J. Parallel Distrib. Comput. 71 (2011) 1518–1531

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A hybrid heuristic–genetic algorithm for task scheduling in heterogeneous processor networks

Mohammad I. Daoud [a], Nawwaf Kharma [b,*]

[a] Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, British Columbia, Canada
[b] Department of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada

ARTICLE INFO

Article history:
Received 1 November 2010
Received in revised form
26 April 2011
Accepted 9 May 2011
Available online 27 May 2011

Keywords:
Genetic algorithms
Task scheduling
List-based scheduling heuristics
Directed acyclic graph
Parallel and distributed processing
Heterogeneous systems

ABSTRACT

Efficient task scheduling on heterogeneous distributed computing systems (HeDCSs) requires the consideration of the heterogeneity of processors and the inter-processor communication. This paper presents a two-phase algorithm, called H2GS, for task scheduling on HeDCSs. The first phase implements a heuristic list-based algorithm, called LDCP, to generate a high quality schedule. In the second phase, the LDCP-generated schedule is injected into the initial population of a customized genetic algorithm, called GAS, which proceeds to evolve shorter schedules. GAS employs a simple genome composed of a two-dimensional chromosome. A mapping procedure is developed which maps every possible genome to a valid schedule. Moreover, GAS uses customized operators that are designed for the scheduling problem to enable an efficient stochastic search. The performance of each phase of H2GS is compared to two leading scheduling algorithms, and H2GS outperforms both algorithms. The improvement in performance obtained by H2GS increases as the inter-task communication cost increases.

1. Introduction

A distributed computing system (DCS) is a group of processors connected *via* a high speed network, which supports the execution of parallel applications. DCSs provide a promising hardware architecture for the execution of computationally intensive scientific applications. The efficiency of executing parallel applications on DCSs depends on the way in which the tasks are scheduled onto the processors. Task scheduling aims to allocate the tasks of an application to the set of available processors, and arrange the execution of the tasks on each processor to minimize the total execution time of the application. The task scheduling problem is an NP-complete problem in most cases [42,31,16,17,24,32,40,9].

In general, task scheduling is presented in two forms: *static* and *dynamic* [48,31,23]. In static scheduling algorithms, all information needed for scheduling, such as the structure of the parallel application, the execution times of individual tasks and the communication costs between tasks, must be known *in advance*. There are several techniques to estimate such information [44]. Static task scheduling takes place during compilation time before running the parallel application. In dynamic scheduling, however, tasks are allocated to processors upon their arrival, and scheduling decisions must be made at *run time* [4,7,23,24,29,31,48]. The domain of application of this paper is *static* task scheduling for DCSs with *heterogeneous* processor, or HeDCSs.

Many parallel applications have long execution times, and hence they require high quality task schedules to minimize their run times. Additionally, the static scheduling time of several scientific and engineering applications is much lower than their run time on DCSs. For example, the execution times of more than 50% of the parallel applications that were run on four real parallel computing systems are between tens to thousands of minutes [25], while the static scheduling times of parallel applications with diverse characteristics, which were scheduled using several static scheduling algorithms, are lower than one second as shown in [42] and discussed in Section 5.2 in this paper. Hence, using complex scheduling techniques to generate high quality task schedules, which reduce the run time of parallel applications, is a justifiable and potentially rewarding pursuit. In particular, the achievement of high performance in commonly used HeDCSs requires efficient scheduling algorithms that are specifically developed for such systems. However, it is easy to demonstrate, using counterexamples, that the best existing scheduling algorithms for HeDCSs generate sub-optimal task schedules [42,40]. Hence, there is much room for the development of better scheduling algorithms for HeDCSs.

In this paper, we propose a new hybrid algorithm, called *Hybrid Heuristic–Genetic Scheduling* (or H2GS) algorithm, which combines two algorithms meant for dealing with the problem of scheduling in HeDCSs. The H2GS algorithm starts by running a heuristic

* Corresponding author.
E-mail addresses: mohammad.daoud@gmail.com (M.I. Daoud), kharma@ece.concordia.ca (N. Kharma).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.05.005

scheduling algorithm, called the *Longest Dynamic Critical Path* (or LDCP) algorithm [12], to quickly generate a high quality task schedule. Next, the H2GS algorithm uses a new Genetic Algorithm, called *Genetic Algorithm for Scheduling* (or GAS), that we propose in this paper, to improve the LDCP-generated schedule. A preliminary version of the GAS algorithm has been partially presented in a conference paper [13] with limited results and analysis. The two-phase feature of the H2GS algorithm allows for the customization of the scheduling process. When the goal of the scheduling process is to avoid low quality task schedules without the restriction of short compilation time, the H2GS algorithm can be adjusted to run the GAS algorithm for a large number of generations in order to generate high quality task schedules. Such an approach is crucial if the search space of the scheduling problem is multimodal [2] and faulty scheduling decisions are expected to have high cost. On the other hand, when the goal is to schedule the application on the HeDCS within a short compilation time, H2GS can be adjusted so the GAS algorithm runs for a small number of generations, in order to generate task schedules quickly, but not necessarily optimally.

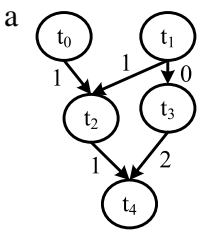
The paper is organized as follows: Section 2 formulates the research problem and introduces some necessary terms. In Section 3, we present related work and discuss two leading task scheduling algorithms for HeDCSs. In Section 4, we present the LDCP and GAS algorithms, then integrate them into one overall algorithm, H2GS. Moreover, in this section an illustrative example is presented that demonstrates the operation of the H2GS algorithm. In Section 5, the performance of the H2GS algorithm is compared to that of HEFT and DLS algorithms. Finally, a conclusion and an overview of future work are provided in Section 6.

2. Problem description

Task scheduling for HeDCSs is the problem of assigning the tasks of a parallel application to the processors of a HeDCS, which have diverse capabilities, and specifying the start execution time of each task. This must be done in a way that respects the precedence constraints among tasks. An efficient schedule is one that minimizes the total execution time, or the *schedule length*, of the parallel application [3,24,32,42].

A parallel application is represented by a directed acyclic graph, or DAG, defined by the tuple (T, E), where T is a set of n tasks and E is a set of e edges. Each, $t_i \in T$, represents a task in the parallel application, which in turn is a set of instructions that must be executed sequentially in the same processor without interruption. Each edge $(t_i, t_j) \in E$ represents a precedence constraint, such that the execution of $t_j \in T$ cannot be started before $t_i \in T$ finishes its execution. If $(t_i, t_j) \in E$, then t_i is a *parent* of t_j and t_j is a *child* of t_i. A task with no parents is called an *entry task*, and a task with no children is called an *exit task*. Each edge $(t_i, t_j) \in$ E has a value that represents the estimated inter-task communication cost required to pass data from the parent task t_i to the child task t_j. Because tasks might need data from their parent tasks, a task can start execution on a processor only when all data required from its parents become available to that processor; at that time the task is marked as *ready*. The speed of the inter-processor communication network is assumed to be much lower than the speed of the intra-processor bus, and consequently the inter-processor communication cost is much higher than the communication cost between tasks scheduled on the same processor. Therefore, when two tasks are scheduled on the same processor the communication cost between these tasks can be ignored.

The HeDCS is represented by a set P of m processors that have diverse capabilities. The $n \times m$ *computation cost matrix* C stores the execution costs of tasks. Each element $c_{i,j} \in C$ represents the estimated execution time of task t_i on processor p_j. Precise calculation of the running times of the tasks on the processors is unfeasible before running the application [36]. One approach to estimate the execution time of task t_i on processor p_j is to use profiling information of t_i and p_j. This estimation method has been used in [11,46]. Another way to estimate the execution time of t_i on p_j is to analyze past observations of the running times of similar tasks on p_j [36,27].



b

task	p0	p1
t0	7	8
t1	6	9
t2	5	8
t3	2	3
t4	2	4

Fig. 1. A sample (a) DAG and (b) computation cost matrix.

All processors in the HeDCS are assumed to be fully connected. Communications between processors occur *via* independent communication units; this allows for concurrent execution of computation of tasks and communications between processors [3,42]. The computation costs of tasks are assumed to be monotonic. In other words, if the computation cost of task t_i on processor p_j is higher than that on processor p_k, then the computation costs of any task on p_j is higher than or equal to that on processor p_k. After scheduling all the tasks of a parallel application on the processors of a HeDCS, the schedule length is defined as the longest finish time of the HeDCS processors. Fig. 1 presents an example of a parallel application consisting of five tasks and a HeDCS with two processors, where the application is represented as a DAG and the execution costs estimated for the five tasks on the HeDCS are shown as a computation cost matrix.

3. Related work

3.1. Task scheduling for DCSs

Static scheduling algorithms can be broadly classified into three main groups: *heuristic algorithms*, *guided random algorithms* and *hybrid algorithms* [34,42,48,19].

3.1.1. Heuristic scheduling algorithms

Heuristic scheduling algorithms move from one point in the search space to another, following a particular rule. Such algorithms, though efficient, search some paths in the search space and ignore others [48]. Heuristic scheduling algorithms can be subdivided into three subgroups: *list-based heuristics*, *clustering heuristics* and *duplication heuristics* [42,48,32].

In list-based scheduling heuristics, each task is assigned a given priority. The tasks are inserted in a list of waiting tasks, such that tasks with higher priority are placed before those with lower priorities. Three steps are then repeated until all the tasks in the list are scheduled: task selection, processor selection and status update. The ready task with the highest priority is removed from the list and selected for scheduling during the task selection phase. In the processor selection phase, the selected task is assigned to the processor that minimizes a predefined cost criterion, such as minimizing the finish execution time of the selected task [42]. Finally, the status of the system is updated in the status update phases. At the end of this process, a valid task schedule is obtained [1,29,32,40,42,44]. Many list-based scheduling heuristics work only if a predefined simplifying assumption is maintained. Some assumptions can be justified in particular contexts, but others cannot be satisfied in real-world applications. Homogeneous processors, unlimited number of processors and no precedence constraints among

tasks, are examples of the simplifying assumptions [20]. Examples of list-based algorithms are the algorithms presented in [42,29,40, 44,15,32].

Clustering heuristics trade off inter-processor communication overhead with parallelization by allocating heavily communicating tasks to the same processor. In such heuristics, the tasks are grouped into an unlimited number of clusters. The tasks that belong to the same cluster will be assigned to the same processor. If the number of created clusters is greater than the number of available processors, clusters are merged so that the number of remaining clusters matches the number of processors. Finally, the clusters are mapped to the available processors, and the local execution of tasks within each processor is determined. In general, the complexity of clustering algorithms tends to be lower than list-based algorithms [42,5,33]. Examples of clustering algorithms are the algorithms introduced in [45,38,44].

Duplication algorithms start by running a clustering or list-based algorithm to create an initial schedule. Next, the tasks that have a large number of dependent tasks are identified and executed redundantly on the processors in which their dependent tasks are allocated to reduce inter-processor communications. Hence, the waiting time of dependent tasks will be reduced. This improvement in performance comes at the cost of increasing the complexity of scheduling process [42,48,5,1]. Examples of this type of heuristics are the algorithms presented in [5,1,30].

3.1.2. Guided random search algorithms

Guided random scheduling algorithms mimic the principles of evolution and natural genetics to evolve near-optimal task schedules. Among the various guided random algorithms, Genetic Algorithms (GAs) are the most widely used for the scheduling problem [42,48,47,18,41]. GA-based scheduling algorithms operate on a *population* of chromosomes that encode possible candidate schedules. During each iteration, or *generation*, of a GA, a set of genetic operators are run on the population to evolve a new population of chromosomes. GA-based scheduling algorithms aim to evolve near-optimal schedules after sufficient number of generations. To guide the search process, GA-based algorithms need to assess the quality, or the *fitness*, of the encoded schedules [48,18,41]. The algorithm developed by Zomaya et al. [48] is an example of this class of algorithms.

Heuristic algorithms move from one point in the search space to another using a particular transition rule. In multimodal problems, this point-to-point transition may mislead the search process. GA-based scheduling algorithms overcome this problem by working on a population of chromosomes in parallel. Hence this reduces the probability of converging to a local optimum. In contrast to heuristic algorithms, which require direct information about the application and the HeDCS to carry out scheduling, GA-based scheduling algorithms operate on chromosomes that represent possible candidate schedules.

3.1.3. Hybrid scheduling algorithms

A hybrid scheduling algorithm combines both heuristic algorithms and GAs. The Genetic List Scheduling (GLS) algorithm [19] is an example of this class of algorithms. The GLS algorithm operates on a *resource* set composed of the processors and the communication buses, and a *user* set composed of the tasks and the inter-task communications. A GA is used to evolve a set of priorities. The evolved priorities are used by a list-based algorithm to generate a task schedule. Each chromosome in the GA population encodes two sets of genes: *user* priorities and *user-resource* priorities. The user priorities genes encode priorities of all users, and they are used to select users for scheduling. The user-resource priorities are used to allocate a resource to the selected user.

3.2. Task scheduling for HeDCSs

Effective task scheduling is an essential issue in obtaining high performance in HeDCSs. Therefore, there are several algorithms proposed in the literature for the problem of task scheduling in HeDCSs. Examples of these algorithms are: Heterogeneous Earliest Finish Time (HEFT) [42], Critical Path on a Processor (CPOP) [42], Dynamic Level Scheduling (DLS) [40], Mapping Heuristic (MH) [15] and Levelized Min Time (LMT) [26].

The DLS and HEFT algorithms are two of the best existing scheduling algorithms for HeDCSs [42], and are employed as benchmark scheduling algorithms in many studies such as [6,37]. Therefore, detailed description of these two algorithms is given in this section.

3.2.1. The Dynamic Level Scheduling (DLS) algorithm

The DLS algorithm uses a quantity called *Dynamic Level* $DL(t_i, p_j)$, which is the difference between the maximum sum of computation costs from task t_i to an exit task, and the earliest start execution time of t_i on processor p_j. In this algorithm, the earliest start execution time of t_i on p_j is defined as the maximum of the *ready time* of t_i on p_j, and the time when p_j finishes the execution of its already scheduled tasks. The DLS algorithm does not schedule tasks between two previously scheduled tasks. To accommodate HeDCSs, the computation cost of a task is taken as the median of the execution times of that task over all processors. Moreover, a new quantity is added to the equation of $DL(t_i, p_j)$ to account for the various execution times of the same task on different processors. At each scheduling step, the DLS algorithm evaluates the DL values for all combinations of ready tasks and available processors. The pair of ready task and available processor that has the highest DL value is chosen for scheduling. The general time complexity of the DLS algorithm is $O(m \times n^3)$, where m is the number of processors and n is the number of tasks. The comparison study presented in [42] shows that the DLS algorithm outperforms the MH and LMT algorithms for different values of DAG size, communication to computation cost ratio, and parallelism factor (these terms will be defined in Section 5). Moreover, the performance of the DLS algorithm is comparable to that of the CPOP algorithm.

3.2.2. The Heterogeneous Earliest Finish Time (HEFT) algorithm

The HEFT algorithm starts by setting the computation costs of tasks and communication costs of edges to their mean values. Each task is assigned a value called *upward rank*. In this algorithm, the upward rank of a task t_i is the largest sum of mean computation costs and mean communication costs along any directed path from task t_i to an exit task. A task list is then generated by sorting all tasks by decreasing order of their upward rank; ties are decided on a random basis. At each scheduling step, the unscheduled task with the highest upward rank value is selected and assigned to the processor that minimizes its finish execution time, using the *insertion-based scheduling policy*. When a processor p_j is assigned a task t_i, the insertion-based scheduling policy considers all possible *idle time slots* on p_j to find a time slot of equal or greater length than the execution time of t_i on p_j. This must be done without violating the precedence constraints among tasks. An idle time slot on processor p_j is defined as the idle time space between the finish execution time and the start execution time of two consecutively scheduled tasks on p_j. The search starts from a time equal to the ready time of t_i on p_j, and proceeds until it finds the first idle time slot with the sufficient length for the computation cost of t_i on p_j. If no such idle time slot is found, the insertion-based scheduling policy inserts the selected task after the last scheduled task on p_j. The HEFT algorithm has a general time complexity of $O(m \times e)$ where m is the number of processors, and e is the number of edges. The time complexity for dense DAGs, in which the number of edges

is proportional to n^2 (where n is the number of tasks), is $O(m \times n^2)$. It has been shown in [42] that the HEFT algorithm outperforms the CPOP, DLS, MH and LMT algorithms for different values of DAG size, communication to computation cost ratio, and parallelism factor.

4. The proposed algorithm

The *Hybrid Heuristic–Genetic Scheduling* (H2GS) algorithm is a two-phase scheduling algorithm. In the first phase, the H2GS algorithm runs a heuristic scheduling algorithm, called the *Longest Dynamic Critical Path* (LDCP) algorithm [12], to quickly generate a high quality task schedule. The schedule generated by the LDCP algorithm is located at an approximate area in the search space around the optimal schedule. In the second phase, a new Genetic Algorithm, called *Genetic Algorithm for Scheduling* (GAS), searches that approximate area to improve the schedule generated by the LDCP algorithm.

4.1. The Longest Dynamic Critical Path (LDCP) algorithm

The LDCP algorithm is a list-based algorithm for task scheduling in HeDCSs [12]. The performance of list-based algorithms depends on the method used to assign priorities to the tasks of a parallel application. During a particular scheduling step, a task must be assigned a high priority if the selection of this task for scheduling ultimately leads to a shorter schedule. In this section, the problem of assigning task priorities in HeDCSs is reviewed and an effective attribute to address this problem is presented. Moreover, a brief description of the LDCP algorithm is provided in Section 4.1.2. A fully detailed description of the LDCP algorithm is given in [12].

4.1.1. Task priorities in HeDCSs

For DCSs with homogeneous processors, the *critical path* (CP) attribute of a DAG provides an effective way for assigning priorities to tasks. For a given DAG, the CP is defined as the path from an entry task to an exit task that has the greatest sum of computation costs of tasks and communication costs of edges [32]. The sum of computation costs of the tasks located on the CP determines the lower bound of the final schedule length. Hence, an efficient list-based scheduling algorithm requires proper scheduling of the tasks located on the CP. On the other hand, when two tasks are scheduled on the same processor, the communication cost between them is zero. Consequently, a CP changes dynamically during the scheduling process. To overcome the dynamic behavior of CPs, Kwok et al. [32] used an efficient attribute, called the *Dynamic Critical Path* (or DCP), to effectively select tasks for scheduling in homogeneous DCSs. The DCP is simply a CP that is computed at each intermediate scheduling step, such that the communication cost among two tasks scheduled on the same processor is considered zero.

In HeDCSs, the various computation costs of the same task on different processors present us with a problem: the DCP computed using the computation costs of tasks on a particular processor may differ from the DCP computed using the computation costs of tasks on another processor. To overcome this problem, previous scheduling algorithms for HeDCSs set the computation costs of tasks to their median values, as in the DLS algorithm [40], or their mean values, as in the HEFT algorithm [42], in order to get a single computation cost for each task. However, these techniques estimate approximate computation costs of tasks and hence, limit the ability of scheduling algorithms to precisely compute the priorities of tasks.

One important attribute that can be used to compute priorities of tasks in HeDCSs precisely is the *Longest Dynamic Critical Path* (LDCP) [12]. The LDCP is explained in Definition 1.

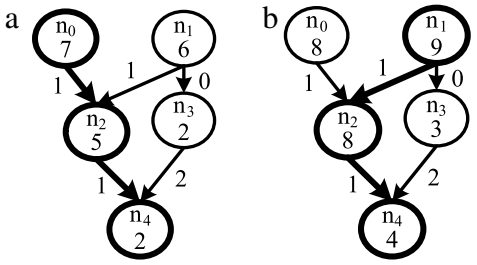


Fig. 2. The DAG in Fig. 1 constructed using the computation costs of tasks on processors (a) p_0 and (b) p_1.

Definition 1. Given a DAG with n tasks and e edges, and a HeDCS with m heterogeneous processors, the *Longest Dynamic Critical Path (LDCP)* during a particular scheduling step is a path of tasks and edges from an entry task to an exit task that has the largest sum of communication costs of edges and computation costs of tasks over all processors. Communication costs between tasks scheduled on the same processor are assumed zero, and the execution constraints are preserved.

For example, consider the application DAG and the computation costs matrix in Fig. 1. At the beginning of the scheduling process, i.e. before scheduling any task, the DCP computed using the computation costs of tasks on processor p_0 is composed from tasks t_0, t_2, t_4 and has a length of 16, as shown in Fig. 2a. However, the DCP computed using the computation costs of tasks on processor p_1 is composed from t_1, t_2, t_4 and has a length of 23, as shown in Fig. 2b. Hence, at the beginning of scheduling process the LDCP is composed from tasks t_1, t_2, t_4 and has a length of 23.

4.1.2. The LDCP algorithm

The LDCP algorithm is a list-based scheduling algorithm for a finite number of heterogeneous processors. In the LDCP algorithm, each scheduling step consists of three phases: task selection, processor selection and status update. These three phases are repeated until all the tasks of the parallel application are scheduled.

A. Task selection phase

The LDCP attribute identifies a set of tasks that play an important role in determining the *provisional* schedule length. To compute the LDCP, a *Directed Acyclic Graph that Corresponds to a Processor* (DAGP), which is explained in Definition 2, is constructed for each processor in the system. These DAGPs are constructed at the beginning of the scheduling process.

Definition 2. Given a DAG with n tasks and e edges and a HeDCS with m heterogeneous processors $\{p_0, p_1, \ldots, p_{m-1}\}$, the *Directed Acyclic Graph that Corresponds to processor* p_j, called $DAGP_j$, is the task graph constructed using the structure of the DAG, with sizes of tasks set to their computation costs on processor p_j.

The $DAGP_0$ and $DAGP_1$ shown in Fig. 2a and b, respectively, correspond to the application DAG and the HeDCS shown in Fig. 1. Through the course of this paper, the task t_i is used to refer to the ith task in the application DAG. The node n_i in $DAGP_j$ corresponds to task t_i in the application DAG with its size set to the computation cost of t_i on processor p_j. Hence, node n_i on $DAGP_j$ *identifies* task t_i on the application DAG.

For each DAGP, all nodes are assigned *upward rank* (URank) values to reflect their priority within that DAGP. The upward rank is explained in Definition 3.

Definition 3. The *upward rank* of a node n_i in a task graph $DAGP_j$, denoted as $URank_j(n_i)$, is recursively defined as:

$$URank_j(n_i) = w_j(n_i) + \max_{n_k \in succ_j(n_i)} \{c_j(n_i, n_k) + URank_j(n_k)\} \quad (1)$$

where $succ_j(n_i)$ is the set of immediate successors of n_i on $DAGP_j$; $w_j(n_i)$ is the size of n_i in $DAGP_j$; $c_j(n_i, n_k)$ is the communication cost between n_i and n_k in $DAGP_j$.

Definition 4. Given a node n_i in a task graph $DAGP_j$, the immediate successor of n_i that satisfies the maximization term in Eq. (1) is called the *upward rank associated successor (URAS)* of node n_i.

The URank values of the nodes in a given DAGP are computed recursively by traversing that DAGP upward starting from exit nodes to entry nodes. The URank value of an exit node is equal to its size. Since we recursively compute the URank values of the nodes in a given DAGP upward starting for the exit nodes, the node with the highest URank value will always be an entry node.

Theorem 1. *The node that has the highest URank value over all DAGPs identifies the entry task of the* LDCP[1].

Theorem 2. *If the tasks on the LDCP are being identified recursively downward starting from the entry task, and node n_i in $DAGP_j$ is used to identify the last identified task on the LDCP, then the URAS of node n_i on $DAGP_j$ identifies the next task on the LDCP.*[1]

The entry task of the LDCP is determined by locating a node n_i that has the highest URank value over all nodes on all DAGPs. The remaining tasks on the LDCP can be identified by recursively traversing the DAGP that contains node n_i. Traversal starts from node n_i and moves downward. During traversal, the nodes that identify the tasks on the LDCP are located using Theorem 2.

Definition 5. During a particular scheduling step, let the set of nodes N be used to identify the tasks on the LDCP. The unscheduled node in N with the highest URank value is defined as the *key node*. The DAGP in which the nodes in N are located is called the *key DAGP*.

Definition 6. During a particular scheduling step if the key node has unscheduled parents, then the unscheduled predecessor of the key node with the highest URank is defined as the *parent key node*.

At each scheduling step the key node, or the parent key node if the key node has unscheduled parents, is used to identify the task that will be selected for scheduling. Ties are broken by selecting the task with the highest number of output edges first; if more than one task exists, the tie is broken on a random basis.

B. Processor selection phase

In this phase, the selected task is assigned to a processor that minimizes its finish execution time. The insertion-based scheduling policy, described in Section 3.2.2, is used to select a processor to execute the selected task and to determine the start execution time of the selected task.

C. Status update phase

When a task is scheduled on a processor, the status of the system must be updated to reflect the new changes. The scheduling of task t_i on processor p_j means that the computation cost of t_i is no more unknown. Hence, the sizes of the nodes that identify t_i are set to the computation cost of t_i on p_j on all DAGPs. Moreover, a value of zero is assigned to all edges that extend between the nodes that identify t_i and the nodes that identify its parents that are scheduled on processor p_j. This must be done for all DAGPs to indicate the zero communication cost between tasks scheduled on the same processor. To reflect these changes, the URank values of the nodes that identify the currently scheduled task and the previously scheduled tasks are updated on all DAGPs.

D. The LDCP algorithm

The LDCP algorithm is formalized in Fig. 3. The LDCP algorithm has a general time complexity of $O(m \times n^3)$ where m is the number of processors, and n is the number of tasks.

As an example, consider the application DAG and the computation cost matrix shown in Fig. 4a and b. A stepwise trace of the LDCP algorithm along with the schedule generated by the LDCP algorithm are shown in Fig. 4c d, respectively. The schedules generated by the DLS and HEFT algorithms are shown in Fig. 4e and f, respectively. The schedule generated by the LDCP algorithm has a length of 64, which is shorter than the schedules generated by the DLS (65.5) algorithm and the HEFT (65.5) algorithm. The last schedule (Fig. 4g), with a length of 61.5, is the result of applying the complete H2GS algorithm, which includes the GAS algorithm explained in Section 4.2.

4.2. Genetic algorithm for scheduling (GAS)

In this section, the Genetic Algorithm for Scheduling (GAS) is presented. The GAS algorithm uses the schedule generated by the LDCP algorithm to create its initial population. The schedule generated by the LDCP algorithm is located at an approximate area in the search space around the optimal schedule. The GAS algorithm searches around that approximate area to improve the LDCP-generated schedule.

The GAS algorithm employs a simple genotype composed of two-dimensional (2-D) chromosomes. New encoding and decoding mechanisms are developed to allow for an efficient mapping between the simple genotype and the complex phenotype, i.e. task schedules. The encoding mechanism uses a simple and fast way to represent task schedules using 2-D chromosomes. The decoding mechanism maps any chromosome in the search space into a valid task schedule, and hence ensures a dense search space. If the scheduling problem is multimodal, a genetic algorithm might prematurely converge to a local optimum when the population diversity is not maintained by the genetic operators during the search process [2]. In order to avoid the problem of premature convergence, customized genetic operators are developed specifically for the task scheduling problem to maintain population diversity and simultaneously enable an efficient stochastic search process. The GAS algorithm can work on top of any other heuristic scheduling algorithm to optimize its output schedules. The operation of the GAS algorithm is formalized in Fig. 5.

A. Schedule encoding

The structure of the task scheduling problem, which is presented as an application DAG that must be allocated to a set of processors, requires a long binary encoded chromosome to include all of the information of a valid task schedule. The encoded schedule has to represent a task schedule in a way that maintains a set of constraints: all the tasks in the application DAG must be scheduled, any task in the application DAG must be executed only once, and the precedence constraints between tasks must be preserved.

In GAS, a 2-D string chromosome is used to encode the task schedule. The 2-D chromosome is composed of a set of substrings, such that the number of substrings in the chromosome is equal to the number of processors in the system. The jth substring in the chromosome represents processor p_j in the computing system. A task schedule is encoded by traversing each processor in the schedule. When processor p_j is traversed, the labels of the tasks scheduled on processor p_j are copied, in the same order, into the jth substring of the chromosome. Therefore, the encoding of a

[1] The Proofs of Theorems 1 and 2 can be found in page 402 of Ref. [12].

```
Construct DAGPs for all processors in the system
while there are unscheduled tasks do
  Find the key DAGP
  Find the key node in the key DAGP
  if the key node has no unscheduled parents then
    Identify the selected task using the key node
  else
    Find the parent key node
    Identify the selected task using the parent key node
  end if
  Compute the finish time of the selected task on every processor in the system
  Find the selected processor that minimizes the finish time of the selected task
  Assign the selected task to the selected processor
  Update the size of the nodes that identify the selected task on all DAGPs
  Update the communication costs on all DAGPs
  Update the URank values of the nodes that identify the scheduled tasks on
    all DAGPs
end while
```

Fig. 3. The LDCP algorithm.

given task schedule will always lead to the same chromosome. For example, consider the computation cost matrix, composed of processors p_0 and p_1, and the application DAG shown in Fig. 1, a valid task schedule for this application is the schedule shown in Fig. 6a. This schedule can be encoded by copying the tasks scheduled on processors p_0 (tasks t_0, t_1, t_3) and p_1 (tasks t_2, t_4), in the same order, into the first and second substrings of the chromosome, respectively, as shown in Fig. 6b. Therefore, the chromosome in Fig. 6b encodes the task schedule in Fig. 6a.

B. Chromosome decoding

The 2-D string chromosome stores two types of genetic information: (i) the assignment of each task in the application DAG to one of the processors in the computing system, and (ii) the execution ordering of tasks assigned to the same processor. The structure of the application DAG along with a list-based scheduling mechanism are used to decode the information stored in the chromosome in order to create a valid task schedule.

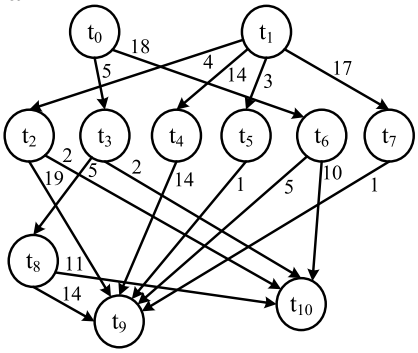
When a chromosome is decoded, the DAG of the unscheduled application is traversed downward, starting from the entry tasks. At each decoding step, the ready tasks of the unscheduled application DAG are defined and assigned to processors according to the location of their corresponding labels in the chromosome. If the label of ready task t_i is located in the jth substring, then task t_i is assigned to processor p_j using the insertion-based scheduling policy described in Section 3.2.2. If, at a given decoding step, the labels of two or more ready tasks are located simultaneously on the jth substring, only the first ready task, with a label preceding the labels of other ready tasks on the jth substring, is selected and assigned to processor p_j. The transversal of tasks of the application continues until all task labels in the chromosome are decoded. Consider the decoding of the chromosome in Fig. 6c that represents a valid schedule for the application DAG and computation cost matrix in Fig. 1. The entry tasks of this application DAG are t_0 and t_1. Although the labels of these entry tasks in the chromosome are located in the substring of processor p_0, the label of t_0 precedes t_1, and hence t_0 is selected and assigned to p_0. In the next step, t_1 is the only ready task, and therefore it is assigned to p_0. After assigning t_0 and t_1, tasks t_2 and t_3 become ready with their labels located in the substrings of p_1 and p_0, respectively; hence, t_2 is assigned to p_1 and t_3 is assigned to p_0. In the final step, t_4 becomes ready with its label located in the substring of p_1, and therefore it is assigned to p_1. The result of this decoding is the schedule shown in Fig. 6a. These decoding steps can also be used to decode the chromosome shown in Fig. 6b to create the schedule presented in Fig. 6a.

This decoding mechanism ensures that the decoding of any chromosome in the search space will always lead to a valid schedule. Moreover, the task schedule produced by the decoding of a given chromosome can also be produced by the decoding of other chromosomes. For example, the decoding of the two chromosomes shown in Fig. 6b and c produces the same schedule shown in Fig. 6a as discussed before. This many-to-one mapping from the genotype to the phenotype parallels the degeneracy of the genetic code in nature [43]. Genetic code degeneracy allows for silent mutations, and has been shown to result in improved genetic diversity in genetic search algorithms [35].

C. Initialization

The first step in the GAS algorithm is the creation of the initial population. The schedule generated by the LDCP algorithm is encoded, and the resulting chromosome is inserted into the initial population. In addition to the chromosome produced

a



b

task	p0	p1
t0	4	6
t1	15	22.5
t2	4	6
t3	13	19.5
t4	10	15
t5	7	10.5
t6	8	12
t7	4	6
t8	12	18
t9	6	9
t10	9	13.5

c

step	task	processor	DAGP
1	t1	p0	DAGP0
2	t0	p1	DAGP1
3	t3	p1	DAGP1
4	t8	p0	DAGP1
5	t4	p0	DAGP0
6	t2	p0	DAGP0
7	t6	p1	DAGP1
8	t5	p1	DAGP1
9	t7	p0	DAGP1
10	t9	p0	DAGP1
11	t10	p0	DAGP0

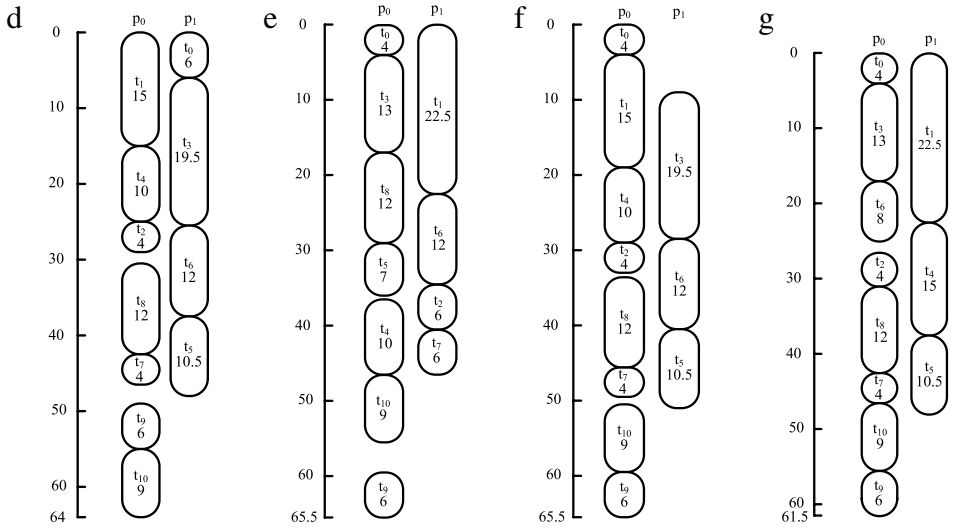


Fig. 4. (a) A DAG of a parallel application, (b) a computation cost matrix of a HeDCS, (c) stepwise trace of the LDCP algorithm, the schedules generated by the (d) LDCP, (e) DLS, (f) HEFT, and (g) H2GS algorithms.

Create the initial *population* using the LDCP-generated schedule

while termination criteria is not met **do**

Evaluate the fitness of the chromosomes in the *population*

Copy the best 10% of the population chromosomes to the *elitism set*

Select chromosomes from the *population* to the *mating pool*

Apply the *swap crossover* operator on the chromosomes in the *mating pool*

Apply the *swap mutation* operator on the chromosomes in the *mating pool*

Combine the chromosomes in the *mating pool* and *elitism set* to generate

the new *population*

end while

Fig. 5. The GAS algorithm.

using the LDCP algorithm, GAS creates a set of processor-based chromosomes and randomly-generated chromosomes to ensure a diverse initial population.

For each processor in the computing system, one processor-based chromosome is created and inserted into the initial population. The processor-based chromosome that corresponds

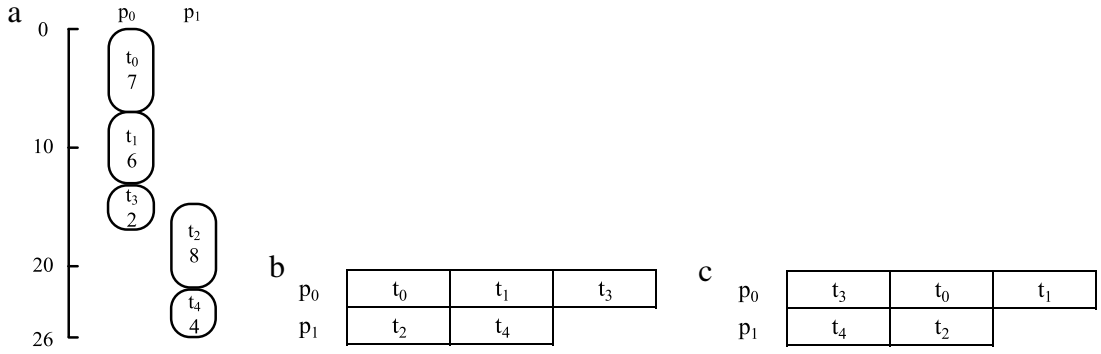


Fig. 6. (a) An example of a task schedule (b) + (c) chromosome examples.

to processor p_j is created by randomly sorting the labels of all tasks and allocating them to the jth substring. No task labels are assigned to the substrings that correspond to the other processors. The randomly-generated chromosomes are created by randomly assigning the labels of all the tasks to the substrings of the chromosome.

The insertion of the LDCP-generated schedule, which enables GAS to exploit the high quality building blocks discovered by LDCP, and the processor-based and randomly-generated chromosomes, which ensure a diverse population, into the initial population of GAS allows for an effective exploration of the search space. To reduce the complexity of the GAS algorithm, the number of chromosomes in the population, or the *population size*, is fixed throughout a GAS run. The smallest population is composed of the chromosome produced using the LDCP algorithm in addition to the processor-based chromosomes. Hence, the size of the smallest population is equal to the number of processors in the system plus 1.

D. Fitness evaluation

The fitness evaluation of a chromosome is quite simple. First, each chromosome in the population is decoded, as described in the chromosome decoding subsection, to create a task schedule. The fitness of a chromosome is equal to $1/l$, where l is the length of its decoded schedule. The fact that the fitness function is simple significantly enhances the performance of the GAS algorithm, and leads to an efficient search process.

E. Selection and elitism

Copies of the fittest 10% of the chromosomes in the population are copied without change to the elitism set. This mechanism guarantees that the best chromosomes are never destroyed by either the crossover or the mutation operators.

A rank-based selection mechanism with replacement is used to select chromosomes for the mating pool. Unlike proportional selection, in which individuals with high fitness values can dominate the population in a few generations since the selection probability of a chromosome is proportional to its fitness, rank-based selection reduces selection pressure for chromosomes with superior fitness values by assigning selection probability to each chromosome based on the rank of this chromosome compared to other individuals [21]. Therefore, for task scheduling problems with multimodal search spaces, rank-based selection maintains population diversity and assists in reducing premature convergence of the population toward a sub-optimal point on the fitness surface [22,21]. The size of the mating pool is equal to 90% of the population size.

F. Swap crossover

Swap crossover works on two chromosomes in the mating pool, called *parent chromosomes*, to produce two *offspring chromosomes* each with genetic material from both parents. For each parent chromosome, one substring is chosen for crossover such that two crossover points are randomly located on each selected substring. Swap crossover exchanges the relative ordering of the tasks delineated by the two crossover points between the two parent chromosomes.

Swap crossover works by creating an intermediate modified chromosome, called the *mask chromosome*, for each parent chromosome. In the mask chromosome, the task-slice located between the two crossover points is removed and replaced by a task-slice that is delineated by the two crossover points from the other parent. If the task labels on the inserted task-slice do not match the task labels on the removed task-slice, then the task labels on the mask chromosome that match the labels on the inserted task-slice, and consequently do not have matching labels on the removed task-slice, are marked as *don't move* (DM). For example, consider the two parent chromosomes shown in Fig. 7a and b. Two crossover points are located on each parent chromosome: the crossover points CP11 and CP12 are located on the chromosome shown in Fig. 7a, and the crossover points CP21 and CP22 are located on the chromosome shown in Fig. 7b. The mask chromosomes that correspond to the parent chromosomes in Fig. 7a and b are shown in Fig. 7c and d, respectively. The DM tasks in the mask chromosomes are marked by asterisks.

To create the offspring chromosome, both the mask chromosome and its parent chromosome are traversed string by string, starting with the first task label of each substring. If the current task label on the parent chromosome is identical to the current task label on the mask chromosome, then it is moved to the offspring chromosome. However, if the current task label on the mask chromosome is marked as DM, then the current task label on the mask chromosome is deleted from both the parent chromosome and mask chromosome. If the current task label on the parent chromosome is different from the current task label on the mask chromosome, only the current task label on the parent chromosome is moved to the offspring chromosome. Finally, before traversing to the next substring, all task labels left on the current substring of the mask chromosome and do not have matching labels on the offspring chromosome, are moved in the same order to the offspring chromosome. Fig. 7e shows the offspring chromosome created by the parent chromosome in Fig. 7a and its mask chromosome in Fig. 7c. Also, the offspring chromosome shown in Fig. 7f is created by the parent chromosome in Fig. 7b and its mask chromosome in Fig. 7d.

As mentioned before, swap crossover copies the relative ordering between the tasks of the inserted task-slice into the offspring chromosome. For example in the chromosomes shown in Fig. 7a, task t_4 is scheduled before task t_2; while in the inserted task-slice copied form the chromosome in Fig. 7b, task t_4 is scheduled after task t_2. Swap crossover copies the relative ordering between tasks t_4 and t_2 from the inserted task-slice to the offspring chromosome. Hence, task t_4 is scheduled after task t_2 in the offspring chromosome as shown in Fig. 7e. Moreover, swap

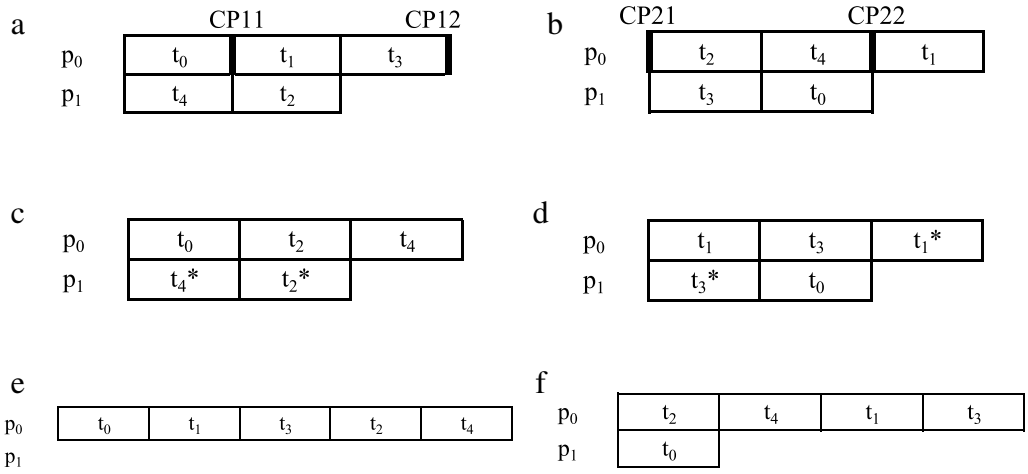


Fig. 7. The swap crossover operator.

crossover ensures that each task in the application DAG is assigned to exactly one processor in the computing system. Therefore, the customized crossover operator employed by GAS improves the population diversity by creating new chromosomes with genetic information inherited form both parents, and concurrently enables an efficient search process.

The swap crossover is applied with crossover probability of P_c to chromosomes in the mating pool. For example, if P_c is equal to 0.7, which matches the value of P_c that is obtained after tuning the GAS algorithm as discussed in Section 5.2.2, then 70% of the chromosomes in the mating pool, on average, are crossed-over.

G. Swap mutation

Swap mutation is applied to chromosomes in the mating pool to enhance the diversity of the population. To mutate a chromosome, two task labels are randomly selected and swapped. Swap mutation is applied with probability of P_m to the chromosomes in the mating pool. As will be discussed in Section 5.2.2, several values of mutation probability are examined to tune the GAS algorithm, and the tuning results suggest that a P_m value of 0.5 enables GAS to obtain the shortest schedules.

After applying the swap crossover and swap mutation operators, the chromosomes in the mating pool are combined with the chromosomes in the elitism set to create the new population.

H. Termination criterion

Although static scheduling takes place offline, a limit must be placed on the running time of the scheduling algorithm to provide a practical solution. The maximum running time of the GAS algorithm depends on the characteristics of the parallel application. As will be discussed in Section 5.2.3, coarse-task parallel applications, or parallel applications that are composed of tasks with relatively long execution times, require high quality task schedules because of the relatively high cost of a faulty scheduling decision. On the other hand, parallel applications that are composed of fine tasks, which execute quickly, require lower quality task schedules, still near-optimal, that can be produced within short compilation times. To provide a practical solution, the GAS algorithm runs for a predetermined number of generations which is set according to the characteristics and real-world context of the parallel application.

5. Results and analysis

In this section, the performance of each phase of the H2GS algorithm is presented in comparison with the DLS and HEFT algorithms, which are two of the best existing scheduling algorithms for HeDCSs as discussed in Section 3.2. An explicit comparison with some other well-known scheduling algorithms for HeDCSs, such as the CPOP, MH and LMT algorithms, is not carried out as the DLS and HEFT algorithms have already been tested against them, and have given better or, at worst, very similar results [42].

To carry out the comparison, the H2GS, DLS and HEFT algorithms are simulated. In order to avoid any bias toward a particular graph structure, two sets of benchmark application graphs are used: randomly-generated application graphs and regular application graphs that correspond to three numerical applications.

5.1. Performance metrics

The performance metrics chosen for the comparison are the Normalized Schedule Length (NSL) [5], speedup [42] and running time of the algorithms. The three metrics are explained below:

- *Normalized Schedule Length (NSL):* the NSL of a task schedule is defined as the normalized schedule length to the lower bound of the schedule length. It is calculated using Eq. (2):

$$NSL = \frac{Schedule\ Length}{\sum_{t_i \in CP_{lower}} c_{i,a}} \tag{2}$$

where the CP_{lower} is the CP of the unscheduled application DAG, based on the computation cost of tasks on the fastest processor p_a. The denominator of Eq. (2) is equal to the sum of computation costs of tasks located on CP_{lower}, when they are executed on p_a.

- *Speedup:* the speedup of a task schedule is the ratio of the serial schedule length obtained by assigning all tasks to the fastest processor, to the parallel execution time of the task schedule.

- *Running Time:* the running time of a scheduling algorithm is defined as the execution time required to produce the output schedule.

The average NSL, speedup and running time over a set of application DAGs are computed for the scheduling algorithms.

5.2. Performance results on random graphs

5.2.1. Creation of random graphs

A set of randomly-generated DAGs is created using a random DAG generator. The random DAG generator has a set of input parameters that determines the characteristics of the generated DAGs. These input parameters are described below:

- *DAG size*, n: the number of tasks in the application DAG.

- *Communication to computation cost ratio*, *CCR*: the average communication cost divided by the average computation cost of the application DAG.

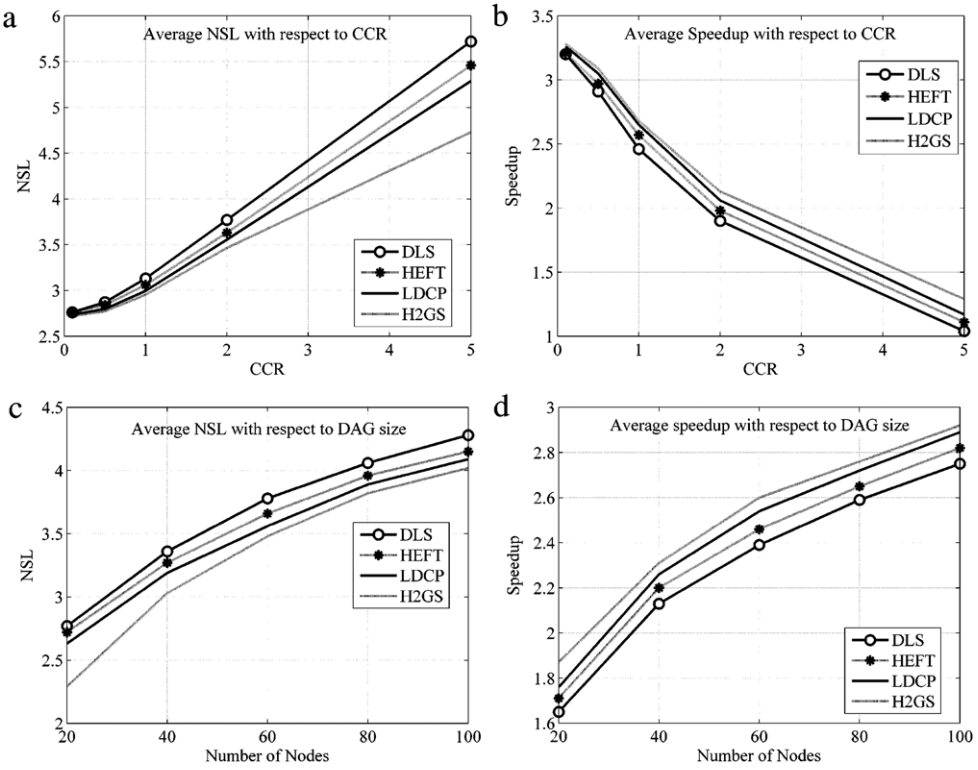


Fig. 8. Average NSL and speedup on random graphs.

- *Parallelism factor*, α: the number of levels of the application DAG is calculated by randomly generating a number, using a uniform distribution with a mean value of $\frac{\sqrt{n}}{\alpha}$, and then rounding it up to the nearest integer. The width of each level is calculated by randomly generating a number using a uniform distribution with a mean value of $\alpha \times \sqrt{n}$, and then rounding it up to the nearest integer [42]. A low α value leads to a DAG with a low parallelism degree [5].

- *Computation cost heterogeneity factor*, h: a high h value indicates high variance of the computation costs of a task, with respect to the processors in the system, and *visa versa*. If the heterogeneity factor is set to 0, the computation cost of a task is the same for all processors. The average computation cost of a task $t_i(\overline{w_i})$ is randomly generated using a uniform distribution with a mean value of W. The value of W, which represents the average computation cost of all tasks in the parallel application, does not affect the comparison results, and it can be any arbitrary value. If there are m processors in the HeDCS, the computation cost of a task t_i for each processor is set by randomly selecting m computation cost values of t_i from the range $[\overline{w_i} \times (1 - \frac{h}{2}), \overline{w_i} \times (1 + \frac{h}{2})]$. The m selected computation cost values of t_i are sorted in an increasing order. The computation cost value of t_i on processor p_0 is set to the first (i.e lowest) computation cost. The computation cost of t_i on processor p_1 is set to the second value. This allocation continues until all processors are processed [42].

The created random DAGs set consists of 2000 application DAGs with four different numbers of processors, varying from 2 to 8 with an increment of 2. For each number of processors, we use five different DAG sizes varying from 20 to 100 nodes with an increment of 20; five different *CCR* values: 0.1, 0.5, 1.0, 2.0 and 5.0; four α values: 0.5, 1.0, 2.0 and 5.0; and five h values: 0.1, 0.2, 0.4, 0.6 and 0.8. Hence, corresponding to each number of processors, a set of 500 application DAGs is created. The large set of random graphs, which consists of 2000 DAGs with diverse characteristics, prevents bias toward one specific scheduling algorithm. The parameter values, which are used in this subsection and the next subsection, are essentially the same as those used by Topcuoglu et al. in [42].

5.2.2. Tuning the GAS algorithm

To select the parameter values of GAS, which include P_c, P_m, population size and number of generations, 50 DAGs are randomly selected from the random DAGs set, which consists of 2000 DAGs, and used to tune GAS. Twenty values of P_m and P_c are tested, varying from 0.05 to 1.00 with an increment of 0.05. Ten values of number of generations are tested, varying from 5 to 50 with an increment of 5. The population size is varied from 10 to 50 with an increment of 5. GAS, on average, achieved the shortest task schedules when P_c and P_m are equal to 0.7 and 0.5, respectively. Moreover, the improvement in performance, in terms of schedule length, obtained by running GAS for more than 20 generations, is less than 10% compared to the schedules obtained using 20 generations. For the number of processors used in this study, a population of 25 chromosomes is found to produce satisfactory results since larger values of population size do not lead to effective improvement in performance and, at the same time, considerably increase the running time of the GAS algorithm. Based on these results, the values of P_c, P_m, population size and number of generations are set to 0.7, 0.5, 25 and 20, respectively, to produce the results reported in this paper. It is worth mentioning that there are several other approaches that can be used to select the parameter values of GAS [14].

5.2.3. Performance results

The NSL and speedup values achieved by each phase of the H2GS algorithm and the DLS and HEFT algorithms are compared with respect to various DAG sizes and CCR values. Moreover, the running time of the scheduling algorithms is studied with respect to the number of nodes.

The NSLs produced by the H2GS, DLS and HEFT algorithms with respect to *CCR* are shown in Fig. 8a. The average NSL of the first phase of H2GS, i.e. the LDCP algorithm, is shorter than the DLS and HEFT algorithms by: (1.0%, 0.9%), (2.4%, 1.6%), (4.4%, 2.0%), (5.7%, 2.2%) and (7.5%, 3.1%), for *CCR* of: 0.1, 0.5, 1.0, 2.0 and 5.0, respectively. Moreover, as shown in Fig. 8a, GAS enhances the

schedules generated by LDCP to generate shorter schedules. The average NSL value of the complete H2GS algorithm is shorter than the DLS and HEFT algorithms by: (1.5%, 1.5%), (3.3%, 2.5%), (5.8%, 3.4%), (8.1%, 4.7%) and (17.3%, 13.3%), for *CCR* of: 0.1, 0.5, 1.0, 2.0 and 5.0, respectively. The first value of each parenthesized pair is the improvement achieved by each phase of the H2GS algorithm over the DLS algorithm, while the second value is the improvement of each phase of H2GS over the HEFT algorithm. This convention for representing results will be adhered throughout this paper, unless an exception is explicitly noted.

The speedup values achieved by the scheduling algorithms with respect to *CCR* are shown in Fig. 8b. The average speedup value of the first phase of the H2GS algorithm is higher than those returned by the DLS and HEFT algorithms by: (1.9%, 1.4%), (4.7%, 2.6%), (7.0%, 2.5%), (8.2%, 4.1%) and (12.3%, 5.0%), for *CCR* of: 0.1, 0.5, 1.0, 2.0 and 5.0, respectively. Furthermore, as shown in Fig. 8b, the GAS algorithm improves the speedup obtained by the LDCP algorithm. The average speedup achieved by the complete H2GS algorithm is greater than those gained by the DLS and HEFT algorithms by: (2.5%, 2.0%), (5.9%, 3.8%), (9.1%, 4.4%), (12.1%, 7.7%) and (24.1%, 16.1%), when the *CCR* is equal to: 0.1, 0.5, 1.0, 2.0 and 5.0, respectively.

In these experiments, the first phase of H2GS outperforms the DLS and HEFT algorithms for all tested *CCR* values in terms of both NSL and speedup. As the value of *CCR* increases, inter-processor communication overhead dominates computation and hence, the performance of the scheduling algorithms tends to degrade. However, as shown in Fig. 8a and b, the first phase of H2GS, or the LDCP algorithm, is more effectively able to deal with the increase in communication cost compared to both the DLS and HEFT algorithms. The ability of the LDCP algorithm to efficiently handle the increase in communication overhead can be explained as follows. As the *CCR* value increases, the LDCP attribute will be dominated by tasks that have high inter-processor communication overhead. Hence, heavily communicating tasks will be identified and selected for scheduling before other tasks. Moreover, at each scheduling step, the insertion-based scheduling policy assigns the selected task to a processor that minimizes its finish executing time. Hence, heavily communicating tasks will be selected and assigned to the same processor if such an assignment leads to a shorter provisional schedule. Finally, the status update phase ensures that the LDCP attribute is effectively updated during the scheduling process. Therefore, heavily communicating tasks will be regularly identified and scheduled to reduce the final schedule length. Moreover, as shown in Fig. 8a and b, the improvement of performance achieved by the second phase of H2GS, or the GAS algorithm, increases as the value of *CCR* increases. Hence, the improvement of performance obtained by the complete H2GS algorithm over the DLS and HEFT algorithms, as well as the LDCP algorithm, increases as *CCR* becomes higher.

The average NSL values achieved by the scheduling algorithms with respect to DAG size are shown in Fig. 8c. The average NSL value of the first phase of H2GS is shorter than those of the DLS and HEFT algorithms by: (5.1%, 3.1%), (5.1%, 2.3%), (5.8%, 2.7%), (4.1%, 1.7%) and (4.3%, 1.3%), for DAG sizes of: 20, 40, 60, 80 and 100, respectively. Moreover, the average NSL value produced by the complete H2GS algorithm is shorter than the DLS and HEFT algorithms by: (17.2%, 15.5%), (9.9%, 7.3%), (7.9%, 4.9%), (6.0%, 3.6%) and (6.1%, 3.2%), when the number of nodes is equal to: 20, 40, 60, 80 and 100, respectively. The speedup values obtained by the scheduling algorithms with respect to DAG size are shown in Fig. 8d. The average speedup achieved by the first phase of H2GS is greater than the DLS and HEFT algorithms by: (6.6%, 3.1%), (6.0%, 2.6%), (6.6%, 3.4%), (4.9%, 2.4%) and (5.0%, 2.4%), when the number of nodes is equal to: 20, 40, 60, 80 and 100, respectively. Furthermore, the average speedup obtained by the complete H2GS algorithm is higher than the DLS and HEFT algorithms by: (13.2%, 9.4%), (8.5%, 5.1%), (9.0%, 5.7%), (6.6%, 4.0%) and (6.3%, 3.6%), for DAG sizes of: 20, 40, 60, 80 and 100, respectively. The first phase of H2GS achieves better results than both the DLS and HEFT algorithms in terms of NSL as well as speedup, for any number of nodes in our range. Furthermore, the second phase of H2GS improves the LDCP-generated schedules, which was created in the first phase, to obtain task schedules of better quality for all tested values of DAG size.

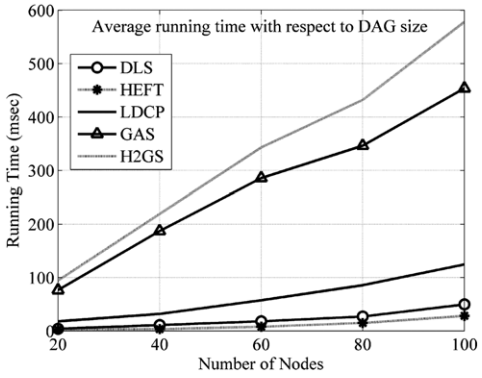


Fig. 9. Average running time with respect to DAG size.

Fig. 9 shows the average running time of each phase of the H2GS algorithm as well as the DLS and HEFT algorithms with respect to the DAG size. The running times of all scheduling algorithms are much lower than the execution times reported for parallel applications run on real parallel computing systems [25]. The average running time of the first phase of H2GS is higher than both the DLS and HEFT algorithms by 189.3% and 454.9%, respectively. The average running time of the complete H2GS algorithm is higher than the DLS and HEFT algorithms by 1428.8% and 2832.0%, respectively. On average 81.1% of the H2GS running time comes from the GAS phase. An advantage to the vast running time difference between the LDCP phase and the GAS phase is that the LDCP algorithm is run first to find high quality task schedules. The GAS algorithm is run for a predefined number of generations to further improve the schedules created by the LDCP algorithm. The running of GAS can be interrupted at any time.

The scheduling process of the DLS, HEFT and LDCP algorithms depends on the relative values of the computation costs of tasks and communication costs of edges with respect to each other, rather than the absolute costs of tasks and edges [40,42,12]. Moreover, all operators of the GAS algorithm do not depend on the absolute values of the computation costs of tasks and communication costs of edges, but they can be affected by the relative costs of tasks and edges. Therefore, the structure, i.e. the allocation of the tasks to the processors and the execution order of the tasks assigned to each processor, of the schedules generated by the DLS, HEFT, LDCP and H2GS algorithms depends only on the relative costs of tasks and edges. For example, assume that the communication costs of all edges in the DAG shown in Fig. 4a is scaled by a factor of a and the values of the computation cost matrix of the HeDCS shown in Fig. 4b are scaled by the same factor a, the structure of the schedules generated by the LDCP, DLS, HEFT and H2GS algorithms for the scaled DAG and the scaled computation cost matrix will be identical to the structure of the schedules shown in Fig. 4d, e, f and g, respectively, that are achieved by the algorithms before scaling the DAG and the computation cost matrix. However, the lengths of schedules created by the LDCP, DLS, HEFT and H2GS algorithms for the scaled DAG and the scaled computation cost matrix are equal to $a \times 64$, $a \times 65.5$, $a \times 65.5$ and $a \times 61.5$, respectively, compared to schedule lengths of 64, 65.5, 65.5 and 61.5, respectively, obtained using the same algorithms before applying the scaling factor. This reasoning suggests that the

cost of faulty scheduling decisions obtained using these algorithms for parallel applications with coarse tasks, i.e. tasks with large computation costs, and intensive inter-task data transfers, i.e. edges with large communication costs, is higher than the cost of faulty schedules computed for applications with fine tasks and small inter-task data transfers.

The two-phase feature of the H2GS algorithm allows for the customization of the scheduling process based on the scheduling problem under consideration. When the cost of faulty scheduling decision is high and the search space of the scheduling problem is multimodal [2], the GAS algorithm can be adjusted to run for a large number of generations such that at each new generation the GAS algorithm exploits the high quality building blocks that are discovered in the previous generations to explore new areas in the search space. The large number of iterations along with the customized genetic operators that are designed to maintain population diversity enable the search process to converge to the global optimum and prevent premature convergence to a local optimum [2]. Another approach to avoiding premature convergence of genetic algorithms, especially when applied to multimodal problems, utilizes a large population consisting of multiple subpopulations within the niches defined by the various optima of the fitness surface. This enhances the diversity of the population and hence improves the search, as witnessed by [39]. On the other hand, when fast scheduling decisions are required, the GAS algorithm can be run for a predefined number of generations in order to generate task schedules within a specific time window, but not necessarily optimally. The characteristics of the search space of a particular application can be estimated from observations of previous runs of this application on the HeDCS.

5.3. Performance results on regular graphs

In this section, the performance of the scheduling algorithms is studied with respect to the application DAGs of three real world parallel algorithms: Gaussian elimination algorithm [42,44], fast Fourier transform algorithm [8,10] and a molecular dynamics code given in [28,42]. Since the structure of the regular graphs is known, there is no need for the parallelism factor parameter. The *CCR* and *h* parameters have the same set of values here as in Section 5.2. The NSL and speedup of the regular graphs will be presented with respect to *CCR*. Hence, the order of parenthesized pairs will correspond to *CCR* values of: 0.1, 0.5, 1.0, 2.0 and 5.0, respectively. This convention for representing results will be adhered throughout this section.

5.3.1. Gaussian elimination

The Gaussian elimination algorithm is characterized by the size of the input matrix. If N is the size of the input matrix, the number of nodes in the task graph is equal to $\frac{N^2+N-2}{2}$ [42]. For the experiments of the Gaussian elimination algorithm, the size of input matrix (N) is used in place of the DAG size (n).

For the NSL comparison, the matrix size is varied from 5 to 20, with an increment of 1, and the number of processors is set to 5. Fig. 10a shows the average NSLs produced by each scheduling algorithm in relation to *CCR*. The average NSL value of the first phase of H2GS is shorter than those of the DLS and HEFT algorithms by: (0.5%, 0.4%), (0.9%, 0.7%), (1.4%, 1.2%), (2.4%, 1.7%) and (3.7%, 2.8%). Moreover, the NSL value of the complete H2GS algorithm is shorter than the DLS and HEFT algorithms by: (1.5%, 1.4%), (2.1%, 1.9%), (2.8%, 2.6%), (4.0%, 3.3%) and (5.8%, 4.9%).

Fig. 10b shows the speedup values of the scheduling algorithms with respect to *CCR* when the number of processors is varied from 2 and 8, with an increment of 2, and the size of the input matrix is set to 20. The first phase of H2GS has a higher speedup value than the DLS and HEFT algorithms by: (0.8%, 0.5%), (1.0%, 0.8%), (2.4%, 1.3%), (2.9%, 1.8%) and (3.7%, 2.8%). The average speedup obtained by the complete H2GS algorithm is greater that the DLS and HEFT algorithms by: (1.9%, 1.6%), (2.3%, 2.2%), (3.9%, 2.8%), (4.3%, 3.2%) and (6.3%, 5.3%).

5.3.2. Fast Fourier transform

The task graph of the fast Fourier transform (FFT) algorithm is characterized by the size of the input vector. For an input vector of size M, the total number of nodes in the task graph is equal to $(2\times M-1)+(M\times \log_2 M)$. As in Gaussian elimination experiments, the size of the input vector (M) is used in place of the DAG size.

To study the NSL values of the scheduling algorithms, the size of the input vector is varied between 2 and 32, incrementing by a power of 2, and the number of processors is set to 5. Fig. 10c shows the average NLS values of the scheduling algorithms with respect to *CCR*. The average NSL obtained by the first phase of H2GS is shorter than the DLS and HEFT algorithms by: (0.8%, 0.9%), (1.6%, 1.3%), (2.5%, 2.3%), (3.0%, 3.4%) and (4.9%, 5.8%). The average NSL value achieved by the complete H2GS algorithm is better than the DLS and HEFT algorithms by: (1.7%, 1.8%), (2.8%, 2.5%), (3.7%, 3.7%), (5.5%, 6.3%) and (7.4%, 9.0%).

Fig. 10d presents the speedup values obtained by the four scheduling algorithms with respect to *CCR*, when the size of the input vector is set to 32, and the number of processors is varied from 2 to 8 with an increment of 2. The first phase of H2GS achieved a greater speedup value than the DLS and HEFT algorithms by: (0.6%, 0.7%), (1.2%, 1.1%), (2.1%, 2.2%), (4.0%, 4.0%) and (5.7%, 8.2%). Moreover, the average speedup value of the complete H2GS algorithm is greater than the DLS and HEFT algorithms by: (1.8%, 1.9%), (2.8%, 2.7%), (4.9%, 5.0%), (7.0%, 6.9%) and (9.5%, 12.0%).

5.3.3. Molecular dynamics code

The performance of the scheduling algorithms is compared with respect to the application DAG of the molecular dynamics code given in [28,42]. Since the number of tasks (41 tasks) and the graph structure are known, only the *CCR* and *h* values are used in this experiment.

Fig. 10e shows the average NSL values of the scheduling algorithms with respect to *CCR* when the number of processors is set to 5. The first phase of H2GS outperforms the DLS and HEFT algorithms in terms of NSL by: (1.2%, 1.4%), (1.5%, 2.3%), (1.8%, 2.9%), (2.8%, 3.2%) and (10.4%, 5.7%). The average NSL value obtained by the complete H2GS algorithm is better than the HEFT and DLS algorithms by: (1.5%, 1.7%), (2.2%, 3.0%), (3.7%, 4.8%), (6.5%, 6.9%) and (14.8%, 10.3%).

Fig. 10f presents the average speedup values of the scheduling algorithms with respect to *CCR*. Since the maximum number of tasks in any level of the molecular dynamics code DAG is less than 7, the number of used processors is varied from 2 to 7 with an increment of 1 [42]. The average speedup value gained by the first phase of H2GS is higher than the DLS and HEFT algorithms by: (0.7%, 0.8%), (1.2%, 3.0%), (5.4%, 7.2%), (6.9%, 9.3%) and (10.0%, 12.5%). Moreover, the speedup values obtained by the complete H2GS algorithm outperform the DLS and HEFT algorithms by: (2.5%, 2.6%), (3.3%, 5.0%), (5.6%, 7.4%), (10.9%, 13.4%) and (15.8%, 18.5%).

For real world applications, the H2GS algorithms outperform the DLS and HEFT algorithms in terms of schedule length and speedup. The general trend of increasing improvement in performance obtained by each phase of the H2GS algorithm over the DLS and HEFT algorithms as *CCR* increases, is observed here as well. This provides clear indication that there is a trend of improved performance with increasing *CCR*.

6. Summary and conclusions

In this paper, we present a new algorithm, called H2GS, for static task scheduling on heterogeneous distributed computing

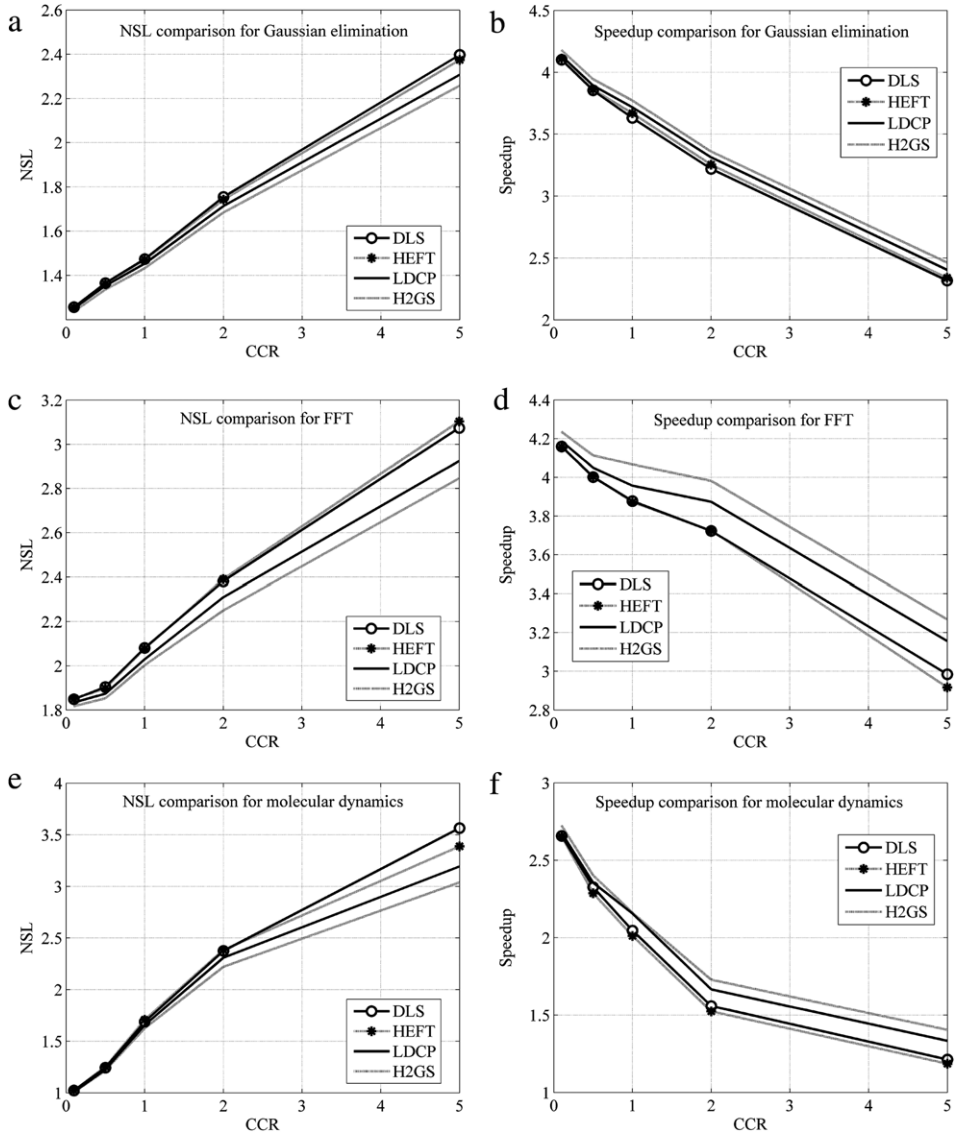


Fig. 10. Average NSL and speedup on regular graphs.

systems (HeDCSs). H2GS is a two-phase algorithm, with the first phase being a list-based scheduling heuristic, called the Longest Dynamic Critical Path algorithm (or LDCP). The LDCP algorithm generates a near-optimal task schedule, which in turn is injected into the second phase of H2GS: an especially designed Genetic Algorithm for Scheduling (or GAS). GAS uses a simple chromosome structure to represent task schedules. A new mapping procedure is developed to map every possible chromosome in the search space into a valid task schedule. Moreover, a set of customized genetic operators are introduced to further optimize the schedule received from the LDCP algorithm. The degree of improvement achieved by GAS depends on the number of generations that GAS is run for, which is user-controllable. A user with strict scheduler execution-time requirements will have to run GAS for a few generations, while one with loose execution-time requirements (e.g. off-line optimization) can run it for a large number of generations to evolve the shortest possible schedules.

On randomly-generated graphs, the H2GS algorithm shows significant improvement, in terms of Normalized Schedule Length and Speedup, over two leading existing scheduling algorithms for HeDCSs: the HEFT and DLS algorithms. H2GS is also compared against HEFT and DLS on three real-world parallel applications: Gaussian elimination, fast Fourier transform and a molecular dynamics code, with favorable results in every case. The improvement in performance achieved by the two-phased H2GS algorithm, over both the DLS and HEFT algorithms tends to increase as *CCR* increases. Due to its ability to generate high quality task schedules at high *CCR* values, the H2GS algorithm provides a practical solution for task scheduling on HeDCSs for applications with high communication costs.

We plan to extend the H2GS algorithm to partially-connected networks of heterogeneous processors. This will allow the use of the H2GS algorithm for a wide range of HeDCSs.

References

[1] I. Ahmad, Y.K. Kwok, On exploiting task duplication in parallel program scheduling, IEEE Trans. Parallel Distrib. Syst. 9 (1998) 872–892.
[2] J. Andre, P. Siarry, T. Dognon, An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization, Adv. Eng. Softw. 32 (2001) 49–60.
[3] R. Bajaj, D.P. Agrawal, Improving scheduling of tasks in a heterogeneous environment, IEEE Trans. Parallel Distrib. Syst. 15 (2004) 107–118.
[4] S. Bansal, P. Kumar, K. Singh, Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs, J. Parallel Distrib. Comput. 65 (2005) 479–491.
[5] S. Bansal, P. Kumar, K. Singh, An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems, IEEE Trans. Parallel Distrib. Syst. 14 (2003) 533–544.

[6] S. Baskiyar, C. Dickinson, Scheduling directed a-cyclic task graphs on a bounded set of heterogeneous processors using task duplication, J. Parallel Distrib. Comput. 65 (2005) 911–921.
[7] W.F. Boyer, G.S. Hura, Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments, J. Parallel Distrib. Comput. 65 (2005) 1035–1046.
[8] Y.C. Chung, S. Ranka, Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors, in: Proc. Supercomputing'92, Minneapolis, MN, 1992, pp. 512–521.
[9] E.G. Coffman, Computer and Jop-Shop Scheduling Theory, John Wiley & Sons Inc, New York, NY, 1976.
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[11] H. Dail, F. Berman, H. Casanova, A decoupled scheduling approach for grid application development environments, J. Parallel Distrib. Comput. 63 (2003) 505–524.
[12] M.I. Daoud, N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 68 (2008) 399–409.
[13] M.I. Daoud, N. Kharma, An efficient genetic algorithm for task scheduling in heterogeneous distributed computing systems, in: Proc. 2006 IEEE Congress on Evolutionary Computation, Vancouver, BC, Canada, 2006, pp. 3258–3265.
[14] A.E. Eiben, Z. Michalewicz, M. Schoenauer, J.E. Smith, Parameter control in evolutionary algorithms, Stud.Comput. Intell. 54 (2007) 19–46.
[15] H. El-Rewini, T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, J. Parallel Distrib. Comput. 9 (1990) 138–153.
[16] H. El-Rewini, T.G. Lewis, H.H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice Hall, New Jersey, NJ, 1994.
[17] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co, New York, NY, 1979.
[18] D.E. Goldberg, Genetic Algorithm in Search, Optimization and Machine Learning, Addison-Wesley, Boston, MA, 1989.
[19] M. Grajcar, Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system, in: Proc. 36th ACM/IEEE Conference on Design automation, New Orleans, LA, 1999, pp. 280–285.
[20] M. Grajcar, Strengths and weaknesses of genetic list scheduling for heterogeneous systems, in: Proc. 2nd International Conference on Application of Concurrency to System Design ACSD'01, Newcastle upon Tyne, UK, 2001, pp. 123–132.
[21] J. Grefenstette, Rank-based selection, in: T. Back, D.B. Fogel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, first ed., Oxford Univ. Press, Oxford, UK, 1997, pp. C2.4.1–C2.4.6.
[22] J.J. Grefenstette, K.A. De Jong, W.M. Spears, Competition-based learning, in: A.L. Meyrowitz, S. Chipman (Eds.), Foundations of Knowledge Acquisition: Machine Learning, Kluwer Academic Publishers, Norwell, MA, 1993, pp. 203–226.
[23] B. Hamidzadeh, L.Y. Kit, D.J. Lilja, Dynamic task scheduling using online optimization, IEEE Trans. Parallel Distrib. Syst. 11 (2000) 1151–1163.
[24] E. Ilavarasan, P. Thambidurai, R. Mahilmannan, Performance effective task scheduling algorithm for heterogeneous computing system, in: Proc. 4th International Symposium on Parallel and Distributed Computing, France, 2005, pp. 28–38.
[25] A. Iosup, C. Dumitrescu, D. Epema, H. Li, L. Wolters, How are real grids used? The analysis of four grid traces and its implications, in: Proc. 7th IEEE/ACM International Conference on Grid Computing, Spain, 2006, pp. 262–269.
[26] M. Iverson, F. Ozguner, G. Follen, Parallelizing existing applications in a distributed heterogeneous environment, in: Proc. 4th Heterogeneous Computing Workshop, Santa Barbara, CA, 1995, pp. 93–100.
[27] M.A. Iverson, F. Ozguner, L. Potter, Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment, IEEE Trans. Comput. 48 (1999) 1374–1379.
[28] S.J. Kim, J.C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proc. International Conference on Parallel Processing, Pennsylvania State University, University Park, PA, 1988, pp. 1–8.
[29] J. Kim, J. Rho, J.-O. Lee, M.-C. Ko, CPOC: Effective static task scheduling for grid computing, in: Proc. 2005 International Conference on High Performance Computing and Communications, Italy, 2005, pp. 477–486.
[30] B. Kuatrachue, T.G. Lewis, Grain size determination for parallel processing, IEEE Softw. 5 (1988) 23–32.
[31] Y.K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (1999) 406–471.
[32] Y.K. Kwok, I. Ahmad, Dynamic critical-path Scheduling: An effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (1996) 506–521.
[33] J. Liou, M.A Palis, A comparison of genetic approaches to multiprocessor scheduling, in: Proc. 11th International Parallel Processing Symp., Geneva, Switzerland, 1997, pp. 152–156.
[34] S. Nesmachnow, H. Cancela, E. Alba, Heterogeneous computing scheduling with evolutionary algorithms, Soft Computing-A Fusion of Foundations, Methodologies and Applications (2010); Available from: http://dx.doi.org/10.1007/s00500-010-0594-y.
[35] M. O'Neill, C. Ryan, Genetic code degeneracy: Implications for grammatical evolution and beyond, in: Proc. 5th European Conference on Artificial Life, Lausanne, Switzerland, 1999, pp. 149–143.
[36] P. Phinjaroenphan, S. Bevinakoppa, P. Zeephongsekul, A method for estimating the execution time of a parallel task on a grid node, in: Lecture Notes in Computer Science 3470, European Grid Conference on Advances in Grid Computing, Amsterdam, Netherlands, 2005, pp. 226–236.
[37] A. Radulescu, A.J.C. van Gemund, Low-cost task scheduling for distributed-memory machines, IEEE Trans. Parallel Distrib. Syst. 13 (2002) 648–658.
[38] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, MIT Press, Cambridge, MA, 1989.
[39] P. Siarry, A. Petrowski, M. Bessaou, A multipopulation genetic algorithm aimed at multimodal optimization, in: A.L. Meyrowitz, S. Chipman (Eds.), Advances in Engineering Software, 33, 2002, pp. 207–213.
[40] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (1993) 175–187.
[41] M. Srinivas, L.M. Patnaik, Genetic algorithms: A survey, Computer 27 (1994) 17–26.
[42] H. Topcuoglu, S. Hariri, M.Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (2002) 260–274.
[43] R.F. Weaver, Molecular Biology, fourth ed., McGraw-Hill Higher Education, Boston, MA, 2008.
[44] M. Wu, D. Dajski, Hypertool: A programming aid for message passing systems, IEEE Trans. Parallel Distrib. Syst. 1 (1990) 330–343.
[45] T. Yang, A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distrib. Syst. 5 (1994) 951–967.
[46] W. Zhang, B. Fang, H. He, H. Zhang, M. Hu, Multisite resource selection and scheduling algorithm on computational grid, in: Proc. 18th International Parallel and Distributed Processing Symp., Los Alamitos, CA, 2004, p. 105.
[47] A.Y. Zomaya, Y.H. Teh, Observations on using genetic algorithms for dynamic load balancing, IEEE Trans. Parallel Distrib. Syst. 12 (2001) 899–911.
[48] A. Zomaya, C. Ward, B. Macey, Genetic scheduling for parallel processor systems: Comparative studies and performance issues, IEEE Trans. Parallel Distrib. Syst. 10 (1999) 795–812.

Mohammad I. Daoud received an M.A.Sc. degree in electrical and computer engineering from Concordia University, Montreal, Quebec, Canada, in 2005, and a Ph.D. degree in electrical and computer engineering from the University of Western Ontario, London, Ontario, Canada, in 2009. He joined the Department of Electrical and Computer Engineering at the University of British Columbia, Vancouver, British Columbia, Canada, as a Postdoctoral Research Fellow in March 2010. His research interests include parallel processing using computer clusters and computational modeling of ultrasound imaging.

Nawwaf Kharma is a graduate of City University and Imperial College, London. His Ph.D. is in Machine Learning, and his research has been in practical applications of Genetic Algorithms with special emphasis on Pattern Recognition & Image Processing. He has authored or co-authored several books and book chapters and numerous journal and conference papers. Dr. Kharma is currently an Associate Professor at the Electrical & Computer Engineering department of Concordia University, Montreal, with an expanding interest in Synthetic Biology for computation.