1996 International Conference on Parallel Processing

FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors

Yu-Kwong Kwok, Ishfaq Ahmad, and Jun Gu

Department of Computer Science
The Hong Kong University of Science and Technology, Hong Kong.

Abstract[1]

The DAG scheduling problem is a rich land of research and a plethora of algorithms for solving this problem have been reported in the literature. However, designing a scheduling algorithm of low complexity without sacrificing performance remains a challenging obstacle from a practical perspective. In this paper, we present a local search-based scheduling algorithm that attempts to meet this challenge. The proposed algorithm is called *Fast Assignment using Search Technique* (FAST). Its overall time complexity is only $O(e)$ where e is the number of edges in the DAG. The algorithm works by first generating an initial solution and then refining it using local neighborhood search. The algorithm outperforms numerous previous algorithms while taking dramatically smaller execution times. The distinctive feature of our research is that the performance evaluation is not carried out using simulation, rather we have tested our proposed algorithm and compared it with other algorithms using a parallel compiler with real applications on the Intel Paragon.

Keywords: Local Search, Multiprocessors, Parallel Processing, Scheduling, Task Graphs.

1 Introduction

To efficiently exploit the tremendous potential of high-performance architectures, the tasks of a parallel application must be carefully decomposed and scheduled to the processors so that the execution time is minimized. When the characteristics of the parallel program, such as execution times of the tasks, amount of communication data, and task dependencies are known *a priori*, scheduling can be done statically. The parallel program, with such known information, can be modeled as a node- and edge-weighted *directed acyclic graph* (DAG), in which the nodes and edges represent tasks and messages, respectively. With such a *static* model, the scheduler is invoked off-line or during compile-time and thus can afford moderate time complexity in order to generate a better schedule. This form of multiprocessor scheduling problem is called *static scheduling* or DAG scheduling.

Static scheduling in most cases is NP-complete [3], [4], and optimal solutions exist only in three simple cases: (i) scheduling a tree-structured DAG with identical node weights to an arbitrary number of processors [3], (ii) scheduling an arbitrary DAG with identical node weights to two processors [3], and (iii) scheduling an interval-ordered DAG to an arbitrary number of processors [3]. Thus, heuristic approaches are sought to tackle the problem under more realistic cases.

There have been a large number of static scheduling heuristics reported in the literature [1], [9], [11], [13]. However, while generating good solutions, very few of them have a low complexity. Thus, most of the algorithms are impractical in a real environment. In a recent study [1], we compared 21 such algorithms and made a number of findings. For example, we observed that an $O(v^3)$ scheduling algorithm (here v denotes the number of nodes in the DAG) can take more than an hour to schedule a DAG with 1,000 nodes (typical in many applications). Taking such a large amount of time to produce a schedule for an application is a major obstacle in using these algorithms with parallel compilers. Some algorithms have low complexities but their solution quality is not satisfactory [1]. The objective of this study is to propose a low complexity DAG scheduling algorithm that can produce efficient schedules.

The most common technique for DAG scheduling is to assign priorities to the nodes of the DAG, and allocate the higher priority nodes to the appropriate time slots on the processors [7]. The solution quality (schedule length) is highly dependent on the accuracy of the priorities. To determine the priorities more accurately, some algorithms spend extra computation steps. Backtracking or recalculation of priorities can incur even higher complexity.

While it is understood that static scheduling is *off-line* and some extra time can be afforded in generating a better solution, the time complexity of an algorithm is an important issue from a practical point of view. In this regard, some of the pioneering work done by Yang and Gerasoulis [18] addressed this problem and proposed some novel techniques for reducing the time complexity of the scheduling algorithms. The objective of our work is to propose an algorithm that has a comparable or lower complexity while producing even better solutions.

We propose a new algorithm using a technique called *local search* [6], [15] which has been successfully applied to solve many NP-hard optimization problems [12]. Our algorithm has two phases. In the first phase, we generate a moderately optimized schedule quickly without spending a large amount of time to generate a good schedule at one shot. Then, in the second phase, we employ the local search technique to refine the schedule. The overall time complexity of our algorithm, which is called *Fast Assignment using Search Technique* (FAST), is only $O(e)$ where e is the number of edges in the DAG. The performance evaluation is carried out not only using simulation for randomly generated task graphs, but the proposed algorithm is also tested and compared it with

1. This research was supported by the Hong Kong Research Grants Council under contract number HKUST 619/94E.

other algorithms using an actual parallel compiler with real applications on the Intel Paragon.

The rest of this paper is organized as follows. In the next section, we introduce the parallel program model used in the DAG scheduling problem. In Section 3, we describe some of the related work on DAG scheduling reported in the literature. In Section 4, we discuss our proposed algorithm. Section 5 contains the performance results, and the final section concludes the paper.

2 Background

In the static scheduling problem, a parallel program is typically modeled by a directed acyclic graph (DAG) $G = (V, E)$, where V is a set of v nodes and E is a set of e directed edges. A node in the DAG represents a task which is a set of instructions that must be executed sequentially in the same processor. The weight on a node is called the *computation cost* of a node n_i and is denoted by $w(n_i)$. The edges in the DAG, each of which is denoted by (n_i, n_j), correspond to the communication messages and precedence constraints among the nodes. The weight on an edge is called the *communication cost* of the edge and is denoted by $c(n_i, n_j)$. The source node of an edge is called the *parent* node while the destination node is called the *child* node. A node with no parent is called an *entry* node and a node with no child is called an *exit* node. A *Critical Path* (CP) of a DAG is a set of nodes and edges constituting a path which has the largest length. The length of the CP is the sum of the computation costs and communication costs along the path.

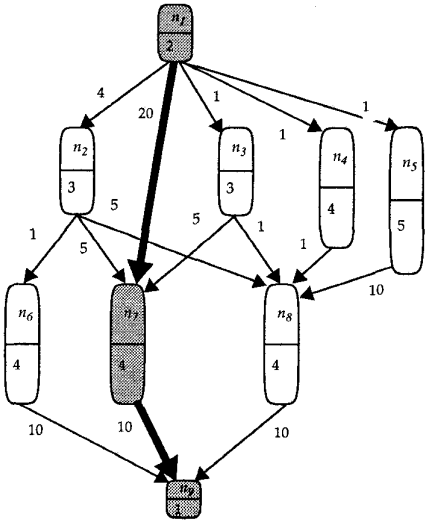
The *communication-to-computation-ratio (CCR)* of a parallel program is defined as its average communication cost divided by its average computation cost on a given system. A node is not *ready* (i.e., cannot start execution) before it gathers all of the messages from its parent nodes. The communication cost among two nodes assigned to the same processor is assumed to be zero. If node n_i is scheduled to processor P, $ST(n_i, P)$ and $FT(n_i, P)$ denote the start time and finish time of n_i on processor P, respectively. After all nodes have been scheduled, the overall execution time or *schedule length*, is defined as $max_i\{FT(n_i, P)\}$ across all processors. The objective of a DAG scheduling algorithm is to assign and schedule the nodes to processors such that the schedule length is minimized without violating the precedence constraints.

The CP provides a lower bound on the schedule length of the DAG. Thus, nodes on the CP must be given higher priorities. To identify a CP node (CPN), two attributes, the *t-level* and *b-level* [18], can be used. The *t-level* of a node n_i is the length of the longest path from an entry node to n_i excluding $w(n_i)$. The *b-level* of a node n_i is the length of the longest path from n_i to an exit node. The CPNs of a DAG are the nodes with the largest sum of *t-level* and *b-level*. Since the computations of *t-level* and *b-level* take $O(e)$ time, the CPNs can be identified also in $O(e)$ time.

For scheduling, the *t-level* and *b-level* of a node can be used in a variety of ways. Some algorithms use *t-level* as a node priority, while some algorithms use *b-level*. Some algorithms use a variant of *b-level*, called the *static b-level* or simply *static level* (SL), in which only the computation costs and not the communication costs are taken into account. There are also two additional attributes called the *as-soon-as-possible* (ASAP) start time and *as-late-as-possible* (ALAP) start time [17]. The ASAP value is just the *t-level*. The ALAP start time is the largest possible start time of a node bounded by the CP length. Thus, the ALAP value is equal to the CP length minus the *b-level* of the node. Given the ASAP and ALAP values, the CPNs, whose ASAP and ALAP values are equal, can be easily identified. Notice that the CP can change during the scheduling process because some of the edges are zeroed if the connected nodes are scheduled to the same processors. To overcome this situation, some algorithms re-compute the values of *t-level* and *b-level*. This increases the complexity since, if the scheduling algorithm takes v steps to schedule all the nodes, the re-computation of priorities can take an overall time of $O(ev)$, which can be $O(v^3)$ in the worst case.

In Figure 1, an example DAG and the values of *SL*, *t-level* (*ASAP*), *b-level*, and *ALAP* of its nodes are provided. The CPNs of the DAG are shown in dark and the edges on the CP are shown with thick arrows.

(a)



(b)

node	SL	t-level	b-level	ALAP
$*n_1$	12	0	37	0
n_2	8	6	23	14
n_3	8	3	23	14
n_4	9	3	20	17
n_5	10	3	30	7
n_6	5	10	15	22
$*n_7$	5	22	15	22
n_8	5	18	15	22
$*n_9$	1	36	1	36

Figure 1: (a) A task graph; (b) The static levels (SLs), *t-levels* (ASAP times), *b-levels*, and ALAP times of the nodes (CPNs are marked by an asterisk).

3 Related Work

A large number of DAG scheduling algorithms have been reported in the literature. In this section, we describe four related scheduling algorithms: the Mobility Directed (MD) algorithm [17], the Earliest Task First (ETF) algorithm [8], the Dynamic Level Scheduling (DLS) algorithm [14], and the Dominant Sequence Clustering (DSC) algorithm [18].

3.1 The MD Algorithm

The MD (Mobility Directed) algorithm selects a node n_i for scheduling based on an attribute called the *relative mobility*, which is defined as dividing the difference between the ALAP and ASAP by $w(n_i)$. At each step, the MD algorithm selects the node with the smallest relative mobility for scheduling. In finding a processor for the node, the MD algorithm schedules the node to the first processor that can accommodate the node. The time complexity of the MD algorithm is $O(v^3)$.

3.2 The ETF Algorithm

The ETF (Earliest Time First) algorithm computes, at each step, the earliest start times for all ready nodes and then selects the one with the smallest start time. When two nodes have the same value of their earliest start times, the ETF algorithm breaks the tie by scheduling the one with the higher static level. The time complexity of the ETF algorithm is $O(pv^2)$.

3.3 The DLS Algorithm

The DLS (Dynamic Level Scheduling) algorithm uses an attribute called *dynamic level* (DL) which is the difference between the *static b-level* of a node and its earliest start time (i.e., *t-level*) on a processor. The node-processor pair that gives the largest value of DL is selected for scheduling. The time complexity of the DLS algorithm is $O(pev)$. The high complexity of the algorithm is due to the pair-wise matching of the nodes to processors.

3.4 The DSC Algorithm

The DSC (Dominant Sequence Clustering) algorithm considers the *Dominant Sequence* (DS) of a graph. The DS is simply the CP of the partially scheduled DAG. The algorithm is briefly described below. The DSC algorithm tracks the CP of the partially scheduled DAG at each step by using the composite attribute (*b-level* + *t-level*) as the priority of a node. The DSC algorithm does not select the node with the highest priority for scheduling unless the node is ready. This is done in order to lower the time complexity of the algorithm because the *t-level* of a node can be computed incrementally and the *b-level* does not change until the node is scheduled. The time complexity of the DSC algorithm is $O((e+v)\log v)$.

4 The Proposed Algorithm

The local search-based algorithm called *Fast Assignment using Search Technique (FAST)*, has two phases:

1) generate an initial schedule using the classical list scheduling method in $O(e)$ time using the given number of available processors;
2) refine the schedule using local neighborhood search in $O(e)$ time.

In the following, we first describe how to determine node priorities and generate an initial schedule in $O(e)$ time. Then we describe how the local search technique can be applied to improve upon the initial schedule.

4.1 Determining Node Priorities

To generate an initial schedule, we employ the traditional list scheduling approach —construct a list and schedule the nodes on the list one by one to the processors. The list is constructed by ordering the nodes according to the node priorities. The list is static so that the order of nodes on the list will not change during the scheduling process. The reason is that as the objective of our algorithm is to produce a good schedule in $O(e)$ time, we do not re-compute the node priorities after each scheduling step while generating the initial schedule. Nonetheless, if the schedule length of the initial schedule is optimized, the subsequent local search process can start at a better solution point and thereby, can generate a better final schedule.

In order to construct a static list of nodes ordered in accurate priorities, we employ a novel method for the arrangement of nodes. As discussed earlier, the CP nodes (CPNs) are the more important nodes because their scheduling order can have a direct impact on the schedule length. Thus, we assign the highest priority to CPNs. However, a CPN cannot be scheduled before all of its parent nodes are scheduled. To tackle this problem, we partition the nodes of the DAG as follows: In a connected graph, an In-Branch Node (IBN) is a node, which is not a CPN, and from which there is a path reaching a Critical Path Node (CPN). An Out-Branch Node (OBN) is a node, which is neither a CPN nor an IBN. For the DAG shown earlier in Figure 1, the CPNs are shown with dark color, the IBNs are uncolored. There is no OBN in this DAG.

When a CPN is considered for scheduling, all the IBNs reaching it as well as its parent CPN must have been scheduled. Thus, the IBNs reaching a CPN must occupy higher positions on the list than the CPN itself. The OBNs are relatively less important nodes so that they can occupy lower positions on the list. To further differentiate the importance among the IBNs, we order the IBNs in a decreasing order of *b-levels*. The OBNs are also ordered in increasing *b-levels*. Given these observations, we can formalize our method of constructing the scheduling list, which is called the CPN-Dominate List, as follow.

CPN-Dominate List:

(1) Make the entry CPN to be the first node in the list. Set *Post* to 2. Let n_x be the next CPN.
Repeat
(2) **If** n_x has all its parent nodes in the list **then**
(3) Put n_x at *Post* in the list and increment *Post*.
(4) **else**
(5) Suppose n_y is the parent node of n_x which is not in the list and has the largest *b-level*. Ties are broken by

choosing the parent node with a smaller *t-level*. If n_y has all its parent nodes in the list, put n_y at *Post* in the list and increment *Post*. Otherwise, recursively include all the ancestor nodes of n_y in the list so that the nodes with larger *b-levels* are considered first.

(6) Repeat step (5) until all the parent nodes of n_x are in the list. Put n_x in the list at *Post*.

(7) **endif**

(8) Make n_x to be the next CPN.

Until all CPNs are in the list.

(9) Append all the OBNs to the list in a decreasing order of *b-level*.

It is obvious that the above procedure for constructing the CPN-Dominate List takes only $O(e)$ time since each edge is visited once.

4.2 The Initial Schedule

Using the CPN-Dominate List, we can schedule the nodes on the list one after another to the processors. Again, in order not to incur high complexity, we do not search for the earliest slot on a processor. Instead, we simply schedule a node to the ready-time of a processor. Initially, the ready-time of all available processors are zero. After a node is scheduled to a processor, the ready-time of that processor is updated. By doing so, a node is scheduled to a processor that allows the earliest start time, which is determined by checking the processor's ready-time with the node's data arrival time (DAT). The DAT of a node can be computed by taking the maximum value among the message arrival times across the its parent nodes. If the parent is scheduled to the same processor as the node, the message arrival time is simply the parent's finish time; otherwise it is equal to the parent's finish time (on a remote processor) plus the communication cost of the edge. Not all processors need to be checked in this process. Instead, we can examine the processors accommodating the parent nodes together with a new processor (if any). The procedure for generating the initial schedule can be formalized below.

InitialSchedule()

(1) Construct the CPN-Dominate List.

Repeat

(2) Remove the first node n_i from the list.

(3) Schedule n_i to the processor, among the processors accommodating the parent nodes of n_i together with a new processor (if any), that allows the earliest start time by checking n_i's DAT with the ready-times of the processors.

Until the list is empty.

The time complexity of *InitialSchedule*() is determined as follows. The first step takes $O(e)$ time. In the repeat loop, the dominant step is the procedure to determine the data arrival time of a node. The cumulative time complexity of this step throughout the execution of the repeat loop is also $O(e)$. Thus, the time complexity of *InitialSchedule*() is again $O(e)$.

To illustrate how *InitialSchedule*() works, we can consider the DAG shown in Figure 1(a) again. The CPN-Dominate List is $\{n_1, n_3, n_2, n_7, n_6, n_5, n_4, n_8, n_9\}$. Note that n_8 is considered after n_6 because n_6 has a smaller *t-level*. Using the CPN-Dominate List, we can consider the initial schedule produced by *InitialSchedule*(). For comparison, the schedules generated by the MD, ETF, and DLS algorithms are shown in Figure 2. The schedule

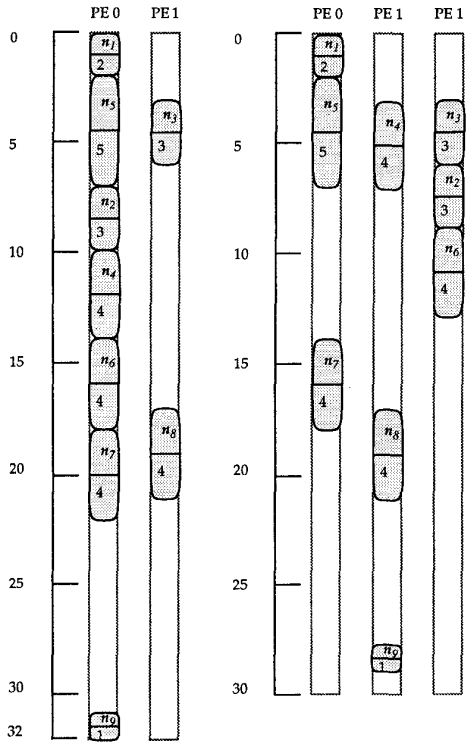


Figure 2: Schedule generated by (a) the MD algorithm (schedule length = 32); (b) the ETF and DLS algorithm (schedule length = 29).

generated by the DSC algorithm is shown in Figure 3. The schedule generated by *InitialSchedule*() is shown in Figure 4(a). Note that the ETF and DLS algorithms generate the same schedule. As can be seen, the MD algorithm produces the worst schedule. This is due to the fact that the MD algorithm does not schedule a node to the earliest possible time slots even though it re-computes priorities at each step. The MD algorithm schedules the node n_4 late because it schedules the node n_5 too early so that it blocks n_4. The schedule generated by the ETF and DLS algorithm is better but is still not satisfactory. The problem is that they schedule the node n_5 early because it has a higher value of static level (SL). But n_5 is in fact not as important as n_2 which should have occupied an earlier slot. As a result, n_7 also starts late and the schedule length cannot be reduced. The DSC algorithm generates a slightly better schedule. The problem with the schedule is that the node n_8 is scheduled to a late time slot because its parent n_4 is not scheduled to the same processor due to the minimization of n_4's start time. The schedule length of generated by *InitialSchedule*() is the shortest even though it does not employ sophisticated (and hence time

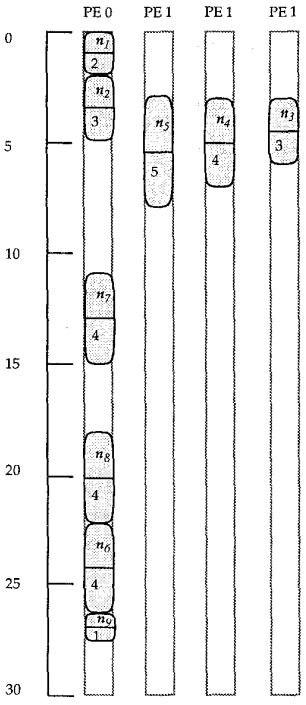


Figure 3: Schedule generated by the DSC algorithm (schedule length = 27).

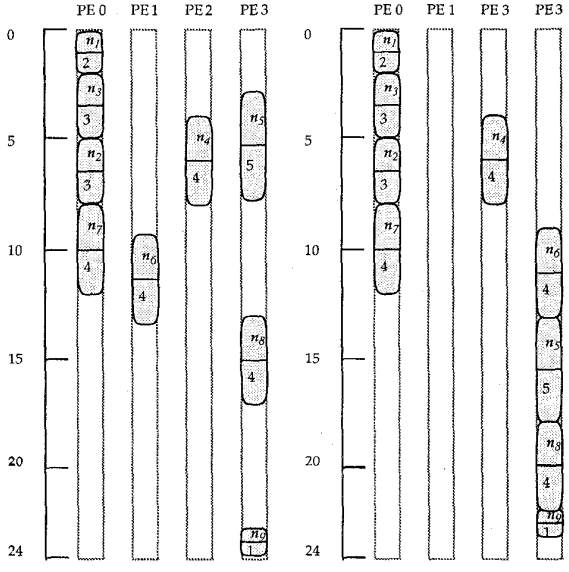


Figure 4: (a) Schedules generated by the *InitialSchedule*() (schedule length = 24); (b) The final schedule after the local search process with node n_6 is transferred to PE 3 (schedule length = 23).

consuming) strategies to compute priorities and to select the best time slots for nodes. However, the initial schedule can be further refined to obtain a better solution, as will be discussed below.

4.3 Local Search

Local search is an old technique for combinatorial optimization. It has been applied to solve NP-hard optimization problems [6]. 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 random selection/ assignment, and pre- and partial selection/assignment, have been developed [5], [15], [16].

Sosic and Gu developed four efficient local search algorithms for a benchmark problem for constrained optimization problems, i.e., the *n*-queen problem. The *Queen_Search1* (QS1) algorithm is a probabilistic local search algorithm that runs in approximately O(*nlogn*) time. The QS2 and QS3 are near linear local search algorithms with random selection/assignment and partial selection/assignment [15]. The QS4 algorithms is a linear time local search algorithm with pre- and partial random selection/assignment [16].

To apply the local search technique to the DAG scheduling problem, we have to define a neighborhood of the initial solution point (i.e., the initial schedule). A simple neighborhood point of a schedule in the solution space is another schedule which is obtained by transferring a node from a processor to another processor.

In the DAG scheduling problem, one method to improve the schedule length is to transfer a *blocking* node from a processor to another processor. The notion of *blocking* is simple: a node is called blocking if removing it from its original processor can make the succeeding nodes to start earlier. In particular, we are interested in transferring the nodes that block the CPNs because the CPNs are the more important nodes. However, high complexity will result if we attempt to locate the actual blocking nodes on all the processors. Thus, in our approach, we only generate a list of *potential* blocking nodes which are the nodes that may block the CPNs. Again, to maintain complexity low, the blocking node list is static and is constructed before the search process starts. A natural choice of blocking node list is the set of IBNs and OBNs because these nodes have the potential to block the CPNs in the processors. In the schedule refinement phase, the blocking node list defines the neighborhood that the local search process will explore.

To illustrate how the search process works, we can consider the initial schedule of the example DAG discussed previously. The blocking node list of the DAG is $\{n_2, n_3, n_4, n_5, n_6, n_8\}$. We can notice that the node n_6 blocks the CPN n_9. In the local search process, it is highly probable that n_6 is selected for transferring. Suppose it is transferred from PE 1 to PE 3. The resulting schedule is shown in Figure 4(b). We can notice that even though the start times of the n_5 and n_8 are increased the final schedule length is shorten.

4.4 The Search-Based Scheduling

Based on the search process mechanisms and the

neighborhood for searching, we can formalize the proposed FAST (Fast Assignment using Search Technique) algorithm below.

The FAST Algorithm:

(1) *InitialSchedule*()
(2) Construct the blocking node list which contains all the IBNs and OBNs.
(3) searchstep = 0;
(4) **do** { /* search */
(5) Pick a node n_i randomly from the blocking node list.
(6) Pick a processor P randomly.
(7) Transfer n_i to P.
(8) If schedule length does not improve, transfer n_i back to its original processor.
(9) } **while** (searchstep++ < MAXSTEP);

The time complexity of the FAST algorithm is determined as follow. As discussed earlier, the procedure *InitialSchedule*() takes $O(e)$ time. The blocking node list can be constructed in $O(v)$ time as the IBNs and OBNs are already identified in the procedure *InitialSchedule*(). In the main loop, the node transferring step takes $O(e)$ time since we have to re-visit all the edges once after transferred the node to a processor in the worst case. For our experiments, the constant MAXSTEP can be as small as 100 even for huge DAGs with tens of thousands of nodes. Indeed, for the results to be presented in the next section, the value of MAXSTEP is fixed at 64. Thus, the overall complexity of the FAST algorithm is $O(e)$.

5 Performance

In this section, we present the performance results of the FAST algorithm and also compare them with those of DSC, MD, ETF, and DLS algorithms. We performed experiments using real workload generated from serial applications by a prototype parallelization and scheduling tool called CASCH (Computer Aided SCHeduling) [2]. The CASCH tool generates a task graphs from a sequential program, uses a scheduling algorithm to perform scheduling, and then generates the parallel code in a scheduled form for the Intel Paragon. The timings for the nodes and edges on the DAG are assigned through a timing database that was obtained through benchmarking. CASCH also provides a graphical interface to interactively run and test various algorithms including the ones discussed in this paper. Instead of just measuring the schedule length through a Gantt chart, we measure the running time of the scheduled code on the Paragon. Various scheduling algorithms, therefore, can be more accurately tested and compared through CASCH using real applications on an actual machine.

We also performed additional experiments with randomly generated large DAGs consisting of thousands of nodes. Notice that the MD and DSC algorithms assume unlimited number of processors while others do not. Thus, to investigate the performance of the algorithms in a fair manner, we give more than enough processors to all the algorithms.

5.1 Real Workload

In our first experiment, we tested the FAST algorithm with the DAGs generated from three real applications: Gaussian elimination, Laplace equation solver and Fast Fourier Transform (FFT) [2], [10], [17]. The Gaussian elimination and Laplace equation solver applications operate on matrices. Thus, the number of nodes in the DAGs generated from these applications are related to the matrix dimension N and is about $O(N^2)$. On the other hand, the FFT application accepts the number of points as input. We examine the performance in three aspects: application execution time, number of processors used and the scheduling algorithm running time.

The results for the Gaussian elimination are shown in Figure 5. In the table shown in Figure 5(a), we normalized the application execution times obtained through all the algorithms with respect to those obtained through the FAST algorithm. We can notice that the programs scheduled by the FAST algorithm are 3% to 15% faster than the other algorithms. Note that the results of the DSC algorithm for matrix dimensions 16 and 32 were not available because the DSC used more than the available Paragon processors in scheduling the parallel program. In general, the DSC algorithm uses $O(v)$ processors. Concerning the number of processors used, the FAST, ETF and DLS algorithms used about the same amount of processors. The number of processors used by all the algorithms is shown in Figure 5(b). The scheduling times of all the algorithms are shown in Figure 5(c). As can be seen, the DSC algorithm is the fastest algorithm and the proposed FAST algorithm is very close to it. The ETF and DLS algorithms running times are relatively large. The MD algorithm is much slower than the other algorithms. Indeed, the MD algorithm is about $O(v)$ times slower than the other algorithms.

The results for the Laplace equation solver are shown in Figure 6. The percentage improvements of the FAST algorithm over the other algorithms are up to 25%. The number of processors used in tis case are about the same for the FAST, MD, ETF and DLS algorithms. The DSC algorithm again uses more processors than the other algorithms. For the scheduling times, the FAST algorithm is the fastest among all the algorithms. The MD algorithm is again $O(v)$ times slower than the other algorithms. The results for the FFT are shown in Figure 7. The FAST algorithm is again better than all the other four algorithms in terms of execution time and running times.

5.2 Random DAGs

To test the scalability and robustness of the FAST algorithm, we performed experiments with very large random DAGs. These DAGs were synthetically generated in the following manner. Given the size of the DAG (i.e., v), we first randomly generated the height of the DAG from a uniform distribution with mean roughly equal to $\sqrt{v}$. For each level, we generated a random number of nodes which was also selected from a uniform distribution with mean roughly equal to $\sqrt{v}$. Then, we connected the

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	1.00	1.00	1.00	1.00
DSC	1.05	1.08	N.A.	N.A.
MD	1.00	1.03	1.08	1.10
ETF	1.00	1.07	1.10	1.15
DLS	1.00	1.08	1.10	1.14

(a) Normalized execution times of Gaussian elimination on the Intel Paragon.

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	4	8	16	32
DSC	5	22	95	128
MD	2	3	4	7
ETF	3	7	16	32
DLS	3	7	16	32

(b) Number of Processors used for the Gaussian elimination.

	Matrix Dimension (Number of Tasks)			
Algorithm	4 (20)	8 (54)	16 (170)	32 (594)
FAST	0.06	0.09	0.15	0.52
DSC	0.04	0.06	0.09	0.21
MD	6.33	6.85	39.54	266.89
ETF	0.02	0.06	0.24	2.41
DLS	0.08	0.09	0.42	4.00

(c) Scheduling times (sec) on a SPARC Station 2 for the Gaussian elimination.

Figure 5: Normalized execution times, number of processors used, and scheduling algorithm running times for the Gaussian elimination for all the scheduling algorithms.

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	1.00	1.00	1.00	1.00
DSC	1.00	1.09	1.13	1.21
MD	1.00	1.12	1.15	1.25
ETF	1.00	1.11	1.14	1.24
DLS	1.00	1.10	1.13	1.23

(a) Normalized execution times of Laplace equation solver on the Intel Paragon.

	Matrix Dimension			
Algorithm	4	8	16	32
FAST	1	4	7	14
DSC	1	13	37	64
MD	1	5	8	13
ETF	1	5	8	16
DLS	1	5	8	15

(b) Number of Processors used for the Laplace equation solver.

	Matrix Dimension (Number of Tasks)			
Algorithm	4 (18)	8 (66)	16 (258)	32 (1026)
FAST	0.05	0.09	0.35	1.28
DSC	0.07	0.11	0.40	4.29
MD	6.23	7.64	111.46	768.90
ETF	0.04	0.05	0.28	3.06
DLS	0.06	0.11	0.55	5.33

(c) Scheduling times (sec) on a SPARC Station 2 for the Laplace equation solver.

Figure 6: Normalized execution times, number of processors used, and scheduling algorithm running times for the Laplace equation solver for all the scheduling algorithms.

nodes from the higher level to lower level randomly. The edge weights were also randomly generated. As the real workload discussed above consists of mainly sparse DAGs, the random DAGs generated were deliberately made denser. The sizes of the random DAGs were varied from 2000 to 5000 with an increment of 1000. For these graphs, we simply measured the schedule length produced by an algorithm. The results for the random DAGs are shown in Figure 8. Note that the MD algorithm was excluded from the comparison because it took more than 8 hours to produce a schedule for a 2000-node DAG. The FAST algorithm performed slightly worse the ETF and DLS algorithm but still outperformed the DSC algorithm by 7% to 12% margin. From Figure 8(b), we note that the DSC algorithm used an unrealistic number of processors. Concerning the scheduling times, we can immediately note from Figure 8(c) that the ETF and DLS algorithms were considerably slower than the FAST and DSC algorithms.

The proposed FAST algorithm outperforms the DSC algorithm both in terms of solution quality and complexity. It also outperforms the ETF, MD, and DLS algorithms most of the time. Furthermore, the ETF, MD, and DLS algorithms need a huge amount of scheduling time to produce a schedule for a realistically large DAG (note that the DAG for the Gaussian elimination application with matrix dimension 128 has 8192 nodes). Thus, the FAST algorithm is more preferable for practical use in parallel program scheduling.

6 Conclusions

The algorithm complexity and quality of the solution can become conflicting goals in the design of efficient scheduling algorithms. Our study indicates that not only does the solution qualities of existing algorithms differ considerably but their running times can vary by even huge margins. In this paper, we have presented a new scheduling algorithm which takes linear time to generate solutions with high quality. We have compared the algorithm with a number of well-known reportedly efficient scheduling algorithms. The results obtained demonstrate that the proposed algorithm is better than the other algorithms in terms of both solution quality and complexity. The major strength of the algorithm is the construction of the CPN-Dominate list in which the priorities of nodes are accurately captured. The list is used to generate an initial schedule upon which the local search process is invoked to refine and produce a final schedule. Nonetheless, there can be cases that the initial schedule is not good enough so that the local search process may get stuck in a poor local minimum point in the solution space.

References

[1] I. Ahmad, Y.K. Kwok and M.Y. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," *submitted for publication.*

Algorithm	Numer of Points: 16	64	128	512
FAST	1.00	1.00	1.00	1.00
DSC	1.03	1.08	1.10	1.15
MD	1.04	1.09	1.11	1.17
ETF	1.02	1.08	1.10	1.15
DLS	1.03	1.07	1.09	1.14

(a) Normalized execution times of FFT on the Intel Paragon.

Algorithm	Numer of Points: 16	64	128	512
FAST	45	12	9	23
DSC	5	12	13	25
MD	5	10	6	21
ETF	3	10	11	11
DLS	7	10	11	11

(b) Number of Processors used for the FFT.

Algorithm	Number of Points (Number of Tasks): 16 (14)	64 (34)	128 (82)	512 (194)
FAST	0.06	0.10	0.12	0.19
DSC	0.07	0.08	0.07	0.10
MD	6.38	9.09	9.87	75.17
ETF	0.05	0.08	0.09	0.16
DLS	0.05	0.18	0.20	0.67

(c) Scheduling times (sec) on a SPARC Station 2 for FFT.

Figure 7: Normalized execution times, number of processors used, and scheduling algorithm running times for FFT for all the scheduling algorithms.

Algorithm	Number of Nodes: 2000	3000	4000	5000
FAST	1.00	1.00	1.00	1.00
DSC	1.08	1.07	1.11	1.12
ETF	0.98	0.97	0.97	0.97
DLS	0.98	0.98	0.97	0.98

(a) Normalized schedule lengths for random DAGs.

Algorithm	Number of Nodes: 2000	3000	4000	5000
FAST	141	195	174	219
DSC	1577	2253	3036	3758
ETF	123	139	162	198
DLS	131	197	166	212

(b) Number of Processors used for random DAGs.

Algorithm	Number of Nodes (Number of Edges): 2000 (81049)	3000 (96847)	4000 (140821)	5000 (179902)
FAST	42.01	46.05	72.36	97.79
DSC	40.37	48.36	86.26	121.98
ETF	748.71	699.83	966.80	2306.42
DLS	814.31	880.36	1182.06	2506.68

(c) Scheduling times (sec) on a SPARC Station 2 for random DAGs.

Figure 8: Normalized schedule lengths, number of processors used and scheduling times for the random DAGs for all the scheduling algorithms.

[2] I. Ahmad, Y.K. Kwok, M.Y. Wu and W. Shu , "CASCH: A Software Tool for Automatic Parallelization and Scheduling of Program on Multiprocessors," *submitted for publication.*

[3] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[4] 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*, no. 5, pp. 287-326, 1979.

[5] J.Gu., "Parallel algorithms and architectures for very fast search," (PhD Thesis) Technical Report UUCS-TR-88-005, Dept. of Computer Science, Univ. of Utah, 1988.

[6] J.Gu, "Local Search for the Satisfiability (SAT) Problem", *IEEE Transactions on Systems and Cybernetics*, vol. 23, no. 3, Jul./Aug. 1993, pp. 1108-1129.

[7] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 19, no. 6, pp. 841-848, Nov. 1961.

[8] J.J.Hwang, Y.C. Chow, F.D. Anger and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, vol. 18, no. 2, pp. 244-257, Apr. 1989.

[9] H.Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, vol. C-33, pp. 1023-1029, Nov. 1984.

[10] R.E.Lord, J.S. Kowalik and S.P. Kumar, "Solving Linear Algebraic Equations on an MIMD Computer," *Journal of the ACM*, 30(1), pp. 103-117, Jan. 1983.

[11] S.S. Pande, D.P. Agrawal and J. Mauney, "A Threshold Scheduling Strategy for Sisal on Distributed Memory Machines," *Journal of Parallel and Distributed Computing*, 21, pp. 223-236, 1994.

[12] C.H.Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[13] B.Shirazi, M. Wang and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Scheduling," *Journal of Parallel and Distributed Computing*, no. 10, pp. 222-232, 1990.

[14] G.C.Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 75-87, Feb. 1993.

[15] R. Sosic 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.

[16] R.Sosic and J.Gu, "Efficient Local Search with Conflict Minimizations," *IEEE Transactions on Data and Knowledge Engineering*, Vol. 6, No. 4, pp. 661-668, Aug. 1994.

[17] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, Jul. 1990.

[18] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951-967, Sep. 1994.