CHAPTER 6

Code Generation for Parallel Computers

6.1 Loop Interchange and Parallelization

As we pointed out in Section 1.2.2, parallelization presents a different set of problems for the translation system because of the need to find parallel regions with sufficient granularity to compensate for the overhead of starting and synchronizing a parallel execution. When dealing with loop nests, this typically means parallelizing loops with a lot of computation. Thus, while vectorization seeks inner loop parallelization, parallelization looks for *outer*, since outer loops should have more computation per iteration. As a result, the purpose of parallelization should be to interchange loops that carry no dependences to the outside, if doing so will not cause them to pick up a carried dependence.

The following examples should explain the distinction.

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

```
   ENDDO
ENDDO
```

If we interchange the loops in this example, we will move the parallel loop to the inside, leaving a recurrence to be calculated on each parallel iteration.

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

If the example were slightly different, we might not be able to find outer loop parallelism.

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

Here the dependence carried by the outer loop has direction vector (<,<) and thus is not interchange sensitive. That is, the outer loop will still carry the dependence after interchange. Without additional transformations, inner loop parallelism is the best we can do.

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

As we shall see, outer loop parallelism can be achieved in this example through loop skewing followed by loop interchange.

These examples lead us to ask the question: When can we move one loop in a loop nest to the outermost position in the nest? That question is answered by the following theorem.

> **Theorem 6.1.** In a nest of loops, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only "=" entries.

Proof. Clearly a loop with all "=" entries in its direction matrix column can be parallelized at the outermost level—it can be moved to the outermost position because the loops inside it will remain in the same relative order and thus all direction vectors in the matrix will still have all "<"

entries in their outermost non-"=" positions. Since it has only "=" entries, this loop will carry no dependences at any level, including the outermost.

Now we must show that these are the only loops that can be parallelized at the outermost level. Consider any other case. If the column contains a ">" entry, it cannot be moved to the outermost level, because that would violate the condition for legal interchange given by Theorem 5.1. If the column contains a "<," it cannot be parallelized at the outermost level because it will always carry a dependence at that level. Q.E.D.

This theorem suggests a general code generation strategy for perfect loop nests based on the direction matrix:

1. If there is at least one loop with an all-"=" direction matrix column, select the outermost such loop, move it to the outermost position, parallelize it, eliminate its column from the direction matrix and repeat step 1 on the remaining loops.
2. Otherwise, pick a loop with the largest number of "<" entries and generate a sequential loop for it at the outermost position. Eliminate its column and any rows corresponding to dependences that it carries at that position (these will be the rows with "<" entries in the column corresponding to the selected loop). Repeat step 1 with the new direction matrix for the remaining loops.

In Section 6.3 we will generalize this algorithm substantially. However, this gives the flavor of the technique—on each step a loop is selected to be in the outermost position, either as a parallel loop or a sequential loop. If a sequential loop is selected, it should be the loop that uncovers the most parallelism in the remaining loops.

As an example, consider the following three-loop nest:

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
         A(I+1,J,K) = A(I,J,K) + X1
         B(I,J,K+1) = B(I,J,K) + X2
         C(I+1,J+1,K+1) = C(I,J,K) + X3
      ENDDO
   ENDDO
ENDDO
```

The direction matrix for this loop is

$$\begin{bmatrix} < = = \\ = = < \\ < < < \end{bmatrix}$$

Since none of the loops can be parallelized at the outermost level right away, the outer loop is sequentialized leaving a direction matrix [= <], which permits the loop on J to be parallelized.

```
DO I = 1, N
   PARALLEL DO J = 1, M
      DO K = 1, L
         A(I+1,J,K) = A(I,J,K) + X1
         B(I,J,K+1) = B(I,J,K) + X2
         C(I+1,J+1,K+1) = C(I,J,K) + X3
      ENDDO
   END PARALLEL DO
ENDDO
```

It should be obvious from this example, that a transformation like loop distribution might make even more parallelism available. For example, if the loop nests were distributed into three distinct loops, each of the loops would have two dimensions of parallelism. Whether this is profitable depends on the minimum granularity for parallelism on the particular target machine. Transformations on a single loop nest are the subject of the next section.

Discuss the differences between loop interchange in vectorization and loop interchange in parallelization. Prove the outermost test. Briefly overview the algorithm.

6.2 Single-Loop Methods

This section will consider the case when you have only a single loop and you wish to make it parallel. What transformations are needed to achieve a profitable parallelization?

6.2.1 Tiling

Suppose we have a simple parallel loop that carries no dependences. Then all the iterations can be executed in parallel. However, there may not be enough work in a single iteration to schedule each iteration as a separate entity. It may be better to group the iterations into sets and make a set be the minimum schedulable unit. This can be expressed as a

source-to-source transformation via strip mining. Consider the simple parallelizable loop:

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

Suppose that there are exactly P processors available to execute the loop, then it could be strip mined as follows:

```
k = CEIL(N/P)
PARALLEL DO I = 1, N, k
   DO i = I, MIN(I+k-1,N)
      A(i) = A(i) + B(i)
   ENDDO
END PARALLEL DO
```

Frequently, this strip mining is done automatically by special hardware, as on the Convex C2 and C3.

If the time required by each iteration of the parallel loop is variable, it may be preferable to choose a block size that is smaller than the number of iterations divided by the number of processors, so that a crude form of load balancing is achieved when processors select a new block of iterations to execute after finishing one block. If the processors can begin work at staggered times, it may also be profitable to vary the block size as the computation proceeds [cite GSS].

6.2.2 Privatization

Scalar and array privatization based on kill analysis

6.2.3 Alignment

As we have discussed previously, loop distribution can be used to eliminate some carried dependences from loops. Consider the following example:

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

If the loop is distributed into two loops, both can be parallelized.

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

```
   B(I) = A(I) + D
ENDDO
```

However, this parallelization may not be desirable because it reduces the granularity of parallelism.

Another approach would be to notice that the problem is caused by the fact that the values of A are used on an iteration from the one on which they are created. We could fix this problem by changing iteration on which the first statement computes its values. We call this transformation *alignment*. Here is the loop after a naive alignment has been performed:

```
DO i = 1, N+1
   IF (i.GT.1) A(i) = B(i-1) + C
   IF (i.LE.N) B(i) = A(i) + D
ENDDO
```

By increasing the number of iterations and executing the two statements on slightly different subsets of those iterations, the carried dependence can be eliminated. The disadvantage of this approach is the overhead of executing the conditionals and the extra iteration. This overhead can be slightly reduced by folding the iterations together, executing the last iteration of the first statement on the same iteration as the first iteration of the second statement.

```
DO i = 1, N
   j = MOD(i-2,N)+1
   A(j+1) = B(j) + C
   B(i) = A(i) + D
ENDDO
```

Note that for $2 \le i \le N$, `j+1=i`. When `i=1`, however, `j=N-1` and `j+1=N`. Thus the total number of iterations is reduced by 1, but we still have the overhead of the MOD calculation on each iteration.

One question that comes to mind is whether it is always possible to use alignment to eliminated carried dependences. Clearly, the answer is no if the carried dependence is involved in a recurrence, as the following example shows:

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

In this case, any alignment that succeeds in eliminating the carried dependence on B will create a carried dependence involving A. The general result is established in the following lemma:

> **Lemma 6.1.** If a collection of statements in a loop form a recurrence, alignment and reordering of statements in the body are not by themselves sufficient to convert all carried dependences to loop-independent dependences while preserving the meaning of the original loop.

Proof. Suppose the contrary. Then for any collection of statements involved in a recurrence in a loop there exists an ordering of the statements and an alignment that has no carried dependences. To preserve correctness, we must preserve every dependence in the transformed code. Thus each statement in the recurrence must have a dependence from it to some other statement in the loop body that is also part of the recurrence. Furthermore, that dependence must be loop-independent. But this is clearly impossible for the last statement in the final order of statements in the recurrence. The only way a dependence can exist from it to another statement in the recurrence is for the dependence to be backward, hence that dependence must be carried. Q.E.D.

But what of a carried dependence that is not involved in a recurrence? Can it always be eliminated by alignment without introducing new carried dependences to take its place? The answer is again no, because of the possibility of an *alignment conflict*—two or more dependences that cannot be simultaneously aligned. Consider the following example:

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

This loop contains two dependences involving the array A, one loop-independent dependence and a loop-carried dependence. Any attempt to align the carried dependence, will transform the loop-independent dependence to a loop-carried dependence.

6.2.4 Code Replication

A closer examination of the alignment conflict example loop from the previous section reveals a way to solve the problem if we can tolerate some extra computation on each parallel iteration. If we replace the refer-

ence to A(I) by a reference to the expression that computes it, the carried dependence can be eliminated.

```
DO I = 1, N
   IF (I.EQ.1) THEN
      t = A(I)
   ELSE
      t = B(I-1) + C
   END IF
   A(I+1) = B(I) + C
   X(I) = A(I+1) + t
ENDDO
```

If the compiler-generated variable t is declared private, the loop has no carried dependences, so its iterations can be run completely in parallel. The cost is the additional computation to determine a value for t. This transformation is called *code replication*.

Once again, we raise the question: are alignment and replication, when combined with statement reordering in the loop body, sufficient to eliminate all carried dependences that are not part of a recurrence?

> **Theorem 6.2.** In a single loop that contains no recurrence and in which the distance of each dependence is a constant independent of the loop index, alignment, replication and statement reordering are sufficient to eliminate all carried dependences.

We will prove this theorem by presenting an algorithm to carry out the desired set of transformations. Note that if a loop contains a recurrence, we can use loop distribution to split it into a set of loops, some of which are recurrences and some of which are not. Then the algorithm we shall present can be applied to the loops containing no recurrence to make them all parallel.

Any loop that contains no recurrence can be represented as a directed acyclic *alignment graph* $G=(V, E)$ in which the vertices represent statements and the edges represent dependences. Let each edge be labeled with its dependence distance.

In this graph an alignment of a particular statement can be represented by an *offset*, representing the number of iterations between its current alignment and its original alignment. Initially each statement in the loop has an offset of zero, because none of them have been moved. However, as the algorithm proceeds, the statement offsets will be adjusted to reflect

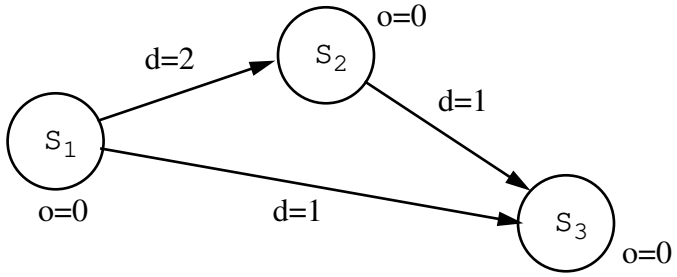
the aggregate effects of the alignments that are performed. Consider the following example loop.

```
     DO I = 1, N
S1      A(I+2) = B(I) + C
S2      X(I+1) = A(I) + D
S3      Y(I) = A(I+1) + X(I)
     ENDDO
```

The initial alignment graph for this example is shown in Figure 6.1

FIGURE 6.1

An alignment graph.



Definition 6.1. Let $o(v)$ denote the offset of vertex v, and $d(e)$ the distance for dependence edge e. We will say that alignment graph $G=(V,E)$ is *carry-free* if, for each edge $e=(v_1,v_2)$, we have

$$o(v_1) + d(e) = o(v_2)$$

The alignment graph in Figure 6.1 is not carry-free. The purpose of an alignment and replication algorithm is to produce, via vertex replication and offset adjustment, an alignment graph that is carry-free. Once we have an alignment graph, the code for the adjusted loop can be generated directly.

Now the basic idea of the alignment and replication algorithm should be clear: We pick a node in the graph to serve as the basis for our alignment and align other nodes with it, replicating when different alignments would be required. This algorithm is given in

FIGURE 6.2

Alignment and replication algorithm.

procedure *align*(*V,E,d,o)*

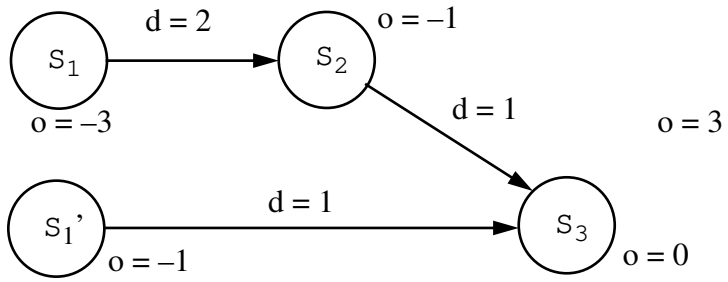
/* *V* and *E* are the vertex and edge sets, respectively */
/* *d* is the dependence distance table */

```
/* o is the offset table, an output variable                */
/* the table entry ⊥ means "undefined"                      */
/* W is a work list for the algorithm

forall v ∈ V do o(v) := ⊥ od;

while V ≠ ∅ do
    select and remove an arbitrary element v0 from V;
    W := {v0}; o(v0) := 0;

    while W≠ ∅ do
        select and remove an arbitrary element v from W;
        foreach edge e incident on v do
            if e = (w,v) then
                if w∈ V then
                    W := W ∪ {w}; o(w) = o(v) – d(e)
                else if o(w) ≠ o(v) – d(e) then /* conflict */
                    create a new vertex w';
                    replace the edge e = (w,v) with e' = (w',v);
                    foreach edge into w do
                        replicate the edge, replacing w with w';
                    od;
                    W := W ∪ {w'}; o(w') = o(v) – d(e)
                end if
            else /* e = (v,w) */
                if w∈ V then
                    W := W ∪ {w}; o(w) = o(v) – d(e)
                else if o(w) ≠ o(v) + d(e) then /* conflict */
                    /* always replicate the upstream vertex */
                    create a replicated vertex v';
                    replace the edge e = (v,w) with e' = (v',w);
                    foreach edge into v do
                        replicate the edge, replacing v with v';
                    od;
                    W := W ∪ {v'}; o(v') = o(w) – d(e)
                end if
            end if
        od
    od
od
end align;
```

If we apply this algorithm to the alignment graph in Figure 6.1 on page 189, choosing S_3 as the root, we get the modified graph in Figure 6.3.

FIGURE 6.3

Modified alignment graph.



All that remains is to show how code is generated from an alignment graph. Note that the offset indicates where in the alignment space each statement should start. Here is a naive alignment for the loop that generated Figure 6.1.

```
     DO I = 1, N+3
S1       IF(I.GE.4) A(I-1) = B(I-3) + C
S1'      IF(I.GE.2 .AND. I.LE.N+1) THEN
            t = B(I-1) + C
         ELSE
            t = A(I+1)
         ENDIF
S2       IF(I.GE.2 .AND.I.LE.N+1) X(I) = A(I-1) + D
S3       IF(I.LE.N) Y(I) = t + X(I)
     ENDDO
```

As with an earlier example, we can repack this code as follows:

```
     DO I = 1, N
        i1 = MOD(I-4,N)+1
S1      A(i1+2) = B(i1) + C
        i2 = MOD(I-2,N)+1
S1'     IF(i2.GE.2) THEN
           t = A(i2) + D
        ELSE
           t = X(I)
        ENDIF
S2      X(i2+1) = A(i2) + D
S3      Y(I) = t + X(I)
     ENDDO
```

Now we need to develop an algorithm to generate this code.

(Algorithm goes here)

6.3 Perfect Loop Nests

6.3.1 Simple Loop Selection

Suppose we are dealing with a perfect loop nest. The principal challenge is to generate the most parallelism with adequate granularity. The principal methods for accomplishing this task will be the use of loop interchange and loop reversal. We will postpone the discussion of other transformations until a later section.

The basis for our algorithm will be the direction matrix for the loop nest as defined in Section 5.1.2 on page 147. Consider the following example.

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

The direction matrix for this set of loops is

$$\begin{bmatrix} < & < & < \\ = & < & = \end{bmatrix}$$

Note that the first dependence in the loop is always carried by the outermost loop level, so the outer loop can never be parallelized, no matter how we interchange the loops. Therefore, the right strategy is to pick an outer loop that leaves the maximum parallelism after it is sequentialized. This is clearly the loop on index `J`. If we rearrange to make this loop parallel, we get the following program, in which the two inner loops can be run in parallel:

```
DO J = 1, M
   PARALLEL DO I = 1, N
      PARALLEL DO K = 1, L
         A(I+1,J+1,K+1) = A(I,J,K)+ A(I+1,J,K+1)
      END PARALLEL DO
   END PARALLEL DO
ENDDO
```

A question that immediately arises is how to tell whether a selected loop arrangement is legal. Consider the following example:

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
         A(I+1,J+1,K+1) = A(I,J,K)+ A(I+1,J+2,K)
      ENDDO
   ENDDO
ENDDO
```

The direction matrix for this nest is

$$\begin{bmatrix} = & < & < \\ < & > & < \end{bmatrix}$$

The `I`-loop cannot be interchanged with the `J`-loop because that would reverse the sense of the second dependence in the loop. However, if we directly swap the `I`-loop and the innermost `K`-loop we get the following direction matrix, which is completely legal:

$$\begin{bmatrix} < & < & = \\ < & > & < \end{bmatrix}$$

If the `K`-loop is sequentialized, the other two loops can now be parallelized:

```
DO K = 1, L
   PARALLEL DO J = 1, M
      PARALLEL DO I = 1, N
         A(I+1,J+1,K+1) = A(I,J,K)+ A(I+1,J+2,K)
      END PARALLEL DO
   END PARALLEL DO
ENDDO
```

In general, the loops in a nest can legally be rewritten in any order that preserves the sense of each dependence in the loop nest—in other words, preserves the property that all the leftmost non-"=" direction in every row of the matrix is "<".

Occasionally, loop reversal can be used to make parallelization possible, as the following example shows:

```
DO I = 2, N+1
   DO J = 2, M+1
      DO K = 1, L
         A(I,J,K) = A(I,J-1,K+1)+ A(I-1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

By examining the direction matrix

$$\begin{bmatrix} = < > \\ < = > \end{bmatrix}$$

we see that we would need to sequentialize both outer loops to parallelize the innermost loop. However, if we reverse the innermost loop and move it to the outermost position, we get the following direction matrix:

$$\begin{bmatrix} < = < \\ < < = \end{bmatrix}$$

which yields parallelism in two dimensions:

```
DO K = L, 1, -1
   PARALLEL DO I = 2, N+1
      PARALLEL DO J = 2, M+1
         A(I,J,K) = A(I,J-1,K+1)+ A(I-1,J,K+1)
      END PARALLEL DO
   END PARALLEL DO
ENDDO
```

These examples lead us to a general strategy for parallel code generation in a parallel loop nest. At each stage in the process, we have a "current direction matrix." The algorithm begins by parallelizing all of the loops that correspond to columns in the direction matrix that have only "=" directions. If any loops remain, use some heuristic to select a loop to sequentialize, removing all direction matrix rows corresponding to dependences carried by the selected loop. This process is continued until all loops have been either parallelized or sequentialized.

FIGURE 6.4

Algorithm for parallel loop selection.

procedure parallelize (*N)*

/* *N* is the loop nest to be parallelized */
/* *M* is the direction matrix for the loop nest */
/* *L* is a worklist of the loops in *N* */

compute the direction matrix *M* for the nest;
let *L* be a list of the loops in the nest *N*;

while $L \neq \emptyset$ **do begin**
  while there exist columns in *M* with all "=" directions **do begin**
    l = loop corresponding to the outermost all-"=" column;

```
            remove l from L;
            generate a parallel loop for l at the outermost position;
            eliminate the column corresponding to l from M
        end;

        if L ≠ Ø then begin
            l = loop selected heuristically;
            remove l from L;
            generate a sequential loop for l at the outermost level;
            remove the column corresponding to l from M;
            remove all rows corresponding to dependences carried by
                loop l from M;
        end
    end
end
```

The remaining question is what heuristic to use to select a loop to sequentialize. The simplest approach would be to select the loop with the most "<" directions, on the theory that this would eliminate the maximum number of rows from the direction matrix. However, this would fail in the case of the following direction matrix:

$$\begin{bmatrix} < & < & = & = \\ < & = & < & = \\ < & = & = & < \\ = & < & = & = \\ = & = & < & = \\ = & = & = & < \end{bmatrix}$$

Even though the outer loop carries the most dependences, it will do no good to sequentialize it, because the three inner loops will need to be sequentialized as well. However, if the three inner loops are sequentialized, the outer loop can be moved to the innermost position and parallelized.

If we view the loops as bit vectors with a "1" in each position corresponding to a dependence that would be carried by this loop in the outermost position. Then the problem of loop selection corresponds to finding a minimal basis among the loops, with "logical or" as the combining operation. In turn, this is the same as the minimum set cover problem,

which is known to be NP-complete. This is the reason that loop selection should be a heuristic.

One way to avoid the problem highlighted by the example is to favor the selection of loops that must be sequentialized before parallelism can be uncovered. Thus, if there exists a loop that can legally be moved to the outermost position and there is a dependence for which that loop has the only "<" direction, sequentialize that loop. If there are several such loops, they will all need to be sequentialized before another loop can be made parallel.

6.3.2 Code Generation with Loop Skewing

As we saw in Chapter 5, loop skewing can be used to create inner loop parallelism by changing "=" directions to "<" directions. The value of skewing to asynchronous parallelization can be illustrated by an example.

```
DO I = 2, N+1
   DO J = 2, M+1
      DO K = 1, L
         A(I,J,K) = A(I,J-1,K)+ A(I-1,J,K)
         B(I,J,K+1) = B(I,J,K) + A(I,J,K)
      ENDDO
   ENDDO
ENDDO
```

The direction matrix for this example is

$$\begin{bmatrix} = & < & = \\ < & = & = \\ = & = & < \end{bmatrix}$$

with the corresponding distance matrix

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

But now we can skew the innermost loop with respect to the two outer loops by substituting

```
k = K+I+J
```

which yields the following code

```
DO I = 2, N+1
   DO J = 2, M+1
      DO k = I+J+1, I+J+L
         A(I,J,k-I-J) = A(I,J-1,k-I-J) + A(I-1,J,k-I-J)
         B(I,J,k-I-J+1) = B(I,J,k-I-J) + A(I,J,k-I-J)
      ENDDO
   ENDDO
ENDDO
```

with the direction matrix

$$\begin{bmatrix} = & < & < \\ < & = & < \\ = & = & < \end{bmatrix}$$

Now the inner loop can be interchanged to the outermost position to yield

```
DO k = 5, N+M+1
   DO I = MAX(2,k-M-L-1), MIN(N+1,k-L-2)
      DO J = MAX(2,k-I-L), MIN(M+1,k-I-1)
         A(I,J,k-I-J) = A(I,J-1,k-I-J) + A(I-1,J,k-I-J)
         B(I,J,k-I-J+1) = B(I,J,k-I-J) + A(I,J,k-I-J)
      ENDDO
   ENDDO
ENDDO
```

in which both of the inner loops can be parallelized.

These ideas are captured in a modification to the parallel code generation algorithm from Figure 6.4. The new algorithm is shown in Figure 6.4.

FIGURE 6.5

Algorithm for parallel loop selection.

procedure parallelize *(N)*

/* *N* is the loop nest to be parallelized */
/* *M* is the direction matrix for the loop nest */
/* *L* is a worklist of the loops in *N* */

compute the direction matrix *M* for the nest;
let *L* be a list of the loops in the nest *N*;

while $L \neq \emptyset$ **do begin**
  while there exist columns in *M* with all "=" directions **do begin**
    l = loop corresponding to the outermost all-"=" column;
    remove *l* from *L*;

```
            generate a parallel loop for l at the outermost position;
            eliminate the column corresponding to l from M
        end;

        if L ≠ Ø then begin
            Let {l1, l2,..., lk} be a collection of loops that can be moved
                to the outermost remaining loop positions and which are
                fully permutable;
            l = loop selected heuristically from this collection to cover
                the most other loops in the collection;
            skew l with respect to the loops in the group that it does not
                 cover;
            interchange l to the outermost position in the collection;
            remove l from L;
            generate a sequential loop for l at the outermost level;
            remove the column corresponding to l from M;
            remove all rows corresponding to dependences carried by
                loop l from M;
        end
    end
end
```

Note that this algorithm makes use of the notion of a fully permutable nest of loops

Definition 6.2. A loop nest is *fully permutable* if the any permutation of the loops in the nest leaves all of the dependence distance vectors lexicographically positive.

Theorem 6.3. Loops l_i through l_j of a nest consisting of loops l_1, l_2,..., l_n are fully permutable if and only if for each dependence in the nest, either it is carried at a level lower than i or its directions D_i through D_j are all "<" or "=".

Theorem 6.4. Any loop nest for which each dependence has a distance vector in which all distances are constants known at compile time can be made fully permutable by loop skewing.

Proof. This is straightforward—simply uses skewing to convert any negative distances to positive. The result must be fully permutable by Theorem 6.3 since all distances are then ≥ 0. For details consult Lam and Wolf [Lam and Wolf]. Q.E.D.

6.3.3 Nests with Fixed Distance Vectors

We will now consider parallel code generation for loop nests in which each of the dependences can be represented by a distance vector in which each distance is a constant known at compile time. On such loop nests a very general technique based on the theory of unimodular transformations can be used.

We begin with the observation that loop interchange, loop skewing and loop reversal are all unimodular *transformations* of the loop indices in a perfect loop nest.

Definition 6.3. A transformation represented by a matrix T is *unimodular* if

1) T is square

2) all the elements of T are integral and

3) the absolute value of the determinant of T is 1.

Note that if T_1 and T_2 are both unimodular, then so is $T_1 \cdot T_2$. Let us examine the three transformations we have been discussing.

1. *Permutation.* Any loop permutation can be expressed as a matrix in which each row and each column has at most a single entry equal to 1. Here are two examples:

$$\text{Identity:} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

$$\text{Loop interchange:} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} j \\ i \\ k \end{bmatrix}$$

2. *Reversal*. A matrix that differs from the identity matrix by having a single negative 1 on the diagonal reverses the loop corresponding to the negative 1.

$$\text{Reversal:} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ -j \\ k \end{bmatrix}$$

3. *Skewing by factor f*. A matrix that differs from the identity by having a single integral value off the diagonal corresponds to skewing.

Skewing:
$$\begin{bmatrix} 1 & 0 & 0 \\ f & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j + fi \\ k \end{bmatrix}$$

The following theorem is easy to prove:

> **Theorem 6.5.** A unimodular transformation T is legal if and only if, for each distance vector d in the loop nest, T$d \succ 0$, that is, Td is lexicographically greater than zero.

Proof. If not then some distance vector is mapped by the transformation into a distance vector that reverses the sense of the dependence, which is clearly not legal.

Given that we are working in a loop nest with only fixed distances, we can make use of Theorem 6.4, which says that any such loop nest can be made fully permutable by loop skewing.

FIGURE 6.6

Skewing-based Loop Selection.

```
procedure parallelize (N)

    /* N is the loop nest to be parallelized          */
    /* M is the direction matrix for the loop nest    */
    /* L is a worklist of the loops in N              */

    compute the direction matrix M for the nest;
    let L be a list of the loops in the nest N;

    while there exist columns in M with all "=" directions do begin
        l = loop corresponding to the outermost all-"=" column;
        remove l from L;
        generate a parallel loop for l at the outermost position;
        eliminate the column corresponding to l from M
    end;

    if L ≠ Ø then begin
        make the remaining loops fully permutable,
            by skewing if necessary;
        l = loop selected heuristically to cover
            the maximum number of other loops;
        skew l with respect to the other loops remaining in L,
            treating it as the innermost loop;
        interchange l to the outermost level;
```

```
        remove l from L;
        generate a sequential loop for l;
        generate parallel loops for the loops remaining in L;
    end
end
```

The algorithm given in this section may result in too much skewing and the practical compiler designer may wish to use a simpler version that skews at most one or two loops with respect to one other loop.

6.4 Fusion and Distribution

6.4.1 General Approach

The methods discussed so far do not take advantage of an important transformation, loop distribution. We have seen that loop distribution can be used to create more parallelism by separating sequential and parallel portions of a computation. As an example consider the loop:

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

The middle statement in this loop is sequential. However, it can be broken out into a separate loop leaving the two remaining statements together in a parallel loop.

```
PARALLEL DO I = 1, N
   A(I) = B(I) + 1
   D(I) = A(I) + X
END PARALLEL DO
DO I = 1, N
   C(I) = A(I) + C(I-1)
ENDDO
```

Lam and Wolf have other kinds of code generation. One uses skewing to parallelize outer loops. It might be worth putting some of those into this section. -- Ken

In this section, we will consider techniques that use loop distribution to break up a loop nest into a collection of subloops and then loop fusion to recombine the subloops to get as much large-grain parallelism as possible.

6.4.2 Single-Loop Methods

We begin with a treatment of methods for parallelization of loops that contain no nested loops. Suppose we have such a loop with more than

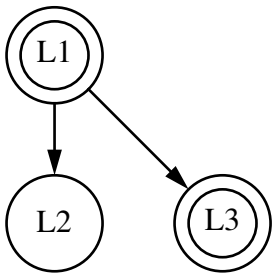
one carried dependence that cannot be eliminated through alignment or distribution. We might begin by maximally distributing the loop over the statements in its body, reducing the loop to a collection of subloops each of which is either parallel or contains a recurrence and hence must be run sequentially. If we apply this transformation to the example in the introduction, we get the following code:

```
L1:  DO I = 1, N
        A(I) = B(I) + 1
     ENDDO
L2   DO I = 1, N
        C(I) = A(I) + C(I-1)
     ENDDO
L3   DO I = 1, N
        D(I) = A(I) + X
     ENDDO
```

Note that we can represent the resulting collection of loops as a directed graph in which the nodes represent loops and the edges represent dependences between the loops as in Figure 6.7.

FIGURE 6.7

Inter-loop dependence graph.



In this diagram the double circles are used to indicate parallel subloops that arise from distribution. It is clear from the examination of this graph that loops L1 and L3 should be fused into a single parallel loop to increase the granularity of parallelism.

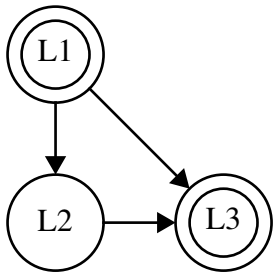
Can we generalize this approach to produce a code generation algorithm? First, we need some way of deciding when we can legally fuse two subloops and when we cannot. Here are a few obvious rules:

1. *Separation Constraint.* Do not fuse a sequential loop with a parallel loop because the result would necessarily be sequential, reducing the total amount of parallelism.

2. *Ordering Constraint.* Do not change the relative ordering of two nodes connected by a dependence edge.

In particular, the ordering constraint implies that you should not fuse loop l_1 into loop l_0 if l_1 depends on another loop l_2 that itself depends upon l_0 but which cannot fuse with l_1. Because of the dependences from l_0 to l_2 to l_1, loop l_2 must be generated *between* l_0 and l_1, but if l_0 and l_1 are fused, this can only happen if l_2 fuses with both. Figure 6.8 contains a modified version of the example in Figure 6.7 in which the result of loop L2 is used in L3. In this case, we cannot fuse L1 and L3 (by the ordering constraint) because L2 cannot be included (by the separation constraint).

FIGURE 6.8 A fusion prevented by the ordering constraint.



One more condition on fusion is required. To understand why, consider the following example code:

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

If we fuse these loops, the loop-independent dependence on `A` will become loop-carried and neither loop will be parallel. (For the moment, we ignore the possibility of using alignment to recover this lost parallelism.)

Definition 6.4. An edge between statements in two different loops is said to be *fusion-preventing* if, after fusing the two loops, the dependence would be carried by the combined loop.

Some fusion preventing dependences actually make the fused loop illegal:

```
DO I = 1, N
```

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

If the loops are fused, the dependence involving A becomes a backward carried antidependence, indicating that the meaning of the program has been changed. Note that some definitions restrict the term fusion preventing to this kind of dependence—one that changes the meaning. However, for the purpose of parallelization, we will use the more inclusive definition because we will not want to fuse loops if that will result in a sequential loop. Thus, our third rule becomes:

3. *Fusion-Preventing Dependence Constraint.* Do not fuse two parallel loops if there is a fusion-preventing dependence between them.

We are now ready to formulate a general fusion procedure. This procedure begins with a graph of parallel and sequential subloops that have resulted from a maximal loop distribution. The principal idea behind the algorithm is to identify at each stage a *cohort,* consisting of a collection of loops that can be run in parallel.

This collection may contain a number of parallel loops, all of which can be fused into a single parallel loop, along with some number of sequential loops which can be run in parallel with the iterations of the parallel loop.

> **Definition 6.5.** A *parallel cohort* is a collection of sequential and parallel loops such that (1) no two parallel loops in the cohort are joined by a fusion-preventing edge and (2) there are no dependence edges between parallel and sequential loops in the cohort.

If one sequential loop in the cohort depends on another, it can be fused with its predecessor, unless the edge joining them is fusion-preventing (in the weaker sense discussed above). Note that we would not want to fuse sequential loops that are not related by a dependence, because this would reduce the available parallelism. In the algorithm below, we will not consider any edge between two sequential loops to be fusion-preventing, because fusion of sequential loops does not really save us much running time.

FIGURE 6.9 Parallel loop fusion.

```
/* The graph G = (N,E) represents the subloops of the original loop */
/* C will be used to represent the current cohort                   */
```

```
/* X is the set of loops that have been eliminated from consideration */
/*     for inclusion in the current cohort C                          */
/* W is a worklist of loops under consideration for membership in C  */
/* S is a worklist of loops to be included in the next cohort after C  */

for each node l ∈ N do
    count(l) := the number of predecessors of l;

S := the set of all elements with count 0;
while S ≠ Ø do begin
l1:  W := S;
    X := Ø;
    while W ≠ Ø do begin
        pick an arbitrary element l ∈ Wand remove it from W;
        C := C ∪ {l};
        for each successor k of l do begin
            count (k) := count (k) – 1;
            if k is a successor by a non-fusible edge then
                X := X ∪ {k}
            end if;
            if count(k) = 0 then
l2:             if k ∉ X
                    then W := W ∪ {k}
                    else S := S ∪ {k}
                endif
            end if
        end
    end;

    publish C as the next parallel cohort;
end
```

This algorithm does not produce a correct fusion according to the three constraints stated earlier because a cohort may contain both parallel and sequential loops. However, if we pose the following problem

> **Theorem 6.6.** The algorithm in Figure 6.9 is correct, in that it finds only correct parallel cohorts, and it produces the minimal number of barriers.

Proof. Correctness follows from the observation that a node is added to the worklist *W* only when all of its predecessors have been assigned to a previous cohort (line *l1*) or when all of its predecessors not in a previous cohort are not joined to it by a non-fusible edge (line *l2*). Note here that

the set X contains all those nodes that cannot be added to the current cohort because of the existence of a non-fusible edge—either a fusion preventing edge between parallel loops or an edge between loops of different types.

The proof of minimality is more difficult. The algorithm is *greedy* because it adds to the current cohort every node that it can before going on to the next cohort. Since the greedy algorithm will always produce the same collection of cohorts, we will refer to this as the greedy collection $C_G=(g_1,g_2,...,g_n)$. Assume there exists a collection of legal parallel cohorts $C = (c_1,c_2,...,c_{n-1})$, with one fewer cohort than the greedy collection. Then there must be a first integer j, such that $1 \le c_j \le n-1$, $c_j \neq g_j$, and $c_i=g_i$ for $1 \le i < j$. Since g_j contains everything that can be put into the next cohort after $g_{j-1}=c_{j-1}$, we must have $c_j \subset g_j$. But this means that we could move all the elements in $g_j - c_j$ into c_j without causing a conflict. Thus we have established that there is a collection of $n-1$ cohorts with $(g_1,g_2,...,g_j)$ as its first j cohorts. We can continue in this manner until we have established that there is a collection of $n-1$ cohorts with $(g_1,g_2,...,g_{n-1})$ as its first $n-1$ cohorts. But this means that $(g_1,g_2,...,g_{n-1})$ is a complete collection of cohorts, contradicting the assumption that the greedy algorithm would not produce a minimal set. □

Although this method minimizes the number of barriers, it could be unsatisfactory because of the handling of sequential loops in the system. It is possible that a very large sequential loop would dominate a cohort generated by this algorithm, when a better load balance would be achieved by splitting the sequential loop between two cohorts.

6.4.2.1 Typed Fusion

We will now consider another code generation model that does not mix parallel and sequential code. In this model, a parallel loop would be terminated by a barrier and all sequential code that must follow that parallel loop or precede the next would appear immediately after the barrier. We can generate optimal code in this model by fusing the parallel loops using an algorithm that can deal with graphs that contain nodes of more than one type, where sequential loops and parallel loops would have different types.

In Figure 6.10, we present the algorithm *TypedFusion*, due to Kennedy and McKinley, that produces an optimal greedy fusion of the nodes of a given type t_0 in a graph that contains fusion-preventing edges and nodes of several types. This algorithm is fast in that it takes $\mathbf{O}(N+E)$ steps in the

worst case, where N is the number of vertices and E is the number of vertices in the graph G. Since any algorithm must traverse the graph at least once, this is clearly the minimum possible.

The basic idea of this algorithm is that, as each node is visited for the first time, the algorithm determines in constant time the exact node of the same type into which it would be fused by the greedy algorithm. If no such node exists, the node is assigned a separate node number. In essence, the algorithm in Figure 6.10 carries out greedy fusion for the single selected type.

> **Definition 6.6.** We define a *bad path* for type t to be a path that begins with a node of type t and either contains a fusion-preventing edge between two nodes of type t, or a node of type different from t.

The algorithm treats a bad path as a fusion-preventing edge. Only nodes of the same type are considered for fusion.

TypedFusion achieves an optimal O(N+E) time bound by choosing, as each new node of the selected type is visited, the correct node with which to fuse in constant time. Intuitively, n can fuse with any of its ancestors from which there is no bad path, but it cannot bypass its predecessors to fuse with an ancestor. It can also fuse with nodes that are neither ancestors nor descendants. *MaxBadPrev*(n) determines the first node m of type t_0 from which there is no bad path to n. It contains the maximum number of a node of type t_0 from which there exists a bad path to the node n being visited. We show that n cannot fuse with that node or any node with lower number, where nodes of type t_0 are numbered by the algorithm in the order whenever they cannot be fused with a previous node of the same type.

We need some neat examples here to show how all this works

The algorithm therefore computes the fusion node by taking the number of the next node of type t_0 after *maxBadPrev*(n). If the result is 0, then n receives the next unassigned number. Consequently, n fuses with the lowest-numbered node of type t_0 from which there is no bad path to n. Clearly, this results in a greedy fusion.

FIGURE 6.10 Typed fusion algorithm.

TypedFusion (G, T, t_0)

INPUT:

G = (N,E) the original typed graph

```
    T is a set of types
    t0 is a specific type for which we will find a minimal fusion
    type(n) is a builtin function that returns the type of a node
OUTPUT:
    L a linear list of fused loops consistent with the constraints
INTERMEDIATE:
    num(n) is the number of the first visit to node n of type t0.
    lastnum is the most recently assigned number.
    maxBadPrev(n) is MAX{num(x) | type(x) = t0 and
        there exists a bad path for t0 from x to n}.
    node(i) is an array that maps numbers to nodes such that
        node(num(x) = x).
    visited is the number of the first node of type t0 in the graph.
    next(i) maps the ith node to the number of the next node of
        the same type.
    W is a working set of nodes ready to be visited.
ALGORITHM:
    /* Initialization */
    lastnum := 0; count(*) := 0; visited := 0; node(*):= 0;
    for each edge e = (m,n) ∈ E docount(n) := count(n) + 1;
    for each node n ∈ N do begin
        maxBadPrev(n) = 0; num(n) := 0; next(n) := 0;
        if count(n) = 0 then W := W ∪ {n} end if;

    /* Iterate over working set, visiting nodes, fusing nodes of type t0 */
    while W ≠ ∅ do begin
        let n be an arbitrary element in W; W := W–{n}; t := type(n);
l1:     if t  = t0 then /* A node of the type being worked on */
            /* Compute node to fuse with. If none, assign a number */
            /*       and add to visited.*/
            if maxBadPrev(n) = 0 then p := visited
            else p := next(maxBadPrev(n)) end if;
            if p ≠ 0 then
                x := node(p);
                num(n) := num(x); maxBadPrev(n) :=
                    MAX(maxBadPrev(n), maxBadPrev(x));
                fuse x and n and call the result n,
                    making all edges out of either be out of n;
            else /* a new node */
                lastnum := lastnum + 1; num(n) := lastnum;
                node(num(n)) := n;
                /* append node n to the end of visited */
                if lastnum = 1 then visited := lastnum
```

```
            else next(lastnum–1) := num(n) end if;
        end if; end if;

        /* Update maxBadPrev for successors and
        /* add to W as appropriate */
l2:     for each node m such that (n,m) ∈ E do begin
            count(m) := count(m) - 1;
            if count(m) = 0 then W := W ∪ {m} end if;
            if t ≠ t0 then
                maxBadPrev(m) := MAX(maxBadPrev(m), maxBadPrev(n))
            else /* t = t0 */
                if type(m) ≠ t0 or (n,m) is fusion-preventing then
                    maxBadPrev(m) := MAX(maxBadPrev(m), num(n))
                else /* equal types and not fusion preventing */
                    maxBadPrev(m) := MAX(maxBadPrev(m),
                        maxBadPrev(n))
                end if;
            end if;
        end
    end while;
```

We note that if there is only one type, this algorithm is an *O(N+E)* algorithm for untyped fusion, the fastest such algorithm known.

Theorem 6.7. Algorithm typed fusion produces a correct fusion, in that it obeys the fusion constraints, and it yields the smallest possible number of nodes of type t_0 after fusion.

Proof. **Correctness.** To establish correctness, we must show that the constraints in the problem definition are observed. First, the separation constraint is clearly satisfied, because the algorithm fuses only nodes of the selected type. To show that it obeys the fusion-preventing constraint, we must establish that it never fuses two nodes joined by a fusion-preventing edge. To show that it satisfies the ordering

constraint, we must show that it never fuses two nodes joined by a path that passes through a node of different type. Since all original edges are left in the graph, this ensures that fusion does not create a cycle in the output DAG. Both of these hypotheses are established by the following Lemma, which relies on the greedy structure of the algorithm.

Lemma 6.2. For each n taken from the worklist W in *TypedFusion*, n cannot fuse with a previously visited node x if there exists there exists a previously-visited node $y \neq x$ such that $type(y) =$

type(n) = t_0, *num*(y) ≥ *num*(x) and y cannot legally be fused with n.

Proof. Suppose there is a node for which the lemma does not hold, then there must exist some first element n_1 for which they do not hold. Since the algorithm always picks the first node of type t_0 in visit order, assume that previous steps of the algorithm always correctly fused with the earliest possible node of the same type. Recall that a *bad path* for type t_0 is one that begins at a node of type t_0 and either contains a fusion-preventing edge between two nodes of that type or passes through a node of a different type. We can fuse n_1 with any node of type t_0 as long as there is no bad path from that node to n_1.

Assume that n_1 can fuse with x even though there exists a y of the same type and larger number that cannot fuse with n_1. Since previous steps of the algorithm correctly fuse with the lowest-numbered node of compatible type, we must assume that y did not fuse with x because it would have been incorrect to do so. Consequently, there must be a bad path for *type*(n_1) from x to y. But since y cannot fuse with n_1, there must also be a bad path from y to n_1. But that means that there must be a bad path from x to n_1, so the nodes cannot be fused. □

The implication of this lemma is that we need only remember the highest-numbered node from which there is a bad path for t_0 to the node being considered. The next node of that type that is visited prior to visiting node n is the node with which n must fuse. If the next visited node is n itself, then it must be assigned a new number because it cannot fuse with any previous node.

TypedFusion computes for each node m the quantity *maxBadPrev*(m), which is the highest numbered node of type t_0 from which there is a bad path for type t_0 to m. A node n may not fuse with a node numbered *maxBadPrev*(n) or lower, by the lemma above. It could, however fuse with p = *next*(*maxBadPrev*(n)).

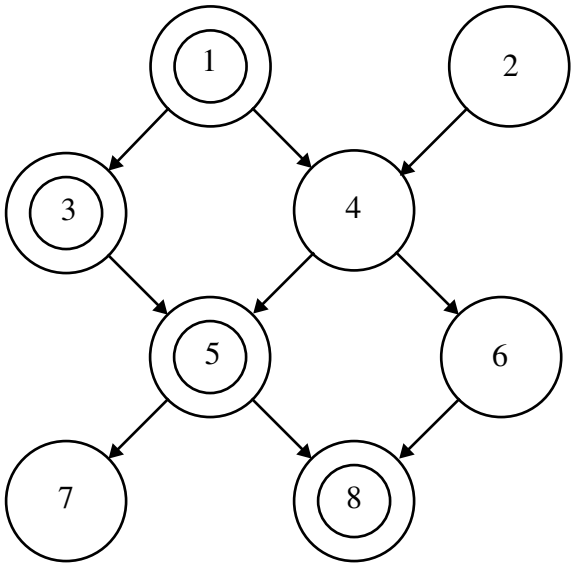
Since all edges in the original graph are left in the final graph (adjusted to reflect fusion) and we cannot fuse nodes joined by a bad path, the output graph after fusion is acyclic. □

Optimality. A greedy algorithm for fusing nodes of the same type repeatedly selects loops from a set for which all predecessors have been assigned to a partition and adds them to the current partition until no more can be added. At that point, it starts a new current partition. In his

dissertation, Callahan proved that if a greedy algorithm is employed on a DAG where all the nodes are of the same type, the resulting fusion is optimal, *i.e.*, it has the smallest possible number of partitions~\cite{Callahan:Dissertation}. The proof here is similar and will be reviewed in the full paper. *TypedFusion* carries out the greedy algorithm for a selected type because, for each node n, it finds the lowest-numbered node in $visited(type(n))$ that can be fused with n. □

As an example of how this partitioning algorithm works, consider the example in Figure 6.11.

FIGURE 6.11 An example subloop graph.



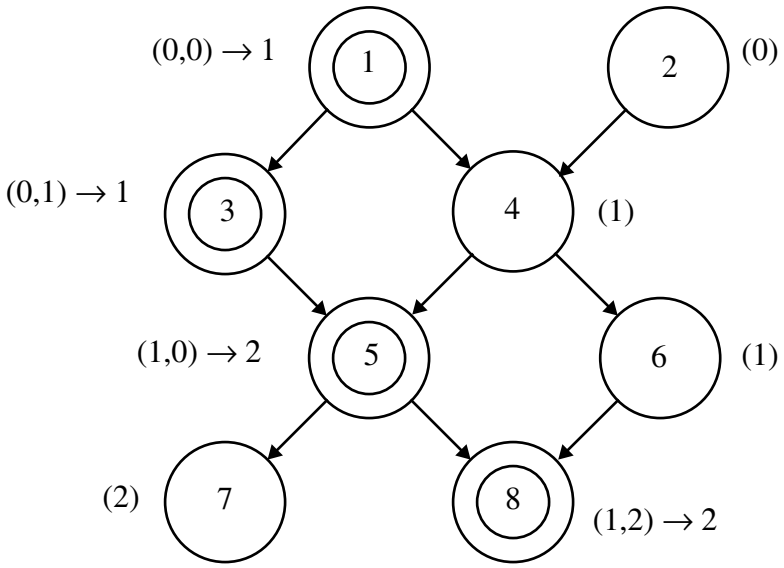
When the algorithm is applied to the nodes of type "parallel," indicated by the double circle, the intermediate annotations are shown in Figure 6.12. Each parallel node is annotated by a tuple of the form

$$(maxBadPrev, p) \rightarrow num$$

where p is as computed in the algorithm and *num* is the final number assigned to the node. Nodes of type "sequential" are annotated with a singleton (*maxBadPrev*) as computed by the algorithm for nodes of type other than t_0.

FIGURE 6.12

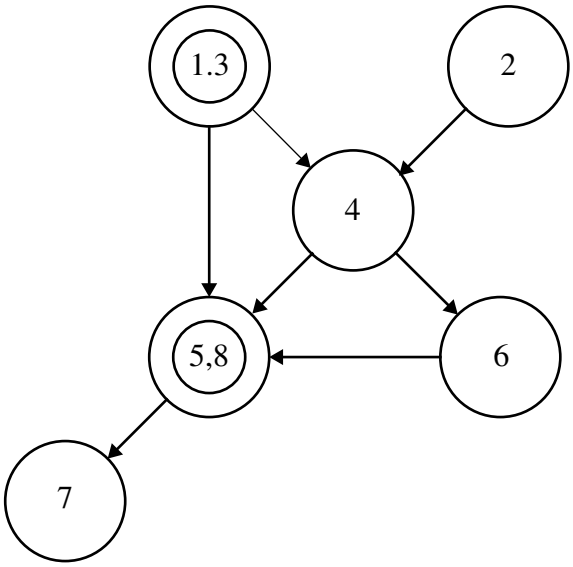
Annotated version of the example graph for type = "parallel."



The result after fusion is shown in Figure 6.13.

FIGURE 6.13

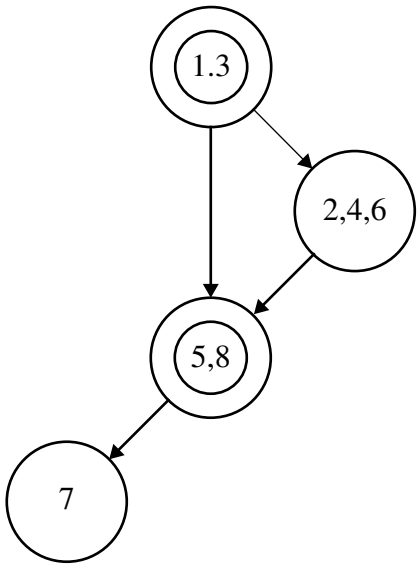
Example after fusing parallel loops.



Note that we can now apply the algorithm *TypedFusion* a second time to fuse all of the sequential loops. We do not necessarily need to fuse them,

but it is useful to know how many sequential "epochs" are needed. The final result is shown in Figure 6.14.

FIGURE 6.14 Example after fusing sequential loops.



It is now clear that the desired schedule is:

1. Parallel loop containing original loops 1 and 3;
2. Sequential loop containing original loops 2, 4 and 6;
3. Parallel loop containing original loops 5 and 8; and
4. Sequential loop containing the original loop 7.

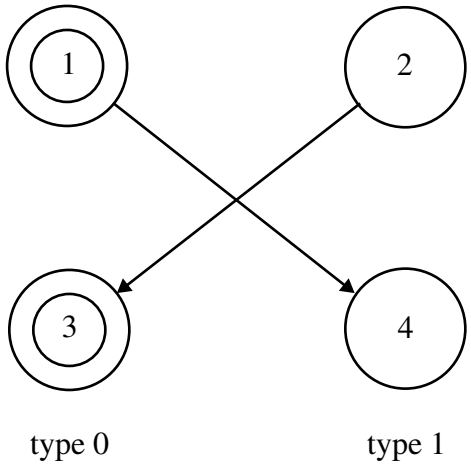
It should be noted that none of the steps in this process takes more than linear time in the size of the loop graph after distribution.

6.4.2.2 Unordered and Ordered Typed Fusion

Once we have developed the notion of typed fusion, we can envision many applications for it. An important example will be given later in the chapter. For now consider the problem of dealing with incompatible loop headers. It may not be convenient or profitable to go to the trouble to fuse two loops that have loop headers that are not compatible because they have different iteration ranges. If we let each loop header be a different type, we can restrict fusion to compatible loop headers by applying algorithm *TypedFusion* once for each type.

It is natural to ask: does this always produce the minimum total number of nodes. To understand this issue, consider the simple dependence graph in Figure 6.15, which illustrates a *type conflict.* Although we can fuse either type separately, we cannot fuse both types because the fusion of one type creates a bad path for the other type. Thus we must select which of the two types we wish to fuse.

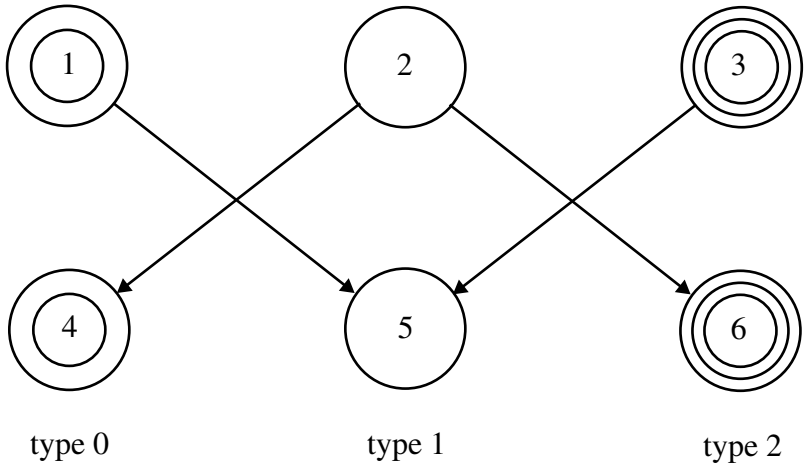
FIGURE 6.15 A type conflict.



The example in Figure 6.16 shows how order can make a difference in the quality of the fusion. If we were to choose to fuse type 1 before either type 0 or type 2, we would create bad paths for both types 0 and 2, preventing all other fusions. However, if we choose to fuse type 0 first, we will create a bad path only for type 1, permitting the two type 2 nodes to fuse as well. The first approach yields a total of 5 nodes, while the second yields 4, a clearly better result.

Is there any way to determine the order in which to fuse types that are otherwise undistinguished. The answer is yes, but it is costly. Kennedy and McKinley have proved that the problem is NP-hard in the number of types [Kennedy-McKinley 93]. Fortunately, type conflicts are fairly rare, so a good heuristic for choosing order may be effective.

FIGURE 6.16 An order-sensitive fusion problem.



The situation is much simpler for the *ordered typed fusion problem,* in which the types can be prioritized according to some criterion, so that it is always more important to minimize the number of nodes of a higher-priority type, no matter how many bad paths for lower-priority types are created. In this case a simple algorithm which calls *TypedFusion* for each type in descending priority order will produce the "optimal result."

FIGURE 6.17 Ordered typed fusion algorithm.

OrderedTypedFusion (*G, T, priority*)

INPUT:

$G = (N,E)$ the original typed graph
T is a set of types
t_0 is a specific type for which we will find a minimal fusion
type(n) is a builtin function that returns the type of a node
priority[t] is an integer priority assigned for each type; note that priorities are unique, i.e., *priority*[t_1] = *priority*[t_2] $\Rightarrow t_1 = t_2$

OUTPUT:

A graph $G' = (N',E')$ in which no further fusion is possible

ALGORITHM:

/* Initialization */
$G' := G$;
for each type t in order of descending priority **do**

TypedFusion (*G', T, t*);

In reality, total orders for types are hard to come by. It is far more typical to find a partial order for types. In other words, in fusion problems that arise in practice there will usually be only a few priority rules that favor one type over another. We will deal with such problems by the procedure in Figure 6.18, which uses topological sort to establish a total priority order for the graph and then call *OrderedTypedFusion*.

FIGURE 6.18

General heuristic typed fusion algorithm.

GeneralTypedFusion (*G, T,* R_T)

INPUT:

$G = (N,E)$ the original typed graph

T is a set of types

R_T is a set of priority rules (t_1,t_2) specifying that t_1 is to be given priority over t_2.

type(*n*) is a builtin function that returns the type of a node

INTERMEDIATE:

priority(*t*) is an integer priority assigned for each type

OUTPUT:

A graph G' = (*N',E'*) in which no further fusion is possible

ALGORITHM:

Apply topological sort to the graph $G_T = (T,R_T)$ to compute a total ordering stored in *priority.*

OrderedTypedFusion (*G, T, priority*);

6.4.3 Multilevel Loop Fusion

The problem of fusion becomes even more difficult when dealing with loop nests because each loop nest could have a different optimal permutation. If we assume that we will only fuse loops that were originally fused in the loop nest before distribution, then different loop permutations can interfere with loop fusion. For example, consider the following loop nest:

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

After distribution, it is clear that each statement will do better with a different outer loop as shown here:

```
PARALLEL DO I = 1, N
   DO J = 1, M
      A(I,J+1) = A(I,J) + C
   ENDDO
END PARALLEL DO

PARALLEL DO J = 1, M
   DO I = 1, N
      B(I+1,J) = B(I.J) + D
   ENDDO
END PARALLEL DO
```

Although each nest has outer loop parallelism, the two nests cannot be fused because they have different outer loops.

It has been shown that loop fusion in the presence of loop interchange is NP-Complete in the number of loops. The following loop illustrates one of the reasons that this problem is so hard.

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

This nest distributes over all four statements to yield the four loops:

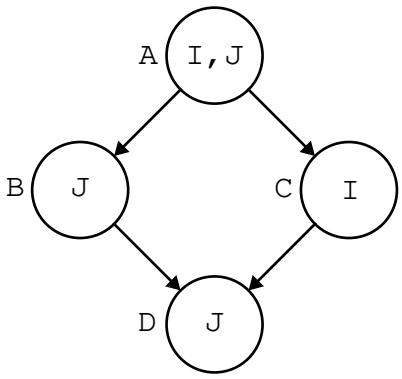
```
DO I = 1, N ! Can be parallel
   DO J = 1, M ! Can be parallel
      A(I,J) = A(I,J) + X
   ENDDO
ENDDO

DO I = 1, N ! Sequential
   DO J = 1, M ! Can be parallel
      B(I+1,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO

DO I = 1, N ! Can be parallel
   DO J = 1, M ! Sequential
      C(I,J+1) = A(I,J) + C(I,J)
   ENDDO
ENDDO

DO I = 1, N ! Sequential
   DO J = 1, M ! Can be parallel
```

```
            D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
         ENDDO
      ENDDO
```

The problem with this loop can be illustrated by the graph in Figure 6.17. In this diagram, each subloop is represented by a node labeled by the variable that is assigned in that subloop. Inside each node the loop indices of the loops that can be parallel are listed.

FIGURE 6.19

Two-level fusion example.



The problem is to determine which loop to fuse the A-loop with. The A-loop produces values which are used in both the B- and C-loops. Hence we would like to fuse with each of those loops. However we cannot fuse with both without giving up parallelism. If we fuse the A-loop with the B-loop, we will need to make the loop on J parallel in the resulting nest. But then a subsequent fusion with the C-loop would force both the I- and J-levels of the result to be sequential. So we must choose. The problem is that we cannot tell by just examining the successors. Although they both look equally good at this point, it is definitely advantageous to fuse with the C-loop. To see why, suppose we fuse with the B-loop instead. The resulting loop could not be fused with the C-loop for the reason we just described. Furthermore, it cannot be fused with the D-loop because we cannot fuse two loops if there is a path in the dependence graph from one to the other that passes through a loop that is not fused into the group. In this case, the path through the C-loop forces us to keep the D-loop separate. The result has two barriers and three cohorts:

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

```
   ENDDO
END PARALLEL DO

PARALLEL DO I = 1, N
   DO J = 1, M
      C(I,J+1) = A(I,J) + C(I,J)
   ENDDO
END PARALLEL DO

PARALLEL DO J = 1, M
   DO I = 1, N
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
   ENDDO
END PARALLEL DO
```

On the other hand, if we fuse the A- and C-loops, we can then fuse the B- and D-loops, which would leave only one barrier and two loop nests.

```
PARALLEL DO I = 1, N
   DO J = 1, M
      A(I,J) = A(I,J) + X
      C(I,J+1) = A(I,J) + C(I,J)
   ENDDO
END PARALLEL DO

PARALLEL DO J = 1, M
   DO I = 1, N
      B(I+1,J) = A(I,J) + B(I,J)
      D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
   ENDDO
END PARALLEL DO
```

This loop is clearly superior in terms of its data reuse and number of processors. The problem is that lookahead is required to determine what to do in situations where fusion decisions need to be made. As it happens, lookahead of arbitrary length can be required.

Because of these problems, most parallel code generation systems use some sort of heuristic to determine how to fuse loops with multiple levels. An example of a heuristic is to fuse with the loop that cannot be fused with one of its successors. The rationale behind this heuristic, which incidentally solves the example problem, is that if a successor cannot be fused with its successor, then you are going to need at least one barrier anyway. So why immediately chose to have two by putting another barrier in front of that successor. This heuristic has been used effectively in at least one parallel code generation system.

Multilevel code generation algorithm here.

6.4.4 Profitability-Based Methods

Because of the need for a minimum granularity for parallelism to be profitable and because there are many alternatives to be considered in parallel code generation, practical methods will need to employ some method for estimating the cost of different code arrangements. One study we conducted at Rice used a particularly effective strategy for discovering parallelizable loops in a Fortran to parallelize many loops that could not be parallelized by existing commercial compilers. Unfortunately, once these loops were parallelized, the code ran substantially slower than before. The problem was insufficient granularity in the newly parallelized loops.

In light of these observations, we can see that some form of performance estimation will need to be incorporated if parallel code generation is to be effective. We will consider the use of a static performance estimation function C, which can be applied to any code fragment to estimate the running time of that fragment. It is not so important that such a function be accurate, but rather that it be good for selecting the better of two alternative code arrangements. In developing a static estimator, key considerations will be the cost of memory references in the code fragment and whether the granularity associated with a parallel loop is sufficient to make parallelization profitable.

One way to use such a cost function in code generation for a loop nest would be to consider all possible permutations and parallelizations and pick the best one. This is impractical for two reasons. First, the total number of alternatives is exponential in the number of loops in a nest and the cost of evaluating the cost estimate is usually significant, which would make this strategy unacceptably slow. Second, many of the loop upper bounds in a loop nest will not be known at compile time. Thus, it may be impossible to precisely estimate the running time. This might be overcome by generating a variety of loop arrangements and selecting the best one at run time, but the number of such arrangements is likely to cause an explosion in code size.

For these reasons, it is common to consider only a subset of the possible code arrangements, based on properties of the cost function itself. As an illustration, we will consider an example method for generating code for a loop nest. This example is based on work by Kennedy and McKinley [].

6.4.4.1 Memory-Based Code Generation

In this section, we will discuss a code-generation heuristic that uses the cost of memory references as the primary consideration in selecting the right permutation for a perfect loop nest. We begin by developing a mea-

sure C_L of the cost of a particular loop when it is innermost in the nest. This cost is essentially an upper bound on the number of cache misses generated by the loop when it is innermost. We shall call this measure *loop cost.* To evaluate loop cost for a given loop, we perform three steps.

1. Subdivide all the references in the loop body into *reference groups*. Two references are in the same reference group if there is a loop independent dependence or a constant-distance carried dependence from one to the other. Intuitively, a cache miss for the first of two references in the same reference group will be the only one experienced by the group because the second reference will find the desired line in cache.
2. For each reference group, determine whether subsequent accesses to the same reference are (a) loop invariant, (b) unit stride or (c) non-unit stride. For case (a) assign the group a reference cost of 1, because only one cache miss will be experienced for the entire loop. For case (b) assign the group a reference cost of equal to the number of iterations of the loop divided by the cache line size (in data items of the type referenced) because of reuse within the cache line. For case (c), assume no reuse within a cache line and assign a cost equal to the number of iterations in the loop.
3. Assign the loop a cost equal to the sum of the reference costs times the aggregate number of times the loop will be executed if it is innermost in the loop nest. This is essentially equal to the product of the loop bounds for the outer loops.

As an example, consider the matrix multiplication inner loop.

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

When the K-loop is innermost the reference cost for C is 1 because it is loop invariant. The reference cost for A is N, because it is non-unit stride and the reference cost for B is N/L, where L is the cache line size, because of the unit stride. Hence the total loop cost for the K-loop is $N^3(1+1/L)+N^2$. When the J-loop is innermost, the cost is $2N^3+N^2$, because the C and B references are non-unit stride and the A reference is invariant. When the I-loop is innermost, we have two unit-stride reference groups and one invariant for a cost of $2N^3/L+N^2$. Thus, the I-loop

has the lowest loop cost and is the preferred innermost loop, from the perspective of cache reuse.

To find a best total order, we simply order the loops from innermost to outermost by increasing loop cost, on the theory that inner loop reuse could also be outer loop reuse if the cache is big enough. This would imply that the following loop order should be selected for best cache performance:

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

Even if the outer loop cache reuse is not achieved, it might be induced by strip mining.

Once a desired loop order is established by the loop cost heuristic, the next step would be to put the loops into a permutation that is as close as possible to the desired permutation—it may not be possible to achieve exactly the desired loop order because some permutations will be illegal. An algorithm for doing this is given in Figure 6.18

FIGURE 6.20

Selecting a nearby permutation.

Input: A desired loop permutation order $O = (i_1,i_2,...,i_n)$ and the original direction matrix D for the loop.

Output: A nearby permutation P.

$P := \emptyset$;
for $i := 1$ **to** n **do begin**
    select the leftmost loop l in O such that $(P_1,P_2,...,P_{i-1},l)$ has no illegal direction vector prefixes;
    remove l from O;
    append l to the end of P, so that it becomes P_i;
end

It is easy to show that if any legal permutation has the desired innermost loop in the innermost position, this algorithm will find a permutation which has the desired loop innermost [Kennedy and McKinley].

This memory-based loop interchange algorithm is but one heuristic approach to folding profitability information into a code generation scheme without suffering exponential execution time. Experiments have shown that it is quite effective in practice [].

6.5 A Code Generation Algorithm

In this section we develop a code generation algorithm for the general case—a loop nest with hierarchy. The main problem here is to decide when to use loop distribution and fusion and which loops to fuse once parallelization of the separate nests is done.

As usual, we begin with a top-down scheme that attempts to parallelize the outermost loop nest. If it succeeds in finding sufficient parallelism, no further effort is required. Otherwise, the outer loops are distributed in an attempt to find more parallelism on inner loops. Throughout we assume that we have a routine, called ParallelizeNest, that finds parallelism in the outermost perfect nest and estimates its profitability. The algorithm is given in Figure 6.19.

FIGURE 6.21 A general code generation algorithm.

```
procedure Parallelize(l );

/* l is the loop nest to be parallelized */
ParallelizeNest(l, success);
    if ~success then begin
        distribute l into loop nests l_1,l_2,...,l_n;
        for i := 1 to n do Parallelize(l_i);
        MergeNests({l_1,l_2,...,l_n});
    end
end Parallelize
```

The key component of this algorithm is the routine *MergeNests*, which fuses the parallelized loops back together whenever possible.

MergeNests uses a method similar to the one used in the algorithm for sequential–parallel fusion given in Figure 6.10. The basic idea is to classify the various loop nests by *type*, where each type represents the outermost loop in the original nest that can be made outermost in the distributed nest without changing one of the critical performance factors, such as parallelism, granularity or memory performance in the innermost

position. As an example, consider the loop nests below. Furthermore, two loops are of different types if the canonical outermost loop in one is parallel and the sequential in the other. As an example, consider the loop nest below.

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

Neither loop in this nest can be parallelized, so it is distributed:

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

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

Each of these loops can be parallelized, the second in two dimensions:

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

PARALLEL DO J = 1, M
   PARALLEL DO I = 1, N
      X(I,J) = A(I,J) + C
   ENDDO
ENDDO
```

The type of the first loop nest is (I–loop, parallel). The type of the second loop nest is (J–loop, parallel). Thus these loops have different types and cannot be merged at the outermost level.

Once the loops are partitioned by type, the outermost loop can be fused and then the loops can be re-merged into a single program.

FIGURE 6.22 Loop fusion for a collection of loop nests.

procedure *MergeNests*(*S*)

/* *S* is the set of loop nests to be parallelized */

let $\{t_1, t_2,...,t_m\}$ be the collection of loop types;
fuse the outermost loops of each type using the algorithm of
Figure 6.18 on page 217;
for each fused group *G* **do begin**
let $\{l_1,l_2,...,l_k\}$ be the loop nests inside fused group *G*;
MergeNests($\{l_1,l_2,...,l_m\}$)
end;
merge the loops of different types, again following Figure 6.10;
end *MergeNests*

The algorithm *MergeNests* in Figure 6.20, will fuse loops whenever it is legal to do so and doing so does not give up a critical performance factor such as total parallelism. Thus it would fail to merge the two loops below, where loops with the same index came from the same original loop.

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

PARALLEL DO J = 1, M
   PARALLEL DO I = 1, N
      X(I,J) = A(I,J) + C
   ENDDO
ENDDO
```

Outer loop fusion could be achieved in this example if the outer loop in the second loop could be sequentialized, but this would give up total parallelism.

One weakness of this scheme is illustrated by another pair of sample loops:

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

PARALLEL DO J = 1, M
   PARALLEL DO I = 1, N
      X(I,J) = A(I,J) + C
   ENDDO
```

```
ENDDO
```

The fusion algorithm in *MergeNests* would fail to interchange the parallel loop on `I` in the second loop; thus it would classify these loop nests as having different types, missing an opportunity to fuse a parallel loop at the outermost level. We note that if these loops were the only ones arising from an original loop nest, the original would be:

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

For this nest, the first call to *Parallelize* would find the `I`-loop and parallelize it at the outermost level. However, there are examples that generate a similar situation, but cannot be parallelized as a single loop, such as this example, slightly modified from Section 6.4.3:

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

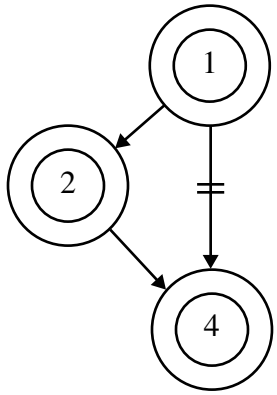
After distribution and parallelization, this becomes:

```
L1   PARALLEL DO J = 1, M
        PARALLEL DO I = 1, N
           A(I,J) = A(I,J) + X
        END PARALLEL DO
     ENDDO

L2   PARALLEL DO J = 1, M
        DO I = 1, N
           B(I+1,J) = A(I,J) + B(I,J)
        ENDDO
     END PARALLEL DO

L3   PARALLEL DO I = 1, N
        DO J = 1, M
           C(I,J+1) = A(I,J) + C(I,J)
        ENDDO
     END PARALLEL DO

L4   PARALLEL DO J = 1, M
```

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

The first second and fourth loops will be classified as a single type. The graph for this type will be as shown in Figure 6.21.

FIGURE 6.23 Parallel and sequential subgraphs.



A greedy fusion algorithm will merge loops L_1 and L_2, leaving loop L_4 separate. The result, after remerging types is:

```
L1  PARALLEL DO J = 1, M
       PARALLEL DO I = 1, N
          A(I,J) = A(I,J) + X
       END PARALLEL DO
       DO I = 1, N
          B(I+1,J) = A(I,J) + B(I,J)
       ENDDO
    END PARALLEL DO

L3  PARALLEL DO I = 1, N
       DO J = 1, M
          C(I,J+1) = A(I,J) + C(I,J)
       ENDDO
    END PARALLEL DO

L4  PARALLEL DO J = 1, M
       DO I = 1, N
          D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
       ENDDO
    ENDDO
```

As we pointed out in Section 6.4.3 on page 216, this is not the optimal merge for this example. However, we are prepared to live with it, given that the problem is NP-complete. Of course, we could use a heuristic like the one suggested in Section 6.4.3, to improve the code slightly.

6.6 An Example

We will conclude the section on code generation in loop nests with a discussion of a particular sample code. The code we choose is a subroutine of ERLEBACHER, a NASA differential equation solver written by Thomas M. Eidson.

FIGURE 6.24 Subroutine *tridvpk* from ERLEBACHER.

```
L1   DO J = 1, JMAXD
        DO I = 1, IMAXD
           F(I,J,1) = F(I,J,1)*B(1)
        ENDDO
     ENDDO

L2   DO K = 2, N-1
        DO J = 1, JMAXD
           DO I = 1, IMAXD
              F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1)*B(K)
           ENDDO
        ENDDO
     ENDDO

L3   DO J = 1, JMAXD
        DO I = 1, IMAXD
           TOT(I,J) = 0.0
        ENDDO
     ENDDO

L4   DO J = 1, JMAXD
        DO I = 1, IMAXD
           TOT(I,J) = TOT(I,J)+D(1)*F(I,J,1)
        ENDDO
     ENDDO

L5   DO K = 2, N-1
        DO J = 1, JMAXD
           DO I = 1, IMAXD
              TOT(I,J) = TOT(I,J)+D(K)*F(I,J,K)
           ENDDO
        ENDDO
```

```
      ENDDO
```

Notice that the loops are already maximally distributed. If we apply the general multilevel code generation algorithm in Figure 6.21 on page 224 to each of these nests, the loops will be parallelized as shown in Figure 6.25. Here note that the loops on index `I` have been left serial to ensure that sequential access takes place on a single processor.

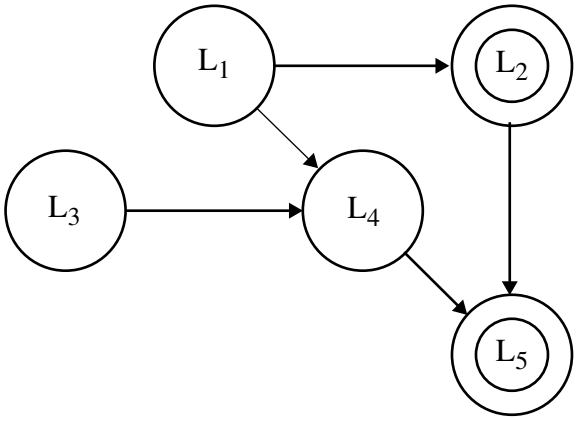
FIGURE 6.25

Subroutine *tridvpk* from ERLEBACHER after single-nest parallelization.

```
L1    PARALLEL DO J = 1, JMAXD
         DO I = 1, IMAXD
            F(I,J,1) = F(I,J,1)*B(1)
         ENDDO
      ENDDO

L2    PARALLEL DO J = 1, JMAXD
         DO K = 2, N-1
            DO I = 1, IMAXD
               F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1)*B(K)
            ENDDO
         ENDDO
      ENDDO

L3    PARALLEL DO J = 1, JMAXD
         DO I = 1, IMAXD
            TOT(I,J) = 0.0
         ENDDO
      ENDDO

L4    PARALLEL DO J = 1, JMAXD
         DO I = 1, IMAXD
            TOT(I,J) = TOT(I,J)+D(1)*F(I,J,1)
         ENDDO
      ENDDO

L5    PARALLEL J = 1, JMAXD
         DO K = 2, N-1
            DO I = 1, IMAXD
               TOT(I,J) = TOT(I,J)+D(K)*F(I,J,K)
            ENDDO
         ENDDO
      ENDDO
```

Typed fusion is then applied, with loops L_1, L_3 and L_4 assigned one type while loops L_2 and L_5 are assigned another. This produces the fusion graph shown in Figure

FIGURE 6.26 Fusion graph for ERLEBACHER.



Two applications of typed fusion at the top level, followed by fusions of the inner loops, produces the final program shown in Figure 6.27.

FIGURE 6.27 Final code for subroutine *tridvpk* from ERLEBACHER.

```
L1   PARALLEL DO J = 1, JMAXD
        DO I = 1, IMAXD
           F(I,J,1) = F(I,J,1)*B(1)
           TOT(I,J) = 0.0
           TOT(I,J) = TOT(I,J)+D(1)*F(I,J,1)
        ENDDO
     ENDDO

L2   PARALLEL DO J = 1, JMAXD
        DO K = 2, N-1
           DO I = 1, IMAXD
              F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1)*B(K)
              TOT(I,J) = TOT(I,J)+D(K)*F(I,J,K)
           ENDDO
        ENDDO
     ENDDO
```

6.7 Historical Notes

To be written