Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Techniques

ISHFAQ AHMAD[1] AND YU-KWONG KWOK[2]

[1]Department of Computer Science
The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
[2]Parallel Processing Laboratory, School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285, USA

Abstract†

Obtaining an optimal schedule for a set of precedence-constrained tasks with arbitrary costs is a well-known NP-complete problem. However, optimal solutions are desired in many situations. In this paper we propose search-based algorithms for determining optimal schedules for moderately large problem sizes. The first algorithm which is based on the A* search technique uses a computationally efficient cost function for guiding the search with reduced complexity. We propose a number of state-pruning techniques to reduce the size of the search space. For further lowering the complexity, we parallelize the search. The parallel version is based on reduced interprocessor communication and is guided by static and dynamic load-balancing schemes to evenly distribute the search states to the processors. We also propose an approximate algorithm that guarantees a bounded deviation from the optimal solution but takes considerably shorter time. Based on an extensive experimental evaluation of the algorithms, we conclude that the parallel algorithm with pruning techniques is an efficient scheme for generating optimal solutions for medium to moderately large problems while the approximate algorithm is a useful alternative if slightly degraded solutions are acceptable.

Keywords: Optimal Scheduling, Task Graphs, Parallel Processing, Parallel A*, State-Space Search Techniques, Multiprocessors.

1 Introduction

Scheduling a parallel program to the processors is crucial for effectively harnessing the computing power of a multiprocessor system. A scheduling algorithm aims to minimize the overall execution time of the program by properly allocating and arranging the execution order of the tasks on the processors such that the precedence constraints among the tasks are preserved. If the characteristics of a parallel program, including task processing times, data dependency and synchronizations, are known *a priori*, the program can be modeled by a node- and edge-weighted *directed acyclic graph* (DAG). The problem of static scheduling of a DAG is in general NP-complete. Hitherto, the problem can be solved in a polynomial-time for only a few highly simplified cases [1], [5], [7]. If the simplifying assumptions of these cases are relaxed, the problem becomes NP-hard in the strong sense. Thus, it is unlikely that the problem in its general form can be solved in a polynomial-time, unless $P = NP$.

In view of the intractability of the scheduling problem, many polynomial-time heuristics are reported to tackle the problem under more pragmatic situations [12], [18]. While these heuristics are shown to be effective in experimental studies, they usually cannot generate optimal solutions, and there is no guarantee in their performance in general. In addition, in the absence of optimal solutions as a reference, the average performance deviation of these heuristics is unknown.

On the other hand, there are many advantages of having optimal schedules: Optimal schedules may be required for critical applications in which performance is the primary objective. Also, optimal solutions for a set of benchmarks problems can serve as a reference to assess the performance of various scheduling heuristics. Moreover, once an optimal schedule for a given problem is determined, it can be re-used for efficient execution of the problem. For obtaining optimal schedules, techniques such as integer programming, state-space search, and branch-and-bound methods can be used [6], [8], [10], [11], [15], [17]. However, the solution space of the problem can be very large (for example, to schedule a DAG with v nodes to p processors, more than p^v possible solutions exist). Furthermore, the solution space in general does not maintain a regular structure to allow state pruning. Thus, a need exists to explore search-based algorithms with efficient state pruning techniques to produce optimal solutions in a short turnaround time.

Kasahara and Narita [9] pioneered the research in using branch-and-bound algorithms for multiprocessor scheduling. However, inter-task communication delays were not considered in the design of their algorithm and such assumption renders the algorithm not useful in more realistic models. Recently, a few other branch-and-bound algorithms for solving the scheduling problem have been reported in the literature [2], [3], [4]. These algorithms also possess one drawback or the other, making them impracticable except for very special cases. For example, some algorithms can handle only restricted DAGs, such as those with unit computation cost and no communication [2], [4]. Some algorithms use more complicated cost functions but their evaluation of a search state computationally is expensive [3]. A huge memory requirement to store the search states is also another common problem.

Our objective in this paper is to propose optimal scheduling schemes that are fast and can be used for problems with practical sizes and without simplifying assumptions. We propose an algorithm based on the A* search technique with an effective yet computationally efficient cost function. The proposed A* algorithm is also equipped with several highly effective state-space pruning techniques, which can dramatically reduce the required scheduling time. The effectiveness of these pruning techniques are analyzed experimentally. We also propose an efficient parallelization methodology for our proposed algorithm. Since a parallel program is executed on multiple processors, it is natural to utilize the same processors to speedup the scheduling of the program. Indeed, using multiple processors to search for an optimal solution not only shortens the computation time but also reduces the memory requirement and allows for a larger problem size. Surprisingly, very little amount of work has been done in parallelizing scheduling algorithms [13]. We also propose a variation of our algorithm which does not provide an optimal solution but guarantees a bounded degradation of the

†. This research was supported by the Hong Kong Research Grants Council under contract number HKUST 734/96E.

solution quality and is much faster. This algorithm can be useful if efficiency, but not an optimal solution, is the primary goal.

The remainder of the paper is organized as follows. Section 2 provides the problem statement. Section 2 contains some of the previous work on generating optimal solutions for scheduling. Section 3 presents the proposed serial, parallel, and approximate algorithms. Section 4 contains the details of our experimental study as well as the experimental results. The last section concludes the paper by providing final remarks.

2 Problem Statement

In static scheduling, a parallel program is modeled by 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 in turn is a set of instructions that must be executed sequentially without preemption in the same processor. The weight associated with a node, which represents the amount of time needed for a processor to execute the task, is called the *computation cost* of a node n_i and is denoted by $w(n_i)$. An edge in the DAG, denoted by (n_i, n_j), corresponds to the communication messages and precedence constraints among the nodes. The weight associated with an edge, which represents the amount of time needed to communicate the data, is called the *communication cost* of the edge and is denoted by $c(n_i, n_j)$. The *communication-to-computation-ratio (CCR)* of a DAG is defined as its average communication cost divided by its average computation cost on a given system.

The source node of an edge directed to a node is called a *parent* of that node. Likewise, the destination node directed from a node is called a *child* of that node. A node with no parent is called an *entry* node and a node with no child is called an *exit* node. The precedence constraints of a DAG dictate that a node 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, $ST(n_i)$ and $FT(n_i)$ denote the start time and finish time of n_i, respectively. After all nodes have been scheduled, the *schedule length* is defined as $max_i\{FT(n_i)\}$ across all nodes. The objective of scheduling is to assign the nodes to the processors and arrange the execution order of the nodes such that the schedule length is minimized and the precedence constraints are preserved.

An example DAG, shown in Figure 1(a), will be used in our discussion. We assume that the processors or *processing elements* (PEs) in the target system do not share memory so that communication solely relies on message-passing. The processors may be heterogeneous or homogeneous. Heterogeneity of processors means the processors have different speeds or processing capabilities. However, we assume every module of a parallel program can be executed on any processor though the computation time needed on different processors may be different. The processors are connected by an interconnection network based on a certain topology. The topology may be fully-connected or of a particular structure such as a hypercube or mesh. Although processors may be heterogeneous, we assume the communication links are homogeneous. That is, a message is transmitted with the same speed on all links. An example processor graph is shown in Figure 1(b).

The arbitrary DAG scheduling problem is an NP-complete problem [7]. However, a few attempts for optimal scheduling of DAGs under more relaxed assumptions have been reported. Chou and Chung [4] proposed an algorithm for optimal unit-computation DAG scheduling on multiprocessors. However, communication among tasks is ignored. Chang and Jiang [2] proposed several state-space search approaches for scheduling DAGs with arbitrary precedence relations. Although the algorithm assumes arbitrary computation costs, communication among tasks is also ignored. Chen and Yu [3] proposed a branch-and-bound algorithm for optimal scheduling of arbitrary structured DAG with arbitrary node- and edge-weights. Their algorithm uses a complicated underestimate cost function to prune the solution space. For generating a new state, the function is computed by first determining all of the complete execution paths extended from the node to be scheduled. To take into consideration inter-processor communication, an exhaustive matching of the execution paths and the processor graph is then performed to determine the minimum communication required. Finally, the finish time of the last exit node is taken as the value of the underestimate cost function. Since the problem considered in that study is the closest to our problem, we will compare our approach with Chen and Yu's algorithm.

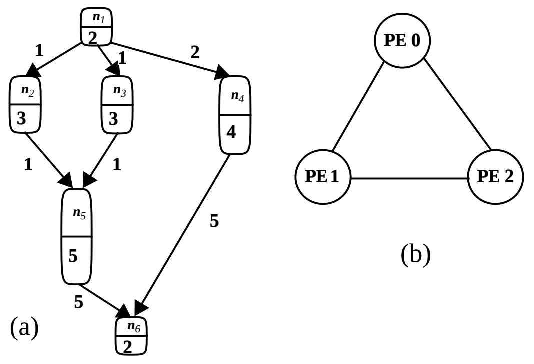


Figure 1: (a) An example DAG; (b) A 3-processor ring target system.

3 The Proposed Algorithms

In this section, we first formulate the scheduling problem in terms of a state-space search, and then define a cost function used for guiding the search. We also describe a number of effective search space pruning techniques to enhance the efficiency of our algorithm. Subsequently, we present the proposed parallel optimal scheduling algorithm. Finally an approximate algorithm for obtaining solutions with a bounded solution quality is presented.

3.1 State-Space Search Formulation

Formulation of a problem in a state-space search framework requires four basic components: *state representation*, *initial state*, *expansion operator*, and *goal state*.

In the context of the scheduling problem, we define these components as follows:

- *State Representation.* State representation describes how a search state represents a partial solution. A state in the search space for the scheduling problem is a partial schedule in which a sub-graph of the DAG is assigned to a certain number of processors.
- *Initial State.* The initial state is the starting state. In the case of scheduling, it is an empty partial schedule.
- *Expansion Operator.* An expansion operator dictates a scheme for constructing larger partial solutions from an existing partial solution. For expanding a search-state, the first node from the list of *ready nodes (*the nodes whose predecessors have been scheduled*)* is selected. The selected node is considered for assignment to each of the available processors. Each possible assignment generates one new state. The next node from the list is then selected,

and state expansion continues in a similar fashion. The state expansion stops when all of the ready nodes have been considered for assignment.

- *Goal State.* A goal state is a solution state and hence the terminating point of a search. In the case of the scheduling problem, it is a complete schedule.

The above components only outline the search scheme for obtaining a solution. To obtain an optimal solution we need an "intelligent" algorithm to navigate the search space using effective exploration techniques. We use the A* algorithm from the area of artificial intelligence [16] to find an optimal solution for the scheduling problem. In the A* algorithm, a cost function $f(s)$ is attached to each state, *s,* in the search-space, and the algorithm always chooses the state with the minimum value of $f(s)$ for expansion. The cost function $f(s)$ is a lower-bound estimate of the exact minimum cost of the search path from the initial state through state *s* to the goal state, denoted by $f^*(s)$. The function $f(s)$ is usually defined by using problem-dependent heuristic information, and is considered to be *admissible* (or *consistent*) if it satisfies $f(s) \le f^*(s)$ for any state *s*. With an admissible function, the A* algorithm guarantees to find an optimal solution.

The function $f(s)$ can be decomposed into two components $g(s)$ and $h(s)$ such that $f(s) = g(s) + h(s)$, where $g(s)$ is the cost from the initial state to state *s*, and $h(s)$ (which is also called the *heuristic function*) is the estimated cost from state *s* to a goal state. Since $g(s)$ represents the actual cost of reaching a state, it is $h(s)$ where the problem-dependent heuristic information is captured. Indeed, $h(s)$ is only an estimate of the actual cost from state *s* to a goal state, denoted by $h^*(s)$. An $h(s)$ is called admissible if it satisfies $h(s) \le h^*(s)$ which in turn implies $f(s) \le f^*(s)$. A properly defined and tightly bounded $h(s)$ (hence $f(s)$) is, therefore, crucial to enhance the search efficiency. One trivial definition of $h(s)$ is to make it zero for any *s*; the search, however, then degenerates to an exhaustive enumeration of states, incurring an exponential time.

For the DAG scheduling problem, our definition of $f(s)$ is aimed at making the computation of the function efficient, since the time required to expand can be very costly. We first define $g(s)$ to be the maximum finish time of all the scheduled nodes. That is, $g(s) = max_i\{FT(n_i)\}$. Obviously $g(s)$ is well-defined in that it essentially represents the length of the partial schedule.

The function $h(s)$ is defined as: $h(s) = max_{n_j \in succ(n_{max})}\{sl(n_j)\}$, where n_{max} is the node corresponding to the value of $g(s)$. Thus, $h(s)$, which can also be easily computed, represents an estimate of the "remaining" schedule length.

Theorem 1: $h(s)$ is admissible.

Proof: Observe that the function $h(s)$ is less than or equal to the time period between the finish time of the exit node, which is lying on the same path as n_{max}, and $FT(n_i)$. Thus, we have $h(s) \le h^*(s)$ for any state *s*, and hence $h(s)$ is admissible. (Q.E.D.)

It should be noted that the simple definition of the heuristic function $h(s)$ permits very efficient implementation of the states expansion process which is critical to enhance the efficiency of the A* algorithm. This issue will be illustrated again later when we describe our experimental results. Furthermore, notice that both $g(s)$ and $f(s)$ are monotone functions.

The algorithm, conforming to the convention, uses two lists: a list called OPEN for keeping the un-expanded states, and a list called CLOSED for keeping the expanded states.

THE SERIAL A* SCHEDULING ALGORITHM:

```
(1) Put the initial state $\Phi$ in the OPEN list and set $f(\Phi) = 0$.
(2) Remove from OPEN the search state $s$ with the smallest $f$, and put it on the list CLOSED.
(3) If $s$ is the goal state, a complete and optimal schedule is found and the algorithm stops; otherwise, go to the next step.
(4) Expand the state $s$ by exhaustively matching all the ready nodes to the processors. Each matching produces a new state $s'$. Compute $f(s') = g(s') + h(s')$ for each new state $s'$. Put all the new states in OPEN. Go to step (2).
```

In the worst case, the A* algorithm can require an exponential time and a large memory space to determine the optimal solution. However, with a properly defined admissible under-estimate function $f(s)$, the algorithm is reasonably efficient on average.

3.2 State-Space Pruning

To enhance the search efficiency we propose to augment the A* algorithm by incorporating a number of state-space pruning techniques outlined below:

Processor Isomorphism: If the target system is composed of homogeneous processors connected by a regular network, generation of equivalent state can be avoided (for a ready node with different processors). To identify isomorphic processors, we need the following definitions.

Definition 1: The ready time of PE i, denoted by RT_i, is defined as the finish time of the last node scheduled to PE i.

Definition 2: Two processors PE i and PE j are isomorphic if:
(i) $neighbors_i = neighbors_j$, and
(ii) $RT_i = RT_j = 0$.

The first condition in Definition 2 requires that the two PEs have the same node-degree in the processor-graph and have the same set of neighboring PEs. According to the second condition, two isomorphic PEs have to be empty. This is a strong requirement. A weaker condition could be: $RT_i = RT_j$ *and* the node currently under consideration for scheduling does not have any predecessor and successor scheduled to either *PE i* and *PE j*. However, verifying this weaker condition increases the time-complexity of scheduling because every nodes scheduled to both processors have to be checked. Thus, we assume the stronger condition for the sake of reducing the time-complexity in state-space pruning.

For example, consider the task graph and processor network shown in Figure 1. Suppose we want to generate new search states by scheduling n_1 to the processors. It is obvious that we need to generate only one search state by assigning n_1 to PE 0. Exhaustively matching n_1 to all three processors is not needed since PE 1 and PE 2 are equivalent to PE 0 at this search step.

Priority Assignment: When more than one nodes are ready for scheduling for generating a new state, not all them need to be considered. Instead, only the node with a higher priority will be examined for scheduling before a node with a lower priority. The rationale is that less important nodes (those with less impact on the final schedule length) should be considered later in the search process so as to avoid regenerating some of the already explored states. If more than one node has the same priority, ties are broken randomly.

Node priorities can be assigned using various attributes. Two common attributes for assigning priority are the *t-level* (top level) and *b-level* (bottom level). The *t-level* of a node n_i is the length of the longest path from an entry node to n_i (excluding n_i). Here, the length of a path is the sum of all the node and edge weights along the path. The *t-level* of n_i highly correlates with n_i's *start time* which is determined after n_i is scheduled to a processor. The *b-level* of a node n_i is the length of the longest path from node n_i to an exit node. The *b-level* of a node is bounded by the length of the *critical path*. A critical path (CP) of a DAG is a path with the longest length; clearly, a DAG can have more than one CP. Both the *t-level* and *b-level* can be computed in $O(e)$ time using standard graph traversal procedures like depth-first search. The *b-level* of a node without the edge costs is called the *static b-level* or simply *static level* (*sl*). The *t-level*s, *b-level*s, and *sl*'s of the DAG depicted in Figure 1(a) are shown in Figure 2.

	sl	b-level	t-level
n_1	12	19	0
n_2	10	16	3
n_3	10	16	3
n_4	6	10	4
n_5	7	12	7
n_6	2	2	17

Figure 2: The *sl*'s (static levels), *b-levels*, and *t-levels* of the DAG shown in Figure 1(a).

In the proposed scheduling algorithms, the ready nodes for scheduling while generating a new state are considered in a decreasing order of *b-level* + *t-level*. That is, the node with the largest value of *b-level* + *t-level* will be chosen for generating a new state.

Node Equivalence: By considering the equivalence relation among the nodes, states leading to the same schedule length can be avoided. By equivalence we mean the two states will lead to schedules with the same schedule length. By using *b-level* and *t-level*, we can define the equivalence relation between two nodes in the DAG.

Definition 3: Two nodes n_i and n_j are ***equivalent*** *if:*
(i) $pred(n_i) = pred(n_j)$,
(ii) $w(n_i) = w(n_j)$, *and*
(iii) $succ(n_i) = succ(n_j)$.

Notice that conditions (i) and (iii) together imply that *t-level*(n_i) = *t-level*(n_j), *and b-level*(n_i) = *b-level*(n_j). With this definition, if two nodes are equivalent, they have the same relationships with the predecessors and successors in that they incur the same amount of communication with the predecessors and successors. Furthermore, they will be *ready* simultaneously and the schedule length will be the same no matter which node is selected first. Thus, only one of the two corresponding new states needs to be stored while the other can be safely discarded. For example, in the task graph shown in Figure 1(a), n_2 and n_3 are equivalent. This is obvious with reference to the values of *b-level*s and *t-level*s shown in Figure 2.

Upper-Bound Solution Cost: In this method, we use an upper bound cost *U* to eliminate a newly generated state. That is, if the state *s* has its $g(s)$ greater than *U*, we can safely discard such a state because it can never lead to an optimal schedule since $g(s)$ is a monotonic increasing function. We use a fast heuristic to determine *U*. The heuristic runs in a linear time and consists of two steps [14]:

(1) Construct a list of tasks ordered in decreasing priorities for the DAG.
(2) Schedule the nodes on the list one by one to the processor that allows the earliest start time.

As both steps take $O(e)$ time, the upper bound cost can be determined in a linear time.

To illustrate how the proposed A* algorithm and the state-space pruning techniques work, we apply the algorithm to schedule the example task graph shown in Figure 1(a) to the 3-processor ring shown in Figure 1(b). The search tree depicting the scheduling steps is shown in Figure 3. Each state in the search tree contains the node chosen for scheduling and the processor to which the node is scheduled. The cost of the state, decomposed into two $g(s)$ and $h(s)$, is also shown. The numbers shown next to some of the states in the search tree indicate the order of state expansion. For this example 26 states are generated and 9 states are expanded. The effectiveness of the proposed A* algorithm is clear when we compare this search tree size with the size of an exhaustive search tree, which contains more than $3^6 = 729$ states.

At the beginning of the search, only n_1 is ready for scheduling and is therefore expanded. Only one state is generated as a result of this expansion. This is because we used the processor isomorphism criterion to avoid generating two equivalent states as initially there is no difference between PE 0, PE 1, and PE 2. This pruning is important since it eliminates a large part of the search space by avoiding the expansion of states situated at a higher level of the search tree. In the next step, this newly generated state is then expanded because it is the only state in the OPEN list. Now four states are generated by scheduling n_2 and n_4 to PE 0 and PE 1. Only two processors are considered because PE 1 and PE 2 satisfy the processor isomorphism criterion at this stage. Note that we could have generated two more states by scheduling n_3 to PE 0 and PE 1. However, since n_2 and n_3 are equivalent nodes by Definition 3, only one of them is chosen for scheduling. Again a large part of the search space is disregarded. In the next step, the state representing the scheduling of n_4 to PE 0 is chosen for expansion because of its least cost. Two states are generated by scheduling n_2 to PE 0 and PE 1. Again we used the equivalence relation between n_2 and n_3 to avoid generating two more states. Since the costs of the two newly generated states are higher than that of a previously generated state, namely the state representing the scheduling of n_4 to PE 1, the latter is chosen for expansion. Next three states are generated because the empty processor PE 2 is different from PE 1, to which n_4 is scheduled, so that processor isomorphism cannot be used. Once again the equivalence relation is used to avoid generating three redundant states. One of the newly generated states, which represents the scheduling of n_2 to PE 0, has the minimum cost, and therefore, is chosen for expansion. As a result of this expansion, only two new states are generated since n_3 is the only ready node and processor PE 2 is not considered due to processor isomorphism. As higher costs states are generated, the search reverts to expanding the left-most state on the second level of the search tree (note that we do not count the initial state as one level).

The state representing the scheduling of n_4 to PE 1 is not generated because it has been visited before on the right-most branch of the search tree. The search then proceeds in a similar fashion and eventually reaching a goal state with a final cost of 14 time units. The path of the search tree representing the optimal scheduling is shown in thick edges in the tree of Figure 3. The final optimal schedule is shown in Figure 4.

3.3 The Parallel A* Algorithm for Scheduling

An efficient parallelization of the above algorithm is non-trivial and requires several design considerations, which will be elaborated below. First, to avoid ambiguity, we will hereafter call the processors executing the parallel A* algorithm as *physical processing elements* (PPEs). The PPEs should be distinguished from the *target processing elements* (TPEs), to which the DAG is to be scheduled. The PPEs are connected by a certain network topology; for instance, the PPEs in the Intel Paragon are connected by a mesh topology.

The initial load distribution phase of the parallel algorithm involves all the PPEs in the system. Every PPE initializes the OPEN list by expanding the initial empty state. Suppose there

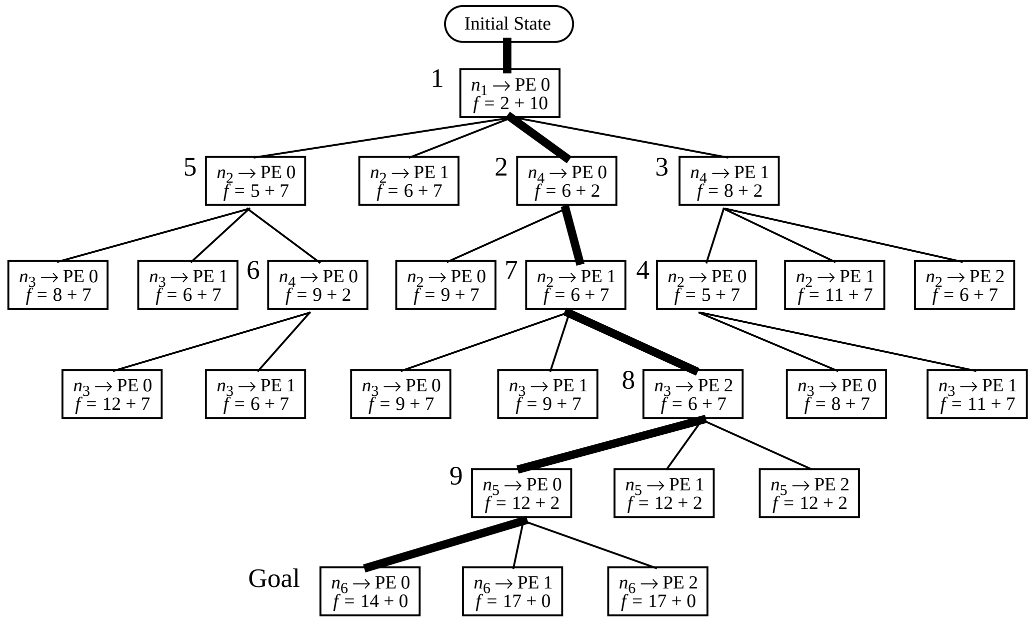


Figure 3: The search tree for scheduling the example task graph shown in Figure 1(a) to the 3-processor ring shown in Figure 1(b).

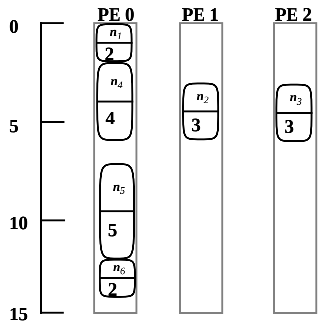


Figure 4: The optimal schedule generated by the A* algorithm (schedule length = 14 time units).

are q PPEs in the parallel machine and each PPE has d neighbor PPEs; for instance, in the case of the Intel Paragon, the value of d is 4. Moreover let there be k states in OPEN. For distributing k states to q PPEs, there are three cases to consider:

Case 1 ($k > q$): Each PPE expands only the initial empty state which results in k new states. Every PPE then gets one state while the extra states are distributed in a round-robin (RR) manner.

Case 2 ($k = q$): Each PPE expands the initial empty state and then gets exactly one state.

Case 3 ($k < q$): Each PPE keeps on expanding states in OPEN starting from the initial empty state until the number of states in OPEN is greater than or equal to q. The list OPEN is then sorted in an increasing order of cost values. Each PPE then gets one state from OPEN in an interleaved manner. That is, the first state in OPEN will go to PE 0, the second state to PE $q-1$, the third state to PE 1, the fourth state to PE $q-2$, and so on. The extra states, if any, will be distributed in a RR fashion. Although there is no guarantee that a best state at the initial level of the search tree will lead to a promising state after some expansions, the algorithm tries to distribute the good states as evenly as possible among all PPEs.

Following the distribution, the PPEs start searching by expanding the states. In the absence of communication among the PPEs, some of the PPEs may work on more promising parts of the search space while others may expand unnecessary states, which are the ones the serial algorithm would not expand. This will inevitably result in a poor speedup since the parallel algorithm would be quite inefficient. To avoid this, the PPEs must periodically communicate with each other to exchange information about the costs of the nodes in the OPEN list. In our scheme, each PPE only communicates with its neighboring PPEs to execute the following local load-balancing algorithm:

ROUND-ROBIN LOAD SHARING:

(1) Determine the average number of un-expanded states N_{avg} in all the OPEN lists.

(2) Every PPE which has the number of local un-expanded states is more than N_{avg}, distribute the surplus states to the deficit PPEs in a round-robin fashion.

The duration of the communication period is set to be T number of expansions, where T is initially $v/2$, then $v/4$, $v/8$, and so on, until it reaches 2 which is the minimum we used. The periods are exponentially decreasing because at the beginning of the search, the costs of the search states differ by a small margin. At such early stages, exploration is more important than exploitation and, therefore, the PPEs should work independently for a longer period of time. When the search reaches the later stages of the search tree, the best cost state tends to have a much smaller cost than the locally best ones.

To avoid excessive overhead, communication is performed only among the neighboring PPEs instead of globally involving all the PPEs. During communication, the neighboring PPEs vote and elect the best cost state, which is then expanded by all the participating PPEs. The resulting new states then go to each neighboring PPE in a RR fashion. Once a goal state is found by any one PPE, it is sent to all the PPEs in the system so that the search can terminate.

Using the above mentioned partitioning, communication, and load balancing schemes, we outline the algorithm as follows.

PARALLEL A* SCHEDULING ALGORITHM:

```
(1)   Expand the initial (empty) state and generate a
      number of new states;
(2)   Eliminate redundant equivalent states;
(3)   Every PPE participates in the initial RR load sharing
      phase;
(4)   Set $i = 2$;
(5)   repeat
(6)     Set $T = \lceil \frac{v}{i} \rceil$;
(7)     repeat /* search */
(8)         Run A* to expand the local states;
(9)     until the number of expanded states is equal to $T$;
(10)    Globally exchange cost information and import
      the best cost state;
(11)    Expand the imported best cost state and perform
      RR load sharing of the generated states;
(12)    Set $i = i \times 2$;
(13)  until a complete schedule is found;
```

It should be mentioned that in our implementation, each PPE only checks for redundant states in its own CLOSED list before adding newly generated states. Although a globally maintained CLOSED list can guarantee no states will be ever re-visited, the communication and synchronization overhead of maintaining the list in a distributed manner can severely limit the scalability of the algorithm. Furthermore, we believe that the states pruning techniques incorporated in the algorithm can effectively discard any redundant states.

When the parallel A* algorithm is applied to schedule the example task graph shown in Figure 1(a) to the 3-processor ring shown in Figure 1(b) using 2 PPEs, the search tree generated, shown in Figure 5, is slightly different. Three more states are generated on the left-most branch of the root in the search tree. This phenomenon can be explained as follows. According to the parallel A* algorithm, both PPEs generate the first two levels of the search tree (note that we do not count the empty initial state as one level) at the beginning as there are not enough states for distribution. One PPE (call it PPE 0) then gets the first and the fourth states, while the other gets the second and the third ones. Initially the period of communication is set to be $6/2 = 3$. Thus, both PPEs work independently until they have expanded 3 states (see Figure 5 for the order of state expansions).

At the time of communication, PPE 0 has fully expanded the right-most branch of the root as shown in Figure 5 (after expansions 1 and 2). For the left-most branch, the states at level 3 (corresponding to assigning n_3 and n_4) are generated. Thus, the best local cost of PPE 0 is 11, corresponding to the state of assigning n_4 to PE 0. On the other hand, PPE 1 has expanded the branch, which contains the goal state, up to the 4th level (corresponding to the states of assigning n_3 to PE 0, 1 and 2). Thus, the best local cost of PPE 1 is 13 (corresponding to assigning n_3 to PE 2).

During the communication, therefore, PPE 0 transfers its best cost state to PPE 1, which is the state of assigning n_4 to PE 0. After the communication, both PPEs expand the best state. However, as a result of the expansion, two states with costs higher than 13 are generated. Therefore, PPE 0 continues to expand the descendant of the best state. On the other hand, PPE 1 reverts to expand its locally best state, the one corresponding to assigning n_3 to PE 2. This expansion leads to generating the parent state of the goal state. Consequently, the goal state is found and PPE 1 broadcasts the result to PPE 0 and the algorithm terminates.

We tested this example on the Intel Paragon using two processors and obtained a speedup of 1.7. Linear speedup is not possible because of the communication overhead and the computation load incurred due to the extra processing in generating more states.

3.4 The Approximate A* Algorithm

According to Pearl *et al.* [16], if a solution with cost bounded by (1+ε) of the optimal cost is good enough, the A* algorithm can be modified to generate such a solution efficiently. We adopt their notations and call the modified A* algorithm, which is in fact an approximate algorithm, as the A_ε^* algorithm. One more list called FOCAL is needed for A_ε^*. This list contains a subset of states currently on the OPEN list. Specifically, we put only the states s' with $f(s') \le (1+\varepsilon) min_{s \in OPEN}\{f(s)\}$. That is, the costs of the states in FOCAL are no greater than the minimum cost in OPEN by a factor (1+ε). Thus A_ε^* only expands states from FOCAL. Furthermore, A_ε^* always finds a solution with cost not exceeding the optimal cost by more than a factor of (1+ε). This is formalized by the following theorem.

Theorem 2: [16] A_ε^* is ε-admissible.

Proof: By definition of FOCAL, the cost of the goal state f_{goal} is within a factor of (1+ε) from the minimum cost f_{min} in FOCAL. Since f is admissible by Theorem 1, we also have $f_{min} \le f_{opt}$. Thus: $f_{goal} \le (1+\varepsilon) f_{min} \le (1+\varepsilon) f_{opt}$ [16]. (Q.E.D.)

4 Performance Results

In this section we present the performance results of the proposed serial and parallel A* algorithms. We first compare the results generated by the serial A* algorithm with that of the branch-and-bound algorithm proposed by Chen and Yu [3]. We then describe the speedup achieved with the parallel A* algorithm implemented on the Intel Paragon, over the proposed serial algorithm. Finally, we also compare the results generated by using the parallel A_ε^* algorithm with those of the exact algorithm.

4.1 Workload

As no widely accepted benchmark graphs exist for the DAG scheduling problem, we believe using random graphs with diverse parameters is appropriate for testing the performance of the algorithms. In our experiments we used three sets of random task graphs, each with a different value of CCR (0.1, 1.0, and 10.0). Each set consists of graphs in which the number of nodes vary from 10 to 32 (with an increment of 2); thus, each set contains 12 graphs. The graphs were randomly generated as follow. First the computation cost of each node in the graph was randomly selected from a uniform distribution with mean equal to 40. Beginning from the first node, a random number indicating the number of children was chosen from a uniform distribution with mean equal to $v/10$. Thus, the connectivity of the graph increases with the size of the graph. The communication cost of an edge was also randomly selected from a uniform distribution with mean equal to 40 times the specified value of CCR. Notice that as we were also interested in the minimum TPEs required for each optimal schedule, we let the algorithms use $O(v)$ TPEs. However, in practice, the algorithms used far less than v TPEs during the search process because redundant states were generated when the algorithms tried to use a new TPE.

4.2 Results of the Serial A* Algorithm

In our first experiment, we ran the serial A* algorithm using a single processor on the Intel Paragon. We also implemented the branch-and-bound algorithm proposed by Chen *et al.* on the same platform. We measured the running times (in seconds) of both algorithms. The results are shown in Table 1. To assess the effectiveness of the pruning techniques,

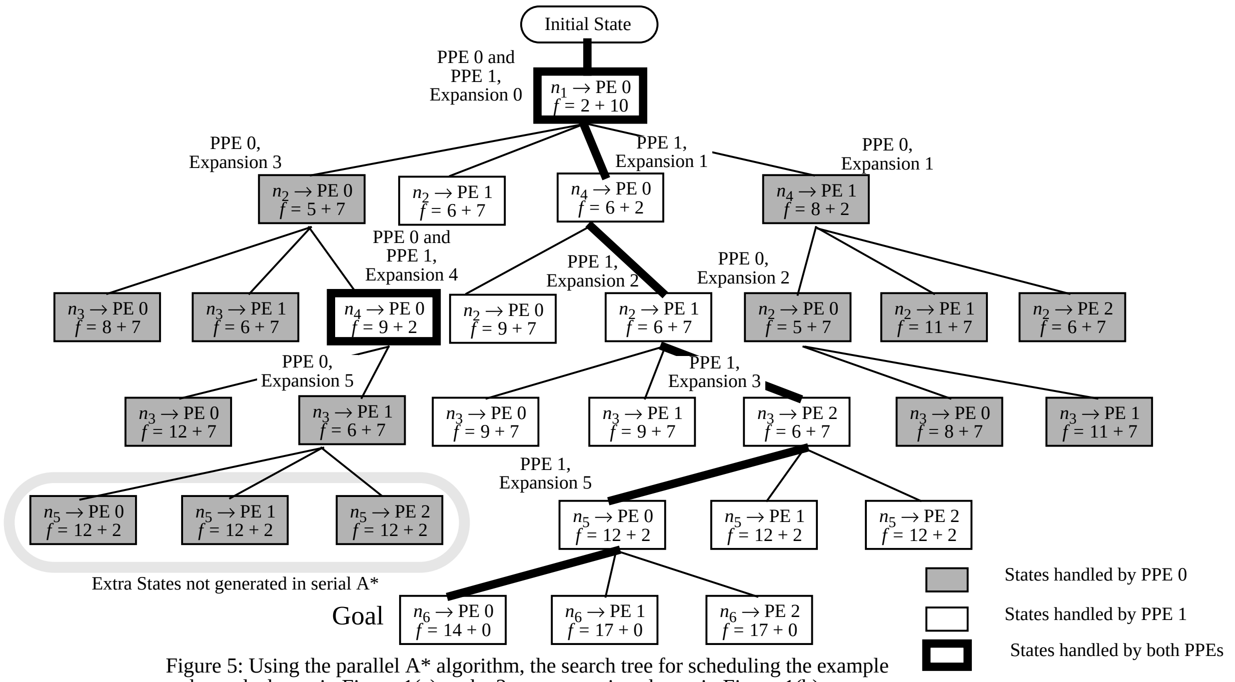

Initial State
PPE 0 and PPE 1, Expansion 0
$n_1 \rightarrow$ PE 0 $f = 2 + 10$
PPE 0, Expansion 3
PPE 1, Expansion 1
PPE 0, Expansion 1
$n_2 \rightarrow$ PE 0 $f = 5 + 7$
$n_2 \rightarrow$ PE 1 $f = 6 + 7$
$n_4 \rightarrow$ PE 0 $f = 6 + 2$
$n_4 \rightarrow$ PE 1 $f = 8 + 2$
PPE 0 and PPE 1, Expansion 4
PPE 1, Expansion 2
PPE 0, Expansion 2
$n_3 \rightarrow$ PE 0 $f = 8 + 7$
$n_3 \rightarrow$ PE 1 $f = 6 + 7$
$n_4 \rightarrow$ PE 0 $f = 9 + 2$
$n_2 \rightarrow$ PE 0 $f = 9 + 7$
$n_2 \rightarrow$ PE 1 $f = 6 + 7$
$n_2 \rightarrow$ PE 0 $f = 5 + 7$
$n_2 \rightarrow$ PE 1 $f = 11 + 7$
$n_2 \rightarrow$ PE 2 $f = 6 + 7$
PPE 0, Expansion 5
PPE 1, Expansion 3
$n_3 \rightarrow$ PE 0 $f = 12 + 7$
$n_3 \rightarrow$ PE 1 $f = 6 + 7$
$n_3 \rightarrow$ PE 0 $f = 9 + 7$
$n_3 \rightarrow$ PE 1 $f = 9 + 7$
$n_3 \rightarrow$ PE 2 $f = 6 + 7$
$n_3 \rightarrow$ PE 0 $f = 8 + 7$
$n_3 \rightarrow$ PE 1 $f = 11 + 7$
PPE 1, Expansion 5
$n_5 \rightarrow$ PE 0 $f = 12 + 2$
$n_5 \rightarrow$ PE 1 $f = 12 + 2$
$n_5 \rightarrow$ PE 2 $f = 12 + 2$
$n_5 \rightarrow$ PE 0 $f = 12 + 2$
$n_5 \rightarrow$ PE 1 $f = 12 + 2$
$n_5 \rightarrow$ PE 2 $f = 12 + 2$
Extra States not generated in serial A*
Goal
$n_6 \rightarrow$ PE 0 $f = 14 + 0$
$n_6 \rightarrow$ PE 1 $f = 17 + 0$
$n_6 \rightarrow$ PE 2 $f = 17 + 0$
States handled by PPE 0
States handled by PPE 1
States handled by both PPEs

Figure 5: Using the parallel A* algorithm, the search tree for scheduling the example task graph shown in Figure 1(a) to the 3-processor ring shown in Figure 1(b).

Table 1: The running times (in seconds) of the Chen *et al.'s* branch-and-bound algorithm and the proposed A* algorithm, with and without pruning, using a single processor on the Intel Paragon for task graphs with CCR equal to 0.1, 1.0, and 10.0.

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Size	Chen	A^*_{full}	A*	Chen .	A^*_{full}	A*	Chen	A^*_{full}	A*
10	202	150	120	289	260	191	367	256	204
12	458	295	245	558	410	312	703	537	428
14	1043	675	567	1349	867	682	1873	903	735
16	2781	1523	1209	3218	2098	1543	3965	2014	1689
18	5231	3109	2465	6345	3568	2863	7624	3731	3025
20	13492	7128	5667	16112	9546	6751	17872	9043	7365
22	29484	15680	12098	32367	15785	13257	34650	19754	15324
24	60129	32045	25688	68492	35689	28462	70326	35789	30257
26	139852	74570	59809	142725	80234	63125	153247	89964	68532
28	289092	153473	125687	309356	179835	142568	324687	180053	152371
30	593412	305673	256892	620605	357923	276581	687001	347900	302674
32	—	652489	525788	—	687924	563284	—	701235	598352

the running times of the A* algorithm without state-space pruning are also shown in the middle column of the table.

As can be observed from the values in the first two columns of the table in Table 1 (results for CCR equal to 0.1) the proposed A* algorithm consistently used much less time than the branch-and-bound algorithm, even without the state-space pruning strategies. This is primarily due to the our algorithm's lower time-complexity for computing the cost of each state. This observation reveals that reducing the complexity of the cost function evaluation method itself can reduce the algorithm's running time. The result for a graph size with 32 nodes was not available for the algorithm by Chen *et al.* because the running time exceeded the limit.

Similar observations can be made about the results with CCR equal to 1.0 and 10.0. There is one major difference, though. The running times spent by both algorithms increased with the value of CCR. This is because with a larger value of CCR, the costs of the intermediate states vary more vigorously and, thus, the search has to explore a wider scope in the search-space.

Comparing the data on the second and third column of Table 1, we can see that the pruning techniques reduce the running times consistently by about 20%. One plausible explanation is that for general graph structures, the proposed A* algorithm behaves more conservatively in that it does not prune many search-states by the solution cost bounding strategy. This is because the f costs of some search-states (where some nodes are sub-optimally scheduled) are less than the upper bound solution cost.

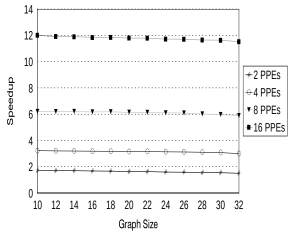
4.3 Results of the Parallel A* Algorithm

In the second experiment, we ran the parallel A* algorithm on the Intel Paragon using 2, 4, 8, and 16 PPEs. The running times used were compared with the serial A* algorithm. The results are presented as speedup plots shown in Figure 6. As can be noticed from the three plots, the speedup of the parallel A* algorithm is moderately less than linear, which is very encouraging. An explanation for the good speedups is that during the communication phases, only neighboring PPEs exchange information on the locally best states. Furthermore, the Intel Paragon has a very fast communication network which permits the PPEs exchange small messages (i.e., the partial nodes assignment and cost) in a short time compared with the processing time for states expansion.

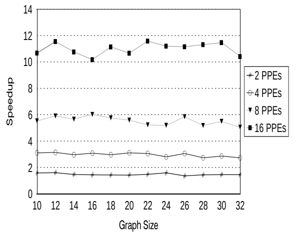
We also observe that the speedup dropped slightly with increasing graph sizes. This is because the parallel A* algorithm tends to generate slightly more search states, which are not generated by the serial algorithm. Another reason is that the communication overhead becomes more significant for larger graph sizes. Comparing the three plots, we find that the speedup curves become more irregular when the value of CCR is higher. This is because as CCR gets higher, the parallel algorithm uses more diverged search directions which are then regulated by the inter-PPE communication.

4.4 Results of the Parallel A_ε* Algorithm

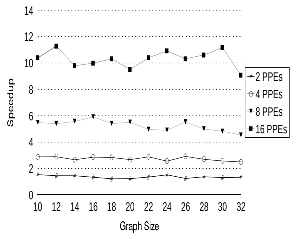
In the last experiment, we ran the parallel A_ε* algorithm using 16 PPEs on the Intel Paragon with approximation factor ε equal to 0.2 and 0.5. The percentage deviations from optimal



(a) CCR = 0.1



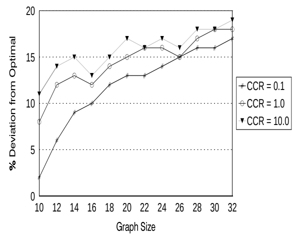
(b) CCR = 1.0



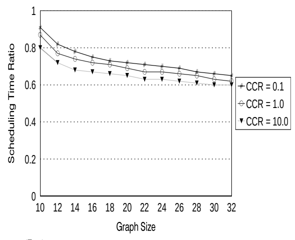
(c) CCR = 10.0

Figure 6: Speedups of the parallel A* algorithm using 2, 4, 8, and 16 PPEs on the Intel Paragon for task graphs with CCR equal to (a) 0.1; (b) 1.0; and (c) 10.0.

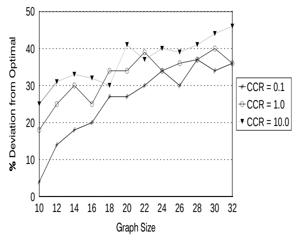
schedule lengths were measured and the scheduling time ratios of the parallel $A_\varepsilon*$ algorithm to the parallel A* algorithm were noted. The results are plotted in Figure 7, indicating that the actual percentage deviations from optimal are not as great as the approximation factor in both cases. This is particularly true for smaller graphs, and is due to the fact that the FOCAL list does not exclude the states leading to an optimal goal for a reasonably effective cost function. Regarding the scheduling time ratios, we find that the computation time saved for each case is of considerable margin—ranges from 10 to 40% for ε equal to 0.2 and from 50 to 70% for ε equal to 0.5. Thus, the parallel $A_\varepsilon*$ algorithm is an attractive choice if slightly inferior to optimal solutions are acceptable.



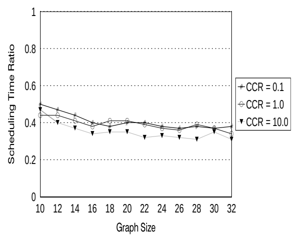
(a) 16 PPEs, ε = 0.2



(b) 16 PPEs, ε = 0.2



(c) 16 PPEs, ε = 0.5



(d) 16 PPEs, ε = 0.5

Figure 7: The deviation from optimal schedule length (plots (a) and (c)) of the schedules generated by the parallel A_ε* algorithm using 16 PPEs on the Intel Paragon and the scheduling time ratio (plots (b) and (d)) of the A_ε* algorithm to the A* algorithm with ε = 0.2 and 0.5.

5 Conclusions

In this paper, we have presented algorithms for optimal static scheduling of an arbitrary DAG to an arbitrary number of processors. The proposed algorithms are based on the A* search technique with a computationally efficient cost function as well as a number of effective state-space pruning techniques. The serial A* algorithm is found to outperform a previously proposed branch-and-bound algorithm by using considerably less time to generate a solution. The parallel A* algorithm, using a dynamic load balancing strategy, yields a close-to-linear speedup. Based on experimental evaluation, the parallel $A_\varepsilon*$ algorithm has shown high capability to prune the search-space; thereby reducing the running time dramatically. The $A_\varepsilon*$ algorithm is scalable and an attractive choice if slightly inferior to optimal solutions are acceptable.

References

[1] H.H. Ali and H. El-Rewini, "The Time Complexity of Scheduling Interval Orders with Communication is Polynomial," *Parallel Processing Letters*, vol. 3, no. 1, 1993, pp. 53-58.

[2] P.C. Chang and Y.S. Jiang, "A State-Space Search Approach for Parallel Processor Scheduling Problems with Arbitrary Precedence Relations," *European Journal of Operational Research*, 77, 1994, pp. 208-223.

[3] G.-H. Chen and J.-S. Yu, "A Branch-And-Bound-With-Underestimates Algorithm for the Task Assignment Problem with Precedence Constraint," Proc. *Int'l Conf. Distributed Computing Systems*, 1990, pp. 494-501.

[4] H.-C. Chou and C.-P. Chung, "Optimal Multiprocessor Task Scheduling Using Dominance and Equivalence Relations," *Computers Operations Research*, 21 (4), 1994, pp. 463-475.

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

[6] R. Correa and A. Ferreira, "On the Effectiveness of Synchronous Parallel Branch-and-Bound Algorithms," *Parallel Processing Letters*, vol. 5, no. 3, 1995, pp. 375-386.

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

[8] R.M. Karp and Y. Zhang, "Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation," *Journal of the ACM*, vol. 40, no. 3, July 1993, pp. 765-789.

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

[10] W.H. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *J. ACM*, vol. 21, Jan. 1974, pp. 140-156.

[11] V. Kumar, K. Ramesh, and V.N. Rao, "Parallel Best-First Search of State-Space Graphs: A Summary of Results," Proc. *7th Int'l Conf. Art. Intell. (AAAI'88)*, Aug. 1988, vol. 1, pp. 122-127.

[12] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, May 1996, pp. 506-521.

[13] —, "Efficient Scheduling of Arbitrary Task Graphs Using A Parallel Genetic Algorithm," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, Nov. 1997, pp. 58-77.

[14] Y.-K. Kwok, I. Ahmad, and J. Gu, "FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," Proc. *25th Int'l Conf. on Parallel Processing*, Aug. 1996, vol. II, pp. 150-157.

[15] N.R. Mahapatra and S. Dutt, "Scalable Global and Local Hashing Strategies for Duplicate Pruning in Parallel A* Graph Search," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 7, July 1997, pp. 738-756.

[16] J. Pearl and J.H. Kim, "Studies in Semi-Admissible Heuristics," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 4, July 1982, pp. 392-399.

[17] D.R. Smith, "Random Trees and the Analysis of Branch-and-Bound Procedures," *Journal of the ACM*, vol. 31, no. 1, Jan. 1984, pp. 163-188.

[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, Sep. 1994, pp. 951-967.