Compile Time Barrier Synchronisation Minimisation

M.F.P. O'Boyle (corresponding author)

Division of Informatics

The University

Edinburgh EH9 3JZ, UK

tel: (+44) 131 650 5117

fax: (+44) 131 667 7209

e-mail: mob@dcs.ed.ac.uk

E.A. Stöhr

Department of Computer Science

The University

Manchester M13 9PL, UK

tel: (+44) 161 275 6134

fax: (+44) 161 275 6204

e-mail: elena@cs.man.ac.uk

Acknowledgement We would like to thank the U.K. Met. Office for access to the Unified Model and CEPBA, UPC for extensive access to their SGI Origin 2000 under the EC TMR program. We would also like to thank Mark Bull, David Haglin and the reviewers for their helpful comments.

Abstract

This paper presents a new compiler approach to minimising the number of barriers executed in parallelised programs. A simple procedure is developed to reduce the complexity of barrier placement by eliminating certain data dependences, without affecting optimality. An algorithm is presented which, provably, places the minimal number of barriers in perfect loop nests and in certain imperfect loop nest structures. This scheme is generalised to accept entire, well-structured control-flow programs containing arbitrary nesting of `IF` constructs, loops and subroutines. It has been implemented in a prototype parallelising compiler and applied to several well-known benchmarks where it has been shown to place significantly fewer synchronisation points than existing techniques. Experiments indicate that on average the number of barriers executed is reduced by 70% and there is a 3 fold improvement in execution time when evaluated on a 32-processor SGI Origin 2000.

keywords: compiler optimisation, synchronisation reduction, efficient parallelisation, barrier minimisation, graph algorithms.

1 Introduction

Many widely-used parallel machines, such as the SGI Origin 2000, HP Exemplar and Sun Enterprise, 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 *cross-processor data dependences* [22] and maintain program semantics [1]. 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

[1]Cross-processor data dependences or non-local dependences [32] occur when a piece of data is written/read on one processor and read/written on another.

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 T3E, synchronisation reduction is essential. In [12] it was shown that, over a range of benchmarks, synchronisation accounted for 37% of the execution time on 16 processors. As the number of processors increases so does the relative overhead of synchronisation, making it a chief obstacle to scalability.

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 provably places the minimal number of barriers and in other cases outperforms existing techniques. Thus, the programmer is relieved of the burden of reasoning about synchronisation placement, without incurring performance degradation, making single - address space parallel architectures more attractive.

Previous work [21] 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.

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 4-6 present powerful algorithms for barrier placement. In Section 7 a review of existing work is presented and in Section 8 the performance of our compiler implementation is evaluated and compared to existing techniques. Section 9 concludes the paper.

2 Approach

This section first presents a motivating example illustrating our approach which is followed by a summary of the algorithm developed in this paper.

2.1 Example

To illustrate the compiler approach developed in this paper, consider the program fragment shown in Figure 1, column 1. By applying the scheme developed in this paper, we eventually determine the barriers needed for the parallel SPMD program shown in the final column. The `C$DOACROSS` directive instructs the runtime system to schedule each of the iterations of the `myid` loop across the available, `NUM_THREADS`, threads. As `NUM_THREADS` equals the number of available processors, each processor executes in parallel just one iteraton of the code within the parallel `myid` loop. Each processor has a number of local variables where `lo` and `hi` are the local loop bounds found after parallelisation and partitioning. Thus, if $N = 64$ and `NUM_THREADS` $= 4$, processor 1 will have the values `myid=1, lo=1, hi= 16`, processor 2 will have the values `myid=2, lo=17, hi=32` etc. In this way we have SPMD execution where each processor executes the same code on different data elements and synchronisation between processors must be inserted by the compiler to ensure correctness.

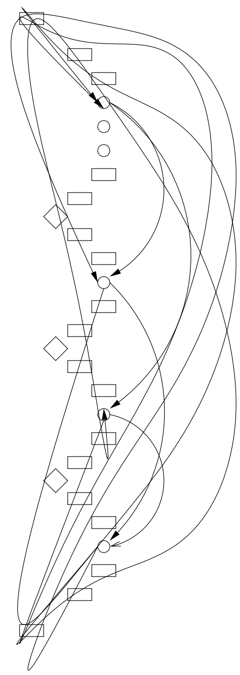
Initially, there are 12 data dependences to consider for synchronisation. A data dependence [35] occurs when two references in a program are to the same data element in memory. If one of these references is a write, then the order in which the references take place in the original program must be preserved in the parallelised program to ensure program semantics are preserved. For further details see [35]. For array P, there are the following four dependences: s1 $\longrightarrow$ s4, s1 $\longrightarrow$ s5, s4 $\longrightarrow$ s1, s5 $\longrightarrow$ s1. For array U, there are two

Original Program (1)

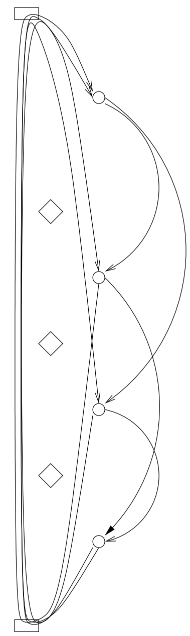
```
DO  STEP = 2,MAXTIMESTEP
L1:  DO  I=1,N
      DO  J=1,N
s1:       P(I,J) = PNEW(I,J)
s2:       U(I,J) = UNEW(I,J)
s3:       V(I,J) = VNEW(I,J)
      ENDDO
     ENDDO

IF:  IF (MOD(STEP,4) .EQ. 0) THEN
L2:    DO  I=1,N-1
        DO  J=1,N
s4:        CAPU(I+1,J) = 0.5 *
           (P(I+1,J) +P(I,J)) *
            U(I+1,J)
        ENDDO
       ENDDO
     ELSE
L3:    DO  I=1,N
        DO  J=1,N-1
s5:        CAPV(I,J+1) = 0.5 *
            (P(I,J+1)+P(I,J))
             * V(I,J+1)
        ENDDO
       ENDDO
     ENDIF
     . . .
L4:  DO  I=1,N-1
      DO  J=1,N-1
s6:     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
```

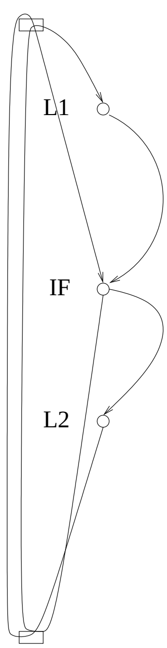
Cross-Processor (2)



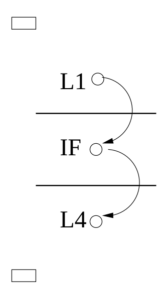
Summarise Loops (3)



Summarise IF (4)



Elimination and Barriers (5)



Parallel Program (6)

```
C$DOACROSS LOCAL(myid,I,J,lo,hi,step)
DO myid = 1, NUM_THREADS
 lo = (N/ NUM_THREADS)* (myid-1) +1
 hi = (N/ NUM_THREADS)* myid

 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()

  IF (MOD(STEP,4) .EQ. 0) THEN
    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
  ELSE
    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
  ENDIF

  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
ENDDO
```

Figure 1: Example illustrating different stages in barrier placement optimisation

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.

The partitioning phase of the compiler determines that four of the dependences are local [32, 22]. This gives the cross-processor dependence graph shown in column 2, where a box represents a `do` or `enddo` control node, a diamond represents an `IF`, `THEN` or `ENDIF` control node and a circle represents a vertex in the dependence graph corresponding to a program statement.

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 and an `IF` construct inside the outer loop. Thus, the perfect loop nests within each branch of the `IF` construct ($L2, L3$) are considered individually, as are the other loop nests ($L1, L4$). Since there are no dependences in each of these loop nests, the next step is to consider each loop nest as a vertex in the graph, shown in column 3. We then similarly consider the `IF` construct as a vertex in the graph, shown in column 4, as there are no dependences whose source and sink are entirely within a branch. By applying the elimination algorithm given in Section 5.1, we can show that only dependences L1 $\longrightarrow$ `IF`, and `IF` $\longrightarrow$ L4 are critical. As long as these dependences are satisfed, so will all other dependences which are therefore redundant and can be removed from further consideration. Now, applying the perfect loop algorithm, described in Section 5, it can be shown that barriers placed just before the `IF` and L4 will satisfy the remaining dependences, as shown in columns 5 and 6.

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 four barriers. The scheme described in this paper places just two 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. If there were, for instance, a data dependence between the two branches of the `IF` construct, the dependence would be loop carried, as only one branch can be executed per iteration. After considering the `IF` construct as a vertex, there would be a

```
overallBarrierPlacement(module)
  IF module contains a subroutine THEN
      FOR each subroutine SB in module
          Perform overallBarrierPlacement(SB)
  ELSE
      Perform barrier placement algorithm (Figure 3)
      Remove killed dependences
      Treat module as a statement
```

Figure 2: The overall barrier placement algorithm

loop carried dependence from the `IF` vertex to itself, which in this case would be discarded as non-essential after elimination, giving precisely the same barriers as shown in column 5.

Although the algorithm is illustrated in the context of an SPMD paralleised program, it is equally applicable to other parallelisation models based on fork/join synchronisation. Knowledge of cross-processor dependences helps reduce the number of dependences to consider, but our algorithm does not require it and may use more basic data dependence information.

2.2 Algorithm Summary

The algorithm developed in this paper assumes that cross-processor dependences are determined by a previous phase of the compiler [22] and our goal, therefore, is to place the smallest number of (executed) barriers to satisfy them. Throughout this paper we refer to barriers *killing* dependences [25], in the sense that the insertion of such a barrier guarentees that the ordering of a program required by the dependence is satisfied and that we need no longer consider the *killed* dependence.

The overall barrier placement algorithm, shown in Figure 2, is a bottom-up approach which starts with the main program, overallBarrierPlacement(`Program`), and applies the recursive barrier placement algorithm, presented in Figure 3, to the statements inside each subroutine. The subroutine is then considered as a statement which is the source or sink of those dependences which were not killed by barriers placed inside the subroutine. The recursive algorithm is again applied to the remainder of the program.

```
barrierPlacement(code_seg)
  Forall top-level constructs con in code_seg
      IF con is an IF construct THEN
          Perform barrierPlacement() on THEN branch
          Perform barrierPlacement() on ELSE branch
          Remove killed dependences, merge branches and consider con as a statement
      IF con is a loop construct THEN
          IF con is a perfect loop nest with just statements in loop body THEN
            Perform perfect loop algorithm (Section 5)
          ELSEIF con is a suitable imperfect loop nest
            Perform imperfect loop algorithm (Section 6)
          ELSE Perform barrierPlacement() on loop body
          Remove killed dependences and consider con as a statement
  Apply FFS algorithm to statements in code_seg
```

Figure 3: The recursive barrier placement algorithm

The recursive algorithm, shown in Figure 3, first determines barrier placement on each branch of an `IF` statement. The `IF` statement is then considered as a single statement which is the source or sink of the combined dependences which were not killed by barriers in either branch.

We split program constructs into suitable categories so as to allow the application of algorithms of lower complexity wherever possible. For loop nests, the recursive algorithm applies either the perfect loop algorithm or the nested loop algorithm (if a loop contains several statements and one further loop nest, which may also contain statements and a further loop nest etc.). In these cases the algorithm places the minimum 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. After applying the barrier placement algorithm to the loops, all of the killed dependences are removed and the loop is considered as a statement. This process is applied recursively until the entire program is considered. If there are any remaining dependences an algorithm, First Fast Sink (FFS), which places the minimal number of barriers in basic blocks, is used. In this way programs containing arbitrary nestings of subroutines, `IF` constructs and loops may be considered. As our approach relies on structured control-flow, control-flow normalisation as a pre-processing phase will be required for those programs containing unstructured gotos.

3 Graph Representation

This section introduces the graph notation used throughout this paper to describe the various parts of the optimisation process. The *control/dependence graph* [21] is a directed graph $CDG = (S, C, E)$, where S are the vertices, corresponding to statements and C, E are the edges corresponding to control-flow [35] and cross-processor data dependences [22] of a program respectively. Such data dependences may be either a flow (from a write reference to a read), output (from a write to a write) or anti (from a read to a write) dependences [35] whose source and sink are on different processors. In our figures, for clarity, we represent vertices of the CDG corresponding to statements by circles, vertices corresponding to controls statements such as `do` and `enddo` by boxes and only illustrate data dependence edges where control-flow is obvious.

We number all statements, including control statements, according to their lexicographic order [10]. This gives a total order to vertices within basic blocks and loop nests where we can determine which vertex or statement precedes another simply by comparing their numeric values. Thus, a dependence with a source in a statement number i and a sink in a statement number j corresponds to an arc $i \longrightarrow j$ in the CDG. In the case of `IF` statements, since we examine each branch independently, in isolation, before merging the results into one vertex, there is never any ambiguity about vertex or statement order. This independent treatment of branches and then the merging of results avoids the need to consider partial orders in the presence of conditional control flow.

As we develop specific algorithms for loop nests, we further examine dependences within these control structures. Let $L = L_1, L_2, \ldots, L_m$ be a loop nest of depth m, where L_k is defined as the loop at *level* k, [35], whose graph contains a header denoted by $do(k)$ and an exit by $enddo(k)$. Dependences with a source and sink within the same iteration are defined as *forward* dependences.

A loop-carried dependence [35], has its source and sink in different iterations of a loop nest. Furthermore, a loop-carried dependence at a particular loop level, with a source i and a sink j, is defined as either a *backward* dependence, if the sink is lexicographically before the source ($j < i$), or a *wrap-around* dependence, if the sink is lexicographically after ($j \geq i$).

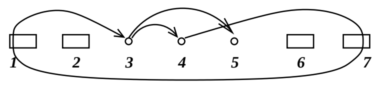
<table>
<tr><th>Loop (1)</th><th>Dependence Graph (2)</th><th>Adjacency Lists (3)</th></tr>
<tr><td>1. do i=2,n
2. do j=1,m
3. a(i,j)=b(i-1,j)
4. b(i,j)=a(i,j)
5. c(i+1,j)=a(i,j)+1
6. enddo
7. enddo</td><td>
</td><td><table><tr><th>Forward</th><th>Backward</th><th>Wrap</th></tr><tr><td>3 ⟶ 4, 5</td><td>Lv1: 4 ⟶ 3
Lv2: -</td><td>Lv1: -
Lv2: -</td></tr></table></td></tr>
</table>

Figure 4: A loop nest, its dependence graph and corresponding adjacency lists

This gives rise to the following property:

Property 1 A backward dependence at loop level k, $i \longrightarrow j$, where i is the source of the dependence and j is the sink, may be killed by any barrier inserted between i and $enddo(k)$ or between $do(k)$ and j. A wrap-around dependence at level k may be killed by any barrier inserted between $do(k)$ and $enddo(k)$.

This follows directly from the fact that a dependence is killed by a barrier placed between its source and sink at an appropriate loop level

Each vertex i may be the source of several dependences, $i \longrightarrow a_{i,p}$, where $a_{i,p}$ is the pth sink whose corresponding source is vertex i. Such dependences can be expressed using ordered adjacency lists [2] where a vertex $a_{i,p}$ is on the adjacency list for vertex i if there is a dependence arc $i \longrightarrow a_{i,p}$. For every loop we divide all loop dependences into forward, backward and wrap-around dependences, at each level, and place them in separate adjacency lists. To illustrate the use of this formulation, consider the loop nest shown in Figure 4. Column 2 graphically describes the dependences of the loop nest which are represented as adjacency lists in column 3. The arcs $3 \longrightarrow 4$ and $3 \longrightarrow 5$ are the two forward dependences with a source in vertex 3 where $a_{3,1} = 4$ and $a_{3,2} = 5$. The arc $4 \longrightarrow 3$ is a backward dependence at level 1 (Lv1) and by Property 1, this level 1 backward dependence can be killed by placing a barrier between vertices 4 and 7 or between 1 and 3.

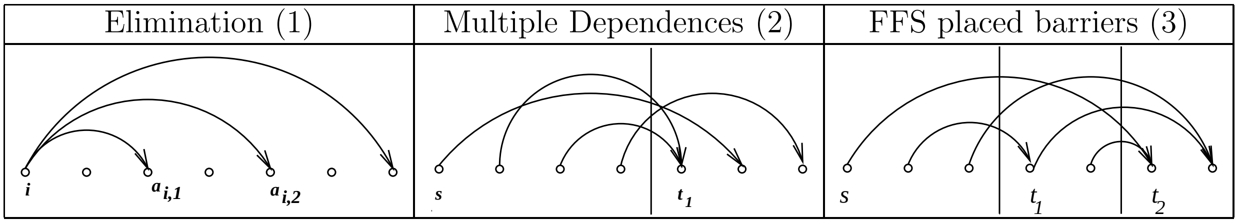


Figure 5: An example illustrating the algorithm for basic blocks

4 Barrier Placement in Basic Blocks

This section briefly describes two techniques used for the efficient insertion of the minimal number of barriers within a basic block. Firstly, it describes an elimination technique to reduce the number of dependences to consider, followed by a description of the First Fast Sink (FFS) algorithm, an efficient synchronisation placement algorithm. These two techniques form the basis for subsequent algorithm phases.

4.1 Dependence Elimination

Throughout this paper, the problem is simplified at each stage and we therefore first present an easy scheme to eliminate from consideration those dependences that are guarenteed to be satisfied. Such dependence elimination does not affect the legality or accuracy of barrier insertion, it merely reduces the problem size for our barrier insertion algorithm.

Consider all those dependences whose source is at vertex i. (See Figure 5, column 1) and let $a_{i,1}$ be the sink of the dependence nearest to i. If we place a barrier between statements i and $a_{i,1}$, all the dependences with sources in i are killed. Thus, for every source, we need only consider the first node of its adjacency list, *eliminating* the remainder. This procedure reduces the number of arcs to be considered, and reduces the computational requirements of later stages. Although more dependences could be removed at this stage, it is unnecessary, as it will not reduce the complexity of the FFS algorithm. For this reason, we do not use cover analysis [25], as the additional expense it will incur does not reduce the cost, or improve the accuracy, of later stages.

4.2 FFS Algorithm

This subsection gives a brief description of the FFS algorithm, which places the minimum number of barriers in basic blocks after elimination has taken place. Consider column 2 of Figure 5. If t_1 is the first sink after a vertex s, then placing a barrier just before t_1 will kill all those dependences with sources between s and t_1. These dependences can be removed from further consideration and the process repeated from the vertex t_1. Using this property the FFS algorithm can be summarised as follows:

Algorithm Starting at vertex 1, find the first sink, t_1, after vertex 1. Place a barrier just before t_1. Then, iteratively, choose t_i as the nearest sink of all the dependences whose sources are at or after t_{i-1}. Place a barrier just before t_i.

For instance, in Figure 5 column 3, the first sink after vertex 1 is $t_1 = 4$. Placing a barrier just before t_1 kills all the dependences with sources between 1 and 4. Similarly, placing a barrier just before $t_2 = 6$ kills all dependences with sources at 4 and 5. As there are no further dependences with sources at or after t_2, all the dependences have been killed. The FFS algorithm allows the determination of those *essential* dependences which must be satisfied by barrier synchronisation:

Definition 1 *Let t_i, $i = 1, \ldots, \ell$, be the "first" sinks of dependences found according to the FFS algorithm. Let q_i be the source of the dependence with a sink t_i whose source is nearest to t_i. The dependences $q_i \longrightarrow t_i$, $i = 1, \ldots, \ell$, are called essential dependences/arcs.*

As any two essential arcs do not intersect, this gives rise to to the following theorem:

Theorem 1 *The number of essential arcs is equal to the minimal number of barriers.*

Proof: See [29]

For examples, and the proof that the algorithm finds the minimal number of barriers in n steps, where n is the number of statements in the basic block, see [29]. The FFS algorithm is used as the basis for the following loop based algorithms.

5 Barrier Placement in Perfect Loop Nests

This section develops an algorithm to efficiently place the minimum number of barriers in a perfect loop nest. Initially, redundant dependences which are guarenteed to be satisfied are eliminated. This is a relatively inexpensive operation and reduces the problem size for the second stage which determines the placement of barriers in the remaining dependence graph.

5.1 Dependence Elimination

We wish to reduce the number of dependences and hence the computational requirements 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 adjacency lists for forward and backward dependences. The absence of forward dependences is handled as a special case (see the end of Section 5.2).

First, we note, that any forward dependence can be considered as a dependence within a basic block. Thus, we can reduce the number of forward dependences in exactly the same manner as described in Section 4.1. As we are considering loop nests, however, further dependence elimination is useful. Consider the case illustrated in column 1 of Figure 6 where $i \longrightarrow j$ is a backward dependence at loop level k and $u \longrightarrow v$ is a forward dependence such that the source u is after the `do` loop header at level k ($u \geq do(k)$) and the sink v is before the sink j ($v \leq j$). Since any barrier which kills $u \longrightarrow v$ will also kill $i \longrightarrow j$, we can remove $i \longrightarrow j$ from the adjacency list of backward dependences at level k.

The procedure for reducing the number of dependences is as follows: At each loop level find the forward dependence with the earliest sink and remove all other backward dependences with later sinks. Similarly, find the forward dependence with the latest source and remove all other backward dependences with earlier sources.

Finally, consider the case illustrated in column 2 of Figure 6. If there are any backward

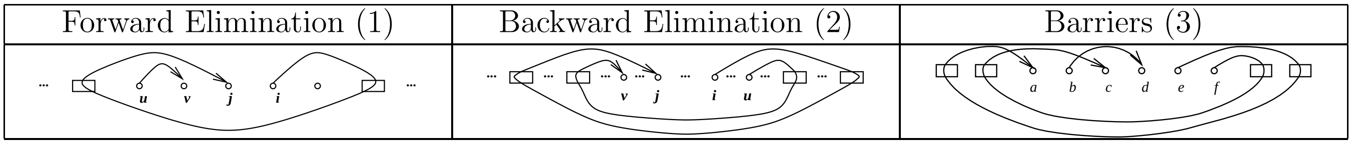


Figure 6: Elimination and barrier placement in perfect loop nests

dependences $i \longrightarrow j$ at say level k and a backward dependence, $u \longrightarrow v$, at a deeper level r, $(r \geq k)$ where the source i is before the source u and the sink j is after the sink v, then $i \longrightarrow j$ can be removed.

The elimination of dependences can be performed in $O(m \cdot n)$ steps, where m is the loop nest depth and n is the number of statements in the loop, and gives rise to the following property:

Property 2 After dependence elimination, all the sinks of backward dependences are before the earliest sink of any forward dependence and all the sources of backward dependences are after the last source of any forward dependences at each corresponding loop level.

5.2 Barrier Placement Algorithm

This section describes the optimal insertion of barriers in perfect loop nests after dependence elimination. It is based on finding the minimum loop nest depth for correct barrier insertion and relies on the FFS algorithm.

Consider the forward dependences within a perfect loop nest L. Let ℓ be the number of essential arcs, as defined in Definition 1, where $q_1 \longrightarrow t_1$ is the first essential arc. Let $x_1, \ldots, x_s$, be the sinks of backward dependences between q_1 and t_1 and let $x_0 = t_1$. By property 2, no backward dependence has a sink after t_1. Using this fact, the following procedure is repeated for every sink x_i.

Algorithm Insert a temporary barrier before x_i, remove all the dependences killed and use the FFS algorithm to kill any remaining forward dependences. If all the dependences are killed with ℓ barriers, then we have found the minimal number of barriers and do not need to proceed with the algorithm. 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 the temporary barriers, then from $k_0, k_1, \ldots, k_s$ choose the minimum level, say k_{min}. Place barriers after $do(k_{min})$, before statement x_{min}, and use the FFS algorithm to kill the remaining dependences.

Example To illustrate this approach, consider the example in Figure 6, column 3. There is only one forward dependence, $q_1 \longrightarrow t_{1,}(b \longrightarrow d)$ and only one node c between $b \longrightarrow d$ which is the sink of a backward dependence. According to our algorithm d is x_0 and c is x_1. Placing a barrier before d does not eliminate all the dependences; the deepest loop level of the remaining dependences is 2 ($f \longrightarrow c$), so $k_0 = 2$. Placing a barrier before c 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 ($e \longrightarrow a$), so $k_1 = 1$. As $k_1 < k_0$, ($k_{min} = k_1$), we place a barrier before c and after $do(1)$. Assuming that each of the loops is executed N times, the two barriers inserted will be executed $N^2 + N$ times.

This algorithm determines the minimal number of times the barriers in a perfect loop nest will be executed. The time complexity of the algorithm is $O(n \cdot d)$, where d is the number of backward dependences after applying elimination. The main result of this section is the following theorem:

Theorem 2 *The algorithm places the minimal number of executed barriers in polynomial time for perfect loop nests.*

Proof For a proof of this theorem see Appendix A.

No forward dependences If there are no forward dependences, then optimal barrier synchronisation insertion is trivial. A barrier is placed after $do(p)$, where p is the maximal level of backward and wrap-around dependences.

6 Barrier Placement in Imperfect Loop Nests

This section considers imperfect loop nests and provably places the minimal number of barriers in certain imperfect loop nest sequences: those loops whose body may contain a

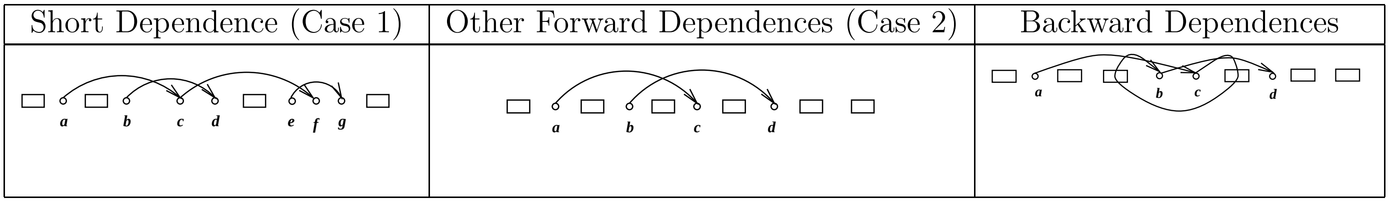


Figure 7: Barrier placement in imperfect loop nests

further loop nest and several statements. Due to the increased complexity of the algorithms developed, we initially further eliminate unnecessary dependences from consideration. To aid presentation of the barrier insertion technique, we divide the algorithm in two : those containing just forward dependences (Section 6.2) and those containing forward and/or backward dependences (Section 6.3). The forward dependence algorithm is further divided into two separate cases depending on the structure of the dependences. The backward dependence algorithm is based on transforming any backward dependence problem to the forward dependence case. Hence, all forms of data dependence within imperfect loop nests are considered.

6.1 Dependence Elimination in Imperfect Loop Nests

Initially, we replace all wrap-around dependences with a forward dependence (see Property 1). This does not affect barrier placement optimality, but allows wrap-around dependences to be optimised using the same techniques used for loop nests with entirely forward dependences. In addition to the elimination described in Section 5, we eliminate covered dependences using the following observation: for every dependence, $u \longrightarrow v$, if there is an dependence whose source and sink are entirely between u and v, then $u \longrightarrow v$ is covered and can be removed from the adjacency list. This elimination procedure is of $O(n^2)$ complexity, where n is the number of statements in the loop.

6.2 Forward Dependences

Without loss of generality we may assume that the loop at the highest loop level m contains a statement involved in a dependence. A dependence $u \longrightarrow v$ with a source before $do(m)$

and a sink after *enddo*(m) is defined as a *long dependence*. All other dependences are defined as *short dependences*. In this section we present two algorithms for imperfect loop nests with forward dependences: first for the case when there is at least one short dependence entirely within the level-m loop and the second for the case when there are no such dependences. This classification allows the use of algorithms with lower complexity to be used wherever possible.

The FFS algorithm places barriers before the sinks of dependences. However, if the sink is at a deeper level than its source, it is cheaper to place the barrier just after the source. For this reason we introduce a new graph to allow optimal barrier placement in imperfect loop nests. Let $H = (V, E)$ be a directed graph and let $H' = (V, E')$ be a directed graph obtained from H by replacing every arc $u \longrightarrow v \in E$ with the source u and the sink v by a reverse arc $v \longrightarrow u \in E'$ with the source v and the sink u. The *Reverse FFS algorithm* is defined as being applied to the graph H if the FFS 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 barrier placement in a loop nest.

6.2.1 Short Dependences inside the innermost Loop (Case 1)

Algorithm Let $u \longrightarrow v$ be the short dependence with the earliest sink of all the dependences entirely within the loop at level-m. Let $x_1, \ldots, x_s$ be the sinks of dependences between u and v and let $x_{s+1} = v$. For each x_j we temporarily place a barrier before it, removing any killed dependences and then apply the FFS algorithm to the remaining dependences after the barrier and the reverse FFS algorithm to the remaining dependences before the barrier. The position with the expected minimum number of executions is chosen.

Example To illustrate this technique, consider the example in Figure 7 (Case 1). There is one short dependence $b \longrightarrow d$ between $do(2), enddo(2)$. There is only one sink c between b and d, hence $x_1 = c$, $x_2 = d$. Placing a barrier before $x_1 = c$ kills the dependences $b \longrightarrow d$ and $a \longrightarrow c$. Applying the FFS algorithm to the remaining dependences after the barrier, gives a further barrier just before f. This gives overall $N^2 + N$ barriers at runtime. Placing a barrier before $x_2 = d$, however, kills $b \longrightarrow d$, $c \longrightarrow f$. Applying the reverse FFS algorithm to the remaining dependences before the barrier, places a further barrier after a, applying FFS to the dependences after the barrier gives a barrier before g, giving overall, $N^2 + 2N$ barriers at runtime. Our algorithm selects the first case.

6.2.2 Other Forward Dependences (Case 2)

In this case there are dependences with either a source or sink between $do(m)$ and $enddo(m)$, but not both. If there are no long dependences, then the FFS algorithm is simply applied to the dependences after $enddo(m)$ and the reverse FFS algorithm to the dependences preceding $do(m)$. Consider now, however, the case when after the elimination there are some long dependences. The algorithm works as follows:

Algorithm For each long dependence, place a temporary barrier just after the source, remove any killed dependences and then apply the FFS algorithm to the remaining dependences after the barrier and the reverse FFS algorithm to the remaining dependences before the barrier. Determine the number of times the barriers will be executed. To complete the algorithm place a temporary barrier just before the first sink of the first long dependence, removing any killed dependences and once again apply the FFS algorithm to the dependences after the barrier and the reverse FFS algorithm to those before. Once again, determine the number of times the barriers will be executed. From these values choose the minimum.

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

Example To illustrate this technique, consider the example in Figure 7 (Case 2). 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 times. Continuing the algorithm, we place a temporary barrier before d. Then, on applying the reverse FFS algorithm, we shall place a barrier after a to kill the dependence $a \longrightarrow c$. In this case the barriers will be executed $N^2 + N$ times and therefore the placement of a barrier after b is selected.

6.3 Backward Dependences

This subsection uses Property 1 to convert backward dependence problems into solely forward dependence ones, allowing the direct application of the algorithms described in Section 6.2. As stated by Property 1, a backward dependence at loop level k may be killed by either a barrier inserted between $do(k)$ and its sink, or by a barrier inserted between its source and $enddo(k)$. However, we do not know in advance which is the best place and therefore all possible positions will be considered.

Algorithm In the first step of the algorithm, for each loop level k, choose at most one sink v of a backward dependence $u \longrightarrow v$. Now, replace the backward dependence $u \longrightarrow v$ by the forward dependence $do(k) \longrightarrow v$ and remove all level-k backward dependences with sinks after v. All other level-k backward dependences with sinks before v are replaced by the dependence $x \longrightarrow enddo(k)$. Now we have a graph containing only forward dependences and can apply the algorithm for forward dependences described above. This procedure is repeated iteratively, with at most one sink of a backward dependence chosen at each level, until the sinks of all the backward dependences have been considered. The minimal expected number of times a set of barriers would be executed determines the final placement.

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 in the loop. This procedure is optimal, but potentially highly expensive. However, by reducing the number of dependences to consider, the procedure is not costly in practice. For those imperfect loop nests where there are two or more enclosed loop nests at a particular level, the procedure described in Section 2.2 is applied.

Example To illustrate this technique, consider the final example in Figure 7. Initially, we do not choose the sink of any backward dependence. In this case the dependence $c \longrightarrow b$ is replaced by the dependence $c \longrightarrow enddo(3)$. Now, we apply the algorithm for forward dependences, 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 choose the sink b and replace the dependence $c \longrightarrow b$ with the dependence $do(3) \longrightarrow b$. In this case we will need a barrier before b and a 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 section gives rise to the following Theorem:

Theorem 3 *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.*

Proof The proof is given in Appendix B.

7 Related Work

Synchronisation is one of the key overheads in parallel computing and techniques to minimise its impact have been the area of research for a number of years. Here we briefly summarise the major themes in synchronisation reduction.

Program transformations to eliminate or decrease the cost of synchronisation associated with loops have been proposed by several researchers [6, 1, 16, 18, 19, 20]. In [17] loop fusion is used to avoid synchronisation between loop nests while in [6] the inverse transformation, loop distribution, is used to move a synchronisation within a loop to a less expensive one between loops. In [35, 13], loop interchange is used to place a parallelised loop outermost and in [1], loop alignment is used to eliminate cross-processor dependences and hence synchronisation. Scheduling of work and data has also been investigated as a means of eliminating cross-processor dependences [3, 16, 21, 32].

Other work has concentrated on using less expensive forms of synchronisation than the barrier. Data and event synchronisation using post/wait statements have been extensively used to synchronise between loop iterations [9, 34, 14, 7] and data elements [30, 31]. In [32], three target synchronisation primitives are considered for an optimising compiler and in [12] this work is extended by exploiting properties of the underlying lazy consistency model. Prakash et al. [24] have proposed runtime methods, such as fuzzy barriers, to reduce the cost of synchronisation, while Philippsen et al. [23] addresses the problem of inserting synchronisations in FORALL statements.

This paper is primarily concerned with reducing the number of barriers inserted given a particular set of cross-processor dependences and a specific synchronisation implementation and is therefore complementary to the above approaches. In [19, 20] a method to eliminate redundant synchronisation 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 [18]. The problem of eliminating redundant or covered dependences was also considered in [15, 25]. Furthermore, in [11], data-flow information is used to help eliminate redundant synchronisation in message-passing.

In a preliminary version of this paper [28], we showed the usefulness of a graph-based approach to barrier synchronisation placement but did not prove the optimality of our technique nor empirically validate it. Quinn et al. [26] 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 examination of loops one at a time can also lead to sub-optimal results. Consider the example in Figure 6, 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.

Tseng [32], after identifying cross-processor dependences, uses an elimination algorithm to remove barriers unnecessarily inserted during the construction of the parallel regions. 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 of how to place barriers so as to kill all dependences at a 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 6, column 3, a barrier would be placed after `f` and between `b` and `d` giving rise to $2N^2$ barriers executed, rather than N^2+N resulting from our scheme. More recent work [12], explores the combination of synchronisation optimisations

	Barriers inserted			Barriers executed			Speedup		
Program	G	O	P	G	O	P	G	O	P
Tomcatv	5	8	14	42	162	242	16.6	11.8	3.73
Swm	11	19	30	181	338	406	7.58	4.92	0.89
Mgrid	13	22	18	18,480	65,102	71,440	6.66	2.17	0.95
Erhs	10	26	6	6148	24,578	6	0.35	0.08	11
Vptst	4	10	16	22	10,280	320	27	0.39	18.59
Btrtst	5	9	16	23	4,240	158,400	10.26	0.155	3.31
Chotst	4	6	12	22	20,040	70,120	16.99	0.51	0.18
Adi	2	2	8	2	2	8	6.23	6.23	4.26
shallow	10	35	80	67	171	766	34.79	16.91	26.13
tred2	15	32	36	20,485	37,311	53,258	6.37	4.99	1.72
vert5	4	13	18	515	2,057	3,084	2.72	0.49	0.42
Average	7.3	16.53	23.05	4,182	14,913	35,745	12.3	4.41	6.47

Figure 8: The number of barriers inserted and executed by the Original(O) and Graph Algorithm (G) and PFA (P) and their speedup on 32 processors of an SGI Origin 2000

and is empirically compared with our approach in Section 8.2.

8 Application

To empirically validate the techniques described in this paper, the graph based algorithm was incorporated into a prototype compiler, MARS [5] and compared to its original synchronisation placement algorithm. Both approaches were then applied to eleven benchmarks. The original algorithm places barriers around parallel loop nests and applies a number of optimisations to eliminate unnecessary synchronisation. Barriers are placed at the outermost possible level within loop nests. Redundant barriers are removed between successive parallel regions and barriers are eliminated altogether, if there does not exist a data dependence with its source and sink within the parallel region. This original scheme is equivalent to SUIF's default model with the additional optimisations of merging SPMD regions and the use of dependence analysis as described in [12].

Although evaluating the new algorithm with respect to the original scheme within the same compiler isolates the effect of the algorithm, it is possible that relative performance improvements are due to poor implementation elsewhere in the compiler. For this reason, an existing

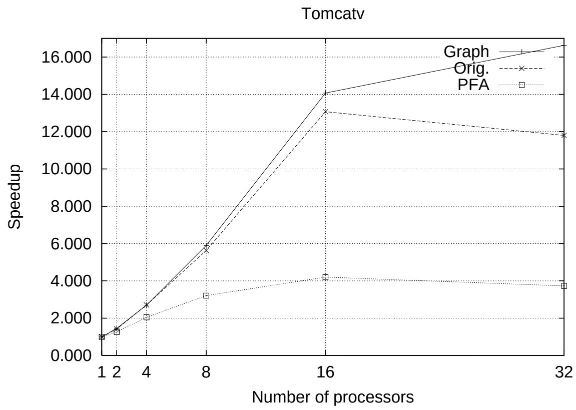


Figure 9: Tomcatv

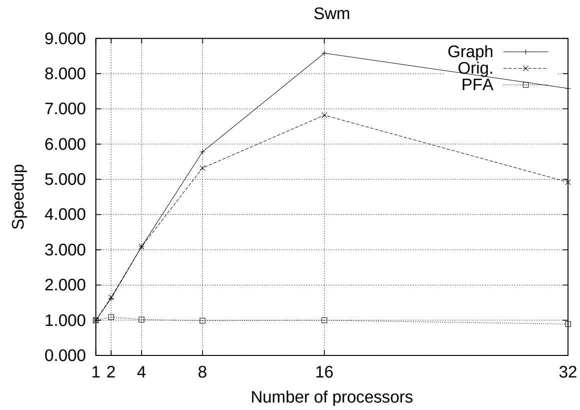


Figure 10: Swm

state of the art commercial parallelising compiler, POWER FORTRAN ACCELERATOR (PFA) is also used as a means of determining whether the gains obtainable are competitive when compared to existing approaches.

The benchmarks include subroutines and general control-flow structure including nested `do` loops and `IF` constructs. The first seven benchmarks *Tomcatv*, *Swm*, *Mgrid*, *Erhs* and *Vptst, Btrtst, Chotst* are from the SPEC '95 and '92 benchmark suites respectively. *Tred2* is from the Eispack suite while *Adi* is a short piece of code described in [8] forming one phase of the alternating direction implicit method. *Shallow* is an implementation of the well-known shallow water equations and finally, *vert5* is based on VertVel, one of the most computationally intensive routines from the U.K. Meteorological Office's Unified Model.

8.1 Results

Each of the benchmarks was compiled by MARS with both the original and graph synchronisation algorithms and the resulting code executed on up to 32 processors of an SGI Origin 2000 [2]. The table shown in Figure 8 summarises the performance of both algorithms and PFA. The number of barriers inserted is shown in the columns headed `Barriers inserted`, while the actual number executed at runtime is shown in the columns headed `Barriers executed`. The final three columns refer to the actual speedup on 32 processors relative to the performance of the original sequential program. In the following paragraphs we analyse

[2]all codes were compiled with the same native f77 compiler with -O3.

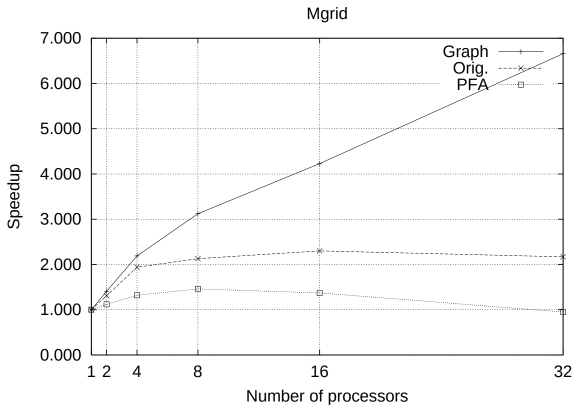


Figure 11: Mgrid

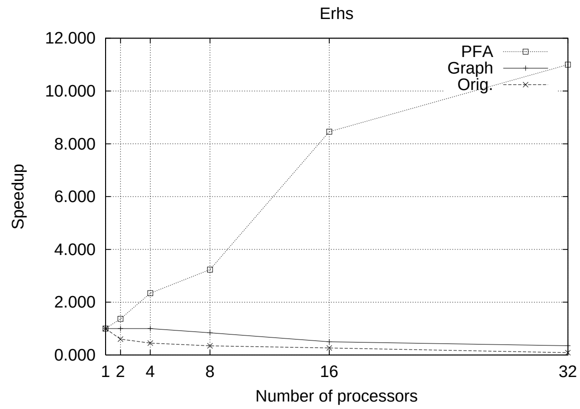


Figure 12: Erhs

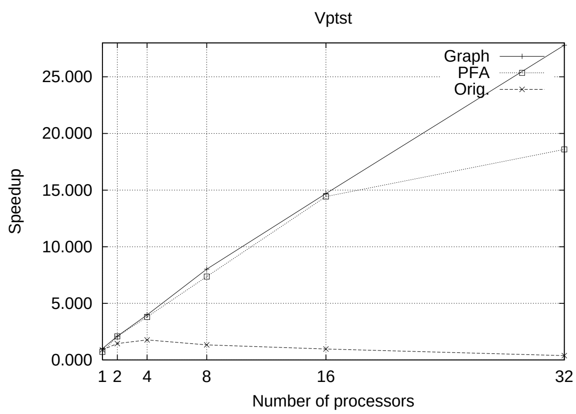


Figure 13: Vptst

the behaviour of our algorithm with respect to each benchmark in greater detail.

Tomcatv The original algorithm inserts almost twice as many barriers as inserted by our graph algorithm. Although the graph algorithm has fewer barrier calls executed at runtime, this is not translated into a noticeable performance benefit until the relative overhead of synchronisation increases for large processor number. In Figure 9, the graph algorithm gives a 41 % improvement in speedup over the original algorithm for the case of 32 processors. As a comparison, PFA fails to reach a quarter of the speedup obtained by the graph algorithm due to excessive synchronisation and a failure to parallelise an inner loop.

Swm The original algorithm incurs approximately twice as many runtime barriers as our new scheme, the effect of which is again most noticeable with a higher number of processors.

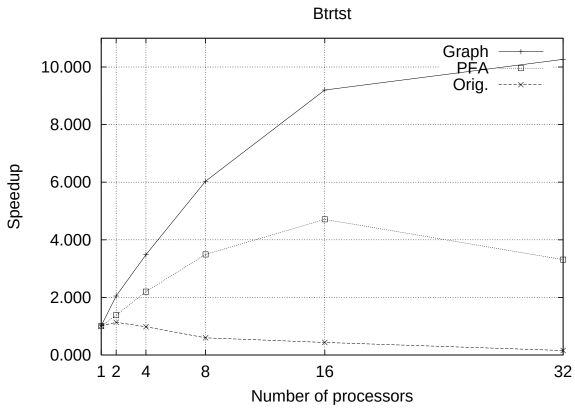


Figure 14: Btrtst

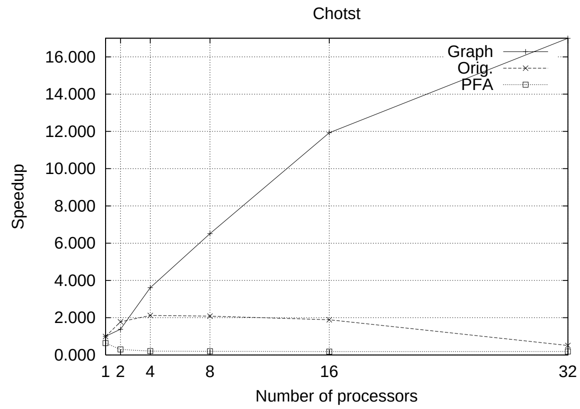


Figure 15: Chotst

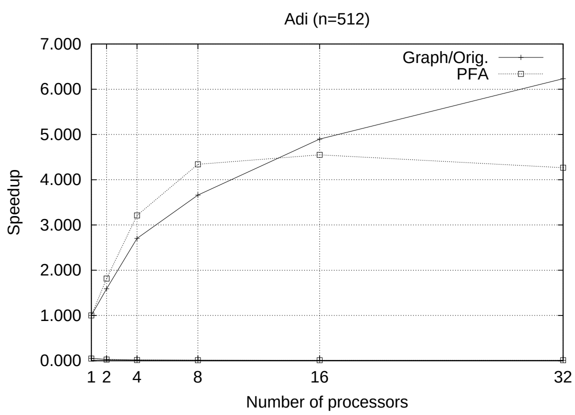


Figure 16: Adi : N = 512

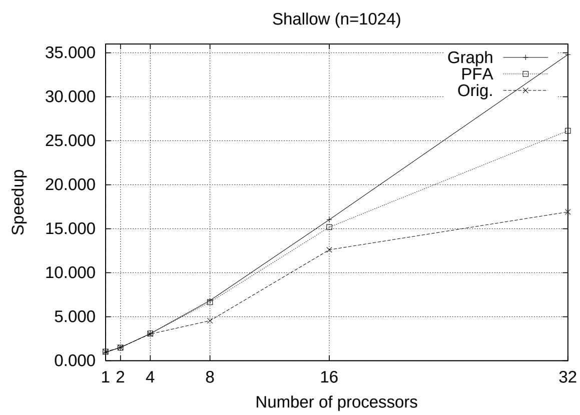


Figure 17: Shallow : N = 1024

The relative improvement, 54%, is greater than that found in `Tomcatv` due to the smaller granularity of work. PFA performs poorly due to conditional parallelisation. Rather than parallelise a loop and incur the synchronisation overhead, at runtime it decides to run it sequentially.

Mgrid Here both approaches incur a large number of runtime synchronisations. However, the graph algorithm executes over three times fewer barriers as the original algorithm. This improves the speedup obtained by a factor of three. PFA once again performs poorly, this time due to excessive synchronisation in the border communication routine.

Erhs Erhs is the largest subroutine within the spec 95 program Applu. Although the graph algorithm reduces the number of barriers executed by 75% and gives a factor of 4 improvement in execution time, this is overshadowed by the absolute poor performance of both

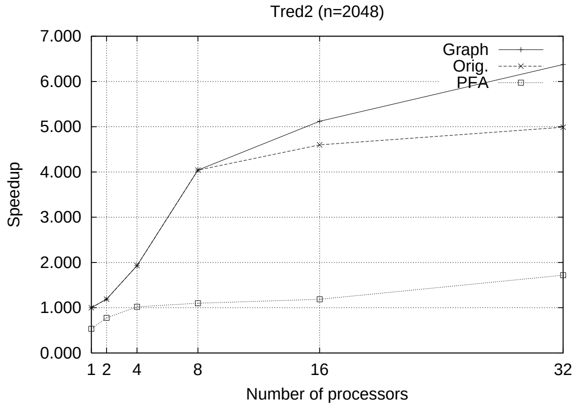


Figure 18: Tred2: N = 2048

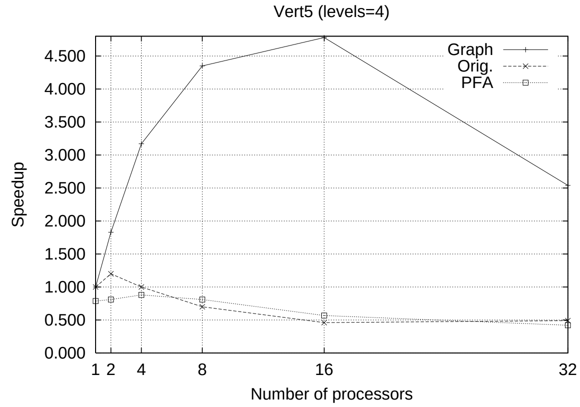


Figure 19: Vert5: Levels = 4

algorithms giving a slow down on 32 processors. This is confirmed when compared to the performance of PFA. The reason for the discrepancy is that both the original and graph algorithm place $O(N^2)$ barriers while PFA places just 6. Currently, MARS does not implement full array privatisation, unlike PFA, and consequently is unable to detect the three main outer loops as parallel. When this optimisation is performed by hand, both algorithms place the same number of barriers and achieve the same performance as PFA.

Vptst The graph algorithm drastically reduces the number of barriers executed giving a dramatic performance improvement. The initial poor performance of the original algorithm is largely due to the placing of a barrier within the outer loop of the innermost 2d loop nest. The graph algorithm gains much of its improvement due to placing the necessary synchronisation at a much higher scope. Our graph algorithm gives a 45% improvement over PFA for 32 processors due to PFA executing ten times as many barriers. PFA performs well compared to the original algorithm as it employs combined global index reordering and loop interchange to place barriers at a higher loop level.

Btrtst Once again our graph algorithm dramatically outperforms the original synchronisation placement scheme. The performance improvement is due to placing barriers at a higher scope as the graph algorithm places the synchronisation outside the main computation subroutine. PFA achieves modest performance gains but slows down after 16 processors.

Chotst Once again our graph algorithm places few barriers in the parallelised code which is translated into just 23 executed calls at runtime. It is a factor of 30 times faster than the

Program	Original	Cross-Processor	After Elimination	% Deps. Remain.
Tomcatv	214	78	25	11.6%
Swm	507	121	46	9.07%
Mgrid	665	605	24	3.6%
Erhs	2787	1700	272	0.59%
Vptst	507	37	3	0.59%
Btrtst	496	28	6	1.2%
Chotst	154	28	6	3.8%
Adi	27	0	0	0 %
shallow	1215	468	73	6%
tred2	505	174	66	13%
vert5	72	35	17	23 %

Figure 20: Data dependences at each stage

naive algorithm and a factor of 90 times faster than PFA, both due to excessive synchronisation.

Adi Here the graph algorithm placed the same number of barriers as the original scheme. In the case of 32 processors our algorithms are both 32% faster than PFA.

Shallow Shallow has very little communication which gives rise to super linear speedup (due to increased cache) for the graph algorithm shown in Figure 17. PFA performs well but the graph algorithm still gives a 33% improvement.

Tred2 Although our graph algorithm clearly gives a good relative performance over the original placement scheme, in absolute terms the executed code is inefficient. The main reason for this inefficiency is due to load imbalance and the relative large amount of communication, $O(N^2)$, relative to computation, $O(N^3)$. As N increases, the relative impact of communication will decrease. PFA performs poorly due to excessive synchronisation.

Vert5 Once again our graph algorithm inserts fewer barriers than the original and outperforms it across all processor sizes. PFA and the original scheme have a similar performance for large processor sizes, where the graph algorithm gives a factor of 4 improvement over both schemes.

8.2 Algorithm Evaluation

Although our approach is, in certain cases optimal, it relies on the use of extensive dependence elimination and specialised algorithms to reduce the cost of finding the best barrier placement. The table shown in Figure 20 shows the number of dependences considered at various stages in the optimisation process. By correctly determining cross-processor dependences [21], we can remove on average 55% of the dependences within the program. The elimination procedures, described in Sections 4, 5 and 6, further reduces this by 83% on average. The final column of Figure 20 shows the percentage of original dependences that need be considered for barrier placement; which is just 7% on average. The overall compilation time overhead for all stages of 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.

8.3 Summary and comparison

Overall, the graph based algorithm improves the average speedup from 4.24 to 12.3 when compared to the original algorithm for 32 processors and thus has a significant impact on execution time. Furthermore, the graph algorithm is capable of achieving good speedup in 10 out of 11 cases. In the one case, where no performance gain was found (`Erhs`), this was due to poor array privatisation rather than barrier synchronisation optimisation.

It is interesting to compare this algorithm with other approaches. Overall, PFA has the average speedup of 6.47 over 32 processors and in all bar one case it is outperformed by the graph algorithm. Although PFA has a completely different parallelisation strategy, it provides a useful comparison in determining the overall impact of a good barrier placement algorithm.

A more accurate comparison can be made with Tseng's barrier placement scheme, described in the context of distributed shared memory (DSM) in [12], as our original algorithm is similar to their approach. Although they are able to reduce the number of barriers inserted for anti-dependences due to the lazy release consistency available in their system [3], more

[3]which is not available on the SGI Origin

barriers are placed overall. They include two Spec codes in their evaluation and in the case of Tomcatv, for instance, they execute twice as many barriers, 81 compared to our 42. Similarly, in the case of Swm their technique executes 245 barriers compared to the 181 executed by our scheme. The speedups obtained on 16 processors are also more modest, probably due to the the cost of the underlying DSM.

9 Conclusions

Reducing synchronisation overhead in parallel programs is essential for scalable performance. This paper has presented an algorithm that provably places the minimal number of barriers synchronisations in 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 computational requirements of the scheme. This algorithm has been incorporated into a parallelising compiler and shown to give significant performance improvement.

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.

Appendix A

Let ℓ be the number of essential arcs. To show that we have found the minimal number of barriers, first note that if we can kill all dependences with ℓ barriers, then, by Theorem 1, this number is minimal.

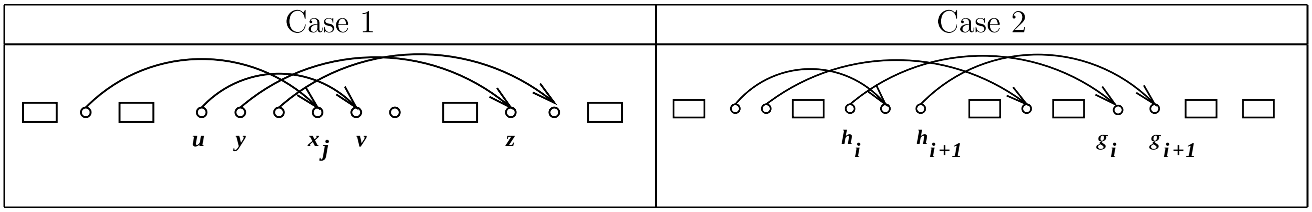


Figure 21: Barrier placement in imperfect loop nests

If we consider how barriers should be inserted inside the loop body then clearly, the first temporary barrier should be inserted between q_1 and t_1, otherwise we will have $\ell+1$ barriers inside the loop body. This barrier may be inserted before t_1 or x_i for some i, $1 \leq i \leq s$. To kill all the remaining forward dependences other barriers should be inserted according to the FFS algorithm. Note that in this way we will also kill the maximum number of backward dependences. If we can not kill all dependences with ℓ barriers, then a barrier should be inserted after $do(i)$ (or before $enddo(i)$) for some i, $i = 1, \ldots, m$. Barriers will be executed the minimal number of times if i is minimal. Thus, if the maximum level of the remaining backward dependences is p, then the cheapest way to kill these dependences is to put a barrier immediately after $do(p)$. If we have considered all the cases where we can place the first temporary barrier within the loop body and have found the minimum of the corresponding maximum levels, then we have found the minimum number of times the barriers will be executed and hence the minimal barrier placement.

Appendix B

First we prove that the algorithm for forward dependences inserts the minimum number of executed barriers. Suppose that we have found a placement of barriers which will be executed the minimum number of times.

Consider the case when there are dependences entirely within the innermost loop (case 1). From all these dependences we choose a dependence $u \longrightarrow v$ with the earliest sink. To kill this dependence there should be a barrier β between u and v.

Suppose that the barrier β is placed after a source of some dependence $y \longrightarrow z$, $u < y < v$.

Let x_j be the earliet sink of a dependence such that $y < x_j \leq v$. Clearly, β would not kill other dependences than a barrier inserted before x_j. Moreover, β will be executed the same number of times as a barrier inserted before x_j. (See figure 21). Thus we can assume that β is inserted before some sink x_j.

Now, if we insert a barrier β just before x_j and remove all dependences killed by β, then it is easy to see that the cheapest way to kill dependences after β is to apply the FFS algorithm to these arcs. The cheapest way to kill dependences before β is to apply the reverse FFS algorithm to these arcs. Thus even if a placement of barriers obtained according to our algorithm may be different from the initial minimal barrier placement, the barriers will be executed the same number of times.

Since we have considered all possible placements of the first temporary barrier between $u \longrightarrow v$ and after applying the FFS algorithm and the reverse FFS algorithm have chosen the placement where the barriers will be executed the minimum number of times, we have found the minimal barrier placement in the corresponding loop.

Now we consider the case when there are no dependences with both a source and sink inside the innermost loop (case 2). Let $R = \{h_i \longrightarrow g_i | h_i < h_{i+1}, i = 1, \ldots, s\}$ be the set of long dependences after dependence elimination. First suppose that there is at least one barrier between h_1 and $do(m)$. We want to see where the last barrier β before $do(m)$ is placed. Suppose that β is placed between h_i and h_{i+1} for some $i = 1, \ldots, s-1$, (or between h_s and $do(m)$.) The barrier β would not kill other dependences than a barrier placed immediately after h_i, since there are no long dependences with sources between h_i and h_{i+1} and after the elimination all short dependences with sinks between h_i and $enddo(m)$ have their sources before h_i. (See figure 21). A barrier inserted just after h_i will not be executed more times than β, since this barrier will not be in a deeper loop nest than β. Thus we can assume that the barrier β is placed immediately after h_i for some i.

Since β is the last barrier before $do(m)$, the cheapest way to kill the dependence $h_{i+1} \longrightarrow g_{i+1}$ is to place a barrier before g_{i+1}. Note that a barrier before g_{i+1} will at the same time kill all short dependences that may be killed by any barrier placed between $enddo(m)$ and g_{i+1}, since all short dependences with sources between $do(m)$ and g_{i+1} have their sinks after

g_{i+1}. Obviously, the cheapest way to kill remaining short dependences is to apply the FFS algorithm to dependences after g_{i+1} and the reverse FFS algorithm to dependences before h_i.

If there are no barriers between h_1 and $do(m)$, then a long dependence with the earliest sink g can be killed only by a barrier placed between $do(m)$ and g. In this case the cheapest way to kill all dependences is to place a barrier before g and then apply the FFS algorithm to remaining dependences after g and the reverse FFS algorithm to remaining dependences before g.

It remains to consider the case when there are backward dependences in the loop. Suppose we have found the placement of barriers which will be executed the minimum number of times. Then some backward dependences at level k are killed by barriers placed between $do(k)$ and their sinks and other by barriers placed between their sources and $enddo(k)$. Let $u \longrightarrow v$ be the level-k backward dependence with the earliest sink which is killed by a barrier placed between $do(k)$ and v. Then all level-k backward dependences with sinks after v will be also killed by this barrier and need not to be considered. Backward dependences with sinks before v should be killed by barriers placed between their sources and $enddo(k)$, since $u \longrightarrow v$ is the level-k dependence with the earliest sink which is killed by a barrier placed between $do(k)$ and v. Thus at most one backward dependence at every level should be replaced by an arc with the source in $do(k)$, $k = 1, \ldots, m$. Since we have proven that the algorithm for forward dependences placed the minimum number of barriers and considered all possible choices of sinks of backward dependences, minimum number of barriers will be also achieved in this case.

References

[1] J.R. Allen, D. Callahan and K. Kennedy, “Automatic Decomposition of Scientific Programs for Parallel Execution”, *Proc. Principles of Programming Languages*, January 1987.

[2] A.V. Aho and J.D. Ullman, “Foundations of Computer Science”, W.H. Freeman and

Company, 1995.

[3] B. Appelbe, S. Doddapaneni and C. Hardnett, "A New Algorithm for Global Optimization for Parallelism and Locality", *7th International Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[4] F. Bodin, L. Kervella and T. Priol, "Fortran-S: A Fortran Interface for Shared Virtual Memory Architecture", *Proc. Supercomputing*, November 1993.

[5] 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*, May 1995.

[6] D. Callahan, "A Global Approach to Detection of Parallelism", Ph.D. Dissertation, Dep. of Computer Science, Rice University, 1987.

[7] D.-K. Chen, "Compiler Optimizations for Parallel Loops with Fine-Grained Synchronization", Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1994.

[8] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka and C.-W. Tseng, "Unified Compilation of Fortran 77D and 90D", *ACM Letters on Programming Languages and Systems*, Vol 2 No 1-4, March-December 1993.

[9] R. Cytron, "Compile-Time Scheduling and Optimizations for Asynchronous Machines", Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1984.

[10] Feautrier P., "Data flow Analysis of Array and Scalar References", Int. Journal of Parallel Prog., 20 (1), 1991.

[11] M. Gupta and E. Schonberg, "Static Analysis to Reduce Synchronization Costs in Data-Parallel Programs", *Proc. Principles of Programming Languages*, January 1996.

[12] H. Han, Chau-Wen Tseng and P. Keleher, "Eliminating Barrier Synchronization for Compiler-Parallelized Codes on Software DSMs", *International Journal of Parallel Programming*, Vol 26(5):pp 591-612, October 1998.

[13] K. Kennedy and K.S. McKinley, "Optimizing for Parallelism and Data Locality", *Proc. International Conference on Supercomputing*, July 1992.

[14] Z. Li, "Compiler Algorithms for Event Variable Synchronization", *Proc. International Conference on Supercomputing*, June 1991.

[15] Z. Li and W. Abu-Sufah, "On Reducing Data Synchronization in Multiprocessed Loops", *IEEE Trans. on Computers*, C-36(1), pp. 105-109, Jan. 1987.

[16] A. Lim and M. Lam, "An Affine Transformation Algorithm to Maximize Parallelism", *Proc. Principles of Programming Languages*, January 1997.

[17] K.S McKinley, "Evaluating Automatic Parallelization for Efficient Execution on Shared-Memory Multiprocessors", *Proc. International Conference on Supercomputing*, July 1994.

[18] 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.

[19] S.P. Midkiff and D.A. Padua, "Compiler Generated Synchronizations for Do Loops", *Proceedings of International Conference on Parallel Processing*, IEEE press, Aug. 1986.

[20] S.P. Midkiff and D.A. Padua, "Compiler Algorithms for Synchronization", *IEEE Trans. Comput.*, C-36(12), pp. 1485-1495, Dec. 87.

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

[22] M.F.P. O'Boyle and F. Bodin, "Compiler Reduction of Synchronisation in Shared Virtual Memory Systems", *Proc. International Conference on Supercomputing*, July 1995.

[23] M. Philippsen and E. Heinz, "Automatic Synchronization Elimination in Synchronous FORALLs", *Proc. The 5th Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[24] D. Prakash, M. Dhagat and R. Bargodia, "Synchronization Issues in Data-parallel Languages", *Proc. International Workshop on Languages and Compiler for Parallel Computing*, pp. 76-95, Aug. 1993.

[25] W. Pugh and D. Wonnacott, "Eliminating False Data Dependences using the Omega Test", *Proc. Programming Languages Design and Implementation*, San Francisco, ACM Press, June 1992.

[26] M. Quinn, P. Hatcher and B. Seevers, "Implementing a Data Darallel Language on a Tightly Coupled Multiprocessors", *Advances in Languages and Compilers for Parallel Processing*, pp.385-401, Pitman, London, 1991.

[27] E.A. Stöhr and M.F.P. O'Boyle, "Barrier Synchronisation Reduction", *Proc. HPCN'97*, April 1997.

[28] E.A. Stöhr and M.F.P. O'Boyle, "A Graph Based Approach to Barrier Synchronisation Minimisation", *Proc. International Conference on Supercomputing*, July 1997.

[29] E.A. Stöhr and M.F.P. O'Boyle, "First Fast Sink: A Compiler Algorithm for Barrier Placement Optimisation", *Future Generation Computer Systems*, 13, pp. 397-406. 1998.

[30] P. Tang, P. Yew and C. Zhu, "Compiler Techniques for Data Synchronisation in Nested Parallel Loops", *Proc. the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[31] P. Tang P. and J.N Zigman, "Reducing Data Communication Overhead for Doacross Loop Nests", *Proc. International Conference on Supercomputing*, Manchester, July 1994.

[32] C.-W. Tseng, "Compiler Optimizations for Eliminating Barrier Synchronization", *Proc. Principles and Practice of Parallel Programming* , Aug. 1995.

[33] M.E. Wolf and M. Lam, "A Data Locality Optimizing Algorithm", *Proc. Programming Language Design and Implementation*, Toronto, June 1991.

[34] M.J. Wolfe, "Multiprocessor Synchronization for Concurrent Loops", *IEEE Software*, pp. 34-42, January 1988.

[35] H. Zima, "Supercompilers for Parallel and Vector Computers", ACM Press, 1991

MICHAEL O'BOYLE received a PhD. in Computer Science from the University of Manchester in 1992. He was formerly a SERC Postdoctoral Research Fellow, a Visiting Research Scientist at IRISA, Rennes, a Visiting Research Fellow at the University of Vienna and a Visiting Scholar at Stanford University. He is currently an EPSRC Advanced Research Fellow, a lecturer in the Institute for Computing Systems Architecture (ICSA) at the University of Edinburgh, where he the leader of the Compiler group, and an honorary lecturer at the University of Manchester. His main research interests are in adaptive and iterative compilation, program transformation theory, embedded systems and automatic compilation for distributed shared memory architectures.

ELENA STOHR studied Mathematics at Moscow State University. She received the Dr. rer. nat. (PhD) in Mathematics from the Academy of Sciences in Berlin and the Dr. sc. in Computer Science from the University of Jena. Her main research interests are compilation for distributed shared memory architectures, compilation techniques for Very Long Instruction Word architectures, optimisation techniques for RISC processors and compiler directed memory coherence. Her research interests also included Group Theory, graph embeddings, communication in networks and applications of Graph Theory in Computer Science. Elena A. Stohr has published a number of papers in the areas of algebra and Computer Science.