Dynamic Processor Self-Scheduling for General Parallel Nested Loops

ZHIXI FANG, MEMBER, IEEE, PEIYI TANG, PEN-CHUNG YEW, SENIOR MEMBER, IEEE, AND CHUAN-QI ZHU

Abstract—This paper proposes a processor self-scheduling scheme for general parallel nested loops in multiprocessor systems. Parallel loops usually constitute most of the execution time in scientific application programs. In a general parallel loop structure, parallel loops, serial loops, and IF–THEN–ELSE constructs are nested in an arbitrary order, and the execution time of the loop body may vary substantially from iteration to iteration. In the proposed scheme, programs are instrumented to allow processors to schedule loop iterations among themselves dynamically at run time without the involvement of the operating system.

The proposed self-scheduling scheme has two levels. At the low level, it uses simple "fetch-and-add" operations to take advantage of the regular structure in the innermost parallel loop nests. At the high level, the irregular structure of the outer loops (parallel or serial) and the IF–THEN–ELSE constructs are handled by using dynamic parallel linked lists. The larger granularity of the processes at the high level can easily justify the added overhead incurred from maintaining such dynamic data structures.

Many self-scheduling techniques proposed so far, such as guided self-scheduling (GSS) [14] and shortest-delay self-scheduling (SDSS) [16] can be incorporated in this scheme. The performance and its possible overhead using this scheme are also analyzed in the paper.

Index Terms—Data synchronization, doacross loops, doall loops, self-scheduling, shared-memory multiprocessor.

I. INTRODUCTION

As device technology approaches its limits, it becomes harder to improve system performance through faster clock rate. Parallel processing using multiple processors becomes a necessity. This can be witnessed by the number of processors in Cray systems, which has increased from one processor in the Cray 1, four processors in the Cray 2, and eight processors in the Cray Y-MP to possibly 64 processors in the future Cray 4 [1]. Exploiting parallelism on multiprocessor systems is much harder than on vector machines. In addition to different parallel algorithms and compiler technology, determining task granularity and scheduling tasks on multiple processors are also vital to the performance of such systems.

Classical static scheduling schemes require the assumption of fixed execution time on tasks. Recent research results [23] show that low variance in the task execution time can also give the performance of the static scheduling schemes. However, they are not applicable here because it is difficult to estimate the execution time, or the distribution of the execution time of tasks on a multiprocessor system. The difficulty stems from the fact that 1) the time spent on synchronization among tasks when they access shared data differs significantly from one execution to another, and can only be determined at run time; 2) the location of data in a memory hierarchy, which is very common in many multiprocessor systems, can cause memory access time to vary widely for memory accesses, and 3) conditional statements with significantly different execution times in each branch can also contribute to the inaccuracy in its estimation.

Dynamic run time scheduling seems to be the only alternative in such an environment. However, it is usually costlier than static compile-time scheduling because it generally requires calling the operating system. This poses a serious problem in determining the granularity of tasks during program partitioning.

It has been shown that more parallelism can be exploited in most programs when the granularity of parallelism becomes smaller [2]. However, large scheduling overhead can easily nullify the performance gained from a large amount of small-grain parallelism. It may even worsen the overall performance if the increased cost of scheduling is excessive (i.e., the performance from a parallel execution can be worse than that from a serial execution). It is thus essential to balance the task granularity with the scheduling overhead.

In most scientific programs, loops are a major source of parallelism [3]. Many compiler techniques have been developed to detect and enhance parallelism in loops [4]–[6]. Because of the regular structure in loops, they are relatively easy to partition and to schedule dynamically. Processor self-scheduling on parallel loops, which allows a processor to "grab" one or several iterations of a loop (by "fetch-and-adding" the loop index variable), was first used on Denelcor HEP multiprocessors [7], and was later used in multiproces-
sor systems such as the Alliant FX/8 [8], the Cray X-MP [9], the Cedar system [12], and the IBM RP3 [10], etc. The idea is to avoid involving the operating system, and decrease the scheduling overhead, by instrumenting parallel loops with the code needed for processors to perform scheduling themselves at run time. The scheduling overhead can be even lower if special hardware is used to facilitate the "fetch-and-add" operations [11], [12], [8].

Few studies have been done on how to instrument multiply-
nested parallel loops for self-scheduling [13], [14]. In [14], a
guided self-scheduling (GSS) scheme is proposed to dynamically change the number of iterations assigned to each processor by dividing the number of remaining iterations with the number of processors, so the workload can be better balanced. However, the parallel loops assumed there have to be perfectly nested so that "implicit loop coalescing" can be used to allocate the iterations of nested parallel loops easily. The self-scheduling scheme in [13] can handle nonperfectly nested parallel loops using implicit loop coalescing, even though it does not allow serial loops to be nested between parallel loops. GSS also assumed parallel loops have no cross-iteration data dependences, so called Doall loops [5], because loops containing cross-iteration data dependences, so called Doacross loops [15], generally perform worse if more than one iteration is assigned to each processor at a time. For example, if there is a data dependence between adjacent iterations, scheduling one iteration on one processor allows processors to start about the same time with some degree of overlap between iterations as described in [15]. However, if we use chunk scheduling (such as GSS) and assign five iterations to each processor, for example, then the next processor cannot continue until the first processor executes the source statement of the data dependence in the fifth iteration. The amount of overlap between tasks is much worse (i.e., about four out of five iterations cannot be overlapped) due to cross-iteration data dependence. The synchronization time is a more dominant factor than the scheduling overhead in Doacross loops.

In this paper, we propose a general scheme to instrument nested parallel loops for processor self-scheduling. Parallel loops may include Doall loops and Doacross loops. The loops can also have parallel and serial loops nested arbitrarily in several levels and the execution time of iterations is unpredictable. Loop bounds in different levels can be functions of the indexes of outer loops. Furthermore, there may exist IF-THEN-ELSE constructs in the loops, and each of their branches may contain other loops (parallel and serial) and other IF-THEN-ELSE constructs. These specifications cover most loops found in parallel programs.

The proposed scheme has two levels. The lower level takes advantage of the regular structure of inner parallel loops by using simple self-scheduling schemes [7], [13]. In this level, GSS [14] for Doall loops and shortest-delay self-scheduling (SDSS) schemes [16], [21] for Doacross loops can be used to enhance performance. The high level uses a task pool to handle irregular outer loop structures such as nested parallel and serial loops and IF-THEN-ELSE constructs.

The rest of the paper is organized as follows. In Section II, we discuss the machine model and some basic concepts used in this paper. In Section III, the self-scheduling scheme is described in detail. In Section IV, the performance and the overhead of the scheme are analyzed. In Section V, we present our conclusions.

II. BACKGROUND

A. The Machine Model

A shared-memory multiprocessor system comprises a number of processors and memory modules connected by an interconnection network. Each processor may have local memory. Shared variables and common data are stored in the shared memory. To support concurrent execution of multiple processes, a set of synchronization primitives is provided. We use a set of synchronization primitives similar to those provided on the Cedar system [17].

Each synchronization variable is an integer variable in the shared memory. A synchronization instruction is an indivisible "test-and-op" operation on a synchronization variable. The format of a synchronization instruction is as follows:

\[
\text{test on } x; \text{ operation on } x
\]

Here \(x\) is the name of the synchronization variable. The test is to be made between the current value of the synchronization variable and an integer provided by the synchronization instruction. The operation is to be applied on the synchronization variable when the test is successful. If the test fails, the operation is not executed. A signal is sent back to the processor that issued the instruction indicating if the test failed or succeeded. The execution of the entire synchronization instruction (i.e., the test and the operation) is indivisible.

The test between the synchronization variable and the integer provided by the instruction can be \(>, \geq, <, \leq, =, \neq, \) and null. A null test means that no test is needed and the operation is always executed. The operation on the variable includes: Fetch, Store, Increment, Decrement and Fetch-and-add(\(k\)). Fetch-and-add(\(k\)) fetches the original value of the variable and then adds \(k\) to the variable. For example, the instruction

\[
\{A < 100; \text{ Fetch}(a) & \text{add}(3)\}
\]

means that if \(A < 100\), then fetch the original value of \(A\) to the local variable \(a\) and add 3 to \(a\). Fetch-and-increment and Fetch-and-Decrement are two special cases of Fetch-and-add(\(k\)) with \(k = 1\) and \(k = -1\), respectively.

A "failure" or "success" signal is sent back to the processor. The following code

\[
\text{again: } \{(S > 0); \text{ Decrement}\}; \\
\text{if (failure) goto again;}
\]

can be used to implement the \(P\) operation on a general semaphore \(S\) [18]. The corresponding \(V\) operation on the semaphore \(S\) is

\[
\{S; \text{ Increment}\};
\]

No test is needed on \(S\) (i.e., a null test).
These synchronization instructions are a subset of the Cedar synchronization instructions [17]. They can also be implemented by using other synchronization instructions. In fact, the self-scheduling scheme proposed in this paper has been implemented in a compiler using Fetch/Increment and Fetch/Decrement instructions [19].

R General Nested Parallel Loops

A general nested parallel loop has the following characteristics:

1. It can be a nonperfectly nested loop.
2. Parallel loops (including Doall loops and Doacross loops) and serial loops can be nested in any arbitrary order.
3. Loop bounds of the nested loops can be functions of the indexes of the outer loops.
4. There can be IF-THEN-ELSE constructs. Each branch of an IF-THEN-ELSE construct can contain other nested loops or other IF-THEN-ELSE constructs.

A typical example of such a loop is shown in Fig. 1. We use solid and dashed left brackets to denote parallel and serial loops, respectively. If there are no loop-independent data dependences [4] among the loops at the same nesting level, these loops can be executed in parallel. However, to simplify our presentation, we assume that loops (parallel or serial) at the same nesting level are executed in sequence. For example, in Fig. 1, parallel loop \( B \), serial loop \( K \) (with its enclosed parallel loops \( C \) and \( D \)) and parallel loop \( E \) are executed in sequence. In PCF Fortran [22], programmers can specify such nonlooping parallelism using parallel section constructs. Our self-scheduling scheme can be easily extended to accommodate such vertical parallelism.

A general nested parallel loop can be transformed into a form in which each innermost loop is a parallel loop, some of which may have only one iteration. For example, the program in Fig. 2(a) has an innermost serial loop \( J_4 \) nested in a parallel loop \( J \). Since processor scheduling is needed only on the parallel loop \( J \), serial loop \( J_4 \) can be viewed as the loop body of the parallel loop \( J \). Hence, from the scheduling point of view, parallel loop \( J \) is an innermost loop which requires scheduling. Nested serial loops \( J_2 \) and \( J_3 \) are enclosed in the outer serial loop \( J_1 \) at the same level as the parallel loop \( J \). They are scalar code which need only one processor to execute. Any scalar code can be treated as a special parallel loop, with loop upper bound being 1. Thus, the nested loop in Fig. 2(a) can be treated as the nested loop in Fig. 2(b).

In light of the above discussion, we assume that

1. All innermost loops in a general parallel nested loop are parallel loops.
2. Scalar code at the same nesting level as other parallel loops is treated as a special parallel loop with loop upper bound being 1.

C. Two-Level Self-Scheduling

Since after we “standardize” a general nested parallel loop structure as described in the last section, all of the innermost loops for scheduling are parallel loops. We should take advantage of their regular loop structure. We can use “fetch-and-op” instructions and schemes such as GSS or SDSS to minimize the scheduling overhead and balance the workload among processors. On the other hand, the higher level structures enclosing these innermost parallel loops (i.e., the outer loops which are not perfectly nested and might have serial loops and IF-THEN-ELSE constructs) can be quite complicated. Implicit loop coalescing, loop distribution, and schemes such as GSS or SDSS, are generally not applicable at this level. A task pool is required to handle such complexity.

We thus propose a two-level self-scheduling scheme. At the high level, each instance\(^1\) of an innermost parallel loop is considered as a task. These tasks are activated according to the precedence relation among them. Active tasks (i.e., those whose predecessors are all completed) are placed in a task pool to be assigned to processors by the high-level self-scheduling scheme. We propose multiple parallel linked lists for the task pool to avoid a serial bottleneck. There are other parallel data structures, such as those described in [24], that can also be used to implement the task pool. The comparison of these parallel data structures is beyond the scope of this paper.

\(^1\) Remember that if an innermost parallel loop is nested in several outer loops, it can have many instances. Each instance is an invocation of the innermost parallel loop with a unique index vector of the enclosing outer loops.
At the low level, each task, which is an instance of an innermost parallel loop, has many loop iterations. Each iteration or several iterations of the loop are partitioned into a subtask to be scheduled on one processor. We can use simple and efficient "fetch-and-op" operations to perform self-scheduling at this level and use GSS or SDSS to enhance the performance.

Such a two-level scheme allows us to find a balance between granularity and scheduling overhead. At the high level, the large granularity of a task can easily justify the relatively large scheduling overhead, which includes resolving the precedence relation, managing the task pool, updating status information, etc. At the low level, each subtask consists of only one or several iterations of a loop. The granularity is much smaller. We switch to a more efficient scheme to take advantage of the regularity of its loop structure. As a matter of fact, a task can even include more than one nested parallel loop as long as low-overhead scheduling schemes still apply. For example, the parallel loops K1 and K2 in the program in Fig. 3(a) can be implicitly coalesced to a single parallel loop K as shown in Fig. 3(b) assuming the loop bound P2 is independent of the loop index K1. The idea is to make a task large enough to offset the scheduling overhead. However, to simplify our presentation, we still assume each task contains only one innermost parallel loop.

D. The Macro-Dataflow Graph and Its Representation

A macro-dataflow graph is used to denote the precedence relation at the high level. We use the program in Fig. 1 as an example. Its macro-dataflow graph is in Fig. 4. Each circular node in the graph is an instance of an innermost parallel loop, and a diamond node is an instance of the condition for an IF-THEN-ELSE construct. A bar denotes an instance of innermost parallel loop B with an outer loop index vector (I, J) = (x, y). A diamond node has two successors: one for the TRUE branch and the other for the FALSE branch. A node becomes active if all of its predecessors are completed. The completion of a diamond node can activate either successor, but not both.

Initially, only the nodes without predecessors are active (e.g., nodes A1 and A2). When a node is completed, it may activate several nodes. For example, the completion of A1 or A2 will activate two instances of the innermost parallel loop B, i.e., B11 and B12 or B21 and B22. We use the array BAR.COUNT to record the number of completed iterations for the outer parallel loops. For example, BAR.COUNT(1:3) will be used to record the number of completed iterations for the outer parallel loop I and loop J.

The completion of an instance of an innermost parallel loop may activate instances of different innermost parallel loops depending on the loop level at which it is completed. For example, the completion of an instance of the parallel loop D activates an instance of the parallel loop C in the next iteration of the serial loop K, or it may activate an instance of the parallel loop E if the iterations of the serial loop K have been exhausted.

The precedence relation in a macro-dataflow graph can be described by a few arrays. Suppose there are m innermost parallel loops in a program, numbered from the top to the bottom. Two arrays DEPTH and BOUND are used to describe all innermost parallel loops. DEPTH(1:m) is used to describe their nesting levels. In particular, the kth element of DEPTH is the number of loops (including both parallel and serial loops) that enclose the kth innermost parallel loop. BOUND(1:m) is used to store the loop bounds of the innermost parallel loops. BOUND(k) stores the loop bound of kth innermost parallel loop. It can be an integer or a pointer to an expression that calculates the loop bound if the loop bound is a function of the indexes of the outer loops. Arrays DEPTH and BOUND for the program in Fig. 1 are shown in Fig. 5.
For each innermost parallel loop, we use an array \( \text{DESCRPT}_i \) (\( 1 \leq i \leq m \)) to describe its enclosing loops. \( \text{DESCRPT}_i \) has \( \text{DEPTH}(i) \) entries. Entry \( \text{DESCRPT}_{i(j)} \) describes the type, the relative position, and other information of the enclosing loop at level \( j \). \( \text{DESCRPT}_i \) is defined as follows:

- **parallel**: array \( [1 \ldots \text{DEPTH}(i)] \) of record
  - **parallel**: indicates whether the loop at this level is parallel or serial.
  - **last**: indicates whether or not the innermost loop \( i \) is the last innermost loop enclosed in this loop.
  - **bound**: shows the loop bound of the enclosing loop at this level. It can be a constant or a pointer to an integer function of the indexes of the outer loops.
- **next**: indicates what the next innermost loop at this level is.
- **conditnl**: indicates whether or not the innermost loop \( i \) is in the TRUE branch of an IF-THEN-ELSE construct.
- **cond_exp**: gives a logical expression if conditnl is “yes.”
- **altern**: indicates the innermost loop in the FALSE branch, if conditnl is “yes.” It may be empty, because the FALSE branch is optional.

The \( \text{DESCRPT}_i \)s for the program in Fig. 1 are shown in Fig. 6. For the sake of clarity, we use letters \( A, B, \ldots, G, H \)
to denote the innermost parallel loops instead of numbers
1, 2, · · ·, 7, 8.

III. ALGORITHMS FOR THE TWO-LEVEL SELF-SCHEDULING SCHEME

We first present the data structure of the task pool for high-
level self-scheduling in Section III-A. The low-level and the
high-level self-scheduling algorithms are presented in Sections
III-B, and III-C, respectively.

A. The Task Pool

Suppose there are \( m \) innermost parallel loops. The task
pool for high-level self-scheduling is composed of \( m \) parallel
linked lists (see Fig. 7), each used to store active instances of
a particular innermost parallel loop. There is an \( m \)-bit control
word \( SW \). Each bit of \( SW \) indicates whether the corresponding
linked list is empty. In particular, \( SW(i) \) (\( 1 \leq i \leq m \)) is 0 if
the \( i \)th linked list is empty; otherwise, it is 1. \( \text{head}(i) \) and \( \text{tail}(i) \)
(\( 1 \leq i \leq m \)) are the pointers for the first and the last entry of
the \( i \)th linked list, respectively. If \( \text{head}(i) = \text{tail}(i) = \text{nil} \), the
\( i \)th linked list is empty. When a processor is searching for a
task in the task pool, it can find the first nonempty linked list
using a leading-one-detection operation on the control word
\( SW \). Then it searches for the first entry that needs processors
in that linked list. This hierarchical approach accelerates the
searching process at the high level. We present a searching
algorithm in Section III-C.

Each entry in a linked list represents an instance of the cor-
responding innermost parallel loop and is called an instance
control block (ICB). The data structure of an ICB is as follows:

ICB: record
  right: a pointer pointing to its right entry in the list.
  left: a pointer pointing to its left entry in the list.
  index: an index variable for the innermost parallel loop.
  pcount: a counter for the number of completed iterations.
  lvec: index vector of the enclosing loops.
end of record

The first set of data fields in an ICB, which includes \( \text{right} \)
and \( \text{left} \), is used to manage the linked list and is updated
by routines \text{DELETE} and \text{APPEND}. The second set of data
fields, which includes \( \text{index} \), \( \text{icount} \), \( \text{pcount} \), and \( \text{lvec} \), is used for self-scheduling to be discussed in the next section.

Subroutines \text{DELETE} and \text{APPEND}, shown in Algorithm
1 and Algorithm 2, are used to delete and append an ICB in a
linked list, respectively. Each has two parameters: \( ip \) is a
pointer to the ICB, and \( i \) is the number of the parallel linked
list containing that ICB. We use \( ip \rightarrow \text{right} \) to denote the right
field of the ICB pointed by \( ip \). Each parallel linked list may be
deleted or appended by multiple processors concurrently. To
guarantee the consistency of the data structure, we use a simple
lock, \( L(i) \) (\( 1 \leq i \leq m \)), for each linked list. Deleting or
appending an ICB in a linked list requires locking the linked
list at the beginning and unlocking it at the end of the opera-
tion. We can use a more sophisticated parallel algorithm
such as in [20] without locking an entire linked list. However,
since there are many linked lists in the high level, processors
can go to the next nonempty linked list when the \( i \)th linked
list is locked by resetting \( SW(i) \) to 0 (so the next leading one
detection instruction will not select the \( i \)th linked list). Not
much concurrency is lost here. If a linked list is not empty
after an ICB is deleted, \( SW(i) \) is set back to 1 before the
linked list is unlocked.

Algorithm 1:

subroutine \text{DELETE}(i, ip)

spin: \{ \( L(i) = 1; \text{Decrement}; \) \}
  if (failure) goto spin; /* lock the linked list */
  \( SW(i) = 0; \) /* reset \( SW(i) \) */
  \( y = ip \rightarrow \text{right}; \)
  \( x = ip \rightarrow \text{left}; \)
  if (\( x \neq \text{nil} \)) then
    \( x \rightarrow \text{right} = y; \) /* change the right pointer of
    the left ICB */
  else
    \( \text{head}(i) = y; \) /* change the head pointer
    \( \text{head}(i) */
  end if;
  if (\( y \neq \text{nil} \)) then
    \( y \rightarrow \text{left} = x; \) /* change the left pointer of the
    right ICB */
  else
    \( \text{tail}(i) = x; \) /* change the tail pointer \( \text{tail}(i) */
    end if;
  if (\( x \neq \text{nil} \) or \( y \neq \text{nil} \)) \( SW(i) = 1; \)
  /* set \( SW(i) \), if the list is nonempty */
  \{ \( L(i); \text{Increment}; \) /* unlock the linked list */
end subroutine

Algorithm 2:

subroutine \text{APPEND}(i, ip)

spin: \{ \( L(i) = 1; \text{Decrement}; \) \}
  if (failure) goto spin; /* lock the linked list */
  \( x \leftarrow \text{tail}(i); \)
  \( SW(i) = 0; \) /* reset \( SW(i) \) */
  \( ip \rightarrow \text{left} = x; \) /* set the left pointer of the
  new ICB */
  \( \text{tail}(i) = ip; \) /* change the tail pointer \( \text{tail}(i) */
  if (\( x \neq \text{nil} \)) then
    \( x \rightarrow \text{right} = ip; \) /* change the right pointer of
    the last ICB */
  else
    \( \text{head}(i) = ip; \) /* change the head pointer
    \( \text{head}(i) */
  end if;
  \( SW(i) = 1; \) /* set \( SW(i) \) */
  \{ \( L(i); \text{Increment}; \) /* unlock the linked list */
end subroutine

B. Low-Level Self-Scheduling

Each processor also keeps the following local variables:

\( i \): the number of the innermost parallel loop the processor
is working on.

\( b \): the loop bound of the innermost parallel loop.
ip: a pointer points to the ICB that corresponds to the instance of the innermost parallel loop.

j: an integer to store index of the iteration currently being executed.

lev: an integer to show the level at which the innermost loop exits when the instance is completed.

targ: a local variable to indicate which innermost loop is the successor.

loc_indexes: the index vector for the enclosing outer loops.

Algorithm 3 is the main procedure for the low-level self-scheduling. Each processor executes the same code in Algorithm 3 with its own set of local variables i, b, ip, and loc_indexes. When a processor begins the execution of the procedure, the value of the local variables, i, b, ip, and loc_indexes should have already been determined.

The code from “start” to “body” checks whether there is any remaining iteration in the instance. If a processor gets the last iteration of the instance (i.e., j = b), it calls DELETE (see Algorithm 1) to delete the corresponding ICB from the task pool, so later processors can get iterations from the next ICB. Notice that an ICB is deleted immediately after all of its iterations are scheduled (not completed). Processors are still executing scheduled iterations in the deleted ICB. The pointer to the deleted ICB is kept in their local variable ip. If all of the iterations of an ICB have been scheduled (i.e., if index > b), the processor counter pcount is decremented to signal the effect, and SEARCH is called to find a new active instance of some innermost loop. Local variables i, ip, b, and loc_indexes are then updated in SEARCH. The subroutine SEARCH is presented in the next section.

Algorithm 3:
start: {ip ← index ≤ b; Fetch(j)&Increment};
If (failure) then
  {ip ← pcount; Decrement};
  SEARCH(i, ip, b, loc_indexes);
  goto start;
else if (j = b) DELETE(i, ip);
end if;

body: {loop body of innermost parallel loop i with index vector loc_indexes and j}

update: {ip ← icount < b − 1; Increment};
if (failure) then
  if (lev − EXIT(i, loc_indexes) ≠ 0) then
    targ = DESCRIPT1(lev).next;
    ENTER(targ, lev);
  end if;
spin: {ip ← pcoun = 1; Decrement};
if (failure) goto spin;
release the ICB pointed by ip;
SEARCH(i, ip, b, loc_indexes);
end if;
goto start;

After a processor completes an iteration, it checks if all of the iterations have been completed. If not (i.e., icount < b − 1), the processor goes back to “start” without changing the values of i, ip, b, and loc_indexes. If all of the iterations have been completed (i.e., icount = b − 1), it activates new instances of innermost parallel loops by adding their ICB’s to the task pool. Two subroutines are used to activate new instances: EXIT and ENTER.

The subroutine EXIT detects the exit level of the current instance of the innermost parallel loop i. The level number is stored in lev. If the completion of the instance does not activate any other instance, EXIT returns 0. If it does, the processor calls ENTER to activate instances of the innermost parallel loop described by DESCRIPT1(lev).next (stored in targ). EXIT and ENTER are presented in the next section.

Notice that when all iterations of an instance are completed, the corresponding ICB must have been deleted from the task pool. But this ICB cannot be deallocated before all of the processors pointing to the ICB have updated their pointers; otherwise, these processors could be trapped after the ICB is deallocated. This is the reason why, in Algorithm 3, the last processor has to wait until the processor counter pcoun becomes 1 and reset it to 0. Then the ICB can be deallocated.
Algorithm 4.

subroutine SEARCH(i, ip, b, loc_indexes)
again: {SW  0; Fetch(temp)};
if (failure) goto again;
i = leading-one-detection(temp);
{L(i) = 1; Decrement}; /* lock the linked list */
if (failure) goto again;
if (SW(i) = 0) then
{L(i); Increment}; /* unlock the linked list */
goto again;
else
SW(i) = 0; /* reset SW(i) */
end if;
ib = head(i);
b = loop bound of the instance obtained from
BOUND(i);
loop: {ip = ipcount  b; Increment};
if (failure) then
y = ip  right;
if (y = nil) then
SW(i) = 1; /* set SW(i) */
{L(i); Increment}; /* unlock this linked list */
goto again;
end if;
if (ip = y); /* move to the right ICB */
goto loop;
end if;
loc_indexes = index vector stored in ip  ivec;
SW(i) = 1; /* set SW(i) */
{L(i); Increment}; /* unlock this linked list */
end subroutine

C. High-Level Self-Scheduling

SEARCH is shown in Algorithm 4. A processor calls
SEARCH to get a new ICB in the task pool (see Fig. 7). It
uses a leading-one-detection operation on the control word
SW to find the first nonempty unlocked linked list, and its
number is stored in the local variable i. Then it searches the
ith linked list from left to right for an ICB that needs pro-
cessors. During the search, other processors may try to delete
ICB's or append new ICB's to the linked list. To prevent de-
etion and appending during the search, we lock the linked list
before searching. After the linked list is locked, it needs to
retest if SW(i) is still 0 because between the time a processor
fetches SW and the time it locks the linked list, other pro-
cessors may have deleted the remaining ICB's and made the
linked list empty (i.e., SW(i) becomes 0). If that is the case,
the processor should unlock the linked list and go back to ac-
tess SW again. Otherwise, the processor proceeds to search
for the ICB whose processor counter, pcnt, does not exceed the
loop upper bound of the corresponding parallel loop.

The function EXIT determines the exit level when an in-
stance of an innermost parallel loop is completed. Exit level
is needed to determine the next innermost parallel loop to
be activated. The algorithm of EXIT is in Algorithm 5. It
checks if the completed innermost parallel loop is the last one
in the enclosing loop. If not, EXIT returns the level of that
enclosing loop (i.e., DEPTH(i) stored in the variable lvl).
If it is the last one, it then checks if its enclosing loop is a
parallel loop or a serial loop. If it is a parallel loop, the
corresponding element of BAR_COUNT is incremented. If
the corresponding element of BAR_COUNT has not reached
the loop bound, the completion of the instance does not ac-
tivate any new innermost parallel loop, and EXIT returns 0.
The loop bound can be obtained from DESCRIPT(i lvl).bound.
If the BAR_COUNT reaches the loop bound, variable lvl
is decremented and the next outer loop is checked. For a serial
enclosing loop, it determines whether the loop is completed by
checking the lvalueth component of loc_indexes and its cor-
responding loop bound. If the serial loop is not completed yet,
EXIT returns lvl after the lvalueth component of loc_indexes is
incremented. The reason for incrementing that component of
loc_indexes will be clear after we present the subroutine EN-
TER. If the serial loop is completed, lvl is decremented and
the function starts over again.

Algorithm 5:

function EXIT(i, loc_indexes)
lvl = DEPTH(i);
first: if (DESCRIPTR(lvl).last) then
if (DESCRIPTR(lvl).parallel) then
 increment the corresponding BAR_COUNTER;
if (BAR_COUNTER equals to the loop bound) then
 talent
 goto first;
else
EXIT = 0;
end if;
else
increment lvalueth index in loc_indexes;
EXIT = lvl;
end if;
end if;
else
EXIT = lvl;
end if;
end function;

The subroutine ENTER(cur, level) activates instances of in-
nermost parallel loop cur at the level level. It creates and
appends the corresponding ICB's to the linked list cur. The
algorithm of ENTER is shown in Algorithm 6.

Algorithm 6:

subroutine ENTER(cur, level)
while (DESCRIPTR(cur, level).cond_initl) do
if (DESCRIPTR(cur, level).cond_exp) then
goto cnlt;

else
  if (FALSE branch is nonempty) then
    cur = DESCRIPT(cur)((level).altern;
  else
    if ((level = EXIT(cur, loc_indexelem)) = 0) then
      cur = DESCRIPT(cur)((level).next;
    else
      return;
    end if;
  end if;
end if;
end while;

cntl: if (level = DEPTH(cur)) then
  level = level + 1
  if (DESCRPT(cur)(level).parallel) then
    M = loop bound obtained from DESCRIPT(cur)(level).bound;
    for k = 1 to M do
      store k to the level element of loc_indexelem;
      call ENTER(cur, level);
      end for;
    else
      store 1 to the level element of loc_indexelem;
      call ENTER(cur, level);
    end if;
  else
    create a new ICB pointed by ptr;
    copy the index vector from loc_indexelem into
    ptr -> ivec;
    ptr -> right = nil;
    ptr -> index = 1;
    ptr -> icount = 0;
    ptr -> pcount = 0;
    APPEND(cur, ptr);
  end if;
end subroutine

If the innermost parallel loop cur is in an IF THEN ELSE construct, its logical condition has to be evaluated. If the condition is FALSE, the variable cur is replaced by the number of the first innermost parallel loop in the FALSE branch (i.e., the innermost parallel loop indicated by DESCRIPT(cur)(level).altern). Since an IF-THEN-ELSE construct could be nested, this process should be repeated until an innermost loop not in an IF-THEN-ELSE construct at this level is found. If the FALSE branch is empty, the procedure EXIT is called much the same way as in Algorithm 3 when an instance is completed. The variable level is modified by the return value of EXIT. If EXIT returns 0, no instances need be activated, and ENTER quits here.

If there are new instances to be activated, several cases need to be considered (see Fig. 8). In Fig. 8(a), loops A and B are at the same level k. The completion of an instance of A activates an instance of B with the same index vector for the enclosing loops. In Fig. 8(b), B is one level deeper than A, and B is enclosed by a parallel loop with loop bound M. In this case, M instances of B should be activated. The first k elements of their index vectors are the same because they are enclosed in the same k outer loops, the (k + 1)th element, which corresponds to the loop J, can range from j to M. These M instances are activated by recursively calling ENTER M times. The case in Fig. 8(c) is similar to that in Fig. 8(b), except that B is enclosed in a serial loop. In this case, only one instance of B is activated. Obviously, if B is nested in s levels deeper than A as shown in Fig. 8(d), all new instances will be activated.

IV. PERFORMANCE AND OVERHEAD

A key requirement for a dynamic scheduling scheme is to keep its run-time scheduling overhead small. There can be three components in the overhead of the proposed scheme. First, a processor must access two shared synchronized variables for each iteration: the index control variable index and the iteration counter icount (see Algorithm 3). Let the average time of this component be $O_1$.

The second component is incurred when SEARCH is executed to search for a new ICB in the task pool. It includes one leading-one-detection operation on SW, search for the ICB that needs processors, and copying the index vector of the enclosing loops. The amount of this overhead depends on how frequently a processor switches from one ICB to another. Let the average time for this overhead be $O_2$. Suppose that the average execution time of innermost parallel loops is $\tau$, and the average number of iterations executed by a processor between two successive calls of SEARCH is $n$. The average scheduling overhead in the second component for an iteration becomes $O_2/n$.

The third component is incurred when a processor is executing EXIT and ENTER to activate new instances. The average time for this overhead is denoted by $O_3$. Notice that only the processor that completes the last iteration of an instance has to call EXIT and ENTER. If the average loop bound of the innermost parallel loops is N, the total number of calls to EXIT and ENTER is one Nth of the total number of iterations. The average scheduling overhead in the third component for an iteration is $O_3/N$. The processor utilization can be estimated by the following expression:

$$
\eta = \frac{\tau}{\tau + O_1 + \frac{O_2}{n} + \frac{O_3}{N}}.
$$

If all innermost parallel loops are Doall loops, we can schedule a chunk of k iterations at a time in the low-level self-scheduling. A processor can execute k iterations serially without scheduling overhead. Note that the average number of chunks executed by a processor between calls of SEARCH becomes $n' = n/k$. But busy-waiting is more likely in this case. $O_2$ will increase to $O_2(k)$, which is a nondecreasing function with respect to $k$. The average number of chunks of iterations for innermost parallel loops is $N' = N/k$. $O_3$ is the same as before. Hence, the processor utilization is

$$
\eta' = \frac{k\tau}{k\tau + O_1 + \frac{kO_2(k)}{n} + k \frac{O_3}{N}}.
$$
or

$$\eta' = \frac{\tau}{\tau + \frac{O_1}{k} + \frac{O_2(k)}{n} + \frac{O_3}{N}}$$  (7)

It is clear from (2) that there is an optimal $k$ which yields maximum processor utilization. That value of $k$ is usually machine-dependent.

V. CONCLUDING REMARKS

We have presented a two-level processor self-scheduling scheme for general parallel nested loops in which parallel loops, serial loops, and IF-THEN-ELSE constructs can be nested in an arbitrary order.

In the low level, we take advantage of the regular structure of the innermost parallel loops to reduce its scheduling overhead. The relatively small granularity of subtasks is helpful in balancing the workload among the processors when the execution time of the subtasks varies substantially. Many self-scheduling techniques such as loop coalescing [13], GSS [14], and SDSS [16] can be used to enhance the performance at this level.

In the high-level, we use a task pool to handle the irregular structure of the outer loops and the IF-THEN-ELSE constructs. A task is an instance of an innermost parallel loop. The larger granularity of the tasks at the high level justifies the use of more complicated structures, such as parallel linked lists. We use multiple parallel linked lists to implement the task pool. Using a leading-one-detection operation, a processor can quickly locate the parallel linked list that needs processors. This makes the search for an active task in this level very efficient.

The basic idea of processor self-scheduling is to avoid the involvement of the operating system. In our self-scheduling scheme, a program is instrumented to become self-contained. The overhead analysis also shows that the scheme can be quite efficient.

REFERENCES


Zhixi Fang (S’82-M’84) received the B.S. degree in mathematics from Fudan University, Shanghai, China, in 1970 and the M.S. and Ph.D. degrees in computer science from the University of Nebraska, Lincoln, in 1982 and 1984, respectively.

From 1984 to 1987, he was an Assistant Professor in the Department of Computer Science, Wichita State University, Wichita, KS. In the summers of 1986 and 1987, he was a Visiting Senior Computer Engineer in the Center for Supercomputing Research and Development at the University of Illinois, Urbana-Champaign. From 1987 to 1988, he was a consulting member of the technical staff at the Concurrent Computer Corporation. He is currently working on parallel compiler and supercomputing system design for Convex Computer Corporation. His current research interests are parallel processing, program restructuring, parallel algorithms for computer vision, scheduling, and synchronization.

Pei Yu Tang was born in Shanghai, China, on June 28, 1946. He graduated from the Department of Mathematics of Fudan University, Shanghai, China, in 1970 and received the Masters degree in computer science from the East China Normal University in 1982. He started his Ph.D. program in the Department of Computer Science at University of Illinois at Urbana-Champaign in 1984.

He was with the Center for Supercomputing Research and Development at UIUC until he completed the Ph.D. degree in 1988. Currently, he is with the Department of Computer Science of the Australian National University. His research interests include parallel computer architectures, restructuring compilers, scheduling, synchronization, and interprocessor communication.

Pen-Chung Yew (S’76-M’80-SM’87) received the B.S.E.E. degree from the National Taiwan University in 1972, the M.S. degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1977, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1981.

Since 1981, he has been involved in the architectural design and hardware implementation of the Cedar multiprocessor system. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering and in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He is also a Senior Computer System Engineer in the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His current research interests are parallel processing, programming architecture, parallelizing compiler, and performance evaluation.

Dr. Yew is a member of the Association for Computing Machinery.

Chuan-Qi Zhu was born in Shanghai, China, in 1943. He received the B.S. degree in mathematics from Fudan University, Shanghai, China in 1965.

From 1965 to 1982, he was with Fudan University where he designed and built two mainframe computers and several special purpose computers. From 1982 to 1984, he was a visiting scholar in the Department of Computer Science, University of Illinois at Urbana-Champaign. Since 1985 he has been on the academic staff in the Center for Supercomputing Research and Development at University of Illinois at Urbana-Champaign. He is currently working on hardware and architecture design of the Cedar supercomputer system. His research interests include interconnection networks, parallel processing, and computer architecture.