Available online at www.sciencedirect.com
SCIENCE DIRECT
Future Generation Computer Systems 22 (2006) 665–675
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Scheduling workflow applications on processors with different capabilities

Zhiao Shi[a], Jack J. Dongarra[a,b,*]

[a] *Innovative Computing Lab, Department of Computer Science, University of Tennessee, Knoxville, TN 37996, USA*
[b] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*

Received 21 July 2005; received in revised form 10 November 2005; accepted 13 November 2005
Available online 9 January 2006

Abstract

Efficient scheduling of workflow applications represented by weighted directed acyclic graphs (DAG) on a set of heterogeneous processors is essential for achieving high performance. The optimization problem is NP-complete in general. A few heuristics for scheduling on heterogeneous systems have been proposed recently. However, few of them consider the case where processors have different capabilities. In this paper, we present a novel list scheduling based algorithm to deal with this situation. The algorithm (SDC) has two distinctive features. First, the algorithm takes into account the effect of Percentage of Capable Processors (PCP) when assigning the task node weights. For two task nodes with same average computation cost, our weight assignment policy tends to give higher weight to the task with small PCP. Secondly, during the processor selection phase, the algorithm adjusts the effective *Earliest Finish Time* strategy by incorporating the average communication cost between the current scheduling node and its children. Comparison study shows that our algorithm performs better than related work overall.

Keywords: Workflow; DAG; Task scheduling; List scheduling; Heterogeneous systems

1. Introduction

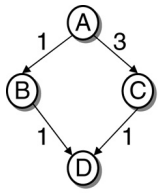
Workflow applications are generally described as collections of tasks that are processed in a well-defined order to accomplish a specific goal [1]. Typically, such applications are carried out by multiple processors connected either locally or in a distributed fashion. A heterogeneous distributed computing system comprises of diverse set of resources interconnected with a high-speed network, thereby supporting efficient executing of computationally intensive applications with different computing needs. The scheduling of workflow applications is highly critical to the performance of heterogeneous distributed computing systems. It deals with the allocation of individual tasks to suitable processors and assignment of the proper order of task execution on each resource. The objective is to minimize the overall completion time or *makespan* [2–5]. A popular representation of a workflow application is the Directed Acyclic Graph (DAG) in which the nodes represent individual application tasks and the directed arcs or edges represent inter-task data dependencies. As the DAG scheduling problem is *NP-complete* [6] in general, a number of heuristics have been proposed. In [7] the authors classified various heuristics into four categories:

- List scheduling algorithms [8–12]
- Clustering algorithms [8,13–15]
- Duplication based algorithms [16–19]
- Guided random search methods [20–25].

Compared to algorithms from the other three categories, list scheduling heuristics usually generate good quality schedules at a reasonable cost. The basic idea of list scheduling algorithms is to make a list (thus the name) of task nodes by first assigning each node some priorities. The list is then formed in decreasing order of priorities [2]. The order of the list admits precedence constraints. While the list is not empty, the algorithm repeatedly removes the first node from the list and allocates it to a processor which optimizes some predefined criteria. Various methods to specify the priorities of nodes and select the best processor have been proposed [2,7,9]. List scheduling heuristics are originally designed for homogeneous systems where processor speed and capability

* Corresponding author at: Innovative Computing Lab, Department of Computer Science, University of Tennessee, Knoxville, TN 37996, USA. Tel.: +1 865 974 8295.
E-mail addresses: shi@cs.utk.edu (Z. Shi), dongarra@cs.utk.edu (J.J. Dongarra).

0167-739X/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2005.11.002



(a)

Task	P1	P2
A	1	1
B	2	∞
C	3	1
D	1	1

(b)

	P1	P2
P1	∞	1
P2	1	∞

(c)

Fig. 1. (a) An example DAG, (b) the computation cost for each node on three machines, (c) the communication cost table.

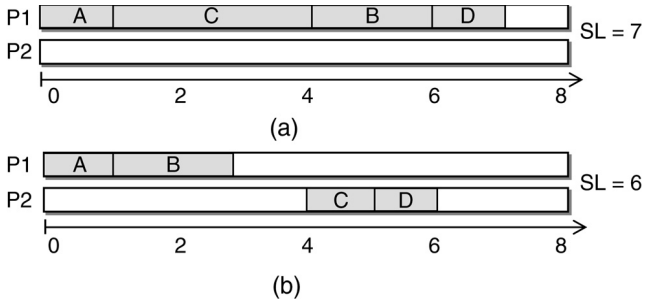


Fig. 2. (a) Schedule for the DAG in Fig. 1 with priority list A, C, B, D, (b) schedule for the DAG in Fig. 1 with priority list A, B, C, D.

and network bandwidth between any pair of processors are the same. It has been extended in two directions. Firstly, several *dynamic* list scheduling algorithms have been introduced [9,26,13]. These algorithms update the priorities of each node and the scheduling list dynamically at each step. Similar to traditional list scheduling algorithms, at each step, the node with highest priority is selected for scheduling. Dynamic list scheduling can potentially generate better schedules. However, these approaches can increase the time complexity of the algorithms significantly. Secondly, a number of new list scheduling algorithms for heterogeneous environment have been proposed [9,7,27]. A comparison of those algorithms reveals that during the processor selection phase: (1) insertion-based policy which allows the possible insertion of a task in an earliest idle time slot between two already-scheduled tasks on a processor is better than non-insertion based counterparts; (2) processor selection criteria that consider the different processor speeds (e.g. *Earliest Finish Time*) outperform those that don't include this factor (e.g. *Earliest Start Time*).

Although the DAG scheduling in general is a well studied problem, most of the algorithms assume that the processors are equally capable, i.e. each processor can execute *all* the tasks with possibly different speeds. While some of the algorithms don't make the assumption explicitly, they don't consider the potential effect of different capabilities either [7,11,12]. Thus, these algorithms suffer in performance when scheduling under this situation. Other algorithms simply become inapplicable without modification. For example, the Critical-Path-on-a-Processor (CPOP) algorithm introduced in [7] allocates all critical tasks onto a single processor in an attempt to minimize the total execution time of the critical tasks. This algorithm fails if none of the processors can process all the critical tasks. Another category of algorithms which becomes unsuitable is the clustering algorithms [8,13–15]. An algorithm of this type allocates tasks into different clusters. Each cluster can contain more than two tasks. When two tasks are assigned to the same cluster, they are executed in the same processor. Under the condition of processors with different capabilities, chances are none of the processors can carry out all the potentially large number of tasks in the same cluster. Therefore, unless effectively modified, clustering algorithms cannot be directly used under these circumstances.

In this paper, we propose a new static list Scheduling algorithm for heterogeneous processors with Different Capabilities (SDC). As found in [28], the methods followed to assign weights to the nodes significantly affect the performance of scheduling algorithms. We suggest a new approach of setting task node's weight. which considers the percentage of capable processor as well as the task's average execution cost among those capable processors. The SDC algorithm selects the task with the highest *b-level* [2] at each step. The selected task is then assigned to a processor which minimizes its *Adjusted Earliest Finish Time (AEFT)* (defined in Section 4) with an insertion-based policy. The AEFT adapts the EFT by including a new term which indicates how large the communication between current node and its children will be on the average provided that it is scheduled on the current processor. Due to resource scarcity, the processor that minimizes EFT for the current scheduling node is not necessarily the best choice because of potentially overwhelming inter-processor communication between the node and its children as shown by the example in Section 4. The algorithm has been tested on a large number of randomly generated problems of different sizes and types. The parametric graph generator is similar to the one designed in [7] but with a different set of parameters. We compare SDC with two other list scheduling algorithms, the Heterogeneous Earliest Finish Time (HEFT) [7] and Dynamic Level Scheduling (DLS) [9]. *Normalized Schedule Length* (NSL) and *Average Percentage Degradation* (APD) [3] are used as the comparison metrics in this paper. The comparison study shows that our algorithm performs considerably better in most cases, especially when the *Communication-to-Computation Ratio* (CCR) and *Percentage of Incapable Processor* (PIP) are large.

The remainder of this paper is organized as follows. In the next section, the scheduling problem and some related terminology are defined. In Section 3, we provide some related work in scheduling for heterogeneous computing systems. Our algorithm (SDC) is introduced in Section 4. Section 5 presents experimental results based on randomly generated task graphs and a real world bioinformatics workflow application graph. Section 6 contains the concluding remarks.

2. Problem description

A scheduling system usually consists of three parts: application, computing environment and scheduling goal. The application and computing environment can be represented by a task graph and resource graph respectively.

(1)Set the weights of task nodes with Eq. (9)
(2)Set the weights of edges with Eq. (2)
(3)Compute the *b-level*s for all tasks by traversing graph upward from the exit node.
(4)Sort the task into a list by non-increasing order of b-level
(5)***while*** the scheduling list is not empty ***do***
(6) Remove the first task v_i from the list for scheduling
(7) ***for*** each processor capable p_j of v_i ***do***
(8) Compute AEFT(n_i, p_j) value with Eq. (11) using insertion-based policy
(9) ***endfor***
(10) Assign task v_i to the processor that minimize AEFT of v_i
(11)***endwhile***

Fig. 3. The SDC algorithm.

2.1. Task graph

The DAG is a generic model of a workflow application consisting of a set of tasks (nodes) among which precedence constraints exist. It is represented by $G = (V, E)$, where V is the set of v tasks that can be executed on a subset of the available processors. E is the set of e directed arcs or edges between the tasks that maintain a partial order among them. The partial order introduces precedence constraints, i.e. if edge $e_{i,j} \in E$, then task v_j cannot start its execution before v_i completes. Matrix D of size $v \times v$ denotes the communication data size, where $d_{i,j}$ is the amount of data to be transferred from v_i to v_j. A task graph is a weighted graph. The weight w_i of a node v_i usually represents its computation cost. The weight of an edge stands for the communication requirement between the connected tasks (the amount of data that must be communicated between them). We introduce a new approach to assign node weight in Section 4.

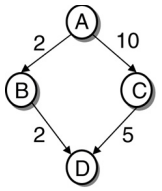
In a given task graph, a root node is called an *entry task* and a leaf node is called *exit task*. We assume that the task graph is a single-entry and single-exit one. If there is more than one exit or entry task, we can always connect them to a zero-cost pseudo exit or entry task with zero-cost edges. This will not affect the schedule.

2.2. Resource graph

A resource graph is an undirected weighted graph (both nodes and edges are weighted). A node of a resource graph represents a processor and an edge denotes the link between a pair of connected processors. The resource graph is a complete graph with p fully connected nodes. The weight of a node represents the processor computation capacity (the amount of computation that can be performed in a unit time). Similarly, the weight of an edge stands for its communication capacity (the amount of data that can go through the link in a unit time). We further assume that all inter-processor communications are performed without contention. This assumption holds since our computing environment consists of processors connected with wide area network links as pointed out in [29].

2.3. Performance criteria

Before presenting the performance criteria, it is necessary to define a few attributes used in the algorithm. The computation



(a)

Task	P1	P2	P3
A	1	∞	2
B	2	∞	1
C	∞	2	4
D	5	3	∞

(b)

	P1	P2	P3
P1	∞	1	1
P2	1	∞	1
P3	1	1	∞

(c)

Fig. 4. (a) An example DAG, (b) the computation cost for each node on three machines, (c) the communication cost table.

cost of task v_i on processor p_j is $w_{i,j}$. If v_i cannot be processed on p_j, then $w_{i,j} = \infty$. The data transfer rates between processors are kept in a matrix dtr of size $p \times p$. The startup cost of communication of each processor is stored in a vector sc of size p. The communication cost $c_{i|m,j|n}$ from task v_i to v_j when task v_i is scheduled on processor p_m and task v_j is scheduled on processor p_n is given by

$$c_{i|m,j|n} = sc_m + \frac{d_{i,j}}{\mathrm{dtr}_{m,n}} \tag{1}$$

we assume that intra-processor communication cost is negligible, i.e. $c_{i|m,j|m} = 0$. The task graph's edge weight is defined as the average communication cost:

$$c_{i,j} = \bar{sc} + \frac{d_{i,j}}{\bar{\mathrm{dtr}}} \tag{2}$$

$\mathrm{EST}(v_i, p_j)$ and $\mathrm{EFT}(v_i, p_j)$ are the earliest execution start time and the earliest execution finish time of task v_i on processor p_j respectively. The entry task can start execution at time 0. Other tasks' EST can be computed by

$$\mathrm{EST}(v_i, p_j) = \max\left\{\mathrm{avail}(v_i, p_j), \max_{v_k \in \mathrm{pred}(v_i)} (FT(v_k, p_{s_k}) + c_{k|s_k,i|j})\right\} \tag{3}$$

where $\mathrm{avail}(v_i, p_j)$ is the earliest time at which processor p_j is ready for task v_i's execution; $\mathrm{pred}(v_i)$ is the set of immediate predecessor tasks of task v_i. The inner max block in Eq. (3) is the time that all the data needed to execute task v_i on processor p_j is available, i.e. the *ready time*. This is obtained by considering all immediate predecessors of task v_i, the time they finish (FT) and the time needed to transfer data from the machine where they actually run on to the machine in consideration p_j. The EFT is defined by

$$\mathrm{EFT}(v_i, p_j) = w_{i,j} + \mathrm{EST}(v_i, p_j). \tag{4}$$

The schedule length L of the DAG is the actual finish time of the exit task v_{exit}.

$$L = \mathrm{FT}(n_{\mathrm{exit}}). \tag{5}$$

Although several performance criteria such as the lateness or the total flow time are suggested in the literature [30], our goal of scheduling in this research is to minimize the scheduling length L (*makespan*).

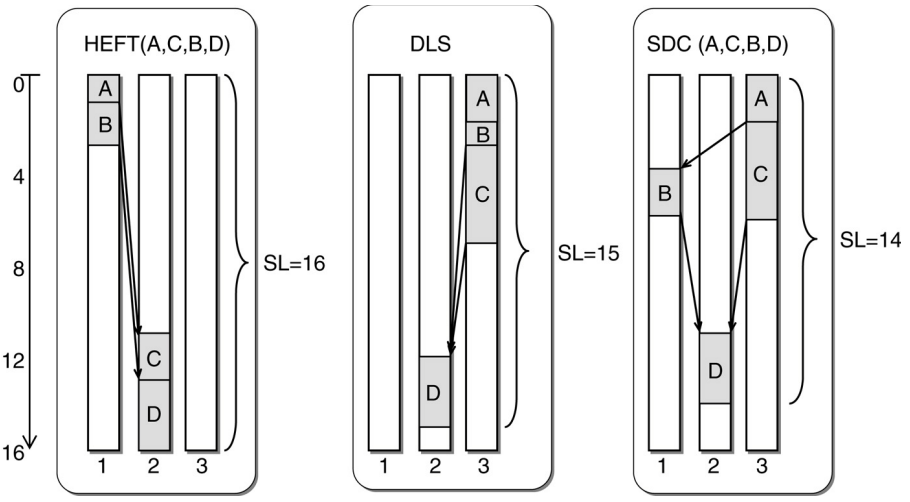


Fig. 5. (a) HEFT algorithm, (b) DLS algorithm, (c) SDC algorithm.

3. Related work

While static list scheduling is a well studied problem, it still gains attention due to its potential of generating good schedules without high time complexity. In [7] two list scheduling algorithms are introduced. The HEFT algorithm significantly outperformed the other algorithms in terms of both performance and cost metrics, including normalized schedule length, speedup, frequency of best results and average running time. The HEFT algorithm selects the task with the highest upward rank (*b-level*) at each step and assigns it to the processor which minimizes its earliest finish time with an insertion-based policy. The task node weight is computed as the average computation cost across all processors and the edge weight is the average communication cost across all edges.

An iterative list scheduling is proposed in a recent study [11]. This algorithm generates an initial solution with moderate quality and then improves it iteratively. The initial step is the same as the HEFT algorithm. Subsequently it modifies the weight of task nodes and edges on each iteration. The processor selection criteria is the minimization of EFT, which is the same as the HEFT algorithm. Simulation results show that an iterative algorithm can produce shorter schedule length than those of HEFT [7], DLS [9] and other algorithms. It is assumed that all tasks can be executed on any of the available processors.

Dynamic Level Scheduling (DLS) algorithm presented in [9] accounts for varying processor speeds as well as descendant effect and resource scarcity. It uses dynamically changing priorities to match tasks with processors at each step. Although this algorithm explicitly considers the effect of different processor capabilities, it has a high time complexity.

Scheduling tasks on processors with different capabilities is studied in [31]. The processors are homogeneous in the sense that each one has the same processing speed. The tasks are independent and cost unit time on each capable processor. The author proves that there exists an optimal solution for this restricted problem and gives a polynomial time algorithm to solve it. In the further research [32], two polynomial algorithms with less time complexities than those proposed in [31] are designed. Both study special cases and thus polynomial solutions are possible. In this paper, we consider situations where tasks have precedence constraints and bear different computation costs on capable processors.

The impact of different weight assignment methods is investigated in [28]. In heterogeneous environment one could use the task computing costs in various ways to compute the weight. In that study, the authors compare different methods such as *mean value*, *median value*, *best value*, *worst value* etc. Results show that significant variations in the makespan underline the dependency of the scheduling algorithms on the weight computing methods. The sensitivity is largely due to the fact that scheduling algorithms have difficulty to access the relative importance of independent tasks effectively.

4. The SDC algorithm

4.1. Setting task node weight

There are various ways to set the weights of task nodes in a heterogeneous setting [28]. For instance, one can take the average value, the best value, etc. In this paper, we consider the effect of scarcity of resources in addition to the average computation cost. We set relatively higher weight to the node with less capable resources. The rationale behind this is that tasks with scarce capable resources should be given higher priority in order to avoid situations that give rise to undesirable effects. This can be best illustrated with an example. In Fig. 1(a)–(c), an example task graph and its resource information are given. Task B and C have the same average computation cost. In addition, B can only be processed on processor 1. Fig. 2(a) shows the schedule when C is scheduled before B. In this case, task C has a smaller earliest finish time when it is assigned on processor 1. Task B has no choice but to be scheduled on processor 1. Assigning task C on processor 1 will delay the starting time of task B thus the whole schedule. To avoid this problem, we can intentionally assign larger weights to tasks with scarce capable processors. As illustrated by Fig. 2(b), task B is considered before task C.

B is still assigned on processor 1. This time C is scheduled on processor 2. The schedule length is reduced from 7 to 6 due to the change of scheduling order between task B and C.

The set of capable processors for node v_i is denoted as

$$CP(v_i) = \{p_k \mid w_{i,k} \neq \infty\}. \tag{6}$$

We define the *Percentage of Capable Processors* (PCP) of node v_i as

$$\mathrm{PCP}(v_i) = \frac{\|CP(v_i)\|}{p} \tag{7}$$

where p is the number of all processors. Thus the *Percentage of Incapable Processors* (PIP) is

$$\mathrm{PIP}(v_i) = 1 - \mathrm{PCP}(v_i). \tag{8}$$

The weight of node v_i is specified as

$$w_i = \frac{\sum_{p_j \in CP(v_i)} w_{i,j}/\|CP(v_i)\|}{\mathrm{PCP}(v_i)}. \tag{9}$$

By applying this specification we give relatively higher weights and thus higher priorities to those task nodes with fewer capable resources. Experimental results in Section 5 show that this method gives better schedules than the one using average computation costs.

4.2. Prioritizing the tasks

This step is essential for list scheduling algorithms. A task processing list is generated by sorting the task by decreasing order of some predefined rank function. In this research, we use *b-level* [2] as the rank function. The *b-level* of node v_i is the length of the longest path from v_i to the exit node. It can be obtained by recursively traversing the task graph from the exit node with time complexity $O(e + v)$.

$$\mathrm{BLEV}(v_i) = w_i + \max_{v_j \in \mathrm{succ}(v_i)} \{\overline{c_{i,j}} + \mathrm{BLEV}(v_j)\} \tag{10}$$

where $\overline{c_{i,j}}$ is the average communication cost of $e_{i,j}$, w_i is the weight of node v_i, and $\mathrm{succ}(v_i)$ is the set of immediate successors of v_i. Ties are broken randomly in order not to introduce high computing cost. The sorted list preserves the precedence constraints among tasks.

4.3. Selecting processors

Various criteria have been proposed to select suitable processors for a task. When scheduling in a homogeneous environment, *Earliest Start Time* is a popular choice [33,8,34], while in heterogeneous settings, using *Earliest Finish Time* as selection criteria gives better schedules [7]. Sih and Lee [9] suggest to select (node, processor) pair that maximize the so-called *Dynamic Level* at each step. They extend the definition of *Dynamic Level* by including the effects of descendant and resource scarcity when scheduling in heterogeneous systems. Furthermore, insertion based policy is better than a non-insertion based one as observed in [3]. Insertion based policy considers scheduling an idle time slot between two already schedule nodes as long as the slot is long enough and inserting the task to the slot doesn't break any precedence constraint.

The *Earliest Finish Time* method apparently fails to consider how well the descendants of current scheduling node v_i matches the selected p_j which minimizes the $\mathrm{EFT}(n_i, p_j)$. This is of particular importance in our computing environment where processors have different capabilities as identified in [9]. We propose a new target function called *Adjusted Earliest Finish Time* (AEFT). The SDC algorithm assigns task to the processor which minimizes the AEFT with an insertion-based policy. The *Adjusted Earliest Finish Time* is defined as

$$\mathrm{AEFT}(v_i, p_j) = \mathrm{EFT}(v_i, p_j) + \frac{1}{\|\mathrm{succ}(v_i)\|} \sum_{v_t \in \mathrm{succ}(v_i)} \sqrt[s_t]{\prod_{w_{t,k} \neq \infty} c_{i|j,t|k}} \tag{11}$$

where $s_t = \|CP(v_t)\|$ is the number of capable processors for task v_t. For each child v_t of v_i, we calculate the geometric average of its communication cost with v_i (assuming it is scheduled on p_j) when v_t is scheduled on each capable processor $p_k \in CP(v_t)$. The second term in Eq. (11) considers how the current node's allocation will affect the communication with its descendant on average. Without the second term, undesirable results can be produced. For example, in the case where $\mathrm{EFT}(v_i, p_j)$ is minimized, due to the scarcity of capable processors to execute v_i, it has to be placed on some processor, say p_k, where communication cost between p_j and p_k can be very expensive. This will undermine the overall quality of the resulting schedule.

4.4. Procedure of the algorithm

The pseudo code of the SDC algorithm is as follows. As with other list scheduling algorithms, the SDC algorithm has two major stages: a *task prioritizing stage* and a *processor selection stage*. The first stage computes the priorities of all the tasks while the second one selects the tasks in the order of their priorities and assigns each selected task on its most desirable processor, which minimizes the task's adjusted finish time. As an illustration, Fig. 4(a) presents a sample DAG. The number next to each edge of the graph corresponds to the amount of data that needs to be transferred from a task to its immediate successor. The cost to execute each of the 4 tasks in the graph on each of three different machines is given in Fig. 4(b). Fig. 4(c) shows the cost to transfer a data unit for any pair of machines. For the simplicity of illustration, we use unit data transfer rate. This is not assumed in the simulation experiments.

Fig. 5 shows the schedules obtained by HEFT, DLS and our SDC algorithm. The schedule length of SDC is shorter than those of the other two algorithms. The scheduling list of HEFT and SDC happens to be same, which is $\{A, C, B, D\}$. DLS algorithm does not maintain a static scheduling list. It selects a pair of (node, processor) that maximize the *Dynamic Level* at each step.

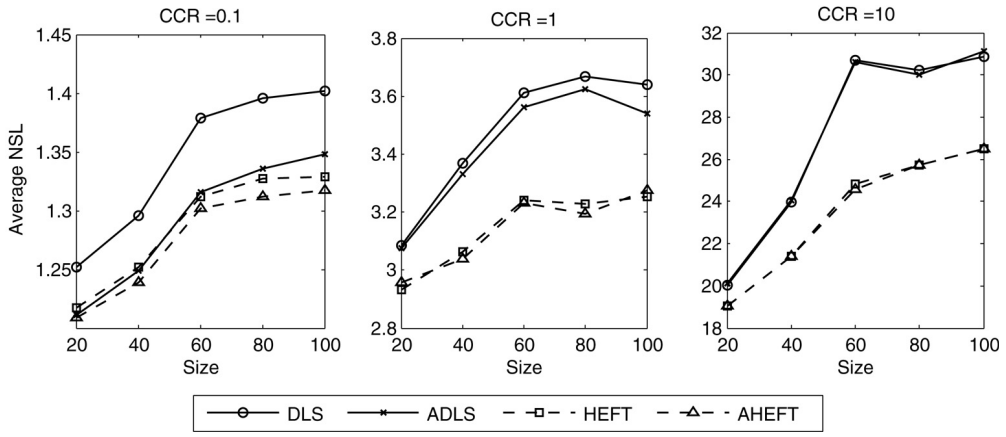


Fig. 6. The effect of weight assignment method on average NSL.

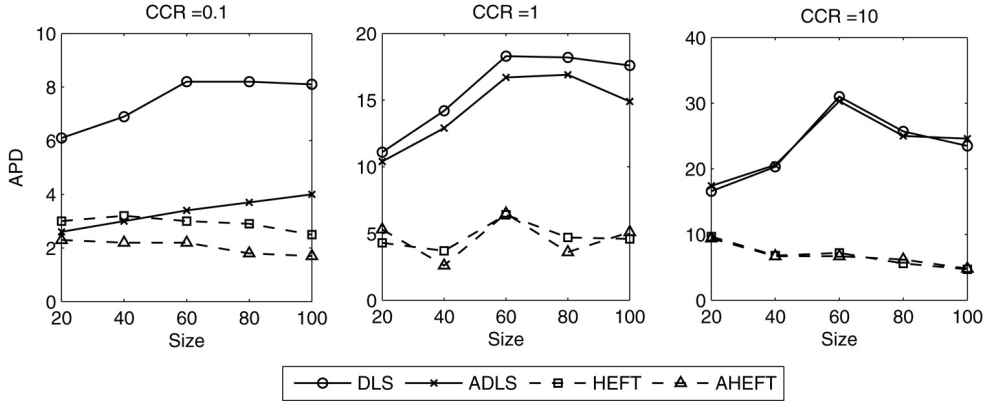


Fig. 7. The effect of weight assignment method on average percentage degradation.

4.5. Time-complexity analysis

We will refer to Fig. 3 when analyzing the time complexity of the algorithm. Line (1) and (2) take $O(vp)$ time. Line (3) can be done in $O(e + v)$ [2]. Sorting tasks in Line (4) takes at most $O(v \log v)$. Lines 5–11 will cost $O(ep^2)$ time. Thus the total time complexity is $O(ep^2)$. For a dense graph when e is proportional to $O(v^2)$, the time complexity becomes $O(v^2p^2)$.

5. Experimental results and discussion

We have evaluated our algorithm with a wide range of graphs. In this section, we present the comparative results of the SDC algorithm and some related work given in Section 3, namely HEFT and DLS. Randomly generated DAGs and a genomic sequence annotation workflow are considered for assessing the algorithms.

5.1. Comparison metrics

The comparisons of the three algorithms are made using the following two measures:

- *Normalized schedule length* (NSL). The principal performance metric of an algorithm is the length of its output schedule. The NSL of an algorithm is defined as:

$$\mathrm{NSL} = \frac{L}{\sum_{v_i \in cp_{\min}} \min_{p_j \in P} \{w_{i,j}\}} \tag{12}$$

where L is the schedule length. $cp_{\min}$ is the critical path of the DAG when the task node weights are evaluated as the minimum computation cost among all capable processors. The denominator represents a lower bound on the schedule length. Such a lower bound may not always be possible to reach and NSL ≥ 1 for any algorithm. We use averaged NSL over a set of DAGs as a comparison metric.

- *Average Percentage Degradation* from the best (APD). APD of an algorithm is the average (over all DAGs) of the percentage of degradation of the schedule lengths L produced by this algorithm from those best schedules. Let G denote a set of DAGs, $G = \{g_1, g_2, \ldots\}$. ALG $= \{\mathrm{alg}_1, \mathrm{alg}_2, \ldots\}$ is the set of algorithms we are comparing. $\mathrm{sl}(\mathrm{alg}_i, g_j)$ represents the schedule length of g_j using algorithm alg_i. The APD of algorithm alg_i over graph set G is defined as:

$$\mathrm{APD}(\mathrm{alg}_i, G) = \frac{\sum_{g_j \in G} \left(\mathrm{sl}(\mathrm{alg}_i, g_j) - \mathrm{sl}\left(\operatorname{argmin}_{\mathrm{alg} \in \mathrm{ALG}} \left(\mathrm{sl}(\mathrm{alg}, g_j) \right), g_j \right) \right)}{\|G\|}. \tag{13}$$

- *Efficiency*. The *speedup* of a task graph is defined as the time required for sequential execution of the graph in a single processor, divided by the time it takes to complete it with

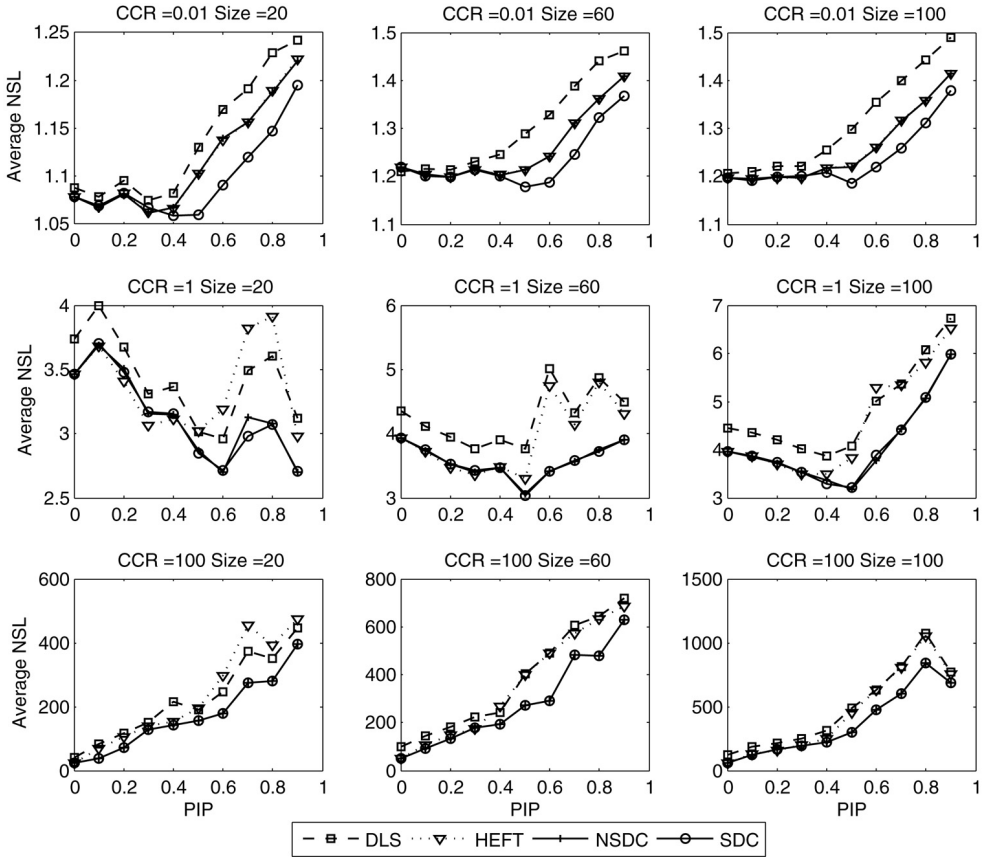


Fig. 8. Average NSL of the algorithms.

N processors. We assume that there is at least one processor can execute all the tasks when comparing efficiencies of various algorithms. The sequential execution time is obtained by assigning all tasks to a single processor that minimizes cumulatively the computation costs. *Efficiency* is the ratio of speedup to the number of processors used. We use efficiency as the metric to test the scalability of our algorithm. The results are presented in Section 5.2.3.3.

5.2. Randomly generated application graphs

5.2.1. Random task graph generation

In the first part of the evaluation, task graphs are generated randomly with the following input parameters:

- Task size in the graph (v). The value of v is assigned from the set {20, 40, 60, 80, 100}.
- Shape parameter of the graph (α). The height of a DAG $h = \frac{\sqrt{v}}{\alpha}$. α gets value from set {0.5, 1.0, 2.0}.
- Average computation cost($\overline{\text{comp}}$). The average computation cost of a task node is the average time required to complete the task on all of its capable processors. The average computation cost of task node v_i ($\overline{\text{comp}}_i$) is generated randomly from normal distribution $N(\overline{\text{comp}}, 0.5\overline{\text{comp}})$. Then the computation cost of v_i on processor p_j ($w_{i,j}$) is from normal distribution $N(\overline{\text{comp}}_i, 0.5\overline{\text{comp}}_i)$. The values of $\overline{\text{comp}}$ is from set {10, 20, 30, 40, 50}.
- Communication-to-Computation Ratio (CCR). The graph's CCR is the ratio of average communication cost to the average computation cost. CCR = {0.01, 0.1, 1.0, 10, 100}.
- Average communication cost($\overline{\text{comm}}$).

$$\overline{\text{comm}} = \text{CCR} * \overline{\text{comp}}. \tag{14}$$

- Percentage of Incapable Processors (PIP). This is defined in Eq. (8). There are two schemes used when setting PIP. In the first set of experiments, we investigate the effect of new weight assignment function on schedule length. The PIP of each task node is randomly generated from uniform distribution (0, 0.9). We want to evaluate how the SDC algorithm performs with respect to PIP in the second set of experiments. The PIP of task node v_i, PIP(v_i) is from normal distribution $N(\overline{\text{PIP}}, 0.5\overline{\text{PIP}})$, where $\overline{\text{PIP}} = 0, 0.1, 0.2, \ldots, 0.9$.

Three sets of experiments are conducted in this part of the evaluation. Experiment set I are designed to examine the effectiveness of weight assignment function described in Section 4.1. Experiment set II accesses the validity of processor selection criteria outlined in Section 4.2. When generating the graphs, each parameter set is repeated 25 times for the first set of experiments and 10 times for the second. This gives 9375 graphs for Experiment set I and 37,500 graphs for Experiment set II. Experiment set III evaluates the efficiency of the algorithm. The processor number varies from 4 to 64. Other parameters are the same as those of experiment set I. Results are presented in Section 5.2.3.

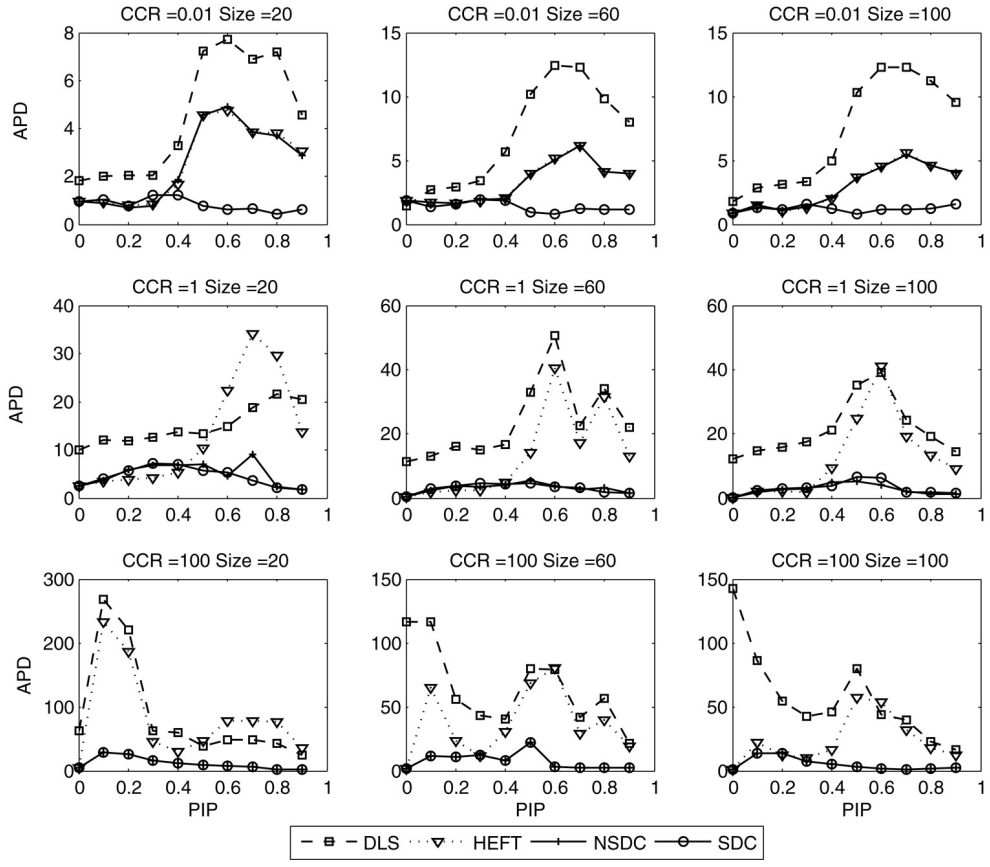


Fig. 9. Average percentage degradation of the algorithms.

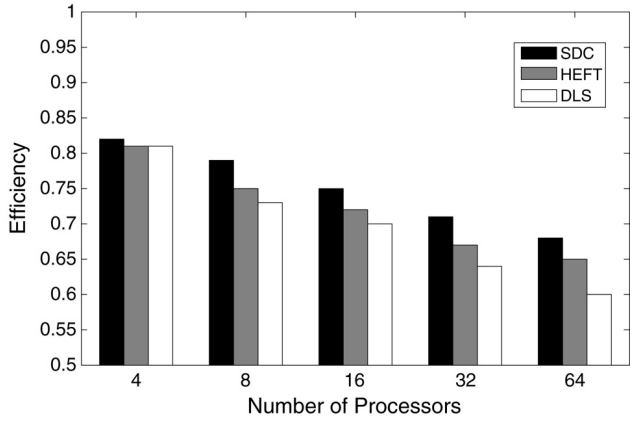


Fig. 10. Efficiency comparison with respect to the number of processors.

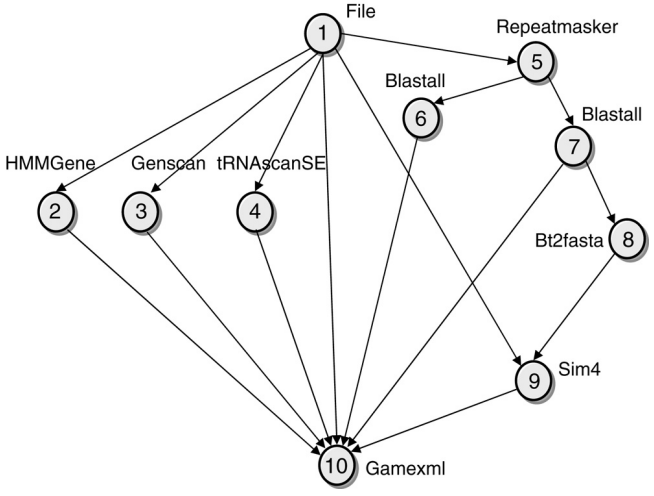


Fig. 11. A genomic sequence annotation workflow.

5.2.2. Generation of resource graph

The resource graph as described in 2.2 is a complete graph. The parameters we need to set are:

- Number of processors (p). In this study, we set $p = 10$.
- Average data transfer rate ($\overline{\text{dtr}}$). This is the average data transfer rate over all combinations of processors. We fix this value as 1. The data transfer rate between p_m and p_n ($\text{dtr}_{m,n}$) is from normal distribution $N(1, 0.5)$. We only uses numbers that are positive.
- Average data transfer size ($\overline{d}$). Since the average data transfer rate is 1, the average data transfer size is the same as the average communication cost. The data size to be transferred from task v_i to v_j is $d_{i,j} \sim N(\overline{\text{dtr}}, 0.5\overline{\text{dtr}})$.
- Startup cost. In this study, we omit the startup cost.

5.2.3. Performance comparison

The algorithm presented in Section 4 has two distinctive features: a new weight assignment method and a modified processor selection criteria. The effects of both features are presented in this section.

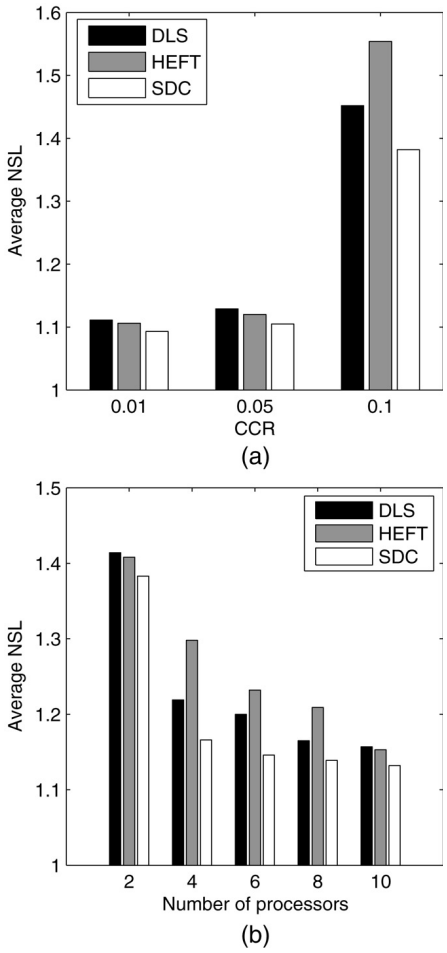


Fig. 12. Comparison of three algorithms on a genomic sequencing annotation workflow.

5.2.3.1. Effect of weight assignment function. Experiment set I investigates how the weight assignment method will impact the average NSL and APD. The results are shown in Figs. 6 and 7 respectively. Two algorithms, namely HEFT and DLS, and their modified counterparts AHEFT and ADLS are compared. The AHEFT (respectively ADLS) is adapted from HEFT (DLS) where our weight assignment method described in Section 4.1 is adopted. We only show the cases where CCR = 0.1, 1 and 10 due to space limits. From Fig. 6 we first observe that for all cases the average NSLs show an increasing trend with respect to the increasing of task graph size. This is due to the fact that the proportion of task nodes other than those on the critical path increases with the task graph size, making it more difficult to achieve the lower bound. We also notice that the adjusted algorithm performs better than their corresponding original version and the degree of improvement varies with respect to CCR. When CCR = 0.1, the improvement of modified DLS over DLS is 5.0% when task size is 100. The average NSL is reduced by 2.2% in the case of AHEFT (the adjusted HEFT algorithm) versus HEFT. When CCR increases to 10, there is no noticeable effect. Remember that we use *b-level* (defined in Eq. (10)) as the priority of task node. When CCR is large, average communication cost dominates BLEV in Eq. (10). Adjustment of task node weight does not really impact the priority, thus does not affect the scheduling list order. On the other hand, when CCR is small, assigning higher weight to those task nodes with large PIP can give higher priority to these tasks and therefore change the scheduling list order. As a result the schedule length is improved.

Fig. 7 depicts the degradation from the best solutions of the algorithms. When CCR = 0.1, the APD of AHEFT is less than 2.1% for all task graph sizes. On average, the APD of DLS is improved by 4.8% when CCR = 0.1. From the first graph of Fig. 7, we notice that the decrease of APD is more perceivable for DLS than for HEFT. In HEFT algorithm, the priority of each task node is set at the beginning and the scheduling list remains unchanged during the whole procedure. However, DLS algorithm reevaluates the *dynamic level* at each scheduling step and selects the (ready node, processor) pair that maximize it. The task weight constantly affects the *dynamic level* values during scheduling process.

5.2.3.2. Effect of processor selection criteria. Experiment set II to the processor selection policy. We investigate how the algorithms will be impacted under graphs with various characteristics. Fig. 8 gives the average NSL values of the algorithms at different CCR, task size and PIP. The DLS, HEFT and NSDC are the algorithms without weight assignment adjustment. When comparing 3 figures on each row, we notice that the average NSLs tend to increase with the increasing task graph size. This is consistent with previous observations in the first set of experiments. When CCR is small, NSDC algorithm performs almost the same as HEFT. This is because the second term of the definition for *Adjusted Earliest Finish Time* (Eq. (11)) is relatively small comparing to EFT. However, due to the significant impact of weight assignment function in the cases of small CCR, SDC generates better schedules overall. This is more obvious when PIP > 0.4. The average NSL of SDC algorithm is better than the DLS algorithm by 4.7%, and the HEFT algorithm by 1.6% when CCR = 0.01. When CCR is large, the communication cost becomes dominant. Scheduling the task without considering communication cost will suffer a huge performance penalty if the children of the node can only be processed on a subset of available resources. The average NSL of SDC is better than the DLS algorithm by 16.4%, and the HEFT by 28.3% when CCR = 100. It is also noticed that when CCR is large, the average NSL is large in that the lower bound of schedule length does not include any communication cost. Our algorithm works better overall especially when the heterogeneity of processor capabilities is considerable.

The Average Percentage Degradation (APD) of the algorithms at different parameters is given in Fig. 9. It can be seen that in almost all cases the APD of our algorithm remains the lowest. The APDs of other two algorithms fluctuate with respect to both PIP and CCR. This indicates that our algorithms are less sensitive to PIP and CCR compared to the other two.

5.2.3.3. Efficiency comparison. We compared the efficiency of three algorithms, namely, SDC, HEFT and DLS. The number of processors used is varied from 4 to 64, incrementing by the power of 2. The rest of the parameters are the same as those

used in Section 5.2.3.1. Fig. 10 shows the comparison with graph size 100 and CCR 0.1. SDC has better efficiency than the other two algorithms consistently. HEFT and DLS have comparable efficiency when processor number is small. With the increase of processor number, HEFT surpasses DLS with respect to efficiency.

5.3. Performance analysis on application graph of a genomic sequence annotation workflow

We further tested our algorithm on a genomic sequence annotation workflow [35]. Fig. 11 shows the task graph. In the figure, the DAG branches after the input sequence *File* node into a sub-DAG of analysis that works on the original input and a sub-DAG that analyzes the input sequence that is masked for repeats with *RepeatMasker*. The unmasked sequence is analyzed further using three software packages, namely *tRNAscanSE*, *Genscan* and *HmmGene* The masked sequence is searched against two databases using *Blastall*. The results from the latter search are further processed by an application (*bt2fasta*). This generates a new database of formatted gene sequences. The unmasked input sequence is then used as input to *Sim4*, which in turn aligns the input sequence to the entries in the newly created database. Results for all analyses are then integrated into an XML file for further interpretation using some annotation tool. In this workflow application, several domain specific softwares are involved. Because of those softwares' special requirements, some of them can only be installed on designated machines while others are available on all processors. This is a good example where processors bear different capabilities in terms of software availability.

There are 10 tasks and 16 edges in the graph. We set the computation units relatively according to the tasks' demands. The transferred data size is also specified approximating the corresponding file size. Processor number is varied between 2 and 10. The PIP of each processor is set randomly. Fig. 12(a) shows the performance of the algorithms with respect to three different CCR values. The highest CCR is set to 0.1 because the workflow is computation-intensive in reality. On the average, SDC performs best among the three algorithms. The performance gain is more notable for larger CCRs. Comparing to DLS, HEFT produces better schedules when CCR is small. When CCR increases to 0.1, the trend is reversed. In [7], where all the processors can handle every task, the conclusion is different. The authors observe that HEFT always obtains smaller average NSL when testing with a modified molecular dynamic task graph. The comparison of three algorithms regarding different processor numbers is given in Fig. 12(b). It is noticed from the figure that since there are at most 4 tasks in any level in the task graph, increasing processor number does not significantly reduce SLR if $p > 4$. The SDC algorithm outperforms the other two algorithms in all cases.

6. Conclusions

In this paper, we presented a new algorithm for scheduling DAG based workflow applications in heterogeneous systems where processors have different capabilities. The algorithm has two distinctive features. First, we suggest considering the effect of tasks' scarcity of capable processors when assigning the task node weights. For two task nodes with the same average computation cost, our weight assignment policy tends to give higher weight to the task with small PCP. Secondly, during the processor selection phase, we adjust the effective *Earliest Finish Time* strategy by incorporating the average communication cost between the current scheduling node and its children. We evaluate the algorithm using a large set of randomly generated task graphs with different characteristics and a real world bioinformatics workflow application. Results show that each feature of the SDC algorithm improves the schedule length. It is noted that the new weight assignment policy impacts the schedule perceivably when CCR is small while the processor selection strategy affects the schedule length more substantially at larger CCR. By combining the two strategies, the SDC algorithm outperforms the other two algorithms overall. Efficiency comparison among the three algorithms reveals that SDC scales well for varying processor number.

Our future work goes in the direction of extending the SDC algorithm to the dynamic environment where processor load, capability and network condition vary during the execution of workflow applications. We also plan to implement the algorithm in a distributed system called GridSolve [36]. Currently, GridSolve does not have the capability to execute workflow applications with user coordination. We aim to enhance it by augmenting the Agent's ability to schedule the whole workflow using SDC algorithm. GridSolve servers usually have different capabilities (software) since they are set up by different service providers. Our algorithm should work better in this situation.

References

[1] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, Tech. Rep., University of Melbourne, Australia, March 2005.
[2] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (4) (1999) 406–471.
[3] Y.-K. Kwok, I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, J. Parallel Distrib. Comput. 59 (3) (1999) 381–422.
[4] H. El-Rewini, T.G. Lewis, H.H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice Hall, Englewood Cliffs, NJ, 1994.
[5] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, K. Kennedy, Task scheduling strategies for workflow-based applications in grids, 2005.
[6] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, 1979.
[7] H. Topcuouglu, S. Hariri, M.Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[8] M.Y. Wu, D.D. Gajski, Hypertool: A programming aid for message-passing systems, IEEE Trans. Parallel Distrib. Syst. 1 (3) (1990) 330–343.
[9] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (2) (1993) 175–187.
[10] H. El-Rewini, T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, J. Parallel Distrib. Comput. 9 (2) (1990) 138–153.
[11] G. Liu, K. Poh, M. Xie, Iterative list scheduling for heterogeneous computing, J. Parallel Distrib. Comput. 65 (5) (2005) 654–665.

[12] R. Sakellariou, H. Zhao, A hybrid heuristic for dag scheduling on heterogeneous systems, in: Proceedings of 13th Heterogeneous Computing Workshop, HCW2004, Santa Fe, NM, 2004.
[13] T. Yang, A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distrib. Syst. 5 (9) (1994) 951–967.
[14] S. Kim, J. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proceedings of International Conf. Parallel Processing, vol. 2, 1988, pp. 1–8.
[15] J. Liou, M. Falls, An efficient clustering heuristic for scheduling dags on multiprocessors, in: Proc. Symp. Parallel and Distributed Processing, 1996.
[16] B. Kruatrachue, T. Lewis, Grain size determination for parallel processing, IEEE Softw. 5 (1) (1988) 23–32.
[17] Y.-C. Chung, S. Ranka, Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors, in: Supercomputing'92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 512–521.
[18] G.-L. Park, B. Shirazi, J. Marquis, DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems, in: IPPS'97: Proceedings of the 11th International Symposium on Parallel Processing, IEEE Computer Society, Washington, DC, 1997, pp. 157–166.
[19] I. Ahmad, Y. Kwok, A new approach to scheduling parallel programs using task duplication, in: Proc. Int'l Conf Parallel Processing, vol. 2, 1994, pp. 47–51.
[20] E.S.H. Hou, N. Ansari, H. Ren, A genetic algorithm for multiprocessor scheduling, IEEE Trans. Parallel Distrib. Syst. 5 (2) (1994) 113–120.
[21] T.D. Braun, H.J. Siegel, N. Beck, L.L. Boloni, A.I. Reuther, M.D. Theys, B. Yao, R.F. Freund, M. Maheswaran, J.P. Robertson, D. Hensgen, A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems, in: HCW'99: Proceedings of the Eighth Heterogeneous Computing Workshop, IEEE Computer Society, Washington, DC, 1999, p. 15.
[22] H. Singh, A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in: Proc. Heterogeneous Computing Workshop, 1996, pp. 86–97.
[23] P. Shroff, D. Watson, N. Flann, K. Freund, Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments, in: Proc. Heterogeneous Computing Workshop, 1996, pp. 98–104.
[24] L. Wang, H. Siegel, V. Roychowdhury, A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments, in: Proc. Heterogeneous Computing Workshop, 1996.
[25] R. Correa, A. Ferreria, P. Rebreyend, Integrating list heuristics into genetic algorithms for multiprocessor scheduling, in: Proc. Eighth IEEE Symp. Parallel and Distributed Processing, SPDP'96, 1996.
[26] Y.-K. Kwok, I. Ahmad, Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (5) (1996) 506–521.
[27] A. Radulescu, A.J.C.V. Gemund, Fast and effective task scheduling in heterogeneous systems, in: HCW'00: Proceedings of the 9th Heterogeneous Computing Workshop, IEEE Computer Society, Washington, DC, 2000, p. 229.
[28] H. Zhao, R. Sakellariou, An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm, in: Proceedings of 9th International Euro-Par Conference, Springer-Verlag, 2003.
[29] H. Casanova, Network modeling issues for grid application scheduling, Int. J. Found. Comput. Sci. 16 (2) (2005) 145–162.
[30] P. Brucker, Scheduling Algorithms, Springer-Verlag, 2004.
[31] R.S. Chang, R.C.T. Lee, On a scheduling problem where a job can be executed only by a limited number of processors, Comput. Oper. Res. 15 (5) (1988) 471–478.
[32] Y.L. Chen, Y.H. Chin, Scheduling unit-time jobs on processors with different capabilities, Comput. Oper. Res. 16 (5) (1989) 409–417.
[33] T.L. Adam, K.M. Chandy, J.R. Dickson, A comparison of list schedules for parallel processing systems, Commun. ACM 17 (12) (1974) 685–690.
[34] J.-J. Hwang, Y.-C. Chow, F.D. Anger, C.-Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, SIAM J. Comput. 18 (2) (1989) 244–257.
[35] S.P. Shah, D.Y. He, J.N. Sawkins, J.C. Druce, G. Quon, D. Lett, G.X. Zheng, T. Xu, B.F. Ouellette, Pegasys: software for executing and integrating analyses of biological sequences, BMC Bioinformatics 5 (2004) 40.
[36] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, J. Dongarra, Recent development in gridsolve, Int. J. High Perform. Comput. Appl. 20 (1) (2006) (special issue) (in press).

Zhiao Shi is currently a PhD student in the Computer Science Department, the University of Tennessee at Knoxville. His research interests include task scheduling for heterogeneous and Grid environments, workflow scheduling algorithms and performance modeling. Previously he received his MS degree in chemical engineering from Kansas State University in 2000.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee, has the position of a Distinguished Research Staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), and an Adjunct Professor in the Computer Science Department at Rice University.

He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports and technical memoranda and he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches. He is a Fellow of the AAAS, ACM, and the IEEE and a member of the National Academy of Engineering.