Efficient Local Search for DAG Scheduling

Min-You Wu, Wei Shu
Department of Electrical and Computer Engineering
The University of New Mexico

Jun Gu
Department of Computer Science
The Hong Kong University of Science and Technology, Hong Kong

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 to improve. 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 applications' requirements at runtime. However, load balancing suffers from run-time overhead due to load information transfer among processors, 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 global load information for load balancing decision. 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 task graph or 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 effort 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 a *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 $\{n_1, n_2, ..., n_n\}$ 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 any parent is called an entry node, whereas a node that does not have any 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 nodes, 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 contemporary static scheduling algorithms, including 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 $T_L(n_i) = T_{critical} - level(n_i)$, where $T_{critical}$ is the length of the critical path, and $level(n_i)$ 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 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. 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-dominate priority. If the next CPN is a ready node, it is put in the CPN-dominate list. For a non-ready CPN, its parent node n_y with the smallest $ALAP$ time is put in the list if all the parents of n_y are already in the list; otherwise, all the ancestor nodes of n_y 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-dominate algorithm utilizes the two important properties of DAG: the critical path and 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 through 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 later '80s [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 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 *FAST*, was given in [16] (see Figure 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 inspected. The time taken for the algorithm is proportional to $e \times MAXSTEP$. $MAXSTEP$ is set to be 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.

---

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++ < $MAXSTEP$)

Figure 1: A random local search algorithm, *FAST*.

The *FAST* algorithm has been modified in [17], which is shown in Figure 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

```
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);
```

Figure 2: The modified *FAST* algorithm.

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].
- The critical path, *CP*, is the longest path in a DAG. The length of the critical path of a DAG is

$$L_{\mathrm{CP}} = \max_{n_i \in V}\{L(n_i)\},$$

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 *pseudo edges* (with zero weights) are inserted to incorporate the initial schedule into the graph. The above definitions of *tlevel*, *blevel*, and the critical path are still applied to the scheduled DAG. Then we define more terms:

- Node n_i has been scheduled on *PE* $pe(n_i)$.
- Let $p(n_i)$ be the *predecessor node* that has been scheduled immediately before node n_i on *PE* $pe(n_i)$. 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* $pe(n_i)$. If node n_i is the last node scheduled on the *PE*, $s(n_i)$ is *null*.

```
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≤i<n} L(n_i), the longest path in DAG;

    /* search */
    while there are nodes in DAG to be scheduled do
    begin
        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 := pick_a_PE_with_Min_L^k, where k = 0, ..., p − 1;
        /* if no improvement */
        if t == pe(n_i) then
            let node n_i stay at PE pe(n_i);
        /* if there are improvements */
        else begin
            move node n_i from PE pe(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;
```

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

A sketch *TASK* algorithm is shown in Figure 3 and the detailed description of the *TASK* algorithm in Figure 4. One of 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 $pe(n_i)$. As long as the initial schedule is correct and every node n_i has available $pe(n_i)$, $p(n_i)$, 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 pseudo edge may be added with zero communication time; that is, no data are transferred along the edge. Step 2 computes the value of *blevel* of each node in the scheduled DAG and initializes *tlevel* for entry nodes. All edges are marked *unvisited*. The variable $next_k$ points to the next node that has not been inspected in PE k. Initially, none of nodes is inspected so $next_k$ 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-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 re-calculated for each PE. To conduct the recalculation at *PE k*, node n_i is pretended to be inserted right in front of $next_k$. Here, $tlevel(n_i)$ can be varied if any of its parent nodes was scheduled to either *PE k* or *PE* $pe(n_i)$. Similarly, $blevel(n_i)$ can be varied if any of its child nodes was initially scheduled to either *PE k* or *PE* $pe(n_i)$. Because the *tlevel*s of its parent nodes are available and the *blevel*s 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 $L(n_i)$'s 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 pseudo edges 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 $L(n_j)$ 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

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 $L^t(n_i) = \min(L^k(n_i), k = 0, ..., p-1)$
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 $s(n_x) = n_i$, $p(n_i) = n_x$, $s(n_i) = n_y$, 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

Figure 4: The detailed description of the *TASK* algorithm.

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, $L(n_j) \leq L_{CP}$. Since the L value of every node is not larger than L_{CP}, L_{CP} will not increase. □

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 in the PE, the *blevel* of a node is invariant *before* it is inspected and the *tlevel* of a node is invariant *after* it is inspected.

Proof: If node n_i is not inspected, then the topological order implies that all descendants of n_i are not inspected. Therefore, the *blevel* of n_i is not changed since the *blevel* of all descendants of n_i are not changed. Once n_i is inspected, then the topological order implies that all ancestors of n_x have been inspected. Because a node is always appended to the previous scheduled nodes in the PE, the *tlevel* of an inspected node remains unchanged. □

Following a topological order of node inspection, we can localize the effect of edge zeroing on the L value of the nodes that have not been inspected. After each move, only the *tlevel* of currently inspected node is computed instead of computing *tlevel*s and *blevel*s of all nodes. Therefore, the time spent on computing L values is significantly reduced.

Theorem 3. 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 pseudo edges in the first step costs $O(n)$. The second step spends $O(e)$ time to compute the *blevel* values. The third step costs $O(n)$ for finding the highest L value. The main computational cost of the algorithm is in step 4. Computing the L value of each node costs $O(D(n_i))$ in inspecting every edge connected to n_i, where $D(n_i)$ is the degree of node n_i. For n steps, the cost is $\sum_{n_i \in V} O(D(n_i)) = O(e)$. 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 *tlevel+blevel* 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 1.

Assume the DAG shown in Figure 5 has been scheduled to three PEs by a DAG scheduling algorithm. The schedule is shown in Figure 6(a), 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 Figure 6(a)). The schedule length is 14. The *blevel* of each node is computed as shown in Table 1. Tables 2 and 3 trace the $tlevel + blevel = L$ values for each step. In Table 2, "$\surd$" 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 "$\surd$" the PE where the node is moved to.

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 Figure 6(b). The L_{CP} of the DAG is reduced to 12. In iterations 2, 3, and 4, moving nodes n_3, n_4, and n_2 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 Figure 6(c). In the following five iterations, nodes n_5, n_7, n_8, n_9 and n_{10} do not move.

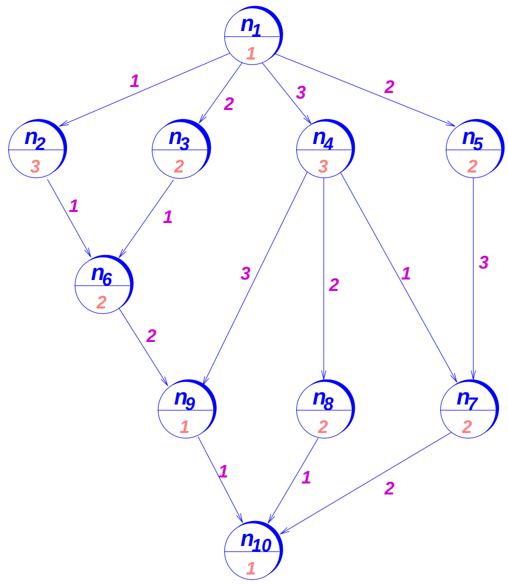

Figure 5: A DAG for Example 1.

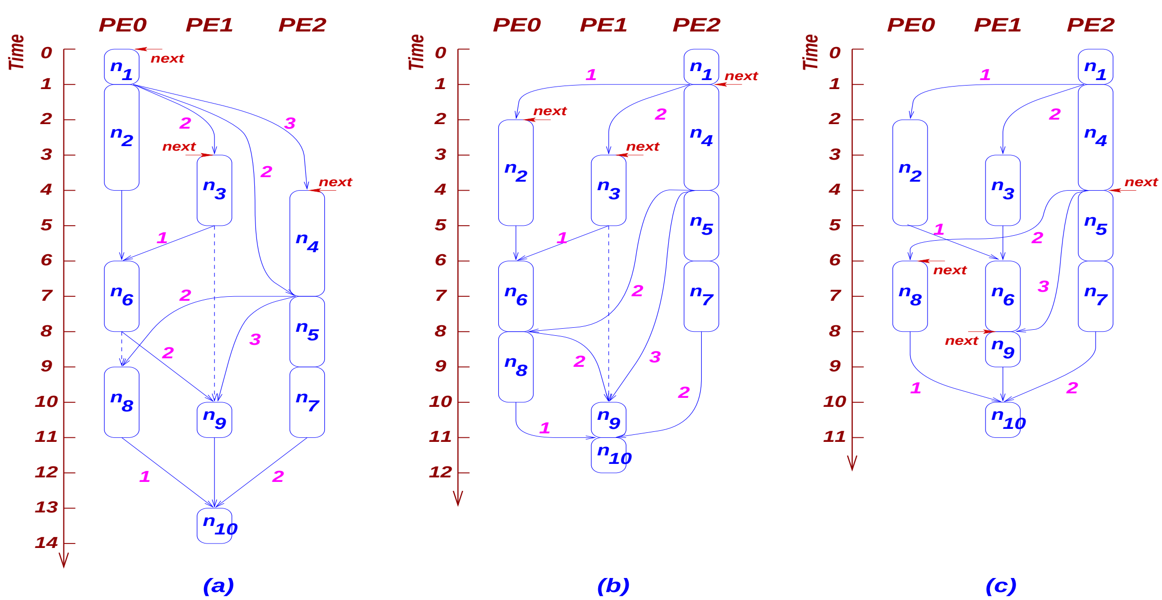

Figure 6: An example of *TASK*'s operations.

Table 1: The Initial *blevel* Value of Each Nodes for Example 1

Node	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
blevel	14	9	9	10	7	6	5	4	2	1

Table 2: The L Values of Ready Nodes for Selecting a Node To Be Inspected

Iteration			
1	n_1 (0+14=14) $\surd$		
2	n_2 (2+9=11),	n_3 (3+9=12) $\surd$,	n_4 (1+10=11)
3	n_2 (2+9=11),	n_4 (1+10=11) $\surd$	
4	n_2 (2+9=11) $\surd$,	n_5 (4+7=11)	
5	n_5 (4+7=11),	n_6 (6+6=12) $\surd$	
6	n_5 (4+7=11) $\surd$,	n_8 (6+4=10),	n_9 (8+2=10)
7	n_7 (6+5=11) $\surd$,	n_8 (6+4=10),	n_9 (8+2=10)
8	n_8 (6+4=10) $\surd$,	n_9 (8+2=10)	
9	n_9 (8+2=10) $\surd$		
10	n_{10} (10+1=11) $\surd$		

Table 3: The L Values of Node n_i on Each PE to Select a PE

Iteration	Node	PE 0	PE 1	PE 2
1	n_1	0+14=14*	0+14=14	0+12=12 $\surd$
2	n_3	3+11=14	3+9 =12*	1+12=13
3	n_4	4+12=16	5+9 =14	1+10=11*
4	n_2	2+9 =11*	5+10=15	4+10=14
5	n_6	6+6 =12*	6+4 =10$\surd$	6+9 =15
6	n_5	5+10=15	8+10=18	4+7 =11*
7	n_7	9+6 =15	9+4 =13	6+5 =11*
8	n_8	6+4 =10*	8+4 =12	8+4 =12
9	n_9	10+3=13	8+2 =10*	8+4 =12
10	n_{10}	10+1=11	10+1=11*	10+1=11

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 1000 to 4000 with an increment of 1000. 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 hundreds graphs.

We evaluated 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] adn the *TASK* algorithm on 4 PEs and 16 PEs, respectively, where "*CPN*" is the CPN-Dominate algorithm, "*FAST*" the modified *FAST* algorithm, and "*TASK*" 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. The 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 $MAXSTEP \times MAXCOUNT = 256$. The *FASTEST* algorithm running on 16 PEs is faster, but still one order of magnitude slower than *TASK*.

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

# of nodes	CCR	Schedule length								Running time (sec)		
		CPN	*sd*	*+FAST*	*sd*	%	*+TASK*	*sd*	%	*CPN*	*FAST*	*TASK*
1000	0.1	2536	27	2535	22	0.06	2529	23	0.3	0.49	52.6	0.51
	1	2820	41	2814	25	0.2	2671	25	5.3	0.10	13.4	0.16
	10	5100	58	5091	47	0.2	4463	44	12.5	0.20	24.1	0.27
2000	0.1	5011	47	5011	50	0.0	4995	37	0.3	1.12	132	1.23
	1	5508	68	5502	55	0.1	5225	48	5.1	0.45	55.1	0.53
	10	10999	168	10979	110	0.2	9472	85	13.9	0.52	66.7	0.64
3000	0.1	7730	45	7730	80	0.0	7577	78	2.0	0.69	87.2	0.88
	1	7705	89	7697	76	0.1	7469	80	3.1	2.30	250	2.51
	10	15622	202	15587	178	0.2	13149	181	15.8	0.91	119	1.20
4000	0.1	10002	99	9997	100	0.05	9925	86	0.8	1.51	228	1.82
	1	10672	112	10646	98	0.2	10144	75	5.0	2.34	287	2.68
	10	21444	349	21420	203	0.1	18379	230	14.3	1.47	173	1.74

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

# of nodes	CCR	Schedule length								Running time (sec)		
		CPN	*sd*	*+FAST*	*sd*	%	*+TASK*	*sd*	%	*CPN*	*FAST*	*TASK*
1000	0.1	663	8	663	8	0.0	652	6	1.7	1.50	55.1	0.57
	1	961	10	960	10	0.1	912	8	5.1	0.35	13.2	0.19
	10	3198	25	3185	22	0.4	3088	24	3.4	0.67	24.0	0.32
2000	0.1	1350	11	1348	11	0.1	1318	12	2.4	3.85	140	1.50
	1	1831	20	1829	17	0.1	1740	17	5.0	1.40	54.2	0.61
	10	6790	41	6789	42	0.01	6479	38	4.6	1.68	63.0	0.74
3000	0.1	2234	32	2234	25	0.0	2156	22	3.5	2.22	89.1	0.96
	1	2340	24	2339	24	0.0	2262	13	3.3	7.50	154	2.66
	10	8768	101	8766	91	0.02	8470	88	3.4	3.06	119	1.28
4000	0.1	2930	11	2928	10	0.07	2777	15	5.2	5.00	198	1.98
	1	2992	18	2992	21	0.0	2864	22	4.3	7.51	167	2.83
	10	13010	89	12990	92	0.2	12457	95	4.3	4.78	173	1.91

Table 6: Comparison for Synthetic DAGs with DSC as Initial Scheduling Algorithm (4 PEs)

# of nodes	CCR	Schedule length								Running time (sec)		
		DSC	*sd*	+*FAST*	*sd*	%	+*TASK*	*sd*	%	*DSC*	*FAST*	*TASK*
1000	0.1	2742	16	2650	22	3.4	2555	24	6.9	1.28	55.0	0.54
	1	3145	33	3002	28	4.5	2756	28	12.4	0.60	16.1	0.16
	10	4450	34	4413	33	0.8	4281	35	3.8	0.76	25.8	0.28
2000	0.1	5332	56	5224	24	2.0	5100	31	4.4	4.23	154	1.33
	1	5845	44	5812	52	0.5	5310	49	9.2	2.83	54.3	0.59
	10	8989	102	8902	97	1.0	8625	56	4.0	2.63	63.2	0.67
3000	0.1	9020	97	8966	75	0.6	7898	77	12.4	5.90	99.2	0.90
	1	7987	88	7883	59	1.3	7587	66	5.0	8.94	287	2.58
	10	12300	138	12289	112	0.1	11847	86	3.7	5.77	123	1.22
4000	0.1	11566	87	11489	78	0.7	10476	36	9.4	11.97	199	1.88
	1	11302	102	11222	98	0.7	10243	93	9.4	12.73	287	2.97
	10	18026	212	17879	165	0.8	17011	168	5.6	9.79	176	1.84

Table 7: Comparison for Synthetic DAGs with DSC as Initial Scheduling Algorithm (16 PEs)

# of nodes	CCR	Schedule length								Running time (sec)		
		DSC	*sd*	+*FAST*	*sd*	%	+*TASK*	*sd*	%	*DSC*	*FAST*	*TASK*
1000	0.1	873	6	867	4	0.7	684	5	22.6	1.23	48.0	0.59
	1	1205	10	1169	6	3.0	975	6	19.1	0.57	13.2	0.22
	10	3328	36	3320	30	0.2	3092	27	7.1	0.74	22.3	0.32
2000	0.1	1785	12	1758	11	1.5	1360	14	23.8	4.35	134	1.60
	1	2487	19	2481	20	0.2	1887	16	24.1	2.62	50.2	0.73
	10	7005	67	6992	70	0.2	6687	54	4.5	2.51	59.8	0.81
3000	0.1	3203	12	3203	24	0.0	2362	18	26.3	5.53	88.2	1.09
	1	3320	45	3292	38	0.8	2580	28	22.3	7.57	267	2.98
	10	8989	102	8962	79	0.3	8432	76	6.2	4.96	107	1.30
4000	0.1	4245	28	4233	33	0.3	3021	27	28.8	10.81	180	2.10
	1	3940	40	3910	35	0.8	3018	33	23.4	11.98	276	3.16
	10	13362	98	13361	105	0.0	12901	56	3.5	8.98	160	2.01

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.

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 1k$\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 *FAST* algorithm and the *TASK* algorithm on different DAGs and 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. *TASK* performs better than *FAST* in most cases and is much faster than *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. *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 algorithms. By utilizing the topological order, it is much faster and of much higher quality than the random local search algorithm.

We have demonstrated that *TASK* was able to reduce drastically the schedule length produced by some well-known algorithms such as DSC and CPN. In the 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.

Table 8: The number of nodes in different matrix sizes and grain sizes for Gaussian elimination

Matrix size	1k×1k				2k×2k			
Grain size	64	32	16	8	64	32	16	8
# of nodes	138	530	2082	8258	530	2082	8258	32898

Table 9: Comparison for Gaussian elimination with CPN as Initial Scheduling Algorithm

Matrix size	Grain size	# of PEs	Schedule length					Running time (sec)		
			CPN	*+FAST*	%	*+TASK*	%	*CPN*	*FAST*	*TASK*
1k×1k	64	4	209.4	209.0	0.2	193.8	7.5	0.01	0.34	0.01
	32	8	109.8	109.8	0.0	97.4	11.3	0.01	1.28	0.02
	16	16	56.5	56.5	0.0	50.1	11.3	0.08	4.55	0.09
	8	32	28.9	28.9	0.0	26.1	9.4	0.62	24.9	0.56
2k×2k	64	8	876.1	876.1	0.0	786.0	10.3	0.01	1.11	0.01
	32	16	449.1	449.1	0.0	397.0	11.6	0.08	4.99	0.09
	16	32	228.3	228.3	0.0	199.4	12.7	0.62	25.3	0.58
	8	64	115.8	115.8	0.0	102.0	12.6	5.32	102	4.19

Table 10: Comparison for Gaussian elimination with DSC as Initial Scheduling Algorithm

Matrix size	Grain size	# of PEs	Schedule length					Running time (sec)		
			DSC	*+FAST*	%	*+TASK*	%	*DSC*	*FAST*	*TASK*
1k×1k	64	4	211.8	193.4	8.7	199.6	5.8	0.01	0.20	0.01
	32	8	97.1	95.2	1.9	95.9	1.3	0.09	1.22	0.02
	16	16	49.4	48.8	1.2	48.4	2.0	0.87	5.68	0.10
	8	32	25.0	24.9	0.4	24.7	1.2	3.79	24.6	0.66
2k×2k	64	8	872.4	817.3	6.3	805.9	7.7	0.09	1.34	0.02
	32	16	392.7	390.6	0.5	384.0	3.2	0.85	5.88	0.10
	16	32	200.2	199.3	0.4	196.1	2.1	3.25	30.2	0.69
	8	64	100.1	100.0	0.1	99.3	0.8	13.8	91.0	5.73

Acknowledgments

This research was partially supported by NSF Grants CCR-9505300 and CCR-9625784, NSERC Research Grant OGP0046423, NSERC Strategic Grant MEF0045793, and NSERC Strategic Grant STR0167029. We would like to thank the anonymous reviewers for their constructive comments.

References

[1] T. L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list scheduling for parallel processing systems. *Communications of ACM*, 17(12):685–690, December 1974.

[2] Y.C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputer '92*, November 1992.

[3] E. Coffman, editor. *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, 1976.

[4] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, June 1990.

[5] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[6] M.R. Gary and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[7] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, (5):287–326, 1979.

[8] J. Gu. Parallel algorithms and architectures for very fast search (PhD Thesis). Technical Report UUCS-TR-88-005, Jul. 1988.

[9] J. Gu. How to solve Very Large-Scale Satisfiability problems. Technical Report UUCS-TR-88-032. 1988, and UCECE-TR-90-002, 1990).

[10] J. Gu. Efficient local search for very large-scale satisfiability problem. *SIGART Bulletin*, 3(1):8–12, Jan. 1992, ACM Press.

[11] J. Gu and Q.P. Gu. Average time complexities of several local search algorithms for the satisfiability problem. Technical Report UCECE-TR-91-004, 1991. Also in *Lecture Notes in Computer Science*, Vol. 834, pp. 146-154, 1994 and to appear in *IEEE Trans. on Knowledge and Data Engineering.*

[12] J. Gu. Local search for satisfiability (SAT) problem. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(4):1108–1129, Jul. 1993, and 24(4):709, Apr. 1994.

[13] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[14] A.A. Khan, C.L. McCreary, and M.S. Jones. A comparison of multiprocessor scheduling heuristics. In *Int'l Conf. on Parallel Processing*, volume II, pages 243–250, August 1994.

[15] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, pages 23–32, January 1988.

[16] Y.K. Kwok, I.Ahmad, and J. Gu. FAST: A low-complexity algorithm for efficient scheduling of DAGs on parallel processors. In *Proc. of Int'l Conference on Parallel Processing*, pages II–150 — II–157, 1996.

[17] Y.K. Kwok and I.Ahmad. FASTEST: A practical low-complexity algorithm for compile-time assignment of parallel programs to multiprocessors. *IEEE Trans. Parallel and Distributed System*, 10(2):147–159, February 1999.

[18] R. Puri and J. Gu. Microword length minimization in microprogrammed controller synthesis. *IEEE Transactions on CAD*, (10):1449–1457, October 1993.

[19] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, 1989.

[20] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel and Distributed System*, 4(2):175–187, February 1993.

[21] R. Sosič and J. Gu. How to search for million queens. Technical Report UUCS-TR-88-008, Dept. of Computer Science, Univ. of Utah, Feb. 1988.

[22] R. Sosič and J. Gu. Fast search algorithms for the n-queens problem. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-21(6):1572–1576, Nov./Dec. 1991.

[23] R. Sosič and J. Gu. Efficient local search with conflict minimization. *IEEE Trans. on Knowledge and Data Engineering*, 6(5):661–668, Oct. 1994.

[24] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[25] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed System*, 5(9):951–967, September 1994.

[26] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message-passing Multiprocessors. *The 6th ACM Int'l Conf. on Supercomputing*, July 1992.

Figure 1. A random local search algorithm, *FAST*.

Figure 2. The modified *FAST* algorithm.

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

Figure 4. The detailed description of the *TASK* algorithm.

Figure 5. A DAG for Example 1.

Figure 6. An example of *TASK*'s operations.

Table 1. The Initial *blevel* Value of Each Nodes for Example 1

Table 2. The L Values of Ready Nodes for Selecting a Node To Be Inspected

Table 3. The L Values of Node n_i on Each PE to Select a PE

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

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

Table 6. Comparison for Synthetic DAGs with DSC as Initial Scheduling Algorithm (4 PEs)

Table 7. Comparison for Synthetic DAGs with DSC as Initial Scheduling Algorithm (16 PEs)

Table 8. The number of nodes in different matrix sizes and grain sizes for Gaussian elimination

Table 9. Comparison for Gaussian elimination with CPN as Initial Scheduling Algorithm

Table 10. Comparison for Gaussian elimination with DSC as Initial Scheduling Algorithm

Min-You Wu received the M.S. degree from the Graduate School of Academia Sinica, Beijing, China, and the Ph.D. degree from Santa Clara University, California. Before he joined the Department of Electrical and Computer Engineering, at the University of New Mexico, where he is currently an Associate Professor, he has held various positions at University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, State University of New York at Buffalo, and University of Central Florida. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 80 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a member of ACM and a senior member of IEEE. He is listed in International Who's Who of Information Technology and Who's Who in America.

Wei Shu received the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1990, the M.S. degree from Santa Clara University in 1984, and the B.S. degree from Hefei Polytechnic University, China, in 1982. Since then, she worked at Yale University, the State University of New York at Buffalo, and University of Central Florida. She is currently an Associate Professor in the Department of Electrical and Computer Engineering, the University of New Mexico Her current interests include dynamic scheduling, runtime support systems for parallel processing, parallel operating systems, and multimedia networks. She is a member of ACM and a senior member of IEEE.

Jun Gu received his BS in Electrical Engineering from the Univ. of Science and Technology of China in 1982 and his PhD in Computer Science from the Univ. of Utah in 1989. Since 1994, he has been a Professor of Electrical and Computer Engineering at the University of Calgary, Canada. He is presently on leave at Hong Kong University of Science and Technology. Jun Gu has been the Associate Editor-in-Chief of IEEE Computer Society Press Editorial Board, an Associate Editor of IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on VLSI Systems, Journal of Global Optimization, Journal of Combinatorial Optimization, and Journal of Computer Science and Technology, and is on the Advisory Board of International Book Series on Combinatorial Optimization. He was a Chair of the 1995 National Academy of Sciences Information Technology Forum and was a Chair of the 1996 NSF special event in celebration of 25 years of research on the satisfiability problem. Jun Gu is a Member of ACM, ISA, ISTS, and INNS, a Senior Member of IEEE, and a Life Member of AAAI.

Min-You Wu, Corresponding Author:

Dept of Electrical and Computer Engr.

The University of New Mexico

EECE Bldg., Room 125

Albuquerque, NM 87131-1356

tel: 505-277-1078

fax: 505-277-1439

e-mail: wu@eece.unm.edu

Wei Shu

Dept of Electrical and Computer Engr.

The University of New Mexico

EECE Bldg., Room 125

Albuquerque, NM 87131-1356

tel: 505-277-1433

fax: 505-277-1439

e-mail: shu@eece.unm.edu

Jun Gu

Computer Science Department

Hong Kong University of Science and Technology

Clear Water Bay, Kowloon

Hong Kong

tel: 852-2358-8766

e-mail: gu@cs.ust.hk