Impact of Self-Scheduling Order on Performance of Multiprocessor Systems

Pei-Yi Tang  Pen-Chung Yew  Chuan-Qi Zhu

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

Processor self-scheduling is an efficient dynamic scheduling for multiprocessors. This paper discusses the impact of the self-scheduling order on the performance of multiply-nested parallel loops.

It is shown that, due to data synchronization for cross-iteration data dependences, the completion time of a multiply-nested loop is reduced when the nesting parallel loops with smaller delays are moved to the inside. The best performance is achieved when a shortest-delay scheduling order is used. The performance of the shortest-delay self-scheduling is compared to other self-scheduling orders and to compile-time static scheduling order proposed elsewhere. Program transformation needed to implement shortest-delay self-scheduling is also included.

1. Introduction

Since Lusk et al. [1] proposed self-scheduling DO-loops for HEP [2], there has been increasing interest in processor self-scheduling for multiprocessor systems. A processor self-scheduling scheme for multiply-nested loops was proposed in [3]. A self-scheduling scheme for even more general parallel loop structures was proposed in [4]. [5] proposed a compiler-guided self-scheduling scheme to improve the performance of singly-nested loops. Processor self-scheduling for singly-nested parallel loops has been implemented in several experimental multiprocessor optimizing compilers [6] [7]. Alliant FX/8 [8] and IBM 3081 [9] also use self-scheduling for processor allocation.

Processor self-scheduling on multiprocessor systems is a dynamic list scheduling [10] [11]. Whenever a processor becomes free, it scans a scheduling list to find the first task in the list whose predecessors are all completed, and starts to execute the task. Scheduling list represents the order in which the tasks are scheduled. In self-scheduling, the precedence relation among tasks can be enforced either by barrier synchronizations [3] or task queues [4]. The data dependences between concurrent tasks can be enforced by explicit low-level synchronizations [12] [13]. Tang et al. showed in [14] that an appropriate scheduling order is needed to prevent deadlocks if data dependences are explicitly synchronized between concurrent tasks.

Scheduling order can affect the performance of a task system. For a parallel nested loop, there can be many scheduling orders that allow deadlock-free execution. Due to the delay caused by explicit synchronizations between concurrent tasks, one scheduling order may result in better performance than the others. The purpose of this paper is to study the impact of different self-scheduling orders on the performance of parallel nested loops. We propose a scheduling order called the shortest-delay order, and show that the performance of using the shortest-delay order is always better than using other scheduling orders. We also present an algorithm to transform a nested loop into a form which allows shortest-delay self-scheduling.

2. Issues of Processor Self-Scheduling

In this section, a model for processor self-scheduling is described and the model will be used throughout the paper. We also address some issues regarding processor self-scheduling.

To execute a program on a multiprocessor system, the program needs to be decomposed into tasks with a precedence relation among these tasks. Tasks are usually iterations of parallel loops or a block of code that can be executed on a processor. If there are data dependences among tasks, they must be satisfied in order to preserve the semantics of the original program [15].

Tasks can be scheduled either statically at compile time or dynamically at run time. In static scheduling, each processor is pre-assigned a number of tasks at com-
3. Performance of Different Self-Scheduling Orders

Given a multiply-nested parallel loop and a limited number of processors, processor self-scheduling order can have a significant impact on the completion time of the loop.

Let us consider a \( n \)-level perfectly-nested loop as shown in Fig. 3.1. The index variables are \( i_1, i_2, \ldots, i_n \), and the corresponding loop upper bounds are \( N_1, N_2, \ldots, N_n \). The loop lower bound and the loop step of each loop are assumed to be 1. The loop body contains \( k \) statements: \( S_1, S_2, \ldots, S_k \). There are \( N_1 \times \cdots \times N_n \) iterations, each of which has an iteration vector \( \vec{i} = (i_1, i_2, \ldots, i_n) \).

\[
\begin{align*}
&\text{do } i_1 = 1, N_1 \\
&\quad \text{do } i_n = 1, N_n \\
&\quad \quad S_1 \\
&\quad \quad \quad \vdots \\
&\quad \quad S_k \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]

Fig. 3.1 A perfectly-nested loop

Consider two statements in the loop body: \( S_p \) and \( S_q \) (\( 1 \leq p, q \leq k \)). If the output value of \( S_p \) is used in \( S_q \), there is a flow dependence from \( S_p \) to \( S_q \). If the value of a variable used in \( S_p \) is updated in \( S_q \), there is an anti-dependence from \( S_p \) to \( S_q \). If the output value of \( S_p \) is updated by \( S_q \), there is an output dependence from \( S_p \) to \( S_q \). \( S_p \) is data dependent on \( S_q \), denoted by \( S_p \xrightarrow{\delta} S_q \), if there is a flow dependence, an anti-dependence or an output dependence from \( S_p \) to \( S_q \). \( S_p \) and \( S_q \) are called source and sink of that dependence, respectively. If \( p < q \), \( S_p \xrightarrow{\delta} S_q \) is a lexically forward dependence (LFD); if \( p \geq q \), \( S_p \xrightarrow{\delta} S_q \) is a lexically backward dependence (LBD).

Each data dependence has a distance vector \( \Delta \) defined as follows. If \( S_p \xrightarrow{\delta} S_q \) and \( S_p \) is in iteration \( \vec{i} = (i_1, i_2, \ldots, i_n) \) and \( S_q \) is in iteration \( \vec{i}' = (i_1', i_2', \ldots, i_n') \), the distance vector of the dependence is \( \Delta = \vec{i} - \vec{i}' - (i_1 - i_1', \ldots, i_n - i_n') \).

Since each iteration is a task and is executed by a single processor, the dependences with zero distance vectors are satisfied by the sequential execution of the task within the processor. The dependences with non-zero distance vectors are called cross-iteration dependences. There are two ways to enforce cross-iteration data dependences: implicit data synchronization and explicit data synchronization. If data dependences are enforced by the precedence relation among tasks, it is called implicit data synchronization. For example, cross-iteration data dependences in a loop can be enforced by forcing a par-

---

**Fig. 2.1** A model of self-scheduling
particular loop to be executed in serial. If data dependences are between two successive loops, we can use a barrier between them to enforce the data dependences. Using precedence relations (i.e. by serializing loops or using barriers) to enforce cross-iteration data dependences will lose parallelism in a program. Alternatively, we can use low-level synchronisation instructions to enforce cross-iteration data dependences directly. The executions of these concurrent tasks can be fully or partially overlapped, depending on the characteristics of the data dependences, and more parallelism can be exploited. Algorithms for generating data-level synchronization instructions to explicitly enforce these data dependences can be found in [16] [12] [13] [17]. In this paper, we assume that there are enough processors in the system and all nesting loops in Fig. 3.1 are parallel loops with explicit data synchronizations for cross-iteration dependences.

Fig. 3.2(a) shows an example of a perfectly-nested loop. The data dependences among the four statements are illustrated by the data dependence graph in Fig. 3.2(b). In Fig. 3.2(b), each node represents a statement and each directed edge represents a data dependence. The distance vector of each cross-iteration dependence is shown as the label of the corresponding edge. There are 16 iterations for the loop, and we assume that both nesting loops are executed as parallel loops, i.e. there is no precedence relation among the tasks.

```
  do i = 1, 4
    do j = 1, 4
      S1: A(i,j) = B(i,j-1) + C(i-1,j)
      S2: C(i,j) = A(i,j) + D(i,j-1)
      S3: B(i,j) = C(i,j) + E(i,j)
      S4: D(i,j) = B(i,j) + F(i,j)
      end do
    end do
```

(a)  (b)

Fig. 3.2 An example of a perfectly-nested loop

3.1. Performance with unlimited number of processors

First, assume that there are unlimited number of processors in the system. For the loop in Fig. 3.2(a), \( P=16 \) processors are enough to achieve the minimum completion time. According to the self-scheduling model described in section 2, 16 processors are self-scheduled to all 16 iterations instantly. If we use the natural lexicographic order of index vectors in self-scheduling, processor 1 will get iteration \((1,1)\), processor 2 will get iteration \((1,2)\) and so on. The execution profile of the loop is shown in Fig. 3.3. We assume that each statement takes one unit of time. Each vertical bold line in Fig. 3.3 represents the execution of an iteration. The iterations are illustrated in their natural lexicographic order, i.e. the first vertical line from the left corresponds to iteration \((1,2)\) and the second line corresponds to iteration \((1,2)\) and so on. Arrows in Fig. 3.3 denote the cross-iteration dependences enforced by explicit data synchronization. In Fig. 3.3, the number near the beginning of each iteration indicates the number of the processor assigned to that iteration and \(s(i,j)\) is the starting time of iteration \((i,j)\). Notice that, due to the dependence \(S_3 \delta S_1\) (see Fig. 3.2(b)), the starting time of an iteration of loop \(j\) is delayed by \(d_j=3\) units of time from that of the previous iteration. Similarly, due to the dependence \(S_2 \delta S_1\), the starting time of an iteration of loop \(i\) is delayed by \(d_i=2\) units of time from the previous iteration. Thus, the completion time of the loop is

\[
T_{\infty} = D_{\min} + B = d_i(N_i-1) + d_j(N_j-1) + B
\]

\[
= 2(4-1) + 3(4-1) + 4 = 19
\]

where \(B\) is the execution time of each iteration (i.e. \(B=4\)) and \(D_{\min}\) is the delay for the starting time of the last iteration (i.e. iteration \((4,4)\)) from that of the first iteration (i.e. iteration \((1,1)\)) of the loop.

Cytroon [18] proposed a Dofacross model for parallel execution of loops. A Dofacross loop is a Do loop with a delay between the starting time of its successive iterations. The motivation of introducing these delays is to eliminate busy-waiting caused by data dependence enforcement. The delays of the nesting loops are calculated in such a way that the source statement of each data dependence is always executed before its sink statement. An algorithm to calculate these Dofacross delays can be found in [18]. Although Dofacross delays will not
be the same as the actual delays observed when the loop is self-scheduled with explicit data synchronization, they characterize the effect of data dependencies on the amount of overlapping between loop iterations. To this end, we can use Doacross delays as a guideline in considering the performance of different self-scheduling orders.

For the loop in Fig. 3.2(a), the Doacross delays calculated by using the algorithm in [18] happen to be the same as the actual delays for processor self-scheduling in Fig. 3.3 with unlimited number of processors (i.e. \( P \geq 16 \)). The starting time \( s(i,j) \) of each iteration is also shown in Fig. 3.3. From Fig. 3.3, we have

\[
s(i,j) = d_i(i-1) + d_j(j-1) \quad (3.1)
\]

\( s(i,j) \) represents the earliest time when iteration \((i,j)\) can start. When there are limited number of processors in the system, iteration \((i,j)\) may start later than \( s(i,j) \) due to the lack of available processors.

3.2. Performance with limited number of processors

In this section, we show that when there are limited number of processors, self-scheduling order has a significant impact on the completion time of a loop. For the loop in Fig. 3.2(a), we examine three scheduling orders: (1) the natural lexicographic order of the original loop, (2) the natural lexicographic order of the loop after loop \( i \) and loop \( j \) are interchanged and (3) the shortest-delay scheduling order based on \( s(i,j) \).

Assume that the self-scheduling order is the natural lexicographic order of the index vector \((i,j)\), i.e. \((1,1), (1,2), ..., (2,1), ..., (4,4)\). Fig. 3.4 shows the execution profile of the loop using \( P = 6 \) processors. The vertical bold lines in Fig. 3.4 represent the executions of 16 iterations in their natural lexicographic order. According to the self-scheduling model in section 2, 6 processors get the first 6 iterations \((1,1), (1,2), ..., (2,2)\) instantly at time \( 0 \). As before, the number of the processor is labeled at the beginning of each iteration. Iteration \((1,1)\) completes at time 4 and processor 1 returns to the processor pool and will get iteration \((2,3)\) immediately. Notice that the next completed iteration is iteration \((2,1)\) which completes at time 6, instead of iteration \((1,2)\) which completes at time 7. After Processor 5 completes iteration \((2,1)\), it is self-scheduled to iteration \((2,4)\). Iteration \((3,1)\) can start at time 4, but there is no processor available until processor 2 finishes iteration \((1,2)\) at time 7. The starting time of iteration \((3,1)\) is thus delayed by 3 units of time due to the unavailability of processors. Due to the delay caused by the dependencies, the starting times for the remaining iterations are also delayed by 3 units of time. The new starting time is represented by a square \( \Box \). Iteration \((4,1)\) is supposed to start at time 9; however, for lack of available processors, it is again delayed by another 3 units of time to time 12. This causes the starting times of iterations \((4,2), (4,3)\) and \((4,4)\) to be also delayed for another 3 units of time (see triangulars in Fig. 3.4). Compared to the completion time of the loop with unlimited number processors (see \((3.0)\)), there is an extra delay \( D' = 3 + 3 = 6 \) for the case of \( 6 \) processors:

\[
T_n = D_{\text{min}} + D' + B = d_i(N_i-1) + d_j(N_j-1) + D' + B \quad (3.2)
\]

\[
= 2(4-1) + 3(4-1) + 6 + 4 = 25
\]

The second self-scheduling order that we want to examine is the natural lexicographic order of the loop after interchanging the loop with smaller delay (i.e. loop \( i \), because \( d_i = 2 \) and \( d_j = 3 \)) into the inside. According to the rule of loop interchanging [19] [20] [21], the semantics of the loop will be preserved after interchanging loop \( i \) and loop \( j \) in Fig. 3.2(a). Loop \( i \) becomes inner loop and the natural lexicographic order of the iteration \((i,j)\) becomes \((1,1), (2,1), ..., (1,2), ..., (1,4)\). Notice that loop interchanging is allowed only when it does not change data dependencies. The delays between successive iterations of loop \( i \) and loop \( j \) are still \( d_i = 2 \) and \( d_j = 3 \), respectively. Fig. 3.5 shows the execution profile using this new self-scheduling order. We can see that the extra delay due to the unavailability of processors, \( D' = 1 + 1 = 2 \), is shorter than before loop interchanging. The reason is that the processors are first assigned to a loop with smaller data dependence delay between successive iterations. Processors start to execute iteration earlier and are released earlier. The released processors can be self-scheduled to the remaining iterations earlier. As a result, the total completion time for the nested loop is reduced.

Fig. 3.4 The execution profile with limited processors
We can push this rule even further by directly considering $s(i,j)$, i.e. the delay of starting time for iteration $(i,j)$ due to data dependences. We can sort $s(i,j)$ and use the order of the sorted $s(i,j)$ as the self-scheduling order. That is, we define the self-scheduling order as follows: iteration $(i_1,j_1)$ is scheduled before iteration $(i_2,j_2)$ if and only if (1) $s(i_1,j_1) < s(i_2,j_2)$ or (2) $s(i_1,j_1) = s(i_2,j_2)$ and $(i_1,j_1) < (i_2,j_2)$. Here, $(i_1,j_1) < (i_2,j_2)$ if and only if (1) $i_1 < i_2$ or (2) $i_1 = i_2$ and $j_1 < j_2$. Using this order in self scheduling, a processor is always scheduled to the iteration with the shortest delay among the tasks to be scheduled. We call this order shortest-delay self-scheduling order. According to the starting times of iterations illustrated in Fig. 3.3, the shortest-delay self-scheduling order is: $(1,1), (2,1), (1,2), (3,1), (2,2), (1,3), (4,1)$, and so on. The execution profile using the shortest-delay scheduling order is shown in Fig. 3.6. There is no extra delay caused by the lack of available processors, i.e. $D' = 0$, and

$$T_b = D_{\text{min}} + D = d_i(N_i - 1) + d_j(N_j - 1) + B$$

$$= 2(4-1)+3(4-1)+4 = 10$$

This is the best completion time we can get because it equals to the completion time with unlimited number of processors (see (3.0)).

### 3.3. Comparison of performance

In this section, we compare the performance of three self-scheduling orders for limited number of processors.

We also compare their performance with that of the static scheduling scheme proposed in [22].

To facilitate our discussion, we use the following formula to describe the completion time of a parallel loop using $P$ processors:

$$T_p = T_{\infty} + D'(P)$$

where $T_{\infty}$ is the optimal completion time with unlimited number of processors and $D'(P) \geq 0$ represents the extra delay due to the limited number of processors. For the parallel loop in Fig. 3.2(a), $T_{\infty} = 10$. Let us call the natural lexicographic order of the loop in Fig. 3.2(a) $j$-inner order. The natural lexicographic order after loop $i$ and loop $j$ are interchanged is called $i$-inner order. For the loop in Fig. 3.2(a), $D'(0)$ for the $j$-inner, $i$-inner and shortest-delay self-scheduling orders are 6, 2 and 0, respectively.

Let us consider the minimum number of processors needed for the optimal completion time for a loop. Let it be denoted by $P$, i.e. $P = \min\{P \mid D'(P) = 0\}$. Following the same process as in section 3.2, we found that, for the loop in Fig. 3.2(a), $P$ for the $j$-inner, $i$-inner and shortest-delay self-scheduling order are 11, 8 and 8, respectively. That is,

$$P_{\text{shortest}} \leq P_{\text{i-outer}} \leq P_{\text{j-outer}}$$

For a doubly-nested loop with both loop bounds $N_i$, we only need to consider the case with $P$ processors, where $N<P \leq P$. Because if $P \leq N$, only the outer loop needs to be parallelized (because we don't have enough processors to make the inner loop run in parallel anyway).
and the loop becomes a singly-nested parallel loop. For the loop in Fig. 3.2(a), we examine the performance of the three self-scheduling orders with \( P \) processors, where \( 5 \leq P \leq 11 \). The extra delay \( D'(P) \) for different number of processors is shown in Table 3.1. From Table 3.1, we have the following for the loop in Fig. 3.2(a)

\[
D'_{\text{shortest}}(P) \leq D'_{j-\text{inner}}(P) \leq D'_{j-\text{inner}}(P) \quad (3.6)
\]

<table>
<thead>
<tr>
<th>( P )</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D'_{j-\text{inner}}(P) )</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( D'_{i-\text{inner}}(P) )</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( D'_{\text{shortest}}(P) )</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1 Additional delay, \( D'(P) \), for self-scheduling with limited processors

For a general doubly-nested loop with both loop bounds equal to \( N \) and the delay of loop \( i \) is less than or equal to that of loop \( j \), we conjecture that (3.5) and (3.6) still hold when \( P > N \). However, it is still an open problem to prove this conjecture.

For general multiply-nested loops, the actual delay in unlimited processor self-scheduling may not be the same as the delay calculated by the Doacsross delay model. However, we can still use the Doacsross delay model to approximate the actual delay for each iteration in unlimited processor self-scheduling. We can then use these estimations to form the shortest-delay order for limited processor self-scheduling. The performance of the shortest-delay self-scheduling is still the best among other scheduling orders because it minimizes the delays for all loop levels. However, a formal proof of this claim is quite difficult because of the dynamic nature of the self-scheduling.

We can also compare the performance of self-scheduling with that of static scheduling. Cytrot [22] proposed an predictor-corrector static scheduling scheme for parallel nested loops using Doacsross delay for each loop. For the loop in Fig. 3.2(a), the minimum number of processors needed for the optimal completion time, \( \hat{P} \), is 8. Table 3.2 shows the performance for the limited processor cases. We can see that \( j\text{-inner} \) static scheduling has a better performance than that of \( i\text{-inner} \) static scheduling. This is because the static scheduling scheme assigns processors starting from outer loops and, thus, gives the outer loops a higher priority in processor allocation. We also notice that the performance for \( P = 5, 6, 7 \) are the same for static scheduling. This is because processors are formed into groups in static scheduling and only 4 processors are actually used. (See [22] for more details of the static scheduling scheme.) Comparing Table 3.1 with Table 3.2, we can see that the shortest-delay self-scheduling is still better than the static scheduling for both \( \hat{P} \) and \( D'(P) \). The \( j\text{-inner} \) self-scheduling should be compared to the \( i\text{-inner} \) static scheduling; the \( i\text{-inner} \) self-scheduling should be compared with the \( j\text{-inner} \) static scheduling because of their different scheduling priorities. In these comparisons, both static scheduling and self-scheduling win in some cases and lose in others.

3.4. Deadlock-free shortest-delay self-scheduling

We have seen that for the loop in Fig. 3.2(a), the performance of the shortest-delay self-scheduling is the best among three scheduling orders considered. In the natural lexicographic orders, the performance is improved after the loop \( i \) with smaller delay is interchanged into the innermost loop. The shortest-delay self-scheduling order is not a natural order. Hence, we have to consider the problem of deadlocks for this scheduling order.

In our self-scheduling model, once a processor gets an iteration, the processor will not be preempted until the completion of the iteration. If a data dependence is not satisfied, the processor will be busy-waiting at the sink of the dependence until the source of the dependence becomes available. In such systems, when the number of processors is limited, deadlocks can occur if a "circular-wait" condition arises [23]. Tang et al. [14] showed that deadlocks can be prevented if an appropriate self-scheduling order is chosen. For parallel loops that do not have a precedence relation among their iterations, we only have to consider the data dependences among iterations. According to [14], deadlocks can be prevented if the data dependences are consistent with the self-scheduling order, i.e. if there is a data dependence from iteration \( \hat{I}_1 \) to \( \hat{I}_2 \), then iteration \( \hat{I}_1 \) must be scheduled before iteration \( \hat{I}_2 \).

For lexically backward dependences (LBD), we have the following theorem:

**Theorem 3.1:**

If there is a lexically backward dependence from iteration \( \hat{I}_1 \) to iteration \( \hat{I}_2 \), then \( s(\hat{I}_1) < s(\hat{I}_2) \), where \( s(\hat{I}_1) \) and \( s(\hat{I}_2) \) are starting times in the Doacsross model for iteration \( \hat{I}_1 \) and \( \hat{I}_2 \), respectively.

Using integer programming to obtain Doacsross delay \( d_i \), \( 1 \leq i \leq n \) (see [18]), Theorem 3.1 can be proved quite easily. Hence, its proof is omitted here. According to the definition of the shortest-delay scheduling order, we conclude that the LBDs are consistent with the shortest delay scheduling order.

Lexically forward dependences (LFD) need special attentions in deadlock prevention. In general, a LFD is
not necessarily consistent with the shortest-delay scheduling order. In this case, we can have three choices:

1. Some LFDs may become redundant after LBDs are enforced by explicit synchronizations [24] [12]. These dependences can be eliminated and, hence, are ignored in deadlock prevention.

2. If a LFD can not be eliminated, we can treat them as a LBD in calculating Doacross delays using the algorithm in [18]. The shortest-delay order based on the newly-calculated loop delays will take care of the LFD.

3. We can use more processors to prevent deadlocks. For example, for the program in Fig. 3.7(a), the Doacross delay calculated by using the data dependence graph in Fig. 3.7(b) will be \( d_i = 1 \) and \( d_j = 3 \). The Doacross execution profile is shown in Fig. 3.7(c). The shortest-delay scheduling order is \( (1,1), (2,1), (3,1), (1,2), (2,2), (3,2), (1,3), (2,3) \) and \( (3,3) \). If there were only two processors, a deadlock will occur: processor 1 will get iteration \( (1,1) \) after it completes iteration \( (1,1) \) because iteration \( (2,1) \) is already assigned to processor 2. Both processor 2 and processor 1 will wait for the data of array \( D \) from iteration \( (1,2) \) and iteration \( (2,2) \), respectively (see dashed arrows in Fig. 3.7(c)), and neither of iterations \( (1,2) \) and \( (2,2) \) was assigned a processor yet. The number of processors should be at least four; otherwise only one loop can be parallel (since both loops have bounds of 3). With at least four processors, deadlocks cannot occur. The problem of determining the number of processors needed to prevent deadlocks caused by LFDs in the shortest-delay self-scheduling remains open. Nevertheless, the deadlocks can be prevented by forcing some loops to be serial when there are not enough processors.

4. Program transformation for shortest-delay self-scheduling

In self-scheduling, processors schedule themselves by fetch-and-adding the loop index variables to obtain an iteration. Fetch-and-add can be implemented as a hardware-supported synchronization instruction. It is available in some multiprocessor systems [25] [26]. For the loop shown in Fig. 3.1, processors can fetch-and-add a single shared variable to get the sequence numbers of the iterations in their lexicographic order. The sequence number is then converted to the corresponding index vector \( \hat{I} \) (see [3]). The shortest-delay self-scheduling order does not conform to any lexicographic order. In this section, we describe a program transformation needed to support the shortest-delay self-scheduling.

Consider the loop in Fig. 3.1. First, we need to perform a data dependence test to build the data dependence graph like the one in Fig. 3.2(b). Then, calculate the Doacross delay \( d_i \geq 0 \) \( (i=1,2,\ldots,n) \) for each nesting loop, using the algorithm proposed in [18]. Notice that \( d_i \) may not be an integer. Since we are only interested in the relative values of the delays for the nesting loops, we can always multiply them with the least common multipliers of their denominators. To simplify our discussion, we assume that \( d_i \)'s are integers. The starting time of iteration \( \hat{I} = (i_1,i_2,\ldots,i_n) \) can be calculated as follows:

\[
    s(\hat{I}) = \sum_{i=1}^{n} d_i i_i - \sum_{i=1}^{n} d_i
\]

(4.0)

Since \( d_i \geq 0 \) and \( 1 \leq i_i \leq N_i \), we have

\[
    0 \leq s(\hat{I}) \leq \sum_{i=1}^{n} d_i N_i - \sum_{i=1}^{n} d_i
\]

(4.1)

The strategy is to use \( t = s(\hat{I}) \) as the index variable of the outmost loop, and use \( i_1, i_2, \ldots, i_{n-1} \) as the index variables for the inner loops. \( i_n \) is determined by \( t, i_1, \ldots, i_{n-1} \) as follows:

\[
    i_n = \left( t - \sum_{i=1}^{n-1} d_i i_i \right) / d_n
\]

(4.2)

The transformed loop has the following form:

---

Fig. 3.7 An example of a nested loop
\[ \text{do } t = 0, 15 \text{. do } i = \max(1, \left\lfloor \frac{(t-7)}{2} \right\rfloor, \min(4, \left\lfloor \frac{(t+2)}{2} \right\rfloor) \text{ if } ((t-2i+5)/3 \text{ is an integer}) \text{ then} \]

\[ j = (t-2i+5)/3 \]

\[ \text{S1. } A(i,j) = B(i-1,j) + C(i-1,j) \]
\[ \text{S2. } C(i,j) = A(i,j) + D(i-1,j) \]
\[ \text{S3. } R(i,j) = C(i,j) + F(i,j) \]
\[ \text{S4. } D(i,j) = B(i,j) + F(i,j) \]

end if

end do

Fig. 4.1 Transformed loop for the shortest–delay self-scheduling

\[ \text{do } t = 0, 15 \text{. do } i = \max(1, \left\lfloor \frac{(t-7)}{2} \right\rfloor, \min(4, \left\lfloor \frac{(t+2)}{2} \right\rfloor) \text{ if } ((t-2i+5)/3 \text{ is an integer}) \text{ then} \]

\[ j = (t-2i+5)/3 \]

\[ \text{S1. } A(i,j) = B(i-1,j) + C(i-1,j) \]
\[ \text{S2. } C(i,j) = A(i,j) + D(i-1,j) \]
\[ \text{S3. } R(i,j) = C(i,j) + F(i,j) \]
\[ \text{S4. } D(i,j) = B(i,j) + F(i,j) \]

end if

end do

where \( LB_k \) and \( UB_k \) are the new lower bound and upper bound of \( i_k \), respectively. \( LB_k \) and \( UB_k \) can be calculated from the index variables of the outer loops \( t, i_1, \ldots, i_{k-1} \) as follows: from (4.0), we have

\[ d_k = t - \frac{1}{k-1} \sum_{i=1}^{k-1} d_i i_k - \frac{1}{k} \sum_{i=1}^{k} d_i i_k + \frac{1}{k} \sum_{i=1}^{k} d_i \]

where \( i_{k+1}, \ldots, i_n \) are not determined yet. Since \( 1 \leq i_k \leq N_k \) and \( d_k \geq 0 \) \((i=k+1, \ldots, n)\), we have

\[ t - \sum_{i=1}^{k-1} d_i i_k - \sum_{i=1}^{k} d_i N_k + \sum_{i=1}^{k} d_i \leq d_k i_k \leq t - \sum_{i=1}^{k-1} d_i i_k + \sum_{i=1}^{k} d_i \]

Remember that \( 1 \leq i_k \leq N_k \). From the above, we have the following formulas for \( LB_k \) and \( UB_k \):

\[ LB_k = \max(1, \left\lfloor \frac{t - \sum_{i=1}^{k-1} d_i i_k - \sum_{i=1}^{k} d_i N_k + \sum_{i=1}^{k} d_i}{d_k} \right\rfloor) \] (4.3)

and

\[ UB_k = \min(N_k, \left\lfloor \frac{t - \sum_{i=1}^{k-1} d_i i_k + \sum_{i=1}^{k} d_i}{d_k} \right\rfloor) \] (4.4)

Let us go back to the loop in Fig. 3.2(a) and use the above formulas to transform it for shortest–delay self-scheduling. We already know that \( d_1 = 2 \), \( d_2 = 3 \) and

\[ N_1 = N_2 = 4 \]. Form (4.1), we have \( 0 \leq t \leq 15 \). From (4.3) and (4.4), we have

\[ \max(1, \left\lfloor \frac{(t-7)/2}{d_k} \right\rfloor) \leq i \leq \min(4, \left\lfloor \frac{(t+2)/2}{d_k} \right\rfloor) \]

The transformed loop is shown in Fig. 4.1.

Let us track the lexicographic order of the transformed loop to see how it actually follows the shortest–delay order of the original loop. First, set \( t = 0 \). Both lower and upper bound of \( i \) are 1. Using \( t = 0 \) and \( i = 1 \), we find that \( j = 1 \). Thus, iteration (1,1) is the first one to be scheduled (see Fig. 3.3 for starting times \( s(i,j) \) of iterations \( (i,j) \)). Next, let \( t = 1 \). Both lower and upper bound of \( i \) are 1 again. But, this time, \( j = 5/3 \) is not an integer. Thus, there is no corresponding iteration. It can be seen from Fig. 3.3 that there is no iteration starts in the horizontal line of \( t = 1 \). When \( t = 2 \), we have \( i = 1, 2 \). If \( i = 1 \), \( j \) is not an integer. If \( i = 2 \) we have \( j = 1 \), and it corresponds to iteration (2,1). Using this algorithm, we can go through all of the iterations of the original loop in their shortest–delay scheduling order as shown in Table 4.1. In Table 4.1, "skip" means that there is no corresponding iteration for that particular \( t \) and \( i \). Only the entries within the trapezoidal iteration space spanned by \( t \) and \( i \) are filled.

### 5. Discussions

### (1) Loop interchange for self-scheduled parallel loops

Table 3.1 shows that the performance is better when a loop with smaller delay is an inner loop. This observation suggests that, if loop nestings are interchangeable,
we should interchange the loops with smaller delay to the inside. This observation is exactly the opposite of the rule used in static scheduling schemes [22], where the loops with smaller delay should be moved to the outside. This is because, in static scheduling, processors are allocated to outer loops first. In other words, static scheduling gives outer loops higher priority in processor allocation. On the contrary, self-scheduling schemes use a lexicographic order which gives a higher priority to the inner loops.

It is very important to note that, using explicit synchronization instructions to enforce data dependences, all nesting loops of a multiply-nested loop can potentially be executed as parallel loops, i.e., all of the iterations can be started at the same time, even though some iterations may wait for a long time for data dependences to be satisfied. With unlimited number of processors, we can afford to assign one processor to each iteration and allow the processors to wait for as long as needed to satisfy the data dependences. However, if we only have limited number of processors, it is very important to determine which nesting loops should be executed as parallel loops in order to optimize processor utilization and minimize busy-wait time. To simplify our argument, let us assume all of the nesting loops of a multiply-nested loop are interchangeable. It is obvious that we should move all of the nesting loops that will be executed as parallel loops to the outside, so after processors are assigned to those loops, the inner loops are essentially a part of the loop body which will be executed on one processor (i.e., executed as serial loops).

The procedure of loop interchange for self-scheduling can thus be described as follows:

1. Determine the number of loops, m, to be executed as parallel loops.
2. Interchange m loops with smallest delay to the outside, if they are interchangeable.
3. Interchange among the m outer parallel loops (if they are interchangeable) so that the delays of the loops are decreasing from outside to the inside, i.e., the loop with the smallest delay is the innermost among the m outer loops.

2 Shortest-delay transformation and loop skewing

The program transformation described in section 4 is used to transform a loop so that the shortest-delay order can be followed when the loop is self-scheduled. We found that when that transformation is applied to a loop with only one recurrent statement (such as a loop for four-point relaxation), we can get the same result as loop skewing proposed in [19] [27] with loop interchanged. In other words, loop skewing combined with loop interchange can be regarded as a degenerated case of the shortest-delay transformation, even though loop skewing is primarily for vector machines.

\begin{verbatim}
d0 i = 1, N
d0 j = 1, N
A(i,j) = \{A(i-1,j) + A(i,j-1)\}/2
end do
end do
\end{verbatim}

(a)

\begin{align*}
&(-1,0) \\
&S & (0,1) \\
&j=1 \\
&i=1 & 2 & 3 & 4 \\
&2 & & & \\
&3 & & & \\
&4 & & &
\end{align*}

(b)

(c)

Fig. 5.1 A single statement recurrence

Consider a single-statement doubly-nested loops as shown in Fig. 5.1(a). Its data dependence graph is shown in Fig. 5.1(b), and its data dependences in the iteration space are shown in Fig. 5.1(c). For vector processors or parallel processors without mechanism to explicitly enforce cross-iteration data dependences, neither loop i nor loop j can be executed in parallel. However, using loop skewing techniques in [19] [27], we can add i-1 to both the lower bound and upper bound of loop j and replace j with j-(i-1) in the loop body as shown in Fig. 5.2(a). Loop j is skewed with respect to loop i. The data dependences of the new loop is shown in Fig. 5.2(c) and the new data dependence graph is shown in Fig. 5.2(b). Now, data dependences on loop i are removed. Using the technique of trapezoidal loop interchange in [21], loop i can be interchanged into the inner loop and then vectorized (see Fig. 5.3).

We can easily arrive at the same result by using our shortest-delay transformation. From Fig. 5.1, we calculate that \(d_j=d_j-1\). Thus, we have \(s(i,j)-i+j-2\), and \(j=s(i,j)-i+2=t-i+2\). Since \(j\) is always an integer, the IF statement in the loop body (see Fig. 4.1) is not needed in this case. The new program is shown in Fig. 5.4, where \(j\) is substituted by \(t-i+2\) in the subscripts. The program in Fig. 5.3 is the same as in Fig. 5.4 if \(j\) is replaced by \(t+1\) in the subscripts.

Since there is only a single statement in the loop body in Fig. 5.1(a), all data dependences are lexically backward dependences. From Theorem 3.1, there cannot be dependences between the instances of S with the same \(t=s(i,j)\). It follows that all statements with the same
s(i,j) can be executed in parallel without synchronization, i.e. the inner loop i can be vectorized. Hence, loop skewing combined with loop interchange is actually a degenerated case of the shortest-delay transformation.

\[
\begin{align*}
do i & = 1, N \hspace{1cm} do i = 1, N+i-1 \\
A(i,j-1+i) &= \frac{A(i-1,j-i+1) + A(i,j-i)}{2} \\
end do & \hspace{1cm} end do 
\end{align*}
\]

Fig. 5.2 The single statement recurrence after loop j is skewed

6. Conclusion

Processor self-scheduling is an efficient dynamic scheduling for multiprocessors. In this paper we discuss the impact of the self-scheduling order on the performance of multiply-nested parallel loops. We show that, due to data synchronization for cross-iteration data dependences, the completion time of a nested loop can be reduced when the nesting loops with smaller delays are interchanged to the inside. The best performance is achieved when a shortest-delay scheduling order is used. The performance of the shortest-delay self-scheduling is also better than that of some compile-time static scheduling schemes. We also include a program transformation needed to implement shortest-delay self-scheduling and found that loop skewing [19] [27] combined with loop interchange is a degenerated case of that transformation.

\[
\begin{align*}
do i & = 0, 2N-2 \\
do j & = max(1, j-N+1), min(N,j) \\
A(i,j-1+i) &= \frac{A(i-1,j-i+1) + A(i,j-i)}{2} \\
end do & \hspace{1cm} end do 
\end{align*}
\]

Fig. 5.4 The single statement recurrence after shortest-delay transformation

REFERENCES


