Synchronization Minimization in a SPMD Execution Model

M.F.P. O'Boyle
Centre for Novel Computing
Department of Computer Science
University of Manchester
Manchester M13 9PL

L. Kervella
IRISA-INRIA
Campus de Beaulieu
35042 Rennes Cedex

F. Bodin
IRISA-INRIA
Campus de Beaulieu
35042 Rennes Cedex

Synchronization Minimization in a SPMD Execution Model

M.F.P. O'Boyle
Centre for Novel Computing
Department of Computer Science
University of Manchester
Manchester M13 9PL

Abstract

This paper presents an algorithm for synchronization placement when using a SPMD execution model, where synchronizations are enforced only when there exists a cross-processor data dependence. In this paper we investigate two scheduling techniques, loop-based and data-based, both of which use a SPMD model. Using scheduling information from previous stages in the compilation process, a new technique to determine potential cross-processor is presented. Given the minimum number of cross-processor data dependences that must be satisfied, a new optimization is used so as to minimize the number of synchronization points needed to satisfy them. This algorithm has been successfully implemented in an experimental compiler. Initial experimental data show this technique to be very effective, outperforming existing methods.

List of Symbols

δ : delta

1 Introduction

A popular method of exploiting parallelism, particularly on distributed memory machines, is the single program multiple data (SPMD [13]) execution model where the array data and loops are distributed across the processors so that each processor works upon a separate section of the array in parallel. Each process executes the same program but operates on different data which are independent portions of an array.

A SPMD model has only one fork at the beginning and one join at the end of a program, but all synchronization points have to be explicitly specified. Compiler placement of synchronization points has to be addressed for efficient execution. Synchronization should only be enforced when there exists a cross-processor data dependence (a dependence whose source and sink are executed on different processors), but synchronization placement requires partitioning and scheduling information from previous stages in the compilation process. Traditionally, shared memory parallel compilers have used a fork & join model to exploit program parallelism. Each parallel DO implied a fork and each ENDDO a join, both of which are expensive operations. Any remaining synchronization was handled by a variety of methods including barriers [44].

Although there has been much research into the minimization of overheads such as communication and load imbalance when using a SPMD model [4], little work has addressed the issue of synchronization overhead. Most SPMD compilers use barrier synchronization which on some machines, such as the KSR-1, is very expensive [30]. However, even if synchronization took zero time, it can increase the idle time of processors and therefore even on machines with fast synchronization such as the Cray T3D, reducing the amount of synchronization is important. Indeed, barrier removal for single address computing has been considered important enough by others, to the extent that pragmas have been added to the programming language to tell the compiler when barriers are unnecessary [32, 6].

In this paper we are concerned with reducing synchronization in single address space computers using a SPMD model of computation. We are interested primarily in distributed-memory computers such as the CRAY T3D, Meiko CS-2, and the KSR-1, but the techniques described in this paper could be applied to symmetric multiprocessors such as the SGI Power Challenge.

1.1 Memory Consistency

Distributed memory machines with a single address space share characteristics with message passing machines and traditional shared memory machines. From a compiler's point of view, memory is either local or non local to a processor and locality exploitation is essential. However, communication is not by message passing but by reading and writing to a single address space common to all processors. This type of machine holds the promise of potential scalability without the difficulty of low level message passing.

Memory consistency in message passing implementations is ensured by only sending and receiving up-to-date values. This may explain the lack of attention given to synchronization removal when using a SPMD model. In single address space computing, however, consistency requires synchronization. Ensuring that a write/read sequence is correctly executed can be done in two ways. If the reader and writer processes (or threads) are scheduled on two different processors then inserting a synchronization between the two ensures correct execution. If, however, the same process (thread) writes and reads the data then the normal sequential execution of the processor will ensure that the correct data are accessed. Using compile time knowledge it is possible to determine if a data dependence can be solely satisfied by sequencing.

Two variants on distributed memory single address space architectures can be found : *distributed shared memory* such as the Cray T3D and Meiko CS-2, or *shared virtual memory* such as the KSR-1 or KOAN system [6]. In *distributed shared memory* machines data allocation is static (or if dynamic, compiler controlled) and is based on the assumption that runtime data allocation is expensive and should be avoided unless there is a clear gain in doing so. This is not true in the case of *shared virtual memory* systems, where there is system support for dynamic re-allocation (or data migration) at the expense of increased system complexity or cost. Exploitation of locality and efficient synchronization is critical to good performance on these machines. Much work has been done on locality [19, 4], but little recent work has focussed on reducing synchronization overhead.

1.2 Partitioning and Scheduling

Data partitioning is generally used for *distributed shared memory* machines (and indeed message passing machines) and forms the basis for the language extensions to Fortran such as HPF, Fortran D [15] and Vienna Fortran [5]. This method can also be used on shared virtual memory machines [29]. Static data layout, alignment and partitioning information can be used to determine the source and sink of data dependences and thus reduce synchronization.

Loop scheduling is frequently used on *shared virtual memory* machines and is similar to the scheduling techniques used for traditional shared memory machines. However, because of the impact of non local memory references, data movement must be considered when scheduling the loops. Loop scheduling information is used to determine the source and sink of data dependences and again reduce synchronization.

The algorithm given in this paper is applicable to all single address machines using a SPMD model whether loop scheduling or data partitioning methods or a combination of the two are used for a particular program. The next section motivates the paper and describes two scheduling methods commonly used on distributed memory machines; a survey of related work is also given. Section 3 describes new algorithms to determine cross-processor data dependences and the placement of barriers to enforce them. Section 4 describes additional optimization techniques to reduce synchronization overhead in the presence of sequential loops. Section 5 presents experiments to illustrate this algorithm, implemented in our compiler. This is then followed by an outline of future work and a summary of the paper.

2 Approach

To provide motivation for this paper, consider the program fragment in figure 1. This is part of a 450 lines FORTRAN program implementing the shallow water equations [26, 18] (and is further described in the experiment section 5.1) . If a fork & join implementation were used, the amount of synchronization would depend on which parts of the program were parallelized. Assuming that each outer loop is parallelized and synchronization is by way of barriers, there will be at least 16 barriers needed, one before and one after each loop nest. If the inner loops were parallelized, the number of synchronization would increase to $6n + 10$, where n is the number of iterations covered by the outer loops.

If a SPMD model were used, which placed synchronization only where there exists cross-processor data dependence, the maximum number of barriers needed is one. By using exact array data flow information [37], it can be shown that there is no output dependence present, as each write is to a different part of an array.

The only data dependences are flow dependences on u between s1 and s9, s2 and s9, s3 and s10, and s4 and s10. These four data dependences can be satisfied by placing a barrier between the last assignment to u in statement s4 and the first assignment to *capu* (s9) which covers or "cuts" each data dependence. However, if loops or data are appropriately aligned, the sources and sinks of these data dependences will be within the same processor, and synchronization will be unnecessary. Thus, by using accurate array data flow information, a SPMD model and scheduling information, the number of synchronization points can be reduced to zero.

```
    do  j=1,n-1
      do  i=1,n-1
s1:      u(i+1,j) = -(psi(i+1,j+1)-psi(i+1,j))/space
      enddo
    enddo
    do i=1,n-1
s2:    u(i+1,n) = -(psi(i+1,1)-psi(i+1,n))/space
    enddo
    do  j=1,n-1
s3:   u(1,j) = -(psi(1,j+1)-psi(1,j))/space
    enddo
s4: u(1,n) = -(psi(1,1)-psi(1,n))/space

    do  j=1,n-1
      do i=1,n-1
s5:      v(i,j+1) = (psi(i+1,j+1)-psi(i,j+1))/space
      enddo
    enddo
    do  i=1,n-1
s6:    v(i,1) = (psi(i+1,1)-psi(i,1))/space
    enddo
    do  j=1,n-1
s7:    v(n,j+1) = (psi(1,j+1)-psi(n,j+1))/space
    enddo
s8: v(n,1) = (psi(1,1)-psi(n,1))/space

    do  j=1,n
      do  i=1,n-1
s9:      capu(i+1,j) = 0.5 * (p(i+1,j)+p(i,j)) * u(i+1,j)
      enddo
    enddo
    do  j=1,n
s10:   capu(1,j) = 0.5 * (p(1,j)+p(n,j)) * u(1,j)
    enddo
```

Figure 1: Shallow Extract.

To determine the processor(s) containing the source and sink of a data dependence it is necessary to know how work and data. The following paragraphs briefly describe two scheduling methods and how they influence synchronization placement.

Loop Scheduling In this approach, the focus is on the scheduling of statement instances to processors. Within loops, this form of scheduling maps loop iterations to processors and is a well-known approach in parallelizing for shared memory machines [31]. How loop iterations are mapped is largely decided by previous stages; either by automatic parallelization or by user directives. We assume each loop is either marked as *parallel* or not. Parallel loops are scheduled such that each processor executes a distinct set of iterations. As parallel loops have no cross-iteration dependences, scheduling distinct set of iterations to each processor implies that there will be no cross-processor data dependence; hence there is no need for synchronization. This fact is well-known and forms the basis of parallelization in the fork & join model. The iterations of the remaining loops are either executed by every processor or by just one, depending on whether enclosed loops are parallel [44].

Data Partitioning In contrast to loop scheduling, data partitioning is concerned with the mapping of array elements to processors. The data partition and mapping of such elements is determined by previous stages in the compilation process. Again this can be either done by compiler methods [29] or by directives [15]. All arrays are assumed to be distributed or partitioned across the processors while scalars are assumed to be replicated within our approach. All arrays are aligned to a common array (or template [5]) which is then partitioned over the processors.

Any given element of two or more arrays is scheduled to the same processor, if they are aligned in that dimension. Thus element a(3,40) will always be scheduled to the same processor as b(3,40), c(3,40), etc, if they have the same alignment. For further discussion of alignment see [21, 28]. Knowledge of the data partition provides accurate information as to the location of the source and sink of a data dependence. In particular, using the "owner-computes model" [43] implies that all write accesses to an array are to local data.

2.1 Loop Scheduling versus Data Partitioning

Although the above loop scheduling and data partitioning techniques come from two quite distinct traditions, they are strongly related from a synchronization point of view. Both schemes can help remove synchronization if the source and sink of a data dependence are scheduled to the same processor. Loop scheduling determines the local iteration space from which data accesses can be determined. Data partitioning, in contrast, determines the local data to be accessed which implies the local iteration space. To illustrate the differences consider the two programs in figures 2.

```
     Dimension a(64),b(64)
     Do i = 1, 63
S1:    a(i+1) = x
S2:    b(i) = a(i+1)
     Enddo
```

(a)

```
     Dimension a(64),b(64)
     Do i = 1,63
S1:    a(i+1) = x
S2:    b(64-i) = a(64-i)
     Enddo
```

(b)

Figure 2: Parallelism with Loop Scheduling (a), Data Partitioning (b)

The first program can be executed in parallel if a loop-based scheme were used, but must be executed sequentially if a data-based approach is used. The converse is true for the second program. With the loop based approach, each processor executes a distinct set of loop iterations (known as the local iteration space) covering each statement within the loop body. Consider the case with 4 processors, and in particular what occurs on processor 2. In both programs, the local iteration spaces associated with S1 and S2 are the same. For processor 2, the iteration space for S1 and S2 is i : 17..32. In the first program, 2.a, there is no cross-processor data dependence as there is no loop carried data dependence. For example, when $i = 32$ $a(33)$ will be written by processor 2 and read by processor 2. In the second program, 2.b, there is a loop carried data dependence and hence synchronization. In particular processor 2 will write $a(17..32)$ but read elements $a(32..47)$, which are written by processor 3. In contrast with the data-based approach, classical data dependence determines whether synchronization is needed. Typically in the second case, however, if no parallelism is available, only one process will execute the loop, reducing the amount of synchronization but incurring data movement.

We now consider data scheduling and again the case with 4 processors, and in particular what occurs on processor 2. We assume that a and b are aligned and block partitioned such that array elements 17..32 of arrays a and b will be scheduled to processor 2. Cross-processor data dependence will occur in the first program, 2.a, at the border elements of the arrays mapped to each processor. The iteration space associated with S1 is i : 16..31 and with S2 i : 17..32 to ensure the owner-compute/local-write rule. To compute $b(32)$, $a(33)$ will be read but this is allocated on processor 3. Furthermore $a(33)$ must be written before $b(32)$ is read, and hence there is a cross-processor data dependence requiring synchronization. In the second program, 2.b, array elements 17..32 of a and b will still be scheduled to processor 2, but the iteration spaces associated with each statement are quite different. The iteration space associated with S1 is still i : 16..31 but with S2 i : 32..47 to ensure the owner-compute/local-write rule. To compute $b(17..32)$ the corresponding elements of a will be accessed, all of which are local, and hence there is no cross-processor data dependence or synchronization.

The crucial point is how data are mapped to processors, rather than the data dependence distance. For example, in figure 2, both a and b are acceded with the same reference pattern. As a and b are mapped identically to processors, there are only local memory references. A data dependence is guaranteed to be local if the read access has the same memory reference as the write in that statement, regardless of the data dependence type or distance. This can be relaxed in the case of 2 or more dimension arrays, where only the partitioned dimensions have to have the same memory reference.

2.2 Related Work

Most previous techniques do not make any assumption on the partitioning of the iteration space or data space across processors. Furthermore, a fork & join model is generally assumed [12, 2, 44, 23, 25]. Some researchers, within this framework, have used kill analysis to remove covered data dependences, particularly in Doacross loops, so as to reduce the number of synchronizations. Midkiff et al. [23, 25] have addressed the problem of removing redundant data dependence to limit the number of synchronization points. A data dependence is said to be redundant if it is *covered* by another synchronized dependence. Within loops, assuming there is only constant dependence distances and no conditional branches, the problem of finding a minimum set of data dependences has been shown to be NP-Hard[24].

Program transformations to eliminate or decrease the cost of synchronizations inside parallel and doacross loops have also been proposed [12, 2, 44, 38, 23, 25]. The program transformations used are loop distribution, loop fusion, and loop alignment. Loop distribution is used to transform a synchronization within one loop nest to a synchronisation between two loop nests. Loop fusion is used to increase granularity and to reduce parallel loop creation overhead (cf. section 4.1). Loop alignment can convert loop carried dependences into a loop independent dependence. To extend the loop alignment applicability it can also be coupled with statement replications; the problem is then NP-Hard when minimizing the amount of replication code [2].

Quinn et al. [38] have proposed a greedy algorithm to minimize the number of synchronization in a basic block. For a set of intervals defined by the sink and source of the data dependences, the algorithm is based on finding the minimum number of points such that every interval spans at least one point. This problem was shown to be polynomial [17]. The global algorithm from Quinn et al. [38] is basic block orientated. It adds synchronizations to loops by going outward from the innermost loop and then by using the basic block algorithm to treat the remaining data dependences.

More recent work has addressed the problem of synchronization insertion for data parallel languages. Prakash et al. [36] have studied how previous techniques could be targeted to the UC data parallel language. They also have proposed runtime methods, such as fuzzy barriers, to reduce the cost of synchronization. The work done by Philippsen et al. [33] addresses the problem of inserting synchronizations in FORALL statements. The method applies at the expression level rather than at the statement level. The main contribution of the method, compare to Quinn et al. is to minimize the number of temporary variables needed to implement efficiently FORALL statements.

The work presented in this paper goes beyond existing research by exploiting the properties of the SPMD execution model. Using compile time knowledge of the program's partition, a new technique to determine data dependences that do not require synchronization has been developed. Furthermore, by using a new method synchronization is placed to minimize runtime overhead. In the next section the intuitive ideas described above will be expanded upon and formalized to give an algorithm which helps reduce the number of synchronizations in a program when loop scheduling or data partitioning is used.

3 Algorithm

The overall compiler algorithm, to reduce synchronization, can be expressed in a simplified manner as follows:

1. Form complete data dependence graph of the program
2. Compute array/loop partition and schedule.
3. Mark as satisfied all data dependences known not to cross a processor boundary based on scheduling information.
4. Use a barrier insertion algorithm to determine the best insert synchronization points and kill covered data dependences.

Step 1 is a basic requirement of much compiler analysis. We require accurate data dependence information to reduce the number of spurious data dependences to be considered [1] and control flow information is needed by step 4 to accurately place barriers. Step 2 is required to determine the location of the source and sink of data dependences. The exact form of the data/loop partition/schedule depends on many criteria including parallelism, locality and load balance as well as synchronization. The determination of the partition/schedule is, however, beyond the scope of this paper. [2]

Step 3 describes the first contribution of this paper. By using information from steps 1 and 2 data dependences can be marked as satisfied if they are known not to cross processor boundaries. Step 4 describes the second contribution of this paper. Once all synchronization inducing data dependences are determined, the minimum number of barriers should be inserted so as to satisfy them. The number of barriers incurred at run-time will depend on the control-flow of the program. In particular, placing barriers inside loops should be avoided.

As they are beyond the scope of this paper, steps 1 and 2 are not further described in this section. After some initial definitions, section 3.1 describes a technique to determine whether a data dependence requires synchronization, while section 3.2. describes the placement of barriers to satisfy those data dependences.

3.1 Determining Dependences That Do Not Require Synchronization

Determining the data dependences that cause cross-processor synchronization depends on how the program is mapped to the machine. If the source and the sink are mapped to processors x and y then if $x \neq y$ synchronization is needed. If $x = y$ then the data dependence is satisfied by sequencing. Determining whether a piece of data is accessed by a unique processor is relatively straightforward *within* a loop nest as illustrated in section 2; across whole programs, the process is more complex. In the most general case, for each processor it is necessary to determine the array sections of the source and sink and determine their equality which is undecidable [14]. Fortunately, in many cases, simpler and much less expensive methods can be used.

If a data dependence involves a private variable, then both accesses are local by definition and hence there is no cross-processor data dependence requiring synchronization. It is assumed that all scalars are privatized, so the remainder of this section is concerned with access to shared arrays.

Initially let $\{Z\}$ denote the set of values ranged over by variable Z. The loops

[1] At present we use the Omega Test [37]

[2] Our compiler uses an automatic data partitioning scheme as described in [29]

in a FORTRAN program surrounding any statement can be represented as a vector of size m, where m is the number of loops or iterators.

Definition 1

$$J = [j_1, j_2, \ldots, j_m]^T \tag{1}$$

are the iterators *surrounding a statement. The values that J ranges over is called the* **iteration space** *denoted by* $\{J\}$

In the example in figure 1, the iterators J surrounding the first assignment statement are as $[j, i]^T$

Consider two data references, `a(` $f(I)$ `)` and `a(`$g(J)$`)`, to a shared array `a`, which form the source and sink of a data dependence. I and J are vectors representing the enclosing iterators and f and g are access functions. Thus, $f(\{I\})$ and $g(\{J\})$ are the set of accessed array elements (or footprint) whose values lie between the lower and upper bounds of array a.[3]

Definition 2 *The* **local iteration space** *of a processor x is the set of iterations of a loop nest J performed by that processor. The local iterators are denoted by the vector $\underline{J_x}$ and the* **local iteration space** *is denoted by* $\{\underline{J_x}\}$

Definition 3 *An array* `a`*, with access* `a(`$f(I)$`)` *in a statement S, is said to be* **uniquely referenced** *if and only if:*

$$f(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) = \emptyset \mid \ \forall x, y \in P, \ x \neq y$$

where P is the set of processors

In other words only processor x accesses array elements $f(\{\underline{I_x}\})$ of array `a` in statement S. Using this property it is possible to determine a criteria for local dependence

Theorem 1 *There is no synchronization required if* `a(`$f(I)$`)` *is* **uniquely referenced** `a(`$g(J)$`)` *is* **uniquely referenced** *and*

$$(f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset) \Rightarrow (x = y) \ \mid \forall x, y \in P$$

In other words, if both source and sink are uniquely referenced and each dependence is within a processor there is no need for synchronisation.

For the purposes of this section, this can be expressed as a stronger sufficient corollary.

Corollary 1 *If* `a(`$f(I)$`)` *is* **uniquely referenced** *by processor x and $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\})$ for all values of $x \in P$ then there is no cross-processor data dependence.*

[3]The functions f, g are not restricted to affine or rational forms .

3.1.1 Loop Scheduling

This section develops two useful theorems which allow the removal of synchronization points when using *affinity* scheduling. When scheduling iterations to processors it is assumed that no iterator is scheduled to more than one processor, i.e.,

$$schedule(I) \Rightarrow \{\underline{I_x}\} \cap \{\underline{I_y}\} = \emptyset \ \mid \ \forall x, y \in P, \ x \neq y \tag{2}$$

Affinity scheduling has the property that it will guarantee the same local iteration space for source and sink if the enclosing iteration spaces are identical and the loops of the same type, i.e., parallel or not. If the two loop are scheduled using affinity then:

$$(\{I\} = \{J\}) \wedge (looptype(I) = looptype(J)) \Rightarrow \{\underline{I_x}\} = \{\underline{J_x}\} \ \mid \ \forall x \in P \tag{3}$$

where *looptype* is either parallel or serial.

The following two theorems are used to prevent synchronizing data dependences in two commonly occurring instances.

Theorem 2 *A data dependence occurring within one Doall loop implies the access functions are one-to-one and there is no need for synchronization.*

For example there is no need for any synchronization in program (a) in figure 2. This is a well known property and is widely used. The access functions must be one-to-one otherwise two distinct loop iterations could refer to the same array element and hence the loop would not be a Doall loop. The next theorem allows data dependences occurring across loops to be examined.

Theorem 3 *If the source and sink of a data dependence occur in different loop nests but the iteration spaces are identical ($\{I\} = \{J\}$), the loops the same type, the accesses functions the same ($f = g$) and one-to-one, then affinity scheduling implies there is no need for synchronization*

For example, in program (a) of figure 3, there is no need for synchronization. Examining the local program (b) of figure 3 after scheduling the loops over 16 processors, we see that the elements written in S1 by processor 2, a(5..8) are read in the second statement. This data dependence is entirely local and does not require any synchronization.

Although the two theorems allow the identification of many internal data dependences, they are conservative in their analysis. The crux of the corollary is showing $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\})$, which in general is undecidable. The equality is certainly true when $f = g$ and $\{\underline{I_x}\} = \{\underline{J_x}\}$, but there are many cases when $f \neq g$

```
    Dimension a(64),b(64)
    Do i = 1,64
S1:   a(i) = x
    Enddo
    Do k = 1,63
S2:   b(k+1) = a(k)
    Enddo
```
(a)

```
    Dimension a(64),b(64)
    Do i = 5,8
S1:   a(i) = x
    Enddo
    Do k = 5,8
S2:   b(k+1) = a(k)
    Enddo
```
(b)

Figure 3: Program with No Synchronization Before and After Loop Scheduling (processor 2).

and $\{\underline{I_x}\} \neq \{\underline{J_x}\}$ but $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\})$ and is compile-time decidable. A useful scheduling technique would be, given two different access functions f and g to an array a, determine the local iteration spaces such that $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\})$.

3.1.2 Data Partitioning

Data partitioning involves the mapping of array elements to processors. In this section we will concentrate on the well known "owner-computes rule" mapping. Other techniques to exploit reduction parallelism [27] and to minimize intra-statement communication in constant communication distance problems [11] can utilize the results of this section.

The owner-computes model implies that all write accesses to an array are to local data. Furthermore, each processor writes to a unique part of an array when executing a particular statement.

$$write(f) \Rightarrow f(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) = \emptyset \ \mid \ \forall x, y \in P, x \neq y \qquad (4)$$

Thus `a(`$f(I)$` )`, is **uniquely referenced**. If, in addition, there is a static data distribution throughout a section of the program, then the same processor will write to the same data elements throughout that section. The following theorem follows trivially:

Theorem 4 *Output data dependences never require synchronization*

For example, consider program (a) in figure 4 and the node program (b) for processor 2 after applying the owner-computes rule. The output dependence from S1 to S2 in figure 4.a can be shown to be local when the partitioned local program is examined in figure 4.b. Before the next theorem can be developed, the following well known property is needed. For further information see [21, 28]

Property 1 *Any given element of two or more arrays is scheduled to the same processor, if they are aligned on a partitioned dimension.*

```
     Dimension a(64),c(64)
     Do i = 1,64
S1:    a(i) = x
     Enddo
     Do k = 1,64
S2:    a(64-k) = c(k)
     Enddo
```
(a)

```
     Dimension a(64),c(64)
     Do i = 5,8
S1:    a(i) = x
     Enddo
     Do k = 56,59
S2:    a(64-k) = c(k)
     Enddo
```
(b)

Figure 4: Program with No Synchronization

Theorem 5 *If a read access to an array* `a(`$f(I)$`)` *on the right hand side of an assignment is aligned on a partitioned index with the write access to array* `b(`$l(I)$`)` *on the left hand side, then all read references to a will be* **uniquely referenced** *and*

$$(l(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) \neq \emptyset) \Rightarrow (x = y) \ \mid \forall x, y \in P$$

In other words aligned partitioned read references are always to local data.

The following theorem is the most useful one when trying to determine local dependences in the case of owner-computes scheduling.

Theorem 6 *If a read array reference is partitioned on aligned indices with the write array reference in a particular statement, no data dependences associated with that read array reference require synchronization*

```
     Dimension a(64),c(64)
     Do i = 1,63
S1:     a(i+1) = x
     Enddo
     Do k = 1,64
S2:     c(64-k) = a(64-k)
     Enddo
```
(a)

```
     Dimension a(64),c(64)
     Do i = 4,7
S1:     a(i+1) = x
     Enddo
     Do k = 56,59
S2:     c(64-k) = a(64-k)
     Enddo
```
(b)

Figure 5: Program with No Synchronization before and After Data Partitioning (processor 2).

This theorem has the important property that all data dependences associated with a read reference can be guaranteed to be local, just by examining the alignment of two arrays within one statement. For example, consider program (a) in figure 5, where a and c are aligned. As the access to a in statement S2 is the same as to c after partitioning, all data dependences to a in program (b) in figure 5 will be local.

Owner-compute scheduling has an asymmetric treatment of data dependences depending on whether the source or sink is a read or write reference. In the case of affinity scheduling, no such asymmetry is present. Loop based scheduling requires the access functions to be one-to-one in order to determine a local data dependence, i.e., two different values of an iterator i access distinct array elements $f(i)$, but the data based approach does not have this requirement. In both loop and data based cases, conservative techniques to determine local data dependence have been used. By using simple tests, it has been possible to avoid calculating the exact array section accessed by each processor, at the expense of accuracy. Future work will investigate if exact array section analysis is beneficial in real programs. Only two compilation schemes have been investigated; future work will also investigate synchronization removal in the presence of other scheduling policies.

3.2 Barrier Insertion

Once we have computed the minimum number of cross-processor data dependences, we need to know where to place synchronization points so as to preserve program semantics but not incur excessive overhead. We use the control flow graph and data dependence information to find the "best" place to insert synchronizations. If considering only a basic block, the problem has been solved by Quinn and Hatcher [38], but this method does not extend to general control flow graphs.

Definition 4 *The Control Flow Graph* (CFG) *is a directed graph* (S, E) *where* S *correspond to basic statements of the program and* E *the set of control relation between two statements. The relation* (S_i, S_j) *refers to an edge in* E *that joins node* S_i *to node* S_j *where* S_i *and* S_j *are adjacent nodes.*

With a general control flow graph, and considering just one data dependence, the problem is similar to a maximum cut algorithm [16]. In our case we have to consider the CFG restricted to all the paths from the source, S_b, to the sink, S_d, of the data dependence. The edges of the restricted CFG can be labeled with the number of times the control flow crosses the edges for a given execution or an approximation of it e.g. loop nest depth. To minimise the execution cost we have to insert barriers such that all the paths from S_b to S_d are cut and the sum of all the labels of the "cut" edges is minimum. This problem, unfortunately, is NP-complete [16] and furthermore, when dealing with more than one data dependence the problem becomes a set of problems.

The algorithm defined below avoids inserting synchronization into loop bodies unless it cannot be avoided. The reason is simple; if a barrier is inserted within a loop then it may be executed n times, where n may be related to the problem size. If a barrier is inserted outside a loop body then it is executed once, so the cost of having unnecessary barriers outside loops is relatively small compared to placing a barrier within a loop body.

Before applying the algorithm we append data dependences to the CFG; if there is a data dependence δ requiring synchronization between two statements S_b and S_d then all edges in all paths in the CFG from S_b and S_d will be marked with the data dependence δ. The problem is then to insert barriers such that any paths from a source to a sink of a data dependence contains at least one barrier. Because barriers are inserted after statements the method looks for a vertex (i.e. a statement) after which the barrier will be inserted. Inserting the barrier means cutting the edge going outward from the vertex. The algorithm proceeds as follow:

1. Determine an edge $E = (S, S')$ to "cut" using the following procedure:

   (a) compute the set of edges marked at least with one data dependence, that are not in a loop,

   (b) if the previous set is not empty, choose the edge with the greater number of data dependence

   (c) if none was found, re-apply the procedure, but allow an extra level of loop

2. Cut the edge $E = (S, S')$ by inserting a barrier ($S^{''}$) after the vertex S.

3. Update the data dependence labels of the CFG, in such fashion that when all the paths from the source to the sink of a data dependence contain a barrier no edges remain labeled with a data dependence. The procedure to update the data dependence labelling is as following:

   Foreach data dependence $\delta\ : S_b \rightarrow S_d$ crossing vertex $S^{''}$ **do**

   (a) Compute the restricted control flow graph, CFG', by restricting CFG to the edges labeled with δ and the corresponding vertices.

   (b) Compute the vertices of CFG' that are dominated and post-dominated [1] by vertex $S^{''}$.

   (c) Remove the data dependence label from those edges whose source is dominated or whose sink is post-dominated by $S^{''}$. For instance, assume that vertex $S^{''}$ dominates vertex S_n and that vertex S_n belongs to a path from S_b to S_d. By inserting a barrier, $S^{''}$, we have ensured that if the control flow goes through the edge whose source is S_n, it was synchronized by the barrier $S^{''}$ because $S^{''}$ dominates S_n. When later barriers are added, the restricted CFG for δ contains only those paths not already having a synchronization. In practice, to avoid excessive dominator and post-dominator computation, for each data dependence we check if all its paths pass through vertex S''. If so the data dependence is removed from all edges.

4. The algorithm is completed when no edges marked with a data dependences remain.

Let m be the number of edges and n be the number of vertices of the control flow graph. For each barrier inserted (potentially n in number) and for each data dependence "cut" (we have at most $nbdep$), we compute a restricted control flow graph at the cost of m operations. On this restricted control flow graph we compute the dominator and post dominator. The complexity of computing the dominator and the post dominator using the simple version of the Lengauer and Tarjan algorithm [20] is bounded by $2 * m * log(n)$. The complexity of the algorithm, not counting the initialization, can be bounded by $O(n * nbdep * m * 2 * m * log(n))$. This bound is much higher than the one achieved in practice. The number of data dependences may be high (a thousand for instance), but the complexity remains small mainly because in many cases we can avoid computing dominators and computation is performed on a restricted graph. Furthermore, as the algorithm proceeds, the number of data dependence to examine decreases and the restricted graph shrinks, both of which reduce the computational complexity.

Figure 6 illustrates the algorithm. The program in figure 6.a has three data dependences (S_1, S_6), (S_3, S_7) and (S_4, S_6) that are represented in the CFG, (6.b), by arrows labeled respectively δ_1 δ_2 and δ_3. The first step of the algorithm consists of choosing an edge. In the example a choice exists because multiple edges are marked with two data dependences. Assume edge (S_3, S_5) is chosen. A barrier is inserted after S_3 (6.c) and the δ_1, δ_2 labels are removed from the edges which are satisfied by the barrier (6.d). Another barrier is inserted after S_4 (6.e). Finally after updating the data dependence labels, none remain(6.f), leaving an unmarked CFG.

4 Optimization

While the barrier insertion algorithm attempts to determine the minimum number of barriers, giving a set of data dependences and scheduling information for a *fixed* program, it may lead to excessive barriers in the case of sequential loops. This section is primarily concerned with well-known transformations that are used to eliminate data dependences within loops before the barrier insertion algorithm is

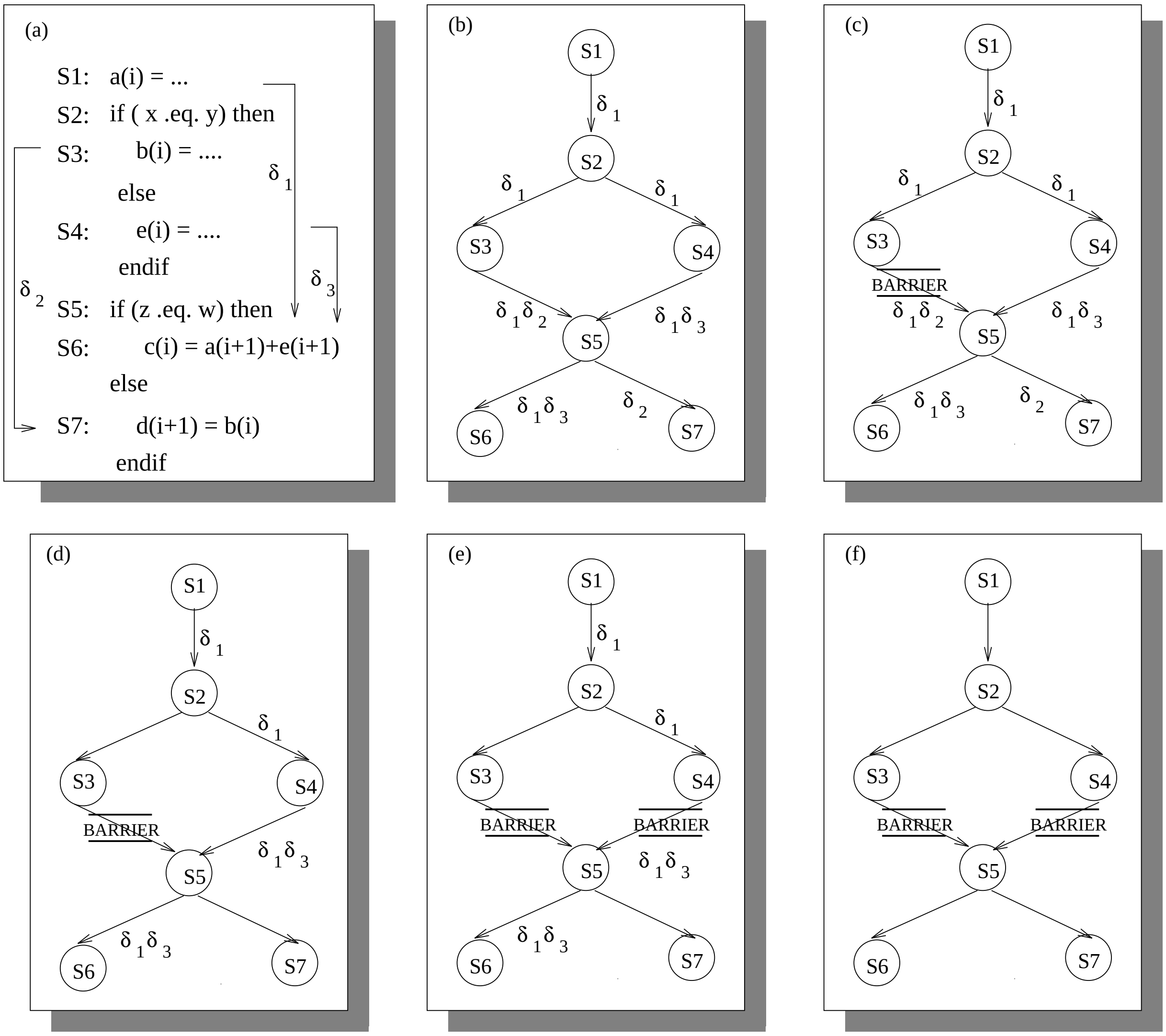

Figure 6: Barrier insertion algorithm steps.

applied, by *restructuring* the program. We also describe how they can be applied within our approach. The main techniques are :

1. Loop distribution
2. Sequential Sections
3. Post/Wait synchronization

4.1 Loop Distribution

Barrier synchronization within inner loops is expensive, but loop distribution can be used to reduce this overhead where its application results in several separate loop nests [12]. Data dependence is preserved but synchronizations are *between* loop bodies, rather than within a loop. The conditions for legal loop distribution are well known [31]. If a data scheduling approach were used in program (a) in figure 2, a barrier would be executed on every iteration, or the program fragment would have to be executed on one processor. If loop distribution were applied, however, to give the form in figure 7, each loop may be executed in parallel, with just one barrier covering the flow dependence between the loop nests.

```
    Do i = 1,63
S1: a(i+1) = x
    Enddo
    Do i = 1,63
S2: b(i) = a(i+1)
    Enddo
```

Figure 7: After loop distribution.

4.2 Sequential Section

If part of a program is completely sequential, then one option is to run it on just one processor. In the case of loop nests around array references, this means that any cross-iteration data dependence will be satisfied by sequencing, but may involve the movement of distributed array data to one processor, before later possible redistribution. Consider program (a) in figure 8, where because of cyclic dependences, it is not possible to apply simple loop distribution. Rather than incurring $O(n)$ barriers it may be preferable to execute this loop sequentially. Because of this dynamic reallocation of data and possible scalability problems, this approach it is more easily implemented when there is *shared virtual memory.* This scheme trades off parallelism and data movement against synchronization overhead.

4.3 Neighbouring Synchronization

It is possible to use a method based upon "neighbouring processor" to reduce the overheads due to sequential sections. Consider, once again, the program (a) in figure 8. If this is partitioned and scheduled in two dimensions then we have the program (b) in figure 8.

Each of the p processors has a process number ($procId$) which waits for the previous processor containing dependent data to terminate before execution. It then notifies the next processor that it may execute. This reduces the number of synchronizations from n barriers to p post/waits. The benefit increases as the number of loops increase. This scheme, however, is more restricted in applicability than barrier synchronizations.

```
Do i = 2,n
  b(i) = b(i-1) + a(i-1)
  a(i) = a(i-1) + b(i-1)
Enddo
```

(a)

```
if (procId .NE. 1) wait(procId-1)
Do i = max(2,ilo),min(n,ihi)
  b(i) = b(i-1) + a(i-1)
  a(i) = a(i-1) + b(i-1)
Enddo
if (procId .NE. numprocs) post(procId+1)
```

(b)

Figure 8: Program with Cyclic Dependence (a), Executed with Wait and Post operations (b).

Sequential loops can be classified as being one of two types; either the iterators occur in an array reference or not [4]. When a sequential iterator does not appear in an array reference it can be viewed as an "updating" or "time-stepping" iterator. The post/wait method is applicable to those cases where the iterator appears in every array reference. For example, a post/wait scheme may be used for the i loop and arrays a, b in figure 8 but cannot be used in figure 9 as i does not appear in the reference $a(k)$. Furthermore, this method can only be used when the array element/ loop iterations to synchronize with are compile-time known. This, in practice, restricts this scheme to affine access functions.

This scheme is similar to the method used when implementing Doacross parallelism, except that synchronization is on processors rather than individual data values. Executing a loop with Post/Wait synchronization ensures data dependence is satisfied so that all data dependences within the wait/post region can be removed. Some coarse-grain wave-front parallelism can be naturally exploited in some cases

[4]parallel iterators in contrast always appear in array references

```
Do i = 1,n
  b(i) = b(i-1) + function_call(b(i-1))
  Do k = 1,n
    a(k) = a(k) +b(i-1)
  Enddo
Enddo
```

Figure 9: Non applicable loop for neighbouring synchronization.

if the loop/array is partitioned over 2 or more dimensions without having to use a hyper-plane method. This is related to work described in [40].

4.4 When to Apply Optimisations

Deciding which method to use depends on applicability and cost. For SVM systems all three methods are applicable; in the case of distributed shared memory, however, sequential sections may be more difficult to implement. Loop distribution is the most favoured option as it reduces the number of synchronizations and can expose parallelism. This is at the cost of doubling the loop enumeration overhead and possible loss of locality [5]. If applicable post/wait synchronization is preferable to sequential sections. It incurs only $O(p)$ (p : number of processors) post/wait synchronizations and may expose parallelism. However, its applicability is restricted and hence sequential sections may have to be considered. Sequential sections should only be used if the time incurred by data movement is less than the the amount due to synchronization. As data movement is proportional to the size of data and synchronization proportional to the enclosing loop nest structure, then by an order of magnitude argument there may be cases where one is preferable to the other; in all others, architectural costs must be considered. This discussion assumes a distributed memory architecture. If this approach were used for symmetric multiprocessors with true shared memory, the data movement overhead is not applicable and sequential sections will be more useful. Which transformation to be used depends on program structure and machine architecture.

This optimization phase is not currently present in our compiler, however, the approach used will be as follows: In the initial stages of compilation, loop distribution should be aggressively used before any parallelisation. A post-partition stages can reverse this via loop fusion, where profitable, as long as there is no intervening barrier. We therefore trade-off an order of magnitude reduction in synchronisations in favour of a constant factor improvement in data re-use.

[5] though it can also improve cache behaviour by removing interferences between two array references

Post/wait synchronisations will be used, where legal, whenever a barrier is placed within a loop. If not legal, sequential sections will be used when the dimension of the surrounding loops is greater than or equal to the dimension of enclosed arrays. Thus, cheaper synchronisation will be used wherever possible and sequential sections only used when the data movement overhead will be less expensive than otherwise excessive synchronisation. This is a post synchronisation placement transformation, where a program with inserted barriers is transformed into a semantically equivalent one with less synchronisation overhead. To summarise, loop distribution should be at the beginning of the compilation process while post/wait synchronisation and sequential sections should take place immediately following the barrier placement algorithm. Once implemented, future work should investigate the applicability and performance impact of each transformation for real programs on existing machines.

5 Experiments

In order to validate our algorithm we incorporated it in the MARS experimental compiler developed at Rennes and Manchester [10]. This compiler is built on top of Sage++[8] and uses an augmented linear algebraic framework. The compiler is written in C++ and consists of approximately 20,000 lines of code, in addition to Sage++.

At present the compiler uses a data oriented approach to partition and schedule program. A description of the overall strategy can be found in [29]. Loop-based partitioning and scheduling techniques are currently being integrated into the system to allow future experimentation. The compiler currently targets SVM systems such as the KSR-1 and KOAN [6] on the Intel Paragon; future work will incorporate distributed shared memory machines such as the Cray T3D and Meiko CS-2 and traditional shared memory machines such as the Cray C90 and Silicon Graphics Power Challenge.

Two programs were used to investigate the effectiveness of our algorithm against an existing commercial compiler, KAP. The experimental results describe the number of barriers inserted and the execution times on the KSR for varying data and processor sizes.

The KAP compiler uses a fork & join model where barriers are placed after parallel loops and sequential sections are executed on one processor by the "master thread". The cost of forking threads dynamically is avoided by the KSR implementation as all threads are created at the start of execution and spin idlely when not involved in parallel execution. Thus a fork & join occurring at a parallel region result in a barrier rather than creation and removal of threads. As a barrier will be incurred at the end of every parallelized Enddo, one of KAPs strategies is to apply loop fusion to reduce the number of synchronizations. KAP differs from our compiler not just in its execution model but in its entire compiler strategy. Nevertheless, it is widely available on KSR-1s and provides a useful comparison.

5.1 Shallow

Shallow is a 494 line FORTRAN program implementation of the shallow water equations [18]. This program was developed from scratch by a programmer without reference to any particular machine as an exercise in systematic development of parallel programs [26]. The program consists of an initialization section followed by a sequential time stepping loop within which each of the 2-d arrays are updated. It is a stencil like problem and has a nearest neighbour reference pattern. There are 17 2-d arrays and 11 scalars. A typical program fragment consists of 1 2-d parallel loop updating the main portion of an array; two 1-d loops updating two borders and a single assignment for a corner element. The time complexity of the program is $O(n^2 \times s)$ and the space complexity $O(n^2)$, where n is the number of grid points in one dimension and s the number of time steps. In all our experiments, the number of steps was fixed at 10.

		No. of processors					
n	Version	1	2	4	8	16	32
64	MARS	2.02	1.11	0.68	0.46	0.41	0.57
	KAP	3.12	2.77	1.43	1.15	1.44	1.14
128	MARS	21.55	10.92	6.32	3.01	1.77	1.24
	KAP	28.69	22.99	10.96	6.96	7.53	5.12
256	MARS	78.62	41.54	20.82	10.61	5.60	3.19
	KAP	110.77	63.55	31.93	15.81	12.46	10.76
512	MARS	162.77	80.51	40.08	20.42	10.31	5.60
	KAP	167.89	107.62	47.43	25.67	13.58	12.70

Table 1: Elapsed times in seconds for Shallow.

The results in table 1 and figures 11 and 12 show that our algorithm and is superior to KAP over all data and processor sizes. On examining the program there were 50 barriers inserted by KAP and 8 by the barrier insertion algorithm. This translated into $28 + 22s$ barriers executed at runtime for KAP and $5 + 3s$ executed for the barrier insertion algorithm. The constant values are the number inserted between initialisation loops, and the s terms for barriers enclosed by the outer time stepping loop in the main part of the program. Asymptotically, we have reduced the number of executed barriers by approximately a factor of 7.

KAP successfully applies aggressive loop fusion to reduce synchronization but our algorithm is even more successful and inserts fewer barriers. Using a simplistic complexity analysis the running time is $O(\frac{n^2 s}{p}) + x$, i.e, the running time is proportional to the serial time divided by the number of processors plus an overhead x. Assuming synchronization is the dominant cost, x will be proportional to the number of barriers. This overhead will dominate when $\frac{n^2}{p}$ is small, ie, when n is small and p is relatively large. This analysis is supported by the actual runtimes for Shallow. For relatively large n (512) and small number of processors (≤ 4), our algorithm is only marginally superior to KAP (c. 15%). In all cases as the granularity of work decreases our algorithm improves and for (n =128, p = 32), is 4 times faster. Larger data sizes were not investigated as the data involved could not fit on one processor and terminate in a reasonable time.

Synchronization, however, is not the only overhead. The work per processor (for a fixed number of time steps) is $O(\frac{n^2}{p})$ and the amount of communication is $O(n)$ due to the column partitioning of data. As p increases relative to n then communication will have an increased impact. When $n = 64, p = 32$ increased communication has outweighed the benefit of more parallelism and has incurred a slow down when compared to 16 processors for our algorithm.

5.2 Tred2

Tred2 is a 148 line "hand-cleaned" version of an Eispack routine, generated for the purposes of an experiment [30]. It reduces a symmetric matrix to a symmetric tridiagonal matrix using and accumulating orthogonal similarity transformations. Gotos were eliminated and induction variables substituted, resulting in a program consisting of two outer serial loops in sequence, within which there were multiple 1 and 2 dimensional parallel loops. Within the program there are two 2 d arrays, three 1-d arrays and five scalars present. The time complexity of the program was $O(n^3)$ and the space complexity $O(n^2)$ where n is the problem size.

The most obvious feature of the results in table 2 and figures 13 and 14, is that KAP performs much worse the algorithm presented in this paper. On examining

		No. of processors					
n	Version	1	2	4	8	16	32
256	MARS	34.26	25.68	14.57	8.26	5.40	7.96
	KAP	41.1	37.4	26.1	22.3	23.0	-
512	MARS	287.56	211.63	125.67	66.81	35.91	24.64
	KAP	334	299	197	165	156	-
1024	MARS	2248.31	1668.57	984.04	532.04	280.52	162.47
	KAP	2622	2145	1411	1145	1053	-

Table 2: Elapsed times in seconds for Tred2.

the program there were only 12 barriers inserted by KAP compared to the 17 by the barrier insertion algorithm. This, however, translated into approximately $4 + 9n + \frac{2n^2}{6}$ barriers executed at runtime for KAP and $5 + 12n$ for the barrier insertion algorithm where the $O(n)$ terms are due to one enclosing loop and the $O(n^2)$ term due to two enclosing loops with a triangular iteration space. This high order term for KAP is due to the parallelization of the loop in figure 10.

```
               DO   J=1,513-II
C*KSR* TILE ( K )
                 DO 8 K=J+1,513-II
                   E(K) = E(K) + Z(K,J) * D(J)
    8            CONTINUE
C*KSR* END TILE
               END DO
```

Figure 10: parallelization result with KAP.

KAP wishes to parallelize the k loop because it is a DOALL loop. The j loop may also be parallelized but it is a reduction loop and requires more treatment to execute in parallel. A fork & join implementation requires a barrier after a parallelized loop, incurring, in this case, 513-ii barriers. Loop interchange could be used to reduce this to 1, but this would have the effect of bad stride access of z. Therefore KAPs heuristic of DOALL parallelism preference and stride-1 access when using a fork & join model leads to excessive synchronization.

The benefit of the barrier insertion algorithm is again most pronounced when the granularity is smallest and overheads dominate. There is a marked drop off after 16 processors for n = 256 however, due to false sharing of the one dimensional arrays. Smaller data sizes were not investigated due to this phenomenum, though it is easy to overcome [9].

Unfortunately, the KAP transformed program no longer runs due to an internal error in the recently updated presto runtime library. Only values for up to 16 processors were gathered before the upgrade occurred.

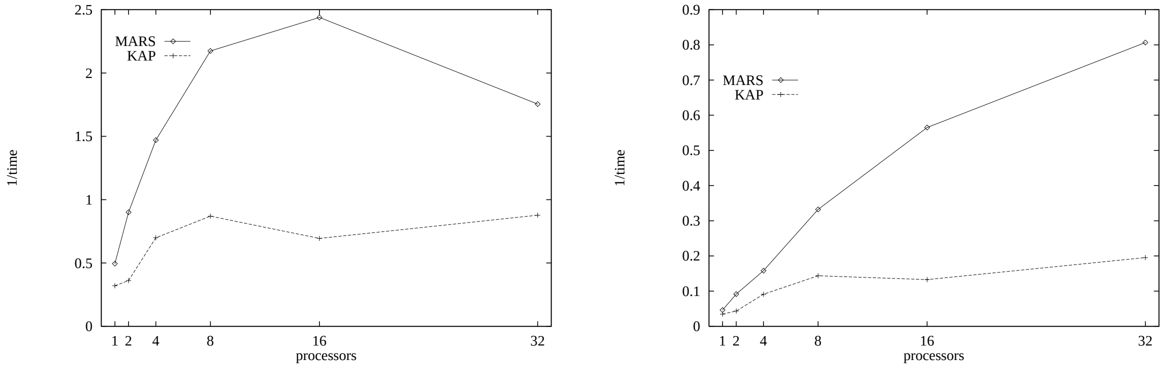

Figure 11: Performance for Shallow, n=64 and n=128.

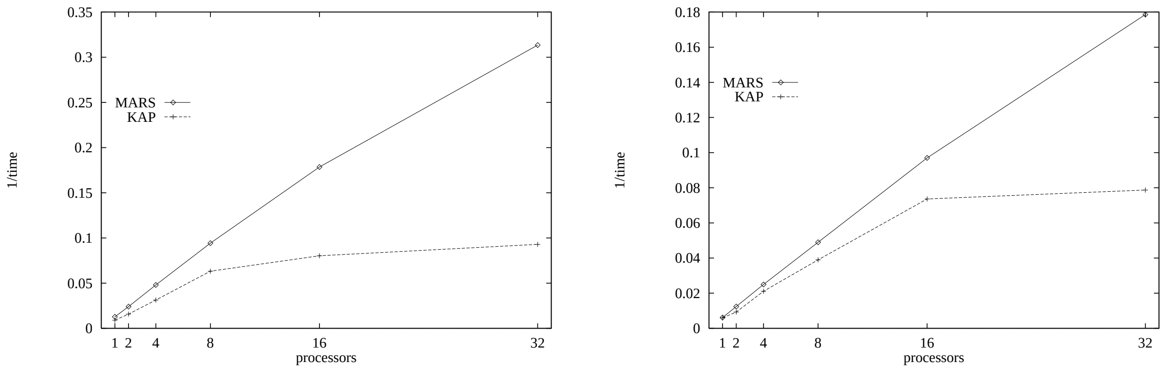

Figure 12: Performance for Shallow, n=256 and n=512.

6 Further Work

Our immediate goal is to examine and implement other partitioning and scheduling techniques. More interestingly, we wish to integrate loop and data based approaches allowing both to be used (where appropriate) in the same program. Initial experience using the KSR-1 [30] shows there are occasions when one is preferable to the other and vice-versa. Furthermore, the optimizations of section 4 must be implemented to give acceptable performance for those programs that contain a significant amount of sequential code.

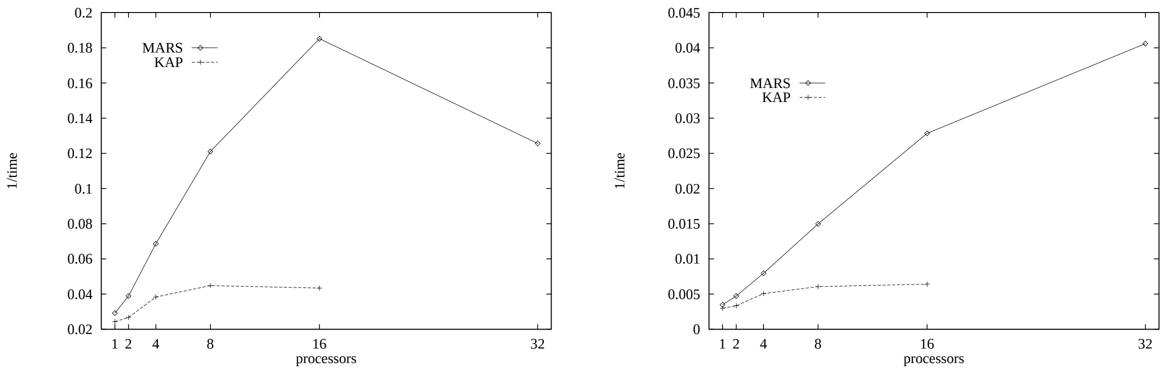

Figure 13: Performance for Tred2, n=256 and n=512.

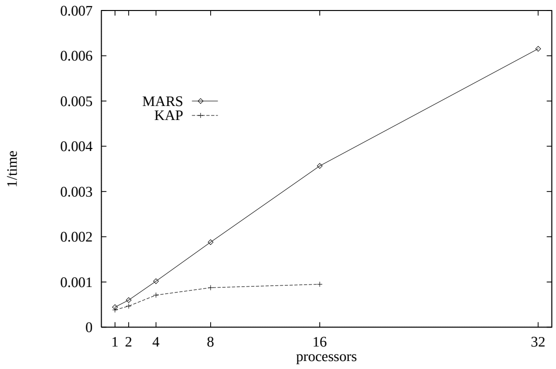

Figure 14: Performance for Tred2, n=1024.

It is assumed that the global trade-off of all costs has been decided at the loop parallelization/ data partitioning stage. However, the new techniques and optimizations described in this paper will affect prior decisions and their impact has to be integrated. It is assumed that scalars are replicated and do not require synchronization. This can be extended further to arrays, thus reducing synchronization but possibly losing parallelism. This will also have an effect on prior partitioning and scheduling decisions.

Synchronization should be proportional to the amount of communication. However just one non-local access requires the same amount of synchronization as if a whole array were accessed. Those data elements requiring non-local access could be isolated from the rest of a loop to advantage. If only part of a non-local array is to be accessed, then only those processors containing that data need to be synchronized with. If a local flag was set when data was written, then the reading processor would only have to wait until it was set, rather than waiting for all processors to reach a barrier. Furthermore, if the writer processor knew who all the readers were, (in the case of affine accesses), then the synchronization could be implemented by non-local writes or post-stores to all the readers. This is, in fact, similar to how message passing machines achieve synchronization, but it requires more analysis [3].

The porting of the back-end to more platforms will be needed to validate this algorithm. It will be interesting to see if this work, whose origins are in research into distributed memory, can provide any benefit to existing shared memory multiprocessors such as the Cray C90.

The approach in this paper should allow the easier porting of programs developed for vector machines to multiprocessors. These programs typically consist of loop sequences with small loop bodies, whose granularity is often considered too small for multiprocessors. If a SPMD model is used, there will be just one fork & join; further synchronizations will only be placed where needed. This will help alleviate the burden on programmers of fusing loop and interchanging them to give better granularity.

7 Summary

In this paper we have described a method whereby synchronization overhead may be reduced in the presence of a SPMD model of computation. Central to our approach is that we attempt to place barriers only when needed rather trying to remove those inserted by default. By using accurate data dependence information and scheduling information, barrier placement can be optimized for a given program. We have implemented the algorithm developed in this paper and initial experimental results are very encouraging showing a large improvement over the default model used on the KSR-1. Future work will build upon this experience.

Acknowledgements

We would like to thank the anonymous referees and Elena Stöhr for their valuable comments on earlier versions of this paper.

References

[1] Aho A., Sethi R., Ullman J., **Compilers Principles, Techniques, and Tools**, Addison Westley, 1986.

[2] Allen J,R, Callahan D. and D. Kennedy K., **Automatic Decomposition of Scientific Programs for Parallel Execution**, Proceedings of Principles of Programming Languages, Munich, ACM Press, January 1987.

[3] Amarasinghe S.P. and Lam M.,**Communication Optimization and Code Generation for Distributed Memory Machines**, Proceedings of Programming Language Design and Implementation, Albuquerque, ACM Press, June 1993.

[4] Anderson J.M. and Lam M.S. **Global Optimizations for Parallelism and Locality on Scalable Parallel Machines**, Proceedings of Programming Languages Design and Implementation, Albuquerque, ACM Press, June 1993

[5] Beckner S., Chapman B. and Zima H., **Vienna Fortran 90** Proceedings of Scalable High Performance Computing Conference, Williamsburg, IEEE Press, April 1992.

[6] Bodin F. and Priol T. **Overview of the Koan Programming Environment for the ipsc/2 and Performance Evaluation of the Because Test Program 2.5.1**, Future Generation Computer Systems, Elsevier Science B.V.,Vol. 10., pp. 391-401, 1994.

[7] Bodin F.. Kervella L. and Priol T. **Fortran-S: A Fortran Interface for Shared Virtual Memory Architecture**, Proceedings of Supercomputing, Portland, IEEE Press, November 1993.

[8] Bodin F., Beckman P., Gannon D., Srinivas J.G.S., **Sage++: A Class Library for Building Fortran and C++ Restructuring Tools**, Proceedings of Object-Oriented Numerics Conference, Oregon, April 1994.

[9] Bodin F., Granston E. and Mountant T. **Evaluating Two Loop Transformations for Reducing Multiple-Writer False Sharing** International Workshop on Languages and Compilers For Parallel Computing, Springer-Verlag in the Lecture Notes in Computer Science, August, 1994.

[10] Bodin F. and O'Boyle M., **MARS: Design and Implementation of a Parallelising Compiler** *in preparation.*

[11] Bouchitte V,, Boulet P., Darte A., Robert Y. **Evaluating Array Expressions on Massively Parallel Machines with Communication/Computation Overlap** Proceedings of Conpar, Linz, Springer-Verlag, September 1994.

[12] Callahan D. **A Global Approach to Detection of Parallelism** , Ph.D Dissertation, Dept of Computer Science, Rice University 1987.

[13] Darema-Rogers F., Norton V.A. Pfister G.F. **Using a Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications**, Research Report RC 11552, IBM T.J. Watson Research Centre, Yorktown Heights, November 1985.

[14] Dowling M.L., **Optimal Code Parallelization using Unimodular Transformations**, Parallel Computing, Vol. 16, pp. 157-171, 1990.

[15] Fox G., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C-W. and Wu MY., **FORTRAN D Language Specification**, Rice COMP TR90-141, Department of Computer Science, Rice University, February 1991.

[16] Garey M., Johnson D., **Computers and Intractability A Guide to the Theory of NP-Completeness**, Freeman 1979.

[17] Gavril F., **Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques and Maximum Independent Set of Chordal Graph**, SIAM Journal on Computing, Vol. 1, No. 2, pp. 1170-1183, 1972,

[18] Hoffmann, G., Swarztrauber, P.N. and Sweet R.A., **Aspects of Using Multiprocessors in Meteorological Modelling**, in Hoffmann G. and Snelling D.F. (eds.), Multiprocessing in Meteorological Models, Springer-Verlag, pp. 125-196,1988.

[19] Kennedy K and McKinley K.S., **Optimizing for Parallelism and Data Locality**, Proceedings of International Conference on Supercomputing, ACM Press, Washington D.C., July 1992.

[20] Lengauer T. and Tarjan R.E **A Fast Algorithm for Finding Dominators in a Flowgraph** , ACM Transactions On Programming Languages and Systems, ACM Press, Vol. 1, No. 1, pp. 121-141,July 1979,

[21] Li J. and Chen M., **Index Domain Alignment: Minimizing Cost of Cross-Referencing between Distributed Arrays**,Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, IEEE Press, October 1990.

[22] McKinley K.S. **Evaluating Automatic Parallelization for Efficient Execution on Shared-Memory Multiprocessors** Proceedings of International Conference on Supercomputing, Manchester, ACM Press, July 1994.

[23] Midkiff S.P. and Padua D.A., **Compiler Generated Synchronizations for Do loops** Proceedings of International Conference on Parallel Processing, IEEE Press, August, 1986.

[24] Midkiff S.P., **Automatic Generation of Synchronization Instruction for Parallel Processors**, MS. Thesis, Univ. Illinois, Urbana-Champaign, CSRD Rep. 588, May 1986.

[25] Midkiff S.P. and Padua D.A.. **Compiler Algorithms for Synchronization** IEEE Transactions on Computers Vol. 36 pp. 1485-1495, December 1987.

[26] O'Boyle M.F.P. and Nisbet A. **Shallow: A Mapping Study**, CNC Tech Rep., Department of Computer Science, University of Manchester, 1992.

[27] O'Boyle M.F.P., **Program and Data Transformations for Efficient Execution of Array Computation on Distributed Memory Architectures**, Technical Report, UMCS-93-1-6, Department. of Computer Science, University of Manchester, January 1993.

[28] O'Boyle M.F.P. and Hedayat G.A. **Data Alignment: Transformations to Reduce Communication on Distributed Memory Architectures**, Proceedings of Scalable High Performance Computing Conference, Williamsburg, IEEE Press, April 1992.

[29] O'Boyle M.F.P. **A Data Partitioning Algorithm for Distributed Memory Compilation**, PARLE '94 Athens, LNCS 817, Springer-Verlag, July 1994.

[30] O'Boyle M.F.P. and Bull J.M., **Expert Programmer vs Parallelising Compiler: A comparative study of two approaches to Shared Virtual Memory**, *to appear in* Scientific Programming.

[31] Padua D.A. and Wolfe M.J., **Advanced Compiler Optimizations for Supercomputers**, Communications of the ACM Vol.29 No.12 pp 1184-1201, December 1986.

[32] Pase D., MacDonald T. and Meltzer A.**MPP Fortran Programming Model** Cray Research 1994.

[33] Philipsen M. and Heinz E.A. **Automatic Synchronization Elimination in Synchronous FORALLs** , Proceedings of the Frontiers of massively parallel computation, IEEE Press, February 1995.

[34] Polychronopoulos C.D., **On Program Restructuring, Scheduling and Communication for Parallel Processor Systems**, Tech Rep. CSRD 595 Centre for SuperComputing Research and Development University of Illinois at Urbana-Champaign, 1986.

[35] Polychronopoulos C.D., **Parallel Programming and Compilers**. Boston M.A. Kluwer, 1988.

[36] Prakash S., Dhagat M. and Bagrodia R., **Synchronization Issues in Data-Parallel Languages**, Proceedings of International Workshop on Languages and Compiler for Parallel Computing, Portland, LNCS 768, Springer-Verlag, August 1993.

[37] Pugh W. and Wonnacott D., **Eliminating False Data Dependences Using the Omega Test**, Proceedings of Programming Languages Design and Implementation, San Francisco, ACM Press, June, 1992.

[38] Quinn M. Hatcher P and Seevers B, **Implementing a Data Parallel Language on a Tightly Coupled Multiprocessors**, in Advances in Languages and Compilers for Parallel Processing, pp. 385-401, Pitman, August 1993.

[39] Schrijver A.**Theory of Linear and Integer Programming** Wiley, Chichester 1986.

[40] Tang P. and Zigman J.N, **Reducing Data Communication Overhead for Doacross Loop Nests**, Proceedings of International Conference on Supercomputing, Manchester, ACM Press, July 1994.

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

[42] Wolf M.E. and Lam M.,**A Data Locality Optimizing Algorithm**, Proceedings of Programming Language Design and Implementation, Toronto, ACM Press, June 1991.

[43] Zima H.P. Bast H,J. and Gerndt H.M, **A Tool for Semi-Automatic MIMD/SIMD parallelization**, Parallel Computing 6, pp1 - 18, 1988.

[44] Zima H., **Supercompilers for Parallel and Vector Computers**, ACM Press, 1991.

List of Figures

A Proofs

Proof of Theorem 1 *Only one processor is the source of uniquely referenced dependence and only one is the sink. If there is a dependence between any two processors they will refer to the same data i.e.* $f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset$. *If all such dependences are within one processor i.e.* $(f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset) \Rightarrow (x = y) \mid \forall x, y \in P$ *then the sequential execution order of the processor will ensure that the access to* $f(\underline{I_x})$ *will have completed before access to* $g(\underline{J_x})$ *is made and therefore there is no need for synchronisation*

Proof of Corollary 1 *If* $f(I)$ *is uniquely referenced and* $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\}) \mid \forall x \in P$*,then* $g(\{J\})$ *is uniquely referenced We have for any processor* y, $f(\{\underline{I_y}\}) = g(\{\underline{J_y}\})$ *hence condition* $f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset$ *can be re-written as* $f(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) \neq \emptyset$. *As* $f(I)$ *is is uniquely referenced, then this is only true when* $x = y$. *Thus* $(f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset) \Rightarrow (x = y)$ *and theorem 1 follows.*

Proof of Theorem 2 *Given condition (2) and* f *is one-to-one, then* $f(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) = \emptyset$ *i.e.* $f(I)$ *is uniquely referenced. As there is only one loop nest* $I \equiv J$ *and condition (3) follows. As this is a Doall loop then* $f = g$ *and* f, g *are one-to-one. Therefore* $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\}), \forall x \in P$ *and corollary 1 follows.*

Proof of Theorem 3 *Given condition (2) and* f *is one-to-one, then* $f(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) = \emptyset$ *i.e.* $f(I)$ *is uniquely referenced.*

As $\{I\} = \{J\}$ *then condition 3 follows.As* $f = g$ *and* f, g *are one-to-one,* $f(\{\underline{I_x}\}) = g(\{\underline{J_x}\})$ *and corollary 1 follows.*

Proof of Theorem 4 *As this is an output dependence* $f(I)$ *and* $g(J)$ *are write accesses. By the owner-computes rule accesses to* $f(I)$ *and* $g(J)$ *are local and uniquely referenced. As there is a static data distribution all writes will be to the same processor i.e.* $(f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset) \Rightarrow (x = y) \mid \forall x, y \in P$ *and theorem 1 follows.*

Proof of Theorem 5 *By condition 4 all write accesses to b will be local and uniquely referenced. If a,b are aligned on a partitioned index then by property 1, all the corresponding elements of a will be local and uniquely referenced and the theorem follows.*

Proof of Theorem 6 *Consider the case (without loss of generality)when* $f(I)$ *is the access function of a write reference in statement,* S_1, $g(J)$ *is the access function of a read reference in* S_2 *and* $l(J)$ *is the access function of the write reference in* S_2. *From the owner-compute rule and condition 4,* $f(I)$ *and* $l(J)$ *are uniquely referenced As the read reference is aligned and* $l(J)$ *is uniquely referenced, then by theorem 4,* $g(J)$ *is uniquely refenced. By a similar argument* $(f(\{\underline{I_x}\}) \cap g(\{\underline{J_y}\}) \neq \emptyset) \Rightarrow (x = y) \mid \forall x, y \in P$ *and theorem 1 holds.*

■

Michael O'Boyle received a Ph.D in Computer Science from the University of Manchester in July 1992. He has recently concluded his position as an SERC Postdoctoral Research Fellow and is now working as a Research Fellow within the Centre for Novel Computing at the University of Manchester. In 1994 he spent six months working with the CAPS group at IRISA.

His main research interests are compilation for distributed memory single address space architectures, formal program transformation representation and compiler impact on shared virtual memory systems especially with respect to coherence.

Lionel Kervella received his diploma in computer science in 1991 from the Institut de Formation Supérieure en Informatique et Communication (IFSIC), Rennes. Currently, he is a Ph.D. candidate at University of Rennes, France. His interests include parallel programming environment and code generation for shared virtual memory systems.

François Bodin was born in Rennes on November 14, 1961. He received a PhD degree in computer science at University of Rennes I (FRANCE) in 1989. In 1990, he spent a year as post-doctoral student at University of Indiana in Pr. Gannon's team. He is currently Assistant Professor at IRISA in France. His interests include compiler technology for fast RISC processors, parallelizer and vectorizer tools.