Compiler Algorithms for Event Variable Synchronization

Zhiyuan Li*
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
305 Talbot Lab., 104 S. Wright St., Urbana, IL 61801
li@csrd.uiuc.edu

May 13, 1991

Abstract

Event variable synchronization is a well-known mechanism for enforcing data dependences in a program that runs in parallel on a shared memory multiprocessor. This paper presents compiler algorithms to automatically generate event variable synchronization code. Previously published algorithms dealt with single parallel loops in which dependence distances are constant and known by the compiler. However, loops in real application programs are often arbitrarily nested. Moreover, compilers are often unable to determine dependence distances. In contrast, our algorithms generate synchronization code based directly on array subscripts and do not require constant distances in data dependences. The algorithms are designed for arbitrarily nested loops, including *triangular* or *trapezoidal* loops.

1 Introduction

On shared memory multiprocessors, the performance of scientific and engineering programs can often be improved by running DO loop iterations in parallel. Some recent simulation studies report tremendous potential parallelism in many application programs. However, the studies also report frequent data dependences across execution threads which must be preserved by means of synchronization [Kum88, CSY90]. Most commercial shared memory multiprocessors provide synchronization mechanisms to preserve data dependences between parallel loop iterations. Many of those mechanisms are either equivalent or close to the event variable synchronization described in the widely circulated PCF document [For88]. Implementation of the event variable synchronization is straightforward on virtually any shared memory multiprocessor.

Manually inserted synchronization code is error-prone. Recently, Callahan, Kennedy, and Subhlok proposed a static analysis to detect errors in user-inserted synchronization code [CKS90]. In this paper, we describe a method that generates event variable synchronization automatically. Midkiff and Padua presented compiler algorithms to generate event posts and waits in single parallel loops, assuming that all data dependences have known constant distances [MP87]. However, more powerful algorithms are needed because loops in real application programs are often arbitrarily nested and compilers are often unable to determine dependence distances [SLY90]. We make two contributions toward meeting this need: First, we present algorithms to deal with arbitrarily nested loops, including the *triangular* or *trapezoidal* loops, in which the lower and upper bounds of inner loops may vary with the index value of the outer loops. Second, our algorithms do not require constant dependence distances. Instead, we generate synchronization code based directly on array subscripts and loop bounds. Of course, we only generate synchronization for a dependence that is proved or assumed to exist between parallel loop iterations by data dependence analysis [WB87, AK87].

2 Background and Assumptions

Program constructs

Given an arbitrary nest of DO loops, a compiler may decide to parallelize all loops or serialize some of the loops. Our task is to generate the necessary synchronization code to support the intended parallelism and to guarantee the correct computation results.

Data dependences between different loop iterations are called *loop carried dependences* [AK87]. We make two assumptions to ensure that explicit synchronization

*This work is supported by the **Department of Energy** Grant DOE DE-FG02-85ER25001.

is needed only for loop carried dependences: First, the processors executing a parallel loop may exit only after all iterations are completed, and second, statements in the same iteration of a parallel loop execute sequentially in their original order.

We restrict the loop bounds to the *singly indexed* form such that each bound expression can be written as $a \times i + b$, where i is the index variable of an outer loop and a and b are invariant expressions. We believe this form is general enough to cover most cases in practical programs. However, we do allow arbitrary loop bound expressions if all loop carried dependences in those loops are due to a class of array references which will be described later in Section 4. Without loss of generality, we assume loop increments of one and that the lower bound of a loop can never be greater than the upper bound. Statements in the DO loop nest are assumed to be `do`, `enddo`, `if`, and assignments.

Event variable synchronization

Event variable synchronization uses two operations, `post` and `wait`, on a logical variable called an *event variable*. The operation `post`(ev) sets the event variable *ev* to `true`; the operation `wait`(ev) busy-waits until *ev* becomes `true`. A data dependence in a parallel loop is enforced by executing a `post` after the dependence source and a `wait` before the dependence sink. The initial value of an event variable is assumed to be `false`.

A data dependence may be a *flow dependence*, which requires a read reference to follow a write reference; an *anti dependence*, which requires a write reference to follow a read reference; or an *output dependence*, which requires a write reference to follow another write reference [Kuc78]. If one reference depends on another, we call the former *dependence sink* and the latter the *dependence source*. In the sample loop in Figure 1(a), a standard data dependence analysis will report a loop carried flow dependence from the A(I1+I2,I2) reference, the source, to the A(I1,I2-2) reference, the sink. However, more information is needed in order to synchronize the dependent references if the doubly nested loops are to be executed in parallel. First, because the A(I1,I2-2) reference should not wait in all loop iterations, a condition must be checked in each iteration to see if a `wait` must really be executed. (Executing unnecessary `wait` may cause deadlock.) We call this condition the *mask predicate*, which is tested by the `if` statement in Figure 1(b). Second, it must be determined which instance of the A(I1+I2,I2) reference should be followed by which instance of the A(I1,I2-2) reference. We call such a correspondence between two dependent references a *contact*, which should be maintained by indexing the event array correctly in `post` and `wait` as in Figure 1(b). In the rest of this paper we discuss how to formulate mask predicates and contacts and how to use them to generate event variable synchronization automatically.

```
DO I1 = 1, N
DO I2 = I1+1, N+1
.. = A(I1,I2-2)
A(I1+I2,I2) = ..
ENDDO
ENDDO
```

(a)

```
DOALL I1 = 1, N
DOALL I2 = I1+1, N+1
IF ((I2.EQ.I1+1)
   .AND. (I1.GE.3))
WAIT(EV(I1-I2+2,I2-2))
.. = A(I1,I2-2)
A(I1+I2,I2) = ..
POST(EV(I1,I2))
ENDDO
ENDDO
```

(b)

Figure 1: An Example of Mask Predicates and Contacts

3 Enhanced data dependence information

Mask predicates and contacts

We explained in the last section that mask predicates and contacts are two pieces of essential information that are not available from standard data dependence analysis. To facilitate further discussion, we define them formally in the following.

Given a loop nest L which has singly indexed loop bounds as defined in Section 2, we consider a (potential) loop carried data dependence, δ, from an array reference R_1 to another array reference R_2. Since the references may be nested in different loops, we use different names for their loop indices: $i_1, i_2, \ldots, i_{l_1}$ for the l_1 loops enclosing R_1, and $j_1, j_2, \ldots, j_{l_2}$ for the l_2 loops enclosing R_2. Fixing a value for each index i_k within its loop bound, we have an instance of R_1 which is denoted by $R_1\langle i_1, i_2, \ldots, i_{l_1}\rangle$. Similarly, we denote an instance of R_2 by $R_2\langle j_1, j_2, \ldots, j_{l_2}\rangle$.

Definition The *mask predicate* of δ is a predicate $\mathcal{F}_\delta(j_1, j_2, \ldots, j_{l_2})$ which is true if and only if $j_1, j_2, \ldots, j_{l_2}$ are within loop bounds and $R_2\langle j_1, j_2, \ldots, j_{l_2}\rangle$ depends on some instances of R_1. When no confusion results, we write $\mathcal{F}$ instead of $\mathcal{F}_\delta$.

Definition A *contact* of δ is a vector $(e_1, e_2, \ldots, e_{l_1})$, where e_k is an integer linear expression in indices $j_1, j_2, \ldots, j_{l_2}$, such that if $\mathcal{F}(j_1, j_2, \ldots, j_{l_2})$ is true, then $R_2\langle j_1, j_2, \ldots, j_{l_2}\rangle$ depends on $R_1\langle e_1, e_2, \ldots, e_{l_1}\rangle$.

Example For the dependence from A(I1+I2,I2) to A(I1,I2-2) in Figure 1(a), we have $\mathcal{F} = (I_2 = I_1 + 1) \wedge (I_1 \geq 3)$. The only contact of the dependence is $(I_1 - I_2 + 2, I_2 - 2)$.

A dependence may have multiple contacts. The following defines a precedence relation between two contacts.

Definition A contact $(e_1, e_2, \ldots, e_{l_1})$ of δ from R_1 to R_2 *precedes* another contact $(h_1, h_2, \ldots, h_{l_1})$ if $R_1\langle e_1, e_2, \ldots, e_{l_1}\rangle$ precedes $R_1\langle h_1, h_2, \ldots, h_{l_1}\rangle$ in the serial execution of L.

Definition The *closest contact* of δ is the contact that is preceded by all other contacts.

Suppose R_1 and R_2 are written as $A(\phi_1, \phi_2, \ldots, \phi_m)$ and $A(\psi_1, \psi_2, \ldots, \psi_m)$, where ϕ_k and ψ_k are integer linear expressions in the loop indices. If dependence δ exists, then some values of $i_1, i_2, \ldots, i_{l_1}$ and $j_1, j_2, \ldots, j_{l_2}$ must satisfy a set of constraints:

$$\phi_k(i_1, i_2, \ldots, i_{l_1}) = \psi_k(j_1, j_2, \ldots, j_{l_2}) \quad (1)$$
$$u_k i_{p(k)} + v_k \le i_k \le u'_k i_{p'(k)} + v'_k \quad (2)$$
$$\tilde{u}_k j_{q(k)} + \tilde{v}_k \le j_k \le \tilde{u}'_k j_{q'(k)} + \tilde{v}'_k \quad (3)$$
$$(i_1 < j_1) \vee (i_1 = j_1) \wedge (i_2 < j_2) \vee \cdots \vee$$
$$(i_1 = j_1) \wedge \cdots \wedge (i_{l_c-1} = j_{l_c-1}) \wedge (i_{l_c} < j_{l_c}). \quad (4)$$

where l_c is the number of loops common to R_1 and R_2, $k = 1, 2, \ldots, m$, and u_k, v_k, $\tilde{u}_k$, and $\tilde{v}_k$ are all integer invariants in L. We call eqs. (1) the *subscript equations*, ineqs. (2) and (3) the *loop bound constraints*, and formula (4) the *precedence constraints*. Note that outer loop indices may appear in inner loop bounds. Hence, $u_1 = \tilde{u}_1 = v_1 = \tilde{v}_1 = 0$, and p, q, $\tilde{p}$, and $\tilde{q}$ are mappings such that $p(k)$, $q(k)$, $\tilde{p}(k)$, $\tilde{q}(k) < k$.

If the compiler decides to serialize at the k^{th} loop level, $1 \le k \le l_c$, then the term in formula (4) that contains the factor $(i_k < j_k)$ should be removed.

From the formulas in (1-4), one can see that data dependences could be arbitrarily complex if we consider arbitrary array subscripts. In this paper, we require R_1 to have a *diagonalizable coefficient matrix*, which is defined below. We believe this definition is general enough for most cases that occur in practical programs. As for R_2, we allow arbitrary linear subscripts.

Coefficient matrix

Definition Consider an m-dimensional array reference, R, whose subscripts contain n index variables. The *coefficient matrix* of R is an $m \times n$ matrix $\mathbf{A}$ whose element $A^{(j,k)}$ is the coefficient of the loop index variable i_k in the j-th dimension of R.

Definition A coefficient matrix $\mathbf{A}$ is *diagonalizable* if there exists a nonsingular integer square matrix $\mathbf{P}$ such that $\mathbf{PA}$ is in the *quasi-diagonal* form

$$\begin{pmatrix} D & 0 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} D \\ 0 \end{pmatrix}$$

where D is a diagonal submatrix.

Example 1) The coefficient matrix of H(2*i, i-1) is $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$, which is diagonalizable.

2) The coefficient matrix of H(2*k, 100) is $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$, which is in the quasi-diagonal form and is thus diagonalizable.
3) The coefficient matrix of H(i, N-2*j), where N is a loop invariant, is $\begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}$, which is in the quasi-diagonal form and is thus diagonalizable.

4) The coefficient matrix of H(i+j, 3) is $\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$, which is not diagonalizable. □

Notice that the number of columns in a coefficient matrix may be less than the number of loops enclosing the array reference, because some index variables may not appear in the array subscripts.

We diagonalize a coefficient matrix by a modified *Gaussian elimination* which, at each elimination step, scales the rows when necessary to keep integer results. It takes at most $2 \times n$ multiplications and n additions to eliminate one element. Therefore, with no more than $2 \times m \times n^2$ multiplications and $m \times n^2$ additions at compile time, a coefficient matrix can either be diagonalized or be recognized as non-diagonalizable.

In the next two sections, we discuss two classes of diagonalizable coefficient matrices which must be dealt with differently. In both sections, we assume the absence of `if` statements in the considered loop nests. `if` statements will be covered in Section 6.

4 Complete Coefficient Matrices

Definition A coefficient matrix is *complete* if the number of its columns equals the number of loops enclosing the array reference. Otherwise, the matrix is called *incomplete*.

If a dependence source R_1 has a diagonalizable and complete coefficient matrix, then the number (m) of the subscript equations must be greater than or equal to the number (l_1) of loops enclosing R_1. Using the transformation that diagonalizes the coefficient matrix, the subscript equations can be rewritten as follows.

$$\begin{array}{rcl} \tilde{a}_1 i_1 & = & \tilde{\psi}_1(j_1, j_2, \ldots, j_{l_2}) \\ \tilde{a}_2 i_2 & = & \tilde{\psi}_2(j_1, j_2, \ldots, j_{l_2}) \\ & \cdots & \\ \tilde{a}_{l_1} i_{l_1} & = & \tilde{\psi}_{l_1}(j_1, j_2, \ldots, j_{l_2}) \\ 0 & = & \tilde{\psi}_{l_1+1}(j_1, j_2, \ldots, j_{l_2}) \\ & \cdots & \\ 0 & = & \tilde{\psi}_m(j_1, j_2, \ldots, j_{l_2}) \end{array} \quad (5)$$

where each $\tilde{a}_k$ is a nonzero integer. We can then use the algorithm below to determine the mask predicate and the contact.

Algorithm 4.1

1. $\mathcal{F}_{int} := (\tilde{a}_1 \mid \tilde{\psi}_1) \wedge (\tilde{a}_2 \mid \tilde{\psi}_2) \wedge \cdots \wedge (\tilde{a}_{l_1} \mid \tilde{\psi}_{l_1})$, where "$|$" indicates "divides".

2. If $m > l_1$, then $\mathcal{F}_{zero} := (\tilde{\psi}_{l_1+1} = 0) \wedge \cdots \wedge (\tilde{\psi}_m = 0)$. Otherwise, $\mathcal{F}_{zero} :=$ `true`.

3. For k from 1 to l_1, $\tilde{i}_k := \tilde{\psi}_k / \tilde{a}_k$.

4. Substitute each $\tilde{i}_k$ into the loop bound constraints, i.e., ineqs. (2), and do:

$$\mathcal{F}_{lb} := \bigwedge_{k=1}^{l_1} (u_k \tilde{i}_{p(k)} + v_k \le \tilde{i}_k \le u'_k \tilde{i}_{p'(k)} + v'_k) \qquad (6)$$

5. Substitute each $\tilde{i}_k$ into the precedence constraints, i.e., formula (4), and do:

$$\mathcal{F}_{pre} := (\tilde{i}_1 < j_1) \vee (\tilde{i}_1 = j_1) \wedge (\tilde{i}_2 < j_2) \vee \cdots \vee (\tilde{i}_1 = j_1) \wedge \cdots \wedge (\tilde{i}_{l_c-1} = j_{l_c-1}) \wedge (\tilde{i}_{l_c} < j_{l_c}). \qquad (7)$$

6. $\mathcal{F} := \mathcal{F}_{int} \wedge \mathcal{F}_{zero} \wedge \mathcal{F}_{lb} \wedge \mathcal{F}_{pre}$.

7. Simplify $\mathcal{F}$.

8. The sought contact is $(\tilde{i}_1, \tilde{i}_2, \ldots, \tilde{i}_{l_1})$. $\square$

In the above, the predicate $\mathcal{F}_{int}$ indicates whether i_1, i_2, ..., i_{l_1} have integer solutions, given fixed $j_1, j_2, \ldots, j_{l_2}$. $\mathcal{F}_{zero}$ indicates whether the last $m - l_1$ equations $\psi_k = 0$ can be satisfied. $\mathcal{F}_{lb}$ indicates whether the integer solutions of $i_1, i_2, \ldots, i_{l_1}$ satisfy the loop bound constraints (i.e., ineqs. (2)). $\mathcal{F}_{pre}$ indicates whether $i_1, i_2, \ldots, i_{l_1}$ satisfies the precedence constraints (i.e., formula (4)). Recall that if some loops in the nest are serialized, the corresponding terms in formula (4) should disappear.

It is clear that the above algorithm is valid for arbitrary loop bound expressions. In the whole algorithm, only Step 4 involves loop bounds: It substitutes the solutions of i indices into the loop bound expressions and derives $\mathcal{F}_{lb}$. This step is valid for arbitrary loop bound expressions.

After $\mathcal{F}$ is formed, it should be simplified as much as possible to reduce the time for evaluating $\mathcal{F}$ at run time. For instance, we can eliminate those factors that are recognized as always true. A detailed discussion on such simplification is beyond the scope of this paper. Nonetheless, the following lemma gives an upper bound on the complexity of $\mathcal{F}$.

Lemma 1 *If the coefficient matrix,* $\mathbf{A}$, *of* R_1 *is both complete and diagonalizable, then* $\mathcal{F}$ *can be evaluated with no more than the following integer operations:* l_1 `mod` *operations,* l_1 *divisions,* $m + 2 \times l_1$ *comparisons,* $2 \times l_1 + m \times l_2$ *multiplications, and* $2 \times l_1 + m \times l_2$ *additions.*

[Proof] In the worst case, none of the components of $\mathcal{F}$ could be evaluated at compile time and therefore must be evaluated at run time. It takes at most $m \times l_2$ multiplications and the same number of additions to calculate $\tilde{\psi}$. Evaluating $\mathcal{F}_{int}$ takes at most l_1 `mod` operations. $\mathcal{F}_{zero}$ takes at most $m - l_1$ comparisons to evaluate. The vector $(\tilde{i}_1, \ldots, \tilde{i}_{l_1})$ can be determined with at most l_1 divisions. Evaluating $\mathcal{F}_{lb}$ takes at most $2 \times l_1$ multiplications and the same number of additions and comparisons, while evaluating $\mathcal{F}_{pre}$ takes at most $l_c \le l_1$ comparisons. □

Corollary 1 *In Lemma 1, if all the loop bounds are invariants of L, then the numbers of multiplications and additions can both be reduced to $m \times l_2$.*

Corollary 2 *In Lemma 1, if every coefficient a_i is either 1 or -1 and all the loop bounds are invariants of L, then $\mathcal{F}$ can be evaluated with at most $m \times l_2$ multiplications, $m \times l_2$ additions, and $m + 2 \times l_1$ comparisons.*

Example We derive $\mathcal{F}$ for the dependence from A(I1+I2, I2) to A(I1, I2-2) in Figure 1(a). Let i_1 and i_2 denote I1 and I2 in A(I1+I2, I2), and let j_1 and j_2 denote I1 and I2 in A(I1,I2-2). We have

$$
\begin{array}{rcl}
\text{Subscript eqs.} & : & i_1 = j_1 - j_2 + 2, \quad i_2 = j_2 - 2 \\
\text{Loop bounds} & : & (1 \le i_1 \le N) \wedge \\
 & & (i_1 + 1 \le i_2 \le N + 1) \\
\Rightarrow \mathcal{F}_{lb} & = & (1 \le j_1 - j_2 + 2 \le N) \wedge \\
 & & (j_1 - j_2 + 3 \le j_2 - 2 \le N + 1) \\
 & = & (j_2 \le j_1 + 1) \wedge (j_1 \le 2j_2 - 5) \\
 & = & (j_2 = j_1 + 1) \wedge (j_1 \ge 3) \\
\text{Precedence} & : & i_1 \le j_1 - 1 \\
\Rightarrow \mathcal{F}_{pre} & = & (j_1 - j_2 + 2 \le j_1 + 1) \\
 & = & (j_2 \ge 1) = True \\
\Rightarrow \mathcal{F} & = & \mathcal{F}_{lb} \wedge \mathcal{F}_{pre} \\
 & = & (j_2 = j_1 + 1) \wedge (j_1 \ge 3).
\end{array}
$$

From the subscript equations, the only contact is $(j_1 - j_2 + 2, j_2 - 2)$. □

Having formulated the mask predicates and the contacts, the compiler can generate synchronization code as follows. First, an event array, *ev*, is declared for every reference R_1 which is a source of data dependences. The number of dimensions of *ev* equals the number, l_1, of loops that enclose R_1. Note that even if a dependence source corresponds to several dependence sinks, it suffices to declare only one event array. Afterwards, `post` and `wait` operations are inserted in the code using the following algorithm.

Algorithm 4.2

Input : 1) An array reference R_1 in the loop nest L as specified in Section 3. Suppose the indices of the enclosing loops are named $i_1, i_2, \ldots, i_{l_1}$. The coefficient matrix of R_1 is diagonalizable and complete. 2) Array references that are recognized or assumed by the compiler as dependent on R_1.

Output : Synchronization code for every dependence from R_1.

Steps : 1. After the statement that issues R_1, insert "`post`(ev($i_1, i_2, \ldots, i_{l_1}$))".

2. For every dependence from R_1, do 2.1-2.2 in the following:

2.1. Use Algorithm 4.1 to formulate the mask predicate $\mathcal{F}(j_1, j_2, \ldots, j_{l_2})$ and the contact $(e_1, e_2, \ldots, e_{l_1})$, where $j_1, j_2, \ldots, j_{l_2}$ are the indices of the loops enclosing the sink reference.

2.2. Before the statement which issues the sink reference, insert "`if` $\mathcal{F}(j_1, j_2, \ldots, j_{l_2})$ `then wait`(ev($e_1, e_2, \ldots, e_{l_1}$))". □

The synchronization code in Figure 1(b) was generated following Algorithm 4.2.

Since Algorithm 4.1 is valid for arbitrary loop bound expressions, if all sources of loop carried dependences in L have complete coefficient matrices, then we can allow L to have arbitrary loop bound expressions.

5 Incomplete Coefficient Matrices

Again, consider a dependence source R_1. By definition, if indices of some loops that enclose R_1 do not appear in the subscripts of R_1, then the coefficient matrix of R_1 is incomplete. We call the missing indices the *implicit indices*, and call the rest the *explicit indices.*

After transforming an incomplete and diagonalizable coefficient matrix to the quasi-diagonal form, the subscript equations can be written as

$$
\begin{array}{rcl}
\tilde{a}_1 i_{y_1} & = & \tilde{\psi}_1(j_1, j_2, \ldots, j_{l_2}) \\
\tilde{a}_2 i_{y_2} & = & \tilde{\psi}_2(j_1, j_2, \ldots, j_{l_2}) \\
 & \cdots & \\
\tilde{a}_n i_{y_n} & = & \tilde{\psi}_n(j_1, j_2, \ldots, j_{l_2}) \\
0 & = & \tilde{\psi}_{n+1}(j_1, j_2, \ldots, j_{l_2}) \\
 & \cdots & \\
0 & = & \tilde{\psi}_m(j_1, j_2, \ldots, j_{l_2})
\end{array}
\qquad (8)
$$

where i_{y_k} are the n explicit indices, $n < l_1$. To formulate the mask predicate, we let $\mathcal{F}_{int}$ be the predicate $(\tilde{a}_1 \mid \tilde{\psi}_1) \wedge (\tilde{a}_2 \mid \tilde{\psi}_2) \wedge \cdots \wedge (\tilde{a}_n \mid \tilde{\psi}_n)$, as in Algorithm 4.1. Likewise, if $m < n$, we let $\mathcal{F}_{zero}$ be the predicate $(\tilde{\psi}_{n+1} = 0) \wedge \cdots \wedge (\tilde{\psi}_m = 0)$. If $m = n$, then we simply let $\mathcal{F}_{zero}$ be true. The remaining task is to formulate $\mathcal{F}_{lb}$ and $\mathcal{F}_{pre}$, and to determine the contacts and the closest contact.

Recall that, in the case of complete coefficient matrices, $\mathcal{F}_{pre}$ and $\mathcal{F}_{lb}$ can be derived by straightforward index substitutions. Here, we must eliminate implicit indices from ineqs. (2) and (4) without being able to substitute directly. This problem would be extremely difficult if we allowed arbitrary loop bound expressions. The following algorithm works for singly indexed bound expressions.

Algorithm 5.1

1. (*Initialize $\mathcal{F}_{lb}$*) For $1 \leq k \leq l_1$, let $\mathcal{F}_{lb}^{(k)}$ be $(u_k i_{p(k)} + v_k \leq i_k \leq u'_k i_{p'(k)} + v'_k)$. Let $\mathcal{F}_{lb}$ be $\bigwedge_{k=1}^{l_1} \mathcal{F}_{lb}^{(k)}$.

2. (*Initialize $\mathcal{F}_{pre}$*) For $1 \leq k \leq l_c$, let $\mathcal{F}_{pre}^{(k)}$ be $(i_1 = j_1) \wedge \ldots \wedge (i_k \leq j_k - 1)$. Let $\mathcal{F}_{pre}$ be $\bigwedge_{k=1}^{l_c} \mathcal{F}_{pre}^{(k)}$.

3. Substitute every explicit index i_{y_k} in $\mathcal{F}_{lb}$ and $\mathcal{F}_{pre}$ with its solution $\tilde{\psi}_k$.

4. For k from l_1 down to 1, if i_k is implicit, then do 4.1-4.7 in the following (to eliminate i_k from $\mathcal{F}_{lb}$ and $\mathcal{F}_{pre}$):

4.1. If i_k is the index of a common loop, then replace $i_k \leq j_k - 1$ in $\mathcal{F}_{pre}$ by $u_k j_{p(k)} + v_k \leq j_k - 1$, and replace $i_k = j_k$ in $\mathcal{F}_{pre}$ by $u_k j_{p(k)} + v_k \leq j_k$.

4.2. Transform all inequalities in $\mathcal{F}_{lb}$ whose only implicit index is i_k to the form of $\alpha \leq i_k$ or $i_k \leq \beta$. For all $\alpha \leq i_k$, take $LB^{(k)} = max(\alpha)$. For all $i_k \leq \beta$, take $UB^{(k)} = min(\beta)$.

4.3. Delete all inequalities $\alpha \leq i_k$ and $i_k \leq \beta$ found in Step 4.2 from $\mathcal{F}_{lb}$ (as they are no longer useful in discovering constraints on the j indices).

4.4. If $LB^{(k)}$ or $UB^{(k)}$ contains any j index, and if $LB^{(k)} \leq UB^{(k)}$ is not implied by current $\mathcal{F}_{lb}$, then add this inequality to $\mathcal{F}_{lb}$ as a new "$\wedge$" factor.

4.5. If u_k or u'_k is nonzero, replace $(u_k i_{p(k)} + v_k \leq i_k \leq u'_k i_{p'(k)} + v'_k)$ in $\mathcal{F}_{lb}$ by $(u_k i_{p(k)} + v_k \leq UB^{(k)}) \wedge (LB^{(k)} \leq u'_k i_{p'(k)} + v'_k) \wedge (u_k i_{p(k)} + v_k \leq u'_k i_{p'(k)} + v'_k)$. (This adds new constraints on $i_{p(k)}$ and $i_{p'(k)}$.)

4.6. In all other inequalities in $\mathcal{F}_{lb}$ of the form $\alpha \leq i_k$, where α contains an implicit index, replace i_k by $UB^{(k)}$. Likewise, in those of the form $i_k \leq \beta$, where β contains an implicit index, replace i_k by $LB^{(k)}$. (Those inequalities were derived in previous iterations of Step 4.)

5. (*Determine the contacts and the closest contact*) For h from 1 to l_c, do 5.1-5.4 in the following:

5.1. For k from h to l_1, do 5.1.1 and 5.1.2 in the following:

5.1.1. For explicit i_k, let $LB^{(k)}$ be the solution of i_k given by eqs. (8). For implicit i_k, if $u_k > 0$, then

$$LB^{(k)} := max(LB^{(k)}, u_k LB^{(p(k))} + v_k),$$

else if $u_k < 0$, then

$$LB^{(k)} := max(LB^{(k)}, u_k UB^{(p(k))} + v_k).$$

5.1.2. For explicit i_k, let $UB^{(k)}$ be the solution of i_k given by eqs. (8). For implicit i_k, if $u'_k > 0$, then

$$UB^{(k)} := min(UB^{(k)}, u'_k UB^{(p'(k))} + v'_k),$$

else if $u'_k < 0$, then

$$UB^{(k)} := min(UB^{(k)}, u'_k LB^{(p'(k))} + v'_k).$$

If $k = h$, then $UB_{save} := UB^{(k)}$, $UB^{(k)} := min(UB^{(k)}, j_k - 1)$.

5.2. The closest contact when $\mathcal{F}_{pre}^{(k)}$ is true is $cc^{(k)} = (UB^{(1)}, UB^{(2)}, \ldots, UB^{(l_1)})$.

5.3. The set of all contacts when $\mathcal{F}_{pre}^{(k)}$ is true is $SC^{(k)} = [LB^{(1)}, UB^{(1)}] \times [LB^{(2)}, UB^{(2)}] \times \ldots \times [LB^{(l_1)}, UB^{(l_1)}]$.

5.4. For implicit $i_{(k)}$, $UB^{(k)} := min(UB_{save}, j_k)$.

$\square$

Two interesting things can be observed from the result of the algorithm above. First, unlike in the case of complete matrices, we may now have multiple contacts. Second, we may even have different forms for the closest contact, depending on which of the predicates $\mathcal{F}_{pre}^{(k)}$ is true. For a flow or output dependence, only the closest contact needs to be maintained explicitly by `wait`. Other contacts will be maintained automatically because the reference instances issued at the dependence source are all write accesses to the same array element and the access order will be maintained by synchronization for the self output dependence. In contrast, for an anti dependence, every contact must be maintained explicitly by `wait`, because the reference instances at the dependence source are read accesses among which no order will be maintained. This issue will be discussed further in Section 7. The algorithm below generates synchronization code for dependence sources which have diagonalizable and incomplete coefficient matrices.

Algorithm 5.2

Input : 1) An array reference R_1 in the loop nest L as specified in Section 3. Suppose the indices of the enclosing loops are named $i_1, i_2, \ldots, i_{l_1}$. The coefficient matrix of R_1 is diagonalizable and incomplete. 2) Array references which are recognized or assumed by the compiler as dependent on R_1.

Output : Synchronization code for every dependence from R_1.

Steps : 1. After the statement which issues R_1, insert `post`$(\text{ev}(i_1, i_2, \ldots, i_{l_1}))$.

2. For every dependence from R_1, formulate the predicates $\mathcal{F}_{int}$, $\mathcal{F}_{zero}$, $\mathcal{F}_{lb}$ and $\mathcal{F}_{pre}^{(k)}$, $1 \leq k \leq l_c$. Let $\tilde{\mathcal{F}} = \mathcal{F}_{int} \wedge \mathcal{F}_{zero} \wedge \mathcal{F}_{lb}$.

3. If R_1 is a write, then for every flow or output dependence from R_1, do the following:

3.1. Before the statement which issues the sink reference, insert “**`if`** $\tilde{\mathcal{F}}$ **`then`**”.

3.2. For $1 \le k \le l_c$, if $\mathcal{F}_{pre}^{(k)}$ is not always false, do the following:

3.2a. Use Algorithm 5.1 to compute $cc^{(k)} = (e_1^{(k)}, e_2^{(k)}, \ldots, e_{l_1}^{(k)})$, the closest contact to be maintained when $\mathcal{F}_{pre}^{(k)}$ is true.

3.2b. If k is the smallest integer such that $\mathcal{F}_{pre}^{(k)}$ is not always false, then insert “`if` $\mathcal{F}_{pre}^{(k)}$ `then wait`(ev($e_1^{(k)}, e_2^{(k)}, \ldots, f_{l_1}^{(k)}$))”; otherwise insert “`elseif` $\mathcal{F}_{pre}^{(k)}$ `then wait`(ev($e_1^{(k)}, e_2^{(k)}, \ldots, f_{l_1}^{(k)}$))”.

3.3. Insert “`endif`” twice.

4. If R_1 is a read, then for every anti dependence from R_1, do the following:

4.1. Before the statement which issues the sink reference, insert “`if` $\mathcal{F}$ `then`”.

4.2. For $1 \le k \le l_c$, if $\mathcal{F}_{pre}^{(k)}$ is not always false, do the following:

4.2a. Use Algorithm 5.1 to compute $SC^{(k)}$, the set of all contacts to be maintained when $\mathcal{F}_{pre}^{(k)}$ is true.

4.2b. If k is the smallest integer such that $\mathcal{F}_{pre}^{(k)}$ is not always false, then insert “`if` $\mathcal{F}_{pre}^{(k)}$ `then`”, and for each contact $(e_1, e_2, \ldots, e_{l_1})$ in $SC^{(k)}$, insert “ `wait`(ev($e_1, e_2, \ldots, e_{l_1}$))”;

4.2c. If k is not the smallest integer required in Step 4.2b, then insert “`elseif` $\mathcal{F}_{pre}^{(k)}$ `then`”, and for each contact $(e_1, e_2, \ldots, e_{l_1})$ in $SC^{(k)}$, insert “`wait`(ev($e_1, e_2, \ldots, e_{l_1}$))”.

4.3. Insert “`endif`” twice. □

The following example illustrates Algorithms 5.1 and 5.2.

Example Take the doubly-nested loop in Figure 2(a). We need to consider 1) the anti dependence, δ_a, from A(I1,I2-1) to A(I2,I2); 2) the output dependence, δ_o, from A(I2,I2) to A(I2,I2); and 3) the flow dependence, δ_f, from A(I2,I2) to A(I1,I2-1).

(a) For δ_a, the coefficient matrix of the source is complete and quasi-diagonal. Following Algorithm 4.1, we can easily determine that (I2, I2 + 1) is the unique contact and that $\mathcal{F}_{\delta_a}$ = (I2 $\le$ N1) $\wedge$ (I2 $\le$ N2 $-$ 1) $\wedge$ (I2 $\le$ I1 $-$ 1) = (I2 $\le$ N2 $-$ 1) $\wedge$ (I2 $\le$ I1 $-$ 1).

(b) For δ_f, the coefficient matrix of the source is diagonalizable and incomplete. Transforming it to the quasi-diagonal form, we derive the subscript equations

$$i_2 = j_1, \quad 0 = j_1 - j_2 + 1,$$

where i_1 and i_2 denote I1 and I2 in the source, and j_1 and j_2 denote I1 and I2 in the sink. $\mathcal{F}_{zero}$ = (I1 $-$ I2 = $-$ 1) is immediately obtained. We now follow Algorithm 5.1 to derive $\mathcal{F}_{lb}$, $\mathcal{F}_{pre}$ and the closest contact.

1. $\mathcal{F}_{lb} := (1 \le i_1 \le N1) \wedge (1 \le i_2 \le N2)$.

2. $\mathcal{F}_{pre}^{(1)} := (i_1 \le j_1 - 1)$; $\mathcal{F}_{pre}^{(2)} := (i_1 = j_1) \wedge (i_2 \le j_2 - 1)$.

3. Substituting the explicit index i_2, we perform

$\mathcal{F}_{lb} := (1 \le i_1 \le N1) \wedge (1 \le j_1 \le N2)$;

$\mathcal{F}_{pre}^{(2)} := (i_1 = j_1) \wedge (j_1 \le j_2 - 1)$.

4.1. Since i_1 is implicit and it belongs to a common loop, we form

$\mathcal{F}_{pre}^{(1)} := (1 \le j_1 - 1)$;

$\mathcal{F}_{pre}^{(2)} := (1 \le j_1) \wedge (j_1 \le j_2 - 1)$.

4.2. $LB^{(1)} := 1$; $UB^{(1)} :=$ N1.

4.3. Delete $1 \le i_1 \le N1$ from $\mathcal{F}_{lb}$.

4.4. $1 \le N1$ does not contain any j index.

Skip 4.5-4.6. Cleaning up the predicates, we have $\mathcal{F}_{lb} = (j_1 \le N2)$, $\mathcal{F}_{zero} = (j_2 - j_1 = 1)$, $\mathcal{F}_{pre}^{(1)} = (2 \le j_1)$, and $\mathcal{F}_{pre}^{(2)} = (j_1 \le j_2 - 1)$.

5. In the first iteration (for h = 1), UB_{save} := N1; $UB^{(1)} := min(N1, j_1 - 1)$; $UB^{(2)} := j_1$;

$cc^{(1)} := (j_1 - 1, j_1)$; $UB^{(1)} := j_1$.

```
DO I1 = 1, N1
DO I2 = 1, N2
A(I2,I2) = ..
 ...
 .. = A(I1,I2-1)
ENDDO
ENDDO
```

(a)

```
DOALL I1 = 1, N1
DOALL I2 = 1, N2
IF ((I2.LE.N2-1) .AND. (I2.LT.I1))
   WAIT(EV1(I2,I2+1))
IF (I1.GE.2) WAIT(EV2(I1-1,I2))
   A(I2,I2) = ...
   POST(EV2(I1,I2))
      ...
IF ((I1.LE.N2).AND.((I2-I1).EQ.1))
THEN IF (I1.GE.2) WAIT(EV2(I1-1,I1))
     ELSEIF (I1.LE.I2-1)) WAIT(EV2(I1,I1))
     ENDIF
ENDIF
.. = A(I1,I2-1)
POST(EV1(I1,I2))
ENDDO
ENDDO
```

(b)

Figure 2: An Example Using Algorithms 5.1 and 5.2

In the second iteration (for h = 2), $UB^{(2)} := j_1$; $cc^{(2)} := (j_1, j_1)$.

(c) For δ_o, the coefficient matrix of the source is diagonalizable and incomplete. Similar to (b), we derive the closest contact (I1-1, I2) and $\mathcal{F}_{\delta_o} = (2 \le j_1)$.

(d) Following Algorithm 5.2, we insert synchronization as shown in Figure 2(b), where EV1 is used for δ_a and EV2 is used for both δ_f and δ_o. □

The following lemma gives an upper bound on the complexity of $\mathcal{F}$ for incomplete, diagonalizable coefficient matrices. For simplicity, we assume that the maximums and the minimums in Algorithm 5.1 can be determined at compile time. Therefore, for example, a compiler can determine which of $UB^{(k)}$ and $j_k - 1$ is greater.

Lemma 2 *If the coefficient matrix,* $\mathbf{A}$*, of* R_1 *is diagonalizable and incomplete, then* $\mathcal{F}_\delta$ *can be evaluated with no more than the following integer operations:* l_1 `mod` *operations,* l_1 *divisions,* $m + 4 \times l_1$ *comparisons,* $2 \times l_1 + m \times l_2$ *multiplications and* $2 \times l_1 + m \times l_2$ *additions.*

[Proof] Compared with the operations required in Lemma 1, here we perform additional two comparisons per iteration in Step 4 in Algorithm 5.1. This amounts to at most $2 \times l_1$ additional comparisons in total.

Corollary 3 *If the coefficient matrix,* $\mathbf{A}$*, of* R_1 *is diagonalizable and incomplete, and if all the loop bounds are invariants in* L*, then* $\mathcal{F}_\delta$ *can be evaluated with the same operations required for diagonalizable and complete matrices (*cf. Corollary 1*).*

[Proof] Invariant loop bounds make the additional comparisons in Lemma 3 unnecessary.

6 IF Statements

In the previous two sections, our discussion was limited to a loop nest L which contains no `if` statements. Ignoring the step control of DO loops, L had straight line code. In this section, we cover `if` statements and control flows. In flow analysis, each `do` loop will be represented as a cycle in the flow graph of L. Parallelization of a `do` loop breaks such a cycle. In order to reflect this fact, we obtain the *sequential skeleton* of L by deleting the `do` and `enddo` statements for every `do` loop that is to be parallelized. We then build a flow graph, *SSG*, of the sequential skeleton. For the sake of clarity, we assume in this section that backward control transfers do not exist. As a result, *SSG* is acyclic. Cycles will be discussed in Section 7.

```
DO I1 = 1, N                  DOALL I1 = 1, N
DO I2 = 1, N                  DOALL I2 = 1, N
IF (C) THEN                   IF (C) THEN
  ...                             ...
  A(I1,I2) = ..                 A(I1,I2) = ..
ELSE                            POST(EV1(I1,I2))
  ...                           POST(EV2(I1,I2))
  ...                         ELSE
  B(I1+1,I2) = ..                 ...
  ...                           B(I1+1,I2) = ..
ENDIF                           POST(EV1(I1,I2))
ENDDO                           POST(EV2(I1,I2))
ENDDO                         ENDIF
                              ENDDO
                              ENDDO
             (a)                           (b)
```

Figure 3: An Example of Branch-Induced POST's

The main problem caused by `if` statements is that an event variable may be posted on some control paths but not on others, which may cause some `wait` operations to busy-wait indefinitely. A straightforward solution to this problem is to add additional `posts` (called *branch-induced* `posts` in this paper) to ensure that the event variable will be posted on every possible control path.

Example The loop in Figure 3(a) gives a basic example, where a distinctive event array is used for each of the dependence sources A(I1,I2) and B(I1+1,I2). A branch-induced `post` is added to each of the two control paths.

Quite often, an array may be referenced on different paths using the same indexing subscripts. We call such identical references on different paths *clones*. For the sake of efficiency, we use the same event array for dependence sources that are clones. The following shows an example.

Example The loop in Figure 4(a) has two clones of A(I1,I2) reference. Both are dependence sources whose corresponding sink is the A(I1-1,I2-1) reference. The same event array EV1 is used for both sources.

The following is a formal definition of clones.

Definition Two dependence sources in L are *clones* if they are references to the same array, their enclosing loop nests have the same depths and the same loop bounds, their subscripts are identical, and they belong to different nodes of *SSG*, the flow graph of the sequential skeleton, in which neither node *dominates* or *post-dominates* the other.

In the above definition, we use the term "dominates" in the same sense as in flow analysis [ASU86]. We

```
DO I1 = 1, N          DOALL I1 = 1, N
DO I2 = 1, N          DOALL I2 = 1, N
IF (C) THEN           IF (C) THEN
  ...                   ...
  A(I1,I2) = ..         A(I1,I2) = ..
ELSE                    POST(EV1(I1,I2))
  ...                 ELSE
  ...                   ...
  A(I1,I2) = ..         A(I1,I2) = ..
ENDIF                   POST(EV1(I1,I2))
. = A(I1-1,I2-1)      ENDIF
ENDDO                 IF ((I1.GE.2).AND.(I2.GE.2))
ENDDO                    WAIT(EV1(I1-1,I2-1))
                      ENDDO
                      ENDDO
       (a)                     (b)
```

Figure 4: An Example of Clones

```
DO I1 = 1, N          DOALL I1 = 1, N
IF (C) THEN           IF (C) THEN
  ...                   ...
  DO I2 = 1, N          DOALL I2 = 1, N
  ...                   ...
  A(I1,I2) = ..         A(I1,I2) = ..
  ENDDO                 POST(EV1(I1,I2))
ELSE                    ENDDO
 ...                  ELSE
ENDIF                   DOALL I2 = 1, N
ENDDO                   POST(EV1(I1,I2))
                        ENDDO
                      ENDIF
                        ...
                      ENDDO
       (a)                     (b)
```

Figure 5: An Example of Complementary Loops

use the term "post-dominates" in the same sense as in [FOW87]: A node in a flow graph G post-dominates another if the former dominates the latter in the reverse graph of G.

Suppose a dependence source R is nested in several loops and some of the loops are on one of the branches of an IF statement. When we add branch-induced `posts` on the other branches, we must also produce some new DO loops on those branches to ensure that each branch-induced `post` is enclosed in a loop nest of the same depth and the same loop bounds as the loop nest enclosing R. The added DO loops are called *complementary loops*. We give a formal definition as follows.

Definition Suppose R is a dependence source in L and S is a code segment of L represented by a node σ in SSG. The DO loops that enclose R but not S are called the *complementary loops* of S (or σ) with respect to R.

Example In Figure 5(a), A(I1,I2) is a dependence source. In Figure 5(b), a branch-induced `post`, enclosed by a complementary loop (`doall` I2), is added to the `else` branch.

The following algorithm generates synchronization code for a loop nest L that has `if` statements.

Algorithm 6.1

1. Identify clone dependence sources in L.

2. Build a flow graph, SSG, for the sequential skeleton of L. 2. For each dependence source in L, declare an event array *ev* as specified in Section 4. Use the same *ev* for clones.

3. Follow Algorithm 4.2 to insert `post` and `wait`. The dependences from clone sources to a sink share the same `wait`.

4. For each dependence source R and its event array *ev*, call *Insert_a_Post*(*ev*, R, *Top*) in Figure 6 to insert branch-induced `posts`, where *Top* is the top node in SSG. □

```
procedure Insert_a_Post(ev, R, σ)

(σ is a node in SSG.)

if σ already contains a post on ev, return.
if any descendant of σ has a post on ev then
    for each descendant σ̃, call Insert_a_Post(ev, R, σ̃)
else determine the complementary loops of σ
    w.r.t. R, and insert a post (on ev) and the complementary loops. in the node σ.
endif
```

Figure 6: A Procedure Called by Algorithm 6.1

In Step 4 of the algorithm, we traverse SSG in a similar way to that in [MP87]. The recursive procedure *Insert_a_Post* in Figure 6 examines a node in SSG and inserts a branch-induced `post` and the complementary loops when necessary.

7 Discussion

Scalar dependences

So far we have ignored dependences between scalar references because scalars are usually either *privatized* or *expanded* into arrays to eliminate unnecessary dependences, such as those due to storage conflicts. If a scalar is neither privatized nor expanded and causes loop carried dependences, event `post` and `wait` can be inserted to enforce the dependences. Scalars can be viewed as a special case of diagonalizable and incomplete coefficient matrices. Therefore Algorithms 5.1 and 5.2 apply.

Constant distances of data dependences

In the cases where the distance of a dependence is known and constant, synchronization is easily inserted by our algorithms. First of all, the contacts would be unique and trivially determinable. The mask predicate would simply test whether an index vector displaced by the distance vector remains within the loop bounds.

Flow graphs with cycles

In Section 6, we considered a flow graph which is a DAG. To deal with a flow graph with cycles, reduce the maximum cycles to single nodes. The reduced graph becomes a DAG. Handling the reduced graph is straightforward, except that `posts` should not be inserted in a reduced node, lest some `wait` may terminate before the dependence is satisfied. Our solution is to add new nodes on the edges which leave the reduced nodes and insert `posts` as needed in the new nodes. However, since the practical importance of this problem is unclear, we do not pursue it further.

Complementary loops

In Algorithm 6.1, we did not handle nodes which have complementary loops in the most efficient way. It is obvious that a single event element suffices to indicate whether an `if` branch is taken in a particular loop iteration. Using this fact, we can avoid inserting complementary loops. A complete discussion is beyond the scope of this paper.

Dependence coverage

If one dependence *covers* others, then only one pair of `post` and `wait` is needed to preserve all those dependences. Some previous works [LAS87, MP87, CKS90] discussed dependence coverage in simple cases such as constant dependence distances. Using mask predicates, more complicated cases can be handled. A necessary condition for one dependence to cover another is that the mask predicate of the latter should imply that of the former. Sufficient conditions must take the control flow into account.

Anti dependences

As Section 5 shows, if the coefficient matrix of an anti dependence source is incomplete, multiple contacts may exist that require the insertion of multiple `waits` before the dependence sink. This is a common weakness among most existing synchronization mechanisms. Instead of performing a `post`, one improvement is to increment a shared counter after each read access at the dependence source. Correspondingly, instead of performing multiple `waits` at the dependence sink, the processor busy-waits until the shared counter accumulates to the number of contacts. Such a counter has been implemented on some research machines such as the Cedar multiprocessor [EPY89], but not on commercial multiprocessors. In real programs, we found anti dependences that do not require explicit synchronization because they can be covered by flow or output dependences. We hope that our further experiments can establish this as a common case.

Related work

Previous works on data synchronization concentrate mainly on devising synchronization mechanisms [Smi85, Sar88, ZY87, SY89]. Only a few works discuss compiler algorithms. The work in [MP87] deals mainly with single parallel loops. Recently, Tang, Yew and Zhu present an algorithm based on special counters [TYZ90]. Their paper considers a subset of our loop bounds and subscripts, but does not address `if` statements. They assume that each data word is associated with a counter, both of which may be accessed atomically. This may double the memory requirement of a program. The practical use of their present algorithm seems to be also limited by a few problems, especially the problem with `if` branches: they often make the access counts difficult to predict. Despite these problems, using counters has certain advantages; as we discussed earlier, some anti dependences may be handled more efficiently. Therefore, it is worthwhile to pursue a better use of counters.

8 Conclusion

We have presented algorithms which gather enhanced data dependence information that is unavailable from conventional analysis. Based on this new information, our algorithms automatically generate event variable synchronization for a wide range of array references. We believe the new information is also useful for other data synchronization mechanisms.

Event variable synchronization is quite flexible. `posts` and `waits` may be inserted in a variety of ways to perform correct synchronization. The strategy adopted in our algorithms may be called a “shortest waiting” strategy, as it allows execution to proceed whenever it is not hindered by data dependences. In this sense, our algorithms support the maximal parallelism. However, whether a “shortest waiting” strategy is the best one depends on the synchronization overhead, which in turn depends on the complexity of loop structures and array subscripts, as well as system factors. In fact, the formulation of mask predicates in this paper provides a basis for evaluating synchronization cost in a "shortest waiting" strategy. Our next goal is to conduct experiments on real programs, collecting data on synchronization complexity and on the validity of our assumptions on loop bounds and array subscripts.

9 Acknowledgement

The author thanks Kathryn McKinley and Chau-Wen Tseng for reading this paper and making helpful comments. He also thanks Merle Levy for her proofreading.

References

[AK87] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transaction on Programming Languages and Systems*, 9(4), October 1987.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPAN Symposium on Principles & Practice of Parallel Programming*, pages 21–31, March 1990.

[CSY90] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture*, June 1990.

[EPY89] P.A. Emrath, D.A. Padua, and P.-C. Yew. Cedar architecture and its software. In *Proceedings of 22nd Hawaii International Conference on System Sciences*, January 1989.

[For88] The Parallel Computing Forum. PCF Fortran language definition, 1 edition, August 1988.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley & Sons, 1978.

[Kum88] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9), September 1988.

[LAS87] Z. Li and W. Abu-Sufah. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers*, C-36(1):105–109, January 1987.

[MP87] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[Sar88] V. Sarkar. Synchronization using counting semaphores. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 627–637, July 1988.

[SLY90] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[Smi85] B. Smith. The architecture of HEP. In *Parallel MIMD Computation: HEP Supercomputer and Its Applications, J. S. Kowalik, ed.*, pages 41–45. The MIT Press, Cambridge, Massachusetts, 1985.

[SY89] H.-M. Su and P.-C. Yew. On data synchronization for multiprocessors. In *Proceedings of 16th Annual International Symposium on Computer Architecture*, pages 416–423, May 1989.

[TYZ90] P. Tang, P.-C. Yew, and C.-Q. Zhu. Compiler techniques for data synchronization in nested parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, July 1990.

[WB87] M.J. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2), April 1987.

[ZY87] C.-Q. Zhu and P.-C. Yew. A scheme to enforce data dependences on large multiprocessor systems. *IEEE Transactions on Software Engineering*, SE-13(6):726–739, June 1987.