PROCESSOR SELF-SCHEDULING FOR MULTIPLE-NESTED PARALLEL LOOPS

Peiyi Tang and Pen-Chung Yew

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

Abstract

Processor self-scheduling is a useful scheme in a multiprocessor system if the execution time of each iteration in a parallel loop is not known in advance and varies substantially, or if there are multiple nestings in parallel loops which makes static scheduling difficult and inefficient. By using efficient synchronization primitives, the operating system is not needed for loop scheduling. The overhead for the processor self-scheduling is small.

We presented a processor self-scheduling scheme for a single-nested parallel loop, and extend the scheme to multiple-nested parallel loops. Barrier synchronization mechanisms in the processors self-scheduling schemes are also discussed.

1. Introduction

Processor scheduling is a problem that must be solved in order to run a program on a multiprocessor system efficiently. The multiprocessor system considered here is a collection of identical processors with a global shared memory. Cray X-MP [1], Cedar [2], Ultracomputer [3] and RP3 [4] are some examples of such systems.

Parallel loops in a program, whose iterations can be executed concurrently on different processors, provide the greatest potential of parallelism to be exploited by multiprocessor systems. It is called a DOALL-loop, if the iterations of a parallel loop are independent. If there are data dependences across iterations of a DO-loop, its iterations can still be executed concurrently on different processors provided that the data dependences are enforced by synchronization across the processors during the execution [8]. This kind of parallel loop is called a DOACROSS-loop [11]. Both DOALL-loops and DOACROSS-loops can be nested in many levels. In this paper, we only consider DOALL-loops.

DOALL-loops can either be recognized by an optimizing compiler like Paraphrase [10] or explicitly specified by a programmer. Processors need to be scheduled properly so that execution time of a DOALL-loop is minimized. A task of scheduling here is one or several iterations of a DOALL-loop. We will only consider non-preemptive scheduling schemes, i.e., once a processor is assigned an iteration, it will continue to execute the iteration until its completion.

If execution time of each iteration of a DOALL-loop is the same, the optimal scheduling is to assign iterations evenly among processors. This scheduling can be done before program execution and is called static scheduling.

If execution time of each iteration is different and is not known until program execution, the scheduling of iterations to processors is more efficient if it can be done dynamically during the program execution, i.e., assign a new iteration to a processor whenever it becomes available. The total number of iterations assigned to each processor may not be equal, but the workload of each processor tends to be balanced. This scheduling is called dynamic scheduling. In this paper, we only consider dynamic scheduling.

Dynamic scheduling will incur scheduling overhead at run time. One technique to reduce overhead is to schedule several iterations (called a chunk of iterations) to a processor at a time. As long as the number of chunks is large enough, workload among processors can still be balanced.

Scheduling overhead can be very large if dynamic scheduling is done by system calls to the operating system. One way to reduce scheduling overhead is to use processor self-scheduling [5] [6] [7]. Rather than issuing a system call to the operating system for scheduling, processors can schedule themselves by fetch-and-adding a shared variable to get loop indices of a chunk of iterations. If a multiprocessor system has efficient hardware-implemented synchronization primitives like those in Cedar [8], Ultracomputer [9] and RP3 [4], scheduling overhead for a chunk of iterations can be reduced quite significantly.

However, so far, all processor self-scheduling schemes only deal with the outermost parallel DO-loop [5] [6] [7]. The rest of the parallel loops nested inside are treated as serial DO-loops. Hence, parallelism of the nested DOALL-loops is not fully exploited.

In this paper, we present a self-scheduling scheme for nested DOALL-loops using Cedar synchronization instructions. Cedar synchronization instructions [8] are briefly introduced in section 2. In section 3, we describe barrier synchronization mechanism needed in processor self-scheduling and processor self-scheduling schemes for single-nested DOALL-loops. In section 4, we present a processor self-scheduling scheme for multiple-nested DOALL-loops. An estimation of overhead for scheduling an iteration and the performance of the proposed processor self-scheduling schemes are given in section 5. In section 6, we have some concluding remarks.
2. Cedar Synchronization Primitives

In a Cedar Multiprocessor System, a variable can be declared as a synchronization variable. A synchronization variable x has two fields: KEY and DATA. The KEY field is for storing synchronization information (which is an integer) and the DATA field is for storing the value of the variable (which can be a floating point number). The format of a Cedar synchronization instruction is as follows:

\{x; test on KEY;
  operation on KEY; operation on DATA\}

Here x is the name (or the address) of the synchronization variable. The "test on x.KEY" specifies the condition to be tested between the KEY field of x (denoted by x.KEY) and a key provided by the instruction (denoted by i.KEY). The test includes \( >, \geq, <, \leq, =, \neq \) and NULL. The NULL test means that no test is needed and therefore the result of the test is always true. The operation on x.KEY can be Increment, Decrement, Add, Fetch, Fetch & Add, Store, Fetch & Increment, Fetch & Decrement and No Action. The operation on DATA can be Fetch, Store and No Action. The execution of the whole synchronization instruction, i.e. test on x.KEY, operations on x.KEY and x.DATA, is done in globally shared memory modules and is indivisible [8]. The operation on x.KEY and the operation on x.DATA are executed only when the result of the test is true. The memory module will inform the processor of a "failure" if the condition of the test is not satisfied, or a "success" if the condition is satisfied and execution of the instruction is completed. For some application, test on x.KEY has to be done repeatedly until the test condition becomes true. A star on the test condition (as shown below) is used to indicate this situation. In other words,

\{x; (test on KEY)*; 
  operation on KEY; operation on DATA \}

is equivalent to

1: \{x; test on KEY; 
  operation on KEY; operation on DATA \}

if failure then goto 1

Cedar synchronization primitives are very effective in handling low level synchronizations required in numerical computations like enforcing data dependences across loop iterations [8]. In those applications, KEY field stores loop iteration numbers and DATA field usually stores the value of the data (which is usually a floating point number). In this paper, we use this synchronization primitives mostly for non-numerical scheduling problems. Hence, only the KEY field is used.

3. Barrier Synchronization

Following are several assumptions used in our processor self-scheduling schemes:

1. We assume that the operating system assigns certain number of processors, say P processors, to a program before its execution. After that, the operating system will not be involved in scheduling.
2. Code for processor self-scheduling is embedded in a user's program, and each processor will execute the same program.
3. For the simplicity of the discussion, we assume that a task in our scheduling is only one iteration. A task which has several iterations can be similarly implemented.

The iterations of a DOALL-loop are scheduled through a shared variable, and after all of the iterations are scheduled, a barrier synchronization is needed for the completion of the DOALL-loop. In this section, we present two barrier synchronization mechanisms and processor self-scheduling schemes for a single-nested DOALL-loop.

In the first barrier synchronization mechanism, processors will be blocked at the barrier until all of the processors complete their tasks and arrive at the barrier. After that, the barrier will open and all of the processors can pass through. There is a counter which counts the number of arriving processors, and it must be reinitialized before the barrier can be reused for the second time. Algorithm 3.1 is a processor self-scheduling scheme for a DOALL-loop using this barrier synchronization mechanism.

Algorithm 3.1

\/* J: loop index for the DOALL-loop with M iterations
   A: a counter to count the number of processors
      that have arrived at the barrier
   B: a variable acting as a barrier */

\/* Initially J=1, A=P and B=0, where P is the total number of the processors assigned to execute the program. */

L1: \{J; \leq M; Fetch(LOCJ)&Increment\}
  if failure then goto \( L_2 \)

  \/* Original Do loop-body with index LOCJ */

  goto \( L_1 \)

  \/* Processors go back to \( L_1 \)
   to get another iteration. */

L2: \{A; \geq 1; Decrement\}
  if failure then
  begin
    J:=1; A:=P; B:=P
    end
  \{B; \geq 0\}; Decrement
  \/* B acts as a barrier and is reinitialized after P processors pass through. */

If a DOALL-loop is enclosed in an outer serial loop as shown in Figure 3.1, the implementation of the outer serial loop is quite simple. Since all of the processors are synchronized at the barrier of the second DOALL-loop \( J_2 \), each processor can have a local copy of loop index variable LOCJ and update it after the execution of the DOALL-loop \( J_2 \). The processor self-scheduling code for the whole program is illustrated in Figure 3.2.

However, a barrier synchronization based on the number of arriving processors has a problem. If the actual number of processors assigned to the program is less than that specified in the program (i.e. P processors), the barrier of \( J_1 \) will never open and all of the processors will be stuck there. In other words, the operating system must assign exactly the same number of processors as specified in the program; otherwise, we will have a deadlock.
However, the precedence relation only requires that all of the iterations of the first DOALL-loop be completed before the second DOALL-loop can be started. In the following, a barrier is controlled by the number of completed iterations of the DOALL-loop instead of by the number of arriving processes. Counter A is initialized to M, the total number of iterations, instead of P. When all of the iterations are completed, the barrier will be opened. Thus, this barrier synchronization mechanism will work even if the operating system assigns fewer than P processors to the program. The processor self-scheduling scheme using this barrier synchronization mechanism is shown in Algorithm 3.2.

**Algorithm 3.2**

/* C: a lock to block processors after P processors have entered the DOALL-loop */
J: loop index for the DOALL-loop with M iterations
A: a counter to count the number of completed iterations
B: a variable acting as a barrier
T: a counter to detect the last processor, which is responsible for reinitializing all of the synchronization variables */

/* Initially C=P, J=1, A=M, B=0 and T=P. */
P is the number of processors assigned to the program. */

{C; (>0)*; Decrement}
/* This is to block processors after P processors have entered the DOALL-loop. */
L₁: {J; ≤M; Fetch(LOCJ)&Increment}
if failure then goto L₂

/* Original Do loop-body with index LOCJ */

{A; >1; Decrement}
if failure then B:=1
/* Counter A counts the number of completed iterations and controls the opening of the barrier. */
goto L₁
/* Processors go back to L₁ to get another iteration. */
L₂: {B; (>0)*; No Action}
/* B acts as a barrier. */
{T; >1; Decrement}
if failure then
/* This is to reinitialize all of the synchronization variables. */
begin
    T:=P; B:=0; A:=M
    J:=1; C:=P
end

Note that all of the synchronization variables in the algorithm will be reinitialized after P processors have passed the variable T. If there are fewer than P processors assigned to the program, some processors must come back and make up the discrepancy, and the synchronization variables will be reinitialized eventually.

Lock C in the beginning of the code is used to prevent the race problem caused by the delay in reinitializing the barrier. Note that there is a delay between the time when all of the P processors pass the barrier and the time when reinitialized. During this period of time, it is possible that some processors may revisit this DOALL-loop again for the next iteration of the outer serial loop. If there is no such a lock to block those fast processors, they will pass the barrier for the second time and start executing the following DOALL-loop before this DOALL-loop is restarted.

The disadvantage of the barrier synchronization mechanism based on the number of completed iterations is that the implementation of the outer serial DO-loop is more complicated. Because processors are no longer synchronized after each iteration of the outer serial DO-loop, the number of the times each processor will traverse the outer serial DO-loop may be different. A shared index variable for the outer serial DO-loop is thus needed. The branch node needed to implement the outer serial DO-loop is given in Algorithm 3.3. Figure 3.3 illustrates the processor self-scheduling code for the entire program in Figure 3.1 using this barrier synchronization mechanism.

**Algorithm 3.3**

/* C: a lock to block processors after P processors have entered the branch node. */
S: a semaphore to guarantee that only one processor can control the outer serial loop
B: a variable acting as a barrier
T: a counter to detect the last processor, which is responsible for reinitializing all of the synchronization variables */

/* Initially C=P, S=1,B=0 and T=P, where P is the number of processors assigned to the program. */

{C; (>0)*; Decrement}
/* This is to block processors after P processors have entered the branch node. */
L₁: {S; >0; Decrement}
if failure then goto L₂
/* This is to allow only one processor to execute the following code. */
if I<N then
begin
    COND:= true; I:=I+1
end
else
begin
    COND:= false; I:= 1
end
/* Update the shared index variable I and resolve the branch condition, which is stored in COND */
B:=1
/* Open the barrier B */
L₂: {B; (>0)*; No Action}
MYCOND:=COND
/* Save the branch condition */
{T; >1; Decrement}
if failure then
begin
4. Processor Self-Scheduling for Nested DOALL-loops

In this section, we extend the processor self-scheduling schemes to nested DOALL-loops. Before going further, we have to define a few terms which will be used later.

If each inner DOALL loop is surrounded immediately by an outer DOALL loop with no scalar code between them, this multiple DOALL loop nestings is a perfectly-nested DOALL structure.

Otherwise, a nested DOALL structure is called a non-perfectly-nested DOALL structure. Figure 4.1 shows an example of non-perfectly-nested DOALL structure. There, we use a left bracket "[" to denote each DOALL-loop nesting. In a non-perfectly-nested DOALL structure, each innermost DOALL loop nesting will contain a loop body. The loop-body with all of its surrounding DOALL-loop nestings form a nested DOALL component. For example, the non-perfectly-nested DOALL structure in Figure 4.1 has three nested DOALL components. A perfectly-nested DOALL structure contains only one nested DOALL component.

We only discuss self-scheduling scheme of a non-perfectly-nested structure. A perfectly-nested DOALL structure is a special case of a non-perfectly-nested DOALL structure.

Assume that a non-perfectly-nested DOALL structure like the one in Figure 4.1 consists of m nested DOALL components. Each nested DOALL component can be characterized by two vectors: an index variable vector and a loop-bound vector. The index variable vector of a nested DOALL component \((I_1, I_2, \ldots, I_d)\) is a vector which consists of the index variables of the loop nestings. Here, \(I_j\) is the index variable of the outermost nesting and \(d\) is the depth of the loop nestings. The loop-bound vector \((N_1, N_2, \ldots, N_d)\) is a vector of the corresponding loop-bounds. For example, the index variable vectors for the three nested DOALL components in Figure 4.1 are \((1, I_2, I_3)\), \((1, I_2, I_3)\), and \((1, I_2, I_3)\), respectively, and their corresponding loop-bound vectors are \((3, 7, 3)\), \((3, 2, 3)\), and \((3, 2, 4)\).

For a nested DOALL component with index variable vector \((I_1, I_2, \ldots, I_d)\) and loop-bound vector \((N_1, N_2, \ldots, N_d)\), there are total of \(N_1 \cdot \cdots \cdot N_d\) iterations for the component. Each of them can be identified by a sequence number from 1 to \(N_1 \cdot \cdots \cdot N_d\). If we split the \(d\) loop nestings at level \(k\) \((1 \leq k \leq d)\) into \((I_1, \ldots, I_k)\) and \((k+1, \ldots, I_d)\), we can notice that there are \(N_{k+1} \cdot \cdots \cdot N_d\) iterations of the component that share common loop indices of the \(k\) outermost nestings \((I_1, \ldots, I_k)\). An iteration with \(I_1=i_{11}, I_2=i_{21}, \ldots, I_k=i_{k1}\) is called an iteration at level \(k\). The sequence number for the iteration is defined to be

\[ i = (i_1 - 1)N_2 \cdot \cdots \cdot N_k + \cdots + (i_{k-1} - 1)N_k + i_k \]

There are \(N_1 \cdot \cdots \cdot N_k\) iterations at level \(k\), each of which can be identified by a sequence number from 1 to \(N_1 \cdot \cdots \cdot N_k\).

Let \((I_1, I_2, \ldots, I_d)\) and \((N_1, N_2, \ldots, N_d)\) be the index variable vector and the loop-bound vector of the \(J\)-th component. It has two parameters \(k_j\) and \(l_j\) \((1 \leq j \leq m, 1 \leq k_j \leq d_j, 1 \leq l_j \leq d_j)\), defined as follows: if its deepest common loop nesting with the \((j-1)\)-th component is at \((p-1)\) level, we will have \(k_j = p\). Similarly, if the deepest common loop nesting it shares with the \((j+1)\)-th component is at \((q-1)\) level, we have \(l_j = q\). From the definition, we have \(l_j = k_{j+l} - 1, 1 \leq j \leq m\). We also define that \(k_j = 1\) and \(l_j = m\). For the non-perfectly-nested DOALL structure in Figure 4.1, we have \(k_1 = 1, k_1 = 2, k_2 = 2, k_3 = 3, k_3 = 3, l_3 = 1\).

Assume that the deepest common loop nesting for the \((j-1)\)-th and the \(j\)-th components is at level \(p-1\), i.e. \(l_j = k_j = p\) (See Figure 4.2). According to the semantics of nested DOALL-loops, an iteration at level \(p-1\) for the \(j\)-th component can not be started until the corresponding iteration for the \((j-1)\)-th component is completed. We use a variable \(B^{j-1}\) to enforce this precedence constraint. When \(B^{j-1} = 1\), the first \(t\) iterations at level \(p-1\) for the \((j-1)\)-th component are completed, and, hence, we can start to schedule the \(j\)-th component for those iterations.

The components of a DOALL loop structure will be scheduled in sequence, and the iterations of each component will be scheduled in the order of their sequence numbers. Thus, the processor self-scheduling schemes proposed here for nested DOALL-loops can also be applied to nested DOACROSS-loops or mixture of nested DOALL-loops and DOACROSS-loops without causing deadlocks.

Processor self-scheduling for the \(j\)-th nested DOALL component is realized by fetch-and-incrementing an index control variable \(I^f\). The initial value of \(I^f\) is 1, which corresponds to the first index vector \((1,1,\ldots,1)\). The maximum value of \(I^f\) is \(M^j = N_1 \cdot \cdots \cdot N_j\), which corresponds to the last index vector \((N_1, N_2, \ldots, N_j)\). Let \(F_{N_1, \ldots, N_j}\) be a function that maps the sequence number of an iteration into its index vector. Table 4.1 shows an example of function \(F_{2,3,2}\): Function \(F_{N_1, \ldots, N_j}\) can be formulated as follows:

\[ F_{N_1, \ldots, N_j}(x) \rightarrow (i_1, i_2, \ldots, i_j) \]

where for \(1 \leq k \leq d_j\), we have

\[ i_k = \left[ \frac{x - 1}{N_{k+1} \cdots N_j} \right] \mod N_k + 1. \]

\([x]\) is the largest integer smaller than \(x\). This computation can be done locally in a processor after it fetches a sequence number from \(I^f\). However, as mentioned before, if the sequence number of an iteration at level \(p-1\) for the \(j\)-th component is larger than \(B^{j-1}\), it can not be started. Given a sequence number \(x\) of an iteration, the sequence number of the iteration at level \(p-1\) is

\[ y = \left[ \frac{x - 1}{N_p \cdots N_j} \right] + 1. \]

So, before starting to execute an iteration of the component, a processor needs to check if \(B^{j-1} \geq y\). If \(B^{j-1} < y\), a processor has to wait until \(B^{j-1}\) becomes larger.

The processor self-scheduling code for the non-perfectly-nested DOALL structure is given in Algorithm 4.1.
Algorithm 4.1

\{ C; ( > 0 ) \}; Decrement

. . .

/* The code for the j-th component starts here. Initially \( i^j = 1 \), \( B^j = 0 \), 
\[ A^j (i_1, i_2, \ldots, i_{j-1}) = N_{i\rightarrow i} \cdots N_{i_{j-1}} \] 
\[ \{ 1 \leq i_1 \leq N_{i}, 1 \leq i_2 \leq N_{i_2}, \ldots, 1 \leq i_{j-1} \leq N_{i_{j-1}} \} \] */

\( L^j : \{ i^j, \leq M_{i^j} ; \text{Fetch}(x) \& \text{Increment} \} \)

/* \( M_{i^j} = N_{i^j} \cdots N_{i} \) */

if failure then goto \( L^j +2 \)

/* \( L^j +2 \) is the beginning of the next component */

\[ y := \left\lfloor (x-1)/(N_{i} \cdots N_{i^j}) \right\rfloor + 1 \]

\( B^j +1 : ( \geq 2y ) ; \) No action

for \( k = 1 \) to \( d_{i^j} \) do

\[ \text{LOCI}_k := \left\lfloor (x-1)/(N_{i} \cdots N_{i^j}) \right\rfloor \mod N_k + 1 \]

/* Compute the index vector from the sequence number \( x \) using function \( \text{F}_{N_{i}, \ldots, N_{i^j}} \) */

. . . original Loop-body with index vector 

\( \{ \text{LOCI}_1, \text{LOCI}_2, \ldots, \text{LOCI}_{i^j} \} \) */

\( A^j (\text{LOCI}_1, \text{LOCI}_2, \ldots, \text{LOCI}_{i^j-1}) ; \geq 1 ; \text{Decrement} \)

if failure then begin

\[ A^j (\text{LOCI}_1, \ldots, \text{LOCI}_{i^j-1}) = N_{i^j} \cdots N_{i} \]

\[ z := \left\lfloor (x-1)/(N_{i} \cdots N_{i^j}) \right\rfloor + 1 \]

\( B^j ; ( \geq z +1 ) ; \text{Increment} \)

end

/* The j-th component ends here. */

\( \}

. . .

\( \}

/* \( B^m ; ( = 1 ) \); No action */

/* \( B^m \) acts as a barrier. */

\( \{ T^j ; > 1 ; \text{Decrement} \} \)

if failure then begin

\( T^j := P \)

\( B^j := 0 ; i^j := 1 \)

\( B^{m} := 0 ; i^{m} := 1 \)

. . .

\( B^{m} := 0 ; i^{m} := 1 \)

\( C := P \)

end

Algorithm 4.1 consists of three portions. First portion is just a single statement

\( \{ C; ( > 0 ) \}; \text{Decrement} \)

which functions as a lock in the beginning of the code to prevent race problem. Lock C is initialized to P, the number of processors assigned to the program. The second portion is the main self-scheduling code for each nested DOALL component. They are the same except for the different parameters such as loop-bound vector, \( k_j \) and \( l_j \). We only present the code for the j-th component.

The main self-scheduling code for each component consists of three parts: loop self-scheduling, the original loop-body and bookkeeping of the completed iterations. We already discussed the "self-scheduling" part. In bookkeeping of the completed iterations, recall that for each iteration at level \( q-1 \), there are total of \( N_q \cdots N_{i} \) iterations for the \((d_j-q)\) innermost loop nestings of the component. We use an element of array \( A^j \) to record the total remaining iterations. It will be decremented by 1 after each iterations of the innermost \((d_j-q)\) loops has been completed.

When \( N_q \cdots N_{i} \) iterations of the innermost \((d_j-q)\) nestings are completed, the element of \( A^j \) is reinitialized. Given the sequence number \( x \) of the iteration fetched from \( i^j \), the sequence number \( z \) of the corresponding iteration at level \( q-1 \) is

\[ z = \left\lfloor \frac{x-1}{N_q \cdots N_{i^j}} \right\rfloor + 1. \]

\( B^j \), which is used to control the execution of the \((j+1)\)-th component, is then updated. Note that \( B^j \) is incremented only when its value is one smaller than the calculated sequence number. Thus, when \( B^j = k \), it is guaranteed that the iterations at level \( q-1 \) with the sequence number from 1 to \( k \) have been completed.

The last portion of Algorithm 4.1 is for barrier synchronization and reinitialization. \( B^m \) acts as a barrier for the entire non-perfectly-nested DOALL structure. In fact, since \( k^m = 1 \), \( B^m \) has only two values: 0 and 1. As in Algorithm 3.2, \( T^j \) is used to detect the last processor leaving the entire DOALL structure. That processor is responsible for reinitializing variables \( B^j, i^j (1 \leq j \leq m) \), \( T \) and \( C \). Array \( A^j (1 \leq j \leq m) \) is reinitialized in the code for each component in the second portion. The barrier synchronization used here is based on the number of completed iterations. If the nested DOALL structure is within an outer serial loop, one has to use algorithm 3.3 to implement branch node.

Notice that iterations for the first nested DOALL component in a non-perfectly-nested DOALL structure can be scheduled without restriction. It is not necessary to check the value of \( B^0 \). This is why we set \( k^m = 1 \) and \( B^0 \) is not used.

The entire processor self-scheduling code for the whole nested DOALL structure in Figure 4.1 is shown in Figure 4.3.

5. Overhead and Performance

Processor self-scheduling allows us to have better utilization of processors. During the program execution time, whenever a processor becomes available, it will grab another iteration and start executing the new iteration as long as the precedence relation in the program is not violated. Hence, the program execution time can actually be improved.

In a large multiprocessor system, global memory accesses will take very significant amount of time, hence, we will only consider global memory accesses (remember that Cedar synchronization primitives are global memory accesses) when we estimate the scheduling overhead. In Algorithm 3.1 and Algo-
Algorithm 3.2, the overhead of scheduling an iteration of a single-nested parallel loop is only one Cedar synchronization instruction, which is one global memory access to the index control variable J. The overhead of scheduling an iteration of multiple-nested parallel loops is two global memory accesses as shown in Algorithm 4.1: one for testing and fetch-and-adding index control variable J' and one for checking variable B^j1-2. The later is not needed in the case of perfectly nested parallel loops, i.e. the scheduling overhead for an iteration of a perfectly nested parallel loop is only one global memory access. This overhead is very small compared to the overhead that would incur if it is done by the operating system.

It is very important to note that the barrier synchronization in the end of parallel loops are also needed in the static scheduling schemes. Hence, they are not extra overhead for the self-scheduling schemes.

Let us compare the self-scheduling scheme with the static scheduling scheme proposed in [12]. Assume that the execution times for each iteration of the nested DOALL structure is the same.

For a single-nested DOALL loop with N iterations, both schemes will schedule N iterations over P processors evenly. The program execution time will be the same for both schemes, i.e. it will be \(\left\lceil \frac{N}{P} \right\rceil\) times the execution time of an iteration. However, processor self-scheduling will require one extra global memory access and, hence, it will have slightly more overhead than static scheduling scheme.

However, for multiple-nested DOALL loops, things can be quite different. Assume that we have a perfectly-nested DOALL-loop structure with loop-bound vector \((N_1, N_2, \ldots, N_d)\). Using static scheduling scheme, we have to find the optimal decomposition of \(P=P_1 \times \cdots \times P_d\), where \(P_i\) is the number of processors assigned to the i-th loop nesting, such that \(\left\lceil \frac{N_1}{P_1} \right\rceil \cdots \left\lceil \frac{N_d}{P_d} \right\rceil\) is minimized [12]. The completion time for this static scheduling is

\[
T_{st} = \tau \left\lceil \frac{N_1}{P_1} \right\rceil \cdots \left\lceil \frac{N_d}{P_d} \right\rceil
\]

(5.1)

where \(\tau\) is the execution time of an iteration. Using processor self-scheduling in Algorithm 4.1, we have

\[
T_{sf} = (\tau + \sigma) \left\lceil \frac{N}{P} \right\rceil,
\]

(5.2)

where \(\sigma\) is the extra time needed for 1 extra Cedar synchronization instruction, and \(N = N_1N_2 \cdots N_d\). Note that

\[
\left\lceil \frac{N}{P} \right\rceil \leq \left\lceil \frac{N_1}{P_1} \right\rceil \cdots \left\lceil \frac{N_d}{P_d} \right\rceil
\]

(5.3)

The equality in (5.3) holds only for some special cases. Let us ignore those special cases and assume that \(\left\lceil \frac{N_1}{P_1} \right\rceil \cdots \left\lceil \frac{N_d}{P_d} \right\rceil = \left\lceil \frac{N}{P} \right\rceil = k\), \(k \neq 0\). \(T_{sf}\) is less than \(T_{st}\) when

\[
\tau > \frac{\sigma}{k} \left\lceil \frac{N}{P} \right\rceil
\]

For the non-perfectly-nested DOALL-loop structure in Figure 4.1, assume that there are \(P=8\) processors. Using the static scheduling scheme in [12], we get the optimal decomposition \(P=1 \times 2 \times 4\), i.e., using 1 processor for the outermost loop nesting, 2 processors for the second outermost loop nesting, and 4 processors for the innermost loop nesting. The completion time is \(9\tau\). Using self-scheduling, the completion time is \(8(\tau + 2\sigma)\). In self-scheduling scheme, a processor is not tied to any specific loop nesting, thus, processors can be better utilized.

6. Concluding Remarks

We present processor self-scheduling scheme for both single- and multiple-nested parallel loops. Two different barrier synchronization mechanism are discussed. The scheduling overheads for these schemes are quite small if synchronization primitives are supported in the system as in Cedar [2].

Processor utilization can be improved over static scheduling scheme, which can lead to better program execution time if the execution time of each loop iteration varies substantially, or if there are multiple loop nestings.

REFERENCES

(12) Constantine D. Polychronopoulos, David J. Kuck and David

DO SERIAL I = 1, N
DO ALL J_1 = 1, M_1
.
. ENDDOALL
DO ALL J_2 = 1, M_2
.
. ENDDOALL
ENDDO SERIAL

(a) (b)

Figure 3.1 An example of parallel program

LOCI := 1

START:

DO ALL J_1
(Algorithm 3.1)

DO ALL J_2
(Algorithm 3.1)

if LOCI < N then
begin
   LOCI := LOCI + 1
   goto START
end

Figure 3.2 Outer serial loop control (scheme 1)

START:

DO ALL J_1
(Algorithm 3.2)

DO ALL J_2
(Algorithm 3.2)

Branch Node
(Algorithm 3.3)

Figure 3.3 Outer serial loop control (scheme 2)

$\begin{array}{c|c|c|c}
 x & i_1 & i_2 & i_3 \\
\hline
1 & 1 & 1 & 1 \\
2 & 1 & 1 & 2 \\
3 & 1 & 2 & 1 \\
4 & 1 & 2 & 2 \\
5 & 1 & 3 & 1 \\
6 & 1 & 3 & 2 \\
7 & 2 & 1 & 1 \\
8 & 2 & 1 & 2 \\
9 & 2 & 2 & 1 \\
10 & 2 & 2 & 2 \\
11 & 2 & 3 & 1 \\
12 & 2 & 3 & 2 \\
\end{array}$

Table 4.1 Function $F_{2,3,2}$

$\begin{bmatrix}
 J_1 = 1, 3 \\
 J_2 = 1, 7 \\
 \ldots \\
 J_3 = 1, 3 \\
 \ldots \\
 K_3 = 1, 4 \\
 \ldots
\end{bmatrix}$

Figure 4.1 A non-perfectly-nested DO ALL structure

Figure 4.2 p-1 outermost common loop nestings
(C; >0*; Decrement)

$L^1$: (I; $\leq 2$; Fetch(x)&Increment)
if failure then goto $L^2$
$(LOCI_1, LOCI_2) \leftarrow F_{3,4}(x)$

/*loop body 1 with $(LOCI_1, LOCI_2)^*$*/

$(A^1(LOCI_1); >1; Decrement)$
if failure then begin
  $A^1(LOCI_1) := 7$
  $z := \lceil(x-1)/7\rceil + 1$
  $(B^1; (=z-1)^*; Increment)$
end
goto $L^1$

$L^2$: (I; $\leq 2$; Fetch(x)&Increment)
if failure then goto $L^1$
y := $\lfloor(x-1)/(2\times2)\rfloor + 1$
$(B^2; (>y)^*; No action)$
$(LOCI_1, LOCI_2, LOCI_3) \leftarrow F_{3,2,3}(x)$

/*loop body 2 with $(LOCI_1, LOCI_2, LOCI_3)^*$*/

$(A^2(LOCI_1, LOCI_2); >1; Decrement)$
if failure then begin
  $A^2(LOCI_1, LOCI_2) := 3$
  $z := \lfloor(x-1)/3\rfloor + 1$
  $(B^2; (=z-1)^*; Increment)$
end
goto $L^2$

$L^3$: (I; $\leq 2$; Fetch(x)&Increment)
if failure then goto $L^4$
y := $\lceil(x-1)/4\rceil + 1$
$(B^3; (>y)^*; No action)$
$(LOCI_1, LOCI_2, LOCK_3) \leftarrow F_{3,4,4}(x)$

/*loop body 3 with $(LOCI_1, LOCI_2, LOCK_3)^*$*/

$(A^3; >1; Decrement)$
if failure then begin
  $A^3 := 3\times2\times4$
  $z := 1$
  $(B^3; (=z-1)^*; Increment)$
end
goto $L^3$

$L^4$: (I; $\leq 1$; No action)
$(T; >1; Decrement)$
if failure then begin
  $T := P$
  $B^4 := 0$; $I^i := 1$
  $B^3 := 0$; $I^i := 1$
  $B^2 := 0$; $I^i := 1$
  $C := P$
end

Figure 4.3 Self-scheduling code for DOALL structure in Figure 4.1