Automatic Decomposition of Scientific Programs for Parallel Execution[†]

Randy Allen
David Callahan
Ken Kennedy

Department of Computer Science
Rice University
Houston, Texas 77251

Received 11/5/86

Abstract

An algorithm for transforming sequential programs into equivalent parallel programs is presented. The method concentrates on finding loops whose separate iterations can be run in parallel without synchronization. Although a simple version of the method can be shown to be optimal, the problem of generating optimal code when loop interchange is employed is shown to be intractable. These methods are implemented in an experimental translation system developed at Rice University.

1. Introduction

If scientific programs can be effectively partitioned into regions that may be executed in parallel, significant gains should be possible by the use of large numbers of processors. At present, development efforts are beginning on machines with the capability of employing a thousand processors. The NYU Ultracomputer [GGKM 83], the Illinois Cedar system [GKLS 84], the Cosmic Cube [Seit 85], and the IBM RP3 [PBGH 85] have all been designed with the specific intent of effectively utilizing large numbers of processors. But without some mechanism for bringing the power of parallel processing to bear on practical problems, its benefits may be lost. We cannot simply shift the burden of parallel processing to the programmer. Programming for parallel machines is difficult and we must provide powerful software tools to assist the application developer if parallel processing is to be effective.

This paper develops the conceptual foundations of an automatic scheme for uncovering implicitly parallel operations within sequential Fortran programs. The goal of this approach is the detection of DO loops whose iterations can be run independently in parallel. While recognition of parallelism in its fullest context is an undecidable problem [Bern 66], the limited scheme we employ can be efficiently implemented.

The methods described in this paper are implemented in an experimental translator, called PFC, in use at Rice University.

2. Parallelism Model

We assume a large number of processors are available to work on a single application. Each processor has some local memory and access to a shared global memory. Access to a global memory location is not automatically synchronized by the hardware; instead, synchronization primitives are available to ensure that shared data is accessed in a desired sequence.

If we are to effectively use a machine of this type, we must keep a large number of processors busy. This requires a sufficient number of independent computation units of *comparable running time*. If we need more than a handful of such units, it seems reasonable to look for them in DO loops, because the body of a DO loop is executed many times with only slight variations among the iterations; hence the running times should be roughly equivalent.

We prefer to identify loops for which no inter-iteration synchronization is required. In such loops we can block load the inputs into processor local memories and compute independently until all the iterations are

[†] Support for this research was provided by IBM Corporation.

done. Of course, we are still faced with the problem of making sure that each iteration represents a computation of sufficient size to justify the synchronization overhead of a parallel computation [FlaK 86]. This issue will not be treated in detail here.

We will present an algorithm that translates programs written in a sequential language, like Fortran, to parallel programs in the same language augmented with simple constructs to manage multiple processors. The principal method for expressing parallelism will be through a variant of the fork-join construct. All processors simultaneously begin executing a collection of code blocks, some of which are *serial blocks* to be executed by a single processor and some of which are *parallel DO's*, whose separate iterations can be executed concurrently by different processors. Each serial block and each iteration of a parallel DO can be viewed as an independent task; tasks are assigned to processors nondeterministically at execution time. When a processor completes a task, it is reassigned to another task that is ready for execution. This process continues so long as there are tasks remaining. When there are no more tasks to be assigned, the processors wait until all other tasks are complete before beginning the next task set.

Our examples use an enhanced version of Fortran to express parallelism. The wait at the end of a task set is specified by a BARRIER statement, which must be executed by all processors before any can continue on. Serial blocks are enclosed between SERIAL and ENDSERIAL delimiters. The first processor to reach a serial block executes it; all others fall through. A parallel DO is indicated by a DOALL keyword. If there are more processors available than DOALL iterations, surplus processors fall through. While these constructs are inadequate for expressing all forms of parallelism, they are sufficient for discussing the central issues in parallel code generation.

Parallel code generation can be viewed as two stages: *decomposition* and *recombination*. The input program is decomposed into blocks of code that are either serial (to be executed by a single processor) or parallel (a parallel DO). These blocks are then recombined into fork-join constructs as described above. The output program consists of clusters of independent tasks separated by barrier synchronization points.

Our goal is to convert as many serial DO loops as possible into parallel loops to exploit the multiple processors in the target machine while minimizing that amount of synchronization overhead. The primary constraint on parallelism is that certain accesses to shared variables must be explicitly serialized to ensure equivalence with the original source program. This requirement can be stated formally as

> **(P1)** If two processes access a shared memory location and if at least one of those processes modifies that location, then accesses to that location must be serialized (by a barrier synchronization point) to preserve the access order of the original program.

Thus we have three problems: detection of loops that can be parallelized, detection of shared variable access which must be explicitly synchronized, and translation of the original program to parallel form. The approach taken here is to break the original program into serial blocks and parallel loops and then insert barrier synchronization points as necessary. Whether a transformation is profitable depends strongly on the architecture of the target machine. To avoid unnecessary restriction of the machine class, we select optimality conditions that are independent of machine size and memory configuration. Initially, we will use the following optimality criteria.

> **(O1)** *If a statement inside a DO loop can be executed in parallel, then some loop containing it will be parallelized.*

> **(O2)** *Subject to* (O1), *the number of barrier synchronization points will be minimized.*

A more precise and effective definition of optimality will be given in the next section.

3. Dependence

A sequential DO loop provides a precise specification of the order in which statements are to be executed: statements within one iteration are executed in the order in which they appear, while different iterations are executed in an order specified by the control parameters of the DO. In a parallel loop, any given iteration of the loop is executed by a single processor, thereby guaranteeing that the execution constraints imposed by statement order will be preserved. However, a parallel loop cannot guarantee that the order in which the loop iterations are performed will be identical to that specified by the control parameters. Thus, if the parallel version is to compute the same result as the sequential loop, the order of iterations must be immaterial to the computation.

Dependence is a relation between the statements of the program that can be used to preserve execution order in transformation systems. The pair (S_1, S_2) is a member of the relation if and only if S_2 *must* be executed after S_1 in order for the results of the program to be correct. In other words, any transformation that merely reorders the execution of statements is safe so long as it preserves the dependences in the code. Since the conversion of a sequential DO loop into a parallel DO loop merely permutes the execution of different iterations, dependence can be used to precisely determine when this transformation is safe.

Kuck has identified three types of data dependences that must be preserved if the meaning of a give program is to be unchanged by transformations [Kuck 78]:

(1) *true dependence* — S_1 stores into a variable which S_2 later uses.

```
S1      X = ...
S2      ... = X
```

(2) *antidependence* — S_1 fetches from a variable that S_2 later stores into.

```
S1      ... = X
S2      X = ...
```

If S_2 were to be executed before S_1, the value of X used in S_1 would be that computed by S_2, which is not correct.

(3) *output dependence* — two statements both store into the same variable.

```
S1      X = ...
S2      X = ...
```

While neither S_1 nor S_2 are affected by a change in execution order, later statements that use X would receive the wrong value if S_1 and S_2 were reversed in execution order.

Dependences may be further classified into two categories: *loop carried* and *loop independent*. This classification arises from the fundamental requirement for a data dependence—the fact that two statements must reference a common memory location, with control flowing from one to the other. The required control flow can occur in one of two ways:

(1) Control can flow from one statement to the second within a single iteration of all the loops, following sequential execution order, as in

```
            DO 100 I = 1, N
S1                  A(I) = ...
S2                  ... = A(I)
100         CONTINUE
```

S_2 uses the value created by S_1 on the same iteration, thus creating a *loop independent* dependence.

(2) Control can flow from one statement to the second because of the iteration of a loop, as in

```
            DO 100 I = 1, N
S1                  A(I) = ...
S2                  ... = A(I-1)
100         CONTINUE
```

Such a dependence is *loop carried*, since it will not exist if the loop is not iterated.

The fundamental difference between loop carried and loop independent dependences is easily understood in the context of the program transformations that preserve them [Alle 83]. Roughly speaking, a loop independent dependence is preserved so long as the original statement order is preserved. Thus, the loops surrounding a loop independent dependence may be permuted at will, without changing the semantics of the statements. A loop carried dependence, however, is strongly dependent upon the order in which the loops are iterated, but completely independent of statement order within the loops.

One final refinement is useful in the case of loop carried dependence. A loop carried dependence arises because of the iteration of one particular loop. For instance, in the following

```
            DO 200 I = 1, N
                DO 100 J = 1, N
S1                  A(I,J) = ...
S2                  ... = A(I-1,J)
100             CONTINUE
200         CONTINUE
```

iteration of the outer loop gives rise to the dependence. In this case, we say that the dependence is *carried* by the loop at nesting level one. So long as the outer loop is iterated sequentially, the dependence will be satisfied. We say that the *level* of the dependence is equal to the nesting level of the loop which gives rise to it.

Because of the importance of loop carried and loop independent dependences to automatic vectorization, there exists a number of fairly precise tests for detecting the presence of these dependences for array variables within loops [Wolf 82, Alle 83, Kenn 80, Bane 76].

4. Parallel Code Generation

4.1. Detecting Parallel Loops

Given the informal definitions of loop carried and loop independent dependences, and the requirements for parallel loops, it is not very hard to derive the following theorem:

> **Theorem 1.** The iterations of a loop can be run on separate processors without synchronization if and only if that loop carries no dependence.

Theorem 1 provides a very simple test for determining when the iterations of a loop may be run in parallel without synchronization—that is, it determines sequential loops which can be correctly run as DOALLs [Kuck 78]. It would be straightforward to develop an algorithm for parallelization based upon this test, but the resulting algorithm would probably not expose much parallelism in practice, because many loops carry dependences and would have to be run sequentially. As a result, a more sophisticated strategy is desirable.

As Theorem 1 points out, loop carried dependences imply a need for communication among processors. This communication, in turns, requires synchronization, which inhibits the type of parallel loops we desire. If this communication can be removed from within a loop body, parallelism can be restored. One transformation for removing such communication is *loop distribution* [Kuck 78]. For instance, in the following code:

```
      DO 100 I = 1, N
S1       C(I) = A(I) + B(I)
S2       D(I) = C(I-1) * B(I)
  100 CONTINUE
```

the I loop carries a dependence from S_1 to S_2, thereby prohibiting parallel execution of the loop. If the I loop is distributed around the two statements

```
      DO 100 I = 1, N
S1       C(I) = A(I) + B(I)
  100 CONTINUE
      DO 200 I = 1, N
S2       D(I) = C(I-1) * B(I)
  200 CONTINUE
```

then the loop carried dependence between the two statements is transformed into a loop independent dependence. As a result, each of the two loops can be executed in parallel, so long as all the processors synchronize at the completion of the first loop. While this code will not execute as fast as a single parallel loop would, it will still execute much faster than the original sequential loop.

From this discussion, it should be evident that a more viable approach to parallel code generation is to convert loop carried dependences into loop independent dependences by distributing loops around the dependences. Not all loop carried dependences can be "spread" across two loops. For instance, the following example

```
      DO 100 I = 1, N
S1       C(I) = A(I) + B(I)
S2       B(I+1) = C(I-1) * A(I)
  100 CONTINUE
```

contains loop carried dependences from S_1 to S_2 (due to C) and from S_2 to S_1 (due to B). Distribution of the loop around the two statements in this example changes the semantics of the code, because it causes the use of B in S_1 to get the values that were extant before the loop was entered, rather than the updated values computed by S_2. As a result, the loop cannot be correctly run in parallel without some form of synchronization.

In general, a loop can be distributed around the statements in its body so long as the bodies of individual *strongly connected* regions are kept together within a single loop copy [Kuck 78]. A strongly connected component of a directed graph is simply a maximal set of nodes and edges contained in a cycle; thus, the strongly connected component which contains a particular node includes all nodes and edges which occur on any path from that node to itself. Using an algorithm developed by Tarjan, a directed graph can be partitioned into strongly connected regions in time that is linear in the number of nodes and edges in the graph. Because strongly connected regions are the "basic blocks" of loop distribution, they are often called *piblocks* [Kuck 78].

From the above discussion, it should be apparent that the only loop carried edges which can be "broken" by loop distribution are those that cross from one strongly connected component to another. Loop carried edges which connect statements in the same piblock cannot be broken by distributing loops. Thus, the piblocks comprising a loop can be separated into two categories: *serial* piblocks (piblocks which contain an internal edge carried by the loop) and *parallel* piblocks (piblocks which do not have internal edges carried by the loop). While a straightforward algorithm can determine these properties and can schedule loops accordingly, such an algorithm would still not make the most effective use of machines close to our model. In general, the larger parallel regions are, the more likely the regions are to make effective use of parallelism on a large multiprocessor. As a result, fusing parallel piblocks together to enhance the granularity of parallel regions should always be profitable. Similarly, fusing serial regions together reduces loop overhead, and should also be beneficial when the serial blocks can not be executed concurrently.

The ideas presented so far in this section allow condition (O1) to be made precise by the following definition

> **Definition.** S is a **parallel statement** if it is enclosed in k loops and for some $i \leq k$, S is not part of a cycle in the dependence subgraph consisting of loop independent dependences and dependences carried at level i or deeper. Such a statement is said to be **parallel at level i**.

If statement S is parallel at level i, and if we execute the outer $i-1$ loops serially, then distribute the level i loop so that S is in a loop at level i by itself, that loop will not carry any dependences and hence can be parallelized. Under this definition, optimality condition (O1) is precise and the optimality of the algorithm is made dependent on the accuracy of the dependence analysis.

For the second problem — synchronizing shared memory accesses — the only accesses which must be synchronized are those between which there is a data dependence. In particular, it is not necessary to consider references related in the transitive closure of the dependence graph since the serialization process has a transitive affect. The second problem can be addressed based on where the endpoints of a data dependence are put after the program is broken into serial and parallel regions (i.e., DOALL's).

> **(P2)** *Both endpoints in the same serial or parallel region.*

Since only a single processor executes this region (or iteration of the loop), condition (P1) cannot be violated.

> **(P3)** *One endpoint in a serial region and the other endpoint in a parallel region.*

In this case, a barrier synchronization point will always be needed between these two regions.

> **(P4)** *Endpoints are in different parallel regions.*

Here a barrier synchronization point will only be needed if the two loops associated with these parallel regions are not combined.

An implication of (P2) is that all dependences contained in a single region can be ignored by the algorithm that orders regions and inserts barrier synchronization points.

4.2. Greedy Code Generator

This section presents an algorithm for solving the parallel code generation problem. Distinguishing parallel from serial regions is based on the dependence analysis discussed in the previous section. The primary task of the algorithm presented here is to find an ordering of these regions that minimizes the number of barrier synchronization points needed between them.

The algorithm is shown in Figure 1. The first step is to distinguish serial and parallel regions; from then on it is basically a greedy algorithm. Beginning with a region which has no incoming dependences, build a maximal set of serial and parallel regions such that barrier synchronization points are not needed between them. When a point is reached such that no additional region can be included, generate code for this set of regions, generate a barrier synchronization point and repeat.

As discussed above, a cycle of dependences defines a basic set of statements which must be executed as a unit. If no dependence in this cycle is carried at level k, then the entire strongly connected region is a parallel region, otherwise it is a serial region. Scalar statements (i.e., statements enclosed in fewer than k loops) are also serial[1]. The main loop is basically a topological sort, modified to delay generating a barrier synchronization point as long as possible.

The set *visited* is the set of strongly connected regions for which code has already been generated. The set *nopreds* is the set of strongly connected regions for which code has not been generated but all of whose immediate predecessors in D_P have been visited. Finally, the set *NotOk* is the set of strongly connected regions which cannot be fused with the current region without violating (P2), (P3) or (P4). Note that this set is reset to

[1] If the statement modifies only variables stored in the private memory associated with each processor, than the statement can be treated both as a serial and as a parallel region.

```
procedure Codegen (S,D,k);
{ S is a collection of statements and }
{ D is the level k dependence graph of S }
break S into strongly-connected regions $P = P_1, \cdots, P_k$;
let $D_P$ be the dependence graph induced on P by D;
{ preds(p) is the set of direct predecessors of p in $D_P$ }
nopreds ← { $p \in P \mid preds(p) = \emptyset$ };
visited ← $\emptyset$;
while ¬empty (nopreds) do begin
  R ← $\emptyset$;
  NotOK ← $\emptyset$;
  while nopreds−NotOK$\neq\emptyset$ do begin
    remove any node p from nopreds−NotOK;
    add p to visited and R;
    call CheckSons(p);
  end;
  { generate parallel code or drop down one level }
  for each connected region $r \in R$ do begin
    if the nodes in r are serial then begin
      let $D_r$ be the level k+1 dependence graph restrict-
        ed to r
      call Codegen(r,$D_r$,k+1);
    end;
    else fuse nodes in r into a single parallel loop;
  end;
  if ¬empty(nopreds)
    then generate a barrier synchronization point;
end;
```

Figure 1. Parallel code generation routine.

empty each time a barrier synchronization point is generated. The subroutine *CheckSons* marks successors of a strongly connected region as *NotOk* when they can not be fused with the node being examined.

The process of generating code includes fusing connected components so that rule (P2) applies and recursing on serial regions to look for parallelism at deeper nesting levels. Many details, such as the need to maintain the topological order as connected components are fused and correct handling of scalar statements, have been omitted for simplicity.

```
procedure CheckSons (v);
{ see if successors can be fused with v }
for each dependence e=(v,w) in $D_p$ do begin
  { check if v and w must be serialized }
  if both v and w are parallel
    then if v and w cannot be fused or e is loop carried
      at level k
      then add w to NotOK;
  else if either v or w is parallel
    then add w to NotOK;
  if preds(w)$\subset$visited
    then add w to nopreds;
end;
```

Figure 2. Check successors.

The second optimality constraint, that every parallel statement be in a loop which is parallelized, is achieved. If a statement S is parallel at level i, then either it is part of a parallel region at a level less than i, or *Codegen* will be called with a set of statements including S and level equal to i. In this case, S will be found to be a parallel region and a parallel loop will be generated around it.

The number of barrier synchronization points needed depends only on the order of the parallel and serial regions in the sequence generated. The ordering generated by the above algorithm is one which needs a minimal number of barrier synchronization points. To prove this, we first abstract away some of the details of the algorithms and examine a more general graph problem.

> **Definition**. Let $G=(V,E)$ be a directed acyclic graph and $\#$ a relation called **inconsistent** *defined on* $V \times V$. Thus, $v\#w$ is read; "v is inconsistent with w". A **consistent partition** *of G is a partition* $P=P_1, \ldots, P_n$ *of V such that the following two conditions hold for each* $v \in P_i$ *and* $w \in P_j$:
>
> *a. If* $<v,w> \in E$ *then* $i \leq j$
>
> *b. If* $v\#w$ *then* $i \neq j$

The mapping of the code generation problem to the problem of finding consistent partitions is clear: V is the set of strongly connected regions of the dependence graph restricted to loop independent edges and loop carried edges that are carried at levels deeper than k. E is the set of inter-region edges of this restricted dependence graph. Two nodes are inconsistent if they must be separated by a barrier synchronization point according to rules (P3) and (P4). A consistent partition is a partial ordering on the nodes in V that does not violate the partial ordering induced by E and that respects the requirements of (P3) and (P4).

Any consistent partition for the code generation problem will induce a valid ordering of the parallel and serial regions and clearly, the greedy algorithm above generates a consistent partition, from now on called the **greedy partition**. The minimal number of barrier synchronization points needed for a code generation problem is thus equal to one less than the number of elements in a consistent partition with the fewest number of elements. We will show that the greedy partition has the fewest number of elements and hence minimizes the number of barrier synchronization points at the outermost level.

Theorem 2. The greedy partition of G has a minimal number of elements.

Proof: Let $GP=G_1,...,G_n$ be the greedy partition of $G=(V,E)$ and let $P=P_1,...,P_m$ by any other consistent partition of G. We show that $n \leq m$ by induction on $|V|$. If $|V|=1$, then $n=1$ and the result is immediate since any partition must have at least one element. Assume that the greedy partition is minimal for all graphs with fewer than k nodes and that $|V|=k$.

An important property of the greedy algorithm is that $G_2,...,G_n$ is the greedy partition of G restricted to the nodes in $V-G_1$. This follows from the fact that the set *NotOK* is reset to empty each time a new element of the partition is started, thus the algorithm is effectively applied recursively to the reduced graph and so the partition generated for the reduced graph is also a greedy partition. If $P_1 \subset G_1$ then $P_2-G_1,...,P_m-G_1$ is a consistent partition of G restricted to $V-G_1$. By the induction hypothesis, we know that $G_2,...,G_n$ is minimal for that restricted graph and so $n-1 \leq m-1$ and the theorem follows.

Otherwise, select $w \in P_1-G_1$ such that $preds(w) \subset G_1$. Since w is not in G_1, at some point w was put into the set *NotOK*. This occurs only if there exists $v \in preds(w)$ such that $w \# v$. By property (a) of a consistent partition, v must be in P_1 since w is, but then we have a contradiction of property (b) which prohibits v and w being in the same element of P. Therefore $P_1 \subset G_1$ and the theorem follows.

A key factor in this proof is the fact $preds(w) \subset G_1$ and $w \notin G_1$ imply that w is inconsistent with at least one of its predecessors. When we add the ability to interchange loop levels, we lose this property and the algorithm is no longer optimal.

4.3. Modifications for the Multi-Loop Case

In the presence of nested loops, the simple greedy algorithm is non-optimal due to the fact that serial regions which have deeper parallelism need to be treated differently from serial regions without deeper parallelism. The greedy algorithm minimizes the number of partitions at the outer level but does not minimize the total number of barrier synchronization points expected.

To correctly handle the multi-loop case, each serial region must be examined for deeper parallelism *before* deciding which partition to put it in. Thus, the first step after breaking a loop into strongly connected regions is to recursively generate parallel code for each serial region that has statements at a deeper nesting level than k. If a serial region, r, has any parallelism deeper, then it is flagged as such: $deeper(r) \leftarrow$ **true** if r has parallelism at a deeper level; otherwise

```
procedure CheckSons (v);
{ see if successors can be fused with v }
for each dependence $e=(v,w)$ in $D_f$ do begin
  { check if v and w must be serialized }
  if both v and w are parallel
    then if v and w cannot be fused or e is loop carried
        at level k
      then add w to NotOK;
  else if either v or w is parallel
    then add w to NotOK;
  else if neither v nor w is scalar and one is deeper
    then add w to NotOK;
  if $preds(w) \subset visited$
    then add w to nopreds;
end;
```

Figure 3. Modification for Nested Loops.

deeper(*r*)←**false**. With the additional information in the *deeper* flags, we modify the procedure *CheckSons* to prevent fusing deeper regions with other regions (see Figure 3). Note that if a serial region has no statements at a deeper nesting level, then it can be fused with a region with deeper parallelism since the barrier synchronization between them is at the outermost nesting level regardless. A serial region with no statements at a deeper nesting level is said to be *scalar*.

Finally, the code to fuse connected regions must be modified. If two deeper regions are in the same partition, they must be completely independent, so we can merge them together so that they share barrier synchronization points. For example, the two loops

```
DO I = 1,N
    DOALL J = 1,N
        A(I,J) = A(I-1,J)+A(I,J)
    ENDDO
    IF (I.LT.N) BARRIER
ENDDO
DO I = 1,N
    DOALL J = 1,N
        B(I,J) = B(I-1,J)+B(I,J)
    ENDDO
    IF (I.LT.N) BARRIER
ENDDO
```

can be merged together into

```
DO I = 1,N
    DOALL J = 1,N
        A(I,J) = A(I-1,J)+A(I,J)
        B(I,J) = B(I-1,J)+B(I,J)
    ENDDO
    IF (I.LT.N) BARRIER
ENDDO
```

The algorithm to perform this merge is straightforward since the two regions are completely independent.

Henceforth, the greedy partition in the nested loop case will refer to the partition built by the greedy algorithm as modified in this section. The proof of optimality in this case can be found in Callahan's dissertation [Call 87].

5. Alignment and Replication

Whenever a value is created on one loop iteration (processor) and used on a different iteration (processor), as in

```
        DO 100 I = 1, N
S1          A(I) = B(I) + C(I)
S2          D(I) = 2*A(I-1)
        100 CONTINUE
```

codegen creates parallel loops by making the dependence loop independent using loop distribution. While the parallelism is enhanced by this transformation, there is still an undesirable synchronization point between the two loops. Note that the need for this synchronization point arises because each iteration of the loop creates a value fetched by another iteration of the loop. If the fetches can be "aligned" by one iteration, so that they occur on the same iteration (and therefore the same processor), as in

```
    DO 100 I = 0, N
        IF (I.GT.0) A(I) = B(I)+C(I)
        IF (I.LT.N) D(I+1) = 2*A(I)
100 CONTINUE
```

the need for synchronization will be eliminated, and the whole loop may be run in parallel. This transformation, called *loop alignment*, has been utilized in other contexts for multiprocessor machines [Padu 79]. The transformation eliminates the need for a synchronization point by moving the offending references to a single processor, where the synchronization will be provided naturally by the sequential execution on the individual processor. It would be nice if loop alignment were applicable in all cases. Unfortunately, it is not. Consider, for instance, the following example:

```
    DO 100 I = 1, N
        A(I) = B(I) + C(I)
        D(I) = A(I) + A(I-1)
100 CONTINUE
```

Once again, *codegen* would split this fragment into two parallel loops. In this case however, loop alignment cannot be directly applied because the second statement requires not only the value computed by the first on the previous iteration, but also the value computed on the present iteration. Aligning the statement for one use throws the other use out of alignment. One way to solve this problem is to compute both the necessary values on each processor by *replicating* the first statement, as in

```
    DO 100 I = 1, N
        A(I) = B(I) + C(I)
        A1(I) = B(I) + C(I)
        D(I) = A1(I) + A(I-1)
100 CONTINUE
```

In this replicated form, loop alignment can be applied to achieve a single parallel loop. Note that the array A1 is temporary to this loop and after alignment, each processor iteration accesses a different element. Hence A1 can be realized as a scalar variable kept in the local memory of each processor.

When can loop alignment and code replication be used to unify parallel regions? The answer is provided in Theorem 3.

Theorem 3. Loop alignment and code replication can be used to eliminate any loop carried true dependence that is not part of a recurrence consisting entirely of loop independent dependences and dependences carried by loops at the same or a deeper nesting level.

The proof of this theorem requires notation which is too detailed to be introduced here, and may be found elsewhere [Call 87]. The implications for the algorithm *codegen* are very positive. Since *codegen* is only concerned with fusing parallel regions that are not part of a recurrence, alignment and replication should always permit two parallel regions to be fused.

While Theorem 3 shows the utility of replication and alignment, it does not indicate the cost of employing those transformations. One unfortunate aspect of replication and alignment is that their effects can chain backwards; that is, a replication created for one alignment may introduce the need for another alignment farther back. Consider, for instance, the following example

```
        DO 100 I = 1, N
            C(I) = 2 * F(I)
            A(I) = B(I) + C(I)
            D(I) = A(I) + A(I-1)
    100 CONTINUE
```

Aligning as before on the dependence carried by A yields

```
        DO 100 I = 1,N+1
            IF (I.LE.N) C(I)=2*F(I)
            IF (2.LE.I) A(I-1) = B(I-1)+C(I-1)
            IF (I.LE.N) A1(I) = B(I)+C(I)
            IF (I.LE.N) D(I) = A1(I)+A(I-1)
    100 CONTINUE
```

While the alignment has eliminated the conflict with respect to A, it has introduced a new conflict with respect to C. As a result, the need for replication and alignment has simply moved farther back.

These observations lead us to see that there exist graphs for which code replication sufficient to align the loop produces an exponential increase in the number of statements in the loop. Furthermore, even for cases not requiring exponential growth, generating optimal output loops may still take exponential time as Theorem 3 shows.

Theorem 4. The problem of finding the minimum amount of code replication sufficient to align a loop is NP-hard in the size of the input loop.

The problem is not NP-complete in general. A tight bound on the size of the minimum sufficient amount of replication is given (in [Call 87]) by $|V|^2 \cdot \max$ {threshold(e) such that e is carried at level k} where V is the set of nodes in the dependence graph. Note that if the magnitude of thresholds of dependences carried at a given level is exponentially larger than $|V|$, then the minimum sufficient amount of replication can also be exponentially larger than $|V|$.

The proof of Theorem 4 utilizes an interesting reduction from 3-satisfiability [AhHU 74]. The following outlines the proof; more details may be found elsewhere [Call 87].

An important concept in the proof (and in replication) is the concept of ***sink nodes***, which are nodes in the dependence graph that have no successors. In the graph shown in Figure 4, if the same alignment value is chosen for both sink nodes (C and D), then node A need not be replicated but node B must be. On the other hand, if the alignment value for C is one more than for D, then node B need not be replicated but node A must be. Further,

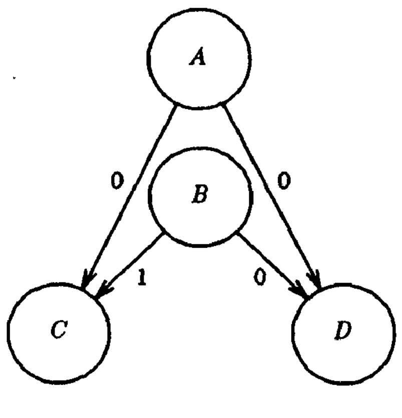

Figure 4. Choices During Replication.

if neither of these situations occurs, then both A and B must be replicated. The need to find relative alignments between sink nodes is the source of the combinatorial complexity for code replication.

Proof (outline): From an instance, B, of 3 CNF Satisfiability, a graph will be constructed. This graph will have $m+m(m-1)/2+7n$ nodes where n is the number of variables in the satisfiability problem and m is the number of clauses. The nodes in the constructed graph can be grouped into three categories. The first consists of all of the sink nodes; each sink node corresponds to one of the variables that appears in B. Assume these variables and nodes are labeled arbitrarily $v_1, \ldots, v_n$. The choices in the 3 CNF satisfiability problem are boolean values for the variables and the choices in the replication problem are alignment values for the sink nodes; the construction of the graph is designed to allow the following relationship between truth assignments that satisfy B and alignment value assignments corresponding to a minimum amount of replication

$$v_i = \begin{cases} \textbf{false} \textit{ if node } v_i \textit{ has alignment value} 3^i \\ \textbf{true} \textit{ if node } v_i \textit{ has alignment value } 2\cdot 3^i \end{cases}$$

The second set of nodes is used to enforce the requirement that each sink node v_i be assigned an alignment value of 3^i or $2\cdot3^i$. There is one node in this set for each unordered pair $<v_i,v_j>$ and the edges leaving that node are shown in Figure 6. This node must be replicated three times unless the each sink node is assigned an appropriate alignment value, in which case in needs to be replicated only twice.

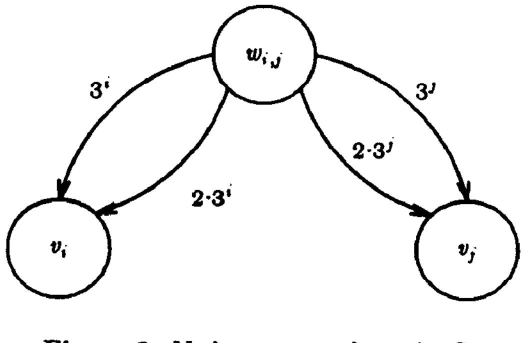

Figure 5. Nodes v_i, v_j and $w_{i,j}$ in G_B

The final set of nodes is used to encode the information in each clause of B. For each clause $F_i = l_1 + l_2 + l_3$ of B, where each l_i is a variable or its negation, there is a subgraph isomorphic to he one shown in Figure??. Each node $f_{i,xyz}$ corresponds to one of the seven combinations of truth values of the literals that satisfy the clause F_i. The thresholds on the edges are constructed so that a node must be replicated if a literal has a truth value different from the value needed for that node. For instance, one node will correspond to the combination where l_1 and l_3 are true but l_3 is false. This node will be replicated if any of these three conditions is not met. If a truth assignment satisfies this clause, then exactly one of the non-sink nodes shown in Figure ?? will not be replicated, otherwise they all will be replicated at least once. If the clause is satisfied, this subgraph will be replicated to only 19 nodes, otherwise more nodes will be needed.

It is shown in [Call 87] that minimum replication requires that alignment values be chosen for each sink node v_i from the set 3^i,$2\cdot3^i$ and that the graph resulting from a minimum replication has exactly $m+3m(m-1)/2+16n$ nodes if and only if B is satisfiable.

Replication interacts with the code generation process described earlier since the goal of replication is to reduce the number of barrier synchronization points. If replicating a particular node does not affect the number barrier synchronization points (i.e., each barrier synchronization point is required anyway) then there is no point in replicating the node. On the other hand, even if replicating a set of nodes reduces the number of barrier synchronization points, it may be that the increased execution time due to the replication is more than the expected cost of a barrier synchronization point.

Since finding the cost of replication is NP-hard and potentially exponential, replication in our implemented depends on heuristics and a short-look ahead to determine whether replication is profitable. This also represents a part of the algorithm that is directly parameterized by the target machine: the cost of barrier synchronization is directly compared to the cost of replication.

6. Loop Interchange

Loop interchange can be used to improve program performance by creating larger parallel regions. For

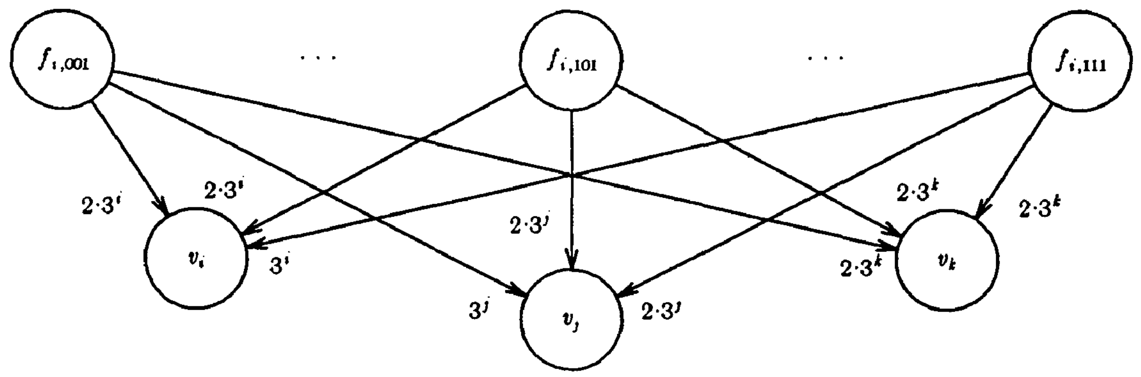

Figure 6. Nodes v_i, v_j and v_k and part of a factor node set in G_B r $F_i = v_i + \overline{v_j} + v_k$

example, in the following loop nest the I loop carries a recurrence and so must be executed serially, but the J loop can be parallelized. However, if we interchange the levels of the loops:

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

then the outer loop can be parallelized and the expected speedup will be greater since the amount of synchronization is reduced and the amount of useful work done between synchronization points is larger.

There may be more than one loop in a nest which could be parallelized when shifted to the outermost level. In the loop nest shown below, both the I loop and the J loop could be shifted to the outermost level and parallelized:

```
DO  I = 1 ,N
    DO  J = 1, N
        X(I,J) = X(I,J) + T*Y(I,J)
    ENDDO
ENDDO
```

The choice of which loop to shift to the outermost position and parallelize is affected by the context of the loop nest. Consider the case where only the J loop can be parallelized in the next loop nest (or strongly connected region in the same loop nest) in the source program, as in:

```
DO  I = 1 ,N
    DO  J = 1, N
        X(I,J)  = X(I,J)  + T*Y(I,J)
        U(I,J)  = U(I-1,J)  + X(I,J)
    ENDDO
ENDDO
```

Note that if we distribute the I loop and interchange loops only around the first strongly connected region, then a fusion preventing dependence is introduced. To avoid a barrier synchronization point between these loop nests, it is necessary (by rule P4) that the outer loops be fused together. To fuse these loops together, the J loop must be in the outermost position for both strongly connected regions.

To add loop interchange to the algorithm in section 4, a decision must be made regarding which level of each loop will be shifted to the outermost position and parallelized. It will be shown that an optimal decision is computationally intractable. The difference added by loop interchange to the consistent partition problem is that consistency of a member of a partition (which implies no barrier synchronization points are needed) is no longer implied by the pairwise consistency of the nodes in that partition. This leads to a generalized notion of consistency and to generalized consistent partition problem. The following definition corresponds the loop interchange problem restricted to a single loop nest in which all dependences that cross between piblocks are loop independent. The function σ corresponds to the

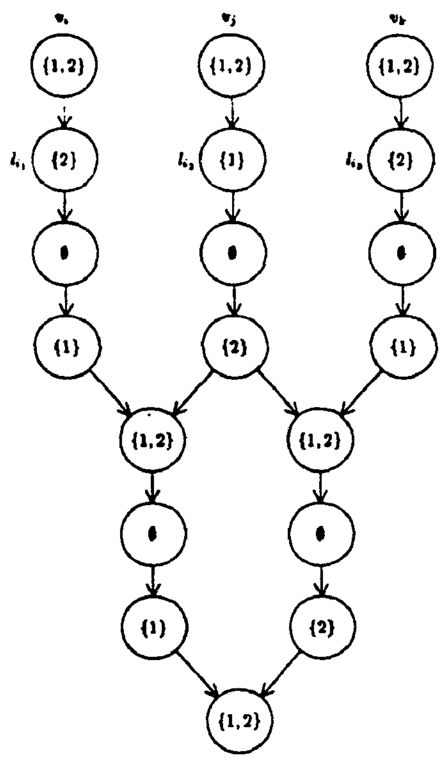

Figure 7.

mapping of strongly connected region into the set of levels which can be parallelized in the outermost position.

> **Definition. An instance of the Restricted Consistent Partition Decision Problem (RCPDP)** consists of a graph $G=(V,E)$, a function σ mapping V into $\{1..k\}$ for some k and a positive integer N. The question is: does there exist a partition, $P=P_1,...,P_N$, of V such that
>
> a. If $<v,w>\in E$, $v\in P_i$, and $w\in P_j$, then $i\leq j$
>
> b. For each connected component R of each member P_i, either $\sigma(v)=\emptyset$ for all $v\in R$ or there exists an integer j such that $j\in\sigma(v)$ for all $v\in R$.

Under this new definition, the problem of minimizing the number of barrier synchronization points will be shown to be NP-complete. This is not surprising since the question of whether a barrier synchronization point is needed between two particular regions can no longer be answered only with information about the two regions involved. The problem will be shown to be NP-hard by showing the above decision problem, based on a restriction of the general problem of code generation with loop interchange, is NP-hard. The significance of the restrictions on the input program is that two parallel regions can be fused if and only if they have the same loop at the outermost level.

> **Theorem 5.** The Restricted Consistent Partition Decision Problem is NP-Hard.

Proof: This theorem will be proved by demonstrating a polynomial reduction from the 3 Conjunctive Normal Form Satisfiability Problem (3SAT). An instance of the 3SAT consists of a boolean expression in conjunctive normal form:

$$B=\prod_{i=1}^{m}F_i$$

where

$$F_i=l_i^1+l_i^2+l_i^3$$

and each literal l_i^j is a variable in the set $\{v_1, \ldots, v_n\}$ or the negation of a variable in that set. The answer to a 3SAT problem is YES if there exists an assignment of logical values **true** and **false** to the variables that satisfies B. A graph, G_B, is constructed from an instance of 3SAT as follows: for each logical variable v_i, there is a node labeled v_i and none of these nodes have any incoming edges; for each clause of three literals, there is a sub-graph isomorphic to the one shown in Figure 7; each node is annotated with a subset of {1,2}. The nodes labeled v_i are each labeled with {1,2}, and every node below the second row is labeled as in Figure 7. The labels of the nodes in the second row depend on the literals in the clause: if l_i^1 is an unnegated instance of variable v_j, then the node corresponding to that literal is annotated with the set {2}. Otherwise, when the literal is a negated variable, the node is labeled with the set {1}. Figure 7 illustrates for the example clause $F_i=v_j+\bar{v}_k+v_l$.

The sets annotating the nodes define a function σ_B from the nodes of G_B into {1,2} and let $N=7$, and so an instance of RCPDP has been constructed from an instance of 3SAT in polynomial time. The next step is to establish that the 3SAT instance is satisfiable if and only if $<G_B,\sigma_B>$ has a consistent partition with seven members. The only real choices in selecting a consistent partition for the graph G_B and function σ_B are which 'loops' to select as the outermost loops for the nodes labeled v_i. The effect of these choices is whether or not any of the nodes in the second row of the subgraphs corresponding to the clauses (see Figure 8) can be part of the first member of the partition

If none of the nodes on the second row can be put in the first partition, then the minimal consistent partition for the subgraph containing that node will have eight members. This is illustrated in Figure 8(b). However, if any of the nodes on the second row can be put in the first partition, then the subgraph containing that node will have a consistent seven partition, as illustrated in Figure 8(a) (the other two case are similar).

A node l on the second row can be put in the first partition if and only if the loop chosen for the variable node that is its immediate ancestor is the same as the loop in the singleton set labeling l. If different loops are selected for a literal node and its ancestor, the two parallel regions they represent can not be fused and so rule P4 will prevent the nodes from being in the same

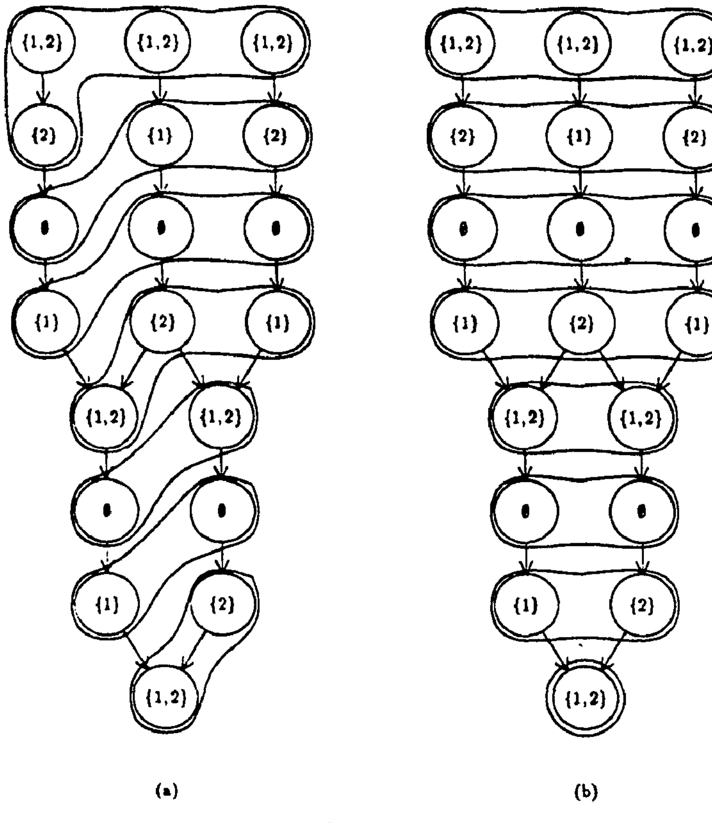

Figure 8.

partition.

Let $T:\{v_1, \ldots, v_n\}$ be a truth assignment that satisfies B. This truth assignment is a guide to selecting outermost loops for the nodes in G_B labeled with variables: if T assigns true to v_i, then select loop 2 for the node v_i, otherwise, select loop 1. Let F_i be a clause of B and G_{F_i} be the subgraph corresponding to F_i. F_i is satisfied by T and so some literal l_i^j is satisfied. If l_i^j is the unnegated variable v_k, then T assigns true to v_k and so loop 2 is selected for node v_k. Loop 2 is the only choice for the node corresponding to l_i^j and hence the node corresponding to l_i^j can be consistently put in the first partition and hence the subgraph G_{F_i} has a consistent seven partition. On the other hand, if l_i^j is the negation of variable v_k, then T assigns false to v_k and so loop 1 is selected for node v_k. Loop 1 is the only choice for the node corresponding to l_i^j and hence the node corresponding to l_i^j can be consistently put in the first partition and again the subgraph G_{F_i} has a consistent seven partition. Since each clause is satisfied, the entire graph as a consistent seven partition.

Assume that G_B has a consistent seven partition, then in each clause subgraph, G_{F_i}, at least one node, l_i^j on the second row is contained in the first partition. Define a truth assignment as follows: if the node l_i^j corresponds to a literal consisting of the unnegated variable v_k, then assign true to v_k, otherwise, l_i^j corresponds to a literal consisting of the negated variable v_k and so assign the value false to v_k. If any variable is not assigned a value by this rule, assign that variable true. This assignment is well defined, otherwise, for some variable v_k, nodes corresponding to literals consisting of both negated and unnegated instance of v_k occur in the first partition and hence the first partition would not be consistent. To show that this truth assignment satisfies B, note that for each clause F_i, there is a literal l_i^j that is true under the above truth assignment and hence F_i is true. Since each clause is satisfied, B is satisfied.

The last two paragraphs have established that B is satisfiable if and only if G_B with σ_B has a consistent seven partition.

Theorem 6. The Code Generation Problem with Loop Interchange is NP-Complete.

Proof: The previous theorem shows the general problem has a subproblem which is NP-hard, this theorem then follows if we can show that the general problem is in NP.

This is straightforward: for each regions that has more than one choice for outermost parallel loop, non-deterministically select one. Based on this choice for outermost loop, apply the algorithm of the Section 3 to obtain a minimal consistent partition in linear time.

7. Conclusions

This paper discusses algorithms for detecting and enhancing parallelism in a sequential program. These algorithms are based on the concept of loop carried dependence and should be quite effective at detecting implicit parallelism while remaining reasonably efficient for most programs.

We have implemented these methods in a parallel code generation system derived from PFC, a vectorizer written at Rice [AllK 84]. The new system analyzes FORTRAN programs, employing all the transformations described in this abstract, and generates parallel Fortran code, similar to that used in the examples, for execution on the IBM RP3 [GGKM 83]. The system also incorporates a complete interprocedural analysis of flow insensitive side effects and performs interprocedural constant propagation [CCKT 86].

References

[AhHU 74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. 1974.

[Alle 83] J.R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, Houston, TX, April 1983.

[AllK 84] J.R. Allen and K. Kennedy, "PFC: a program to convert Fortran to parallel form," *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, MD, 1984, 186-203.

[AllK 85] J.R. Allen and K. Kennedy, "A parallel programming environment," *IEEE Software 2(4)*, July 1985, 21-29.

[Bane 76] U. Banerjee, "Data dependence in ordinary programs," Report 76-837, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, November 1976.

[Bern 66] A.J. Bernstein, "Analysis of programs for parallel processing," *IEEE Trans. Electronic Computers 15(5)*, October 1966.

[Call 87] D.Callahan "A Global Approach to Detection of Parallelism," Ph.D. Dissertation, Dept. of Computer Science, Rice University, Houston, TX, January 1987.

[CCKT 86] D.Callahan, K.Cooper, K Kennedy, and L.Torczan, "Interprocedural Constant Propagation" *Proceedings SIGPLAN '86 Symp. on Compiler Construction,* SIGPLAN Notices V21,No. 7, July 1986

[Cytr 82] R. Cytron, "Compile-time scheduling and optimization for asynchronous machines," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, August 1982.

[FlaK 85] H.P. Flatt and K. Kennedy, "Performance of Parallel Processors," Rice COMP TR85-22, Department of Computer Science, Rice University, Houston, Texas, June 1985.

[GGKM 83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU ultracomputer -- designing an MIMD shared memory parallel computer," *IEEE Trans. on Computers c-32(2)*, February 1983, 175-189.

[GKLS 84] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "Cedar," *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, MD, 1984, 251-275.

[Kenn 80] K. Kennedy, "Automatic translation of Fortran programs to vector form," Rice Technical Report 476-029-4, Rice University, October 1980.

[Kuck 78] D.J. Kuck, *The Structure of Computers and Computations* Volume 1, John Wiley and Sons, New York, 1978.

[Padu 79] D.A. Padua, "Multiprocessors: discussion of some theoretical and practical problems," TR UIUCDCS-R-79-90, University of Illinois at Urbana-Champaign, Urbana, Ill., November 1979.

[PBGH 85] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J.Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton and J. Weiss, "The IBM research parallel processor prototype (RP3): introduction and architecture," RC 11060, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 1985.

[Seit 85] C.L. Seitz, "The Cosmic Cube," *Comm. ACM 28(1)*, January 1985, pp. 22-33.

[Wolf 82] M.J. Wolfe, "Optimizing supercompilers for supercomputers," TR UIUCDCS-R-82-1105, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1982.