Efficient Local Search for DAG Scheduling

Min-You Wu, Senior Member, IEEE, Wei Shu, Senior Member, IEEE, and
Jun Gu, Senior Member, IEEE

Abstract—Scheduling DAGs to multiprocessors is one of the key issues in high-performance computing. Most realistic scheduling algorithms are heuristic and heuristic algorithms often have room for improvement. The quality of a scheduling algorithm can be effectively improved by a local search. In this paper, we present a fast local search algorithm based on topological ordering. This is a compaction algorithm that can effectively reduce the schedule length produced by any DAG scheduling algorithm. Thus, it can improve the quality of existing DAG scheduling algorithms. This algorithm can quickly determine the optimal search direction. Thus, it is of low complexity and extremely fast.

Index Terms—DAG scheduling, multiprocessors, fast local search, quality, complexity.

1 INTRODUCTION

Scheduling computations onto processors is one of the crucial components of a parallel processing environment. They can be performed at compile-time or runtime. Scheduling performed at compile-time is called static scheduling. Scheduling performed at runtime is called dynamic scheduling. The flexibility inherent in dynamic scheduling allows adaptation to unforeseen application requirements at runtime. However, load balancing suffers from runtime overhead due to load information transfers among processors, the load balancing decision-making process, and communication delay due to task relocation. Furthermore, most runtime scheduling algorithms utilize neither the characteristics information of application problems nor the global load information for load balancing decisions. The major advantage of static scheduling is that the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared to dynamic scheduling. Static scheduling can utilize the knowledge of problem characteristics to reach a well-balanced load.

We consider static scheduling algorithms that schedule an edge-weighted directed acyclic graph (DAG), also called a task graph or a macro-dataflow graph, to a set of homogeneous processors to minimize the completion time. Since the static scheduling problem is NP-complete in its general forms [6] and optimal solutions are known in restricted cases [3], [5], [7], there has been considerable research efforts in this area, resulting in many heuristic algorithms [19], [24], [4], [25], [20], [2], [14]. In this paper, instead of suggesting a new scheduling algorithm, we present an algorithm that can improve the scheduling quality of the existing scheduling algorithms by using a fast local search technique. This algorithm, called TASK (Topological Assignment and Scheduling Kernel), systematically minimizes a given schedule in topological order. In each move, the dynamic cost of a node is used to quickly determine the search direction. It can effectively reduce the length of a given schedule.

This paper is organized as follows: In the next section, we review DAG scheduling algorithms. In Section 3, the local search technique is described. The random local search algorithm is discussed in Section 4. In Section 5, we propose a new local search algorithm, TASK. Performance data and comparisons are presented in Section 6. Finally, Section 7 concludes this paper.

2 DAG SCHEDULING

A directed acyclic graph (DAG) consists of a set of nodes \( n_1, n_2, \ldots, n_n \) connected by a set of edges, each of which is denoted by \( e_{i,j} \). Each node represents a task and the weight of node \( n_i, w(n_i) \), is the execution time of the task. Each edge represents a message transferred from one node to another node and the weight of edge \( e_{i,j} \), \( w(e_{i,j}) \), is equal to the transmission time of the message. The communication-to-computation ratio (CCR) of a parallel program is defined as its average communication cost divided by its average computation cost on a given system. In a DAG, a node that does not have a parent is called an entry node, whereas a node that does not have a child is called an exit node. A node cannot start execution before it gathers all of the messages from its parent nodes. In static scheduling, the number of nodes, the number of edges, the node weight, and the edge weight are assumed to be known before program execution. The weight between two nodes assigned to the same processing element (PE) is assumed to be zero.

The objective in static scheduling is to assign nodes of a DAG to PEs such that the schedule length or makespan is minimized without violating the precedence constraints. There are many approaches that can be employed in static scheduling. In the classical approach [13], also called
list scheduling, the basic idea is to make a priority list of node, and then assign these nodes one by one to PEs. In the scheduling process, the node with the highest priority is chosen for scheduling. The PE that allows the earliest start time is selected to accommodate this node. Most of the reported scheduling algorithms are based on this concept of employing variations in the priority assignment methods, such as HLF (Highest Level First), LP (Longest Path), LPT (Longest Processing Time), and CP (Critical Path) [1], [24], [15]. In the following, we review some of the contemporary static scheduling algorithms, including the MCP, DSC, DLS, and CPN methods.

The Modified Critical Path (MCP) algorithm is based on the as-late-as-possible (ALAP) time of a node [24]. The ALAP time is defined as \( T_{\text{ALAP}}(n_i) = T_{\text{critical}} - \text{level}(n_i) \), where \( T_{\text{critical}} \) is the length of the critical path and \( \text{level}(n_i) \) is the length of the longest path from node \( n_i \) to an exit node, including node \( n_i \) [5]. The MCP algorithm was designed to schedule a DAG on a bounded number of PEs. It sorts the node list in the increasing ALAP order. The first node in the list is scheduled to the PE that allows the earliest start time, considering idle time slots. Then, the node is deleted from the list and this operation repeats until the list is empty.

The Dominant Sequence Clustering (DSC) algorithm is designed based on an attribute for a task graph called the dominant sequence (DS) [25]. A DS is defined, for a partially scheduled task graph, as the path with the maximum sum of communication costs and computation costs in the graph. Nodes on the DS are considered to be relatively more important than others. The ready nodes with the highest priority will be scheduled first. Then, the priorities of the child nodes of the scheduled node will be updated and this operation repeats until all nodes are scheduled. The dynamic cost is used to quickly determine the critical path length. This idea has been incorporated into our TASK algorithm to reduce its complexity.

The Dynamic Level Scheduling (DLS) algorithm determines node priorities by assigning an attribute, called dynamic level (DL), to each node at every scheduling step [20]. DL is the difference between the static level and the message-ready time. DLS computes DL for each ready node on all available processors. Suppose \( DL(n_i, J) \) is the largest among all pairs of ready nodes and available processors, from schedule \( n_i \) to processor \( J \). Repeat this process until all nodes are scheduled.

Recently, a new algorithm has been proposed by using the Critical Path Node (CPN) [16]. This algorithm is based on the CPN-dominant priority. If the next CPN is a ready node, it is put in the CPN-dominant list. For a nonready CPN, its parent node, \( n_p \), with the smallest ALAP time is put in the list if all the parents of \( n_p \) are already in the list. Otherwise, all the ancestor nodes of \( n_p \) are recursively included in the list before the CPN node is in the list. The first node in the list is scheduled to the PE that allows the earliest start time. Then, the scheduled node is removed from the list and this operation repeats until the list is empty. The CPN-dominant algorithm utilizes the two important properties of DAG: the critical path and the topological order. It potentially generates a good schedule.

Although these algorithms produce relatively good schedules, they are usually not optimal. Sometimes, the generated schedule is far from optimal. In this paper, we propose a fast local search algorithm, TASK, to improve the quality of schedules generated by an initial scheduling algorithm.

### 3 Local Search

Local search was one of the early techniques for combinatorial optimization. It has been applied to solve NP-hard optimization problems [12]. The principle of local search is to refine a given initial solution point in the solution space by searching the neighborhood of the solution point. Recently, a number of efficient heuristics for local search, i.e., conflict minimization [8], [21], random selection/assignment [22], [23], and pre and partial selection/assignment [22], [23], have been developed.

There are several significant local search solutions to the scheduling problems. The SAT1 algorithm was the first local search algorithm developed for the satisfiability problem during the late 1980s [8], [9], [10], [11]. This scheduling problem is well-known as a Max-Satisfiability problem. A local search solution to the SAT problem was applied to solve several large scale industrial scheduling problems.

Two basic strategies have been used in a local search. The first one is a random search in which the local search direction is randomly selected. If the initial solution point is improved, it moves to the refined solution point. Otherwise, another search direction is randomly selected. The random strategy is simple and effective for some problems, such as the \( n \)-queens problem [21]. However, it may not be efficient for other problems such as the microword length minimization [18] and the DAG scheduling problem.

The second strategy utilizes certain criteria to find a search direction that will most likely lead to a better solution point. In the microword length minimization [18], a compatibility class is considered only when moving some nodes from the class may reduce the cost function. This strategy effectively reduces the search space by guiding the search toward a more promising direction. The local search algorithm presented in this paper uses this strategy. With carefully selected criteria, a local search for DAG scheduling becomes very efficient and the scheduling quality can be improved significantly.

### 4 Random Local Search Algorithm

A number of local search algorithms for scheduling have been presented [16], [17]. A random local search algorithm for DAG scheduling, named \textit{FAST}, was given in [16] (see Fig. 1). In this algorithm, a node is randomly picked and then moved to a randomly selected PE. If the schedule length is reduced, the move is accepted. Otherwise, the node is moved back to its original PE. Each move, successful or not, takes \( O(e) \) time to compute the schedule length, where \( e \) is the number of edges in the graph. To reduce its complexity, a constant \( MAXSTEP \) is defined to limit the number of steps so that only \( MAXSTEP \) nodes are
searchstep = 0
do {
    pick a node \( n_i \) randomly
    pick a PE \( P \) randomly
    move \( n_i \) to PE \( P \)
    if schedule length does not improve
        move \( n_i \) back to its original PE
} while (searchstep++ < MAXSTEP)

Fig. 1. A random local search algorithm, FAST.

inspected. The time taken for the algorithm is proportional to \( e \times MAXSTEP \). \( MAXSTEP \) is set to 64 [16]. Moreover, randomly selected nodes and PEs may not be able to significantly reduce the length of a given schedule. Even if the \( MAXSTEP \) is equal to the number of nodes, leading to a complexity of \( O(en) \), the random search algorithm still cannot provide a satisfactory performance.

The FAST algorithm has been modified in [17], which is shown in Fig. 2. The major improvement is that it uses a nested loop for a probabilistic jump. The total number of search steps is \( MAXSTEP \times MAXCOUNT \). \( MARGIN \) is used to reduce the number of steps. \( MAXSTEP \) is set to 8, \( MAXCOUNT \) to 64, and \( MARGIN \) to 2 [17]. A parallel version of the FAST algorithm is named FASTEST. A speedup from 11.93 to 14.45 on 16 PEs has been obtained for FASTEST [17].

5 LOCAL SEARCH WITH TOPOLOGICAL ORDERING FOR SCHEDULING

We propose a fast local search algorithm utilizing topological ordering for effective DAG scheduling. The algorithm is called TASK (Topological Assignment and Scheduling Kernel). In this algorithm, the nodes in the DAG are inspected in a topological order. In this order, it is not required to visit every edge to determine whether the schedule length is reduced. The time spent on each move can be drastically reduced so that inspecting every node in a large graph becomes feasible. Also, in this order, we can compact the given schedule systematically.

For a given graph, in order to describe the TASK algorithm succinctly, several terms are defined as follows:

- \( tlevel(n_i) \), the largest sum of communication and computation costs at the top level of node \( n_i \), i.e., from an entry node to \( n_i \), excluding its own weight, \( w(n_i) \) [26].
- \( blevel(n_i) \), the largest sum of communication and computation costs at the bottom level of node \( n_i \), i.e., from \( n_i \) to an exit node [26].
- \( CP \), the critical path, is the longest path in a DAG. The length of the critical path of a DAG is

\[
L_{CP} = \max_{n_i \in V}\{L(n_i)\},
\]

where \( L(n_i) = tlevel(n_i) + blevel(n_i) \) and \( V \) is the node set of the graph.

The TASK algorithm is applied to a previously scheduled DAG. In this case, a scheduled DAG is constructed, which contains scheduling and execution order information [25]. To enforce the execution order in each PE, some pseudoeedges (with zero weights) are inserted to incorporate the initial schedule into the graph. The above definitions for \( tlevel, blevel, \) and \( CP \) are still applied to the scheduled DAG.

Then, we define more terms:

- Node \( n_i \) has been scheduled on PE \( pe(n_i) \).

BestSL = infinity; searchcount = 0; /* BestSL: Best schedule length */
repeat
    searchstep = 0; counter = 0;
    do {
        pick a node \( n_i \) randomly
        pick a PE \( P \) randomly
        move \( n_i \) to PE \( P \)
        if schedule length does not improve
            move \( n_i \) back to its original PE and increment counter;
        otherwise set counter to 0;
    } while (searchstep++ < MAXSTEP and counter < MARGIN);
    if BestSL > SL(NewSchedule) then /* SL(S): Schedule length of schedule S */
        BestSchedule = NewSchedule;
        BestSL = SL(NewSchedule);
    endif
    NewSchedule = Randomly pick a node from the critical path and move it to another processor;
until (searchcount++ > MAXCOUNT);

Fig. 2. The modified FAST algorithm.
procedure TASK (DAG.Schedule)
begin
  /* initialization */
  Construct a scheduled DAG;
  for node i := 0 to n - 1 do
    \[ L(n_i) := tlevel(n_i) + blevel(n_i); \]
    \[ L_{CP} := \max_{0 \leq i < n} L(n_i), \]
    the longest path in DAG;
  /* search */
  while there are nodes in DAG to be scheduled do
    i := pick.a.node.with.Max.L(n_i);
    for each PE k
      obtain \( L^k(n_i) \) by moving \( n_i \) to PE k;
      \[ t := \text{pick.a.PE.with.Min.} L^k, \]
      where \( k = 0, \ldots, p - 1; \)
    /\* if no improvement */
    if \( t = p(e(n_i)) \) then
      let node \( n_i \) stay at PE \( p(e(n_i)) \);
    /\* if there are improvements */
    else begin
      move node \( n_i \) from PE \( p(e(n_i)) \) to PE \( t \);
      modify.pseudo.edges.in.DAG;
      propagate.tlevel.of.n_i.to.its.children;
    end;
    mark \( n_i \) as being scheduled;
  end;
end;

Fig. 3. TASK: Topological Assignment and Scheduling Kernel, a local search algorithm based on topological ordering for fast scheduling.

- Let \( p(n_i) \) be the predecessor node that has been scheduled immediately before node \( n_i \) on PE \( p(e(n_i)) \). If node \( n_i \) is the first node scheduled on the PE, \( p(n_i) \) is null.
- Let \( s(n_i) \) be the successor node that has been scheduled immediately after node \( n_i \) on PE \( p(e(n_i)) \). If node \( n_i \) is the last node scheduled on the PE, \( s(n_i) \) is null.

A sketch TASK algorithm is shown in Fig. 3 and the detailed description of the TASK algorithm in Fig. 4. One of the characteristics of this TASK algorithm is its independence from the algorithm that was used to generate the initial schedule. A node is labeled as \( n_i \) and its current PE number is \( p(e(n_i)) \). As long as the initial schedule is correct and every node \( n_i \) has available \( p(e(n_i)), p(n_i), \) and \( s(n_i) \) nodes, application of the local compaction algorithm guarantees that the new schedule of the graph is better than or equal to the initial one.

The input of the algorithm is a given DAG schedule generated by any heuristic DAG scheduling algorithm. First, a scheduled DAG is constructed. A pseudoedge may be added with zero communication time, that is, no data are transferred along the edge. Step 2 computes the value of \( blevel \) for each node in the scheduled DAG and initializes \( tlevel \) for entry nodes. All edges are marked unvisited. The variable \( \text{next}_{k} \) points to the next node that has not been inspected in PE \( k \). Initially, none of nodes is inspected, so \( \text{next}_{k} \) points to the first node in PE \( k \).

In Step 3, a ready node \( n_i \), with the maximum value \( L(n_i) = tlevel(n_i) + blevel(n_i) \), is selected for inspection. Ties are broken by \( tlevel(n_i) \). For the same \( tlevel(n_i) \), ties are broken randomly. A node is ready when all its parents have been inspected. In this way, the nodes are inspected in a topological order. Although other topological orders, such as \( blevel \), \( tlevel \), or \( CPN\text{-dominate} \), can be used, \( tlevel + blevel \) has been shown to be a good indicator for the order of inspection [24], [25].

To inspect node \( n_i \), in Step 4, the value:

\[ L(n_i) = tlevel(n_i) + blevel(n_i) \]

is recalculated for each PE. To conduct the recalculation at PE \( k \), node \( n_i \) is pretended to be inserted right in front of \( \text{next}_{k} \). Here, \( tlevel(n_i) \) can be varied if any of its parent nodes was scheduled to either PE \( k \) or PE \( p(e(n_i)) \). Similarly, \( blevel(n_i) \) can be varied if any of its child nodes was initially scheduled to either PE \( k \) or PE \( p(e(n_i)) \). Because the \( tlevels \) of its parent nodes are available and the \( blevels \) of its child nodes are unchanged, the value of \( L(n_i) \) in every PE can be easily computed. The values indicate the degree of improvement by a local search. With the new \( L(n_i) \) recalculated for every PE, node \( n_i \) is then moved to the PE that allows the minimum value of \( L(n_i) \). If node \( n_i \) has been moved to PE \( t \), the corresponding pseudoeedges are modified in Step 5. The \( tlevel \) of \( n_i \) is propagated to its children so that, when a node becomes ready, its \( tlevel \) can be computed. This process continues until every node is inspected.

The TASK algorithm satisfies the following properties.

**Theorem 1.** The critical path length \( L_{CP} \) will not increase after each step of the TASK algorithm.

**Proof.** The \( L(n_i) \) of node \( n_i \) is determined by the longest path that includes \( n_i \). Assume \( L(n_i) \) of node \( n_j \) increases as a result of moving node \( n_i \). Then, \( n_i \) and \( n_j \) must be on the same path from an entry node to an exit node. Because \( L(n_j) \) increases, this path must be the longest path that includes \( n_j \) and it determines the value of \( L(n_j) \). If this path determines the value of \( L(n_i) \), too, \( L(n_i) = L(n_j) \). Otherwise, a longer path determines \( L(n_i) \) and \( L(n_i) > L(n_j) \). In each step, \( L(n_i) \) will not increase and \( L(n_i) \leq L_{CP} \). Thus, \( L(n_i) \leq L_{CP} \). Since the \( L \) value of every node is not larger than \( L_{CP} \), \( L_{CP} \) will not increase. \( \square \)

If \( n_i \) is a node on a critical path, reduction of its \( L(n_i) \) value implies the reduction of the critical path length of the entire graph. (It may not immediately reduce the critical path length in the case of parallel critical paths.) If \( n_i \) is not a node on a critical path, reducing its \( L(n_i) \) value does not reduce the critical path length immediately. However, it increases the possibility of length reduction in a later step.

In the TASK algorithm, \( tlevel \) and \( blevel \) values are reused so that the complexity in determining \( L \) is reduced. The following theorems explain how the topological order makes the complexity reduction possible.

**Theorem 2.** If the nodes in a DAG are inspected in a topological order and each ready node is appended to the previous node list.
Step 1. Constructing a scheduled DAG:

For each node $n_i$ that is not the last node in a PE

let $n_j = s(n_i)$, if there exists no $e_{i,j}$, create a pseudo edge $e_{i,j}$ from $n_i$ to $n_j$ with $w(e_{i,j}) = 0$

Step 2. Initialization:

For each node $n_i$

compute $blevel(n_i)$ by considering pseudo edges

if it is an entry node, mark $n_i$ as ready and initialize $tlevel(n_i) = 0$

Mark every $e_{i,j}$ as unvisited

For each PE $k$

let $next_k$ point to the first node in the PE

Step 3. Selection:

Pick the ready node $n_i$ with the highest value of $L(n_i) = tlevel(n_i) + blevel(n_i)$

Ties are broken by $tlevel(n_i)$; for the same $tlevel(n_i)$, ties are broken randomly

Step 4. Inspection:

For each PE $k$, recompute $L^k(n_i)$ by assuming $n_i$ be moved to PE $k$ and inserted before $next_k$

Find a PE $t$ such that $L'(n_i) = \min(L^k(n_i), k = 0, ..., p - 1)$

Step 5. Compaction:

If $t = pe(n_i)$ /* node $n_i$ will stay at PE $t$ */

let $next_t = s(n_i)$

else /* move node $n_i$ from PE $r = pe(n_i)$ to PE $t$ */

let $n_l = p(n_i)$ and $n_m = s(n_i)$

delete edge $e_{l,i}$ if it is a pseudo edge

delete edge $e_{i,m}$ if it is a pseudo edge

if no edge $e_{l,m}$ previously exists

create a pseudo edge $e_{l,m}$ with $w(e_{l,m}) = 0$ and mark it as visited

let $s(n_l) = n_m$ and $p(n_m) = n_l$, and $next_r = n_m$

let $pe(n_i) = t$

let $n_y = next_t$ and $n_x = p(n_y)$; delete edge $e_{x,y}$ if it is a pseudo edge

create a pseudo edge $e_{x,i}$ if no edge $e_{x,i}$ previously exists

create a pseudo edge $e_{i,y}$ if no edge $e_{i,y}$ previously exists

let $s(n_y) = n_i$, $p(n_i) = n_x$, $s(n_i) = n_y$, and $p(n_y) = n_i$

Step 6. Propagation of $tlevel$

For each child node of node $n_i$, say $n_j$

mark edge $e_{i,j}$ as visited

if all incoming edges of $n_j$ are marked as visited

mark $n_j$ as ready and compute $tlevel(n_j)$

Repeat Steps 3-6 until all nodes are inspected

Fig. 4. The detailed description of the TASK algorithm.

The time complexity of the TASK algorithm is $O(e + np)$, where $e$ is the number of edges, $n$ is the number of nodes, and $p$ is the number of PEs.

Proof. Insertion of pseudoedges in Step 1 costs $O(n)$. Step 2 spends $O(e)$ time to compute the $blevel$ values. Step 3
To complete inspection of a node, a target PE must be selected from all the $p$ PEs, resulting in the cost of $O(np)$. Therefore, the total cost is $O(e + np)$. □

The TASK algorithm shares some concepts with the DSC algorithm [25]. The topological order is used to avoid repeated calculation of the dynamic critical path so that the complexity can be reduced. The task selection criteria of $t_{level} + b_{level}$ has been used in the MD [24] and DSC algorithms. It measures the importance of a node for scheduling and is proven as an efficient criteria of node selection. The TASK algorithm is different from the DSC algorithm in many aspects. DSC is an algorithm that schedules a DAG onto an unbounded number of clusters, whereas TASK is a local search algorithm that improves an existing schedule on a bounded number of processors. Although both DSC and TASK algorithms aim to reduce schedule length, DSC realizes it by merging clusters, whereas TASK realizes it by moving nodes among processors. In DSC, the merging of clusters is based on the gain in reduction of edges between a node and its parents. TASK goes one step further by considering the possible gain in reduction of edges between the node and its children, which potentially results in a better and more efficient decision.

In the following, we use an example to illustrate the operation of the TASK algorithm.

**Example.** Assume the DAG shown in Fig. 5 has been scheduled to three PEs by a DAG scheduling algorithm. The schedule is shown in Fig. 6a, in which three pseudo (dashed) edges have been added to construct a scheduled DAG: one from node $n_6$ to node $n_8$, one from node $n_3$ to node $n_9$, and one from node $n_4$ to node $n_5$ (not
shown in Fig. 6a). The schedule length is 14. The blevel of each node is computed as shown in Table 1. Tables 2 and 3 trace the $L$ values for each step. In Table 2, \"√\" indicates the node with the largest $L$ value and is to be inspected in the current step. In Table 3, \"*\" indicates the original PE and \"√\" is the PE to where the node is moved.

First, there is only one ready node, $n_1$, which is a CP node. Its $L$ value on PE 0 is:

$$L^0(n_1) = 0 + 14 = 14.$$ 

Then, the $L$ values on other PEs are computed:

$$L^1(n_1) = 0 + 14 = 14,$$

$$L^2(n_1) = 0 + 12 = 12,$$

as shown in Table 3. Thus, node $n_1$ is moved from PE 0 to PE 2, as shown in Fig. 6b. The $L_{CP}$ of the DAG is reduced to 12. In iterations 2, 3, and 4, moving nodes $n_2, n_3,$ and $n_4$ does not reduce any $L$ value. In iteration 5, node $n_6$ is moved from PE 0 to PE 1 as the $L$ value is reduced from 12 to 11, as shown in Fig. 6c. In the following five iterations, nodes $n_5, n_7, n_8, n_9,$ and $n_{10}$ do not move.

### Table 1

<table>
<thead>
<tr>
<th>Node</th>
<th>$n_1$</th>
<th>$n_2$</th>
<th>$n_3$</th>
<th>$n_4$</th>
<th>$n_5$</th>
<th>$n_6$</th>
<th>$n_7$</th>
<th>$n_8$</th>
<th>$n_9$</th>
<th>$n_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>blevel</td>
<td>14</td>
<td>9</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2

The $L$ values of ready nodes for selecting a node to be inspected

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Node</th>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n_1$ (0+14=14)</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$n_2$ (2+9=11), $n_3$ (3+9=12), $n_4$ (1+10=11)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$n_2$ (2+9=11), $n_4$ (1+10=11)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$n_2$ (2+9=11), $n_5$ (4+7=11)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$n_5$ (4+7=11), $n_6$ (6+6=12)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$n_5$ (4+7=11), $n_8$ (6+4=10), $n_9$ (8+2=10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$n_7$ (6+5=11), $n_8$ (6+4=10), $n_9$ (8+2=10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$n_8$ (6+4=10), $n_9$ (8+2=10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$n_9$ (8+2=10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$n_{10}$ (10+1=11)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3

The $L$ values of node $n_i$ on each PE to select a PE

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Node</th>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n_1$</td>
<td>0+14=14*</td>
<td>0+14=14</td>
<td>0+12=12 √</td>
</tr>
<tr>
<td>2</td>
<td>$n_3$</td>
<td>3+11=14</td>
<td>3+9 =12*</td>
<td>1+12=13</td>
</tr>
<tr>
<td>3</td>
<td>$n_4$</td>
<td>4+12=16</td>
<td>5+9 =14</td>
<td>1+10=11*</td>
</tr>
<tr>
<td>4</td>
<td>$n_2$</td>
<td>2+9 =11*</td>
<td>5+10=15</td>
<td>4+10=14</td>
</tr>
<tr>
<td>5</td>
<td>$n_6$</td>
<td>6+6 =12*</td>
<td>6+4 =10√</td>
<td>6+9 =15</td>
</tr>
<tr>
<td>6</td>
<td>$n_5$</td>
<td>5+10=15</td>
<td>8+10=18</td>
<td>4+7 =11*</td>
</tr>
<tr>
<td>7</td>
<td>$n_7$</td>
<td>9+6 =15</td>
<td>9+4 =13</td>
<td>6+5 =11*</td>
</tr>
<tr>
<td>8</td>
<td>$n_8$</td>
<td>6+4 =10*</td>
<td>8+4 =12</td>
<td>8+4 =12</td>
</tr>
<tr>
<td>9</td>
<td>$n_9$</td>
<td>10+3=13</td>
<td>8+2 =10*</td>
<td>8+4 =12</td>
</tr>
<tr>
<td>10</td>
<td>$n_{10}$</td>
<td>10+1=11</td>
<td>10+1=11*</td>
<td>10+1=11</td>
</tr>
</tbody>
</table>

#### 6 Performance Study

In this section, we present the performance results of the TASK algorithm and compare the TASK algorithm to the random local search algorithm, FAST. We performed experiments using synthetic DAGs as well as real workload generated from the Gaussian elimination program.
We use the same random graph generator in [17]. The synthetic DAGs are randomly generated graphs consisting of thousands of nodes. These large DAGs are used to test the scalability and robustness of the local search algorithms. These DAGs were synthetically generated in the following manner. Given $N$, the number of nodes in the DAG, we first randomly generated the height of the DAG from a uniform distribution with the mean roughly equal to $\sqrt{N}$. For each level, we generated a random number of nodes which were also selected from a uniform distribution with a mean roughly equal to $\sqrt{N}$. Then, we randomly connected the nodes from the higher level to the lower level. The edge weights were also randomly generated. The sizes of the random DAGs were varied from 1,000 to 4,000 with an increment of 1,000. Three values of the communication-computation-ratio (CCR) were selected to be 0.1, 1, and 10. The weights of the nodes and edges were generated randomly so that the average value of CCR corresponded to 0.1, 1, or 10. Performance data are the average over two hundred graphs.

We evaluate performance of these algorithms in two aspects: the schedule length generated by the algorithm and the running time of the algorithm. Tables 4 and 5 show the comparison of the modified FAST algorithm [17] and the TASK algorithm on four PEs and 16 PEs, respectively, where “CPN” is the CPN-Dominate algorithm, “FAST” is the modified FAST algorithm, and “TASK” is the TASK algorithm. The comparison is conducted for different sizes and different CCRs. The CPN-Dominate algorithm [16] generates the initial schedules. For the schedule length, the value in the column “CPN” is the length of the initial schedule. The value in the column “+FAST” is for initial scheduling plus the random local search algorithm. And, the value in the column “+TASK” is for initial scheduling plus the TASK algorithm.

### Table 4
Comparison for Synthetic DAGs with CPN as Initial Scheduling Algorithm (Four PEs)

<table>
<thead>
<tr>
<th># of nodes</th>
<th>CCR</th>
<th>Schedule length</th>
<th>Running time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPN  sd</td>
<td>+FAST  sd</td>
</tr>
<tr>
<td>1000</td>
<td>0.1</td>
<td>2536  27</td>
<td>2535  22</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2820  41</td>
<td>2814  25</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>5100  58</td>
<td>5091  47</td>
</tr>
<tr>
<td>2000</td>
<td>0.1</td>
<td>5011  47</td>
<td>5011  50</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>5508  68</td>
<td>5502  55</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10999 168</td>
<td>10979 110</td>
</tr>
<tr>
<td>3000</td>
<td>0.1</td>
<td>7730  45</td>
<td>7730  80</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>7705  89</td>
<td>7697  76</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>15622 202</td>
<td>15587 178</td>
</tr>
<tr>
<td>4000</td>
<td>0.1</td>
<td>10002 99</td>
<td>9997 100</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10672 112</td>
<td>10646 98</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>21444 349</td>
<td>21420 203</td>
</tr>
</tbody>
</table>

### Table 5
Comparison for Synthetic DAGs with CPN as Initial Scheduling Algorithm (16 PEs)

<table>
<thead>
<tr>
<th># of nodes</th>
<th>CCR</th>
<th>Schedule length</th>
<th>Running time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPN  sd</td>
<td>+FAST  sd</td>
</tr>
<tr>
<td>1000</td>
<td>0.1</td>
<td>663   8</td>
<td>663   8</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>961   10</td>
<td>960   10</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>3198  25</td>
<td>3185  22</td>
</tr>
<tr>
<td>2000</td>
<td>0.1</td>
<td>1350  11</td>
<td>1348  11</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1831  20</td>
<td>1829  17</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>6790  41</td>
<td>6789  42</td>
</tr>
<tr>
<td>3000</td>
<td>0.1</td>
<td>2234  32</td>
<td>2234  25</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2340  24</td>
<td>2339  24</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>8768  101</td>
<td>8766  91</td>
</tr>
<tr>
<td>4000</td>
<td>0.1</td>
<td>2930  11</td>
<td>2928  10</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2992  18</td>
<td>2992  21</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>13010 89</td>
<td>12990 92</td>
</tr>
</tbody>
</table>
Column “sd,” following each schedule value, is its standard deviation. The “%” columns, following “+FAST” and “+TASK,” are the percentage of improvement in the initial schedule. The running times of the CPN-Dominate algorithm, the modified FAST algorithm, and the TASK algorithm are also shown in the tables. It can be seen that TASK is much more effective and faster than FAST. The search order with the L value is superior to the random search order. In Table 5, for CCR = 10 on 16 PEs, the improvement ratio drops. In this case, the degree of parallelism to exploit is maximized and there is not much to do with it. The FAST algorithm is about two orders of magnitude slower than TASK, partly because $\text{MAXSTEP} \times \text{MAXCOUNT} = 256$.

The FASTEST algorithm running on 16 PEs is faster, but still one order of magnitude slower than TASK.

Tables 6 and 7 show the comparison with DSC [25] as the initial scheduling algorithm. The cluster merging algorithm shown in [26] maps the clusters to processors. The CPN-Dominate algorithm generates a better schedule for DAGs with smaller CCR and DSC is more efficient when CCR is large. For smaller CCR, DSC is not very good. Therefore, TASK produces a large improvement ratio. On the other hand, DSC is particularly suited for large CCR and TASK is unable to improve much from its result. In general, less improvement can be obtained by the TASK algorithm for a better schedule. This is because a good schedule leaves less room for improvement. The TASK algorithm normally provides uniformly consistent performance. That is, the schedule produced by TASK does not depend much on the initial schedule.

<table>
<thead>
<tr>
<th># of nodes</th>
<th>CCR</th>
<th>Schedule length</th>
<th>Running time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DSC</td>
<td>sd</td>
</tr>
<tr>
<td>1000</td>
<td>0.1</td>
<td>2742</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3145</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4450</td>
<td>34</td>
</tr>
<tr>
<td>2000</td>
<td>0.1</td>
<td>5332</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5845</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8989</td>
<td>102</td>
</tr>
<tr>
<td>3000</td>
<td>0.1</td>
<td>9020</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7987</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>12300</td>
<td>138</td>
</tr>
<tr>
<td>4000</td>
<td>0.1</td>
<td>11566</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11302</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18026</td>
<td>212</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of nodes</th>
<th>CCR</th>
<th>Schedule length</th>
<th>Running time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DSC</td>
<td>sd</td>
</tr>
<tr>
<td>1000</td>
<td>0.1</td>
<td>873</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1205</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3328</td>
<td>36</td>
</tr>
<tr>
<td>2000</td>
<td>0.1</td>
<td>1785</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2487</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7005</td>
<td>67</td>
</tr>
<tr>
<td>3000</td>
<td>0.1</td>
<td>3203</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3320</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8989</td>
<td>102</td>
</tr>
<tr>
<td>4000</td>
<td>0.1</td>
<td>4245</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3940</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>13362</td>
<td>98</td>
</tr>
</tbody>
</table>

**TABLE 6**
Comparison for Synthetic DAGs with DSC as Initial Scheduling Algorithm (Four PEs)

**TABLE 7**
Comparison for Synthetic DAGs with DSC as Initial Scheduling Algorithm (16 PEs)
We also tested the local search algorithms with the DAGs generated from a real application: Gaussian elimination with partial pivoting. The Gaussian elimination program operates on matrices. The matrix is partitioned by columns. The finest grain size of this column partitioning scheme is a single column. However, this fine-grain partition generates too many nodes in the graph. For example, the fine-grain partition of a \( 1 \times 1k \) matrix generates a DAG of 525,822 nodes. To reduce the number of nodes, a medium-grain partition is used. Table 8 lists the number of nodes in different matrix sizes and grain sizes (number of columns). The CCR is between 0.1 and 0.8. These graphs are generated by the Hypertool from an annotated sequential Gaussian elimination program [24]. The comparisons of the \textit{FAST} algorithm and the \textit{TASK} algorithm on different DAGs and a different number of PEs are shown in Tables 9 and 10, where Table 9 uses CPN as the initial scheduling algorithm and Table 10 uses DSC as the initial scheduling algorithm. In general, a cluster algorithm such as DSC performs well when communication of a DAG is heavy. Therefore, it generates better schedules for Gaussian elimination. \textit{TASK} performs better than \textit{FAST} in most cases and is much faster than \textit{FAST}.

7 CONCLUSION AND FUTURE WORKS

A local search is an effective method for solving NP-hard optimization problems. It can be applied to improve the quality of existing scheduling algorithms. \textit{TASK} is a low-complexity, high-performance local search algorithm for static DAG scheduling. It can quickly reduce the schedule length produced by any DAG scheduling algorithm. By utilizing the topological order, it is much faster and of much higher quality than the random local search algorithm.

We have demonstrated that \textit{TASK} was able to drastically reduce the schedule length produced by some well-known algorithms such as DSC and CPN. In future work, a comparison with the best scheduling algorithms such as MCP [24] will be conducted. A preliminary comparison showed that a small improvement was observed since the MCP produces very good results already.
ACKNOWLEDGMENTS

This research was partially supported by the U.S. National Science Foundation Grants CCR-9505300 and CCR-9625784, the Natural Sciences and Engineering Research Council of Canada Research Grant OGP0046423, the Natural Science and Engineering Research Council of Canada Strategic Grant MEF0045793, and the Natural Science and Engineering Research Council of Canada Strategic Grant STRO167029. We would like to thank the anonymous reviewers for their constructive comments.

REFERENCES


