Compaction of Schedules and a Two-Stage Approach for Duplication-Based DAG Scheduling

Doruk Bozdağ, Füsun Özgüner, *Senior Member*, *IEEE*, and Umit V. Catalyurek

Abstract—Many DAG scheduling algorithms generate schedules that require prohibitively large number of processors. To address this problem, we propose a generic algorithm, SC, to minimize the processor requirement of any given valid schedule. SC preserves the schedule length of the original schedule and reduces processor count by merging processor schedules and removing redundant duplicate tasks. To the best of our knowledge, this is the first algorithm to address this highly unexplored aspect of DAG scheduling. On average, SC reduced the processor requirement 91, 82, and 72 percent for schedules generated by PLW, TCSD, and CPFD algorithms, respectively. SC algorithm has a low complexity ($O(|\mathcal{N}|^3)$) compared to most duplication-based algorithms. Moreover, it decouples processor economization from schedule length minimization problem. To take advantage of these features of SC, we also propose a scheduling algorithm SDS, having the same time complexity as SC. Our experiments demonstrate that schedules generated by SDS are only 3 percent longer than CPFD ($O(|\mathcal{N}|^4)$), one of the best algorithms in that respect. SDS and SC together form a two-stage scheduling algorithm that produces schedules with high quality and low processor requirement, and has lower complexity than the comparable algorithms that produce similar high-quality results.

Index Terms—Scheduling and task partitioning, task duplication, algorithms, multiprocessor systems.

1 Introduction

Enhanced performance and reduced cost of commodity hardware boosted employment of distributed memory multiprocessor systems (DMMS) in supercomputing fields, such as climate research, physical simulations, image processing, and database systems. In many applications, an efficient parallel version of the application program is not available. Therefore, parallelism is achieved by partitioning the program into smaller chunks, called *tasks*, and scheduling these tasks on target DMMS to minimize the overall execution time, or *schedule length*.

The goal of partitioning is to represent the program in the form of a *directed acyclic graph (DAG)* consisting of tasks with appropriate grain size [1], [2]. Task computation and intertask communication times in the DAG are determined according to the target DMMS architecture via estimation and benchmarking techniques [3], [4], [5]. Dependencies among tasks incur inevitable communication overhead when tasks are assigned to different processors of the DMMS. The aim of DAG scheduling is to minimize parallel execution time by assigning tasks onto DMMS in a way to reduce interprocessor communication. DAG scheduling problem is shown to be NP-complete in its general form [6]. Optimal polynomial solutions exist only for limited cases where either the costs [7], [8], [9] or the shape [10], [11], [12], [13] of the input DAG is restricted.

In general, heuristic algorithms proposed in the literature offer trade-offs between the quality and the time complexity of scheduling solutions. On one end of the spectrum, there are low time complexity techniques, such as list scheduling, that produce relatively low-quality solutions [14], [15]. More recently developed duplication-based scheduling techniques lie on the other end of the spectrum [2], [10], [16], [17], [18], [19]. By assigning some of the tasks redundantly on multiple processors, algorithms using this method avoid large amount of interprocessor communication. Although time complexity of duplication-based algorithms are significantly higher, quality of solutions is usually much better than nonduplication-based techniques [10], [20].

The number of processors required is also an important concern regarding the practicality of a schedule. Many algorithms in the literature assume unlimited number of available processors and fail to truly address this issue. Some algorithms on the other hand, are designed for scheduling on bounded number of processors. However, in addition to producing longer schedules due to this limitation, these algorithms do not have a mechanism to *minimize* the processor requirement either. In this work, we propose a generic algorithm, called *schedule compaction* (SC), to minimize the processor requirement of any given valid schedule. SC algorithm preserves the schedule length of the original schedule and reduces processor count by merging processor schedules and removing redundant duplicate tasks. It can be

- *D. Bozdağ is with the Department of Electrical and Computer Engineering, The Ohio State University, 333 W. 10th Avenue, 3190 Graves Hall, Columbus, OH 43210. E-mail: bozdag.1@osu.edu.*
- *F. Özgüner is with the Department of Electrical and Computer Engineering, The Ohio State University, 205 Dreese Laboratory, 2015 Neil Avenue, Columbus, OH 43210. E-mail: ozguner@ece.osu.edu.*
- *U.V. Catalyurek is with the Department of Biomedical Informatics, The Ohio State University, 333 W. 10th Avenue, 3172B Graves Hall, Columbus, OH 43210 and the Department of Electrical and Computer Engineering, The Ohio State University, 316 Dreese Laboratory, 2015, Neil Avenue, Columbus, OH 43210. E-mail: umit@bmi.osu.edu.*

Manuscript received 5 Mar. 2008; revised 15 Oct. 2008; accepted 9 Dec. 2008; published online 18 Dec. 2008.
Recommended for acceptance by A. Pietracaprina.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-03-0088.
Digital Object Identifier no. 10.1109/TPDS.2008.260.

TABLE 1
DAG Scheduling Algorithms and Their Properties

Non-duplication based algorithms		
Algorithm	Time Complexity	Classification
DSC [24]	$O(\|\mathcal{E}\|log\|\mathcal{N}\|)$	Cluster-based
LC [25]	$O(\|\mathcal{E}\|\|\mathcal{N}\|)$	Cluster-based
DCP [15]	$O(\|\mathcal{N}\|^3)$	List
MD [4]	$O(\|\mathcal{N}\|^3)$	Cluster-based
EZ [26]	$O(\|\mathcal{E}\|^2)$	Cluster-based
Duplication based algorithms		
Algorithm	Time Complexity	Classification
TDS [7]	$O(\|\mathcal{N}\|^2)$	Partial duplication
LWB [27]	$O(\|\mathcal{N}\|^2)$	Partial duplication
PLW [11]	$O(\|\mathcal{N}\|(\|\mathcal{E}\| + \|\mathcal{N}\|log\|\mathcal{N}\|))$	Partial duplication
DFRN [20]	$O(\|\mathcal{N}\|^3log\|\mathcal{N}\|)$	Partial duplication
LCTD [28]	$O(\|\mathcal{N}\|^3log\|\mathcal{N}\|)$	Full duplication
TCSD [22]	$O(\|\mathcal{N}\|^3log\|\mathcal{N}\|)$	Full duplication
PY [9]	$O(\|\mathcal{N}\|^2(\|\mathcal{E}\| + \|\mathcal{N}\|log\|\mathcal{N}\|))$	Full duplication
CPFD [10]	$O(\|\mathcal{N}\|^4)$	Full duplication
BTDH [16]	$O(\|\mathcal{N}\|^4)$	Full duplication
DSH [2]	$O(\|\mathcal{N}\|^4)$	Full duplication

applied to the output of any scheduling algorithm, regardless of the bound on the number of available processors or the restrictions on task duplication. To the best of our knowledge, except for our preliminary study [21], this is the first work to address this highly unexplored aspect of DAG scheduling.

The worst-case time complexity of SC algorithm is $O(|\mathcal{N}|^3)$, where $|\mathcal{N}|$ is the number of tasks in the DAG. Duplication-based algorithms that generate shortest schedules usually have higher time complexities than $O(|\mathcal{N}|^3)$. Furthermore, SC decouples processor economization from schedule length minimization problem. To take advantage of these features of SC, we also propose a duplication-based scheduling algorithm called *sub-DAG based scheduling* (SDS). SDS has a worst-case time complexity of $O(|\mathcal{N}|^3)$, yet, as our experiments demonstrate, it generates schedules comparable to the best algorithms in the literature. Performance of SDS is evaluated in comparison to three duplication-based algorithms, CPFD [10], PLW [11], and TCSD [22]. CPFD has a time complexity of $O(|\mathcal{N}|^4)$, however, it has been shown several times that it is one of the best algorithms in the literature to generate the shortest schedules [10], [20], [22], [23]. PLW has a relatively lower time complexity (see Table 1), and TCSD can be viewed as a compromise between PLW and CPFD in terms of quality versus time complexity trade-off.

The rest of the paper is organized as follows: In Section 2, we give an overview of previous work on DAG scheduling. Section 3 introduces basic concepts and definitions. We give details of the SC and SDS algorithms in Sections 4 and 5, respectively. Experimental evaluation of our work is presented in Section 6 and we conclude in Section 7.

2 Related Work

DAG scheduling algorithms can be divided into two classes with respect to whether they allow task duplication or not (see Table 1). In nonduplication-based scheduling, two main approaches are list scheduling and cluster-based scheduling. In list scheduling [15], each task is first assigned a priority. Then the tasks are considered in descending order of priorities for scheduling on a set of available processors. Although they have relatively low time complexity, the quality of schedules generated by algorithms in this class are generally worse than that of algorithms in other classes.

In cluster-based scheduling, processors are treated as clusters and the completion time is minimized by moving tasks among clusters [4], [24], [25], [26]. At the end of clustering, heavily communicating tasks are assigned to the same processor. While this reduces interprocessor communication, it may lead to load imbalances or idle time slots when tasks are ordered on processors with respect to precedence constraints in the DAG. For more information about nonduplication-based algorithms, refer to the excellent survey by Kwok and Ahmad in [29].

Duplication-based scheduling is also an NP-complete problem [9]. To improve the start time of a target task, algorithms in this class first select a task that the target task has a dependency. By duplicating the selected task on the same processor with the target task, interprocessor communication can be avoided. Duplication-based algorithms can be divided into two classes with respect to selection of the task to duplicate. Full duplication-based algorithms [2], [9], [10], [16], [22], [28] allow tasks from higher levels in the DAG to be considered for duplication, whereas partial duplication-based algorithms [7], [11], [20], [27], [30] impose some restrictions during the task selection process. Such restrictions lead to time versus quality trade-off between these two subclasses. In general, space complexity of full duplication-based algorithms is $O(|\mathcal{N}|^2)$ and that of partial duplication-based algorithms is $O(|\mathcal{E}|)$.

Wu et al. [31] proposed a low time complexity algorithm based on local search to reduce the schedule length produced by any nonduplication-based scheduling algorithm. The idea of improving existing algorithms has also been studied by Radulescu and Van Gemund [32]. They proposed a method to reduce the time complexity of nonduplication-based algorithms without significantly affecting the quality of solutions. Nevertheless, we are not aware of any work that focuses on minimizing the processor requirement of an existing schedule. Duplication-based algorithms especially suffer from large processor requirement [10] due to large number of replicated tasks. There is also plenty of room for improvement for algorithms designed for scheduling on bounded number of processors [14], [33], [34], [35], [36], [37] as well.

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 $(n_p, n_t) \in \mathcal{E}$ for each task n_p (*parent*) that n_t (*child*) depends on. A child task depends on its parents such that its execution 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 weight w_t represents the expected computation time associated with task n_t and the cost $c_{p,t}$ of a directed edge (n_p, n_t) represents the required communication time to send data from n_p to n_t if they are assigned to different processors. Here, it is assumed that the communication network is fully connected and there is no contention in the links. *Critical path* on a DAG is defined as the path from an entry task to an exit task, along which sum of the computational weights and communication costs is maximum. The bottom level b_t of a

task n_t is computed by traversing the DAG upward starting from exit tasks. It is defined as $b_t = w_t + \max_{(n_t,n_c)\in\mathcal{E}}(c_{t,c} + b_c)$. The bottom level of an exit task is defined to be equal to its computation weight.

The set of processors in a homogeneous computing environment is denoted by $\mathcal{P}$, where each processor P_ℓ in this set can compute and communicate simultaneously. We call the schedule consisting of tasks assigned to a processor along with their start times as a *partial schedule*. Since mapping a set of partial schedules onto a set of homogeneous processors is trivial, we use the terms partial schedule and processor interchangeably and use the same notation to refer them. For a task n_t scheduled on processor P_ℓ, $st(n_t, P_\ell)$ and $ft(n_t, P_\ell)$ denote the *start time* and the *finish time* of n_t on P_ℓ, respectively. Note that with nonpreemptive execution of tasks, $ft(n_t, P_\ell) = st(n_t, P_\ell) + w_t$. Schedule length is denoted by σ and computed as follows:

$$\sigma = max_{n_t\in\mathcal{N},P_\ell\in\mathcal{P}}\{ft(n_t, P_\ell)\}. \quad (1)$$

Tasks scheduled on each partial schedule are sorted in increasing start time order. We call the execution order of a task n_t on a partial schedule P_ℓ as the *position* of n_t on P_ℓ and denote it by $pos(n_t, P_\ell)$. The position of the task with the smallest start time on a partial schedule is 1.

Consider an edge (n_p, n_t) in a DAG and a partial schedule P_ℓ in a valid schedule corresponding to this DAG. If n_p is not scheduled on P_ℓ or if it is scheduled at a position greater than $pos(n_t, P_\ell)$, n_t is required to receive data from a copy of n_p residing on some other partial schedule. Then, the copy of n_t on P_ℓ is called an *off-processor child* of n_p.

On the other hand, if a copy of n_p is scheduled on P_ℓ at a position smaller than $pos(n_t, P_\ell)$, the copy of n_t on P_ℓ is called a *local child* of n_p.

4 Compaction of Schedules onto Fewer Partial Schedules

In this section, details of the SC algorithm will be presented. The goal of the SC algorithm is to reduce the processor requirement of a valid input schedule without degrading the schedule length.

4.1 Overview of the SC Algorithm

SC algorithm consists of three phases, where actual schedule compaction is achieved in the third phase. In this phase, partial schedules are considered in pairs and a pair is merged by rescheduling tasks on both partial schedules onto a new partial schedule P_m. On P_m, a task may need to be rescheduled to start earlier or later than its original scheduled time due to limited available time slots. For a merge operation to be successful, all of the following four conditions should hold for P_m:

1. A task on P_m should receive data from all of its parents before its scheduled start time.
2. A task having off-processor children should finish early enough such that its off-processor children receive data before their scheduled start time.
3. A task having local children should be scheduled for execution before all of its local children.
4. The length of P_m should not exceed the overall schedule length σ.

Throughout Section 4, we will refer to these conditions as *Validity Conditions (VCs)*. The first two phases of the SC algorithm are concerned with improving the effectiveness of the third phase. In the first two phases, tasks on partial schedules are reorganized to improve their flexibility with respect to *VC* 1 and *VC* 2. Moreover, the number of task duplicates are reduced to help satisfy *VC* 4.

In a valid input schedule, a task may have a copy on more than one partial schedule. Except for exit tasks, each task copy in a schedule either has a local or an off-processor child. Here, we note that if a copy of a task n_t finishes early enough, it can provide data to all of n_t's off-processor children. To formalize the finish time requirement of such a copy, we give the following definition:

Definition 1. *Latest finish time $lft(n_t)$ of a task n_t is the latest time that at least one copy of n_t must finish so that dependencies between n_t and its off-processor children are preserved.*

To make use of this observation in SC algorithm, *exactly* one copy of n_t is held responsible for providing data to n_t's off-processor children. Selected copy is *fixed* on the partial schedule it resides and other copies are freed from satisfying any off-processor children dependency. Formally:

Definition 2. *A copy of task n_t is said to be fixed on partial schedule P_ℓ if it is constrained to finish before or on $lft(n_t)$ on P_ℓ.*

By fixing a copy of a task to satisfy all off-processor children dependencies, the sole purpose of the nonfixed copies becomes satisfying local children dependencies. Thus, nonfixed copies are no longer required to meet *VC* 2, which introduces more flexibility in Phase 3. Selection of the fixed copy for each task is carried out in the first phase of the SC algorithm. In this phase, tasks are also shifted to start as late as possible without violating *VC* 3. The purpose is to convert as many local children dependencies as possible to off-processor children dependencies, which eventually helps reducing the number of task duplicates. Before further explanation, we define *data arrival time* as follows:

Definition 3. *Data arrival time $dat(n_t, n_p)$ from a task n_p to its off-processor child n_t is the time that n_t is guaranteed to receive data from the fixed copy of n_p.*

Once a local child n_t of a task n_p is shifted to start later in Phase 1, it is possible that its start time becomes greater than $dat(n_t, n_p)$. In that case, we convert the local child dependency to an off-processor child dependency to relieve the copy of n_p on P_ℓ from *VC* 3 with respect to its child n_t. If n_p on P_ℓ has no more local child dependency to fulfill, it becomes redundant. Such redundant task copies are discarded in the second phase of the SC algorithm. As a result, the number of task duplicates is reduced to allow more free time slots, which helps satisfying *VC* 4 in Phase 3. In addition to identification of redundant duplicates, the second phase of SC algorithm is also concerned with computation of *earliest start time* of each task.

Definition 4. *Earliest start time $est(n_t, P_\ell)$ of a task n_t on a partial schedule P_ℓ denotes the earliest time that n_t will receive data from all of its parents and be ready for execution on P_ℓ.*

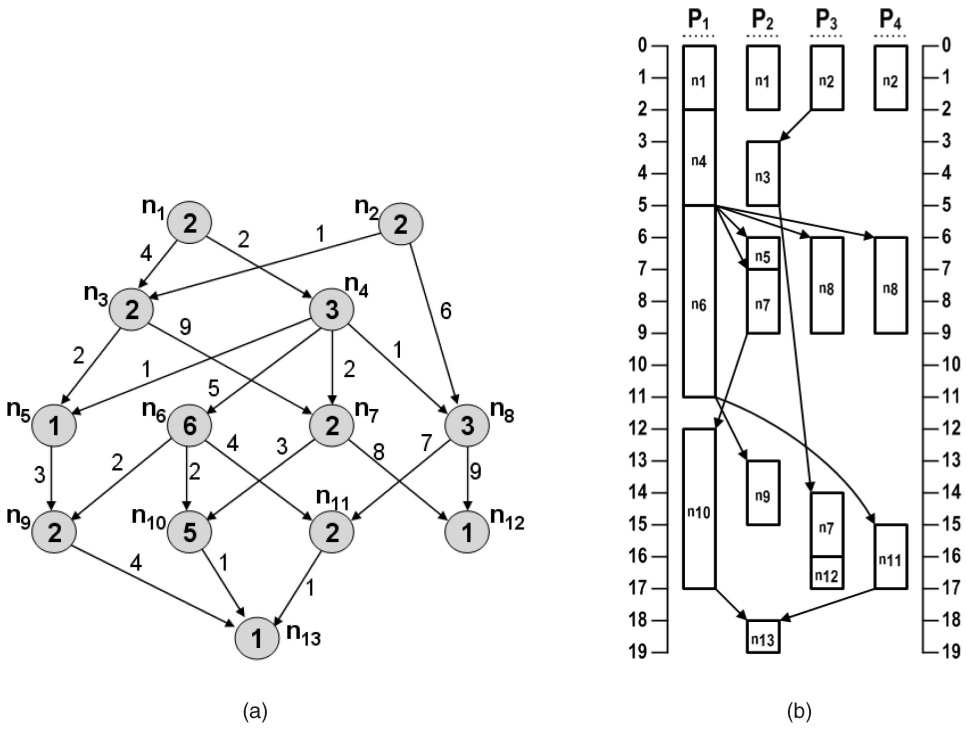

Fig. 1. (a) A sample DAG. (b) A sample schedule consisting of four partial schedules, with $\sigma = 19$.

In Phase 2, est of each task is computed and fixed tasks are shifted back to start as early as possible without violating *VC* 1 and *VC* 3. The purpose of shifting back the fixed copy of a task is to reduce the est of its off-processor children, therefore providing more flexibility for *VC* 1 in Phase 3 of the SC algorithm.

Given the state of the schedule as well as est, lft, and fixed copy information at the end of Phase 2, in Phase 3, partial schedules are merged as much as possible to reduce the processor requirement. The rest of this section is dedicated to explain the details and further optimizations regarding the individual phases of the SC algorithm.

4.2 Phase 1: Compute lft, Select Fixed Copies, and Shift Tasks

There are three objectives in the first phase of the SC algorithm: to determine *latest finish time* of each task, to select task copies to fix on partial schedules, and to shift tasks to finish as late as possible (see Algorithm 1). In this phase, tasks are considered one by one in nondecreasing bottom level order to ensure that a task is always considered before its parents.

Algorithm 1. Compute lft, select fixed copies and shift tasks

1: **for each** n_t with nondecreasing bottom level order **do**
2: Initialize $lft(n_t)$
3: Fix a copy of n_t if n_t has off-processor children
4: Shift all copies of n_t, $lft(n_t) \leftarrow ft(n_t, source(n_t))$

Let n_t denote the task being considered in a particular iteration of the for loop in Algorithm 1 and $\mathcal{Q}(n_t)$ denote the set of partial schedules that have at least one off-processor child of n_t. If $\mathcal{Q}(n_t)$ is empty, then the only constraint on finish time of n_t is that it cannot finish later than the schedule length σ due to *VC* 4. In that case, $lft(n_t)$ is set to σ. Otherwise, it is initialized as follows:

$$lft(n_t) = min_{(n_t,n_c)\in\mathcal{E}, P_\ell\in\mathcal{Q}(n_t)}\{st(n_c, P_\ell) - c_{t,c}\}. \quad (2)$$

As mentioned in Section 4.1, it is sufficient to fix a single copy of n_t to start before or on $lft(n_t)$ to satisfy *VC* 2. In deciding which copy of a task n_t to fix, there are two cases to consider. In the first case, on every partial schedule that contain a copy of n_t, the task succeeding n_t has a start time greater than $lft(n_t)$. Then, we fix the copy of n_t on the partial schedule that gives the minimum difference between $lft(n_t)$ and the start time of the task succeeding n_t on the same partial schedule; so that fixing the copy results in the smallest possible idle time. In the second case, there is at least one partial schedule P_k on which $lft(n_t)$ is greater than the start time of the task succeeding n_t on P_k. In such a case, we fix the copy that finishes the earliest. The partial schedule that the fixed copy of n_t resides is called the *source* of n_t and denoted by $source(n_t)$. $source(n_t)$ is set to $NULL$ if n_t is not fixed on any partial schedule (i.e., when $\mathcal{Q}(n_t)$ is empty).

Consider an example input schedule shown in Fig. 1b corresponding to the DAG in Fig. 1a. There are four partial schedules in this example and the arrows on the schedule show important off-processor child dependencies. Tasks n_1, n_5, n_8, n_9, n_{12}, and n_{13} do not have off-processor children, hence no copy of these tasks are fixed. On the other hand, all remaining tasks have at least one off-processor child. Tasks n_3, n_4, n_6, n_{10}, and n_{11} have single copies and they are fixed on the processors they reside. Finally, the copies of tasks n_2 and n_7 are fixed on P_3 and P_2, respectively. In Fig. 1b, finish time of each fixed task is equal to its latest finish time.

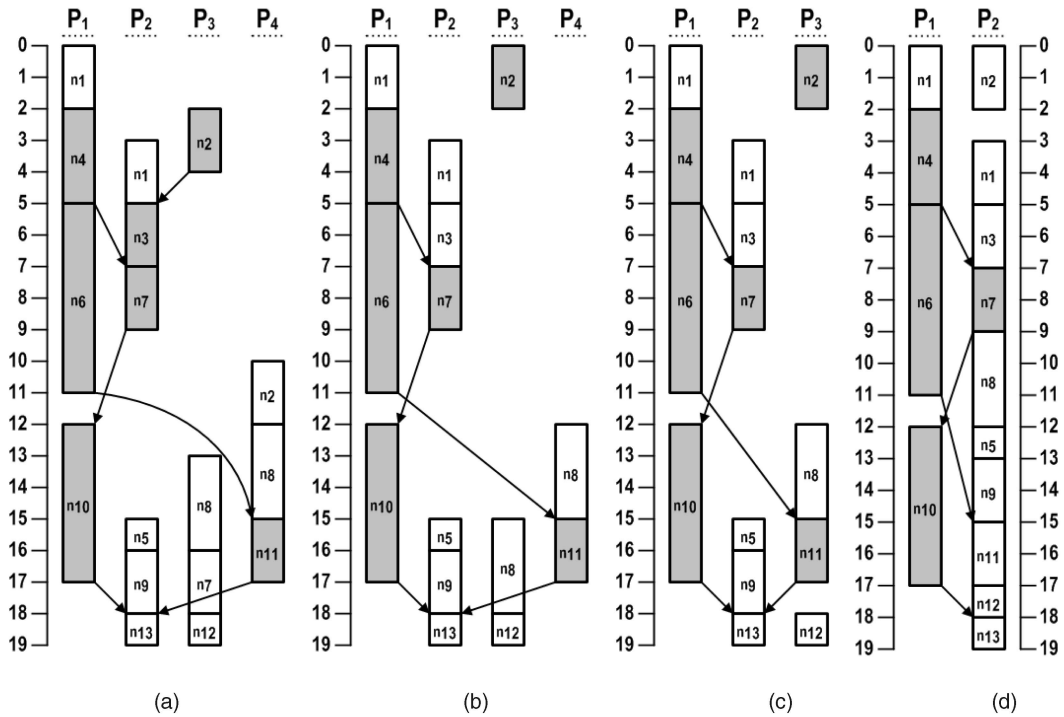

Fig. 2. Running trace of the SC algorithm. (a) After shifting tasks to later time slots. (b) After removing redundant copies. (c) After merging P_3 and P_4. (d) After merging P_2 and P_3.

Once a copy n_t is fixed, all of its copies are shifted to finish as late as possible by making use of the trailing idle time slots toward the end of partial schedules. During the shift operation, a copy of n_t is allowed to jump over other tasks on the same partial schedule provided that they are not local children of n_t. The fixed copy of n_t is also shifted, however, it is not allowed to finish after $lft(n_t)$ to ensure *VC* 2 is not violated. Once the fixed copy is shifted, we set $lft(n_t)$ to the new finish time of the fixed copy, which may be smaller than current $lft(n_t)$. This helps relaxing *VC* 1 for n_t's off-processor children.

The example schedule at the end of Phase 1 of the SC algorithm is given in Fig. 2a. Note that task n_5 on P_2 jumped over n_7 since n_7 is not a local child of n_5. In Fig. 2, fixed task copies are shown shaded.

4.3 Phase 2: Compute est and Remove Redundant Copies

The aim of the second phase is to compute *earliest start time* of each task copy and to discard redundant duplicates. A task copy is considered redundant if it does not have any local or off-processor child. A redundant task copy can be removed from the schedule without violating any dependency. Main steps of this phase are given in Algorithm 2.

In the loop shown in lines 11-19 of Algorithm 2, $est(n_t, P_\ell)$ is computed for each copy of task n_t on each partial schedule P_ℓ. While computing est, children dependencies are also checked to identify redundant copies. If a copy of task n_t is needed to be kept on a partial schedule P_ℓ to satisfy a local or an off-processor child dependency, $keep(n_t, P_\ell)$ is set to 1 to denote that this copy is not redundant.

Algorithm 2. Compute est and remove redundant copies

```
1: for each task n_t in nondecreasing bottom level order do
2:     for each partial schedule P_ℓ that has a copy of n_t do
3:         if P_ℓ = source(n_t) then
4:             if Q(n_t) = ∅ then
5:                 Free the fixed copy of n_t
6:             else
7:                 keep(n_t, source(n_t)) ← 1
8:         if n_t is an exit task or keep(n_t, P_ℓ) = 1 then
9:             est(n_t, P_ℓ) ← 0
10:            Shift n_t to later slots on P_ℓ
11:            for each parent n_p of n_t do
12:                if n_p is not scheduled before n_t on P_ℓ then
13:                    est(n_t, P_ℓ) ← max{est(n_t, P_ℓ), dat(n_t, n_p)}
14:                Q(n_p) ← Q(n_p) ∪ P_ℓ
15:            else if source(n_p) = P_ℓ or
16:                    dat(n_t, n_p) > st(n_t, P_ℓ) then
17:                keep(n_p, P_ℓ) ← 1
18:            else
19:                C(n_p, P_ℓ) ← C(n_p, P_ℓ) ∪ {(n_t, dat(n_t, n_p))}
20:        else
21:            Discard copy of n_t on P_ℓ
22:            if C(n_t, P_ℓ) ≠ ∅ then
23:                Q(n_t) ← Q(n_t) ∪ P_ℓ
24:                for each (n_c, dat(n_c, n_t)) ∈ C(n_t, P_ℓ) do
25:                    est(n_c, P_ℓ) ← max{est(n_c, P_ℓ), dat(n_c, n_t)}
26: Shift back fixed task copies and update their lft
```

In Algorithm 2, fixed copy of each task n_t is considered after all nonfixed copies are considered. Since some task copies will be discarded in this phase, $\mathcal{Q}(n_t)$ may become empty. In that case, the fixed copy of n_t is no longer required to finish at $lft(n_t)$ since it does not have any off-processor

children anymore. Thus, the fixed copy is changed to a nonfixed copy. This case is handled in line 5 of Algorithm 2. On the other hand, if a fixed task still has off-processor children, $keep(n_t, source(n_t))$ is set to 1, so that the fixed copy will be retained.

There are three possible ways that data dependency between the copy of task n_t on P_ℓ and its parent n_p can be satisfied. In the first case, n_t is an off-processor child of n_p and receives data from the fixed copy of n_p. In this case, P_ℓ is included in $\mathcal{Q}(n_p)$ and $est(n_t, P_\ell)$ is updated as given in line 13 of Algorithm 2. Therefore, maximum data arrival time from such parents of n_t determines $est(n_t, P_\ell)$. In the remaining two cases, n_p is scheduled to start before n_t on P_ℓ. In the second case, either $dat(n_t, n_p) > st(n_t, P_\ell)$ or n_p is fixed on P_ℓ. Therefore, n_t is a local child of n_p and the copy of n_p on P_ℓ is essential to fulfill the child dependency. Then, in line 17 of Algorithm 2, we set $keep(n_p, P_\ell)$ to 1 so that n_p will not be discarded later on. Finally, in the third case, as a result of shifting to later slots in Phase 1 as well as in line 10 of Algorithm 2, n_t may start later than $dat(n_t, n_p)$. Hence, both the fixed copy and the copy on P_ℓ of n_p can be used to satisfy the child dependency. If the fixed copy of n_p is used to satisfy the child dependency, then the copy of n_p on P_ℓ may become redundant and can be removed from the schedule. However, for that to happen, all children of n_p on P_ℓ should start later than $dat(n_t, n_p)$. Since all children of n_p may not have been considered yet, n_t is put into a list $\mathcal{C}(n_p, P_\ell)$. This is the list of local children of n_p on P_ℓ that can be converted to off-processor children of n_p if the copy of n_p on P_ℓ is discarded. $dat(n_t, n_p)$ is also recorded together with n_t in this list.

A task n_t is always considered after its children in Algorithm 2. Thus, while considering the copy of n_t on P_ℓ, $keep(n_t, P_\ell)$ has already been set to 1 if that copy is essential to satisfy any child dependency. On the other hand, if $keep(n_t, P_\ell)$ has not been set, this copy of n_t is discarded as given in line 21. If $\mathcal{C}(n_t, P_\ell)$ is not empty, then local children of n_t in this list are converted to off-processor children of n_t, and P_ℓ is added to $\mathcal{Q}(n_t)$. est of each of these local children is also updated using the previously recorded data arrival time from the fixed copy of n_t in line 25.

Fig. 2b shows the schedule in our running example at the end of this phase. Since n_8 and n_{12} can receive data from the fixed copies of n_2 (on P_3) and n_7 (on P_2), respectively, n_2 on P_4 and n_7 on P_3 are removed as they became redundant. Furthermore, n_3 on P_2 is no longer fixed as the removed copy of n_7 on P_3 was the only off-processor child candidate of n_3. To relax *VC* 1 for their children, fixed tasks are shifted back to start as early as possible (no jumps allowed). Their lfts are also reduced in line 26 of Algorithm 2. In the example, only the fixed copy of n_2 on P_3 is affected by this and shifted back with $lft(n_2)$ set to 2. Note that by reducing $lft(n_2)$, children n_3 and n_8 of n_2 have more room to be shifted back, providing a flexibility in terms of *VC* 1.

4.4 Phase 3: Merge Partial Schedules

Algorithm 3 outlines Phase 3 of the SC algorithm. In this phase, pairs of partial schedules are considered for merging by utilizing idle slots on partial schedules. As explained in Section 4.1, the resulting partial schedule in a merge operation is required to meet all *Validity Conditions*. This phase consists of several rounds and a partial schedule is merged with at most one other in a single round. This approach ensures a smaller time complexity at the expense of a possible, yet slight improvement in the final processor requirement.

Algorithm 3. Merge partial schedules

1: $\mathcal{M} \leftarrow \mathcal{P}$
2: **loop** ▷ Each iteration corresponds to a round
3: $\mathcal{S} \leftarrow \{< P_\ell, P_k > | P_\ell \in \mathcal{P}$ and $P_k \in \mathcal{P}$, and $P_\ell \in \mathcal{M}$ or $P_k \in \mathcal{M}$, and $sim_{\ell,k} \geq 0\}$
4: **break-loop** if $\mathcal{S} = \emptyset$
5: $\mathcal{M} \leftarrow \emptyset$
6: **for all** $< P_\ell, P_k > \in \mathcal{S}$ in nonincreasing sim order **do**
7: $P_m \leftarrow$ an empty partial schedule, $t \leftarrow \sigma$
8: $q_\ell \leftarrow$ position of the last task on P_ℓ
9: $q_k \leftarrow$ position of the last task on P_k
10: **while** $q_\ell \geq 1$ or $q_k \geq 1$ **do**
11: $j \leftarrow schedule\_task(t, P_\ell, P_k, q_\ell, q_k)$
12: **if** $j = -1$ **then**
13: **go to** line 6 (skip $< P_\ell, P_k >$ pair)
14: $q_j \leftarrow q_j - 1$
15: **for each** task n_t scheduled on P_m **do**
16: **if** $source(n_t) = P_\ell$ **or** $source(n_t) = P_k$ **then**
17: $\mathcal{Q}(n_t) \leftarrow \mathcal{Q}(n_t) - \{P_\ell, P_k\}$
18: **if** $\mathcal{Q}(n_t) \neq \emptyset$ **then**
19: $source(n_t) \leftarrow P_m$
20: **else**
21: $source(n_t) \leftarrow NULL$
22: **else if** $ft(n_t, P_m) < lft(n_t)$ **then**
23: $source(n_t) \leftarrow P_m$, $lft(n_t) \leftarrow ft(n_t, P_m)$
24: $est(n_t, P_m) \leftarrow 0$
25: **for each** parent n_p of n_t **do**
26: **if** n_p is not scheduled before n_t on P_m **then**
27: $est(n_t, P_m) \leftarrow max\{est(n_t, P_m), dat(n_t, n_p)\}$
28: Shift back fixed task copies on P_m and update lfts
29: $P_\ell \leftarrow P_m$, $\mathcal{P} \leftarrow \mathcal{P} - \{P_k\}$
30: $\mathcal{M} \leftarrow \mathcal{M} \cup \{P_\ell\}$
31: $\mathcal{S} \leftarrow \mathcal{S} - \{< P_i, P_j >\}$ for $i \in \{\ell, k\}$ or $j \in \{\ell, k\}$

In order to decide which pairs of partial schedules to merge at a particular round, priorities are assigned to each possible pair based on a pairwise similarity measure. Let et_ℓ denote the sum of the execution times of the tasks scheduled on partial schedule P_ℓ and σ_ℓ denote the finish time of the task at the last position on P_ℓ. Note that since the last task on P_ℓ is either a fixed task that finishes at its lft or it already finishes at σ, it should not be scheduled to finish later than σ_ℓ. Let $et_{\ell,k}$ denote the sum of the execution times of the tasks common to partial schedules P_ℓ and P_k. Then, we compute the similarity $sim_{\ell,k}$ between P_ℓ and P_k as follows:

$$sim_{\ell,k} = \begin{cases} \frac{et_{\ell,k}}{min\{et_\ell, et_k\}}, & \textbf{if } et_\ell + et_k - et_{\ell,k} \leq max\{\sigma_\ell, \sigma_k\} \\ -1, & \textbf{otherwise}. \end{cases} \quad (3)$$

The numerator in this equation is the amount of overlap between P_ℓ and P_k. The denominator is the maximum amount of overlap that could be achieved if tasks on one of the partial schedules were subset of the tasks on the other one, in which case P_ℓ and P_k are perfectly similar. It is also checked if the smallest possible partial schedule length of

$et_\ell + et_k - et_{\ell,k}$ that would be achieved by merging P_ℓ and P_k violates *VC* 4, or *VC* 2 for the task at the last position on either of these partial schedules. If it does, then $sim_{\ell,k}$ is set to -1 to denote this. The set of partial schedule pairs to be considered at each round is computed in line 3 of Algorithm 3 and denoted by $\mathcal{S}$. At least one of the partial schedules in a pair in $\mathcal{S}$ must be an element of set $\mathcal{M}$, which is the set of partial schedules successfully merged in the preceding round. This approach bounds the time complexity of the algorithm as will be explained in Section 4.5. Algorithm 3 terminates if set $\mathcal{S}$ becomes empty.

The pairs in set $\mathcal{S}$ are considered in nonincreasing similarity order for merging. Let P_ℓ and P_k denote the partial schedules in the pair being considered. A temporary partial schedule P_m is constructed by considering tasks on P_ℓ and P_k one by one. Let q_ℓ and q_k denote the position of the tasks being considered on P_ℓ and P_k, respectively. q_ℓ and q_k are initialized as the position of the last task on respective partial schedules. To keep track of the possible finish time of the next task to be scheduled on P_m, a counter t is initialized to σ in line 7 of Algorithm 3.

Until all tasks on P_ℓ and P_k are considered, the task pointed by either q_ℓ or q_k is selected and scheduled on P_m by calling the $schedule\_task$ function in line 11. The selected task is always scheduled at the first position on P_m, hence tasks are placed on P_m in reverse direction with respect to the time axis. If the task pointed by q_ℓ is selected, $schedule\_task$ function returns ℓ, and q_ℓ is decremented in line 14 to point to the task at position $q_\ell - 1$ on P_ℓ. Similar arguments hold for q_k. On the other hand, if merging P_ℓ and P_k is detected to be infeasible, $schedule\_task$ function returns -1 and the next pair in $\mathcal{S}$ is considered for merging. $schedule\_task$ function will be explained in more detail later in this section.

If P_ℓ and P_k are merged without any dependency violations, some optimizations are carried out between lines 15 and 28 before P_m replaces these two partial schedules. Consider a task n_t that was previously fixed on P_ℓ or P_k. The selection mechanism in $schedule\_task$ function ensures that all children of n_t on P_k and P_ℓ are scheduled at positions after $pos(n_t, P_m)$ on P_m. Therefore, n_t will no longer have off-processor children on either P_ℓ or P_k as they will become local children; and P_ℓ and P_k are removed from set $\mathcal{Q}(n_t)$ in line 17. If $\mathcal{Q}(n_t)$ becomes empty as a result of this, n_t is freed from satisfying any child dependency. Otherwise, it is fixed on P_m. If a nonfixed copy of a task n_t on P_m has a finish time smaller than $lft(n_t)$, n_t is fixed on P_m with $lft(n_t)$ equal to this finish time in line 23. Again, this is done to relax *VC* 1 for n_t's off-processor children in the later iterations of the algorithm.

Consider a task n_t which was scheduled on either P_ℓ or P_k before merging and was an off-processor child of a task n_p. On the resulting partial schedule P_m, it is possible that a copy of n_p might have been scheduled at a position before $pos(n_t, P_m)$. Therefore, n_t on P_m will no longer be an off-processor child of n_p. Furthermore, latest finish times of parents of n_t of which n_t is an off-processor child might have been reduced at previous iterations of the current round (due to line 28). As a result, $est(n_t, P_m)$ might have been reduced. Therefore, in the loop in line 25, est of such tasks are recomputed based on the current state of the schedule. Then, fixed tasks are shifted to start as early as possible on P_m based on the updated est information and their lfts are updated, if they are reduced, in line 28.

Finally, P_m is stored as P_ℓ together with the fixed task, est, and lft information relevant to P_m in line 29 of Algorithm 3. Also P_k is removed from set $\mathcal{P}$. P_ℓ is then inserted in set $\mathcal{M}$ and unconsidered pairs in set $\mathcal{S}$ that include either P_ℓ or P_k are removed to ensure that each partial schedule is merged at most once in a round in line 31. In the remainder of this section, we give details of the $schedule\_task$ function called in line 11 of Algorithm 3.

4.4.1 Selecting a Task to Schedule on P_m

Let n_ℓ denote the task at position q_ℓ on P_ℓ and n_k denote the task at position q_k on P_k. $schedule\_task$ function selects one of these tasks as the next task to be scheduled on P_m in a reasonable way. The selected task is denoted by n_a and the other task is denoted by n_b. Partial schedules that n_a and n_b reside are denoted by P_a and P_b, respectively. If $q_\ell = 0$, n_a is set to n_k, and if $q_k = 0$, then n_a is set to n_ℓ trivially. Otherwise, between n_ℓ and n_k, the one having the greater start time is initially assigned to n_a.

As will be explained shortly, $schedule\_task$ function schedules the selected task at the first position on P_m, and we use t to denote the latest allowed finish time of the selected task on P_m. Initially, we assume that n_a will be scheduled on P_m to finish at t. If n_a is a fixed task, however, it will be fixed on P_m as well and will finish at $min(t, lft(n_a))$. Therefore, if n_a is fixed on P_a and $lft(n_a) < t$, scheduling n_a on P_m will leave a gap of $t - lft(n_a)$ on P_m. To avoid such a gap, n_a and n_b are swapped if n_b is not a fixed task.

The selected task is always scheduled at the first position on P_m. Therefore, a task selected for scheduling before another one will be at a later position on P_m, hence will have a greater start time. If n_a is selected before n_b for scheduling on P_m, then n_b will have a smaller start time compared to the case where n_b is selected before n_a. In the former case, n_b may not be able to start later than its est on P_m (which is approximated by $est(n_b, P_b)$ here) and violate *VC* 1. n_a and n_b are swapped if n_b should be selected before n_a to be able to satisfy *VC* 1.

4.4.2 Scheduling the Selected Task on P_m

To schedule the selected task n_a, we first check if there is an unconsidered copy of n_a that resides on P_b. If this is the case, since the copy of n_a on P_b will be considered for duplication on P_m later on, n_a is not scheduled to avoid multiple copies on P_m. Furthermore, if n_a was fixed on P_a, it would fixed on P_b and $lft(n_a)$ would be reduced to $ft(n_a, P_b)$. Then, $schedule\_task$ function terminates and returns a.

If there is no unconsidered copy of n_a on P_b, n_a is scheduled to finish at t on P_m. If n_a is fixed on P_a with $lft(n_a) < t$, its finish time is changed to $lft(n_a)$ on the condition that P_b is not the only partial schedule in $\mathcal{Q}(n_a)$. The reason is that if $\mathcal{Q}(n_a) = \{P_b\}$, this means that the only off-processor children of n_a reside on P_b. Since n_a is fixed on P_a, off-processor children of n_a on P_b must have greater start times than $ft(n_a, P_a)$. Therefore, they should have already been scheduled on P_m with start times greater than t. As a result, n_a does not have any restriction to start at $lft(n_t)$, in this case.

After scheduling n_a, t is set to $st(n_a, P_m)$, and feasibility of P_m as well as earliest start time constraints of n_a are checked. If t becomes negative or smaller than $est(n_a, P_a)$, $schedule\_task$ function returns -1. Otherwise it returns a.

Note that $est(n_a, P_m)$ would have been smaller than $est(n_a, P_a)$, however, computing $est(n_a, P_m)$ here brings along a significant time complexity. Therefore, we approximate $est(n_a, P_m)$ via the precomputed upper bound $est(n_a, P_a)$.

In our example in Fig. 2b, merging P_1 with any other partial schedule violates *VC* 4, hence its similarity is to other partial schedules is marked with -1. Among other possible pairs, $< P_3, P_4 >$ is selected as the first pair to be merged since $sim_{3,4} = \frac{3}{5}$ is the largest. n_a and n_b are initialized to n_{12} and n_{11}, respectively. First, n_{12} is scheduled on P_m (which will become P_3 in Fig. 2c) since it has a greater start time. Then, n_8 is assigned to n_a, however, since an unconsidered copy of n_8 exists on P_4, it is skipped. Subsequently, n_{11}, n_8, and n_2 are scheduled on P_m with decreasing start time order, and P_m is assigned to P_3 since merging is successful (Fig. 2c). In the second round, $< P_2, P_3 >$ pair is the only pair considered for merging. Here, after n_{13} and n_{12}, n_{11} is scheduled on P_m even though its start time is smaller than n_9. The reason is that if n_9 was selected before n_{11}, n_{11} would start at 13 and its dependency to n_6 would be violated. $< P_2, P_3 >$ pair is also merged successfully and the resulting schedule is given in Fig. 2d.

4.5 Complexity of SC

We make two reasonable assumptions about the input schedule while computing the time complexity of SC algorithm. First, a task can not have more than one copy on a partial schedule, hence the number of tasks on a partial schedule can not exceed $|\mathcal{N}|$. Second, initial number of processors $|\mathcal{P}|$ is smaller than or equal to $|\mathcal{N}|$.

In lft computation step of Phase 1, all copies of all children of each task are considered, resulting in time complexity of $O(|\mathcal{E}||\mathcal{P}|)$. Allowing jumps during task shifting requires $O(|\mathcal{N}|)$ operations for each task. Since all copies of all tasks are considered, complexity of this step is $O(|\mathcal{N}|^2|\mathcal{P}|)$, which is also the time complexity of Phase 1. est computation step in Algorithm 2 considers either all parents (line 11) or all children (line 24) of a task for all copies of the task. Therefore, the time complexity of Phase 2 is $O(|\mathcal{E}||\mathcal{P}|)$.

At each round of Algorithm 3, at least one of the partial schedules in each pair in $\mathcal{S}$ should have been successfully merged in the previous round, hence should be an element of $\mathcal{M}$. During similarity computation, similarity between each partial schedule in $\mathcal{M}$ and all other partial schedules in $\mathcal{P}$ are computed. Similarity of two of partial schedules can be computed in $O(|\mathcal{N}|)$ by using a mark array of size $|\mathcal{N}|$. Let m_i and p_i denote the sizes of sets $\mathcal{M}$ and $\mathcal{P}$ at the beginning of ith round, respectively. Since a partial schedule can be merged with at most one other in a single round: $0 \leq m_{i+1} \leq \frac{p_i}{2}$ and $\frac{p_i}{2} \leq p_{i+1} \leq p_i - 1$. In the worst-case scenario, only one pair is merged at each round. Then, the cumulative number of pairs considered over all rounds is $(|\mathcal{P}| - 1)^2 + \sum_{i=2}^{|\mathcal{P}|-1}(|\mathcal{P}| - i)$, where $(|\mathcal{P}| - 1)^2$ term is due to the first round at which all partial schedules are in $\mathcal{M}$. Hence, the time complexity of similarity computation is $O(|\mathcal{N}||\mathcal{P}|^2)$. The while loop in line 10 iterates over the tasks on partial schedules being merged. Since $schedule\_task$ function is $O(1)$, overall time complexity of the while loop is also $O(|\mathcal{N}||\mathcal{P}|^2)$. The for loop in line 15 of Algorithm 3 has a time complexity of $O(|\mathcal{E}|)$ and it is executed only after a successful merge operation. Since at most $|\mathcal{P}| - 1$ merge operations can be successful in the entire Phase 3, overall time complexity of this loop is $O(|\mathcal{E}||\mathcal{P}|)$.

Resulting time complexity of SC algorithm is $O(|\mathcal{N}|^2|\mathcal{P}|)$. For input schedules assuming unbounded number of processors $|\mathcal{P}|$ is at most $|\mathcal{N}|$ and the time complexity of SC becomes $O(|\mathcal{N}|^3)$.

5 Scheduling Sub-DAGs onto Partial Schedules

SDS algorithm is designed to take advantage of existence of SC algorithm, hence it creates a minimized partial schedule for each task on a separate processor. Basically, each task is first scheduled on a temporary partial schedule P_t and its parents (and parents of parents etc.) are duplicated on P_t one by one until there is no potential improvement in the start time of the task by further duplication. Although this scheme requires as many processors as the number of tasks, processor requirement will be minimized by applying SC algorithm after SDS while maintaining the achieved schedule length.

SDS algorithm (Algorithm 4) starts with scheduling each of the entry tasks on a new processor. For each task n_i, we always set $eft(n_i)$ to $ft(n_i, P_i)$. In other words, we assume that among all partial schedules, n_i finishes the earliest on P_i since P_i is dedicated to achieve minimum possible finish time for n_i. After entry tasks are scheduled, partial schedules of the remaining tasks are constructed one by one. Task n_i whose partial schedule will be constructed next is selected among the tasks that have all their parents already scheduled.

Definition 5. *Among parents of a task n_j that are not scheduled on P_t at a position before n_j, the one that has the largest data arrival time is called the critical parent of n_j.*

Algorithm 4. SDS

1: Schedule each entry task n_i on a new partial schedule P_i with finish time and eft of w_i
2: **for each** nonentry task n_i **do**
3: ▷ all parents of n_i must have already been scheduled
4: Compute dat for each parent of n_i
5: Fill $parents_i$ array, $idx_i \leftarrow 0$
6: $n_p \leftarrow critical\_parent(n_i)$
7: Schedule n_i on P_i to start at $dat(n_i, n_p)$
8: $P_t \leftarrow P_i$, $n_b \leftarrow n_i$, $\Delta \leftarrow \infty$
9: **while** n_p exists **and**
10: idle time before $st(n_b, P_t)$ is greater than w_p **do**
11: **if** $pos(n_b, P_t) = 1$ **and**
12: $max\{st(n_b, P_t) - \Delta, dat(n_b, n_q)\} \leq eft(n_p)$ **then**
13: **if** $st(n_b, P_t) - eft(n_p) > ft(n_i, P_t) - ft(n_i, P_i)$
14: **then**
15: $P_i \leftarrow P_p$, append tasks on P_t to P_i
16: **break-while**
17: Call $duplicate$ function to duplicate n_p on P_t and to determine new n_b and Δ
18: $n_p \leftarrow critical\_parent(n_b)$
19: **if** $ft(n_i, P_t) < ft(n_i, P_i)$ **then**
20: $P_i \leftarrow P_t$
21: **break-while** if while loop iterated $|\mathcal{N}| \times \kappa$ times
22: $eft(n_i) \leftarrow ft(n_i, P_i)$

In Algorithm 4, we use the function $critical\_parent$ to determine the critical parent of a given task according to Definition 5. In lines 4 and 5, we sort parents of n_i in nonincreasing dat order and store them in array $parents_i$. We use a variable idx_i to keep the index of the current critical parent of n_i in $parents_i$. When $critical\_parent$ is called for a task n_j, we simply check whether the parent at position idx_j in $parents_j$ is duplicated on P_t at a position before n_j or not. If it is, we increment idx_j and repeat the check. Otherwise, the parent at position idx_j is returned as the critical parent of n_j. Initially, idx_i is set to 0, since the first parent in $parents_i$ has the greatest dat and is not currently duplicated on P_t. In lines 6 and 7, this first critical parent is assigned to n_p and n_i is scheduled on P_i to start at $dat(n_i, n_p)$.

During construction of P_i, n_i always remains at the last position on P_i and n_i's parents, parents of parents, etc., are duplicated on P_i one by one to improve $ft(n_i, P_i)$. From this point on, we use P_t to denote the current state of the partial schedule being constructed and P_i to denote the best state so far. P_t may have more tasks scheduled on it than P_i and $ft(n_i, P_t)$ may be greater than $ft(n_i, P_i)$. However, by further duplication on P_t, $ft(n_i, P_t)$ may get smaller than $ft(n_i, P_i)$, in which case we update P_i in line 20.

Definition 6. *If a task n_j on P_t can start earlier when the task at position $pos(n_j, P_t) - 1$ starts earlier, n_j is called a computation-bounded task. Otherwise, it is called a communication-bounded task. The latter occurs when n_j starts at $dat(n_j, n_p)$, where n_p is the critical parent of n_j.*

Definition 7. *Among communication-bounded tasks on P_t, the one at the latest position is called the bottleneck task and is denoted by n_b.*

Note that all tasks starting later than the bottleneck task are computation bounded and their start time can be reduced if n_b can start earlier. Therefore, the aim of the while loop in lines 9-21 is to determine the bottleneck task and improve its start time by duplicating its critical parent. Since n_i is the only task on P_i in line 8, we initialize n_b with n_i. The while loop iterates as long as there exists a bottleneck task n_b with a critical parent n_p and the total idle time before $st(n_b, P_t)$ is greater than w_p.

Definition 8. *Minimum decrease needed in $st(n_b, P_t)$ so that a task at a later position than n_b on P_t will become communication bounded is called target shift and denoted by Δ.*

In line 17, n_p is duplicated on P_t at a position just before n_b as will be explained in more detail below. Suppose that n_b was at the first position on P_t and n_p is duplicated immediately before n_b on P_t to finish at $eft(n_p)$. Let n_q denote the new critical parent of n_b. If $dat(n_b, n_q) < eft(n_p)$, then n_b can be shifted to start at $eft(n_p)$. However, if $st(n_b, P_t) - \Delta < eft(n_p)$, then tasks at later positions than n_b will remain computation-bounded. Since $eft(n_p)$ is the earliest possible finish time for n_p and n_b should start after n_p, $st(n_i, P_i)$ cannot get smaller any further in this scenario. In order to take advantage of this observation, we check if $eft(n_p)$ is greater than both $st(n_b, P_t) - \Delta$ and $dat(n_b, n_q)$ in lines 11 and 12. If this is true, complete partial schedule of n_p is inserted before n_b to ensure that n_p finishes at $eft(n_p)$ on P_t. Then, tasks already on P_t are appended without leaving any idle time in line 15. We carry out this operation only if it improves the overall partial schedule length and finalize P_i in line 16. Operations in lines 11-16 significantly reduce the runtime of the algorithm while not changing the resulting P_i.

If $eft(n_p)$ does not satisfy the conditions above, we duplicate n_p on P_t and determine the next bottleneck task in line 17. After the duplication, critical parent of the new bottleneck task is determined and best schedule P_i is updated if $ft(n_i, P_i)$ improves. In line 21, P_i is finalized if the the while loop iterated $|\mathcal{N}| \times \kappa$ times, where κ is a constant. This check is used to limit the worst-case time complexity and does not have a significant practical impact on SDS algorithm. After construction of P_i ends, we set $eft(n_i)$ to $ft(n_i, P_i)$ in line 22.

5.1 Task Duplication on P_t

$Duplicate$ function in Algorithm 5 outlines the procedure for duplicating critical parent n_p of n_b onto P_t. In the first line of this algorithm, we would delete the copy of n_p on P_t if it had been duplicated previously. Then, we set idx_p to 0 so that all parents of n_p will be considered by $critical\_parent$ function calls for n_p until P_i is finalized. If critical parent n_r of n_p exists, we initialize $st(n_p, P_t)$ with $dat(n_p, n_r)$. If finish time of the task at position $pos(n_b, P_t) - 1$ is greater than $dat(n_p, n_r)$, we set $st(n_p, P_t)$ to the finish time of this task. Then, n_p is duplicated at the position immediately before n_b to start at $st(n_p, P_t)$ in line 7.

Algorithm 5. Duplicate n_p on P_t: *duplicate* function

1: Delete the previously scheduled copy of n_p on P_t if exists
2: $idx_p \leftarrow 0$, $st(n_p, P_t) \leftarrow 0$
3: $n_r \leftarrow critical\_parent(n_p)$
4: $st(n_p, P_t) \leftarrow dat(n_p, n_r)$ if n_r exists
5: $n_j \leftarrow$ task at $pos(n_b, P_t) - 1$
6: $st(n_p, P_t) \leftarrow max\{st(n_p, P_t), ft(n_j, P_t)\}$ if n_j exists
7: Duplicate n_p on P_t immediately before n_b to start at $st(n_p, P_t)$, $ft \leftarrow ft(n_p, P_t)$
8: **for each** task n_j in increasing $pos(n_j, P_t)$ order with $pos(n_j, P_t) > pos(n_p, P_t)$ **do**
9: $st(n_j, P_t) \leftarrow ft$
10: $n_k \leftarrow critical\_parent(n_j)$
11: $st(n_j, P_t) \leftarrow max\{ft, dat(n_j, n_k)\}$ if n_k exists
12: $ft \leftarrow ft(n_j, P_t)$
13: $\Delta \leftarrow \infty$
14: **for each** task n_j in decreasing $pos(n_j, P_t)$ order
15: $n_k \leftarrow critical\_parent(n_j)$
16: **if** n_k exists **then**
17: **if** $st(n_j, P_t) = dat(n_j, n_k)$ **then**
18: ▷ Check if n_j is communication bounded
19: $n_b \leftarrow n_j$
20: **break-for**
21: $\Delta \leftarrow min\{\Delta, st(n_j, P_t) - dat(n_j, n_k)\}$

After duplicating n_p, we update start times of the tasks at later positions than n_p. First, start time of each task n_j is set equal to the finish time of the task preceding it in line 9. Then, it is set to $dat(n_j, n_k)$ if $dat(n_j, n_k)$ is greater, where n_k is the critical parent of n_j. The $critical\_parent$ function call

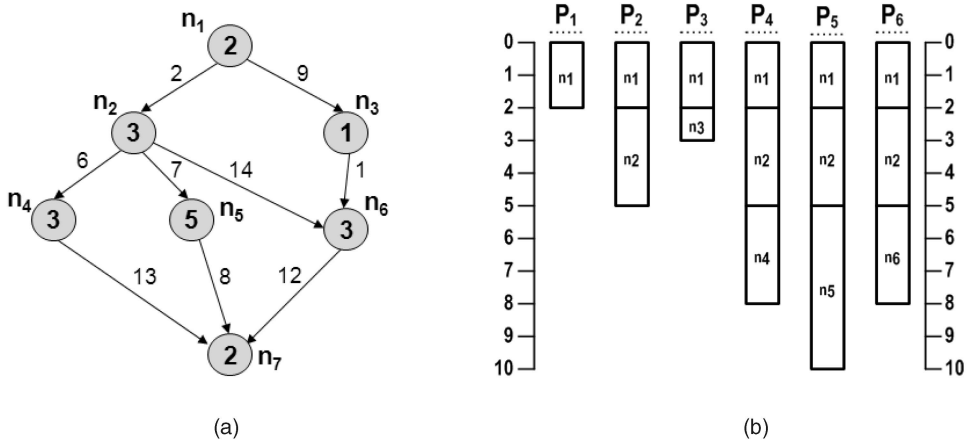

Fig. 3. (a) A sample DAG. (b) Partial schedules for tasks n_1-n_6 generated by SDS algorithm.

in line 10 also ensures that a task n_j at a later position than n_p will never have n_p as its critical parent on P_t anymore.

After updating the start times, we search for the next bottleneck task n_b and target shift Δ associated with it in lines 13-21. We go through each task n_j starting from the task at the last position on P_t. When a communication-bounded task is encountered, it is assigned to n_b and *duplicate* function terminates. Otherwise, if the amount of reduction required in $st(n_j, P_t)$ to make n_j communication-bounded is less than Δ, we assign Δ this reduction amount in line 21.

5.2 A Scheduling Example

In order to demonstrate the scheduling mechanism of SDS, an example DAG is given in Fig. 3a. Partial schedules for tasks n_1, n_2, n_3, n_4, and n_5 are relatively easier to construct, hence they will not be discussed here. These partial schedules as well as the one for n_6 are given in Fig. 3b. Now consider task n_6 for scheduling. It has two parents: n_2 and n_3, and the corresponding data arrival times are $dat(n_6, n_2) = 19$ and $dat(n_6, n_3) = 4$. Hence, n_2 becomes the first element of the array $parents_6$ and n_3 becomes the second one in line 5 of Algorithm 4. $critical\_parent(n_6)$ call in line 6 returns n_2 and n_6 is scheduled on P_6 to start at $dat(n_6, n_2) = 19$. The element after n_2 in $parents_6$ is n_3 with $dat(n_6, n_3) = 4$. Since this is smaller than $eft(n_2) = 5$, the if statement in lines 11 and 12 is true. Then, partial schedule P_2 is copied before task n_6 on P_6, so as to have n_2 finish at $eft(n_2)$, and task n_6 is appended to this partial schedule in line 15. Final view of partial schedule P_6 is given in Fig. 3b.

Scheduling of task n_7 starts with filling $parents_7$ array with n_4, n_6, and n_5, respectively, since $dat(n_7, n_4) = 21$, $dat(n_7, n_6) = 20$, and $dat(n_7, n_5) = 18$. Hence, n_7 is initially scheduled on P_7 to start at 21 and n_4 is assigned to n_p. Then, in line 17 of Algorithm 4, *duplicate* function is called to schedule a copy of n_4 on P_7. In line 3 of Algorithm 5, $critical\_parent(n_4)$ call returns n_2. Then, n_4 is duplicated on P_7 to start at $dat(n_4, n_2) = 11$. In the first for loop in Algorithm 5, $critical\_parent(n_7)$ call returns n_6. Since $dat(n_7, n_6) = 20$ is greater than $ft(n_4, P_7) = 14$, n_7 is shifted to start at 20 on P_7. In the second for loop, the bottleneck task n_b is determined to be n_7, since it is the communication-bounded task with the latest start time on P_7. The snapshot of partial schedule P_7 as well as other relevant partial schedules before the next iteration of the while loop in Algorithm 4 are displayed in Fig. 4a. The while loop is then executed to duplicate n_6 on P_7 and the resulting partial schedule is given in Fig. 4b. Note that $ft(n_7, P_7)$ increased to 24, hence the partial schedule in Fig. 4a is still stored as the best partial schedule. At this point, n_7 is a computation-bounded task and n_6 becomes the bottleneck task. Therefore, the critical parent n_2 of n_6 is duplicated on P_7 as shown in Fig. 4c. Then, n_4 becomes the bottleneck task and its critical parent n_2 is assigned to n_p. Note that even though n_2 has already been scheduled on P_7, it starts execution after n_4. Therefore, it is not useful to satisfy dependency to n_4. Previously scheduled copy of n_2 is removed in line 1 of Algorithm 5 and then n_2 is duplicated to start before n_4 as shown in Fig. 4d. Finally, n_5 and n_1 are scheduled on P_7 to achieve the partial schedules in Figs. 4e and 4f, respectively. When SC is applied after SDS, processor requirement reduces from 7 to 2 for this example.

5.3 Complexity of SDS

The computation of *dat* for each parent of each task and sorting them takes $O(|\mathcal{N}|^3)$ in line 4 of Algorithm 4. At each iteration of the while loop in line 9, exactly one task is duplicated on P_t. However, since a task may be duplicated more than once on P_t, we need the bound in line 21 to limit the theoretical time complexity of the SDS algorithm. In practice, the number of tasks duplicated on a partial schedule is many fewer than $|\mathcal{N}|$. Therefore, even with a κ value of 1, the while loop is unlikely to be broken in line 21 for any partial schedule. Even if it is broken, increase in schedule length is likely to be small. The *duplicate* function in Algorithm 5 traverses tasks on P_t once in each for loop. Since the while loop in line 9 and the for loop in line 2 are also $O(|\mathcal{N}|)$, overall time complexity due to *duplicate* function calls is $O(|\mathcal{N}|^3)$.

For every task n_j duplicated on P_t, idx_j associated with $parents_j$ is reset to 0 in line 2 of Algorithm 5 and it is incremented by later $critical\_parent(n_j)$ calls. In the worst-case, idx_j goes through all elements of $parents_j$, resulting in $O(|\mathcal{N}|)$ complexity for each task duplicated on P_t. Since total number of duplications for a single partial schedule is bounded by $|\mathcal{N}| \times \kappa$ and at most $|\mathcal{N}|$ partial schedules are constructed, overall time complexity of *critical_parent*

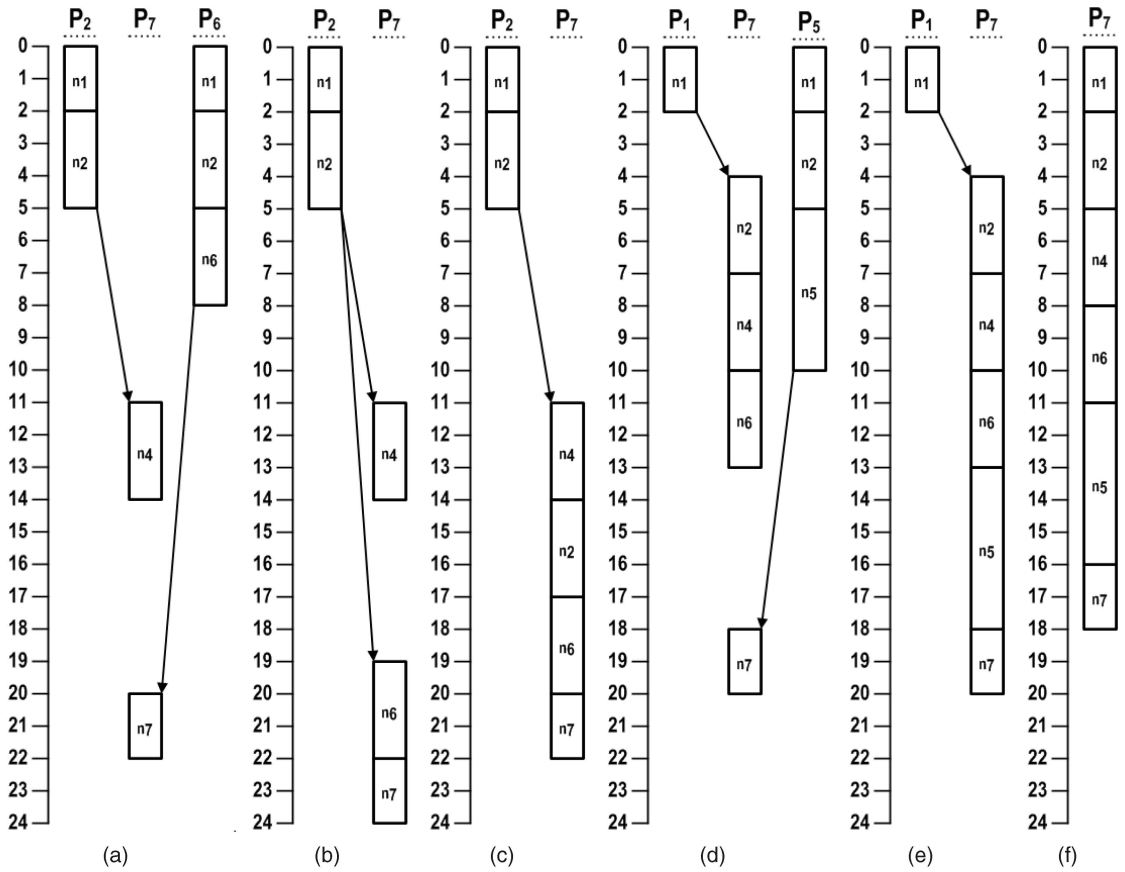

Fig. 4. Snapshots from scheduling steps of partial schedule P_7.

function is $O(|\mathcal{N}|^3)$. Consequently, SDS algorithm has a worst-case time complexity of $O(|\mathcal{N}|^3)$. Since at most $|\mathcal{N}|$ tasks can be scheduled on a single partial schedule, the space complexity of SDS algorithm is $O(|\mathcal{N}|^2)$.

6 Experimental Results

We evaluated the performance of the proposed algorithms on random DAGs as well as DAGs from three application classes. We generated random DAGs with three varying parameters. The first one is the average number of parents of tasks to control dependencies in the DAG. Usually this parameter does not change significantly with problem size for DAGs from the same application domain. However, it may get different values for various 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 time associated with edges to average task computation time in the DAG. As the third parameter, we varied the number of tasks to see the impact of problem size on scheduling quality and processor count.

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 time from interval $(0, 20]$ with uniform probability. Finally, each edge is assigned a communication time from interval $(0, 20 \times CCR]$ to approximate the desired CCR.

We also tested the algorithms on task graphs from LU decomposition (LU) [17], Laplace equation (LE) [4], and Gaussian elimination (GE) [4] 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 and communication times are generated using the same method for random DAG experiments. For every combination of parameter values and for each DAG type, the results in this section correspond to the average results from three distinct DAGs. While presenting a result for a varying parameter, the results are averaged over all tested values of the remaining parameters.

Experimental results for the proposed algorithms are presented in comparison to CPFD [10], PLW [11], and TCSD [22] algorithms. In Figs. 5 and 6, normalized schedule lengths obtained by each algorithm while varying the number of tasks, CCR and the number of parents are presented. Normalized schedule length (NSL) is defined as the ratio of the schedule length obtained by an algorithm to the sum of the computation costs of the tasks on the critical path of the DAG [15]. Note that since NSL is simply a lower bound on the schedule length, an NSL value greater than 1 does not mean that the schedule is not optimal. The results indicate that with varying number of parents for random DAGs and varying number of tasks for all DAGs, CPFD provides the smallest NSL on average. Proposed SDS algorithm also performs well, within only 3 percent of CPFD on the average for our test set. On the other hand, PLW and TCSD are 30 and 12 percent worse than CPFD, respectively, on average. The results are similar

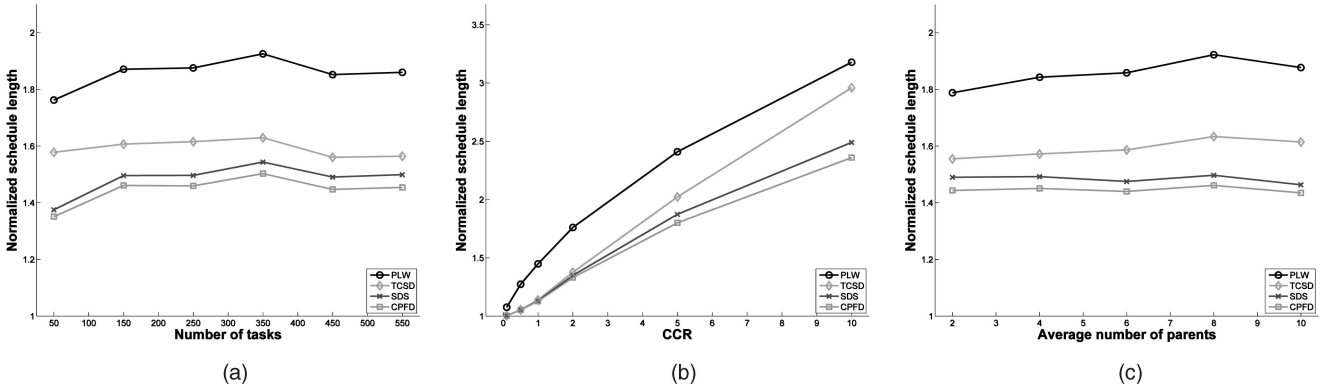

Fig. 5. Normalized schedule length for scheduling random DAGs.

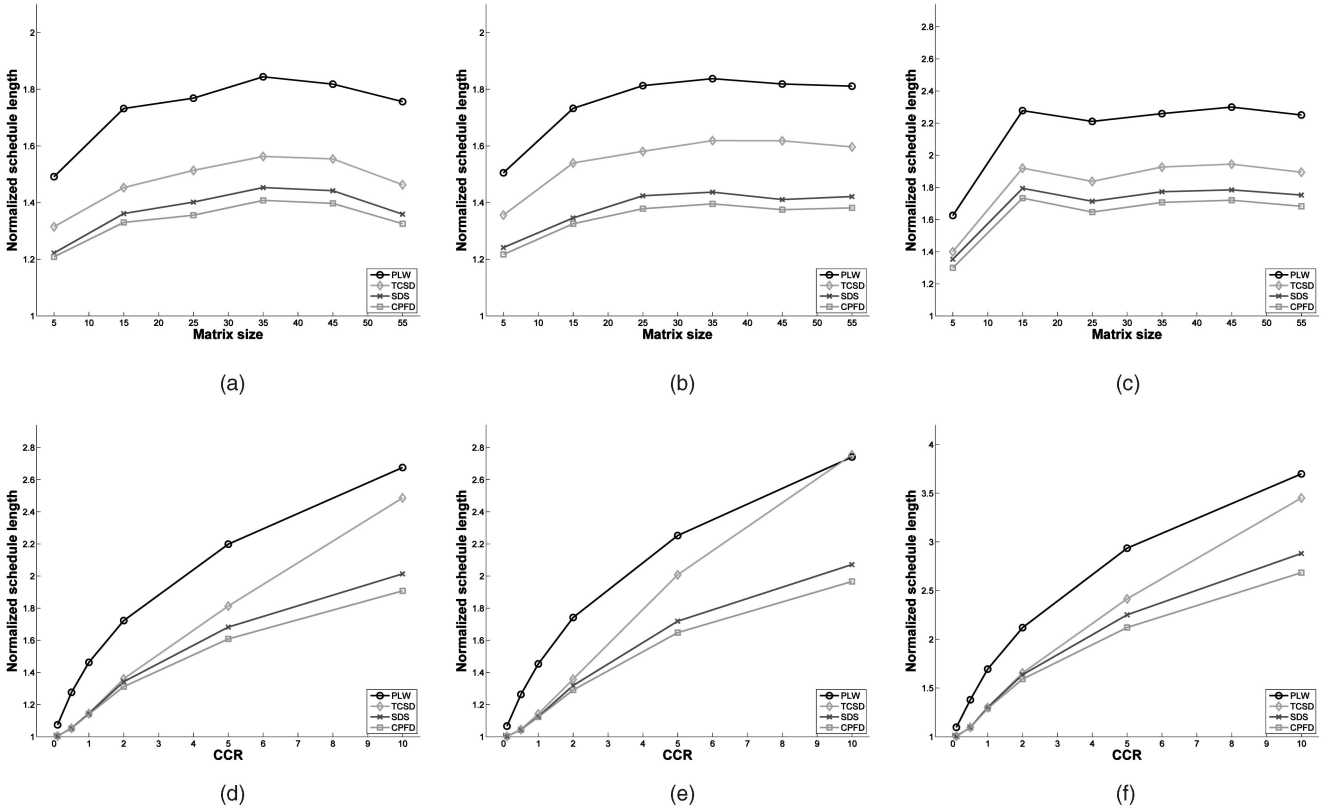

Fig. 6. NSL for application DAGs. (a) and (d) LU decomposition. (b) and (e) Gaussian elimination. (c) and (f) Laplace equation.

when CCR is varied, however, the spread between the algorithms slightly increases with increasing CCR. The reason is that, as communication times get larger compared to computation times in a DAG, task duplication becomes more important to avoid expensive communications. Therefore, algorithms with better task duplication strategies perform even better with increasing CCR. The results show that all algorithms achieve NSL values close to 1 when CCR is 0.1. On the other, extreme with $\mathrm{CCR} = 10$, schedules generated by SDS are 6 percent longer than that by CPFD on average, whereas PLW and TCSD perform 38 and 31 percent worse, respectively. From these experiments, we conclude that SDS clearly outperforms similar time complexity algorithms PLW and TCSD and it is comparable to CPFD which has a considerably higher time complexity.

In the next set of experiments, we evaluate the performance of the proposed SC algorithm in reducing required number of processors. The results of these experiments are given in Figs. 7 and 8. In legends of these figures, X+SC indicates that SC is applied on the schedule generated by algorithm X. Note that reduction in the required number of processors is not attempted within the SDS algorithm since it is designed to be used together with the SC algorithm. Therefore, we do not present the number of processors required by the SDS algorithm alone, which would be equal to the number of tasks in the input DAG.

In Fig. 7a, the number of processors required by schedules generated by PLW, TCSD, and CPFD on random DAGs changes in proportion to the number of tasks. For the three application DAG classes, we consider the number of tasks change with the square of the matrix size. This explains the

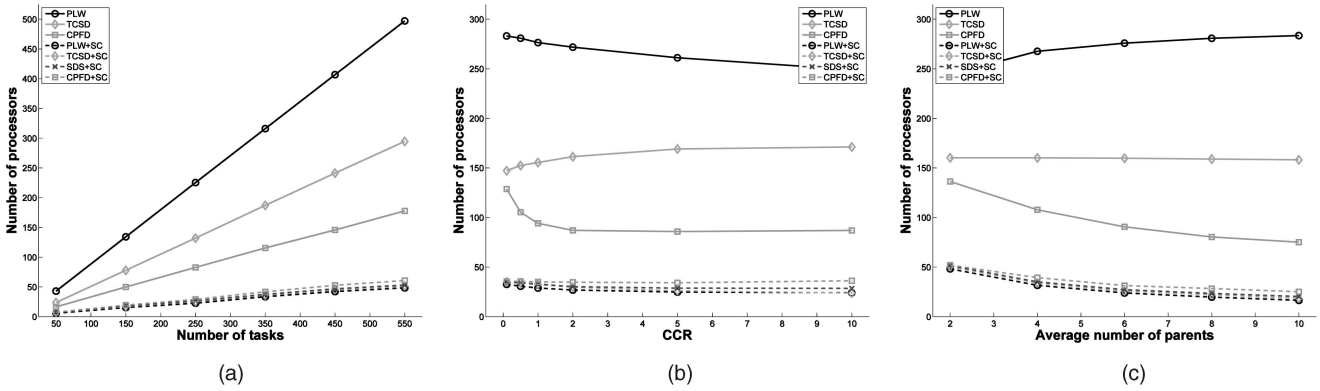

Fig. 7. Number of processors required for scheduling random DAGs.

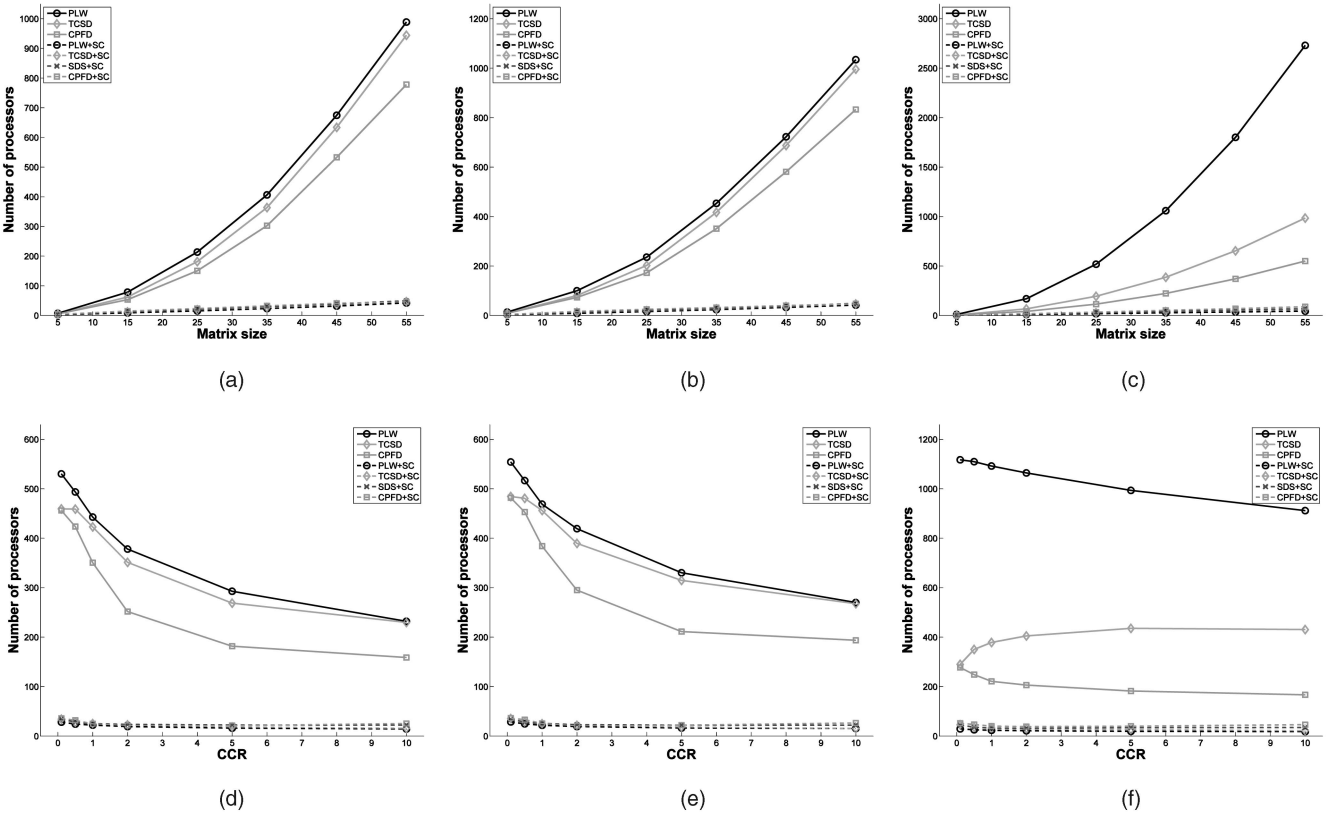

Fig. 8. Number of processors required for scheduling application DAGs. (a) and (d) LU decomposition. (b) and (e) Gaussian elimination. (c)-(f) Laplace equation.

super-linear increase in the processor requirement with increasing matrix size in Figs. 8a, 8b, and 8c. As CCR increases, schedule length usually increases as well leading to more available time on processors to insert tasks. Therefore, processor requirement for all algorithms decreases, or at least, does not increase significantly with increasing CCR as shown in Figs. 7b, 8d, 8e, and 8f. Similar reasoning applies to Fig. 7c, where schedule length tends to increase with increasing number of parents for the random DAGs.

The results indicate that the processor requirement of original CPFD schedules is smaller than TCSD, which, in turn, is smaller than PLW on average. This confirms that CPFD not only generates the shortest schedules, but also efficient in terms of processor usage. TCSD follows CPFD in that respect whereas processor requirement of PLW is very large even though it generates significantly longer schedules. When we apply SC algorithm on the schedules generated by these algorithms, as well as SDS, the processor requirement dramatically reduces, 91 percent for PLW, 82 percent for TCSD, and 72 percent for CPFD schedules averaged over our entire test set. Furthermore, the number of processors required by these schedules becomes inversely proportional to the schedule length, which demonstrates the effectiveness of the SC algorithm. In other words, the compaction mechanism in SC is aggressive enough to fit longer schedules into smaller number of processors and provide a compensation for loss in schedule length by gain in the processor requirement. Another improvement by SC is very slow increase in required number of processors with increasing number of tasks in a DAG. This leads to modest processor requirement even for large applications. For example, a GE

TABLE 2
Experimental Results on Three Real DAGs

DAG	Algorithm	NSL	$\|P\|$	
			w/o SC	w/ SC
Strassen Mat. Mul.	PLW	1.01	12	7
	TCSD	1	15	10
	CPFD	1	11	10
	SDS	1	–	10
TCE-S	PLW	1.02	18	6
	TCSD	1	9	4
	CPFD	1	8	4
	SDS	1	–	4
TCE-D	PLW	1.01	17	4
	TCSD	1	10	3
	CPFD	1	10	4
	SDS	1	–	3

application with matrix size 55 can be scheduled on a medium-sized cluster with 64 processors using CPFD+SC algorithm. Compared to more than 800 processors required by the original CPFD solution, it can be deduced that SC algorithm is an essential step to be able to utilize high-quality solution of CPFD on large scale applications.

We have carried out another set of experiments on three real DAGs to show the effectiveness of the SC algorithm. These DAGs represent real instances of Strassen Matrix Multiplication [38] having 29 tasks, Tensor Contraction Engine [39] with single (TCE-S) and double (TCE-D) excitations having 33 and 24 tasks, respectively. As the results in Table 2 illustrate, SC algorithm reduced the processor requirement significantly in all three cases.

Overall, SDS+SC algorithm with $O(|\mathcal{N}|^3)$ time complexity provides the most desirable solution in terms of time complexity, schedule length as well as the number of processors. PLW+SC and TCSD+SC with complexities of $O(|\mathcal{N}|^3)$ and $O(|\mathcal{N}|^3 log|\mathcal{N}|)$, respectively, require marginally smaller number of processors than SDS+SC, however, the length of schedules generated by these algorithms are significantly larger. CPFD+SC on the other hand generates slightly shorter schedules than SDS+SC. However, in addition to using larger number of processors to do that, it has a significantly larger time complexity of $O(|\mathcal{N}|^4)$ due to CPFD, which introduces a serious handicap especially for large scale applications.

7 Conclusion

We developed a novel algorithm to minimize the processor requirement of schedules generated by any scheduling algorithm. When applied on a valid schedule, the proposed SC algorithm compacts the schedule on fewer number of processors without increasing the schedule length. Experiments on DAGs representing three applications as well as random DAGs verified that SC algorithm dramatically reduces the processor requirement of the schedules generated by CPFD, PLW, and TCSD algorithms. For our test set, the reduction was 91 percent for PLW, 82 percent for TCSD, and 72 percent for CPFD algorithms on average. SC algorithm has a time complexity of $O(|\mathcal{N}|^3)$, which is smaller than most duplication-based scheduling algorithms.

To take advantage of existence of SC that minimizes processor count, we have also proposed SDS algorithm that focuses on minimizing the schedule length only. SDS also has $O(|\mathcal{N}|^3)$ time complexity and we showed that the quality of schedules generated by SDS is very close to CPFD, one of the best algorithms in that respect. Since the time complexity of SDS is smaller than CPFD by $O(|\mathcal{N}|)$, we conclude that SDS+SC provides a highly desirable algorithm combination in terms of small schedule length, low time complexity, and low processor requirement.

Acknowledgments

This research was supported in part by the US National Science Foundation under Grants #CNS-0643969, #CCF-0342615, #CNS-0403342, and the US Department of Energy SciDAC Grant #DE-FC02-06ER2775.

References

[1] A. Gerasoulis and T. Yang, "On the Granularity and Clustering of Directed Acyclic Task Graphs," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 6, pp. 686-701, June 1993.
[2] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software,* vol. 5, no. 1, pp. 23-32, Jan. 1988.
[3] M. Cosnard and M. Loi, "Automatic Task Graph Generation Techniques," *Parallel Processing Letters,* vol. 5, no. 4, pp. 527-538, Dec. 1995.
[4] M.-Y. Wu and D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 3, pp. 330-343, July 1990.
[5] M. Iverson, F. Özgüner, and L. Potter, "Statistical Prediction of Task Execution Times Through Analytical Benchmarking for Scheduling in a Heterogeneous Environment," *IEEE Trans. Computers,* vol. 48, no. 12, pp. 1374-1379, Dec. 1999.
[6] M. Garey and D. Johnson,, *Computers and Intractability, A Guide to the Theory of NP Completeness.* W.H. Freeman and Co., 1979.
[7] S. Darbha and D. Agrawal, "Optimal Scheduling Algorithm for Distributed Memory Machines," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 1, pp. 87-95, Jan. 1998.
[8] C. Park and T. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication," *IEEE Trans. Computers,* vol. 51, no. 4, pp. 444-448, Apr. 2002.
[9] C. Papadimitriou and M. Yannakakis, "Towards an Architecture Independent Analysis of Parallel Algorithms," *SIAM J. Computing,* vol. 19, pp. 322-328, Apr. 1990.
[10] I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 9, pp. 872-892, Sept. 1998.
[11] M. Palis, J. Liou, and D. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 1, pp. 46-54, Jan. 1996.
[12] S. Baskiyar, "Scheduling Task In-Trees on Distributed Memory Systems," *Proc.15th Int'l Parallel and Distributed Processing Symp.,* p. 6, Apr. 2001.
[13] T. Wajdi and A. Imitaz, "Optimal Algorithm for Tree Scheduling with Unit Time Communication Delays," *Proc. IEE Computers and Digital Techniques,* vol. 148, no. 2, pp. 79-88, Mar. 2001.
[14] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing,* vol. 18, no. 2, pp. 244-257, 1989.
[15] Y.-K. Kwok and I. Ahmad, "Dynamic Critical Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 506-521, May 1996.
[16] Y. Chung and S. Ranka, "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," *Proc. Supercomputing,* pp. 512-521, Nov. 1992.
[17] T. Tsuchiya, T. Osada, and T. Kikuno, "Genetics-Based Multiprocessor Scheduling Using Task Duplication," *Microprocessors and Microsystems,* vol. 22, no. 3, pp. 197-207, Aug. 1998.
[18] C.-H. Yang, P. Lee, and Y.-C. Chung, "Improving Static Task Scheduling in Heterogeneous and Homogeneous Computing Systems," *Int'l Conf. Parallel Processing,* pp. 45-45, Sept. 2007.

[19] L. Zhou and S. Shi-Xin, "A Genetic Scheduling Algorithm Based on Knowledge for Multiprocessor System," *Proc. Int'l Conf. Comm. Circuits and Systems,* pp. 900-904, July 2007.
[20] G.-L. Park, B. Shirazi, and J. Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems," *Proc. 11th Int'l Parallel Processing Symp.,* pp. 157-166, Apr. 1997.
[21] D. Bozdag, F. Ozguner, E. Ekici, and U. Catalyurek, "A Task Duplication Based Scheduling Algorithm Using Partial Schedules," *Proc. Int'l Conf. Parallel Processing,* pp. 630-637, June 2005.
[22] G. Li, D. Chen, D. Wang, and D. Zhang, "Task Clustering and Scheduling to Multiprocessors with Duplication," *Proc. Int'l Parallel and Distributed Processing Symp.,* p. 8, 22-26, Apr. 2003.
[23] C. Boeres and V. Robello, "Cluster-Based Static Scheduling: Theory and Practice," *Proc. 14th Symp. Computer Architecture and High Performance Computing,* pp. 133-140, Oct. 2002.
[24] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 9, pp. 951-967, Sept. 1994.
[25] S. Kim and J. Browne, "A General Approach to Mapping of Parallel Computation Upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing,* vol. 3, pp. 1-8, 1988.
[26] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.
[27] J. Colin and P. Chretienne, "C.P.M. Scheduling with Small Computation Delays and Task Duplication," *Operations Research,* pp. 680-684, 1991.
[28] B. Shirazi, H. Chen, and J. Marquis, "Comparative Study of Task Duplication Static Scheduling Versus Clustering and Non-Clustering Techniques," *Concurrency: Practice and Experience,* vol. 7, no. 5, pp. 371-390, Aug. 1995.
[29] Y.-K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel and Distributed Computing,* vol. 59, no. 3, pp. 381-422, Dec. 1999.
[30] Y. Ruan, G. Liu, Q. Li, and T. Jiang, "An Efficient Scheduling Algorithm for Dependent Tasks," *Proc. Fourth Int'l Conf. Computer and Information Technology,* pp. 456-461, Sept. 2004.
[31] M.-Y. Wu, W. Shu, and J. Gu, "Efficient Local Search for Dag Scheduling." *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 6, pp. 617-627, June 2001.
[32] A. Radulescu and A. van Gemund, "Low-Cost Task Scheduling for Distributed-Memory Machines," *IEEE Trans. Parallel and Distributed Systems,* vol. 13, no. 6, pp. 648-658, June 2002.
[33] S. Bansal, P. Kumar, and K. Singh, "An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 6, pp. 533-544, June 2003.
[34] S. Pasham and W.-M. Lin, "Efficient Task Scheduling with Duplication for Bounded Number of Processors," *Proc. 11th Int'l Conf. Parallel and Distributed Systems,* vol. 1, pp. 543-549, July 2005.
[35] H. El-Rewini and T. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *J. Parallel and Distributed Computing,* vol. 9, no. 2, pp. 138-153, June 1990.
[36] G. Sih and E. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 2, pp. 75-87, Feb. 1993.
[37] G. Li, Y. Zhang, Y. Lin, and Y. Huang, "Scalable Duplication Strategy with Bounded Availability of Processors," *Proc. 10th Int'l Conf. Parallel and Distributed Systems,* pp. 267-274, July 2004.
[38] G.H. Golub and C.F.V. Loan, *Matrix Computations,* third ed. Johns Hopkins Univ. Press, 1996.
[39] A.A. Auer et al., "Automatic Code Generation for Many-Body Electronic Structure Methods: The Tensor Contraction Engine," *Molecular Physics,* vol. 104, pp. 211-228, 2006.

Doruk Bozdağ received the BS degrees in electrical and electronic engineering and physics from Boğaziçi University, Turkey, in 2002, and the MS degree in electrical and computer engineering from the Ohio State University in 2005. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering at the Ohio State University. His research interests include scheduling algorithms for multiprocessor systems, high-performance computing, parallel and combinatorial algorithms for scientific computing, and bioinformatics applications.

Füsun Özgüner received the MS degree in electrical engineering from Istanbul Technical University in 1972, and the PhD 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 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, Department of Electrical and Computer Engineering, where she is presently a professor. She served as the interim department chair from September 2004 to 2006. 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, on-chip multiprocessing, and wireless networks. She has served as an associate editor of the *IEEE Transactions on Computers* and program committees of several international conferences. She recently served as the general chair of the 2007 International Conference on Pervasive Services. She is a senior member of the IEEE.

Umit V. Catalyurek received the BS, MS, and PhD degrees in computer engineering and information science from Bilkent University, Turkey, in 1992, 1994, and 2000, respectively. He is an associate 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, runtime systems for data-intensive computing, and high-performance computing in biomedicine.