and upper loop bound defines an inequality dividing the iteration space into two half spaces: those that are iterations in the loop (the *positive* half space), and those that are not (the *negative* half space). The conjunction (logical AND) of all the linear equalities represents the intersection of the positive half spaces, which defines a convex polyhedron, which we call the *iteration space* for the loop nest. A *convex polyhedron* has the property that if

```

i = 0;
while (i<100) {
    <some statements not involving i>
    i = i+1;
}

```

(a) A while-loop with obvious limits.

```

i = 0;
while (1) {
    <some statements>
    i = i+1;
}

```

(b) It is unclear when or if this loop terminates.

```

i = 0;
while (i<n) {
    <some statements not involving i or n>
    i = i+1;
}

```

(c) We don't know the value of n , so we don't know when this loop terminates.

Figure 11.9: Some while-loops

two points are in the polyhedron, all points on the line between them are also in the polyhedron. All the iterations in the loop are represented by the points with integer coordinates found within the polyhedron described by the loop-bound inequalities. And conversely, all integer points within the polyhedron represent iterations of the loop nest at some time.

```

for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;

```

Figure 11.10: A 2-dimensional loop nest

Example 11.6: Consider the 2-dimensional loop nest in Fig. 11.10. We can model this two-deep loop nest by the 2-dimensional polyhedron shown in Fig. 11.11. The two axes represent the values of the loop indexes i and j . Index i can take on any integral value between 0 and 5; index j can take on any integral value such that $i \leq j \leq 7$. \square

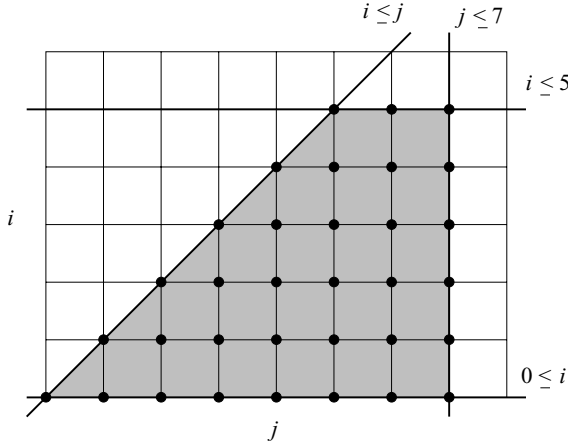


Figure 11.11: The iteration space of Example 11.6

Iteration Spaces and Array-Accesses

In the code of Fig. 11.10, the iteration space is also the portion of the array A that the code accesses. That sort of access, where the array indexes are also loop indexes in some order, is very common. However, we should not confuse the space of iterations, whose dimensions are loop indexes, with the data space. If we had used in Fig. 11.10 an array access like $Z[2*i, i+j]$ instead of $Z[j, i]$, the difference would have been apparent.

11.3.2 Execution Order for Loop Nests

A sequential execution of a loop nest sweeps through iterations in its iteration space in an ascending lexicographic order. A vector $\mathbf{i} = [i_0, i_1, \dots, i_n]$ is *lexicographically less than* another vector $\mathbf{i}' = [i'_0, i'_1, \dots, i'_n]$, written $\mathbf{i} < \mathbf{i}'$, if and only if there exists an $m < \min(n, n')$ such that $[i_0, i_1, \dots, i_m] = [i'_0, i'_1, \dots, i'_m]$ and $i_{m+1} < i'_{m+1}$. Note that $m = 0$ is possible, and in fact common.

Example 11.7: With i as the outer loop, the iterations in the loop nest in Example 11.6 are executed in the order shown in Fig. 11.12. \square

11.3.3 Matrix Formulation of Inequalities

The iterations in a d -deep loop can be represented mathematically as

$$\{\mathbf{i} \text{ in } Z^d \mid \mathbf{B}\mathbf{i} + \mathbf{b} \geq \mathbf{0}\} \quad (11.1)$$

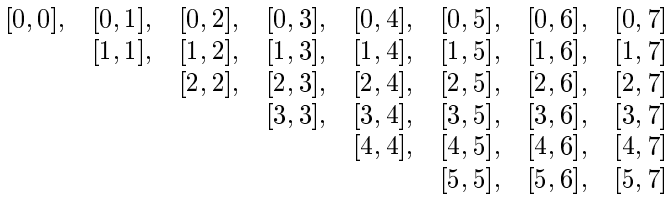


Figure 11.12: Iteration order for loop nest of Fig. 11.10

Here,

1. Z , as is conventional in mathematics, represents the set of integers — positive, negative, and zero,
2. \mathbf{B} is a $d \times d$ integer matrix,
3. \mathbf{b} is an integer vector of length d , and
4. $\mathbf{0}$ is a vector of d 0's.

Example 11.8: We can write the inequalities of Example 11.6 as in Fig. 11.13. That is, the range of i is described by $i \geq 0$ and $i \leq 5$; the range of j is described by $j \geq i$ and $j \leq 7$. We need to put each of these inequalities in the form $ui + vj + w \geq 0$. Then, $[u, v]$ becomes a row of the matrix \mathbf{B} in the inequality (11.1), and w becomes the corresponding component of the vector \mathbf{b} . For instance, $i \geq 0$ is of this form, with $u = 1$, $v = 0$, and $w = 0$. This inequality is represented by the first row of \mathbf{B} and top element of \mathbf{b} in Fig. 11.13.

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 11.13: Matrix-vector multiplication and a vector inequality represents the inequalities defining an iteration space

As another example, the inequality $i \leq 5$ is equivalent to $(-1)i + (0)j + 5 \geq 0$, and is represented by the second row of \mathbf{B} and \mathbf{b} in Fig. 11.13. Also, $j \geq i$ becomes $(-1)i + (1)j + 0 \geq 0$ and is represented by the third row. Finally, $j \leq 7$ becomes $(0)i + (-1)j + 7 \geq 0$ and is the last row of the matrix and vector. \square

Manipulating Inequalities

To convert inequalities, as in Example 11.8, we can perform transformations much as we do for equalities, e.g., adding or subtracting from both sides, or multiplying both sides by a constant. The only special rule we must remember is that when we multiply both sides by a negative number, we have to reverse the direction of the inequality. Thus, $i \leq 5$, multiplied by -1 , becomes $-i \geq -5$. Adding 5 to both sides, gives $-i + 5 \geq 0$, which is essentially the second row of Fig. 11.13.

11.3.4 Incorporating Symbolic Constants

Sometimes, we need to optimize a loop nest that involves certain variables that are loop-invariant for all the loops in the nest. We call such variables *symbolic constants*, but to describe the boundaries of an iteration space we need to treat them as variables and create an entry for them in the vector of loop indexes, i.e., the vector \mathbf{i} in the general formulation of inequalities (11.1).

Example 11.9: Consider the simple loop:

```
for (i = 0; i <= n; i++) {
    ...
}
```

This loop defines a one-dimensional iteration space, with index i , bounded by $i \geq 0$ and $i \leq n$. Since n is a symbolic constant, we need to include it as a variable, giving us a vector of loop indexes $[i, n]$. In matrix-vector form, this iteration space is defined by

$$\left\{ i \text{ in } Z \mid \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}.$$

Notice that, although the vector of array indexes has two dimensions, only the first of these, representing i , is part of the output — the set of points lying with the iteration space. \square

11.3.5 Controlling the Order of Execution

The linear inequalities extracted from the lower and upper bounds of a loop body define a set of iterations over a convex polyhedron. As such, the representation assumes no execution ordering between iterations within the iteration space. The original program imposes one sequential order on the iterations, which is the lexicographic order with respect to the loop index variables ordered from the outermost to the innermost. However, the iterations in the space can be executed in any order as long as their data dependences are honored (i.e.,

the order in which writes and reads of any array element are performed by the various assignment statements inside the loop nest do not change).

The problem of how we choose an ordering that honors the data dependences and optimizes for data locality and parallelism is hard and is dealt with later starting from Section 11.7. Here we assume that a legal and desirable ordering is given, and show how to generate code that enforce the ordering. Let us start by showing an alternative ordering for Example 11.6.

Example 11.10: There are no dependences between iterations in the program in Example 11.6. We can therefore execute the iterations in arbitrary order, sequentially or concurrently. Since iteration $[i, j]$ accesses element $Z[j, i]$ in the code, the original program visits the array in the order of Fig. 11.14(a). To improve spatial locality, we prefer to visit contiguous words in the array consecutively, as in Fig. 11.14(b).

This access pattern is obtained if we execute the iterations in the order shown in Fig. 11.14(c). That is, instead of sweeping the iteration space in Fig. 11.11 horizontally, we sweep the iteration space vertically, so j becomes the index of the outer loop. The code that executes the iterations in the above order is

```
for (j = 0; j <= 7; j++)
  for (i = 0; i <= min(5,j); i++)
    Z[j,i] = 0;
```

□

Given a convex polyhedron and an ordering of the index variables, how do we generate the loop bounds that sweep through the space in lexicographic order of the variables? In the example above, the constraint $i \leq j$ shows up as a lower bound for index j in the inner loop in the original program, but as an upper bound for index i , again in the inner loop, in the transformed program.

The bounds of the outermost loop, expressed as linear combinations of symbolic constants and constants, define the range of all the possible values it can take on. The bounds for inner loop variables are expressed as linear combinations of outer loop index variables, symbolic constants and constants. They define the range the variable can take on for each combination of values in outer loop variables.

Projection

Geometrically speaking, we can find the loop bounds of the outer loop index in a two-deep loop nest by *projecting* the convex polyhedron representing the iteration space onto the outer dimension of the space. The projection of a polyhedron on a lower-dimensional space is intuitively the shadow cast by the object onto that space. The projection of the two-dimensional iteration space in Fig. 11.11 onto the i axis is the vertical line from 0 to 5; and the projection onto

$Z[0, 0]$,	$Z[1, 0]$,	$Z[2, 0]$,	$Z[3, 0]$,	$Z[4, 0]$,	$Z[5, 0]$,	$Z[6, 0]$,	$Z[7, 0]$
	$Z[1, 1]$,	$Z[2, 1]$,	$Z[3, 1]$,	$Z[4, 1]$,	$Z[5, 1]$,	$Z[6, 1]$,	$Z[1, 7]$
		$Z[2, 2]$,	$Z[3, 2]$,	$Z[4, 2]$,	$Z[5, 2]$,	$Z[6, 2]$,	$Z[7, 2]$
			$Z[3, 3]$,	$Z[4, 3]$,	$Z[5, 3]$,	$Z[6, 3]$,	$Z[7, 3]$
				$Z[4, 4]$,	$Z[5, 4]$,	$Z[6, 4]$,	$Z[7, 4]$
					$Z[5, 5]$,	$Z[6, 5]$,	$Z[7, 5]$

(a) Original access order.

$Z[0, 0]$
$Z[1, 0]$, $Z[1, 1]$
$Z[2, 0]$, $Z[2, 1]$, $Z[2, 2]$
$Z[3, 0]$, $Z[3, 1]$, $Z[3, 2]$, $Z[3, 3]$
$Z[4, 0]$, $Z[4, 1]$, $Z[4, 2]$, $Z[4, 3]$, $Z[4, 4]$
$Z[5, 0]$, $Z[5, 1]$, $Z[5, 2]$, $Z[5, 3]$, $Z[5, 4]$, $Z[5, 5]$
$Z[6, 0]$, $Z[6, 1]$, $Z[6, 2]$, $Z[6, 3]$, $Z[6, 4]$, $Z[6, 5]$
$Z[7, 0]$, $Z[7, 1]$, $Z[7, 2]$, $Z[7, 3]$, $Z[7, 4]$, $Z[7, 5]$

(b) Preferred order of access.

$[0, 0]$
$[0, 1]$, $[1, 1]$
$[0, 2]$, $[1, 2]$, $[2, 2]$
$[0, 3]$, $[1, 3]$, $[2, 3]$, $[3, 3]$
$[0, 4]$, $[1, 4]$, $[2, 4]$, $[3, 4]$, $[4, 4]$
$[0, 5]$, $[1, 5]$, $[2, 5]$, $[3, 5]$, $[4, 5]$, $[5, 5]$
$[0, 6]$, $[1, 6]$, $[2, 6]$, $[3, 6]$, $[4, 6]$, $[5, 6]$
$[0, 7]$, $[1, 7]$, $[2, 7]$, $[3, 7]$, $[4, 7]$, $[5, 7]$

(c) Preferred order of iterations.

Figure 11.14: Reordering the accesses and iterations for a loop nest

the j axis is the horizontal line from 0 to 7. When we project a 3-dimensional object along the z axis onto a 2-dimensional x and y plane, we eliminate variable z , losing the height of the individual points and simply record the 2-dimensional footprint of the object in the x - y plane.

Loop bound generation is only one of the many uses of projection. Projection can be defined formally as follows. Let S be an n -dimensional polyhedron. The projection of S onto the first m of its dimensions is the set of points (x_1, x_2, \dots, x_m) such that for some $x_{m+1}, x_{m+2}, \dots, x_n$, vector $[x_1, x_2, \dots, x_n]$ is in S . We can compute projection using *Fourier-Motzkin elimination*, as follows:

Algorithm 11.11: Fourier-Motzkin elimination.

INPUT: A polyhedron S with variables x_1, x_2, \dots, x_n . That is, S is a set of linear constraints involving the variables x_i . One given variable x_m is specified to be the variable to be eliminated.

OUTPUT: A polyhedron S' with variables $x_1, \dots, x_{m-1}, x_{m+1}, \dots, x_n$ (i.e., all the variables of S except for x_m) that is the projection of S onto dimensions other than the m th.

METHOD: Let C be all the constraints in S involving x_m . Do the following:

1. For every pair of a lower bound and an upper bound on x_m in C , such as

$$\begin{array}{rcl} L & \leq & c_1 x_m \\ & & c_2 x_m \leq U \end{array}$$

create the new constraint

$$c_2 L \leq c_1 U$$

Note that c_1 and c_2 are integers, but L and U may be expressions with variables other than x_m .

2. If integers c_1 and c_2 have a common factor, divide both sides by that factor.
3. If the new constraint is not satisfiable, then there is no solution to S ; i.e., the polyhedra S and S' are both empty spaces.
4. S' is the set of constraints $S - C$, plus all the constraints generated in step 2.

Note, incidentally, that if x_m has u lower bounds and v upper bounds, eliminating x_m produces up to uv inequalities, but no more. \square

The constraints added in step (1) of Algorithm 11.11 correspond to the implications of constraints C on the remaining variables in the system. Therefore, there is a solution in S' if and only if there exists at least one corresponding

solution in S . Given a solution in S' the range of the corresponding x_m can be found by replacing all variables but x_m in the constraints C by their actual values.

Example 11.12: Consider the inequalities defining the iteration space in Fig. 11.11. Suppose we wish to use Fourier-Motzkin elimination to project the two-dimensional space away from the i dimension and onto the j dimension. There is one lower bound on i : $0 \leq i$ and two upper bounds: $i \leq j$ and $i \leq 5$. This generates two constraints: $0 \leq j$ and $0 \leq 5$. The latter is trivially true and can be ignored. The former gives the lower bound on j , and the original upper bound $j \leq 7$ gives the upper bound. \square

Loop-Bounds Generation

Now that we have defined Fourier-Motzkin elimination, the algorithm to generate the loop bounds to iterate over a convex polyhedron (Algorithm 11.13) is straightforward. We compute the loop bounds in order, from the innermost to the outer loops. All the inequalities involving the innermost loop index variables are written as the variable's lower or upper bounds. We then project away the dimension representing the innermost loop and obtain a polyhedron with one fewer dimension. We repeat until the bounds for all the loop index variables are found.

Algorithm 11.13: Computing bounds for a given order of variables.

INPUT: A convex polyhedron S over variables v_1, \dots, v_n .

OUTPUT: A set of lower bounds L_i and upper bounds U_i for each v_i , expressed only in terms of the v_j 's, for $j < i$.

METHOD: The algorithm is described in Fig. 11.15. \square

Example 11.14: We apply Algorithm 11.13 to generate the loop bounds that sweep the iteration space of Fig. 11.11 vertically. The variables are ordered j, i . The algorithm generates these bounds:

$$\begin{array}{ll} L_i : & 0 \\ U_i : & 5, j \\ L_j : & 0 \\ U_j : & 7 \end{array}$$

We need to satisfy all the constraints, thus the bound on i is $\min(5, j)$. There are no redundancies in this example. \square

```

 $S_n = S$ ; /* Use Algorithm 11.11 to find the bounds */
for (  $i = n$ ;  $i \geq 1$ ;  $i - -$  ) {
     $L_{v_i}$  = all the lower bounds on  $v_i$  in  $S_i$ ;
     $U_{v_i}$  = all the upper bounds on  $v_i$  in  $S_i$ ;
     $S_{i-1}$  = Constraints returned by applying Algorithm 11.11
           to eliminate  $v_i$  from the constraints  $S_i$ ;
}
/* Remove redundancies */
 $S' = \emptyset$ ;
for (  $i = 1$ ;  $i \leq n$ ;  $i + +$  ) {
    Remove any bounds in  $L_{v_i}$  and  $U_{v_i}$  implied by  $S'$ ;
    Add the remaining constraints of  $L_{v_i}$  and  $U_{v_i}$  on  $v_i$  to  $S'$ ;
}

```

Figure 11.15: Code to express variable bounds with respect to a given variable ordering

[0, 0],	[1, 1],	[2, 2],	[3, 3],	[4, 4],	[5, 5]
[0, 1],	[1, 2],	[2, 3],	[3, 4],	[4, 5],	[5, 6]
[0, 2],	[1, 3],	[2, 4],	[3, 5],	[4, 6],	[5, 7]
[0, 3],	[1, 4],	[2, 5],	[3, 6],	[4, 7]	
[0, 4],	[1, 5],	[2, 6],	[3, 7]		
[0, 5],	[1, 6],	[2, 7]			
[0, 6],	[1, 7]				
[0, 7]					

Figure 11.16: Diagonalwise ordering of the iteration space of Fig. 11.11

11.3.6 Changing Axes

Note that sweeping the iteration space horizontally and vertically, as discussed above, are just two of the most common ways of visiting the iteration space. There are many other possibilities; for example, we can sweep the iteration space in Example 11.6 diagonal by diagonal, as discussed below in Example 11.15.

Example 11.15: We can sweep the iteration space shown in Fig. 11.11 diagonally using the order shown in Fig. 11.16. The difference between the coordinates j and i in each diagonal is a constant, starting with 0 and ending with 7. Thus, we define a new variable $k = j - i$ and sweep through the iteration space in lexicographic order with respect to k and j . Substituting $i = j - k$ in the inequalities we get:

$$\begin{array}{rcl}
 0 \leq & j - k & \leq 5 \\
 j - k \leq & j & \leq 7
 \end{array}$$

To create the loop bounds for the order described above, we can apply Algorithm 11.13 to the above set of inequalities with variable ordering k, j .

$$\begin{array}{ll} L_j : & k \\ U_j : & 5 + k, 7 \\ L_k : & 0 \\ U_k : & 7 \end{array}$$

From these inequalities, we generate the following code, replacing i by $j - k$ in array accesses.

```
for (k = 0; k <= 7; k++)
  for (j = k; j <= min(5+k,7); j++)
    Z[j,j-k] = 0;
```

□

In general, we can change the axes of a polyhedron by creating new loop index variables that represent affine combinations of the original variables, and defining an ordering on those variables. The hard problem lies in choosing the right axes to satisfy the data dependences while achieving the parallelism and locality objectives. We discuss this problem starting with Section 11.7. What we have established here is that once the axes are chosen, it is straightforward to generate the desired code, as shown in Example 11.15.

There are many other iteration-traversal orders not handled by this technique. For example, we may wish to visit all the odd rows in an iteration space before we visit the even rows. Or, we may want to start with the iterations in the middle of the iteration space and progress to the fringes. For applications that have affine access functions, however, the techniques described here cover most of the desirable iteration orderings.

11.3.7 Exercises for Section 11.3

Exercise 11.3.1: Convert each of the following loops to a form where the loop indexes are each incremented by 1:

- a) for (i=10; i<50; i=i+7) X[i,i+1] = 0;.
- b) for (i= -3; i<=10; i=i+2) X[i] = X[i+1];.
- c) for (i=50; i>=10; i--) X[i] = 0;.

Exercise 11.3.2: Draw or describe the iteration spaces for each of the following loop nests:

- a) The loop nest of Fig. 11.17(a).
- b) The loop nest of Fig. 11.17(b).

```

for (i = 1; i < 30; i++)
    for (j = i+2; j < 40-i; j++)
        X[i,j] = 0;

```

(a) Loop nest for Exercise 11.3.2(a).

```

for (i = 10; i <= 1000; i++)
    for (j = i; j < i+10; j++)
        X[i,j] = 0;

```

(b) Loop nest for Exercise 11.3.2(b).

```

for (i = 0; i < 100; i++)
    for (j = 0; j < 100+i; j++)
        for (k = i+j; k < 100-i-j; k++)
            X[i,j,k] = 0;

```

(c) Loop nest for Exercise 11.3.2(c).

Figure 11.17: Loop nests for Exercise 11.3.2

c) The loop nest of Fig. 11.17(c).

Exercise 11.3.3: Write the constraints implied by each of the loop nests of Fig. 11.17 in the form of (11.1). That is, give the values of the vectors \mathbf{i} and \mathbf{b} and the matrix \mathbf{B} .

Exercise 11.3.4: Reverse each of the loop-nesting orders for the nests of Fig. 11.17.

Exercise 11.3.5: Use the Fourier-Motzkin elimination algorithm to eliminate i from each of the sets of constraints obtained in Exercise 11.3.3.

Exercise 11.3.6: Use the Fourier-Motzkin elimination algorithm to eliminate j from each of the sets of constraints obtained in Exercise 11.3.3.

Exercise 11.3.7: For each of the loop nests in Fig. 11.17, rewrite the code so the axis i is replaced by the major diagonal, i.e., the direction of the axis is characterized by $i = j$. The new axis should correspond to the outermost loop.

Exercise 11.3.8: Repeat Exercise 11.3.7 for the following changes of axes:

- Replace i by $i + j$; i.e., the direction of the axis is the lines for which $i + j$ is a constant. The new axis corresponds to the outermost loop.
- Replace j by $i - 2j$. The new axis corresponds to the outermost loop.

! Exercise 11.3.9: Let A , B , and C be integer constants in the following loop, with $C > 1$ and $B > A$:

```
for (i = A; i <= B; i = i + C)
    Z[i] = 0;
```

Rewrite the loop so the incrementation of the loop variable is 1 and the initialization is to 0, that is, to be of the form

```
for (j = 0; j <= D; j++)
    Z[E*j + F] = 0;
```

for integers D , E , and F . Express D , E , and F in terms of A , B , and C .

Exercise 11.3.10: For a generic two-loop nest

```
for (i = 0; i <= A; i++)
    for(j = B*i+C; j <= D*i+E; j++)
```

with A through E integer constants, write the constraints that define the loop nest's iteration space in matrix-vector form, i.e., in the form $\mathbf{B}\mathbf{i} + \mathbf{b} = \mathbf{0}$.

Exercise 11.3.11: Repeat Exercise 11.3.10 for a generic two-loop nest with symbolic integer constants m and n as in

```
for (i = 0; i <= m; i++)
    for(j = A*i+B; j <= C*i+n; j++)
```

As before, A , B , and C stand for specific integer constants. Only i , j , m , and n should be mentioned in the vector of unknowns. Also, remember that only i and j are output variables for the expression.

11.4 Affine Array Indexes

The focus of this chapter is on the class of affine array accesses, where each array index is expressed as affine expressions of loop indexes and symbolic constants. Affine functions provide a succinct mapping from the iteration space to the data space, making it easy to determine which iterations map to the same data or same cache line.

Just as the affine upper and lower bounds of a loop can be represented as a matrix-vector calculation, we can do the same for affine access functions. Once placed in the matrix-vector form, we can apply standard linear algebra to find pertinent information such as the dimensions of the data accessed, and which iterations refer to the same data.

11.4.1 Affine Accesses

We say that an array access in a loop is *affine* if

1. The bounds of the loop are expressed as affine expressions of the surrounding loop variables and symbolic constants, and
2. The index for each dimension of the array is also an affine expression of surrounding loop variables and symbolic constants.

Example 11.16: Suppose i and j are loop index variables bounded by affine expressions. Some examples of affine array accesses are $Z[i]$, $Z[i + j + 1]$, $Z[0]$, $Z[i, i]$, and $Z[2 * i + 1, 3 * j - 10]$. If n is a symbolic constant for a loop nest, then $Z[3 * n, n - j]$ is another example of an affine array access. However, $Z[i * j]$ and $Z[n * j]$ are not affine accesses. \square

Each affine array access can be described by two matrices and two vectors. The first matrix-vector pair is the \mathbf{B} and \mathbf{b} that describe the iteration space for the access, as in the inequality of Equation (11.1). The second pair, which we usually refer to as \mathbf{F} and \mathbf{f} , represent the function(s) of the loop-index variables that produce the array index(es) used in the various dimensions of the array access.

Formally, we represent an array access in a loop nest that uses a vector of index variables \mathbf{i} by the four-tuple $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$; it maps a vector \mathbf{i} within the bounds

$$\mathbf{B}\mathbf{i} + \mathbf{b} \geq \mathbf{0}$$

to the array element location

$$\mathbf{F}\mathbf{i} + \mathbf{f}$$

Example 11.17: In Fig. 11.18 are some common array accesses, expressed in matrix notation. The two loop indexes are i and j , and these form the vector \mathbf{i} . Also, X , Y , and Z are arrays with 1, 2, and 3 dimensions, respectively.

The first access, $X[i - 1]$, is represented by a 1×2 matrix \mathbf{F} and a vector \mathbf{f} of length 1. Notice that when we perform the matrix-vector multiplication and add in the vector \mathbf{f} , we are left with a single function, $i - 1$, which is exactly the formula for the access to the one-dimensional array X . Also notice the third access, $Y[j, j + 1]$, which, after matrix-vector multiplication and addition, yields a pair of functions, $(j, j + 1)$. These are the indexes of the two dimensions of the array access.

Finally, let us observe the fourth access $Y[1, 2]$. This access is a constant, and unsurprisingly the matrix \mathbf{F} is all 0's. Thus, the vector of loop indexes, \mathbf{i} , does not appear in the access function. \square

ACCESS	AFFINE EXPRESSION
$x[i-1]$	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$
$y[i,j]$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$y[j,j+1]$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$y[1,2]$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$
$z[1,i,2*i+j]$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

Figure 11.18: Some array accesses and their matrix-vector representations

11.4.2 Affine and Nonaffine Accesses in Practice

There are certain common data access patterns found in numerical programs that fail to be affine. Programs involving sparse matrices are one important example. One popular representation for sparse matrices is to store only the nonzero elements in a vector, and auxiliary index arrays are used to mark where a row starts and which columns contain nonzeros. Indirect array accesses are used in accessing such data. An access of this type, such as $X[Y[i]]$, is a nonaffine access to the array X . If the sparsity is regular, as in banded matrices having nonzeros only around the diagonal, then dense arrays can be used to represent the subregions with nonzero elements. In that case, accesses may be affine.

Another common example of nonaffine accesses is linearized arrays. Programmers sometimes use a linear array to store a logically multidimensional object. One reason why this is the case is that the dimensions of the array may not be known at compile time. An access that would normally look like $Z[i,j]$ would be expressed as $Z[i * n + j]$, which is a quadratic function. We can convert the linear access into a multidimensional access if every access can

be decomposed into separate dimensions with the guarantee that none of the components exceeds its bound. Finally, we note that induction-variable analyses can be used to convert some nonaffine accesses into affine ones, as shown in Example 11.18.

Example 11.18: We can rewrite the code

```
j = n;
for (i = 0; i <= n; i++) {
    Z[j] = 0;
    j = j+2;
}
```

as

```
j = n;
for (i = 0; i <= n; i++) {
    Z[n+2*i] = 0;
}
```

to make the access to matrix Z affine. \square

11.4.3 Exercises for Section 11.4

Exercise 11.4.1: For each of the following array accesses, give the vector \mathbf{f} and the matrix \mathbf{F} that describe them. Assume that the vector of indexes \mathbf{i} is i, j, \dots , and that all loop indexes have affine limits.

- a) $X[2 * i + 3, 2 * j - i]$.
- b) $Y[i - j, j - k, k - i]$.
- c) $Z[3, 2 * j, k - 2 * i + 1]$.

11.5 Data Reuse

From array access functions we derive two kinds of information useful for locality optimization and parallelization:

1. *Data reuse*: for locality optimization, we wish to identify sets of iterations that access the same data or the same cache line.
2. *Data dependence*: for correctness of parallelization and locality loop transformations, we wish to identify *all* the data dependences in the code. Recall that two (not necessarily distinct) accesses have a data dependence if instances of the accesses may refer to the same memory location, and at least one of them is a write.

In many cases, whenever we identify iterations that reuse the same data, there are data dependences between them.

Whenever there is a data dependence, obviously the same data is reused. For example, in matrix multiplication, the same element in the output array is written $O(n)$ times. The write operations must be executed in the original execution order;³ we can exploit the reuse by allocating a register to hold one element of the output array while it is being computed.

However, not all reuse can be exploited in locality optimizations; here is an example illustrating this issue.

Example 11.19: Consider the following loop:

```
for (i = 0; i < n; i++)
    Z[7*i+3] = Z[3*i+5];
```

We observe that the loop writes to a different location at each iteration, so there are no reuses or dependences on the different write operations. The loop, however, reads locations 5, 8, 11, 14, 17, ..., and writes locations 3, 10, 17, 24, The read and write iterations access the same elements 17, 38, and 59 and so on. That is, the integers of the form $17 + 21j$ for $j = 0, 1, 2, \dots$ are all those integers that can be written both as $7i_1 + 3$ and as $3i_2 + 5$, for some integers i_1 and i_2 . However, this reuse occurs rarely, and cannot be exploited easily if at all. \square

Data dependence is different from reuse analysis in that one of the accesses sharing a data dependence must be a write access. More importantly, data dependence needs to be both correct and precise. It needs to find all dependences for correctness, and it should not find spurious dependences because they can cause unnecessary serialization.

With data reuse, we only need to find where most of the exploitable reuses are. This problem is much simpler, so we take up this topic here in this section and tackle data dependences in the next. We simplify reuse analysis by ignoring loop bounds, because they seldom change the shape of the reuse. Much of the reuse exploitable by affine partitioning resides among instances of the same array accesses, and accesses that share the same *coefficient matrix* (what we have typically called **F** in the affine index function). As shown above, access patterns like $7i + 3$ and $3i + 5$ have no reuse of interest.

11.5.1 Types of Reuse

We first start with Example 11.20 to illustrate the different kinds of data reuses. In the following, we need to distinguish between the access as an instruction in

³There is a subtle point here. Because of the commutativity of addition, we would get the same answer to the sum regardless of the order in which we performed the sum. However, this case is very special. In general, it is far too complex for the compiler to determine what computation is being performed by a sequence of arithmetic steps followed by writes, and we cannot rely on there being any algebraic rules that will help us reorder the steps safely.

a program, e.g., $x = Z[i, j]$, from the execution of this instruction many times, as we execute the loop nest. For emphasis, we may refer to the statement itself as a *static access*, while the various iterations of the statement as we execute its loop nest are called *dynamic accesses*.

Reuses can be classified as *self* versus *group*. If iterations reusing the same data come from the same static access, we refer to the reuse as self reuse; if they come from different accesses, we refer to it as group reuse. The reuse is *temporal* if the same exact location is referenced; it is *spatial* if the same cache line is referenced.

Example 11.20: Consider the following loop nest:

```
float Z[n];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/3;
```

Accesses $Z[j]$, $Z[j + 1]$, and $Z[j + 2]$ each have self-spatial reuse because consecutive iterations of the same access refer to contiguous array elements. Presumably contiguous elements are very likely to reside on the same cache line. In addition, they all have self-temporal reuse, since the exact elements are used over and over again in each iteration in the outer loop. In addition, they all have the same coefficient matrix, and thus have group reuse. There is group reuse, both temporal and spatial, between the different accesses. Although there are $4n^2$ accesses in this code, if the reuse can be exploited, we only need to bring in about n/c cache lines into the cache, where c is the number of words in a cache line. We drop a factor of n due to self-spatial reuse, a factor of c due to spatial locality, and finally a factor of 4 due to group reuse. \square

In the following, we show how we can use linear algebra to extract the reuse information from affine array accesses. We are interested in not just finding how much potential savings there are, but also which iterations are reusing the data so that we can try to move them close together to exploit the reuse.

11.5.2 Self Reuse

There can be substantial savings in memory accesses by exploiting self reuse. If the data referenced by a static access has k dimensions and the access is nested in a loop d deep, for some $d > k$, then the same data can be reused n^{d-k} times, where n is the number of iterations in each loop. For example, if a 3-deep loop nest accesses one column of an array, then there is a potential savings factor of n^2 accesses. It turns out that the dimensionality of an access corresponds to the concept of the *rank* of the coefficient matrix in the access, and we can find which iterations refer to the same location by finding the *null space* of the matrix, as explained below.

Rank of a Matrix

The rank of a matrix \mathbf{F} is the largest number of columns (or equivalently, rows) of \mathbf{F} that are linearly independent. A set of vectors is *linearly independent* if none of the vectors can be written as a linear combination of finitely many other vectors in the set.

Example 11.21: Consider the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

Notice that the second row is the sum of the first and third rows, while the fourth row is the third row minus twice the first row. However, the first and third rows are linearly independent; neither is a multiple of the other. Thus, the rank of the matrix is 2.

We could also draw this conclusion by examining the columns. The third column is twice the second column minus the first column. On the other hand, any two columns are linearly independent. Again, we conclude that the rank is 2. \square

Example 11.22: Let us look at the array accesses in Fig. 11.18. The first access, $X[i-1]$, has dimension 1, because the rank of the matrix $[1 \ 0]$ is 1. That is, the one row is linearly independent, as is the first column.

The second access, $Y[i, j]$, has dimension 2. The reason is that the matrix

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

has two independent rows (and therefore two independent columns, of course). The third access, $Y[j, j+1]$, is of dimension 1, because the matrix

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

has rank 1. Note that the two rows are identical, so only one is linearly independent. Equivalently, the first column is 0 times the second column, so the columns are not independent. Intuitively, in a large, square array Y , the only elements accessed lie along a one-dimensional line, just above the main diagonal.

The fourth access, $Y[1, 2]$ has dimension 0, because a matrix of all 0's has rank 0. Note that for such a matrix, we cannot find a linear sum of even one row that is nonzero. Finally, the last access, $Z[i, i, 2 * i + j]$, has dimension 2. Note that in the matrix for this access

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

the last two rows are linearly independent; neither is a multiple of the other. However, the first row is a linear “sum” of the other two rows, with both coefficients 0. \square

Null Space of a Matrix

A reference in a d -deep loop nest with rank r accesses $O(n^r)$ data elements in $O(n^d)$ iterations, so on average, $O(n^{d-r})$ iterations must refer to the same array element. Which iterations access the same data? Suppose an access in this loop nest is represented by matrix-vector combination \mathbf{F} and \mathbf{f} . Let \mathbf{i} and \mathbf{i}' be two iterations that refer to the same array element. Then $\mathbf{F}\mathbf{i} + \mathbf{f} = \mathbf{F}\mathbf{i}' + \mathbf{f}$. Rearranging terms, we get

$$\mathbf{F}(\mathbf{i} - \mathbf{i}') = \mathbf{0}.$$

There is a well-known concept from linear algebra that characterizes when \mathbf{i} and \mathbf{i}' satisfy the above equation. The set of all solutions to the equation $\mathbf{F}\mathbf{v} = \mathbf{0}$ is called the *null space* of \mathbf{F} . Thus, two iterations refer to the same array element if the difference of their loop-index vectors belongs to the null space of matrix \mathbf{F} .

It is easy to see that the null vector, $\mathbf{v} = \mathbf{0}$, always satisfies $\mathbf{F}\mathbf{v} = \mathbf{0}$. That is, two iterations surely refer to the same array element if their difference is $\mathbf{0}$; in other words, if they are really the same iteration. Also, the null space is truly a vector space. That is, if $\mathbf{F}\mathbf{v}_1 = \mathbf{0}$ and $\mathbf{F}\mathbf{v}_2 = \mathbf{0}$, then $\mathbf{F}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{0}$ and $\mathbf{F}(c\mathbf{v}_1) = \mathbf{0}$.

If the matrix \mathbf{F} is *fully ranked*, that is, its rank is d , then the null space of \mathbf{F} consists of only the null vector. In that case, iterations in a loop nest all refer to different data. In general, the dimension of the null space, also known as the *nullity*, is $d - r$. If $d > r$, then for each element there is a $(d - r)$ -dimensional space of iterations that access that element.

The null space can be represented by its basis vectors. A k -dimensional null space is represented by k independent vectors; any vector that can be expressed as a linear combination of the basis vectors belongs to the null space.

Example 11.23: Let us reconsider the matrix of Example 11.21:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

We determined in that example that the rank of the matrix is 2; thus the nullity is $3 - 2 = 1$. To find a basis for the null space, which in this case must be a single nonzero vector of length 3, we may suppose a vector in the null space to

be $[x, y, z]$ and try to solve the equation

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

If we multiply the first two rows by the vector of unknowns, we get the two equations

$$\begin{aligned} x + 2y + 3z &= 0 \\ 5x + 7y + 9z &= 0 \end{aligned}$$

We could write the equations that come from the third and fourth rows as well, but because there are no three linearly independent rows, we know that the additional equations add no new constraints on x , y , and z . For instance, the equation we get from the third row, $4x + 5y + 6z = 0$ can be obtained by subtracting the first equation from the second.

We must eliminate as many variables as we can from the above equations. Start by using the first equation to solve for x ; that is, $x = -2y - 3z$. Then substitute for x in the second equation, to get $-3y = 6z$, or $y = -2z$. Since $x = -2y - 3z$, and $y = -2z$, it follows that $x = z$. Thus, the vector $[x, y, z]$ is really $[z, -2z, z]$. We may pick any nonzero value of z to form the one and only basis vector for the null space. For example, we may choose $z = 1$ and use $[1, -2, 1]$ as the basis of the null space. \square

Example 11.24: The rank, nullity, and null space for each of the references in Example 11.17 are shown in Fig. 11.19. Observe that the sum of the rank and nullity in all the cases is the depth of the loop nest, 2. Since the accesses $Y[i, j]$ and $Z[1, i, 2 * i + j]$ have a rank of 2, all iterations refer to different locations.

Accesses $X[i-1]$ and $Y[j, j+1]$ both have rank-1 matrices, so $O(n)$ iterations refer to the same location. In the former case, entire rows in the iteration space refer to the same location. In other words, iterations that differ only in the j dimension share the same location, which is succinctly represented by the basis of the null space, $[0, 1]$. For $Y[j, j+1]$, entire columns in the iteration space refer to the same location, and this fact is succinctly represented by the basis of the null space, $[1, 0]$.

Finally, the access $Y[1, 2]$ refers to the same location in all the iterations. The null space corresponding has 2 basis vectors, $[0, 1]$, $[1, 0]$, meaning that all pairs of iterations in the loop nest refer to exactly the same location. \square

11.5.3 Self-Spatial Reuse

The analysis of spatial reuse depends on the data layout of the matrix. C matrices are laid out in row-major order and Fortran matrices are laid out in column-major order. In other words, array elements $X[i, j]$ and $X[i, j+1]$ are

ACCESS	AFFINE EXPRESSION	RANK	NULL- ITY	BASIS OF NULL SPACE
$x[i-1]$	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$	1	1	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$y[i,j]$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	2	0	
$y[j,j+1]$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	1	1	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
$y[1,2]$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$	0	2	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$z[1,i,2*i+j]$	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	2	0	

Figure 11.19: Rank and nullity of affine accesses

contiguous in C and $X[i, j]$ and $X[i + 1, j]$ are contiguous in Fortran. Without loss of generality, in the rest of the chapter, we shall adopt the C (row-major) array layout.

As a first approximation, we consider two array elements to share the same cache line if and only if they share the same row in a two-dimensional array. More generally, in an array of d dimensions, we take array elements to share a cache line if they differ only in the last dimension. Since for a typical array and cache, many array elements can fit in one cache line, there is significant speedup to be had by accessing an entire row in order, even though, strictly speaking, we occasionally have to wait to load a new cache line.

The trick to discovering and taking advantage of self-spatial reuse is to drop the last row from the coefficient matrix \mathbf{F} . If the resulting *truncated* matrix has rank that is less than the depth of the loop nest, then we can assure spatial locality by making sure that the innermost loop varies only the last coordinate of the array.

Example 11.25: Consider the last access, $Z[1, i, 2 * i + j]$, in Fig. 11.19. If we

delete the last row, we are left with the truncated matrix

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

The rank of this matrix is evidently 1, and since the loop nest has depth 2, there is the opportunity for spatial reuse. In this case, since j is the inner-loop index, the inner loop visits contiguous elements of the array Z stored in row-major order. Making i the inner-loop index will not yield spatial locality, since as i changes, both the second and third dimensions change. \square

The general rule for determining whether there is self-spatial reuse is as follows. As always, we assume that the loop indexes correspond to columns of the coefficient matrix in order, with the outermost loop first, and the innermost loop last. Then in order for there to be spatial reuse, the vector $[0, 0, \dots, 0, 1]$ must be in the null space of the truncated matrix. The reason is that if this vector is in the null space, then when we fix all loop indexes but the innermost one, we know that all dynamic accesses during one run through the inner loop vary in only the last array index. If the array is stored in row-major order, then these elements are all near one another, perhaps in the same cache line.

Example 11.26: Note that $[0, 1]$ (transposed as a column vector) is in the null space of the truncated matrix of Example 11.25. Thus, as mentioned there, we expect that with j as the inner-loop index, there will be spatial locality. On the other hand, if we reverse the order of the loops, so i is the inner loop, then the coefficient matrix becomes

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

Now, $[0, 1]$ is not in the null space of this matrix. Rather, the null space is generated by the basis vector $[1, 0]$. Thus, as we suggested in Example 11.25, we do not expect spatial locality if i is the inner loop.

We should observe, however, that the test for $[0, 0, \dots, 0, 1]$ being in the null space is not quite sufficient to assure spatial locality. For instance, suppose the access were not $Z[1, i, 2 * i + j]$ but $Z[1, i, 2 * i + 50 * j]$. Then, only every fiftieth element of Z would be accessed during one run of the inner loop, and we would not reuse a cache line unless it were long enough to hold more than 50 elements. \square

11.5.4 Group Reuse

We compute group reuse only among accesses in a loop sharing the same coefficient matrix. Given two dynamic accesses $\mathbf{Fi}_1 + \mathbf{f}_1$ and $\mathbf{Fi}_2 + \mathbf{f}_2$, reuse of the same data requires that

$$\mathbf{Fi}_1 + \mathbf{f}_1 = \mathbf{Fi}_2 + \mathbf{f}_2$$

or

$$\mathbf{F}(\mathbf{i}_1 - \mathbf{i}_2) = (\mathbf{f}_2 - \mathbf{f}_1).$$

Suppose \mathbf{v} is one solution to this equation. Then if \mathbf{w} is any vector in the null space of \mathbf{F} , $\mathbf{w} + \mathbf{v}$ is also a solution, and in fact those are all the solutions to the equation.

Example 11.27: The following 2-deep loop nest

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    Z[i,j] = Z[i-1,j];
```

has two array accesses, $Z[i, j]$ and $Z[i - 1, j]$. Observe that these two accesses are both characterized by the coefficient matrix

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

like the second access, $Y[i, j]$ in Fig. 11.19. This matrix has rank 2, so there is no self-temporal reuse.

However, each access exhibits self-spatial reuse. As described in Section 11.5.3, when we delete the bottom row of the matrix, we are left with only the top row, $[1, 0]$, which has rank 1. Since $[0, 1]$ is in the null space of this truncated matrix, we expect spatial reuse. As each incrementation of inner-loop index j increases the second array index by one, we in fact do access adjacent array elements, and will make maximum use of each cache line.

Although there is no self-temporal reuse for either access, observe that the two references $Z[i, j]$ and $Z[i - 1, j]$ access almost the same set of array elements. That is, there is group-temporal reuse because the data read by access $Z[i - 1, j]$ is the same as the data written by access $Z[i, j]$, except for the case $i = 1$. This simple pattern applies to the entire iteration space and can be exploited to improve data locality in the code. Formally, discounting the loop bounds, the two accesses $Z[i, j]$ and $Z[i - 1, j]$ refer to the same location in iterations (i_1, j_1) and (i_2, j_2) , respectively, provided

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

Rewriting the terms, we get

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

That is, $j_1 = j_2$ and $i_2 = i_1 + 1$.

Notice that the reuse occurs along the i -axis of the iteration space. That is, the iteration (i_2, j_2) occurs n iterations (of the inner loop) after the iteration

(i_1, j_1) . Thus, many iterations are executed before the data written is reused. This data may or may not still be in the cache. If the cache manages to hold two consecutive rows of matrix Z , then access $Z[i - 1, j]$ does not miss in the cache, and the total number of cache misses for the entire loop nest is n^2/c , where c is the number of elements per cache line. Otherwise, there will be twice as many misses, since both static accesses require a new cache line for each c dynamic accesses. \square

Example 11.28: Suppose there are two accesses

$$A[i, j, i + j] \text{ and } A[i + 1, j - 1, i + j]$$

in a 3-deep loop nest, with indexes i, j , and k , from the outer to the inner loop. Then two accesses $\mathbf{i}_1 = [i_1, j_1, k_1]$ and $\mathbf{i}_2 = [i_2, j_2, k_2]$ reuse the same element whenever

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}.$$

One solution to this equation for a vector $\mathbf{v} = [i_1 - i_2, j_1 - j_2, k_1 - k_2]$ is $\mathbf{v} = [1, -1, 0]$; that is, $i_1 = i_2 + 1$, $j_1 = j_2 - 1$, and $k_1 = k_2$.⁴ However, the null space of the matrix

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

is generated by the basis vector $[0, 0, 1]$; that is, the third loop index, k , can be arbitrary. Thus, \mathbf{v} , the solution to the above equation, is any vector $[1, -1, m]$ for some m . Put another way, a dynamic access to $A[i, j, i + j]$, in a loop nest with indexes i, j , and k , is reused not only by other dynamic accesses $A[i, j, i + j]$ with the same values of i and j and a different value of k , but also by dynamic accesses $A[i + 1, j - 1, i + j]$ with loop index values $i + 1$, $j - 1$, and any value of k . \square

Although we shall not do so here, we can reason about group-spatial reuse analogously. As per the discussion of self-spatial reuse, we simply drop the last dimension from consideration.

The extent of reuse is different for the different categories of reuse. Self-temporal reuse gives the most benefit: a reference with a k -dimensional null space reuses the same data $O(n^k)$ times. The extent of self-spatial reuse is limited by the length of the cache line. Finally, the extent of group reuse is limited by the number of references in a group sharing the reuse.

⁴It is interesting to observe that, although there is a solution in this case, there would be no solution if we changed one of the third components from $i + j$ to $i + j + 1$. That is, in the example as given, both accesses touch those array elements that lie in the 2-dimensional subspace S defined by “the third component is the sum of the first two components.” If we changed $i + j$ to $i + j + 1$, none of the elements touched by the second access would lie in S , and there would be no reuse at all.

11.5.5 Exercises for Section 11.5

Exercise 11.5.1: Compute the ranks of each of the matrices in Fig. 11.20. Give both a maximal set of linearly independent columns and a maximal set of linearly independent rows.

$$\begin{array}{ccc}
 \begin{bmatrix} 0 & 1 & 5 \\ 1 & 2 & 6 \\ 2 & 3 & 7 \\ 3 & 4 & 8 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 12 & 15 \\ 3 & 2 & 2 & 3 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 5 & 6 & 3 \end{bmatrix} \\
 \text{(a)} & \text{(b)} & \text{(c)}
 \end{array}$$

Figure 11.20: Compute the ranks and null spaces of these matrices

Exercise 11.5.2: Find a basis for the null space of each matrix in Fig. 11.20.

Exercise 11.5.3: Assume that the iteration space has dimensions (variables) i , j , and k . For each of the accesses below, describe the subspaces that refer to the following single elements of the array:

a) $A[i, j, i + j]$

b) $A[i, i + 1, i + 2]$

! c) $A[i, i, j + k]$

! d) $A[i - j, j - k, k - i]$

! **Exercise 11.5.4:** Suppose array A is stored in row-major order and accessed inside the following loop nest:

```

for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
    for (k = 0; k < 100; k++)
      <some access to A>

```

Indicate for each of the following accesses whether it is possible to rewrite the loops so that the access to A exhibits self-spatial reuse; that is, entire cache lines are used consecutively. Show how to rewrite the loops, if so. Note: the rewriting of the loops may involve both reordering and introduction of new loop indexes. However, you may not change the layout of the array, e.g., by changing it to column-major order. Also note: in general, reordering of loop indexes may be legal or illegal, depending on criteria we develop in the next section. However, in this case, where the effect of each access is simply to set an array element to 0, you do not have to worry about the effect of reordering loops as far as the semantics of the program is concerned.

a) $A[i+1, i+k, j] = 0$.

!! b) $A[j+k, i, i] = 0$.

c) $A[i, j, k, i+j+k] = 0$.

!! d) $A[i, j-k, i+j, i+k] = 0$.

Exercise 11.5.5: In Section 11.5.3 we commented that we get spatial locality if the innermost loop varies only as the last coordinate of an array access. However, that assertion depended on our assumption that the array was stored in row-major order. What condition would assure spatial locality if the array were stored in column-major order?

! Exercise 11.5.6: In Example 11.28 we observed that the existence of reuse between two similar accesses depended heavily on the particular expressions for the coordinates of the array. Generalize our observation there to determine for which functions $f(i, j)$ there is reuse between the accesses $A[i, j, i+j]$ and $A[i+1, j-1, f(i, j)]$.

! Exercise 11.5.7: In Example 11.27 we suggested that there will be more cache misses than necessary if rows of the matrix Z are so long that they do not fit in the cache. If that is the case, how could you rewrite the loop nest in order to guarantee group-spatial reuse?

11.6 Array Data-Dependence Analysis

Parallelization or locality optimizations frequently reorder the operations executed in the original program. As with all optimizations, operations can be reordered only if the reordering does not change the program's output. Since we cannot, in general, understand deeply what a program does, code optimization generally adopts a simpler, conservative test for when we can be sure that the program output is not affected: we check that the operations on any memory location are done in the same order in the original and modified programs. In the present study, we focus on array accesses, so the array elements are the memory locations of concern.

Two accesses, whether read or write, are clearly *independent* (can be reordered) if they refer to two different locations. In addition, read operations do not change the memory state and therefore are also independent. Following Section 11.5, we say that two accesses are *data dependent* if they refer to the same memory location and at least one of them is a write operation. To be sure that the modified program does the same as the original, the relative execution ordering between every pair of data-dependent operations in the original program must be preserved in the new program.

Recall from Section 10.2.1 that there are three flavors of data dependence:

1. *True dependence*, where a write is followed by a read of the same location.

2. *Antidependence*, where a read is followed by a write to the same location.
3. *Output dependence*, which is two writes to the same location.

In the discussion above, data dependence is defined for dynamic accesses. We say that a static access in a program depends on another as long as there exists a dynamic instance of the first access that depends on some instance of the second.⁵

It is easy to see how data dependence can be used in parallelization. For example, if no data dependences are found in the accesses of a loop, we can easily assign each iteration to a different processor. Section 11.7 discusses how we use this information systematically in parallelization.

11.6.1 Definition of Data Dependence of Array Accesses

Let us consider two static accesses to the same array in possibly different loops. The first is represented by access function and bounds $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$ and is in a d -deep loop nest; the second is represented by $\mathcal{F}' = \langle \mathbf{F}', \mathbf{f}', \mathbf{B}', \mathbf{b}' \rangle$ and is in a d' -deep loop nest. These accesses are data dependent if

1. At least one of them is a write reference and
2. There exist vectors \mathbf{i} in Z^d and \mathbf{i}' in $Z^{d'}$ such that
 - (a) $\mathbf{B}\mathbf{i} \geq \mathbf{0}$,
 - (b) $\mathbf{B}'\mathbf{i}' \geq \mathbf{0}$, and
 - (c) $\mathbf{F}\mathbf{i} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

Since a static access normally embodies many dynamic accesses, it is also meaningful to ask if its dynamic accesses may refer to the same memory location. To search for dependencies between instances of the same static access, we assume $\mathcal{F} = \mathcal{F}'$ and augment the definition above with the additional constraint that $\mathbf{i} \neq \mathbf{i}'$ to rule out the trivial solution.

Example 11.29: Consider the following 1-deep loop nest:

```
for (i = 1; i <= 10; i++) {
    Z[i] = Z[i-1];
}
```

This loop has two accesses: $Z[i-1]$ and $Z[i]$; the first is a read reference and the second a write. To find all the data dependences in this program, we need to check if the write reference shares a dependence with itself and with the read reference:

⁵Recall the difference between static and dynamic accesses. A static access is an array reference at a particular location in a program, while a dynamic access is one execution of that reference.

1. *Data dependence between $Z[i - 1]$ and $Z[i]$.* Except for the first iteration, each iteration reads the value written in the previous iteration. Mathematically, we know that there is a dependence because there exist integers i and i' such that

$$1 \leq i \leq 10, 1 \leq i' \leq 10, \text{ and } i - 1 = i'.$$

There are nine solutions to the above system of constraints: $(i = 2, i' = 1)$, $(i = 3, i' = 2)$, and so forth.

2. *Data dependence between $Z[i]$ and itself.* It is easy to see that different iterations in the loop write to different locations; that is, there are no data dependencies among the instances of the write reference $Z[i]$. Mathematically, we know that there does not exist a dependence because there do not exist integers i and i' satisfying

$$1 \leq i \leq 10, 1 \leq i' \leq 10, i = i', \text{ and } i \neq i'.$$

Notice that the third condition, $i = i'$, comes from the requirement that $Z[i]$ and $Z[i']$ are the same memory location. The contradictory fourth condition, $i \neq i'$, comes from the requirement that the dependence be nontrivial — between different dynamic accesses.

It is not necessary to consider data dependences between the read reference $Z[i - 1]$ and itself because any two read accesses are independent. \square

11.6.2 Integer Linear Programming

Data dependence requires finding whether there exist integers that satisfy a system consisting of equalities and inequalities. The equalities are derived from the matrices and vectors representing the accesses; the inequalities are derived from the loop bounds. Equalities can be expressed as inequalities: an equality $x = y$ can be replaced by two inequalities, $x \geq y$ and $y \geq x$.

Thus, data dependence may be phrased as a search for integer solutions that satisfy a set of linear inequalities, which is precisely the well-known problem of *integer linear programming*. Integer linear programming is an NP-complete problem. While no polynomial algorithm is known, heuristics have been developed to solve linear programs involving many variables, and they can be quite fast in many cases. Unfortunately, such standard heuristics are inappropriate for data dependence analysis, where the challenge is to solve many small and simple integer linear programs rather than large complicated integer linear programs.

The data dependence analysis algorithm consists of three parts:

1. Apply the GCD (Greatest Common Divisor) test, which checks if there is an integer solution to the equalities, using the theory of linear Diophantine equations. If there are no integer solutions, then there are no data dependences. Otherwise, we use the equalities to substitute for some of the variables thereby obtaining simpler inequalities.
2. Use a set of simple heuristics to handle the large numbers of typical inequalities.
3. In the rare case where the heuristics do not work, we use a linear integer programming solver that uses a branch-and-bound approach based on Fourier-Motzkin elimination.

11.6.3 The GCD Test

The first subproblem is to check for the existence of integer solutions to the equalities. Equations with the stipulation that solutions must be integers are known as *Diophantine equations*. The following example shows how the issue of integer solutions arises; it also demonstrates that even though many of our examples involve a single loop nest at a time, the data dependence formulation applies to accesses in possibly different loops.

Example 11.30: Consider the following code fragment:

```
for (i = 1; i < 10; i++) {
    Z[2*i] = 10;
}
for (j = 1; j < 10; j++) {
    Z[2*j+1] = 20;
}
```

The access $Z[2 * i]$ only touches even elements of Z , while access $Z[2 * j + 1]$ touches only odd elements. Clearly, these two accesses share no data dependence regardless of the loop bounds. We can execute iterations in the second loop before the first, or interleave the iterations. This example is not as contrived as it may look. An example where even and odd numbers are treated differently is an array of complex numbers, where the real and imaginary components are laid out side by side.

To prove the absence of data dependences in this example, we reason as follows. Suppose there were integers i and j such that $Z[2 * i]$ and $Z[2 * j + 1]$ are the same array element. We get the Diophantine equation

$$2i = 2j + 1.$$

There are no integers i and j that can satisfy the above equation. The proof is that if i is an integer, then $2i$ is even. If j is an integer, then $2j$ is even, so $2j + 1$ is odd. No even number is also an odd number. Therefore, the equation

has no integer solutions, and thus there is no dependence between the read and write accesses. \square

To describe when there is a solution to a linear Diophantine equation, we need the concept of the *greatest common divisor* (GCD) of two or more integers. The GCD of integers a_1, a_2, \dots, a_n , denoted $\gcd(a_1, a_2, \dots, a_n)$, is the largest integer that evenly divides all these integers. GCD's can be computed efficiently by the well-known Euclidean algorithm (see the box on the subject).

Example 11.31: $\gcd(24, 36, 54) = 6$, because $24/6$, $36/6$, and $54/6$ each have remainder 0, yet any integer larger than 6 must leave a nonzero remainder when dividing at least one of 24, 36, and 54. For instance, 12 divides 24 and 36 evenly, but not 54. \square

The importance of the GCD is in the following theorem.

Theorem 11.32: The linear Diophantine equation

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

has an integer solution for x_1, x_2, \dots, x_n if and only if $\gcd(a_1, a_2, \dots, a_n)$ divides c . \square

Example 11.33: We observed in Example 11.30 that the linear Diophantine equation $2i = 2j + 1$ has no solution. We can write this equation as

$$2i - 2j = 1.$$

Now $\gcd(2, -2) = 2$, and 2 does not divide 1 evenly. Thus, there is no solution.

For another example, consider the equation

$$24x + 36y + 54z = 30.$$

Since $\gcd(24, 36, 54) = 6$, and $30/6 = 5$, there is a solution in integers for x , y , and z . One solution is $x = -1$, $y = 0$, and $z = 1$, but there are an infinity of other solutions. \square

The first step to the data dependence problem is to use a standard method such as Gaussian elimination to solve the given equalities. Every time a linear equation is constructed, apply Theorem 11.32 to rule out, if possible, the existence of an integer solution. If we can rule out such solutions, then answer “no”. Otherwise, we use the solution of the equations to reduce the number of variables in the inequalities.

Example 11.34: Consider the two equalities

$$\begin{array}{rcl} x - 2y + z & = & 0 \\ 3x + 2y + z & = & 5 \end{array}$$

The Euclidean Algorithm

The *Euclidean algorithm* for finding $\gcd(a, b)$ works as follows. First, assume that a and b are positive integers, and $a \geq b$. Note that the GCD of negative numbers, or the GCD of a negative and a positive number is the same as the GCD of their absolute values, so we can assume all integers are positive.

If $a = b$, then $\gcd(a, b) = a$. If $a > b$, let c be the remainder of a/b . If $c = 0$, then b evenly divides a , so $\gcd(a, b) = b$. Otherwise, compute $\gcd(b, c)$; this result will also be $\gcd(a, b)$.

To compute $\gcd(a_1, a_2, \dots, a_n)$, for $n > 2$, use the Euclidean algorithm to compute $\gcd(a_1, a_2) = c$. Then recursively compute $\gcd(c, a_3, a_4, \dots, a_n)$.

Looking at each equality by itself, it appears there might be a solution. For the first equality, $\gcd(1, -2, 1) = 1$ divides 0, and for the second equality, $\gcd(3, 2, 1) = 1$ divides 5. However, if we use the first equality to solve for $z = 2y - x$ and substitute for z in the second equality, we get $2x + 4y = 5$. This Diophantine equation has no solution, since $\gcd(2, 4) = 2$ does not divide 5 evenly. \square

11.6.4 Heuristics for Solving Integer Linear Programs

The data dependence problem requires many simple integer linear programs be solved. We now discuss several techniques to handle simple inequalities and a technique to take advantage of the similarity found in data dependence analysis.

Independent-Variables Test

Many of the integer linear programs from data dependence consist of inequalities that involve only one unknown. The programs can be solved simply by testing if there are integers between the constant upper bounds and constant lower bounds independently.

Example 11.35: Consider the nested loop

```
for (i = 0; i <= 10; i++)
  for (j = 0; j <= 10; j++)
    Z[i,j] = Z[j+10,i+11];
```

To find if there is a data dependence between $Z[i, j]$ and $Z[j + 10, i + 11]$, we ask if there exist integers i, j, i' , and j' such that

$$\begin{aligned} 0 &\leq i, j, i', j' \leq 10 \\ i &= j' + 10 \\ j &= i' + 11 \end{aligned}$$

The GCD test, applied to the two equalities above, will determine that there *may* be an integer solution. The integer solutions to the equalities are expressed by

$$i = t_1, j = t_2, i' = t_2 - 11, j' = t_1 - 10$$

for any integers t_1 and t_2 . Substituting the variables t_1 and t_2 into the linear inequalities, we get

$$\begin{aligned} 0 &\leq t_1 && \leq 10 \\ 0 &\leq t_2 && \leq 10 \\ 0 &\leq t_2 - 11 && \leq 10 \\ 0 &\leq t_1 - 10 && \leq 10 \end{aligned}$$

Thus, combining the lower bounds from the last two inequalities with the upper bounds from the first two, we deduce

$$\begin{aligned} 10 &\leq t_1 \leq 10 \\ 11 &\leq t_2 \leq 10 \end{aligned}$$

Since the lower bound on t_2 is greater than its upper bound, there is no integer solution, and hence no data dependence. This example shows that even if there are equalities involving several variables, the GCD test may still create linear inequalities that involve one variable at a time. \square

Acyclic Test

Another simple heuristic is to find if there exists a variable that is bounded below or above by a constant. In certain circumstances, we can safely replace the variable by the constant; the simplified inequalities have a solution if and only if the original inequalities have a solution. Specifically, suppose every lower bound on v_i is of the form

$$c_0 \leq c_i v_i \text{ for some } c_i > 0$$

while the upper bounds on v_i are all of the form

$$c_i v_i \leq c_0 + c_1 v_1 + \dots + c_{i-1} v_{i-1} + c_{i+1} v_{i+1} + \dots + c_n v_n$$

where c_i is nonnegative. Then we can replace variable v_i by its smallest possible integer value. If there is no such lower bound, we simply replace v_i with $-\infty$.

Similarly, if every constraint involving v_i can be expressed in the two forms above, but with the directions of the inequalities reversed, then we can replace variable v_i with the largest possible integer value, or by ∞ if there is no constant upper bound. This step can be repeated to simplify the inequalities and in some cases determine if there is a solution.

Example 11.36: Consider the following inequalities:

$$\begin{array}{rclcl} 1 & \leq & v_1, v_2 & \leq & 10 \\ 0 & \leq & v_3 & \leq & 4 \\ v_2 & \leq & v_1 & & \\ & & v_1 & \leq & v_3 + 4 \end{array}$$

Variable v_1 is bounded from below by v_2 and from above by $v_3 + 4$. However, v_2 is bounded from below only by the constant 1, and v_3 is bounded from above only by the constant 4. Thus, replacing v_2 by 1 and v_3 by 4 in the inequalities, we obtain

$$\begin{array}{rclcl} 1 & \leq & v_1 & \leq & 10 \\ 1 & \leq & v_1 & & \\ & & v_1 & \leq & 8 \end{array}$$

which can now be solved easily with the independent-variables test. \square

The Loop-Residue Test

Let us now consider the case where every variable is bounded from below and above by other variables. It is commonly the case in data dependence analysis that constraints have the form $v_i \leq v_j + c$, which can be solved using a simplified version of the *loop-residue test* due to Shostak. A set of these constraints can be represented by a directed graph whose nodes are labeled with variables. There is an edge from v_i to v_j labeled c whenever there is a constraint $v_i \leq v_j + c$.

We define the *weight* of a path to be the sum of the labels of all the edges along the path. Each path in the graph represents a combination of the constraints in the system. That is, we can infer that $v \leq v' + c$ whenever there exists a path from v to v' with weight c . A cycle in the graph with weight c represents the constraint $v \leq v + c$ for each node v on the cycle. If we can find a negatively weighted cycle in the graph, then we can infer $v < v$, which is impossible. In this case, we can conclude that there is no solution and thus no dependence.

We can also incorporate into the loop-residue test constraints of the form $c \leq v$ and $v \leq c$ for variable v and constant c . We introduce into the system of inequalities a new dummy variable v_0 , which is added to each constant upper and lower bound. Of course, v_0 must have value 0, but since the loop-residue test only looks for cycles, the actual values of the variables never becomes significant. To handle constant bounds, we replace

$$\begin{aligned} v &\leq c \text{ by } v \leq v_0 + c \\ c &\leq v \text{ by } v_0 \leq v - c. \end{aligned}$$

Example 11.37: Consider the inequalities

$$\begin{aligned} 1 &\leq v_1, v_2 \leq 10 \\ 0 &\leq v_3 \leq 4 \\ v_2 &\leq v_1 \\ 2v_1 &\leq 2v_3 - 7 \end{aligned}$$

The constant upper and lower bounds on v_1 become $v_0 \leq v_1 - 1$ and $v_1 \leq v_0 + 10$; the constant bounds on v_2 and v_3 are handled similarly. Then, converting the last constraint to $v_1 \leq v_3 - 4$, we can create the graph shown in Fig. 11.21. The cycle v_1, v_3, v_0, v_1 has weight -1 , so there is no solution to this set of inequalities. \square

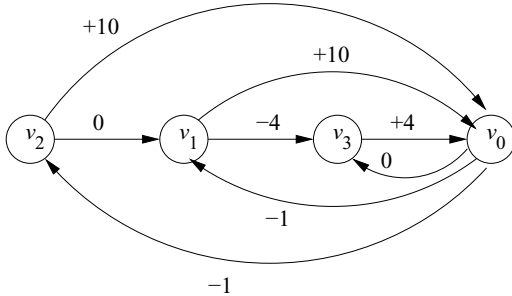


Figure 11.21: Graph for the constraints of Example 11.37

Memoization

Often, similar data dependence problems are solved repeatedly, because simple access patterns are repeated throughout the program. One important technique to speed up data dependence processing is to use *memoization*. Memoization tabulates the results to the problems as they are generated. The table of stored solutions is consulted as each problem is presented; the problem needs to be solved only if the result to the problem cannot be found in the table.

11.6.5 Solving General Integer Linear Programs

We now describe a general approach to solving the integer linear programming problem. The problem is NP-complete; our algorithm uses a branch-and-bound approach that can take an exponential amount of time in the worst case. However, it is rare that the heuristics of Section 11.6.4 cannot resolve the problem, and even if we do need to apply the algorithm of this section, it seldom needs to perform the branch-and-bound step.

The approach is to first check for the existence of rational solutions to the inequalities. This problem is the classical linear-programming problem. If there is no rational solution to the inequalities, then the regions of data touched by the accesses in question do not overlap, and there surely is no data dependence. If there is a rational solution, we first try to prove that there is an integer solution, which is commonly the case. Failing that, we then split the polyhedron bounded by the inequalities into two smaller problems and recurse.

Example 11.38: Consider the following simple loop:

```
for (i = 1; i < 10; i++)
    Z[i] = Z[i+10];
```

The elements touched by access $Z[i]$ are $Z[1], \dots, Z[9]$, while the elements touched by $Z[i + 10]$ are $Z[11], \dots, Z[19]$. The ranges do not overlap and therefore there are no data dependences. More formally, we need to show that there are no two dynamic accesses i and i' , with $1 \leq i \leq 9$, $1 \leq i' \leq 9$, and $i = i' + 10$. If there were such integers i and i' , then we could substitute $i' + 10$ for i and get the four constraints on i' : $1 \leq i' \leq 9$ and $1 \leq i' + 10 \leq 9$. However, $i' + 10 \leq 9$ implies $i' \leq -1$, which contradicts $1 \leq i'$. Thus, no such integers i and i' exist. \square

Algorithm 11.39 describes how to determine if an integer solution can be found for a set of linear inequalities based on the Fourier-Motzkin elimination algorithm.

Algorithm 11.39: Branch-and-bound solution to integer linear programming problems.

INPUT: A convex polyhedron S_n over variables v_1, \dots, v_n .

OUTPUT: “yes” if S_n has an integer solution, “no” otherwise.

METHOD: The algorithm is shown in Fig. 11.22. \square

Lines (1) through (3) attempt to find a rational solution to the inequalities. If there is no rational solution, there is no integer solution. If a rational solution is found, this means that the inequalities define a nonempty polyhedron. It is relatively rare for such a polyhedron not to include any integer solutions — for that to happen, the polyhedron must be relatively thin along some dimension and fit between integer points.

Thus, lines (4) through (9) try to check quickly if there is an integer solution. Each step of the Fourier-Motzkin elimination algorithm produces a polyhedron with one fewer dimension than the previous one. We consider the polyhedra in reverse order. We start with the polyhedron with one variable and assign to that variable an integer solution roughly in the middle of the range of possible values if possible. We then substitute the value for the variable in all other polyhedra, decreasing their unknown variables by one. We repeat the same process until

- 1) apply Algorithm 11.11 to S_n to project away variables v_n, v_{n-1}, \dots, v_1 in that order;
- 2) let S_i be the polyhedron after projecting away v_{i+1} , for $i = n-1, n-2, \dots, 0$;
- 3) **if** S_0 is empty **return** “no”;
/* There is no rational solution if S_0 , which involves only constants, has unsatisfiable constraints */
- 4) **for** ($i = 1; i \leq n; i++$) {
- 5) **if** (S_i does not include an integer value) **break**;
- 6) pick c_i , an integer in the middle of the range for v_i in S_i ;
- 7) modify S_i by replacing v_i by c_i ;
- 8) }
9) **if** ($i == n+1$) **return** “yes”;
- 10) **if** ($i == 1$) **return** “no”;
- 11) let the lower and upper bounds on v_i in S_i be l_i and u_i , respectively;
- 12) recursively apply this algorithm to $S_n \cup \{v_i \leq \lfloor l_i \rfloor\}$ and $S_n \cup \{v_i \geq \lceil u_i \rceil\}$;
- 13) **if** (either returns “yes”) **return** “yes” **else return** “no”;

Figure 11.22: Finding an integer solution in inequalities

we have processed all the polyhedra, in which case an integer solution is found, or we have found a variable for which there is no integer solution.

If we cannot find an integer value for even the first variable, there is no integer solution (line 10). Otherwise, all we know is that there is no integer solution including the combination of specific integers we have picked so far, and the result is inconclusive. Lines (11) through (13) represent the branch-and-bound step. If variable v_i is found to have a rational but not integer solution, we split the polyhedron into two with the first requiring that v_i must be an integer smaller than the rational solution found, and the second requiring that v_i must be an integer greater than the rational solution found. If neither has a solution, then there is no dependence.

11.6.6 Summary

We have shown that essential pieces of information that a compiler can glean from array references are equivalent to certain standard mathematical concepts. Given an access function $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$:

1. The dimension of the data region accessed is given by the rank of the matrix \mathbf{F} . The dimension of the space of accesses to the same location is given by the nullity of \mathbf{F} . Iterations whose differences belong to the null space of \mathbf{F} refer to the same array elements.

2. Iterations that share self-temporal reuse of an access are separated by vectors in the null space of \mathbf{F} . Self-spatial reuse can be computed similarly by asking when two iterations use the same row, rather than the same element. Two accesses $\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1$ and $\mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$ share easily exploitable locality along the \mathbf{d} direction, if \mathbf{d} is the particular solution to the equation $\mathbf{F}\mathbf{d} = (\mathbf{f}_1 - \mathbf{f}_2)$. In particular, if \mathbf{d} is the direction corresponding to the innermost loop, i.e., the vector $[0, 0, \dots, 0, 1]$, then there is spatial locality if the array is stored in row-major form.
3. The data dependence problem — whether two references can refer to the same location — is equivalent to integer linear programming. Two access functions share a data dependence if there are integer-valued vectors \mathbf{i} and \mathbf{i}' such that $\mathbf{B}\mathbf{i} \geq \mathbf{0}$, $\mathbf{B}'\mathbf{i}' \geq \mathbf{0}$, and $\mathbf{F}\mathbf{i} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

11.6.7 Exercises for Section 11.6

Exercise 11.6.1: Find the GCD's of the following sets of integers:

- a) $\{16, 24, 56\}$.
- b) $\{-45, 105, 240\}$.
- ! c) $\{84, 105, 180, 315, 350\}$.

Exercise 11.6.2: For the following loop

```
for (i = 0; i < 10; i++)
    A[i] = A[10-i];
```

indicate all the

- a) True dependences (write followed by read of the same location).
- b) Antidependences (read followed by write to the same location).
- c) Output dependences (write followed by another write to the same location).

! Exercise 11.6.3: In the box on the Euclidean algorithm, we made a number of assertions without proof. Prove each of the following:

- a) The Euclidean algorithm as stated always works. In particular, $\gcd(b, c) = \gcd(a, b)$, where c is the nonzero remainder of a/b .
- b) $\gcd(a, b) = \gcd(a, -b)$.
- c) $\gcd(a_1, a_2, \dots, a_n) = \gcd(\gcd(a_1, a_2), a_3, a_4, \dots, a_n)$ for $n > 2$.

- d) The GCD is really a function on sets of integers; i.e., order doesn't matter. Show the *commutative law* for GCD: $\gcd(a, b) = \gcd(b, a)$. Then, show the more difficult statement, the *associative law* for GCD: $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$. Finally, show that together these laws imply that the GCD of a set of integers is the same, regardless of the order in which the GCD's of pairs of integers are computed.
- e) If S and T are sets of integers, then $\gcd(S \cup T) = \gcd(\gcd(S), \gcd(T))$.

! Exercise 11.6.4: Find another solution to the second Diophantine equation in Example 11.33.

Exercise 11.6.5: Apply the independent-variables test in the following situation. The loop nest is

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    for (k=0; k<100; k++)
```

and inside the nest is an assignment involving array accesses. Determine if there are any data dependences due to each of the following statements:

- a) $A[i, j, k] = A[i+100, j+100, k+100]$.
- b) $A[i, j, k] = A[j+100, k+100, i+100]$.
- c) $A[i, j, k] = A[j-50, k-50, i-50]$.
- d) $A[i, j, k] = A[i+99, k+100, j]$.

Exercise 11.6.6: In the two constraints

$$\begin{array}{rcl} 1 & \leq & x \leq y - 100 \\ 3 & \leq & x \leq 2y - 50 \end{array}$$

eliminate x by replacing it by a constant lower bound on y .

Exercise 11.6.7: Apply the loop-residue test to the following set of constraints:

$$\begin{array}{l} 0 \leq x \leq 99 \quad y \leq x - 50 \\ 0 \leq y \leq 99 \quad z \leq y - 60 \\ 0 \leq z \leq 99 \end{array}$$

Exercise 11.6.8: Apply the loop-residue test to the following set of constraints:

$$\begin{array}{l} 0 \leq x \leq 99 \quad y \leq x - 50 \\ 0 \leq y \leq 99 \quad z \leq y + 40 \\ 0 \leq z \leq 99 \quad x \leq z + 20 \end{array}$$

Exercise 11.6.9: Apply the loop-residue test to the following set of constraints:

$$\begin{array}{ll} 0 \leq x \leq 99 & y \leq x - 100 \\ 0 \leq y \leq 99 & z \leq y + 60 \\ 0 \leq z \leq 99 & x \leq z + 50 \end{array}$$

11.7 Finding Synchronization-Free Parallelism

Having developed the theory of affine array accesses, their reuse of data, and the dependences among them, we shall now begin to apply this theory to parallelization and optimization of real programs. As discussed in Section 11.1.4, it is important that we find parallelism while minimizing communication among processors. Let us start by studying the problem of parallelizing an application without allowing any communication or synchronization between processors at all. This constraint may appear to be a purely academic exercise; how often can we find programs and routines that have such a form of parallelism? In fact, many such programs exist in real life, and the algorithm for solving this problem is useful in its own right. In addition, the concepts used to solve this problem can be extended to handle synchronization and communication.

11.7.1 An Introductory Example

Shown in Fig. 11.23 is an excerpt of a C translation (with Fortran-style array accesses retained for clarity) from a 5000-line Fortran multigrid algorithm to solve three-dimensional Euler equations. The program spends most its time in a small number of routines like the one shown in the figure. It is typical of many numerical programs. These often consist of numerous for-loops, with different nesting levels, and they have many array accesses, all of which are affine expressions of surrounding loop indexes. To keep the example short, we have elided lines from the original program with similar characteristics.

The code of Fig. 11.23 operates on the scalar variable T and a number of different arrays with different dimensions. Let us first examine the use of variable T . Because each iteration in the loop uses the same variable T , we cannot execute the iterations in parallel. However, T is used only as a way to hold a common subexpression used twice in the same iteration. In the first two of the three loop nests in Fig. 11.23, each iteration of the innermost loop writes a value into T and uses the value immediately after twice, in the same iteration. We can eliminate the dependences by replacing each use of T by the right-hand-side expression in the previous assignment of T , without changing the semantics of the program. Or, we can replace the scalar T by an array. We then have each iteration (j, i) use its own array element $T[j, i]$.

With this modification, the computation of an array element in each assignment statement depends only on other array elements with the same values for the last two components (j and i , respectively). We can thus group all

```

for (j = 2; j <= jl; j++)
  for (i = 2, i <= il, i++) {
    AP[j,i]      = ...;
    T            = 1.0/(1.0 + AP[j,i]);
    D[2,j,i]     = T*AP[j,i];
    DW[1,2,j,i]  = T*DW[1,2,j,i];
  }
for (k = 3; k <= kl-1; k++)
  for (j = 2; j <= jl; j++)
    for (i = 2; i <= il; i++) {
      AM[j,i]    = AP[j,i];
      AP[j,i]    = ...;
      T          = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...;
      D[k,j,i]   = T*AP[j,i];
      DW[1,k,j,i] = T*(DW[1,k,j,i] + DW[1,k-1,j,i])...;
    }
...
for (k = kl-1; k >= 2; k--)
  for (j = 2; j <= jl; j++)
    for (i = 2; i <= il; i++)
      DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];

```

Figure 11.23: Code excerpt of a multigrid algorithm

operations that operate on the (j, i) th element of all arrays into one computation unit, and execute them in the original sequential order. This modification produces $(j_l - 1) \times (i_l - 1)$ units of computation that are all independent of one another. Notice that second and third nests in the source program involve a third loop, with index k . However, because there is no dependence between dynamic accesses with the same values for j and i , we can safely perform the loops on k inside the loops on j and i — that is, within a computation unit.

Knowing that these computation units are independent enables a number of legal transforms on this code. For example, instead of executing the code as originally written, a uniprocessor can perform the same computation by executing the units of independent operation one unit at a time. The resulting code, shown in Fig. 11.24, has improved temporal locality, because results produced are consumed immediately.

The independent units of computation can also be assigned to different processors and executed in parallel, without requiring any synchronization or communication. Since there are $(j_l - 1) \times (i_l - 1)$ independent units of computation, we can utilize at most $(j_l - 1) \times (i_l - 1)$ processors. By organizing the processors as if they were in a 2-dimensional array, with ID's (j, i) , where $2 \leq j < j_l$ and $2 \leq i < i_l$, the SPMD program to be executed by each processor is simply the body in the inner loop in Fig. 11.24.

```

for (j = 2; j <= jl; j++)
  for (i = 2; i <= il; i++) {
    AP[j,i]      = ...;
    T[j,i]       = 1.0/(1.0 + AP[j,i]);
    D[2,j,i]     = T[j,i]*AP[j,i];
    DW[1,2,j,i]  = T[j,i]*DW[1,2,j,i];
    for (k = 3; k <= kl-1; k++) {
      AM[j,i]    = AP[j,i];
      AP[j,i]    = ...;
      T[j,i]     = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...;
      D[k,j,i]   = T[j,i]*AP[j,i];
      DW[1,k,j,i] = T[j,i]*(DW[1,k,j,i] + DW[1,k-1,j,i])...;
    }
    ...
    for (k = kl-1; k >= 2; k--)
      DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
  }

```

Figure 11.24: Code of Fig. 11.23 transformed to carry outermost parallel loops

The above example illustrates the basic approach to finding synchronization-free parallelism. We first split the computation into as many independent units as possible. This partitioning exposes the scheduling choices available. We then assign computation units to the processors, depending on the number of processors we have. Finally, we generate an SPMD program that is executed on each processor.

11.7.2 Affine Space Partitions

A loop nest is said to have k degrees of parallelism if it has, within the nest, k parallelizable loops — that is, loops such that there are no data dependencies between different iterations of the loops. For example, the code in Fig. 11.24 has 2 degrees of parallelism. It is convenient to assign the operations in a computation with k degrees of parallelism to a processor array with k dimensions.

We shall assume initially that each dimension of the processor array has as many processors as there are iterations of the corresponding loop. After all the independent computation units have been found, we shall map these “virtual” processors to the actual processors. In practice, each processor should be responsible for a fairly large number of iterations, because otherwise there is not enough work to amortize away the overhead of parallelization.

We break down the program to be parallelized into elementary statements, such as 3-address statements. For each statement, we find an *affine space partition* that maps each dynamic instance of the statement, as identified by its loop indexes, to a processor ID.

Example 11.40: As discussed above, the code of Fig. 11.24 has two degrees of parallelism. We view the processor array as a 2-dimensional space. Let (p_1, p_2) be the ID of a processor in the array. The parallelization scheme discussed in Section 11.7.1 can be described by simple affine partition functions. All the statements in the first loop nest have this same affine partition:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

All the statements in the second and third loop nests have the following same affine partition:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

□

The algorithm to find synchronization-free parallelism consists of three steps:

1. Find, for each statement in the program, an affine partition that maximizes the degree of parallelism. Note that we generally treat the statement, rather than the single access, as the unit of computation. The same affine partition must apply to each access in the statement. This grouping of accesses makes sense, since there is almost always dependence among accesses of the same statement anyway.
2. Assign the resulting independent computation units among the processors, and choose an interleaving of the steps on each processor. This assignment is driven by locality considerations.
3. Generate an SPMD program to be executed on each processor.

We shall discuss next how to find the affine partition functions, how to generate a sequential program that executes the partitions serially, and how to generate an SPMD program that executes each partition on a different processor. After we discuss how parallelism with synchronizations is handled in Sections 11.8 through 11.9.9, we return to Step 2 above in Section 11.10 and discuss the optimization of locality for uniprocessors and multiprocessors.

11.7.3 Space-Partition Constraints

To require no communication, each pair of operations that share a data dependence must be assigned to the same processor. We refer to these constraints as “space-partition constraints.” Any mapping that satisfies these constraints creates partitions that are independent of one another. Note that such constraints can be satisfied by putting all the operations in a single partition. Unfortunately, that “solution” does not yield any parallelism. Our goal is to create

as many independent partitions as possible while satisfying the space-partition constraints; that is, operations are not placed on the same processor unless it is necessary.

When we restrict ourselves to affine partitions, then instead of maximizing the number of independent units, we may maximize the degree (number of dimensions) of parallelism. It is sometimes possible to create more independent units if we can use *piecewise* affine partitions. A piecewise affine partition divides instances of a single access into different sets and allows a different affine partition for each set. However, we shall not consider such an option here.

Formally, an affine partition of a program is *synchronization free* if and only if for every two (not necessarily distinct) accesses sharing a dependence, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ in statement s_1 nested in d_1 loops, and $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ in statement s_2 nested in d_2 loops, the partitions $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ and $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ for statements s_1 and s_2 , respectively, satisfy the following *space-partition constraints*:

- For all \mathbf{i}_1 in Z^{d_1} and \mathbf{i}_2 in Z^{d_2} such that
 - a) $\mathbf{B}_1 \mathbf{i}_1 + \mathbf{b}_1 \geq \mathbf{0}$,
 - b) $\mathbf{B}_2 \mathbf{i}_2 + \mathbf{b}_2 \geq \mathbf{0}$, and
 - c) $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$,

it is the case that $\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$.

The goal of the parallelization algorithm is to find, for each statement, the partition with the highest rank that satisfies these constraints.

Shown in Fig. 11.25 is a diagram illustrating the essence of the space-partition constraints. Suppose there are two static accesses in two loop nests with index vectors \mathbf{i}_1 and \mathbf{i}_2 . These accesses are dependent in the sense that they access at least one array element in common, and at least one of them is a write. The figure shows particular dynamic accesses in the two loops that happen to access the same array element, according to the affine access functions $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1$ and $\mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$. Synchronization is necessary unless the affine partitions for the two static accesses, $\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1$ and $\mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$, assign the dynamic accesses to the same processor.

If we choose an affine partition whose rank is the maximum of the ranks of all statements, we get the maximum possible parallelism. However, under this partitioning some processors may be idle at times, while other processors are executing statements whose affine partitions have a smaller rank. This situation may be acceptable if the time taken to execute those statements is relatively short. Otherwise, we can choose an affine partition whose rank is smaller than the maximum possible, as long as that rank is greater than 0.

We show in Example 11.41 a small program designed to illustrate the power of the technique. Real applications are usually much simpler than this, but may have boundary conditions resembling some of the issues shown here. We shall use this example throughout this chapter to illustrate that programs with

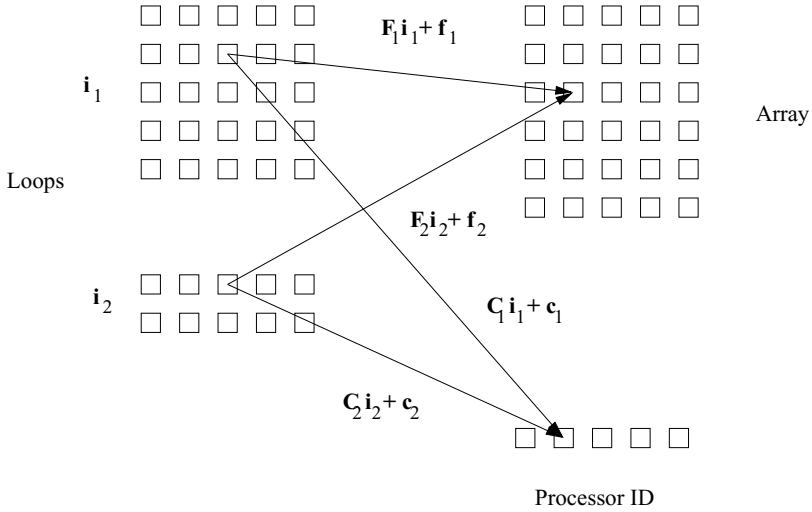


Figure 11.25: Space-partition constraints

affine accesses have relatively simple space-partition constraints, that these constraints can be solved using standard linear algebra techniques, and that the desired SPMD program can be generated mechanically from the affine partitions.

Example 11.41: This example shows how we formulate the space-partition constraints for the program consisting of the small loop nest with two statements, s_1 and s_2 , shown in Figure 11.26.

```

for (i = 1; i <= 100; i++)
  for (j = 1; j <= 100; j++) {
    X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
    Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
  }

```

Figure 11.26: A loop nest exhibiting long chains of dependent operations

We show the data dependences in the program in Figure 11.27. That is, each black dot represents an instance of statement s_1 , and each white dot represents an instance of statement s_2 . The dot located at coordinates (i, j) represents the instance of the statement that is executed for those values of the loop indexes. Note, however, that the instance of s_2 is located just below the instance of s_1 for the same (i, j) pair, so the vertical scale of j is greater than the horizontal scale of i .

Notice that $X[i, j]$ is written by $s_1(i, j)$, that is, by the instance of statement s_1 with index values i and j . It is later read by $s_2(i, j + 1)$, so $s_1(i, j)$ must

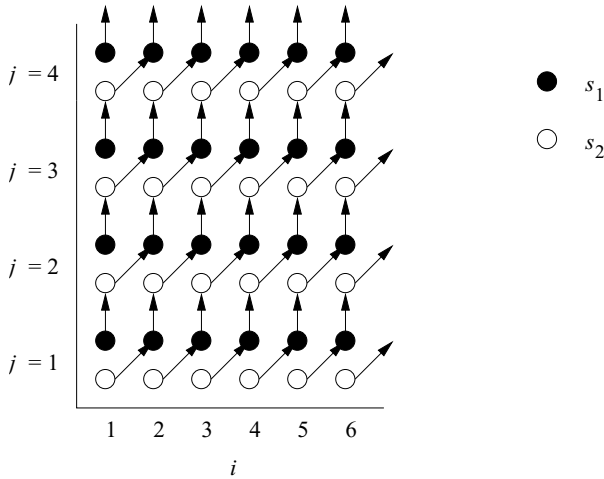


Figure 11.27: Dependences of the code in Example 11.41

precede $s_2(i, j + 1)$. This observation explains the vertical arrows from black dots to white dots. Similarly, $Y[i, j]$ is written by $s_2(i, j)$ and later read by $s_1(i + 1, j)$. Thus, $s_2(i, j)$ must precede $s_1(i + 1, j)$, which explains the arrows from white dots to black.

It is easy to see from this diagram that this code can be parallelized without synchronization by assigning each chain of dependent operations to the same processor. However, it is not easy to write the SPMD program that implements this mapping scheme. While the loops in the original program have 100 iterations each, there are 200 chains, with half originating and ending with statement s_1 and the other half originating and ending with s_2 . The lengths of the chains vary from 1 to 100 iterations.

Since there are two statements, we are seeking two affine partitions, one for each statement. We only need to express the space-partition constraints for one-dimensional affine partitions. These constraints will be used later by the solution method that tries to find all the independent one-dimensional affine partitions and combine them to get multidimensional affine partitions. We can thus represent the affine partition for each statement by a 1×2 matrix and a 1×1 vector to translate the vector of indexes $[i, j]$ into a single processor number. Let $\langle [C_{11} C_{12}], [c_1] \rangle, \langle [C_{21} C_{22}], [c_2] \rangle$, be the one-dimensional affine partitions for the statements s_1 and s_2 , respectively.

We apply six data dependence tests:

1. Write access $X[i, j]$ and itself in statement s_1 ,
2. Write access $X[i, j]$ with read access $X[i, j]$ in statement s_1 ,
3. Write access $X[i, j]$ in statement s_1 with read access $X[i, j - 1]$ in statement s_2 ,

4. Write access $Y[i, j]$ and itself in statement s_2 ,
5. Write access $Y[i, j]$ with read access $Y[i, j]$ in statement s_2 ,
6. Write access $Y[i, j]$ in statement s_2 with read access $Y[i-1, j]$ in statement s_1 .

We see that the dependence tests are all simple and highly repetitive. The only dependences present in this code occur in case (3) between instances of accesses $X[i, j]$ and $X[i, j-1]$ and in case (6) between $Y[i, j]$ and $Y[i-1, j]$.

The space-partition constraints imposed by the data dependence between $X[i, j]$ in statement s_1 and $X[i, j-1]$ in statement s_2 can be expressed in the following terms:

For all (i, j) and (i', j') such that

$$\begin{array}{ll} 1 \leq i \leq 100 & 1 \leq j \leq 100 \\ 1 \leq i' \leq 100 & 1 \leq j' \leq 100 \\ i = i' & j = j' - 1 \end{array}$$

we have

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c_1 \end{bmatrix} = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} c_2 \end{bmatrix}$$

That is, the first four conditions say that (i, j) and (i', j') lie within the iteration space of the loop nest, and the last two conditions say that the dynamic accesses $X[i, j]$ and $X[i, j-1]$ touch the same array element. We can derive the space-partition constraint for accesses $Y[i-1, j]$ in statement s_2 and $Y[i, j]$ in statement s_1 in a similar manner. \square

11.7.4 Solving Space-Partition Constraints

Once the space-partition constraints have been extracted, standard linear algebra techniques can be used to find the affine partitions satisfying the constraints. Let us first show how we find the solution to Example 11.41.

Example 11.42: We can find the affine partitions for Example 11.41 with the following steps:

1. Create the space-partition constraints shown in Example 11.41. We use the loop bounds in determining the data dependences, but they are not used in the rest of the algorithm otherwise.
2. The unknown variables in the equalities are $i, i', j, j', C_{11}, C_{12}, c_1, C_{21}, C_{22}$, and c_2 . Reduce the number of unknowns by using the equalities due to the access functions: $i = i'$ and $j = j' - 1$. We do so using Gaussian elimination, which reduces the four variables to two: say $t_1 = i = i'$, and $t_2 = j = j' - 1$. The equality for the partition becomes

$$\begin{bmatrix} C_{11} - C_{21} & C_{12} - C_{22} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} c_1 - c_2 - C_{22} \end{bmatrix} = 0$$

3. The equation above holds for all combinations of t_1 and t_2 . Thus, it must be that

$$\begin{aligned} C_{11} - C_{21} &= 0 \\ C_{12} - C_{22} &= 0 \\ c_1 - c_2 - C_{22} &= 0 \end{aligned}$$

If we perform the same steps on the constraint between the accesses $Y[i-1, j]$ and $Y[i, j]$, we get

$$\begin{aligned} C_{11} - C_{21} &= 0 \\ C_{12} - C_{22} &= 0 \\ c_1 - c_2 + C_{21} &= 0 \end{aligned}$$

Simplifying all the constraints together, we obtain the following relationships:

$$C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1.$$

4. Find all the independent solutions to the equations involving only unknowns in the coefficient matrix, ignoring the unknowns in the constant vectors in this step. There is only one independent choice in the coefficient matrix, so the affine partitions we seek can have at most a rank of one. We keep the partition as simple as possible by setting $C_{11} = 1$. We cannot assign 0 to C_{11} because that will create a rank-0 coefficient matrix, which maps all iterations to the same processor. It then follows that $C_{21} = 1$, $C_{22} = -1$, $C_{12} = -1$.
5. Find the constant terms. We know that the difference between the constant terms, $c_2 - c_1$, must be -1 . We get to pick the actual values, however. To keep the partitions simple, we pick $c_2 = 0$; thus $c_1 = -1$.

Let p be the ID of the processor executing iteration (i, j) . In terms of p , the affine partition is

$$\begin{aligned} s_1 : [p] &= [1 \quad -1] \begin{bmatrix} i \\ j \end{bmatrix} + [-1] \\ s_2 : [p] &= [1 \quad -1] \begin{bmatrix} i \\ j \end{bmatrix} + [0] \end{aligned}$$

That is, the (i, j) th iteration of s_1 is assigned to the processor $p = i - j - 1$, and the (i, j) th iteration of s_2 is assigned to processor $p = i - j$. \square

Algorithm 11.43: Finding a highest-ranked synchronization-free affine partition for a program.

INPUT: A program with affine array accesses.

OUTPUT: A partition.

METHOD: Do the following:

1. Find all data-dependent pairs of accesses in a program for each pair of data-dependent accesses, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ in statement s_1 nested in d_1 loops and $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ in statement s_2 nested in d_2 loops. Let $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ and $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ represent the (currently unknown) partitions of statements s_1 and s_2 , respectively. The space-partition constraints state that if

$$\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$$

then

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$$

for all \mathbf{i}_1 and \mathbf{i}_2 , within their respective loop bounds. We shall generalize the domain of iterations to include all \mathbf{i}_1 in Z^{d_1} and \mathbf{i}_2 in Z^{d_2} ; that is, the bounds are all assumed to be minus infinity to infinity. This assumption makes sense, since an affine partition cannot make use of the fact that an index variable can take on only a limited set of integer values.

2. For each pair of dependent accesses, we reduce the number of unknowns in the index vectors.

(a) Note that $\mathbf{F}\mathbf{i} + \mathbf{f}$ is the same vector as

$$\begin{bmatrix} \mathbf{F} & \mathbf{f} \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix}$$

That is, by adding an extra component 1 at the bottom of column-vector \mathbf{i} , we can make the column-vector \mathbf{f} be an additional, last column of the matrix \mathbf{F} . We may thus rewrite the equality of the access functions $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$ as

$$\begin{bmatrix} \mathbf{F}_1 & -\mathbf{F}_2 & (\mathbf{f}_1 - \mathbf{f}_2) \end{bmatrix} \begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix} = 0$$

- (b) The above equations will in general have more than one solution. However, we may still use Gaussian elimination to solve the equations for the components of \mathbf{i}_1 and \mathbf{i}_2 as best we can. That is, eliminate as many variables as possible until we are left with only variables that cannot be eliminated. The resulting solution for \mathbf{i}_1 and \mathbf{i}_2 will have the form

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix} = \mathbf{U} \begin{bmatrix} \mathbf{t} \\ 1 \end{bmatrix}$$

where \mathbf{U} is an upper-triangular matrix and \mathbf{t} is a vector of free variables ranging over all integers.

- (c) We may use the same trick as in Step (2a) to rewrite the equality of the partitions. Substituting the vector $(\mathbf{i}_1, \mathbf{i}_2, 1)$ with the result from Step (2b), we can write the constraints on the partitions as

$$\begin{bmatrix} \mathbf{C}_1 & -\mathbf{C}_2 & (\mathbf{c}_1 - \mathbf{c}_2) \end{bmatrix} \mathbf{U} \begin{bmatrix} \mathbf{t} \\ 1 \end{bmatrix} = 0$$

3. Drop the nonpartition variables. The equations above hold for all combinations of \mathbf{t} if

$$\begin{bmatrix} \mathbf{C}_1 & -\mathbf{C}_2 & (\mathbf{c}_1 - \mathbf{c}_2) \end{bmatrix} \mathbf{U} = \mathbf{0}.$$

Rewrite these equations in the form $\mathbf{A}\mathbf{x} = \mathbf{0}$, where \mathbf{x} is a vector of all the unknown coefficients of the affine partitions.

4. Find the rank of the affine partition and solve for the coefficient matrices. Since the rank of an affine partition is independent of the value of the constant terms in the partition, we eliminate all the unknowns that come from the constant vectors like \mathbf{c}_1 or \mathbf{c}_2 , thus replacing $\mathbf{A}\mathbf{x} = \mathbf{0}$ by simplified constraints $\mathbf{A}'\mathbf{x}' = \mathbf{0}$. Find the solutions to $\mathbf{A}'\mathbf{x}' = \mathbf{0}$, expressing them as \mathbf{B} , a set of basis vectors spanning the null space of \mathbf{A}' .
5. Find the constant terms. Derive one row of the desired affine partition from each basis vector in \mathbf{B} , and derive the constant terms using $\mathbf{A}\mathbf{x} = \mathbf{0}$.

□

Note that Step 3 ignores the constraints imposed by the loop bounds on variables \mathbf{t} . The constraints are only stricter as a result, and the algorithm must therefore be safe. That is, we place constraints on the \mathbf{C} 's and \mathbf{c} 's assuming \mathbf{t} is arbitrary. Conceivably, there would be other solutions for the \mathbf{C} 's and \mathbf{c} 's that are valid only because some values of \mathbf{t} are impossible. Not searching for these other solutions may cause us to miss an optimization, but cannot cause the program to be changed to a program that does something different from what the original program does.

11.7.5 A Simple Code-Generation Algorithm

Algorithm 11.43 generates affine partitions that split computations into independent partitions. Partitions can be assigned arbitrarily to different processors, since they are independent of one another. A processor may be assigned more than one partition and can interleave the execution of its partitions, as long as operations within each partition, which normally have data dependences, are executed sequentially.

It is relatively easy to generate a correct program given an affine partition. We first introduce Algorithm 11.45, a simple approach to generating code for a single processor that executes each of the independent partitions sequentially. Such code optimizes temporal locality, since array accesses that have several uses are very close in time. Moreover, the code easily can be turned into an SPMD program that executes each partition on a different processor. The code generated is, unfortunately, inefficient; we shall next discuss optimizations to make the code execute efficiently.

The essential idea is as follows. We are given bounds for the index variables of a loop nest, and we have determined, in Algorithm 11.43, a partition for the accesses of a particular statement s . Suppose we wish to generate sequential code that performs the action of each processor sequentially. We create an outermost loop that iterates through the processor IDs. That is, each iteration of this loop performs the operations assigned to a particular processor ID. The original program is inserted as the loop body of this loop; in addition, a test is added to guard each operation in the code to ensure that each processor only executes the operations assigned to it. In this way, we guarantee that the processor executes all the instructions assigned to it, and does so in the original sequential order.

Example 11.44: Let us generate code that executes the independent partitions in Example 11.41 sequentially. The original sequential program is from Fig. 11.26 is repeated here as Fig. 11.28.

```

for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }

```

Figure 11.28: Repeat of Fig. 11.26

In Example 11.42, the affine partitioning algorithm found one degree of parallelism. Thus, the processor space can be represented by a single variable p . Recall also from that example that we selected an affine partition that, for all values of index variables i and j with $1 \leq i \leq 100$ and $1 \leq j \leq 100$, assigned

1. Instance (i, j) of statement s_1 to processor $p = i - j - 1$, and
2. Instance (i, j) of statement s_2 to processor $p = i - j$.

We can generate the code in three steps:

1. For each statement, find all the processor IDs participating in the computation. We combine the constraints $1 \leq i \leq 100$ and $1 \leq j \leq 100$ with one of the equations $p = i - j - 1$ or $p = i - j$, and project away i and j to get the new constraints

- (a) $-100 \leq p \leq 98$ if we use the function $p = i - j - 1$ that we get for statement s_1 , and
 - (b) $-99 \leq p \leq 99$ if we use $p = i - j$ from statement s_2 .
2. Find all the processor IDs participating in any of the statements. When we take the union of these ranges, we get $-100 \leq p \leq 99$; these bounds are sufficient to cover all instances of both statements s_1 and s_2 .
 3. Generate the code to iterate through the computations in each partition sequentially. The code, shown in Fig. 11.29, has an outer loop that iterates through all the partition IDs participating in the computation (line (1)). Each partition goes through the motion of generating the indexes of all the iterations in the original sequential program in lines (2) and (3) so that it can pick out the iterations the processor p is supposed to execute. The tests of lines (4) and (6) make sure that statements s_1 and s_2 are executed only when the processor p would execute them.

The generated code, while correct, is extremely inefficient. First, even though each processor executes computation from at most 99 iterations, it generates loop indexes for 100×100 iterations, an order of magnitude more than necessary. Second, each addition in the innermost loop is guarded by a test, creating another constant factor of overhead. These two kinds of inefficiencies are dealt with in Sections 11.7.6 and 11.7.7, respectively. \square

```

1)   for (p = -100; p <= 99; p++)
2)       for (i = 1; i <= 100; i++)
3)           for (j = 1; j <= 100; j++) {
4)               if (p == i-j-1)
5)                   X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
6)               if (p == i-j)
7)                   Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
8)           }
```

Figure 11.29: A simple rewriting of Fig. 11.28 that iterates over processor space

Although the code of Fig. 11.29 appears designed to execute on a uniprocessor, we could take the inner loops, lines (2) through (8), and execute them on 200 different processors, each of which had a different value for p , from -100 to 99. Or, we could partition the responsibility for the inner loops among any number of processors less than 200, as long as we arranged that each processor knew what values of p it was responsible for and executed lines (2) through (8) for just those values of p .

Algorithm 11.45: Generating code that executes partitions of a program sequentially.

INPUT: A program P with affine array accesses. Each statement s in the program has associated bounds of the form $\mathbf{B}_s \mathbf{i} + \mathbf{b}_s \geq \mathbf{0}$, where \mathbf{i} is the vector of loop indexes for the loop nest in which statement s appears. Also associated with statement s is a partition $\mathbf{C}_s \mathbf{i} + \mathbf{c}_s = \mathbf{p}$ where \mathbf{p} is an m -dimensional vector of variables representing a processor ID; m is the maximum, over all statements in program P , of the rank of the partition for that statement.

OUTPUT: A program equivalent to P but iterating over the processor space rather than over loop indexes.

METHOD: Do the following:

1. For each statement, use Fourier-Motzkin elimination to project out all the loop index variables from the bounds.
2. Use Algorithm 11.13 to determine bounds on the partition ID's.
3. Generate loops, one for each of the m dimensions of processor space. Let $\mathbf{p} = [p_1, p_2, \dots, p_m]$ be the vector of variables for these loops; that is, there is one variable for each dimension of the processor space. Each loop variable p_i ranges over the union of the partition spaces for all statements in the program P .

Note that the union of the partition spaces is not necessarily convex. To keep the algorithm simple, instead of enumerating only those partitions that have a nonempty computation to perform, set the lower bound of each p_i to the minimum of all the lower bounds imposed by all statements and the upper bound of each p_i to the maximum of all the upper bounds imposed by all statements. Some values of \mathbf{p} may thereby have no operations.

The code to be executed by each partition is the original sequential program. However, every statement is guarded by a predicate so that only those operations belonging to the partition are executed. \square

An example of Algorithm 11.45 will follow shortly. Bear in mind, however, that we are still far from the optimal code for typical examples.

11.7.6 Eliminating Empty Iterations

We now discuss the first of the two transformations necessary to generate efficient SPMD code. The code executed by each processor cycles through all the iterations in the original program and picks out the operations that it is supposed to execute. If the code has k degrees of parallelism, the effect is that each processor performs k orders of magnitude more work. The purpose of the first transformation is to tighten the bounds of the loops to eliminate all the empty iterations.

We begin by considering the statements in the program one at a time. A statement's iteration space to be executed by each partition is the original iteration space plus the constraint imposed by the affine partition. We can generate

tight bounds for each statement by applying Algorithm 11.13 to the new iteration space; the new index vector is like the original sequential index vector, with processor ID's added as outermost indexes. Recall that the algorithm will generate tight bounds for each index in terms of surrounding loop indexes.

After finding the iteration spaces of the different statements, we combine them, loop by loop, making the bounds the union of those for each statement. Some loops end up having a single iteration, as illustrated by Example 11.46 below, and we can simply eliminate the loop and simply set the loop index to the value for that iteration.

Example 11.46: For the loop of Fig. 11.30(a), Algorithm 11.43 will create the affine partition

$$\begin{aligned}s_1 : p &= i \\ s_2 : p &= j\end{aligned}$$

Algorithm 11.45 will create the code of Fig. 11.30(b). Applying Algorithm 11.13 to statement s_1 produces the bound: $p \leq i \leq p$, or simply $i = p$. Similarly, the algorithm determines $j = p$ for statement s_2 . Thus, we get the code of Fig. 11.30(c). Copy propagation of variables i and j will eliminate the unnecessary test and produce the code of Fig. 11.30(d). \square

We now return to Example 11.44 and illustrate the step to merge multiple iteration spaces from different statements together.

Example 11.47: Let us now tighten the loop bounds of the code in Example 11.44. The iteration space executed by partition p for statement s_1 is defined by the following equalities and inequalities:

$$\begin{aligned}-100 &\leq p \leq 99 \\ 1 &\leq i \leq 100 \\ 1 &\leq j \leq 100 \\ i - p - 1 &= j\end{aligned}$$

Applying Algorithm 11.13 to the above creates the constraints shown in Fig. 11.31(a). Algorithm 11.13 generates the constraint $p + 2 \leq i \leq 100 + p + 1$ from $i - p - 1 = j$ and $1 \leq j \leq 100$, and tightens the upper bound of p to 98. Likewise, the bounds for each of the variables for statement s_2 are shown in Fig. 11.31(b).

The iteration spaces for s_1 and s_2 in Fig. 11.31 are similar, but as expected from Fig. 11.27, certain limits differ by 1 between the two. The code in Fig. 11.32 executes over this union of iteration spaces. For example, for i use $\max(1, p + 1)$ as the lower bound and $\min(100, 101 + p)$ as the upper bound. Note that the innermost loop has 2 iterations except that it has only one the first and last time it is executed. The overhead in generating loop indexes is thus reduced by an order of magnitude. Since the iteration space executed is larger than either that of s_1 and s_2 , conditionals are still necessary to select when these statements are executed. \square

```

for (i=1; i<=N; i++)
    Y[i] = Z[i]; /* (s1) */
for (j=1; j<=N; j++)
    X[j] = Y[j]; /* (s2) */

```

(a) Initial code.

```

for (p=1; p<=N; p++) {
    for (i=1; i<=N; i++)
        if (p == i)
            Y[i] = Z[i]; /* (s1) */
    for (j=1; j<=N; j++)
        if (p == j)
            X[j] = Y[j]; /* (s2) */
}

```

(b) Result of applying Algorithm 11.45.

```

for (p=1; p<=N; p++) {
    i = p;
    if (p == i)
        Y[i] = Z[i]; /* (s1) */
    j = p;
    if (p == j)
        X[j] = Y[j]; /* (s2) */
}

```

(c) After applying Algorithm 11.13.

```

for (p=1; p<=N; p++) {
    Y[p] = Z[p]; /* (s1) */
    X[p] = Y[p]; /* (s2) */
}

```

(d) Final code.

Figure 11.30: Code for Example 11.46

$$j: \quad i - p - 1 \leq j \leq i - p - 1$$

$$1 \leq j \leq 100$$

$$i: \quad p + 2 \leq i \leq 100 + p + 1$$

$$1 \leq i \leq 100$$

$$p: \quad -100 \leq p \leq 98$$

(a) Bounds for statement s_1 .

$$j: \quad i - p \leq j \leq i - p$$

$$1 \leq j \leq 100$$

$$i: \quad p + 1 \leq i \leq 100 + p$$

$$1 \leq i \leq 100$$

$$p: \quad -99 \leq p \leq 99$$

(b) Bounds for statement s_2 .

Figure 11.31: Tighter bounds on p , i , and j for Fig. 11.29

11.7.7 Eliminating Tests from Innermost Loops

The second transformation is to remove conditional tests from the inner loops. As seen from the examples above, conditional tests remain if the iteration spaces of statements in the loop intersect but not completely. To avoid the need for conditional tests, we split the iteration space into subspaces, each of which executes the same set of statements. This optimization requires code to be duplicated and should only be used to remove conditionals in the inner loops.

To split an iteration space to reduce tests in inner loops, we apply the following steps repeatedly until we remove all the tests in the inner loops:

1. Select a loop that consists of statements with different bounds.
2. Split the loop using a condition such that some statement is excluded from at least one of its components. We choose the condition from among the boundaries of the overlapping different polyhedra. If some statement has all its iterations in only one of the half planes of the condition, then such a condition is useful.
3. Generate code for each of these iteration spaces separately.

Example 11.48: Let us remove the conditionals from the code of Fig. 11.32. Statements s_1 and s_2 are mapped to the same set of partition ID's except for

```

for (p = -100; p <= 99; p++)
  for (i = max(1,p+1); i <= min(100,101+p); i++)
    for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
      if (p == i-j-1)
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
      if (p == i-j)
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }

```

Figure 11.32: Code of Fig. 11.29 improved by tighter loop bounds

the boundary partitions at either end. Thus, we separate the partition space into three subspaces:

1. $p = -100$,
2. $-99 \leq p \leq 98$, and
3. $p = 99$.

The code for each subspace can then be specialized for the value(s) of p contained. Figure 11.33 shows the resulting code for each of the three iteration spaces.

Notice that the first and third spaces do not need loops on i or j , because for the particular value of p that defines each space, these loops are degenerate; they have only one iteration. For example, in space (1), substituting $p = -100$ in the loop bounds restricts i to 1, and subsequently j to 100. The assignments to p in spaces (1) and (3) are evidently dead code and can be eliminated.

Next we split the loop with index i in space (2). Again, the first and last iterations of loop index i are different. Thus, we split the loop into three subspaces:

- a) $\max(1, p+1) \leq i < p+2$, where only s_2 is executed,
- b) $\max(1, p+2) \leq i \leq \min(100, 100+p)$, where both s_1 and s_2 are executed, and
- c) $101+p < i \leq \min(101+p, 100)$, where only s_1 is executed.

The loop nest for space (2) in Fig. 11.33 can thus be written as in Fig. 11.34(a).

Figure 11.34(b) shows the optimized program. We have substituted Fig. 11.34(a) for the loop nest in Fig. 11.33. We also propagated out assignments to p , i , and j into the array accesses. When optimizing at the intermediate-code level, some of these assignments will be identified as common subexpressions and re-extracted from the array-access code. \square

```

/* space (1) */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
        }

/* space (3) */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */

```

Figure 11.33: Splitting the iteration space on the value of p

11.7.8 Source-Code Transforms

We have seen how we can derive from simple affine partitions for each statement programs that are significantly different from the original source. It is not apparent from the examples seen so far how affine partitions correlate with changes at the source level. This section shows that we can reason about source code changes relatively easily by breaking down affine partitions into a series of primitive transforms.

Seven Primitive Affine Transforms

Every affine partition can be expressed as a series of primitive affine transforms, each of which corresponds to a simple change at the source level. There are seven kinds of primitive transforms: the first four primitives are illustrated in Fig. 11.35, the last three, also known as *unimodular transforms*, are illustrated in Fig. 11.36.

The figure shows one example for each primitive: a source, an affine partition, and the resulting code. We also draw the data dependences for the code before and after the transforms. From the data dependence diagrams, we see that each primitive corresponds to a simple geometric transform and induces a relatively simple code transform. The seven primitives are:

```

/* space (2) */
for (p = -99; p <= 98; p++) {
    /* space (2a) */
    if (p >= 0) {
        i = p+1;
        j = 1;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* space (2b) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        j = i-p-1;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        j = i-p;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* space (2c) */
    if (p <= -1) {
        i = 101+p;
        j = 100;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
    }
}

```

(a) Splitting space (2) on the value of i .

```

/* space (1); p = -100 */
X[1,100] = X[1,100] + Y[0,100]; /* (s1) */

/* space (2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1]; /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1]; /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p]; /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}

/* space (3); p = 99 */
Y[100,1] = X[100,0] + Y[100,1]; /* (s2) */

```

(b) Optimized code equivalent to Fig. 11.28.

Figure 11.34: Code for Example 11.48

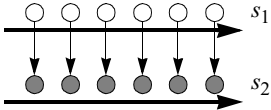
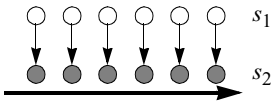
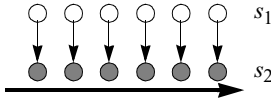
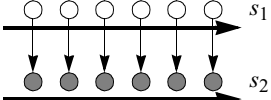
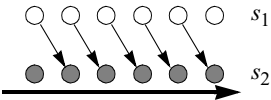
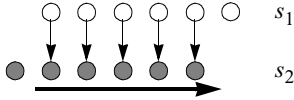
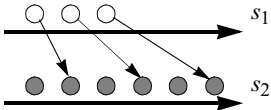
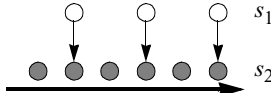
SOURCE CODE	PARTITION	TRANSFORMED CODE
<pre>for (i=1; i<=N; i++) Y[i] = Z[i]; /*s1*/ for (j=1; j<=N; j++) X[j] = Y[j]; /*s2*/</pre> 	<p>Fusion</p> $s_1 : p = i$ $s_2 : p = j$	<pre>for (p=1; p<=N; p++){ Y[p] = Z[p]; X[p] = Y[p]; }</pre> 
<pre>for (p=1; p<=N; p++){ Y[p] = Z[p]; X[p] = Y[p]; }</pre> 	<p>Fission</p> $s_1 : i = p$ $s_2 : j = p$	<pre>for (i=1; i<=N; i++) Y[i] = Z[i]; /*s1*/ for (j=1; j<=N; j++) X[j] = Y[j]; /*s2*/</pre> 
<pre>for (i=1; i<=N; i++) { Y[i] = Z[i]; /*s1*/ X[i] = Y[i-1]; /*s2*/ }</pre> 	<p>Re-indexing</p> $s_1 : p = i$ $s_2 : p = i - 1$	<pre>if (N>=1) X[1]=Y[0]; for (p=1; p<=N-1; p++){ Y[p]=Z[p]; X[p+1]=Y[p]; } if (N>=1) Y[N]=Z[N];</pre> 
<pre>for (i=1; i<=N; i++) Y[2*i] = Z[2*i]; /*s1*/ for (j=1; j<=2N; j++) X[j]=Y[j]; /*s2*/</pre> 	<p>Scaling</p> $s_1 : p = 2 * i$ $(s_2 : p = j)$	<pre>for (p=1; p<=2*N; p++){ if (p mod 2 == 0) Y[p] = Z[p]; X[p] = Y[p]; }</pre> 

Figure 11.35: Primitive affine transforms (I)

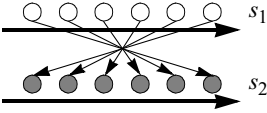
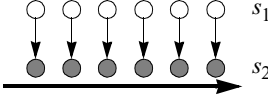
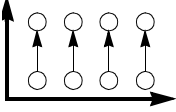
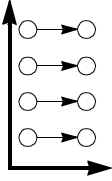
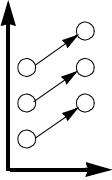
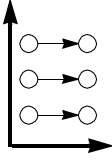
SOURCE CODE	PARTITION	TRANSFORMED CODE
<pre> for (i=0; i<=N; i++) Y[N-i] = Z[i]; /*s1*/ for (j=0; j<=N; j++) X[j] = Y[j]; /*s2*/ </pre> 	<p>Reversal</p> $s_1 : p = N - i$ $(s_2 : p = j)$	<pre> for (p=0; p<=N; p++){ Y[p] = Z[N-p]; X[p] = Y[p]; } </pre> 
<pre> for (i=1; i<=N; i++) for (j=0; j<=M; j++) Z[i,j] = Z[i-1,j]; </pre> 	<p>Permutation</p> $\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$	<pre> for (p=0; p<=M; p++) for (q=1; q<=N; i++) Z[q,p] = Z[q-1,p]; </pre> 
<pre> for (i=1; i<=N+M-1; i++) for (j=max(1,i+N); j<=min(i,M); j++) Z[i,j] = Z[i-1,j-1]; </pre> 	<p>Skewing</p> $\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	<pre> for (p=1; p<=N; p++) for (q=1; q<=M; q++) Z[p,q-p] = Z[p-1,q-p-1]; </pre> 

Figure 11.36: Primitive affine transforms (II)

Unimodular Transforms

A unimodular transform is represented by just a unimodular coefficient matrix and no constant vector. A *unimodular matrix* is a square matrix whose determinant is ± 1 . The significance of a unimodular transform is that it maps an n -dimensional iteration space to another n -dimensional polyhedron, where there is a one-to-one correspondence between iterations of the two spaces.

1. *Fusion*. The fusion transform is characterized by mapping multiple loop indexes in the original program to the same loop index. The new loop fuses statements from different loops.
2. *Fission*. Fission is the inverse of fusion. It maps the same loop index for different statements to different loop indexes in the transformed code. This splits the original loop into multiple loops.
3. *Re-indexing*. Re-indexing shifts the dynamic executions of a statement by a constant number of iterations. The affine transform has a constant term.
4. *Scaling*. Consecutive iterations in the source program are spaced apart by a constant factor. The affine transform has a positive nonunit coefficient.
5. *Reversal*. Execute iterations in a loop in reverse order. Reversal is characterized by having -1 as a coefficient.
6. *Permutation*. Permute the inner and outer loops. The affine transform consists of permuted rows of the identity matrix.
7. *Skewing*. Iterate through the iteration space in the loops at an angle. The affine transform is a unimodular matrix with 1's on the diagonal.

A Geometric Interpretation of Parallelization

The affine transforms shown in all but the fission example are derived by applying the synchronization-free affine partition algorithm to the respective source codes. (We shall discuss how fission can parallelize code with synchronization in the next section.) In each of the examples, the generated code has an (outermost) parallelizable loop whose iterations can be assigned to different processors and no synchronization is necessary.

These examples illustrate that there is a simple geometric interpretation of how parallelization works. Dependence edges always point from an earlier instance to a later instance. So, dependences between separate statements not nested in any common loop follows the lexical order; dependences between

statements nested in the same loop follow the lexicographic order. Geometrically, dependences of a two-dimensional loop nest always point within the range $[0^\circ, 180^\circ)$, meaning that the angle of the dependence must be below 180° , but no less than 0° .

The affine transforms change the ordering of iterations such that all the dependences are found only between operations nested within the same iteration of the outermost loop. In other words, there are no dependence edges at the boundaries of iterations in the outermost loop. We can parallelize simple source codes by drawing their dependences and finding such transforms geometrically.

11.7.9 Exercises for Section 11.7

Exercise 11.7.1: For the following loop

```
for (i = 2; i < 100; i++)
    A[i] = A[i-2];
```

- What is the largest number of processors that can be used effectively to execute this loop?
- Rewrite the code with processor p as a parameter.
- Set up and find one solution to the space-partition constraints for this loop.
- What is the affine partition of highest rank for this loop?

Exercise 11.7.2: Repeat Exercise 11.7.1 for the loop nests in Fig. 11.37.

Exercise 11.7.3: Rewrite the following code

```
for (i = 0; i < 100; i++)
    A[i] = 2*A[i];
for (j = 0; j < 100; j++)
    A[j] = A[j] + 1;
```

so it consists of a single loop. Rewrite the loop in terms of a processor number p so the code can be partitioned among 100 processors, with iteration p executed by processor p .

Exercise 11.7.4: In the following code

```
for (i = 1; i < 100; i++)
    for (j = 1; j < 100; j++)
/* (s) */    A[i,j] =
                (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])/4;
```

```

for (i = 0; i <= 97; i++)
    A[i] = A[i+2];

```

(a)

```

for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + C[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1];
        }

```

!(b)

```

for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + A[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1] + B[i,j,k];
        }

```

!(c)

Figure 11.37: Code for Exercise 11.7.2

the only constraints are that the statement s that forms the body of the loop nest must execute iterations $s(i-1, j)$ and $s(i, j-1)$ before executing iteration $s(i, j)$. Verify that these are the only necessary constraints. Then rewrite the code so that the outer loop has index variable p , and on the p th iteration of the outer loop, all instances of $s(i, j)$ such that $i + j = p$ are executed.

Exercise 11.7.5: Repeat Exercise 11.7.4, but arrange that on the p th iteration of the outer loop, instances of s such that $i - j = p$ are executed.

! Exercise 11.7.6: Combine the following loops

```

for (i = 0; i < 100; i++)
    A[i] = B[i];
for (j = 98; j >= 0; j = j-2)
    B[i] = i;

```

into a single loop, preserving all dependencies.

Exercise 11.7.7: Show that the matrix

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

is unimodular. Describe the transformation it performs on a two-dimensional loop nest.

Exercise 11.7.8: Repeat Exercise 11.7.7 on the matrix

$$\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

11.8 Synchronization Between Parallel Loops

Most programs have no parallelism if we do not allow processors to perform any synchronizations. But adding even a small constant number of synchronization operations to a program can expose more parallelism. We shall first discuss parallelism made possible by a constant number of synchronizations in this section and the general case, where we embed synchronization operations in loops, in the next.

11.8.1 A Constant Number of Synchronizations

Programs with no synchronization-free parallelism may contain a sequence of loops, some of which are parallelizable if they are considered independently. We can parallelize such loops by introducing synchronization barriers before and after their execution. Example 11.49 illustrates the point.

```

for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
for (i = 0; i < n; i++)
    for (j = 1; j < n; j++)
        X[i,j] = g(X[i,j] + X[i,j-1]);

```

Figure 11.38: Two sequential loop nests

Example 11.49: In Fig. 11.38 is a program representative of an ADI (Alternating Direction Implicit) integration algorithm. There is no synchronization-free parallelism. Dependences in the first loop nest require that each processor works on a column of array X ; however, dependences in the second loop nest require that each processor works on a row of array X . For there to be no communication, the entire array has to reside on the same processor, hence there

is no parallelism. We observe, however, that both loops are independently parallelizable.

One way to parallelize the code is to have different processors work on different columns of the array in the first loop, synchronize and wait for all processors to finish, and then operate on the individual rows. In this way, all the computation in the algorithm can be parallelized with the introduction of just one synchronization operation. However, we note that while only one synchronization is performed, this parallelization requires almost all the data in matrix X to be transferred between processors. It is possible to reduce the amount of communication by introducing more synchronizations, which we shall discuss in Section 11.9.9. \square

It may appear that this approach is applicable only to programs consisting of a sequence of loop nests. However, we can create additional opportunities for the optimization through code transforms. We can apply loop fission to decompose loops in the original program into several smaller loops, which can then be parallelized individually by separating them with barriers. We illustrate this technique with Example 11.50.

Example 11.50: Consider the following loop:

```
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];      /* (s1) */
    W[A[i]] = X[i];         /* (s2) */
}
```

Without knowledge of the values in array A , we must assume that the access in statement s_2 may write to any of the elements of W . Thus, the instances of s_2 must be executed sequentially in the order they are executed in the original program.

There is no synchronization-free parallelism, and Algorithm 11.43 will simply assign all the computation to the same processor. However, at the least, instances of statement s_1 can be executed in parallel. We can parallelize part of this code by having different processors perform different instances of statement s_1 . Then, in a separate sequential loop, one processor, say numbered 0, executes s_2 , as in the SPMD code shown in Fig. 11.39. \square

11.8.2 Program-Dependence Graphs

To find all the parallelism made possible by a constant number of synchronizations, we can apply fission to the original program greedily. Break up loops into as many separate loops as possible, and then parallelize each loop independently.

To expose all the opportunities for loop fission, we use the abstraction of a *program-dependence graph* (PDG). A program dependence graph of a program

```

X[p] = Y[p] + Z[p];    /* (s1) */
/* synchronization barrier */
if (p == 0)
    for (i=1; i<=n; i++)
        W[A[i]] = X[i]; /* (s2) */

```

Figure 11.39: SPMD code for the loop in Example 11.50, with p being a variable holding the processor ID

is a graph whose nodes are the assignment statements of the program and whose edges capture the data dependences, and the directions of the data dependence, between statements. An edge from statement s_1 to statement s_2 exists whenever some dynamic instance of s_1 shares a data dependence with a *later* dynamic instance of s_2 .

To construct the PDG for a program, we first find the data dependences between every pair of (not necessarily distinct) static accesses in every pair of (not necessarily distinct) statements. Suppose we determine that there is a dependence between access \mathcal{F}_1 in statement s_1 and access \mathcal{F}_2 in statement s_2 . Recall that an instance of a statement is specified by an index vector $\mathbf{i} = [i_1, i_2, \dots, i_m]$ where i_k is the loop index of the k th outermost loop in which the statement is embedded.

1. If there exists a data-dependent pair of instances, \mathbf{i}_1 of s_1 and \mathbf{i}_2 of s_2 , and \mathbf{i}_1 is executed before \mathbf{i}_2 in the original program, written $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$, then there is an edge from s_1 to s_2 .
2. Similarly, if there exists a data-dependent pair of instances, \mathbf{i}_1 of s_1 and \mathbf{i}_2 of s_2 , and $\mathbf{i}_2 \prec_{s_1 s_2} \mathbf{i}_1$, then there is an edge from s_2 to s_1 .

Note that it is possible for a data dependence between two statements s_1 and s_2 to generate both an edge from s_1 to s_2 and an edge from s_2 back to s_1 .

In the special case where statements s_1 and s_2 are not distinct, $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ if and only if $\mathbf{i}_1 \prec \mathbf{i}_2$ (\mathbf{i}_1 is lexicographically less than \mathbf{i}_2). In the general case, s_1 and s_2 may be different statements, possibly belonging to different loop nests.

Example 11.51: For the program of Example 11.50, there are no dependences among the instances of statement s_1 . However, the i th instance of statement s_2 must follow the i th instance of statement s_1 . Worse, since the reference $W[A[i]]$ may write any element of array W , the i th instance of s_2 depends on all previous instances of s_2 . That is, statement s_2 depends on itself. The PDG for the program of Example 11.50 is shown in Fig. 11.40. Note that there is one cycle in the graph, containing s_2 only. \square

The program-dependence graph makes it easy to determine if we can split statements in a loop. Statements connected in a cycle in a PDG cannot be



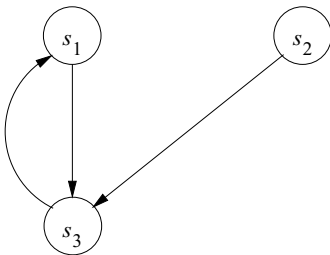
Figure 11.40: Program-dependence graph for the program of Example 11.50

split. If $s_1 \rightarrow s_2$ is a dependence between two statements in a cycle, then some instance of s_1 must execute before some instance of s_2 , and vice versa. Note that this mutual dependence occurs only if s_1 and s_2 are embedded in some common loop. Because of the mutual dependence, we cannot execute all instances of one statement before the other, and therefore loop fission is not allowed. On the other hand, if the dependence $s_1 \rightarrow s_2$ is unidirectional, we can split up the loop and execute all the instances of s_1 first, then those of s_2 .

```

for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];          /* (s1) */
    for (j = i; j < n; j++) {
        X[i,j] = Y[i,j]*Y[i,j]; /* (s2) */
        Z[j] = Z[j] + X[i,j];   /* (s3) */
    }
}
    
```

(a) A program.



(b) Its dependence graph.

Figure 11.41: Program and dependence graph for Example 11.52.

Example 11.52: Figure 11.41(b) shows the program-dependence graph for the program of Fig. 11.41(a). Statements s_1 and s_3 belong to a cycle in the graph and therefore cannot be placed in separate loops. We can, however, split statement s_2 out and execute all its instances before executing the rest of the computation, as in Fig. 11.42. The first loop is parallelizable, but the second is not. We can parallelize the first loop by placing barriers before and after its parallel execution. \square


```

for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
        X[i,j] = Y[i,j]*Y[i,j];    /* (s2) */
for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];              /* (s1) */
    for (j = i; j < n; j++)
        Z[j] = Z[j] + X[i,j];      /* (s3) */
}

```

Figure 11.42: Grouping strongly connected components of a loop nest

11.8.3 Hierarchical Time

While the relation $\prec_{s_1 s_2}$ can be very hard to compute in general, there is a family of programs to which the optimizations of this section are commonly applied, and for which there is a straightforward way to compute dependencies. Assume that the program is block structured, consisting of loops and simple arithmetic operations and no other control constructs. A statement in the program is either an assignment statement, a sequence of statements, or a loop construct whose body is a statement. The control structure thus represents a hierarchy. At the top of the hierarchy is the node representing the statement of the whole program. An assignment statement is a leaf node. If a statement is a sequence, then its children are the statements within the sequence, laid out from left to right according to their lexical order. If a statement is a loop, then its children are the components of the loop body, which is typically a sequence of one or more statements.

```

s0;
L1: for (i = 0; ...) {
    s1;
    L2: for (j = 0; ...) {
        s2;
        s3;
    }
    L3: for (k = 0; ... )
        s4;
    s5;
}

```

Figure 11.43: A hierarchically structured program

Example 11.53: The hierarchical structure of the program in Fig. 11.43 is shown in Fig. 11.44. The hierarchical nature of the execution sequence is high-

lighted in Fig. 11.45. The single instance of s_0 precedes all other operations, because it is the first statement executed. Next, we execute all instructions from the first iteration of the outer loop before those in the second iteration and so forth. For all dynamic instances whose loop index i has value 0, the statements s_1 , L_2 , L_3 , and s_5 are executed in lexical order. We can repeat the same argument to generate the rest of the execution order. \square

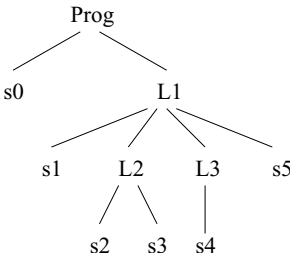


Figure 11.44: Hierarchical structure of the program in Example 11.53.

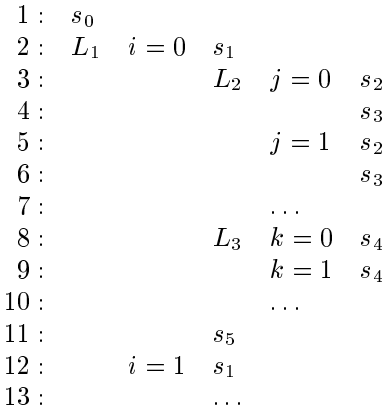


Figure 11.45: Execution order of the program in Example 11.53.

We can resolve the ordering of two instances from two different statements in a hierarchical manner. If the statements share common loops, we compare the values of their common loop indexes, starting with the outermost loop. As soon as we find a difference between their index values, the difference determines the ordering. Only if the index values for the outer loops are the same do we need to compare the indexes of the next inner loop. This process is analogous to how we would compare time expressed in terms of hours, minutes and seconds. To compare two times, we first compare the hours, and only if they refer to

the same hour would we compare the minutes and so forth. If the index values are the same for all common loops, then we resolve the order based on their relative lexical placement. Thus, the execution order for the simple nested-loop programs we have been discussing is often referred to as “hierarchical time.”

Let s_1 be a statement nested in a d_1 -deep loop, and s_2 in a d_2 -deep loop, sharing d common (outer) loops; note $d \leq d_1$ and $d \leq d_2$ certainly. Suppose $\mathbf{i} = [i_1, i_2, \dots, i_{d_1}]$ is an instance of s_1 and $\mathbf{j} = [j_1, j_2, \dots, j_{d_2}]$ is an instance of s_2 .

$\mathbf{i} \prec_{s_1 s_2} \mathbf{j}$ if and only if either

1. $[i_1, i_2, \dots, i_d] \prec [j_1, j_2, \dots, j_d]$, or
2. $[i_1, i_2, \dots, i_d] = [j_1, j_2, \dots, j_d]$, and s_1 appears lexically before s_2 .

The predicate $[i_1, i_2, \dots, i_d] \prec [j_1, j_2, \dots, j_d]$ can be written as a disjunction of linear inequalities:

$$(i_1 < j_1) \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \dots \vee (i_1 = j_1 \wedge \dots \wedge i_{d-1} = j_{d-1} \wedge i_d < j_d)$$

A PDG edge from s_1 to s_2 exists as long as the data-dependence condition and one of the disjunctive clauses can be made true simultaneously. Thus, we may need to solve up to d or $d + 1$ linear integer programs, depending on whether s_1 appears lexically before s_2 , to determine the existence of one edge.

11.8.4 The Parallelization Algorithm

We now present a simple algorithm that first splits up the computation into as many different loops as possible, then parallelizes them independently.

Algorithm 11.54: Maximize the degree of parallelism allowed by $O(1)$ synchronizations.

INPUT: A program with array accesses.

OUTPUT: SPMD code with a constant number of synchronization barriers.

METHOD:

1. Construct the program-dependence graph and partition the statements into strongly connected components (SCC's). Recall from Section 10.5.8 that a strongly connected component is a maximal subgraph of the original whose every node in the subgraph can reach every other node.
2. Transform the code to execute SCC's in a topological order by applying fission if necessary.
3. Apply Algorithm 11.43 to each SCC to find all of its synchronization-free parallelism. Barriers are inserted before and after each parallelized SCC.

□

While Algorithm 11.54 finds all degrees of parallelism with $O(1)$ synchronizations, it has a number of weaknesses. First, it may introduce unnecessary synchronizations. As a matter of fact, if we apply this algorithm to a program that can be parallelized without synchronization, the algorithm will parallelize each statement independently and introduce a synchronization barrier between the parallel loops executing each statement. Second, while there may only be a constant number of synchronizations, the parallelization scheme may transfer a lot of data among processors with each synchronization. In some cases, the cost of communication makes the parallelism too expensive, and we may even be better off executing the program sequentially on a uniprocessor. In the following sections, we shall next take up ways to increase data locality, and thus reduce the amount of communication.

11.8.5 Exercises for Section 11.8

Exercise 11.8.1: Apply Algorithm 11.54 to the code of Fig. 11.46.

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        B[i,j] = Y[i,j] + A[i] + A[j]; /* (s2) */
```

Figure 11.46: Code for Exercise 11.8.1

Exercise 11.8.2: Apply Algorithm 11.54 to the code of Fig. 11.47.

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    B[i] = B[i] + A[i]; /* (s2) */
    for (j=0; j<100; j++)
        C[j] = Y[j] + B[j]; /* (s3) */
}
```

Figure 11.47: Code for Exercise 11.8.2

Exercise 11.8.3: Apply Algorithm 11.54 to the code of Fig. 11.48.

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    for (j=0; j<100; j++)
        B[j] = A[i] + Y[j]; /* (s2) */
    C[i] = B[i] + Z[i]; /* (s3) */
    for (j=0; j<100; j++)
        D[i,j] = A[i] + B[j]; /* (s4) */
}

```

Figure 11.48: Code for Exercise 11.8.3

11.9 Pipelining

In pipelining, a task is decomposed into a number of stages to be performed on different processors. For example, a task computed using a loop of n iterations can be structured as a pipeline of n stages. Each stage is assigned to a different processor; when one processor is finished with its stage, the results are passed as input to the next processor in the pipeline.

In the following, we start by explaining the concept of pipelining in more detail. We then show a real-life numerical algorithm, known as successive over-relaxation, to illustrate the conditions under which pipelining can be applied, in Section 11.9.2. We then formally define the constraints that need to be solved in Section 11.9.6, and describe an algorithm for solving them in Section 11.9.7. Programs that have multiple independent solutions to the time-partition constraints are known as having outermost *fully permutable loops*; such loops can be pipelined easily, as discussed in Section 11.9.8.

11.9.1 What is Pipelining?

Our initial attempts to parallelize loops partitioned the iterations of a loop nest so that two iterations that shared data were assigned to the same processor. Pipelining allows processors to share data, but generally does so only in a “local,” way, with data passed from one processor to another that is adjacent in the processor space. Here is a simple example.

Example 11.55: Consider the loop:

```

for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];

```

This code sums up the i th row of Y and adds it to the i th element of X . The inner loop, corresponding to the summation, must be performed sequentially

Time	Processors		
	1	2	3
1	$X[1] += Y[1,1]$		
2	$X[2] += Y[2,1]$	$X[1] += Y[1,2]$	
3	$X[3] += Y[3,1]$	$X[2] += Y[2,2]$	$X[1] += Y[1,3]$
4	$X[4] += Y[4,1]$	$X[3] += Y[3,2]$	$X[2] += Y[2,3]$
5		$X[4] += Y[4,2]$	$X[3] += Y[3,3]$
6			$X[4] += Y[4,3]$

Figure 11.49: Pipelined execution of Example 11.55 with $m = 4$ and $n = 3$.

because of the data dependence;⁶ however, the different summation tasks are independent. We can parallelize this code by having each processor perform a separate summation. Processor i accesses row i of Y and updates the i th element of X .

Alternatively, we can structure the processors to execute the summation in a pipeline, and derive parallelism by overlapping the execution of the summations, as shown in Fig. 11.49. More specifically, each iteration of the inner loop can be treated as a stage of a pipeline: stage j takes an element of X generated in the previous stage, adds to it an element of Y , and passes the result to the next stage. Notice that in this case, each processor accesses a column, instead of a row, of Y . If Y is stored in column-major form, there is a gain in locality by partitioning according to columns, rather than by rows.

We can initiate a new task as soon as the first processor is done with the first stage of the previous task. At the beginning, the pipeline is empty and only the first processor is executing the first stage. After it completes, the results are passed to the second processor, while the first processor starts on the second task, and so on. In this way, the pipeline gradually fills until all the processors are busy. When the first processor finishes with the last task, the pipeline starts to drain, with more and more processors becoming idle until the last processor finishes the last task. In the steady state, n tasks can be executed concurrently in a pipeline of n processors. \square

It is interesting to contrast pipelining with simple parallelism, where different processors execute different tasks:

- Pipelining can only be applied to nests of depth at least two. We can treat each iteration of the outer loop as a task and the iterations in the inner loop as stages of that task.
- Tasks executed on a pipeline may share dependences. Information pertaining to the same stage of each task is held on the same processor; thus results generated by the i th stage of a task can be used by the i th stage

⁶ Remember that we do not take advantage of the assumed commutativity and associativity of addition.

of subsequent tasks with no communication cost. Similarly, each input data element used by a single stage of different tasks needs to reside only on one processor, as illustrated by Example 11.55.

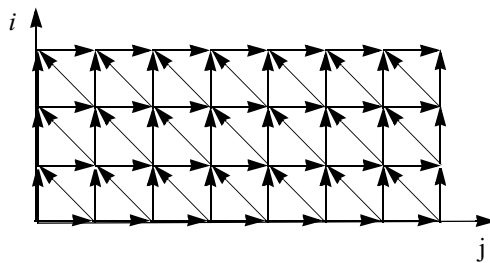
- If the tasks are independent, then simple parallelization has better processor utilization because processors can execute all at once without having to pay for the overhead of filling and draining the pipeline. However, as shown in Example 11.55, the pattern of data accesses in a pipelined scheme is different from that of simple parallelization. Pipelining may be preferable if it reduces communication.

11.9.2 Successive Over-Relaxation (SOR): An Example

Successive over-relaxation (SOR) is a technique for accelerating the convergence of relaxation methods for solving sets of simultaneous linear equations. A relatively simple template illustrating its data-access pattern is shown in Fig. 11.50(a). Here, the new value of an element in the array depends on the values of elements in its neighborhood. Such an operation is performed repeatedly, until some convergence criterion is met.

```
for (i = 0; i <= m; i++)
  for (j = 0; j <= n; j++)
    X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2])
```

(a) Original source.



(b) Data dependences in the code.

Figure 11.50: An example of successive over-relaxation (SOR)

Shown in Fig. 11.50(b) is a picture of the key data dependences. We do not show dependences that can be inferred by the dependences already included in the figure. For example, iteration $[i, j]$ depends on iterations $[i, j - 1]$, $[i, j - 2]$ and so on. It is clear from the dependences that there is no synchronization-free parallelism. Since the longest chain of dependences consists of $O(m + n)$ edges, by introducing synchronization, we should be able to find one degree of parallelism and execute the $O(mn)$ operations in $O(m + n)$ unit time.

In particular, we observe that iterations that lie along the 150° diagonals⁷ in Fig. 11.50(b) do not share any dependences. They only depend on the iterations that lie along diagonals closer to the origin. Therefore we can parallelize this code by executing iterations on each diagonal in order, starting at the origin and proceeding outwards. We refer to the iterations along each diagonal as a *wavefront*, and such a parallelization scheme as *wavefronting*.

11.9.3 Fully Permutable Loops

We first introduce the notion of *full permutability*, a concept useful for pipelining and other optimizations. Loops are *fully permutable* if they can be permuted arbitrarily without changing the semantics of the original program. Once loops are put in a fully permutable form, we can easily pipeline the code and apply transformations such as blocking to improve data locality.

The SOR code, as it written in Fig. 11.50(a), is not fully permutable. As shown in Section 11.7.8, permuting two loops means that iterations in the original iteration space are executed column by column instead of row by row. For instance, the original computation in iteration [2,3] would execute before that of [1,4], violating the dependences shown in Fig. 11.50(b).

We can, however, transform the code to make it fully permutable. Applying the affine transform

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

to the code yields the code shown in Fig. 11.51(a). This transformed code is fully permutable, and its permuted version is shown in Fig. 11.51(c). We also show the iteration space and data dependences of these two programs in Fig. 11.51(b) and (d), respectively. From the figure, we can easily see that this ordering preserves the relative ordering between every data-dependent pair of accesses.

When we permute loops, we change the set of operations executed in each iteration of the outermost loop drastically. The fact that we have this degree of freedom in scheduling means that there is a lot of slack in the ordering of operations in the program. Slack in scheduling means opportunities for parallelization. We show later in this section that if a nest has k outermost fully permutable loops, by introducing just $O(n)$ synchronizations, we can get $O(k - 1)$ degrees of parallelism (n is the number of iterations in a loop).

11.9.4 Pipelining Fully Permutable Loops

A loop with k outermost fully permutable loops can be structured as a pipeline with $O(k - 1)$ dimensions. In the SOR example, $k = 2$, so we can structure the processors as a linear pipeline.

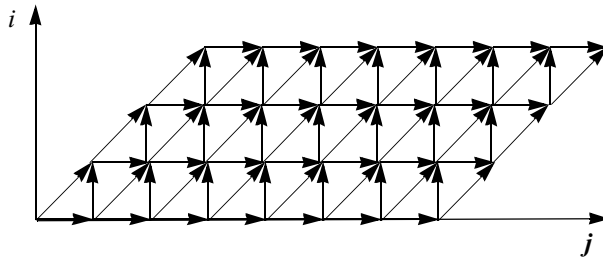
⁷I.e., the sequences of points formed by repeatedly moving down 1 and right 2.


```

for (i = 0; i <= m; i++)
  for (j = i; j <= i+n; j++)
    X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])

```

(a) The code in Fig. 11.50 transformed by $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.



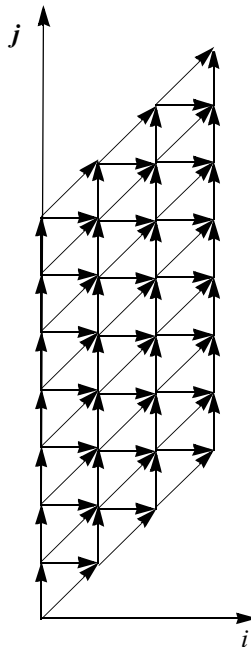
(b) Data dependences of the code in (a).

```

for (j = 0; j <= m+n; j++)
  for (i = max(0,j); i <= min(m,j), i++)
    X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2])

```

(c) A permutation of the loops in (a).



(d) Data dependences of the code in (b).

Figure 11.51: Fully permutable version of the code Fig. 11.50

We can pipeline the SOR code in two different ways, shown in Fig. 11.52(a) and Fig. 11.52(b), corresponding to the two possible permutations shown in Fig. 11.51(a) and (c), respectively. In each case, every column of the iteration space constitutes a task, and every row constitutes a stage. We assign stage i to processor i , thus each processor executes the inner loop of the code. Ignoring boundary conditions, a processor can execute iteration i only after processor $p - 1$ has executed iteration $i - 1$.

```

/* 0 <= p <= m */
for (j = p; j <= p+n; j++) {
    if (p > 0) wait (p-1);
    X[j-p+1] = 1/3 * (X[j-p] + X[j-p+1] + X[j-p+2]);
    if (p < min (m,j)) signal (p+1);
}

```

(a) Processors assigned to rows.

```

/* 0 <= p <= m+n */
for (i = max(0,p); i <= min(m,p); i++) {
    if (p > max(0,i)) wait (p-1);
    X[p-i+1] = 1/3 * (X[p-i] + X[p-i+1] + X[p-i+2]);
    if (p < m+n) & (p > i) signal (p+1);
}

```

(b) Processors assigned to columns.

Figure 11.52: Two pipelining implementations of the code from Fig. 11.51

Suppose every processor takes exactly the same amount of time to execute an iteration and synchronization happens instantaneously. Both these pipelined schemes would execute the same iterations in parallel; the only difference is that they have different processor assignments. All the iterations executed in parallel lie along the 135° diagonals in the iteration space in Fig. 11.51(b), which corresponds to the 150° diagonals in the iteration space of the original code; see Fig. 11.50(b).

However, in practice, processors with caches do not always execute the same code in the same amount of time, and the time for synchronization also varies. Unlike the use of synchronization barriers which forces all processors to operate in lockstep, pipelining requires processors to synchronize and communicate with at most two other processors. Thus, pipelining has relaxed wavefronts, allowing some processors to surge ahead while others lag momentarily. This flexibility reduces the time processors spend waiting for other processors and improves parallel performance.

The two pipelining schemes shown above are but two of the many ways in which the computation can be pipelined. As we said, once a loop is fully

permutable, we have a lot of freedom in how we wish to parallelize the code. The first pipeline scheme maps iteration $[i, j]$ to processor i ; the second maps iteration $[i, j]$ to processor j . We can create alternative pipelines by mapping iteration $[i, j]$ to processor $c_0i + c_1j$, provided c_0 and c_1 are positive constants. Such a scheme would create pipelines with relaxed wavefronts between 90° and 180° , both exclusive.

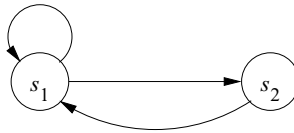
11.9.5 General Theory

The example just completed illustrates the following general theory underlying pipelining: if we can come up with at least two different outermost loops for a loop nest and satisfy all the dependences, then we can pipeline the computation. A loop with k outermost fully permutable loops has $k - 1$ degrees of pipelined parallelism.

Loops that cannot be pipelined do not have alternative outermost loops. Example 11.56 shows one such instance. To honor all the dependences, each iteration in the outermost loop must execute precisely the computation found in the original code. However, such code may still contain parallelism in the inner loops, which can be exploited by introducing at least n synchronizations, where n is the number of iterations in the outermost loop.

```
for (i = 0; i < 100; i++) {
  for (j = 0; j < 100; j++)
    X[j] = X[j] + Y[i,j];    /* (s1) */
  Z[i] = X[A[i]];           /* (s2) */
}
```

(a)



(b)

Figure 11.53: A sequential outer loop (a) and its PDG (b)

Example 11.56: Figure 11.53 is a more complex version of the problem we saw in Example 11.50. As shown in the program dependence graph in Fig. 11.53(b), statements s_1 and s_2 belong to the same strongly connected component. Because we do not know the contents of matrix A , we must assume that the access in statement s_2 may read from any of the elements of X . There is a true dependence from statement s_1 to statement s_2 and an antidependence from

statement s_2 to statement s_1 . There is no opportunity for pipelining either, because all operations belonging to iteration i in the outer loop must precede those in iteration $i + 1$. To find more parallelism, we repeat the parallelization process on the inner loop. The iterations in the second loop can be parallelized without synchronization. Thus, 200 barriers are needed, with one before and one after each execution of the inner loop. \square

11.9.6 Time-Partition Constraints

We now focus on the problem of finding pipelined parallelism. Our goal is to turn a computation into a set of pipelinable tasks. To find pipelined parallelism, we do not solve directly for what is to be executed on each processor, like we did with loop parallelization. Instead, we ask the following fundamental question: What are all the possible execution sequences that honor the original data dependences in the loop? Obviously the original execution sequence satisfies all the data dependences. The question is if there are affine transformations that can create an alternative schedule, where iterations of the outermost loop execute a different set of operations from the original, and yet all the dependences are satisfied. If we can find such transforms, we can pipeline the loop. The key point is that if there is freedom in scheduling operations, there is parallelism; details of how we derive pipelined parallelism from such transforms will be explained later.

To find acceptable reorderings of the outer loop, we wish to find one-dimensional affine transforms, one for each statement, that map the original loop index values to an iteration number in the outermost loop. The transforms are legal if the assignment can satisfy all the data dependences in the program. The “time-partition constraints,” shown below, simply say that if one operation is dependent upon the other, then the first must be assigned an iteration in the outermost loop no earlier than that of the second. If they are assigned in the same iteration, then it is understood that the first will be executed after than the second within the iteration.

An affine-partition mapping of a program is a *legal-time partition* if and only if for every two (not necessarily distinct) accesses sharing a dependence, say

$$\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$$

in statement s_1 , which is nested in d_1 loops, and

$$\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$$

in statement s_2 nested in d_2 loops, the one-dimensional partition mappings $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ and $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ for statements s_1 and s_2 , respectively, satisfy the *time-partition constraints*:

- For all \mathbf{i}_1 in Z^{d_1} and \mathbf{i}_2 in Z^{d_2} such that

$$\text{a) } \mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2,$$

- b) $\mathbf{B}_1 \mathbf{i}_1 + \mathbf{b}_1 \geq \mathbf{0}$,
- c) $\mathbf{B}_2 \mathbf{i}_2 + \mathbf{b}_2 \geq \mathbf{0}$, and
- d) $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$,

it is the case that $\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 \leq \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2$.

This constraint, illustrated in Fig. 11.54, looks remarkably similar to the space-partition constraints. It is a relaxation of the space-partition constraints, in that if two iterations refer to the same location, they do not necessarily have to be mapped to the same partition; we only require that the original relative execution order between the two iterations is preserved. That is, the constraints here have \leq where the space-partition constraints have $=$.

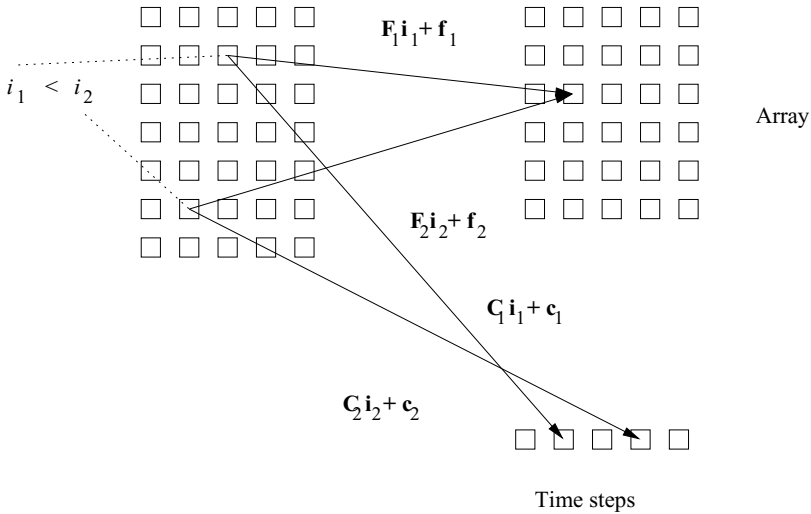


Figure 11.54: Time-Partition Constraints

We know that there exists at least one solution to the time-partition constraints. We can map operations in each iteration of the outermost loop back to the same iteration, and all the data dependences will be satisfied. This solution is the only solution to the time-partition constraints for programs that cannot be pipelined. On the other hand, if we can find several independent solutions to time-partition constraints, the program can be pipelined. Each independent solution corresponds to a loop in the outermost fully permutable nest. For instance, there is only one independent solution to the timing constraints extracted from the program in Example 11.56, where there is no pipelined parallelism. As another instance, there are two independent solutions to the SOR code example of Section 11.9.2.

Example 11.57: Let us consider Example 11.56, and in particular the data dependences of references to array X in statements s_1 and s_2 . Because the

access is not affine in statement s_2 , we approximate the access by modeling matrix X simply as a scalar variable in dependence analysis involving statement s_2 . Let (i, j) be the index value of a dynamic instance of s_1 and let i' be the index value of a dynamic instance of s_2 . Let the computation mappings of statements s_1 , and s_2 be $\langle [C_{11}, C_{12}], c_1 \rangle$ and $\langle [C_{21}], c_2 \rangle$, respectively.

Let us first consider the time-partition constraints imposed by dependences from statement s_1 to s_2 . Thus, $i \leq i'$, the transformed (i, j) th iteration of s_1 must be no later than the transformed i' th iteration of s_2 ; that is,

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + c_1 \leq C_{21}i' + c_2.$$

Expanding, we get

$$C_{11}i + C_{12}j + c_1 \leq C_{21}i' + c_2.$$

Since j can be arbitrarily large, independent of i and i' , it must be that $C_{12} = 0$. Thus, one possible solution to the constraints is

$$C_{11} = C_{21} = 1 \quad \text{and} \quad C_{12} = c_1 = c_2 = 0.$$

Similar arguments about the data dependence from s_2 to s_1 and s_2 back to itself will yield a similar answer. In this particular solution, the i th iteration of the outer loop, which consists of the instance i of s_2 and all instances (i, j) of s_1 , are all assigned to timestep i . Other legal choices of C_{11} , C_{21} , c_1 , and c_2 yield similar assignments, although there might be timesteps at which nothing happens. That is, all ways to schedule the outer loop require the iterations to execute in the same order as in the original code. This statement holds whether all 100 iterations are executed on the same processor, on 100 different processors, or anything in between. \square

Example 11.58: In the SOR code shown in Fig. 11.50(a), the write reference $X[j+1]$ shares a dependence with itself and with the three read references in the code. We are seeking computation mapping $\langle [C_1, C_2], c \rangle$ for the assignment statement such that

$$\begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c \end{bmatrix} \leq \begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} c \end{bmatrix}$$

if there is a dependence from (i, j) to (i', j') . By definition, $(i, j) \prec (i', j')$; that is, either $i < i'$ or $(i = i' \wedge j < j')$.

Let us consider three of the pairs of data dependences:

1. True dependence from write access $X[j+1]$ to read access $X[j+2]$. Since the instances must access the same location, $j+1 = j'+2$ or $j = j'+1$. Substituting $j = j'+1$ into the timing constraints, we get

$$C_1(i' - i) - C_2 \geq 0.$$

Since $j = j' + 1$, $j > j'$, the precedence constraints reduce to $i < i'$. Therefore,

$$C_1 - C_2 \geq 0.$$

2. Antidependence from read access $X[j+2]$ to write access $X[j+1]$. Here, $j+2 = j'+1$, or $j = j'-1$. Substituting $j = j'-1$ into the timing constraints, we get

$$C_1(i' - i) + C_2 \geq 0.$$

When $i = i'$, we get

$$C_2 \geq 0.$$

When $i < i'$, since $C_2 \geq 0$, we get

$$C_1 \geq 0.$$

3. Output dependence from write access $X[j+1]$ back to itself. Here $j = j'$. The timing constraints reduce to

$$C_1(i' - i) \geq 0.$$

Since only $i < i'$ is relevant, we again get

$$C_1 \geq 0.$$

The rest of the dependences do not yield any new constraints. In total, there are three constraints:

$$\begin{aligned} C_1 &\geq 0 \\ C_2 &\geq 0 \\ C_1 - C_2 &\geq 0 \end{aligned}$$

Here are two independent solutions to these constraints:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The first solution preserves the execution order of the iterations in the outermost loop. Both the original SOR code in Fig. 11.50(a) and the transformed code shown in Fig. 11.51(a) are examples of such an arrangement. The second solution places iterations lying along the 135° diagonals in the same outer loop. The code shown in Fig. 11.51(b) is an example of a code with that outermost loop composition.

Notice that there are many other possible pairs of independent solutions. For example,

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

would also be independent solutions to the same constraints. We choose the simplest vectors to simplify code transformation. \square

11.9.7 Solving Time-Partition Constraints by Farkas' Lemma

Since time-partition constraints are similar to space-partition constraints, can we use a similar algorithm to solve them? Unfortunately, the slight difference between the two problems translates into a big technical difference between the two solution methods. Algorithm 11.43 simply solves for $\mathbf{C}_1, \mathbf{c}_1, \mathbf{C}_2$, and \mathbf{c}_2 , such that for all \mathbf{i}_1 in Z^{d_1} and \mathbf{i}_2 in Z^{d_2} if

$$\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$$

then

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2 \mathbf{i}_2 + \mathbf{c}_2.$$

The linear inequalities due to the loop bounds are only used in determining if two references share a data dependence, and are not used otherwise.

To find solutions to the time-partition constraints, we cannot ignore the linear inequalities $\mathbf{i} \prec \mathbf{i}'$; ignoring them often would allow only the trivial solution of placing all iterations in the same partition. Thus, the algorithm to find solutions to the time-partition constraints must handle both equalities and inequalities.

The general problem we wish to solve is: given a matrix \mathbf{A} , find a vector \mathbf{c} such that for all vectors \mathbf{x} such that $\mathbf{Ax} \geq \mathbf{0}$, it is the case that $\mathbf{c}^T \mathbf{x} \geq \mathbf{0}$. In other words, we are seeking \mathbf{c} such that the inner product of \mathbf{c} and any coordinates in the polyhedron defined by the inequalities $\mathbf{Ax} \geq \mathbf{0}$ always yields a nonnegative answer.

This problem is addressed by *Farkas' Lemma*. Let \mathbf{A} be an $m \times n$ matrix of reals, and let \mathbf{c} be a real, nonzero n -vector. Farkas' lemma says that either the *primal* system of inequalities

$$\mathbf{Ax} \geq \mathbf{0}, \quad \mathbf{c}^T \mathbf{x} < \mathbf{0}$$

has a real-valued solution \mathbf{x} , or the *dual* system

$$\mathbf{A}^T \mathbf{y} = \mathbf{c}, \quad \mathbf{y} \geq \mathbf{0}$$

has a real-valued solution \mathbf{y} , but never both.

The dual system can be handled by using Fourier-Motzkin elimination to project away the variables of \mathbf{y} . For each \mathbf{c} that has a solution in the dual system, the lemma guarantees that there are no solutions to the primal system. Put another way, we can prove the negation of the primal system, i.e., we can prove that $\mathbf{c}^T \mathbf{x} \geq \mathbf{0}$ for all \mathbf{x} such that $\mathbf{Ax} \geq \mathbf{0}$, by finding a solution \mathbf{y} to the dual system: $\mathbf{A}^T \mathbf{y} = \mathbf{c}$ and $\mathbf{y} \geq \mathbf{0}$.

Algorithm 11.59: Finding a set of legal, maximally independent affine time-partition mappings for an outer sequential loop.

About Farkas' Lemma

The proof of the lemma can be found in many standard texts on linear programming. Farkas' Lemma, originally proved in 1901, is one of the *theorems of the alternative*. These theorems are all equivalent but, despite attempts over the years, a simple, intuitive proof for this lemma or any of its equivalents has not been found.

INPUT: A loop nest with array accesses.

OUTPUT: A maximal set of linearly independent time-partition mappings.

METHOD: The following steps constitute the algorithm:

1. Find all data-dependent pairs of accesses in a program.
2. For each pair of data-dependent accesses, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ in statement s_1 nested in d_1 loops and $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ in statement s_2 nested in d_2 loops, let $\langle \mathbf{C}_1, c_1 \rangle$ and $\langle \mathbf{C}_2, c_2 \rangle$ be the (unknown) time-partition mappings of statements s_1 and s_2 , respectively. Recall the time-partition constraints state that

- For all \mathbf{i}_1 in Z^{d_1} and \mathbf{i}_2 in Z^{d_2} such that
 - a) $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$,
 - b) $\mathbf{B}_1 \mathbf{i}_1 + \mathbf{b}_1 \geq \mathbf{0}$,
 - c) $\mathbf{B}_2 \mathbf{i}_2 + \mathbf{b}_2 \geq \mathbf{0}$, and
 - d) $\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$,

it is the case that $\mathbf{C}_1 \mathbf{i}_1 + c_1 \leq \mathbf{C}_2 \mathbf{i}_2 + c_2$.

Since $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ is a disjunctive union of a number of clauses, we can create a system of constraints for each clause and solve each of them separately, as follows:

- (a) Similarly to step (2a) in Algorithm 11.43, apply Gaussian elimination to the equations

$$\mathbf{F}_1 \mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2 \mathbf{i}_2 + \mathbf{f}_2$$

to reduce the vector

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix}$$

to some vector of unknowns, \mathbf{x} .

- (b) Let \mathbf{c} be all the unknowns in the partition mappings. Express the linear inequality constraints due to the partition mappings as

$$\mathbf{c}^T \mathbf{D} \mathbf{x} \geq 0$$

for some matrix \mathbf{D} .

- (c) Express the precedence constraints on the loop index variables and the loop bounds as

$$\mathbf{A} \mathbf{x} \geq 0$$

for some matrix \mathbf{A} .

- (d) Apply Farkas' Lemma. Finding \mathbf{x} to satisfy the two constraints above is equivalent to finding \mathbf{y} such that

$$\mathbf{A}^T \mathbf{y} = \mathbf{D}^T \mathbf{c} \text{ and } \mathbf{y} \geq 0.$$

Note that $\mathbf{c}^T \mathbf{D}$ here is \mathbf{c}^T in the statement of Farkas' Lemma, and we are using the negated form of the lemma.

- (e) In this form, apply Fourier-Motzkin elimination to project away the \mathbf{y} variables, and express the constraints on the coefficients \mathbf{c} as $\mathbf{E} \mathbf{c} \geq 0$.
- (f) Let $\mathbf{E}' \mathbf{c}' \geq 0$ be the system without the constant terms.
3. Find a maximal set of linearly independent solutions to $\mathbf{E}' \mathbf{c}' \geq 0$ using Algorithm B.1 in Appendix B. The approach of that complex algorithm is to keep track of the current set of solutions for each of the statements, then incrementally look for more independent solutions by inserting constraints that force the solution to be linearly independent for at least one statement.
4. From each solution of \mathbf{c}' found, derive one affine time-partition mapping. The constant terms are derived using $\mathbf{E} \mathbf{c} \geq 0$.

□

Example 11.60: The constraints for Example 11.57 can be written as

$$\begin{bmatrix} -C_{11} & -C_{12} & C_{21} & (c_2 - c_1) \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geq 0$$

$$\begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geq 0$$

Farkas' lemma says that these constraints are equivalent to

$$\begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix} [z] = \begin{bmatrix} -C_{11} \\ -C_{12} \\ C_{21} \\ c_2 - c_1 \end{bmatrix} \text{ and } z \geq 0.$$

Solving this system, we get

$$C_{11} = C_{21} \geq 0 \text{ and } C_{12} = c_2 - c_1 = 0.$$

Notice that these constraints are satisfied by the particular solution we obtained in Example 11.57. \square

11.9.8 Code Transformations

If there exist k independent solutions to the time-partition constraints of a loop nest, then it is possible to transform the loop nest to have k outermost fully permutable loops, which can be transformed to create $k-1$ degrees of pipelining, or to create $k-1$ inner parallelizable loops. Furthermore, we can apply blocking to fully permutable loops to improve data locality of uniprocessors as well as reducing synchronization among processors in a parallel execution.

Exploiting Fully Permutable Loops

We can create a loop nest with k outermost fully permutable loops easily from k independent solutions to the time-partition constraints. We can do so by simply making the k th solution the k th row of the new transform. Once the affine transform is created, Algorithm 11.45 can be used to generate the code.

Example 11.61: The solutions found in Example 11.58 for our SOR example were

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Making the first solution the first row and the second solution the second row, we get the transform

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

which yields the code in Fig. 11.51(a).

Making the second solution the first row instead, we get the transform

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

which yields the code in Fig. 11.51(c). \square

It is easy to see that such transforms produce a legal sequential program. The first row partitions the entire iteration space according to the first solution. The timing constraints guarantee that such a decomposition does not violate any data dependences. Then, we partition the iterations in each of the outermost loop according to the second solution. Again this must be legal because we are dealing with just subsets of the original iteration space. The same goes for the rest of the rows in the matrix. Since we can order the solutions arbitrarily, the loops are fully permutable.

Exploiting Pipelining

We can easily transform a loop with k outermost fully permutable loops into a code with $k - 1$ degrees of pipeline parallelism.

Example 11.62: Let us return to our SOR example. After the loops are transformed to be fully permutable, we know that iteration $[i_1, i_2]$ can be executed provided iterations $[i_1, i_2 - 1]$ and $[i_1 - 1, i_2]$ have been executed. We can guarantee this order in a pipeline as follows. We assign iteration i_1 to processor p_1 . Each processor executes iterations in the inner loop in the original sequential order, thus guaranteeing that iteration $[i_1, i_2]$ executes after $[i_1, i_2 - 1]$. In addition, we require that processor p waits for the signal from processor $p - 1$ that it has executed iteration $[p - 1, i_2]$ before it executes iteration $[p, i_2]$. This technique generates the pipelined code Fig. 11.52(a) and (b) from the fully permutable loops Fig. 11.51(a) and (c), respectively. \square

In general, given k outermost fully permutable loops, the iteration with index values (i_1, \dots, i_k) can be executed without violating data-dependence constraints, provided iterations

$$[i_1 - 1, i_2, \dots, i_k], [i_1, i_2 - 1, i_3, \dots, i_k], \dots, [i_1, \dots, i_{k-1}, i_k - 1]$$

have been executed. We can thus assign the partitions of the first $k - 1$ dimensions of the iteration space to $O(n^{k-1})$ processors as follows. Each processor is responsible for one set of iterations whose indexes agree in the first $k - 1$ dimensions, and vary over all values of the k th index. Each processor executes the iterations in the k th loop sequentially. The processor corresponding to values $[p_1, p_2, \dots, p_{k-1}]$ for the first $k - 1$ loop indexes can execute iteration i in the k th loop as long as it receives a signal from processors

$$[p_1 - 1, p_2, \dots, p_{k-1}], \dots, [p_1, \dots, p_{k-2}, p_{k-1} - 1]$$

that they have executed their i th iteration in the k th loop.

Wavefronting

It is also easy to generate $k - 1$ inner parallelizable loops from a loop with k outermost fully permutable loops. Although pipelining is preferable, we include this information here for completeness.

We partition the computation of a loop with k outermost fully permutable loops using a new index variable i' , where i' is defined to be some combination of all the indices in the k permutable loop nest. For example, $i' = i_1 + \dots + i_k$ is one such combination.

We create an outermost sequential loop that iterates through the i' partitions in increasing order; the computation nested within each partition is ordered as before. The first $k - 1$ loops within each partition are guaranteed to be parallelizable. Intuitively, if given a two-dimensional iteration space, this transform groups iterations along 135° diagonals as an execution of the outermost loop. This strategy guarantees that iterations within each iteration of the outermost loop have no data dependence.

Blocking

A k -deep, fully permutable loop nest can be blocked in k -dimensions. Instead of assigning the iterations to processors based on the value of the outer or inner loop indexes, we can aggregate blocks of iterations into one unit. Blocking is useful for enhancing data locality as well as for minimizing the overhead of pipelining.

Suppose we have a two-dimensional fully permutable loop nest, as in Fig. 11.55(a), and we wish to break the computation into $b \times b$ blocks. The execution order of the blocked code is shown in Fig. 11.56, and the equivalent code is in Fig. 11.55(b).

If we assign each block to one processor, then all the passing of data from one iteration to another that is within a block requires no interprocessor communication. Alternatively, we can coarsen the granularity of pipelining by assigning a column of blocks to one processor. Notice that each processor synchronizes with its predecessors and successors only at block boundaries. Thus, another advantage of blocking is that programs only need to communicate data accessed at the boundaries of the block with their neighbor blocks. Values that are interior to a block are managed by only one processor.

Example 11.63: We now use a real numerical algorithm — Cholesky decomposition — to illustrate how Algorithm 11.59 handles single loop nests with only pipelining parallelism. The code, shown in Fig. 11.57, implements an $O(n^3)$ algorithm, operating on a 2-dimensional data array. The executed iteration space is a triangular pyramid, since j only iterates up to the value of the outer loop index i , and k only iterates to the value of j . The loop has four statements, all nested in different loops.

Applying Algorithm 11.59 to this program finds three legitimate time dimensions. It nests all the operations, some of which were originally nested in 1- and 2-deep loop nests into a 3-dimensional, fully permutable loop nest. The code, together with the mappings, is shown in Fig. 11.58.

The code-generation routine guards the execution of the operations with the original loop bounds to ensure that the new programs execute only operations

```

for (i=0; i<n; i++)
    for (j=1; j<n; j++) {
        <S>
    }

```

(a) A simple loop nest.

```

for (ii = 0; ii<n; ii+=b)
    for (jj = 0; jj<n; jj+=b)
        for (i = ii*b; i <= min(ii*b-1, n); i++)
            for (j = ii*b; j <= min(jj*b-1, n); j++) {
                <S>
            }

```

(b) A blocked version of this loop nest.

Figure 11.55: A 2-dimensional loop nest and its blocked version

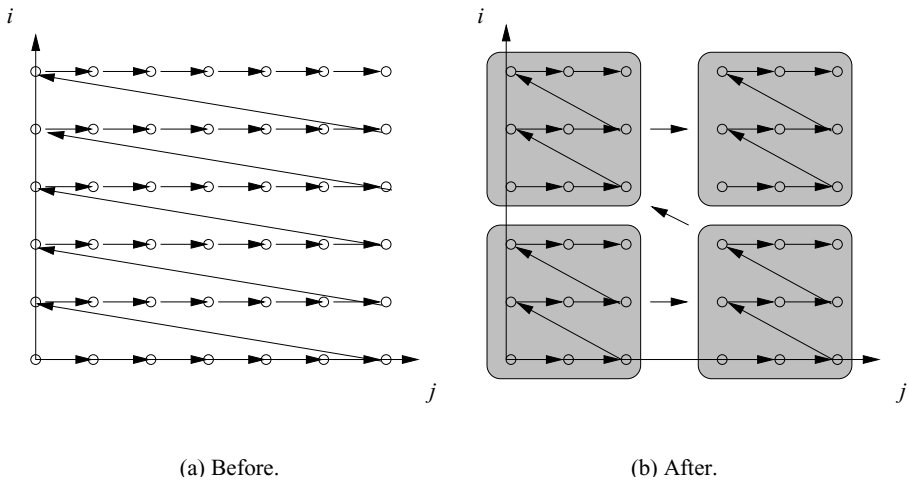


Figure 11.56: Execution order before and after blocking a 2-deep loop nest.

```

for (i = 1; i <= N; i++) {
    for (j = 1; j <= i-1; j++) {
        for (k = 1; k <= j-1; k++)
            X[i,j] = X[i,j] - X[i,k] * X[j,k];
        X[i,j] = X[i,j] / X[j,j];
    }
    for (m = 1; m <= i-1; m++)
        X[i,i] = X[i,i] - X[i,m] * X[i,m];
    X[i,i] = sqrt(X[i,i]);
}

```

Figure 11.57: Cholesky decomposition

```

for (i2 = 1; i2 <= N; i2++)
    for (j2 = 1; j2 <= i2; j2++) {
        /* beginning of code for processor (i2,j2) */
        for (k2 = 1; k2 <= i2; k2++) {

            // Mapping: i2 = i, j2 = j, k2 = k
            if (j2 < i2 && k2 < j2)
                X[i2,j2] = X[i2,j2] - X[i2,k2] * X[j2,k2];

            // Mapping: i2 = i, j2 = j, k2 = j
            if (j2 == k2 && j2 < i2)
                X[i2,j2] = X[i2,j2] / X[j2,j2];

            // Mapping: i2 = i, j2 = i, k2 = m
            if (i2 == j2 && k2 < i2)
                X[i2,i2] = X[i2,i2] - X[i2,k2] * X[i2,k2];

            // Mapping: i2 = i, j2 = i, k2 = i
            if (i2 == j2 && j2 == k2)
                X[k2,k2] = sqrt(X[k2,k2]);
        }
        /* ending of code for processor (i2,j2) */
    }
}

```

Figure 11.58: Figure 11.57 written as a fully permutable loop nest

that are in the original code. We can pipeline this code by mapping the 3-dimensional structure to a 2-dimensional processor space. Iterations $(i2, j2, k2)$ are assigned to the processor with ID $(i2, j2)$. Each processor executes the innermost loop, the loop with the index $k2$. Before it executes the k th iteration, the processor waits for signals from the processors with ID's $(i2 - 1, j2)$ and $(i2, j2 - 1)$. After it executes its iteration, it signals processors $(i2 + 1, j2)$ and $(i2, j2 + 1)$. \square

11.9.9 Parallelism With Minimum Synchronization

We have described three powerful parallelization algorithms in the last three sections: Algorithm 11.43 finds all parallelism requiring no synchronizations, Algorithm 11.54 finds all parallelism requiring only a constant number of synchronizations, and Algorithm 11.59 finds all the pipelinable parallelism requiring $O(n)$ synchronizations where n is the number of iterations in the outermost loop. As a first approximation, our goal is to parallelize as much of the computation as possible, while introducing as little synchronization as necessary.

Algorithm 11.64, below, finds all the degrees of parallelism in a program, starting with the coarsest granularity of parallelism. In practice, to parallelize a code for a multiprocessor, we do not need to exploit all the levels of parallelism, just the outermost possible ones until all the computation is parallelized and all the processors are fully utilized.

Algorithm 11.64: Find all the degrees of parallelism in a program, with all the parallelism being as coarse-grained as possible.

INPUT: A program to be parallelized.

OUTPUT: A parallelized version of the same program.

METHOD: Do the following:

1. Find the maximum degree of parallelism requiring no synchronization: Apply Algorithm 11.43 to the program.
2. Find the maximum degree of parallelism that requires $O(1)$ synchronizations: Apply Algorithm 11.54 to each of the space partitions found in step 1. (If no synchronization-free parallelism is found, the whole computation is left in one partition).
3. Find the maximum degree of parallelism that requires $O(n)$ synchronizations. Apply Algorithm 11.59 to each of the partitions found in step 2 to find pipelined parallelism. Then apply Algorithm 11.54 to each of the partitions assigned to each processor, or the body of the sequential loop if no pipelining is found.
4. Find the maximum degree of parallelism with successively greater degrees of synchronizations: Recursively apply Step 3 to computation belonging to each of the space partitions generated by the previous step.

□

Example 11.65: Let us now return to Example 11.56. No parallelism is found by Steps 1 and 2 of Algorithm 11.64; that is, we need more than a constant number of synchronizations to parallelize this code. In Step 3, applying Algorithm 11.59 determines that there is only one legal outer loop, which is the one in the original code of Fig. 11.53. So, the loop has no pipelined parallelism. In the second part of Step 3, we apply Algorithm 11.54 to parallelize the inner loop. We treat the code within a partition like a whole program, the only difference being that the partition number is treated like a symbolic constant. In this case the inner loop is found to be parallelizable and therefore the code can be parallelized with n synchronization barriers. □

Algorithm 11.64 finds all the parallelism in a program at each level of synchronization. The algorithm prefers parallelization schemes that have less synchronization, but less synchronization does not mean that the communication is minimized. Here we discuss two extensions to the algorithm to address its weaknesses.

Considering Communication Cost

Step 2 of Algorithm 11.64 parallelizes each strongly connected component independently if no synchronization-free parallelism is found. However, it may be possible to parallelize a number of the components without synchronization and communication. One solution is to greedily find synchronization-free parallelism among subsets of the program dependence graph that share the most data.

If communication is necessary between strongly connected components, we note that some communication is more expensive than others. For example, the cost of transposing a matrix is significantly higher than just having to communicate between neighboring processors. Suppose s_1 and s_2 are statements in two separate strongly connected components accessing the same data in iterations \mathbf{i}_1 and \mathbf{i}_2 , respectively. If we cannot find partition mappings $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ and $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ for statements s_1 and s_2 , respectively, such that

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 - \mathbf{C}_2 \mathbf{i}_2 - \mathbf{c}_2 = \mathbf{0},$$

we instead try to satisfy the constraint

$$\mathbf{C}_1 \mathbf{i}_1 + \mathbf{c}_1 - \mathbf{C}_2 \mathbf{i}_2 - \mathbf{c}_2 \leq \delta$$

where δ is a small constant.

Trading Communication for Synchronization

Sometimes we would rather perform more synchronizations to minimize communication. Example 11.66 discusses one such example. Thus, if we cannot

parallelize a code with just neighborhood communication among strongly connected components, we should attempt to pipeline the computation instead of parallelizing each component independently. As shown in Example 11.66, pipelining can be applied to a sequence of loops.

Example 11.66: For the ADI integration algorithm in Example 11.49, we have shown that optimizing the first and second loop nests independently finds parallelism in each of the nests. However, such a scheme would require that the matrix be transposed between the loops, incurring $O(n^2)$ data traffic. If we use Algorithm 11.59 to find pipelined parallelism, we find that we can turn the entire program into a fully permutable loop nest, as in Fig. 11.59. We then can apply blocking to reduce the communication overhead. This scheme would incur $O(n)$ synchronizations but would require much less communication. \square

```

for (j = 0; j < n; j++)
    for (i = 1; i < n+1; i++) {
        if (i < n) X[i,j] = f(X[i,j] + X[i-1,j])
        if (j > 0) X[i-1,j] = g(X[i-1,j],X[i-1,j-1]);
    }

```

Figure 11.59: A fully permutable loop nest for the code of Example 11.49

11.9.10 Exercises for Section 11.9

Exercise 11.9.1: In Section 11.9.4, we discussed the possibility of using diagonals other than the horizontal and vertical axes to pipeline the code of Fig. 11.51. Write code analogous to the loops of Fig. 11.52 for the diagonals: (a) 135° (b) 120° .

Exercise 11.9.2: Figure 11.55(b) can be simplified if b divides n evenly. Rewrite the code under that assumption.

```

for (i=0; i<100; i++) {
    P[i,0] = 1; /* s1 */
    P[i,i] = 1; /* s2 */
}
for (i=2; i<100; i++)
    for (j=1; j<i; j++)
        P[i,j] = P[i-1,j-1] + P[i-1,j]; /* s3 */

```

Figure 11.60: Computing Pascal's triangle

Exercise 11.9.3: In Fig. 11.60 is a program to compute the first 100 rows of Pascal's triangle. That is, $P[i, j]$ will become the number of ways to choose j things out of i , for $0 \leq j \leq i < 100$.

- a) Rewrite the code as a single, fully permutable loop nest.
- b) Use 100 processors in a pipeline to implement this code. Write the code for each processor p , in terms of p , and indicate the synchronization necessary.
- c) Rewrite the code using square blocks of 10 iterations on a side. Since the iterations form a triangle, there will be only $1 + 2 + \cdots + 10 = 55$ blocks. Show the code for a processor (p_1, p_2) assigned to the p_1 th block in the i direction and the p_2 th block in the j direction, in terms of p_1 and p_2 .

```

for (i=0; i<100; i++) {
    A[i, 0,0] = B1[i]; /* s1 */
    A[i,99,0] = B2[i]; /* s2 */
}
for (j=1; j<99; j++) {
    A[ 0,j,0] = B3[j]; /* s3 */
    A[99,j,0] = B4[j]; /* s4 */
}
for (i=0; i<99; i++)
    for (j=0; j<99; j++)
        for (k=1; k<100; k++)
            A[i,j,k] = (A[i,j,k-1] + A[i-1,j,k-1] +
                        A[i+1,j,k-1] + A[i,j-1,k-1] +
                        A[i,j+1,k-1])/5; /* s5 */

```

Figure 11.61: Code for Exercise 11.9.4

! Exercise 11.9.4: Repeat Exercise 11.9.2 for the code of Fig. 11.61. However, note that the iterations for this problem form a 3-dimensional cube of side 100. Thus, the blocks for part (c) should be $10 \times 10 \times 10$, and there are 1000 of them.

! Exercise 11.9.5: Let us apply Algorithm 11.59 to a simple example of the time-partition constraints. In what follows, assume that the vector \mathbf{i}_1 is (i_1, j_1) , and vector \mathbf{i}_2 is (i_2, j_2) ; technically, both these vectors are transposed. The condition $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ consists of the following disjunction:

- i. $i_1 < i_2$, or
- ii. $i_1 = i_2$ and $j_1 < j_2$.

The other equalities and inequalities are

$$\begin{array}{rcl}
2i_1 + j_1 - 10 & \geq & 0 \\
i_2 + 2j_2 - 20 & \geq & 0 \\
i_1 & = & i_2 + j_2 - 50 \\
j_1 & = & j_2 + 40
\end{array}$$

Finally, the time-partition inequality, with unknowns c_1 , d_1 , e_1 , c_2 , d_2 , and e_2 , is

$$c_1 i_1 + d_1 j_1 + e_1 \leq c_2 i_2 + d_2 j_2 + e_2.$$

- a) Solve the time-partition constraints for case i — that is, where $i_1 < i_2$. In particular, eliminate as many of i_1 , j_1 , i_2 , and j_2 as you can, and set up the matrices D and A as in Algorithm 11.59. Then, apply Farkas' Lemma to the resulting matrix inequalities.
- b) Repeat part (a) for the case ii , where $i_1 = i_2$ and $j_1 < j_2$.

11.10 Locality Optimizations

The performance of a processor, be it a part of a multiprocessor or not, is highly sensitive to its cache behavior. Misses in the cache can take tens of clock cycles, so high cache-miss rates can lead to poor processor performance. In the context of a multiprocessor with a common memory bus, contention on the bus can further add to the penalty of poor data locality.

As we shall see, even if we just wish to improve the locality of uniprocessors, the affine-partitioning algorithm for parallelization is useful as a means of identifying opportunities for loop transformations. In this section, we describe three techniques for improving data locality in uniprocessors and multiprocessors.

1. We improve the temporal locality of computed results by trying to use the results as soon as they are generated. We do so by dividing a computation into independent partitions and executing all the dependent operations in each partition close together.
2. *Array contraction* reduces the dimensions of an array and reduces the number of memory locations accessed. We can apply array contraction if only one location of the array is used at a given time.
3. Besides improving temporal locality of computed results, we also need to optimize for the spatial locality of computed results, and for both the temporal and spatial locality of read-only data. Instead of executing each partition one after the other, we interleave a number of the partitions so that reuses among partitions occur close together.

11.10.1 Temporal Locality of Computed Data

The affine-partitioning algorithm pulls all the dependent operations together; by executing these partitions serially we improve temporal locality of computed data. Let us return to the multigrid example discussed in Section 11.7.1. Applying Algorithm 11.43 to parallelize the code in Fig 11.23 finds two degrees of parallelism. The code in Fig 11.24 contains two outer loops that iterate through the independent partitions serially. This transformed code has improved temporal locality, since computed results are used immediately in the same iteration.

Thus, even if our goal is to optimize for sequential execution, it is profitable to use parallelization to find these related operations and place them together. The algorithm we use here is similar to that of Algorithm 11.64, which finds all the granularities of parallelism starting with the outermost loop. As discussed in Section 11.9.9, the algorithm parallelizes strongly connected components individually, if we cannot find synchronization-free parallelism at each level. This parallelization tends to increase communication. Thus, we combine separately parallelized strongly connected components greedily, if they share reuse.

11.10.2 Array Contraction

The optimization of array contraction provides another illustration of the trade-off between storage and parallelism, which was first introduced in the context of instruction-level parallelism in Section 10.2.3. Just as using more registers allows for more instruction-level parallelism, using more memory allows for more loop-level parallelism. As shown in the multigrid example in Section 11.7.1, expanding a temporary scalar variable into an array allows different iterations to keep different instances of the temporary variables and to execute at the same time. Conversely, when we have a sequential execution that operates on one array element at a time serially, we can contract the array, replace it with a scalar, and have each iteration use the same location.

In the transformed multigrid program shown in Fig. 11.24, each iteration of the inner loop produces and consumes a different element of AP , AM , T , and a row of D . If these arrays are not used outside of the code excerpt, the iterations can serially reuse the same data storage instead of putting the values in different elements and rows, respectively. Figure 11.62 shows the result of reducing the dimensionality of the arrays. This code runs faster than the original, because it reads and writes less data. Especially in the case when an array is reduced to a scalar variable, we can allocate the variable to a register and eliminate the need to access memory altogether.

As less storage is used, less parallelism is available. Iterations in the transformed code in Fig. 11.62 now share data dependences and no longer can be executed in parallel. To parallelize the code on P processors, we can expand each of the scalar variables by a factor of P and have each processor access its own private copy. Thus, the amount by which the storage is expanded is

```

for (j = 2, j <= jl, j++)
  for (i = 2, i <= il, i++) {
    AP          = ...;
    T           = 1.0/(1.0 +AP);
    D[2]        = T*AP;
    DW[1,2,j,i] = T*DW[1,2,j,i];
    for (k=3, k <= kl-1, k++) {
      AM        = AP;
      AP        = ...;
      T         = ...AP -AM*D[k-1]...;
      D[k]      = T*AP;
      DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...;
    }
    ...
    for (k=kl-1, k>=2, k--)
      DW[1,k,j,i] = DW[1,k,j,i] +D[k]*DW[1,k+1,j,i];
  }

```

Figure 11.62: Code of Fig. 11.23 after partitioning (Fig. 11.24) and array contraction

directly correlated to the amount of parallelism exploited.

There are three reasons it is common to find opportunities for array contraction:

1. Higher-level programming languages for scientific applications, such as Matlab and Fortran 90, support array-level operations. Each subexpression of array operations produces a temporary array. Because the arrays can be large, every array operation such as a multiply or add would require reading and writing many memory locations, while requiring relatively few arithmetic operations. It is important that we reorder operations so that data is consumed as it is produced and that we contract these arrays into scalar variables.
2. Supercomputers built in the 80's and 90's are all vector machines, so many scientific applications developed then have been optimized for such machines. Even though vectorizing compilers exist, many programmers still write their code to operate on vectors at a time. The multigrid code example of this chapter is an example of this style.
3. Opportunities for contraction are also introduced by the compiler. As illustrated by variable T in the multigrid example, a compiler would expand arrays to improve parallelization. We have to contract them when the space expansion is not necessary.

Example 11.67: The array expression $Z = W + X + Y$ translates to

```

for (i=0; i<n; i++) T[i] = W[i] + X[i];
for (i=0; i<n; i++) Z[i] = T[i] + Y[i];

```

Rewriting the code as

```

for (i=0; i<n; i++) { T = W[i] + X[i]; Z[i] = T + Y[i] }

```

can speed it up considerably. Of course at the level of C code, we would not even have to use the temporary T , but could write the assignment to $Z[i]$ as a single statement. However, here we are trying to model the intermediate-code level at which a vector processor would deal with the operations. \square

Algorithm 11.68: Array contraction.

INPUT: A program transformed by Algorithm 11.64.

OUTPUT: An equivalent program with reduced array dimensions.

METHOD: A dimension of an array can be contracted to a single element if

1. Each independent partition uses only one element of the array,
2. The value of the element upon entry to the partition is not used by the partition, and
3. The value of the element is not live on exit from the partition.

Identify the contractable dimensions — those that satisfy the three conditions above — and replace them with a single element. \square

Algorithm 11.68 assumes that the program has first been transformed by Algorithm 11.64 to pull all the dependent operations into a partition and execute the partitions sequentially. It finds those array variables whose elements' live ranges in different iterations are disjoint. If these variables are not live after the loop, it contracts the array and has the processor operate on the same scalar location. After array contraction, it may be necessary to selectively expand arrays to accommodate for parallelism and other locality optimizations.

The liveness analysis required here is more complex than that described in Section 9.2.5. If the array is declared as a global variable, or if it is a parameter, interprocedural analysis is required to ensure that the value on exit is not used. Furthermore, we need to compute the liveness of individual array elements, conservatively treating the array as a scalar would be too imprecise.

11.10.3 Partition Interleaving

Different partitions in a loop often read the same data, or read and write the same cache lines. In this and the next two sections, we discuss how to optimize for locality when reuse is found across partitions.

Reuse in Innermost Blocks

We adopt the simple model that data can be found in the cache if it is reused within a small number of iterations. If the innermost loop has a large or unknown bound, only reuse across iterations of the innermost loop translates into a locality benefit. Blocking creates inner loops with small known bounds, allowing reuse within and across entire blocks of computation to be exploited. Thus, blocking has the effect of capitalizing on more dimensions of reuse.

Example 11.69: Consider the matrix-multiply code shown in Fig. 11.5 and its blocked version in Fig. 11.7. Matrix multiplication has reuse along every dimension of its three-dimensional iteration space. In the original code, the innermost loop has n iterations, where n is unknown and can be large. Our simple model assumes that only the data reused across iterations in the innermost loop is found in the cache.

In the blocked version, the three innermost loops execute a three-dimensional block of computation, with B iterations on each side. The block size B is chosen by the compiler to be small enough so that all the cache lines read and written within the block of computation fit into the cache. Thus reused data across iterations in the third outermost loop can be found in the cache. \square

We refer to the innermost set of loops with small known bounds as the *innermost block*. It is desirable that the innermost block include all the dimensions of the iteration space that carry reuse, if possible. Maximizing the lengths of each side of the block is not as important. For the matrix-multiply example, 3-dimensional blocking reduces the amount of data accessed for each matrix by a factor of B^2 . If reuse is present, it is better to accommodate higher-dimensional blocks with shorter sides than lower-dimensional blocks with longer sides.

We can optimize locality of the innermost fully permutable loop nest by blocking the subset of loops that share reuse. We can generalize the notion of blocking to exploit reuses found among iterations of outer parallel loops, also. Observe that blocking primarily interleaves the execution of a small number of instances of the innermost loop. In matrix multiplication, each instance of the innermost loop computes one element of the array answer; there are n^2 of them. Blocking interleaves the execution of a block of instances, computing B iterations from each instance at a time. Similarly, we can interleave iterations in parallel loops to take advantage of reuses between them.

We define two primitives below that can reduce the distance between reuses across different iterations. We apply these primitives repeatedly, starting from the outermost loop until all the reuses are moved adjacent to each other in the innermost block.

Interleaving Inner Loops in a Parallel Loop

Consider the case where an outer parallelizable loop contains an inner loop. To exploit reuse across iterations of the outer loop, we interleave the executions of

a fixed number of instances of the inner loop, as shown in Fig. 11.63. Creating two-dimensional inner blocks, this transformation reduces the distance between reuse of consecutive iterations of the outer loop.

<pre>for (i=0; i<n; i++) for (j=0; j<n; j++) <S></pre>	<pre>for (ii=0; ii<n; ii+=4) for (j=0; j<n; j++) for (i=ii; i<min(n, ii+4); i++) <S></pre>
--	---

(a) Source program.

(b) Transformed code.

Figure 11.63: Interleaving 4 instances of the inner loop

The step that turns a loop

```
for (i=0; i<n; i++)
  <S>
```

into

```
for (ii=0; ii<n; ii+=4)
  for (i=ii; i<min(n, ii+4); i++)
    <S>
```

is known as *stripmining*. In the case where the outer loop in Fig. 11.63 has a small known bound, we need not stripmine it, but can simply permute the two loops in the original program.

Interleaving Statements in a Parallel Loop

Consider the case where a parallelizable loop contains a sequence of statements s_1, s_2, \dots, s_m . If some of these statements are loops themselves, statements from consecutive iterations may still be separated by many operations. We can exploit reuse between iterations by again interleaving their executions, as shown in Fig. 11.64. This transformation *distributes* a stripmined loop across the statements. Again, if the outer loop has a small fixed number of iterations, we need not stripmine the loop but simply distribute the original loop over all the statements.

We use $s_i(j)$ to denote the execution of statement s_i in iteration j . Instead of the original sequential execution order shown in Fig. 11.65(a), the code executes in the order shown in Fig. 11.65(b).

Example 11.70: We now return to the multigrid example and show how we exploit reuse between iterations of outer parallel loops. We observe that references $DW[1, k, j, i]$, $DW[1, k-1, j, i]$, and $DW[1, k+1, j, i]$ in the innermost loops of the code in Fig. 11.62 have rather poor spatial locality. From reuse analysis, as discussed in Section 11.5, the loop with index i carries spatial

<pre> for (i=0; i<n; i++) { <S1> <S2> ... } </pre>	<pre> for (ii=0; ii<n; ii+=4) { for (i=ii; i<min(n,ii+4); i++) <S1> for (i=ii; i<min(n,ii+4); i++) <S2> ... } </pre>
---	---

(a) Source program.

(b) Transformed code.

Figure 11.64: The statement-interleaving transformation

locality and the loop with index k carries group reuse. The loop with index k is already the innermost loop, so we are interested in interleaving operations on DW from a block of partitions with consecutive i values.

We apply the transform to interleave statements in the loop to obtain the code in Fig. 11.66, then apply the transform to interleave inner loops to obtain the code in Fig. 11.67. Notice that as we interleave B iterations from loop with index i , we need to expand variables AP, AM, T into arrays that hold B results at a time. \square

11.10.4 Putting it All Together

Algorithm 11.71 optimizes locality for a uniprocessor, and Algorithm 11.72 optimizes both parallelism and locality for a multiprocessor.

Algorithm 11.71: Optimize data locality on a uniprocessor.

INPUT: A program with affine array accesses.

OUTPUT: An equivalent program that maximizes data locality.

METHOD: Do the following steps:

1. Apply Algorithm 11.64 to optimize the temporal locality of computed results.
2. Apply Algorithm 11.68 to contract arrays where possible.
3. Determine the iteration subspace that may share the same data or cache lines using the technique described in Section 11.5. For each statement, identify those outer parallel loop dimensions that have data reuse.
4. For each outer parallel loop carrying reuse, move a block of the iterations into the innermost block by applying the interleaving primitives repeatedly.

$$\begin{array}{cccc}
s_1(0), & s_2(0), & \dots, & s_m(0), \\
s_1(1), & s_2(1), & \dots, & s_m(1), \\
s_1(2), & s_2(2), & \dots, & s_m(2), \\
s_1(3), & s_2(3), & \dots, & s_m(3), \\
s_1(4), & s_2(4), & \dots, & s_m(4), \\
s_1(5), & s_2(5), & \dots, & s_m(5), \\
s_1(6), & s_2(6), & \dots, & s_m(6), \\
s_1(7), & s_2(7), & \dots, & s_m(7), \\
\dots, & & &
\end{array}$$

(a) Original order.

$$\begin{array}{cccc}
s_1(0), & s_1(1), & s_1(2), & s_1(3), \\
s_2(0), & s_2(1), & s_2(2), & s_2(3), \\
\dots, & & & \\
s_m(0), & s_m(1), & s_m(2), & s_m(3), \\
s_1(4), & s_1(5), & s_1(6), & s_1(7), \\
s_2(4), & s_2(5), & s_2(6), & s_2(7), \\
\dots, & & & \\
s_m(4), & s_m(5), & s_m(6), & s_m(7), \\
\dots, & & &
\end{array}$$

(b) Transformed order.

Figure 11.65: Distributing a stripmined loop

5. Apply blocking to the subset of dimensions in the innermost fully permutable loop nest that carries reuse.
6. Block outer fully permutable loop nest for higher levels of memory hierarchies, such as the third-level cache or the physical memory.
7. Expand scalars and arrays where necessary by the lengths of the blocks.

□

Algorithm 11.72: Optimize parallelism and data locality for multiprocessors.

INPUT: A program with affine array accesses.

OUTPUT: An equivalent program that maximizes parallelism and data locality.

METHOD: Do the following:

1. Use the Algorithm 11.64 to parallelize the program and create an SPMD program.

```

for (j = 2, j <= jl, j++)
  for (ii = 2, ii <= il, ii+=b) {
    for (i = ii; i <= min(ii+b-1,il); i++) {
      ib          = i-ii+1;
      AP[ib]      = ...;
      T           = 1.0/(1.0 +AP[ib]);
      D[2,ib]     = T*AP[ib];
      DW[1,2,j,i] = T*DW[1,2,j,i];
    }
    for (i = ii; i <= min(ii+b-1,il); i++) {
      for (k=3, k <= kl-1, k++)
        ib          = i-ii+1;
        AM          = AP[ib];
        AP[ib]      = ...;
        T           = ...AP[ib]-AM*D[ib,k-1]...;
        D[ib,k]     = T*AP;
        DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...;
      }
    ...
    for (i = ii; i <= min(ii+b-1,il); i++)
      for (k=kl-1, k>=2, k--) {
        DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
        /* Ends code to be executed by processor (j,i) */
      }
  }
}

```

Figure 11.66: Excerpt of Fig. 11.23 after partitioning, array contraction, and blocking

2. Apply Algorithm 11.71 to the SPMD program produced in Step 1 to optimize its locality.

□

11.10.5 Exercises for Section 11.10

Exercise 11.10.1: Perform array contraction on the following vector operations:

```

for (i=0; i<n; i++) T[i] = A[i] * B[i];
for (i=0; i<n; i++) D[i] = T[i] + C[i];

```

Exercise 11.10.2: Perform array contraction on the following vector operations:

```

for (j = 2, j <= jl, j++)
  for (ii = 2, ii <= il, ii+=b) {
    for (i = ii; i <= min(ii+b-1,il); i++) {
      ib          = i-ii+1;
      AP[ib]      = ...;
      T           = 1.0/(1.0 +AP[ib]);
      D[2,ib]     = T*AP[ib];
      DW[1,2,j,i] = T*DW[1,2,j,i];
    }
    for (k=3, k <= kl-1, k++)
      for (i = ii; i <= min(ii+b-1,il); i++) {
        ib          = i-ii+1;
        AM          = AP[ib];
        AP[ib]      = ...;
        T           = ...AP[ib]-AM*D[ib,k-1]...;
        D[ib,k]     = T*AP;
        DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...;
      }
    ...
    for (k=kl-1, k>=2, k--) {
      for (i = ii; i <= min(ii+b-1,il); i++)
        DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
      /* Ends code to be executed by processor (j,i) */
    }
  }
}

```

Figure 11.67: Excerpt of Fig. 11.23 after partitioning, array contraction, blocking, and inner-loop interleaving

```

for (i=0; i<n; i++) T[i] = A[i] + B[i];
for (i=0; i<n; i++) S[i] = C[i] + D[i];
for (i=0; i<n; i++) E[i] = T[i] * S[i];

```

Exercise 11.10.3: Stripmine the outer loop

```

for (i=n-1; i>=0; i--)
  for (j=0; j<n; j++)

```

into strips of width 10.

11.11 Other Uses of Affine Transforms

So far we have focused on the architecture of shared memory machines, but the theory of affine loop transforms has many other applications. We can apply affine transforms to other forms of parallelism including distributed memory

machines, vector instructions, SIMD (Single Instruction Multiple Data) instructions, as well as multiple-instruction-issue machines. The reuse analysis introduced in this chapter also is useful for data *prefetching*, which is an effective technique for improving memory performance.

11.11.1 Distributed Memory Machines

For distributed memory machines, where processors communicate by sending messages to each other, it is even more important that processors be assigned large, independent units of computation, such as those generated by the affine-partitioning algorithm. Besides computation partitioning, a number of additional compilation issues remain:

1. *Data allocation.* If processors use different portions of an array, they each only have to allocate enough space to hold the portion used. We can use projection to determine the section of arrays used by each processor. The input is the system of linear inequalities representing the loop bounds, the array access functions, and the affine partitions that map the iterations to processor IDs. We project away the loop indices and find for each processor ID the set of array locations used.
2. *Communication code.* We need to generate explicit code to send and receive data to and from other processors. At each synchronization point
 - (a) Determine the data residing on one processor that is needed by other processors.
 - (b) Generate the code that finds all the data to be sent and packs it into a buffer.
 - (c) Similarly, determine the data needed by the processor, unpack received messages, and move the data to the right memory locations.

Again, if all accesses are affine, these tasks can be performed by the compiler, using the affine framework.

3. *Optimization.* It is not necessary for all the communications to take place at the synchronization points. It is preferable that each processor sends data as soon as it is available, and that each processor does not start waiting for data until it is needed. Such optimizations must be balanced by the goal of not generating too many messages, since there is a nontrivial overhead associated with processing each message.

Techniques described here have other applications as well. For example, a special-purpose embedded system may use coprocessors to offload some of its computations. Or, instead of demand fetching data into the cache, an embedded system may use a separate controller to load and unload data into and out of the cache, or other data buffers, while the processor operates on other data. In these cases, similar techniques can be used to generate the code to move data around.

11.11.2 Multi-Instruction-Issue Processors

We can also use affine loop transforms to optimize the performance of multi-instruction-issue machines. As discussed in Section 10.5, the performance of a software-pipelined loop is limited by two factors: cycles in precedence constraints and the usage of the critical resource. By changing the makeup of the innermost loop, we can improve these limits.

First, we may be able to use loop transforms to create innermost parallelizable loops, thus eliminating precedence cycles altogether. Suppose a program has two loops, with the outer being parallelizable and the inner not. We can permute the two loops to make the inner loop parallelizable and so create more opportunities for instruction-level parallelism. Notice that it is not necessary for iterations in the innermost loop to be completely parallelizable. It is sufficient that the cycle of dependences in the loop be short enough so that all the hardware resources are fully utilized.

We can also relax the limit due to resource usage by improving the usage balance inside a loop. Suppose one loop only uses the adder, and another uses only the multiplier. Or, suppose one loop is memory bound and another is compute bound. It is desirable to fuse each pair of loops in these examples together so as to utilize all the functional units at the same time.

11.11.3 Vector and SIMD Instructions

Besides multiple-instruction issue, there are two other important forms of instruction-level parallelism: vector and SIMD operations. In both cases, the issue of just one instruction causes the same operation to be applied to a vector of data.

As mentioned previously, many early supercomputers used vector instructions. Vector operations are performed in a pipelined manner; the elements in the vector are fetched serially and computations on different elements are overlapped. In advanced vector machines, vector operations can be *chained*: as the elements of the vector results are produced, they are immediately consumed by operations of another vector instruction without having to wait for all the results to be ready. Moreover, in advanced machines with *scatter/gather* hardware, the elements of the vectors need not be contiguous; an index vector is used to specify where the elements are located.

SIMD instructions specify that the same operation be performed on contiguous memory locations. These instructions load data from memory in parallel, store them in wide registers, and compute on them using parallel hardware. Many media, graphics, and digital-signal-processing applications can benefit from these operations. Low-end media processors can achieve instruction-level parallelism simply by issuing one SIMD instruction at a time. Higher-end processors can combine SIMD with multiple-instruction issue to achieve higher performance.

SIMD and vector instruction generation share many similarities with locality

optimization. As we find independent partitions that operate on contiguous memory locations, we stripmine those iterations and interleave these operations in innermost loops.

SIMD instruction generation poses two additional difficulties. First, some machines require that the SIMD data fetched from memory be aligned. For example, they might require that 256-byte SIMD operands be placed in addresses that are multiples of 256. If the source loop operates on just one array of data, we can generate one main loop that operates on aligned data and extra code before and after the loop to handle those elements at the boundary. For loops operating on more than one array, however, it may not be possible to align all the data at the same time. Second, data used by consecutive iterations in a loop may not be contiguous. Examples include many important digital-signal-processing algorithms, such as Viterbi decoders and fast Fourier transforms. Additional operations to shuffle the data around may be necessary to take advantage of the SIMD instructions.

11.11.4 Prefetching

No data-locality optimization can eliminate all memory accesses; for one, data used for the first time must be fetched from memory. To hide the latency of memory operations, *prefetch instructions* have been adopted in many high-performance processors. Prefetch is a machine instruction that indicates to the processor that certain data is likely to be used soon, and that it is desirable to load the data into the cache if it is not present already.

The reuse analysis described in Section 11.5 can be used to estimate when caches misses are likely. There are two important considerations when generating prefetch instructions. If contiguous memory locations are to be accessed, we need to issue only one prefetch instruction for each cache line. Prefetch instructions should be issued early enough so that the data is in the cache by the time it are used. However, we should not issue prefetch instructions too far in advance. The prefetch instructions can displace data that may still be needed; also the prefetched data may be flushed before it is used.

Example 11.73: Consider the following code:

```
for (i=0; ii<3; i++)
    for (j=0; j<100; j++)
        A[i,j] = ...;
```

Suppose the target machine has a prefetch instruction that can fetch two words of data at a time, and that the latency of a prefetch instruction takes about the time to execute six iterations of the loop above. The prefetch code for the above example is shown in Fig. 11.68.

We unroll the innermost loop twice, so a prefetch can be issued for each cache line. We use the concept of software pipelining to prefetch data six iterations before it is used. The prolog fetches the data used in the first six iterations. The


```

for (i=0; i<3; i++) {
    for (j=0; j<6; j+=2)
        prefetch(&A[i,j]);
    for (j=0; j<94; j+=2) {
        prefetch(&A[i,j+6]);
        A[i,j] = ...;
        A[i,j+1] = ...;
    }
    for (j=94; j<100; j++)
        A[i,j] = ...;
}

```

Figure 11.68: Code modified to prefetch data

steady state loop prefetches six iterations ahead as it performs its computation. The epilog issues no prefetches, but simply executes the remaining iterations. □

11.12 Summary of Chapter 11

- ◆ *Parallelism and Locality from Arrays.* The most important opportunities for both parallelism and locality-based optimizations come from loops that access arrays. These loops tend to have limited dependences among accesses to array elements and tend to access arrays in a regular pattern, allowing efficient use of the cache for good locality.
- ◆ *Affine Accesses.* Almost all theory and techniques for parallelism and locality optimization assume accesses to arrays are affine: the expressions for the array indexes are linear functions of the loop indexes.
- ◆ *Iteration Spaces.* A loop nest with d nested loops defines a d -dimensional iteration space. The points in the space are the d -tuples of values that the loop indexes can assume during the execution of the loop nest. In the affine case, the limits on each loop index are linear functions of the outer loop indexes, so the iteration space is a polyhedron.
- ◆ *Fourier-Motzkin Elimination.* A key manipulation of iteration spaces is to reorder the loops that define the iteration space. Doing so requires that a polyhedral iteration space be projected onto a subset of its dimensions. The Fourier-Motzkin algorithm replaces the upper and lower limits on a given variable by inequalities between the limits themselves.
- ◆ *Data Dependences and Array Accesses.* A central problem we must solve in order to manipulate loops for parallelism and locality optimizations is whether two array accesses have a data dependence (can touch the

same array element). When the accesses and loop bounds are affine, the problem can be expressed as whether there are solutions to a matrix-vector equation within the polyhedron that defines the iteration space.

- ◆ *Matrix Rank and Data Reuse.* The matrix that describes an array access can tell us several important things about that access. If the rank of the matrix is as large as possible (minimum of the number of rows and number of columns), then the access never touches the same element twice as the loops iterate. If the array is stored in row- (column-)major form, then the rank of the matrix with the last (first) row deleted tells us whether the access has good locality; i.e., elements in a single cache line are accessed at about the same time.
- ◆ *Data Dependence and Diophantine Equations.* Just because two accesses to the same array touch the same region of the array does not mean that they actually access any element in common. The reason is that each may skip some elements; e.g., one accesses even elements and the other accesses odd elements. In order to be sure that there is a data dependence, we must solve a Diophantine (integer solutions only) equation.
- ◆ *Solving Diophantine Linear Equations.* The key technique is to compute the greatest common divisor (GCD) of the coefficients of the variables. Only if that GCD divides the constant term will there be integer solutions.
- ◆ *Space-Partition Constraints.* To parallelize the execution of a loop nest, we need to map the iterations of the loop to a space of processors, which can have one or more dimensions. The space-partition constraints say that if two accesses in two different iterations share a data dependence (i.e., they access the same array element), then they must map to the same processor. As long as the mapping of iterations to processors is affine, we can formulate the problem in matrix-vector terms.
- ◆ *Primitive Code Transformations.* The transformations used to parallelize programs with affine array accesses are combinations of seven primitives: loop fusion, loop fission, re-indexing (adding a constant to loop indexes), scaling (multiplying loop indexes by a constant), reversal (of a loop index), permutation (of the order of loops), and skewing (rewriting loops so the line of passage through the iteration space is no longer along one of the axes).
- ◆ *Synchronization of Parallel Operations.* Sometimes more parallelism can be obtained if we insert synchronization operations between steps of a program. For example, consecutive loop nests may have data dependences, but synchronizations between the loops can allow the loops to be parallelized separately.
- ◆ *Pipelining.* This parallelization technique allows processors to share data, by synchronously passing certain data (typically array elements) from one

processor to an adjacent processor in the processor space. The method can improve the locality of the data accessed by each processor.

- ◆ *Time-Partition Constraints.* To discover opportunities for pipelining, we need to discover solutions to the time-partition constraints. These say that whenever two array accesses can touch the same array element, then the access in the iteration that occurs first must be assigned to a stage in the pipeline that occurs no later than the stage to which the second access is assigned.
- ◆ *Solving Time-Partition Constraints.* Farkas' Lemma provides a powerful technique for finding all the affine time-partition mappings that are allowed by a given loop nest with array accesses. The technique is essentially to replace the primal formulation of the linear inequalities that express the time-partition constraints by their dual.
- ◆ *Blocking.* This technique breaks each of several loops in a loop nest into two loops each. The advantage is that doing so may allow us to work on small sections (blocks) of a multidimensional array, one block at a time. That, in turn, improves the locality of the program, letting all the needed data reside in the cache while working on a single block.
- ◆ *Stripmining.* Similar to blocking, this technique breaks only a subset of the loops of a loop nest into two loops each. A possible advantage is that a multidimensional array is accessed a "strip" at a time, which may lead to the best possible cache utilization.

11.13 References for Chapter 11

For detailed discussions of multiprocessor architectures, we refer the reader to the text by Hennessy and Patterson [9].

Lamport [13] and Kuck, Muraoka, and Chen [6] introduced the concept of data-dependence analysis. Early data-dependence tests used heuristics to prove a pair of references to be independent by determining if there are no solutions to Diophantine equations and systems of real linear inequalities: [5, 6, 26]. Maydan, Hennessy, and Lam [18] formulated the data-dependence test as integer linear programming and showed that the problem can be solved exactly and efficiently in practice. The data-dependence analysis described here is based on work by Maydan, Hennessy, and Lam [18] and Pugh and Wonnacott [23], which in turn use techniques of Fourier-Motzkin elimination [7] and Shostak's algorithm [25].

The 70's and early 80's saw the use of loop transformations to improve vectorization and parallelization: loop fusion [3], loop fission [1], stripmining [17], and loop interchange [28]. There were three major experimental parallelizer/vectorizing projects going on at the time: Parafrase led by Kuck at the University of Illinois Urbana-Champaign [21], the PFC project led by Kennedy

at Rice University [4], and the PTRAN project led by Allen at IBM Research [2].

McKellar and Coffman [19] first discussed using blocking to improve data locality. Lam, Rothbert, and Wolf [12] provided the first in-depth empirical analysis of blocking on caches for modern architectures. Wolf and Lam [27] used linear-algebra techniques to compute data reuse in loops. Sarkar and Gao [24] introduced the optimization of array contraction.

Lamport [13] was the first to model loops as iteration spaces and used hyperplaning (a special case of an affine transform) to find parallelism for multiprocessors. Affine transforms have their root in systolic-array algorithm design [11]. Intended as parallel algorithms directly implemented in VLSI, systolic arrays require communication to be minimized along with parallelization. Algebraic techniques were developed to map the computation onto space and time coordinates. The concept of an affine schedule and the use of Farkas' Lemma in affine transformations were introduced by Feautrier [8]. The affine-transformation algorithm described here is based on work by Lim et al. [15, 14, 16].

Porterfield [22] proposed one of the first compiler algorithms to prefetch data. Mowry, Lam, and Gupta [20] applied reuse analysis to minimize the prefetch overhead and gain an overall performance improvement.

1. Abu-Sufah, W., D. J. Kuck, and D. H. Lawrie, "On the performance enhancement of paging systems through program analysis and transformations," *IEEE Trans. on Computing* **C-30**:5 (1981), pp. 341–356.
2. Allen, F. E., M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," *J. Parallel and Distributed Computing* **5**:5 (1988), pp. 617–640.
3. Allen, F. E. and J. Cocke, "A Catalogue of optimizing transformations," in *Design and Optimization of Compilers* (R. Rustin, ed.), pp. 1–30, Prentice-Hall, 1972.
4. Allen, R. and K. Kennedy, "Automatic translation of Fortran programs to vector form," *ACM Transactions on Programming Languages and Systems* **9**:4 (1987), pp. 491–542.
5. Banerjee, U., *Data Dependence in Ordinary Programs*, Master's thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1976.
6. Banerjee, U., *Speedup of Ordinary Programs*, Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1979.
7. Dantzig, G. and B. C. Eaves, "Fourier-Motzkin elimination and its dual," *J. Combinatorial Theory*, **A(14)** (1973), pp. 288–297.
8. Feautrier, P., "Some efficient solutions to the affine scheduling problem: I. One-dimensional time," *International J. Parallel Programming* **21**:5 (1992), pp. 313–348,

9. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.
10. Kuck, D., Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Transactions on Computers* **C-21**:12 (1972), pp. 1293–1310.
11. Kung, H. T. and C. E. Leiserson, "Systolic arrays (for VLSI)," in Duff, I. S. and G. W. Stewart (eds.), *Sparse Matrix Proceedings* pp. 256–282. Society for Industrial and Applied Mathematics, 1978.
12. Lam, M. S., E. E. Rothberg, and M. E. Wolf, "The cache performance and optimization of blocked algorithms," *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 63–74.
13. Lamport, L., "The parallel execution of DO loops," *Comm. ACM* **17**:2 (1974), pp. 83–93.
14. Lim, A. W., G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," *Proc. 13th International Conference on Supercomputing* (1999), pp. 228–237.
15. Lim, A. W. and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," *Proc. 24th ACM SIGPLAN-SIG-ACT Symposium on Principles of Programming Languages* (1997), pp. 201–214.
16. Lim, A. W., S.-W. Liao, and M. S. Lam, "Blocking and array contraction across arbitrarily nested loops using affine partitioning," *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2001), pp. 103–112.
17. Loveman, D. B., "Program improvement by source-to-source transformation," *J. ACM* **24**:1 (1977), pp. 121–145.
18. Maydan, D. E., J. L. Hennessy, and M. S. Lam, "An efficient method for exact dependence analysis," *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 1–14.
19. McKeller, A. C. and E. G. Coffman, "The organization of matrices and matrix operations in a paged multiprogramming environment," *Comm. ACM*, **12**:3 (1969), pp. 153–165.
20. Mowry, T. C., M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (1992), pp. 62–73.

21. Padua, D. A. and M. J. Wolfe, “Advanced compiler optimizations for supercomputers,” *Comm. ACM*, **29**:12 (1986), pp. 1184–1201.
22. Porterfield, A., *Software Methods for Improving Cache Performance on Supercomputer Applications*, Ph.D. Thesis, Department of Computer Science, Rice University, 1989.
23. Pugh, W. and D. Wonnacott, “Eliminating false positives using the omega test,” *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 140–151.
24. Sarkar, V. and G. Gao, “Optimization of array accesses by collective loop transformations,” *Proc. 5th International Conference on Supercomputing* (1991), pp. 194–205.
25. R. Shostak, “Deciding linear inequalities by computing loop residues,” *J. ACM*, **28**:4 (1981), pp. 769–779.
26. Towle, R. A., *Control and Data Dependence for Program Transformation*, Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1976.
27. Wolf, M. E. and M. S. Lam, “A data locality optimizing algorithm,” *Proc. SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 30–44.
28. Wolfe, M. J., *Techniques for Improving the Inherent Parallelism in Programs*, Master’s thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1978.