A Graph Based Approach to Barrier Synchronisation Minimisation

E.A. Stöhr
Department of Computer Science
The University
Manchester M13 9PL, UK

M.F.P. O'Boyle
Department of Computer Science
The University of Edinburgh
Edinburgh EH9 3JZ, UK

Abstract

This paper presents a new graph theoretic approach to minimising the number of barriers in parallelised programs. A simple procedure to reduce the complexity of barrier placement, without affecting optimality, is developed. A new algorithm is then presented which places provably the minimal number of barriers in perfect loop nests. This technique is extended so as to place the minimal number of barriers in certain imperfect loop nest structures. This scheme is generalised to accept entire programs and implemented in a prototype parallelising compiler where it has been applied to several well-known benchmarks and shown to place significantly fewer synchronisation points than an existing commercial compiler.

1 Introduction

Many widely-used parallel machines, such as the SGI challenge, Convex Exemplar and Sequent NumaQ, provide a shared address space. Such architectures are attractive as they are easier to program than message passing systems, speeding the implementation and porting of parallel programs. This ease of programming is, however, gained at a cost. As message-passing programs explicitly encode all data transfer and synchronisation between processors, the user is able to minimise these overheads, albeit at the cost of additional effort. Shared address space architectures, in contrast, use a read/write model with synchronisation inserted at appropriate points to honour data dependences and maintain program semantics. Barriers are a widely used form of synchronisation but are typically more expensive than a read or write access and it is important for good performance that redundant synchronisation is avoided. Synchronisation also increases idle time, due to load imbalance, and can sequentialise sections of a program. Thus, even on architectures with fast synchronisation mechanisms, such as the Cray T3D, synchronisation reduction is essential.

However, minimising synchronisation overhead increases the complexity of the programming task and reduces the attractiveness of shared memory programming. This paper offers another approach. It describes a compiler-based algorithm that automatically places synchronisation efficiently. In the most commonly occurring cases, it places provably the minimal number of barriers and in all other cases outperforms existing techniques. Thus, the programmer is relieved from the burden of reasoning about synchronisation placement, without incurring performance degradation, making single - address space parallel architectures more attractive.

Our previous work [7] incorporated ideas from distributed-memory computing and used a SPMD execution model. Using a cross-processor dependence, rather than program structure approach, it was possible to significantly reduce the number of synchronisations by utilising data and work layout information. This reduction of synchronisation was, however, largely local in nature. In particular, it was relatively naive in determining whether a barrier could satisfy several cross processor dependences and in determining the minimal number of barriers for a particular program. Furthermore, there was no examination of the optimality, nor an investigation into the actual complexity of the approach. This paper develops a framework for globally determining the smallest number of synchronisations for a given program. It is based upon a graph formulation of the problem and exploits control and data dependence structure within a program. We develop and implement a compiler algorithm that attempts to insert the smallest number of synchronisations and we show when it is optimal.

This paper makes the following contributions:

- A technique that safely eliminates many dependences, reducing the complexity of the algorithm for barrier placement, without affecting optimality.
- An algorithm that provably determines the minimal number of barriers in perfect loop nests.
- An algorithm that provably determines the minimal number of barriers in certain imperfect loop nests.
- An algorithm that places fewer barriers in the entire program than existing methods.

The remainder of the paper is structured as follows: Section 2 describes our approach and presents an illustrative example. Section 3 describes the graph framework within which the algorithms are developed. Sections 4 and 5 present powerful algorithms for barrier placement. In section 6 a review of existing work is presented and in section 7 the performance of our compiler implementation is evaluated. Section 8 concludes the paper.

2 Approach

This section summarises our approach and presents an illustrative example.

1. **Repeat**
   (a) For each loop L
      i. Perform dependence elimination
      ii. **If** perfect(L) **Then**
         - **If** $\exists$ forward dependence
            – Apply FFS
            – **If** remaining dependences **Then** apply maximum level
         - **Else** Place barriers after $do(p)$, where p is the maximal level of dependences
      iii. **Elseif** imperfect(L) **Then**
         - **If** all dependences are forward
            – **If** $\exists$ short dependence at level m **Then** apply FFS and reverse FFS
            – **elseif** $\exists$ other short dependences **Then** kill a long dependence; apply FFS and reverse FFS
            – **elseif** $\exists$ only long dependences **Then** place a barrier before $do(k)$ and the smallest sink of remaining dependences
         - **Else** Apply backward dependence algorithm
   (b) Remove killed dependences and consider each L as a statement
2. **Until** no further loop nests
3. For a basic block
   (a) Perform dependence elimination
   (b) Apply FFS

Figure 1: Algorithm

2.1 Algorithm Summary

Once cross-processor dependences [8] are determined, we wish to place the smallest number of barriers to kill them. Determining the fewest number of barriers to kill all dependences is an expensive operation. The first step in our algorithm, shown in figure 1, is therefore to prune the data dependence graph of non-*critical* data dependences. A loop-specific algorithm is then applied which exploits the specific control-structure of nested loops. It categorises data dependences by their direction and the loop nest level at which they can be satisfied. For perfect loop nests, it places the minimal number of barriers. In addition, for those imperfectly nested loops where a loop contains several statements and one further loop nest (which may also contain statments and a further loop nest etc.), it also places the minimal number of barriers. When there are two or more enclosed loop nests, they are initially treated separately before being considered as simple vertices in their enclosing loop's dependence graph. This process is applied recursively until the entire program is considered. If there is any remaining straight line code an algorithm, First Fast Sink (FFS)[12], which places the minimal number of barriers, is used.

Our approach relies on structured control-flow and therefore requires control-flow normalisation as a pre-processing phase. Functions and subroutines do not present a particular problem as they can be considered, for the purposes of this optimisation, simply as statements which may be the source and sink of several dependences. The algorithms developed in this paper exploit basic block and loop structure but do not make special provision for IFs. In fact, optimisation of general nested IF constructs considerably complicates the presentation of our scheme and is not considered here. For further details on optimising synchronisation in the presence of IF constructs see [13]. For the purposes of this paper, if both paths of an IF construct are considered to be taken, the algorithm may be safely applied.

2.2 Example

To illustrate the compiler approach developed in this paper, consider the program fragment, extracted from the shallow benchmark program, shown in figure 2, column 1. By applying the scheme developed in this paper, we eventually derive the node program shown in the final column, where lo and hi are the local loop bounds found after parallelisation and partitioning.

Initially, there are 12 dependences to consider. For array P, there are the following four dependences : s1 $\longrightarrow$ s4, s1 $\longrightarrow$ s5, s5 $\longrightarrow$ s1, s4 $\longrightarrow$ s1. For array U, there are two dependences: s2 $\longrightarrow$ s4, s4 $\longrightarrow$ s2, For array V, there are again two dependences s3 $\longrightarrow$ s5, s5 $\longrightarrow$ s3. CAPU and CAPV also have two dependences each namely: s4 $\longrightarrow$ s6, s6 $\longrightarrow$ s4 and s5 $\longrightarrow$ s6, s6 $\longrightarrow$ s5.

All of these dependences are shown as arcs in column 2, where a box represents a `do` or `enddo` control node and a circle represents a simple vertex in the dependence graph corresponding to a program statment.

If we partition by columns then it can be shown that four of the dependences are local, giving the cross-processor dependence graph shown in column 3. As can be seen from the diagram, it is far from trivial to determine the minimal number of barriers even for a small code fragment. On examining the loop structure, there is more than one loop nest inside the outer loop and thus each perfect loop nest $(L1, L2, L3, L4)$ is considered individually. The next step is to consider each loop nest as a vertex in the graph. By applying the elimination algorithm given in section 4, we can show that only dependences L1 $\longrightarrow$ L2, L2 $\longrightarrow$ L4 and L3 $\longrightarrow$ L4 are critical and that all other dependences can be ignored.

Now applying the perfect loop algorithm, described in section 4, it can be shown that barriers placed just before L2 and L4 will kill the remaining dependences, as shown in column 5. Most existing schemes would either place a fork and join around each parallelised loop or a barrier after each perfect loop nest giving a total of 4 barriers. Our previous scheme [7] would place a barrier after the first DO header, the first perfect loop nest and after the second perfect loop nest giving 3 barriers. The scheme described in this paper places just 2 barriers which is the minimal number for this program. Although this is a simple example, it illustrates *some* of the basic points of the approach described in this paper.

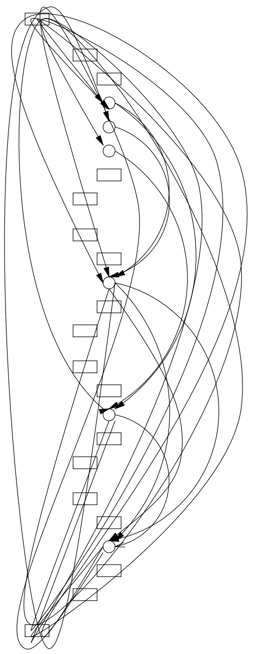
3 Graph representation

The control flow graph, augmented with cross-processor data dependences, is the main program representation used in this paper. The *control/dependence graph* is a directed graph $CDG = (S, C, E)$, where the set of nodes S corresponds to the set of statements of a program, the set of arcs C corresponds to the set of control relations and the set of arcs E cor-

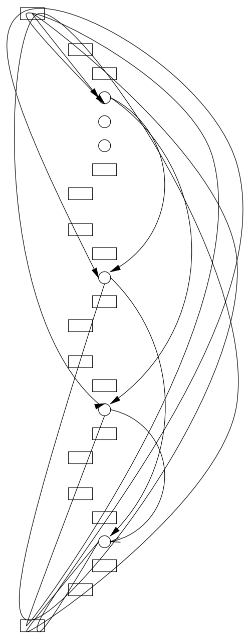
Original Program

```
DO  STEP = 2,MAXTIMESTEP
L1:  DO  I=1,N
       DO  J=1,N
1:       P(I,J) = PNEW(I,J)
2:       U(I,J) = UNEW(I,J)
3:       V(I,J) = VNEW(I,J)
       ENDDO
     ENDDO
.
.
L2:  DO  I=1,N-1
       DO  J=1,N
4:      CAPU(I+1,J) = 0.5 *
         (P(I+1,J) +P(I,J)) *
          U(I+1,J)
       ENDDO
     ENDDO
.
.
L3:  DO  I=1,N
       DO  J=1,N-1
5:       CAPV(I,J+1) = 0.5 *
          (P(I,J+1)+P(I,J))
           * V(I,J+1)
       ENDDO
     ENDDO
.
.
L4:  DO  I=1,N-1
       DO  J=1,N-1
6:       PNEW(I,J) = PSOLD(I,J)
          - DD  * (CAPU(I+1,J)
          - CAPU(I,J)) - DD *
         (CAPV(I,J+1) - CAPV(I,J))
       ENDDO
     ENDDO
.
ENDDO
```

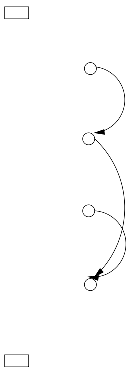
Original Dependences



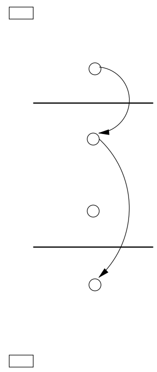
Cross-Processor



After Elimination



Loop Alg. and Barriers



Node Program

```
DO  STEP = 2,MAXTIMESTEP
 DO  I=1,N
  DO  J=lo,hi
   P(I,J) = PNEW(I,J)
   U(I,J) = UNEW(I,J)
   V(I,J) = VNEW(I,J)
  ENDDO
 ENDDO
call barrier()
.
 DO  I=1,N-1
  DO  J=lo,hi
   CAPU(I+1,J) = 0.5 * (P(I+1,J)
     +P(I,J)) * U(I+1,J)
  ENDDO
 ENDDO
.
.
 DO  I=1,N
  DO  J=lo,min(hi,n-1)
   CAPV(I,J+1) = 0.5 * (P(I,J+1)
     +P(I,J)) * V(I,J+1)
  ENDDO
 ENDDO
call barrier()
.
 DO  I=1,N-1
  DO  J=lo,min(hi,n-1)
   PNEW(I,J) = PSOLD(I,J) - DD
    * (CAPU(I+1,J) - CAPU(I,J))
    - DD * (CAPV(I,J+1)
    - CAPV(I,J))
  ENDDO
 ENDDO
.
ENDDO
```

Figure 2: Synchronisation example

responds to the cross-processor data dependences between two statements.

Let $L = L_1, L_2, \ldots, L_m$, $m \geq 1$, be a loop nest where L_k, $1 \leq k \leq m$, is the loop at *level* k. A loop L_k has a body containing statements and/or further loop nests. Let G be a graph of the loop L whose vertices either correspond to program statements or the entry and exit control points of a loop denoted by $do(k)$ and $enddo(k)$. Dependences with a source and sink within the same iteration are called *forward* dependences. The dependence vector of a forward dependence is $\overrightarrow{0}$ and the source of a forward dependence is always less than the sink. A loop-carried dependence, with a source j and a sink i, is called either a *backward* dependence, if $i < j$, or a *wrap-around* dependence, if $i \geq j$. Loop-carried dependences have non-zero dependence vectors and the *level* of a dependence is defined as the position of the leftmost nonzero component of the dependence vector.

Property 1: A backward dependence $i \longrightarrow b_i^k$ at level k may be killed by any barrier inserted between i and $enddo(k)$ or between $do(k)$ and b_i^k. A wrap-around dependence at level k may be killed by any barrier inserted between $do(k)$ and $enddo(k)$.

The data dependences of a loop can be expressed using ordered incidence lists. We divide all dependences into forward, backward and wrap-around dependences, at each level, and place them in separate incidence lists so that a vertex v is on the incidence list for vertex u, if there is a corresponding dependence from u to v. There are $2m+1$ incidence lists: one for forward dependences and $2m$ for backward/wrap-around dependences at each level. The algorithms described in the following two sections use the incidence lists to determine barrier placement.

4 Perfect Loop Nests

This section describes the elimination of dependences and barrier placement for perfect loop nests.

4.1 Elimination

We wish to reduce the number of dependences and hence the complexity of the algorithm for barrier placement within loop nests. If there is at least one forward dependence, then any barrier inserted to kill this dependence will kill all wrap-around dependences at the appropriate level. In this subsection, we suppose that there is at least one forward dependence and therefore need only consider the incidence lists for forward and backward dependences. The absence of forward dependences is handled as a special case later (see section 4.3.)

First, we perform dependence elimination as described in [12]: for every source vertex i, remove all dependences except for the dependence with the smallest sink. Let $i \longrightarrow b_i$ be a backward dependence at loop level k and let $j \longrightarrow a_j$ be a forward dependence such that $j \geq do(k)$ and $a_j \leq b_i$. Since any barrier which kills $j \longrightarrow a_j$ will kill $i \longrightarrow b_i$, we can remove $i \longrightarrow b_i$ from the incidence list of backward dependences at level k. (See column 1 of figure 3).

The procedure for reducing the number of dependences is as follows: for each k, $k = 1, \ldots, m$, choose the smallest number, say a_j, from the list of sinks, $a_1, \ldots, a_{n-1}$ such that $do(k) < j, a_j < enddo(k)$. Remove all level ℓ dependences $i \longrightarrow b_i$ such that $b_i \geq a_j, \ell \leq k$. From the list of forward dependences choose the biggest j, $do(k) < j < enddo(k)$ such that there is an arc $j \longrightarrow a_j$ with $do(k) < a_j < enddo(k)$. Remove all level ℓ dependences $i \longrightarrow b_i$ such that $i \leq j$, $l \leq k$.

If there are any backward dependences $i \longrightarrow b_i^k$ and $j \longrightarrow b_j^r$, where $k \leq r$, $i \leq j$ and $b_j^r \leq b_i^k$, then any barrier inserted to kill $j \longrightarrow b_j^r$ will kill $i \longrightarrow b_i^k$ (see column 2 of figure 3). Thus, we can remove $i \longrightarrow b_i^k$ from the list of backward dependences at level k. The elimination of dependences can be performed in $O(m \cdot n)$ steps. Performing dependence elimination gives rise to the following useful property.

Property 2: After performing the elimination of non-critical dependences, all the sinks of backward dependences are smaller than the smallest sink of forward dependences and all the sources of backward dependences are greater than the largest source of the forward dependences at corresponding loop levels.

4.2 Barrier placement in perfect loop nests

This section describes the optimal insertion of barriers in perfect loop nests on a reduced dependence graph. A loop nest is *perfect* if, for each $1 \leq j < m-1$, the body of L_j is L_{j+1}. Firstly, we briefly describe the FFS (*Fast First Sink*) algorithm [12], which places the minimal number of barriers in basic blocks with n statements in at most n steps. This algorithm is used as the basis for barrier placement in loops. After elimination has taken place, the algorithm works as follows: starting at vertex 1, find the first sink, t_1, after vertex 1. Iteratively, choose t_i as the first sink after t_{i-1} of all the dependences with sources greater than or equal to t_{i-1}. Let $q_i \longrightarrow t_i$, $q_i \geq t_{i-1}$, $\forall i$ be the set of *essential arcs* determined by this algorithm. If we insert barriers just before $t_1, t_2, \ldots, t_\ell$, all the dependences within a basic block will be killed.

The algorithm, for perfect loop nests, works as follows: from all the forward dependences find the essential arcs and initially insert a temporary barrier before each sink t_i. If all the dependences are killed, then we have found the minimal number of barriers and need not proceed further. If, however, there are remaining backward dependences which were not killed, further work is necessary.

Maximal Level Let $q_1 < x_1 < x_2 < \ldots < x_s < t_1$ be all the nodes such that x_i is a sink of some backward dependence. (see Property 2.) Let k_0 be the deepest loop nest level of all the remaining backward dependences; k_0 is placed on the *maximum level list.* The following procedure is repeated for all x_i, $i = s, s-1, \ldots, 1$. We insert a temporary barrier before x_i, remove all dependences killed, find the essential arcs from the remaining set of forward dependences and place temporary barriers before the sinks of the essential arcs. If all the dependences are killed, then we have found the optimal number of barriers. Otherwise, find the largest level k_i of all remaining backward dependences and place k_i on the maximum level list. Now, if in all cases some dependences were not killed by ℓ barriers, then from $k_0, k_1, \ldots, k_s$ choose the minimum level, say k_{min}. Place barriers before statement x_{min}, the sinks of the essential dependences and after $do(k_{min})$. This algorithm determines the minimal number of times barriers for a perfect loop nest will be executed.

Example To illustrate this approach, consider the example in figure 3, column 3. There is only one forward dependence, $q_1 \longrightarrow t_1, (b \longrightarrow d)$ and only one node $x_1 = c$ between $b \longrightarrow d$ as the sink of a backward dependence. Placing a

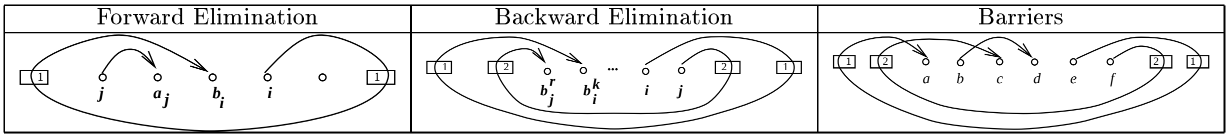


Figure 3: Elimination and Barrier Placement in Perfect Loop Nests

barrier before t_1, does not eliminate all the dependences; the deepest loop level of the remaining dependences is 2 ($f \longrightarrow c$), hence $k_0 = 2$. Placing a barrier before x_1, kills $f \longrightarrow c$ and $b \longrightarrow d$, but again does not eliminate all dependences. However, the deepest loop level of the remaining dependences is now 1 ($e \longrightarrow a$), hence $k_1 = 1$. As k_1 is smaller than k_0, ($k_{min} = k_1$), we place a barrier before $x_1 = c$ and after $do(1)$. Suppose that each of the loops is executed N times then the 2 barriers inserted will be executed N^2+N times. All other placements are more expensive and hence our selection is optimal.

4.3 No Forward Dependences

If there are no forward dependences, then a barrier is placed after $do(p)$, where p is the maximal level of backward and wrap-around dependences. This is also the optimal placement of barrier synchronisation.

The time complexity is $O(n \cdot (d+1))$, where d is the number of backward dependences after applying elimination.

Theorem 1 *The algorithm places the minimal number of barriers in polynomial time for perfect loop nests.*

Proofs of theorems are omitted due to space constraints; for further details see [14]

5 Barrier placement in imperfect loop nests

This section considers certain imperfect loop nest sequences: loops whose body may contain a further loop nest and several statements. To aid presentation, we will divide the algorithm for these nested loop structures into two cases. Those containing just forward dependences and those containing forward and/or backward dependences.

5.1 Forward Dependences

A dependence, $u \longrightarrow v$, with a source before $do(m)$ and a sink after $enddo(m)$ is called a *long dependence*. All other dependences are called *short dependences*. Initially, we consider a case when at least one short dependence has a source and sink between $do(m)$ and $enddo(m)$. Next, the case when no short dependence has a source and sink between $do(m)$ and $enddo(m)$ is addressed. Finally, the third case of only long dependences is considered.

Initially, we replace all wrap-around dependences at level k, $k = 1, \ldots, m$, with an arc $do(k) \longrightarrow enddo(k)$ (see Property 1) and consider this arc as a forward dependence. Since barrier placement in nested loop sequences is more complicated than for perfect loops nests, in addition to the elimination described in section 4.1, we will further reduce the number of dependences to consider by removing covered dependences using the following property:

Property 3: For every arc, $u \longrightarrow v$, if there is an arc, $u' \longrightarrow v'$, such that $u \le u'$ and $v \ge v'$ then $u \longrightarrow v$ is covered and can be removed from the incidence list.

A new graph is introduced to aid the placement of barriers in the least expensive positions, i.e. in those locations where the number of times they are executed at runtime is minimal. Let H be a graph and let H' be a graph obtained from H by replacing every arc $u \longrightarrow v \in H$ with the source u and the sink v by a reverse arc $v \longrightarrow u \in H'$ with the source v and the sink u. The *Reverse Fast First Sink Algorithm* is defined as being applied to the graph H if the Fast First Sink Algorithm is applied to the graph H'. This reverse formulation allows the use of the existing FFS algorithm in determining the least costly positions for barriers in a loop nest.

5.1.1 Short Dependences inside level m.

In this case there is at least one short dependence with both a source and a sink between $do(m)$ and $enddo(m)$. Let $u \longrightarrow v$ be a dependence with the smallest sink of such short dependences. All dependences with sources less than u and sinks greater than v have already been removed from the incidence list during elimination and need not be considered further. Let $x_1, \ldots, x_s$ be the sinks of dependences with sources less than $do(m)$ such that $u < x_1 < x_2 < \ldots < x_s < x_{s+1} = v$. For every x_j, $j = 1, \ldots, s+1$, temporarily place a barrier before x_j, remove any killed dependences and then apply the FFS algorithm to the remaining arcs to the right of x_j and the reverse FFS algorithm to the remaining arcs to the left of x_j. The position with the expected minimum number of executions is chosen.

The FFS algorithm and reverse FFS are applied in this manner in order to place a barrier at the lowest loop level possible to kill a dependence. Obviously, a barrier placed deep inside a loop nest will be executed more frequently than one with less enclosing loops.

Example To illustrate this technique, consider the example in figure 4. There is one short dependence $b \longrightarrow d$ within $do(2), enddo(2)$. There is only one sink c between $b \longrightarrow d$, hence $x_1 = c$, $x_2 = d$. Placing a barrier before $x_1 = c$ kills dependence $b \longrightarrow d$ and $a \longrightarrow c$. Applying the FFS to the dependences to the right, gives a barrier before f. This gives $N^2 + N$ barriers at runtime.

Placing a barrier before $x_2 = d$, however, kills $b \longrightarrow d$, $c \longrightarrow f$. Applying reverse FFS to the remaining dependences to the left, places a barrier after a, applying FFS to the dependences to the right gives a barrier before g, giving N^2+2N barriers at runtime. Our algorithm selects the first case.

5.1.2 Other Short Dependences

In this case there is no short dependence with both a source and a sink between $do(m)$ and $enddo(m)$, but there are other short dependences. Let R be the set of long dependences after dependence elimination. Let $h \longrightarrow g \in R$. Since we have performed elimination, there are no dependences with both a source and a sink between h and g (Property 3). The algorithm works as follows: for every dependence

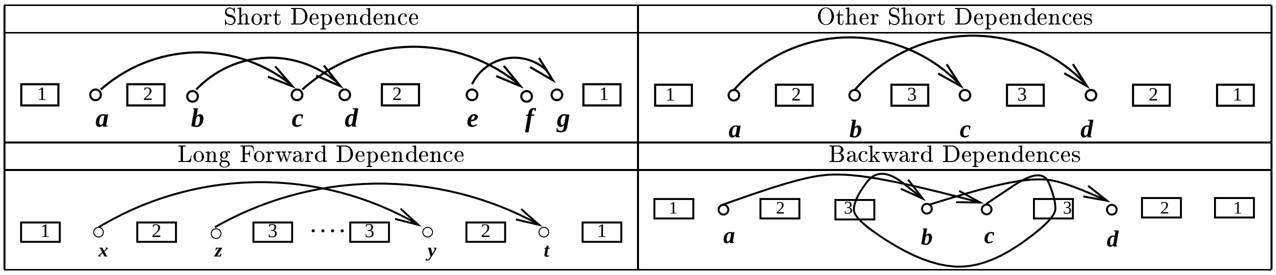


Figure 4: Barrier Placement in Imperfect Loop Nests

$h \longrightarrow g \in R$ place a temporary barrier just after h, remove any killed dependences and then apply the FFS algorithm to the remaining arcs to the right of h and the reverse FFS algorithm to the remaining arcs to the left of h. Estimate the number of times the barriers will be executed. Then place a temporary barrier just before g, remove any killed dependences and then apply the FFS algorithm to the remaining arcs to the right of h and the reverse FFS algorithm to the remaining arcs to the left of h. Once again estimate the number of times the barriers will be executed. From these estimates for every long dependence choose the minimum.

Example To illustrate this technique, consider the example in figure 4. In this example there is one long dependence $b \longrightarrow d$ and one short dependence $a \longrightarrow c$. Initially, we place a temporary barrier just after the source of the long dependence b. This barrier kills all dependences and will be executed N^2 number of times. Continuing the algorithm, we place a temporary barrier before d. Then, on applying the reverse FFS, we shall place a barrier after a to kill the dependence $a \longrightarrow c$. In this case the barriers will be executed $N^2 + N$ number of times and therefore the placement of a barrier after b is selected.

If there were no long dependences, then the FFS algorithm is simply applied to the arcs to the right of $enddo(m)$ and the reverse FFS algorithm to the arcs to the left of $do(m)$.

5.1.3 Long Dependences

The final case to consider is where there are no short dependences. In this case for every $k = 2, \ldots, m$ place a temporary barrier before $do(k)$, remove any killed dependences and then place a barrier before the smallest sink of the remaining dependences. Choose the position in which the barriers will be executed the minimum number of times.

Example In figure 4 the minimum number of execution in achieved when one barrier is placed before $do(2)$ and another before t.

The time complexity for the placement of barriers in imperfectly nested loops with only forward dependences can be shown to be $O(n^2)$, where n is the number of statements.

5.2 Backward Dependences

This subsection uses Property 1 to convert backward dependence problems into solely forward dependence ones, which have been solved in the previous subsection. As stated by Property 1, a backward dependence $u \longrightarrow v$ at level k may be killed by either a barrier inserted between $do(k)$ and v, or by a barrier inserted between u and $enddo(k)$. However, we do not know in advance which is the best place and therefore all possible positions will be considered.

In the first step of the algorithm at most one temporary barrier is placed immediately before some sink of a backward dependence at each loop level k. If there are any remaining backward dependences not killed by the temporary barriers, then barriers should be inserted between the sources and $enddo(k)$. To do this, each remaining backward dependence $x \longrightarrow y$ on level k is replaced by a dependence $x \longrightarrow enddo(k)$ and then the algorithm for forward dependences described above is applied. This procedure is repeated iteratively, with again at most one barrier placed before the sink of a backward dependence, until the sinks of all backward dependences have been considered. The number of times barriers are to be executed is estimated and the minimal value determines the position of the barriers. The time complexity for this algorithm is $O(r^m \cdot n^2)$, where r is the number of backward dependences and n is the number of statements.

Example To illustrate this technique, consider the example in figure 4. Initially, we do not insert any temporary barrier before the sinks of backward dependences. In this case the dependence $c \longrightarrow b$ is replaced by the dependence $c \longrightarrow enddo(3)$. We continue to apply the algorithm described above and find that the best placement of barriers is before $enddo(3)$ and after a. The number of times the barriers will be executed is $N^3 + N$. Now, we place a temporary barrier just before b and in this case we will need an additional barrier before d to kill $b \longrightarrow d$. The number of times the barriers will be executed is $N^3 + N^2$. Comparing each case we find that the best barrier placement is just after a and just before $enddo(3)$.

This procedure is optimal but potentially highly expensive. However, by reducing the number of dependences to consider, the procedure is not costly in practice.

Theorem 2 *The algorithm places the minimum number of barriers in those imperfect loop nests where each loop contains at most one further loop nest and 0 or more statements.*

For those imperfect loop nests where there are two or more enclosed loop nests at a particular level, the procedure described in section 2.1 is applied.

6 Related Work

This section describes related work and compares it to our approach. In [5, 6] a method to eliminate redundant syn-

Program	Original	Cross-Processor	After Elimination
Adi	27	13	9
cgm	251	24	8
shallow	1215	468	73
tred2	505	174	66
vert5	72	35	17

Figure 5: Data dependences at each stage

chronisation points based on kill/cover analysis is presented. It was shown that the problem of finding the minimal number of synchronisation points is NP-complete [4]. The problem of eliminating redundant or covered dependences was also considered in [3]. Quinn et al. [11] have proposed an algorithm similar in spirit to that presented in this paper. Apart from a small technical error in their algorithm, they minimise the number of synchronisations in a basic block and attempt to extend the technique to loop nests. Crucially, however, there is no consideration of data dependence distance or loop level. Thus, if a dependence had its source and sink within an innermost loop, a barrier would be placed between them even if the dependence was carried at a higher level. Their examining loops one at a time can also lead to sub-optimal results. Consider the example in figure 3, column 3. Barriers would be placed before `a` and `d` resulting in $2N^2$ barriers executed rather than N^2+N resulting from our scheme.

More recent work has also addressed the problem of synchronisation insertion for data parallel languages. Prakash et al. [10] have studied how previous techniques could be targeted to the UC data parallel language. They have proposed runtime methods, such as fuzzy barriers, to reduce the cost of synchronisation. Philippsen and Heinz [9] used program restructuring as an optimisation technique for elimination of redundant barriers in synchronous FORALLs.

In [7, 8], techniques to determine cross-processor dependences based on data and work layout are described. Tseng [15] takes a similar approach and uses, potentially more accurate, communication analysis to identify cross-processor dependences. After identifying cross-processor dependences, he uses an elimination algorithm to remove barriers unnecessarily inserted during the construction of the parallel regions. When it is profitable, any remaining barriers are replaced with a less expensive form of synchronisation such as counters. The algorithm for barrier placement in loop nests is, however, sub-optimal when there are loop carried dependences. Excessive barriers may be inserted as there is no consideration as how to place barriers so as to kill all dependences at the minimal runtime cost. This will become increasingly important as the size of the parallel regions created by their compiler increases. For instance, in the example in figure 3, column 3, a barrier would be place after `f` and between `b` and `d` giving rise to $2N^2$ barriers executed rather than N^2+N resulting from our scheme.

7 Application

To empirically validate the techniques described in this paper, the graph based algorithm was incorporated into the MARS compiler [1] as a separate phase after partitioning and scheduling. The new compiler phase consisted of approximately 2700 lines of C++. The updated compiler was applied to four well known benchmarks and one other program.

Adi is a short, 29 line, piece of code described in [2] forming one phase of alternate direction integration. *Cgm* is the conjugate gradient code from the NAS benchmarks which works on a sparse input matrix where the benchmark section is 257 lines long. *Shallow* is a 454 lines implementation of the well-known shallow water equations and *Tred2* is a 148 line "hand-cleaned" version of an Eispack routine. Finally, *vert5* is a 176 line program based on VertVel, one of the most computationally intensive routines from the U.K. Meteorological Office's (UKMO) Unified Model.

The number of dependences considered at each stage in the algorithm was recordered as was the eventual number of barriers inserted and executed. As well as recording MARS behaviour with these programs, POWER FORTRAN Accelerator (PFA), a state-of-the art commercial compiler, available on the SGI, was used as a comparison.

7.1 Results

The table, shown in figure 5, shows the number of dependences considered at various stages in the optimisation process. Determining cross-processor dependences removes between 50 % and 90 % of the dependences; the elimination procedure further reduces this number by between 40 % and 80%. The compilation time overhead for the barrier placement algorithm is now less than 0.01 sec for each of the test programs, using a Sun Sparc 4 workstation and gcc. The improvement in compilation time over our previous scheme (up to 20 mins for Shallow) is such that synchronisation placement is no longer an issue in MARS's compilation time.

The table shown in figure 6 summarises the performance of both systems. The number of barriers inserted is shown in the columns headed `static`, while the actual number executed at runtime is shown in the columns headed `executed`[1]. PFA uses a fork and join execution model, so for the purposes of comparison each fork and join is considered as a barrier. In each case MARS outperforms PFA, but there is a wide variation in the relative improvement. In *cgm* the improvement of MARS over PFA is marginal. However, PFA, rather than the alternative of inserting excessive synchronisation, makes a significant part of the program into a large sequential section. MARS, on the other hand, completely parallelises the entire application. In the small Adi example, MARS inserts no barriers between the inner loop nests as there is no cross-processor dependence; the two barriers, at the start and end of the program, are the minimum needed for all parallel programs. The barrier placement algorithm is particularly effective in the case of the Shallow benchmark. This program contains many dependences, but the careful placement of barriers can simultaneously cover approximately 7 cross-processor dependences; 73 dependences are covered by 10 barriers. In the

[1] P. denotes the actions of PFA, M. denotes the actions of MARS

Program	M. static	P. static	M. executed	P. executed	M. Speedup	P. Speedup	Relative Improv.
Adi	2	8	2	8	3.52	3.21	1.09
cgm	7	12	1157	1592	1.06	0.98	1.08
shallow	10	80	67	766	2.62	2.51	1.04
tred2	15	36	5,125	13,332	2.66	2.34	1.14
vert5	4	18	515	3084	2.49	2.21	1.13

Figure 6: Results

cases of tred2 and vert5, each barrier covers approximately 4 cross-processor dependences. For each of these programs there is also an improvement over PFA by a factor of up to 11 in the number of barriers executed at runtime.

To see the effect of the barrier reduction on performance times, the PFA and MARS generated codes were executed on a 4 processor SGI challenge; the relative speedup of each compiler over the sequential version is shown in the final three columns of figure 6. Although for all programs MARS shows an improvement, it is relative modest especially when considering the difference in the number of barriers executed. This is largely due to the small number of processors in our system. For example, in Shallow, the work per processor is $O(\frac{n^2}{p})$ but the number of barriers is constant (for a fixed number of time-steps). As the number of processors increases, so does the overhead of synchronisation. This is borne out in practice as can be seen in previous work [7, 15]. It is hoped that further experimental work on a larger system will be possible.

8 Conclusions and Further Work

This paper has presented an algorithm that provably places the minimal number of barriers in basic blocks and loops with lower complexity than existing schemes. The algorithm places fewer barriers in the entire program than all previous schemes and incorporates a dependence elimination mechanism to reduce the complexity of the scheme. This algorithm has been incorporated into a parallelising compiler and shown to outperform an existing commercial compiler.

Although we restrict our attention to FORTRAN programs in our implementation, this technique could be used for other programming languages and generalised to other constructs. In particular *while do, do while* and *repeat until* loop structures could be incorporated into this technique. The algorithm was implemented in a parallelising compiler based on data partitioning. However, it is equally applicable to compilers using other parallelisation models. Although knowledge of cross-processor dependences helps reduce the number of dependences to consider, our algorithm does not require it and thus it may easily be integrated in any system containing basic data dependence information.

Our implementation performed well on the selected programs, however, further experimentation is necessary to validate the approach. In particular it is necessary to investigate the compile-time cost when tackling codes many thousands of lines in length and the performance of generated code on large systems.

Acknowledgement We would to thank the U.K. Met. Office for access to the Unified Model

References

[1] F. Bodin and M.F.P O'Boyle, A Compiler Strategy for SVM, *the Third Workshop on Languages, Compilers and Runtime Systems for Scalable Computing*, Kluwer Press, New York, May 1995.

[2] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy K, C. Koelbel, S. Ranka and C-W. Tseng , Unified Compilation of Fortran 77D and 90D, *ACM Letters on Programming Languages and Systems*, Vol 2 Nos 1-4, March - December 1993.

[3] Z. Li, W. Abu-Sufah, On reducing data synchronization in multiprocessed loops, *IEEE Trans. on Computers*, C-36(1), pp 105-109, Jan. 1987.

[4] S.P. Midkiff, Automatic generation of synchronization instructions for parallel processors. M.S. Thesis, University of Illinois at Urbana-Champaign, CSRD Rep. 588, May 1986.

[5] S.P. Midkiff, D.A. Padua, Compiler generated synchronizations for do loops, *Proceedings of International Conference on Parallel Processing*, IEEE press, New York, Aug. 1986.

[6] S.P. Midkiff, D.A. Padua, Compiler algorithms for synchronization, *IEEE Trans. Comput.*, C-36(12), pp. 1485-1495, Dec. 87.

[7] M.F.P. O'Boyle, L. Kervella, F. Bodin, Synchronization Minimization in a SPMD Execution Model, *J. of Parallel and Distributed Computing* 29, pp. 196-210, 1995.

[8] M.F.P. O'Boyle, F. Bodin, Compiler Reduction of Synchronisation in Shared Virtual Memory Systems *9th ACM International Conference on Supercomputing*, ACM Press, July 1995.

[9] M. Philippsen, E. Heinz, Automatic synchronization elimination in synchronous FORALLs. In *Frontiers'95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995.

[10] D. Prakash, M. Dhagat, R. Bargodia, Synchronization issues in data-parallel languages. *Proceedings of International Workshop on Languages and Compiler for Parallel Computing*, pp. 76-95, LNCS 768, Springer Verlag, Aug. 1993.

[11] M. Quinn, P. Hatcher and B. Seevers, Implementing a data parallel language on a tightly coupled multiprocessors. In *Advances in Languages and Compilers for Parallel Processing*, pp.385-401. Pitman, London, 1991.

[12] E Stöhr and M O'Boyle, Barrier Synchronisation Reduction, *to appear at HPCN'97.*

[13] E Stöhr, Barrier Synchronisation Reduction in IF Constructs, *CNC Tech Rep.*, University of Manchester, September 1996.

[14] E Stöhr and M.O'Boyle A Graph based Approach to Barrier Synchronisation Reduction Minimisation, *CNC Tech Rep.*, University of Manchester, December 1996.

[15] C.-W. Tseng, Compiler optimizations for eliminating barrier synchronization, *5th ACM Symposium on Principles& Practice of Parallel Programming* , ACM Press, Aug. 1995