

Scalable Duplication Strategy with Bounded Availability of Processors

Guodong Li Youtao Zhang Yongjin Lin Yaochun Huang
Dept. of Computer Science, University of Texas at Dallas, Richardson 75080, USA
Email: magicflute@263.net

Abstract

In this paper we present a new task selection scheme and a list-scheduling algorithm for scheduling DAGs onto homogeneous/heterogeneous systems with bounded availability of processors. Specifically, forbidden duplications and superfluous duplications are eliminated in the task selection phase and the processor selection phase respectively. Simulation results show that the proposed algorithm outperforms other high performance algorithms when the availability of processors is restrained.

Keywords *task duplication, list scheduling, heterogeneous systems, mapping, multiprocessor*

1. Introduction

Duplication based allocation and scheduling is a relatively new approach for scheduling tasks onto multiprocessors. Task duplication based scheduling algorithms [3-16] duplicate certain tasks to minimize communication costs among dependent tasks. The idea is to map some tasks redundantly in trade of less inter-task communication overhead. However, the duplication based scheduling problem has been shown to be NP-complete [10]. Thus, many proposed algorithms are based on heuristics, and a large majority of them are only applicable to unbounded number of homogeneous processors. Although some algorithms take machine availability into consideration, they focus on extending the method for unbounded number of processors to the bounded case. So far there is no work concentrating on obtaining reasonable trade-off between duplications and bounded processors. Our tests show that, unfortunately, duplication without restriction will have great negative effect on the final schedule length. Hence, some duplication should be forbidden to minimize the overall finish time. Previous works pay little attention on this issue, and their performance cannot degrade gracefully when the number of available multiprocessors dwindles.

The task scheduling problem has been studied by many research groups for the heterogeneous systems [2, 7]. Most duplication-based algorithms for heterogeneous systems are extended from corresponding algorithms for homogeneous systems, and thus exhibit similar

characteristics. They face the same problem we mentioned.

This paper proposes a new task priority assignment strategy along with a duplication-based scheduling algorithm for both homogeneous and heterogeneous systems, with emphasis on bounded availability of processors. The contribution of our algorithm include:

- ✧ We point out that existing algorithms cannot adjust to the bounded case gracefully because they pay little attention to a fact: in order to save processor space for subsequent tasks or more important duplications, some duplication operations should be forbidden.
- ✧ We design a static priority assignment approach to eliminate forbidden duplications and superfluous duplications. It is superior to previous approaches because it fully considers the conflict between duplication and bounded availability of processors.
- ✧ We develop a list-scheduling algorithm incorporating our priority assignment approach and some methods adopted by cluster-based algorithms.

In sections 2-6 we give related works, the node selection scheme, the scheduling algorithm, some discussion and experimental results respectively.

2. Related Works

Duplication-Based algorithms can be classified into two categories in terms of the task duplication approach used [13]: Scheduling with Partial Duplication (SPD) and Scheduling with Full Duplication (SFD). SPD algorithms only duplicate limited number of parents of a node to maintain low complexity, while SFD algorithms attempt to duplicate all the parents of a join node to achieve minimal schedule length. In general, SFD algorithms have better performance than SPD algorithms, but their complexities are high.

Duplication-based algorithms can also be classified as list-scheduling algorithms and clustering algorithms. In list-scheduling algorithms, tasks are first assigned priorities according to their levels or ranks, and are put in a list in decreasing order of their priorities. They differ mainly in the way by which task priorities are assigned. Failure to schedule important tasks first may result in inefficient scheduling. In contrast, clustering-based

algorithms attempt to arrange tasks involving in heavy communication into the same cluster; and the ancestors of the task under consideration are absorbed into the cluster so that overall schedule length of the whole cluster is reduced. After clusters are constructed, one processor is assigned to each cluster. If no enough processors are available, some merging methods will be used to merge two or more clusters into one larger cluster. However, these merging methods are generally separated from the clustering methods. And, deciding which clusters should be merged and in which order the merging operations should be preformed is difficult. Thus, list-scheduling algorithms are more preferable for the case of bounded availability of processors.

Table 1 summarizes some renowned duplication based scheduling algorithms: LDBS (Levelized Duplication Based Scheduling)[3], SD (Selective Duplication)[4], ITDS (Improved TDS)[5], TCSD (Task Clustering and Scheduling based on Duplication)[6], BTDH (Bottom-Up Top-Down Duplication Heuristic)[7], DHS (Duplication Scheduling Heuristic) [8], PY [9], CPFD/ECPFD (Critical Path Fast Duplication)[10], TDS (Task Duplication based Scheduling)[11], DFRN (Duplication First and Reduction Next)[12], TCS (Task Clustering and Scheduling)[13], and LCTD (Linear Clustering with Task Duplication)[14]. In this table, we indicate the categories each algorithm belongs to.

In the simulation section we will compare the performance of our algorithm against those of ECPFD, LDBS, SD and TCSD. ECPFD and LDBS are the representatives of list scheduling algorithms for bounded processors. ECPFD assigns higher priorities to nodes on the critical path, and the duplication process is applied recursively to the very important parents. In addition, ECPFD determinates suitable processor and appropriate slot for a candidate node by testing all possible processors. Similarly, LDBS schedules tasks according to their levels or ranks. For a ready task, LDBS attempts to minimize its earliest start times by duplicating its critical immediate predecessor into an appropriate slot.

Table 1. Task Duplication Scheduling Algorithms

Algorithms	Characteristics
DHS	SFD, List-scheduling, Unbounded, Homogeneous
BTDH	SFD, List-scheduling, Unbounded, Homogeneous
SD	SFD, List-scheduling, Bounded, Homogeneous
ECPFD	SFD, List-scheduling, Bounded, Heterogeneous
DFRN	SPD, List-scheduling, Bounded, Homogeneous
LDBS	SPD, List-scheduling, Bounded, Heterogeneous
LCTD	SPD, Cluster-based, Bounded, Homogeneous
TDS	SPD, Cluster-based, Unbounded, Homogeneous
ITDS	SFD, Cluster-based, Unbounded, Homogeneous
PY	SFD, Cluster-based, Bounded, Homogeneous
TCS	SFD, Cluster-based, Bounded, Homogeneous
TCSD	SFD, Cluster-based, Bounded, Homogeneous

SD tends to avoid redundant duplications in some extend: it duplicates the nodes only if the performance can be improved. The task sequence generation strategy of SD is similar to that of ECPFD. And, in the processor selection phase, SD tries to duplicate the most important immediate parent of a node under consideration. As another cluster-based algorithm, TCSD is an extension of PY and TCS. It iteratively absorbs a critical node into a cluster, and recalculates the next critical path. The scheduling is constructed by visiting the task in reverse topological order. TCSD attempts to reduce the number of processors being consumed by deleting some redundant clusters.

2.1 Task Selection Scheme

In classical list scheduling strategies, priority strategies having been adopted so far include:

1. simple-level. Independent tasks are grouped into the same level. LDBS's first version and DERN schedule tasks in ascending order of simple levels.
2. s-level. It is the largest sum of computation costs along a path from the task to an exit task. DSH and BTDH schedule tasks in descending order of s-levels, while TDS in ascending order of s-level.
3. b-level. It is the largest sum of computation and communication costs along the longest path from current task to the exit task. LDBS's second version and HEFL[15] schedule tasks in descending order of s-level.
4. t-level. It is the largest sum of computation and communication costs along the longest path from the entry task to the task. MH [1] schedules tasks in decreasing order of t-level. In general, scheduling in descending order of b-level tends to schedule critical path nodes first while scheduling in ascending order of t-level tends to schedule nodes in a topological order [16].
5. critical path based. The relative importance of the tasks is in the following order: critical path tasks, in-branch tasks, out-branch tasks. ECPFD and SD schedule tasks according to this order.
6. combined-level. It is the mix of above levels. In CPOP [15] tasks are scheduled in descending order of the summation of b-levels and t-levels. In DPS [17], priorities are determined by the differences between b-levels and t-levels.
7. relative mobility. It involves the difference of the latest start times and earliest start times of the tasks. HRMS [18] uses relative mobility to identify free tasks to be executed.

Dynamic list scheduling approach, in which task priorities are determined after each scheduling step, may also be used. However, this approach is not suitable for communication-intensive and irregular DAGs (Directed Acyclic Graphs).

2.2 The Problem

Now we an example to illustrate how the conflict between duplication and bounded processors affects the overall schedule length.

Consider an example DAG depicted in Fig.1. Suppose that only two processors are available. The priority sequence with respect to critical path or relative mobility is $\{v_2, v_5, v_6, v_1, v_3, v_4, v_7\}$ (v_5 and v_6 are interchangeable, so do v_3 and v_4), and the corresponding scheduling is indicated in Fig.1.a. The priority sequence with respect to s-level or t-level or l-level or simple-level or combined-level approach is $\{v_2, v_1, v_5, v_6, v_3, v_4, v_7\}$ (or other similar sequences), and the corresponding scheduling is indicated in Fig.1.b. However, the scheduling with sequence $\{v_2, v_1, v_5, v_3, v_6, v_4, v_7\}$, as shown in Fig.1.c, has better schedule length. In fact, to achieve an optimal scheduling for this case, no duplication should be made.

These kinds of duplications, which should not be made, are called forbidden duplications. Another term, superfluous duplication [4], is referred to the duplication that doesn't contribute toward performance improvement. In other words, if a duplication is unable to reduce the overall schedule length, it is superfluous and should not be made for the sake of saving processor space. We will propose an approach to eliminate both forbidden duplications and superfluous duplication.

3. Task Selection Strategy

3.1. Model and Notations

A parallel program is usually represented by a Directed Acyclic Graph (DAG), which is defined by the tuple (V, E) , where V and E are the set of tasks (nodes) and the set of edges respectively. The edge $e_{i,j} \in E$ represents the precedence constraint between the task v_i and v_j . W is a $v \times q$ computation cost matrix in which each $\tau_{i,j}$ gives v_i 's computation cost on processor PE_j , here v and q are the number of nodes and number of processors respectively.

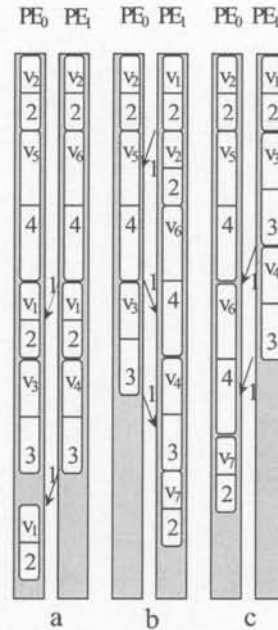
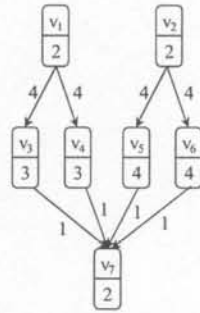


Fig.1 An example DAG and its scheduling

D is a $v \times v$ matrix of communication data, where $d_{i,k}$ is the amount of data required to be transmitted from task v_i to task v_k . The data transfer rates between processors are stored in matrix B of size $q \times q$.

The average computation cost of task v_i is defined as:

$$\bar{w}_i = \sum_{1 \leq j \leq q} \frac{w_{i,j}}{q}$$

The communication cost of the edge $e_{i,k}$, which is for transferring data from task v_i on PE_m to task v_k on PE_n , is defined by

$$c_{i,k} = \text{data}_{i,k} / B_{m,n}$$

The average communication cost of an edge $e_{i,k}$ is defined by $\bar{c}_{i,k} = \text{data}_{i,k} / B'$ where B' is the average transfer rate among all processors.

For homogeneous systems, these denotations could be simplified such that τ_i represents the computation cost for task v_i and $c_{i,j}$ is the communication cost for edge $e_{i,j}$.

When two tasks, v_i and v_j , are assigned to the same processor, $c_{i,j}$ is assumed to be zero. The term iparent is used to represent immediate parent. The earliest start time, est_i , and the earliest completion time, ect_i , are the earliest times that a task v_i starts and finishes its execution respectively. $est(i, m)$ and $ect(i, m)$ are the earliest start time and earliest finish time that task v_i starts and finishes execution on PE_m respectively. For simplification purpose, we define some terms for homogeneous systems here; they can be easily extended to heterogeneous systems. A message arriving time from v_j to v_i , $mat_{j,i}$, is the time that the message from v_j arrives at v_i . If v_i and v_j are scheduled on the same processor, $mat_{j,i}$ becomes ect_j ; otherwise, $mat_{j,i} = ect_j + c_{j,i}$. For a join node v_i , its arriving time $mat_i = \max \{mat_{j,i} \mid v_j \text{ is } v_i\text{'s iparent}\}$. In addition, its critical iparent, which is denoted as $CIP(v_i)$, provides the largest mat to the join node. That is, $v_j = CIP(v_i)$ if and only if $mat_{j,i} \geq mat_{k,i}$ for all k where v_k is the iparent of v_i and $k \neq j$ (if multiple nodes satisfy this constraint, arbitrarily select one). The critical iparent of an entry node is defined to be NULL. After all tasks in a graph are scheduled, the schedule length, also called makespan, is the largest finish time of exit tasks. The objective of the scheduling problem is to determine the assignment and scheduling of tasks such that minimal schedule length is obtained.

3.2 Task Priority Assignment

We begin with the case of homogeneous systems, and then extend it to the case of heterogeneous systems.

Consider the simple DAG in Fig.1, v_3 and v_4 conflict with v_2 's duplication, and v_5 and v_6 conflict with v_1 's duplication. Duplicating v_1 or v_2 may have negative effect

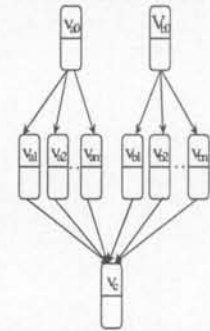


Fig 2. A simple DAG prone to generating forbidden dup.

on the overall schedule length. Existing duplication-based algorithms fail to make correct decision on whether v_1 or v_2 should be duplicated or not. In fact, they cannot discriminate v_3 from v_4 , and v_5 from v_6 . Consequently, v_3 and v_4 are always scheduled together or consecutively, so do v_5 and v_6 . However, to obtain an optimal scheduling, v_3 and v_4 should possess different priorities. Priorities should be assigned in a way that v_5 is scheduled first, then v_3 , then v_6 , and finally v_7 .

There are two intuitive methods to eliminate forbidden duplications. The first one is to examine whether these duplications exist and remove them from the processors. However, identifying them in an existing scheduling is not an easy job because this kind of examination operation will involve removing each suspected duplication tentatively and checking whether the overall schedule length is improved. Furthermore, removing duplications from an existing scheduling is complicated and time-consuming. Another intuitive method is to predict the effect of a duplication: if it will increase the final schedule length due to its conflict with subsequent tasks, then it is a forbidden duplication. However, it is very difficult to make accurate prediction without generating the whole scheduling. This method is even more complicated than the first one.

Now we propose a strategy to solve this problem. Since a forbidden duplication may force its conflicting tasks to start execution late, we could schedule these tasks, by assigning higher priorities to them, before corresponding forbidden duplications are made. In this way, no removing or prediction is needed.

Consider the DAG depicted in Fig.2, supposed that $v_{a1}, v_{a2}, \dots, v_{ai}, i < m$ and $v_{b1}, v_{b2}, \dots, v_{bj}, j < n$ have been scheduled, while $v_{a(i+1)}, \dots, v_{am}$ and $v_{b(j+1)}, \dots, v_{bn}$ are to be scheduled. Note that scheduling v_{a1}, \dots, v_{am} and v_{b1}, \dots, v_{bn} involve the duplication of v_{a0} or v_{b0} respectively. Thus $v_{a(i+1)}, \dots, v_{am}$ and $v_{b(j+1)}, \dots, v_{bn}$ conflict with the duplication of v_{a0} or v_{b0} respectively. The next task should be scheduled (that is, the task with highest priority), e.g. v_x , is determined by:

$$\begin{aligned} & \text{if } \tau_{a(i+1)} + \dots + \tau_{am} > \tau_{b(j+1)} + \dots + \tau_{bn} \text{ then } x = k \mid \tau_{ak} = \\ & \max \{ \tau_{a(i+1)}, \dots, \tau_{am} \}; \\ & \text{if } \tau_{a(i+1)} + \dots + \tau_{am} < \tau_{b(j+1)} + \dots + \tau_{bn} \text{ then } x = k \mid \tau_{bk} = \\ & \max \{ \tau_{b(j+1)}, \dots, \tau_{bn} \}. \\ & \text{otherwise, } x = k \mid \tau_k = \min \{ \tau_{a(i+1)}, \dots, \tau_{am}, \tau_{b(j+1)}, \dots, \\ & \tau_{bn} \} \text{ or } \max \{ \tau_{a(i+1)}, \dots, \tau_{am}, \tau_{b(j+1)}, \dots, \tau_{bn} \}; \end{aligned}$$

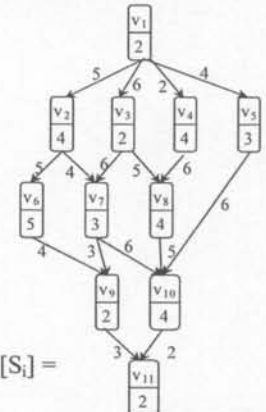
For instance, for the DAG in Fig.1, initially $\tau_3 + \tau_4 = 6 < \tau_5 + \tau_6 = 8$, thus v_5 is selected for scheduling; then because $\tau_3 + \tau_4 = 6 > \tau_6 = 4$, v_3 is selected; and then because $\tau_4 = 3 < \tau_6 = 4$, the turn is on v_6 , and finally on v_4 . This scheduling is optimal for the case of two processors. Thus, v_5 's priority should be larger than v_3 's, and v_3 's priority larger than v_6 's. The key idea is to assign higher priorities to those tasks that will have more severe negative influence on the overall schedule length.

Now we introduce some concepts. Set S_i consists of all v_i 's descendents (including v_i); Set T_i consists of all v_i 's descendents and the edges connecting them (including v_i). $[S_i]$ represents the sum of computation costs of all nodes in S_i , $[T_i]$ represents the sum of computation cost of all nodes and all communication costs of all edges in S_i . We call $[S_i]$ the accumulative computation costs of v_i , $[T_i]$ the accumulative computation and communication costs of v_i . And, c_rank_i and b_rank_i , both integers, are the priorities to be assigned to task v_i . The operator $[]$ represent the computation/communication costs of all elements in the set it parenthesizes.

Let $j \rightarrow i$ denote that v_j is a descendent of v_i . We define that $v_i \rightarrow v_j$. Let $p(i)$ denote the index of v_i 's parent.

$$\begin{aligned} S_i &= \bigcup_{j \rightarrow i} v_j; [S_i] = \sum_{j \rightarrow i} \tau_j; \\ T_i &= \bigcup_{j \rightarrow i} v_j \cup \bigcup_{j \rightarrow i \wedge k \rightarrow i \wedge e_{j,k} \in E} e_{j,k}; \\ [T_i] &= \sum_{j \rightarrow i} \tau_j + \sum_{j \rightarrow i \wedge k \rightarrow i \wedge e_{j,k} \in E} c_{j,k}; \\ c_rank_i &= [\bigcup_{p(i)=p(i) \wedge [S_j] < [S_i]} S_j]; \\ b_rank_i &= [\bigcup_{p(i)=p(i) \wedge [T_j] < [T_i]} T_j]; \end{aligned}$$

If v_i is an exit node, $S_i = T_i = \{v_i\}$, $[S_i] = [T_i] = c_rank_i = b_rank_i = \tau_i$.



Priorities of Tasks				
Node	[S]	[T]	c_rank	b_rank
1	35	104	35	104
2	20	47	33	72
3	17	47	24	85
4	14	27	17	33
5	9	17	9	17
6	9	16	9	16
7	11	25	13	34
8	10	17	10	17
9	4	7	4	7
10	6	8	8	8
11	2	2	2	2

Fig.3 An example DAG and its nodes' ranks

To understand the meaning of c_rank , consider a node v_i . Suppose $v_1, \dots, v_k, \dots, v_m$ is a permutation of v_i 's siblings such that $[S_1] < \dots < [S_k] < [S_i] < [S_{k+1}] < \dots < [S_m]$ (if two siblings, e.g. v_i' and v_j' , satisfy $[S_i] = [S_j]$, we will break the tie by defining $[S_i] < [S_j]$ if $i < j$ and $[S_j] < [S_i]$ if $i > j$. We may also use a tuple $[[S_i], i]$ for comparison), then v_i 's rank with respect to computation cost, i.e. c_rank_i , is given by

$$c_rank_i = [\bigcup_{j \rightarrow i} v_j \cup \bigcup_{1 \leq k < i} v_k]$$

In other words, c_rank_i is the maximum sum of the computation costs of the descendents of v_i and v_i 's siblings whose accumulative computation costs are less than v_i 's. Similar analysis can be applied to b_rank_i , in which communication costs are taken into account. Either c_rank or b_rank can be used as the task's priority.

For example, for the DAG depicted in Fig.1, $[S_3] = [S_4] = 5$, but $[[S_3], 3] < [[S_4], 4]$, thus $c_rank_3 = [v_7 \cup v_3] = \tau_7 + \tau_3 = 5$, $c_rank_4 = [v_7 \cup v_3 \cup v_4] = \tau_7 + \tau_3 + \tau_4 = 8$. Following similar procedure, we have $c_rank_5 = 6$, $c_rank_6 = 10$. The task sequence is $\{v_1, v_6, v_4, v_5, v_3, v_7\}$.

It is optimal for the case of two homogeneous processors.

Fig.3 gives an example DAG and its relative parameters. The task sequences in descending order of c_ranks and b_ranks are $\{1, 2, 3, 4, 7, 8, 5, 6, 10, 9, 11\}$ and $\{1, 3, 2, 7, 4, 5, 8, 6, 10, 9, 11\}$ respectively.

For heterogeneous systems, we use average task computation costs and average communication costs to calculate the priorities of all tasks. What we need to do is replace the τ_i and $c_{i,k}$ with $\bar{\tau}_i$ and $\bar{c}_{i,k}$ respectively, and then follow the same procedure as we do for homogeneous systems.

4. Scalable Duplication-Based Scheduling

Algorithm (SDS)

SDS is a list scheduling algorithm that uses an insertion-based duplication approach to assign a task to one of the processors to minimize this task's earliest time. Task v_i can start execution on processor PE_m if and only if it have received all data from its parents. At first, tasks are assigned priorities according to their c_ranks or b_ranks , and then the task with highest priority is considered to execute. If an idle processor is available, it will be tested. When more than one processors return the same value of ect_i , then SDS is in favor of the processor involving less duplication cost. This operation is to eliminate superfluous duplications.

SDS Algorithm .

Begin

Compute c_ranks (or b_ranks) for all tasks by traversing the DAG upward, starting from the exit task;

Sort the tasks in a scheduling list in increasing order of c_rank (or b_rank) values, breaking tie by in favor of the one with less computation cost;

Set processor_set to be empty;

while there are unscheduled tasks in the list **do**

 Select the task, e.g. n_i , from the head of the list for scheduling.

if there is an idle processor, e.g. PE_m , **then** processor_set = processor_set \cup $\{PE_m\}$;

endif

for each processor PE_m in the processor_set **do**

 call minimize_est(v_i , m), record the return value ect_i and the duplication cost;

endfor

 Assign task n_i to the processor PE_j that minimizes ect_i , breaking tie by choosing the one with less duplication cost.

endwhile

End

Procedure minimize_est(node v_i , processor m) attempts to minimize v_i 's earliest start time on processor PE_m by duplicating v_i 's ancestors which locate on critical edges. We use the concept of critical edges introduced in a cluster-based algorithm [6]. Now we can assume that v_i 's ancestors already on processor PE_m constitutes a cluster $C[i,m]$. At each step, the edge that crosses $C[i,m]$ and locates on the critical path of current configuration is

called the current critical edge of $C[i,m]$. Namely, the critical edge is the edge from which the current CIP emanates. Suppose $e_{k,l}$ is the current critical edge, then duplicating v_k on processor PE_m may decrease the value of ect_i . This procedure repeatedly identifies the new critical edge and tries to duplicate the node from which the critical edge emanates on processor PE_m until est_i cannot be improved any more.

Procedure minimize_est(node v_i , processor m)

Begin

Find first suitable slot $[v_j, v_{j+1}]$ on m for v_i such that $est_{j+1} - \max\{ect(j, m), mat_i\} \geq \tau_i$, record the $ect(i, m)$ as $ect'(i, m)$;

while (**true**) **do**

 Identify the current critical edge of $C[i, m]$, e.g. $e_{k,j}$;

if no critical edge exists **then**

 Eliminate superfluous duplications;

return $ect'(i, m)$ and the duplication cost;

 Find first suitable slot $[v_j, v_{j+1}]$ for v_k on m ;

if suitable slot exists **then**

 Duplicate v_k in the slot on m ;

 Add τ_k into the duplication cost;

 Temporarily update the value of $ect(i, m)$;

else

 Eliminate superfluous duplications;

return $ect'(i, m)$ and the duplication cost;

endif

if $ect(i, m) > ect'(i, m)$ **then**

 Undo the duplication of v_k on m ;

 Eliminate superfluous duplications;

return $ect'(i, m)$ and the duplication cost;

else

$ect'(i, m) = ect(i, m)$;

endif

endwhile

end

Procedure minimize_est() performs the operation of eliminating superfluous duplications before it returns. A duplication is termed as superfluous if the data arrival time of the node under consideration (the duplication's child or descendent), with and without duplication, turned out to be the same. This operation is done by simply checking whether the duplications made in this call can be removed without leading to the increase of ect_i .

Fig.4 (at the last page) gives the scheduling trace and the scheduling result for the DAG depicted in Fig.3. Tasks are scheduled in descending order of b_rank . Note that assigning v_7 to PE_0 , PE_1 or PE_2 will lead to the same value of ect_7 , but the duplication cost on PE_1 is smallest (no duplication is made), so PE_1 is selected. Also note that v_6 's finish time, with or without v_2 's duplication on PE_0 , is the same, so v_2 's duplication is a superfluous and shouldn't be made.

The time complexity of SDS is determined as follows. The calculation of c_ranks or b_ranks need $O(v_e)$ time at most where v and e are the number of tasks and edges respectively. We need to call minimize_est() $O(v)$ times for scheduling each task because we have to test

each processor in the processor_set; when extremely bounded number of processors are provided, this procedure will be called $O(1)$ times. The time complexity of minimize_est() is $O(e)$ because we need to find the critical edges at each step. Thus the overall time complexity is $O(v^2e)$.

5. Discussion

Some other algorithms, such as ECPFD, TCSD and SD, make certain effort to reduce the number of processors consumed, which in turn may alleviate the conflict between duplication and bounded space. For example, ECPFD refrains from duplicating out-branch nodes initially to spare some space; TCSD deletes some superfluous clusters to avoid consuming new processors. In addition, SD notices that in many duplication algorithms, some duplication is not useful and should be forbidden. However, for the case of bounded availability of processors, these algorithms fail to eliminate a duplication which may increase the schedule length of current configuration but will eventually reduce the overall schedule length, thus they do little to the problem.

In the following, we present some optimality result for the case of unbounded availability of homogeneous systems to show that SDS maintains good characteristics as other high-performance duplication based algorithms such as TCSD do (they use the same duplication strategy on the processor selection phase). Due to space constraint, we don't give the proofs here.

Theorem 1. For out-tree DAGs, SDS generates an optimal scheduling for unbounded homogeneous processors.

Theorem 2. For fork-join DAGs (diamond DAGs), SDS generates an optimal scheduling for unbounded homogeneous processors.

Theorem 3. Assuming unbounded homogeneous processors, for arbitrary DAGs, the schedules generated by SDS are at most twice as the optimal ones. Moreover, if the granularity of DAG is larger than $(1-\epsilon)/\epsilon$ for $0 < \epsilon \leq 1$, the schedule length generated is at most $(1+\epsilon)$ times as the optimal one.

6. Performance Evaluation

In this section we compare the performance of SDS with four high-performance duplication-based scheduling algorithms, i.e. CPFD/ECPFD, TCSD, SD, and LDBS. These algorithms have been proven to be superior to some other algorithms including HEFL, DLS, TDS, TDS, TCS, PY, etc. in many cases [3][6][10].

These four algorithms along with SDS are applied to diverse sets of application with varying characteristics. For homogeneous systems, TCSD, SD, CPFD and SDS are tested in the case of both bounded and unbounded availabilities of processors; for heterogeneous systems, two versions of LDBS and SDS are tested in the case of

bounded availability of processors.

The reason why we test our algorithms on random DAGs is two-fold: (1) Random DAGs are valid representatives of all kinds of DAGs appearing in real time applications. Thus our simulation doesn't bias on any specific application. (2) Most previous works conduct extensive simulation on random DAGs, and obtain favorite results. Operating on the same data set enable us to compare SDS with them.

We begin with the homogeneous case. The input DAGs are generated randomly with the numbers of tasks ranging from 100 to 800 nodes and the numbers of edges varying from 1,00 to 2,000. Additionally, the number of predecessors and successors varies from one to 50, and the computation costs vary from 10 to 1,000 time units. The CCR (communication to computation ratio) of a DAG is defined as its average communication cost divided by its average computation cost. In these tests the CCR values used are 0.1, 1.0, 5.0, and 10.0.

We generate 3000 random DAGs for testing. The first comparison is to compare the schedule lengths of three algorithms with SDS in terms of the normalized schedule lengths (NSLs), which is obtained by dividing the output schedule length by the sum of computation costs on the critical path [2]. Table 2 shows the average NSLs produced by SDS, CPFD, SD and TCSD. Note that TCSD is a cluster-based SFD algorithm, it has shown to outperform other algorithms including TDS, CPFD, PY and TCS in the case of unbounded availability of homogeneous processors. Simulation results shows that SDS is slightly inferior to TCSD and CPFD in this case, this can be ascribed to the possibility that SDS schedules some relative unimportant tasks first. However, when the number of available processors decreases, SDS gain advantages in terms of performance over both TCSD and SD more significantly because SDS takes extra effort to eliminate forbidden and superfluous duplications.

Table 2. Average NSLs of four algorithms for random DAGs with various numbers of nodes when unbounded homogeneous processors are available

Algorithms	Number of Nodes				
	100	200	300	400	Avg.
SDS	1.51	1.52	1.63	1.74	1.58
CPFD	1.51	1.49	1.61	1.66	1.57
SD	2.03	2.22	2.44	2.55	2.32
TCSD	1.46	1.49	1.54	1.56	1.52

Table 3 and table 4 compare the performance degradation of four algorithms in terms of average NSLs with respect to the number of processors available. The numbers of nodes in test DAGs are 300 or 600 and the CCRs vary from 0.1 to 10. These data show that our algorithm achieves comparable performance to that in the case of unbounded availability of processors. And, compared with three other algorithms, the performance of SDS is less sensitive to the number of available processors. Table 5 shows the performance degradation

of SDS with respect to the percentage of the number available processors to that of required processors. The performance degradation becomes significant as the number of processors is decreased to be around or less than 20%.

Table 3. Average NSLs of four algorithms for random DAGs with 300 nodes with respect to the number of available homogeneous processors

Algorithms	Number of Processors				
	20	40	60	80	Avg.
SDS	1.86	1.72	1.70	1.66	1.74
ECPFD	1.93	1.77	1.69	1.61	1.75
SD	2.76	2.58	2.47	2.38	2.55
TCSD	2.04	1.85	1.72	1.69	1.82

Table 4. Average NSLs of four algorithms for random DAGs with 800 nodes with respect to the number of available homogeneous processors

Algorithm ms	Number of Processors				
	20	40	60	80	Avg.
SDS	2.01	1.85	1.76	1.72	1.82
ECPFD	2.13	1.97	1.83	1.75	1.92
SD	3.05	2.82	2.68	2.49	2.76
TCSD	2.73	2.45	2.14	1.86	2.30

Table 5. Performance degradation of SDS with respect to the number of required processors

Number of nodes	Percentage of the number available processors to that of required processors			
	80%	60%	40%	20%
200	1.07	1.18	1.32	1.65
400	1.09	1.24	1.48	1.88
600	1.13	1.31	1.55	1.96
800	1.16	1.35	1.68	2.06

Algorithms are generally more sensitive to the value of CCR than to the number of nodes. Table 6 shows the ratio of schedule length generated by ECPFD, SD and TCSD over that of SDS provided that only 50% of required processors for each algorithm are provided. It may be noted that differences between the performances of various algorithms become more significant with larger value of CCR.

Table 6. Ratio of schedule lengths generated by three algorithms over that of SDS

Algorithms	CCR				Avg.
	0.1	1	5	10	
ECPFD	1.01	1.05	1.12	1.14	1.08
SD	1.44	1.49	1.54	1.62	1.52
TCSD	1.26	1.28	1.29	1.33	1.29

Among duplication-based algorithms, SPD algorithms duplicate limited number of parents of a node to achieve low complexity, the problem of forbidden duplications in these algorithms are not as severe as in SFD algorithms. However, the performances of SPD algorithms are not favorable. Unfortunately, cluster-based

SFD algorithms such as TCSD also behavior bad in the case of constrained availability of processors. In general, SFD list scheduling algorithms are more adaptable to this case than cluster-based algorithms do. For example, ECPFD and SD can achieve satisfactory schedule length for various kinds of DAGs. However, SD algorithm only check current node's most important immediate parents, thus suffers from the same problems as SPD algorithms. In order to minimize the start time of current node, ECPFD resorts to minimize the start time of its CIP by duplicating the parents of this CIP on the same processor, this operation duplicates as many ancestors as possible on the current processor while the earliest start time of current node will not increase if this time depends on the new CIP (after the original CIP has been duplicated). Moreover, this unrestricted duplication operation will lead to a certain extend of performance degradation when the number of available processors is limited. In contrast, identifying the new critical edge and duplicating the new CIP at each step, as SDS does, will be more advantageous. The simulation results verify this conclusion.

Now we look at the simulation results for heterogeneous systems. A heterogeneous system is randomly created based on the number of processors q . We use a random graph generator similar to that in [2] to control the range percentage of computation costs on processors and communication costs on each link. The rest data in matrix W , D and B are created randomly. Graph sizes vary from 40 to 200 nodes, and the CCR is 0.1, 1, 5, or 10. The normalized schedule length in this case is obtained by dividing the output schedule length by the sum of minimum computation costs of the nodes on the critical path [6]. The performance of SDS is compared with against LDBS because LDBS has been shown to outperform several other efficient algorithms such as HEFL. Fig.5 shows the results when the number of nodes is from 40 to 200 and the CCR is from 0.1 to 10. The average NSLs of SDS is roughly 15% on average smaller than that of LDBS. In fact, LDBS's duplication strategy is similar to that of CPFD that always tries to minimize the start time of the CIP of the node under consideration. As we have mentioned, this strategy is inefficient under the circumstance of bounded availability of processors.

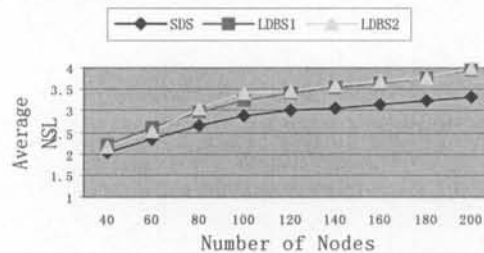


Fig.5. Average NSLs of SDS and LDBS for Heterogeneous systems

In general we use *b_ranks* as tasks' priorities. When the CCR is large, more duplications could be made to reduce a node's earliest start time, in this case we can *c_ranks* instead of *b_ranks* as tasks' priorities.

7. Conclusion

In this paper we present a new task selection scheme along with a scheduling algorithm SDS for scheduling DAGs onto a system of homogeneous or heterogeneous processors. The objective of the task selection strategy is to eliminate forbidden duplications. In addition, superfluous duplications are eliminated during the scheduling process. Simulation results show that SDS outperforms other algorithms when only bounded number of processors is available. Based on the performance evaluation result, SDS is a viable solution for DAG scheduling problem in this case.

References

- [1] B.P. Gan; S.Y. Huang. "The modified mapping heuristic algorithm", Second International Conference on Algorithms and Architectures for Parallel Processing, 1996.
- [2] H. Topcuoglu, S. Hariri, M.Y.Wu. "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing", IEEE Trans. Parallel and Distributed Systems, 13(3), 260-274, Mar. 2002.
- [3] A. Dogan, F. Ozguner. "LDBS: A Duplication Based Scheduling Algorithm for Heterogeneous Computing Systems", Proceeding of the International Conference on Parallel Processing, 2002
- [4] S. Bansal, P. Kumar, K. Singh. "An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems", IEEE Trans. Parallel and Distributed Systems, 14(6), 533-544, Jan. 2003.
- [5] C.I. Park, T.Y. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication", IEEE Trans. Computers, 51(4), 444-448, 2002.
- [6] G.D. Li, D.X. Chen, D.M. Huang, D.F. Zhang. "Task Clustering and Scheduling to Multiprocessors with Duplication", 17th International Parallel and Distributed Processing Symposium, 2003.
- [7] Y.C.Chung, S.Ranka. "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors", Proc. Supercomputing '92, 1992.
- [8] B.Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing", IEEE Software, 23-32, Jan.1988.
- [9] C. Papadimitriou and M. Yannakakis, "Toward an Architecture Independent Analysis of Parallel Algorithms, SIAM J.Computing, 19, 322-328, 1990.
- [10] I. Ahmad, Y.K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling", IEEE Trans. Parallel and Distributed Systems, 9(9), 872-892, Sep. 1998.
- [11] S. Darbha, D.P. Agrawal. "Optimal Scheduling Algorithm for Distributed-Memory Machines", IEEE Trans. Parallel and Distributed Systems, 9(1), 97-95, Jan. 1998.
- [12] G.L. Park, B. Shirazi, J. Marquis. "Mapping of Parallel

- Tasks to Multiprocessors with Duplication", Proc. of the 12th International Parallel Processing Symposium, 1998.
- [13] M.A. Palis, J.C. Liou, D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures", IEEE Trans. Parallel and Distributed Systems, 7(1), 46-55, Jan. 1996.
- [14] B. Shirazi, H. Chen, J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques", Concurrency: Practice and Experience, 7, 371-389, Aug. 1995.
- [15] Topcuoglu, S.Hariri, M.Y. Wu. "Task scheduling algorithms for heterogeneous processors". In IPPS/SPDP Workshop on Heterogeneous Computing, San Francisco, CA, 2001.
- [16] A. Gerasoulis, T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors" Journal of Parallel and Distributed Computing, 16(4), 276-291, 1992.
- [17] I. Ahmad, M.K. Dhodhi, R. Ul-Mustafa. "DPS: dynamic priority scheduling heuristic for heterogeneous computing systems", IEE Proceedings on Computers and Digital Techniques, 145(6), 411-418, 1998.
- [18] W.Y. Chan; C.K. Li. "Scheduling tasks in DAG to heterogeneous processor system", Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998.

Scheduling Trace of SDS					
Ste	Node	PE0	PE1	PE2	N. Dup.
1	1	2*	-	-	-
2	3	4*	4	-	-
3	2	8	6*	-	1
4	7	11	11*	11	3
5	4	8	15	6*	1
6	5	7*	14	9	-
7	8	15	16	12*	3
8	6	16	16*	17	-
9	10	20	21	19*	7
10	9	18*	19	22	6
11	11	23*	23	23	-

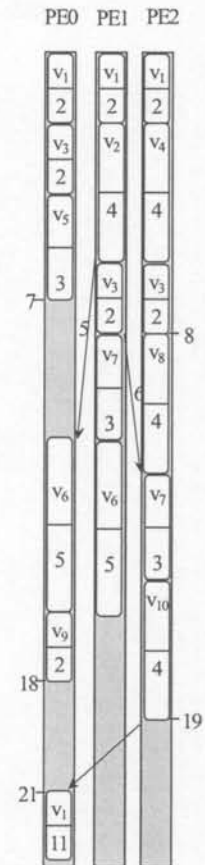


Fig 4. Running trace and result for the DAG in Fig.3