Benchmarking the Task Graph Scheduling Algorithms

Yu-Kwong Kwok[1] and Ishfaq Ahmad[2]

[1]Parallel Processing Laboratory, School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285, USA

[2]Department of Computer Science
The Hong Kong University of Science and Technology, Hong Kong

Abstract†

The problem of scheduling a weighted directed acyclic graph (DAG) to a set of homogeneous processors to minimize the completion time has been extensively studied. The NP-completeness of the problem has instigated researchers to propose a myriad of heuristic algorithms. While these algorithms are individually reported to be efficient, it is not clear how effective they are and how well they compare against each other. A comprehensive performance evaluation and comparison of these algorithms entails addressing a number of difficult issues. One of the issues is that a large number of scheduling algorithms are based upon radically different assumptions, making their comparison on a unified basis a rather intricate task. Another issue is that there is no standard set of benchmarks that can be used to evaluate and compare these algorithms. Furthermore, most algorithms are evaluated using small problem sizes, and it is not clear how their performance scales with the problem size. In this paper, we first provide a taxonomy for classifying various algorithms into different categories according to their assumptions and functionalities. We then propose a set of benchmarks which are of diverse structures without being biased towards a particular scheduling technique and still allow variations in important parameters. We have evaluated 15 scheduling algorithms, and compared them using the proposed benchmarks. Based upon the design philosophies and principles behind these algorithms, we interpret the results and discuss why some algorithms perform better than the others.

Keywords: Performance Evaluation, Benchmarks, Multiprocessors, Parallel Processing, Scheduling, Task Graphs, Scalability.

1 Introduction

The problem of scheduling a weighted directed acyclic graph (DAG), also called a task graph or macro-dataflow graph, to a set of homogeneous processors to minimize the completion time, has intrigued researchers even before the advent of parallel computers. The problem is NP-complete in its general forms [15], and polynomial-time solutions are known only for a few restricted cases [11]. Since tackling the scheduling problem in an efficient manner is imperative for achieving a meaningful speedup from a parallel or distributed system, it continues to be a focus of great attention from the research community. Considerable research efforts expended in solving the problem have resulted in a myriad of heuristic algorithms. While each heuristic is individually reported to be efficient, it is not clear how effective these algorithms are and how they compare against each other on a unified basis.

The objectives of this study include proposing a set of benchmarks and using them to evaluate the performance of a set of DAG scheduling algorithms (DSAs) with various parameters and performance measures. Since a large number of DSAs have been reported in the literature with radically different assumptions, it is important to demarcate these algorithms into various classes according to their assumptions about the program and machine model. A performance evaluation and comparison study should provide answers to the following questions:

- **What are the important performance measures?** The performance of a DSA is usually measured in terms of the quality of the schedule (the total duration of the schedule) and the running time of the scheduling algorithm. Sometimes, the number of target processors allocated is also taken as a performance parameter. One problem is that usually there is a trade-off between the first two performance measures; that is, efforts to obtain better solution often incur a higher time-complexity. Furthermore, using more processors can possibly result in a better solution. Another problem is that most algorithms are evaluated using small problem sizes, and it is not known how the algorithms scale with the problem size.
- **What problem parameters affect the performance?** The performance of DSAs, in general, tends to bias towards the problem graph structure. In addition, other parameters such as the communication-to-computation ratio, the number of nodes and edges in the graph, and the number of target processors also affect the performance of a DSA. Thus, it is important to measure the performance of DSAs by robustly testing them with various ranges of such parameters.
- **What benchmarks should be used?** There does not exist any set of benchmarks that can be considered as a standard to evaluate and compare various DSAs on a unified basis. The most common practice is to use random graphs. The use of task graphs derived from various parallel applications is also common. However, in both cases, there is again no standard that can provide a robust set of test cases. Therefore, there is a need for a set of benchmarks that are representative of various types of synthetic and real test cases. These test cases should be diverse without being biased towards a particular scheduling technique and should allow variations in important parameters.
- **How does the performance of DSAs vary?** Since most DSAs are based on heuristics techniques, bounds on their performance levels and variations from the optimal solution are not known. In addition, the average, worst, and best case performance of these algorithms is not known. Furthermore, since not all DSAs make identical assumptions, they must be segregated and evaluated within various categories.
- **Why some algorithms perform better?** Although some qualitative and quantitative comparisons of some DSAs have been carried out in the past (see [16], [19], [30]), they mainly presented experimental results without giving a rationale of why some algorithms performs well and some do not. The previous studies were also limited to a few algorithms and did not make a comprehensive evaluation. The design philosophies and characteristics of various algorithms must be understood in order to assess their merits and deficiencies. The qualitative analyses can ensue some future guidelines for designing even better heuristics.

In this paper, we describe a performance study of various DSAs with the aim of providing answers to the questions posed above. First, we define the DAG scheduling problem in the next section, and provide an overview of various fundamentals scheduling techniques and attributes that are shared by a vast number of DSAs in Section 3. This is followed by a chronological summary and a taxonomy of various DSAs reported in the literature presented in Section 4. Since it is not the objective of this research to provide a survey on this topic, the purpose of this taxonomy is to set a context in which we select a set of algorithms for benchmarking. We select 15 algorithms and explain their major characteristics. All of these algorithms have been implemented on a common platform and tested using the same suite of benchmark task graphs with a wide

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

range of parameters which will be introduced in Section 5. Comparisons are made within each group whereby these algorithms are ranked from the performance and complexity standpoints. Section 6 includes the results and comparisons and Section 7 concludes the paper.

2 The Model

We consider the general model assumed for a task graph that has been commonly used by many researchers (see [8] for explanation). Some simplifications in the model are possible, and will be introduced later. We assume the system consists of a number of identical (homogeneous) processors. Although scheduling on heterogeneous processors is also an interesting problem, we confine the scope of this study to homogeneous processors only. The number of processors can be limited (given as an input parameter to the scheduling algorithm) or unlimited.

The DAG is a generic model of a parallel program consisting of a set of processes (nodes) among which there are dependencies. 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. A node with no parent is called an entry node and a node with no child is called an exit node. The weight on a node is called the *computation cost* of a node n_i and is denoted by $w(n_i)$. The graph also has directed edges representing a partial order among the tasks. The partial order introduces a precedence-constrained directed acyclic graph (DAG) and implies that if $n_i \rightarrow n_j$, then n_j is a child which cannot start until its parent n_i finishes and sends its data to n_i. The weight on an edge is called the *communication cost* of the edge and is denoted by $c(n_i, n_j)$. This cost is incurred if n_i and n_j are scheduled on different processors and is considered to be zero if n_i and n_j are scheduled on the same processor. 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.

The node and edge weights are usually obtained by estimation or profiling [13], [32]. Scheduling of a DAG is performed statically (i.e., at compile time) since the information about the DAG structure and costs associated with nodes and edges must be available *a priori*. The objective of DAG scheduling is to find an assignment and the start times of tasks to processors such that the schedule length (i.e., the overall duration of the schedule) is minimized such that the precedence constraints are preserved.

3 Characteristics of Scheduling Algorithms

The general DAG scheduling problem has been shown to be NP-complete [15], and remains intractable even with highly simplifying assumptions applied to the task and machine models [26], [27]. Nevertheless, polynomial-time algorithms for some special cases have been reported: [4], [11].

In view of the intractability of the problem, researchers have resorted to designing efficient heuristics which can find good solutions within a reasonable amount of time. Most scheduling heuristic algorithms are based on the *list scheduling* technique. The basic idea in list scheduling is to assign priorities to the nodes of the DAG and place the nodes in a list arranged in descending order of priorities. The node with a higher priority is examined for scheduling before a node with a lower priority; if more than one node has the same priority, ties are broken using some method.

There are, however, numerous variations in the methods of assigning priorities and maintaining the ready list, and criteria for selecting a processor to accommodate a node. We describe below some of these variations and show that they can be used to characterize most scheduling algorithms.

***Assigning Priorities to Nodes*:** Two major attributes for assigning priorities 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 in the DAG (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 earliest possible start time. The *t-level* of a node is a dynamic attribute because the weight of an edge may be zeroed when the two incident nodes are scheduled to the same processor. The *b-level* of a node n_i is the length of the longest path from node n_i to an exit node and is bounded by the length of the *critical path*. A critical path (CP) of a DAG, is a path from an entry node to an exit node, whose length is the maximum. Different DSAs have used the *t-level* and *b-level* attributes in a variety of ways. Some algorithms assign a higher priority to a node with a smaller *t-level* while some algorithms assign a higher priority to a node with a larger *b-level*. Still some algorithms assign a higher priority to a node with a larger (*b-level* – *t-level*). In general, scheduling in descending order of *b-level* tends to schedule critical path nodes first while scheduling in ascending order of *t-level* tends to schedule nodes in a topological order [16].

***Insertion* vs. *Non-Insertion*:** When determining the start time of a node on a processor *P*, some algorithms only consider scheduling a node after the last node on *P*. Some algorithms also consider other idle time slots on *P* and may insert a node between two already scheduled nodes.

***Critical-Path-Based* vs. *Non-Critical-Path-Based*:** Critical-path-based algorithms determine scheduling order or give a higher priority to a critical-path node (CPN). Non-critical-path-based algorithms do not give special preference to CPNs; they assign priorities simply based on the levels or other attributes of the nodes.

***Static List* vs. *Dynamic List*:** The set of ready nodes are often maintained as a *ready list*. Initially, the ready list includes only the entry nodes. After a node is scheduled, the nodes freed by the scheduled node are inserted into the ready list such that the list is sorted in descending order of node priorities. The list can be maintained in two ways: A ready list is static if it is constructed before scheduling starts and remains the same throughout the whole scheduling process. A ready list is called dynamic if it is rearranged according to the changing node priorities.

***Greedy* vs. *Non-Greedy*:** In assigning a node to a processor, most scheduling algorithms attempt to minimize the start-time of a node. This is a greedy strategy. However, some algorithms do not necessarily minimize the start-time of a node but consider other factors as well.

***Time-Complexity*:** The time-complexity of a DSA is usually expressed in terms of the number of node, *v*, the number of edges, *e*, and the number of processors, *p*. The major steps in an algorithm include a traversal of the DAG and a search of slots in the processors to place a node. Simple static priority assignment in general results in a lower time-complexity while dynamic priority assignment inevitably leads to a higher time-complexity of the scheduling algorithm. Backtracking can incur a very high time complexity and is therefore not usually employed.

4 A Classification of DAG Scheduling Algorithms

The static DAG scheduling problem has been tackled with large variations in the task graph and machines models. Figure 1 provides a classification and chronological summary of various static DSAs. Since it is not the purpose of this paper to provide a survey of such algorithms, this summary is by no means complete (a more extensive taxonomy on the general scheduling problem has been proposed in [8]). Furthermore, a complete overview of the literature is beyond the scope of this paper. Nevertheless, we believe our classification scheme can be extended to most of the reported DSAs.

Earlier algorithms have made radically simplifying assumptions about the task graph representing the program and the model of the parallel processor system [11]. These algorithms assume the graph to be of a special structure such as a tree, forks-join, etc. In general, however, parallel programs come in a variety of structures, and as such many recent algorithms are designed to tackle arbitrary graphs. These algorithms can be further divided into two categories. Some algorithms assume the computational costs of all the tasks to be uniform [11] whereas other algorithms assume the computational costs of tasks to be arbitrary. Some of the earlier work has also assumed the inter-task communication to be zero, that is, the task graph contains precedence but without cost. The problem becomes less complex in the absence of communication delays. Furthermore, scheduling with

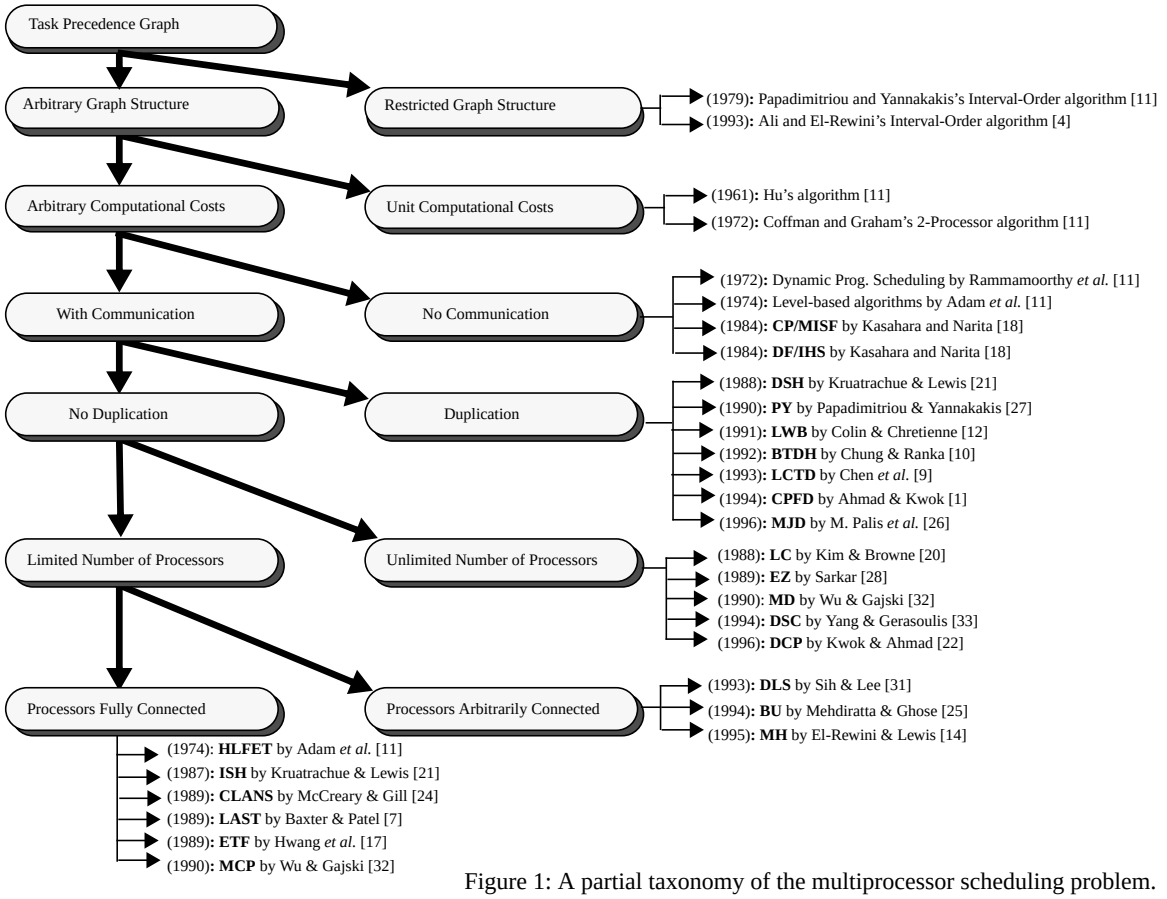


Figure 1: A partial taxonomy of the multiprocessor scheduling problem.

communication delays is NP-complete.

Scheduling with communication may be done using duplication or without duplication. The rationale behind the task-duplication based (TDB) scheduling algorithms is to reduce the communication overhead by redundantly allocating some nodes to multiple processors. In duplication-based scheduling, different strategies can be employed to select ancestor nodes for duplication.

Non-TDB algorithms assuming arbitrary task graphs with arbitrary costs on nodes and edges can be divided into two categories: some scheduling algorithms assume the availability of unlimited number of processors, while some other algorithms assume a limited number of processors. The former class of algorithms are called the UNC (*unbounded number of clusters*) scheduling algorithms [3] and the latter the BNP (*bounded number of processors*) scheduling algorithms [3]. In both classes of algorithms, the processors are assumed to be fully-connected and no attention is paid to link contention or routing strategies used for communication. The technique employed by the UNC algorithms is also called *clustering* (see [14], [16], [20], [26], [34] for details). At the beginning of the scheduling process, each node is considered a cluster. In the subsequent steps, two clusters† are merged if the merging reduces the completion time. This merging procedure continues until no cluster can be merged. The rationale behind the UNC algorithms is that they can take advantage of using more processors to further reduce the schedule length. However, the clusters generated by the UNC may need a post-processing step for mapping the clusters onto the processors because the number of processors available may be less than the number of clusters.

A few algorithms have been designed to take into account the most general model in which the system is assumed to consist of an arbitrary network topology, of which the links are not contention-free. These algorithms are called the APN (*arbitrary processor network*) scheduling algorithms [3]. In addition to scheduling tasks, the APN algorithms also schedule messages on the network communication links.

In the following we list the algorithms chosen in our study. For detailed descriptions and characteristics we refer the reader to the sited publications. To narrow the scope of this paper, we do not consider TDB algorithms (for a more detailed overview of such algorithm, see [1]).

- **BNP Scheduling Algorithms:** HLFET [11], ISH [21], MCP [32], ETF [17], DLS [31], and LAST [7].
- **UNC Scheduling Algorithms:** EZ [28], LC [20], DSC [34], MD [32], and DCP [22].
- **APN Scheduling Algorithms:** MH [14], DLS [31], BU [25] and BSA [2].

5 Benchmark Graphs

In our study, we propose and use a suite of benchmark graphs consisting of 5 different sets. The generation techniques and characteristics of these benchmarks are described as follows:

5.1 Peer Set Graphs

The *Peer Set Graphs* (PSGs) are example task graphs used by various researchers and documented in publications. These graphs are usually small in size but are useful in that they can be used to trace the operation of an algorithm by examining the schedule produced. A detailed description of the graphs is provided in Section 6.1.

5.2 Random Graphs with Optimal Solutions

These are random graphs for which we have obtained optimal solutions using a branch-and-bound algorithm. We call these graph *random graphs with optimal solutions using branch-and-bound* (RGBOS). This suite of random task graphs consists of three subsets of graphs with different CCRs (0.1, 1.0, and 10.0). Each subset consists of graphs in which the number of nodes vary from

†. We use the term cluster and processor interchangeably since in the UNC scheduling algorithms, merging a single node cluster to another cluster is analogous to scheduling a node to a processor.

10 to 32 with increments of 2, thus, totalling 12 graphs per set. The graphs were randomly generated as follows: First the computation cost of each node in the graph was randomly selected from a uniform distribution with the mean equal to 40 (minimum = 2 and maximum = 78). Beginning with the first node, a random number indicating the number of children was chosen from a uniform distribution with the mean equal to $v/10$. The communication cost of each edge was also randomly selected from a uniform distribution with the mean equal to 40 times the specified value of CCR. To obtain optimal solutions for the task graphs, we applied a parallel A* algorithm [23] to the graphs. Since generating optimal solutions for arbitrarily structured task graphs takes exponential time, it is not feasible to obtain optimal solutions for large graphs.

5.3 Random Graphs with Pre-Determined Optimal Schedules

These are *random graphs with pre-determined optimal solutions* (RGPOS). The method of generating graphs with known optimal schedules is as follows: Suppose that the optimal schedule length of a graph and the number of processors used are specified as L_{opt} and *p*, respectively. For each PE *i*, we randomly generate a number x_i from a uniform distribution with mean v/p . The time interval between 0 and L_{opt} of PE *i* is then randomly partitioned into x_i sections. Each section represents the execution span of one task, thus, x_i tasks are "scheduled" to PE *i* with no idle time slot. In this manner, *v* tasks are generated so that every processor has the same schedule length. To generate an edge, two tasks n_a and n_b are randomly chosen such that $FT(n_a) < ST(n_b)$. The edge is made to emerge from n_a to n_b . As to the edge weight, there are two cases to consider: (i) the two tasks are scheduled to different processors, and (ii) the two tasks are scheduled to the same processor. In the first case the edge weight is randomly chosen from a uniform distribution with maximum equal to $(ST(n_b) - FT(n_a))$ (the mean is adjusted according to the given CCR value). In the second case the edge weight can be an arbitrary positive integer because the edge does not affect the start and finish times of the tasks which are scheduled to the same processor. We randomly chose the edge weight for this case according to the given CCR value. Using this method, we generated three sets of task graphs with three CCRs: 0.1, 1.0, and 10.0. Each set consists of graphs in which the number of nodes vary from 50 to 500 in increments of 50; thus, each set contains 10 graphs.

5.4 Random Graphs without Optimal Schedules

The fourth set of benchmark graphs, referred to as *random graphs with no known optimal solutions* (RGNOS), consists of 250 randomly task graphs. The method of generating these random task graphs is the same as that in RGBOS. However, the sizes of these graphs are much larger, varying from 50 nodes to 500 nodes with increments of 50. For generating the complete set of 250 graphs, we varied three parameters: *size*, *communication-to-computation ratio* (CCR), and *parallelism*. Five different values of CCR were selected: 0.1, 0.5, 1.0, 2.0 and 10.0. The parallelism parameter determines the *width (*defined as the largest number of non-precedence-related nodes in the DAG*)* of the graph. Five different values of parallelism were chosen: 1, 2, 3, 4 and 5. A parallelism of 1 means the average width of the graph is $\sqrt{v}$, a value of 2 means the graph has an average width of $2\sqrt{v}$, and so on. Our main rationale for using these large random graphs as a test suite is that they contain as their subset a variety of graph structures. This avoids any bias that an algorithm may have towards a particular graph structure.

5.5 Traced Graphs

The last set of benchmark graphs, called *traced graphs* (TG), represent some of the numerical parallel application programs obtained via a parallelizing compiler [3]. We use Cholesky factorization graphs for this category.

6 Performance Results and Comparison

In this section, we present the performance results and comparisons of the 15 scheduling algorithms which were implemented on a SUN SPARC IPX workstation with all of the benchmarks described above. The algorithms are compared within their own classes, although some comparison of UNC and BNP algorithms are also carried out. The comparisons are made using the following six measures.

- **Normalized Schedule Length (NSL):** The main performance measure of an algorithm is the schedule length of its output schedule. The NSL of an algorithm is defined as: $NSL = L / \left(\sum_{n_i \in CP} w(n_i) \right)$, where L is the schedule length. It should be noted that the sum of computation costs on the CP represents a lower bound on the schedule length. Such lower bound may not always be possible to achieve, and the optimal schedule length may be larger than this bound.
- **Number of Processors Used:** The number of processors used is another important measure of an algorithm and it varies widely for different algorithms. The number of processors used are measured for the BNP and UNC algorithms.
- **Running Time of the Algorithms:** The running time of a scheduling algorithm is another important performance measure because a long running time can severely limit the scalability of an algorithm.

6.1 Results for the Peer Set Graphs

The results of applying the UNC and BNP algorithms to the PSG are shown in Table 1. The APN algorithms were not applied to this set of example graphs because many network topologies are possible as test cases making a fair comparison quite difficult. As can be seen from the table, the schedule lengths produced vary considerably, despite the small sizes of the graphs. This phenomenon is contrary to our expectation that the algorithms would generate the same schedule lengths for most of the cases. It also indicates that the performance of various DSAs is more sensitive to the diverse structures of the graphs rather than their sizes. A plausible explanation for this pathological observation is that the ineffective scheduling technique employed in some algorithms leads to mistakes made in the earlier stages of the scheduling process so that long schedule lengths are produced.

Table 1: Schedule lengths generated by the UNC and BNP algorithms for the PSGs.

Source of task graph	UNC Algorithms					BNP Algorithms					
	LC	EZ	MD	DSC	DCP	HLFET	ISH	ETF	LAST	MCP	DLS
Ahmad and Kwok [2] (a 13-node graph)	485	717	430	404	392	454	454	473	445	454	454
Al-Maasarani [5] (a 16-node graph)	44	44	50	49	44	45	45	44	53	45	44
Al-Mouhamed [6] (a 17-node graph)	39	40	38	38	38	41	38	41	43	40	41
Shirazi *et al.* [29] (a 11-node graph)	39	32	28	30	28	28	33	28	42	33	33
Colin and Chretienne [12] (a 9-node graph)	15	14	15	14	14	14	14	14	14	14	14
Gerasoulis and Yang [16] (a 7-node graph)	25	20	22	18	18	18	18	18	18	21	18
Kruatrachue and Lewis [21] (a 15-node graph)	19	16	11	15	11	11	11	11	15	11	11
McCreary and Gill [24] (a 9-node graph)	212	159	159	160	149	180	180	180	149	180	180
Chung and Ranka [10] (a 11-node graph)	46	42	35	37	35	35	40	35	46	40	40
Wu and Gajski [32] (a 18-node graph)	420	540	420	390	390	390	390	390	470	390	390
Yang and Gerasoulis [33] (a 7-node graph)	19	20	18	16	16	19	16	16	16	16	16

Among the UNC algorithms, the DCP algorithm consistently generate the best solutions. However, there is no single BNP algorithm which outperform all the others. In summary, we make the following observations:

- The greedy BNP algorithms give very similar schedule lengths as can be seen from the results of HLFET, ISH, ETF, MCP and DLS.
- Non-greedy and non-CP-based UNC algorithms in general perform worse than the greedy BNP algorithms.
- CP-based algorithms perform better than non-CP-based ones (DCP, DSC, MD and MCP perform better than others).
- Among the CP-based algorithms, dynamic-list algorithms perform better than static-list ones (DCP, DSC and MD in general perform better than MCP).

6.2 Results for RGBOS benchmarks

The results of the UNC and BNP algorithms for the RGBOS benchmarks are shown in Table 2 and Table 3, respectively. Since

optimal solutions for specific network topologies are not known, the APN algorithms were again not applied to these benchmarks. Table 2 includes the percentage degradations from the optimal schedule lengths produced by the UNC algorithms. The overall average degradations for each algorithm are given in the last row. As can be seen, when CCR is 0.1, both MD and DCP generate optimal solutions for half of the test cases, and the overall average degradation is less than 2%. Other algorithms generate optimal solutions for a few number of cases and the overall average degradation is larger. Among all the UNC algorithms, the DCP algorithm performs the best.

Table 2: The percentage degradations from the optimal solutions of for the RGBOS benchmarks (UNC algorithms).

CCR	0.1					1.0					10.0				
Algorithms	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP
Graph Size 10	0.0	0.0	0.0	0.0	0.0	5.2	8.7	0.0	0.0	0.0	3.1	12.7	6.8	9.4	0.8
12	1.4	4.0	0.5	3.3	4.3	2.3	6.1	1.9	7.9	2.3	4.0	7.0	4.0	0.0	0.0
14	7.8	2.2	0.7	2.5	3.6	9.4	13.9	5.6	6.4	0.0	3.7	6.7	9.2	0.0	0.0
16	0.0	0.9	0.0	0.0	0.0	8.8	3.6	0.9	0.9	4.3	14.0	2.2	0.0	3.4	0.0
18	6.1	5.7	5.8	4.4	3.5	6.0	1.8	0.0	7.6	0.0	10.4	19.8	5.8	11.5	2.4
20	7.0	1.9	3.3	5.6	2.1	3.8	6.7	4.3	6.0	3.9	0.3	0.0	2.9	0.2	0.0
22	5.8	0.0	0.0	0.7	0.0	0.0	7.8	4.1	0.0	0.0	1.5	22.6	11.6	3.4	2.2
24	7.3	2.5	3.7	1.4	1.9	0.2	1.5	6.6	4.8	4.4	14.0	6.7	11.2	10.7	0.0
26	5.2	3.3	0.0	0.1	0.0	8.3	14.1	9.9	9.6	0.0	18.1	3.5	7.1	11.4	5.4
28	7.8	0.0	0.0	4.2	0.0	2.3	14.9	4.2	10.6	0.0	20.7	8.9	11.1	13.9	1.1
30	0.1	1.4	0.0	3.1	0.0	0.0	12.2	3.7	3.0	0.0	5.6	7.1	13.5	3.5	11.6
32	4.6	3.3	1.3	0.4	0.5	3.5	2.2	1.0	5.3	2.8	0.9	3.4	0.8	4.2	0.0
No. of Opt.	2	3	6	2	6	2	0	2	2	7	0	1	1	2	6
Avg. Dev.	4.4	2.1	1.3	2.1	1.3	4.1	7.8	3.5	5.2	1.5	8.0	8.4	7.0	6.0	2.0

Table 3: The percentage degradations from the optimal solutions of for the RGBOS benchmarks (BNP algorithms).

CCR	0.1					1.0					10.0				
Algorithms	HLFETISH	ETF	LAST	MCP	DLS	HLFETISH	ETF	LAST	MCP	DLS	HLFETISH	ETF	LAST	MCP	DLS
Graph Size 10	0.0	2.9	1.4	7.1	3.6	0.0	4.2	0.0	9.3	0.0	0.0	0.0	5.2	10.9	7.0	17.1	2.3	2.0
12	4.3	2.3	4.4	6.7	3.2	0.0	4.2	0.3	0.6	9.9	0.7	5.2	7.4	2.7	0.5	2.7	4.5	0.0
14	2.4	1.2	0.2	2.9	0.4	1.2	0.0	0.0	0.0	3.8	7.3	4.5	2.2	12.1	7.7	13.8	5.1	0.5
16	0.0	3.0	0.0	0.0	3.0	1.6	9.5	1.1	0.2	3.2	2.6	7.8	8.0	8.2	1.0	1.2	0.0	4.1
18	2.0	0.8	3.5	3.4	5.9	5.6	10.5	1.3	0.0	0.0	4.7	1.7	5.1	2.0	8.1	7.3	4.2	0.8
20	5.9	2.9	2.6	10.0	0.3	6.0	6.9	7.8	2.8	8.3	3.2	3.0	5.7	3.8	3.5	5.7	0.0	8.2
22	5.8	0.7	0.1	2.4	1.8	5.4	8.8	0.9	2.5	0.2	4.4	2.8	8.0	3.8	0.3	3.0	7.6	5.5
24	0.5	4.9	3.2	3.1	4.0	4.1	10.9	2.4	2.2	1.4	1.3	4.1	1.5	3.0	0.6	18.8	5.2	4.4
26	2.5	0.0	4.5	0.0	1.4	0.0	8.4	6.3	6.7	3.8	1.8	1.7	2.2	4.6	9.7	11.7	7.4	1.6
28	0.0	6.6	0.4	3.1	0.2	0.4	6.7	0.0	4.0	0.0	0.0	8.4	9.0	6.4	6.5	17.5	5.2	4.7
30	3.2	2.8	0.0	1.2	2.9	4.8	5.1	0.0	5.1	7.6	8.0	2.3	6.9	3.0	5.5	3.4	8.9	13.7
32	6.8	0.1	1.3	6.1	3.8	3.2	13.6	7.2	5.8	10.3	5.4	0.3	18.0	0.0	7.3	20.6	0.8	0.0
No. of Opt.	3	1	2	2	0	3	1	4	2	3	2	1	0	1	0	0	2	2
Avg. Dev.	2.8	2.4	1.8	3.8	2.5	2.7	7.4	2.3	3.3	4.0	3.3	3.5	6.6	5.0	4.8	10.2	4.3	3.8

Table 3 indicates that the BNP algorithms generate fewer optimal solutions compared to the DCP algorithm. The overall average degradations are also higher than that of the DCP algorithm. However, compared with other UNC algorithms, the MCP, ETF, ISH, and DLS algorithms perform better both in terms of the number of optimal solutions generated and the overall degradations. Among all the BNP algorithms, the MCP algorithm performs the best while the LAST algorithm performs the worst.

We summarize our observations from Table 2 and Table 3 as follows.

- Greedy BNP algorithms have shown higher capability in generating optimal solutions than the non-greedy and non-CP-based UNC algorithms with DCP as the only exception.
- CP-based algorithms are clearly better than the non-CP-based ones as can be seen from the results of DCP and MCP.

6.3 Results for the RGPOS Benchmarks

The results of applying the UNC and BNP algorithms to RGPOS benchmarks are shown in Table 4 and Table 5, respectively. Since optimal solutions for specific network topologies are not known, the APN algorithms were again not applied to the RGPOS task graphs. In Table 4, the percentage degradations from the optimal schedule lengths of the UNC algorithms are shown. The overall average degradations for each algorithm are again shown in the last row of the table. As can be seen, when CCR is 0.1, the DCP generates optimal solutions for more than half of the test cases and the overall average degradation is less than 2%. Other algorithms generate optimal solutions for a few number of cases and the overall average degradation is larger. The percentage degradations in general increase with CCRs. When CCR is 10.0, none of the UNC algorithms except DCP can generate any optimal solution. The results given in Table 5 indicate that the BNP algorithms generate a similar number of optimal solutions and values of percentage degradations. When CCR is 10.0, none of the BNP algorithms generates any optimal solutions. In summary, the results of Table 4 and Table 5 lead to similar conclusions as those made in Section 6.2.

Table 4: The percentage degradations from the optimal solutions of for the RGPOS benchmarks (UNC algorithms).

CCR	0.1					1.0					10.0				
Algorithms	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP
Graph Size 50	0.8	7.9	5.7	3.5	0.0	21.3	9.6	0.8	13.4	17.5	17.6	12.8	2.7	23.6	9.2
100	2.6	4.5	5.9	6.1	0.6	0.0	0.0	4.8	14.8	0.0	11.8	21.5	6.6	19.7	3.4
150	0.0	11.1	4.9	8.6	0.0	0.0	29.9	9.1	9.8	0.0	29.9	17.9	20.7	6.7	10.4
200	3.1	0.3	0.0	8.3	0.0	7.2	20.4	14.3	9.0	5.7	17.1	5.8	5.9	14.1	10.8
250	0.0	0.0	3.6	0.0	0.0	5.8	12.8	7.6	14.9	0.0	4.9	12.4	2.1	14.0	4.8
300	5.8	3.7	2.8	5.4	5.2	18.2	32.2	11.5	4.8	0.0	15.7	4.0	15.0	12.9	13.0
350	5.5	15.1	0.3	4.2	0.0	7.0	0.6	11.5	0.0	0.0	10.2	5.0	4.1	7.9	1.2
400	0.0	4.8	5.7	2.5	0.0	11.4	19.8	1.4	1.7	0.0	29.8	21.3	11.8	2.1	0.6
450	0.5	7.4	0.0	6.7	0.0	11.6	17.9	9.5	22.0	12.6	25.2	21.0	7.5	22.8	0.0
500	2.2	13.3	2.5	3.7	5.6	19.8	27.2	10.7	0.0	0.0	6.7	6.6	8.2	11.5	10.7
No. of Opt.	3	1	2	1	7	2	1	0	2	7	0	0	0	0	1
Avg. Dev.	2.0	6.8	3.1	4.9	1.1	10.2	17.0	8.1	9.0	3.6	16.9	12.8	8.5	13.5	6.4

Table 5: The percentage degradations from the optimal solutions of for the RGPOS benchmarks (BNP algorithms).

CCR	0.1					1.0					10.0				
Algorithms	HLFETISH	ETF	LAST	MCP	DLS	HLFETISH	ETF	LAST	MCP	DLS	HLFETISH	ETF	LAST	MCP	DLS
Graph Size 50	0.0	7.8	6.8	11.8	1.6	0.0	13.5	3.8	14.2	23.5	13.8	10.1	4.3	1.0	15.3	13.0	8.1	5.0
100	9.5	2.8	0.9	8.2	2.5	6.9	4.2	4.3	7.2	0.0	3.2	0.0	5.2	1.9	2.3	10.4	5.8	0.7
150	6.3	3.7	9.1	0.0	7.7	1.3	10.0	22.1	5.1	12.8	10.0	0.0	3.0	23.9	8.1	20.3	10.3	13.6
200	7.4	0.0	2.9	4.8	0.0	2.8	2.5	7.7	10.2	14.2	5.1	12.4	3.2	11.1	12.6	22.7	12.1	4.8
250	0.0	0.0	0.0	10.7	0.0	1.2	16.5	16.2	0.0	0.8	14.2	5.0	9.7	17.5	7.1	11.1	3.8	7.3
300	12.4	7.4	6.5	0.3	4.7	1.9	0.6	6.3	7.3	16.5	3.6	10.2	3.1	7.7	9.2	3.4	9.0	0.7
350	3.3	4.2	5.6	0.0	6.4	2.4	20.3	0.0	0.0	17.6	0.0	4.2	28.5	7.3	10.5	29.4	14.0	11.9
400	0.0	7.1	0.0	3.7	2.2	0.0	5.1	1.1	4.3	7.9	7.1	2.5	2.3	10.6	14.1	18.0	0.9	3.0
450	6.4	0.8	0.0	4.6	0.0	1.8	2.6	4.9	10.6	9.1	1.0	0.4	18.6	13.7	21.6	20.1	2.2	17.9
500	6.1	2.9	5.0	6.6	7.6	9.5	6.1	3.5	11.1	25.3	4.8	14.6	7.2	23.8	19.2	3.1	6.9	12.9
No. of Opt.	3	2	3	2	3	2	0	1	2	1	1	2	0	0	0	0	0	0
Avg. Dev.	5.1	3.7	3.7	5.1	3.3	2.8	8.1	7.0	7.0	12.8	6.3	5.9	8.5	11.8	12.0	15.2	7.3	7.8

6.4 Results for the RGNOS Benchmarks

Since the optimal solutions for the RGNOS benchmarks are not known, we evaluate and compare the algorithms with a more extensive range of parameters including graph sizes, CCRs, and parallelisms.

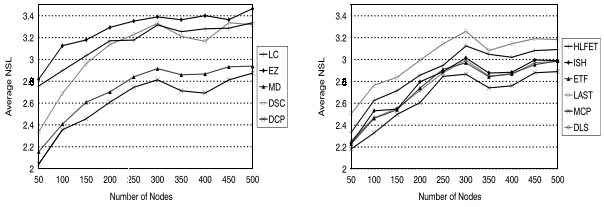
6.4.1 Comparing Schedule Lengths

The average NSLs for the BNP, UNC, and APN scheduling algorithms are given in Figure 2. Each curve in the plots is the average of 25 tests cases with various CCRs and parallelism. Figure 2 reveals that the behavior of these algorithms is consistent in terms of their relative performance for various number of nodes in the graph. Among the BNP scheduling algorithms, the performance of the MCP algorithm is the best while the LAST algorithm is outperformed by all other algorithms. We also observe that the NSLs for all the algorithms show a slightly increasing trend if the task graph size is increased. This is because the proportion of nodes other than those on the CP increases making it more difficult to achieve the lower bound. For the UNC scheduling algorithms, we observe that the DCP and MD algorithms perform significantly better as compared to the rest of the algorithms. The NSLs for the DSC and LC algorithms are similar.

Although the BNP algorithms are designed for a limited number of processors (as an input parameter), we ran each algorithm with a very large number of processors such that the number of processors became virtually unlimited. From this experiment, we noted the average number of processors used by these algorithms for each graph size (the number of processors used is shown later in Figure 3). In another experiment, we reduced the number processors to 50% of that average. Since no significant difference in the NSLs as well as the relative performance of these algorithms can be observed, we do not include those results in this paper. One possible reason for this phenomenon is that the schedule length is dominated by the scheduling of CP nodes. In the case of a very large number of processors, the non-CP nodes are spread across many processors, while in the case of a fewer number of processors, these nodes are packed together without making much impact on the overall schedule length.

For the APN scheduling algorithms, the target architectures included an 8-processor ring, an 8-processor hypercube, a 4×2 mesh, and an 8-processor clique. The average NSLs for these experiments are shown in Figure 2(c). Each point on the curve now represents the average of 100 NSLs. One reason for the much larger NSLs in these cases is that the numbers of processors used were

(intentionally) much smaller. For example, a 500-node task graph is scheduled to 8 processors†. The results of the APN algorithms suggest that there can be substantial difference in the performance of these algorithms. For example, significant differences are observed between the NSLs of BSA and BU. The performance of DLS is relatively stable with respect to the graph size while MH yields fairly long schedule lengths for large graphs. As can be seen, the BSA algorithm performs admirably well for large graphs. The main reason for the better performance of BSA is an efficient scheduling of communication messages that can have a drastic impact on the overall schedule length. In terms of the impact of the topology, we find that all algorithms perform better on the networks with more communication links. However, these results are excluded due to space limitations.



(a) UNC algorithms (b) BNP algorithms

(c) APN algorithms

Figure 2: Average NSL of the UNC, BNP and APN algorithms for RGNOS benchmarks.

6.4.2 Number of Processors Used

The number of processors used by an algorithm is an important performance measure especially for the algorithms that are designed for using an unlimited number of processors. The BNP algorithms are designed for a bounded number of processors, but as explained earlier, were tested them with a very large number (virtually unlimited number) of processors; we then noted the number of processors actually used.

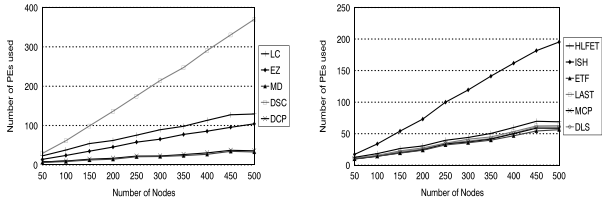
Figure 3(a) shows the average number of processors used by the BNP scheduling algorithms. The DLS algorithm uses the smallest number of processors, even compared to ETF although both algorithms share similar concepts. The numbers of processors used by the MCP and ETF algorithms are close. On the other hand, these numbers for the HLFET and ISH are also similar.

Figure 3(b) shows the average number of processors used by the UNC scheduling algorithms. As can be seen, the DSC algorithm uses a large number of processors. This is because it uses a new processor for every node whose start time cannot be reduced on a processor already in use. The LC and EZ algorithms also use more processors than others because they pay no attention on the use of processors. In contrast, the DCP algorithm has a special processor finding strategy: as long as the schedule length is not affected, it tries to schedule a child to a processor holding its parent even though its start time may not reduce. The MD algorithm also uses relatively smaller number of processors because, to schedules a node to a processor, it first scans the already used processors.

6.4.3 Algorithm Running Times

In this section, we compare the running times of all the algorithms. Table 6 shows the running times of the BNP scheduling algorithms for various number of nodes in the task graph. Each value in the table again is the average of 25 cases. The MCP

†. The number of processors used by a typical UNC algorithm is very large—the LC algorithm, for instance, uses more than 100 processors for a 500-node task graph



(a) UNC algorithms (b) BNP algorithms

Figure 3: The average number of processors used for the RGNOS benchmarks.

algorithm is found to be the fastest algorithm while DLS and ETF are slower than the rest. The large running times of the DLS and ETF algorithms are primarily due to exhaustive calculations of the start times of all of the ready tasks on all of the processors. The running time of LAST and HLFET are also large while ISH takes reasonable amounts of time. Based on these running time results, the BNP algorithms can be ranked in the order: MCP, ISH, HLFET, LAST, and (DLS, ETF).

From the running times of UNC scheduling algorithms shown in Table 6, we observe that the LC and DSC algorithms yield the minimum running time. The running times of MD, EZ and DCP are close. Based on these running time results, these algorithms can be ranked in the order: LC, DSC, EZ, DCP, and MD. For the APN scheduling algorithms, the BU algorithm is found to be the fastest. The running times of the MH and BSA algorithms are close while those of the DLS algorithm are relatively large. Based on these results, in terms of running times, these algorithms can be ranked in the order: BU, BSA, MH, and DLS.

Table 6: Average running times (in seconds) for all the algorithms using the RGNOS benchmarks.

	BNP Algorithms						UNC Algorithms					APN Algorithms			
Graph Sizes	HLFET	ISH	ETF	LAST	MCP	DLS	LC	EZ	MD	DSC	DCP	MH	DLS	BSA	BU
50	0.1	0.1	0.1	0.2	0.1	0.1	0.1	0.4	0.5	0.1	0.4	0.3	1.8	0.2	0.1
100	0.4	0.1	0.3	0.5	0.1	0.3	0.1	1.2	1.3	0.1	1.1	1.8	17.5	1.5	0.1
150	0.7	0.2	0.7	1.1	0.1	0.8	0.2	3.2	3.6	0.2	3.1	6.1	77.6	5.3	0.2
200	1.1	0.4	1.3	2.0	0.2	1.4	0.3	5.4	6.4	0.3	5.6	13.8	181.8	11.5	0.4
250	1.7	0.6	3.4	3.5	0.3	3.2	0.4	9.6	10.3	0.5	9.2	28.5	473.6	22.7	1.0
300	2.2	0.9	5.0	5.0	0.4	4.8	0.6	13.8	14.6	0.7	13.9	44.9	799.5	36.7	1.6
350	3.0	1.2	7.9	7.2	0.6	7.5	0.8	20.8	22.3	1.0	21.8	72.2	1329.6	57.1	2.5
400	3.5	1.5	10.3	9.3	0.7	9.4	1.0	31.6	36.9	1.2	32.1	93.6	2093.9	79.5	3.7
450	4.5	2.1	17.5	12.7	0.9	16.4	1.3	43.9	47.9	1.8	44.6	132.5	3342.2	103.6	5.7
500	5.4	2.5	20.2	16.4	1.1	17.9	1.6	52.3	63.5	2.3	58.6	189.3	4334.8	147.0	7.7

6.5 Results for Traced Graphs

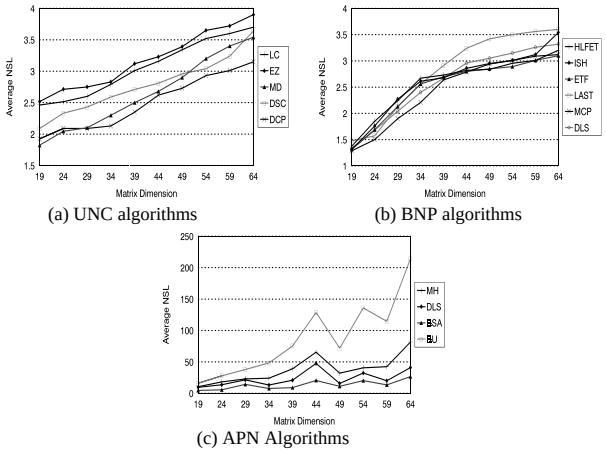
For the *traced graphs* (TG) representing the parallel numerical application, Cholesky factorization, the results are shown in Figure 4. Since the application operates on matrices, the graph sizes depend on the matrix dimensions. For a matrix dimension of N, the graph size is $O(N^2)$. We note that the performance of the BNP algorithms is quite similar with the exception that LAST performs much worse. By contrast, the performance of the UNC algorithms is much diverse. The relative performance of the APN algorithms is quite similar for both applications.

7 Conclusions and Future Work

In this paper, we have presented the results of an extensive performance study of 15 DSAs. Our study has revealed several important findings:

- For both the BNP and UNC classes, algorithms emphasizing the accurate scheduling of nodes on the critical-path are in general better than the other algorithms.
- Dynamic critical-path is better than static critical-path, as demonstrated by both the DCP and DSC algorithms.
- Insertion is better than non-insertion—for example, a simple algorithm such as ISH employing insertion can yield dramatic performance.
- Dynamic priority is in general better than static priority, although it can cause substantial complexity gain—for example the DLS and ETF algorithms have higher complexities. However, this is not always true—one exception, for example, is that the MCP algorithm using static priorities performs the best in the BNP class.

We have provided a set of benchmarks which provide a variety of test cases including two kinds of graphs with optimal solutions.



(a) UNC algorithms (b) BNP algorithms

(c) APN Algorithms

Figure 4: Average NSL for Cholesky factorization task graphs.

These can be good test cases for evaluating and comparing future algorithms.

The current research concentrates on further elaboration of various techniques, such as reducing the scheduling complexities, improving computation estimations, and incorporating network topology and communication traffic. A promising avenue for solving the first two problems is by parallelizing static scheduling on the target parallel machine where the user program executes [2].

In UNC algorithms, clusters obtained through scheduling are assigned to a bounded number of processors. All nodes in a cluster must be scheduled to the same processor. This property makes the cluster scheduling algorithms more complex than the standard BNP scheduling algorithms. Two such algorithms called Sarkar's assignment algorithm and Yang's RCP algorithm are described in [28] and [33], respectively. Sarkar's algorithm combines the cluster merging and ordering nodes into one step, considering the execution order. RCP merges clusters without considering the execution order, which may lead to a poor decision on merging. However, RCP has a lower complexity. Both algorithms are simple and do not utilize the information provided by the UNC scheduling. Generally, cluster scheduling is a relatively unexplored area. More effective algorithms are to be designed. It would be an interesting study to compare the BNP approach with the UNC+CS approach.

The APN algorithms can be fairly complicated because they take into account more parameters. Further research is required in this area, and the effects of topology and routing strategy need to be determined.

References

[1] I. Ahmad and Y.-K. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," Proc. of *Int'l Conf. Parallel Processing*, vol. II, pp. 47-51, Aug. 1994.

[2] —, "A Parallel Approach to Multiprocessor Scheduling," Proc. of *Int'l Parallel Processing Symposium*, Apr. 1995, pp. 289-293.

[3] I. Ahmad, Y.-K. Kwok, M.-Y. Wu and W. Shu, "Automatic Parallelization and Scheduling of Programs on Multiprocessors using CASCH," Proc. of *Int'l Conf. Parallel Proc.*, Aug. 1997, pp. 288-291.

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

[5] A. Al-Maasarani, *Priority-Based Scheduling and Evaluation of Precedence Graphs with Communication Times*, M.S. Thesis, King Fahd University of Petroleum and Minerals, Saudi Arabia, 1993.

[6] M.A. Al-Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs," *IEEE Trans. Software Engineering*, vol. 16, no. 12, Dec. 1990, pp. 1390-1401.

[7] J. Baxter and J.H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," Proc. of *Int'l Conf. Parallel Processing*, vol. II, pp. 217-222, Aug. 1989.

[8] T.L. Casavant and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. on Soft. Eng.*, vol. 14, no. 2, pp. 141-154, Feb. 1988.

[9] H. Chen, B. Shirazi and J. Marquis, "Performance Evaluation of A Novel Scheduling Method: Linear Clustering with Task Duplication," Proc. of *Int'l Conf. on Parallel and Dist. Sys.*, pp. 270-275, Dec. 1993.

[10] Y.C. Chung and S. Ranka, "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," *Proc. of Supercomputing'92*, pp. 512-521, Nov. 1992.

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

[12] J.Y. Colin and P. Chretienne, "C.P.M. Scheduling with Small Computation Delays and Task Duplication," *Operations Research*, pp. 680-684, 1991.

[13] M. Cosnard and M. Loi, "Automatic Task Graph Generation Techniques," *Parallel Proc. Lett.*, Dec. 1995, pp. 527-538.

[14] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138-153, Jun. 1990.

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

[16] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276-291, Dec. 1992.

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

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

[19] A.A. Khan, C.L. McCreary and M.S. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," Proc. of *Int'l Conf. on Parallel Processing*, vol. II, pp. 243-250, Aug. 1994.

[20] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. of Int'l Conference on Parallel Processing*, vol. II, pp. 1-8, Aug. 1988.

[21] B. Kruatrachue and T.G. Lewis, "Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems," Technical Report, Oregon State University, Corvallis, OR 97331, 1987.

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

[23] —, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Technique," *submitted for publication*.

[24] C. McCreary and H. Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing," *Communications of the ACM*, vol. 32, pp. 1073-1078, Sep. 1989.

[25] N. Mehdiratta and K. Ghose, "A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessor," Proc. of *Int'l Conf. on Parallel Processing*, vol. II, pp. 151-154, Aug. 1994.

[26] M.A. Palis, J.-C. Liou, and D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Trans. Parallel and Dist. Systems*, vol. 7, no. 1, Jan. 1996, pp. 46-55.

[27] —, "Towards an Architecture-Independent Analysis of Parallel Algorithms," *SIAM Journal on Computing*, vol. 19, no. 2, pp. 322-328, Apr. 1990.

[28] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.

[29] B. Shirazi, H. Chen and J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-clustering Techniques," *Concurrency: Practice and Experience*, vol. 7(5), Aug. 1995, pp. 371-390.

[30] 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.

[31] 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.

[32] 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.

[33] T. Yang and A. Gerasoulis, "List Scheduling with and without Communication Delays," *Parallel Comp.*, (19), 1993, pp. 1321-1344.

[34] —, "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.