A Task Duplication Based Bottom-Up Scheduling Algorithm for Heterogeneous Environments*

Doruk Bozdağ†, Umit Catalyurek‡†, Füsun Özgüner†

†The Ohio State University
Dept. of Electrical and Computer Engineering
Columbus, OH 43210 USA
{bozdagd, ozguner}@ece.osu.edu

‡The Ohio State University
Dept. of Biomedical Informatics
Columbus, OH 43210 USA
umit@bmi.osu.edu

Abstract

We propose a new duplication-based DAG scheduling algorithm for heterogeneous computing environments. Contrary to the traditional approaches, proposed algorithm traverses the DAG in a bottom-up fashion while taking advantage of task duplication and task insertion. Experimental results on random DAGs and three different application DAGs show that the makespans generated by the proposed DBUS algorithm are much better than those generated by the existing algorithms, HEFT, HCPFD and HCNF.

1 Introduction

Task scheduling for multiprocessor systems has been a well studied problem for many decades. Numerous algorithms have been proposed to achieve speedup on parallel applications represented in the form of *directed acyclic graphs (DAGs)*. Task scheduling problem is NP-complete [9], therefore proposed approaches are mainly heuristics except for some special cases [2, 6].

There are many algorithms that produce high quality solutions [1, 5] for the case of homogeneous computing systems. However, scheduling in heterogeneous computing systems is a far more complicated problem due to non-uniform processor speeds and communication link bandwidths. Two of the most important classes of scheduling algorithms are list-based and cluster-based algorithms. List-based scheduling is very popular for heterogeneous environments due to its low complexity and good quality of resulting schedules [2, 3, 8, 10, 12, 13]. Cluster based scheduling on the other hand, is not extensively investigated except a few studies [4, 7].

One of the most popular algorithms in heterogeneous DAG scheduling is the HEFT algorithm [13]. HEFT schedules each task on the processor that provides the earliest possible finish time and employs task insertion. However, HEFT does not allow task duplication. HCPFD [10] is a duplication based algorithm and similar to HEFT, a task is scheduled on a processor that provides the earliest finish time. In addition, the critical parent of the task is also duplicated on the selected processor only if this duplication improves the task's finish time. HCPFD does not allow task insertion, therefore tasks can only be scheduled after the latest scheduled task on a processor. In contrast to HCPFD, another duplication based algorithm, HCNF [3], checks on all processors if duplicating the task together with its critical parent can yield a better finish time for the task, before selecting where to schedule the task.

We have chosen the above three algorithms, HEFT, HCPFD and HCNF, to compare with the DBUS algorithm we are proposing in this work, since these algorithms have different scheduling properties and comparable time complexities (see Table 1). There are many other algorithms for heterogeneous DAG scheduling with slightly different focus. For example, LDBS algorithm [8] aims at minimizing the schedule length at the expense of a high time complexity of $O(|\mathcal{N}|^3|\mathcal{E}||\mathcal{P}|^2)$, where $|\mathcal{N}|$ and $|\mathcal{E}|$ denote the number of tasks and edges in a DAG and $|\mathcal{P}|$ denotes the number of processors. On the other hand, FCP algorithm [12] aims just the opposite; the performance of this algorithm is

*This research was supported in part by the National Science Foundation under Grants #CCF-0342615, #ANI-0330612, #CNS-0426241, NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003, Sandia National Laboratories under Doc.No: 283793.

1-4244-0054-6/06/$20.00

Algorithm	Task insertion	Restrictions on duplication	Complexity
HEFT [13]	Yes	No duplication	$O(\|\mathcal{N}\|^2\|\mathcal{P}\|)$
HCPFD [10]	No	Only critical parent of a task	$O(\|\mathcal{N}\|^2\|\mathcal{P}\|)$
HCNF [3]	No	Only critical parent of a task	$O(\|\mathcal{N}\|^2(log\|N\| + \|P\|)$
DBUS	Yes	No restriction	$O(\|\mathcal{N}\|^2\|\mathcal{P}\|^2)$

Table 1. Comparison of scheduling algorithms

slightly worse than HEFT, however, it has a smaller time complexity of $O(|\mathcal{N}|log|\mathcal{P}| + |\mathcal{E}|)$. Finally, TDS algorithm [2] is proved to produce optimal schedules if a set of conditions is met. However, TDS algorithm is designed for networks with homogeneous links and does not handle heterogeneous links.

In this work, we propose a novel scheduling algorithm DBUS that benefits both from task insertion and task duplication. The proposed algorithm traverses the DAG in a bottom-up fashion contrary to the traditional approaches and does not impose any restrictions on the number of task duplication. We start with some preliminaries in the next section and the details of the DBUS algorithm are presented in Section 3. Experimental results presenting the performance of DBUS in comparison with HEFT, HCPFD and HCNF algorithms are presented in Section 4 and we conclude with Section 5.

2 Preliminaries

A DAG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ consists of a set of nodes $\mathcal{N}$ representing the tasks and a set of directed edges $\mathcal{E}$ representing dependencies among tasks. The edge set $\mathcal{E}$ contains edges $(n_p, n_j) \in \mathcal{E}$ for each task n_p (*parent*) that n_j (*child*) depends on. A child task depends on its parent tasks such that the execution of a child task cannot start before it receives data from all of its parents. A task having no parents is called an *entry task* whereas a task having no children is called an *exit task*.

The set of processors in a heterogeneous computing environment is denoted by $\mathcal{P}$, where each processor in this set is assumed to execute each task without preemption. The non-zero weight $w_{i,\ell}$ of a task n_i represents the expected execution time of the task on processor p_ℓ and it is assumed to be known a priori. There are several techniques in the literature such as statistical prediction [11] and analytical benchmarking [14] to estimate these weights. The average execution time of a task n_i can be found as $\overline{w_i} = \frac{1}{|\mathcal{P}|}\sum_{\ell=1}^{|\mathcal{P}|} w_{i,\ell}$.

The communication volume associated with a directed edge (n_i, n_j) is represented by $v_{i,j}$. Let $s_{\ell,k}$ and $t_{\ell,k}$ denote the communication startup cost and the expected time required to transfer a single unit of data between processors p_ℓ and p_k, respectively. The communication cost from task n_i scheduled on processor p_ℓ to task n_j scheduled on processor p_k is calculated as $c(n_i, p_\ell, n_j, p_k) = s_{\ell,k} + v_{i,j} \times t_{\ell,k}$. Here, it is assumed that the links in the heterogeneous computing network is contention free. The average communication cost associated with edge (n_i, n_j) is defined as $\overline{c_{i,j}} = \overline{s} + \overline{t} \times v_{i,j}$, where $\overline{s}$ is the average communication startup cost and $\overline{t}$ is the average time required to transfer a unit of data in the network.

The DBUS algorithm to be introduced in the next section schedules the tasks in the DAG in a bottom-up fashion, scheduling all children of a task before scheduling the task itself. Therefore, for the sake of simplicity in the presentation, the finish time of the first scheduled task (exit task) is taken to be 0 and the start time of a task n_i scheduled on processor p_ℓ is computed as

$$st(n_i, p_\ell) = ft(n_i, p_\ell) + w_{i,\ell}$$

Here, $ft(n_i, p_\ell)$ denotes the finish time of task n_i on processor p_ℓ. If a task n_i is not scheduled on a processor p_ℓ, its start time is set to 0, i.e. $st(n_i, p_\ell) = 0$. Schedule length is defined as

$$\text{Schedule Length} = \max_{n_j \in \mathcal{N}, p_\ell \in \mathcal{P}} st(n_j, p_\ell)$$

The objective of proposed scheduling algorithm is to schedule the tasks of a DAG such that the schedule length is minimized.

We define $cover\_lst(n_i, p_\ell, p_k)$ as the *latest start time* of task n_i on processor p_k such that the children of n_i on p_ℓ can receive data in time from the copy of n_i on p_k. A special case, $cover\_lst(n_i, p_{-1}, p_k)$, denotes the latest start time of n_i on p_k regardless of any dependency between n_i and its children. $cover\_lst$ can be computed as:

$$cover\_lst(n_i, p_\ell, p_k) = slot(p_k, w_{i,k}, lst_{\ell,k}(n_i))$$

where $slot(p_k, w, t)$ denotes the start of first empty slot on processor p_k of size w before time t. Recall that our time line is backward, hence $slot(p_\ell, w, t) \geq t$. $lst_{\ell,k}(n_i)$ is the latest allowable start time of task n_i on processor p_k to satisfy dependency requirements between n_i and its children scheduled on p_ℓ. $lst_{\ell,k}(n_i)$ can be computed as follows:

$$lst_{\ell,k}(n_i) = \begin{cases} w_{i,k} & \text{if } \ell = -1 \\ \max_{(n_i,n_j)\in\mathcal{E}}(st(n_j,p_\ell)) & \\ \quad + w_{i,k} & \text{if } \ell = k \\ \max_{(n_i,n_j)\in\mathcal{E}}(st(n_j,p_\ell)+ & \\ \quad c(n_i,p_k,n_j,p_\ell)) + w_{i,k} & \text{if } \ell \neq k \end{cases}$$

n_i	$w_{i,1}$	$w_{i,2}$	$w_{i,3}$	$\overline{w_i}$
n_1	2	1	3	2.00
n_2	4	2	7	4.33
n_3	6	5	8	6.33
n_4	2	2	1	1.67
n_5	2	3	3	2.67

Table 2. Entry $w_{i,\ell}$ represents execution time of task n_i on processor p_ℓ.

3 DBUS: Duplication-based Bottom-Up Scheduling Algorithm

The DBUS algorithm consists of a critical path-based listing phase followed by a duplication-based scheduling phase. In the traditional approach of top-down DAG traversal, duplication is carried out when a task's finish time can be improved with duplication of its critical parent on the same processor with the task itself. This means that a task with multiple children can be duplicated on multiple processors if its duplication helps improving its children's finish time on those processors. The drawback in this approach is that, the duplication of the parent task on each of such processors is done independently. Therefore the positions of the parent task on relevant processors are decided without any optimization with respect to each other. By traversing the DAG in a bottom up fashion, the proposed algorithm first schedules all children of a task before scheduling the task on as many processors as necessary. Consequently, it is more likely to make good duplication decisions since all copies of the parent task are considered at the same time. In addition, the stop criterion for duplication is not determined by the number of duplications already carried out, but by the quality of the current schedule. Since the number of beneficial duplications may differ significantly across different problems, DBUS offers superior performance compared to algorithms that limit duplication by the number of duplicates. Furthermore, the DBUS algorithm is an insertion based algorithm that allows tasks to be scheduled at the first available time slot that can accommodate themselves. Task insertion based algorithms have better chances of finding shorter schedule lengths compared to non-insertion based ones. In the remainder of this section, the phases of the DBUS algorithm are presented in more detail.

3.1 Listing Phase

DBUS algorithm schedules tasks in a bottom-up fashion. Therefore, in order to prevent any dependency violation, a task is said to be *ready* for scheduling only if all of its children are already scheduled. At any point during the progress of the algorithm, there may be more than one task ready for scheduling. Among these tasks, which one to schedule next is a heuristic choice. There are techniques in the literature that assigns priorities to tasks, so that the task having the highest priority among the ready tasks is selected to be scheduled next. We have implemented 6 popular techniques, namely, prioritizing based on critical path, top-levels, bottom-levels, static top-levels, static bottom-levels and average task weights [13, 10]. While using one of these techniques as the main method, we used another one as a tie breaker. We tested every possible permutation of the mentioned techniques and decided on prioritizing based on critical path as the main method and static top-levels as the tie breaker being the best choices for our DBUS algorithm.

In critical path based prioritizing, nodes on the critical path are determined and they are attempted to be scheduled before other tasks. The bottom level $b\_level(n_i)$ of a task n_i is computed by traversing the DAG upward starting from the exit tasks. It is defined as $b\_level(n_i) = \overline{w_i} + \max_{(n_i,n_j)\in\mathcal{E}}(\overline{c_{i,j}} + b\_level(n_j))$. The b_level of an exit task n_i is defined to be $\overline{w_i}$. Similarly, top level $t\_level(n_i)$ for task n_i is computed by traversing the DAG downward starting from the entry tasks and is defined as $t\_level(n_i) = \max_{(n_j,n_i)\in\mathcal{E}}(\overline{w_j} + \overline{c_{j,i}} + t\_level(n_j))$. Static top level, denoted by st_level, is defined similar to t_level, however communication cost is not taken into account: $st\_level(n_i) = \max_{(n_j,n_i)\in\mathcal{E}}(\overline{w_j} + t\_level(n_j))$. The t_level and st_level of an entry task are both defined to be 0.

A node n_i is on the critical path (CP) if $t\_level(n_i) + b\_level(n_i) = \max_{n_j\in\mathcal{N}}(t\_level(n_j) + b\_level(n_j))$. The listing heuristic presented in Algorithm 1 determines and sorts the CP nodes in non-increasing t_level order. In order to schedule a CP task, all of its children should have been scheduled. The recursive ADDTOLIST function makes sure that the children of each task are inserted into the priority list $\mathcal{R}$ at an earlier position than the task itself. The child of the task that has a higher st_level will be considered for scheduling before others and hence it is inserted into $\mathcal{R}$ at an earlier position.

In the example given in Figure 1, a DAG with 5 tasks is to be scheduled on a heterogeneous system with three

Algorithm 1 Listing

```
1: function LISTING(G)
2:     calculate b_level(n_i), t_level(n_i) and st_level(n_i) ∀n_i ∈ N
3:     cp_list ← CP tasks in non-increasing t_level order
4:     ADDTOLIST(R, 0, cp_list)
5:     return R
6: function ADDTOLIST(R, pos, list)
7:     for ℓ = 0 to length(list) do
8:         child_list ← children of list[ℓ] ∉ R in non-increasing st_level order
9:         pos ← ADDTOLIST(R, pos, child_list)
10:        R[pos] ← list[ℓ]
11:        pos ← pos + 1
12:    return pos
```

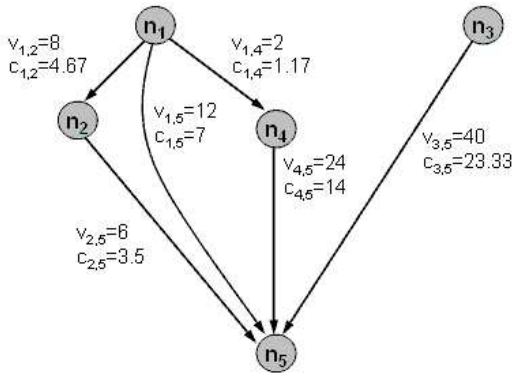

Figure 1. An example DAG. $v_{i,j}$ represents communication volume and $c_{i,j}$ represents average communication time on the example system between tasks n_i and n_j.

p_ℓ	$t_{\ell,1}$	$t_{\ell,2}$	$t_{\ell,3}$
p_1	-	1	0.25
p_2	1	-	0.5
p_3	0.25	0.5	-

Table 3. Entry $t_{\ell,k}$ represents per unit data transfer time from processor p_ℓ to processor p_k.

processors. Entry $w_{i,\ell}$ in Table 2 represents the execution time of task n_i on processor p_ℓ and entry $t_{\ell,k}$ in Table 3 represents per unit data transfer time from processor p_ℓ to processor p_k. For simplicity in the presentation, communication start time ($s_{\ell,k}$) is considered negligible. There are two weights associated with each edge in the DAG. The first one corresponds to communication volume $v_{i,j}$, and the second one corresponds to average communication cost $\overline{c_{i,j}}$ between tasks n_i and n_j. According to Table 3, the average per unit data transfer time between two distinct processors, $\overline{t}$, is computed to be 0.58 time units, therefore $\overline{c_{i,j}} = 0.58 \times v_{i,j}$. Please note that average execution and communication times are only relevant for the listing phase, and have no significance in the actual scheduling phase. The results of the listing phase for the example DAG is given in Table 4. First, n_5 is inserted into $\mathcal{R}$ since it is on the critical path and has no child. It is followed by n_3, which is the only task other than n_5 on the critical path. Among the remaining tasks, n_2 and n_4 have the largest st_levels, and n_2 is randomly chosen as the next task. Finally n_4 and n_1 are inserted in $\mathcal{R}$ into last two positions.

3.2 Scheduling Phase

The pseudocode of the scheduling phase of the DBUS algorithm is given in Algorithm 2. DBUS schedules the tasks in a bottom-up fashion as determined in the listing phase. The next task in $\mathcal{R}$ to be considered for scheduling is denoted by n_t. Initially, n_t is considered for duplication on every processor, rather than only on processors that a child of n_t is scheduled. This choice allows exploration of scheduling alternatives where a better latest start time (lst) for n_t can be found by scheduling n_t on a processor that none of its children has been scheduled. Therefore $cover\_lst(n_t, p_m, p_m)$ is calculated for each processor $p_m \in \mathcal{P}$. Note that

n_i	$t\_level(n_i)$	$b\_level(n_i)$	$t\_level(n_i) + b\_level(n_i)$	$st\_level(n_i)$	position in $\mathcal{R}$
n_1	0.00	21.50	21.50	0.00	5
n_2	6.67	10.50	17.17	2.00	3
n_3	0.00	32.33	32.33	0.00	2
n_4	3.17	18.33	21.50	2.00	4
n_5	29.67	2.67	32.33	6.33	1

Table 4. Results of the listing phase (Algorithm 1) for the example DAG in Figure 1.

Algorithm 2 DBUS: Duplication-based Bottom-Up Scheduling Algorithm

1: **for** $i = 0$ to $length(\mathcal{R})$ **do**
2: $n_t \leftarrow \mathcal{R}[i]$
3: $\mathcal{D} \leftarrow \emptyset$
4: $\mathcal{C} \leftarrow \{p_m | p_m \in \mathcal{P}$ and $\exists (n_t, n_j) \in \mathcal{E}$ such that $st(n_j, p_m) \neq 0\}$
5: **if** $\mathcal{C} = \emptyset$ **then** ▷ n_t is an exit task
6: schedule n_t on the processor that gives $\min_{p_\ell \in \mathcal{P}}\ cover\_lst(n_t, p_{-1}, p_\ell)$
7: **else**
8: INSERT $(lstQ, p_m, cover\_lst(n_t, p_m, p_m))$ for all $p_m \in \mathcal{P}$
9: **while** $\mathcal{C} \neq \emptyset$ **do**
10: $< p_\ell, lst\_level > \leftarrow$ EXTRACTMAX $(lstQ)$
11: **if** $p_\ell \in \mathcal{C}$ **then**
12: $min\_cover\_lst \leftarrow \min_{p_k \in lstQ}\ cover\_lst(n_t, p_\ell, p_k)$
13: $p_k \leftarrow$ the processor associated with min_cover_lst
14: **if** $min\_cover\_lst \geq lst\_level$ **then**
15: SCHEDULE (n_t, p_ℓ, lst_level)
16: **else**
17: SCHEDULE $(n_t, p_k, min\_cover\_lst)$
18: UPDATE $(lstQ, p_k, min\_cover\_lst)$
19: $free\_lst \leftarrow cover\_lst(n_t, p_{-1}, p_\ell)$
20: **if** $free\_lst < lst\_level$ **then**
21: INSERT $(lstQ, p_\ell, free\_lst)$

$cover\_lst(n_t, p_m, p_m)$ is the latest start time of n_t on p_m such that n_t will be executed before its children scheduled on p_m. Processors are inserted into a priority queue ($lstQ$) using their $cover\_lst$ times as their keys by the INSERT function in step 8 of the algorithm. If a task needs to be scheduled on a processor, the key of that processor represents the task's computed latest start time. As a greedy choice, we will go over the processors in that queue one-by-one in non-increasing key order and check whether the duplication on that processor is required or another processor can cover for that one.

The set $\mathcal{C}$ is defined as the set of processors that should be *covered* by task n_t. A processor p_ℓ is referred to as covered for task n_i only if all children of n_i on p_ℓ are guaranteed to receive data from a copy of n_i before their scheduled start time. Initially, all processors on which at least one of n_t's children is scheduled are included in the set $\mathcal{C}$.

The **while** loop in steps 9-21 of the algorithm terminates only after all processors in set $\mathcal{C}$ are covered. At each iteration of the loop, the maximum value in the $lstQ$ is extracted and assigned to lst_level and p_ℓ is assigned the processor associated with this value. If p_ℓ is in set $\mathcal{C}$, it is searched (step 12) if duplication of n_t on a processor other than p_ℓ can provide a better latest start time than lst_level while providing data to all children of n_t on p_ℓ. If no such processor is found, n_t is duplicated on p_ℓ to start at lst_level (step 15).

If a duplicate of n_t on another processor p_k that can cover its children on p_ℓ with a smaller latest start time (called min_cover_lst) can be found, n_t is duplicated on p_k to start at min_cover_lst. Then, p_k, min_cover_lst pair is inserted into the $lstQ$ by the UPDATE function as min_cover_lst is smaller than the current lst_level. The reason is that, processor p_k may be used to cover some other processor (by rescheduling n_t on an earlier time on p_k) as long as min_cover_lst is not the maximum value in the $lstQ$ (step 18). Since the children of n_t on p_ℓ are now covered by the dupli-

Algorithm 3 Schedule

```
1: function SCHEDULE(n_t, p_ℓ, lst)
2:     if p_ℓ ∈ 𝒟 then
3:         remove duplicate of n_t on p_ℓ        ▷ new copy can cover all tasks that the existing one covers
4:     st(n_t, p_ℓ) ← lst
5:     𝒟 ← 𝒟 ∪ {p_ℓ}
6:     for each p_m ∈ 𝒞 do
7:         if lst ≥ cover_lst(n_t, p_m, p_ℓ) then
8:             𝒞 ← 𝒞 − {p_m}
```

n_t	$lstQ$: p_ℓ, $cover\_lst$	p_k, min_cover_lst	Dest. proc.	Covered
n_5	$p_1, 2$		p_1	
	$p_1^*, 8$	$p_3, 20$	p_1	p_1
n_3	$p_3, 8$			
	$p_2, 5$			
	$p_1^*, 12$	$p_2, 10$	p_2	p_1
n_2	$p_3, 7$			
	$p_2, 2$			
	$p_1^*, 10$	$p_3, 9$	p_3	p_1
n_4	$p_2, 2$			
	$p_3, 1$			
	$p_3^*, 12$	$p_2, 11$	p_2	p_2, p_3
n_1	$p_2^*, 11$			
	$p_1^*, 10$			
	$p_2, 11$			
n_1	$p_1^*, 10$	$p_3, 8$	p_3	p_1
	$p_3, 3$			

Table 5. Scheduling steps for the example DAG in Figure 1.

cate on p_k, n_t is no longer required to start before its children on p_ℓ. Consequently, the latest start time of n_t on p_ℓ regardless of any children dependency, called $free\_lst$, is computed (step 19). If $free\_lst$ is smaller than the lst_level, p_ℓ can still be used to cover other processors, and hence p_ℓ, $free\_lst$ pair is inserted into $lstQ$ in step 21.

SCHEDULE function given in Algorithm 3 schedules the given task n_t to start at the provided lst value on the destination processor p_ℓ. Notice that another copy of task n_t may have already been scheduled on any of the processors, therefore this scheduling may result in duplication. If a duplicate of n_t already exists on p_ℓ, then it is removed since the new one can cover all the tasks that the existing one was covering. After duplication, all processors in set $\mathcal{C}$ are examined to see if they are also covered with the current duplication and the covered ones are removed from set $\mathcal{C}$.

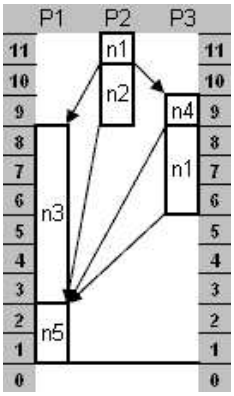

Figure 2. Schedule generated by the DBUS algorithm for the example in Figure 2.

Table 5 shows the scheduling steps for the example DAG in Figure 1. The identity of the tasks are given under column n_t in the order they are considered for scheduling. All p_ℓ, $cover\_lst$ pairs in the $lstQ$ are listed in the second column. Here, $cover\_lst$ is used to represent $cover\_lst(n_t, p_\ell, p_\ell)$. The processors marked with an asterisk superscript in this column are the ones in set $\mathcal{C}$ during the current iteration. While considering a pair in the $lstQ$, the corresponding p_k, min_cover_lst pair is given in the next column, whenever it is calculated. The destination processor that the task being considered is duplicated at the end of the iteration and the processors initially in set $\mathcal{C}$ that are covered by this duplication are given in the last two columns, respectively.

The algorithm starts with scheduling the exit task n_5 on p_1 which provides the shortest execution time. Then n_3 becomes the next task in $\mathcal{R}$ and corresponding $lstQ$ is constructed. n_3 is considered for duplication on p_1 since p_1 is on top of $lstQ$, and $p_1 \in \mathcal{C}$. Since min_cover_lst is greater than $cover\_lst$, n_3 is scheduled on p_1 to start at 8. Since p_1 is the only task in $\mathcal{C}$ and it is covered, no more duplication is considered for n_3. When scheduling n_2, again p_1 is on top of $lstQ$ initially. However, this time p_2 provides a min_cover_lst of 10, which is smaller than

$cover\_lst$. Therefore, n_2 is scheduled on p_2 to start at min_cover_lst, and it covers p_1. n_4 is scheduled on p_3 in a similar way to n_2. Note that by considering all processors rather than just the ones in set $\mathcal{C}$, DBUS algorithm was able to schedule tasks n_2 and n_4 with better start times that it would otherwise. Finally, when scheduling n_1, the set $\mathcal{C}$ consists of all three processors, since each of them has at least one of n_1's children already scheduled on it. First, p_3 with $cover\_lst$ of 12 is considered for scheduling n_3. However, since min_cover_lst provided by p_2 is smaller, n_1 is scheduled on p_2. This duplication helps covering p_2 as well as p_3. Since set $\mathcal{C}$ is still non-empty after this duplication, another duplication is considered for n_1. Note that since p_2 and p_3 are now covered, $free\_lst$ for each of them is calculated and inserted into $lstQ$. In the next iteration for n_1, p_2 is on top of the $lstQ$. However, since p_2 is no longer in set $\mathcal{C}$, it is skipped and the next pair, p_1, 10, in $lstQ$ is considered. This time, min_cover_lst provided by p_3 is smaller than $cover\_lst$, therefore n_1 is duplicated on p_3 to cover the last processor p_1 in $\mathcal{C}$. Please note that using task insertion and making processor p_3 available for future duplications after it had been covered allowed better utilization of idle slots. The resulting schedule is presented in Figure 2.

The time complexity of the DBUS algorithm is dominated by steps 12, 15 and 17 . Computation of $cover\_lst$ requires $O(|\mathcal{N}|)$ by properly storing information about which children of each task are duplicated on each processor. Thus, step 12 introduces a complexity of $O(|\mathcal{N}||\mathcal{P}|)$ which is also the complexity of SCHEDULE function at steps 15 and 17. Together with the **while** loop at step 9 and the **for** loop at step 1, the overall complexity of the DBUS algorithm is $O(|\mathcal{N}|^2|\mathcal{P}|^2)$.

4 Experimental Results

We evaluated the performance of the proposed DBUS algorithm on random DAGs as well as three application DAGs on random heterogeneous configurations. We generated random DAGs with three varying parameters. The first one is the average number of parents of a task to control dependencies in the DAG. Usually this parameter does not change significantly with problem size for DAGs of the same application. However, it may have different values for different applications. We used random DAGs to evaluate the effect of this parameter. The second parameter is *communication to computation ratio (CCR)*, which is defined as the ratio of average communication volume to average task execution weight. Here, task execution weight is the amount of unit computation to be carried out to completely execute a task. As the third parameter, we varied the number of tasks to see the impact of problem size on scheduling quality.

In random DAG experiments, the number of tasks is selected from set $\{50, 150, 250, 350, 450, 550\}$, CCR from set $\{0.1, 0.5, 1, 5, 10\}$ and average number of parents from set $\{4, 8, 12, 16, 20\}$. For a given number of tasks and average number of parents, first a random DAG topology is generated. Then each task is assigned an execution weight from interval $(0, 2 \times 10^7]$ with uniform probability. Finally, each edge is assigned a communication volume from interval $(0, 2 \times 10^7 \times CCR]$ to approximate the desired CCR.

We also tested the algorithms on tasks graphs from LU decomposition (LU), Laplace equation (LE) and Gaussian elimination (GE) applications. For these applications, the shape of the DAG is determined by the application. Therefore, we only investigated the effects of matrix size and CCR. The matrix size is chosen from set $\{5, 15, 25, 35, 45, 55\}$ while execution weights and communication volumes are generated similar to random DAG experiments.

We scheduled the generated DAGs on heterogeneous configurations with 16 processors unless specified otherwise. Heterogeneity of processors are simulated by randomly choosing the time required to complete a unit computation (t_c) for each task from interval $(0, 2 \times 10^{-7}]$ for each processor. Similarly, link heterogeneity is simulated by randomly selecting the time required to transfer a unit data (t_w) from interval $(0, 2 \times 10^{-7}]$ for each link. For each and every combination of parameter values and for each application, 30 DAGs are generated and average results are presented. While presenting a result for a varying parameter, the results are averaged over all tested values of the remaining parameters. Finally, the schedule lengths generated by each algorithm are normalized by that generated by HEFT.

The results on random DAGs (Figures 3 and 4) show that the DBUS algorithm generates the shortest schedules on the average among the four tested algorithms. The performance of DBUS is especially better than HEFT for small number of tasks and large CCR. The effect of CCR can be explained by the fact that task duplication is more useful when communication costs are relatively high. With the same reasoning, the performance of HCPFD and HCNF improves with increasing CCR compared to HEFT as well. However HCNF outperforms HEFT only when the number of tasks is smaller than 150 or CCR is greater than 7. As the number of parents increases, task insertion becomes less effective. The reason is that, greater number of dependencies impose a larger earliest finish time for each task and prohibits them to take advantage of idle slots. As shown in Figure 4, this leads to decreased performance gap between HEFT and HCNF. On the other hand, task duplication

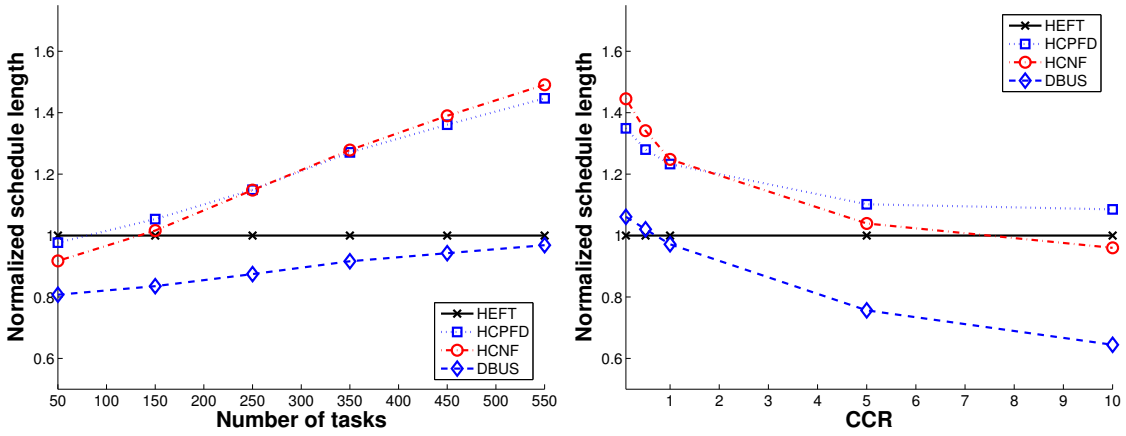

Figure 3. Normalized schedule length for random DAGs while varying (a) number of tasks (b) CCR.

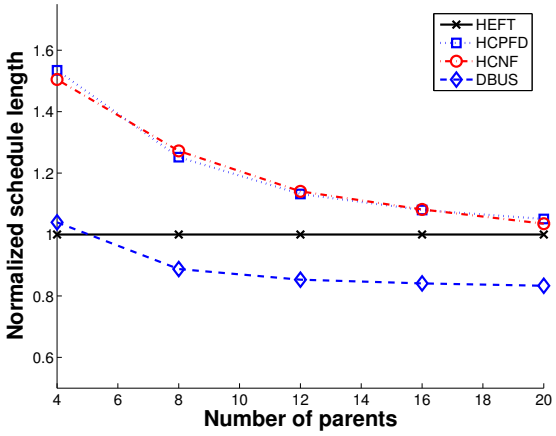

Figure 4. Normalized schedule length for random DAGs while varying number of parents.

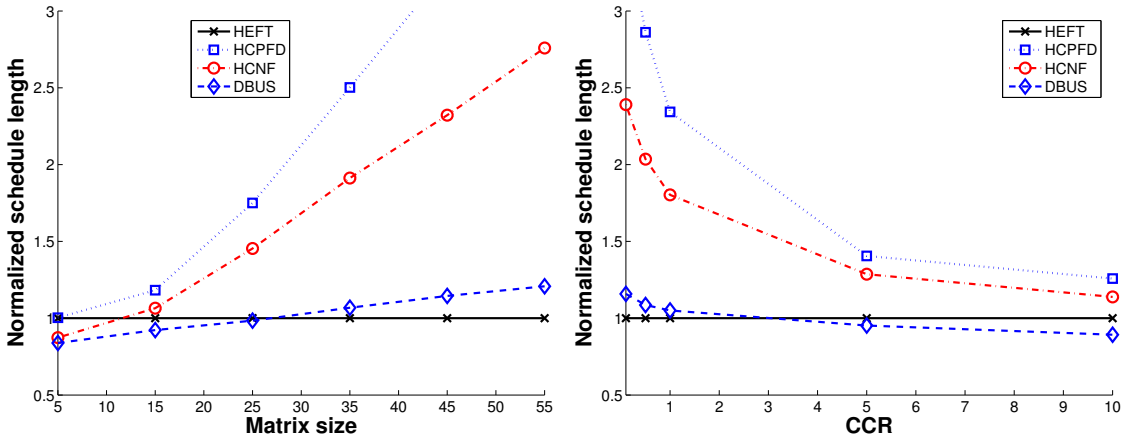

Figure 5. Normalized schedule length for Laplace Equation DAGs while varying (a) matrix size (b) CCR.

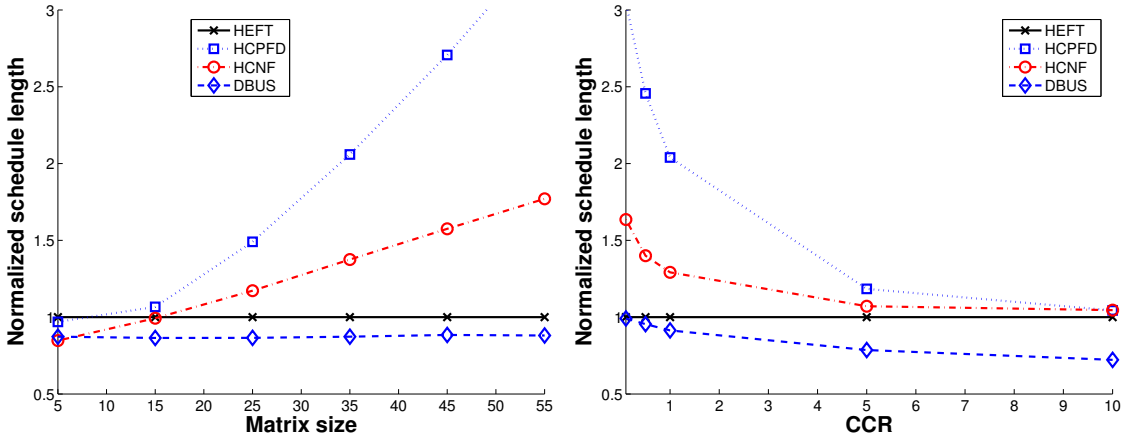

Figure 6. Normalized schedule length for LU Decomposition DAGs while varying (a) matrix size (b) CCR.

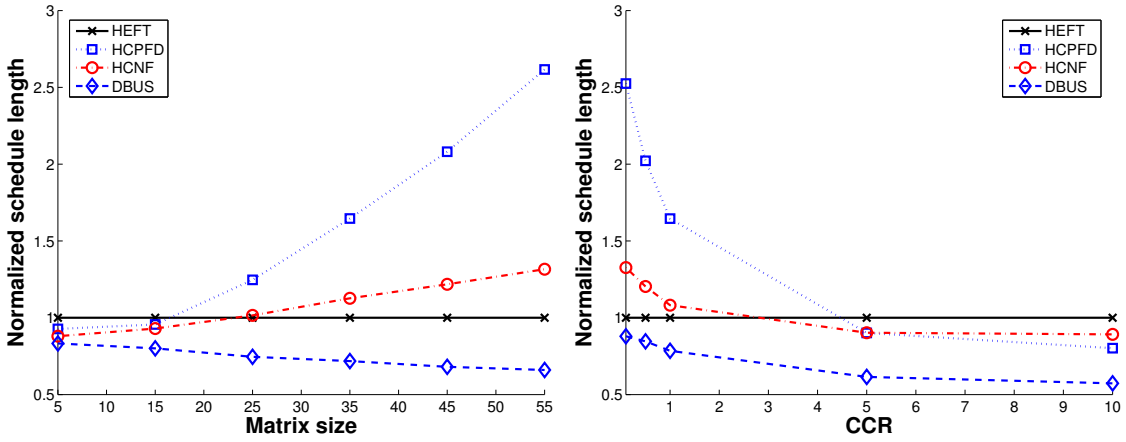

Figure 7. Normalized schedule length for Gaussian Elimination DAGs while varying (a) matrix size (b) CCR.

still proves to be effective with increasing dependencies, therefore DBUS performs better than HEFT as the number of parents increases.

For the application DAGs (Figures 5–7), DBUS significantly outperforms other algorithms except for LE task graphs where HEFT generates results with similar quality as DBUS. This can be explained by the the fact that tasks in LE graphs have at most two parents. Therefore, task duplication is not very useful except for large CCR. Similar to random DAG results, duplication based algorithms tend to improve their schedule quality with increasing CCR compared to HEFT. However, with increasing matrix size m, HCPFD and HCNF schedule qualities get worse quickly compared to HEFT and DBUS. The reason is that the number of tasks in these application graphs grow proportional to m^2, whereas average number of parents does not change significantly. Therefore with increasing m there are relatively more tasks independent of each other, which makes task insertion more effective. Consequently, as m increases non-insertion based algorithms suffer compared to insertion based ones. Furthermore, task duplication may start to be less effective than task insertion. Thus, utilizing some of the idle slots with duplication instead of task insertion may degrade the overall performance. This may explain why DBUS performs worse than HEFT when matrix size is large for the LE task graphs.

We also evaluated the impact of heterogeneity on scheduling quality. We define link heterogeneity as $\frac{max(t_w)}{min(t_w)}$ and processor heterogeneity as $\frac{max(t_c)}{min(t_c)}$. In order to obtain desired heterogeneity, t_c and t_w are chosen from interval $[1-x, 1+x]$ instead of the default interval $(0, 2\times10^{-7}]$. Therefore heterogeneity is equal to the ratio $\frac{1+x}{1-x}$. With appropriate values for x, we varied one type of heterogeneity from 1 to 200 while keeping the other at 200. For this experiment, we used the GE DAG with matrix size 55, since it was the largest application DAG we had generated. We generated weights corresponding to each CCR value, then averaged the results. Figure 8 shows that the effect of heterogeneity on DBUS is almost the same relative to HEFT and HCNF algorithms. In contrast, relative performance of HCPFD is adversely affected with increasing heterogeneity especially with processor heterogeneity.

As the final experiment, the effect of number of available processors on schedule length is investigated. We used the GE DAG with matrix size 55 for this experiment as well. Results in Figure 9 shows that the performance gap between DBUS and other algorithms first increases then decreases with increasing number of processors. The reason for decreasing performance gap is due to law of diminishing returns. Since DBUS generates a high quality schedule with smaller number of processors, it becomes more difficult to improve the schedule length even though the number of processors is increased. However, other algorithms have larger room for improvement, therefore they benefit from additional processors more than DBUS does. Still, DBUS performs significantly better than other algorithms in all cases.

5 Conclusions

In this work, we have developed a novel duplication-based bottom-up scheduling algorithm, called DBUS. DBUS is a list-based scheduling algorithm that traverses the DAG in a bottom-up fashion contrary to the traditional approaches and does not impose any restrictions on the number of task duplication. We tested the DBUS

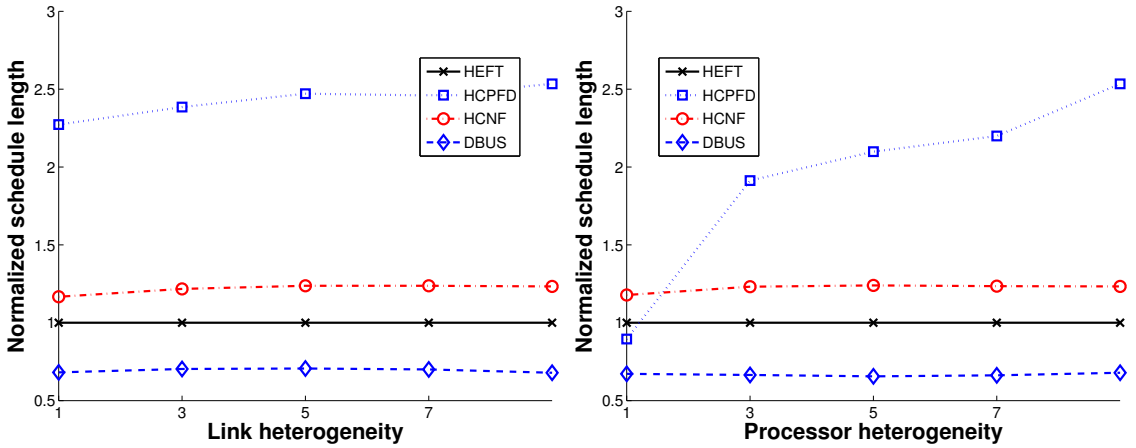

Figure 8. Normalized schedule length while varying heterogeneity for (a) links (b) processors.

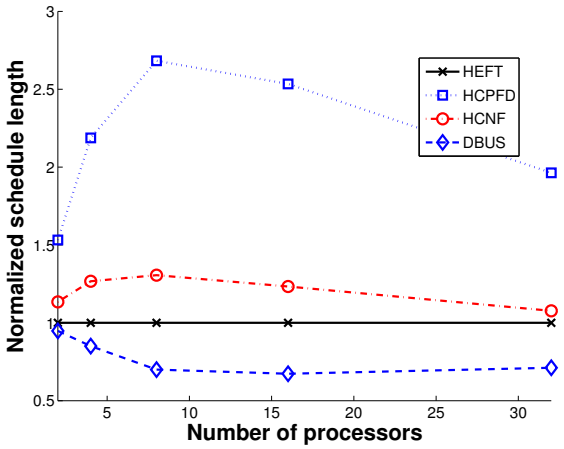

Figure 9. Normalized schedule length while varying the number of available processors for scheduling.

algorithm on combinations of three different application DAGs on randomly generated heterogeneous computing configurations. Experimental evaluation validated that DBUS produces superior results compared to the existing algorithms HEFT, HCPFD and HCNF.

References

[1] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, September 1998.

[2] R. Bajaj and D. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, February 2004.

[3] S. Bakiyar and P. SaiRanga. Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 97–103, October 2003.

[4] C. Boeres, J. Filho, and V. Rebello. A cluster-based strategy for scheduling task on heterogeneous processors. *Symposium on Computer Architecture and High Performance Computing*, pages 214–221, October 2004.

[5] D. Bozdağ, F. Özgüner, E. Ekici, and U. Catalyurek. A task duplication based scheduling algorithm using partial schedules. *Proceedings of International Conference on Parallel Processing*, pages 630–637, June 2005.

[6] T.-Y. Choe and C.-I. Park. A task duplication based scheduling algorithm with optimality condition in heterogeneous systems. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 531–536, August 2002.

[7] B. Cirou and E. Jeannot. Triplet: A clustering scheduling algorithm for heterogeneous systems. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 231–236, September 2001.

[8] A. Doğan and F. Özgüner. LDBS: A duplication based scheduling algorithm for heterogeneous computing systems. *Proceedings of the International Conference on Parallel Processing*, pages 352–359, August 2002.

[9] M. Garey and D. Johnson. *Computers and Intractability, A Guide to the Theory of NP Completeness*. W.H. Freeman and Co., 1979.

[10] T. Hagras and J. Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 107–115, April 2004.

[11] M. Iverson, F. Özgüner, and L. Potter. Statistical prediction of task execution times through analytical benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, December 1999.

[12] A. Radulescu and A. van Gemund. Fast and effective task scheduling in heterogeneous systems. *Proceedings of the Heterogeneous Computing Workshop*, pages 229–238, May 2000.

[13] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[14] J. Yang, A. Khokhar, S. Sheikh, and A. Ghafoor. Estimating execution time for parallel tasks in heterogeneous processing (hp) environment. *International Parallel Processing Symposium Workshop on Heterogeneous Computing*, pages 23–28, April 1994.

Biographies

Doruk Bozdağ is a graduate student in the Department of Electrical and Computer Engineering at The Ohio State University. His research interests include scheduling algorithms for multiprocessor systems, parallel graph algorithms and high-performance computing. He received his M.S. in Electrical and Computer Engineering from The Ohio State University in 2005 and B.S. in Electrical and Electronic Engineering and B.S. in Physics from Boğaziçi University, Turkey, in 2002.

Umit Catalyurek is an Assistant Professor in the Department of Biomedical Informatics at The Ohio State University, and has a joint faculty appointment in the Department of Electrical and Computer Engineering. His research interests include combinatorial scientific computing, grid computing, and runtime systems and algorithms for high-performance and data-intensive computing. He received his PhD, M.S. and B.S. in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.

Füsun Özgüner received the M.S. degree in electrical engineering from the Istanbul Technical University in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975. She worked at the I.B.M. T.J. Watson Research Center with the Design Automation group for one year and joined the faculty at the Department of Electrical Engineering, Istanbul Technical University in 1976. Since January 1981 she has been with The Ohio State University, where she is presently a Professor and the Interim Chair of Electrical and Computer Engineering. Her current research interests are parallel and fault-tolerant architectures, heterogeneous distributed computing, reconfiguration and communication in parallel architectures, real-time parallel computing and communication, and wireless networks. Dr. Özgüner has served as an associate editor of the IEEE Transactions on Computers and on program committees of several international conferences.