Journal of Parallel and Distributed Computing **59**, 381–422 (1999)
Article ID jpdc.1999.1578, available online at http://www.idealibrary.com on IDEAL®

Benchmarking and Comparison of the Task Graph Scheduling Algorithms[1]

Yu-Kwong Kwok

Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong
E-mail: ykwok@eee.hku.hk

and

Ishfaq Ahmad

Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
E-mail: iahmad@cs.ust.hk

Received September 2, 1998; revised March 19, 1999; accepted June 14, 1999

The problem of scheduling a parallel program represented by a weighted directed acyclic graph (DAG) to a set of homogeneous processors for minimizing the completion time of the program has been extensively studied. The NP-completeness of the problem has stimulated researchers to propose a myriad of heuristic algorithms. While most of these algorithms are reported to be efficient, it is not clear how they compare against each other. A meaningful performance evaluation and comparison of these algorithms is a complex task and it must take into account a number of issues. First, most scheduling algorithms are based upon diverse assumptions, making the performance comparison rather meaningless. Second, there does not exist a standard set of benchmarks to examine these algorithms. Third, most algorithms are evaluated using small problem sizes, and, therefore, their scalability is unknown. In this paper, we first provide a taxonomy for classifying various algorithms into distinct categories according to their assumptions and functionalities. We then propose a set of benchmarks that are based on diverse structures and are not biased toward a particular scheduling technique. We have implemented 15 scheduling algorithms and compared them on a common platform by using the proposed benchmarks, as well as by varying important problem parameters. We interpret the results based upon the design philosophies and principles behind these algorithms, drawing inferences why some algorithms perform better than others. We

[1] This research was supported by a grant from the Hong Kong Research Grants Council under Contracts HKUST 734/96E and HKUST 6076/97E. A preliminary version of this work was presented at the 12th International Parallel Processing Symposium (IPPS'98), Orlando, FL.

0743-7315/99 $30.00

also propose a performance measure called *scheduling scalability* (SS) that captures the collective effectiveness of a scheduling algorithm in terms of its solution quality, the number of processors used, and the running time.

Key Words: 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 in order to minimize the completion time, has intrigued researchers for quite some time [22]. The problem is NP-complete in its general form [18], and polynomial-time solutions are known only for a few restricted cases [13, 17]. Since efficient scheduling imperative for achieving a meaningful speedup from a parallel or distributed system, it continues to spur interest among 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 these algorithms 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 of DSAs 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 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 a better solution often incur a higher time-complexity. On one extreme, one can try to find an optimal or close-to-optimal solution using, for instance, a branch-and-bound technique [24] or other time-consuming search methods. On the other extreme, one can try to employ a fast technique which can yield an adequate solution. 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 very clear how they scale with the problem size. Thus, there is a need to determine a performance measure that should be an indicative parameter of a DSA's scalability, as well as of the trade-off between its solution quality and running time.

- *What problem parameters affect the performance?* The performance of DSAs, in general, tends to bias toward a particular 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. 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 a 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, again in both cases, there is 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, as well as real, test cases. These test cases should be diverse without being biased toward 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 solutions are not known. In addition, the average, worst, and best case performances of these algorithms are not known. Furthermore, since the existing DSAs are based on different assumptions, they must be segregated and evaluated, within distinct categories.

- *Why some algorithms perform better?* Some qualitative and quantitative comparisons of some DSAs have been carried out in the past (see [19, 25, 39]), but they mainly presented experimental results without giving a rationale of why some algorithms perform 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 stem 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 and provide an overview of various fundamental scheduling techniques and attributes that are shared by a vast number of DSAs. Next, we provide a chronological summary and a taxonomy of various DSAs reported in the literature. Since a survey on this topic is not the objective, the purpose of this taxonomy is to set a context wherein we select a set of algorithms for benchmarking. We select 15 algorithms and examine their salient characteristics. We have implemented these algorithms on a common platform and tested them using the same suite of benchmark task graphs with a wide range of parameters. We make comparisons within each group whereby these algorithms are ranked from the performance and complexity standpoints. We also define a new performance measure called *scheduling scalability* which captures the collective effectiveness of a scheduling algorithm in terms of its solution quality, the number of processors used, and the running time.

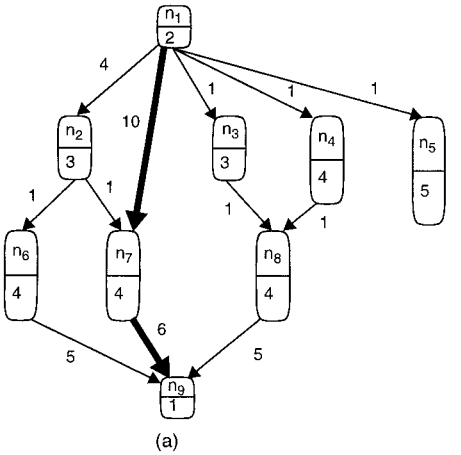
The rest of this paper is organized into six sections. In the next section, we describe the generic DAG model. In Section 3, we describe the basic scheduling techniques and a number of concepts that can be used to explain various algorithms in the later discussion. Section 4 provides a taxonomy and a brief survey of

various DSAs. The same section also includes brief descriptions and characteristics of the DSAs chosen for our performance study. Section 5 describes a set of benchmarks that we propose and subsequently use for performance evaluation. Section 6 includes the results and comparisons and Section 7 concludes the paper.

2. THE DAG MODEL

We consider the general model assumed for a task graph that has been commonly used by many researchers (see [10, 13] 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 could 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 small example DAG is shown in Fig. 1. Formally, a DAG consists of v nodes, $n_1, n_2, ..., n_v$, that can be executed on any of the available processors. 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 has one or more inputs. When all inputs are available, the node is triggered to execute. After its execution, it generates its outputs. 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 e directed edges representing a partial order among the tasks. The partial order introduces a precedence-constrained directed acyclic graph 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_j. 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



(a)

node	SL	t-level	b-level	ALAP
*n_1	11	0	23	0
n_2	8	6	15	8
n_3	8	3	14	9
n_4	9	3	15	8
n_5	5	3	5	18
n_6	5	10	10	13
*n_7	5	12	11	12
n_8	5	8	10	13
*n_9	1	22	1	22

(b)

FIG. 1. (a) A task graph; (b) the static levels (SLs), t-levels, b-levels, and ALAPs of the nodes.

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 [15, 42] for which the parameters are determined using benchmark profiling techniques [23]. 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*.

If node n_i is scheduled to processor P, $ST(n_i, P)$ and $FT(n_i, P)$ denote the start time and finish time of n_i on processor P, respectively. After all nodes have been scheduled, the schedule length is defined as $\max_i\{FT(n_i, P)\}$ across all processors. The objective of DAG scheduling is to find an assignment and the start times of tasks to processors such that the schedule length 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 [18], and remains intractable even with severe assumptions applied to the task and machine models [32, 34, 41]. Nevertheless, polynomial-time algorithms for some special cases have been reported; Hu [20] devised a linear-time algorithm to solve the problem of scheduling a uniform node-weight free-tree to an arbitrary number of processors. Coffman and Graham [13] devised a quadratic-time algorithm to solve the problem of scheduling an arbitrarily structured DAG with uniform node-weights to two processors. The third case is to schedule an interval-ordered DAG with uniform node-weights to an arbitrary number of processors. A DAG is called interval-ordered if every two precedence-related nodes can be mapped to two non-overlapping intervals on the real number line. Papadimitriou and Yannakakis [33] designed a linear-time algorithm to tackle the problem. In all these three cases, communication between tasks is ignored. Recently, Ali and El-Rewini [6] showed that interval-ordered DAG with uniform edge weights, which are equal to the node weights, can also be optimally scheduled in polynomial-time.

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 start time, denoted by $T_S(n_i)$, which is determined after n_i is scheduled to a processor. 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. Thus, the path reaching a node, whose length determines the *t-level* of the node, may cease to be the longest one.

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. In Fig. 1a, the edges of the CP are shown with thick arrows.

Variations in the computation of the *b-level* of a node are possible. Most DSAs examine a node for scheduling only after all the parents of the node have been scheduled. In this case, the *b-level* of a node is a constant until after it is scheduled to a processor. However, some algorithms allow the scheduling of a child before its parents. In that case, the *b-level* of a node becomes a dynamic attribute. In these algorithms, the value of $T_S(n_i)$ for any node n_i is not fixed until all the nodes are scheduled, allowing the insertion of a node to a time slot created by pushing some earlier scheduled nodes downward. It should be noted that some scheduling algorithms do not take into account the edge weights in computing the *b-level*. To distinguish such a definition of *b-level* from the one described above, we call it the *static b-level* or simply static level (SL).

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. The composite attribute (*b-level* − *t-level*) is a compromise between the previous two cases. The attributes for the example DAG are shown in Fig. 1b). Note that the nodes of the CP (n_1, n_7, n_9 which are marked by an asterisk) can be identified with their values of (*b-level* + *t-level*) because all of them have the maximum value 23.

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). Noncritical-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, thus, is usually not 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. Table 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 [10]). 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 [1, 13]. These algorithms assume the graph to be of a special structure such as a tree or forks-join. 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 [13, 20], whereas other algorithms assume the computational costs of tasks to be arbitrary. Some of the earlier work has also assumed the intertask 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 communication delays is NP-complete.

Scheduling with communication may be done with 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

TABLE 1

A Partial Taxonomy of the Multiprocessor Scheduling Problem

Algorithm	Special graph structure	Unit computational costs	With communication	Duplication	Unlimited number of processors	Processor network topology
Scheduling for restricted graphs						
Hu's algorithm (1961) [20]	Tree	Yes	No	No	Yes	Clique
Papadimitriou and Yannakakis's Interval-Order algorithm (1979) [33]	Interval-order	Yes	No	No	Yes	Clique
Coffman and Graham's 2-Processor algorithm (1972) [13]	—	Yes	No	No	Yes	Clique
Ali and El-Rewini's Interval-Order algorithm (1993) [6]	Interval-Order	Yes	Yes	No	Yes	Clique
Dynamic Prog. Scheduling by Rammamoorthy *et al.* (1972) [36]	—	No	No	No	Yes	Clique
Level-based algorithms by Adam *et al.* (1974) [1]	—	No	No	No	Yes	Clique
CP/MISF by Kasahara & Narita (1984) [24]	—	No	No	No	Yes	Clique
DF/IHS by Kasahara & Narita (1984) [24]	—	No	No	No	Yes	Clique
Scheduling for unrestricted graphs						
Task duplication based (TDB) scheduling algorithms						
DSH by Kruatrachue & Lewis (1988) [27]	—	No	Yes	Yes	Yes	Clique
PY by Papadimitriou & Yannakakis (1990) [34]	—	No	Yes	Yes	Yes	Clique
LWB by Colin & Chretienne (1991) [14]	—	No	Yes	Yes	Yes	Clique

BTDH by Chung & Ranka (1992) [12]	—	No	Yes	Yes	Yes	Clique
LCTD by Chen *et al.* (1993) [11]	—	No	Yes	Yes	Yes	Clique
CPFD by Ahmad & Kwok (994) [2]	—	No	Yes	Yes	Yes	Clique
MJD by M. Palis *et al.* (1996) [32]	—	No	Yes	Yes	Yes	Clique
DFRN by Park *et al.* (1997) [35]	—	No	Yes	Yes	Yes	Clique
Unbounded number of clusters (UNC) scheduling algorithms						
LC by Kim & Browne (1988) [26]	—	No	Yes	No	Yes	Clique
EZ by Sarkar (1989) [37]	—	No	Yes	No	Yes	Clique
MD by Wu & Gajski (1990) [42]	—	No	Yes	No	Yes	Clique
DSC by Yang & Gerasoulis (1994) [44]	—	No	Yes	No	Yes	Clique
DCP by Kwok & Ahmad (1996) [28]	—	No	Yes	No	Yes	Clique
Bounded number of processors (BNP) scheduling algorithms						
HLFET by Adam *et al.* (1974) [1]	—	No	Yes	No	No	Clique
ISH by Kruatrachue & Lewis (1987) [27]	—	No	Yes	No	No	Clique
CLANS by McCreary & Gill (1989) [30]	—	No	Yes	No	No	Clique
LAST by Baxter & Patel (1989) [9]	—	No	Yes	No	No	Clique
ETF by Hwang *et al.* (1989) [21]	—	No	Yes	No	No	Clique
MCP by Wu & Gajski (1990) [42]	—	No	Yes	No	No	Clique
Arbitrary processor network (APN) scheduling algorithms						
MH by El-Rewini & Lewis (1990) [16]	—	No	Yes	No	No	Arbitrary
BU by Mehdiratta & Ghose (1994) [31]	—	No	Yes	No	No	Arbitrary
DLS by Sih & Lee (1993) [40]	—	No	Yes	No	No	Arbitrary

employed to select ancestor nodes for duplication. A common technique, however, is to recursively duplicate ancestor nodes in a bottom-up fashion, as is done in the recently proposed DFRN algorithm [35]. A more extensive discussion and evaluation of TDB scheduling algorithms can be found in [2].

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 an 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 [5] and the latter the BNP (*bounded number of processors*) scheduling algorithms [5]. 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 [16, 19, 26, 32, 44] for details). At the beginning of the scheduling process, each node is considered a cluster. In the subsequent steps, two clusters[2] 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 postprocessing 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 [5]. In addition to scheduling tasks, the APN algorithms also schedule messages on the network communication links.

To narrow the scope of this paper, we do not consider TDB algorithms (for a more detailed overview of such algorithms, see [2]).

4.1. BNP Scheduling Algorithms

In the following, we discuss six BNP scheduling algorithms: HLFET [1], ISH [27], MCP [42], ETF [21], DLS [40], and LAST [9]. The major characteristics of these algorithms are summarized in Table 2. In the table, p denotes the number of processors given. Some of the complexities do not have this parameter because the algorithms use $O(v)$ processors.

The HLFET algorithm. The HLFET (highest level first with estimated times) algorithm [1] is one of the simplest scheduling algorithms. The algorithm schedules a node to a processor that allows the earliest start time. The main problem with HLFET is that in calculating the SL of a node, it ignores the communication costs on the edges.

[2] 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.

TABLE 2

Some of the BNP Scheduling Algorithms and Their Characteristics

Algorithm	Priority	CP-Based	List Type	Greedy	Complexity
HLFET	SL	No	Static	Yes	$O(v^2)$
ISH	SL	No	Static	Yes	$O(v^2)$
LAST	ALAP	Yes	Static	Yes	$O(v^2 \log v)$
ETF	SL	No	Static	Yes	$O(pv^3)$
MCP	$SL - T_S$	No	Dynamic	Yes	$O(pv^3)$
DLS	edge weights	No	Dynamic	Yes	$O(v(v+e))$

The ISH Algorithm. The ISH (insertion scheduling heuristic) algorithm [27] uses a simple but effective idea of using holes created by the partial schedules. The algorithm first picks an unscheduled node with the highest SL and schedules it to a processor that allows the earliest start time, and thus essentially possesses the same drawback as the HLFET algorithm. The ISH algorithm tries to "insert" other unscheduled nodes from the ready list into the idle time slot before the node just scheduled.

The MCP algorithm. The MCP (modified critical path) algorithm [42] uses the ALAP time of a node as a priority. The ALAP time of a node is computed by first computing the length of CP and then subtracting the *b-level* of the node from it. Thus, the ALAP times of the nodes on the CP are just their *t-levels*. The MCP algorithm first computes the ALAP times of all the nodes and then constructs a list of nodes in ascending order of ALAP times. Ties are broken by considering the ALAP times of the children of a node. The algorithm then schedules the nodes on the list one by one such that a node is scheduled to a processor that allows the earliest start time using the insertion approach.

The ETF algorithm. The ETF (Earliest Time First) algorithm [21] computes, at each step, the earliest start times for all ready nodes and then selects the one with the smallest start time. Here, the earliest start time of a node is computed by examining the start time of the node on all processors exhaustively. When two nodes have the same value of their earliest start times, the algorithm breaks the tie by scheduling the one with the higher SL. Thus, a node with a higher SL does not necessarily get scheduled first because the algorithm gives a higher priority to a node with the earliest start time.

The DLS algorithm. The DLS (dynamic level scheduling) algorithm [40] uses an attribute called dynamic level (DL) which is the difference between the SL of a node and its earliest start time on a processor. Then, similar to the ETF algorithm, the DLS algorithm constructs a pool of ready nodes. At each scheduling step, the algorithm computes the DL for every node in the ready pool on all processors. The node-processor pair that gives the largest value of DL is selected for scheduling. This mechanism is very similar to the one used by the ETF algorithm. However, there is one difference between the ETF algorithm and the DLS algorithm: the former always schedules the node with the minimum earliest start time and uses SL merely to break ties, while the latter tends to schedule nodes in descending order of SLs at the beginning of scheduling process but tends to schedule nodes in ascending order of *t-levels* (i.e., the earliest start times) near the end of the scheduling process.

The LAST algorithm. The LAST (localized allocation of static tasks) algorithm [9] is not a list scheduling algorithm and uses for node priority an attribute called D_NODE, which depends only on the incident edges of a node. The main goal of the LAST algorithm is to minimize the overall communication. This goal, however, does not necessarily lead to the minimization of the completion time. One of the consequences of using D_NODE is that a node may be selected for scheduling before some of its parents. Thus, the earliest start time of a node cannot be fixed

until the scheduling process terminates. Furthermore, a node may need to be inserted into any idle time slot on a processor. The LAST algorithm ignores node weights in the node selection precess.

4.2. UNC Scheduling Algorithms

We select five UNC scheduling algorithms, EZ [37], LC [26], DSC [44], MD [42], and DCP [28] for our performance study. Table 3 includes some of the characteristics of these algorithms.

The EZ algorithm. The EZ (edge-zeroing) algorithm [37] selects clusters for merging based on edge weights. At each step, the algorithm finds the edge with the largest weight. The two clusters incident by the edge are merged if the merging (thereby zeroing the largest weight) does not increase the completion time. After two clusters are merged, the ordering of nodes in the resulting cluster is based on the SLs of the nodes.

The LC algorithm. The LC (linear clustering) algorithm [26] merges nodes to form a single cluster, based on the CP. The algorithm first determines the set of nodes constituting the CP. It then schedules all of the CP nodes to a single processor at once. These nodes and all edges incident on them are then removed from the DAG. The algorithm zeroes the edges on the entire CP at once. However, when an edge is zeroed, the CP may change. The edge that should be zeroed next may not be on the original CP.

The DSC algorithm. The DSC (dominant sequence clustering) algorithm [44] considers the dominant sequence (DS) of a graph. The DS is simply the CP of the partially scheduled DAG. The DSC algorithm tracks the CP of the partially scheduled DAG at each step by using the composite attribute (*b-level* + *t-level*) as the priority of a node. The DSC algorithm does not select the node with the highest priority for scheduling unless the node is ready. This is done in order to lower the time complexity of the algorithm because the *t-level* of a node can be computed incrementally and the *b-level* does not change until the node is scheduled. The algorithm scans through all clusters to find the one that allows the minimum start time of the node, provided that such selection will not delay the start time of a not yet scheduled CP node.

The MD algorithm. The MD (mobility directed) algorithm [42] selects a node n_i for scheduling, based on an attribute called the relative mobility, which is defined as:

$$\frac{\text{Cur\_CP\_Length} - (\text{b-level}(n_i) + \text{t-level}(n_i))}{w(n_i)}.$$

Cur_CP_Length is the length of the current longest path in the DAG. If a node is on the current CP of the partially scheduled DAG, the sum of its *b-level* and *t-level* is equal to the current CP length. Thus, the relative mobility of a node is zero if it is on the current CP. At each step, the MD algorithm selects the node with the smallest relative mobility for scheduling. In testing whether a cluster can accommodate a node, the MD algorithm scans from the earliest idle

TABLE 3

Some of the UNC Scheduling Algorithms and Their Characteristics

Algorithm	Priority	List	CP-based	Greedy	Complexity
LC	SL	Dynamic	No	No	$O(e(v+e))$
EZ	SL + t-level	Static	No	Yes	$O(v(v+e))$
MD	SL + t-level	Dynamic	Yes	Yes	$O((e+v)\log v)$
DSC	b-level + t-level	Dynamic	Yes	No	$O(v^3)$
DCP	b-level + t-level	Dynamic	Yes	No	$O(v^3)$

time slot on the cluster and schedules the node into the *first* idle time slot that is large enough for the node. An idle time slot can be created or made larger by possibly pulling some already scheduled nodes downward.

The DCP algorithm. The DCP (dynamic critical path) algorithm [28] is designed, based on an attribute which is similar to relative mobility. The DCP algorithm uses a look-ahead strategy to find a better cluster for a given node. In addition to computing the value of $T_S(n_i)$ on a cluster, the DCP algorithm also computes the value of $T_S(n_c)$ on the same cluster, where n_c is the child of n_i that has the largest communication and is called the critical child of n_i. The DCP algorithm schedules n_i to the cluster that gives the minimum value of the sum of these two attributes. This look-ahead strategy can potentially avoid scheduling a node to a cluster that has no room to accommodate a heavily communicated child of the node. The DCP algorithm examines all the existing clusters for a node while the MD algorithm only tests from the first cluster and stops after finding one that has a large enough idle time slot.

4.3. APN Scheduling Algorithms

In the following, we discuss four such algorithms, namely, the MH [16], DLS [40], BU [31], and BSA [3] algorithms. Some of their characteristics are given in Table 4.

The MH algorithm. The MH (mapping heuristic) algorithm [16] initializes a ready node list that contains all entry nodes ordered in decreasing priorities. Each node is scheduled to a processor that gives the smallest start time. In calculating the start time of a node, a routing table is maintained for each processor. The table contains the information as to which path to route messages from the parent nodes to the nodes under consideration. After a node is scheduled, all of its ready successor nodes are appended to the ready node list.

The DLS algorithm. The DLS (dynamic level scheduling) algorithm [40] described earlier can also be used as an APN scheduling algorithm. To use it as a APN scheduling algorithm, it requires the message routing method to be supplied by the user. The T_S of a node is then computed according to how the messages from the parents of the node are routed.

The BU algorithm. The BU (bottom up) algorithm [31] first finds the CP of the DAG and then assigns all the nodes on the CP to the same processor at once. Afterward the algorithm assigns the remaining nodes in a reversed topological order to the processors. The node assignment is guided by a load-balancing processor selection heuristic which attempts to balance the load across all given processors. After all the nodes are assigned to some processors, the BU algorithm tries to schedule the communication messages among them using a channel allocation heuristic which tries to keep the hop count of every message roughly a constant constrained by the processor network topology. Different network topologies require different channel allocation heuristics.

TABLE 4

Some of the APN Scheduling Algorithms and Their Characteristics

Algorithm	Node priority	CP-based	Message routing	Complexity
MH	SL	No	Needs routing table	$O(v(p^3v+e))$
DLS	$SL-T_S$	Yes	Needs routing table	$O(v^3p^2)$
BU	—	Yes	Hard-coded	$O(v^2 \log v)$
BSA	—	Yes	Adaptive	$O(p^2ev)$

The BSA algorithm. The BSA (bubble scheduling and allocation) algorithm [3] constructs a schedule incrementally by first injecting all the nodes to the pivot processor, defined as the processor with the highest degree. The algorithm then tries to improve the start time of each node (hence, "bubbling" up nodes) by transferring it to one of the adjacent processors of the pivot processor, if the migration can improve the start time of the node. This is because after a node migrates, the space it occupies on the pivot processor is released and can be used for its successor nodes on the pivot processor. After all nodes on the pivot processor are considered, the algorithm selects the next processor in the processor list to be the new pivot processor. The process is repeated by changing the pivot processor in a breadth-first order.

5. BENCHMARK GRAPHS

In our study, we propose and use a suite of benchmark graphs consisting of five 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 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 [4] 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 Predetermined Optimal Schedules

These are *random graphs with predetermined 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 nonprecedence-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 toward a particular graph structure.

5.5. Traced Graphs

The last set of benchmark graphs, called *traced graphs* (TG), represents some of the numerical parallel application programs obtained via a parallelizing compiler [5]. We use two sets of such graphs: Cholesky factorization and mean value analysis.

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 comparisons 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

$$NSL = \frac{L}{\sum_{n_i \in CP} w(n_i)},$$

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 a lower bound may not always be possible to achieve, and the optimal schedule length may be larger than this bound.

- *Pairwise and global comparisons.* In the pairwise comparison, we measure the number of times an algorithm produced better, worse, or equal schedule length compared to each other algorithm within the same class. In the global comparison, an algorithm is collectively compared with all other algorithms in the same class.

- *Number of best solutions*. For a set of experiments, we count the number of times an algorithm produced the shortest schedule length compared to other algorithms.

- *Average degradation from the best*. When an algorithm does not produce the best schedule length within its class, we compare its degradation from the best solution. The average degradation is calculated as the average of these degradations across all such cases.

- *Number of processor 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. Although the BNP algorithms are designed for a limited number of processors, their performance depends upon this number. Thus, in order to make a fair comparison with the objective of producing a better schedule length, all BNP algorithms were first tested by providing a very large number of processors. The numbers of processors actually used were then noted.

- *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.

- *Scheduling scalability* (*SS*). This is a new performance measure defined by us and is elaborated in Section 6.6. The SS of an algorithm is a collective indicator of whether an algorithm can scale its performance well for larger problem sizes.

6.1. Results for the Peer Set Graphs

The results of applying the UNC and BNP algorithms to the PSG are shown in Table 5. 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.

Among the UNC algorithms, the DCP algorithm consistently generate the best solutions. However, there is no single BNP algorithm which outperforms 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.
- Nongreedy 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 6 and Table 7, respectively. Since optimal solutions for specific network topologies are not known, the APN algorithms were again not applied to these benchmarks. Table 6 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 7 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.

TABLE 5

Scheduling Lengths Generated by the UNC and BNP Algorithms for the PSGs

	UNC Algorithms					BNP Algorithms					
Source of task graph	LC	EZ	MD	DSC	DCP	HLFET	ISH	ETF	LAST	MCP	DLS
Ahmad & Kwok [3] (a 13-node graph)	485	717	430	404	392	454	454	473	445	454	454
Al-Maasarani [7] (a 16-node graph)	44	44	50	49	44	45	45	44	53	45	44
Al-Mouhamed [8] (a 17-node graph)	39	40	38	38	38	41	38	41	43	40	41
Shirazi *et al.* [38] (a 11-node graph)	39	32	28	30	28	28	33	28	42	33	33
Colin & Chretienne [14] (a 9-node graph)	15	14	15	14	14	14	14	14	14	14	14
Gerasoulis & Yang[19] (a 7-node graph)	25	20	22	18	18	18	18	18	18	21	18
Kruatrachue & Lewis [27] (a 15-node graph)	19	16	11	15	11	11	11	11	15	11	11
McCreary & Gill [30] (a 9-node graph)	212	159	159	160	149	180	180	180	149	180	180
Chung & Ranka [12] (a 11-node graph)	46	42	35	37	35	35	40	35	46	40	40
Wu & Gajski [42] (a 18-node graph)	420	540	420	390	390	390	390	390	470	390	390
Yang & Gerasoulis [43] (a 7-node graph)	19	20	18	16	16	19	16	16	16	16	16

TABLE 6

The Percentage Degradations from the Optimal Solutions for RGBOS (UNC Algorithms)

CCR	0.1					1.0					10.0				
Algorithms Graph size	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP
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 7

The Percentage Degradations from the Optimal Solutions for RGBOS (BNP Algorithms)

CCR	0.1						1.0						10.0					
Algorithms Graph size	HLFET	ISH	ETF	LAST	MCP	DLS	HLFET	ISH	ETF	LAST	MCP	DLS	HLFET	ISH	ETF	LAST	MCP	DLS
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 8

The Percentage Degradations from the Optimal Solutions for RGBOS (UNC Algorithms)

CCR	0.1					1.0					10.0				
Algorithms Graph size	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP	LC	EZ	MD	DSC	DCP
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 9

The Percentage Degradations from the Optimal Solutions for RGBOS (BNP Algorithms)

CCR	0.1						1.0						10.0					
Algorithms Graph size	HLFET	ISH	ETF	LAST	MCP	DLS	HLFET	ISH	ETF	LAST	MCP	DLS	HLFET	ISH	ETF	LAST	MCP	DLS
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
30	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

We summarize our observations from Table 6 and Table 7 as follows:

- Greedy BNP algorithms have shown higher capability in generating optimal solutions than the nongreedy 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 8 and Table 9, 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 8 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 9 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 8 and Table 9 lead to similar conclusions as those made in Section 6.2.

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 Fig. 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

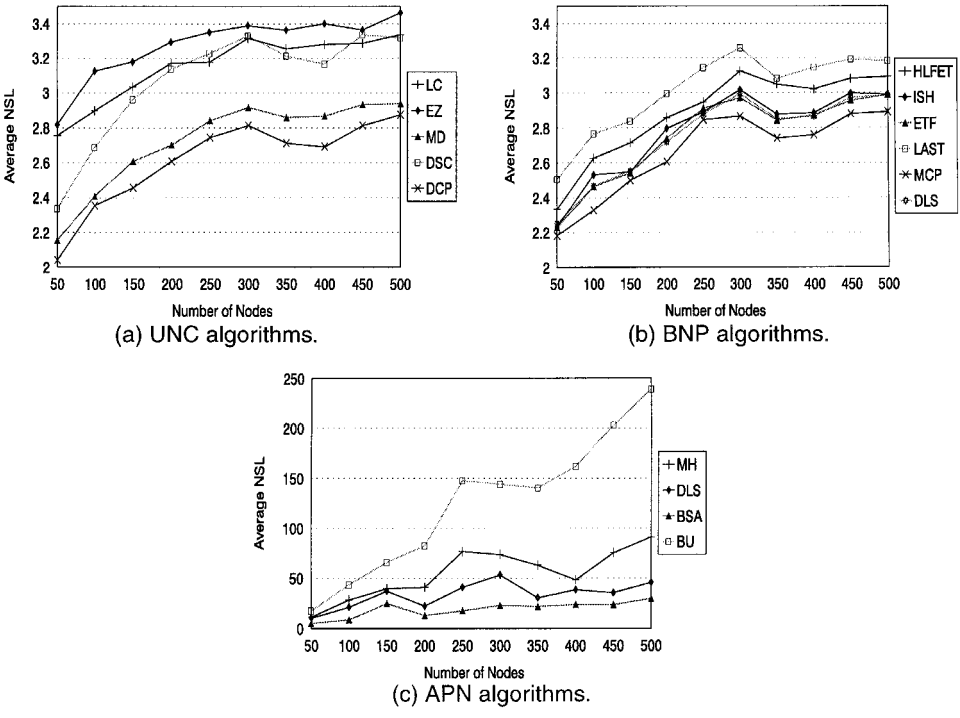


FIG. 2. Average NSL of the UNC, BNP, and APN algorithms for RGNOS benchmarks.

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 Fig. 12). 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 Fig. 2c. 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 eight

processors.[3] 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 ache MSLs 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.

We also measured the average NSLs of the algorithms against CCRs and parallelisms. As we found that parallelism does not have a significant effect on the NSLs, we have not included those results here. The average NSLs plotted against CCRs, however, are included and shown in Fig. 3. We can observe that the average NSLs increase slightly when CCR increases from small to medium range. The increase in average NSLs becomes more palpable when CCR is large (10.0). This indicates that when the edge-weights are large, the algorithms can make more mistakes.

6.4.2. Pairwise Comparison

Next, we present a pair-wise and a global comparison among all the algorithms by observing the number of times each algorithm performs better, worse, or the same, compared to every other algorithm in 250 test cases. This comparison for the UNC scheduling algorithms is given in a graphical form shown in Fig. 4. Here, each box compares two algorithms: the algorithm on the left side and the algorithm on the top. Each box contains three numbers preceded by "$>$," "$<$," and "$=$" signs, indicating the number of times the algorithm on the left performs better, worse, and the same, respectively, compared to the algorithm shown on the top. For example, the DCP algorithm performs better than the DSC algorithm in 220 cases, worse in 6 cases, and the same in 24 cases. For the global comparison, an additional box ("ALL") for each algorithm compares that algorithm with all other algorithms combined. These results clearly indicate that the DCP algorithm is better than all other algorithms. The DCP algorithm's performance is followed by that of the MD algorithm. Both DCP and MD outperform EZ and LC by a large margin while DSC is marginally better than LC. Based on these results, we rank these UNC algorithms in the following order: DCP, MD, DSC, LC, and EZ. Interestingly, this ranking is the same as using the NSLs shown in Fig. 2.

The pair-wise and global comparison of BNP scheduling algorithms is depicted in Fig. 5. Based on these results, we rank these BNP algorithms in the following order: MCP, ISH, DLS, HLFET, ETF, and LAST. This ranking essentially indicates the quality of scheduling based on how often an algorithm performs better than the others.

[3] 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.

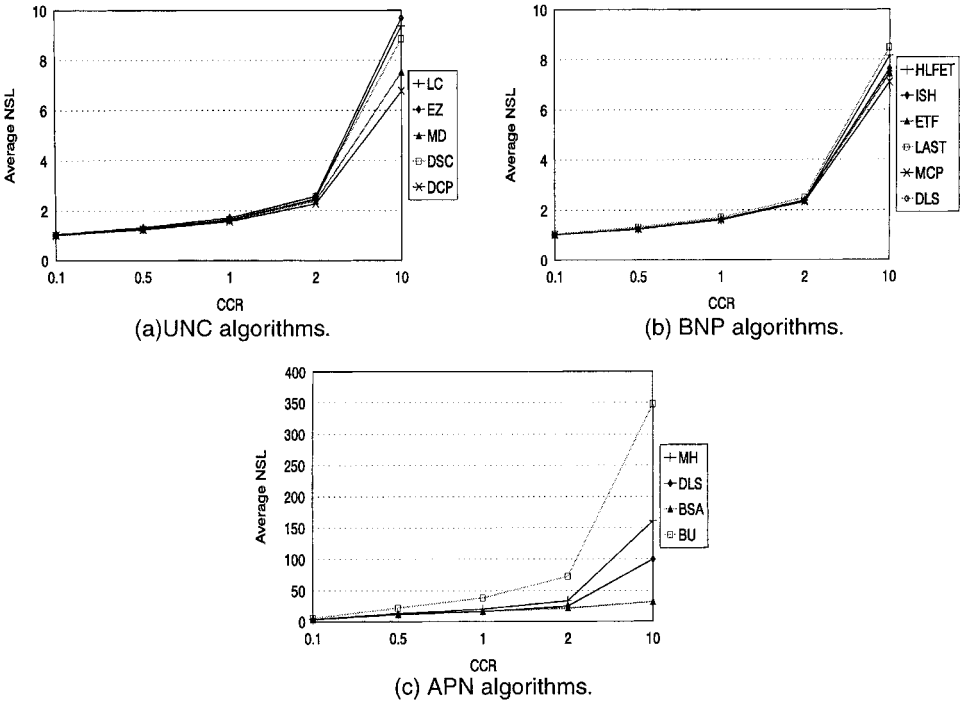


FIG. 3. Average NSL vs CCR.

A comparison between the BNP and UNC algorithms is also carried out, as shown in Fig. 6. Here each BNP algorithm is compared with each UNC algorithm. Similarly, each UNC algorithm is compared with each BNP algorithm. From this comparison, we can make a number of interesting observations. Contrary to the intuitive assumption, not all UNC algorithms are better than the BNP algorithms—only DCP outperforms each BNP algorithm. On the other hand, the HLFET, ISH, MCP, and DLS algorithms outperform each UNC algorithm, except DCP. No BNP algorithm is outperformed by all UNC algorithms.

In the pair-wise comparison of the APN scheduling algorithms shown in Fig. 7. BSA outperforms the other three algorithms in a large number of cases while DLS

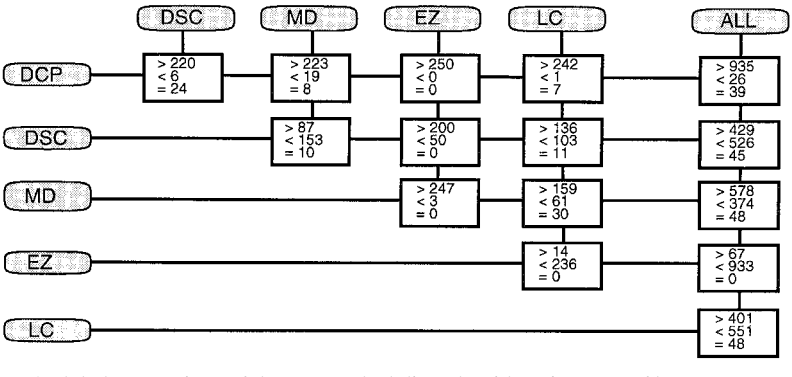


FIG. 4. A global comparison of the UNC scheduling algorithms in terms of better, worse, and equal performance for the RGNOS benchmarks.

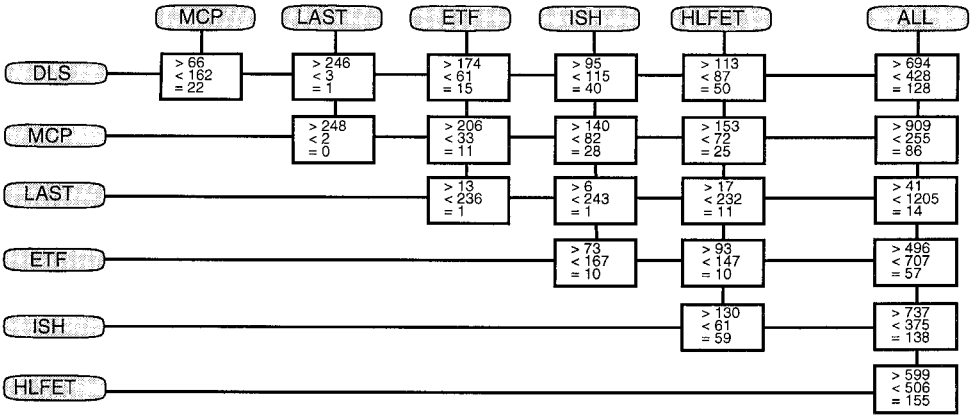


FIG. 5. A global comparison of the BNP scheduling algorithms in terms of better, worse, and equal performance for the RGNOS benchmarks.

performs better than MH. The BU algorithm is outperformed by all other algorithms. In terms of performance, these algorithms can be ranked in the order: BSA, DLS, MH, and BU.

6.4.3. Best Solutions and Degradations from the Best

The Kiviat graphs depicted in Fig. 8, Fig. 9, and Fig. 10 show the number of times each algorithm yielded the best solution out of 250 test cases (for the ANP scheduling algorithms, there are 1000 test cases). If the shaded area in a Kiviat graph clusters closely around the 0% axis, the algorithm generates optimal solutions for most of the cases. In the category of the UNC scheduling algorithms, the DCP algorithm generates the best solution in about 90% of the cases. For the BNP scheduling algorithms, the MCP algorithm generates the best solution for 141

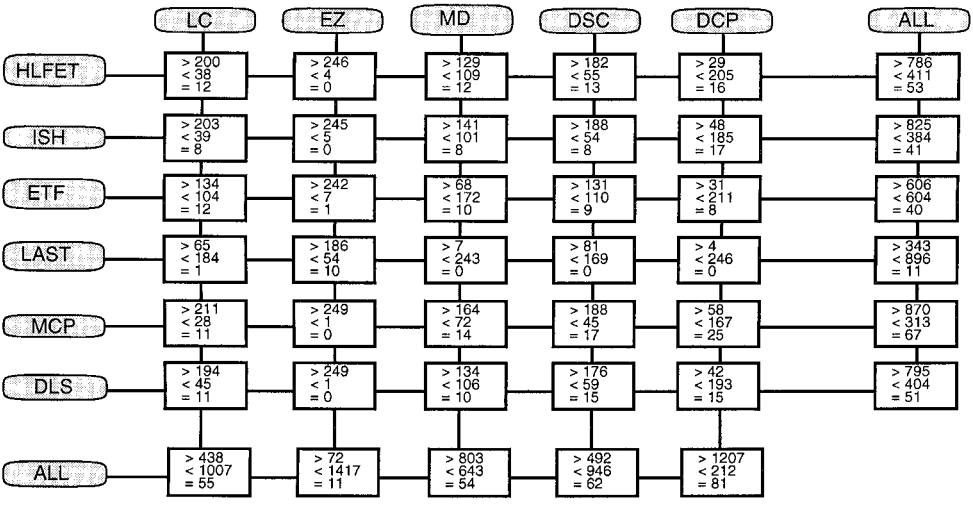


FIG. 6. A global comparison of the BNP scheduling algorithms versus the UNC scheduling algorithms in terms of better, worse, and equal performance for the RGNOS benchmarks.

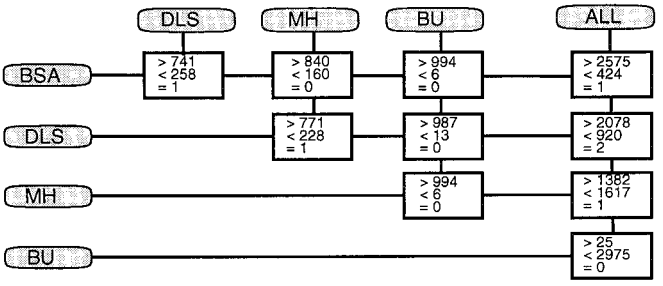


FIG. 7. A comparison of the APN scheduling algorithms in terms of better, worse, and equal performance across all topologies for the RGNOS benchmarks.

times—which is more than 50% of the number of test cases. Similarly, in the category of the ANP algorithms, the BSA algorithm generates the best solution in about 60% of the cases. The LAST, EZ, and BU algorithms do not generate the best solution in a single case.

Next, we compare the average degradation in the schedule length with respect to the best case. For each of the 250 test cases, we compare the schedule lengths produced by all of the algorithms in the same class and take the shortest schedule length as the "pseudo-optimal." We then take the percentage difference between a longer schedule length and the "pseudo-optimal." The average percentage degradation is taken as the average of all cases when an algorithm produces a schedule length longer than "pseudo-optimal." An algorithm may perform better than other algorithm in a large number of cases and also generate a good average NSL, but it may generate extremely poor schedules in some cases. This measure essentially indicates how poor an algorithm's performance is.

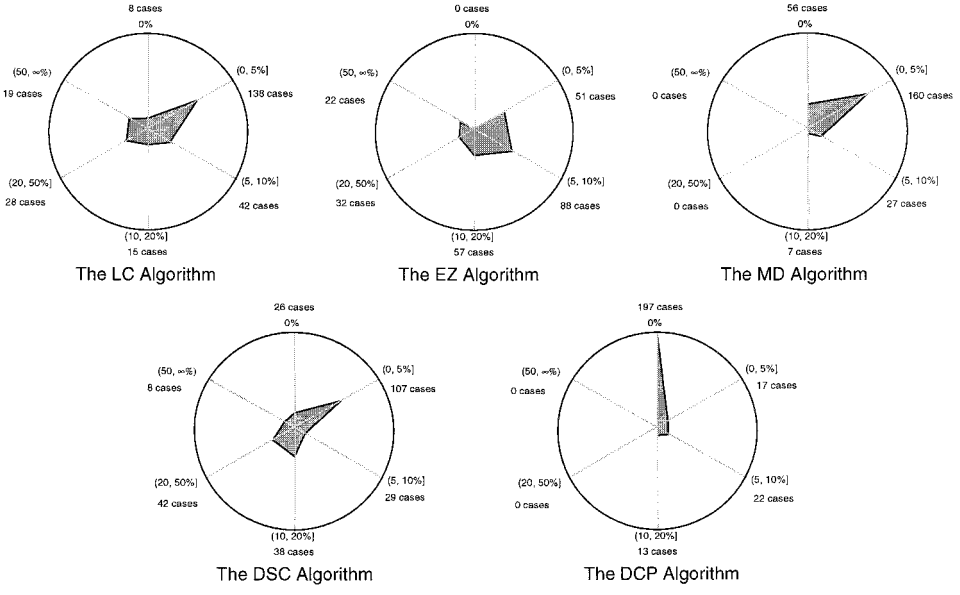


FIG. 8. The number of cases with percentage degradation from the best solutions for RGNOS (UNC algorithms).

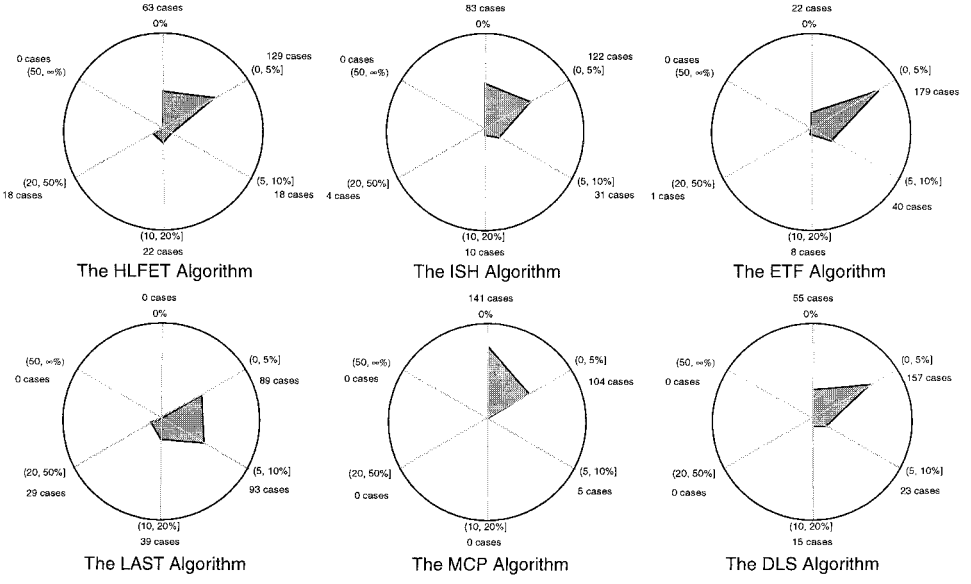


FIG. 9. The number of cases with percentage degradation from the best solutions for RGNOS (BNP algorithms).

Figure 11 depicts the degradation from the best solution for all three classes of the algorithms. For the UNC algorithms, the percentage degradation is very small for the DCP algorithm—in the graph it is visible only when the graph size is 50, 100, or 250. The MD and LC algorithms perform closely and their degradations are usually between 2% to 4%. On the other hand, the percentage degradations for the EZ and LC algorithms are similar.

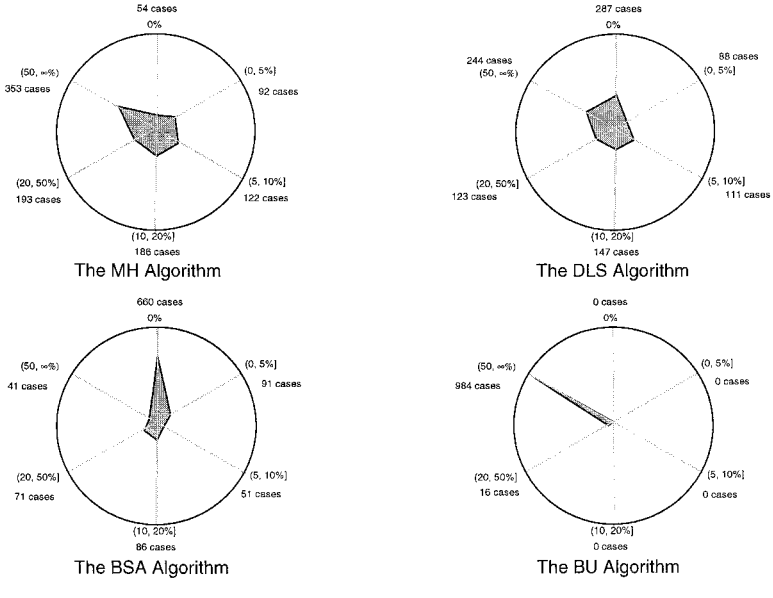


FIG. 10. The number of cases with various ranges of percentage degradation from the best solutions for the RGNOS benchmarks (APN algorithms).

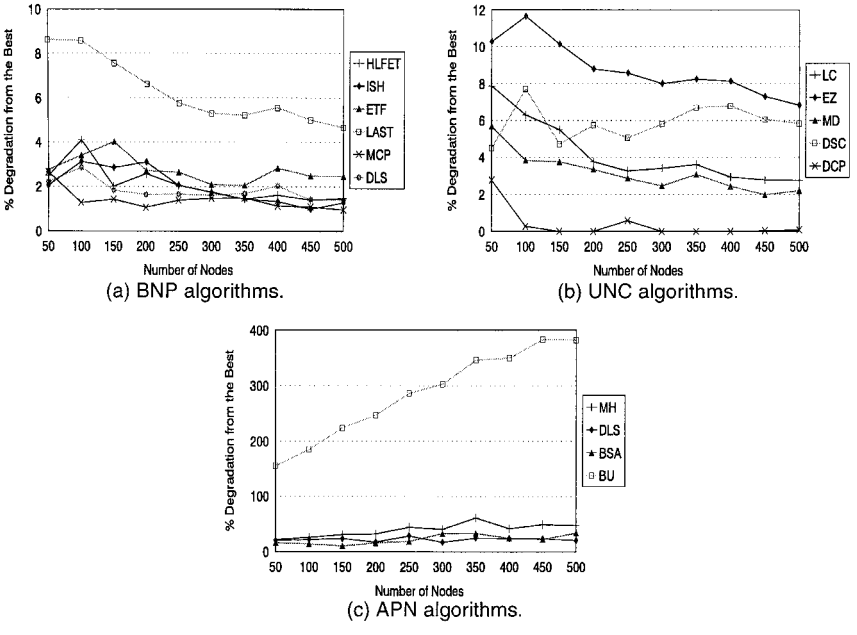


FIG. 11. The average percentage degradation from the best solutions for the RGNOS benchmarks.

We observe that, with the exception of LAST, the percentage degradations for other BNP algorithms are quite close. We can also see that while there is a great difference between the performance of the MCP and ETF algorithms shown earlier in Fig. 9, the difference between their percentage degradations are merely about 2 to 3%. This indicates that there can be large variation in the performance of the MCP algorithm. Similarly, the ISH algorithm sometimes has a large degradation compared to DLS and HLFET, although it outperforms them by a large margin when comparing the number of best solutions.

In the case of the APN scheduling algorithms, a large variation shown in Fig.11c can be noticed. The MH, DLS, and BSA algorithms generally yield less than 40% variations but the BU algorithm incurs a considerably large variation which indicates that it can generate extremely long schedules.

6.4.4. 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, we tested them with a very large number (virtually unlimited number) of processors; we then noted the number of processors actually used.

Figure 12b 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.

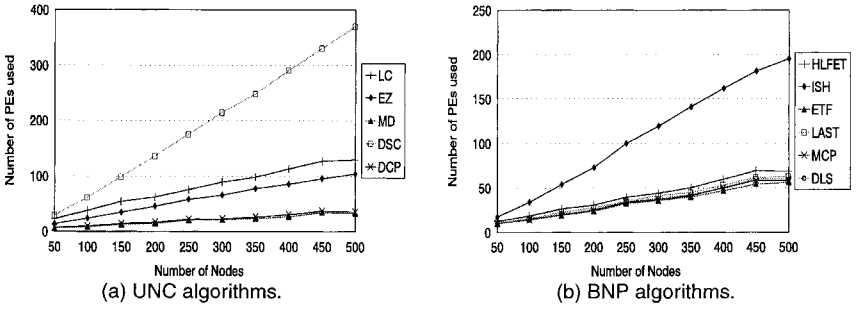


FIG. 12. The average number of processors used for the RGNOS benchmarks.

Figure 12a 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.

6.4.5. Algorithm Running Times

In this section, we compare the running times of all the algorithms. Table 10 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 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 10, 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.

6.5. Results for Traced Graphs

For the two sets of *traced graphs* (TG) representing two parallel numerical applications, Cholesky factorization and mean value analysis, the results are shown

TABLE 10

Average Running Times (in Seconds) for All 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

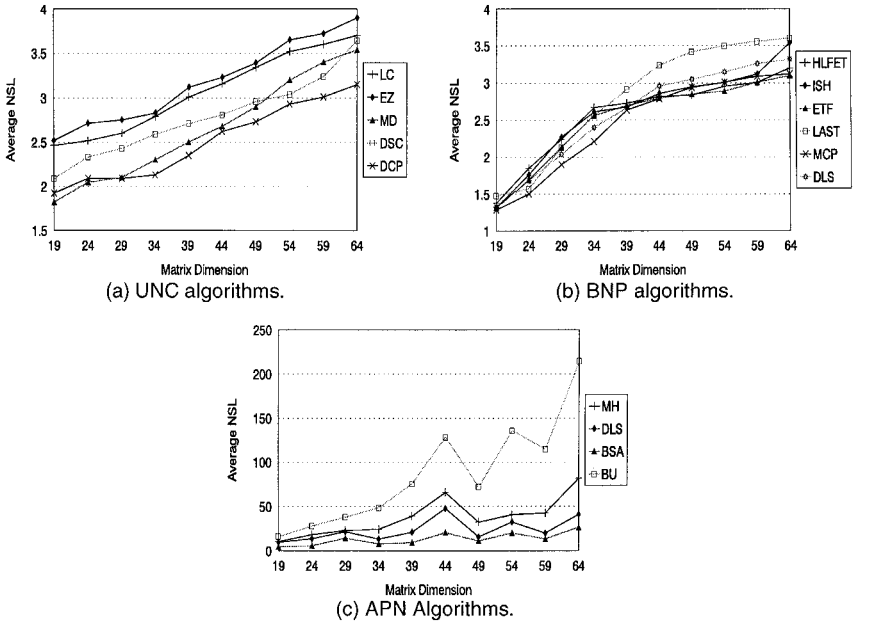


FIG. 13. Average NSL for Cholesky factorization task graphs.

in Figs. 13 and 14, respectively. Since these applications operate on matrices, the graph sizes depend on the matrix dimensions. For a matrix dimension of N, the graph size is $O(N^2)$. We varied matrix dimensions from 19 to 64 with increments of 5 and, consequently, the graph sizes varied from about 200 to 1600.

We note that for both applications, the performance of the BNP algorithms is quite similar with the exception that LAST performs much worse. By contrast, the

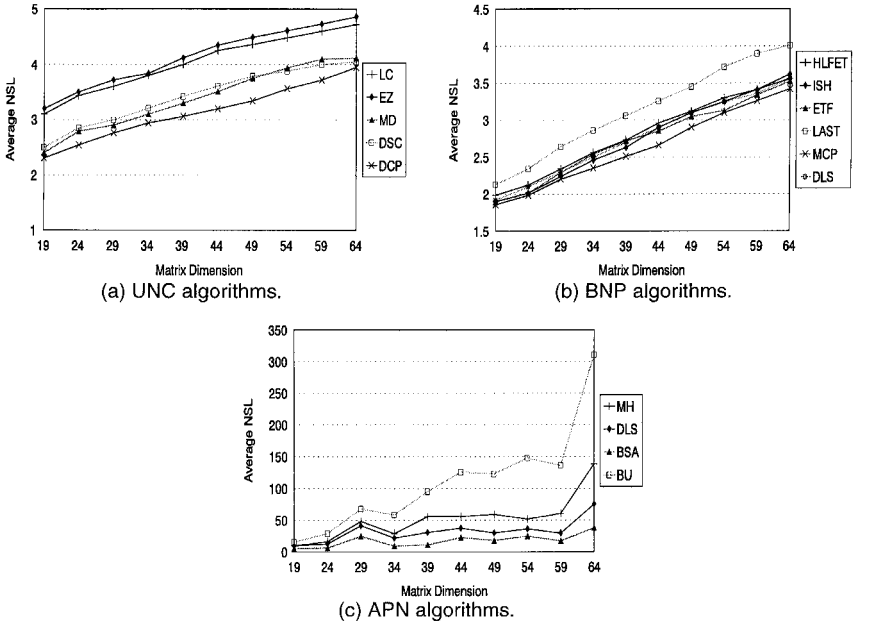


FIG. 14. Average NSL for mean value analysis task graphs.

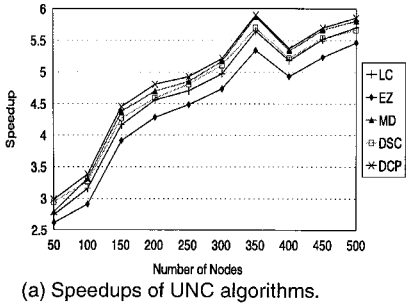
performance of the UNC algorithms is very diverse. The relative performance of the APN algorithms is quite similar for both applications.

6.6. Scheduling Scalability (SS)

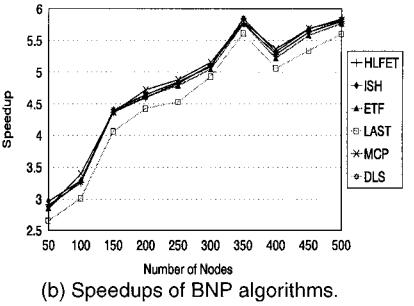
The objective of scheduling scalability (SS) is to capture the combined effectiveness of a DSA in terms of its solution quality, the number of processors used, and its running time in finding the solution. The SS of a DSA is defined as

$$SS = \left(\frac{\sum_{i=1}^{v} w(n_i)}{L}\right) \cdot \left(\frac{v}{p}\right) \cdot \left(\frac{\log v}{\log t}\right),$$

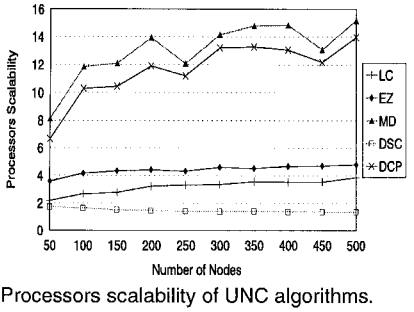
where L is the schedule length and t is the running time of the scheduling algorithm.



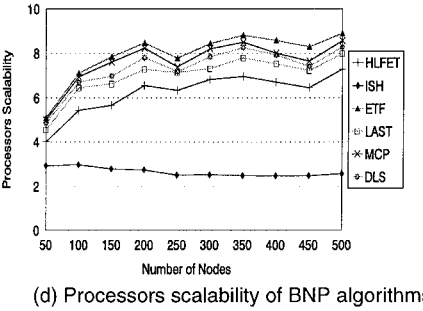
(a) Speedups of UNC algorithms.



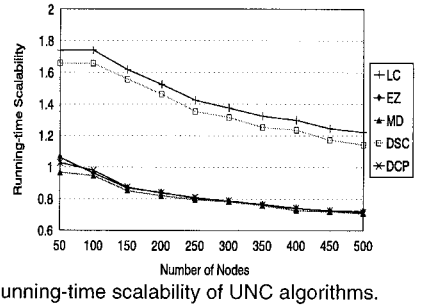
(b) Speedups of BNP algorithms.



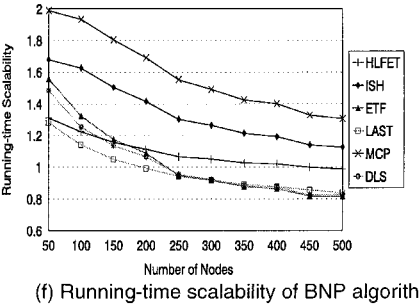
(c) Processors scalability of UNC algorithms.



(d) Processors scalability of BNP algorithms.



(e) Running-time scalability of UNC algorithms.



(f) Running-time scalability of BNP algorithms.

FIG. 15. The speedups, processor scalability, and running-time scalability of UNC and BNP algorithms.

The first factor in SS is the speedup of the schedule, which measures the quality of solution. The second factor, called the *processor scalability*, measures the effectiveness in using processors. The third factor, called the *running-time scalability*, is determined as follows: The ratio of the logarithms of the graph size and running time effectively measures the exponents of the complexity of the algorithms. For example, suppose an algorithm has a complexity $O(v^4)$; then,

$$\frac{\log v}{\log t} = \frac{\log v}{\log C_1 v^4} = \frac{\log v}{C_2 + 4 \log v} \approx \frac{1}{4} \qquad \text{for s ufficiently large } v,$$

where C_1 and C_2 are constants. Thus, using the running-time scalability, we can compare the relative running times of scheduling algorithms more accurately.

If an algorithm generates short schedules with a large number of processors and long running times, its SS will have a small value. On the other hand, the SS of a fast algorithm, which generates moderately long schedules using a small number of processors, will have a large value. To attain a high value of SS, an algorithm must simultaneously achieve the conflicting goals of good solution quality, efficient use of resources, and low time-complexity.

We first separately examine the speedups, processors scalability, and running-time scalability of the UNC and BNP algorithms for the RGNOS benchmarks (see Fig. 15). As can be seen, the speedups of the all the algorithms yield similar patterns. The speedups increase with the graph sizes because a larger graph has a higher parallelism. The processors scalabilities, on the other hand, are very diverse. Among the UNC algorithms, MD and DCP perform dramatically better than the others because their strategies in using processors are more economical. The DSC and LC algorithms use $O(v)$ processors, as we see in Fig. 15. For the BNP algorithms, the processors scalabilities differ by a small margin, with the exception of ISH which also uses $O(v)$ processors. For running-time scalabilities, LC and DSC perform the best among the UNC algorithms. For the BNP algorithms, MCP demonstrates considerably higher running-time scalability than the others.

The SS plots of the UNC and BNP algorithms are shown in Fig. 16. We can observe that combining the three factors, the SS of DCP is the best among all the UNC algorithms. The SS of MD is also quite high among the other UNC algorithms. The SS of DSC shows a decreasing trend with the increasing graph size,

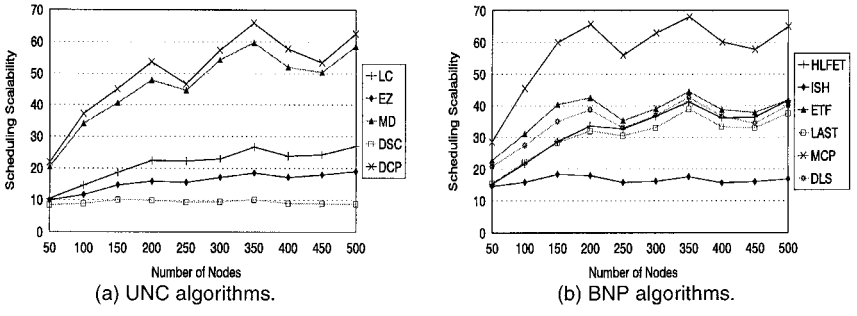


FIG. 16. Scheduling scalability.

while the values of SS of other UNC algorithms show an increasing trend. For the BNP algorithms, the SS of MCP is much higher than the others indicating that it is highly scalable. Indeed, the SS of MCP is even slightly higher than that of DCP. The other BNP algorithms, with the exception of ISH, have very similar SS values.

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 noninsertion. For example, a simple algorithm such as ISH employing insertion can yield a 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. These can be good test cases for evaluating and comparing future algorithms.

We have also proposed a performance measure called the scheduling scalability. To attain a high scheduling scalability, a DSA has to achieve the conflicting goals of good solution quality, efficient utilization of processors, and short running times. Indeed, we find that while the DCP algorithm produces much shorter schedules than the MCP algorithm, both algorithms have similar scheduling scalability because the time-complexity of the MCP algorithm is lower. To achieve a higher scheduling scalability, the time-complexity of a DSA must be reduced without compromising its solution quality.

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 programs execute [3, 29].

In UNC algorithms, clusters obtained through scheduling are assigned to a bounded number of processors. All nodes it 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 [37, 43], 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.

ACKNOWLEDGMENTS

The authors thank the anonymous referees and Professor David Lilja for their helpful comments.

REFERENCES

1. T. L. Adam, K. M. Chandy, and J. Dickson, A comparison of list scheduling for parallel processing systems, *Comm. ACM* **17**, No. 12 (Dec. 1974), 685–690.
2. I. Ahmad and Y. K. Kwok, On exploiting task duplication in parallel program scheduling, *IEEE Trans. Parallel Distrib. Systems* **9**, No. 9 (Sept. 1998), 872–892.
3. I. Ahmad and Y. K. Kwok, On parallelizing the multiprocessor scheduling problem, *IEEE Trans. Parallel Distrib. Systems* **11**, No. 4 (Apr. 1999), 414–432.
4. I. Ahmad and Y. K. Kwok, Optimal and near-optimal allocation of precedence-constrained task to parallel processors: Defying the high complexity using effective search technique, *in* "Proc. 1998 Int'l Conf. Parallel Processing," pp. 424–431, Aug. 1998.
5. I. Ahmad, Y. K. Kwok, M.-Y. Wu, and W. Shu, Automatic parallelization and scheduling of programs on multiprocessors using CASH, *in* "Proc. 1997 Int'l Conf. Parallel Processing," pp. 288–291, Aug. 1997.
6. H. H. Ali and H. El-Rewini, The time complexity of scheduling interval orders with communication is polynomial, *Parallel Process. Lett.* **3**, No. 1 (1993), 53–58.
7. 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.
8. M. A. Al-Mouhamed, Lower bound on the number of processors and time for scheduling precedence graphs with communication costs, *IEEE Trans. Software Eng.* **16**, No. 12 (Dec. 1990), 1390–1401.
9. J. Baxter and J. H. Patel, The LAST algorithm: A heuristic-based static task allocation algorithm, *in* "Proc. 1998 Int'l Conf. Parallel Processing," Vol. II, pp. 217–222, Aug. 1989.
10. T. L. Casavant and J. G. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, *IEEE Trans. Software Eng.* **14**, No. 2 (Feb. 1988), 141–154.
11. H. Chen, B. Shirazi, and J. Marquis, Performance evaluation of a novel scheduling method: Linear clustering with task duplication, *in* "Proc. Int'l Conf. Parallel and Distributed Systems," pp. 270–275, Dec. 1993.
12. Y. C. Chung and S. Ranka, Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors, *in* "Proc. Supercomputing '92," pp. 512–521, Nov. 1992.
13. E. G. Coffman and R. L. Graham, Optimal scheduling for two-processor systems, *Acta Inform.* **1** (1972), 200–213.
14. J. Y. Colin and P. Chretienne, C.P.M. scheduling with small computation delays and task duplication, *Oper. Res.* **39**, No. 4 (July–Aug. 1991), 680–684.

15. M. Cosnard and M. Loi, Automatic task graph generation techniques, *Parallel Process. Lett.* **5**, No. 4 (Dec. 1995), 527–538.

16. H. El-Rewini and T. G. Lewis, Scheduling parallel programs onto arbitrary target machines, *J. Parallel Distrib. Comput.* **9**, No. 2 (June 1990), 138–153.

17. E. B. Fernadez and B. Bussell, Bounds on the number of processors and time for multiprocessor optimal schedules, *IEEE Trans. Comput.* **C-22**, No. 8 (Aug. 1973), 745–751.

18. M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, CA, 1979.

19. A. Gerasoulis and T. Yang, A comparison of clustering heuristics for scheduling DAGs on multiprocessors, *J. Parallel Distrib. Comput.* **16**, No. 4 (Dec. 1992), 276–291.

20. T. C. Hu, Parallel sequencing and assembly line problems, *Oper. Res.* **19**, No. 6 (Nov. 1961), 841–848.

21. J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, *SIAM J. Comput.* **18**, No. 2 (Apr. 1989), 244–257.

22. K. Hwang and Z. Xu, "Scalable Parallel Computing: Technology, Architecture, Programming," McGraw–Hill, New York, 1998.

23. K. Hwang, Z. Xu, and M. Arakawa, Benchmark evaluation of the IBM SP2 for parallel signal processing, *IEEE Trans. Parallel Distrib. Systems* **7**, No. 5 (May 1996), 522–536.

24. H. Kasahara and S. Narita, Practical multiprocessor scheduling algorithms for efficient parallel processing, *IEEE Trans. Comput.* **C-33**, No. 11 (Nov. 1984), 1023–1029.

25. A. A. Khan, C. L. McCreary, and M. S. Jones, A comparison of multiprocessor scheduling heuristic, *in* "Proc. 1994 Int'l Conf. Parallel Processing," Vol. II, pp. 243–250, Aug. 1994.

26. S. J. Kim, and J. C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, *in* "Proc. 1988 Int'l Conf. Parallel Processing," Vol. II, pp. 1–8, Aug. 1988.

27. B. Kruatrachue and T. G. Lewis, Duplication scheduling heuristic (DSH): A new precedence task scheduler for parallel processor systems, Technical Report, Oregon State University, Corvallis, OR 97331, 1987.

28. Y. K. Kwok, and I. Ahmad, Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors, *IEEE Trans. Parallel Distrib. Systems* **7**, No. 5 (May 1996), 506–521.

29. Y. K. Kwok, and I. Ahmad, Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm, *J. Parallel Distrib. Comput.* **47**, No. 1 (Nov. 1997), 58–77.

30. C. McCreary and H. Gill, Automatic determination of grain size for efficient parallel processing, *Comm. ACM* **32**, No. 9 (Sept. 1989), 1073–1078.

31. N. Mehdiratta and K. Ghose, A bottom-up approach to task scheduling on distibuted memory multiprocessor, *in* "Proc. 1994 Int'l Conf. Parallel Processing," Vol. II, pp. 151–154, Aug. 1994.

32. M. A. Palis, J.-C. Liou, and D. S. L. Wei, Task clustering and scheduling for distributed memory parallel architectures, *IEEE Trans. Parallel Distrib. Systems* **7**, No. 1 (Jan. 1996), 46–55.

33. C. H. Papadimitriou and M. Yannakakis, Scheduling interval-ordered tasks, *SIAM J. Comput.* **8**, No. 3 (Aug. 1979), 405–409.

34. C. H. Papadimitriou and M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms, *SIAM J. Comput.* **19**, No. 2 (Apr. 1990), 322–328.

35. G. -L. Park, B. Shirazi, and J. Marquis, DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems, *in* "Proc. 11th Int'l Parallel Processing Symposium," pp. 157–166, Apr. 1997.

36. C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, Optimal scheduling strategies in a multiprocessor system, *IEEE Trans. Comput.* **C-21**, No. 2 (Feb. 1972), 137–146.

37. V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," MIT Press, Cambridge, MA, 1989.

38. B. Shirazi, H. Chen, and J. Marquis, Comparative study of task duplication static scheduling versus clustering and non-clustering techniques, *Concurrency: Practice and Experience* **7**, No. 5 (Aug. 1995), 371–390.

39. R. Shirazi, M. Wang, and G. Pathak, Analysis and evalutation of heuristic methods for static scheduling, *J. Parallel Distrib. Comput.* **10**, No. 3 (Nov. 1990), 222–232.

40. G. C. Sih and E. A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, *IEEE Trans. Parallel Distrib. Systems* **4**, No. 2 (Feb. 1993), 75–87.

41. J. Ullman, NP-complete scheduling problems, *J. Comput. System Sci.* **10** (1975), 384–393.

42. M.-Y. Wu and D. D. Gajski, Hypercool: a programming aid for message-passing systems, *IEEE Trans. Parallel Distrib. Systems* **1**, No. 3 (July 1990), 330–343.

43. T. Yang and A. Gerasoulis, List scheduling with and without communication delays, *Parallel Comput.* **19** (1993), 1321–1344.

44. T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, *IEEE Trans. Parallel Distrib. Systems* **5**, No. 9 (Sept. 1994), 951–967.

YU-KWONG KWOK received his B.Sc. degree in computer engineering from the University of Hong Kong in 1991, the M.Phil. and Ph.D. degrees in computer science from the Hong Kong University of Science and Technology in 1994 and 1997, respectively. Currently, he is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar for one year in the parallel processing laboratory at the School of Electrical and Computer Engineering at Purdue University. His research interests include software support for parallel and distributed computing, heterogeneous cluster computing, and distributed multimedia systems. He is a member of the IEEE Computer Society and the ACM.

ISHFAQ AHMAD received a B.Sc. degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1985. He received his M.S. degree in computer engineering and Ph.D. degree in computer science, both from Syracuse University in 1987 and 1992, repectively. At present, he is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology (HKUST). His research interests are in the areas of parallel programming tools, scheduling and mapping algorithms for scaleable architectures, video compression, and interactive multimedia systems. He is director of Multimedia Technology Research Center at HKUST, where he and his colleagues are working on a number of research projects related to information technology, in particular in the areas of video coding and interactive multimedia systems in a distributed environment using high-performance computing for emerging applications. He has published over 100 technical papers in refereed journals and conferences. He has served on the program committees of numerous international conferences and has guest edited several journals. He is serving on the editorial board of IEEE Concurrency, Cluster Computing, and IEEE Transactions on Circuits and Systems for Video Technology. He is a member of the IEEE Computer Society.