Execution of Regular DO Loops on Asynchronous Multiprocessors

Pei Ouyang

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012-1185

Abstract

This paper studies issues concerning parallel execution of regular Fortran DO loops on an asynchronous shared-memory multiprocessor, where each iteration is the basic unit to be executed by a single processing element. An iteration is a dependent predecessor of another iteration if execution of the latter iteration has to wait until execution of the former iteration has completed. During the execution of a DO loop, an iteration will pass through four states, namely, idle, pending, ready, and finished states. An iteration is idle if none of its dependent predecessors have completed; an iteration is pending if some of its dependent predecessors have completed, but not all; an iteration is ready if all its dependent predecessors have completed, but itself has not; otherwise, an iteration is finished. In addition, an iteration without any dependent predecessors is called an initial iteration, which can only have ready and finished states. Via describing an execution scheme, this paper studies the characteristics of Fortran DO loops which are related to the efficiency of the execution. Specifically, this paper investigates (1) the number of initial iterations, (2) the maximum number of ready iterations at any instances during the execution, (3) the maximum number of pending iterations at any instances during the execution, (4) a hash function to disperse different pending iterations, and (5) the parallel execution time.

\footnote{This research was partially supported by NSF under grant number CCR-89-6949.}

\footnote{Author’s electronic addresses: ouyang@spunky.cs.nyu.edu, (212) 998-3083.}
1 Introduction

As DO loops account for the major parallelism in Fortran programs, how to execute a DO loop efficiently in parallel environments is an important issue. Researches on executing DO loops have been done for various environments, including systolic arrays[6], vector processors[2,8], VLIW processors[1,11], synchronous multiprocessors[3], and asynchronous multiprocessors[7,9].

There are many DO loops where the dependence distances between iterations can be determined at compiling time[6,9]. Let us call this type of DO loops regular DO loops. This paper studies issues concerning executing regular Fortran DO loops on asynchronous shared-memory multiprocessors such as Ultracomputer[4]. Although executing DO loops on environments such as vector processors and VLIW processors has appealing results, the execution of DO loops on asynchronous shared-memory multiprocessors is interesting because which might be the only available machines. In addition, when the DO loops contain IF statements whose conditions are hard to be accurately predicted true or false at compiling time, it is hard to execute DO loops efficiently on vector processors and VLIW processors.

The execution scheme used in this paper is quite straightforward: each execution unit is an iteration and is scheduled to be executed by a free processor greedily. Although the concept of the scheme is simple, several nontrivial issues have to be considered, namely, what is the space needed to implement such scheme, how does one execution unit inform its dependent successors in an efficient way, and how to determine if the parallel execution is superior to the sequential execution. Such issues of executing regular Fortran DO loops on asynchronous shared-memory multiprocessors cannot be found in the existent literatures and are the subjects of this paper.

The organization of this paper is as follows. Section 2 defines the model and terms used in our discussion, as well as presents an overview of our execution scheme. In section 3, the data structure used in our scheme is examined in detail. Specifically, the number of initial, ready, and pending iterations are calculated, and a hash function to disperse different pending iterations are presented. Section 4 shows the time needed to execute a DO loop under our execution scheme. Finally, section 5 gives a conclusion.

2 Background

The loops that will be considered in this paper are normalized DO loops as below:

\[
\begin{align*}
\text{DO } & I_1 = 1, \ U_1 \\
\text{DO } & I_2 = 1, \ U_2 \\
& \vdots \\
\text{DO } & I_n = 1, \ U_n \\
\text{loop body} \\
\text{ENDDO} \\
& \vdots \\
\text{ENDDO}
\end{align*}
\]

Our execution scheme takes each iteration as the basic unit to be scheduled for execution. In other words, there are totally \( \prod_{i=1}^{n} U_i \) execution units in the above loop. For convenience of our presentation, an iteration will be represented by its values of induction variables. For example, iteration \( [i_1, i_2, \ldots, i_n] \) means the iteration when induction variable \( I_k \) is equal to \( i_k \), for \( 1 \leq k \leq n \). Furthermore, the nested DO loop will be modelled by an iteration space and several dependence vectors. The iteration space corresponding to the above loop is the Cartesian space \( [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n] \), and a dependence vector \( d_i = [d_{i_1}, \ldots, d_{i_n}] \) for the above loop is used to describe that iteration \( [s_1, \ldots, s_n] \) must be executed after iteration \( [s_1 - d_{i_1}, \ldots, s_n - d_{i_n}] \).

We call iteration \( [s_1, \ldots, s_n] \) the dependent predecessor of iteration \( [s_1, \ldots, s_n] \). In our execution scheme, an iteration can be executed only after all its dependent predecessors have been completed. During the execution of a DO loop, an iteration will pass through four states, namely, idle, pending, ready, and finished states. An iteration is idle if none of its dependent predecessors have completed; an iteration is pending if some of its dependent predecessors have completed, but not all; an iteration is ready if all its dependent predecessors have completed, but itself has not; otherwise, an iteration is finished. In addition, an iteration without any dependent predecessors is called an initial iteration, which can only have ready and finished states. It is useful to represent an iteration space and its associated dependence vectors as a dependence graph \( G = (V, E) \), where each vertex in \( V \) corresponds to an iteration in the iteration space, and \( < v_1, v_2 > \) is in \( E \) if the iteration corresponding to \( v_1 \) is a dependent predecessor of the iteration corresponding \( v_2 \). A longest path of \( G \) is a path \( p = v_0 v_1 \ldots v_l \) such that \( v_i \in V \) for \( 0 \leq i \leq l, \ < v_i, v_{i+1} > \in E \) for \( 0 \leq i \leq l - 1 \), and \( l \) is the maximum over all such paths. Note that the

---

Footnote 1: If the execution time of a single iteration is too small when compared to the synchronization time, several iterations can be grouped into an execution unit[5]. This is another subject of researches and we will not discuss it here.
parallel execution time of a DO loop can be expressed as a function of the length of the longest path.

**Example 1** Consider the following program:

\[
\begin{align*}
&\text{DO } I_1 = 1, 17 \\
&\text{DO } I_2 = 1, 17 \\
&\quad a(I_1, I_2) = a(I_1 - 1, I_2 - 3) \text{ or } a(I_1 - 3, I_2 - 1) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

The associated iteration space is \([1,17] \times [1,17]\), and the associated dependence vectors are \([1,3]\) and \([3,1]\). In figure 1, the iteration space is represented by a \(17 \times 17\) grid. All the iterations shown in figure 1(a) by \(\Box\)'s are initial iterations, and all the iterations shown in figure 1(b) by \(\bigcirc\)'s can be at ready state simultaneously. Note that the number of \(\bigcirc\)'s in figure 1(b) is also the maximum number of iterations at any instances during the execution. Finally, also shown in figure 1(b) by a dashed line is one of the DO loop's longest paths, which have the length equal to 8.

Now let us give an overview of our execution scheme. The execution scheme uses the following three "pools" to store iterations:

- **INIT**: a data structure used to store initial iterations.
- **READY**: a FIFO queue used to store ready iterations.
- **PENDING**: a data structure used to store pending iterations.

Note that an iteration can be stored by its induction variables instead of the whole loop body. The execution scheme is as follows. A free processor first tries to fetch an iteration from INIT to execute. If INIT is empty, then an iteration is fetched from the READY. If READY is also empty, then the corresponding nested DO loop has been finished. Whenever a processor finishes executing an iteration \(ci\), it installs each dependent successor \(si\) as follows:

**Algorithm 1**

- Try to find the entry for \(si\) in PENDING;
- if \(si\) is not in PENDING then
  - if \(si\) has no other dependent predecessors then
    - install \(si\) in READY;
  - else
    - install \(si\) in PENDING;
- endif
- else
- At the \(si\) entry in PENDING, mark \(ci\) finished;
- endif

if \(si\) has no other unfinished dependent predecessors then
move \(si\) from PENDING to READY;
endif
endif

The above execution scheme is quite naive but seems unavoidable in the execution of nested DO loops under asynchronous shared-memory multiprocessor environments. However, some techniques can be imposed on this execution scheme to improve the performance. As mentioned in section 1, the iterations can be grouped into larger execution units to compromise between computation and synchronization times. In addition, the data structure READY can be implemented as a priority queue where the priority of a ready iteration is a function of its expected execution time, the number of its dependent successors, the length of the longest path to an iteration without dependent successors, and so forth. This paper will not discuss these techniques, yet which can be applied to our scheme easily when needed.

3 Properties of Initial, Ready, and Pending Iterations

In this section, characteristics of initial, ready, and pending iterations are studied. First of all, our execution scheme needs to identify initial iterations and put them in the data structure INIT. A naïve method to find all such iterations would be sweeping through each of the iterations to check if it has dependent predecessors. This is described by the following algorithm:

**Algorithm 2**

- for each iteration \([I_1, \ldots, I_n]\) in the iteration space \([1, U_1] \times \cdots \times [1, U_n]\) do
- for each dependence vector \(d_i = [d_{i1}, d_{i2}, \ldots, d_{in}]\), for \(1 \leq i \leq m\), check if \([I_1 - d_{i1}, \ldots, I_n - d_{in}]\) is within the range \([1, U_1] \times \cdots \times [1, U_n]\);
- if any dependence vector makes the condition satisfied, then the iteration \([I_1, \ldots, I_n]\) should not be put in INIT;
- otherwise, \([I_1, \ldots, I_n]\) should be put in INIT;
- end
- end */ for */

In spite of the simplicity of the above algorithm, the time needed to execute the algorithm is \(O(m \prod_{i=1}^{n} U_i)\). Much time can be saved if we can avoid scanning non-initial iterations. Therefore, we need to know which iterations are initial iterations. From Example 1 in Section 2, it can be observed that initial iterations...
locate at the “borders” of the iteration space. For example, consider an iteration space $[1, 10] \times [1, 10]$. For a dependence vector $[2, 0]$, the set of iterations $\{(x, y) | 1 \leq x \leq 2, 1 \leq y \leq 10\}$ will not depend on any other iterations via the dependence vector $[2, 0]$. Hence this set of iterations will consist of all the initial iterations if there is only one dependence vector $[2, 0]$. Suppose we have another dependence vector $[1, -3]$, then the set of iterations $\{(x, y) | 1 \leq x \leq 1, 1 \leq y \leq 10\} \cup \{(x, y) | 1 \leq x \leq 10, 8 \leq y \leq 10\}$ will not depend on any other iterations via the dependence vector $[1, -3]$. As an initial iteration does not depend on any other iterations via any dependence vectors, the set of initial iterations for the dependence vectors $[2, 0]$ and $[1, -3]$ is $\{(x, y) | 1 \leq x \leq 2, 1 \leq y \leq 10\} \cap (\{(x, y) | 1 \leq x \leq 1, 1 \leq y \leq 10\} \cup \{(x, y) | 1 \leq x \leq 10, 8 \leq y \leq 10\})$. The following theorem formally state which iterations are initial iterations:

**Theorem 1** Let $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$ be the iteration space, $d_i = [d_{i1}, d_{i2}, \ldots, d_{im}]$, for $1 \leq i \leq m$, be $m$ dependence vectors. Let

$$I_{ij} = \begin{cases} \{[e_1, \ldots, e_n] \in S | 1 \leq e_j \leq d_{ij}\} & \text{if } d_{ij} > 0 \\ \emptyset & \text{if } d_{ij} = 0 \\ \{[e_1, \ldots, e_n] \in S | U_j + d_{ij} < e_j \leq U_j\} & \text{if } d_{ij} < 0 \end{cases}$$

Then an iteration $k$ is in $I = \bigcap_{i=1}^m \bigcup_{j=1}^n I_{ij}$ if and only if iteration $k$ is an initial iteration in $S$.

**proof.** If $k = [k_1, \ldots, k_n]$ is in $I$, we show that it is impossible that $k$ depends on any other iterations in the iteration space $S$. Suppose on the contrary that $k$ depends on some other iteration, then there must exist an iteration $k' = [k'_1, \ldots, k'_n]$ such that $k = k' + d_s$ for some fixed $s$. Since $k \in \bigcap_{i=1}^m \bigcup_{j=1}^n I_{ij}$, $k$ must be in $I_{st}$ for some fixed $t$. Since $k = k' + d_s$, we have $k_t = k'_t + d_{st}$.

However,

- if $d_{st} > 0$, then $k'_t = k_t - d_{st} \leq d_{st} - d_{st} = 0$;
- if $d_{st} = 0$, then $I_{st} = \emptyset$; $k$ cannot be in $I_{st}$;
- if $d_{st} < 0$, then $k'_t < k_t - d_{st} \leq U_t + d_{st} - d_{st} = U_t$.

These cases imply that $k$ cannot depend on any other iterations in $S$, contradicting to our assumption. Hence we conclude that $k$ does not depend on any other iterations in $S$.

Conversely, if $k \notin I$, then $k \notin \bigcup_{j=1}^n I_{sj}$ for some fixed $s$. In other words, $k \notin I_{sj}$ for all $1 \leq j \leq n$. Let us consider the following cases:

- if $d_{sj} > 0$, then $d_{sj} < k_j \leq U_j$;
- if $d_{sj} = 0$, then $1 \leq k_j \leq U_j$;
- if $d_{sj} < 0$, then $1 \leq k_j \leq U_j + d_{sj}$.

Define $k' = [k_1 - d_{s1}, \ldots, k_n - d_{sn}]$. It is clear that $k'$ must be in the iteration space $S$ and $k = k' + d_s$. That is, $k$ depends on $k'$. This completes our proof. □
An algorithm generating all the initial iterations is described next. For clarity, let us describe the algorithm by an example first. Consider an iteration space $[1, 10] \times [1, 10] \times [1, 10]$ and dependence vectors $d_1 = [0, 2, 3]$, $d_2 = [1, -1, 2]$ and $d_3 = [3, 1, 1]$. Let $d_i$ be denoted by $[d_{i1}, d_{i2}, d_{i3}]$ for $1 \leq i \leq 3$. The algorithm recursively divide each dimension into regions until it reaches the last dimension. Initially, dimension 1 is divided into three regions $[1, 1]$, $[2, 3]$, and $[4, 10]$ according to $d_{11}$, $d_{21}$ and $d_{31}$. By doing this way, we have

- For each iteration in the subspace $[1, 1] \times [1, 10] \times [1, 10]$, it may depend on other iterations only via $d_1$;
- For each iteration in the subspace $[2, 3] \times [1, 10] \times [1, 10]$, it may depend on other iterations only via $d_1$ or $d_2$;
- For each iteration in the subspace $[4, 10] \times [1, 10] \times [1, 10]$, it may depend on other iterations via $d_1$, $d_2$, or $d_3$.

With dimension 1 restricted to the region $[1, 1]$, dimension 2 will be divided, using $d_{12}$ only, into $[1, 2]$ and $[3, 10]$. By doing this way, we have

- For each iteration in the subspace $[1, 1] \times [1, 2] \times [1, 10]$, it may not depend on any other iterations;
- For each iteration in the subspace $[1, 1] \times [3, 10] \times [1, 10]$, it may depend on other iterations only via $d_1$.

Finally, with dimension 1 and 2 restricted to the region $[1, 1]$ and $[1, 2]$ respectively, initial iterations in the region $[1, 1] \times [1, 2] \times [1, 10]$ are generated. Table 1 summarizes the generation procedure for this example and Algorithm 3 describes the general procedure.

**Algorithm 3** Let $[1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$ be the iteration space, $d_i = [d_{i1}, d_{i2}, \ldots, d_{im}]$, for $1 \leq i \leq m$, be $m$ dependence vectors. The algorithm will generate the initial iteration indices by recursive calls to the procedure $\text{iter}(k, D)$. Let $x_1, \ldots, x_n, y_1, \ldots, y_n$ be global variables. At the main routine, $\text{iter}(1, \{d_1, \ldots, d_m\})$ is called.

**procedure** $\text{iter}(k, D)$

/* $k$ is the depth of the loop under considered */
/* $D$ is a set containing dependence vectors */
if $k = n$ then
   $x_n = \max\{\{1\} \cup \{U_1 + d_{in} + 1 \mid d_i \in D, d_{in} \leq 0\}\}$
   $y_n = \min\{\{U_1\} \cup \{d_{in} \mid d_i \in D, d_{in} \geq 0\}\}$
   generate iterations in the Cartesian space $[x_1, y_1] \times \cdots \times [x_n, y_n]$;
else
   $T = \{[f(d_{ik}), d_i] \mid d_i \in D, d_{ik} \neq 0, |d_{ik}| < U_k\}$
   $f(d_{ik}) = \begin{cases} d_{ik} & \text{if } d_{ik} > 0 \\ U_k + d_{ik} & \text{if } d_{ik} < 0 \end{cases}$
   $D' = \{d_i \in D \mid U_k < d_{ik} \leq 0\}$;
   $\text{finished} = \text{false}$;
   $y_n = 0$;
   while (not finished) do
      $x_k = y_k + 1$;
      if $T = \emptyset$ then
         $y_k = U_k$;
      else
         $y_k = \text{smallest}(T)$; /* smallest(T) is the smallest value among all $f(d_{ik})$'s in the elements of T. */
         iter($k+1, D'$);
      endif
      if $T = \emptyset$ then
         $\text{finished} = \text{true}$;
      else
         for each $[f(d_{ik}), d_i] \in T$ that $f(d_{ik}) = y_k$ do
            if $d_{ik} > 0$ then
               insert $d_i$ into $D'$;
            else /* $d_{ik} < 0$ */
               delete $d_i$ from $D'$;
            endif
         endfor
         remove $[f(d_{ik}), d_i]$ from $T$;
      endif
   endwhile
end while
endif
endprocedure

The above algorithm is faster than algorithm 1 because those iteration indices which do not belong to INIT are not generated. However, algorithm 3 does waste time for sweeping through “empty blocks” when the last dimension $n$ is empty. In this case, algorithm 3 can be revised so that the role of dimension $n$ is replaced by a nonempty dimension $k$ where $x_k$ is always less than or equal to $y_k$. In addition, generating the INIT need not be accomplished at compiling time. Algorithm 3 can be updated to fit into a run-time self-scheduling scheme, which eliminates the space needed for INIT.

Next we consider the space requirement for the data structure READY. To determine the space requirement, we have to know the maximum number of ready iterations at any instances during the execution:

**Theorem 2** Let $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$ be the iteration space, $d_i = [d_{i1}, d_{i2}, \ldots, d_{im}]$, for $1 \leq i \leq m$, be $m$ dependence vectors. Then the maximum number of ready iterations at any instances during the execution is less than or equal to $\min\{\prod_{i=1}^{m} U_{j_i} - \prod_{j=1}^{m} (U_{j_i} - |d_{ij_i}|) \mid 1 \leq i \leq m\}$.
Table 1: Generating initial iterations for the iteration space \([1,10] \times [1,10] \times [1,10]\) and dependence vectors \(d_1 = [0, 2, 3], d_2 = [1, -1, 2]\) and \(d_3 = [3, 1, 1]\).

<table>
<thead>
<tr>
<th>dimension 1</th>
<th>dimension 2</th>
<th>dimension 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>([d_{11}, d_{21}, d_{31}])</td>
<td>([d_{12}])</td>
<td>([d_{13}, d_{14}])</td>
</tr>
<tr>
<td>([d_{13}, d_{23}])</td>
<td>([d_{13}])</td>
<td>([d_{13}, d_{15}])</td>
</tr>
<tr>
<td>([2,3])</td>
<td>([1,2])</td>
<td>([1,10])</td>
</tr>
<tr>
<td>([d_{12}, d_{22}])</td>
<td>([d_{23}])</td>
<td>([d_{23}])</td>
</tr>
<tr>
<td>([1,2])</td>
<td>([1,2])</td>
<td>([1,2])</td>
</tr>
<tr>
<td>([1,10])</td>
<td>([1,10])</td>
<td>([1,10])</td>
</tr>
<tr>
<td>([4,10])</td>
<td>([d_{12}, d_{22}, d_{32}])</td>
<td>([d_{23}, d_{33}])</td>
</tr>
<tr>
<td>([d_{13}, d_{23}, d_{33}])</td>
<td>([d_{23}, d_{33}])</td>
<td>([d_{23}, d_{33}])</td>
</tr>
<tr>
<td>([1,1])</td>
<td>([1,1])</td>
<td>([1,1])</td>
</tr>
</tbody>
</table>

proof. Let \(I_{ij}\) be the same as in theorem 1. Then for each dependence vector \(d_i\), \(\cup_{j=1}^{n} I_{ij}\) is the set of iterations which do not depend on any other iterations via \(d_i\). In addition, note that when each iteration completes, at most one iteration can be activated via dependence vector \(d_i\). Therefore, the maximum number of ready iterations at any instances during the execution is less than or equal to \(\min \{ | \cup_{j=1}^{n} I_{ij} | \} \) \(\forall i \leq m\), where \( | \cup_{j=1}^{n} I_{ij} | \) denotes the size of the set \(\cup_{j=1}^{n} I_{ij}\).

Without loss of generality, we assume that \(d_{ik} \geq 0\) for \(1 \leq i \leq m\), and \(1 \leq k \leq n\) below in computing the value of \( | \cup_{j=1}^{n} I_{ij} | \). According to the definition of \(I_{ij}\), we have \(\cup_{j=1}^{n} I_{ij} = \cup_{j=1}^{n} \{x_1, \ldots, x_n \in S | 1 \leq x_j \leq d_{ij}\} = \{x_1, \ldots, x_n \in S | 1 \leq x_j \leq d_{ij}\} = \{x_1, \ldots, x_n \in S | 1 \leq x_j \leq d_{ij}\}\). Therefore, we have \( | \cup_{j=1}^{n} I_{ij} | = \prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - d_{ij})\). This completes our proof.

From Theorem 2, it is enough to allocate \(c \min \{ \prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - d_{ij}) \} \) \(\forall i \leq m\) unit space to READY, where \(c\) is a constant representing the space need for each entry of iterations. Note that the value obtained in Theorem 2 also represents the maximum number of processing elements that can be used simultaneously under our execution scheme.

Now let us determine the space required by PENDING. To do this, we have to know the maximum possible number of pending iterations at any instances during the execution.

**Theorem 3** Let \(S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]\) be the iteration space, \(d_i = [d_{i1}, d_{i2}, \ldots, d_{in}]\), for \(1 \leq i \leq m\), be \(m\) dependence vectors. Then the number of pending iterations at any instances during the execution cannot exceed \(\sum_{i=1}^{m} (\prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - d_{ij}))\).

proof. When an iteration becomes pending, it must be “activated” by one of its dependence predecessors. For a dependence vector \(d_i\), the number of pending iterations that are activated by \(d_i\) cannot exceed \(\prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - d_{ij})\), as can be seen from Theorem 2. Therefore, the total number of pending iterations cannot exceed \(\sum_{i=1}^{m} (\prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - d_{ij}))\).

From the above theorem, it is enough to allocate \(c \sum_{i=1}^{m} (\prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - d_{ij}))\) unit space to PENDING, where \(c\) is a constant representing the space need for each entry of iterations.

In addition to deciding the space bound for PENDING, we also need an access scheme to accomplish the action “try to find the entry for \(si\) in PENDING” in algorithm 1. The following algorithm describes the access scheme:

**Algorithm 4** Let \(c = [c_1, \ldots, c_n] = [\sum_{i=1}^{m} d_{i1}, \ldots, \sum_{i=1}^{m} d_{in}]\). Assume that \(c_i \leq U_i\) for \(1 \leq i \leq n\), which should account for most cases in practice. We will represent PENDING as a hash table with entries indexed by \(1, 2, \ldots, P\), where \(P\) is equal to \(\prod_{j=1}^{m} U_j - \prod_{j=1}^{m} (U_j - c_j)\). Then an iteration \([x_1, \ldots, x_n]\) will be assigned to a bucket of the entry indexed by \(r\), where \(r\) is computed as below (assume that \(c_j \geq 0\) for \(1 \leq j \leq n\) for clarity):\(^2\)

```c
/* Find the initial iteration \([y_1, \ldots, y_n]\) which is the ancestor of \([x_1, \ldots, x_n]\) via dependence vector \(c\) */
\(a = \min \{ \lfloor \frac{x_i - 1}{d_{ij}} \rfloor | 1 \leq j \leq n, c_j \neq 0 \};\)
\(p = \min \{ \lfloor \frac{y_i - 1}{c_j} \rfloor | 1 \leq j \leq n, c_j \neq 0 \};\)
\([y_1, \ldots, y_n] = [x_1 - a * c_1, \ldots, x_n - a * c_n];\)
/* find \(s\), the sum of sizes from block 1 to block \(p\), where block \(i\) is the Cartesian space \([c_{i-1} + 1, U_i]\) */
\(s = \sum_{i=1}^{p} (U_j - c_j) \prod_{j=1}^{m} (U_j - d_{ij});\)
/* find \(t\), the address of \([y_1, \ldots, y_n]\) within block \(p\), */
```

\(^2\)We will define \(\sum_{e \in \emptyset} f(s) \equiv 0\) and \(\prod_{e \in \emptyset} f(s) \equiv 1\).
where block \( p \) is the Cartesian space \([c_1 + 1, U_1] \times \cdots \times [c_p + 1, U_p] \times [1, c_p] \times [1, U_{p+1}] \times \cdots \times [1, U_n] \) * 

let \( [i_1, \ldots, i_n] = \{y_1 = c_1, \ldots, y_{p-1} = c_{p-1}, y_{p}, \ldots, y_n\} \); 

let \([v_1, \ldots, v_n] = \sum_{i=1}^{n} (i - 1) \prod_{j=i+1}^{n} v_j \); 

t = s + t + 1; 

Each of the initial iterations for the vector \( c \) will be mapped to a unique number in the range from 1 to \( P \), which can be observed from the fact that the algorithm in essence just divides the initial iterations for dependence vector \( c \) into at most \( n \) \( n \)-dimensional “blocks”, and then orders the iterations in each block according to row-major order. With this access scheme, it is expected that each entry of the hash table will usually store only one iteration. This is because in most cases, all the dependent predecessors of iteration \( I + d_1 + \ldots + d_m \) have iteration \( I \) as a dependent ancestor. Hence when iteration \( I \) is pending, iteration \( I + d_1 + \ldots + d_m \) cannot become pending as it requires at least one of its dependent predecessor be finished, which in turn requires iteration \( I \) be finished. An entry of the hash table may contain more than one iterations only when those iterations near the iteration space boundaries are being executed and some components of dependence vectors are negative. For example, for the iteration space \([1, 10] \times [1, 10] \times [1, 10] \) and dependence vectors \([0, 1, -2], [1, -2, 1], \) and \([1, 0, 2] \), iteration \([2, 2, 2] \) and iteration \([4, 1, 3] \) may be at the pending states simultaneously.

Finally, to support the above access scheme, we prove that the number of entries in PENDING is less than or equal to the bound we got in Theorem 3, i.e., \( \prod_{i=1}^{n} U_i - \prod_{i=1}^{n} (U_i - \lfloor \sum_{j=1}^{m} |d_{ij}| \rfloor) \) \( \leq \sum_{i=1}^{n} (\prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - \lfloor d_{ij} \rfloor)) \).

We need a lemma first:

**Lemma 4** Let \([1, U_1] \times \cdots \times [1, U_n] \) be the iteration space, \( d_1 = [d_{11}, d_{12}, \ldots, d_{1n}] \) and \( d_2 = [d_{21}, d_{22}, \ldots, d_{2n}] \) be two dependence vectors. Also assume that \( U_j \geq |d_{1j}| + |d_{2j}| \) for \( 1 \leq j \leq n \). Then we have \( \prod_{i=1}^{n} U_i - \prod_{i=1}^{n} (U_i - \lfloor d_{ij} \rfloor) \leq \prod_{i=1}^{n} (\prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - \lfloor d_{ij} \rfloor)) \).

**Proof.** See appendix.

With this lemma, we now show that the number of entries in PENDING is less than or equal to the bound we got in theorem 3:

**Theorem 5** Let \( S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n] \) be the iteration space, \( d_i = [d_{i1}, d_{i2}, \ldots, d_{in}] \), for \( 1 \leq i \leq m \), be \( m \) dependence vectors. Assume that \( U_j \geq \sum_{i=1}^{m} |d_{ij}| \) for \( 1 \leq j \leq n \). Then we have \( \prod_{i=1}^{n} U_j - \prod_{i=1}^{n} (U_j - \lfloor \sum_{i=1}^{m} |d_{ij}| \rfloor) \leq \sum_{i=1}^{m} (\prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - \lfloor d_{ij} \rfloor)) \).

**Proof.**

\[
\prod_{i=1}^{n} U_j - \prod_{i=1}^{n} (U_j - \lfloor \sum_{i=1}^{m} |d_{ij}| \rfloor) \\
\leq (\prod_{i=1}^{n} U_j - \prod_{i=1}^{n} (U_j - \lfloor d_{ij} \rfloor)) \\
+ (\prod_{i=1}^{n} U_j - \prod_{i=1}^{n} (U_j - \lfloor \sum_{i=1}^{m} |d_{ij}| \rfloor)) \\
\leq (\prod_{i=1}^{n} U_j - \prod_{i=1}^{n} (U_j - \lfloor d_{ij} \rfloor)) \\
+ (\prod_{i=1}^{n} U_j - \prod_{i=1}^{n} (U_j - \lfloor \sum_{i=1}^{m} |d_{ij}| \rfloor)) \\
\leq \cdots \\
\leq \sum_{i=1}^{m} (\prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - \lfloor d_{ij} \rfloor))
\]

**4 Execution Time of DO Loops**

In this section, we will consider when the parallel execution is superior to the sequential execution of the nested DO loops. Because of the synchronization cost, parallel execution of a nested DO loop is not necessarily faster than sequential execution of the same loop. Therefore, the compiler should estimate both parallel and sequential time to make the right choice.

Let \( t \) denote the average execution time of an iteration, \( s \) denote the synchronization time required by algorithm 1, and \( l \) denote the number of iterations on the longest path in the dependence graph. Then the sequential execution time is \( t \prod_{i=1}^{n} U_i \) and the parallel execution time is \( (t + ms)l \), where \( n \) is the depth of the nested DO loop, \( U_i \)'s are the upper bounds of induction variables, and \( m \) is the number of dependence vectors. Therefore, parallel execution is preferred to sequential execution when \( (t + ms)l < t \prod_{i=1}^{n} U_i \), i.e., when \( \frac{ms \cdot l}{t} < \prod_{i=1}^{n} U_i - 1 \).

The remaining question now is how to find the value of \( t \), that is, the length of the longest path in the dependence graph. First of all, let us define the set of source iterations, \( C \), for the longest path. For each iteration \( s_{ij} = [s_{1j}, \ldots, s_{nj}] \in C, s_{ij} \) could only be either 1 or \( U_j \). Specifically, \( s_{ij} \) could be 1 only when there exists some \( d_{ij} > 0 \), \( s_{ij} \) could be \( U_j \) only when there exists some \( d_{ij} < 0 \). If for some fixed dimension \( j \) and all dependence vectors \( d_i \), \( d_i \)'s are all zeros, then dimension \( j \) can be regarded as nonexistent in solving this problem. The following theorem tells us how to find the upper bound of the longest path:

**Theorem 6** Let \( S = [1, U_1] \times \cdots \times [1, U_n] \) be the iteration space, \( d_i = [d_{i1}, \ldots, d_{in}] \), for \( 1 \leq i \leq m \), be \( m \) dependence vectors. In addition, let \( C = \{s_1, s_2, \ldots, s_h\} \)
be the set of source iterations. Then the length of the longest path in the dependence graph cannot exceed 
\( \max\{v_1, \ldots, v_n\} \), where \( v_i \) is obtained by solving the following integer programming problem:

\[
\begin{align*}
\text{max} & \quad v_i = x_1 + x_2 + \ldots + x_m \\
\text{subject to} & \quad 1 \leq s_{ik} + \sum_{j=1}^{m} x_j d_{jk} \leq U_k \quad \text{for} \quad 1 \leq k \leq n \\
& \quad x_j \geq 0 \quad \text{for} \quad 1 \leq j \leq m
\end{align*}
\]

proof. In a dependence graph, any iteration reachable from \( s_i \) can be expressed as \( s_i + x_1 d_1 + x_2 d_2 + \ldots + x_m d_m \), where \( x_j \) is greater than or equal to 0 for \( 1 \leq j \leq m \). According to the definition of the longest path, all the iterations in a longest path must be within the iteration space \( S \). Specifically, the “sink” iteration of the longest path must be within the iteration space \( S \) also. That is,

\[
1 \leq s_{ik} + \sum_{j=1}^{m} x_j d_{jk} \leq U_k \quad \text{for} \quad 1 \leq k \leq n
\]

Therefore, the length of the longest path starting from \( s_i \) in the dependence graph cannot exceed the value \( v_i \) defined by the above integer programming problem.

Now we show that considering only the source iterations in \( C \) is sufficient. Suppose there is a longest path from \( a = [a_1, \ldots, a_n] \) to \( a + \sum_{j=1}^{m} x_j d_j \), where \( 1 < a_k < U_k \) for some \( k \). Let us consider the following cases:

- If \( 1 < a_k < a_k + \sum_{j=1}^{m} x_j d_{jk} \leq U_k \), then there must exist some \( d_j \) such that \( d_{jk} > 0 \). Consequently, there must exist some \( s_i \in C \) such that \( s_{ik} = 1 \). In addition, it is obvious that \( 1 < U_k + \sum_{j=1}^{m} x_j d_{jk} < U_k \).
- If \( 1 \leq a_k + \sum_{j=1}^{m} x_j d_{jk} < a_k < U_k \), then there must exist some \( d_j \) such that \( d_{jk} < 0 \). Consequently, there must exist some \( s_i \in C \) such that \( s_{ik} = U_k \). In addition, it is obvious that \( 1 < U_k + \sum_{j=1}^{m} x_j d_{jk} < U_k \).
- If \( 1 < a_k = a_k + \sum_{j=1}^{m} x_j d_{jk} < U_k \), then we have \( 1 = 1 + \sum_{j=1}^{m} x_j d_{jk} < U_k \) and \( 1 < U_k + \sum_{j=1}^{m} x_j d_{jk} = U_k \).

All these cases imply that we can find a longest path with its source iteration in \( C \) and having the length no less than \( x_1 + \ldots + x_m \). This completes our proof. □

Some comments follows. First, since the leftmost nonzero entry of a dependence vector is always positive, the graph is acyclic. As a consequence, the maximum value of the integer programming problem is always bounded. Second, efficiency can be improved by using linear programming methods such as simplex method to find the value of \( l \). For large \( U_j \)’s and small \( d_{ij} \)’s, which should be the common cases, the error is expected to be small. Finally, since most nested DO loops have depth less than or equal to 3 [10], it is usually that no more than \( 4 \) source iterations need to be considered 3.

5 Conclusion

In an asynchronous multiprocessor, executing a nested DO loop involves scheduling iterations to be executed as soon as possible. This kind scheduling is necessary even if grouping is applied. Although the concept is easily understood, the implementation of such execution scheme involves several issues. This paper first sketches the execution scheme in section 2, and then discusses the implementation issues in section 3, where the space bound for the implementation is found, as well as an addressing scheme is proposed. Because of the synchronization costs, parallel execution of a nested DO loop is not necessarily beneficial. The choice of whether or not executing a DO loop in parallel is studied in section 4, where finding the length of the longest path in a dependence graph is transformed to a couple of integer programming problems.

Some issues remain to be studied. First, it is preferred to find a lower space bound for the data structure PENDING, as well as its associated addressing scheme. Second, a more efficient and accurate algorithm to find the length of the longest path for a dependence graph is wanted. Thirdly, we would like to extend our execution scheme to more generic DO loops where an outer loop can contain several single statements and other DO loops inside. Finally, to improve the efficiency of the execution, it is also important to consider hardware supports for execution schemes.

References


\[ \text{Note that for all source iterations, the value of the first dimension is always 1.} \]
sis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1984.


Appendix

Proof of Lemma 4.

Define $A_k = \{j | 1 \leq j \leq k, d_{ij}d_{kj} < 0, \text{ and } |d_{1j}| \geq |d_{kj}|\}$, $B_k = \{j | 1 \leq j \leq k, d_{ij}d_{kj} < 0, \text{ and } |d_{kj}| > |d_{1j}|\}$, and $C_k = \{j | 1 \leq j \leq k, d_{ij}d_{kj} \geq 0\}$. We prove by induction on $k$ that

$$
\prod_{j=1}^{k} (U_j - |d_{1j}|) + \prod_{j=1}^{k} (U_j - |d_{kj}|) - \prod_{j=1}^{k} (U_j - |d_{kj} + d_{1j}|) \\
\leq \prod_{j \in A_k} U_j - \prod_{j \in A_k} (U_j - |d_{1j}|) + \prod_{j \in B_k} (U_j - |d_{kj}|) + \prod_{j \in C_k} (U_j - |d_{1j}| - |d_{kj}|) \\
+ \prod_{j \in A_k} (U_j - |d_{1j}|) \prod_{j \in B_k} (U_j - |d_{kj}|) \prod_{j \in C_k} (U_j - |d_{1j}| - |d_{kj}|) \\
(1)
$$

When $k = 1$, (1) is true since, by the definition of dependence vectors, $d_{11} \geq 0$ and $d_{21} \geq 0$.

Suppose (1) is true when $k = p - 1$, we prove that (1) is still true when $k = p$.

Case 1: $d_{1p}d_{2p} \geq 0$:

$$
(U_p - |d_{1p}|) \prod_{j=1}^{p-1} (U_j - |d_{1j}|) + (U_p - |d_{2p}|) \prod_{j=1}^{p-1} (U_j - |d_{2j}|) - (U_p - |d_{1p} + d_{2p}|) \prod_{j=1}^{p-1} (U_j - |d_{1j} + d_{2j}|) \\
= U_p \prod_{j=1}^{p-1} (U_j - |d_{1j}|) + \prod_{j=1}^{p-1} (U_j - |d_{2j}|) - \prod_{j=1}^{p-1} (U_j - |d_{1j} + d_{2j}|) \\
- d_{1p} \prod_{j=1}^{p-1} (U_j - |d_{1j}| - |d_{2j}|) + d_{2p} \prod_{j=1}^{p-1} (U_j - |d_{1j} + d_{2j}|) \\
\leq U_p \prod_{j=1}^{p-1} U_j - \prod_{j \in A_{p-1}} (U_j - |d_{1j}|) + \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) + \prod_{j \in C_{p-1}} (U_j - |d_{1j}| - |d_{2j}|) \\
+ \prod_{j \in A_{p-1}} (U_j - |d_{1j}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j \in C_{p-1}} (U_j - |d_{1j}| - |d_{2j}|) \\
[ \text{ by induction hypothesis } ]
$$

Case 2: $d_{1p}d_{2p} < 0$ and $|d_{1p}| \geq |d_{2p}|$:

$$
(U_p - |d_{1p}|) \prod_{j=1}^{p-1} (U_j - |d_{1j}|) + (U_p - |d_{2p}|) \prod_{j=1}^{p-1} (U_j - |d_{2j}|) - (U_p - |d_{1p} + d_{2p}|) \prod_{j=1}^{p-1} (U_j - |d_{1j} + d_{2j}|) \\
= U_p \prod_{j=1}^{p-1} (U_j - |d_{1j}|) + \prod_{j=1}^{p-1} (U_j - |d_{2j}|) - \prod_{j=1}^{p-1} (U_j - |d_{1j} + d_{2j}|) \\
- |d_{1p}| \prod_{j=1}^{p-1} (U_j - |d_{1j}| - |d_{2j}|) + |d_{2p}| \prod_{j=1}^{p-1} (U_j - |d_{1j} + d_{2j}|) \\
\leq U_p \prod_{j=1}^{p-1} U_j - \prod_{j \in A_{p-1}} (U_j - |d_{1j}|) + \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) + \prod_{j \in C_{p-1}} (U_j - |d_{1j}| - |d_{2j}|) \\
+ \prod_{j \in A_{p-1}} (U_j - |d_{1j}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j \in C_{p-1}} (U_j - |d_{1j}| - |d_{2j}|) \\
[ \text{ by induction hypothesis } ]
$$

9
\[+ U_p \prod_{i \in A_{p-1}} (U_j - |d_{ij}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j, j \in C_{p-1}} (U_j - |d_{ij}| - |d_{2j}|) - |d_{1p}| \prod_{i \in A_{p-1}} (U_j - |d_{ij}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j, j \in C_{p-1}} (U_j - |d_{ij}| - |d_{2j}|)\]

because

\[\prod_{j \in A_{p-1}} (U_j - |d_{ij}|) + |d_{2j}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j, j \in C_{p-1}} (U_j - |d_{ij}| - |d_{2j}|)\]

\[\prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j, j \in C_{p-1}} (U_j - |d_{ij}| - |d_{2j}|) \leq -\prod_{j \in C_{p-1}} (U_j - |d_{1j}| - |d_{2j}|),\]

\[|d_{1p}| \prod_{j \in A_{p-1}} (U_j - |d_{ij}|) \leq 0\]

\[\prod_{j \in A_{p-1}} (U_j - |d_{ij}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j, j \in C_{p-1}} (U_j - |d_{ij}| - |d_{2j}|)\]

\[\prod_{j \in A_{p-1}} (U_j - |d_{ij}|) \prod_{j \in B_{p-1}} (U_j - |d_{2j}|) \prod_{j, j \in C_{p-1}} (U_j - |d_{ij}| - |d_{2j}|)\]

Case 3: \(d_{1p}d_{2p} < 0\) and \(|d_{2p}| > |d_{1p}|\); This case is similar to case 2 and hence is omitted.

This completes our proof of inequality (1). Since

\[\prod_{j \in A_k} (U_j - |d_{ij}|) \prod_{j \in B_k} (U_j - |d_{2j}|) \prod_{j \in C_k} (U_j - |d_{ij}| - |d_{2j}|) \geq \prod_{j \in A_k} (U_j - |d_{ij}|) \prod_{j \in B_k} (U_j - |d_{2j}|) \prod_{j \in C_k} (U_j - |d_{ij}| - |d_{2j}|)\]

for any \(k\), the lemma is proved by setting \(k\) to \(n\). \(\square\)