An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling

Annie S. Wu, Han Yu, Shiyuan Jin, Kuo-Chi Lin, and Guy Schiavone, *Member*, *IEEE*

Abstract—We have developed a genetic algorithm (GA) approach to the problem of task scheduling for multiprocessor systems. Our approach requires minimal problem specific information and no problem specific operators or repair mechanisms. Key features of our system include a flexible, adaptive problem representation and an incremental fitness function. Comparison with traditional scheduling methods indicates that the GA is competitive in terms of solution quality if it has sufficient resources to perform its search. Studies in a nonstationary environment show the GA is able to automatically adapt to changing targets.

Index Terms—Genetic algorithm, task scheduling, parallel processing.

1 Introduction

The problem of scheduling a set of dependent or independent tasks to be processed in a parallel fashion is a well-studied area. Examples of such problems include the scheduling of jobs onto a fixed set of machines in a manufacturing plant, the scheduling of aircraft takeoffs and landings onto one or more landing strips, and the scheduling of meeting rooms to multiple events of varying size and length. With the development of parallel computing, a new version of this problem has gained increasing attention. A program can be decomposed into a set of smaller tasks. These tasks are likely to have dependencies and, consequently, precedence requirements. The goal of a scheduler is to assign tasks to available processors such that precedence requirements between tasks are satisfied and the overall length of time required to execute the entire program, the *schedule length* or *makespan*, is minimized. This problem of scheduling of tasks to be executed on a multiprocessor computer is one of the most challenging problems in parallel computing.

We introduce a novel genetic algorithm (GA) approach to the problem of multiprocessor task scheduling. Two unique features distinguish this GA from a traditional GA algorithm. First, it uses a flexible representation style which allows the GA to evolve both the structure and the value of the solutions. This flexibility is expected to improve a GA's ability to identify and retain good building blocks. Second, this GA uses a dynamically incremental fitness function which starts out rewarding for simpler goals, gradually increasing the difficulty of the desired fitness values or goals until a full solution is found. As a result, our GA places no restrictions on the individuals that can be formed and does not require special operators or repair mechanisms to ensure validity. Rather, it attempts to give partial fitness for invalid individuals that contain some valid subsequences of tasks and encourages the formation of successively longer valid subsequences. In addition, the design of our algorithm and representation allows for dynamic target changes with no interruption to the learning process. This GA does not need to be reinitialized, adjusted, or even stopped when a target change occurs.

We compare the performance of the GA with that of three traditional scheduling methods: Insertion Scheduling Heuristic (ISH) [28], Duplication Scheduling Heuristic (DSH) [28], and Critical Path Fast Duplication (CPFD) [2]. ISH and DSH are well-known list scheduling heuristic methods [28], [37]. CPFD consistently outperformed other algorithms (including DSH) in a comparision of duplication-based scheduling algorithms [2]. DSH and CPFD allow task duplication in solutions; ISH does not. We compare the performance on the GA and these algorithms on nine problems that have been used in similar comparison studies. In addition, we investigate the behavior of our GA in nonstationary environments where a target goal changes over time.

2 Task Scheduling in Parallel Systems

Multiprocessor scheduling problems can be classified into many different classes based on characteristics of the program and tasks to be scheduled, the multiprocessor system, and the availability of information. El Rewini et al. [13] give a general taxonomy of scheduling problems and discusses differences between classes.

We focus on a deterministic scheduling problem in which there exist precedence relations among the tasks to be scheduled and in which task duplication is allowed. A *deterministic scheduling* problem is one in which all information about the tasks and their relations to each other, such as execution time and precedence relations, are known to the scheduling algorithm in advance. Such problems, also known as *static scheduling* problems, contrast to *nondeterministic scheduling* problems in which some information about tasks and their relations may be undeterminable until runtime, i.e., task execution time and precedence relations may be determined by data input.

- *A.S. Wu and H. Yu are with the School of Computer Science, University of Central Florida, Orlando, FL 32816-2362. E-mail: {aswu, hyu}@cs.ucf.edu.*
- *S. Jin and G. Schiavone are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816. E-mail: shiyuan_jin@yahoo.com and guy@ist.ucf.edu.*
- *K.-C. Lin is with the Institute for Simulation and Training, 3280 Progress Drive, Orlando, FL 32826. E-mail: klin@pegasus.cc.ucf.edu.*

Manuscript received 31 Mar. 2003; revised 25 Oct. 2003; accepted 5 Feb. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0046-0303.

Within the class of deterministic scheduling problems, this work focuses on problems with the following characteristics:

1. Precedence relations among the tasks exist. Precedence relations among tasks determine the order in which tasks must be performed.
2. Communication costs exist. Communication cost is the cost to transmit messages from a task on one processor to a succeeding task on a different processor. Communication cost between two tasks on the same processor is assumed to be zero.
3. Task duplication is allowed. The same task may be assigned to more than one processor to reduce communication costs and schedule length.
4. The multiprocessor system consists of a limited number of fully connected processors.

We represent a parallel program as a directed acyclic graph (DAG), $G = (V, E)$, where V is a set of nodes each of which represents a component subtask of the program and E is a set of directed edges that specify both precedence constraints and communication paths among nodes. Hereafter, we will use the terms *node* and *task* interchangeably. In the DAG model, each node label gives the execution time for the corresponding task and each edge label gives communication time required to pass data from one node to the next if the two nodes are assigned to different processors during program execution. A task cannot start until all of its predecessor tasks are complete.

Fig. 1 shows two example DAGs from [44]. We compare performance of the GA with that of ISH, DSH, and CPFD on these two problems as well as on the other problems listed in Table 1.

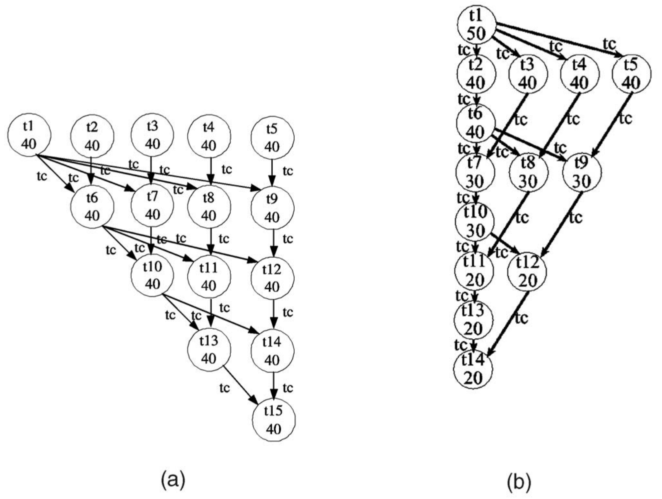


Fig. 1. Description of task dependencies for (a) the Gauss-Jordan algorithm and (b) LU decomposition.

3 Background

Kwok and Ahmad [31] present a comprehensive review and classification of deterministic or static scheduling algorithms. Among the most common methods is a class of methods called *list scheduling* techniques. List scheduling techniques assign a priority to each task to be scheduled then sort the list of tasks in decreasing priority. As processors become available, the highest priority task in the task list is assigned to be processed and removed from the list. If more than one task has the same priority, selection from among the candidate tasks is typically random. ISH [28] is a list scheduling heuristic that was developed to optimize scheduling DAGs with communication delays. ISH extends a basic list scheduling heuristic from Hu [26] by attempting to insert ready tasks into existing communication delay slots. DSH [28] improves upon ISH by using task duplication to reduce the start time of tasks within a schedule. DSH was one of the first approaches to use task duplication which reduces interprocessor communication time by scheduling tasks redundantly to multiple processors. CPFD [2] is another task duplication approach which prioritizes tasks based on whether or not they are part of the critical path (longest path) of a DAG.

GAs have been applied to the task scheduling problem in a number of ways [1], [4], [9], [12], [13], [25], [30], [44]. The two main approaches appear to be: methods that use a GA in combination with other list scheduling techniques and methods that use a GA to evolve the actual assignment and order of tasks into processors.

A number of studies have used GAs to determine task priorities for list scheduling techniques. In [1], [12], each individual in a GA is a vector of length n where n is the number of tasks to be scheduled. Each value of the vector represents a task priority for Task $t_i, i = 0, \ldots, n$. Tasks are ordered by increasing i. The initial population consists of one individual with priority value based on the longest path to an exit node on the DAG and the remaining individuals consisting of randomly permuted priority values from the first individual. Traditional crossover and mutation operators are used to generate new individuals. The job of the GA is to generate new combinations of priority values. Tasks are sorted based on priority value, then are scheduled using basic list scheduling techniques. Kwok and Ahmad [30] use

TABLE 1
Selected Test Problems

Problem	#Nodes	Source	Note
P1	15	Tsuchiya, et al. [44]	Gauss-Jordan alg. (comm. time: 25)
P2	15	Tsuchiya, et al. [44]	Gauss-Jordan alg. (comm. time: 100)
P3	14	Tsuchiya, et al. [44]	LU decomposition (comm. time: 20)
P4	14	Tsuchiya, et al. [44]	LU decomposition (comm. time: 80)
P5	15	Kruatrachue and Lewis [29]	
P6	17	Al-Mouhamed [3]	
P7	18	Wu and Gajski [49]	
P8	16	Wu and Gajski [49]	Laplace (comm. time: 40)
P9	16	Wu and Gajski [49]	Laplace (comm. time: 160)

a coarse-grained parallel GA in combination with a list scheduling heuristic. Individuals are again vectors of length n, where n is the number of tasks to be scheduled. The elements of a vector represent the tasks themselves and the order of the tasks gives the relative task priorities. As with other ordering problems such as the Travelling Salesman Problem, a number of order-based crossover operators are discussed. Mutation involves swapping tasks.

Alternatively, GAs have also been used to directly evolve task assignment and order in processors. Hou et al. [25] use a GA to evolve individuals consisting of multiple lists, with each list representing the tasks assigned to one processor. Crossover exchanges tasks between corresponding processors from two different individuals. Mutation exchanges tasks within a single individual. This approach restricts the actions of genetic operators to ensure the validity of evolved individuals. As a result, some parts of the search space may be unreachable. Correa et al. [9] improve upon Hou's method to allow the entire search space to be searched. Tsuchiya et al. [44] implement a GA scheduler that allows task duplication: One task may be assigned to multiple processors. They compare their GA to DSH and show that the GA is able to find comparable or better solutions. All of these GA approaches require special methods to ensure the validity of the initial population and to ensure the validity of offspring generated by crossover and mutation. In other words, all individuals generated by these systems must represent "executable" schedules. Zomaya et al. [50] incorporate heuristics in the generation of the initial population of a GA and perform a thorough study of how GA performance varies with changing parameter settings.

4 Algorithm Design

Many GA practitioners have experienced the "you get what you ask for" lesson. A carefully designed GA stubbornly refuses to find useful solutions. Closer examination reveals that the error is human; that the GA is actually giving you what you asked for, but you asked for the wrong thing. Typical GA designs incorporate a large number of arbitrary human decisions which can potentially bias the algorithm's performance. For example, Hou's method [25] uses restricted genetic operators which was found to render parts of the solution space unreachable and the order in which tasks are specified in [1], [12] affects the likelihood that two tasks will be crossed over.

Keeping this lesson in mind, we attempt to minimize the amount of arbitrary human input in our GA design, particularly in our problem representation and fitness function. We implement a novel GA approach for scheduling tasks for parallel execution on multiprocessor systems. Our GA extends the traditional GA [24], [18] in two key ways.

First, we use a dynamically adaptive representation which allows a GA to evolve both the structure and value of the solutions. Our problem representation is variable in length, uses a location independent encoding, and may contain noncoding regions (regions which do not contribute to encoding a solution). Both valid and invalid individuals may evolve.

Second, we use a dynamically adaptive, incremental fitness function which initially rewards for simple goals and gradually increases the difficulty of the goals over the generations. Fitness is evaluated in the same way, regardless of whether an individual encodes a valid or invalid solution. Given the number of possible orderings of tasks in processors, the percentage of valid orderings is very small. If a GA is not restricted to only work with valid individuals, the chance of randomly finding a valid ordering, let alone a good valid ordering, may be very low. Restrictions, however, may introduce unexpected biases in the system and may require extensive revision with each new problem. Instead of using special operators or repair mechanisms to restrict a system to only generate valid individuals, our GA attempts to give partial fitness for invalid individuals that contain some valid subsequences of tasks to encourage the formation of successively longer valid subsequences. Previous work has shown that gradually increasing the difficulty of a GA fitness function can result in the formation of more complex solutions [36].

```
procedure GA
  {
  initialize population;
  while termination condition not satisfied do
     {
     evaluate current population;
     select parents;
     apply genetic operators to parents to create offspring;
     set current population equal to the new offspring population;
     }
  }
```

Fig. 2. Basic steps of a typical genetic algorithm.

The basic algorithm is the same as a traditional GA as shown in Fig. 2. Details that are specific to our system are described below.

4.1 Problem Representation

The importance of tightly linked or compactly encoded building blocks in a GA representation has long been recognized [24], [18], [17]. Compactly arranged building blocks (building blocks with low defining length) are expected to be more likely to be transmitted as a whole by the genetic operators during a reproduction event [14]. Location independent problem representations, where the information content is not fixed at specific locations on a GA individual, have been proposed in a number of studies as a way to help a GA identify and maintain tightly linked building blocks. Such representations allow for rearrangement of encoded information [7], [5], [16], [19], [23], [35], [40], [46], [48], overlapping encodings which can be more space efficient [7], [42], [45], and the appearance of noncoding regions which affects crossover probability [7], [32], [15], [33], [38], [43], [46], [47]. In some location independent representations, the arrangement of encoded information will determine what is expressed [19], [23], [35] even though the actual encoded content is not determined by its location. We use such a representation here: The meaning of an encoded element is independent of its location on an individual, but its location determines whether or not it is expressed.

Each individual in a GA population consists of a vector of cells. We define a *cell* to be a task and processor pair: (t, p). Each cell indicates that Task t is assigned to be processed on Processor p. The number of cells in an individual may vary, so individuals in a GA population will vary in length. Fig. 3 shows an example individual. The first cell of this individual assigns Task 4 to Processor 1, the next cell assigns Task 2 to Processor 4, etc. This representation requires that the number of processors and number of tasks to be performed are known in advance. The problem itself defines the number of tasks to be performed and their dependencies on each other. We

```
(4,1)(2,4)(7,3)(2,3)(4,1)(5,4)(6,3)(1,1)(3,2)
```

Fig. 3. An example individual.

assume that the number of available processors is also defined in advance.

The cells on an individual determine which tasks are assigned to which processors. The order in which the cells appear on an individual determines the order in which the tasks will be performed on each processor. Individuals are read from left to right to determine the ordering of tasks on each processor. For example, the individual shown in Fig. 3 results in the processor assignments shown in Fig. 4. Invalid task orderings will have their fitness value penalized by the fitness function.

The same task may be assigned more than once to different processors. The individual in Fig. 3 assigns Task 2 to processors 3 and 4. Tasks cannot be assigned to the same processor more than once. If a task-processor pair appears more than once on an individual, only the first (leftmost, since individuals are read from left to right) pair is active. Any remaining identical pairs are essentially noncoding regions. In the example from Fig. 3, the second instance of `(4,1)` is not scheduled into the processor lists in Fig. 4.

As each (task, processor) pair is read from left to right, all active pairs are placed into FIFO queues based on their processor specification. The content of each queue indicates the tasks that will be performed on each processor. The order of the tasks in each queue indicates the order in which the tasks are assigned to be executed on each processor. Thus, the order in which tasks will be performed on each processor depends on the order in which the task-processor pairs appear on an individual.

The initial population is initialized with randomly generated individuals. Each individual consists of exactly one copy of each task. As a result, the length of all individuals in an initial population is equal to the number of tasks in the target DAG. Each task is randomly assigned to a processor.

4.2 Genetic Operators

Slight modifications to crossover and mutation are necessary to work with this representation. The modified versions of these genetic operators are described here.

4.2.1 Crossover

Recall that each individual consists of a vector of task-processor pairs or cells. Crossover exchanges substrings of cells between two individuals. This allows the GA to explore new solutions while still retaining parts of previously discovered solutions.

Processor 1	Task 4	Task 1	
Processor 2	Task 3		
Processor 3	Task 7	Task 2	Task 6
Processor 4	Task 2	Task 5	

Fig. 4. Assignment of tasks from the individual in Fig. 3.

All experiments described here use random one-point crossover. Random crossover involves two parent individuals. A crossover point is randomly chosen for each parent. The segments to the right of the crossover points are exchanged to form two offspring. Fig. 5 shows an example of random crossover. The crossover rate gives the probability that a pair of parents will undergo crossover. In addition, if a crossover operation generates an offspring individual that exceeds the maximum allowed genome length, crossover does not occur. Parents that do not crossover transform unchanged into offspring. Parents that do not crossover may still undergo mutation.

4.2.2 Mutation

The mutation rate indicates the probability that a cell will be changed. As a result, the expected number of mutations per individual is equal to the mutation rate multiplied by the length of an individual. If a cell is selected to be mutated, then either the task number or the processor number of that cell will be randomly changed.

4.3 Fitness Function

A number of factors are expected to contribute to the fitness of an individual. The fitness function separates the evalution into two parts. The first part, $task\_fitness$, focuses on ensuring that all tasks are performed and scheduled in valid orders. The second part, $processor\_fitness$, attempts to minimize processing time. The actual fitness, $fitness$, of a GA individual is a weighted sum of the above two partial fitness values.

4.3.1 Calculating $task\_fitness$

The $task\_fitness$ component of the fitness function evaluates whether all tasks are represented and in valid order. A pair of tasks is independent if neither task relies on the data output from the other task for execution. The scheduling of a pair of tasks to a single processor is valid if the pair is independent or if the order in which they are assigned to the processor matches the order of their dependency. The scheduling of a group of tasks to a single processor is valid

```
Randomly select parent 1 crossover point: 2
Randomly select parent 2 crossover point: 4
      Parent 1                        (4,1)(2,4)| (3,3)(2,3)(5,4)(1,1)(3,2)
      Parent 2             (4,3)(3,3)(5,2)(3,4)| (2,4)(3,3)
Random crossover produces
      Offspring 1                     (4,1)(2,4)| (2,4)(3,3)
      Offspring 2          (4,3)(3,3)(5,2)(3,4)| (3,3)(2,3)(5,4)(1,1)(3,2)
```

Fig. 5. Random one-point crossover randomly selects crossover points on each parent and exchanges the left segments to form offspring.

if the order of every pair of tasks in the group is valid. A solution is valid if all of its processor schedules are valid.

Because of the complexity of the solutions, we develop an incremental fitness function that changes over time. We initially reward for finding short valid sequences of tasks. Over time, we increase the length of the sequences that can be rewarded, encouraging the GA to find and maintain longer valid sequences. Eventually, the valid sequences will be long enough that the individuals will represent full valid solutions. This strategy rewards for small steps toward the goal, to encourage the algorithm to find the complete goal.

The $task\_fitness$ component of an individual's fitness is based on two main components: the percentage of valid sequences of a given length and the percentage of the total tasks specified by an individual. Initially, the fitness function will reward for short sequences of valid tasks. A sequence of tasks is valid if the tasks in the sequence are arranged in a valid chronological order. When the average fitness of the GA population exceeds a threshold fitness, the length of the sequence for which the GA searches is increased, thus increasing the difficulty of the fitness function.

Calculating raw fitness: The raw fitness of an individual reflects the percentage of sequences of a given length in an individual that are valid sequences. For example, suppose we are working with Problem P4. Processor 3 in Fig. 4 has been assigned three tasks. If the current sequence length is two, Processor 3 contains two sequences of length two, but only one *valid* sequence of length two, the sequence `Task2-Task6`. The sequence `Task7-Task2` is not a valid sequence because Task 7 cannot be executed before Task 2.

Assume that the problem to be solved involves P processors and T tasks. Evolution will occur in eras, $era = 0, 1, 2, \ldots, E$. Initially, $era = 0$. The maximum era count, $E \leq T$, is a user defined parameter value. The era counter, era, is increased when the average population fitness exceeds a user defined threshold, *thresh*, and when the number of individuals with the current maximum fitness exceeds a user defined threshold, $thresh\_maxfit$. Unless otherwise specified, we use $thresh = 0.75$ and $thresh\_maxfit = 0.1$.

Let $numtasks(p), p = 1, \ldots, P$ indicate the number of tasks assigned to processor *p*. To calculate the raw fitness of a processor, we need to consider two things: the first $era + 1$ (or fewer) tasks assigned to the processor and all task sequences of length $era + 2$. The first component is important because as era increases, the likelihood of processors containing fewer than $era + 2$ tasks increases. We need to reinforce the GA for these shorter sequences in order for them to eventually build up to the measured sequence length.

We will first determine the contribution of the first $era + 1$ or fewer tasks in a processor. Let

$$subseq(p) = \begin{cases} 1 & \text{if } numtasks(p) > 0 \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

and let

$$valseq(p) = \begin{cases} 1 & \text{if the first } era + 1 \text{ or fewer tasks in Processor } p \\ & \text{are in valid order} \\ 0 & \text{otherwise.} \end{cases} \qquad (2)$$

Equations (1) and (2) refer to individual processors. To calculate the contribution over all processors (the contribution for the entire individual), we let

$$Subseq = \sum_{p=1}^{P} subseq(p),$$

$$Valseq = \sum_{p=1}^{P} valseq(p).$$

We will next determine the contribution of all sequences of length $era + 2$ in a processor. Let

$$s(p) = \#\ \text{sequences of length } era + 2 \text{ in Processor } p \qquad (3)$$

and let

$$v(p) = \#\ \text{valid seq. of length } era + 2 \text{ in Processor } p. \qquad (4)$$

Combining (3) and (4) to determine the contribution over all processors, we let

$$S = \sum_{p=1}^{P} s(p), \quad V = \sum_{p=1}^{P} v(p).$$

The $raw\_fitness$ for an individual is then calculated with the following equation:

$$raw\_fitness = \frac{Valseq + V}{Subseq + S}. \qquad (5)$$

Calculating the task ratio: In addition to encouraging the system to find valid sequences of tasks, we also want to encourage the system to include at least one copy of each task in each solution. We define the $task\_ratio$ to be the percentage of distinct tasks from the total tasks in the problem that are represented in an individual. The $task\_ratio$ is calculated with the following equation:

$$task\_ratio = \frac{number\ of\ distinct\ tasks\ specified\ on\ an\ individual}{total\ number\ of\ tasks\ in\ the\ problem}. \qquad (6)$$

This factor penalizes solutions that do not contain at least one copy of every task. Once all tasks are represented in an individual, this penalty becomes null.

Effective fitness: The effective $task\_fitness$ of an individual is the product of (5) and (6).

$$task\_fitness = raw\_fitness * task\_ratio. \qquad (7)$$

This value makes up the first component of the fitness of a GA individual.

4.3.2 Calculating $processor\_fitness$

The $processor\_fitness$ component of the fitness function encourages the formation of shorter solutions that minimize makespan. As the length of a solution can only be measured if a solution exists, $processor\_fitness = 0$ for all individuals that do not encode valid solutions.

Suppose t is the runtime for a solution represented by an individual. Let $serial\_len$ equal the length of time required to complete all tasks serially on a single processor and let $super\_serial\_len = P * serial\_len$, where P equals the number of processors. Any reasonable solution should give $t \ll super\_serial\_len$, making $super\_serial\_len$ a safe but reasonable upper bound to solution runtime. The goal of the GA is to minimize t. The $processor\_fitness$ first calculates the difference between $super\_serial\_len$ and t then calculates what proportion of $super\_serial\_len$ this difference represents:

$$processor\_fitness = \frac{super\_serial\_len - t}{super\_serial\_len}.$$

As a result, $processor\_fitness$ is inversely proportional to t. As the runtime of a solution decreases, the amount that $processor\_fitness$ contributes to the individual's full fitness increases.

It is important to note that, although the theoretical maximum value of $processor\_fitness$ is 1.0, in practice, this value cannot be achieved. For $processor\_fitness$ to equal 1.0, the runtime, t, of a solution would have to be zero. Since all tasks obviously require nonzero runtime, t will never be zero for valid individuals.

4.3.3 Calculating $fitness$

The $task\_fitness$ component of the fitness function rewards for the formation of valid solutions. If an individual encodes a valid solution, the $processor\_fitness$ component of the fitness function rewards for shorter solutions. The full fitness of an individual is a weighted sum of the $task\_fitness$ and $processor\_fitness$:

$$fitness = (1 - b) * task\_fitness + b * processor\_fitness,$$

where $0.0 \leq b \leq 1.0$. Unless otherwise specified, we use $b = 0.2$.

If an individual encodes a valid solution, the above equation holds. It is important to note that, even though the theoretical maximum value of $fitness$ is 1.0, this value will never be achieved in practice because the $processor\_fitness$ component of the fitness function can never reach 1.0 in practice.

If an individual does not encode a valid solution, we are unable to evaluate $processor\_fitness$. As a result, $processor\_fitness = 0$ and

$$\begin{aligned} fitness &= (1 - b) * task\_fitness + b * 0 \\ &= (1 - b) * task\_fitness. \end{aligned}$$

The value of $fitness$ is returned to the GA by the fitness function as the fitness of an individual. The complexity of this fitness function (and this GA[1]) is $O(PT^2)$.

5 Experimental Details

The following parameter settings were empirically determined to be good values for our GA. Unless otherwise specified, we use the following values in our experiments:

Population size :	400
Crossover type :	Random one-point
Crossover rate :	0.8
Mutation rate :	0.005
Selection method :	Tournament
Stopping condition :	3,000 generations.

We use a variable length representation with a maximum length of two times the number of tasks in the problem, $2 \times T$.

TABLE 2
Minimum Makespan Found by ISH, DSH, CPFD, and GA

Test	ISH	DSH	CPFD	GA		
Problem				Best	Average	95% CI
P1	300	300	300	300	300	0
P2	500	400	400	400	430	5.28
P3	260	260	260	260	263.4	2.06
P4	400	330	330	330	370	6.41
P5	650	539	446	438	445.92	6.05
P6	41	37	37	37*	37.78*	0.24*
P7	450	370	330	350	380.6	4.76
P8	760	760	760	760	782.8	5.63
P9	1220	1030	1040	1040	1101.8	14.23

*CI = confidence interval. * In a second set of runs in which the population size was doubled, the GA found a minimum makespan of 36 and average makespan of 36.92 with a 95 percent confidence interval of 0.17.*

5.1 Comparison with Traditional List Scheduling Methods

We first compare the performance of the GA with that of ISH, DSH, and CPFD on nine problems. Table 2 shows the best solutions obtained for each problem by each method. Because the GA is a stochastic algorithm, we perform 50 runs for each problem and also report its average results. The GA outperforms traditional methods on one problem (P5), performs as well as the best traditional method on six problems, and achieves the second best performance on two problems (P7 and P9). Doubling the GA population size allows the GA to also outperform traditional methods on Problem P6. Results indicate that, given sufficient resources, the GA is able to equal or outperform traditional scheduling methods.

Interestingly, the data in Table 2 suggest that the advantages of task duplication in these scheduling methods are particularly noticeable on problems with longer communication times. Problems P1 and P2 share the same DAG and differ only in their communication times: Problem P2 has a significantly longer communication time than Problem P1. The same holds true for Problems P3 and P4 and Problems P8 and P9. Problems P2, P4, P5, P7, and P9 have communication times that are larger than the task execution times (significantly larger for P5) and show noticeable improvement when using methods that allow task duplication. Problems P1, P3, P6, and P8 have communication times that are equal or less than task execution times and show little improvement with the addition of task duplication.

An examination of scalability to larger problems finds that GA performance declines as the problem size increases. GAs tend to require larger populations to maintain performance as problem size increases, e.g., when P4 is scaled up to be a 27-node problem, a GA using population size 400 finds a minimum makespan of 680; a GA using population size 800 finds a minimum makespan of 650.

1. The computational effort of a GA is typically dominated by its fitness evaluation.

TABLE 3
Average Number of Generations and Average Clock Time (in Seconds) Using a GA

Test problem	Average (95% CI) generations to best solution	Average (95% CI) time to best solution	Average (95% CI) time for one run
P1	682.26 (191.30)	30.70 (11.01)	129.37 (11.18)
P2	1011.88 (219.07)	54.49 (13.37)	164.20 (14.54)
P3	934.34 (213.50)	28.15 (6.73)	95.05 (4.43)
P4	1333.36 (247.21)	60.78 (13.89)	140.17 (12.10)
P5	871 (180.38)	36.16 (7.74)	137.98 (12.39)
P6	1375.46 (246.84)	80.36 (16.92)	187.35 (15.52)
P7	1316.62 (248.24)	77.03 (14.46)	178.36 (14.90)
P8	1168.18 (195.33)	73.97 (12.77)	192.12 (12.98)
P9	1627.72 (237.77)	130.83 (20.15)	248.69 (28.43)

CI = confidence interval.

These results indicate that a GA requires sufficient resources in order to find good solutions.

A comparison of execution times finds that the cost for having sufficient resources is a longer execution time. ISH, DSH, and CPFD consistently post runtimes of less than one second for the problems that we tested. The GA requires significantly longer execution times. Table 3 gives the average number of generations and seconds to find a good solution using the GA. Traditional methods clearly outperform the GA in terms of execution time.

Fig. 6 shows an example of how a typical GA run proceeds. Fig. 6a shows the evolution of population fitness. The top line shows the best population fitness at each generation. The bottom line shows the average population fitness at each generation. The vertical lines indicate the generations at which the era counter is incremented. The start of each era is indicated at the top of the graph. The average population fitness climbs within each era. Each time the era counter is incremented, however, the difficulty level of the fitness function increases and the average fitness of the population drops. After about six eras in this run, there are apparently enough valid task sequences to allow the remaining eras to increment once per generation until the maximum $era = 15$ is reached. Fig. 6b shows the evolution of runtime or makespan in the same run. The minimum makespan can only be calculated from a valid solution. Early in the run, with lower values of era, valid solutions are found only sporadically. Over time, valid solutions are found more consistently and the minimum makespan decreases steadily.

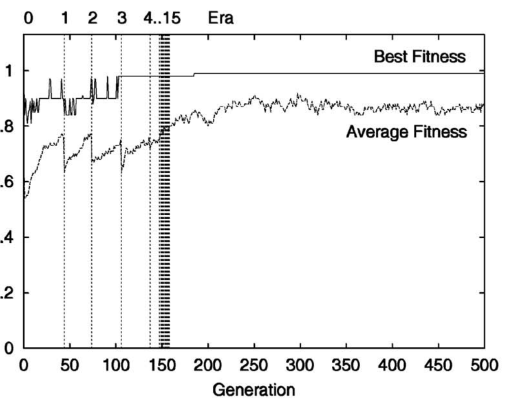
5.2 Comparison Using Heterogeneous Processors

We also compare the four algorithms in a more complex environment in which the processors are heterogeneous. Processors 1 and 3 remain unchanged, we double the processing time for processor 2, and we triple the processing time for processor 4. Table 4 compares the quality of the solutions found. Our GA exhibits the best performance on five of the problems. On Problems P3 and P4, CPFD and DSH, respectively, perform equally as well as the GA. On Problems P1 and P5, the GA comes in second to DSH. Among the traditional methods, DSH performs better than CPFD.

5.3 Performance Sensitivity to GA Fitness Function

The fitness function of a GA can have a significant impact on the effectiveness of the algorithm. Our fitness function has several parameters that can vary. We test the sensitivity of the GA to variations in these values. Specifically, we examine GA performance on problems P1 and P3 where $b \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$ and $thresh \in \{0.25, 0.5, 0.75, 1.0\}$.

Figs. 7 and 8 show the results for Problem P1. Similar results were obtained for Problem P3. Fig. 7 shows the percent of runs in each experiment that find at least one valid solution. Fig. 8a shows the minimum makespan achieved by the GA and Fig. 8b shows the minimum makespan found in the final generation averaged over



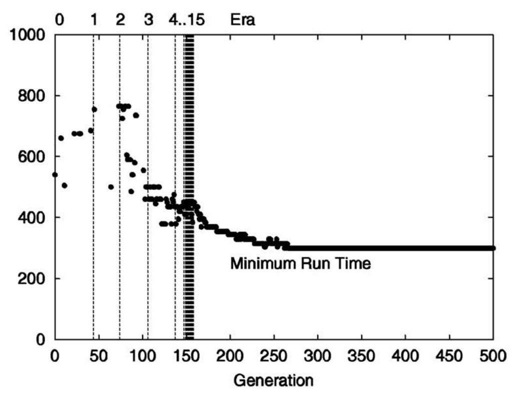


Fig. 6. Evolution of (a) population fitness and (b) minimum makespan in response to increasing eras.

TABLE 4
Minimum Makespan Found by ISH, DSH, CPFD, and GA on a Heterogeneous Problem

Test	ISH	DSH	CPFD	GA		
Problem				Best	Average	95% CI
P1	300	300	345	315	333.7	3.70
P2	500	440	460	420	462	8.91
P3	320	310	260	260	288.4	5.50
P4	400	350	360	350	396.9	8.13
P5	549	470	606	539	559.9	8.65
P6	46	42	43	40	42.94	0.60
P7	470	410	370	360	413.2	10.33
P8	840	840	860	810	889.4	14.40
P9	1220	1130	1210	1060	1187	23.95

CI = confidence interval.

50 runs. Data are obtained for each combination of b and $thresh$ values. The x-axis labels indicate the $b/thresh$ combination for each set of runs.

Intermediate values of b consistently produce good performance, finding a minimum makespan of 300 and average best makespans of 300 or slightly above. The narrow confidence intervals in Fig. 8b indicate that most runs are able to find a best solution at or close to 300. Varying $thresh$ appears to have little impact on the results.

Extreme values of b have a noticeable negative impact on GA performance. Setting $b = 0.0$ produces suboptimal, though still respectable, results, with minimum makespans as high as 315. When $b = 0.0$, the fitness function consists of only the $task\_fitness$ portion which focuses only on finding valid solutions (it rewards for valid substrings of tasks and rewards for having at least one copy of each task). The length (makespan) of a solution is irrelevant as the fitness function does not give any reward for shorter solutions. As a result, the GA is able to find solutions; however, the lack of pressure for smaller solutions is apparent as all of the GA runs with $b = 0.0$ find significantly longer solutions that those runs with intermediate values of b. The larger 95 percent confidence interval indicates a wider range of makespan values found.

Setting $b = 1.0$ makes it difficult for the GA to find valid solutions. Fig. 7 shows that the GA is unable to find a valid solution in every run when $b = 1.0$. When valid solutions are found, however, the GA finds good solutions, although not as

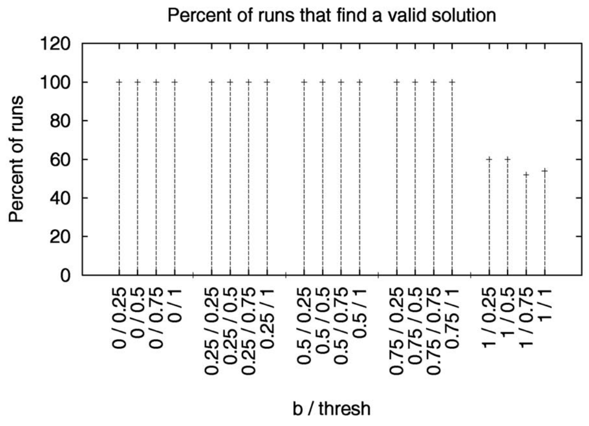


Fig. 7. Problem P1: Percent of runs that find a valid solution. X-axis indicates $b/thresh$ values.

consistently as with intermediate values of b. When $b = 1.0$, the fitness function consists only of the $processor\_fitness$ component which is activated only if an individual encodes a valid solution. Only when an individual encodes a valid solution will the fitness function return a nonzero value. As a result, there is no feedback for partial solutions. With no fitness reward for partial solutions consisting of short valid task orderings, the GA has difficulty finding valid solutions. Thus, rewarding for valid partial orderings appears to be an important component of the algorithm's success.

The $thresh$ parameter appears to have little impact on quality of solutions found for intermediate values of b. For $b = 0.0$ and $b = 1.0$, performance declines with increasing values of $thresh$.

5.4 Performance in Nonstationary Environments

The results from Section 5.1 indicate that, while a GA can find very competitive solutions, its execution times are likely to be longer than traditional methods. Why then would one choose to use a GA over faster traditional methods? We expect the strengths of this GA approach to be in its flexibility and adaptability in nonstationary environments [6], [8], [11], [10], [20], [21], [22], [27], [34], [39], [41]. Nonstationary problem environments are those in which the desired solution changes over time. Such environments can be difficult for traditional scheduling

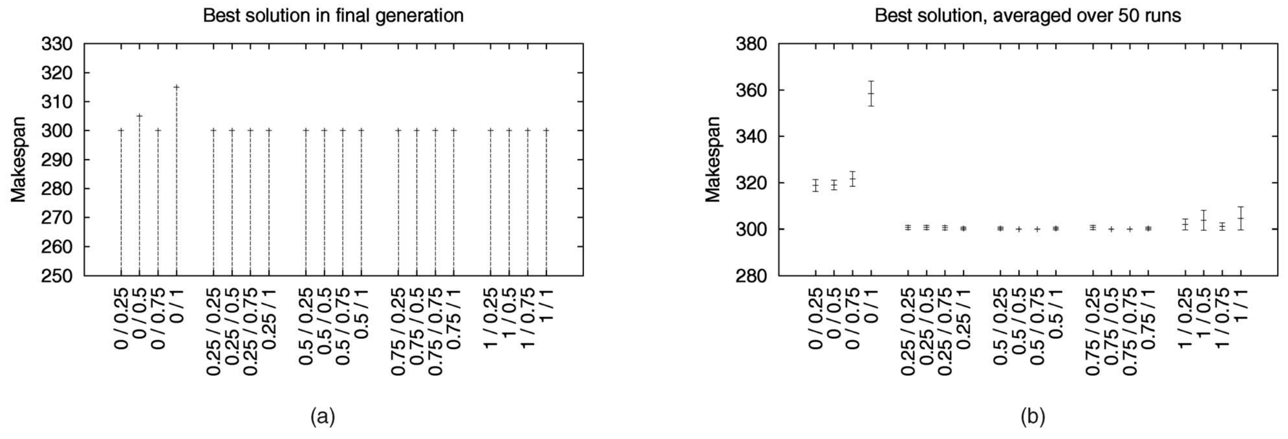


Fig. 8. Problem P1: X-axis indicates $b/thresh$ values. (a) Minimum makespan. (b) Average best makespan averaged over 50 runs* with 95 percent confidence intervals. *When $b = 1.0$, not all 50 runs are able to find valid solutions. The average values shown are calculated only from those runs that do find valid solutions.

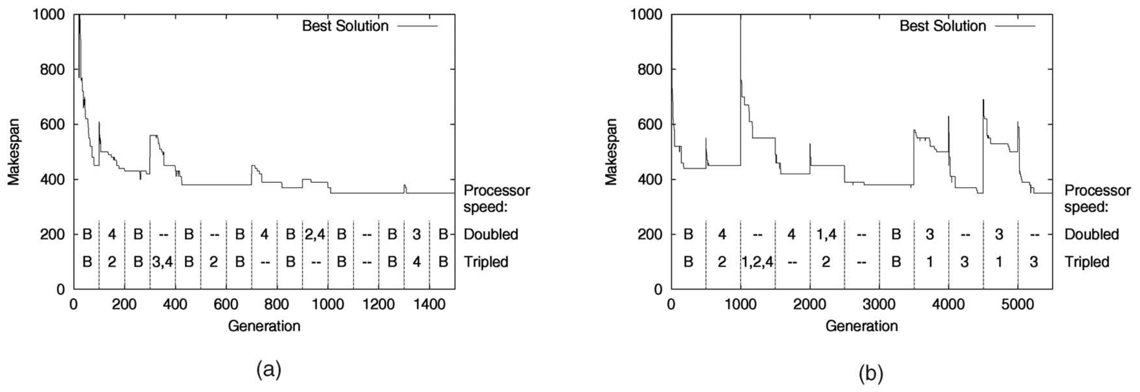


Fig. 9. Two example GA runs. Evolution of best solution in a nonstationary environment. "B" indicates base target. Integer values indicate modified processors.

algorithms; most must be restarted, many reconfigured or reprogrammed, for each new situation. We believe that the flexibility of our GA and its representation will allow it to automatically adapt to changes in the fitness evaluation with no change or interruption to the algorithm itself.

The experiments in this section investigate the GA's ability to adapt in a nonstationary environment where processor speeds can change over time. Once a GA run has started, no human intervention or interruptions are allowed; the GA must adapt automatically to changes in the target problem. Within a multiprocessor system, processor loads may vary depending on the number of tasks under execution and how they are distributed among the processors. As a processor's load increases, its execution speed is expected to decrease. Ideally, as processor loads change, the system will automatically redistribute workload among the processors to take advantage of processors with low load and minimize assignments to processors with high load. An algorithm that is able to adapt automatically to such changes can significantly improve the efficiency of managing a multiprocessor system. In addition, the underlying problem has now changed and become more difficult: multiprocessor task scheduling for *heterogeneous* processors.

A GA run begins by finding a task schedule for four identical processors at minimal load. We call this situation the *base target*. We change the target problem by increasing processor speeds to double and triple the minimal speed. At fixed intervals, each processor has a 30 percent chance of doubling its speed and a 20 percent chance of tripling its speed. We call these situations *modified targets*. We test intervals of $I = \{100, 500\}$ generations.

Fig. 9 shows, in two example runs, how the evolved solutions of a GA change as processor speed changes. "B" indicates base target. Integer values indicate modified processors. Processors are randomly selected to be modified. The optimal makespan for the base target is 330. Fig. 9a shows an example run with $I = 100$ where each interval with a modified target is followed by an interval with the base target. Fig. 9b shows an example run with $I = 500$ where multiple consecutive intervals can have modified targets. The base target is assigned to generations 0-500 and generations 3,000-3,500 to provide a baseline comparison.

Results indicate that this GA approach is able to automatically adapt to changes in the target solution. In both examples, the GA continues to improve the solutions generated throughout a run. As expected, makespan increases sharply after a target change to a modified target, but solutions immediately begin to improve. Fig. 9b shows less stable solutions than Fig. 9a. We speculate that there are two potential causes for this difference. First, longer intervals of 500 generations give the GA more time to optimize solutions and converge the population for the current target. As a result, it is less likely that the population will have solutions that perform well for other modified targets. Second, not having a base target in between each modified target gives the GA no time to "neutralize" its solutions between modified targets. Overall, the GA is able to evolve near optimal solutions in both experiments, even when processor speeds are increased.

6 Conclusions

We describe a novel GA approach to solving the multiprocessor task scheduling problem. We improve upon previous GA applications to this problem by eliminating the need for special operators or repair mechanisms to ensure formation of valid solutions and using partial solutions to direct the GA search. Our approach extends a traditional GA in two ways: it uses a variable length, location independent problem representation and it uses a time varying incremental fitness function that rewards for increasingly complex partial solutions. We empirically study the strengths and weaknesses of this method and compare its performance to that of three well-known traditional algorithms for solving this problem. In addition, we investigate the adaptability of this method in nonstationary environments.

We first compare the performance of our GA with three traditional scheduling techniques: ISH, DSH, and CPFD. The GA outperforms traditional methods on two out of nine problems. It performs comparably on the remaining problems. The GA's ability appears to depend on sufficient availability of resources. As problem size increases, the GA requires a larger population or more generations to maintain performance. As we increase the resources (e.g., population size) available to the GA, it is able to find better solutions. The trade off for the increased resources used by the GA is a significantly longer execution time than traditional methods. Results also indicate that algorithms that allow task duplication have an advantage over algorithms that do not allow task duplication, particularly when communication costs are high.

We repeat the same set of experiments in a more complex environment consisting of heterogeneous processors. The GA achieves the top performance on seven out of nine problems and is second-best on two problems. Overall, the GA appears to be the most flexible algorithm for problems using heterogeneous processors. We speculate that heterogeneous processors make it more difficult for list scheduling algorithms to accurately estimate task priority.

We next examine the GA's sensitivity to fitness function parameters. We use a two part fitness function which 1) encourages the formation of valid solutions by rewarding incrementally larger partial solutions over time and 2) encourages the formation of optimal solutions by rewarding for shorter valid solutions. Experimental results indicate that both components are necessary for the evolution of optimal solutions. Removing the first component from the fitness function removes the partial credit for partial solutions which makes it difficult for the GA to find valid solutions at all. Rewards for partial solutions appear to be important for directing the GA's search. Removing the second component removes the selection pressure for shorter makespans resulting in valid solutions with slightly longer makespans.

Finally, we examine the GA's behavior in a nonstationary environment consisting of dynamically changing heterogenous processors. We believe that the strength of our approach is that it can be applied to different scheduling problems with no reconfiguration. As a result, it should be able to dynamically monitor the progress of a parallelized program on a multiprocessor system and reschedule the program's tasks as necessary to maximize processor availability. We randomly select processors and double or triple their processing speed for intervals of time. These modified targets may or may not be interspersed with base targets in which all processors have normal processing speed (and, hence, are homogeneous). Results indicate that the GA is able to adapt automatically to changes in the problem to be solved. Although performance decreases after a target change, the GA immediate begins to improve solutions and is ultimately able to find near optimal solutions even when one or more processors are significantly slower than normal.

The advantages of the GA approach presented here are that it is simple to use, requires minimal problem specific information, and is able to effectively adapt in dynamically changing environments. The primary disadvantage of this approach is that it has a longer execution time than many traditional scheduling methods.

References

[1] I. Ahmad and M.K. Dhodhi, "Multiprocessor Scheduling in a Genetic Paradigm," *Parallel Computing,* vol. 22, pp. 395-406, 1996.

[2] I. Ahmad and Y. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 9, pp. 872-892, Sept. 1998.

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

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

[5] J.D. Bagley, "The Behavior of Adaptive Systems which Employ Genetic and Correlation Algorithms," PhD dissertation, Univ. of Michigan, 1967.

[6] J. Branke, "Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems," *Proc. Congress on Evolutionary Computation,* pp. 1875-1882, 1999.

[7] D.S. Burke, K.A. De Jong, J.J. Grefenstette, C.L. Ramsey, and A.S. Wu, "Putting More Genetics into Genetic Algorithms," *Evolutionary Computation,* vol. 6, no. 4, pp. 387-410, 1998.

[8] H.G. Cobb, "An Investigation into the Use of Hypermutation as an Adaptive Operator in Genetic Algorithms Having Continuous, Time-Dependent Nonstationary Environments," Technical Report AIC-90-001, Naval Research Lab., Washington DC, 1990.

[9] R.C. Correa, A. Ferreira, and P. Rebreyend, "Scheduling Multiprocessor Tasks with Genetic Algorithms," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 8, pp. 825-837, 1999.

[10] K.A. De Jong, "An Analysis of the Behavior of a Class of Genetic Adaptive Systems," PhD dissertation, Univ. of Michigan, 1975.

[11] K. Deb and D.E. Goldberg, "An Investigation of Niche and Species Formation in Genetic Function Optimization," *Proc. Int'l Conf. Genetic Algorithms,* pp. 42-50, 1989.

[12] M.K. Dhodhi, I. Ahmad, and I. Ahmad, "A Multiprocessor Scheduling Scheme Using Problem-Space Genetic Algorithms," *Proc. IEEE Int'l Conf. Evolutionary Computation,* pp. 214-219, 1995.

[13] H. El-Rewini, T.G. Lewis, and H.H. Ali, *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[14] L.J. Eshelman, R.A. Caruana, and J.D. Schaffer, "Biases in the Crossover Landscape," *Proc. Third Int'l Conf. Genetic Algorithms,* pp. 10-19, 1989.

[15] S. Forrest and M. Mitchell, "Relative Building-Block Fitness and the Building-Block Hypothesis," *Foundations of Genetic Algorithms 2,* pp. 109-126, 1992.

[16] R.W. Franceschini, A.S. Wu, and A. Mukherjee, "Computational Strategies for Disaggregation," *Proc. Ninth Conf. Computer Generated Forces and Behavioral Representation,* 2000.

[17] D.R. Frantz, "Non-Linearities in Genetic Adaptive Search," PhD dissertation, Univ. of Michigan, Ann Arbor, 1972.

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

[19] D.E. Goldberg, B. Korb, and K. Deb, "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex Systems,* vol. 3, pp. 493-530, 1989.

[20] D.E. Goldberg and R.E. Smith, "Nonstationary Function Optimization Using Genetic Algorithms with Dominance and Diploidy," *Proc. Int'l Conf. Genetic Algorithms,* pp. 59-68, 1987.

[21] J.J. Grefenstette, "Genetic Algorihtms for Changing Environments," *Proc. Conf. Parallel Problem Solving from Nature (PPSN),* vol. 2, pp. 137-144, 1992.

[22] "Evolvability in Dynamic Fitness Lanscapes: A Genetic Algorithm Approach," *Proc. Conf. Congress on Evolutionary Computation (CEC),* pp. 2031-2038, 1999.

[23] G.R. Harik, "Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms," PhD dissertation, Univ. of Michigan, Ann Arbor, 1997.

[24] J.H. Holland, *Adaptation in Natural and Artificial Systems.* Univ. of Michigan Press, 1975.

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

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

[27] H. Kita, Y. Yabumoto, N. Mori, and Y. Nishikawa, "Multi-objective Optimization by Means of the Thermodynamical Genetic Algorithm," *Proc. Conf. Parallel Problem Solving from Nature,* pp. 504-512, 1996.

[28] B. Kruatrachue and T.G. Lewis, "Duplication Scheduling Heuristic, a New Precedence Task Scheduler for Parallel Systems," Technical Report 87-60-3, Oregon State Univ., 1987.

[29] B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software,* vol. 5, no. 1, pp. 23-32, 1988.

[30] Y. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm," *J. Parallel and Distributed Computing,* vol. 47, no. 1, pp. 58-77, 1997.

[31] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys,* vol. 31, no. 4, pp. 406-471, 1999.

[32] W.B. Langdon and R. Poli, "Fitness Causes Bloat," *Soft Computing in Eng. Design and Manufacturing,* pp. 13-22, 1997.

[33] J.R. Levenick, "Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology," *Proc. Fourth Int'l Conf. Genetic Algorithms,* pp. 123-127, 1991.

[34] W. Liles and K. De Jong, "The Usefulness of Tag Bits in Changing Envrionments," *Proc. Congress on Evolutionary Computation,* pp. 2054-2060, 1999.

[35] F.G. Lobo, K. Deb, D.E. Goldberg, G. Harik, and L. Wang, "Compressed Introns in a Linkage Learning Genetic Algorithm," *Proc. Third Conf. Genetic Programming,* pp. 551-558, 1998.

[36] J.D. Lohn, G.L. Haith, S.P. Columbano, and D. Stassinopoulos, "A Comparison of Dynamic Fitness Schedules for Evolutionary Design of Amplifiers," *Proc. First NASA/DoD Workshop Evolvable Hardware,* pp. 87-92, 1999.

[37] B.S. Macey and A.Y. Zomaya, "A Performance Evaluation of CP List Scheduling Heuristics for Communication Intensive Task Graphs," *Proc. Joint 12th Int'l Parallel Processing Symp. and Ninth Symp. Parallel and Distributed Prog.,* pp. 538-541, 1998.

[38] H.A. Mayer, "ptGAs: Genetic Algorithms Using Promoter/ Terminator Sequences," PhD dissertation, Univ. of Salzburg, Germany, 1997.

[39] N. Mori, H. Kita, and Y. Nishikawa, "Adaptation to a Changing Environment by Means of the Thermodynamical Genetic Algorithm," *Proc. Conf. Parallel Problem Solving from Nature,* pp. 513-522, 1996.

[40] J. Paredis, "The Symbiotic Evolution of Solutions and Their Representations," *Proc. Sixth Int'l Conf. Genetic Algorithms,* pp. 359-365, 1995.

[41] R.E. Smith, "Diploidy Genetic Algorithms for Search in Time Varying Environments," *Proc. Ann. Southeast Regional Conf. ACM,* pp. 175-179, 1987.

[42] T. Soule and A.E. Ball, "A Genetic Algorithm with Multiple Reading Frames," *Proc. Genetic and Evolutionary Computation Conf.,* 2001.

[43] T. Soule, J.A. Foster, and J. Dickinson, "Code Growth in Genetic Programming," *Genetic Programming,* Cambridge, Mass.: MIT Press, pp. 215-233, 1996.

[44] T. Tsuchiya, T. Osada, and T. Kikuno, "Genetic-Based Multiprocessor Scheduling Using Task Duplication," *Microprocessors and Microsystems,* vol. 22, pp. 197-207, 1998.

[45] A.S. Wu, "Non-Coding Segments and Floating Building Blocks for the Genetic Algorithm," PhD dissertation, Univ. of Michigan, Ann Arbor, 1995.

[46] A.S. Wu and I. Garibay, "The Proportional Genetic Algorithm: Gene Expression in a Genetic Algorithm," *Genetic Programming and Evolvable Machines,* vol. 3, no. 2, pp. 157-192, 2002.

[47] A.S. Wu and R.K. Lindsay, "Empirical Studies of the Genetic Algorithm with Non-Coding Segments," *Evolutionary Computation,* vol. 3, no. 2, pp. 121-147, 1995.

[48] A.S. Wu and R.K. Lindsay, "A Comparison of the Fixed and Floating Building Block Representation in the Genetic Algorithm," *Evolutionary Computation,* vol. 4, no. 2, pp. 169-193, 1996.

[49] 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, pp. 330-343, 1990.

[50] A.Y. Zomaya, C. Ward, and B. Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 8, pp. 795-812, 1999.

Annie S. Wu received the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor. She is an assistant professor in the School of Computer Science and director of the Evolutionary Computation Laboratory at the University of Central Florida (UCF), Orlando.

Han Yu received the BS degree from Shanghai Jiao Tong University and the MS degree from UCF. He is a PhD student in the School of Computer Science at the University of Central Florida (UCF).

Shiyuan Jin received the BS degree from the Zhejiang Institute of Technology and the MS degree from the University of Central Florida (UCF), Orlando. He is a PhD student in the Department of Electrical and Computer Engineering (ECE) at UCF.

Kuo-Chi Lin received the PhD degree in aerospace engineering from the University of Michigan. He holds a joint faculty position between the Institute for Simulation and Training and the Department of Aerospace Engineering at the University of Central Florida (UCF), Orlando. He has more than 25 years of experience in the field of modeling and simulation. He is currently working on dynamic data-driven application systems.

Guy Schiavone received the PhD degree in engineering science from Dartmouth College in 1994, and the BEEE degree from Youngstown State University in 1989. He is an assistant professor with the Department of Electrical and Computer Engineering (ECE) at the University of Central Flodida (UCF), Orlando. He held previous positions as visiting assistant professor in the UCF Computer Science Department, senior research scientist at the University of Michigan, Ann Arbor, and research scientist at the Institute for Simulation and Training. He has lead projects and published in the areas of electromagnetics, distributed and parallel processing, terrain databases, 3D modeling, analysis and visualization, and interoperability in distributed simulations for training. He is a member of the IEEE and IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.