Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 68 (2008) 399–409
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Research Note

A high performance algorithm for static task scheduling in heterogeneous distributed computing systems

Mohammad I. Daoud, Nawwaf Kharma*

Electrical and Computer Engineering, Concordia University, 1455 de Maisonneuve, West, S-H-961 Montreal, Que., Canada H3G 1M8

Received 19 September 2006; received in revised form 30 April 2007; accepted 9 May 2007
Available online 28 July 2007

Abstract

Effective task scheduling is essential for obtaining high performance in heterogeneous distributed computing systems (HeDCSs). However, finding an effective task schedule in HeDCSs requires the consideration of both the heterogeneity of processors and high interprocessor communication overhead, which results from non-trivial data movement between tasks scheduled on different processors. In this paper, we present a new high-performance scheduling algorithm, called the longest dynamic critical path (LDCP) algorithm, for HeDCSs with a bounded number of processors. The LDCP algorithm is a list-based scheduling algorithm that uses a new attribute to efficiently select tasks for scheduling in HeDCSs. The efficient selection of tasks enables the LDCP algorithm to generate high-quality task schedules in a heterogeneous computing environment. The performance of the LDCP algorithm is compared to two of the best existing scheduling algorithms for HeDCSs: the HEFT and DLS algorithms. The comparison study shows that the LDCP algorithm outperforms the HEFT and DLS algorithms in terms of schedule length and speedup. Moreover, the improvement in performance obtained by the LDCP algorithm over the HEFT and DLS algorithms increases as the inter-task communication cost increases. Therefore, the LDCP algorithm provides a practical solution for scheduling parallel applications with high communication costs in HeDCSs.

Keywords: Task scheduling; Directed acyclic graph; Heuristics; Parallel processing; Heterogeneous systems

1. Introduction

A distributed computing system, or DCS, is a group of processors connected *via* a high speed network that supports the execution of parallel applications. The efficiency of executing parallel applications on DCSs critically depends on the method used to schedule the tasks of the parallel application onto the available processors. In DCSs, interprocessor communication is an unavoidable overhead of the execution of parallel programs. This overhead occurs when tasks allocated to different processors exchange data. The creation of high quality task schedules becomes more critical when the parallel application is executed on DCSs with heterogeneous processors, or HeDCSs (heterogeneous distributed computing systems). In addition to the tradeoff between the speedup gained through parallelization and the overhead of interprocessor communication, scheduling algorithms for HeDCSs have to consider the various execution times of the same task on different processors. A faulty scheduling decision in HeDCSs may limit the performance of the system by the capabilities of the slowest processors.

In general, task scheduling algorithms for DCSs are classified into two classes: *static* and *dynamic*. 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 [18]. Static task scheduling takes place during compile time before running the parallel application. In contrast, scheduling decisions in dynamic scheduling algorithms are made at *run time*. The objective of dynamic scheduling algorithms includes not only creating high quality task schedules, but also minimizing the run time scheduling overheads [4,6,8,9,11,14,20]. In this paper, static scheduling is addressed, as it allows the use of sophisticated scheduling algorithms to create high quality task schedules without introducing run time scheduling overheads.

* Corresponding author. Fax: +1 514 848 2802.
E-mail address: kharma@ece.concordia.ca (N. Kharma).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.05.015

Static task scheduling for DCSs, in general, is shown to be an NP-complete problem [9,13,16,17], and many static scheduling algorithms based on heuristics are proposed in the literature [1–4,7,9–11,13,15–19]. One important class of scheduling heuristics is *list-based* algorithms [14]. In list-based scheduling algorithms, each task is assigned a given priority. Three steps are then repeated until all tasks of the parallel application are scheduled: task selection, processor selection and status update. The highest-priority unscheduled task is 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. Finally, the status of the system is updated in the status update phase. At the end of this process, a valid schedule is obtained [1,11,13,16–18]. Examples of list-based algorithms are: heterogeneous earliest finish time (HEFT) [17], critical path on a processor (CPOP) [17], critical path on a cluster (CPOC) [11], dynamic level scheduling (DLS) [16], modified critical path (MCP) [18], mapping heuristic (MH) [15] and dynamic critical path (DCP) [13].

Many parallel applications have long execution times and hence, they require high quality task schedules to minimize their run times. Moreover, in typical scientific and engineering applications, compile time, including the static scheduling time, is much lower than run time. Hence, increasing scheduling complexity to create high quality task schedules, which reduce the run time of parallel applications, will improve the overall performance of DCSs.

HeDCSs, such as heterogeneous clusters, are in common use. To obtain high-performance in HeDCSs, efficient scheduling algorithms that are developed specifically for HeDCSs must be used to schedule the tasks of parallel applications. There are several algorithms for task scheduling on HeDCSs, such as: HEFT, CPOP, DLS, MH, levelized min time (LMT) [10] and heterogeneous N-predecessor duplication (HNPD) [5]. Topcuoglu et al. [17] presented a performance comparison study of the HEFT, CPOP, DLS, MH and LMT algorithms for different values of DAG size, communication to computation cost ratio (CCR) and parallelism factor (the DAG size, CCR, and parallelism factor will be defined in Section 5). In their study, the performance of the HEFT algorithm outperforms the CPOP, DLS, MH and LMT algorithms. Moreover, the performance of the DLS algorithm outperforms the MH and LMT algorithms. The CPOP algorithm and the DLS algorithm achieved comparable results. The performance of the HEFT and HNPD algorithms is compared in [5], where the latter combines both list-based scheduling and multiple task duplication. When the number of processors is equal to one-forth the number of tasks, the HEFT algorithm outperforms the HNPD algorithm for CCR values less than or equal to one, while the HNPD algorithm outperforms the HEFT algorithm for CCR values greater than 1. On the other hand, for unlimited number of processors the HNPD algorithm outperforms the HEFT algorithm. Since the HNPD algorithm employs multiple task duplication, the HNPD algorithm requires a greater number of processors than the HEFT algorithm to achieve the same schedule length.

In this paper, a new list-based algorithm, called the *longest dynamic critical path* (LDCP) algorithm, for static task scheduling in HeDCSs with limited numbers of processors is presented. The motivation behind this algorithm is to generate the high-quality task schedules that are necessary to achieve high performance in HeDCSs with limited numbers of processors. The remainder of this paper is organized as follows: in Section 2, we define the research problem and some necessary terms. Section 3 introduces the problem of assigning task priorities in HeDCSs along with a new attribute to effectively address this problem. Section 4 introduces the LDCP algorithm. Simulation results are presented in Section 5. Finally, a conclusion and an overview of future work are given in Section 6.

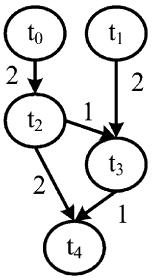
2. Problem definition

In static task scheduling for HeDCSs, the 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 task $t_i \in T$ represents a task in the parallel application, and each edge $(t_i, t_j) \in E$ represents a precedence constraint and a communication message between tasks t_i and t_j. If $(t_i, t_j) \in E$, then the execution of $t_j \in T$ cannot be started before $t_i \in T$ finishes its execution. The source task t_i of an edge (t_i, t_j) is a *parent* of the sink task t_j, while 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*. Associated with each edge (t_i, t_j) is a value $d_{i,j}$ that represents the amount of data to be transmitted from task t_i to task t_j [2,9,13,17].

The HeDCS is represented by a set P of m processors that have diverse capabilities. The $n \times m$ *computation cost matrix* W stores the execution costs of tasks. Each element $w_{i,j} \in W$ represents the estimated execution time of task t_i on processor p_j. All processors are assumed to be fully connected. Communications between processors occur *via* independent communication units; this allows for concurrent execution of computation tasks and communications between processors [2,17]. 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.

The communication cost between two processors p_k and p_l depends on the network initialization at processors p_k and p_l in addition to the communication time on the network. The time required to initialize the network at the sender and receiver processors is considered to be ignorable compared to the communication time on the network [2,9]. The data transfer rate between any two processors on the network is assumed to be fixed and constant [2,9]. Therefore, the communication cost of an edge (t_i, t_j) is equal to the amount of data transmitted from task t_i to task t_j, or $d_{i,j}$, divided by the data transfer rate of the network. Without loss of generality, the data transfer rate of interprocessor network is assumed to be unity [9]. Hence, the communication cost of an edge (t_i, t_j) is equal to $d_{i,j}$ given that tasks t_i and t_j are scheduled on different processors. Since the data transfer rate of the intra-processor bus is much higher

a



b

Task	p_0	p_1
t_0	5	8
t_1	11	15
t_2	4	7
t_3	6	7
t_4	1	2

Fig. 1. An example of a DAG and a computation cost matrix.

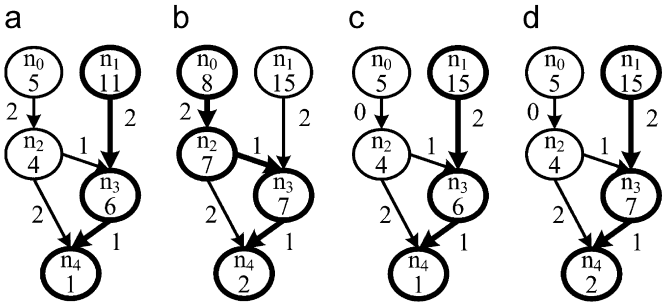


Fig. 2. The DAG in Fig. 1 constructed ((a) and (b)) before scheduling any task and using the computation costs of tasks on processors (a) p_0 and (b) p_1, and ((c) and (d)) after scheduling tasks t_0 and t_2 on p_0 and task t_1 on p_1 and setting the computation costs of the unscheduled tasks to their values on processors (c) p_0 and (d) p_1.

than the data transfer rate of the interprocessor network, the communication cost between two tasks scheduled on the same processor is taken as zero. A task can start execution on a processor only when all data from its parents become available to that processor; at that time the task is marked as *ready*. Tasks must be scheduled and assigned to processors in a way that minimizes the total run time, or the *schedule length*, of the parallel application [2,9,13,17]. An example of a DAG of a parallel application and a computation cost matrix of a HeDCS with two processors is shown in Fig. 1.

3. Task priorities in HeDCSs

The performance of list-based scheduling algorithms depends highly on the method used to assign priorities to tasks. A task must be assigned a high priority if the selection of this task for scheduling during the current step ultimately leads to a shorter schedule length.

For 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 for which the sum of the computation costs of tasks and the communication costs of edges is maximal. 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. [13] used an efficient attribute, called the DCP attribute, to effectively select tasks for scheduling in homogeneous computing systems. 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 same 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, or their mean values, as in the HEFT algorithm, 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 LDCP. The LDCP is explained in Definition 1.

Definition 1. Given a DAG with n tasks and e edges, and a HeDCS with m heterogeneous processors, the **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 scheduling, the DCP computed using the computation costs of tasks on processor p_0 is composed of tasks t_1, t_3, t_4, and has a length of 21, as shown in Fig. 2a. However, the DCP computed using the computation costs of tasks on processor p_1 is composed of tasks t_0, t_2, t_3, t_4, and has a length of 28, as shown in Fig. 2b. Hence, at the start of scheduling the LDCP is composed of tasks t_0, t_2, t_3, t_4, and has a length of 28.

If tasks t_0 and t_2 are scheduled on processor p_0 and task t_1 is scheduled on processor p_1, then the longest path, computed by setting the computation costs of unscheduled tasks to their values on processor p_0, will be composed of tasks t_1, t_3 and t_4, and will have a length of 25, as shown in Fig. 2c. However, the longest path computed by using the computation costs of the unscheduled tasks on processor p_1 will also be composed of tasks t_1, t_3 and t_4, but will have a length of 27, as shown in Fig. 2d. Hence, the LDCP will be composed from tasks t_1, t_3, t_4 and will have a length of 27. It is worth noting that in these calculations the communication cost between tasks t_0 and t_2 is set to 0. It should also be noted that during a particular scheduling step, the LDCP is not unique, and as such it is possible to have two or more paths of tasks and edges from

entry tasks to exit tasks that satisfy Definition 1. However, all LDCPs must have the same length.

4. The LDCP algorithm

In the LDCP algorithm, each scheduling step consists of three phases: task selection, processor selection and status update.

4.1. Task selection phase

The LDCPs identify a set of tasks that play an important role in determining the *provisional* schedule length. To compute the LDCPs 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 the DAGP. The upward rank is defined 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 nodes that have the highest URank value over all DAGPs identify the entry tasks of all LDCPs.*

Proof. If node n_i, which is located on DAGP$_j$, has the highest URank value over all DAGPs, then the length of any LDCP is equal to the URank value of node n_i. Hence, node n_i identifies the entry task of at least one of the LDCPs. Moreover, each LDCP must have an entry task with URank value equal to the highest URank value over all DAGPs. □

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

Proof. If node n_i on DAGP$_j$ identifies task t_i on a LDCP and its URAS is node n_k, then the URank value of node n_k is equal to the length of the portion of that LDCP that extends between the exit task of that LDCP and the task located immediately after task t_i on that LDCP. Hence, n_k identifies the next task after t_i on that LDCP. □

The LDCP algorithm chooses one of the LDCPs, which is called the *selected LDCP*, as follows. The entry task of the selected LDCP is determined by locating a node n_i that has the highest URank value over all nodes on all DAGPs. If there are more than one node with the highest URank value, then ties are broken by selecting the node with the highest number of output edges first; if more than one node exists, the tie is broken on a random basis. This tie-breaking strategy guarantees a higher priority for tasks with greater numbers of children, without introducing a significant increase in the time complexity of the LDCP algorithm. The remaining tasks on the selected 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 selected LDCP are located using Theorem 2, and ties are broken in the same way used to select the entry task of the selected LDCP.

Definition 5. During a particular scheduling step, let the set of nodes N be used to identify the tasks on the selected 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 predecessors of the key node with the highest URank value are defined as the **parent key nodes**.

At each scheduling step, if the key node does not have any unscheduled parent, then the key node is used to identify the

```
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 execution constraints on all DAGPs
  update the temporary zero-cost edges on the DAGP associated with the
    selected processor
  update the URank values of the nodes that identify the scheduled tasks on
    all DAGPs
end while
```

Fig. 3. The LDCP algorithm.

task that will be selected for scheduling. Otherwise, the parent key node with the highest number of output edges is used to identify the selected task. If more than one parent key node has the highest number of output edges, then the tie is broken on a random basis.

4.2. Processor selection phase

In this phase, the selected task is assigned to a processor that minimizes its finish execution time using the *insertion-based scheduling* policy [17]. 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. 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 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.

4.3. Status update phase

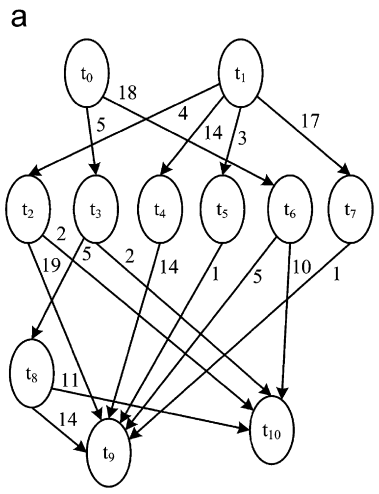
When a task is scheduled on a processor, the status of the system must be updated to reflect the new changes. The scheduling of task t_i on processor p_j means that the computation cost of t_i is no longer unknown. Hence, the sizes of the nodes that identify t_i are set to the computation cost of t_i 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. The insertion of task t_i into processor p_j will result in new execution constraints. These execution constraints are shown on all DAGPs by adding a zero-cost edge from the node that identifies t_i to the node that identifies the task scheduled after t_i on p_j (if any), and another zero-cost edge from the node that identifies the task scheduled before t_i on p_j (if any) to the node that identifies t_i. Moreover, the execution constraints between the nodes that identify the tasks scheduled right before and right after task t_i on processor p_j are removed from all DAGPs.

To enable the LDCPs to include new ready tasks as the scheduling process advances, each time a new task is allocated to processor p_j the LDCP algorithm adds temporary zero-cost edges into $DAGP_j$ from the node that identifies the last task scheduled on p_j (task t_k) to all the ready nodes that do not communicate with task t_k. This must be done after removing the pervious temporary zero-cost edges from $DAGP_j$. 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.

4.4. The proposed LDCP algorithm

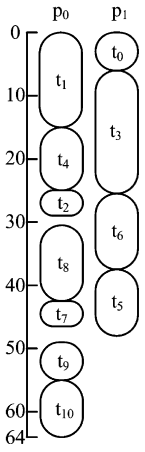
The proposed LDCP algorithm is formalized in Fig. 3. The LDCP algorithm has a time complexity of $O(m \times n^3)$ where m is the number of processors, and n is the number of tasks. In comparison, the time complexity of the DLS and HEFT algorithms is $O(m \times n^3)$ and $O(m \times e)$, respectively, where e is the number of edges. For dense DAGs in which the number of edges is proportional to n^2, the time complexity of the HEFT algorithm is $O(m \times n^2)$.

As an illustration, consider the application DAG and the computation cost matrix shown in Fig. 4a and b. The schedule generated by the LDCP algorithm along with a stepwise trace of the LDCP algorithm are shown in Fig. 4c and d, 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 stepwise trace of the DLS and HEFT algorithms are shown in Fig. 4e and f, respectively.



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



d

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

e

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

f

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

Fig. 4. (a) and (b) A sample DAG and computation cost matrix, (c) the schedule generated by the LDCP algorithm, ((d), (e) and (f)) stepwise trace of the (d) LDCP, (e) DLS and (f) HEFT algorithms.

5. Results and analysis

In this section, we compare the performance of the LDCP algorithm to two scheduling algorithms for HeDCSs with limited numbers of processors: the HEFT and DLS algorithms. An explicit comparison with some other well-known scheduling algorithms for HeDCSs, such as CPOP, MH and LMT, is not carried out as the HEFT and DLS algorithms have already been tested against them, and have given better or at worst very similar results [17].

To test the performance of the scheduling algorithms, a simulation environment for computer clusters is built and run on an IBM Pentium IV computer. The LDCP algorithm as well as the HEFT and DLS algorithms are implemented. Two sets of parallel application graphs, which correspond to both random application DAGs and DAGs of parallel numerical applications, are created. The scheduling algorithms are run on the application graphs to generate output schedules. Finally, a group of performance metrics is applied to the schedules generated by the three scheduling algorithms.

The performance metrics chosen for the comparison are the normalized schedule length (NSL) and speedup [3,17]. The NSL of a given schedule is defined as the schedule length divided by the lowest possible value of the schedule length. It is calculated using:

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

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.

The speedup of a schedule is defined as the ratio of the schedule length obtained by assigning all task to the fastest processor, to the parallel execution time of the task schedule.

5.1. Performance results on random graphs

A set of randomly generated graphs is created by varying a set of parameters that determines the characteristics of the

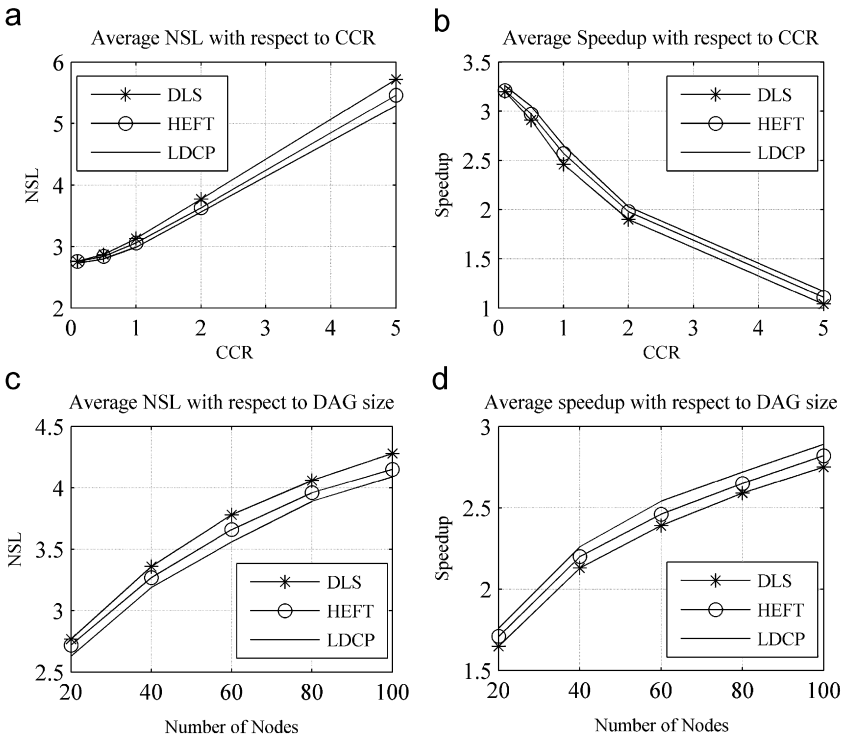


Fig. 5. Performance results on random graphs.

generated DAGs. These 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.
- *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 [17]. A low α value leads to a DAG with a low parallelism degree [3].
- *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 *vice 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 does not affect the performance results of the scheduling algorithms. 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 $\left[\overline{w_i} \times \left(1 - \frac{h}{2}\right), \overline{w_i} \times \left(1 + \frac{h}{2}\right)\right]$. 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 [17].

The 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. The large set of random graphs, which consists of 2000 DAGs with diverse characteristics, prevents bias towards 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 [17].

The NSLs produced by the LDCP, HEFT and DLS algorithms for the various *CCR* values are shown in Fig. 5a. The average NSL value of 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. The first value of each parenthesized pair is the improvement achieved by the LDCP algorithm over the DLS algorithm, while the second value is the improvement of the LDCP 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 three algorithms with respect to certain *CCR* values are shown in Fig. 5b. The average speedup value of the LDCP algorithm is higher than those returned by the DLS and HEFT algorithms by: (1.9%, 1.4%), (4.7%, 2.6%),

Table 1
A global comparison of the scheduling algorithms

		LDCP	HEFT	DLS
LDCP	Better		1612 (80.6%)	1678 (83.9%)
	Equal	–	178 (8.9%)	89 (4.5%)
	Worse		210 (10.5%)	233 (11.6%)
HEFT	Better	210 (10.5%)		1466 (73.3%)
	Equal	178 (8.9%)	–	156 (7.8%)
	Worse	1612 (80.6%)		378 (18.9%)
DLS	Better	233 (11.6%)	378 (18.9%)	
	Equal	89 (4.5%)	156 (7.8%)	–
	Worse	1678 (83.9%)	1466 (73.3%)	

(7.0%, 2.5%), (8.2%, 4.1%) and (12.3%, 5.0%), when the *CCR* is equal to: 0.1, 0.5, 1.0, 2.0 and 5.0, respectively.

In these experiments, the LDCP algorithm outperforms the DLS and HEFT algorithms for all tested *CCR* values in terms of both NSL and speedup. As the value of *CCR* increases, interprocessor communication overhead dominates computation and hence, the performance of all three scheduling algorithms tends to degrade. However, as shown in Fig. 5a and b, 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 is progressively dominated by tasks with high interprocessor communication overheads. 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 execution finishing 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.

The average NSL and speedup values gained by the three algorithms with respect to DAG size are shown in Fig. 5c and d, respectively. The average NSL value of the LDCP algorithm 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. The average speedup gained by the LDCP algorithm is greater than those of 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. Hence, the LDCP algorithm 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.

The number of times each scheduling algorithm produced better, equal or worse schedules, compared to each of the other two algorithms for the 2000 randomly generate DAGs, is shown in Table 1. Each cell in Table 1 compares the schedule length generated by the algorithm in the leftmost column to the algorithm in the top row. The percentage in the parentheses is calculated by dividing the number on the left by the total number of DAGs. As shown in Table 1, the LDCP algorithm has superior performance compared to the DLS and HEFT algorithms based on occurrences of better results.

5.2. 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: the Gaussian elimination algorithm [17,18], the fast Fourier transform algorithm [7] and a molecular dynamics code given in [12,17]. 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 5.1.

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 $\frac{N^2+N-2}{2}$ [17]. For the experiments of the Gaussian elimination algorithm, the size of the input matrix (N) is used in place of the DAG size (n). For the NSL comparison, the matrix size used in the experiments is varied from 5 to 20, with an increment of 1, and the number of processors is set to 5. The average NSLs produced by each scheduling algorithm in relation to *CCR* are shown in Fig. 6a. The average NSL value of the LDCP algorithm 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%). In this subsection, all the results are presented with respect to the value of *CCR*. Hence, the parenthesized pairs are arranged to present the results achieved with *CCR* values of 0.1, 0.5, 1.0, 2.0 and 5.0, respectively. The average 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

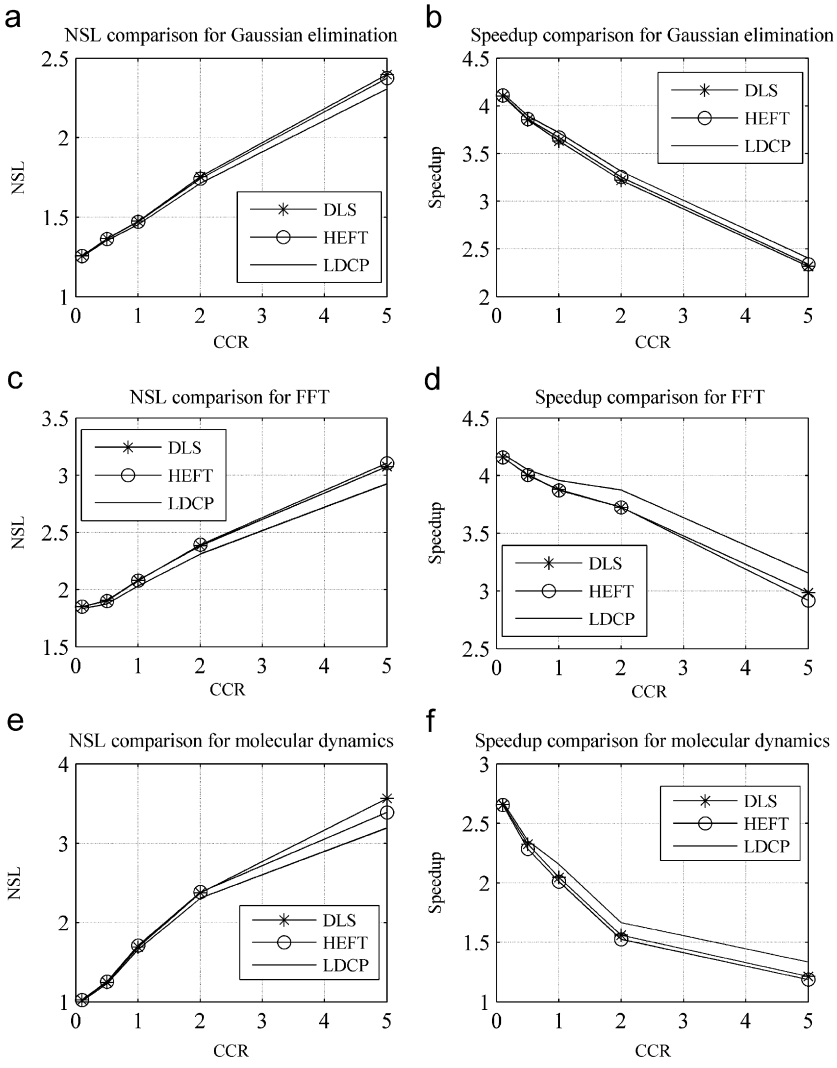


Fig. 6. Performance results on regular graphs.

set to 20 are shown in Fig. 6b. The LDCP algorithm 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 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 the Gaussian elimination algorithm experiments, the size of the input vector (M) is used in place of the DAG size. To study the NSL values of the three 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. The average NLS values of the scheduling algorithms with respect to *CCR* are shown in Fig. 6c. The average NSL obtained by the LDCP algorithm 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 speedup values obtained by the three 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, are shown in Fig. 6d. The LDCP algorithm gained 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%).

Finally, the performance of the three scheduling algorithms is compared to each other with respect to the application DAG of the molecular dynamics code given in [12,17]. Since the number of tasks (41 tasks) and the graph structure are known, only the *CCR* and h values are used in this experiment. The average NSL values of the scheduling algorithms with respect to *CCR* when the number of processors is set to 5 are shown in Fig. 6e. On average, the LDCP algorithm 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 speedup values of the scheduling algorithms with respect to *CCR* are presented in Fig. 6f. Since the maximum

number of tasks in any level of the molecular dynamics code DAG is less than 7, the number of processors used is varied from 2 to 7 with an increment of 1 [17]. The average speedup value gained by the LDCP algorithm 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%).

For real world applications, the LDCP algorithm outperforms the DLS and HEFT algorithms in terms of schedule length and speedup. The general trend of increasing improvement in performance obtained by the LDCP 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. Conclusion and future work

We present a new list-based scheduling algorithm, called the longest dynamic critical path (LDCP) algorithm, for the problem of scheduling in heterogeneous distributed computing systems (HeDCSs). The LDCP algorithm uses a new attribute, called LDCP, to accurately identify the priorities of tasks in HeDCSs. The LDCP attribute is an intuitive extension to the widely used concept of critical path (CP), meant to accommodate the heterogeneous nature of HeDCSs. The LDCP attribute is designed to reflect the fact that a single DAG may have more than one CP, if scheduled on more than one non-identical processor. In computing LDCP, the communication overhead between tasks scheduled on the same processor is neglected.

The performance of the LDCP algorithm is compared to two of the best existing scheduling algorithms for HeDCSs: the HEFT [17] and DLS [16] algorithms. The comparative study is based on both randomly generated application DAGs and DAGs that correspond to three real-world numerical applications. The LDCP algorithm significantly outperforms both the HEFT and DLS algorithms in terms of normalized schedule length (NSL) and speedup. The improvement in performance achieved by the LDCP algorithm over the DLS and HEFT algorithms tends to increase as *CCR* increases. For a *CCR* value of 5, the LDCP algorithm achieved NSLs that are 3.7–10.4% shorter that the DLS algorithm, and 2.8–5.8% shorter than the HEFT algorithm. Moreover, the speedup achieved by the LDCP algorithm, when *CCR* is equal to 5, is 3.7–12.3% higher than the DLS algorithm, and 2.8–12.5% higher than the HEFT algorithm. Based on its superior performance at high *CCR* values, the LDCP algorithm appears to be a practical solution for task scheduling on HeDCSs for applications with high communication costs.

We plan to extend the LDCP algorithm to partially connected networks of heterogeneous processors. The extended LDCP algorithm will be tested on HeDCSs with larger numbers of processors and arbitrary interprocessor communication networks. The results reported in this paper suggest the use of the LDCP attribute with other scheduling optimization techniques, such as task duplication, to develop better scheduling algorithms for HeDCSs.

References

[1] I. Ahmad, Y.K. Kwok, On exploiting task duplication in parallel program scheduling, IEEE Trans. Parallel Distributed Systems 9 (9) (1998) 872–892.
[2] R. Bajaj, D.P. Agrawal, Improving scheduling of tasks in a heterogeneous environment, IEEE Trans. Parallel Distributed Systems 15 (2) (2004) 107–118.
[3] S. Bansal, P. Kumar, K. Singh, An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems, IEEE Trans. Parallel Distributed Systems 14 (6) (2003) 533–544.
[4] S. Bansal, P. Kumar, K. Singh, Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs, J. Parallel Distributed Comput. 65 (4) (2005) 479–491.
[5] S. Baskiyar, C. Dickinson, Scheduling directed a-cyclic task graphs on a bounded set of heterogeneous processors using task duplication, J. Parallel Distributed Comput. 65 (8) (2005) 911–921.
[6] W.F. Boyer, G.S. Hura, Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments, J. Parallel Distributed Comput. 65 (9) (2005) 1035–1046.
[7] Y.C. Chung, S. Ranka, Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors, in: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, Minneapolis, MN, USA, 1992, pp. 512–521.
[8] B. Hamidzadeh, L.Y. Kit, D.J. Lilja, Dynamic task scheduling using online optimization, IEEE Trans. Parallel Distributed Systems 11 (11) (2000) 1151–1163.
[9] E. Ilavarasan, P. Thambidurai, R. Mahilmannan, Performance effective task scheduling algorithm for heterogeneous computing system, in: Proceedings of the Fourth International Symposium on Parallel and Distributed Computing, France, 2005, pp. 28–38.
[10] M. Iverson, F. Ozguner, G. Follen, Parallelizing existing applications in a distributed heterogeneous environment, in: Proceedings of the Fourth Heterogeneous Computing Workshop, 1995, pp. 93–100.
[11] J. Kim, J. Rho, J.-O. Lee, M.-C. Ko, CPOC: effective static task scheduling for grid computing, in: Proceedings of the 2005 International Conference on High Performance Computing and Communications, Italy, 2005, pp. 477–486.
[12] S.J. Kim, J.C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proceedings of the International Conference on Parallel Processing, Pennsylvania State University, University Park, PA, USA, 1988, pp. 1–8.
[13] Y.K. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distributed Systems 7 (5) (1996) 506–521.
[14] Y.K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surveys 31 (4) (1999) 406–471.
[15] H. El-Rewini, T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, J. Parallel Distributed Comput. 9 (2) (1990) 138–153.
[16] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distributed Systems 4 (2) (1993) 175–187.
[17] H. Topcuoglu, S. Hariri, M.Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distributed Systems 13 (3) (2002) 260–274.
[18] M. Wu, D. Dajski, Hypertool: a programming aid for message passing systems, IEEE Trans. Parallel Distributed Systems 1 (3) (1990) 330–343.
[19] T. Yang, A. Gerasoulis, DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distributed Systems 5 (9) (1994) 951–967.
[20] A. Zomaya, C. Ward, B. Macey, Genetic scheduling for parallel processor systems: comparative studies and performance issues, IEEE Trans. Parallel Distributed Systems 10 (8) (1999) 795–812.

Mohammad I. Daoud is a PhD candidate in the Department of Electrical and Computer Engineering at University of Western Ontario, London, Canada. He received his Bachelor of Engineering degree from An-Najah National University, Nablus, Palestine in 2001 and Master of Applied Science degree from Concordia University, Montreal, Canada in 2005. His research interests include parallel processing, numerical analysis of 3-D ultrasonic imaging, and evolutionary computation.

Nawwaf Kharma is an Associate Professor with the ECE Department of Concordia University, Montreal, Canada. He specializes in the extension and application of Evolutionary Computation methodologies to real-world problems, such as Pattern Recognition and Process Scheduling. He is a member of ACM-SIGEVO.