# Synchronization Minimization in a SPMD Execution Model

François Bodin, Lionel Kervella, Mike O' Boyle

▶ **To cite this version:**

François Bodin, Lionel Kervella, Mike O' Boyle. Synchronization Minimization in a SPMD Execution Model. [Research Report] RR-2387, INRIA. 1994. inria-00074288

**HAL Id: inria-00074288**
**https://inria.hal.science/inria-00074288**

Submitted on 24 May 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Synchronization Minimization in a SPMD Execution Model*

F. Bodin and L. Kervella and M. O' Boyle

**N° 2387**

Septembre 1994

PROGRAMME 1

ISSN 0249-6399

# Synchronization Minimization in a SPMD Execution Model

F. Bodin and L. Kervella and M. O' Boyle

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet CAPS

Rapport de recherche n ° 2387 — Septembre 1994 — 20 pages

**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 dependence. In this paper we investigate two scheduling techniques, loop based and data based, both of which use an SPMD model. Using scheduling information from previous stages in the compilation process, potential cross-processor dependences are determined. Given the minimum number of cross-processor data dependences that must be satisfied, an optimization technique based on a maximal cut algorithm 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.

**Key-words:** Synchronization, SPMD, global adress space, parallel loops, control and data parallelism.

*(Résumé : tsvp)*

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

# Optimisation des synchronisations avec le modèle d'exécution SPMD

**Résumé :** Ce rapport présente un algorithme pour le placement des synchronisations dans un programme parallèle s'éxécutant selon le modèle SPMD (un seul programme pour plusieurs flots de données). L'algorithme admet deux techniques de parallélisation :le parallélisme fonctionnel (boucle parallèle) ainsi que le parallèlisme par distribution de données.

A partir de la répartition des calculs sur les processeurs, les dépendances entre processeursont déterminées. Un algorithme similaire à celui de l'obtention de la "coupure maximale" est utilisé pour déterminer le nombre minimun de points de synchronisation nécéssaires. Cet algorithme a été mis en œuvre au sein d'un prototype et les premières expériences montrent l'éfficacité de notre méthode.

**Mots-clé :** Synchronisation, SPMD, espace d'adressage global, boucles parallèles, parallèlisme par distribution du calcul et des données.

# 1 Introduction

A popular method of exploiting parallelism, particularly on distributed memory machines, is the single program multiple data, SPMD [12], 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 catered for. The optimal compiler placement of synchronization points has to be addressed for efficient execution. synchronizations should only be enforced when there exists a cross-processor dependence but determining this 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 were expensive operations. Any remaining synchronization was handled by a variety of methods including barriers [40]. [1]

Although there has been much research into the minimization of overheads such as communication and load imbalance when using an 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 [28]. 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 [30], [6].

In this paper we are concerned with reducing synchronization in single address space computers using an 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 traditional shared memory multiprocessors such as the Sequent Balance, if a SPMD model were used.

## 1.1 Memory Consistency

Distributed memory machines with a single address space share characteristics from 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.

[1] The cost of forking threads dynamically can be avoided by creating them at the start of execution and letting them spin idlely when not involved in parallel execution. However, a synchronization is still required at the fork and join points.

Memory consistency in message-passing implementations is ensured by only sending and receiving up to date values, where synchronization occurs when communicating. This may explain the lack of attention given to synchronization removal when using an SPMD model. In single address space computing, however, consistency is ensured by 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 to two different processors then inserting a synchronization between the two ensures that the write takes place before the read. If, however, the same process (thread) writes and reads the data then normal von Neumann execution sequence will ensure that the correct data is accessed. This is preferable as it is a less expensive operation. Using compile time knowledge it is possible to determine if a dependence can be solely satisfied by sequencing.

Two variants on distributed memory single address can be found : *distributed shared memory* such as the Cray T3D, 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 [2] 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 [18] [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 [13] and Vienna Fortran [5]. This method can also be used on shared virtual memory machines [27]. Static data layout, alignment and partitioning information is used to determine the source and sink of dependences and thus reduce synchronization.

Loop based scheduling is frequently used when using *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 partitioning the loops. Loop scheduling information is used to determine the source and sink of dependences and again reduce synchronization.

The algorithm given in this paper is applicable to all single address machines using a SPMD whether loop or data based scheduling 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 the algorithm to determine cross-processor dependences and the placement of barriers to cover them. Section 4 describes additional optimization techniques to reduce synchronization overhead in the presence of sequential loops. Section 5 validates

[2] or data migration

experiments to illustrate this algorithm, implemented in our compiler, which 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 [24], [17]. If a fork/join implementation were used, the amount of synchronization would depend on which parts of the program were parallelized. Assuming that each loop nest is parallelized and synchronization is by way of barriers, there will be at least 16 barriers needed, one before and after each loop nest. This is assuming that the outer loops are parallelized in each case. If the inner loops were parallelized, the number of synchronization would increase to $6n+10$.

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

The only dependences are flow dependences on $U$ between S1 and S9, S2 and S9, S3 and S10, and statement S4 and S10. These four 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 dependence. However if loops or data are appropriately aligned the source and sinks of these dependences will be within the same processor, and synchronization will be unnecessary. Thus by using accurate array data flow information, an SPMD model and scheduling information, the number of synchronization points can be reduced to zero.

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. The following sub sections 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 [29] How loop iterations are mapped is largely decided by previous stages; either by automatic parallelization or by user directives. We consider that 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, then scheduling distinct set of iterations to each processor implies that there will no inter-processor 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 [40].

**Data Scheduling** In contrast to loop scheduling, data scheduling is concerned with the mapping of array elements to processors. The data partition and mapping of such elements

```
    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

is determined by previous stages in the compilation process. Again this can be either done by compiler methods [27] or by directives [13]. All arrays are assumed to be distributed or partitioned across the processors while scalars are assumed to be replicated. 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 [19],[26]. Knowledge of the data partition provides accurate information as to the location of the source and sink of a dependence. Using the "owner-computes model" [39] implies that all write accesses to an array are to local data.

## 2.1 Loop versus Data Partitioning

Although these partition/scheduling techniques come from two quite distinct traditions, they are strongly related from a synchronization point of view. Both schemes can remove synchronization if the source and sink of a dependence are scheduled to the same processor. Loop partitioning/scheduling determines the local iteration space from which data accesses can be determined. Data partitioning/scheduling, 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: (a) Loop Parallelism, (b) Index Parallelism

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 processor ($p = 4$) 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, there is no inter-processor dependence as there is no cross-iteration dependence. For example when $i = 32$, $a(33)$ will be written by processor 2 and read by processor 2. In the second example, there is a cross-iteration data dependences and hence synchronization. In particular processor 2 will write $a(17..32)$ but read elements $b(32..47)$ 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.

Inter-processor dependence will occur in the first example, if data scheduling is used, at the border elements of processor. Again consider the case with 4 processor ($p = 4$) and in particular what occurs on processor 2. Array elements 17..32 of arrays $a$ and $b$ will be scheduled to this 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. Element $b(32)$ will read $a(33)$ but this is allocated on processor 3. Furthermore $a(33)$ must be written before it is read by $b(32)$ and hence there is an interprocessor dependence requiring synchronization. In the second case 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. 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 synchronization.

The crucial point is how data is mapped to processors, rather than the dependence distance. Both $a$ and $b$ in the second example share the same reference and as they are mapped to processors in the same manner there are no non-local memory references. A dependence is guaranteed to be internal if the read access has the same memory reference as the write in that statement, regardless of the dependence type or distance. This can be relaxed in the case of 2 or more dimension arrays, where only the partitioned indices have to have the same memory reference. In both cases, however loop distribution could be used to remove synchronization occurring in the middle of the loops.

## 2.2 Related Work

Most of the previous techniques do not make any assumption on the partitioning of the iteration space or data space across processors, furthermore a fork and join model is generally assumed [11, 2, 40, 21, 23]. Some researches, within this framework, have used kill analysis to remove covered dependences, particularly in Doacross loops, so as to reduce the number of synchronizations. Midkiff et al. [21, 23] 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 dependence has be shown to be NP-Hard[22].

Program transformations to eliminate or decrease the cost of synchronizations inside parallel and doacross loops have also been proposed [11, 2, 40, 34, 21, 23]. The program transformations used are loop distribution, loop fusion and loop alignment. Loop distribution is used to move a synchronization in a parallel loop between the parallel loop resulting from the distribution. 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 dependences. 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. [34] have proposed a greedy algorithm to minimize the number of synchronization in a basic block. The algorithm is based on finding, for a set of intervals defined by the sink and source of the dependencies, the minimum number of points such that every intervals spans at least one point. This problem was shown to be polynomial[16]. The global algorithm, from Quinn et al. [34], adds synchronization to loops, going outward from the innermost loop, then uses the basic block algorithm to treat the remaining dependencies.

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 program's partition it is possible to determine dependences that do not require synchronization. Furthermore by using a method (based on a maximal cut algorithm) synchronization are placed so 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 or data scheduling is used.

# 3 Algorithm

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

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

Step 1 is a basic requirement of much compiler analysis. We require accurate data dependence information to reduce the number of spurious dependences to be considered [3] 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 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. [4]

Step 3 describes the first contribution of this paper. By using information from steps 1 and 2 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

[3] At present we use the Omega Test [33]

[4] Our compiler uses an automatic data partitioning scheme as described in [27]

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. This step uses a heuristic algorithm to achieve this.

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 4.1 describes a technique to determine whether a dependence requires synchronization, while section 4.2. describes the placement of barriers to satisfy those dependences.

The loops in a FORTRAN program surrounding any statement can be represented as an $m \times 1$ vector 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 $[J, I]^T$

**Definition 2** *A* **data reference** *$ref$ is a read or write reference to a scalar or array within a program. $ref_k$ is the $k$th reference within a statement $S$.*

For example $U(I+1, J)$ is a data reference to the array $U$

**Definition 3** *An* **access function** *$f$ determines the array elements accessed within a data reference*

Frequently access functions are affine functions of the enclosing iteration space. For example the access function of the reference U(I+1,J) can be represented as $f(J)$ where $f(J)$ is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} J \\ I \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

**Definition 4** *A* **data dependence** *$\delta$ is a tuple of the following form:*

$$\delta = (source,\ sink,\ Statout,\ Statin,\ type)$$

*where $source, sink$ designate the data references forming the dependence the which occur in statements $Statout$, $Statin$ respectively. type indicates the form of the dependence; output, flow or anti.*

## 3.1 Determining Dependences That Do Not Require Synchronization

Determining the dependences that cause inter-processor synchronization depends on how the program is mapped to the machine. If the source and the sink are mapped to processors $p_x$ and $p_y$ then if $x \neq y$ synchronization is needed. If $x = y$ then the dependence will be 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 slightly more involved. In the general case for each processor it is necessary to determine the array sections of the source and sink and determine their equality. Fortunately, simpler and much less expensive methods can be used.

If a dependence involves a private variable, then both accesses are local by definition and hence there is no cross-processor dependence requiring synchronization. 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, a($f(I)$) and a($g(J)$), to a shared array $a$, which form the source and sink of a dependence. $I$ and $J$ are vectors representing the enclosing iterators and $f$ and $g$ are access functions. Let $\{f\}$ be the set of values of the function $f$, so $\{f(I)\}$ and $\{g(J)\}$ are the set of array elements whose values lie between the lower and upper bounds of array $a$.[5]

**Definition 5** *The* **local iteration space** *of a processor $p_x$ 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 6** *An access to an array $a$, $g(J)$ in a statement $S$ is* **uniquely referenced** *by a processor $p_x$ if and only if for all values of $\underline{J_x}$ only processor $p_x$ accesses array elements $\{g(\underline{J_x})\}$ 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 $f(\underline{I_x})$ is* **uniquely referenced** *by processor $p_x$ and $g(\underline{J_y})$ is* **uniquely referenced** *by processor $p_y$ and $y = x$.*

For the purposes of this section, this can be expressed as a more useful corollary.

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

This corollary forms the criteria for determining synchronization points for both scheduling techniques. The requirement for equality $\{f(\underline{I})\} = \{g(\underline{J})\}$ can be relaxed. $\{f(\underline{I})\}$ may be a subset of $\{g(\underline{J})\}$ as long as $g(J)$ is uniquely referenced and vice-versa. This leads to two criteria either of which must hold for a dependence to be local and not requiring synchronization.

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

**Criterion 1:**

$$\{f(\underline{I_x})\} \subseteq \{g(\underline{J_x})\} \mid \forall x \in P \tag{2}$$

$$\{g(\underline{J_x})\} \cap \{g(\underline{J_y})\} = \emptyset \mid \forall x, y \in P\, x \neq y \tag{3}$$

**Criterion 2:**

$$\{f(\underline{I_x})\} \supseteq \{g(\underline{J_x})\} \mid \forall x \in P \tag{4}$$

$$\{f(\underline{I_x})\} \cap \{f(\underline{I_y})\} = \emptyset \mid \forall x, y \in P\, x \neq y \tag{5}$$

Criterion 1 states that, for every processor, the array elements accessed by the source of the dependence are a sub-set of the sink elements and there are no two processors accessing the same sink elements. Criterion 2 has a similar interpretation in that the array elements accessed by the sink of the dependence are a sub-set of the source elements and there are no two processors accessing the same source element.

### 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.

$$schedule(I) \Rightarrow \underline{I_x} \neq \underline{I_y} \mid \forall x, y \in P\, x \neq y \tag{6}$$

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:

$$\{I\} = \{J\} \wedge looptype(I) = looptype(J) \Rightarrow \{\underline{I_x}\} = \{\underline{J_x}\} \mid \forall x \in P \tag{7}$$

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

**Theorem 2** *A data dependence, loop independent, occurring within one Doall loop implies 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 bijective other wise 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 dependences occurring across loops to be examined.

**Theorem 3** *If the source and sink of a 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 bijective, 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, we see that the elements written in S1, a(5..8) are read in the second statement. This dependence is entirely local and does not require any synchronization.

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

(b)

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

Although the two theorems allow the identification of many internal data dependences, they are conservative in their analysis. The crux of the theorems is showing $\{f(\underline{I_x})\} = \{g(\underline{J_x})\}$. This 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 $\{f(\underline{I_x})\} = \{g(\underline{J_x})\}$. A useful scheduling technique would be, given two different access functions $f$ and $g$ to, say, an array $a$, determine the local iteration spaces such that $\{f(\underline{I_x})\} = \{g(\underline{J_x})\}$. For example in figure 3, if the access were a(i) in S1 and a(k+3) in S2, then the local iteration space for the second processor should be 5 to 8 for the first loop and 2 to 5 for the second. This would ensure local dependence and there would be no need for synchronization. This scheduling is similar to first applying loop alignment [11] and then using affinity scheduling.

### 3.1.2 Data Scheduling

Data scheduling involves the mapping of array elements to processors. In this section we will concentrate on a well known "owner-computes rule" mapping. Other techniques to exploit reduction parallelism [25] and to minimize intra-statement communication in constant communication distance problems [10] 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.

$$write(f) \Rightarrow \{f(\underline{I_x})\} \cap \{f(\underline{I_y})\} = \emptyset \forall x, y \in P, x \neq y \tag{8}$$

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.

$$write(f) \wedge write(g) \Rightarrow \{f(\underline{I_x})\} \subseteq \{g(\underline{J_x})\} \mid \forall x \in P \tag{9}$$

or

$$write(f) \wedge write(g) \Rightarrow \{f(\underline{I_x})\} \supseteq \{g(\underline{J_x})\} \mid \forall x \in P \quad (10)$$

The following theorem follows trivially from these properties.

**Theorem 4** *Output dependences never require synchronization*

```
      Dimension a(64),b(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),b(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 Before and After Data Scheduling (processor 2)

For example consider the program (a) in figure 4. The output dependence from S1 to S2 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 [19], [26]

**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 b on the right hand side of an assignment statement is partitioned along an index aligned with the write access to array a on the left hand side, then all read references to b by a will be local if they have the same access function.*

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 exactly aligned on all partitioned indices with the write array reference in a particular statement, no dependencies associated with that read array reference require synchronization*

This theorem has the important property that all 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 the 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 dependences to $a$ in program (b) in figure 5 will be local.

```
      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 Scheduling (processor 2)

Owner-compute scheduling has an asymmetric treatment of 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 bijective in order to determine a local dependence i.e. two different values of an iterator $i$ access distinct array elements $f(i)$, but the data based approach does have this requirement. In both loop and data based cases, conservative techniques to determine local 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 scheduling schemes have been investigated; future work will investigate synchronization removal in the presence of other scheduling policies.

## 3.2 Maximal Cut

Once we have computed the minimum number of interprocessor dependencies, 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 dependencies information to find the "best" places to insert synchronizations. When considering just one dependence this problem can be reduced to a maximal cut algorithm which unfortunately is NP complete [15]. For this reason we use a heuristic algorithm based on the control flow graph which is defined as follows:

**Definition 7** *The Control Flow Graph ($CFG$) is a directed graph $(S, E)$ where $S$ are basic statements of the program and $E$ the set of control relation between two statements. An entry and exit nodes are added for every loops of the program. 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.*

If there is a dependence requiring synchronisation between two statements $S_b$ and $S_d$ then all paths in the $CFG$ from $S_b$ and $S_d$ will be marked with a $\delta$.

We use a heuristic based on the number of data dependences marking an edge and the enclosing loop nest depth to determine where to place barriers. Once this has been