A Task Duplication Based Scheduling Algorithm Using Partial Schedules*

Doruk Bozdağ†, Füsun Özgüner†, Eylem Ekici†, Umit Catalyurek‡
† Department of Electrical and Computer Engineering
‡ Department of Biomedical Informatics
The Ohio State University
{bozdagd,ozguner,ekici}@ece.osu.edu, umit@bmi.osu.edu

Abstract

We propose a novel replication-based two-phase scheduling algorithm designed to achieve DAG scheduling with small makespans and high efficiency. In the first phase, the schedule length of the application is minimized using a novel approach that utilizes partial schedules. *In the second phase, the number of processors required is minimized by eliminating and merging these partial schedules. Experimental results on random DAGs show that the makespans generated by the proposed algorithm are slightly better than those generated by the well known CPFD algorithm whereas the number of processors used is less than half of what is needed by CPFD solutions.*

1 Introduction

Low cost, high performance commodity computing and network hardware turned the distributed memory multiprocessor systems (DMMS) into viable alternatives to the traditional supercomputers. In order to achieve high performance with a DMMS, efficient task partitioning of the application and scheduling of those tasks onto parallel processors is of utmost importance. Assuming that the application has already been partitioned into tasks and the task dependencies are represented as a *directed acyclic graph (DAG)*, the goal of scheduling is to minimize the execution time, also referred to as *schedule length* or *makespan*, of the application by allocating tasks on DMMS such that the precedence constraints are preserved [2]. DAG scheduling problem is shown to be an NP-complete problem in its general form [7]. Optimal polynomial solutions exist only for limited cases where either the cost or the shape of the DAG is restricted [6, 14]. In general, proposed heuristic algorithms offer trade-offs between the quality and the complexity of the solutions. Besides the schedule length, various cost criteria such as the number of processors used are also taken into account in many cases.

Task duplication is a relatively new approach for DAG scheduling [1, 3, 6, 14, 15, 17]. This kind of scheduling is also an NP-complete problem [13]. Performance of the duplication based algorithms are superior to non-duplication based ones in terms of generating smaller schedule lengths. However, this is usually achieved at the expense of higher time complexity and larger number of processors.

In this work, we propose a novel two-phase duplication-based scheduling algorithm, called *DUPS*. The first phase, called *minSL*, aims to minimize the schedule length by minimizing the earliest finish time of each task on a new processor. The schedule on the processor associated with a task is called the *partial schedule* of that task. In the second phase, called *minNP*, the required number of processors is minimized by eliminating and merging the partial schedules without increasing the schedule length found in the first phase. A nice feature of *minNP* is that, with small modifications, it can be applied to the output of the existing scheduling algorithms to improve their efficiency. Performance evaluation of DUPS is presented in comparison to the CPFD algorithm which is shown to be a superior algorithm compared to other related scheduling algorithms in terms of producing the smallest schedule length with requiring a moderate amount of processors [1, 4, 8].

2 Related Work

An excellent survey and taxonomy of task scheduling algorithms and some benchmarking techniques are presented in [11, 12]. DAG scheduling algorithms can be divided into two with respect to whether they allow task duplication or not. Non-duplication based algorithms aim to keep the number of processors required minimal

*This research was supported in part by the National Science Foundation under Grants #CCF-0342615, #ACI-0203846, #ANI-0330612, #CNS-0426241, NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003, Sandia National Laboratories under Doc.No: 283793.

and to achieve low-complexity scheduling without allowing any task duplication. In task duplication based algorithms, on the other hand, some tasks are duplicated on more than one processor to eliminate the interprocessor communication cost.

Two common approaches in non-duplication based scheduling are list scheduling and cluster-based scheduling. In list scheduling [16, 18], each task in a DAG is first assigned a priority. Then the tasks are considered in non-ascending order of priorities for scheduling on a set of available processors. Despite the fact that the quality of their schedules is usually worse than that of other algorithm classes, low complexity of the list-based algorithms still make them attractive alternatives.

In cluster-based scheduling, processors are treated as clusters and the completion time is minimized by moving tasks among clusters [10, 18]. At the end of clustering, heavily communicating tasks are assigned to the same processor, reducing the interprocessor communication. However, after clustering is completed, merging the clusters to fit into a smaller number of processors without degrading the schedule is usually the bottleneck for this class of algorithms.

The proposed algorithm is a duplication-based static scheduling algorithm and it differs from the previous algorithms by addressing the minimization of the schedule length and the number of processors used as separate problems to be optimized in two distinct phases.

3 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 that n_j depends on. The task n_p, from which the directed edge originates, is called the *parent task* and the task n_j, to which the directed edge points, is called the *child task*. 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*. A *non-join task* and a *join task* are defined as tasks having a single and multiple parents, respectively.

The weight w_j of a node n_j represents the computational weight associated with the respective task, whereas the cost $c_{p,j}$ of a directed edge (n_p, n_j) represents the communication cost between tasks n_p and n_j. In the DAG scheduling problem, if a task scheduled on one processor depends on a task scheduled on another processor, an interprocessor communication is required between those processors. However, if both of the tasks are assigned to same processor, the communication requirement is considered negligible. In the DAG shown in Figure 1(a), the numbers in the circles represent the computational weights of the respective nodes and the numbers on the edges represent the communication costs. In literature, several methods exist to make good estimations of computational weights and communication costs in a DAG [9, 18].

Earliest start time $est(n_j, P_\ell)$ of a task $n_j \in \mathcal{N}$ for a processor $P_\ell \in \mathcal{P}$ denotes the earliest time that n_j can start execution on P_ℓ, where $\mathcal{P}$ represents the set of homogeneous processors. Since a task can only start execution when all of its inputs are ready on a processor, $est(n_j, P_\ell)$ can be computed as $max\{min\{ft(n_p, P_\ell), ft(n_p, P_k) + c_{p,j}\}\}$ for each parent n_p of n_j and $P_k \neq P_\ell$ where $ft(n_p, P_k)$ denotes the *finish time* of task n_p on processor P_k. If n_p is not copied on P_k, $ft(n_p, P_k)$ is considered to be infinity. Please note that, we assume all links in the interprocessor network are contention free and processors can compute and communicate simultaneously. Execution of a task does not necessarily start at its est since another task may be scheduled to execute at that time slot on the same processor. For a task n_j scheduled on processor P_ℓ, $st(n_j, P_\ell)$ denotes the *start time* of n_j on P_ℓ. Note that, with non-preemptive execution of tasks, $ft(n_j, P_\ell) = st(n_j, P_\ell) + w_j$. If $est(n_j, P_\ell)$ is lower bounded due to a parent n_p of n_j which is not scheduled on P_ℓ to finish before $st(n_j, P_\ell)$, that parent is called the *critical parent* of n_j for processor P_ℓ.

Schedule length (SL) is the most important performance indicator of scheduling algorithms. It is defined as $SL = max_{n_j \in \mathcal{N}, P_\ell \in \mathcal{P}}\{ft(n_j, P_\ell)\}$. A widely used metric to evaluate the schedule length is the *normalized schedule length (NSL)* [1]. NSL is defined as the ratio of the parallel schedule length to the sum of the computational weights along the critical path, where *critical path* is defined as the path from an entry task to an exit task, along which the sum of the computational weights and communication costs is maximum. Another metric is the number of processors required by the generated parallel schedules. This metric measures how efficiently an algorithm utilizes the processors in scheduling the tasks.

4 The Algorithm

In the following subsections, the two main phases of the Duplication-based Scheduling Algorithm Using Partial Schedules (DUPS) and their relationship are described in more detail.

4.1 minSL: Minimizing the Schedule Length

The *minSL* algorithm exploits the assumption that an infinite number of processors are available, by creating a minimized partial schedule for each task on a separate processor. Basically, each task is first scheduled on a

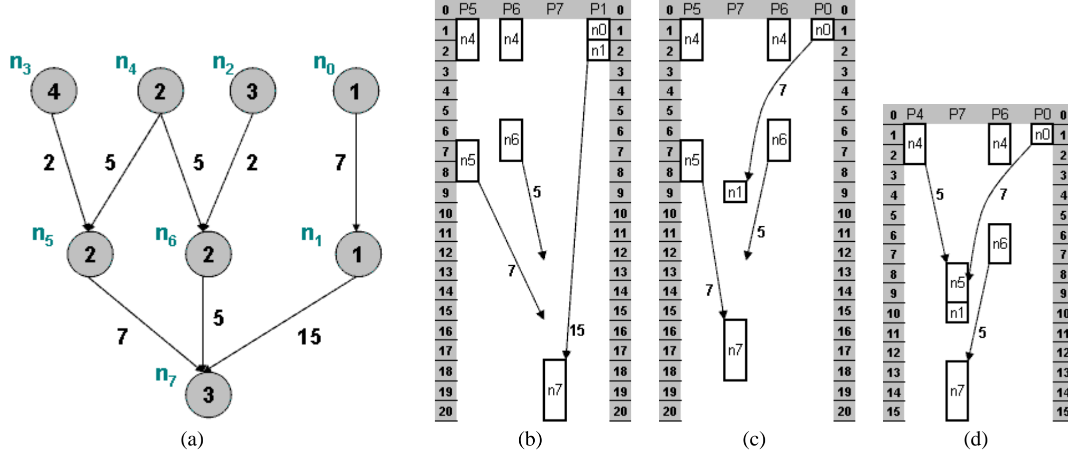

Figure 1. (a) DAG (b) Initializing n_7 (c) Duplicating n_1 on P_7. (d) Duplicating n_5 on P_7.

Algorithm 1 minSL

1: schedule each entry task n_s on a new processor P_s with $st(n_s, P_s) \leftarrow 0$
2: **while** there are unscheduled tasks **do**
3: $\quad n_t \leftarrow$ a randomly chosen task whose parents' partial schedules have been determined
4: $\quad$ **if** n_t is a non-join task **then**
5: $\quad\quad$ copy its parent's (n_p) partial schedule to P_t and set $st(n_t, P_t) \leftarrow ft(n_p, P_t)$
6: $\quad$ **else**
7: $\quad\quad st(n_t, P_t) \leftarrow est(n_t, P_t)$
8: $\quad\quad n_i \leftarrow n_t$ and $n_c \leftarrow$ critical parent of n_i
9: $\quad\quad$ **while** n_c exists **and** $(SL_{t,current}-$ idle time on $P_t) + w_c < SL_{t,best}$ **do**
10: $\quad\quad\quad$ compute $est(n_c, P_t)$ and schedule n_c on P_t
11: $\quad\quad\quad$ put tasks scheduled after n_c into *rList*
12: $\quad\quad\quad$ sort tasks in the *rList* and reschedule on P_t
13: $\quad\quad\quad$ **if** $SL_{t,current} < SL_{t,best}$ **then**
14: $\quad\quad\quad\quad P_{t,best} \leftarrow P_{t,current}$
15: $\quad\quad\quad$ find the critical child n_i and critical task n_c

---

new processor, then its parents (and parents of parents etc.) are duplicated on that processor one by one until there is no potential improvement by further duplication. Since each task n_j's partial schedule will be constructed on a new processor P_j, we will also use P_j to denote the partial schedule of n_j. Although this scheme requires as many processors as the number of tasks, the second phase of the *DUPS* algorithm will minimize the processor requirement by maintaining the achieved SL at this phase.

minSL algorithm (Algorithm 1) starts with scheduling each of the entry tasks on a new processor. Then, the partial schedules of the remaining tasks are constructed one by one. In step 3, one of tasks whose parents have already been scheduled is randomly selected as the *target task* (n_t), which is the task whose partial schedule will be constructed next. If n_t is a non-join task, the best partial schedule can be obtained by duplicating its parent's partial schedule onto a new processor P_t, called the *target processor*, and scheduling n_t to start immediately after its parent (step 5).

If the target task is a join task, first it is scheduled on a new processor P_t to start at its est in step 7. Then n_t's partial schedule is tried to be improved by duplicating tasks onto P_t (steps 9-15). The next task to be duplicated on P_t is called the *critical task* n_c, which is the critical parent of the *critical child* n_i, chosen among the tasks scheduled on P_t. In step 8, we set n_i to n_t and n_c becomes the critical parent of n_t. Please note that at the current moment the only way to improve the partial schedule P_t is duplicating critical parent of n_t on P_t.

Duplication of n_c onto P_t can create a potential improvement (step 9) only if the sum of the computational weights on P_t after duplication is smaller than the smallest partial schedule length obtained so far. If this is the case, n_c is scheduled to start either at the first idle slot after $est(n_c, P_t)$ or at $st(n_i, P_t)$, whichever is earlier (step 10). After n_c is scheduled, all tasks that were scheduled to start after its start time, including n_i, are pushed into the *rList* for rescheduling in order to avoid any overlaps and to utilize the idle slots better (step 11). Since n_c may also be the critical parent of some tasks scheduled on P_t other than n_i, those tasks may now start earlier as well. Sometimes such tasks can start earlier than the tasks scheduled to start before themselves in the previous iteration. Therefore, in step 12, the tasks in the *rList* are sorted with respect to their est's on P_t to allow each task to start as early as possible. While doing that the precedence of each previous n_c to its n_i is preserved to make sure that they will not again become a

critical child and parent pair for the same partial schedule. Furthermore, any duplicate of n_c in the *rList* is removed. This situation only occurs if current n_c was also the critical task for a previous critical child and hence it had been duplicated on P_t; but it had been scheduled to start after current n_i. Since n_c will be scheduled before current n_i in the current iteration, that copy of n_c becomes redundant.

Tasks in the *rList* are rescheduled on P_t such that if a task's est on P_t is earlier than the finish time of the task scheduled just before itself, it is scheduled at the finish time of that task; otherwise, it is scheduled at its est. If the duplication results in a better partial schedule than the best obtained so far, it is saved as the best partial schedule in step 14.

After the first pass of the loop in steps 9-15, partial schedule P_t may be improved by duplicating the critical parent of a task scheduled on P_t. Here, *minSL* chooses the critical child (n_i) as the latest starting task on P_t that starts at its est and has a critical parent. The rationale behind this choice is that the tasks scheduled to start after the critical child n_i start later than their est, and they can start earlier if an idle time slot can be created just before their start time. However, since n_i already starts at its est, it cannot start earlier on P_t and blocks those tasks unless its critical parent is duplicated on P_t to improve its est. Clearly, the choice of the task to be replicated is an heuristic decision. We have also experimented duplicating the critical parents of non-critical children and observed that it usually did not improve or slightly improved the schedule.

For the sample DAG displayed in Figure 1(a), consider the scheduling of task n_7. Since n_7 is a join task, at step 7 of Algorithm 1, it is initialized to start at its est, which is 17, on a new processor P_7 (Figure 1(b)). Since the sum of current computational weights and the computational weight of the critical task n_1 ($SL_{t,current}-$ idle time on $P_7 + w_1$ = 4) is less than the best schedule length so far (20); there is a potential to improve the schedule. Hence n_7's critical parent n_1 is duplicated on P_7 (Figure 1(c)). Since n_7 is the only task after n_1 on P_7, it is inserted into the *rList* and rescheduled, resulting a new best schedule length of 18. Then, n_7 and n_5 become the next critical child and the critical task respectively (step 15). n_5 is duplicated on P_7 to start its est (7) and both n_1 and n_7 are inserted into the *rList*. Finally they are rescheduled on P_7 resulting a new best partial schedule of length 15 (Figure 1(d)).

In the next iteration, n_c becomes n_6 with an est of 7. However, task n_5 is already scheduled to start at that time on P_7, therefore, n_6 is scheduled at the first idle slot after its est (Figure 2(a)). Note that at the end of this iteration SL did not improve. However, the algorithm does not stop here, since the schedule can still potentially be improved as it will be in the next iteration when n_4 becomes n_c and its duplication result in a new best partial schedule of length 13 (Figure 2(b)). Please note that after the duplication and sorting, n_6 was able to start before n_5 which allowed SL to improve by 1 compared to the case where n_5 is forced to start before n_6. After this point, duplication of the next critical task n_3 would not improve SL and the schedule on P_7 is stored as the partial schedule of task n_7. Note that although *minSL* stops here, as mentioned earlier it is possible to improve the schedule by duplicating a parent (n_0) of a non-critical child (n_1), as displayed in Figure 2(c). Our algorithm chooses not to search for such cases, because the high runtime cost do not amortize the negligible improvements.

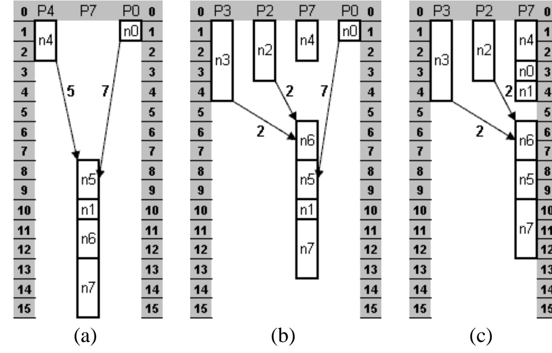

Figure 2. (a) Duplicating n_6 on P_7 (b) Duplicating n_4 on P_7 (c) Partial schedule on P_7 if n_0 is duplicated

4.2 minNP: Minimizing the Number of Processors Used

The *minNP* algorithm tries to minimize the number of processors required by the schedule generated by *minSL* without degrading the achieved schedule length. *minNP* consists of subsequent calls to two subroutines. First, redundant partial schedules are discarded by *elimProcs*, then the remaining ones are merged, as much as possible, by *mergeSchedules*.

elimProcs (Algorithm 2) checks whether a task is duplicated on a partial schedule other than its own to finish early enough such that it can fulfill the dependency requirements of its children. If it is, task's partial schedule is discarded and its duplicates on other partial schedules are shifted to later idle slots as much as possible to relax the *latest finish time* (lft) constraints of the task's parents. lft of a task is defined as the time that at least one of its duplicates should finish execution so that its children will receive data before their start time.

In *elimProcs*, all processors are considered one by

Algorithm 2 elimProcs

1: **for all** P_t with non-increasing SL_t order **do**
2:   **if** target task n_t on P_t is an exit task **then**
3:     $lft(n_t) \leftarrow SL; \quad fixed(n_t) \leftarrow P_t$
4:   **else**
5:     $\mathcal{D}_t = \{P_\ell | (n_t, n_d) \in \mathcal{E}$ and $ft(n_t, P_\ell) > st(n_d, P_\ell)\}$
6:     $t_k \leftarrow$ start time of the next task after n_t on P_k
7:     **if** $\mathcal{D}_t = \emptyset$ **then**
8:       $\mathcal{P} \leftarrow \mathcal{P} - \{P_t\}$
9:     **else**
10:       $lft(n_t) \leftarrow min_{P_\ell \in \mathcal{D}_t}\{st(n_d, P_\ell) - c_{t,d}\}$
11:       **if** $lft(n_t) \geq ft(n_t, P_\ell)$ for $P_\ell \in \mathcal{P}$ such that $SL_\ell > SL_t$ **and** $min_{P_k \in (\mathcal{P} - \{P_t\})}\{t_k - ft(n_t, P_k)\} = t_\ell - ft(n_t, P_\ell)$ **then**
12:         $P_c \leftarrow P_\ell; \quad \mathcal{P} \leftarrow \mathcal{P} - \{P_t\}$
13:       **else**
14:         $P_c \leftarrow P_t$
15:       $fixed(n_t) \leftarrow P_c$
16:     $ft(n_t, P_k) \leftarrow t_k$ for all $P_k \in \mathcal{P} - \{P_c\}$
17:     $ft(n_t, P_c) \leftarrow min\{t_c, lft(n_t)\}$

one in non-increasing partial schedule length in order to ensure that a task is always considered after its children. One duplicate of each task finishing no later than the lft is fixed on the processor it is scheduled and not allowed to finish later than its lft. Each task is fixed on at most one processor, and array *fixed* is used to keep track of the processors where each task is fixed. There is no lft restriction on the remaining duplicates.

If the target task n_t, the task that its partial schedule is created on the processor P_t, is an exit task, it is scheduled to finish at SL and *fixed* on P_t (step 3). For a non-exit n_t on the other hand, first the processors hosting its dependent children are found in step 5. If there is no such child, P_t is no longer needed (step 8). Otherwise, $lft(n_t)$ is calculated considering all dependent children n_d. Then, it is checked whether a duplicate of n_t on $P_\ell \neq P_t$ starts before $lft(n_t)$. In such a case, P_ℓ is labeled as the *candidate processor* P_c. If there are more than one such duplicates, ties are broken by choosing the processor on which the idle time immediately after n_t is the smallest. If no P_c is found, P_t is labeled as P_c. In step 15, n_t is fixed on P_c so that it cannot be scheduled to finish after $lft(n_t)$ on P_c in later steps.

In step 16, all duplicates of n_t except the one on P_c are shifted to finish at the start time of the next task after each of them. The duplicate on P_c is scheduled to finish at the smaller of $lft(n_t)$ and the start time of the next task on P_c.

An example DAG and the partial schedules after

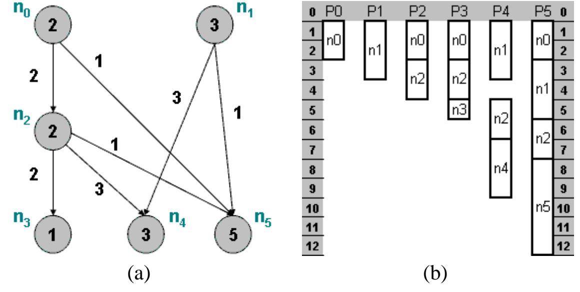

Figure 3. (a) DAG (b) Partial schedules after the *minSL* algorithm

minSL is given in Figure 3. *minNP* starts with considering P_5 which is the longest partial schedule. Since n_5 and similarly n_4 and n_3 are exit tasks, they are simply shifted to finish at SL. For the next longest partial schedule, P_2, it is checked if there is a dependent child that requires the partial schedule of n_2. In this example, n_2 is duplicated before its children on all relevant processors, therefore it has no dependent child and P_2 is discarded. The duplicates of n_2 are shifted to later slots as shown in Figure 4(a). By the same argument, partial schedule of n_1 is also discarded (Figure 4(b)).

The situation is different for n_0 since n_2 on P_4 is a dependent child of n_0 and $lft(n_0)$ is 5 due to this child. Out of the two duplicates of n_0 on processors P_3 and P_5 both of which finish earlier than $lft(n_0)$, the one on P_5 is chosen as the candidate since the idle time following n_0 is the smallest on P_5. As a result, n_0 is fixed on P_5 and P_0 is discarded. The resulting schedule is shown in Figure 4(c).

Further reduction in the number of processors can be achieved by merging the partial schedules by utilizing the idle slots. The *mergeSchedules* subroutine (Algorithm 3) designed for this purpose considers two processors at a time in non-increasing SL order and tries to merge the tasks on these processors without violating any dependency constraints. Before merging begins, *data send time (dst)*, the time that all children of a task finishes receiving data from the task is calculated. dst will be used to remove redundant duplicates during merging.

Out of the two processors being considered, the one having the longer (shorter) SL is labeled as $P_\ell(P_s)$. A merged schedule is constructed on a temporary processor P_m, which is filled back to front, by considering tasks one by one. To keep track of the start time of the last scheduled task, a counter t is initialized to SL in step 4. Next, the latest starting unconsidered non-redundant (with a dst larger than t) task from P_ℓ or P_s is picked as the next task to be merged (n_m) by the *pickTask* subroutine. If P_m becomes infeasible at any

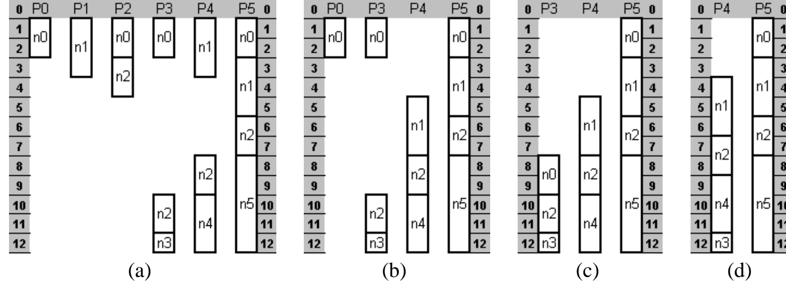

Figure 4. (a) Shifting n_2 (b) Shifting n_1 (c) After *elimProcs* (d) After *mergeSchedules*

Algorithm 3 mergeSchedules

1: $dst(n_i) \leftarrow lft(n_i) + max_{n_j}\{c_{i,j}\}\ \forall (n_i, n_j) \in \mathcal{E}$
2: **for all** P_ℓ with non-increasing SL_ℓ order **do**
3: **for all** P_s starting from the one after P_ℓ in non-increasing SL_s order **do**
4: $t \leftarrow SL$
5: **while** $n_m \leftarrow$ pickTask(P_ℓ, P_s) **do**
6: **if** n_m is fixed on P_ℓ or P_s **then**
7: $ft(n_m, P_m) \leftarrow min\{lft(n_m), t\}$
8: **else**
9: $ft(n_m, P_m) \leftarrow t$
10: $t \leftarrow st(n_m, P_m) \leftarrow ft(n_m, P_m) - w_m$
11: **if** $t < 0$ **then**
12: merge is not feasible; skip P_s
13: **else**
14: **for all** parents n_p of n_m which has no unconsidered duplicate on P_ℓ or P_s **do**
15: **if** $st(n_m, P_m) < lft(n_p) + c_{p,m}$ **then**
16: merge is not feasible; skip P_s
17: $P_\ell \leftarrow P_m$; $\mathcal{P} \leftarrow \mathcal{P} - \{P_s\}$
18: fix tasks previously fixed on P_s on P_ℓ

point, merging the current processor pair is canceled and the next pair is considered. If two schedules are merged without any dependency problems, P_m is stored as P_ℓ and P_s is discarded (step 17). Moreover, any task that was previously fixed on P_s is fixed on P_ℓ. In the example displayed in Figure 4, partial schedules P_3 and P_4 are merged and the resulting schedule is stored as P_4 whereas P_3 is discarded (Figure 4(d)).

After n_m is selected, it is tentatively scheduled on P_m and the feasibility of the schedule is checked. If n_m is fixed on P_ℓ or P_s, it is scheduled to finish at the minimum of $lft(n_m)$ and t (step 7). Otherwise, it is scheduled to finish at t. If this makes the start time of n_m negative then schedule is infeasible (step 12). We also need to verify if $st(n_m, P_m)$ is larger than $est(n_m, P_m)$. Here, when computing the est, we do not consider the parents duplicated on P_ℓ or P_s and have not been scheduled onto P_m yet. Clearly such parents will be scheduled before n_m on P_m (recall that we are filling P_m back-to-front). However, if data from any of the parents that will not be duplicated on P_m arrives after the scheduled start time of n_m, again P_m becomes infeasible (steps 14-16).

minNP algorithm can be used to increase the efficiency of other scheduling algorithms with some modification in *elimProcs*. In the current form, *elimProcs* exploits the structure of the partial schedules generated by *minSL*, and it discards a processor P_i if a duplicate of n_i on another processor can provide n_i's output to its dependent children in a timely manner. Since schedules generated by other algorithms are not guaranteed to have such a structure, *elimProcs* should be altered. In the modified *minNP* algorithm, *MminNP*, *elimProcs* is changed such that the lft calculation and task shifting steps are done in separate loops for a more accurate lft computation. The duplicate of each task starting closest to its lft is *fixed*, and non-fixed tasks are shifted to later slots on the other processors. The *mergeSchedules* subroutine is used without any modification.

5 Experimental Results

We evaluated the performance of the proposed DUPS algorithm on random DAGs and compared to the CPFD algorithm [1]. We also applied *MminNP*, to the output of CPFD to investigate the reduction in the number of processors used. This version of CPFD is called *CPFD+MminNP*.

The parameters of interest are the number of tasks (N), *communication to computation ratio (CCR)*, and the number of parent tasks (p) to control the number of dependencies. In the generated DAGs, N varies between 50 and 550, CCR between 0.1 and 10, and p between 2 and 10. For each parameter set 14 random DAGs are generated, resulting in 2520 DAGs. We have implemented both DUPS and CPFD using C++ and compiled using GNU gcc 3.2. All experiments are carried out on a Linux box equipped with 2.8GHz Pentium IV CPU and 512MB memory.

Due to lack of space we are only presenting the aver-

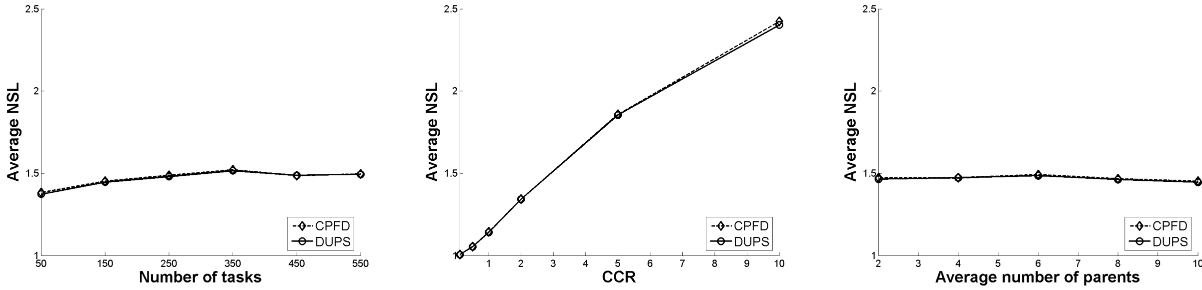

Figure 5. Average NSL while varying number of tasks, CCR and number of parents.

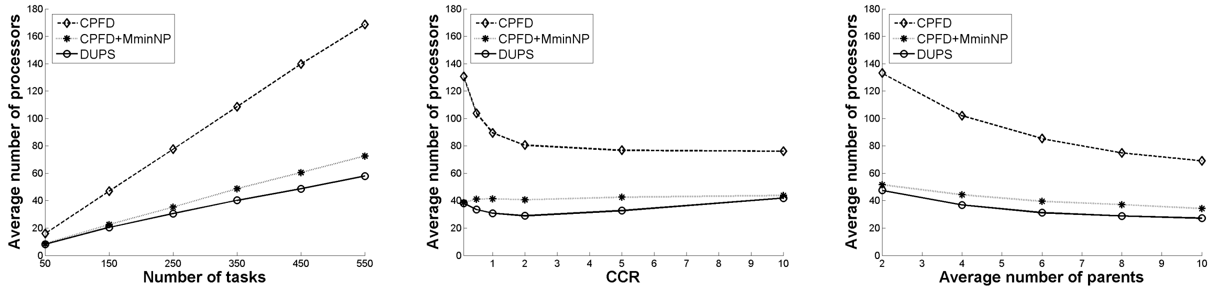

Figure 6. Average number of processors while varying number of tasks, CCR and number of parents.

age performance results in this paper. Detailed results for different values of DAG parameters can be found in [5]. NSL and the processor requirement of the schedules generated by DUPS, CPFD and CPFD+MminNP algorithms are presented in Figures 5-6. The results indicate that, the NSL of the schedules by DUPS and CPFD are very close to each other where DUPS is marginally (0.34%) better. Furthermore, the number of processors required by DUPS is only 37% of what is required by CPFD. Similarly, the processor requirement of CPFD+MminNP is 45% of CPFD. It is expected that as N increases, SL would also increase. However it is important that SL scale well with the change in N. Results illustrate that DUPS scales well both in terms of NSL and the number of processors used.

CCR is defined as the ratio of the average communication cost to the average computation cost in a DAG. The larger the CCR, the more important the communication costs become in scheduling. If on the average, communication takes considerably larger amount of time than computation, i.e. if CCR is large, the processors tend to be idle for a larger amount of time. The reason is that the child tasks have to wait until all the data from their parents are communicated before their execution can start. Therefore scheduling algorithms usually suffer more with larger CCR. Figure 6 displays that both CPFD and DUPS suffer with large CCR and the performance improvement of DUPS over CPFD is not significant in terms of NSL. The difference in the number of processors used for DUPS and CPFD+MminNP is the largest when CCR is around 1.0 to 5.0 and it is relatively smaller at the extreme cases of CCR (0.1 and 10.0).

The final parameter of interest is the average number of parents of the tasks. Let E be number of edges in the DAG. Then p can be computed as $p = \frac{E}{N}$. As E increases for a constant N, each task will depend on a larger number of tasks which may complicate the scheduling problem. The effect of this parameter has not been investigated thoroughly in previous related studies. Our experiments demonstrated that NSL does not change considerably with p. This result can be interpreted as the change in SL is highly correlated with the change in the sum of the computation costs along the critical path, therefore increase in SL with p is proportional to this sum. On the other hand, the processor requirement decreases slightly as the number of dependencies increase. The reason is that, increasing SL with p, results in larger time slots to be utilized on the processors. Thus, tasks can fit on a smaller number of processors. Please also refer to [5] for detailed experimental results.

The run time comparison of DUPS and CPFD as well as the run time of DUPS phases and MminNP are shown in Figure 7. Please note that we have coded both algorithms for correct functionality but did not optimize them for run time. The results show that DUPS has a smaller run time although both algorithms have the same time complexity of $O(N^4)$. CPFD schedules each task

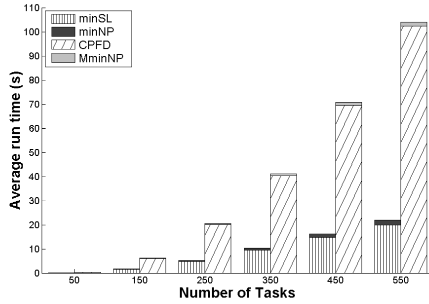

Figure 7. Runtime comparison of CPFD+MminNP and DUPS

on all processors that contain any of its parents before choosing the best processor to schedule the task. On the contrary, in DUPS, only a single partial schedule is constructed for each task, thus avoiding several scheduling attempts to find the best choice.

6 Conclusion and Future Work

The schedules produced by the proposed DUPS algorithm has been shown to require only 37% of the number of processors required by the well known CPFD algorithm while achieving the same schedule length on the average. We also applied the second phase of the proposed algorithm after CPFD. The results showed that we were able to reduce the processor requirement of CPFD to its 45%, however, still considerably larger than that of DUPS.

Testing DUPS on real world applications and comparing it with algorithms that aim to optimize different metrics are included in our future work plan. We also plan to generalize the algorithm to suit for problems with tighter constraints.

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] I. Ahmad and Y.-K. Kwok. On parallelizing the multiprocessor scheduling problem. *IEEE Transactions on Parallel and Distributed Systems*, 10(4):414–432, April 1999.

[3] S. Bansal, P. Kumar, and K. Singh. An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):533–544, June 2003.

[4] C. Boeres and V. Robello. Cluster-based static scheduling: Theory and practice. *Proceedings of the* 14th *Symposium on Computer Architecture and High Performance Computing*, pages 133–140, October 2002.

[5] D. Bozdağ. A task duplication based scheduling algorithm using partial schedules. Master's thesis, The Ohio State University, 2005.

[6] S. Darbha and D. Agrawal. Optimal scheduling algorithm for distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):87–95, January 1998.

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

[8] L. Guodong, C. Daoxu, W. Daming, and Z. Defu. Task clustering and scheduling to multiprocessors with duplication. *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[9] 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.

[10] Y.-K. Kwok and I. Ahmad. Dynamic critical path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[11] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, December 1999.

[12] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[13] C. Papadimitriou and M. Yannakakis. Towards an architecture independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19:322–328, April 1990.

[14] C. Park and T. Choe. An optimal scheduling algorithm based on task duplication. *IEEE Transactions on Computers*, 51(4):444–448, April 2002.

[15] G. Park, B. Shirazi, and J. Marquis. Mapping of parallel tasks to multiprocessors with duplication. *Proceedings of the* 31st *Annual Hawaii International Conference on System Sciences*, 7:96–105, January 1998.

[16] A. Radulescu and A. van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):648–658, June 2002.

[17] M. Wu, W. Shu, and J. Gu. Efficient local search for dag scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):617–627, June 2001.

[18] M.-Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.