Compiler Techniques for Data Synchronization in Nested Parallel Loops

Pei-Yi Tang\textsuperscript{1,2} \hspace{1cm} Pen-Chung Yew\textsuperscript{1} \hspace{1cm} Chuan-Qi Zhu\textsuperscript{1}

Department of Computer Science \hspace{1cm} Center for Supercomputing Research and Development \hspace{1cm} Computer Center
The Australian National University \hspace{1cm} University of Illinois at U-C \hspace{1cm} Fudan University
Canberra, ACT 2601 \hspace{1cm} Urbana, IL 61801 \hspace{1cm} Shanghai, China
Australia

Abstract

The major source of parallelism in ordinary programs is \texttt{do} loops. When loop iterations of parallelized loops are executed on multiprocessors, the cross-iteration data dependences need to be enforced by synchronization between processors. Existing data synchronization schemes are either too simple to handle general nested loop structures with non-trivial array subscript functions or insufficient due to the large run-time overhead.

In this paper, we propose a new synchronization scheme based on two data-oriented synchronization instructions: \texttt{synch_read(x,s)} and \texttt{synch_write(x,s)}. We present the algorithm to compute the ordering number, \textit{s}, for each data access. Using our scheme, a parallelizing compiler can parallelize a general nested loop structure with complicated cross-iteration data dependences. If the computations of ordering numbers cannot be done at compile time, the run-time overhead is smaller than the other existing run-time schemes.

1. Introduction

In recent years, MIMD multiprocessors and multicomputers have drawn much attention from both academic and industrial communities. According to Bell [1], 29 companies have built such machines since 1983, and 22 have put products on the market, for example Cray XMP, BBN Butterfly, Alliant FX/8, Convex C-2, Sequent, Encore, Intel iPSC, and INMOS Transputers. Large research projects such as Cedar [2] and RP3 [3] are indicative of the strong interests in larger multiprocessors systems in academia. It is believed that with multiple processors running concurrent tasks (processes) in parallel, MIMD machines can exploit more parallelism than SIMD vector processors.

\texttt{do} loops are a major source of concurrent tasks in programs. Each iteration of a loop can potentially be a concurrent task. If there are no dependences across iterations, they can be executed by multiple processors independently as parallel \texttt{do all} loop. If there are cross-iteration dependences, the loop iterations are either executed sequentially, or in parallel with the cross-iteration dependences being enforced by synchronization. In the latter case, the execution of the loop iterations can be fully or partially overlapped. This kind of parallel loops is called \texttt{do acr} loops [4]. The cross-iteration dependences can be control or data dependences. According to [5], the control dependences can be converted to data dependences. This paper addresses inter-processor synchronization to enforce data dependences for general nested \texttt{do acr} loops in shared-memory multiprocessors.

There have been some works on multiprocessor data synchronization for \texttt{do acr} loops [6] [7] [8] [9] [10]. Mikkil and Padua [6] and Wolfe [8] suggested inserting statement-level synchronization instructions such as \texttt{Set/Wait} and \texttt{Send/Wait} in the loop body to enforce data dependences. These schemes can, however, only handle constant-distance data dependences and are mainly used for singly nested \texttt{do acr} loops. Su and Yew [7] has also proposed a process-orientated data synchronization scheme for constant-distance data dependences. A recent empirical study on array subscripts and data dependences [11] shows that only small portion of data dependences (13.65\%) are of constant distance that can be determined at compile time. In other words, 86.35\% of data dependences have to be enforced by executing the relevant loops sequentially in the above schemes.

Unlike [6] [7] [8], Zhu and Yew [9] proposed a runtime data synchronization scheme based on the sequential access order in the original programs on each data element. They extended the Full/Empty tag of the memory word in Denelcor's HEP [12] to an integer. Data dependences are
enforced by applying Read-Test-Operation type instructions controlled by the original access order on each data element. The scheme can enforce any data dependencies at the cost of quite a large run-time overhead of accessing globally shared tables and barrier synchronizations. The data synchronization scheme proposed in this paper is similar to but different from that of [9] in the following respects:

1. [9] uses an operational approach to determine the access order for each data element at run time. The operational approach incurs a lot of run-time accesses to global shared-tables and barrier synchronization variables. The proposed scheme uses computational approach to compute the access orders at compile time (if all of the information such as subscript functions and loop bounds are available). If the computation cannot be carried out at compile time, it can be performed locally within each processor at run time. In large multiprocessor systems, the access time to the global shared-memory is far larger than the cycle time of the processors. The actual computation time is also small because the deepest nesting level of nested loops in most programs does not exceed 4. The run-time overhead of our scheme is much smaller than that of [9].

2. The proposed scheme can handle many-to-one mapping array references efficiently (an array reference has a many-to-one mapping if many instances of the enclosing loop index vectors map onto the same array element).

3. The proposed scheme can be used in general non-perfectly nested loops.

4. The proposed scheme does not unnecessarily enforce input data dependences, allowing more parallelism to be exploited.

The rest of the paper is organized as follows. In Section 2, we define access order and data dependences, and propose data synchronization instructions for their enforcement. In Section 3, we present algorithms to generate data synchronization instructions. We give our concluding remarks in Section 4.

2. Data Dependences and Synchronization Instructions

2.1. Access Order and Data Dependences

A variable (a scalar or an array element) can be accessed (read or written) several times in the original sequential program. The access order is a linear order, $\prec$, of all read and write accesses to the variable during the lifetime of the program ($A_i \prec A_j$ means that access $A_i$ is before access $A_j$, where $A \in \{R, W\}$). In the parallel execution of the program, the access order of each variable must be maintained, with the exception that the successive reads between two writes may be executed in any order. For example, in the program in Fig. 1, the array element $a(2,3)$ is accessed for seven times and the linear access order is shown in Fig. 2(a). The element $a(2,3)$ is first writ-

```
do i = 1, 3
  do j = 1, 3
    r1: $a(i,j) = \ldots$
    .
    $\ldots = a(k, i) + \ldots$
    $a(j, k) = \ldots$
  end do
end do
```

Fig. 1 An example of general nested loop
ten by reference $r3$ in loop iteration $(i,j,k)=(1,2,3)$. It is then overwritten by the same reference in loop iteration $(i, j, k)=(2, 2, 3)$ and so on. The linear access order is $W_1W_2W_3R_4R_5W_6R_7$.

The order between two successive accesses, at least one of which is a write, is called direct data dependence. The order from a write to a read is called flow dependence. The order from a read to a write is an anti-dependence. The order between two successive writes is an output dependence. When the program is executed in parallel, these three types of data dependences must be satisfied; otherwise, the parallel execution will have a different result. The order between two successive reads (sometimes called input dependence) need not be enforced in parallel execution. In fact, the transitive closure of direct dependence (called indirect dependence) is a partial order among all the accesses to the variable. It is the partial order rather than the original linear order that has to be enforced in parallel execution. The partial order of the dependences for array element $a(2,3)$ is shown in Fig. 2(b).

Among the three types of data dependences, only flow dependences are true dependences. The anti- and output dependences are caused by the repeated use of variables. If each assignment uses a new variable, the anti- and output dependences can be eliminated. However, this would require the expensive dynamic memory allocation [10]. In this paper, we consider the enforcement of all these types of data dependences.

2.2. Data Synchronization Instructions

To enforce the data dependences, synchronization instructions to coordinate data accesses from different processors are needed. There are three kinds of data synchronization instructions [7]: (1) statement-oriented instructions such as P/V operations on semaphores, Lock/Unlock in the Cray XMP, Advance/Wait in the Alliant FX/8, Set(Send)/Wait proposed in [6] [8], (2) process-oriented
instructions proposed in [7], and (3) data-oriented instructions such as Full/Empty tag scheme in Denelcor’s HEP [12], integer-key scheme in the Cedar machine [2] [9] and bit-map scheme proposed in [14].

In this paper, we propose two data-oriented synchronization instructions similar to Cedar synchronization instructions [9] called *sync_read* and *sync_write*. As in the Cedar machine, we assume that each synchronization data variable is associated with an integer word called *KEY*. The *KEY* is a time stamp to control when an access to the variable can be performed. The initial value of *KEY* is always 0. Each time the variable is accessed (either read or written), its *KEY* is incremented. We use *sync_read* and *sync_write* to replace the regular read and write in the original sequential program, respectively. To enforce data dependences, a *sync_write* will not be performed until the value of *KEY* equals the total number of previous accesses to the variable in the original sequential program. In this paper, when we refer to the temporal relation between data accesses using words “before” “after” “previous” “most recent” etc., we mean the linear order of accesses to the variable, ⪯, in the original sequential program. In our scheme, *sync_read* corresponding to successive reads without interleaving writes are allowed to be executed in any order. The semantics of *sync_read* and *sync_write* are shown in Fig. 3. In Fig. 3, x is the address or the name of a synchronization variable, s is an integer provided by the processor issuing the *sync_read* or *sync_write* and is used to be compared with the *KEY* before the read or the write can take place. For a write access, the *sync_write* can be performed only when the *KEY* equals s. For a read access, the *sync_read* can be executed only when the *KEY* is larger than s. The processor has to re-issue the same *sync_write* or *sync_read* instruction if the test condition is false. A *with* statement represents a critical region that can only be executed exclusively. It is assumed that the hardware in shared memory guarantees that the *sync_read* and *sync_write* instructions be executed indivisibly as in the Cedar machine [15], NYU Ultracomputer [20] and IBM RP3 [3].

The integer issued with a *sync_read* or a *sync_write*, s, is called the *ordering number* of the access. In order to enforce data dependences using *sync_write* and *sync_read* operations, the ordering number s is determined by the following rules:

**rule 1:**

For a write access, W, the ordering number, s(W), is equal to the total number of previous accesses.

**rule 2:**

For a read access, R, the ordering number, s(R), is equal to the ordering number of the most recent write access. If there is no previous write access, the ordering number is -1.

In rule 2, if there is no previous write access for a read access, any negative number for s will allow the *sync_read* to be executed because the initial value of *KEY* is 0. Here, it is simply -1.

The following theorem shows that all data dependences will be enforced in parallel execution using the proposed synchronization scheme.

**Theorem:**

All flow, anti- and output dependences of a variable will be satisfied in parallel execution using the *sync_write* and *sync_read* with the ordering numbers determined by rule 1 and rule 2.

**Proof:**

It is not difficult to prove, by induction, that all of the *sync_read* and *sync_write* will be eventually executed in parallel execution. The proof is straightforward and is omitted here. Let W_i and W_j be two write accesses to the same variable with W_i< W_j. From rule 1, we have s(W_i)< s(W_j). Because the value of *KEY* is incremented

---

1 ⪯ is the linear order of accesses to the variable in the original sequential program.
each time the variable is accessed, \( W_i \) must be executed before \( W_j \) in parallel execution and the output dependence from \( W_i \) to \( W_j \) will be satisfied. Consider a write, \( W_i \), and a read \( R_j \) with \( W_i \prec R_j \). We have \( s(W_i) \leq s(R_j) \) from rules 1 and 2. Since \( R_j \) will not be executed until \( s(R_j) < KEY \), i.e., \( s(W_i) < KEY \), and \( W_i \) is executed only when \( s(W_i) = KEY \), \( W_i \) must be executed before \( R_j \), and the flow dependence from \( W_i \) to \( R_j \) will be satisfied. Finally, consider \( R_i \) and \( W_j \) with \( R_i \prec W_j \). There are \( s(W_j) \geq 0 \) accesses before \( W_j \) in the original sequential program. Assume that \( W_j \) is executed before \( R_i \) in parallel execution. Since \( W_j \) is executed only when \( KEY = s(W_j) \) and the initial value of \( KEY \) is 0, there must be \( s(W_j) \) operations executed before \( W_j \) in the parallel execution. Since \( R_i \) is now executed after \( W_j \), there must be at least one operation, either \( W_k \) or \( R_k \) with \( W_k \prec W_j \) or \( W_j \prec R_k \) executed before \( W_j \) in the parallel execution. This means that the output dependence from \( W_j \) to \( W_i \) or the flow dependence from \( W_j \) to \( R_k \) would be violated. This is impossible from the above established conclusions. #QED

Fig. 2(b) also shows the ordering numbers, \( s \), for all the data accesses to \( a(1, 3) \). Notice that \( K_4 \) and \( K_5 \) have the same ordering number, because they have the same most recent write access, \( W_3 \). They are allowed to be executed in any order in parallel execution.

As long as the appropriate ordering number for each synch_write and synch_read is used, all of the statements in a nested loop can be executed in parallel. Theoretically, we can exploit parallelism with statement level granularity, i.e., each statement is assigned to a processor. In practice, however, such scheme would cause extremely high scheduling overhead. Hence, one iteration or a group of iterations are usually allocated to a processor at a time. Given a general nested loop like the one in Fig. 1, some loops may be identified as parallel doall loops by data dependence analysis [18]. The remaining loops can be executed either as sequential do loops or parallel doacr loops depending on the percentage of the computation of the loop body that can be overlapped. The problem to decide which loops to be executed as parallel loops and which loops as sequential loops is beyond the scope of this paper. Given a general nested loop, the data synchronization scheme presented in this paper is valid for any combination of parallel and sequential loops. For example, the nested loop in Fig. 1 has 9 loop iterations for loop body \( \{X\} \) with the enclosing index vectors \( i, j \), \( 1 \leq i, j \leq 3 \), and 27 loop iterations for loop body \( \{Y\} \) with the enclosing index vectors \( i, j, k \), \( 1 \leq i, j, k \leq 3 \). If it is profitable, our data synchronization scheme allows all nesting loops to be parallelized and all 36 loop iterations be treated as parallel tasks, eliminating any needs for barrier synchronization. We can use processor self-scheduling schemes [16] [17] to allocate all 36 loop iterations to the processors dynamically. The parallelized program is shown in Fig. 4. Notice that we use cobegin...coend construct to describe the parallelism between loop body \( \{X\} \) and doacr loop \( k \). Since reads and writes can be trivially distinguished in assignment statements, we use syntax \( \$[x ; s] \) to denote either synch_read\( (x,s) \) or synch_write\( (x,s) \). Further, the ordering

\[
\begin{align*}
dacr & i = 1, 3 \\
dacr & j = 1, 3 \\
& \begin{align*}
& \text{begin} \\
& \text{r1: } \$[a(i, j) ; s1(i, j)] = ... \quad \{X\} \\
& \quad \text{.} \\
& \quad \text{end} \\
& \text{doacr } k = 1, 4 \\
& \text{r2: } ... = \$[a(k, i) ; s2(i, j, k)] + ... \quad \{Y\} \\
& \quad \text{.} \\
& \quad \text{end doacr} \\
& \text{coend} \\
& \text{end doacr} \\
& \text{end doacr}
\end{align*}
\]

Fig. 4 Parallelized nested loop

number used to write a \((2, 3)\) by r1 in iteration \((i,j)=(2,3)\) is denoted by \(s1(2,3)\). According to the earlier discussion, \(s1(2,3)\) should be 2 (see Fig. 2).

3. Generating Data Synchronization Instructions

To parallelize nested do loops, it is straightforward to replace ordinary reads and writes with synch_reads and synch_writes. However, the key problem is to determine the ordering number for each data access. In this section, algorithms to compute ordering numbers are presented. The computation can be performed at compile time (if the needed information is available) or at run time.

3.1. General Nested Loops and Array Subscripts

The general form of a non-perfectly nested loop is shown in Fig. 5. We assume that all loops are normalized with lower bound and loop stride being 1. Both \(\{X\}\) and \(\{Y\}\) in the loop body can be any sequence of assignment statements and other instances of non-perfectly nested loops. Recursively, inner nested loop can be another non-perfectly nested loop. When a non-perfectly nested loop is executed in parallel, any of its do loops can be a parallel loop and \(\{X\}\), inner nested loop and \(\{Y\}\) in each loop body can also be executed in parallel (see Fig. 1 and Fig. 4).

\[
\begin{align*}
do & i = 1, N \\
& \{X\}; \\
& \text{inner nested loop}; \\
& \{Y\}; \\
& \text{end do}
\end{align*}
\]

Fig. 5 The general form of non-perfectly nested loop
Assume that an m-dimensional array A appears in a nested loop k times. Each appearance is called a reference, denoted by \( r_i, 1 \leq i \leq k \). The references are numbered in the lexical order. That is, if reference \( r_q \) is lexically before reference \( r_p \), then \( q < p \). If two references appear in the same assignment statement, then their lexical order can be determined by the order in the parse tree traversal of the statement. Suppose that an array reference is enclosed within \( n \) nesting loops with index variables \( i_1, i_2, \ldots, i_n \) (\( i_1 \) the outermost loop and \( i_n \) the innermost loop). Each reference has the form:

\[
A(\vec{f}_1, \vec{f}_2, \ldots, \vec{f}_n)
\]

where each subscript function \( f_i, 1 \leq i \leq m \), is a single-variable linear function \( ax_i+b_i \), where a and b are integer constants and \( i \in \{i_1, i_2, \ldots, i_n\} \). There is no restriction on which index variable is used in a subscript function. An index variable can appear in the subscript functions of different dimensions in the same or different references, leading to coupled subscript functions [19], which is one of the major causes for non-constant distance data dependences. Simple data synchronization schemes such as those in [6] [7] [8] cannot handle data dependences of non-constant distances.

3.2. Ordering Numbers for Write Accesses

We first consider the algorithms to compute the ordering numbers for write accesses. Suppose that reference \( r_p \) is a write reference and it is enclosed in \( s \) loops. The index vector of \( r_p \) is

\[
\hat{I}_p = (i_1, \ldots, i_c, i_{c+1}, \ldots, i_s)
\]

where \( c \leq s \). For a particular \( \hat{I}_p \), reference \( r_p \) accesses the array element

\[
A(\hat{f}_p(\hat{I}_p)) = A(b_1, b_2, \ldots, b_m)
\]

According to the rules described in Section 2.2, the ordering number is the total number of previous accesses to \( A(b_1, \ldots, b_m) \).

Through ordinary inexact data dependence tests\(^\text{2}\) such as GCD test and Banerjee's Inequality test [18], we can determine whether reference \( r_p \) is dependent on reference \( r_q \), \( 1 \leq q \leq k \). Such information becomes available after the data dependence analysis to determine what loops to parallelize [13]. If \( r_p \) and \( r_q \) are independent, then there is no previous access by \( r_q \) to the same array element \( A(b_1, \ldots, b_m) \). Let \( r_p \) depend on \( r_q \). The number of previous accesses by \( r_q \) is denoted by \( P_{wp}(\hat{I}_p) \). The total number of previous accesses, denoted as \( P_{wp}(\hat{I}_p) \), is

\[
P_{wp}(\hat{I}_p) = \sum_{r_p \text{ depends on } r_q} P_{wp}(\hat{I}_p)
\]

Next, we present the algorithm to compute \( P_{wp}(\hat{I}_p) \).

Assume that reference \( r_q \)

\[\begin{align*}
A(f_{q1}, f_{q2}, \ldots, f_{qm})
\end{align*}\]

is enclosed in \( t \) loops, where the outermost \( c \) (\( c \leq t, c \leq s \)) loops are common with reference \( r_p \). We denote the index vector of reference \( r_q \) as

\[
\hat{I}_q = (i_{c1}, \ldots, i_{ce}, j_{c+1}, \ldots, j_t)
\]

Let the upper bounds of the enclosing loops be

\[
\hat{UB} = (JUB_1, \ldots, JUB_v, JUB_{c+1}, \ldots, JUB_t).
\]

We assume that all loop upper bounds are known. We want to find the number of previous instances of \( \hat{I}_q \) in which reference \( r_p \) refers to the same array element \( A(b_1, b_2, \ldots, b_m) \). We have a system of \( m \) equations

\[
\begin{align*}
(f_{q1}(\hat{I}_q)) &= b_1 \\
(f_{q2}(\hat{I}_q)) &= b_2 \\
& \quad \vdots \\
(f_{qm}(\hat{I}_q)) &= b_m
\end{align*}
\]

where each left hand side is a single-variable linear function or an integer constant (i.e., a degenerated linear function). If it is a constant on the left hand side, we have to check whether the constants on both sides are equal. If they are not equal, there is no solution for the set of equations and \( P_{wp}(\hat{I}_p) = 0 \).\(^\text{3}\) In any case, the system of equations can be solved in \( O(m) \) time. It is important to note that some index variables of \( \hat{I}_q \) may not appear in the system of equations (2) and, therefore, are not determined after (2) is solved. It is for this reason that there may be many instances of \( \hat{I}_q \) before \( I_p \).

For the determined index variables in \( \hat{I}_q \) we need to do the following check to see if their values are integers within the corresponding loop bounds:

1. **integer check**: check whether all determined index variables are integers. If not, \( P_{wp}(\hat{I}_p) = 0 \).

2. **boundary check**: check whether all determined index variables are within their loop bounds. If not, then \( P_{wp}(\hat{I}_p) = 0 \).

3. **consistency check**: check whether all determined values of each index variable are equal. (It is possible that an index variable have many determined values because it may appear in several equations in (2).) If not, then \( P_{wp}(\hat{I}_p) = 0 \).

To determine the number of previous instances of \( \hat{I}_q \), we divide it into two subvectors: \( \hat{I}_q = (i_{c1}, \ldots, i_{ce}) \) and \( (i_{c+1}, \ldots, j_t) \). Likewise, \( \hat{I}_p \) is divided into \( \hat{I}_p = (i_1, \ldots, i_c) \) and \( (i_{c+1}, \ldots, i_s) \). \( UB \) is also divided into \( IUB = (JUB_1, \ldots, JUB_v) \) and \( JUB = (JUB_{c+1}, \ldots, JUB_t) \). Let \( X \) denote the number of instances of \( I_p = (i_1, \ldots, i_c) \) lexicographically before \( \hat{I}_p = (i_1, \ldots, i_c) \). \( \hat{I}_p = (i_1, \ldots, i_c) \) is lexicographically before \( (i_1, \ldots, i_c) \) if

\[\text{Even though } r_p \text{ depends on } r_q \text{ by inexact dependence tests, it is still possible that } P_{wp}(\hat{I}_p) = 0 \text{ for this particular instance of } r_p \text{ in iteration } I_p.\]

\[\text{181}\]
Algorithm 1

input: \( i \leq c, p, q, \Delta, \overrightarrow{I}_q, \overrightarrow{I}_p, \overrightarrow{IUB} \)

\[
\begin{align*}
\Delta &= (\delta_1, \ldots, \delta_c) \\
\overrightarrow{I}_q &= (i_1, \ldots, i_c) \\
\overrightarrow{I}_p &= (i_1, \ldots, i_c) \\
\overrightarrow{IUB} &= (\overrightarrow{IUB}_1, \ldots, \overrightarrow{IUB}_c) \\
\end{align*}
\]

output: the number of instances of \( \overrightarrow{I}_q = (i_1, \ldots, i_c) \) lexicographically before \( \overrightarrow{I}_p = (i_1, \ldots, i_c) \)

Function INSTANCE \((i, c, p, q, \Delta, \overrightarrow{I}_p, \overrightarrow{IUB})\)

\[
\begin{align*}
\delta_l &= \begin{cases} 
> & \text{if } i_l \text{ is determined and } i_l > i_l \\
< & \text{if } i_l \text{ is determined and } i_l < i_l \\
= & \text{if } i_l \text{ is determined and } i_l = i_l \\
* & \text{if } i_l \text{ is not determined by (2)} 
\end{cases} \\
\text{Given all the above information, } X \text{ can be computed by the function INSTANCE shown in Algorithm 1.}
\end{align*}
\]

Function INSTANCE inputs three subvectors:

\[
\Delta = (\delta_1, \ldots, \delta_c) \\
\overrightarrow{I}_q = (i_1, \ldots, i_c) \\
\overrightarrow{IUB} = (\overrightarrow{IUB}_1, \ldots, \overrightarrow{IUB}_c)
\]

where \( 1 \leq l \leq c \), and outputs the number of instances of subvector \( \overrightarrow{I}_q = (i_1, \ldots, i_c) \) lexicographically before subvector \( \overrightarrow{I}_p = (i_1, \ldots, i_c) \). Recall that some variables of \( \overrightarrow{I}_q = (i_1, \ldots, i_c) \) may not be determined by (2) and it is for this reason that there may be many instances of \( \overrightarrow{I}_q \). In particular, a function call of INSTANCE with \( \overrightarrow{I}_q = \overrightarrow{I}_p = \overrightarrow{IUB} \) produces the required value of \( X \).

When INSTANCE \((i, c, p, q, \Delta, \overrightarrow{I}_p, \overrightarrow{IUB})\) is evaluated, if \( \delta_l = "*" \) (i.e., \( i_l \) is determined and \( i_l = i_l \) we simply recursively call INSTANCE \((i+1, c, p, q, \Delta, \overrightarrow{I}_p, \overrightarrow{IUB})\) at the next level. If \( \delta_l = "<" \) (i.e., \( i_l < i_l \)) is lexicographically after \( \overrightarrow{I}_p = (i_1, \ldots, i_c) \) and the function call returns \( 0 \). If \( \delta_l = "<" \) (i.e., \( i_l < i_l \)), the function call returns the product of the loop upper bounds of the loops with \( \delta_l = "*" \), \( l+1 \leq l \leq c \). This is because as long as \( i_l \) is determined and \( i_l < i_l \), each undetermined \( i_l \), \( l+1 \leq l \leq c \), can take any integer value within its loop bound and all the resulting instances of \( \overrightarrow{I}_q \) are lexicographically before \( \overrightarrow{I}_p = (i_1, \ldots, i_c) \). In case all of \( i_l \), \( l+1 \leq l \leq c \), are determined, the function call returns \( 1 \). If \( \delta_l = "*" \) (i.e., \( i_l \) is not determined by (2)), \( i_l \) can take any value less than \( i_l \) or equal to \( i_l \) to make \( \overrightarrow{I}_q \) lexicographically before \( \overrightarrow{I}_p \). If \( i_l \) takes the values less than \( i_l \) (there are \( i_l-1 \) such values), we have the same situation as \( \delta_l = "<" \). If \( i_l \) takes the same value as \( i_l \), we have the same situation as \( \delta_l = "=" \). Algorithm 1 also shows how the boundary condition for \( l = c \) is handled.

Let us use the nested loop in Fig. 1 as an example to illustrate the computation of the ordering number for a write access. Consider the ordering number for the write reference \( r_1 \) (i.e., \( p=1 \)) in iteration \( I_p = (i, j) = (2, 3) \). We want to compute

\[
P_{1}(2, 3) = P_{11}(2, 3) + P_{21}(2, 3) + P_{31}(2, 3).
\]

The array element accessed is \( a(2, 3) \). To calculate \( P_{31}(2, 3) \), we have

\[
j' = 2 \quad k = 3
\]

\[\text{The inexact data dependence tests would show that } r_1 \text{ depends on } r_1, r_2 \text{ and } r_3.\]
where \( j \) and \( k \) are from \( \hat{I}_q = (i_1, j, k) \) (notice \( q = 3 \)). Among the variables of \( I_q \), \( i_r \) is not determined by (2). According to the notations we have introduced, \( I_q \) is divided into \( I_q = (i_1, j, k) \) (notice \( i_r \)). We have \( Y = 1 \). Comparing \( I_r = (i_1, j, k) \) with \( I_q = (i_2, k) \), we have \( \Delta = (*, k) \). \( X \) is computed by calling INSTANCE with \( l = 1 \). The function returns \( 2-1 = 1 \) plus the result of recursive call of INSTANCE at level \( l = 2 \). The latter returns 1. Therefore, \( X = 2 \) and \( P_{21}(2, 3) = X = 2 \). In calculating \( P_{21}(2, 3) \) (i.e., \( q = 2 \)), we have \( I_q = (3, j, k, 2) \).

Comparing \( \bar{I}_r = (3, j, 2) \) with \( \bar{I}_p = (2, 3) \), we have \( \Delta = (>^*, k) \). With \( \Delta = (>^*, k) \), the INSTANCE function call returns 0 and we have \( P_{21}(2, 3) = 0 \). For \( P_{11}(2, 3) \) (i.e., \( q = 1 \)), we have \( \Delta = (>^*, k) \). Because \( q - p \), the INSTANCE function call returns 0 also. The final result is \( P_{1}(2, 3) = 2 \).

Let us consider the time complexity of finding the ordering number for a write access. To solve the system of equations (2) and make integer, boundary and consistency checks takes \( O(m) \). To build the orientation vector \( \Delta \) (see (3)) takes \( O(c) \). It takes \( O(c - l + 1)^2 \) to compute \( \hat{Y} \). The complexity of Algorithm 1 is \( O\left(\frac{1}{2} (c-l+1)^2\right) \). In other words, it takes \( O\left(\frac{1}{2} c^2\right) \) to compute \( X \). Let the deepest loop nesting level be \( n \). We have \( c \leq l = n \). Since there are \( k \) references in the program, the total time complexity of finding the ordering number for a write access is

\[
O\left(\frac{1}{2} k n^2 + k n + k m\right)
\]

(4)

3.3. Ordering Numbers for Read Accesses

Now, we consider the ordering numbers for read accesses. Suppose that reference \( r_p \) is a read. In iteration \( I_p \), reference \( r_p \) reads array element

\[
\Lambda(I_p, I_q, I_r, I_s) = A(b_1, b_2, \ldots, b_m).
\]

According to rule 2 in Section 2.2, the ordering number for a read access is that of the most recent write access to the same array element. Two steps are used to find the most recent write access:

1. For each write reference \( r_q \) on which \( r_p \) depends, decide by inexact dependence tests, \( \Delta = (s, k) \) against the most recent access to the same array element.\(^5\)

2. From the candidate write accesses, select the most recent one by comparing their index vectors. If there are no candidate write accesses, the ordering number for \( r_p \) in \( I_p \) is -1.

The procedure MOST_RECENT in Algorithm 2 is used for step 1. All of the assumptions and notations of references \( r_p \) and \( r_q \) in Section 3.2 still hold except \( r_p \) is a read reference and \( r_q \) a write reference. As mentioned before, because some variables in \( I_q \) may not be determined by (2), there may be many instances of \( I_q = (i_1, \ldots, i_r) \) lexicographically before \( \bar{I}_p = (i_1, \ldots, i_r) \). Procedure MOST_RECENT finds the most recent one among them.

Procedure MOST_RECENT inputs vectors

\[
\Delta = (\delta_1, \ldots, \delta_c)
\]

\[
\bar{I}_p = (i_1, \ldots, i_r)
\]

\[
\bar{I}_q = (i_1, \ldots, i_l)
\]

\[
\bar{I} = (IUB_1, \ldots, IUB_c)
\]

and determines the values for undetermined variables in \( \bar{I}_q \) such that resulting \( \bar{I}_q \) is the most recent instance lexicographically before \( \bar{I}_p \). The procedure first searches for the

Algorithm 2

input: \( c, p, q \)

\[
\Delta = (\delta_1, \ldots, \delta_c)
\]

\[
\bar{I}_p = (i_1, \ldots, i_r)
\]

\[
\bar{I}_q = (i_1, \ldots, i_l)
\]

\[
\bar{I} = (IUB_1, \ldots, IUB_c)
\]

output: the most recent instance of \( \bar{I}_q = (i_1, \ldots, i_r) \)

lexicographically before \( \bar{I}_p = (i_1, \ldots, i_r) \)

Procedure MOST_RECENT (\( c, p, q, \Delta, \bar{I}_p, \bar{I}_q, \bar{I}, var \bar{I}_q \))

find the leftmost \( \delta_i \) such that \( \delta_i = '<' \) or \( \delta_i = '>' \):

if (no such \( \delta_i \) is found) then

/* all \( \delta_i \), 1 \leq i \leq c, are '=' or '*' */

if (\( q < p \)) then

\( i_w = i_r \) for all \( \delta_w = '*' \);

else

find the rightmost \( \delta_w \) such that \( \delta_w = '*' \) and \( i_w > 1 \):

if (no such \( \delta_w \) is found) then

return (no_instance);

else

\( i_w = i_w - 1 \);

\( i_r = i_r \) for all the remaining \( \delta_r = '*' \);

end if;

end if:

else

/* now, \( \delta_i \) (1 \leq i \leq c) is either '<' or a '>' */

if (\( \delta_i = '<' \)) then

\( i_w = i_w \) for all \( \delta_w = '*' \) with \( w < x \);

\( i_r = IUB_x \) for all \( \delta_r = '*' \) with \( x > z \);

else

/* \( \delta_w = '>' \)

find the rightmost \( \delta_w \) on the left of \( \delta_z \) (\( w < z \))

such that \( \delta_w = '*' \) and \( i_w > 1 \);

if (no such \( \delta_w \) is found) then

return (no_instance);

else

\( i_w = i_w - 1 \);

\( i_r = i_r \) for all \( \delta_r = '*' \) with \( x < w \);

\( i_r = IUB_y \) for all \( \delta_r = '*' \) with \( y > z \);

end if;

end if;

end if;

return (done);

\(^5\) Some write reference \( r_p \) may not have previous access for this particular instance of \( I_p \), even though \( r_p \) depends on \( r_q \) by inexact dependence tests.
Fig. 6 A general nested loop

leftmost component that is "<" or ">" in $A$. If no such component is found, all components of $A$ are "=" or "*". In this case, if $q < p$, the most recent instance of $I_q$ is equal to $I_p$. If $q > p$, the lowest instance of $I_q$ is $< I_p$. This can be obtained by searching the rightmost undetermined variable in $I_q$, 1 less than its counterpart in $I_p$ and keeping the remaining undetermined variables in $I_q$ equal to their counterparts in $I_p$. For example, suppose that $c = 3$ and the lower bounds of the three common loops are $IUB = (40, 40, 40)$. If $I_p = (i_1, i_2, i_3) = (7, 6, 8)$ and $I_q = (i_1', i_2', i_3') = (i_1, 5, i_3)$ (i.e., $A = (x, <, *)$), the most recent instance of $(i_1', 5, i_3')$ is $(7, 5, 40)$. If $q > p$, the most recent instance of $(i_1', 5, i_3')$ is $(7, 6, 7)$.

If the leftmost component of $A$ that is neither "=" nor "*" is found to be $\delta = < $, the most recent instance of $I_q$ can be obtained by replacing the undetermined components to the left of $i_w$, if any, with their counterparts in $I_p$. The undetermined components to the right of $i_w$, if any, should take the largest values. The loop upper bounds. For example, if $(i_1, i_2, i_3) = (7, 6, 8)$ and $(i_1', i_2', i_3') = (i_1', 5, i_3')$, (i.e., $A = (x, <, *)$), the most recent instance of $(i_1', 5, i_3')$ is $(7, 5, 40)$. If $\delta = > $, we need to see if we can find the rightmost $\delta = > $ on the left of $\delta$ such that $i_w$ is greater than 1. Then the most recent instance can be obtained by replacing the undetermined $i_w$ with $i_w - 1$ and the other undetermined components to its left, $i_x$, with the corresponding $i_x$. If there are any undetermined components to the right of $i_x$, simply replace them with the corresponding loop upper bounds. If no such $\delta = >$ can be found, there is no previous instance of $(i_1', \cdots, i_x')$.

After the most recent instance of $I_q = (i_1', \cdots, i_x')$, has been found, it can be concatenated with the largest instance of the second subvector $(i_{x+1}, \cdots, i_j)$ to form $I_q$. The largest instance of $(i_{x+1}, \cdots, i_j)$ is obtained by assigning the corresponding loop upper bounds to its undetermined components, if any.

Before we present the algorithm in step 2 to select the most recent write access among the candidates obtained in step 1, we need to describe the nesting tree of a program. A program can be regarded as a sequence of assignment statements and nested loops defined in Fig. 5. The nesting relationship of all of the loops and references in a program can be represented by an ordered nesting tree, where the root node represents the entire program, each intermediate node a do loop and each leaf node a reference. If loop $i$ or a reference is directly nested in another loop $j$, the node for loop $i$ or the reference is a child of the node for loop $j$. The do loops and the references in the outermost level are children of the root node. There is a total order among the children of each node determined by the lexical order in the program. For example, loop $11$ encloses loops $12$ and $14$ in Fig. 6. Loop $12$ is lexically before loop $14$. Therefore, loop $14$ is the last child of loop $12$. There are five references in the program in Fig. 6. The nesting tree of the program is shown in Fig. 7. It is not difficult for a compiler to generate such nesting trees.

Suppose that there are $u$ candidates from write references $r_{q_1}, \cdots, r_{q_u}$ with index vectors $I_{q_1}, \cdots, I_{q_u}$ respectively. Each candidate is a write access and is represented by the reference-index-vector pair $(r_{q_i}, I_{q_i})$. Let $Q$ be the set of such a candidates $Q = \{ (r_{q_1}, I_{q_1}), \cdots, (r_{q_u}, I_{q_u}) \}$. Function SELECT in Algorithm 3 is used to select the most recent element from $Q$ using the nesting tree of the program.

SELECT is a recursive function with three parameters: $Q$, $NP$ and $l$. $Q$ is the set of candidates, $NP$ is the pointer to the root of a subtree considered. $l$ is the level of the root of the subtree. The level of the root of the nesting tree is 0. The most recent write access among all the $u$ candidates can be obtained by calling SELECT($Q$, rootptr, $1$-0), where $Q = \{ (r_{q_1}, I_{q_1}), \cdots, (r_{q_u}, I_{q_u}) \}$ and "rootptr" is the pointer to the root of the nesting tree.
Algorithm 3  
inputs: $l$: nesting level of the subtree  
        $Q$: set of candidates  
        $NP$: pointer to the subtree  
output: the most recent access from $Q$

Function SELECT($Q$, $NP$, $l$)  
Let $C$ be the last child of the node pointed 
        by $NP$ with descendents in $Q$;  
        $Q_1 = \{ A \in Q \mid A$ is a descendent of $C \}$;  
if ($|Q_1|=1$) then  
        return($Q_1$);  
else  
        $NP = \text{pointer to } C$;  
        $Q_2 = \{ (r_q,t_q) \in Q_1 \mid$ the $l+1$-th component of $t_q$ 
        has the largest value $\}$;  
if ($|Q_2|=1$) then  
        return($Q_2$);  
else  
        return(SELECT($Q_2$, $NP$, $l+1$));  
end if;  
end if; 

There are two reductions of $Q$ in function SELECT. Let $C$  
be the last child of the node pointed by $NP$ with descendents in $Q$. Recall that the children of a node is ordered by their lexical order in the program. Set $Q$ is first reduced to the set of the descendents of $C$. The rationale behind this reduction is to seek the most recent instance, we need only to consider the descendents of the last child by the lexical order. If $Q_1$ is a singleton set, we have the answer; otherwise, we continue with the second reduction. The second reduction is from $Q_1$ to the set of accesses (denoted as $Q_2$) with largest index value for the $l+1$-th component in their index vectors. The reason for this reduction is that the references with a larger index value of the common loop are after those with smaller index values. If there is still more than one access in $Q_2$, function SELECT is recursively invoked at the next level.

Let us look at an example. Suppose that there are candidates, from references $r_1, \cdots, r_5$ in the program in Fig. 6 and the corresponding index vectors are $I_1=(3,2), I_2=(4,8,7), I_3=(3,2,1), I_4=(4,5)$ and $I_5=(4,4)$. Originally, we have $Q=\{(r_1,f_1), \cdots, (r_5,f_5)\}$ and $NP$ points to the root node, $r$, of the nesting tree (see Fig. 7). After the first reduction, $Q_1$ is the same as $Q$ because the root has only one child. After the second reduction, $Q_1$ is reduced to $Q_2=\{(r_2,f_2),(r_3,f_4),(r_5,f_5)\}$. In SELECT($Q_2$, $NP$, $1$), $Q_2$ is reduced to $Q_1=\{(r_4,f_2),(r_5,f_5)\}$ after the first reduction. After the second reduction, $Q_1$ is further reduced to $Q_2=\{(r_4,f_4)\}$. Thus, the most recent instance among the five candidates is reference $r_4$ with index vector $I_4=(4,5)$.

Let us estimate the time complexity to find the most recent write access to the same array element for a read access. To solve the system of equations (2) takes $O(m)$.

The complexity of Algorithm 2 is $O(c)$. If there are $k \leq k$ write references, the total time to find the $u \leq k$ candidates for the most recent write access is $O(k(c+m))$. Let $n$ be the deepest nesting level. Since $c \leq n$, $O(k(c+m))=O(k(n+m))$. The complexity of Algorithm 3 is calculated as follows. The process of finding the set of children that are ancestors of the references in $Q$ takes $O(un)$ because $|Q|=u$ and the height of the nesting tree is $n$. To build $Q_1$ for the first reduction takes also $O(un)$. To build $Q_2$ for the second reduction takes $O(2u)$. In the worst case, the function SELECT needs to be recursively called $n$ times with the diminishing tree height. Since $u \leq k$, the overall time complexity of Algorithm 3 is $O(\frac{1}{2}n(2kn+2k))$. Thus, the time to find the most recent write access to the same array element is $O(kn^2+2kn+km)$. If the ordering number of that write access is not available, another $O(\frac{1}{2}kn^2+kn+km)$ (see (4)) is needed. Hence, in the worst case, the time complexity of computing the ordering number for a read access is

$$O(\frac{3}{2}kn^2+3kn+2kn)$$

(5)

Algorithms to compute the ordering numbers both for write and read accesses have been presented. If all of the information such as loop bounds, coefficients and constants in subscript functions are available at compile time, all these ordering numbers can be computed and stored in the local memory of each processor before the parallel program starts. If this is not the case, the computations of ordering numbers have to be performed at run time. Notice that all these runtime computations can be done locally in each processor and no accesses to global shared memory are needed. It is important to note that the array dimension, $m$, the deepest loop nesting level, $n$, and the number of references, $k$, are all very small in real programs. As shown in [11], 92-42% of array references are one-dimensional or two-dimensional. The deepest loop nesting level $n \leq 4$ is quite common in most programs. Therefore, the time for local computation of ordering numbers will be much less than the global accesses to shared tables and synchronization barriers in [9] when the number of processors is large.

4. Concluding Remarks

By executing asynchronous tasks in parallel, multiprocessors can explore more parallelism in programs than vector processors. The major source of concurrent tasks in ordinary programs are loop iterations. Most existing multiprocessors can only execute the parallel loops whose loop iterations are independent (called doall loops). To execute parallel doall loops correctly, the cross-task data dependences need to be enforced by synchronization between processors. The previously proposed data synchronization schemes are either too simple to handle general nested loop structures with non-trivial array subscript functions [6] [8] [7], or will incur too much run-time overhead [9] [10].
We have proposed a new synchronization scheme based on two data-oriented synchronization instructions: synch_read(x,s) and synch_write(x,s). We presented algorithms to compute the ordering number, s, for each data access. Using our scheme, a parallelizing compiler can parallelize not only a singly-nested loop but also a general nested loop structure with cross-iteration data dependences. If the computations of ordering numbers cannot be done at compile time, the run-time overhead is very small (compared to other run-time schemes [9] [10]) because each processor can perform the computation locally and no global accesses to shared tables and synchronization barriers are needed.

References


