Proceedings of the 2007 IEEE Symposium on Computational
Intelligence in Scheduling (CI-Sched 2007)

A Hybrid GA-based Scheduling Algorithm for Heterogeneous Computing Environments

Han Yu

Abstract— **We design a hybrid algorithm to schedule the execution of a group of dependent tasks for heterogeneous computing environments. The algorithm consists of two elements: a genetic algorithm(GA) to map tasks to processors, and a heuristic-based approach to assign the execution order of tasks. This algorithm takes advantage of both the exploration power of GA and the heuristics embedded in the scheduling problem, so it can effectively reduce the search space while not sacrificing the search quality. The experiments show that this algorithm performs consistently better than Heterogeneous Earliest-Finish-Time(HEFT) without incurring much computational cost. Multiple runs of the algorithm can further improve the search result.**

I. Introduction

Scheduling a group of dependent tasks on parallel processors is an intensively studied problem in parallel computing. By decomposing a computation into smaller tasks and executing the tasks on multiple processors, we can potentially reduce the total execution time of the computation.

The scheduling problem is typically given by a group of dependent tasks along with a group of interconnected processors. The data dependency and execution precedence among tasks can be described with a directed acyclic graph (DAG). The goal of scheduling is to minimize the total execution time of tasks, also known as *makespan*, by assigning the tasks to the processors. There are many variations of scheduling problems, with different assumptions on the interconnection and processing ability of processors. Traditional scheduling problems assume a homogeneous computing environment in which all processors have the same processing abilities and they are fully connected. Recent studies have been diverted to scheduling for heterogeneous computing environments in which the execution time of a task may vary among different processors, not all processors are directly connected, and the bandwidth of communication links connecting processors may also be different. In addition, some scheduling problems allow a task to be executed on multiple processors, while other problems restrict the execution of a task on only one processor.

In this paper, we focus on the study of scheduling for heterogeneous computing environments. We assume that processors are fully connected with the same communication links (i.e., with the same bandwidths), but they have different processing abilities. In addition, each task can only be executed on one processor. The communication time between two dependent tasks should be taken into account if they are assigned to different processors. We also assume a static computing model in which the dependence relations and the execution times of tasks are known a priori and do not change over the course of scheduling and task execution. In addition, all processors are fully available to the computation on the time slots they are assigned.

Han Yu is with the Physical Realization Research Center of Motorola Labs, 1301 E. Algonquin Road Room 1014, Schaumburg, Illinois 60196. (Phone: 847-538-2545; Fax: 847-576-2111; Email: a37377@motorola.com)

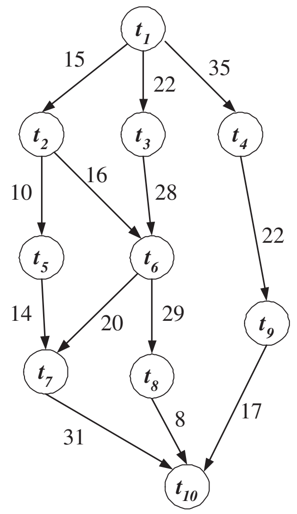

Fig. 1. An example DAG containing ten tasks.

Figure 1 shows an example DAG that contains ten tasks, t_1 to t_{10}. The arrows represent data dependencies among tasks. Two tasks are dependent if the execution of one task relies on the execution result of the other. The numbers represent the communication times needed to transfer data between two dependent tasks. Table I lists the execution times of each task on three processors, P_1, P_2, and P_3. Figure 2 shows an execution schedule of tasks with a makespan of 153.

For a pair of dependent tasks, t_i and t_j, if the execution of t_j depends on the output from the execution of t_i, then t_i is the predecessor of t_j, and t_j is the successor of t_i. We use $prod(t)$ and $succ(t)$ to denote the set of predecessor tasks and successor tasks of task t, respectively.

II. Related Work

The search for an optimal solution to the problem of multiprocessor scheduling has been proven to be NP-hard except for some special cases [5]. Numerous approaches have been developed to solve the problem. These approaches can be mainly classified into two categories: deterministic approaches and non-deterministic approaches.

Deterministic approaches attempt to exploit the heuristics extracted from the nature of the problem in guiding the search for a solution. They are efficient algorithms as the search

TABLE I
THE COMPUTATION TIMES OF TEN TASKS IN FIGURE 1 ON THREE PROCESSORS, P_1, P_2, AND P_3.

	P_1	P_2	P_3
t_1	27	21	30
t_2	19	15	18
t_3	35	21	27
t_4	17	12	26
t_5	11	18	14
t_6	19	32	28
t_7	26	15	31
t_8	22	22	20
t_9	22	31	22
t_{10}	16	19	24

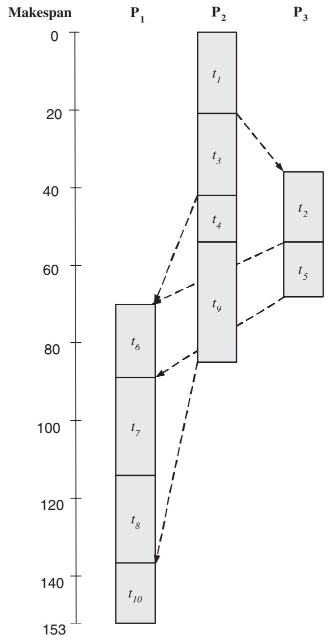

Fig. 2. A schedule for the task graph in Figure 1 on three processors. The makespan of the schedule is 153.

is narrowed down to a very small portion of the solution space; however, the performance of these algorithms is heavily dependent on the effectiveness of the heuristics. Therefore, they are not likely to produce consistent results on a wide range of problems.

Of all the deterministic approaches, many of them belong to *list scheduling algorithms*. The search in list scheduling algorithms is divided into two phases: in the first phase, a priority value is given to each task according to some criteria; in the second phase, tasks are assigned to processors in decreasing order of their priorities. ISH [9], DSH [9], MCP [18], and CPFD [2] are typical list scheduling approaches to homogeneous computing systems, while HEFT [14] and CPOP [14] are list scheduling algorithms designed for heterogeneous computing systems. A drawback of list scheduling algorithms is that the static assignment of task priority is not able to capture the dynamics in task execution because less important tasks may be scheduled earlier, causing the delay of execution of more important tasks. Dynamic approaches, such as DCP [10], attempt to overcome this problem by overlapping the phases of task order assignment and task scheduling.

Another group of deterministic algorithms is *clustering algorithms* [8], [19]. These algorithms assume that there are an unlimited number of processors available to task execution. Clustering algorithms will use as many processors as possible in order to reduce the makespan of the schedule. If, however, the number of processors used by a schedule is more than the number actually available in a given problem, a mapping process is required to merge the tasks in the proposed schedule onto the actual number of available processors.

Contrary to deterministic algorithms, non-deterministic algorithms incorporate a combinatoric process in the search for solutions. Non-deterministic algorithms typically require sufficient sampling of candidate solutions in the search space and have shown robust performance on a variety of scheduling problems. Genetic algorithms [6], [11], [16], [15], [17], simulated annealing [3], [7], [12], and tabu search [13] have been successfully applied to task scheduling. Non-deterministic algorithms, however, are less efficient and have much higher computational cost than deterministic algorithms.

Genetic algorithms have been widely used to evolve solutions for many multiprocessor scheduling problems. These GA approaches vary in their encoding schemes, the implementation of genetic operators, and methods for solution evaluation. However, due to the large solution space that a GA is required to cover, the search generally incurs considerably high computational cost. Some GA approaches use specially designed encoding methods or genetic operators to reduce the search space, but they may also result in bias during the search and affect the quality of solutions [1], [6], [4].

III. ALGORITHM DESIGN

The basic idea of our algorithm is to exploit the advantages of both the evolutionary and heuristic-based algorithms while avoiding their drawbacks. We restrict the use of GA to perform task-to-processor mapping only while using a heuristic approach to determine the order of tasks assigned to the same processor.

A. A Genetic Algorithm for Task Mapping

We use a simple genetic algorithm to perform task mapping, i.e., assign the execution of each task to one of the available processors. The solution of the GA is encoded with a linear list of integers, with each integer representing the processor to which a task is assigned. Suppose there are n tasks and m available processors. The value of each integer in the solution ranges from 1 to m, and there are n integers in each solution. If each task has a unique id (from task 1 to n), the i-th integer represents the processor to which task i is assigned. The search space of a GA, therefore, is m^n. Figure 3 shows the corresponding GA individual to the schedule in Figure 2.

2	3	2	2	3	1	1	1	2	1

Fig. 3. The individual that encodes the solution in Figure 2. Each integer in the individual represents the processor that a task is mapped to.

Elements in a simple GA are employed: one-point crossover, fixed rate mutation, and tournament selection. When an integer is chosen to be mutated, it is replaced by another randomly generated integer ranging from 1 to m.

B. A Heuristic-based Algorithm for Task Order Assignment

For each individual in the population, we need to determine the order of tasks to be executed on each processor and calculate the makespan of the schedule. We apply a heuristic-based algorithm for task order assignment. Tasks are assigned one by one, and in each step, we choose one of the tasks that are ready for execution. A task is *ready* if it has no predecessor or all its predecessor tasks are already scheduled. Among all ready tasks, the priority is given to the one whose execution is critical to reduce the makespan of a schedule. The calculation of the makespan of a schedule on a task level is based on two aspects: the completion time of the task and the execution time of all tasks that depend on the execution of this task. The completion time of a task can be calculated, but the latter aspect cannot be accurately determined as the execution order of successor tasks is not known yet. We use the notion of upward rank to give an estimation. The upward rank of a task is used in HEFT to determine the priority of task assignment [14]. The upward rank of a task t, $u_r(t)$, can be calculated recursively with the following equation:

$$u_r(t) = \begin{cases} 0 & \text{if } succ(t) = \phi \\ max(avg(t_j) + comm & \\ (t, t_j)), \forall t_j \in succ(t) & \text{otherwise} \end{cases} \tag{1}$$

where $comm(t, t_j)$ denotes the communication time between tasks t and t_j, and $avg(t_j)$ denotes the average execution time of task t_j on all processors. A task that does not have any successor tasks receives an upward rank of zero, while the upward rank of all other tasks is the largest of the sum of the average computation time of its successor task and the communication time between the task and the successor task. The upward rank of a task is independent of the processors to which the task and all its successor tasks are assigned. Therefore, the upward rank can be calculated before task mapping. Table II shows the average computation times and upward ranks of all tasks in Figure 1.

The procedure of task order assignment consists of multiple phases, and in each phase we assign the execution of one task. We define S_{ready} to be the set of all ready tasks. In each phase, we first initialize S_{ready} that includes all tasks that has no predecessors. Then for each task in S_{ready}, we calculate its completion time on the designated processor. We always choose the task whose sum of completion time and upward rank is the largest. The selected task is appended to the execution task queue of the processor and removed from S_{ready}. New ready tasks, if there are any, are added to S_{ready}.

TABLE II
THE AVERAGE COMPUTATION TIMES AND UPWARD RANKS OF TEN TASKS IN FIGURE 1 ON THREE PROCESSORS, P_1, P_2, AND P_3.

	Avg. Computation Time	Upward Rank
t_1	26	198.67
t_2	17.33	137
t_3	27.67	149
t_4	18.33	83.67
t_5	14.33	88.67
t_6	26.33	94.67
t_7	24	50.67
t_8	21.33	27.67
t_9	25	36.67
t_{10}	19.67	0

We repeat the above steps until all tasks have been scheduled. After that, we calculate the makespan of the schedule which in turn determines the fitness of the GA individual.

C. Algorithm Procedure

Combining both the GA and heuristic-based algorithm, we show the procedure of the hybrid algorithm as follows:

```
a) calculate the average execution time
   and upward rank of each task
b) initialize GA population
c) for each generation, do
   c.1) for each individual, do
      c.1.1) map tasks to processors
             according to the individual
      c.1.2) determine the order of task
             execution
      c.1.3) calculate the makespan of
             the schedule
   c.2) perform tournament selection to
        produce individuals for the next
        generation
   c.3) perform crossover and mutation on
        selected individuals
d) select the best solution in the final
   generation as the solution of the
   algorithm
```

The detailed procedure of determining order of task execution order (step c.1.2) is:

```
a) initialize the set of ready tasks
   that includes tasks with no
   predecessors
b) while the set of ready tasks is not
   empty, do
   b.1) for each ready task, do
        b.1.1) calculate the completion
               time of the task on the
               assigned processor
        b.1.2) calculate the sum of its
               completion time and upward
               rank
   b.2) select the task whose sum is the
       largest among all ready tasks and
       assign the task to the processor
```

```
b.3) update the set of ready tasks by
     removing the scheduled task and
     adding new ready tasks
```

IV. Experiments

A. Experimental Design

We evaluate the performance of our algorithm on task graphs with variable characteristics. We test thirteen groups of task graphs, with each group having a different communication to computation ratio (i.e., the ratio of the average communication time between dependent tasks and the average computation time of all tasks in a graph) and the density of the graph (which is given by the average number of outgoing tasks for a given task). Both the communication to computation ratio and the average number of outgoing tasks have baseline settings of 1.0 and range between 0.5 and 2. For each group, we randomly generate ten task graphs. All task graphs contain ten tasks to be scheduled on three available processors. The average computation time of a task on a processor is 20. Both the computation time of a task and the communication time of dependent tasks follow Poisson distributions.

We run the GA fifty times for each task graph. For each run, we calculate the speedup of the solution using the following equation:

$$speedup = \frac{serial\_execution\_time}{makespan} \tag{2}$$

where $serial\_execution\_time$ is the sum of the average computation times of all tasks. We use $serial\_execution\_time$ to approximately calculate the makespan of a schedule if all tasks are serially assigned to the same processor. The higher the speedup, the more effective the distribution of task execution on parallel processors.

For each task graph, we calculate both the average speedup of solutions over fifty runs with 95% confidence interval (CI) and the speedup of the best solution found in fifty runs. Then we calculate the average value of these results for ten task graphs in each group.

Table III lists the GA parameters used in the experiment, which exhibits the best performance from a preliminary experiment on various parameter settings.

TABLE III
Parameter settings used in the experiment.

Parameter	Value
Population Size	200
Number of Generations	200
Crossover Rate	0.9
Mutation Rate	0.01
Selection Scheme	Tournament
Tournament Size	2

In addition, we evaluate the performance of HEFT on the same thirteen groups of task graphs and calculate the average speedup of solutions for each group. HEFT is a list scheduling

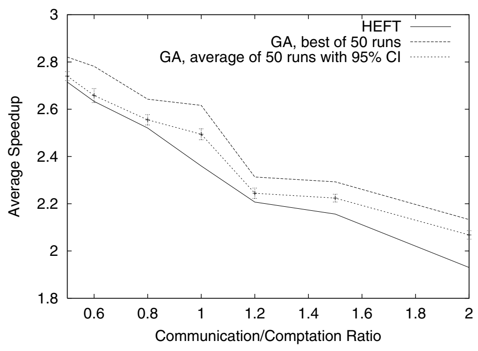

Fig. 4. The comparison of performance between the hybrid algorithm and HEFT on task graphs with varying communication to computation ratios. The result is given by the average speedup of schedules for ten task graphs in each test case. For the hybrid algorithm, we plot both the highest speedup found in fifty runs for each task graph and the average speedup of schedules in fifty runs with 95% confidence interval (CI).

algorithm and the priority of tasks is based on their upward ranks. As a deterministic algorithm, HEFT is run only once for each task graph.

B. Experimental Results and Analysis

Figure 4 shows the comparison between the hybrid algorithm and HEFT on task graphs with varying communication to computation ratios. The average number of outgoing tasks is fixed at 1.0. For GA runs, we show the average speedup of both the best solutions and the average solutions (along with 95% confidence interval) in each test case. The results indicate that the speedup of schedules decreases quickly as the communication to computation ratio increases. The hybrid algorithm performs consistently better than HEFT in all test cases. The gap on the performance of these two algorithms is more noticeable in test cases with higher communication to computation ratios (e.g., with a ratio of 2.0). To schedule a task graph with a high communication to computation ratio, proper assignment of dependent tasks on processors is essential to avoid or reduce high communication costs. The use of the GA for task mapping enables the hybrid algorithm to search for a larger solution space than HEFT, so it is more likely to find better mappings for tasks. Figure 4 also indicates that a better result can be found if we run the algorithm sufficient number of times (in our case is fifty runs for each task graph). The hybrid algorithm is also efficient, with an average execution time of 0.15 seconds for each run.

Figure 5 shows the comparison between the hybrid algorithm and HEFT on task graphs with varying densities. The communication to computation ratio is fixed at 1.0. A higher speedup can be achieved when the density of task graph is lower (i.e., lower degrees of dependency between tasks). The hybrid algorithm outperforms HEFT in all test cases. Again, running the algorithm multiple times allows us to find better solutions than a single run.

We also study the effectiveness of genetic algorithms by looking into the progress of search during a GA run. Figure 6 shows the average makespan and the makespan of the best

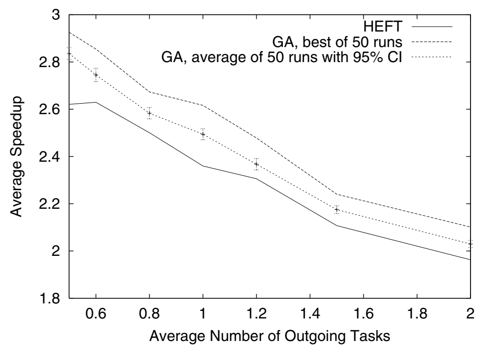

Fig. 5. The comparison on the performance between the hybrid algorithm and HEFT on task graphs with varying densities. The result is given by the average speedup of schedules for ten task graphs in each test case. For the hybrid algorithm, we plot both the highest speedup found in fifty runs for each task graph and the average speedup of schedules in fifty runs with 95% confidence interval (CI).

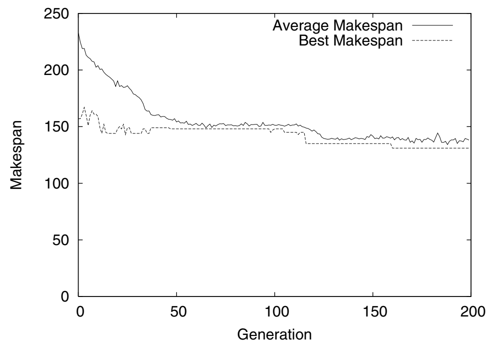

Fig. 6. The average makespan of solutions and the makespan of the best solution in each generation in a typical GA run.

solution found in each generation in a typical GA run. We observe that the makespan decreases continuously throughout the run, despite momentary increases in some generations. This result indicates that the GA search is not likely to be stuck in local optima, a typical problem for local search approaches. The best solution, with a makespan of 131, is first found in generation 160.

We perform additional experiments to evaluate the effectiveness of using GAs in our hybrid algorithm. In the first experiment, we run the algorithm by turning off crossover (set the crossover rate to 0 while keep the mutation rate at 0.01), simulating a hill climbing search process. In the second experiment, we run the algorithm without mutation (set the mutation rate to 0 while keep the crossover rate at 0.9). The same thirteen groups of task graphs are used. The average speedup of solutions with 95% confidence interval is calculated for the performance study. Tables IV and V show the performance comparison of these two experiments and the results from GA runs with the standard parameter settings in Table III. GA runs with both crossover and mutation produce consistently better solutions than runs that perform crossover or mutation only. GAs with both crossover and mutation also exhibit more stable performance with a smaller confidence interval within fifty runs. This result indicates to us that both the exploration and exploitation during the GA search process are important to reach the high performance of our hybrid algorithm. Suppressing either of them can deteriorate the search quality of GA.

V. Conclusions

We design a hybrid algorithm for scheduling tasks on heterogeneous processors. This algorithm incorporates a GA-based search to map tasks to processors while using a heuristic-based approach to assign the order of task execution for each processor. As a result, this algorithm can cover a larger search space than deterministic scheduling approaches without incurring high computational cost. The experiments show that this algorithm outperforms HEFT, a widely used deterministic algorithm for heterogeneous computing systems, with a higher speedup on task execution. The advantage of this algorithm is more noticeable if proper assignment of tasks on processors is critical to locate high quality solutions.

We plan to extend our study with additional experiments to evaluate the performance of this algorithm. We will test the algorithm on larger task graphs (i.e., more tasks) and network systems (more processors), and variable degrees of heterogeneity among processors and tasks.

Our algorithm can potentially be improved with a different method for calculating the upward ranks of tasks. In the current implementation, the calculation of upward ranks is performed before the GA begins and is independent of the processor that a task is assigned to. We use the average computation time of a task in the calculation. Alternatively, we can calculate the upward rank after task mapping is finished. As all tasks have been assigned a processor for execution, we can use the computation time of a task on its assigned processor when calculating the upward rank. This modification may result in more accurate estimation of the makespan and may further improve the quality of solutions.

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 Transactions on Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872–892, 1998.

[3] S. W. Bollinger and S. F. Midkiff, "Processor and link assignment in multicomputers using simulated annealing," in *Proceedings of the International Conference on Parallel Processing*, 1988, pp. 1–7.

[4] M. K. Dhodhi, I. Ahmad, and I. Ahmad, "A multiprocessor scheduling scheme using problem-space genetic algorithms," in *Proc. IEEE Int'l Conf. on Evolutionary Computing*, 1995, pp. 214–219.

[5] M. R. Garey and D. S. Johnson, *Computers and intractability, a guide to the theory of NP-Completeness*. New York: W. H. Freeman, 1979.

[6] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113–120, 1994.

[7] K. Hwang and J. Xu, "Mapping partitioned program modules onto multicomputer nodes using simulated annealing," in *Proceedings of the International Conference on Parallel Processing*, 1990, pp. 292–293.

[8] S. J. Kim and J. C. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," in *International Conference on Parallel Processing*, vol. 2, 1988, pp. 1–8.

[9] B. Kruatrachue and T. G. Lewis, "Duplication Scheduling Heuristic, a new precedence task scheduler for parallel systems," Oregon State University, Tech. Rep. 87-60-3, 1987.

TABLE IV
THE PERFORMANCE COMPARISON AMONG THREE SETTINGS OF GAS, A STANDARD GA WITH PARAMETER SETTINGS IN TABLE III, A GA WITH MUTATION ONLY, AND A GA WITH CROSSOVER ONLY, ON TASK GRAPHS WITH VARYING COMMUNICATION TO COMPUTATION RATIOS.

Communication to Computation Ratio	Standard GA Runs Avg. Speedup (95% CI)	Mutation Only Runs Avg. Speedup (95% CI)	Crossover Only Runs Avg. Speedup (95% CI)
0.5	2.7403 (0.0189)	2.6843 (0.0259)	2.7152 (0.0237)
0.6	2.6579 (0.0295)	2.5914 (0.0317)	2.6414 (0.0298)
0.8	2.5552 (0.0220)	2.5133 (0.0294)	2.5323 (0.0250)
1.0	2.4941 (0.0232)	2.4333 (0.0321)	2.4577 (0.0272)
1.2	2.2444 (0.0219)	2.2011 (0.0251)	2.2348 (0.0194)
1.5	2.2238 (0.0165)	2.1965 (0.0214)	2.2028 (0.0206)
2.0	2.0679 (0.0178)	2.0336 (0.0211)	2.0530 (0.0199)

TABLE V
THE PERFORMANCE COMPARISON AMONG THREE SETTINGS OF GAS, A STANDARD GA WITH PARAMETER SETTINGS IN TABLE III, A GA WITH MUTATION ONLY, AND A GA WITH CROSSOVER ONLY, ON TASK GRAPHS WITH WITH VARYING DENSITIES.

Average Number of Outgoing Tasks	Standard GA Runs Avg. Speedup (95% CI)	Mutation Only Runs Avg. Speedup (95% CI)	Crossover Only Runs Avg. Speedup (95% CI)
0.5	2.8357 (0.0248)	2.7451 (0.0331)	2.8209 (0.0272)
0.6	2.7452 (0.0284)	2.6916 (0.0327)	2.7090 (0.0301)
0.8	2.5832 (0.0241)	2.5305 (0.0296)	2.5357 (0.0282)
1.0	2.4941 (0.0232)	2.4333 (0.0321)	2.4577 (0.0272)
1.2	2.3673 (0.0242)	2.3236 (0.0280)	2.3504 (0.0260)
1.5	2.1750 (0.0165)	2.1437 (0.0213)	2.1492 (0.0189)
2.0	2.0295 (0.0155)	1.9877 (0.0208)	2.0148 (0.0165)

[10] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel & Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.
[11] ——, "Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 58–77, 1997.
[12] A. K. Nanda, D. DeGroot, and D. Stenger, "Scheduling directed task graphs on multiprocessors using simulated annealing algorithms," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992.
[13] S. C. S. Porto and C. C. Ribeiro, "A tabu search approach to task scheduling on heterogeneous processors under precedence constraints," *International Journal of High-Speed Computing*, vol. 7, no. 2, 1995.
[14] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel & Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.
[15] T. Tsuchiya, T. Osada, and T. Kikuno, "Genetic-based multiprocessor scheduling using task duplication," *Microprocessors and Microsystems*, vol. 22, pp. 197–207, 1998.
[16] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogenous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8–22, 1997.
[17] A. S. Wu, H. Yu, S. Jin, G. Schiavone, and K.-C. Lin, "An incremental genetic algorithm approach to multiprocessor scheduling," *IEEE Transactions on Parallel & Distributed Systems*, vol. 15, no. 9, pp. 824–834, 2004.
[18] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Transactions on Parallel & Distributed Systems*, vol. 1, no. 3, pp. 330–343, 1990.
[19] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel & Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.