Compiler Reduction of Synchronisation in Shared Virtual Memory Systems

Michael O'Boyle
Centre for Novel Computing
Department of Computer Science
University of Manchester
Manchester M13 9PL
email: mob@cs.man.ac.uk

François Bodin
IRISA-INRIA
Campus de Beaulieu
35042 Rennes Cedex
email: bodin@irisa.fr

Abstract

This paper presents a new compiler method that reduces the number of executed barriers on shared virtual memory architectures using dependence and scheduling information. Wait/post synchronisation is used in sequential sections to further reduce synchronisation overheads and maintain locality. The work presented in this paper goes beyond existing research by exploiting the properties of the SPMD execution model and unifies the treatment of loop and data based partitioning. Using compile time knowledge of the data and work layout, a new technique is described which determines dependences that do not require synchronisation is described. Furthermore, synchronisations are placed so as to minimise runtime overhead. These techniques have been implemented in an experimental compiler, MARS, where initial results are very promising, outperforming existing methods

1 Introduction

Shared Virtual Memory (SVM) architectures provide a single address space on distributed hardware with no physically shared memory. Instead the distributed local memories collectively provide a virtual address space shared by all processors. SVM promises the benefit of the ease of programming found in shared-memory multiprocessors with the scalability of message-passing multiprocessors. The compiler is relieved from the difficulties of producing optimised message-passing code, but must now be concerned with data locality and synchronisation overhead. This paper is concerned with reducing synchronisation overhead without adversely affecting locality. On machines such as the KSR-1, synchronisation is very expensive and exploitation of locality is essential for scalable performance [23]. Much work has been concerned with locality optimisation [14],[30]. but little recent attention has been paid to synchronisation overheads. This paper describes a new compiler technique that minimises synchronisation overhead when using a SPMD execution model [10]. Furthermore data movement is kept to a minimum in sequential sections of a program by using post/wait rather than barrier synchronisation. Both of these techniques take advantage of compile-time knowledge of the location of the source and sink of dependences.The techniques described in this paper are applicable if either traditional loop based parallelisation or data partitioning is used, and thus unify the treatment of synchronisation.

Section 2 describes the impact on synchronisation of both loop and data based scheduling, while section 3 formalises the approach for incorporation as a compiler algorithm. Section 4 describes experimental results on two program sections derived from the U.K. Meteorological Offices forecasting program, comparing this method with an existing commercial compiler KAP.

1.1 Motivation

To illustrate the methods described in this paper, consider the program fragment in figure 1. If a fork/join implementation is used [14], there will be at least 8 barriers, one before and after each parallelised loop nest. This is assuming that all outer loops are parallelised; parallelising innermost loops would increase the number of barriers to $2 \times \frac{N+1-II}{2} + 4$.

If a SPMD execution model is used and the loop and data partitioned/scheduled to give the node program in figure 1, the number of barriers can be reduced to 1. Elements $e[lo..hi]$ are written in statements S1 and S2 where lo and hi are the local iterator bounds after SPMD code generation. The same processor writes to the same data locally and hence there is no *cross-processor* dependence requiring synchronisation. The same is true for the flow dependence on e from S1 to S2 and the anti-dependence from S2 to S4. All the dependences on gg are also local except the flow dependence from S3 to S5 when computing privatised variable g (i.e. g is replicated on all processors). The access to $gg(n+1-ii)$ is, in general, non-local and therefore requires synchronisation. The barrier may be placed anywhere between S3 and S5. In general, the optimal placement will depend on other dependences present in the program.

There are occasions where *cross-processor* dependences occur within a loop nest, as in the second loop nest in figure 2. One solution is to run the loop on just one processor but this can lead to excessive data movement between parallel and sequential regions. Rather than incur barrier overheads or data movement, each processor can evaluate it's local data sequentially before synchronising with the next, as shown in the node program in figure 2 where z refers to the processor number. Rather than redistributing the data we have, in effect, redistributed, the control flow.

Source Program

```
          DO J = 1, N+1-II
            DO K = J+1 ,N+1-II
S1            E(K) = E(K)+Z(K,J)*D(J)
            ENDDO
          ENDDO
          DO J = 1, N+1-II
S2          GG(J) = E(J)+Z(J,J)*D(J)
          ENDDO
          DO J = 1, N+1-II
            DO K = J+1 ,N+1-II
S3            GG(J) = GG(J)+Z(K,J)*D(K)
            ENDDO
          ENDDO
          DO J = 1, N+1-II
S4          E(J) = GG(J)
          ENDDO
S5        G = GG(N+1-II)
```

Node Program

```
          DO J = 1, N+1-II
            DO K = max(J+1,lo) ,min(N+1-II,hi)
S1            E(K) = E(K)+Z(K,J)*D(J)
            ENDDO
          ENDDO
          DO J = max(1,lo), min(N+1-II,hi)
S2          GG(J) = E(J)+Z(J,J)*D(J)
          ENDDO
          DO J = max(1,lo), min(N+1-II,hi)
            DO K = J+1, N+1-II
S3            GG(J) = GG(J)+Z(K,J)*D(K)
            ENDDO
          ENDDO
          DO J = max(1,lo), min(N+1-II,hi)
S4          E(J) = GG(J)
          ENDDO
          call Barrier()
S5        G = GG(N+1-II)
```

Figure 1: Program Extract

1.2 Related Work and Contribution

The contribution of this paper is a compiler algorithm that reduces the number of executed barriers on shared virtual memory architectures based on dependence and scheduling information. Wait/post synchronisation is used in sequential sections to further reduce synchronisation overheads and maintain locality. Previous techniques do not make any assumption on the partitioning of the iteration or data space across processors, furthermore a fork and join model is generally assumed [9, 1, 32, 27]. Some researchers, within this framework, have used kill analysis to remove covered dependences, particularly in doacross loops, so as to reduce the number of synchronisations. Midkiff et al. [17, 19] have addressed the problem of removing redundant data dependence to limit the number of synchronisation points. In [28], Doacross parallelism is exploited on a distributed memory machine where the main goals are exploitation of parallelism and amortising communication overhead for stencil like problems. Program transformations to eliminate or decrease the cost of synchronisations inside parallel and doacross loops have also been proposed [9, 1, 18, 17, 19]. The program transformations used are loop distribution, loop fusion, and loop alignment. Loop distribution is used to transform a synchronisation 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. 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 minimising the amount of replication code [1].

More recent work has addressed the problem of synchronisation insertion for data parallel languages. Prakash et al. [26] 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 synchronisation. The work done by Philippsen et al. [25] addresses the problem of inserting synchronisations in FORALL statements. The method applies at the expression level rather than at the statement level. The main contribution is minimising 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. Coupled with compile time knowledge of the data and work layout it is possible to eliminate dependences that do not require synchronisation. In our framework, barriers are placed so as to mimimise runtime overhead due to synchronisation. In addition, we utilise post/wait synchronisation to further reduce runtime overhead in sequential sections while maintaining data locality.

2 Partitioning and Scheduling

The algorithm given in this paper relies on a SPMD execution model. The SPMD model requires only one fork at the beginning and one join at the end of a program but all synchronisation points have to be explicitly catered for. Synchronisation placement has to be addressed by the compiler for efficient execution. Synchronisations need only be enforced when there exists a cross-processor dependence but determining this requires scheduling information from previous stages in the compilation process. In contrast, shared memory parallel compilers have traditionally used a fork/join model to exploit program parallelism. Each parallel DO implied a fork and each ENDDO a join, both of which were expensive operations. Any remaining synchronisation was handled by a variety of methods including barriers [32]

To determine the processor(s) containing the source and sink of dependence it is necessary to know how work and data are scheduled to them. This will depend on whether loop or data based partitioning/scheduling is used.

Loop Scheduling Loop based scheduling is frequently used when using shared virtual memory machines. However because of the impact of non-local memory references, data movement must be considered when partitioning the loops.

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 dependence; hence there is no need for synchro-

Parallel and Sequential

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

Wait and Post Node Program

```
Do t = 1,n
  Do i = lo,hi
    a(i) = a(i) +1
    b(i) = b(i) +1
    c(i) = c(i) +1
  Enddo

wait(z-1)
  Do i = lo,hi
    a(i) = b(i-1) +a(i)
    b(i) = a(i-1) +b(i)
    c(i) = c(i-1) +b(i)
  Enddo
post(z)
Enddo
```

Figure 2: Wait and Post

nisation. The iterations of the remaining loops are either executed by every processor or by just one, depending on whether enclosed loops are parallel [32].

Data Scheduling Data partitioning is generally used for distributed memory machines [2, 3] and is used for shared virtual memory machines [22] by the MARS compiler [8].

In contrast to loop scheduling, data scheduling is concerned with the mapping of array elements to processors. The data alignment, partition and mapping of such elements is determined by previous stages in the compilation process. Using the "owner-computes model" [31] implies that all write accesses to an array are to local data. Static data layout, alignment and partitioning information is used to determine the source and sink of dependences and thus reduce synchronisation.

2.1 Loop versus Data Scheduling

Although these partition/scheduling techniques come from two quite distinct traditions, they are strongly related from a synchronisation point of view. Both schemes can remove synchronisation if the source and sink of a dependence are scheduled to the same processor. Loop scheduling determines the local iteration space from which data accesses can be determined. Data scheduling, in contrast, determines the local data to be accessed which implies the local iteration space. To illustrate these approaches consider the two programs in figures 3. The first source program, (A), can be executed in parallel if a loop based scheme is used, but must be executed sequentially if a data based approach is used. The converse is true for the second program,(B).

With the loop scheduling, 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 processor ($p = 4$) and in particular what occurs on processor 2 ($z = 2$). In both programs, (A),(B), the local iteration spaces associated with S1 and S2 are the same. In the first node program, (C), there is no cross-processor dependence as there is no cross-iteration dependence in the source program (A). For example when $i = 32$, $a(33)$ will be written by processor 2 and read by processor 2. In the second program, (B), there is a cross-iteration data dependences and hence there would be synchronisation. In particular processor 2 would write $a(17..32)$ but read elements $b(32..47)$ which would be written by processor 3. Typically in this case, however, the loop will be executed sequentially by just one processor incurring data movement.

We now consider data scheduling where a and b are aligned and block-partitioned. Cross-processor dependence will occur in the first source program (A), between the border elements of the local arrays resident on each processor. Array elements 17..32 of arrays a and b will be scheduled to processor 2. The iteration space associated with S1 is i : 16..31 and with S2 i : 17..32 to ensure the owner-compute/local-write rule. When computing element $b(32)$, $a(33)$ will be read, but this is allocated on processor 3. Furthermore $a(33)$ must be written before calculating $b(32)$ and hence there is an cross-processor dependence requiring synchronisation. In the second program, (B), the same array elements of a and b will be scheduled to processor 2, but the iteration spaces associated with each statement are quite different, as shown in the node program (D). Elements of $b(17..32)$ will access the corresponding elements of a, all of which are local, and hence there is no inter-processor dependence or synchronisation. The crucial point is how data is mapped to processors, rather than the dependence distance, determines synchronisation.

When post/wait synchronisation is to be used, the form of synchronisation will again depend on the scheduling method used. Consider the source program in figure 4 where there is a cross-iteration anti-dependence on b. Loop scheduling requires that the processors execute in increasing number order while data scheduling requires the opposite. This surprising result is due to the owner-computes rule where each statement may have its own separate local iteration space, which can affect the ordering of statements and hence the synchronisation order. If the processors were executed in loop schedule order, for example, processor 1 would write to $b(17)$, as it is local, before processor 2 can read it Reversing the order preserves the program's semantics. The remainder of this paper formalises these ideas and develops new techniques to reduce synchronisation overhead in the presence of loop and data scheduling.

Loop Parallel (A)	Data Parallel (B)

<pre>   Dimension a(64),b(64)
     Do i = 1, 63
S1:   a(i+1) = x
S2:   b(i) = a(i+1)
     Enddo</pre>
	<pre> Dimension a(64),b(64)
	     Do i = 1,63
	S1: a(i+1) = x
	S2: b(64-i) = a(64-i)
	     Enddo</pre>

Node Program after Loop Scheduling (C)	**Node Program after Data Scheduling (D)**

<pre>   Dimension a(64),b(64)
     Do i = 17, 32
S1:   a(i+1) = x
S2:   b(i) = a(i+1)
     Enddo</pre>
	<pre> Dimension a(64),b(64)
	     Do i = 1,63
	       if (16.LE.i).AND.(i.LE.31)
	S1: a(i+1) = x
	       if (32.LE.i).AND.(i.LE.47)
	S2: b(64-i) = a(64-i)
	     Enddo</pre>

Figure 3: Loop and Data Parallel Source and Node Programs

3 Algorithm

Synchronisation reduction is just one phase of an optimising compiler [8]. In this paper we assume that the loop tiling or data partitioning has already taken place and may not be altered. The synchronisation phase can be expressed in a simplified manner as follows:

1. Mark as satisfied all dependences known not to cross a processor boundary based on scheduling information.
2. Determine the best place to globally place synchronisation points and kill covered dependences
3. If there is an inner loop dependence use wait/post synchronisation if possible

Once all synchronisation inducing 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; our heuristic algorithm tries to achieve this. When this cannot be avoided, post/wait synchronisation is used if possible. Section 3.1 describes a technique to determine cross-processor dependence, while section 3.2 briefly outlines the placement of barriers to satisfy those dependences. Sections 3.3. to 3.7 describe the conditions under which wait/post synchronisation may be used and the form of the program transformation.

3.1 Determining Dependences That Do Not Require Synchronisation

Determining the data dependences that cause cross-processor synchronisation 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 $x \neq y$ synchronisation is needed. If $x = y$ then the data dependence is satisfied by sequencing. 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 [12]. 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 synchronisation. It is assumed that all scalars are privatised, so the remainder of this section is concerned with access to shared arrays.

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

Definition 1 *The* **local iteration space** *of a processor $x \in P$, is the set of iterations of a loop nest J performed by that processor, where P is the set of processors . The local iterators are denoted by the vector $\underline{J_x}$ and the* **local iteration space** *is denoted by $\{\underline{J_x}\}$*

Definition 2 *An array* a*, with access $\mathtt{a}(f(I))$ in a statement S, is said to be* **uniquely referenced** *if and only if:*

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

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 synchronisation required if $\mathtt{a}(f(I))$ is* **uniquely referenced**, *$\mathtt{a}(g(J))$ is* **uniquely referenced** *and*

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

In other words, if both source and sink are uniquely referenced and if there is a data dependence it is within a processor, then there is no need for synchronisation. Due to space restrictions, all proofs are omitted.

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

Corollary 1 *If $\mathtt{a}(f(I))$ 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.*

3.1.1 Loop Scheduling

This section develops two useful theorems which allow the removal of synchronisation points when using *affinity* scheduling. When scheduling iterations to processors it is assumed

Source	Loop Scheduling	Data Scheduling

`Do i = 1,63`
`  b(i-1) =x`
`  a(i+1) = b(i)`
`Enddo`
	`wait(z-1)`
	`Do i = 17,32`
	`  b(i-1) =x`
	`  a(i+1) = b(i)`
	`Enddo`
	`post(z)`
		`wait(z+1)`
		`Do i = 1,63`
		`if (18.LE.i).AND.(i.LE.33)`
		`  b(i-1) =x`
		`if (16.LE.i).AND.(i.LE.31)`
		`  a(i+1) = b(i)`
		`Enddo`
		`post(z)`

Figure 4: Wait and Post Synchronisation for Loop and Data Scheduling

that no iterator is scheduled to more than one processor, i.e.,

$$schedule(I) \Rightarrow \{\underline{I_x}\} \cap \{\underline{I_y}\} = \emptyset \mid \forall x, y \in P,\ x \neq y \quad (1)$$

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:

$$(\{I\} = \{J\}) \wedge (type(I) = type(J)) \Rightarrow \{\underline{I_x}\} = \{\underline{J_x}\} \mid \forall x \in P \quad (2)$$

where *type* is either parallel or serial.

The following two theorems are used to prevent synchronising 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 synchronisation.*

For example there is no need for any synchronisation in program (A) in figure 3. 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 ($\{\underline{I_x}\} = \{\underline{J_x}\}$), $\forall x \in P$, 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 synchronisation*

For example, in program (a) of figure 5, there is no need for synchronisation. Examining the local program (b) of figure 5 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 synchronisation.

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 $f \neq g$ 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 [20] and to minimise intra-statement communication in constant communication distance problems [11] can utilise 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.

$$write(f) \Rightarrow f(\{\underline{I_x}\}) \cap f(\{\underline{I_y}\}) = \emptyset \mid \forall x, y \in P, x \neq y \quad (3)$$

Thus $\mathtt{a}(f(I))$, 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 synchronisation*

For example, consider program (a) in figure 6 and the node program (b) for processor 2 after applying the owner-computes rule. The output dependence from S1 to S2 in figure 6(a) can be shown to be local when the partitioned local program is examined in figure 6(b). Before the next theorem can be developed, the following well known property is needed. For further information see [15, 21]

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

Theorem 5 *If a read access to an array $\mathtt{a}(f(I))$ on the right hand side of an assignment is aligned on a partitioned index with the write access to array $\mathtt{b}(l(I))$ 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 synchronisation*

Source Program	Node Program

`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 5: Program with No Synchronisation using Affinity Scheduling

Source Program	Node Program

`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 6: Program with No Synchronisation using Data Scheduling

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 (B) in figure 3, where a and b are aligned. As the access to a in statement S2 is the same as to b after partitioning, all data dependences to a in node program (D) are 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.

3.2 Barrier Insertion

Once we have computed the minimum number of cross-processor dependences, we need to know where to place synchronisation points so as to preserve program semantics but not to incur excessive overhead. We use the control flow graph (CFG) and data dependence information to find the "best" place to insert synchronisations. When considering just one dependence this problem can be reduced to a maximal cut algorithm which unfortunately is NP complete [13]. Barriers are inserted so as to cover as many dependences as possible, while avoiding placement within nested loops wherever possible. Once placed, all covered dependences are marked as satisfied and only the remainder need be considered. The full details of this iterative scheme and its effect are described in [7].

3.3 Wait/Post

While the barrier insertion algorithm determines the minimum number of barriers, giving a set of dependences and scheduling information for a program, it can lead to barriers in inner loops. This section is primarily concerned with a transformations, that try to eliminate dependences within loops. It is related to doacross parallelism, except that synchronisation is on processors rather than individual data values and has similarities with the macro-tiles described in [29]. Some coarse-grain wave-front parallelism is naturally exploited in some cases if a loop or array is partitioned over 2 or more dimensions without having to use a hyperplane method. The stages for correctly generating efficient correct code can be summarised as follows:

1. Determine Synchronisation Distance/Direction
2. Determine Processors to Synchronise with
3. Transform Program

3.4 Synchronisation Distance/Direction

The synchronisation direction defines the (partial) order threads or processors must execute in and is defined as:

Definition 3 *Synchronisation Direction is the sign of the following $x - y$ where x, y are the processor containing the respective source and sink of a dependence.*

Consider the processors $1..p$, if the synchronisation direction is positive, then the processors must execute in the order $1, 2, 3..p$. A negative direction implies the ordering $p, p-1, p-2, p-3..1$. In this paper we restrict our attention to programs with constant synchronisation direction. This has the benefit of simple code generation. Programs with more complex dependences would require additional transformations. Thus the kernel code for a thread or processor z, $z \in 1..p$ with a 1 dimensional partition is of the form shown in figure 7 where $k = 1$ is the synchronisation direction if positive (i.e. wait for the previous one to finish),

$k = -1$ if the synchronisation direction is negative. This can be extended to 2 or more dimensions corresponding to tiled loops or partitioned arrays.

Definition 4 *Synchronisation Distance is the value of the following $x - y$ where x, y are the processor containing the respective source and sink of a dependence.*

In the case where the distance and hence the location of the source and sink of a dependence is precisely known, more efficient code generation is possibe and may allow exploitation of parallelism in a manner similar to cycle shrinking. If processor 3 has to wait for only processor 1 to complete but processor 4 has to wait for only processor 2 to complete, processors 1 and 2 can execute in parallel as can 3 and 4, 5 and 6 etc., if the dependence distance is constant and the source and sink are two processors apart, ($k = 2$).

```
wait (z-k)
Do i = lo,hi
  ...
Enddo
post (z)
```

Figure 7: Generic code

3.4.1 Loop Scheduling

The synchronisation direction is determined by simply examining the dependence direction of each data dependence corresponding to the partitioned iterator. The dependence matrix D is used [30], where each row corresponds to the distance in loop iterations between accesses to the same part of memory. Constant dependences will have constant entries while constant direction dependences will be described by +1 or -1 . Those dependences with varying direction are marked as *. The synchronisation direction for a particular loop may change depending on the loop level the wait/post code is to be inserted. For instance the dependence vector in figure 8 is $[1, -1]^T$ when considering the whole loop i, j, but changes to [1] when considering just the j loop [32]. The synchronisation distance changes accordingly as well. Thus if the j loop is partitioned, $k = -1$ if the wait is place outermost but $k = 1$ if placed surrounding just j.

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

Figure 8: Dependence Distance Example

3.4.2 Data Scheduling

Data scheduling depends on the mapping of array elements rather than iterators to processors and thus it is the relative alignment that is important. As described earlier, (section 3.1), if an access is perfectly aligned within a statement then it will be local requiring no dependence. Any difference implies inter-processor synchronisation. If the sign of the difference remains the same or there is a constant direction then this synchronisation scheme can be applied. For data scheduling a communication matrix C [11] is used whose row entries represent the direction/distance in each array index between successive accesses to the same array element. In figure 8 the communication matrix is the same as the data dependence matrix i.e. $C = [1, -1]^T$.

Unlike data dependence this does not vary at different loop levels. The type of data dependence, however, affects the direction of synchronisation. Table 9 summarises the affect data dependence type has on the synchronisation direction/distance. Flow dependence is, as expected, the same as with loop scheduling. Anti-dependence, however, requires synchronisation in the opposite direction to the communication direction. This property would allow the work described in [28] to be extended to anti-dependences. Only flow and anti-dependence need be considered as in owner computes data scheduling, output dependences are always local.

Dependence	sign($C_{i,j}$) = -1	sign($C_{i,j}$) = +1
flow	-1	+1
anti	+1	-1

Figure 9: Synchronisation Direction

3.5 Synchronising Processors

In order to correctly place synchronisation primitives, it is necessary to determine those accesses that incur cross-processor dependence. Only those dependences occurring in a partitioned/parallelised dimension need be considered for synchronisation as the remainder are guaranteed to be local/sequentialised and do not require further attention.

Let $\mathcal{P}$ be the partition matrix whose non-null entries define those indexes/iterators to be distributed. Let $S = \mathcal{P}D$ or $S = \mathcal{P}C$ be the partitioned synchronisation matrices and S' be the matrix where only rows of S corresponding to non-zero rows of $\mathcal{P}$ are maintained. This way non-partitioned entries are removed.

For constant distance synchronisation the processor to synchronise with depends on the distance and the number of elements/iterators scheduled to each processor. In the case of just one dimensional dependence the k term in figure 7 is of the form $k = \lceil \frac{s}{bls} \rceil$ where s is the synchronisation distance, and bls represents the number of elements/iterations scheduled per processor (i.e. blocksize). This is assuming that the number of data element/ loop iterations is evenly distributed i.e. by block for a constant distant statements. With 2 or more dimensions,the source of one dependence may occur on more than one processor in more than one dimension.

Let $Z = [z_1, ..z_{rank(\mathcal{P})}]^T$ be the processor vector, whose elements represent a processor identifier in a particular dimension i.e. $Z = [z_1, z_2]^T = [3, 2]^T$ means the processors are to be considered as a two dimensional array and the element in the 3rd row and 2nd column is referred to.

In figure 8, $S' = [1, -1]^T$; if there is only one element or iteration per processor ($bls = 1$) then processor $[z_1, z_2]^T$ will synchronise with $[z_1, z_2]^T - [1, -1]^T = [z_1 - 1, z_2 + 1]^T$ or it's north-east neighbour. For blocksizes greater than 1 the north, east and north-east neighbours $[z_1 - 1, z_2]^T, [z_1, z_2 + 1]^T, [z_1 - 1, z_2 + 1]^T$ need to be synchronised with. Thus the processors to synchronise with will depend on the synchronisation distance/direction and blocksize.

Honouring all these synchronisation vectors will produce correct programs but may be overly conservative introducing more than are strictly required. If processor u is to synchronise with v which in turn must synchronise with w, a synchronisation dependence between u and w does not have to explicitly satisfied as as it is implicitly satisfied. It can be shown that all dependences that are a positive integer combination of others need not be considered. Therefore if $S^{'} = [1,-1]^T$ and $bls > 1$, processor $[z_1, z_2]^T$, need not synchronise with $[z_1 - 1, z_2 + 1]^T$ as $[1,-1]^T = [1,0]^T + [0,-1]^T$

3.6 Legality Criteria

Before the program transformation for wait/post scheduling can be defined, it is necessary to describe when it may be legally applied.

Let X be those synchronisation dependences to be explicitly satisfied. The synchronisation dependence direction for any dependence, must be constant in a particular dimension. i.e. the order of processor synchronisation must be consistent throughout the whole execution.

$$\forall i, j | X_{i,j} \neq * \tag{4}$$

Constant synchronisation direction is not sufficient. The synchronisation directions must not lead to deadlock by having a cyclic dependence. Let x_i be a column vector of X. This acyclic constaint can be expressed as:

$$\nexists a_i \in \mathbb{Z}^+ \ | \ \sum_{i=1} a_i x_i = 0 \tag{5}$$

For instance synchronisation directions $[1,0]^T, [0,1]^T$ do not deadlock, but $[1,1]^T$, $[-1,-1]^T$ do as $[1,1]^T + 1 \times [-1,-1]^T = 0$

3.7 Transformation

Given the dependences to be honoured, X, we insert a number of wait $\mathcal{S}^W$ and post $\mathcal{S}^P$ synchronisation statements around a loop nest i.e.

where

$$\mathcal{S}^W = (\mathcal{S}_1^W, \ldots, \mathcal{S}_{rank(X)}^W) \tag{6}$$

and

$$\mathcal{S}^P = (\mathcal{S}_1^P, \ldots, \mathcal{S}_{rank(X)}^P) \tag{7}$$

Each wait statement has the form

$$\mathcal{S}_k^W = wait(Z_1 - fX_{1,k}, Z_2 - fX_{2,k}, \ldots, Z_n - fX_{n,k}) \tag{8}$$

and post statements

$$\mathcal{S}_k^P = post(Z_1, Z_2, \ldots, Z_n) \tag{9}$$

For loop scheduling $f = 1$, in other words, it is solely the synchronisation distance that determines the synchronisation order. For data scheduling, however $f = 1$ if there is a flow dependence and -1 if it is an anti-dependence as described in section 3.4. If the example in figure 8 is partitioned in 2 dimensions, this will give give the local program shown in figure 10.

```
wait(z1-1,z2)
wait(z1,z2+1)
Do i = max(2,lo0), min(n,hi0)
  Do j = max(2,lo1),min(n,hi1)
    b(i,j) = b(i-1,j+1) +a(i,j)
  Enddo
Enddo
post(z1,z2)
post(z1,z2)
```

Figure 10: Dependence Distance Example

4 Experiments

In order to validate our method we incorporated it in the MARS compiler developed at IRISA and Manchester [8]. This compiler is built on top of Sage++[5] 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 data scheduling; loop scheduling techniques will be integrated into the system. The compiler currently targets SVM systems including the KSR-1, ICL Goldrush and KOAN [4] on the Intel Paragon

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

4.1 Vert Vel

The U.K. Meteorological Office's (UKMO) Unified Model includes both data assimilation and prediction for the atmosphere and ocean. VertVel is one of the most computationally intensive routines from this program. It calculates the divergence from mass-weighted horizontal velocity then derives the mass-weighted vertical velocity field. To simplify the experiment several parameters were set equal to one parameter n giving a serial time complexity of $O(n^3)$ and a space complexity of $O(n^2)$. The code was relatively easy to parallelise for both compiler's the only difference being in the number of synchronisation inserted. The figures in Table 1 show the elapsed time in seconds for varying data and processor sizes when using KAP and our compiler.

KAP tiled most of the loops inserting 12 barriers into the code which translated into $6n^2 + 6n$ at runtime. MARS inserted just one barrier which translated to n^2 barriers at runtime. KAP varies from 1.5 to 4 times slower than our scheme. This is most marked at small granularities (n small relative to p) as the overhead due to synchronisation will begin to dominate. False sharing is responsible for the slow downs on 32 processors for data sizes 128 and 256, however false sharing elimination transformations based on [6] when implemented will improve the performance of our scheme for these values.

4.2 Simple Loop

This program is a simplification of a program construct found elsewhere in the (UKMO) and is shown in figure 2.

Both KAP and our compiler parallelise the first i loop, but KAP runs the second loop nest sequentially while our compiler places post and wait synchronisations around it. Both versions produce programs with the same parallel work

		No. of processors					
N	Version	1	2	4	8	16	32
128	Compiler	27.04	17.40	12.46	14.72	12.75	20.35
	KAP	41.99	36.36	34.38	47.40	35.88	48.32
256	Compiler	220.65	129.28	77.94	74.70	65.38	82.07
	KAP	328.61	308.67	187.75	236.58	250.21	239.98
512	Compiler	1835.04	1188.95	971.10	693.20	382.09	345.06
	KAP	2566.32	1953.82	1256.90	1231.05	1130.61	1215.79
1024	Compiler	*	14077.28	9222.40	4778.08	3072.08	2201.42
	KAP	*	*	11743.60	7556.64	5937.64	6819.26

Table 1: Elapsed times in seconds for Vert Vel

		No. of processors					
N	Version	1	2	4	8	16	32
64	Compiler	0.01	0.02	0.04	0.08	0.19	0.71
	KAP	0.27	0.33	0.67	0.59	0.26	0.30
128	Compiler	0.04	0.05	0.07	0.12	0.31	1.18
	KAP	1.37	1.19	1.37	1.23	0.84	0.75
256	Compiler	0.15	0.15	0.18	0.24	0.48	1.93
	KAP	0.34	0.54	0.54	1.41	2.17	2.82
512	Compiler	0.56	0.53	0.53	0.65	1.04	3.42
	KAP	1.21	2.00	1.91	2.30	8.12	8.41
1024	Compiler	2.22	1.84	1.83	2.00	2.59	6.60
	KAP	4.65	7.46	7.13	7.81	8.90	9.70
2048	Compiler	8.90	6.94	6.84	8.05	8.03	14.07
	KAP	18.25	28.9	27.55	30.89	31.65	36.51

Table 2: Elapsed times in seconds for Simple Loop

complexity of $3\frac{n^2}{p} + 3n^2$ i.e. $O(n^2)$ but differ in their overheads. KAP incurs $2n$ barriers due to the tiling of the first loop plus $O(n^2)$ data movement. The data in effect is redistributed twice on every iteration of the t loop. The post wait implementation incurs no data movement overhead but has $2n$ barriers an $n \times p$ post/wait synchronisations. This scheme will be most effective when n is large relative to p

Examining the results in table 2 broadly follows this analysis. The wall clock times for KAP are in some instances much larger than for our compiler, sometimes up to factor of 6 times larger. This is due to the large overhead of operating system support for page migration. The times given are the best over a number of runs for each value of n and p. While the variation was around 10% for our compiler the times varied by a factor of 20 for the KAP implementation, particularly for the large data sizes. The only occasion when KAP outperformed our compiler was when excessive false sharing occurred for small values of n and large values of p.

5 Summary

In this paper we have described a method whereby synchronisation overhead may be reduced by using dependence and scheduling information. We have unified the treatment of loop and data based parallelisation schemes with respect to synchronisation. In sequential sections we have used post/wait synchronisation to preserve locality without excessive synchronisation overhead. We have implemented the algorithm developed in this paper and initial experimental results are very encouraging showing a large improvement over an existing commercial compiler.

In this paper it has been assumed that the partitioning and scheduling of the program has already been decided. Obviously the overhead of synchronisation should effect this decision. Future work will investigate the impact of our synchronisation minimisation techniques on such decisions and use a transformations such as loop distribution to give an integrated solution.

References

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

[2] 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.

[3] 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

[4] 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, In Because Workshop, October 1992.

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

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

[7] Bodin F., Kervella L. and O'Boyle M.F.P, **Synchronization Minimization in a SPMD Execution Model** IRISA Tech Rep, 1994.

[8] Bodin F. and O'Boyle M.F.P., **A Compiler Strategy for SVM**, *to appear* Workshop for Language, Compiler and Run-time Systems for Scalable Performance. New York, May 1995.

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

[10] Darema-Rogers F., Norton V.A. and 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.

[11] Darte A. and Robert Y., **Mapping uniform loop nest onto distributed memory architectures**, Parallel Computing 20 (1194) pp. 679-710.

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

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

[14] 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.

[15] 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

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

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

[18] Midkiff S.P., **Automatic generation of synchronization instruction for parallel processors**, MS. Thesis, Univ. Illinois, Urbana-Champaign, CSRD Rep. 588, May 1986.

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

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

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

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] Wolfe M.J., **More Iteration Space Tiling**, Proceedings of Supercomputing '89, IEEE Press, November 1989.

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

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

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