J Supercomput (2008) 43: 77–97
DOI 10.1007/s11227-007-0139-z

A performance study of multiprocessor task scheduling algorithms

Shiyuan Jin · Guy Schiavone · Damla Turgut

Published online: 1 June 2007
© Springer Science+Business Media, LLC 2007

Abstract Multiprocessor task scheduling is an important and computationally difficult problem. A large number of algorithms were proposed which represent various tradeoffs between the quality of the solution and the computational complexity and scalability of the algorithm. Previous comparison studies have frequently operated with simplifying assumptions, such as independent tasks, artificially generated problems or the assumption of zero communication delay. In this paper, we propose a comparison study with realistic assumptions. Our target problems are two well known problems of linear algebra: LU decomposition and Gauss–Jordan elimination. Both algorithms are naturally parallelizable but have heavy data dependencies. The communication delay will be explicitly considered in the comparisons. In our study, we consider nine scheduling algorithms which are frequently used to the best of our knowledge: min–min, chaining, A*, genetic algorithms, simulated annealing, tabu search, HLFET, ISH, and DSH with task duplication. Based on experimental results, we present a detailed analysis of the scalability, advantages and disadvantages of each algorithm.

Keywords Task scheduling · Parallel computing · Heuristic algorithms · Communication delay

1 Introduction

Scheduling a set of dependent or independent tasks for parallel execution on a set of processors is an important and computationally complex problem. Parallel program can be decomposed into a set of smaller tasks that generally have dependencies. The goal of task scheduling is to assign tasks to available processors such that

S. Jin · G. Schiavone · D. Turgut (✉)
School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816-2362, USA
e-mail: turgut@eecs.ucf.edu

precedence requirements between tasks are satisfied and the overall time required to execute all tasks, the *makespan*, is minimized. There are various variants of this problem, depending on whether we consider communication delays or not, whether the multiprocessor systems are heterogeneous or homogeneous and other considerations. Various studies have proven that finding an optimal schedule is an NP-complete problem even in the simplest forms [1–3].

As finding an optimal solution is not feasible, a large number of algorithms were proposed which attempt to obtain a near-optimal solution for various variants of the multiprocessor task scheduling problem. These algorithms usually trade the computational complexity of the scheduling algorithm itself to the quality of the solution. Algorithms based on complex, iterative search can usually (but not always) outperform simple one-pass heuristics, but their computational complexity makes them less scalable. The comparison of the various approaches is made difficult by the lack of an agreed benchmark problem, and the variety of assumptions made by the developers.

In this paper, we consider the problem of mapping directed acyclic task graphs with communication delays onto a homogeneous cluster of processing elements. As the most frequently used architectures for parallel computation are currently cluster computers with independent hard drives, this is a realistic model which the user of a parallel computing system would encounter. Furthermore, we had chosen to use as our benchmark, two widely used algorithms in linear algebra, the LU decomposition the and Gauss–Jordan elimination.

We compare the performance of nine typical scheduling algorithms: min–min [4], chaining [5], A* [6], genetic algorithms [1], simulated annealing [7], tabu search [8], as well as three popular list scheduling heuristics [9]: Highest Level First Known Execution Times (HLFET) [10], Insertion Scheduling Heuristic (ISH) [11], and Duplication Scheduling Heuristic (DSH) [12]. Since some of the heuristics were not originally designed to solve the problem in the presence of communication delays, we have adapted the algorithms to take into consideration the communication delays in these cases.

The rest of the paper is organized as follows. The next section presents related research work. The structure of the problem and assumptions are described in Sect. 3. The considered scheduling algorithms and the adaptations which were required to handle communication delays are discussed in detail in Sect. 4. Section 5 presents simulation results and analysis. Conclusions are offered in Sect. 6.

2 Related work

The importance of the multiprocessor task scheduling problem led to several comparative studies. Hwok et al. [13] extensively classified, described and compared twenty-seven scheduling algorithms through a nine-task problem. However, the study did not consider algorithms such as GA, simulated annealing or tabu search, which are frequently used in scheduling problems. In addition, the small size of the target problem prevents us from drawing scalability conclusions. Braun et al. [14] present a comparison of eleven heuristics for mapping meta-tasks without communication delays onto a heterogeneous cluster of processors. This paper simplifies the heuristic scheduling by assuming there are no dependencies among tasks. Davidovic et al. [15] reports on a study focused on the comparison of list scheduling approaches.

The proposed task scheduling algorithms span a wide variety of approaches. Although some scheduling algorithms are based on simple, one-shot heuristics, a significant research effort was targeted towards more complex, iterative search algorithms such as genetic algorithms, simulated annealing, tabu search and A*.

Genetic algorithms have received much attention in parallel computing, mainly because GAs are effective in solving such NP-hard problems. Hou et al. [1] reported that the results of GA were within 10% of the optimal schedules. Their results are based on task graphs with dependencies but without communication delays. The method proposed in [16], though very efficient, does not search all of the solution space. Due to the strict ordering that only the highest priority ready task can be selected for scheduling, there can be many valid schedules omitted from the search. Correa et al. [2] proposed modifications to the approach in [16] to broaden the search space to include all the valid solutions. This modified approach was tested on task graphs that represent well-known parallel programs. Wu et al. [17] proposed a novel GA which allows both valid and invalid individuals in the population. This GA uses an incremental fitness function and gradually increases the difficulty of fitness values until a satisfactory solution is found. This approach is not scalable to large problems since much time is spent evaluating invalid individuals that may never become valid ones. Moore [18] applies parallel GA to the scheduling problem and compares its accuracy with mathematically predicted expected value. More GA approaches are found in [16, 19–22].

List scheduling techniques are also widely used in task scheduling problems. List scheduling techniques assign a priority to tasks that are ready to be executed based on a particular heuristic, then sort the list of tasks in decreasing priority. As a processor becomes available, the highest priority task in the task list is assigned to that processor and removed from the list. If more than one task has the same priority, selection from these candidate tasks is typically random. Ibarra et al. [4] proposed a heuristic algorithm for scheduling independent tasks onto identical and nonidentical processors. Djordjevic and Tosic [5] proposed a single pass deterministic algorithm, chaining, based on list scheduling techniques. In list scheduling heuristics, only the tasks that are ready are considered for mapping. Chaining overcomes this constraint by allowing even the nonready tasks for mapping. Therefore, tasks can be selected for mapping in any order, irrespective of the task dependencies. However, this algorithm does not allow duplication of tasks.

Some researchers proposed a combination of GAs and list heuristics. Correa et al. [2] proposed a modified GA by the use of list heuristics in the crossover and mutation in a pure genetic algorithm. This method is said to dramatically improve the quality of the solutions that can be obtained with both a pure genetic algorithm and a pure list approach. Unfortunately, the disadvantage is that the running time is much larger than when running the pure genetic algorithm. Similarly, Auyeung et al. [23] proposed a genetic algorithm which considers a combination of four list scheduling heuristics, Highest Level First (HLF), Highest Co-level First (HCF), Largest Number of Successors First (LNSF), Largest Processing Time First (LPTF), by assigning a weight to each method, respectively. It outperforms any one of the four list scheduling methods. This paper also addresses the scalability issue by increasing the problem size. However, it assumes no communication delay between processors and no task duplications. Ahmad et al.[24] addressed a different encoding of the chromosomes based on the list scheduling heuristic. Each task is assigned a priority. These priorities are encoded in the chromosome, instead of the actual mapping of the tasks to the processors. An actual mapping is then generated for each chromosome by selecting a ready task with the highest priority and allocating it to the first available processor. The priorities in one of the initial chromosomes are based on the length of the path from that task to a task that does not have any successors. The other individuals of the initial population are obtained by randomly modifying the priorities of the first chromosome. The results obtained were consistently better than those obtained by [1].

Search techniques originating from artificial intelligence have also been applied in this area. Kwoka et al. [25] proposed A* and parallel A* based algorithms and found that the A* algorithm outperforms a previously proposed branch-and-bound algorithm by using considerably less time to generate a solution. The parallel A* has demonstrated an almost linear speedup. Borriello et al. [26] use simulated annealing to solve scheduling problems. Onbasioglu et al. [27] combine simulated annealing with hill climbing to solve similar problems. Porto et al. [28] apply the tabu search algorithm to the static task scheduling problem in a heterogeneous multiprocessor environment under precedence constraints. More task scheduling algorithms are addressed in [29–38].

A special class of task scheduling approaches are the ones which allow task duplication. These approaches allow the execution of the same task on multiple nodes if the output data is needed by multiple downstream tasks. These approaches are trading additional computational power for lower communication overhead. Although in practice, the cost of computation prevents the use of task duplication approaches, whenever the expense can be justified, these approaches can significantly reduce the makespan of computation [12, 19, 39–45].

In addition, some researchers analyze the task scheduling problems based on the unique feature of the structure of task graphs. Amoura et al. [46] propose a graph theoretical model to the Gaussian elimination method. When scheduling, the proposed algorithms, namely, GCO (Generalized Column-Oriented Scheduling), MGCO (the Modified Generalized Column-Oriented Scheduling), and BS (Blocking Scheduling) take the triangular architecture parameters into consideration. Suruma and Sha [47] propose an algorithm based on a newly developed framework called collision graph. The scheduling model is tested in a message scheduling problem in a network environment in cases where network traffic is either regular or irregular. Suruma and Sha [48] build on top of the collision graph concept introduced in [49] to investigate the compile-time analysis and run-time scheduling of point-to-point message transmissions done to minimize the communication overhead. Liu [50] analyzes worse-case error bounds on three parallel processing models. The author mathematically analyzes the differences of performance between nontask duplication and task duplication, and discusses scheduling policies over “wide” and “narrow” tasks. We believe greedy task duplication and “wide” and “narrow” task policies discussed in the paper correspond with the task duplication and b-level concept respectively in the list scheduling algorithms of our paper.

3 Task scheduling problem

We consider a directed acyclic task graph $G = \{V, E\}$ of n vertices. Each vertex $v \in V = \{T_1, T_2, \dots, T_n\}$ in the graph represents a task. Our aim is to map every task to a set $P = \{P_1, P_2, \dots, P_p\}$ of p processors. Each task T_i has a weight W_i associated with it, which is the amount of time the task takes to execute on any one of the p homogeneous processors. Each directed edge $e_{ij} \in E$ indicates a dependency between the two tasks T_i and T_j that it connects. If there is a path from vertex T_i to vertex T_j in the graph G, then T_i is the predecessor of T_j and T_j is the successor of T_i. The successor task cannot be executed before all its predecessors have been executed and their results are available at the processor at which the successor is scheduled to execute.

Each edge of the graph has a weight C_{ij} associated with it, which is proportional to the amount of communication delay required for the results of task T_i to reach T_j if both tasks are allocated to different processors. The communication time is assumed to be zero when the tasks are allocated to the same processor. We assume a fully connected cluster in which each processor is connected to every other processor. The connections are full-duplex meaning that the data communication can take place in either direction simultaneously. Also, each processor can send and/or receive data to and from any number of processors simultaneously. A task is "ready" to execute on a processor if all of its predecessors have completed execution and their results are available at the processor on which the task is scheduled to execute. If the next task to be executed on a processor is not yet ready, the processor remains idle until the task is ready.

We assume that the node and the edge weights within each graph correspond to execution times and the communication times respectively, and their values are known in advance. The input to all of the heuristics consists of $n \times 1$ matrix of execution times W_i of each task, and an $n \times n$ matrix of communication times C_{ij}. If the element C_{ij} is equal to zero, it means that there is no direct dependency between the task T_i and T_j. As the graph is acyclic, the following conditions hold: $\forall_i\ C_{ii} = 0$, and $\forall_{i,j}$ if $C_{ij} = 0$, then $C_{ji} = 0$.

Our goal is to find a schedule which is a mapping of tasks to processors that minimizes the makespan. The makespan of a schedule can be defined as the time it takes from the instant the first task begins execution to the instant at which the last task completes execution. A schedule that overlaps computation with communication to the maximum possible extent shortens the overall makespan.

In the simulation study, we tested two different types of task graphs—the LU and Gauss–Jordan task graphs. To test scalability, the size of the task graphs increases from 14 tasks to 35 tasks for LU decomposition and from 15 to 36 tasks for Gauss–Jordan elimination algorithm. Figures 1 and 2 show the general structure of n tasks for the Gauss–Jordan elimination and the LU decomposition task graphs, respectively.

4 Scheduling algorithms

In this section, we introduce the nine scheduling algorithms considered in our comparison study: min–min, chaining, A*, genetic algorithms, simulated annealing, tabu

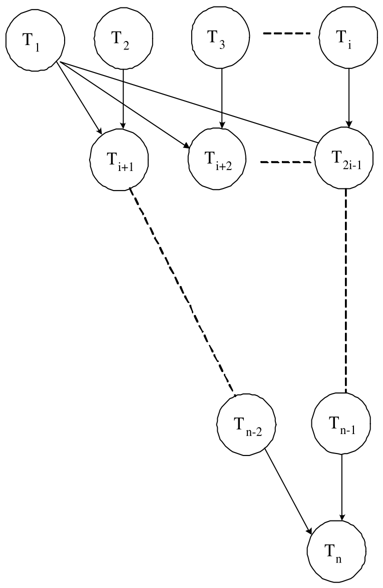


Fig. 1 A task graph for the Gauss–Jordan elimination algorithm

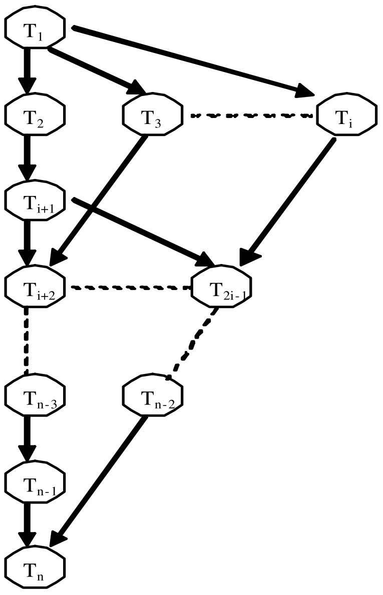


Fig. 2 A task graph for the LU decomposition algorithm

search, HLFET, ISH, and DSH. When necessary, we also present the adaptations performed on the algorithms such that they take into consideration the communication delay.

4.1 Min–min

The min–min heuristic, as presented by Ibarra and Kim [4], can be used to schedule a set of tasks without dependencies onto a heterogeneous multiprocessor system. Min–min is a simple heuristic, which selects a task T_i with the minimum completion time on any processor from U, the set of unmapped tasks, and schedules it onto the processor on which it has the minimum completion time.

The same algorithm with small modifications can be applied to the problem with task dependencies. In the presence of task dependencies, all the tasks from U cannot be compared, as completion times cannot be calculated for a task if all of its predecessors are not yet mapped to some processor. Therefore, only those tasks for which all the predecessors have already been mapped (i.e., those tasks for which none of their predecessors are in U) can be selected for comparison of completion times. Secondly, calculation of the completion times involves both the execution time of the task and the time at which the task is ready to execute. The min–min heuristic is very simple, easy to implement and it was one of the fastest algorithms compared.

4.2 Chaining

Chaining, proposed by Djordjevic and Tosic [5], is a single pass deterministic algorithm based on list scheduling techniques. In list scheduling heuristics, only the tasks that are ready are considered for mapping. Chaining overcomes this constraint by allowing even the nonready tasks for mapping. Therefore, tasks can be selected for mapping in any order, irrespective of the task dependencies.

Chaining tries to partition the task graph among the processors and it does not allow duplication of the tasks. The algorithm starts with a partially scheduled task graph which is the original task graph to which two dummy tasks with zero execution time, s and q are added. Every processor starts by executing the dummy task s and finishes by executing the dummy task q. All other tasks are executed only once. As many ρ-edges as the processors are added between s and q. Each ρ-edge from s to q represents the execution schedule of a single processor. In each iteration of the algorithm, there are two major stages—selection of a task and selection of a ρ-edge. In the task selection, first a task and then the ρ-edge is selected. A task that has the least mobility, i.e., a task with large execution time and large communication delays, is selected in the task selection step. The task is moved to a ρ-edge by placing in which the length of the longest path passing through the task that is minimum. The partially scheduled task graph is updated by removing all the non ρ-edges between the current task and the tasks which are already on the selected ρ-edge. This process is continued until all the tasks have been allocated to some processor (moved to a ρ-edge).

4.3 Genetic algorithms

Genetic algorithms try to mimic the natural evolution process and generally start with an initial population of individuals, which can either be generated randomly or based on some other algorithm. Each individual is an encoding of a set of parameters that uniquely identify a potential solution of the problem. In each generation, the population goes through the processes of crossover, mutation, fitness evaluation and selection. During crossover, parts of two individuals of the population are exchanged in order to create two entirely new individuals which replace the individuals from which they evolved. Each individual is selected for crossover with a probability of crossover rate. Mutation alters one or more genes in a chromosome with a probability of mutation rate. For example, if the individual is an encoding of a schedule, two tasks are picked randomly and their positions are interchanged. A fitness function calculates the fitness of each individual, i.e., it decides how good a particular solution is. In the selection process, each individual of the current population is selected into the new population with a probability proportional to its fitness. The selection process ensures that individuals with higher fitness values have a higher probability to be carried onto the next generation, and the individuals with lower fitness values are dropped out. The new population created in the above manner constitutes the next generation, and the whole process is terminated either after a fixed number of generations or when a stopping criteria is met. The population after a large number of generations is very likely to have individuals with very high fitness values which imply that the solution represented by the individual is good; it is very likely to achieve an acceptable solution to the problem.

There are many variations of the general procedure described above. The initial population may be generated randomly, or through some other algorithm. The search space, i.e., the domain of the individuals, can be limited to the set of valid individuals, or extended to the set of all possible individuals, including invalid individuals. The population size, the number of generations, the probabilities of mutation and crossover are some of the other parameters that can be varied to obtain a different genetic algorithm.

4.4 A* search

The A* algorithm is a best-first search algorithm, originally from the area of artificial intelligence. It is a tree search algorithm that starts with a null solution, and proceeds to a complete solution through a series of partial solutions. The root, or the null solution means none of the tasks are allocated to any processor. The tree is expanded by selecting a task and allocating it to all possible processors. Each allocation is a different partial solution; therefore, each node has p children. At any node, the partial solution has one more task mapped than its parent node. The total number of nodes in the tree is limited to a predetermined constant in order to avoid an exponential execution time. The procedure is explained in detail by Kafil et al. [51], for the problem without communication delays. The cost function $f(n)$ at any node is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the maximum of the machine availability times, and $h(n)$ is a lower bound estimate of time required for the remaining tasks to execute. In order to limit the size of the tree, the nodes with the largest $f(n)$ are removed, whenever the number of nodes exceeds the maximum allowable number of nodes.

This algorithm is not directly applicable to the problem with communication delays. Since given a partial solution, the machine availability times cannot be calculated, unless, all the predecessors of the task have already been allocated, and their finishing times are known. Therefore, we have modified the algorithm such that, at each step, we only select those tasks whose predecessors have already been mapped. This considerably restricts the search space, but this is the only way in which the machine availability times of all the processors can be calculated.

4.5 Simulated annealing

Simulated annealing is a Monte Carlo [52] approach for the optimization functions. This approach derives from the roughly analogous physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure. The simulated annealing process lowers the temperature by slow stages until the system "freezes" and no further changes occur. At each temperature, the simulation must proceed long enough for the system to reach a steady state or equilibrium. Like GA, simulated annealing is also a non-deterministic algorithm.

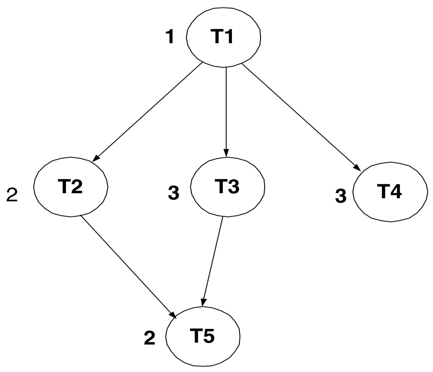
We use a simplified simulated annealing approach to this task scheduling problem. First, we assume that the tasks are labeled according to the topological order in the task graph. All the tasks assigned to the same processor are scheduled sequentially based on their labels. Second, the temperature goes down in every generation during execution. It differs from the normal SA approach that temperature stays until the system reaches a steady state. Third, in every generation, a new solution is created by randomly modifying the current solution (remove a task, or switch the two tasks). The new solution is evaluated by the fitness function. The accept function decides if the new solution is acceptable or not, based on the evaluation result. It is accepted only when its fitness value is higher than the current one, or lower than the current one within an acceptance threshold. If it is accepted, the new solution replaces the current one. Otherwise, it is discarded. Fourth, the algorithm stops after the temperature reaches a predefined value. The procedure of simulated annealing is as follows:

Step 1: Initialize solution.
Step 2: Estimate initial temperature.
Step 3: Evaluate solution.
Step 4: If the new solution is accepted, update the old one.
Step 5: Adjust temperature.
Step 6: If the temperature reaches a pre-defined value, then stop the search; otherwise, generate new solution, and go to step 3.

4.6 Tabu search

Tabu search [8, 28] is a neighborhood search technique that tries to avoid local minima and attempts to guide the search towards a global minimum. Tabu search starts with an initial solution, which can be obtained by applying a simple one-pass heuristic, and scans the neighborhood of the current solution—that is, all the solutions that differ from the current one by a single move. For the multiprocessor task-scheduling problem, a move consists of moving a task from one processor to some other processor, or changing the order of execution of a task within the list of tasks scheduled to a processor. This technique considers all the moves in the immediate neighborhood, and accepts the move which results in the best makespan.

Fig. 3 An example task graph



4.7 Highest level first with estimated times (HLFET)

The Highest Level First with Estimated Times (HLFET) algorithm, proposed by Adam et al. [10], is a list scheduling algorithm which assigns scheduling priority on each node based on static *b-level* [13], which is the length of a longest path from the node to an exit node without considering the length of the edge (or communication time). For instance, in Fig. 3, suppose the number on the left side of each circle represents the task execution time. The longest path from T_1 to the exit node T_5 is T_1, T_3, T_5. So, the static *b-level* of $T_1 = 1 + 3 + 2 = 6$; the static *b-level* of T_4 is 3 because T_4 itself is an exit node. Similarly, the *b-levels* of T_2 and T_3 are 4 and 5, respectively.

Once all predecessors of this task have been scheduled, the task is put on a ready list. The ready list is made in a descending order of static *b-level*. Nodes with the same static *b-level* values are selected randomly. To achieve better scheduling, the first task in the ready list is always scheduled to a processor that allows the earliest execution using noninsertion approach. The ready list is updated and the scheduling process is repeated until all tasks are scheduled. Using static *b-level* simplifies the scheduling because static *b-level* is constant throughout the whole scheduling process; however, it is not optimal since it did not take into an account the communication time as a factor to task scheduling priority. Moreover, as HLFET uses no-insertion approach, an idle time slot is not utilized, which affects performance improvement.

4.8 Insertion scheduling heuristic (ISH)

The Insertion Scheduling Heuristic (ISH) algorithm, proposed by Kruatrachue and Lewis [11], improves the HLFET algorithm by utilizing the idle time slots in the scheduling. Initially, it uses the same approach as HLFET to make a ready list based on static *b-level* and schedule the first node in the ready list using the non-insertion approach. The difference is that, once the scheduling of this node creates an idle slot, ISH checks if any task in the ready list can be inserted into the idle slot but cannot be scheduled earlier on the other processors. Schedule such tasks as many as possible into the idle slot.

4.9 Duplication scheduling heuristic (DSH)

Duplication Scheduling Heuristic (DSH), proposed by Kruatrachue and Lewis [12], differs from the HLFET and ISH algorithms that allow no task cloning or duplication. DSH algorithm duplicates some predecessors in different processors so that each child can start as earlier as possible by eliminating communication delay. Once a node creates an idle slot, the algorithm tries to duplicate as many predecessors as possible into the slot only if the duplicated predecessors can improve the start time of this node. Initially, the static *b-level* of each node based on the DAG diagram is calculated and all nodes placed in a descending order based on the static *b-level*. The ready node n_i with the highest static *b-level* is selected as a candidate and scheduled first and tested on every processor. If node n_i creates an idle time slot in one processor, the parent n_p of this node which is not scheduled in this processor and makes the longest waiting time is considered to be duplicated. If successful, the start-time of node n_i is adjusted and n_p is considered as a candidate. n_p's parents are tracked back and this process is recursively repeated until either no duplication is allowed or an entry node is encountered. The selected task n_i should be scheduled in the processor that offers the minimum start time.

Ahmad and Kwok [53] extend DSH into a new algorithm called Bottom-Up Top-Down Duplication Heuristic (BTDH). The major improvement of the BTDH algorithm over the DSH algorithm is that the BTDH keeps on duplicating the predecessors of a node even if the duplication time slot is completely used up in the hope that the start time will eventually be minimized. However, the authors pointed out that the performance of BTDH and DSH is close.

5 Performance evaluation

5.1 Experimental setup

We have run all the nine algorithms on five instances of various sizes of both the Gauss–Jordan elimination problem and the LU factorization problem. In our simulation, we assume the number of processors is four. The problem sizes, the computation and communication time considered were summarized in Table 1. For the nondeterministic scheduling algorithms, the simulations were run 100 times and the average values reported.

The number of tasks we choose for LU and GJ algorithms is based on their "layered" structure diagrams shown in Figs. 4 and 5, respectively. Let us explain the

Table 1 Experimental setup

Problem	No. of tasks	Computation time	Communication time
Gauss–Jordan elimination	15, 21, 28, 36	40 s/task	100 s
LU factorization	14, 20, 27, 35	10 s bottom layer task, plus 10 s for every layer	80 s

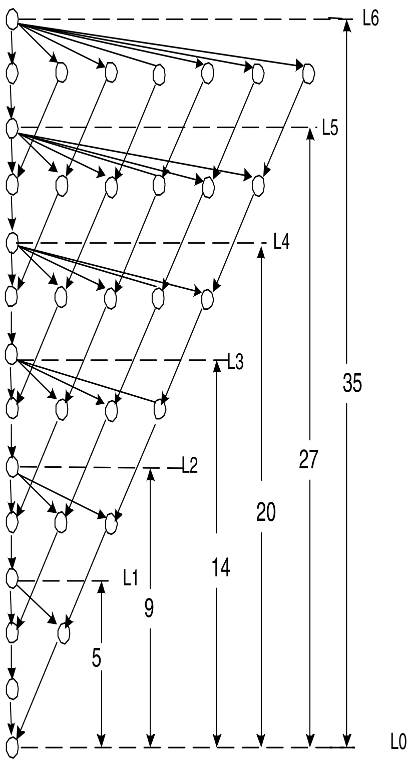


Fig. 4 A task graph for LU decomposition algorithm with 35 tasks

reasoning behind the number of tasks chosen for our performance study. We use a number of dashed lines to help explain the chosen tasks' numbers.

Figure 4 is an example of LU algorithm with 35 tasks. Each layer starts with one task followed by a number of tasks. This LU diagram increments from Layer L1 into L2, L3, L4, L5, and L6, thus the number of task increasing from 9 into 14, 20, 27, and 35, respectively.

Figure 5 shows an example of GJ algorithm with 36 tasks. The numbers of tasks we choose in GJ algorithm is more straightforward. Layers L0, L1, L2, L3, L4, L5, L6, L7 contain 3, 6, 10, 15, 21, 28,and 36 tasks, respectively. Therefore, the total numbers of tasks between layer L0 and L1, L2, L3, L4, L5, L6, and L7 are 3, 10, 15, 21, 28, and 36.

Considering our assumption of four processors, a number of tasks longer than 35 would not yield qualitative differences, especially considering that the additional tasks are the same type as the previous ones.

As discussed thus far, task duplication is an effective way to decrease a makespan. Therefore, we implemented a general GA approach that allows task duplication in scheduling LU and GJ problems. We use elitist selection, meaning that the best individual always survives in one generation, to keep the best result into the next generation. All GA parameters of this simulation are shown in Table 2.

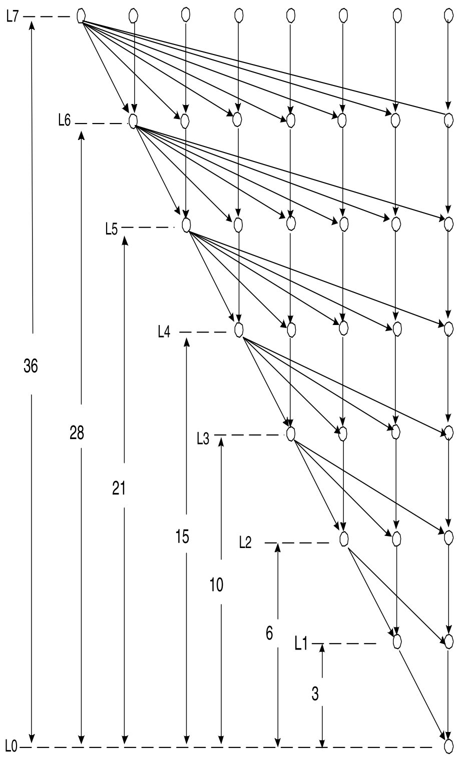


Fig. 5 A task graph for GJ algorithm with 36 tasks

Table 2 GA implementation parameters

Parameters	Value
Population size	30
Mutation rate	0.01
Crossover rate	0.7
Stopping criteria	Stops when it converges within a predefined threshold or when the max. no. of generation exceeds 6,000

5.2 Simulation metrics

The quality of results is measured by two metrics: makespan and computational cost. The makespan is represented by the time required to execute all tasks; the computational cost is the execution time of an algorithm. A good scheduling algorithm should yield short makespan and low computational cost. In our simulation, we evaluate Gauss–Jordan elimination and LU factorization algorithms based on different number of tasks, with the former increasing from 15, 21, 28 to 36 and the latter from 14, 20, 27 to 35. Other parameters such as task computation time and intercommunication delays are given in Table 1. An ideal algorithm must perform well on different problems with different sizes.

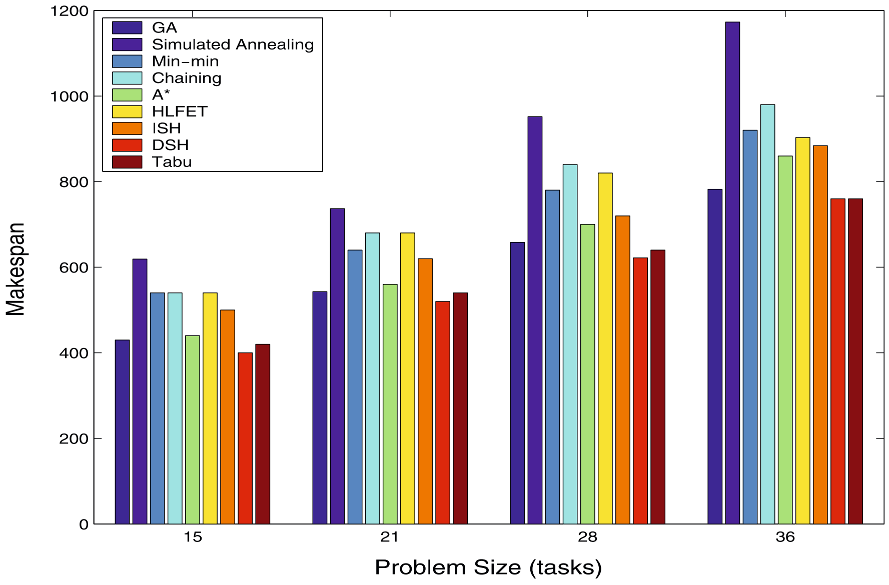


Fig. 6 Makespans for Gauss–Jordan elimination problem for variable task sizes and the nine scheduling algorithms

5.3 Simulation results

The makespan of the obtained solutions are represented on Fig. 6 for the Gauss–Jordan elimination and on Fig. 7 for the LU factorization. The execution time of the algorithms is presented in Fig. 8 for the Gauss–Jordan elimination and in Fig. 9 for the LU factorization. In addition, we analyze the effect of our tabu search results by varying the length of tabu list and the maximum number of moves allowed.

We will present our results centered around a number of observations.

5.3.1 One-shot heuristics vs. iterative search

A cursory look at the graphs of the execution time (Figs. 8 and 9) tells us that the scheduling algorithms can be clearly separated into two classes: one-shot heuristics with very short execution times (min–min, chaining, HLFET, ISH and DSH) and iterative search algorithms with significant execution times (genetic algorithms, simulated annealing, tabu search and A*). One-shot heuristics create a solution based on various criteria, without searching through a subset of the solution space. Iterative search algorithms however, are considering a larger number of possible solutions before returning a preferred solution, therefore they can take a significant amount of time. On the other hand, most iterative search algorithms can return "current best" solutions, if stopped at any given moment of time. Depending on the implementation, A* might be an exception to this, as it might potentially search through incomplete solutions.

GA has been shown to be an effective algorithm in solving *NP*-complete problems. For such problems, however, it is a challenge to predict when the algorithm converges

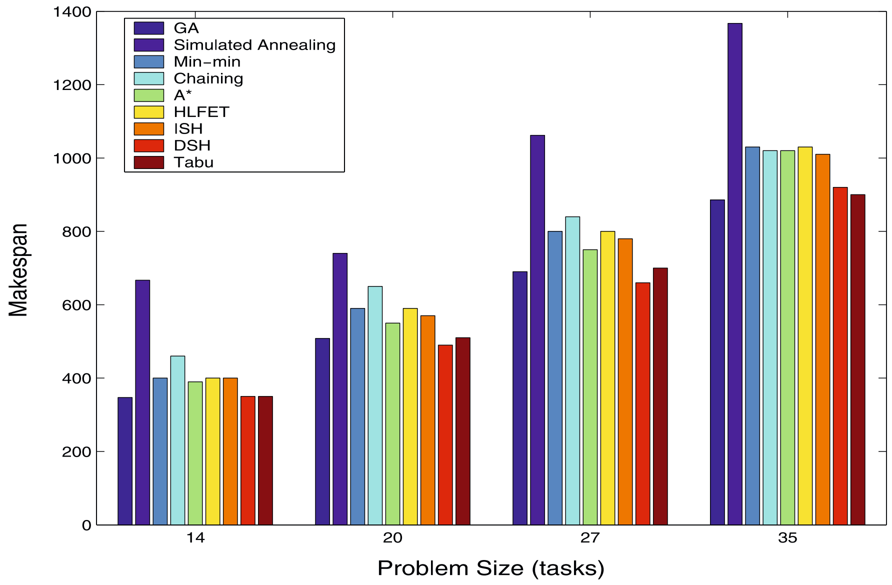


Fig. 7 Makespans for LU factorization problem for variable task sizes and the nine scheduling algorithms

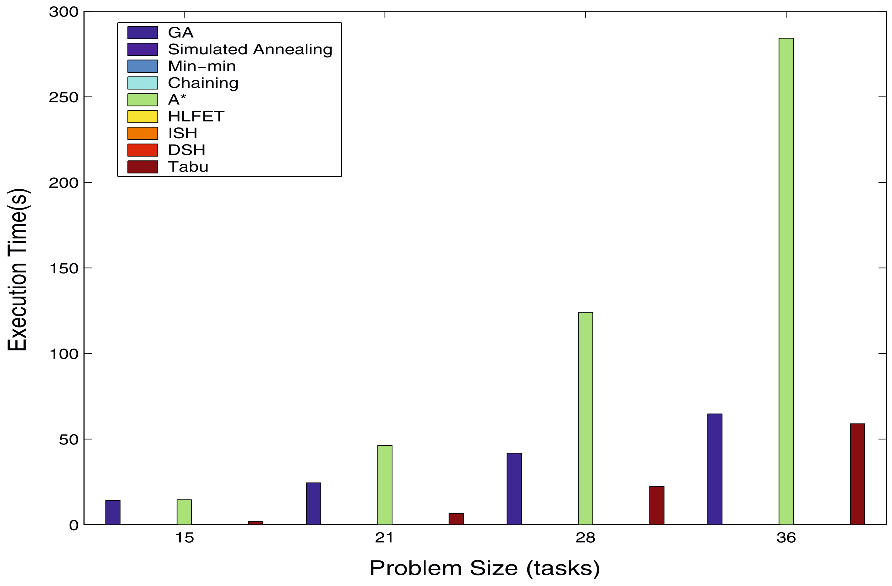


Fig. 8 Execution time of the nine scheduling algorithms for the Gauss–Jordan elimination problem of variable task sizes

and what the optimal solution is, since the best solution is unknown. A number of researchers have analyzed GA convergence issues theoretically. Gao [54] indicated that there is a tradeoff between the convergence and the long-time performance in

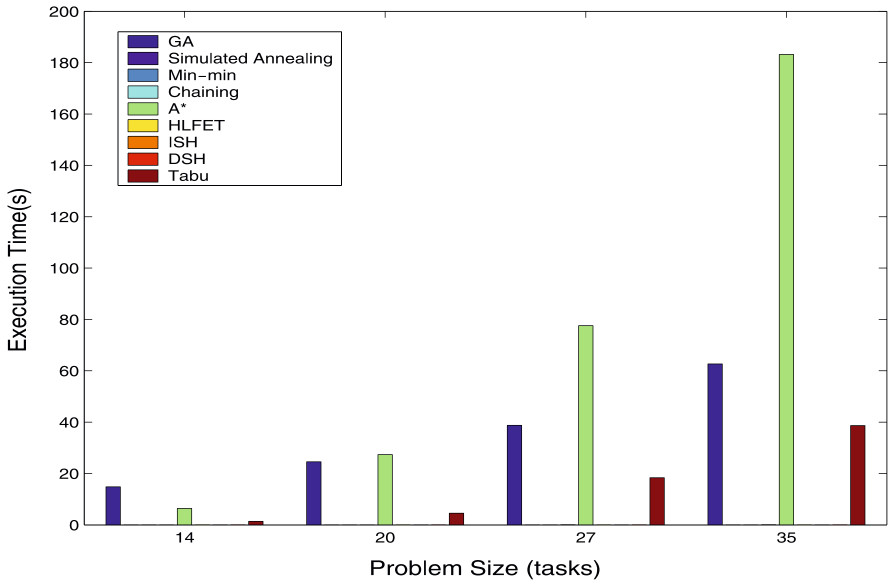


Fig. 9 Execution time of the nine scheduling algorithms for the LU factorization problem of variable task sizes

genetic algorithms. Rudolph [55] considered that a CGA (Canonical Genetic Algorithm) would never converge to the global optimum, while the variants of CGAs that always maintain the best solution are shown to converge to the global optimum. However, in [55], only the probability that a global solution being generated by mutation was discussed, which has a little or no practical use for real-world applications. In addition, all these existing literature assume a fixed length of individual when analyzing convergence; however, our GA uses varied length of individual due to an unexpected number of tasks duplicated. Through literature reviews, we found that many current theoretical research on GA convergences are not applicable to the real-world problems that need to be solved in a reasonable time. Therefore, we terminate our GA algorithm when the best result is converged within a predefined threshold. Otherwise, we stop the algorithm when the maximum number of generation exceeds 6,000.

The deciding factor between one-shot heuristics and iterative search algorithms is whether the improved quality of solutions justifies the computational expense of the iterative search algorithm. Our results show that genetic algorithms, tabu search and A* had outperformed all the one-shot heuristics without task duplication. Simulated annealing on the other hand, had the worst performance among all the scheduling algorithms tested. The makespan difference between the best performing iterative search (genetic algorithm) and the best performing one-shot heuristic without duplication (ISH) was about 60–80 s depending on the problem type and size. As the execution time of the genetic algorithm was of the same magnitude, for our target problems, executed only once, the choice is undecided. However, whenever the execution time of the target problem is larger, or a schedule can be reused multiple times, the investment in the iterative search algorithm will yield overall better results.

We conclude that the expense of the iterative search algorithms are justified for large problems and/or problems where the schedule can be reused multiple times. The best iterative search algorithms were found to be the genetic algorithms and tabu search.

5.3.2 One-shot heuristics without task duplication

We found a consistent ordering between the performance of the one-shot heuristics, with ISH being the best, followed in order by HLFET, min–min and chaining. The improvement of ISH over HLFET is justified by the fact that ISH is able to take advantage of idle time slots, HLFET only appends tasks without insertion, which can result in some idle time slots.

5.3.3 One-shot heuristics with task duplication

Duplication Scheduling Heuristic (DSH) was the clear winner of our comparison study, providing a very low scheduling time (characteristic to one-shot heuristic algorithms) and the solutions with the shortest makespan. This, of course, is obtained at a cost of higher utilization of the available computational resources, by duplicate execution of the tasks.

Depending on the organizational or administrative setting in which the computation takes place, this might or might not be justifiable. If the user has exclusive access to a cluster computer, the use of the duplication scheduling is clearly justified. If the user is charged by the computation capacity used, duplication scheduling leads to extra cost which needs to be balanced against its benefits.

5.3.4 Extended study of tabu search

There are two constants in the tabu search algorithm that can be varied. The constant *nitertabu* specifies the size of the tabu list (i.e., for how many iterations a move must be considered tabu). If the best move at any point is in the tabu list, the tabu search does not accept it, but proceeds to the next best move. The moves continue to get rejected until a move, which is not already in the tabu list, is found. If there is any such move, the move is accepted. If there is no such move, the algorithm updates its tabu list and goes to the next iteration without changing the current solution. The algorithm halts when the number of moves in the immediate history without improvement exceeds a certain level, given by a constant *nmaxmoves*. The constant *nmaxmoves* specifies the maximum number of moves without improvement in the makespan that can be allowed. Clearly, *nitertabu* should be less than *nmaxmoves*. Varying *nitertabu* did not have any considerable effect on the quality of the results. The values of 100 and 200 were used, but the results were identical in both cases as well as the execution times.

A detailed description of the technique is presented by Porto et al. [28]. The paper makes observations that the quality of solutions obtained is not affected much by the different choices of parameters. Varying *nmaxmoves*, however, affects the amount of time it takes by the tabu search. We used three different values (200, 500, and 1,000) for *nmaxmoves*. The results showed that increasing *nmaxmoves* does not guarantee a

Table 3 Results of tabu search by varying nmaxmoves (M: Makespan; Time: Execution time)

Problem	*nmaxmoves*									
	50		100		200		500		1000	
	Time	M	Time	M	Time	M	Time	M	Time	M
Lu14	0.46	350	0.74	350	1.35	350	2.13	350	6.71	350
Gj15	0.6	420	1.03	420	1.98	420	2.96	420	8.35	420
Lu20	1.53	510	2.54	510	4.54	510	9.36	510	21.30	510
Gj21	2.05	540	3.61	540	6.48	540	13.80	540	29.00	540
Lu27	4.6	760	12.41	700	18.35	700	32.16	700	167.34	700
Gj28	10.55	640	14.11	640	22.38	640	64.10	640	86.57	640
Lu35	15.54	930	23.26	930	38.63	930	122	920	384.57	900
Gj36	14.37	840	39.48	760	59.96	760	115.97	760	207.74	760

better solution. Even in the case when the solutions were better, it could be seen that the improvement in the solution was not comparable to the increase in the execution time. For example, increasing *nmaxmoves* from 200 to 1000 increased the execution time by $1,000\%$; however, the improvement was a mere 3.33%. On the other hand, *nmaxmoves* should be large enough to obtain a good solution for large task graphs. We focus on how the results varied with the parameters of the algorithms. The detailed results are presented in Table 3.

6 Conclusions

In this paper, we compared nine scheduling algorithms for multiprocessor task scheduling with communication delays. Duplication Scheduling Heuristic (DSH) had provided short scheduling time and schedules with the shortest makespan, with the additional expense occurring by duplication of tasks on multiple processors. We conclude that from a purely performance point of view, DSH (ones-shot heuristic algorithm) is the best solution, but its deployment needs to be subject of a careful cost-benefit analysis. One-shot heuristic algorithms without task duplication can provide adequate performance and fast scheduling time: Insertion Scheduling Heuristic (ISH) had proven to be the best of this group. The next group, scheduling algorithms based on iterative search such as genetic algorithms, simulated annealing, tabu search, and A* require an order of magnitude longer computation time, but (with the exception of simulated annealing) had yielded better solutions with a shorter makespan than the one-shot heuristic algorithms without task duplication. In this group, the best solutions were obtained by genetic algorithms and tabu search.

We conclude that the use of these algorithms are justified whenever the scheduling can be done off-line, there is a need for repeated execution of the schedules or the makespan of the application is significantly longer than the scheduling time.

References

1. Hou E, Ansari N, Ren H (1994) A genetic algorithm for multiprocessor scheduling. IEEE Trans Parallel Distrib Syst 5(2):113–120
2. Correa R, Ferreira A, Rebreyend P (1999) Scheduling multiprocessor tasks with genetic algorithms. IEEE Trans Parallel Distrib Syst 10:825–837
3. Bokhari S (1999) On the mapping problem. IEEE Trans Comput 30(3):207–214
4. Ibarra O, Kim C (1977) Heuristic algorithms for scheduling independent tasks on non-identical processors. J Assoc Comput Mach 24(2):280–289
5. Djordjevic G, Tosic M (1996) A heuristic for scheduling task graphs with communication delays onto multiprocessors. Parallel Comput 22(9):1197–1214
6. Russell S, Norvig P (2003) Artificial intelligence, a modern approach. Pearson Education, Ch 5, pp 139–172
7. Chamberlain R, Edelman M, Franklin M, Witte E (1988) Simulated annealing on a multiprocessor. In: Proceedings of the 1988 IEEE international conferences on computer design: VLSI in computers and processors, pp 540–544
8. Tian Y, Sannomiya N, Xu Y (2000) A tabu search with a new neighborhood search technique applied to flow shop scheduling problems. In: Proceedings of the 39th IEEE conference on decision and control, vol 5, pp 4606–4611
9. Macey B, Zomaya A (1998) A performance evaluation of CP list scheduling heuristics for communication intensive task graphs. In: Proc joint 12th intl parallel processing symp and ninth symp parallel and distributed prog, pp 538–541
10. Adam T, Chandy K, Dickson J (1974) A comparison of list schedules for parallel processing systems. ACM Commun 17:685–690
11. Kruatrachue B, Lewis T (1987) Duplication scheduling heuristic DSH: A new precedence task scheduler for parallel processing systems. PhD thesis, Oregon State University, Corvallis, OR
12. Kruatrachue B, Lewis T (1998) Grain size determination for parallel processing. IEEE Softw 5(1):23–32
13. Hwok Y, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv 31(4):407–471
14. Braun T, Siegel H, Beck N et al (1999) A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing system. In: Eighth heterogeneous computing workshop (HWC), pp 15–23
15. Davidovic T, Crainic T (2006) Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems. Comput Oper Res 33:2155–2177
16. Zomaya A, Ward C, Macey B (1999) Genetic scheduling for parallel processor systems comparative studies and performance issues. IEEE Trans Parallel Distrib Syst 10(8):795–812
17. Wu A, Yu H, Jin S, Lin K, Schiavone G (2004) An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Trans Parallel Distrib Syst 15(9):824–834
18. Moore M (2004) An accurate parallel genetic algorithm to schedule tasks on a cluster. Parallel Comput 30:567–583
19. Yao W, You J, Li B (2004) Main sequences genetic scheduling for multiprocessor systems using task duplication. Microprocess Microsyst 28:85–94
20. Kwok Y, Ahmad I (1997) Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. J Parallel Distrib Comput 47(1):58–77
21. Ceyda O, Ercan M (2004) A genetic algorithm for multilayer multiprocessor task scheduling. In: TENCON 2004. IEEE region 10 conference, vol 2, pp 68–170
22. Cheng S, Huang Y (2003) Scheduling multi-processor tasks with resource and timing constraints using genetic algorithm. In: 2003 IEEE international symposium on computational intelligence in robotics and automation, vol 2, pp 624–629
23. Auyeung A, Gondra I, Dai H (2003) Evolutionary computing and optimization: Multi-heuristic list scheduling genetic algorithm for task scheduling. In: Proceedings of the 2003 ACM symposium on applied computing, pp 721–724
24. Ahmad I, Dhodhi M (1996) Multiprocessor scheduling in a genetic paradigm. Parallel Comput 22(3):395–406
25. Kwoka Y, Ahmad I (2005) On multiprocessor task scheduling using efficient state space search approaches. J Parallel Distrib Comput 65:1515–1532
26. Borriello G, Miles D (1994) Task scheduling for real-time multi-processor simulations real-time operating systems and software. In: Proceedings of RTOSS '94, 11th IEEE workshop, pp 70–73

27. Onbasioglu E, Ozdamar L (2003) Optimization of data distribution and processor allocation problem using simulated annealing. J Supercomput 25(3):237–253
28. Porto S, Ribeiro C (1995) A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. Int J High-Speed Comput 7(1):45–71
29. Baptiste P (2003) A note on scheduling multiprocessor tasks with identical processing times. Comput Oper Res 30:2071–2078
30. Baruah S (2001) Scheduling periodic tasks on uniform multiprocessors. Inf Process Lett 80:97–104
31. Singh G (2001) Performance of critical path type algorithms for scheduling on parallel processors. Oper Res Lett 29:17–30
32. Hanen C, Munier A (2001) An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. Discrete Appl Math 108:239–257
33. Blazewicz J, Drozdowski M, Formanowicz P, Kubiak W, Schmidt G (2000) Scheduling preemptable tasks on parallel processors with limited availability. Parallel Comput 26:1195–1211
34. Manoharan S (2001) Effect of task scheduling on the assignment of dependency graphs. Parallel Comput 27:257–268
35. Amoura A, Bampis E, Manoussakis Y, Tuza Z (1999) A comparison of heuristics for scheduling multiprocessor tasks on three dedicated processors. Parallel Comput 25:49–61
36. Cai X, Lee C, Wong T (2000) Multiprocessor task scheduling to minimize the maximum tardiness and the total completion time. IEEE Trans Robotics Autom 16:824–830
37. Bandyopadhyay T, Basak S, Bhattacharya S (2004) Multiprocessor scheduling algorithm for tasks with precedence relation. In: TENCON 2004. 2004 IEEE region 10 conference, vol 2, pp 164–167
38. Lundberg L (2002) Analyzing fixed-priority global multiprocessor scheduling. In: Proceedings of eighth IEEE on real-time and embedded technology and applications symposium, pp 145–153
39. Kwok Y (2003) On exploiting heterogeneity for cluster based parallel multithreading using task duplication. J Supercomput 25(1):63–72
40. Ahmad I, Kwok Y (1994) A new approach to scheduling parallel program using task duplication. In: Proceedings of international conference on parallel processing, vol 2, pp 47–51
41. Kang O, Kim S (2003) A task duplication based scheduling algorithm for shared memory multiprocessors. Parallel Comput 29(1):161–166
42. Darbha S, Agrawal D (1997) A task duplication based scalable scheduling algorithm for distributed memory systems. J Parallel Distrib Comput 46:15–26
43. Kruatrachue B (1987) Static task scheduling and grain packing in parallel processing systems. PhD thesis, Electrical and Computer Engineering Department, Oregon State University
44. Tsuchiya T, Osada T, Kikuno T (1998) Genetics-based multiprocessor scheduling using task duplication. Microprocess Microsyst 22:197–207
45. Park G, Shirazi B, Marquis J (1998) Mapping of parallel tasks to multiprocessors with duplication. In: Proceedings of the thirty-first Hawaii international conference on system sciences, vol 7, pp 96–105
46. Amoura A, Bampis E, König J-C (1998) Scheduling algorithms for parallel Gaussian elimination with communication costs. IEEE Trans Parallel Distrib Syst 9(7):679–686
47. Surma D, Sha EH-M (1995) Compile-time communication scheduling on parallel systems. Research report TR-95-3, Dept of Computer Science and Engineering, University of Notre Dame
48. Surma D, Sha EH-M, Passos N (1998) Collision graph based communication reduction techniques for parallel systems. Int J Comput Appl 5(1):11–22
49. Surma D, Sha EH-M (1995) Application specific communication scheduling on parallel systems. In: Eighth international conference on parallel and distributed computing systems, pp 137–139
50. Liu Z (1998) Worst-case analysis of scheduling heuristics of parallel systems. Parallel Comput 24:863–891
51. Kafil M, Ahmad I (1997) Optimal task assignment in heterogeneous computing systems. In: 6th heterogeneous computing workshop (HCW'97), p 135
52. Eliasi R, Elperin T, Bar-Cohen A (1990) Monte Carlo thermal optimization of populated printed circuit board. IEEE Trans Components, Hybrids, Manuf Technol 13:953–960
53. Ahmad I, Kwok Y (1998) On exploiting task duplication in parallel program scheduling. IEEE Trans Parallel Distrib Syst 9(9):872–892
54. Gao Y (1998) An upper bound on the convergence rates of canonical genetic algorithms. Complex Int 5:1–14
55. Rudolph G (1994) Convergence analysis of canonical genetic algorithms. IEEE Trans Neural Networks 5(1):96–101

Shiyuan Jin is currently a Ph.D. student with the School of Electrical Engineering and Computer Science at University of Central Florida. He received B.S. degree from the Zhejiang Institute of Technology, China, and the M.S. degree from the University of Central Florida (UCF), Orlando. His research interests include genetic algorithms, parallel and distributed processing, and optimization.

Guy Schiavone received the Ph.D. degree in engineering science from Dartmouth College in 1994, and the BEEE degree from Youngstown State University in 1989. Dr. Schiavone has over 10 years of teaching and research experience working in the areas of computer modeling and simulation, numerical analysis, image processing and electromagnetics. He also has over 5 years experience working as a research engineer and technician for General Motors. He has numerous published papers and conference presentations in the areas of electromagnetics, terrain database analysis and visualization, and Distributed Interactive Simulation.

Damla Turgut is an Assistant Professor with the School of Electrical Engineering and Computer Science at University of Central Florida. She is affiliated with Networking and Mobile Computing Laboratory (NetMoC), Interdisciplinary Information Science Laboratory (I2Lab), and Institute for Simulation and Training (IST) at UCF. She received her B.S., M.S., and Ph.D. degrees from the Computer Science and Engineering Department of University of Texas at Arlington in 1994, 1996, and 2002 respectively. She has been included in the WHO's WHO among students in American Universities and Colleges in 2002. She has been awarded outstanding research award and has been recipient of the Texas Telecommunication Engineering Consortium (TxTEC) fellowship. She is a member of IEEE, member of the ACM, and the Upsilon Pi Epsilon honorary society. Her research interests include wireless networking, mobile computing, embodied agents, distributed systems, and software engineering.