Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors using A Parallel Genetic Algorithm

Yu-Kwong Kwok[†] and Ishfaq Ahmad

Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

Email: {*csricky, iahmad*}@*cs.ust.hk*

January 1997
Revised: July 1997

To appear in the Journal of Parallel and Distributed Computing
Special Issue on Parallel Evolutionary Computing

Abstract

Given a parallel program represented by a task graph, the objective of a scheduling algorithm is to minimize the overall execution time of the program by properly assigning the nodes of the graph to the processors. This multiprocessor scheduling problem is NP-complete even with simplifying assumptions, and becomes more complex under relaxed assumptions such as arbitrary precedence constraints, and arbitrary task execution and communication times. The present literature on this topic is a large repertoire of heuristics that produce good solutions in a reasonable amount of time. These heuristics, however, have restricted applicability in a practical environment because they have a number of fundamental problems including high time complexity, lack of scalability, and no performance guarantee with respect to optimal solutions. Recently, genetic algorithms (GAs) have been widely reckoned as a useful vehicle for obtaining high quality or even optimal solutions for a broad range of combinatorial optimization problems. While a few GAs for scheduling have already been suggested, in this paper we propose a novel GA-based algorithm with an objective to simultaneously meet the goals of high performance, scalability, and fast running time. The proposed Parallel Genetic Scheduling (PGS) algorithm itself is a parallel algorithm which generates high quality solutions in a short time. By encoding the scheduling list as a chromosome, the PGS algorithm can potentially generate an optimal scheduling list which in turn leads to an optimal schedule. The major strength of the PGS algorithm lies in its two efficient genetic operators: the order crossover and mutation. These operators effectively combine the building-blocks of good scheduling lists to construct better lists. The proposed algorithm is evaluated through a robust comparison with two heuristics best known in terms of performance and time complexity. It outperforms both heuristics while taking considerably less running time. When evaluated with random task graphs for which optimal solutions are known, the PGS algorithm generates optimal solutions for more than half of the test cases and close-to-optimal for the other half.

---

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

1 Introduction

To effectively harness the computing power of high-performance parallel computing systems, it is crucial to employ a judicious scheduling algorithm for proper allocation and sequencing of tasks on the processors. Given a parallel program modeled by a node- and edge-weighted *directed acyclic graph* (DAG), finding an optimal schedule with a minimum turnaround time without violating precedence constraints among the tasks is well known to be an NP-complete problem [10], [13], [39]. Only for a few simplified cases [4], [10], [13], [29], [32], the problem can be solved by a polynomial-time algorithm. If the simplifying assumptions of these cases are relaxed, the problem becomes NP-hard in the strong sense. Thus, it is unlikely that the general scheduling problem can be solved in a polynomial-time, unless $P = NP$. Therefore, state-space search techniques are considered as the only resort for finding optimal solutions [8], [9]. However, most of these techniques are still designed to work under restricted environments and are usually not applicable to general situations. Furthermore, a state-space search incurs an exponential time in the worst case. As such, with these techniques even moderately larger problems cannot be solved in a reasonable amount of time.

Due to the intractability of the scheduling problem and the ineffectiveness of state-space search techniques, many polynomial-time heuristics are reported to tackle the problem under more pragmatic situations [1], [3], [14], [26]. The rationale of these heuristics is to sacrifice optimality for the sake of reduced time complexity. While these heuristics are shown to be effective in experimental studies, they usually cannot generate optimal solutions, and there is no guarantee in their performance in general. The major weakness of these heuristics is that they usually employ a deterministic greedy strategy which can be sensitive to the scheduling parameters such as the number of target processors available and the structure of the input task graphs. Improving the performance of a heuristic generally increases its complexity. Furthermore, these heuristics are usually not scalable in terms of their running time and solution quality for large problem sizes.

In view of the drawbacks of the existing sequential scheduling heuristics, we aim at designing a new scheduling scheme which has a high capability to generate optimal solutions and is also fast and scalable. To obtain high quality solutions, we devise a genetic formulation of the scheduling problem, in which scheduling lists (ordering of tasks for scheduling) are systematically combined by using computationally efficient operators so as to determine an optimal scheduling list. To achieve a reduced time complexity, the proposed algorithm is parallelized. The algorithm not only scales well with the number of processors but can also handle general DAGs without making simplifying assumptions.

Inspired by the Darwinian concept of evolution, genetic algorithms [6], [11], [16], [18], [36], [40] are global search techniques which explore different regions of the search space simultaneously by keeping track of a set of potential solutions called a population. According to the *Building-block Hypothesis* [16] and the *Schema Theorem* [16], a genetic algorithm systematically combines the good building-blocks of some selected individuals in the population to generate

better individuals for survival in a new generation through employing genetic operators such as crossover and mutation. Another attractive merit of genetic search is that the parallelization of the algorithm is possible. With these distinctive algorithmic merits, genetic algorithms are becoming more widely used in many areas to tackle the quest for optimal solutions in optimization problems. Indeed, genetic algorithms have been applied to the data partitioning problem [7], the graph partitioning problem [2], the robotic control problem [15], the standard cell placement problem [33], etc.

We formulate the scheduling problem in a genetic search framework based on the observation that if the tasks of a parallel program are arranged properly in a list, an optimal schedule may be obtained by scheduling the tasks one by one according to their order in the list. With this concept, we encode each chromosome to be a valid scheduling list, one in which the precedence constraints among tasks are preserved. We also design two genetic operators: the order crossover and mutation. These operators effectively combine the good features of existing scheduling lists to form better lists. Using random task graphs for which optimal schedules are known, we have found that the proposed algorithm can generate optimal solutions for a majority of the test cases. Furthermore, when compared with two efficient scheduling heuristics, the proposed algorithm outperforms them while taking much less computation time due to its effective parallelization.

The remainder of the paper is organized as follow. In the next section we provide the problem statement. In Section 3 we give a background of genetic search by presenting a brief survey of genetic techniques. In Section 4 we present the proposed parallel genetic scheduling algorithm. Examples are used to illustrate the functionality of the proposed technique. In Section 5 we describe our experimental study and its results. We also describe some related work on using genetic algorithms for scheduling in Section 6. Finally, we provide some concluding remarks and future research directions in the last section.

2 Problem Statement

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

The source node of an edge incident on a node is called a *parent* of that node. Likewise, the

destination node emerged from a node is called a *child* of that node. A node with no parent is called an *entry* node and a node with no child is called an *exit* node. The precedence constraints of a DAG dictate that a node cannot start execution before it gathers all of the messages from its parent nodes. The communication cost among two nodes assigned to the same processor is assumed to be zero. Thus, the *data available time* (DAT) of a node depends heavily on the processor to which the node is scheduled. If node n_i is scheduled, $ST(n_i)$ and $FT(n_i)$ denote the start-time and finish-time of n_i, respectively. After all nodes have been scheduled, the *schedule length* is defined as $max_i\{FT(n_i)\}$ across all nodes. The objective of scheduling is to minimize the schedule length by proper allocation of the nodes to the processors and arrangement of execution sequencing of the nodes without violating the precedence constraints. We summarize in Table 1 the notations used in the paper. An example DAG, shown in Figure 1(a), will be used as an example in the subsequent discussion.

Table 1: Definitions of some notations.

Symbol	Definition
n_i	Node number of a node in the parallel program task graph
$w(n_i)$	Computation cost of node n_i
$c(n_i, n_j)$	Communication cost of the directed edge from node n_i to n_j
v	Number of nodes in the task graph
e	Number of edges in the task graph
p	Number of processing elements (PEs) in the target system
b-level	Bottom level of a node
t-level	Top level of a node
ALAP	As Late As Possible start-time of a node
DAT	Data available time of a node on a particular PE
$ST(n_i)$	Start-time of node n_i
$FT(n_i)$	Finish-time of node n_i
CCR	Communication-to-computation Ratio
SL	Schedule Length
PPE	Physical Processing Elements (on which the PGS algorithm is executed)
μ_c	Crossover Rate
μ_m	Mutation Rate
N_p	Population Size
N_g	Number of Generations
f	Fitness value of a chromosome

The processing elements (PEs) in the target system may be heterogeneous or homogeneous. Heterogeneity of PEs means that the PEs have different speeds or processing capabilities; we assume the communication links are homogeneous. However, we assume every module of a parallel program can be executed on any PE though the computation time needed on different PEs may be different. The PEs are connected by an interconnection network based on a certain topology. The topology may be fully-connected or of a particular structure such as a hypercube or mesh. That is, a message is transmitted with the same speed on all links. Using this model, a multiprocessor network can be represented by an undirected graph. An example processor

graph is shown in Figure 1(b).

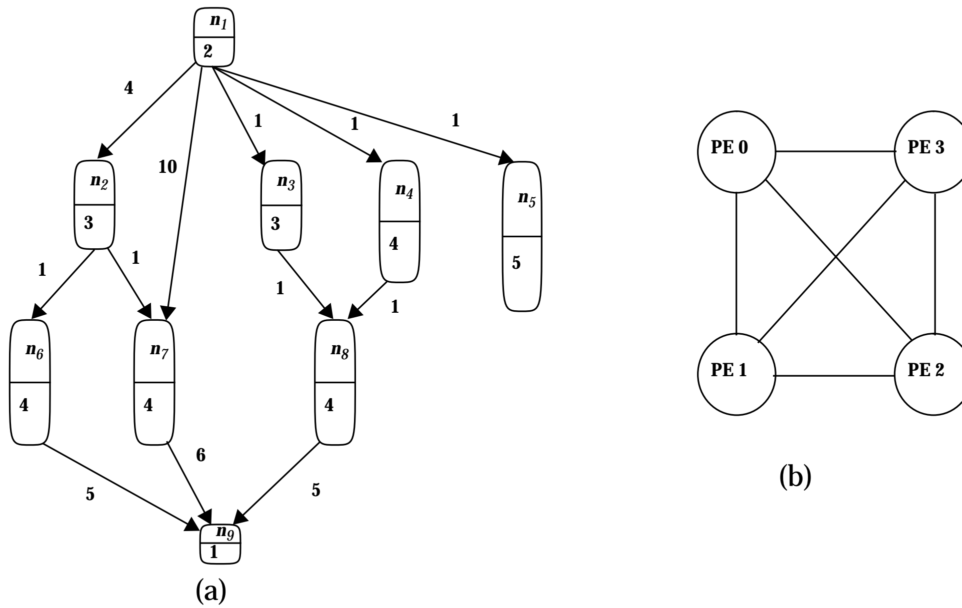


Figure 1: (a) An example DAG; (b) A 4-processor fully-connected target system.

The problem of optimally scheduling a DAG in a polynomial-time has been solved for only three simple cases. The first case is to schedule a free-tree with uniform node weights to arbitrary number of processors. Hu [20] suggested a linear-time algorithm to solve the problem. The second case is to schedule an arbitrarily structured DAG with uniform node weights to two processors. Coffman and Graham [10] devised a quadratic time algorithm for solving this problem. The third case is to schedule an *interval-order*ed DAG with uniform node weights to arbitrary number of processors. Papadimitriou and Yannakakis [29] designed a linear time algorithm to tackle the problem. In all these cases, communication among the tasks is ignored. Recently, Ali and El-Rewini [4], [12] have shown that interval-ordered DAG with uniform edge weights, which are equal to the node weights, can also be optimally scheduled in polynomial-time. Ullman [39] and Garey *et al.* [13] have shown that for more general cases, the scheduling problem is NP-complete.

3 Overview of Genetic Search Techniques

In this section we present a brief review of standard genetic algorithms (SGA). This will be followed by a discussion of different models of parallel genetic algorithms (PGA).

3.1 Standard Genetic Algorithms

Genetic algorithms (GAs), introduced by Holland in the 1970's [18], are search techniques that are designed based on the concept of evolution [6], [11], [16], [36]. In simple terms, given a well-defined search space in which each point is represented by a bit string, called a *chromosome*, a GA is applied with its three genetic search operators—*selection*, *crossover*, and *mutation*—to

transform a population of chromosomes with the objective of improving the quality of the chromosomes. A GA is usually employed to determine the optimal solution of a specific objective function. The search space, therefore, is defined as the solution space so that each feasible solution is represented by a distinct chromosome. Before the search starts, a set of chromosomes is randomly chosen from the search space to form the initial population. The three genetic search operations are then applied one after the other to obtain a new generation of chromosomes in which the expected quality over all the chromosomes is better than that of the previous generation. This process is repeated until the stopping criterion is met and the best chromosome of the last generation is reported as the final solution. An outline of a generic GA is as follows. The detailed mechanism of the three operators will be discussed in detail afterwards.

Standard Genetic Algorithm (SGA):

(1) Generate initial population;
(2) **while** number of generations not exhausted **do**
(3)   **for** *i* = 1 **to** PopulationSize **do**
(4)   Randomly select two chromosomes and apply the crossover operator;
(5)   Randomly select one chromosome and apply mutation operator;
(6)   **endfor**
(7)   Evaluate all the chromosomes in the population and perform selection;
(8) **endwhile**
(9) Report the best chromosome as the final solution.

In nature we observe that stronger individuals survive, reproduce, and hence transmit their good characteristics to subsequent generations. This natural selection process inspires the design of the selection operator in the GAs. Given a generation of chromosomes, each of them is evaluated by measuring its fitness which is, in fact, the quality of the solution the chromosome represents. The fitness value is usually normalized to a real number between 0 and 1, and the higher the value, the fitter the chromosome. Usually a proportionate selection scheme is used. With this scheme, a chromosome with a fitness value f is allocated f / f_{avg} offspring in the subsequent generation, where f_{avg} is the average fitness value of the population. Thus, a chromosome with a larger fitness value is allocated more offsprings while a chromosome with a smaller fitness value, for example, less than the average, may be discarded in the next generation.

GAs are least affected by the continuity properties of the search space unlike many other heuristic approach. For instance, many researchers have found that GAs are better than simulated annealing [16] as well as tabu search [11], both of which operate on one single solution only.

For an efficient GA search, in addition to a proper solution structure, it is necessary that the initial population of solutions be a diverse representative of the search space. Furthermore, the solution encoding should permit:

- a large diversity in a small population;

- easy crossover and mutation operations; and
- an easy computation of the objective function.

3.2 Genetic Search Operators

In this section we review the mechanism and characteristics of two important standard genetic operators: crossover and mutation. A less commonly used operator, called inversion, is also discussed.

Crossover is a crucial operator of GAs and is applied after selection. While selection is used to improve the overall quality of the population, crossover is used to explore the search space to find better individuals. Pairs of chromosomes are selected randomly from the population for application of the crossover operator. In the simplest approach, a point is chosen randomly as the crossover point. The two chromosomes then exchange the portions beyond the crossover point to generate two new chromosomes. A simple example of the standard crossover is given in Figure 2(a). The rationale is that after the exchange the newly generated chromosomes may contain the good characteristics from both the parent chromosomes and hence, possess a higher fitness value. Nevertheless the newly generated chromosomes may be worse than their parents. With respect to this, the crossover operator is not always applied to the selected pair of chromosomes. It is applied with a certain pre-specified probability called the crossover rate, denoted by μ_c. There are a number of variants of standard crossover operators. These include

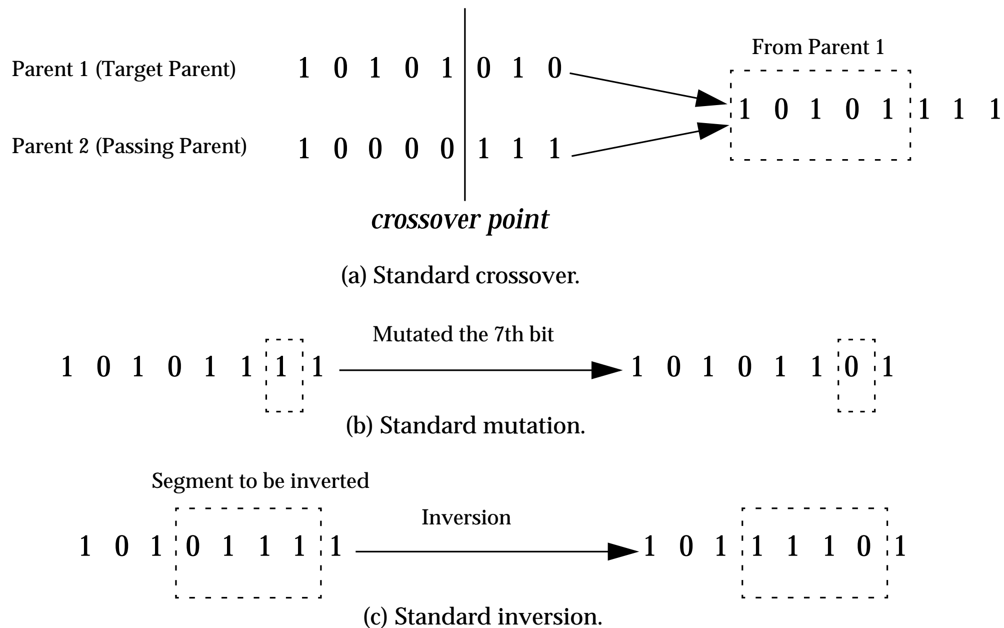


Figure 2: Examples of the standard (a) crossover operator; (b) mutation operator; and (c) inversion operator on a binary coded chromosome.

the *order crossover*, *partially-mapped crossover* (PMX), and *cycle crossover* [16], [33]. The characteristics of these variants will be described later in the paper.

Mutation is a genetic operator for recovering the good characteristics lost during crossover.

Mutation of a chromosome is achieved by simply flipping a randomly selected bit of the chromosome. A simple example of a standard mutation is given in Figure 2(b). Like a crossover, mutation is applied with a certain probability called the mutation rate which denoted by μ_m. Although mutation is a secondary search operator, it is useful for escaping from the local minima. For instance, suppose all the chromosomes have converged to 0 at a certain bit position while the optimal solution has a 1 at that position. Crossover cannot regenerate a 1 at that position but mutation may be able to. If crossover is the only operator used, then as new patterns evolve out of the old ones, there is invariably a loss in pattern diversity and hence the breadth of the search domain.

An inversion operator takes a random segment in a chromosome and reverses it. A simple example of the standard inversion is given in Figure 2(c). The advantage of the inversion operator is as follows. There are some groups of properties, or genes which would be advantageous for offsprings to inherit together from one parent. Such groups of genes, which interact to increase the fitness of the offspring which inherits them, are said to be *co-adapted*. If two genes are close to each other in the chromosome, the probability of being split up, when the crossover operator divides the chromosome into two segments, will be less.

3.3 Control Parameters

A GA is governed by a number of parameters: population size N_p, number of generations N_g, crossover rate μ_c, and mutation rate μ_m. Finding appropriate values for these parameters requires extensive experimentations [6], [16], [36]. Even with appropriate parameters, optimal solutions cannot be guaranteed due to the probabilistic nature of GAs. Grefenstette [17] proposed using the *scaling* method for preventing a *premature convergence*, a scenario in which chromosomes of a population become homogeneous and converge to a sub-optimal chromosome. Scaling involves re-adjusting the fitness values of solutions in order to sustain a steady selective pressure in the population so that a premature convergence may be avoided. To tune the control parameters *on-the-fly*, Srinivas and Patnaik [35] proposed an adaptive method which is driven by the idea of sustaining diversity in a population without affecting its convergence properties. Their algorithm [35] protects the best solutions in each generation from being disrupted by crossover and mutation. Extensive experiments have shown that this adaptive strategy can help prevent GA's getting stuck at a local minimum.

3.4 Parallel Genetic Algorithms

The inherent parallelism in GAs can be exploited to enhance their search efficiency. In contrast to simulated annealing which is intrinsically sequential and thus hard to parallelize [28], parallelism in GAs is easier to exploit [11], [16], [22], [27]. One of the approaches of parallelizing a GA is to divide the population into q partitions, where q is the number of *physical processing elements* (PPEs) on which the parallel GA is executed[†]. Each subpopulation in a PPE contains more than one chromosome. Each PPE then runs a separate copy of the original GA with its own

†. A PPE should be distinguished from a PE of the target system to which a DAG is scheduled.

partition of the population. Such a parallel GA (PGA) is called the coarse-grained PGA. Two less common types of PGAs are the fine-grained PGA and the micro-grained PGA. In a fine-grained PGA exactly one chromosome is assigned to each PPE. In this type of PGA, it is the topology of the PPE network that determines the degree of population isolation, and hence diversity, of the individuals in the whole population. In a micro-grained PGA, only a single population is maintained, and parallelism comes from the use of multiple PPEs to evaluate the individual fitness function. We employ the coarse-grained parallelization approach.

In a coarse-grained PGA, there are many possible schemes for the PPEs to communicate and exchange information about solutions [22], [27], [30]. The objective of communication is to transfer fitter chromosomes from their local populations to other PPEs for parallel exploitation. There are two major models of communication: the *isolated island* model and the *connected island* model.

In an isolated island model, PPEs work independently and communicate only at the end of the whole search process to select the best solution among all the PPEs. No migration of fitter chromosomes is performed. Obviously linear speedup can be achieved. Nevertheless, a PPE may waste all its processing cycles when it gets stuck at a poor population without the knowledge that other PPEs are searching in more promising regions of the search space. This in turn is a result of partitioning the population, which reduces the diversity of the chromosomes in the search space.

In a connected island model, PPEs communicate periodically to exchange the information about their solutions found thus far. It is common for the best chromosome found to be broadcast to all PPEs so that every PPE may devote the processing power towards the most promising direction. There are two variants of the connected island model. The first is that PPEs communicate in a synchronous fashion, that is, all PPEs participate in a communication phase simultaneously. While this scheme may be easy to implement, its drawback is that the communication cost paid for by the information exchange can be a significant overhead, limiting the achievable speedup. The second variant is that PPEs communicate in an *event-driven* manner, that is, PPEs communicate only when necessary. Also, not all PPEs participate in a communication phase. The merit of this scheme is that the communication overhead is lower. However, a drawback is that this scheme is more difficult to implement. Thus, most PGAs employ the synchronous connected island model.

Finally, some researchers have found that in a PGA, PPEs can use different sets of control parameters to further increase the population diversity [37], [38]. This can help avoiding premature convergence.

4 The Proposed Parallel Genetic Algorithm for Scheduling

In this section we describe the proposed parallel genetic scheduling algorithm. We first present a scrutiny of the list scheduling method, which is necessary for the proposed genetic formulation for the scheduling problem. We then proceed to describe the chromosome encoding

scheme, the design of the genetic operators, and the selection of the control parameters, and finally, the parallelization of the algorithm.

4.1 A Scrutiny of List Scheduling

Classical optimal scheduling algorithms, like Hu's algorithm [20] and Coffman *et al.* 's algorithm [10], are based on the list scheduling approach in which the nodes of the DAG are first arranged as a list such that the ordering of the nodes in the list preserves the precedence constraints. In the second step, beginning from the first node in the list, each node is removed and scheduled to a PE that allows an earliest start-time. Hereafter we refer to this second step as the *start-time minimization* step, which is outlined as follows.

Start-time Minimization:

(1) $\forall j$, ReadyTime(PE_j) = 0;
(2) **while** the scheduling list L is not empty **do**
(3) remove the first node n_i from L;
(4) Min_ST = ∞;
(5) **for** j = 0 **to** p-1 **do**
(6) This_ST = *max*{ReadyTime(PE_j), DAT(n_i, PE_j)};
(7) **if** This_ST < Min_ST **then** Min_ST = This_ST; Candidate = PE_j; **endif**
(8) **endfor**
(9) schedule n_i to Candidate; ReadyTime(Candidate) = Min_ST + $w(n_i)$;
(10) endwhile

An optimal ordering of nodes in the list is required to generate an optimal schedule using the list scheduling approach. For instance, in Hu's algorithm [20], the scheduling list is constructed by using a node labeling process which proceeds from the top level leave nodes of the free-tree down to the root node. Such labeling leads to an optimal ordering of the nodes in that the nodes in the list, when scheduled, will occupy the earliest possible time slot in the processors. Unfortunately, while optimal scheduling lists can be easily constructed for certain restricted cases (e.g., a unit-weight free-tree as in the case of Hu's algorithm), such lists cannot be determined for arbitrary DAGs. Indeed, there are an exponential number of legitimate lists for a DAG that can be used for scheduling. An exhaustive search for an optimal list is clearly not a feasible approach.

Recent heuristics use node priorities to construct scheduling lists. Node priorities can be assigned using various attributes. Two frequently used attributes for assigning priority are the *t-level* (top level) and *b-level* (bottom level). The *t-level* of a node n_i is the length of the longest path from an entry node to n_i (excluding n_i). Here, the length of a path is the sum of all the node and edge weights along the path. The *t-level* of n_i highly correlates with n_i's start-time which is determined after n_i is scheduled to a processor. The *b-level* of a node n_i is the length of the longest path from node n_i to an exit node. The *b-level* of a node is bounded by the length of the *critical path* (CP). A CP of a DAG, is a path with the longest length; clearly, a DAG can have more than one CP. Some scheduling algorithms do not take into account the edge weights in

computing the *b-level;* to distinguish such definition of *b-level* from the one described above, we call it the *static b-level* or simply *static level* (*sl*). Some other DAG scheduling algorithms employ an attribute called *ALAP* (As-Late-As-Possible start-time). To compute these attributes, only two traversals of the DAG are needed. Thus, these attributes can be computed in $O(e+v)$ time. For example, the *t-level*'s, *b-level*'s, and *sl*'s of the DAG depicted in Figure 1(a) are shown in Figure 3. Note that the nodes of the CP are marked by an asterisk.

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

Figure 3: The static levels (*sl*'s), *t-level*'s, *b-level*'s and ALAP's of the nodes.

Using different combinations of the above attributes, some algorithms have demonstrated better performance than the others. For example, let us consider the schedules for the task graph shown in Figure 1(a) produced by the DCP algorithm [23] and the MCP algorithm [41], which are shown in Figure 4(a) and Figure 4(b), respectively. Note that the schedule generated by the DCP algorithm is an optimal schedule (schedule length = 16 time units). The scheduling order of the MCP algorithm is: $n_1, n_4, n_2, n_3, n_7, n_6, n_8, n_5, n_9$, which is an increasing ALAP ordering. However, this order is sub-optimal. On the other hand, the DCP algorithm does not schedule node following a static topological order, and optimizes the use of available time slots in the processors by scheduling some more important descendants first. The DCP algorithm is described in detail in [23].

Let us analyze the schedule† produced by the DCP algorithm from another perspective: If we are given a list $\{n_1, n_2, n_7, n_4, n_3, n_8, n_6, n_9, n_5\}$ and we schedule the nodes on the list one by one using the start-time minimization strategy, we will get a slightly different schedule with the same length (shown in Figure 5) assuming four processors are available. Another list, $\{n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9\}$, can also result in the same schedule. Thus, we can view the start-time minimization method as a mapping $M: \Pi \rightarrow S$ which maps the set Π of topologically ordered lists to the set S of valid schedules. However, such a mapping is not surjective. That is, when we are given an optimal schedule, it is not always possible to find a corresponding list which can lead to the schedule by the start-time minimization strategy.

For example, the optimal schedule generated by the DCP algorithm, shown in Figure 4(a), cannot be generated by any list using start-time minimization. The reason is that the node n_6 does not start at the earliest possible time, the time right after n_7 finishes. On the other hand,

†. Note that the DCP algorithm assumes the availability of unlimited number of processors [23], albeit it uses only three.

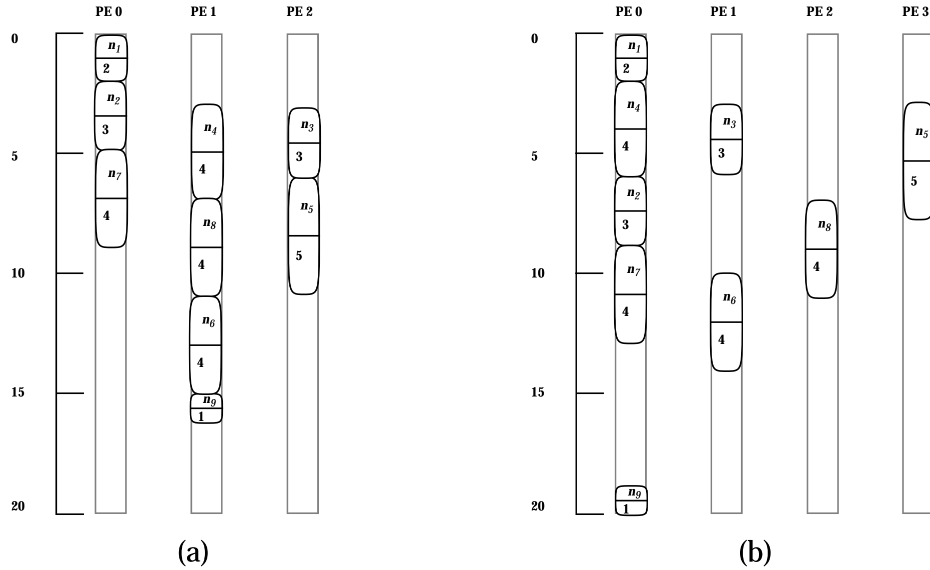


Figure 4: The schedules for the example DAG shown in Figure 1(a) generated by (a) the DCP algorithm (schedule length = 16 time units); (b) the MCP algorithm (schedule length = 20 time units).

such a mapping is not injective either because distinct lists can be mapped to the same schedule. The scheduling problem is then reduced to the problem of finding a list which can be mapped to an optimal schedule. In fact most of the list scheduling algorithms can be analyzed using this framework. The major differences in these algorithms are: (i) the method of implementing a different function M (i.e., a different space and time assignment strategy, which may not be start-time minimization); and (ii) the method of selecting scheduling lists from the set Π. Some algorithms optimize the former while constructing lists by a simple method. Other algorithms, such as the MCP algorithm, optimize the latter while using the start-time minimization strategy as the mapping M. A few algorithms, such as the DCP algorithm, optimize both.

4.2 A Genetic Formulation of the Scheduling Problem

The likelihood of the existence of lists leading to optimal schedules using the start-time minimization technique is very high, albeit it has not been proven that such a list always exists. Since an optimal schedule is not unique, the list which can lead to an optimal schedule, therefore, is not unique. We call such lists as the optimal lists. There are a number of members in the set Π which are qualified to be optimal lists. A solution neighborhood can then be defined for genetic search. Specifically, we can start from an initial list from which we obtain an initial schedule. We can then systematically modify the ordering within the list in a way such that the nodes are still in topological order (i.e., the precedence constraints are still satisfied). From the new list we obtain a new schedule. If the schedule is better, we adopt it; otherwise we test another modified list.

Based on the above analysis, we give the proposed genetic formulation of the scheduling problem as follows.

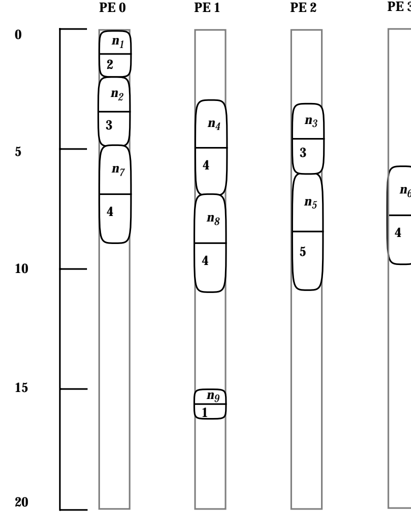


Figure 5: The schedule generated by the start-time minimization method (schedule length = 16 time units).

Encoding: We make a valid scheduling list as a chromosome. A valid scheduling list is one in which the nodes of the DAG are in a topological order. For example, the list $\{n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9\}$ is a valid chromosome for the DAG shown in Figure 1(a).

Fitness of a Chromosome: Fitness value is defined as: $(\sum w(n_i) - SL)/\sum w(n_i)$, where the schedule length *SL* is determined by using the start-time minimization method. The fitness of a chromosome is therefore always bounded between 0 and 1. For example, the list used by the MCP algorithm for the DAG shown in Figure 1(a) has a fitness value of $\frac{10}{30} = 0.3333$.

Generation of the Initial Population: An initial population is generated from a set of scheduling lists which are constructed by ALAP ordering, *b-level* ordering, *t-level* ordering, *sl* ordering and a random topological ordering, etc. These different orderings not only provide the necessary diversity but also represent a population with a higher fitness than a set of totally random topological orderings. A whole population is then generated from these orderings by performing random valid swapping of nodes in the lists.

In the next section we describe the design of the mutation and crossover operators. Since the selection mechanism is related to the migration process, we will discuss this aspect when we present the parallel genetic algorithm.

4.3 Genetic Operators

As the standard crossover and mutation operators may violate precedence constraints, we need to use other well-defined genetic operators. The inversion operator is not considered because it can obviously generate invalid scheduling lists. We consider three kinds of crossover operators: the order crossover [16], [33], [40], the partially-mapped crossover (PMX) [16], [33], and the cycle crossover [16], [33]. By using small counter-examples, we show that the PMX and cycle crossover operators may also produce invalid lists. Therefore in the proposed algorithm,

only the order crossover operator is used. We also describe a a mutation operator based on node-swapping.

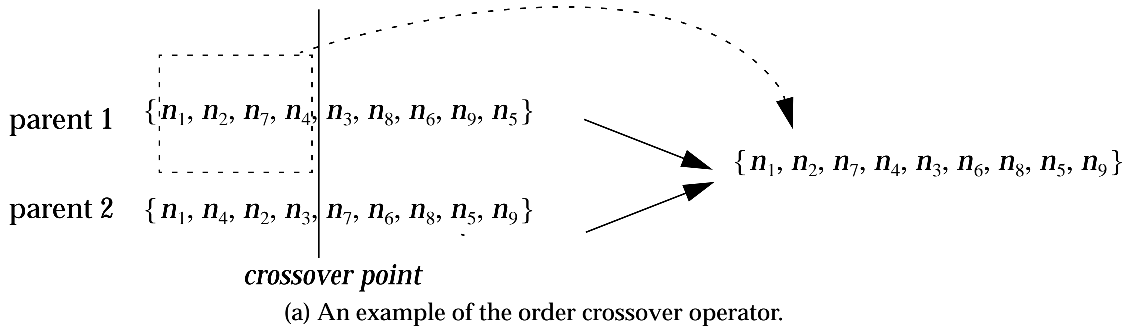
Order Crossover Operator: We consider a single-point *order crossover* operator. That is, given two parents, we first pass the left segment (i.e., the segment on the left of the crossover point) from the first parent, called parent 1, to the child. Then we construct the right fragment of the child by taking the remaining nodes from the other parent, called parent 2, in the same order. An example of the crossover operator is given in Figure 6(a). Note that the chromosomes shown in the figure are all valid topological ordering of the DAG in Figure 1(a). The left segment $\{n_1, n_2, n_7, n_4\}$ of parent 1 is passed directly to the child. The nodes in the right segment $\{n_3, n_8, n_6, n_9, n_5\}$ of parent 1 are then appended to the child according to their order in parent 2. This order crossover operator is easy to implement and permits fast processing. The most important merit is that it never violates the precedence constraints, as dictated by following theorem.

Theorem 1: *The order crossover operator always produces a valid scheduling list from two valid parent chromosomes.*

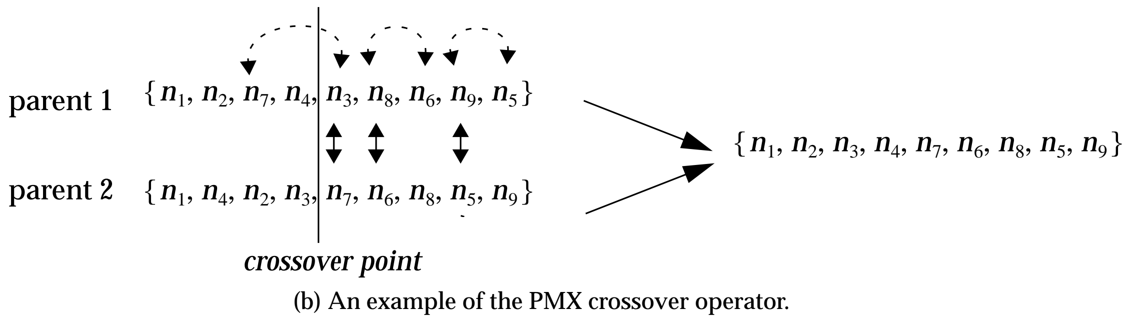
Proof: Suppose we are given two parent chromosomes $\{n_{i_1}, n_{i_2}, \ldots, n_{i_v}\}$ and $\{n_{j_1}, n_{j_2}, \ldots, n_{j_v}\}$. Let the crossover point be chosen at the *k*th position. Then after applying the order crossover, the child will be $\{n_{i_1}, n_{i_2}, \ldots, n_{i_k}, n_{j_x}, \ldots, n_{j_y}\}$ for some indices *x* and *y*. Obviously the precedence constraints among any two nodes at or before the *k*th position will be respected. For a node at or before the *k*th position and a node after the *k*th position, the precedence constraint (if any) will also be respected because their relative positions are the same as in parent 1. Finally for any two nodes after the *k*th position, the precedence constraint (if any) will also be respected because their relative positions are the same as in parent 2, by definition of the order crossover operator. (Q.E.D.)

The order crossover operator as defined above has the potential to properly combine the accurate task orderings of the two parent chromosomes so as to generate a scheduling list which can lead to a shorter schedule. This is because the “good” portions of a parent chromosome is a subsequence of the list which is an optimal scheduling ordering of the nodes in the subsequence. These good portions are essentially the building-blocks of an optimal list, and an order crossover operation can potentially pass such building-blocks to an offspring chromosome from which a shorter schedule may be obtained.

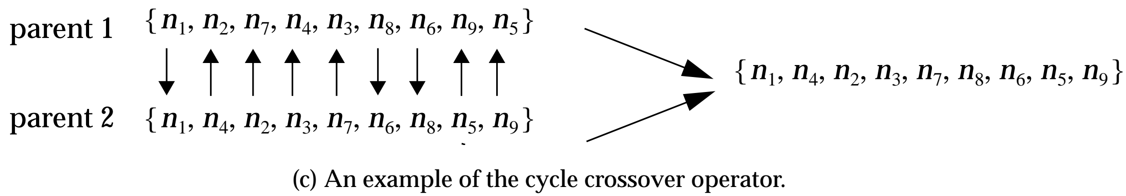
PMX Crossover Operator: A single point partially-mapped crossover can be implemented as follows. First a random crossover point is chosen. Then we consider the segments following the crossover point in both parents as the partial mapping of the genes to be exchanged in the first parent to generate the child. To do this, we first take corresponding genes from the two right segments of both parents, and then locate both these genes in the first parent and exchange them. Thus a gene in the right segment of the first parent and a gene at the same position in the second parent will define which genes in the first parent have to be swapped to generate the child. An example of the crossover operator is given in Figure 6(b). The pairs (n_3, n_7), (n_8, n_6), and (n_9, n_5) are situated at the same locations in both parents (note that the pairs (n_6, n_8) and (n_5, n_9) are



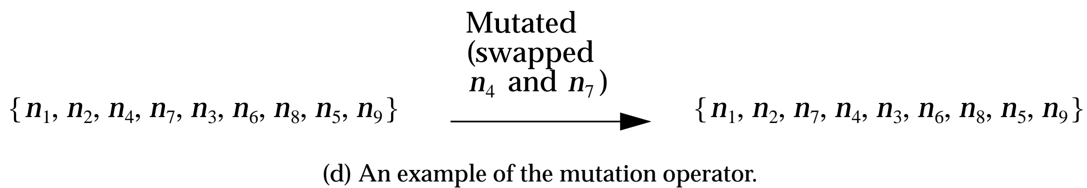
(a) An example of the order crossover operator.



(b) An example of the PMX crossover operator.



(c) An example of the cycle crossover operator.



(d) An example of the mutation operator.

Figure 6: Examples of the (a) order crossover, (b) PMX crossover, (c) cycle crossover, and (d) mutation operators.

implicitly handled also). Their corresponding positions in parent 1 are swapped to generate the child. Then the remaining nodes in parent 1 are copied to the child to finish the crossover. Unfortunately, unlike the order crossover operator, PMX crossover may produce invalid scheduling lists. An example of such scenario is shown in Figure 7. As can be seen, in the resulting chromosome, the positions of n_2 and n_4 violate their precedence relationship. This is also true for n_3 and n_5.

Cycle Crossover Operator: A single point cycle crossover operator can be implemented as

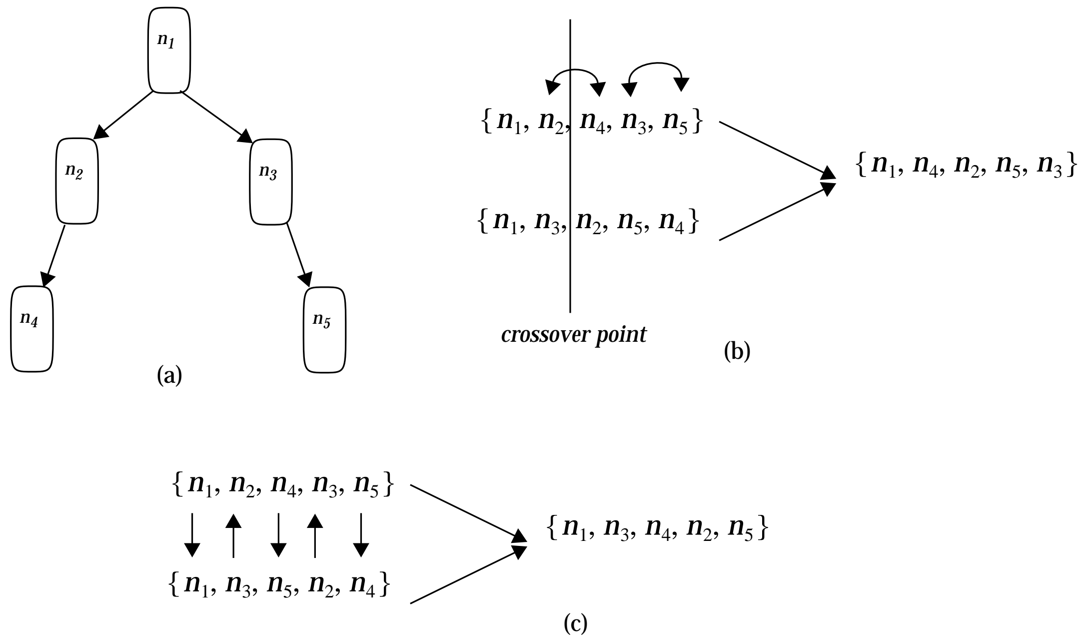


Figure 7: (a) A simple task graph; (b) An example of generating an invalid ordering of the graph by using the PMX crossover operator; (c) An example of generating an invalid ordering of the graph by using the cycle crossover operator.

follows. We start at position 1 in parent 1 and copy the gene to location 1 of the child. Then we examine the gene at position 1 in parent 2. This gene cannot be copied to the child since the child's corresponding position has been occupied. We then locate this gene from parent 1 and suppose it is found in position *i*. We copy this gene to position *i* of the child. Similarly the gene at position *i* in parent 2 cannot be copied. We again locate this gene from parent 1 and copy it to the child. This process is repeated until we encounter a gene in parent 2 which has already been copied to the child from parent 1. This completes one cycle. Another cycle is then initiated at the earliest position of the child that has not been occupied and the copying is performed from parent 2 to the child. Thus, in alternate cycles, the child inherits genes from both parents and the genes are copied at the same locations.

An example of the crossover operator is given in Figure 6(c). As can be seen in the figure, n_1 is first copied to the child from parent 1. But the corresponding node in parent 2 is also n_1 and, therefore, a cycle is completed. We start over again at position 2 of parent 2. We first copy n_4 from parent 2 to the child. Then we find that the node in position 2 of parent 1 is n_2 so that we copy n_2 from parent 2 to position 3 of the child. This time the corresponding node is n_7 and so we copy n_7 from parent 2 to position 5 of the child. Since the node in position 5 of parent 1 is n_3, we copy n_3 from parent 2 to position 4 of the child. Now we encounter n_4 in parent 1 which has already been copied from parent 2 to the child. Thus, the second cycle is completed. The third cycle starts from parent 1 again and nodes n_8 and n_6 are copied to the child at positions 6 and 7,

respectively. The last cycle starts from parent 2 again and nodes n_5 and n_9 are copied to the child at position 8 and 9, respectively. Unfortunately the cycle crossover operator may also generate invalid scheduling lists. An example of such situation is shown in Figure 7. As can be seen, after the crossover, the positions of n_2 and n_4 violate their precedence relationship.

Since both the PMX crossover operator and the cycle crossover operator does not guarantee valid scheduling lists, they are not employed in the proposed algorithm.

Mutation Operator: A valid topological order can be transformed into another topological order by swapping some nodes. For example, the scheduling list used by the MCP algorithm—$\{n_1, n_4, n_2, n_3, n_7, n_6, n_8, n_5, n_9\}$—can be transformed into an optimal list $\{n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9\}$ by swapping n_2 and n_4. Not every pairs of nodes can be swapped without violating the precedence constraints. Two nodes are *interchangeable* if they are not lying on the same path in the DAG. Using a pre-processing depth-first traversal of the DAG, we can check whether two nodes are interchangeable in a constant time during the search. This implies that we can efficiently test whether two randomly selected nodes are interchangeable and if so swap them and check the new schedule length. Such swapping actually defines a random search neighborhood. The size of the neighborhood is $O(v^2)$ since there are $O(C_2^v)$ pairs of interchangeable nodes. We define the mutation operator as a swap of two interchangeable nodes in a given chromosome. This operator captures the major characteristic of mutation, which is to randomly perturb the chromosome in such a way that a lost genetic feature can be recovered when the population is becoming homogeneous. An example is given in Figure 6(d) where two interchangeable nodes n_4 and n_7 are randomly chosen and their positions in the list are swapped.

4.4 Control Parameters

As Tanese [37], [38] has suggested, if the parallel processors executing a parallel genetic algorithm use heterogeneous control parameters, the diversity of the global population can be more effectively sustained. To implement this strategy, we use adaptive control parameters as suggested by Srinivas *et al.* [35]. The adaptive crossover rate μ_c is defined as follows:

$$\mu_c = \frac{k_c(f_{max} - f')}{(f_{max} - f_{avg})}$$

where f_{max} is the maximum fitness value in the local population, f_{avg} is the average fitness value, f' is the fitness value of the fitter parent for the crossover, and k_c is a positive real constant less than 1.

The adaptive mutation rate μ_c is defined as follows:

$$\mu_m = \frac{k_m(f_{max} - f)}{(f_{max} - f_{avg})}$$

where f is the fitness value of the chromosome to be mutated and k_m is a positive real constant less than 1.

Using the above adaptive crossover and mutation rate, the best chromosome is protected from disruption by crossover and mutation. On the other hand, when the population tends to become more homogeneous, both rates increase because f_{avg} will be about the same as f_{max}. Thus, under such a situation, chromosomes are more likely to be perturbed. This helps to prevent a pre-mature convergence to a sub-optimal solution. Note that even though the initial setting of the crossover rate and mutation rate is the same for all the parallel processors, the adaptive strategy gradually leads to the desired heterogeneity of the parameters among the processors.

Two other control parameters which are critical to the performance of a GA are the population size N_p and the number of generation N_g. Usually N_p and N_g are fixed for all problem sizes. However, this is not appropriate because larger problems require more time for exploration and exploitation. We therefore vary these two parameters linearly according to the problem size. Specifically, we set $N_p = k_p v$ and $N_g = k_g v$, where k_p and k_g are real constants.

4.5 Parallelization

The global population is partitioned into q sub-populations, where q is the number of PPEs. For efficiency we use a synchronous connected island model, in which PPEs communicate periodically to exchange the fittest individual and the communication is a synchronous voting such that the fittest individual is broadcast to all the PPEs. In other words sub-populations network topology is not considered. The reason is that the communication delay is insensitive to the distance between the PPEs of the Intel Paragon.

For the migration and selection of chromosomes, we adopt the following strategy. When the PPEs communicate, only the best chromosome migrates. When the fittest chromosome is imported, the worst chromosome in the local population is discarded while the fittest chromosome and the locally best chromosome are protected from the rank-based selection process. That is, in addition to having a higher expected share of offsprings, they are guaranteed to be retained in the new generation.

The period of communication for the PPEs is set to be T number of generations, which follows an exponentially decreasing sequence: initially $\left\lceil \frac{N_g}{2} \right\rceil$, then $\left\lceil \frac{N_g}{4} \right\rceil$, $\left\lceil \frac{N_g}{8} \right\rceil$, and so on. The rationale is that at the beginning of the search, the diversity of the global population is high. At such early stages, exploration is more important than exploitation; therefore, the PPEs should work on the local sub-population independently for a longer period of time. When the search reaches the later stages, it is likely that the global population converges to a number of different fittest chromosomes. Thus, exploitation of more promising chromosomes is needed to avoid unnecessary work on optimizing the locally best chromosomes that may have smaller fitness values than the globally best chromosomes.

With the above design considerations, the Parallel Genetic Scheduling (PGS) algorithm is outlined below.

Parallel Genetic Scheduling (PGS):

(1) Generate a local population with size equal to N_p/q by perturbing pre-defined topological orderings of the DAG (e.g., ALAP ordering, ***b-level*** ordering, etc.).
(2) $i = 2$;
(3) repeat
(4) $T = 0$;
(5) **repeat**
(6) **for** $j = 1$ **to** N_p/q **do**
(7) Using the current crossover and mutation rates, applied the two operators to randomly chosen chromosomes.
(8) **endfor**
(9) Evaluate the local population.
(10) **until** $++T = \lceil N_g/i \rceil$;
(11) Accept the best chromosome from a remote PPE and discard the worst local chromosome accordingly.
(12) Explicitly protect the best local and remote chromosome and adapt new crossover and mutation rates.
(13) $i = i \times 2$
(14) **until** the total number of generations elapsed equal to N_g;

5 Performance Results

To examine the efficacy of the PGS algorithm, we have implemented it on the Intel Paragon using the C language and tested it using different suites of task graphs. In the first two experiments, we aimed at investigating the absolute solution quality of the algorithm by applying it to two different sets of random task graphs for which the optimal solutions are known. As no widely accepted benchmark graphs exist for the DAG scheduling problem, we believe using random graphs with diverse parameters is appropriate for testing the performance of the algorithm. To compare the PGS algorithm with the existing techniques, we choose two extreme examples. The first is the DCP algorithm [23] which has been compared to the best known six heuristic algorithms (DLS [34], MCP [41], MD [41], ETF [21], DSC [42] and EZ [31]), and is known to be considerably better in terms of performance but has a slightly higher complexity. The other is the DSC algorithm [42] which is widely known and has been considered by many studies to be one of the best in terms of time complexity with a reasonable performance. In all the experiments, we assumed the target processors are fully-connected and homogeneous.

5.1 Workload

The first suite of random task graphs consists of three sets of graphs with different CCRs: 0.1, 1.0, and 10.0. Each set consists of graphs in which the number of nodes vary from 10 to 32 with increments of 2, and thus, each set contains 12 graphs. The graphs within the same set have the same value of CCR. 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 mean equal to 40 (minimum = 2 and maximum = 78). Then beginning from the first node, a random number indicating the number of children was chosen from a uniform distribution with mean equal to

$\frac{v}{10}$. Thus, the connectivity of the graph increases with the size of the graph. The communication cost of each edge was also randomly selected from a uniform distribution with mean equal to 40 times the specified value of CCR. Hereafter we call this suite of graphs the type-1 random task graphs.

To obtain optimal solutions for the task graphs, we applied a parallel A* algorithm to the graphs. For details of the A* algorithm, the reader is referred to [24]. Since generating optimal solutions for arbitrarily structured task graphs takes exponential time, it is not feasible to obtain optimal solutions for large graphs. On the other hand, to investigate the scalability of the PGS algorithm, it is desirable to test it with larger task graphs for which optimal solutions are known. To resolve this problem, we employed a different strategy to generate the second suite of random task graphs. Rather than trying to find out the optimal solutions after the graphs are randomly generated, we set out to generate task graphs with *given* optimal schedule lengths and number of processors used in the optimal schedules.

The method to generate task 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 SL_{opt} and p, respectively. Then for each PE i, we randomly generate a number x_i from a uniform distribution with mean $\frac{v}{p}$. The time interval between 0 and SL_{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 with increments of 50; thus, each set contains 10 graphs. The graphs within the same set have the same value of CCR. Hereafter we call this suite of graphs the type-2 random task graphs.

We also used a suite of regularly structured task graphs which represent a number parallel numerical algorithms. The suite comprises graphs representing the parallel Gaussian elimination algorithm [41], the parallel Laplace equation solver [41], and the parallel LU-decomposition algorithm [25]. As all these algorithms operate on matrices, the sizes of the task graphs vary with the matrix-sizes N in that $v = O(N^2)$. We used matrix-sizes from 9 to 18 with increments of 1.

5.2 Comparison against Optimal Solutions

Since the performance and efficiency of the PGS algorithm critically depend on the values of

N_g and N_p, in the first experiment we tested the algorithm using the type-1 random task graphs with varying N_g and N_p. First we fixed N_g by setting k_g as 20 and varied N_p by setting k_p as 2, 3, 5, and 10. Then we fixed N_p by setting k_p as 10 and varied N_g by setting k_g as 2, 5, 10, and 20. We used 8 PPEs on the Paragon for all these cases. In all the experiments the initial values of μ_c and μ_m were set to 0.6 and 0.02 respectively. In addition, both k_c and k_m were also fixed at 0.6 and 0.02.

The percentage deviations from optimal solutions for the three values of CCR were noted and are shown in Table 3. In the table, the total number of optimal solutions generated and the average percentage deviations for each CCR are also shown. Note that the average percentage deviations are calculated by dividing the total deviations by the number of non-optimal cases only. These average deviations, therefore, indicate more accurately the performance of the PGS algorithm when it is not able to generate optimal solutions. As can be noticed from the table, the PGS algorithm generated optimal solutions for over half of all the cases. The effect of k_p (hence N_p) was not very significant, though. We can see that the average deviations decrease slightly with increasing k_p. The effect of CCR is in fact more profound in that there are more deviations with increasing CCR. This is because as the edge weights become larger, there are more variations in the start-times of nodes and hence, it is more difficult to determine an optimal list. Finally, we observe that the PGS algorithm generated more optimal solutions for smaller problems.

Table 2: The percentage deviations from optimal schedule lengths for the type-1 random task graphs with three CCRs using four values of population-size constant: k_p = 2, 3, 5, and 10; 8 PPEs were used for all the cases.

k_p	2			3			5			10		
CCR	0.1	1.0	10.0	0.1	1.0	10.0	0.1	1.0	10.0	0.1	1.0	10.0
Graph Size 10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	0.0	9.6	12.0	0.0	9.6	7.0	0.0	9.6	7.0	0.0	8.8	7.0
18	0.0	0.0	17.3	0.0	0.0	8.9	0.0	0.0	8.9	0.0	0.0	8.9
20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
22	3.3	6.8	16.0	3.3	2.8	16.0	0.0	0.0	12.0	0.0	0.0	12.0
24	4.3	5.3	0.0	2.3	0.0	0.0	2.3	0.0	0.0	2.3	0.0	0.0
26	0.0	0.0	14.1	0.0	0.0	14.1	0.0	0.0	10.0	0.0	0.0	10.0
28	0.0	8.9	0.0	0.0	4.9	0.0	0.0	4.9	0.0	0.0	4.9	0.0
30	0.0	8.5	0.0	0.0	4.5	0.0	0.0	4.5	0.0	0.0	4.5	0.0
32	6.8	2.9	6.1	2.8	0.0	0.0	2.8	0.0	0.0	2.8	0.0	0.0
No. of Opt.	9	6	7	9	8	8	10	9	8	10	9	8
Avg. Dev.	4.8	7.0	13.1	2.8	5.4	11.5	2.5	6.3	9.5	2.5	6.1	9.5

The results with a fixed value of N_p but with varying values of N_g are shown in Table 3. These results indicate that the effect of k_g is more significant than that of k_p. The number of optimal solutions generated for the cases with smaller number of generations were notably less than that of the cases with larger number of generations. The average percentage deviations were also larger for smaller number of generations. When k_g was 10 or higher, the PGS

Table 3: The percentage deviations from optimal schedule lengths for the type-1 random task graphs with three CCRs using four values of number-of-generation constant: k_g = 2, 5, 10, and 20; 8 PPEs were used for all the cases.

k_g	2			5			10			20		
CCR	0.1	1.0	10.0	0.1	1.0	10.0	0.1	1.0	10.0	0.1	1.0	10.0
Graph Size 10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	14.8	17.7	14.9	2.5	16.6	14.6	0.0	11.0	9.1	0.0	8.8	7.0
18	27.9	8.2	17.6	21.0	3.5	16.0	7.8	0.0	9.0	0.0	0.0	8.9
20	0.0	30.3	19.1	0.0	24.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
22	5.5	15.4	25.1	4.0	14.5	23.3	0.0	0.0	14.3	0.0	0.0	12.0
24	19.5	0.0	23.8	11.9	0.0	19.2	4.9	0.0	7.8	2.3	0.0	0.0
26	14.0	30.2	20.1	12.8	20.5	16.0	0.0	0.0	13.3	0.0	0.0	10.0
28	24.3	21.8	11.4	18.0	12.4	0.0	0.0	6.5	0.0	0.0	4.9	0.0
30	15.8	26.5	19.4	12.2	17.1	10.0	6.3	8.6	3.3	0.0	4.5	0.0
32	20.1	23.2	19.0	15.7	14.1	11.3	4.2	0.0	1.7	2.8	0.0	0.0
No. of Opt.	4	4	3	4	4	4	8	8	5	10	9	8
Avg. Dev.	17.7	21.7	18.9	12.2	15.4	13.8	5.8	6.5	8.4	2.5	6.1	9.5

algorithm attained a performance comparable to those shown earlier in Table 3. The effect of CCR was again respectable. An intuitive observation is that larger N_g and N_p in general can lead to better performance but the execution time required will then become larger too. Indeed there is a trade-off between better performance and higher efficiency.

Based on the results shown in Table 3 and Table 3, in the subsequent experiments k_p and k_g were fixed as 5 and 10, respectively. That is, $N_p = 5v$ and $N_g = 10v$, unless otherwise stated.

In the next experiment we aimed to investigate the effect of the number of PPEs on the performance of the algorithm. We applied the PGS algorithm to the type-1 random task graphs on the Paragon using 2, 4, 8, and 16 PPEs. Again the percentage deviations from optimal solutions were noted. These results are shown in Table 4 and indicate that the PGS algorithm demonstrated similar performance for 2, 4, and 8 PPEs. When 16 PPEs were used, the performance degraded by a slightly larger margin. One explanation for this phenomenon is that using more PPEs implies that the size of a local population is smaller and thus leads to smaller diversity of the local population. Premature convergence is then more likely to result. The effect of CCR was again quite considerable in that the average deviations increase with increasing CCRs.

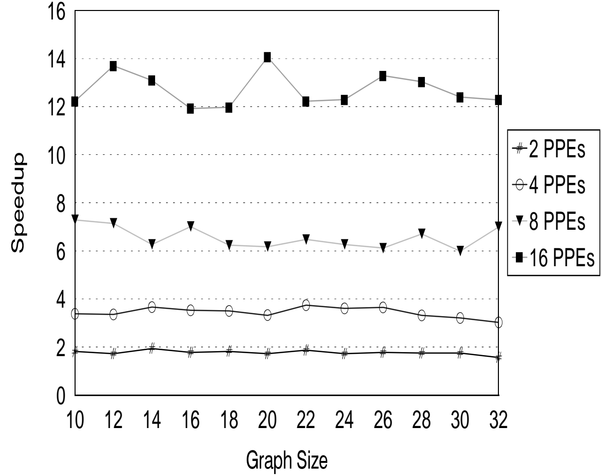
To examine the effectiveness of parallelization, we observed the execution times of the PGS algorithm using different number of PPEs. As a reference, we also ran the PGS algorithm using 1 PPE on the Paragon. The average execution times and speedups are shown in Figure 8. As can be seen from the speedup plot, the PGS algorithm demonstrated a slightly less than linear speedup. This is because the migration of best chromosomes contributed considerable communication overhead.

Table 4: Results of the PGS algorithm compared against optimal solutions (percentage deviations) for the type-1 random task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR	0.1				1.0				10.0			
No. of PPEs	2	4	8	16	2	4	8	16	2	4	8	16
Graph Size 10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.1	0.0	0.0	0.0	8.0
12	0.0	0.0	0.0	2.3	0.0	0.0	0.0	16.4	0.0	0.0	0.0	5.1
14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.4	0.0	0.0	0.0	9.3
16	0.9	0.6	0.6	0.7	9.2	8.1	7.9	17.8	7.6	6.7	6.2	16.3
18	6.2	5.2	5.0	6.0	0.0	0.0	0.0	5.2	8.0	7.2	7.2	7.7
20	0.0	0.0	0.0	5.3	0.0	0.0	0.0	11.0	0.0	0.0	0.0	0.3
22	0.0	0.0	0.0	4.6	0.0	0.0	0.0	18.0	13.0	12.1	12.0	24.4
24	4.0	3.3	2.9	5.1	0.0	0.0	0.0	10.3	6.2	5.4	5.2	12.0
26	6.6	5.3	5.0	14.6	0.0	0.0	0.0	9.4	10.2	10.3	9.7	32.5
28	4.7	3.2	3.1	7.2	6.1	5.3	5.1	10.4	0.0	0.0	0.0	10.4
30	4.3	2.6	2.2	4.5	6.3	5.3	4.9	11.0	2.3	2.3	2.1	2.3
32	3.3	2.3	2.1	3.3	0.0	0.0	0.0	13.2	1.2	1.1	1.0	2.0
No. of Opt.	5	5	5	2	9	9	9	0	5	5	5	0
Avg. Dev.	4.3	3.2	3.0	5.4	7.2	6.2	6.0	11.0	7.0	6.4	6.2	10.8

Graph Size	Running Times (secs)
10	4.12
12	4.64
14	4.86
16	5.02
18	5.32
20	5.79
22	6.03
24	6.34
26	6.89
28	7.12
30	7.63
32	7.82

(a) Average running times using 1 PPE.



(b) Average speedups.

Figure 8: (a) The average running times of the PGS algorithm for the type-1 random task graphs with three CCRs using 1 PPE on the Intel Paragon; (b) the average speedups of the PGS algorithm for 2, 4, 8, and 16 PPEs.

From the above results we find that the number of generations used critically affects the performance of the PGS algorithm. In view of this, we ran the algorithm with larger number of generations to see how far it can approach optimal. We used 8 PPEs on the Paragon and other parameters remained the same. The results are shown in Figure 11. Note that the results for the smaller number of generations shown earlier in Table 3 are also included. We notice that when the number of generations is increased to $40v$, the PGS algorithm is almost optimal.

In the next experiment we used the type-2 random task graphs, which are significantly larger graphs, to examine the performance of the PGS algorithm. Again the percentage deviations from

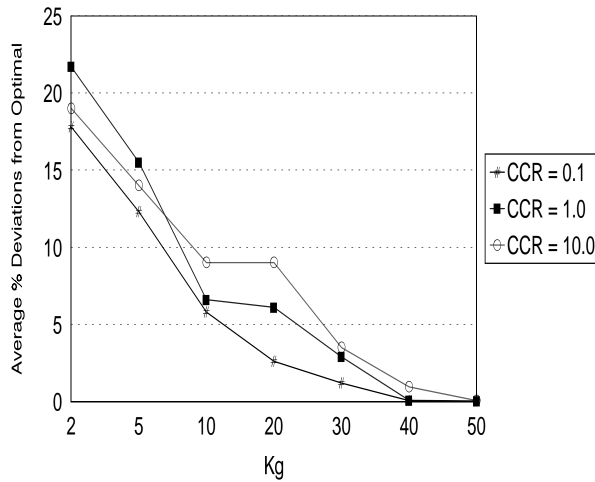


Figure 9: The average percentage deviations from the optimal of the solutions generated by the PGS algorithm for the type-1 random task graphs with three CCRs using 8 PPEs on the Intel Paragon.

optimal solutions were noted. The results are shown in Table 5. We notice that the percentage deviations are much larger than that of the type-1 random task graphs. Nonetheless the PGS algorithm was able to generate optimal solutions for these larger graphs. We also measured the running times of the algorithm which are shown in Figure 8. The speedup was also slightly less than linear due to the chromosomes migration process.

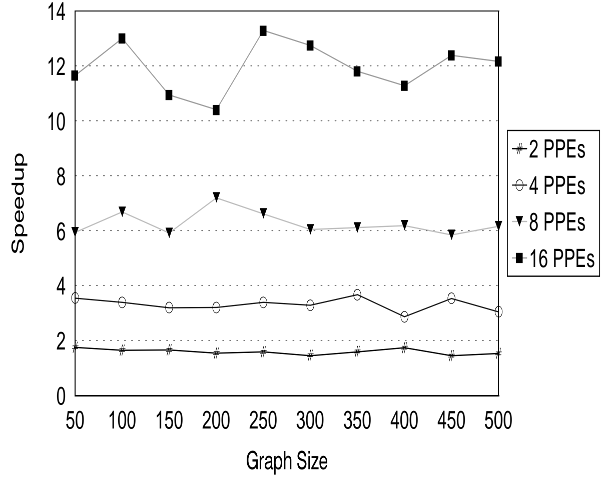
Table 5: Results of the PGS algorithm compared against optimal solutions (percentage deviations) for the type-2 random task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR	0.1				1.0				10.0			
No. of PPEs	2	4	8	16	2	4	8	16	2	4	8	16
Graph Size 50	0.0	12.4	2.4	1.5	6.6	0.0	0.0	0.0	3.4	17.2	5.8	15.2
100	0.0	0.4	9.5	0.0	0.0	0.0	0.0	7.7	0.0	32.0	0.0	35.8
150	16.7	2.7	0.0	0.0	0.0	16.2	20.1	20.0	10.9	0.0	11.6	13.4
200	0.0	0.0	1.4	0.0	2.7	2.4	23.1	25.9	14.6	26.6	0.0	23.1
250	0.0	2.1	6.5	0.0	0.0	24.5	0.0	0.0	35.0	0.0	21.0	0.0
300	12.7	0.0	5.1	15.9	0.0	0.0	0.0	0.0	24.0	0.0	0.0	12.0
350	0.0	0.0	18.0	0.0	4.5	13.7	6.5	8.9	7.97	3.1	0.0	0.0
400	0.0	3.2	6.9	17.9	12.8	13.6	0.0	10.3	0.0	35.1	38.2	1.4
450	0.0	0.0	10.0	1.0	0.0	0.0	0.0	0.0	17.5	24.6	16.0	2.6
500	0.0	18.5	3.3	3.6	0.0	24.3	0.0	0.0	0.0	0.0	20.5	37.7
No. of Opt.	8	4	1	5	6	4	7	5	3	4	4	2
Avg. Dev.	14.7	6.5	7.0	8.0	6.6	15.8	16.6	14.6	16.2	23.1	18.9	17.7

As for the type-1 random task graphs, we also tested the PGS algorithm with a larger number of generations. Again 8 PPEs were used and other parameters remained the same. The results are shown in Figure 11 indicating that the PGS algorithm is almost optimal when the number of generations used is $50v$.

Graph Size	Running Times (secs)
50	16.41
100	26.10
150	44.96
200	61.54
250	102.60
300	156.98
350	184.75
400	270.32
450	370.77
500	489.32

(a) Average running times using 1 PPE.



(b) Average speedups.

Figure 10: (a) The average running times of the PGS algorithm for the type-2 random task graphs with three CCRs using 1 PPE on the Intel Paragon; (b) the average speedups of the PGS algorithm for 2, 4, 8, and 16 PPEs.

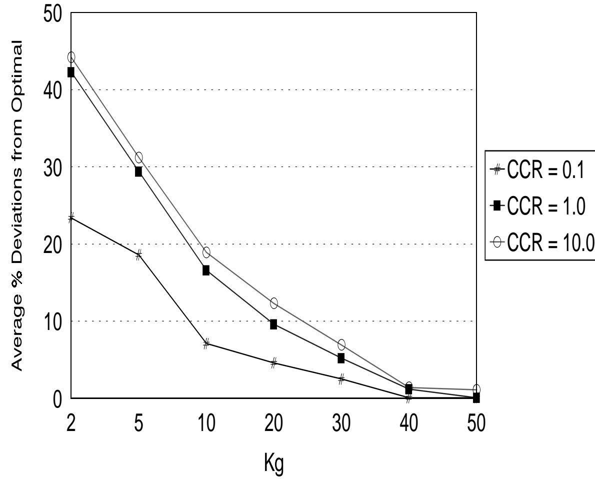


Figure 11: The average percentage deviations from the optimal of the solutions generated by the PGS algorithm for the type-2 random task graphs with three CCRs using 8 PPEs on the Intel Paragon.

5.3 Results on Regular Graphs

In the last experiment we aimed to compare the performance of the PGS algorithm with the DSC [42] and DCP [23] algorithms. We used three sets of regular task graphs representing three parallel numerical algorithms: Gaussian elimination, LU-decomposition, and Laplace equation solver. The size of these graphs vary as the input matrix sizes for these numerical algorithms. We used 10 graphs for each CCR with matrix sizes varied from 9 to 18. The sizes of the 10 graphs then varied roughly from 30 to 300. It should be noted that these graphs are sparse graphs compared with the random graphs used in the previous experiments.

As optimal solutions for these regular graphs are not available, we computed the ratios of the schedule lengths generated by the PGS algorithm to that of the DSC and DCP algorithm using 2, 4, 8, and 16 PPEs on the Paragon. The results of comparing the PGS algorithm with the DSC algorithm for the Gaussian elimination graphs are shown in Table 7 which also includes the

Table 6: Ratios of the schedule lengths generated by the PGS algorithm to that of the DSC algorithm for the Gaussian elimination task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR	0.1				1.0				10.0			
No. of PPEs	2	4	8	16	2	4	8	16	2	4	8	16
Matrix Size 9	0.60	0.63	0.56	0.64	0.57	0.59	0.53	0.55	0.57	0.57	0.44	0.57
10	0.64	0.68	0.68	0.70	0.55	0.52	0.52	0.52	0.74	0.74	0.74	0.95
11	0.58	0.60	0.57	0.58	0.66	0.66	0.72	0.60	0.81	0.61	0.62	0.62
12	0.59	0.62	0.62	0.62	0.55	0.50	0.53	0.50	0.51	0.51	0.59	0.51
13	0.58	0.62	0.62	0.62	0.61	0.57	0.56	0.57	0.94	0.45	0.80	0.76
14	0.59	0.60	0.57	0.60	0.63	0.59	0.46	0.47	0.52	0.70	0.45	0.52
15	0.62	0.72	0.66	0.62	0.61	0.61	0.61	0.61	0.55	0.55	0.57	0.54
16	0.51	0.56	0.55	0.59	0.67	0.67	0.67	0.67	0.40	0.61	0.68	0.48
17	0.53	0.57	0.51	0.53	0.58	0.66	0.66	0.58	0.46	0.56	0.73	0.73
18	0.78	0.69	0.75	0.73	0.53	0.65	0.65	0.65	0.75	0.85	0.75	0.93
Avg. Ratio	0.60	0.63	0.61	0.62	0.60	0.60	0.59	0.57	0.62	0.61	0.64	0.66

average ratios. Here the PGS algorithm outperformed the DSC algorithm in all cases. The improvement of the PGS algorithm over the DSC algorithm declines slightly with increasing number of PPEs and larger values of CCR.

The results of comparing the PGS algorithm with the DCP algorithm for the Gaussian elimination task graphs are shown in Table 7. These results reveal that the performance of both algorithms were roughly the same. Indeed the PGS algorithm generated the same schedule lengths as that of the DCP algorithm for more than half of the cases.

The results for the LU-decomposition graphs are shown in Table 8 and Table 8. From Table 8, we can observe that for this type of regular graph, the PGS algorithm again generated much better schedules compared with that of the DSC algorithm. On the other hand, the PGS algorithm produced slightly inferior solutions compared to that of the DCP algorithm. An explanation for this phenomenon is that the LU-decomposition graphs have multiple critical paths, which make minimization of schedule lengths much more difficult. Nonetheless, the PGS algorithm outperformed the DCP algorithm with respectable margins for a number of cases.

The results for the Laplace equation solver graphs are shown in Table 10 and Table 10. Again we find that the PGS algorithm consistently outperformed the DSC algorithm for all but two cases. On the other hand, the overall performance of the DCP algorithm was slightly better, which is presumably because all the paths in a Laplace equation solver graph are critical paths. The average differences in schedule lengths between the two algorithms, however, are within 20%.

Table 7: Ratios of the schedule lengths generated by the PGS algorithm to that of the DCP algorithm for the Gaussian elimination task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR		0.1				1.0				10.0			
No. of PPEs		2	4	8	16	2	4	8	16	2	4	8	16
Matrix Size	9	1.00	1.05	0.93	1.07	1.03	1.07	0.97	1.00	1.00	1.00	0.77	1.00
	10	0.94	1.00	1.00	1.02	1.06	1.00	1.00	1.00	1.00	1.00	1.00	1.29
	11	1.00	1.02	0.97	1.00	1.00	1.00	1.08	0.91	1.30	0.98	1.00	1.00
	12	0.95	1.00	1.00	1.00	1.09	1.00	1.05	1.00	1.00	1.00	1.16	1.00
	13	0.94	1.00	0.99	1.00	1.06	1.00	0.98	1.00	1.32	0.63	1.12	1.07
	14	0.99	1.00	0.96	1.00	1.18	1.12	0.86	0.88	1.00	1.35	0.87	1.00
	15	0.93	1.09	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00	1.04	0.98
	16	0.91	1.00	0.99	1.06	0.99	1.00	1.00	1.00	0.80	1.22	1.35	0.95
	17	1.00	1.08	0.96	1.00	0.88	1.00	1.00	0.88	0.63	0.76	1.00	1.00
	18	1.05	0.93	1.02	0.99	0.82	1.00	1.00	1.00	1.00	1.14	1.00	1.24
Avg. Ratio		0.97	1.02	0.98	1.01	1.01	1.02	0.99	0.97	1.01	1.01	1.03	1.05

Table 8: Ratios of the schedule lengths generated by the PGS algorithm to that of the DSC algorithm for the LU-decomposition task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR		0.1				1.0				10.0			
No. of PPEs		2	4	8	16	2	4	8	16	2	4	8	16
Matrix Size	9	0.67	0.74	0.68	0.73	0.78	0.71	0.69	0.66	0.77	0.70	0.66	0.77
	10	0.56	0.58	0.56	0.59	0.60	0.63	0.65	0.58	0.65	0.80	0.64	0.72
	11	0.59	0.64	0.59	0.63	0.67	0.78	0.72	0.72	0.83	0.78	0.70	0.66
	12	0.59	0.57	0.58	0.54	0.76	0.74	0.68	0.68	0.75	0.61	0.61	0.60
	13	0.63	0.64	0.65	0.60	0.57	0.52	0.49	0.58	0.60	0.57	0.52	0.63
	14	0.79	0.76	0.79	0.76	0.54	0.56	0.53	0.51	0.80	0.73	0.84	0.73
	15	0.58	0.55	0.54	0.55	0.54	0.52	0.60	0.59	0.58	0.57	0.58	0.66
	16	0.55	0.53	0.52	0.57	0.66	0.75	0.76	0.66	0.82	0.68	0.70	0.77
	17	0.76	0.74	0.70	0.76	0.63	0.68	0.62	0.69	0.60	0.62	0.49	0.59
	18	0.74	0.67	0.70	0.65	0.71	0.74	0.76	0.67	0.69	0.65	0.63	0.59
Avg. Ratio		0.65	0.64	0.63	0.64	0.65	0.66	0.65	0.63	0.71	0.67	0.64	0.67

From the results for these three types of regular graphs, we find that the PGS algorithm can generate much better solutions than the DSC algorithm. Also the results indicate that the performance of the PGS algorithm is comparable to that of the DCP algorithm. These results should be interpreted with reference to the running times also. These timing results are shown in Figure 8. The running times of the DSC and DCP algorithm using 1 PPE on the Paragon are also shown. Furthermore, the speedups of the PGS algorithm are computed with respect to the DCP algorithm in order to directly compare the relative efficiency of the two algorithms. We find that the relative speedups were also moderately less than linear.

According to the results shown above, it appears that the number of generations is the most significant limiting factor of the performance of the PGS algorithm. We tested the PGS algorithms for the three types of regular graphs again with k_g set to be 20 and 30. We used 8

Table 9: Ratios of the schedule lengths generated by the PGS algorithm to that of the DCP algorithm for the LU-decomposition task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR		0.1				1.0				10.0			
No. of PPEs		2	4	8	16	2	4	8	16	2	4	8	16
Matrix Size	9	1.00	1.10	1.01	1.08	1.10	1.00	0.98	0.94	1.18	1.07	1.00	1.17
	10	1.02	1.05	1.02	1.07	1.06	1.12	1.15	1.03	1.01	1.25	1.00	1.12
	11	1.00	1.09	1.00	1.06	0.93	1.08	1.00	1.00	1.25	1.18	1.06	1.00
	12	1.09	1.05	1.08	1.00	1.12	1.09	1.00	1.00	1.22	1.00	1.00	0.98
	13	1.01	1.03	1.05	0.96	1.09	1.00	0.94	1.12	1.19	1.14	1.04	1.24
	14	1.08	1.04	1.09	1.04	1.07	1.10	1.04	1.00	1.10	1.00	1.15	1.00
	15	1.07	1.02	1.00	1.02	1.05	1.00	1.15	1.14	1.02	1.01	1.02	1.16
	16	1.05	1.00	0.99	1.09	1.00	1.13	1.15	1.00	1.22	1.01	1.04	1.14
	17	1.09	1.07	1.00	1.09	1.05	1.12	1.02	1.14	1.19	1.23	0.96	1.16
	18	1.09	1.00	1.03	0.97	1.00	1.04	1.07	0.95	1.17	1.10	1.08	1.00
Avg. Ratio		1.05	1.05	1.03	1.04	1.05	1.07	1.05	1.03	1.15	1.10	1.03	1.10

Table 10: Ratios of the schedule lengths generated by the PGS algorithm to that of the DSC algorithm for the Laplace equation solver task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

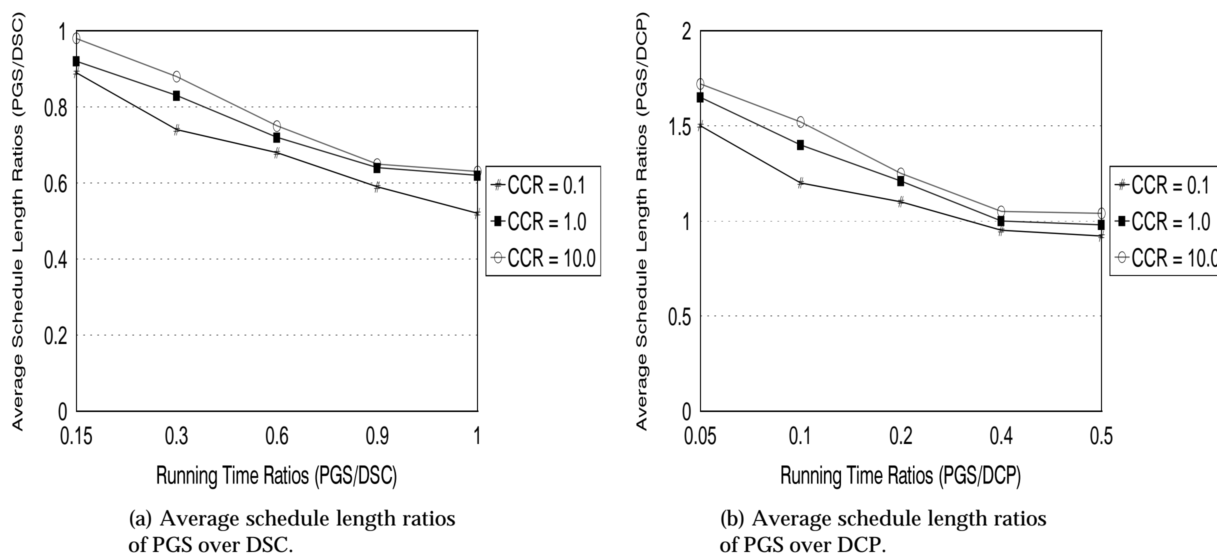
CCR		0.1				1.0				10.0			
No. of PPEs		2	4	8	16	2	4	8	16	2	4	8	16
Matrix Size	9	0.81	0.83	0.70	0.65	0.86	0.94	0.89	0.71	0.49	0.60	0.52	0.72
	10	0.64	0.56	0.65	0.61	0.95	0.64	0.64	0.81	0.56	0.94	0.76	0.76
	11	0.68	0.81	0.62	0.68	0.64	0.64	0.80	0.52	0.62	0.80	0.62	0.92
	12	0.46	0.50	0.50	0.53	0.62	0.52	0.52	0.75	0.82	0.56	0.57	0.53
	13	0.58	0.67	0.57	0.58	0.99	0.98	0.96	0.72	0.99	0.76	0.84	0.69
	14	0.55	0.81	0.72	0.68	0.88	1.01	0.62	0.84	0.60	0.60	0.65	0.61
	15	0.57	0.69	0.57	0.61	0.84	0.67	0.74	0.77	0.84	0.46	0.64	0.49
	16	0.64	0.67	0.70	0.82	0.54	0.80	0.66	0.60	0.73	0.89	0.58	0.67
	17	0.89	0.74	0.95	0.86	0.63	0.61	0.84	0.49	0.65	0.53	0.52	0.52
	18	0.63	0.65	0.73	0.76	0.78	0.87	1.04	0.89	0.46	0.98	0.98	0.86
Avg. Ratio		0.65	0.69	0.67	0.68	0.77	0.77	0.77	0.71	0.68	0.71	0.67	0.68

PPEs on the Paragon. Other parameters remained the same. The average schedule length ratios of PGS to DSC and DCP are depicted by the plot shown in Figure 8. For comparison, the ratios shown earlier are also included in the plot. Note that in the horizontal axis the ratios of the algorithm's running times, rather than the number of generations used, are indicated. When the ratio of running times approaches 0.5 (i.e., number of generations is about $30v$), the PGS algorithm outperformed the DCP algorithm for small values of CCR (i.e., 0.1 and 1.0) and showed almost the same performance when CCR is 10. It should be noted that even though the number of generations was increased by a factor of 2 and 3, the PGS algorithm was still faster than the DCP algorithm by a factor of roughly 3 and 1.8 respectively.

From these results we can conclude that the efficiency of the PGS algorithm is much higher than the DCP algorithm. Thus, the PGS algorithm is a viable choice for scheduling if a parallel

Table 11: Ratios of the schedule lengths generated by the PGS algorithm to that of the DCP algorithm for the Laplace equation solver task graphs with three CCRs using 2, 4, 8, and 16 PPEs on the Intel Paragon.

	CCR	0.1				1.0				10.0			
	No. of PPEs	2	4	8	16	2	4	8	16	2	4	8	16
Matrix Size	9	1.25	1.28	1.08	1.00	1.22	1.33	1.25	1.00	0.94	1.16	1.00	1.38
	10	1.19	1.05	1.22	1.13	1.48	1.00	1.00	1.27	0.85	1.43	1.16	1.16
	11	1.00	1.20	0.91	1.00	1.00	1.00	1.26	0.82	1.00	1.29	1.00	1.48
	12	0.92	1.00	1.00	1.05	1.20	1.00	1.01	1.45	1.56	1.06	1.07	1.00
	13	1.00	1.16	0.98	1.00	1.48	1.46	1.43	1.08	1.67	1.29	1.42	1.16
	14	0.87	1.29	1.14	1.08	1.22	1.40	0.86	1.16	1.00	1.00	1.08	1.01
	15	1.00	1.20	1.00	1.06	1.38	1.10	1.21	1.27	1.59	0.87	1.21	0.93
	16	1.00	1.05	1.09	1.28	1.00	1.48	1.22	1.11	1.25	1.52	1.00	1.14
	17	1.23	1.02	1.30	1.18	1.08	1.05	1.45	0.84	1.24	1.02	1.00	1.00
	18	1.07	1.10	1.23	1.29	1.07	1.19	1.42	1.21	0.67	1.41	1.42	1.24
	Avg. Ratio	1.05	1.14	1.10	1.11	1.21	1.20	1.21	1.12	1.18	1.20	1.14	1.15



(a) Average schedule length ratios of PGS over DSC.

(b) Average schedule length ratios of PGS over DCP.

Figure 13: The average ratios of schedule lengths for all the regular task graphs with three CCRs using 8 PPEs on the Intel Paragon: (a) PGS vs. DSC and (b) PGS vs. DCP.

processing platform is available.

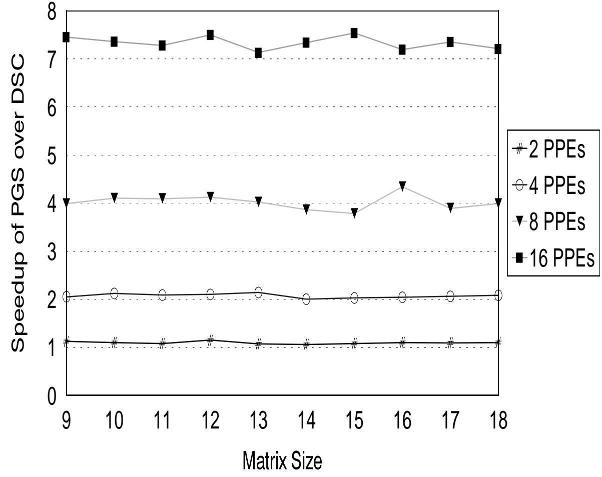
6 Related Work

In most of the previous work on multiprocessor scheduling using a genetic search, SGA with standard genetic operators is commonly adopted. Furthermore, these algorithms do not exploit the inherent parallelism in genetic search, and are sequential.

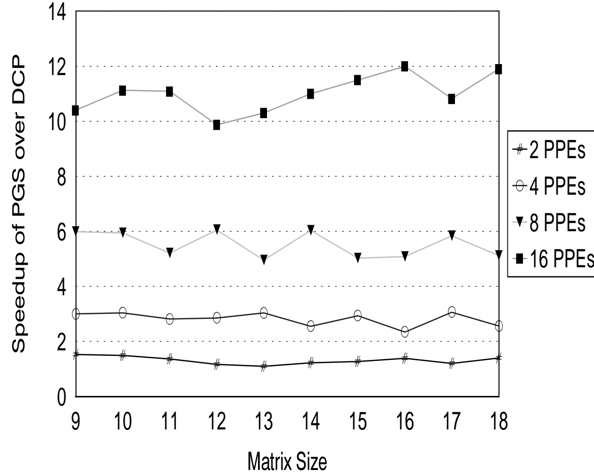
Benten and Sait [5] proposed a genetic algorithm for scheduling a DAG to a limited number of fully-connected processors with a contention-free communication network. In their scheme, each solution or schedule is encoded as a chromosome containing v alleles, each of which is an

Matrix Size	Running Times (secs) DSC	DCP	PGS
9	2.12	5.17	6.17
10	2.16	5.55	6.15
11	3.54	6.73	8.73
12	3.67	7.92	9.92
13	4.89	9.63	11.63
14	5.98	12.43	13.43
15	7.98	16.34	18.34
16	9.67	19.32	20.32
17	11.56	23.89	25.89
18	13.66	28.25	30.25

(a) Average running times using 1 PPE.



(b) Average speedups of PGS over DSC.



(c) Average speedups of PGS over DCP.

Figure 12: (a) The average running times of the PGS algorithm for all the regular task graphs with three CCRs using 1 PPE on the Intel Paragon; (b) the average speedups of the PGS algorithm over the DSC algorithm; (c) the average speedups of the PGS algorithm over the DCP algorithm.

ordered pair of task index and its assigned processor index. With such encoding the design of genetic operators is straightforward. Standard crossover is used because it always produces valid schedules as offsprings and is computationally efficient. Mutation is simply a swapping of the assigned processors between two randomly chosen alleles. For generating an initial population, Benten and Sait use a technique called “pre-scheduling” in which N_p random permutations of numbers from 1 to v are generated. The number in each random permutation represents the task index of the task graph. The tasks are then assigned to the PEs uniformly: the first $\frac{v}{p}$ tasks in a permutation are assigned to PE 0, the next $\frac{v}{p}$ tasks to PE 1, and so on. In their simulation study using randomly generated task graphs with a few tenths of nodes, their algorithm was shown to outperform the ETF algorithm proposed by Hwang *et al.* [21].

Hou, Ansari, and Ren [19] also proposed a scheduling algorithm using genetic search in which each chromosome is a collection of lists, each of which represents the schedule on a distinct processor. Thus, each chromosome is not a linear structure but is a two-dimensional structure instead. One dimension is a particular processor index and the other is the ordering of tasks scheduled on the processor. Using such an encoding scheme poses a restriction on the schedules being represented: the list of tasks within each processor in a schedule is ordered in ascending order of their *topological height*, which is defined as the largest number of edges from an entry node to the node itself. This restriction also facilitates the design of the crossover operator. In a crossover, two processors are selected from each of two chromosomes. The list of tasks on each processor is cut into two parts. Then the two chromosomes exchange the two lower parts of their task lists correspondingly. It is shown that this crossover mechanism always produces valid offsprings. However, the height restriction in the encoding may cause the search to be incapable of obtaining the optimal solution. It is because the optimal solution may not obey the height ordering restriction at all. Hou *et al.* then incorporated a heuristic technique to lower the likelihood of such pathological situation. Mutation is simpler in design. In a mutation, two randomly chosen tasks with the same height are swapped in the schedule. As to the generation of the initial population, N_p randomly permuted schedules obeying the height ordering restriction are generated. In their simulation study using randomly generated task graphs with a few tenths of nodes, their algorithm was shown to produce schedules within 20% from optimal solutions.

Ahmad and Dhodhi [2] proposed a scheduling algorithm using a variant of genetic algorithm called *simulated evolution*. They employ a problem-space neighborhood formulation in that a chromosome represents a list of task priorities. Since task priorities are dependent on the input DAG, different set of task priorities represent different problem instances. First, a list of priorities is obtained from the input DAG. Then the initial population of chromosomes are generated by randomly perturbing this original list. Standard genetic operators are applied to these chromosomes to determine the fittest chromosome which is the one based on which a list scheduling heuristic is applied, the resulting assignment and sequencing gives the shortest schedule length for the *original* problem. The genetic search, therefore, operates on the problem-space instead of the solution-space as is commonly done. The rationale of this approach is that good solutions of the problem instances in the problem-space neighborhood are expected to be good solutions for the original problem as well.

7 Conclusions

We have presented a parallel genetic algorithm, called the PGS algorithm, for multiprocessor DAG scheduling. The major motivation of using a genetic search approach is that the recombinative nature of a genetic algorithm can potentially determine an optimal scheduling list leading to an optimal schedule. Using well-defined crossover and mutation operators, the PGS algorithm judiciously combines good building-blocks of scheduling lists to construct better lists. Parallelization of the algorithm is based on a novel approach in that the parallel processors

communicate to exchange the best chromosomes with exponentially decreasing periods. As such, the parallel processors perform exploration of the solution-space at the early stages and exploitation at the later stages.

In our experimental studies, we have found that the PGS algorithm generates optimal solutions for more than half of all the cases in which random task graphs were used. In addition, the PGS algorithm demonstrates an almost linear speedup and is therefore scalable.

While the DCP algorithm has already been shown to outperform many of the leading algorithms, the PGS algorithm is even better since it generates solutions with comparable quality while using significantly less time due to its effective parallelization. The PGS algorithm outperforms the DSC algorithm in terms of both the solution quality and running time. An extra advantage of the PGS algorithm is its scalability, and with the use of more parallel processors, the algorithm can also be used for scheduling large task graphs.

Although the PGS algorithm has shown encouraging performance, further improvements are possible if we can determine an optimal set of control parameters, including crossover rate, mutation rate, population size, number of generations, and number of parallel processors used. However, finding an optimal parameters set for a particular genetic algorithm is hitherto an open research problem.

Acknowledgments

We would like to thank the referees for their constructive comments and suggestions which have greatly improved the quality of this paper.

References

[1] T.L. Adam, K.M. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Communications of the ACM*, vol. 17, Dec. 1974, pp. 685-690.

[2] Imtiaz Ahmad and M.K. Dhodhi, "Multiprocessor Scheduling in a Genetic Paradigm," *Parallel Computing*, vol. 22, no. 3, Mar. 1996, pp. 395-406.

[3] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," *International Symposium on Parallel Architectures, Algorithms, and Networks*, Beijing, China, Jun. 1996, pp. 207-213.

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

[5] S. Ali, S.M. Sait, and M.S.T. Benten, "GSA: Scheduling and Allocation using Genetic Algorithm," Proceedings of *EURO-DAC'94*, 1994, pp. 84-89.

[6] W. Atmar, "Notes on Simulation of Evolution," *IEEE Trans. Neural Networks*, vol. 5, no. 1, Jan. 1994, pp. 130-147.

[7] R. Chandrasekharam, S. Subhramanian, and S. Chaudhury, "Genetic Algorithm for Node Partitioning Problem and Applications in VLSI Design," *IEE Proceedings*, vol. 140, no. 5, Sep. 1993, pp. 255-260.

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

[9] H.-C. Chou and C.-P. Chung, "Optimal Multiprocessor Task Scheduling Using Dominance and Equivalence Relations," *Computers Operations Research*, vol. 21, no. 4, 1994, pp. 463-475.

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

[11] L.D. Davis (Ed.), *The Handbook of Genetic Algorithms,* New York, Van N. Reinhold, 1991.

[12] H. El-Rewini and H.H. Ali, "On Considering Communication in Scheduling Task Graphs on Parallel Processors," *Journal of Parallel Algorithms and Applications*, vol. 3, 1994, pp. 177-191.

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

[14] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAG's on Multiprocessors," *Journal of Parallel and Distributed Computin*g, vol. 16, no. 4, Dec. 1992, pp. 276-291.

[15] M.A.C. Gill and A.Y. Zomaya, "Genetic Algorithms for Robot Control," Proc. 1995 *IEEE Int'l Conf. on Evolutionary Computation*, vol. 1, pp. 462-466.

[16] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading, Mass., 1989.

[17] J.J. Grefenstette, "Optimization of Control Parameters for Genetic Algorithms," *IEEE Trans. Systems, Man, and Cybernetics*, vol. SMC-16, no. 1, Jan./Feb. 1986, pp. 122-128.

[18] J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, Mich., 1975.

[19] E.S.H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans. Parallel and Distributed Sys.,* vol. 5, no. 2, Feb. 1994, pp. 113-120.

[20] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 19, no. 6, Nov. 1961, pp. 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 Journal of Computing,* vol. 18, no. 2, Apr. 1989, pp. 244-257.

[22] A. Kapsalis, G.D. Smith, V.J. Rayward-Smith, and T.C. Fogarty, "A Unified Paradigm for Parallel Genetic Algorithms," *Evolutionary Computing, AISB Workshop (Selected Papers)*, 1994, pp. 131-149.

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

[24] Y.-K. Kwok and I. Ahmad, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Technique," *submitted for publication.*

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

[26] C. McCreary, A.A. Khan, J.J. Thompson, and M.E. McArdle, "A Comparison of Heuristics for Scheduling DAG's on Multiprocessors," Proc. *Int. Parallel Processing Sym.,* 1994, pp. 446-451.

[27] H. Muhlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," Proc. *Int'l Conf. on Genetic Algorithms*, 1989, pp. 416-421.

[28] T.M. Nabhan and A.Y. Zomaya, "A Parallel Simulated Annealing Algorithm with Low Communication Overhead," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 12, Dec. 1995, pp. 1226-1233.

[29] C.H. Papadimitriou and M. Yannakakis, "Scheduling Interval-Ordered Tasks," *SIAM J. Computing,* 8, 1979, pp. 405-409.

[30] M. Rebaudengo and M. Sonza Reorda, "An Experimental Analysis of the Effects of Migration in Parallel Genetic Algorithms," Proceedings of *Euromicro Workshop on Parallel and Distributed Processing,* 1993, pp. 232-238.

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

[32] R. Sethi, "Scheduling Graphs on Two Processors," *SIAM Journal of Computing*, vol. 5, no. 1, Mar. 1976, pp. 73-82.

[33] K. Shahookar and P. Mazumder, "A Genetic Approach to Standard Cell Placement using Meta-Genetic Parameter Optimization," *IEEE Trans. CAD of Integrated Circuits and Systems,* vol. 9, no. 5, May 1990, pp. 500-511.

[34] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, pp.75-187, Feb. 1993.

[35] M. Srinivas and L.M. Patnaik, "Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms," *IEEE Trans. Sys., Man and Cybernetics*, vol. 24, no. 4, Apr. 1994, pp. 656-667.

[36] —, "Genetic Algorithms: A Survey," *IEEE Computer,* Jun. 1994, pp. 17-26.

[37] R. Tanese, "Parallel Genetic Algorithm for a Hypercube," Proc. *Int'l Conf. on Genetic Algorithms*, 1987, pp. 177-183.

[38] —, "Distributed Genetic Algorithms," Proc. *Int'l Conf. Genetic Alg.*, 1989, pp. 434-439.

[39] J. Ullman, "NP-Complete Scheduling Problems," *J. Comp. Sys. Sci.,* 10, 1975, pp. 384-393.

[40] L. Wang, H.J. Siegel, and V.P. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments," Proc. *5th Heterogeneous Computing Workshop*, 1996, pp. 72-85.

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

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