Supporting Graph-Based Real-Time Applications in Distributed Systems*

Cong Liu and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

The processing graph method (PGM) is a widely used framework for modeling applications with producer/consumer precedence constraints. PGM was originally developed by the U.S. Navy to model signal-processing applications where data communications exist among connected tasks. Prior work has shown how to schedule PGM-specified systems on uniprocessors and globally-scheduled multiprocessors. In this paper, this work is extended to enable such systems to be supported in a distributed collection of multicore machines. In such a context, pure global and partitioned scheduling approaches are problematic. Moreover, data communication costs must be considered. In this paper, a clustered scheduling algorithm is proposed for soft real-time PGM-specified distributed task systems for which bounded deadline tardiness is acceptable. This algorithm is effective in reducing data communication costs with little utilization loss. This is shown both analytically and via experiments conducted to compare it with an optimal integer linear programming solution.

1 Introduction

In work on real-time scheduling in distributed systems, task models where no inter-task precedence constraints exist, such as the periodic and the sporadic task models, have received much attention. However, in many real-time systems, applications are developed using processing graphs [8, 11], where vertices represent sequential code segments and edges represent precedence constraints. For example, multimedia applications and signal-processing algorithms are often specified using directed acyclic graphs (DAGs) [10, 11]. With the growing prevalence of multicore platforms, it is inevitable that such DAG-based real-time applications will be deployed in distributed systems where multicore machines are used as per-node computers. One emerging example application where such a deployment is expected is fractionated satellite systems [9]. Such a system consists of a number of wirelessly-connected small satellites, each of which may be controlled by a multicore machine. The overall collection of such machines is expected to support DAG-based real-time workloads such as radar and signal-processing subsystems. To support such workloads, efficient scheduling algorithms are needed. Motivated by applications such as this, this paper is directed at supporting real-time DAG-based applications in distributed systems.

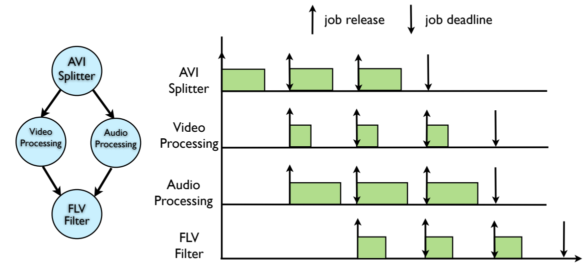

Figure 1: Example DAG.

We view a distributed system as a collection of clusters of processors, where all processors in a cluster are locally connected (i.e., on a multicore machine). A DAG-based task system can be deployed in such a setting by (*i*) assigning tasks to clusters, and (*ii*) determining how to schedule the tasks in each cluster. In addressing (*i*), overheads due to data communications among connected tasks must be considered since tasks within the same DAG may be assigned to different clusters. In addressing (*ii*), any employed scheduling algorithm should seek to minimize utilization loss.

Within a cluster, precedence constraints can easily be supported by viewing all deadlines as hard and executing tasks sporadically (or periodically), with job releases adjusted so that successive tasks execute in sequence. Fig. 1 shows an example multimedia application, which transforms an input AVI video file into a FLV video file (AVI and FLV are two types of multimedia container formats). This application is represented by a DAG with four sporadic tasks. (DAG-based systems are formerly defined in Sec. 2.) Fig. 1 shows a global-earliest-deadline-first (GEDF) schedule for this application on a two-processor cluster. (It suffices to know here that the kth job of the AVI splitter task, the video processing task [or the audio processing task], and the FLV filter task, respectively, must execute in sequence.) As seen in this example, the timing guarantees provided by the sporadic model ensure that any DAG executes correctly as long as no deadlines are missed.

*Work supported by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

Unfortunately, if all deadlines of tasks assigned to the same cluster must be viewed as hard, then significant processing capacity must be sacrificed, due to either inherent schedulability-related utilization loss—which is unavoidable under most scheduling schemes [7]—or high runtime overheads—which typically arise in optimal schemes that avoid schedulability-related loss [4]. In systems where less stringent notions of real-time correctness suffice, such utilization loss can be avoided by viewing deadlines as soft. In this paper, such systems are our focus; the notion of soft real-time correctness we consider is that deadline tardiness is bounded.

To the best of our knowledge, in all prior work on supporting DAG-based applications in systems with multiple processors (multiprocessors or distributed systems), either *global* or *partitioned* scheduling has been assumed. Under global scheduling, tasks are scheduled from a single run queue and may migrate across processors; in contrast, under partitioned schemes, tasks are statically bound to processors and per-processor schedulers are used. Partitioned approaches are susceptible to bin-packing-related schedulability limitations, which global approaches can avoid. Indeed, if bounded deadline tardiness is the timing constraint of interest, then global approaches can often be applied on multiprocessor platforms with no loss of processing capacity [14]. However, the virtues of global scheduling come at the expense of higher runtime overheads. In work on ordinary sporadic (not DAG-based) task systems, clustered scheduling, which combines the advantages of both global and partitioned scheduling, has been suggested as a compromise [3, 6]. Under clustered scheduling, tasks are first partitioned onto clusters of cores, and intra-cluster scheduling is global.

In distributed systems, clustered scheduling algorithms are a natural choice, given the physical layout of such a system. Thus, such algorithms are our focus here. Our specific objective is to develop clustered scheduling techniques and analysis that can be applied to support DAGs, assuming that bounded deadline tardiness is the timing guarantee that must be ensured. Our primary motivation is to develop such techniques for use in distributed systems, where different clusters are physically separated; however, our results are also applicable in settings where clusters are tightly coupled (e.g., each cluster could be a socket in a multi-socket system). Our results can be applied to systems with rather sophisticated precedence constraints. To illustrate this, we consider a particularly expressive DAG-based formalism, the processing graph method (PGM) [13]. PGM was first developed by the U.S. Navy to model signal-processing applications where producer/consumer relationships exist among tasks. In a distributed system, it may be necessary to transfer data from a producer in one cluster to a consumer in another through an inter-cluster network, which could cause a significant amount of data communication overhead. Thus, any proposed scheduling algorithm should seek to minimize such inter-cluster data communication.

Related work. To our knowledge, DAGs have not been considered before in the context of clustered real-time scheduling algorithms, except for the special cases of partitioned and global approaches.[1] An overview of work on scheduling DAGs in a distributed system under partitioned approaches (which we omit here due to space constraints) can be found in [15]. The issue of scheduling PGM graphs on a uniprocessor was extensively considered by Goddard in his dissertation [11]. Goddard presented techniques for mapping PGM nodes to tasks in the rate-based-execution (RBE) task model [12], as well as conditions for verifying the schedulability of the resulting task set under a rate-based, earliest-deadline-first (EDF) scheduler.

In recent work [15], we extended Goddard's work and showed that a variety of global scheduling algorithms can ensure bounded deadline tardiness in general DAG-based systems with no utilization loss on multiprocessors, including algorithms that are less costly to implement than optimal algorithms.

Contributions. In this paper, we show that sophisticated notions of acyclic precedence constraints can be efficiently supported under clustered scheduling in a distributed system, provided bounded deadline tardiness is acceptable. The types of precedence constraints we consider are those allowed by PGM. We propose a clustered scheduling algorithm called CDAG that first partitions PGM graphs onto clusters, and then uses global scheduling approaches within each cluster. We present analysis that gives conditions under which each task's maximum tardiness is bounded. Any clustered approach is susceptible to some bin-packing-related utilization loss; however, the conditions derived for CDAG show that for it, such loss is small. To assess the effectiveness of CDAG in reducing inter-cluster data communications, we compare it with an optimal integer linear programming (ILP) solution that minimizes inter-cluster data communications when partitioning PGM graphs onto clusters. We assume that tasks are specified using a rate-based task model that generalizes the periodic and sporadic task models.

Organization. The rest of this paper is organized as follows. Sec. 2 describes our system model. In Sec. 3, the ILP formulation of the problem and the polynomial-time CDAG algorithm and its analysis are presented. Sec. 4 evaluates the proposed algorithm via an experimental study. Sec. 5 concludes.

[1]The problem of non-real-time DAG scheduling in parallel and distributed systems has been extensively studied. An overview on such work can be found in [16].

2 Preliminaries

In this section, we present an overview of PGM and describe the assumed system architecture. For a complete description of the PGM, please see [13].

PGM specifications. An acyclic PGM graph system [13] consists of a set of acyclic PGM graphs (DAGs), each with a distinct source node. Each PGM graph contains a number of nodes connected by edges. A node can have outgoing or incoming edges. A source node, however, can only have outgoing edges. Each directed edge in a PGM graph is a typed first-in-first-out queue, and all nodes in a graph are assumed to be reachable from the graph's source node. A producing node transports a certain number of data units[2] to a consuming node, as indicated by the data type of the corresponding queue. Data is appended to the tail of the queue by the producing node and read from the head by the consuming node. A queue is specified by three attributes: a *produce* amount, a *threshold*, and a *consume* amount. The produce amount specifies the number of data units appended to the queue when the producing node completes execution. The threshold amount specifies the minimum number of data units required to be present in the queue in order for the consuming node to process any received data. The consume amount is the number of data units dequeued when processing data. We assume that the queue that stores data is associated with the consuming node. That is, data is stored in memory local to the corresponding consuming node.[3] The only restriction on queue attributes is that they must be non-negative integral values and the consume amount must be at most the threshold. In the PGM framework, a node is *eligible* for execution when the number of data units on each of its input queues is over that queue's threshold. Overlapping executions of the same node are disallowed. For any queue connecting nodes T_l^j and T_l^k in a PGM graph T_l, we let ρ_l^{jk} denote its produce amount, θ_l^{jk} denote its threshold, and σ_l^{jk} denote the consume amount. If there is an edge from task T_l^k to task T_l^h in graph T_l, then T_l^k is called a *predecessor task* of T_l^h. We let $pred(T_l^h)$ denote the set of all predecessor tasks of T_l^h. We define the *depth* of a task within a graph to be the number of edges on the longest path between this task and the source task of the corresponding graph.

Example. Fig. 2(a) shows an example PGM graph system consisting of two graphs T_1 and T_2 where T_1 contains four nodes with four edges and T_2 contains two nodes with one edge. We will use this example to illustrate other concepts throughout the paper.

In the PGM framework, it is often assumed that each source node executes according to a rate-based pattern (see below). Note that, even if all source nodes execute according to a periodic/sporadic pattern, non-source nodes may still execute following a rate-based pattern [11, 15]. The execution rate of a node within a PGM graph can be calculated based upon the attributes of the node's incoming edges using techniques presented in [11, 15].

[2]We assume that data units are defined so that the *number* of such units produced reflects the total *size* of the data produced.

[3]It is more reasonable to associate a queue with the corresponding consuming node than with the producer node because this enables the consumer to locally detect when the queue is over threshold and react accordingly.

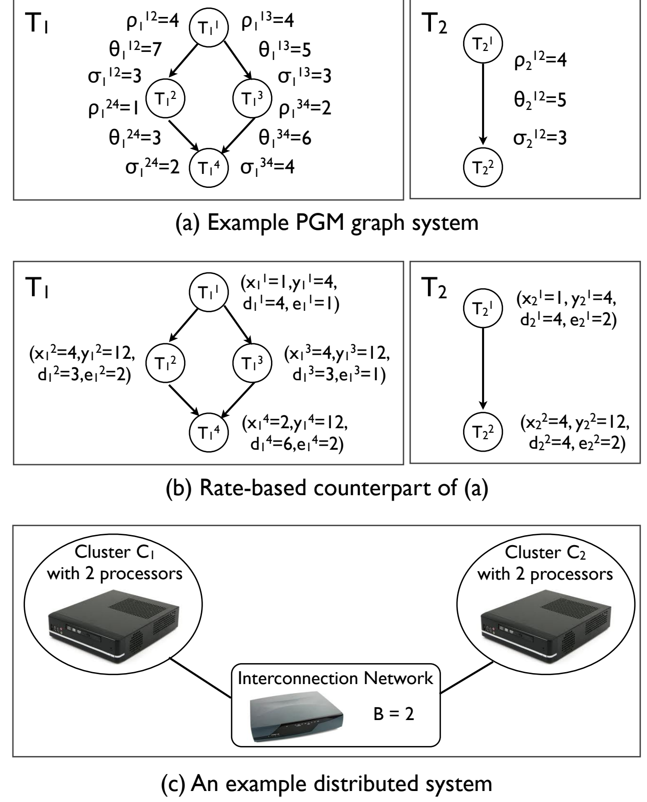

Figure 2: Example system used throughout the paper.

According to the rate-based task model [12,15], each task T_l^h within a PGM graph T_l is specified by four parameters: $(x_l^h, y_l^h, d_l^h, e_l^h)$. The pair (x_l^h, y_l^h) represents the maximum execution rate of T_l^h: x_l^h is the maximum number of invocations of the task in any interval $[j \cdot y_l^h, (j+1) \cdot y_l^h)$ $(j \geq 0)$ of length y_l^h; such an invocation is called a *job* of T_l^h. x_l^h and y_l^h are assumed to be non-negative integers. Additionally, $d_l^h = y_l^h / x_l^h$ is the task's relative deadline, and e_l^h is its worst-case execution time. The jth job of T_l^h is denoted $T_{l,j}^h$. We denote its release time (when T_l^h is invoked) as $r_{l,j}^h$ and its (absolute) deadline as $d_{l,j}^h = r_{l,j}^h + d_l^h$. The *utilization of* T_l^h, u_l^h, is defined to be $e_l^h \cdot \frac{x_l^h}{y_l^h}$. The *utilization of a PGM graph* T_l is defined to be $u_l = \sum_{T_l^h \in T_l} u_l^h$. The widely-studied sporadic task model is a special case of the rate-based task model. In the sporadic task model, a task is released no sooner than every p time units, where p is the task's period. In the rate-based task model, the notion of a "rate" is much more general.

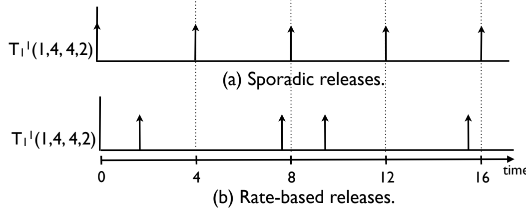

Figure 3: Sporadic and rate-based releases.

Example. Fig. 2(b) shows the rate-based counterpart of the PGM graphs in Fig. 2(a). Fig. 3 shows job release times for task $T_1^1(1,4,4,1)$ of graph T_1. In inset (a), jobs are released sporadically, once every four time units in this case. Inset (b) shows a possible job-release pattern that is not sporadic. As seen, the second job is released at time 7.5 while the third job is released at time 9.5. The separation time between these jobs is less than that seen in the sporadic schedule.

A job $T_{l,j}^h$ may be restricted from beginning execution until certain jobs of tasks in $pred(T_l^h)$ have completed (i.e., $T_{l,j}^h$ cannot start execution until receiving the required data from these jobs). We denote the set of such *predecessor jobs* as $pred(T_{l,j}^h)$. $pred(T_{l,j}^h)$ can be defined precisely by examing the execution rates of all tasks on any path from the source node of T_l to T_l^h [15].

If a job $T_{l,j}^h$ completes at time t, then its *tardiness* is defined as $max(0, t - d_{l,j}^h)$ and its *response time* is defined as $max(0, t - r_{l,j}^h)$. A PGM graph's tardiness is the maximum of the tardiness of any job of any of its tasks. Note that, when a job of a task misses its deadline, the release time of the next job of that task is not altered. However, jobs of the same task still cannot execute in parallel.

System architecture. We consider the problem of scheduling a set of n acyclic PGM graphs on φ clusters $C_1, C_2, ..., C_\varphi$. Each cluster C_i contains λ_i processors. Clusters are connected by a network. Let B denote the minimum number of data units that can be transferred between any two clusters per time unit. Similarly, let b denote the minimum number of data units that can be transferred between any two processors within the same cluster per time unit. If the system is a distributed collection of multicore machines, then B is impacted by the speed and bandwidth of the communication network and b is impacted by the speed of the data transfer bus on a multicore chip. In this case, $b \gg B$, as local on-chip data communication is generally much faster than communication across a network.

Example. An example distributed system containing two clusters each with two processors interconnected by a network is shown in Fig. 2(c).

3 Algorithm CDAG

In this section, we propose Algorithm CDAG, a clustered-scheduling algorithm that ensures bounded tardiness for DAG-based systems on distributed clusters. Since inter-cluster data communication can be expensive, CDAG is designed to reduce such communication.

CDAG consists of two phases: an *assignment phase* and an *execution phase*. The assignment phase executes offline and assigns each PGM graph to one or more clusters. In the execution phase, PGM graphs are first transformed to ordinary sporadic tasks (where no precedence constraints arise), and then scheduled under a proposed clustered scheduling algorithm.

3.1 Assignment Phase

The assignment phase assigns acyclic PGM graphs (or DAGs for short) to clusters in a way such that the inter-cluster data communication cost is reduced. Note that the total utilization of the DAGs (or portions of DAGs) assigned to a cluster must not exceed the total capacity of that cluster. CDAG contains an assignment algorithm that is designed to partition DAGs onto clusters such that both the inter-cluster data communication cost and any bin-packing-related utilization loss are minimized.

To provide a better understanding of the problem of partitioning DAGs onto clusters with minimum inter-cluster data communication cost, we first formulate it as an ILP, which provides an optimal solution. (Note that, due to potential utilization loss arising from partitioning DAGs, the ILP approach may not find a feasible assignment. However, if the ILP approach cannot find a feasible solution, then the given task set cannot be assigned to clusters under any partitioning algorithm.)

Definition 1. For any edge e_i^{jk} of DAG T_i, its *edge data weight*, w_i^{jk}, is defined to be $\rho_i^{jk} \cdot \frac{x_i^j}{y_i^j}$. A larger edge data weight indicates a larger data communication cost between tasks connected by the corresponding edge. For any edge e_i^{jk} of DAG T_i , its *communication cost* ϖ_i^{jk} is defined to be w_i^{jk} if the corresponding connected nodes T_i^j and T_i^k are assigned to different clusters, and 0 otherwise.

Note that the above definition does not consider data consuming rates. This is because data is stored in memory local to the consuming node, as discussed in Sec. 2. Thus, only produced data needs to be transferred.

Example. For task T_1^1 of DAG T_1 in Fig. 2, we can use previous techniques [11, 15] to calculate its execution rates, which are $x_1^1 = 1$ and $y_1^1 = 4$. Thus, for edge e_1^{12} of T_1, its edge data weight $w_1^{12} = \rho_1^{12} \cdot \frac{x_1^1}{y_1^1} = 4 \cdot \frac{1}{4} = 1$. Intuitively, node T_1^1 produces one data unit on average on edge e_1^{12} per time unit, given its execution rate $(x_1^1, y_1^1) = (1, 4)$.

Definition 2. The *total communication cost* of any given DAG-based system τ, denoted ϖ_{sum}, is given by $\sum_{T_i \in \tau} \sum_{e_i^{jk} \in T_i} \varpi_i^{jk}$.

ILP formulation. We are given a set τ of tasks (each task corresponds to a node in a DAG) and a set ξ of clusters. To reduce clutter in the ILP formulation, we denote tasks more simply as $T_1, T_2, ...$, and let $w_{i,j}$ be the data weight of the edge connecting tasks T_i and T_j ($w_{i,j} = 0$ if i and j are not connected). (This simplified notation is used only for the ILP formulation.)

For all $T_i \in \tau$, let $x_{i,k}$ be a binary decision variable that equals 1 when task T_i is assigned to cluster C_k, and 0 otherwise. For all $(T_i, T_j) \in \tau \times \tau$, let $y_{i,j}$ be a binary decision variable that equals 1 if tasks T_i and T_j are assigned to the same cluster, and 0 otherwise.

Our goal is to minimize the total communication cost. An ILP formulation of this optimization problem is then:

Minimize

$$\sum_{i\in\tau}\sum_{j\in\tau} w_{i,j}\cdot(1-y_{i,j}) \qquad (1)$$

subject to the constraints below. Note that by Defs. 1 and 2, (1) represents the total communication cost.

- Each task must be assigned to one cluster:

$$\sum_{C_k\in\xi} x_{i,k} = 1, \forall T_i \in \tau.$$

- The total utilization of all tasks assigned to a cluster must not exceed the total capacity of that cluster:

$$\sum_{T_i\in\tau} x_{i,k}\cdot u_i \leq \lambda_k, \forall C_k \in \xi.$$

- $y_{i,j}$ should be 1 when two tasks are assigned to the same cluster, and 0 otherwise:

$$y_{i,j} \leq x_{i,k} - x_{j,k} + 1, \forall (T_i, T_j) \in \tau\times\tau, \forall C_k \in \xi,$$

$$y_{i,j} \leq -x_{i,k} + x_{j,k} + 1, \forall (T_i, T_j) \in \tau\times\tau, \forall C_k \in \xi.$$

By solving the ILP above, we obtain an optimal assignment that gives the minimum total communication cost as long as there exists a feasible assignment.

Example. Consider assigning DAGs T_1 and T_2 in Fig. 2 to two clusters. T_1 has a utilization of $1/4+2/3+1/3+1/3 = 19/12$ and T_2 has a utilization of $1/2 + 2/3 = 7/6$. By formulating this assignment problem as an ILP according to the above approach, an optimal solution is to assign all tasks of T_1 to the first cluster and all tasks of T_2 to the second cluster, which leads to $\varpi_{sum} = 0$.

A polynomial-time assignment algorithm. Although the ILP solution is optimal, it has exponential time complexity. We now propose a polynomial-time algorithm to assign DAGs to clusters. This algorithm tries to minimize the total communication cost, which is achieved by locally minimizing communication costs when assigning each DAG.

Definition 3. For any task T_i^j of DAG T_i, its *task data weight*, w_i^j, is defined to be $\sum_{T_i^j\to T_i^k} w_i^{jk}$, where $T_i^j \to T_i^k$ denotes that T_i^j has an outgoing edge to T_i^k.

ASSIGN

$u(C_i)$: CAPACITY OF CLUSTER C_i, INITIALLY $u(C_i) = \lambda_i$

ξ: A LIST OF CLUSTERS $\{C_1, C_2, ..., C_\varphi\}$

ζ: A LIST OF DAGS $\{T_1, T_2, ..., T_n\}$

PHASE 1:
1 Order DAGs in ζ by largest average data weight first
2 Order clusters in ξ by smallest capacity first
3 **for** each DAG T_i in ζ in order
4   **for** each cluster C_j in ξ in order
5   **if** $u_i \leq u(C_j)$
6   Assign all tasks of T_i to C_j
7   Remove T_i from ζ; $u(C_j) := u(C_j) - u_i$

PHASE 2:
8 **for** each DAG T_i in ζ in order
9   Order tasks within T_i by smallest depth first, then order tasks within T_i and at the same depth by largest task data weight first
10 Order clusters in ξ by largest capacity first
11 **for** each task T_i^k of DAG T_i in ζ in order
12   **for** each cluster C_j in ξ in order
13   **if** $u_i^k \leq u(C_j)$ **then**
14   Assign T_i^k to C_j; $u(C_j) := u(C_j) - u_i^k$
15   **else** Remove C_j from ξ

Figure 4: Psuedocode of the assignment algorithm.

Definition 4. For any DAG T_i, its *average data weight*, w_i, is defined to be $\frac{\sum_{e_i^{jk}\in T_i} w_i^{jk}}{E_i}$, where E_i is the total number of edges in T_i. A DAG T_i's *data weight* is defined to be $\sum_{e_i^{jk}\in T_i} w_i^{jk}$.

Example. For DAG T_1 in Fig. 2, by Def. 1, $w_1^{12} = w_1^{13} = 1$, $w_1^{24} = 1/3$, and $w_1^{34} = 2/3$. Thus, by Def. 4, T_1 has an average data weight of $\frac{3}{4}$. By Def. 3, task T_1^1 of T_1 has a data weight of $w_1^1 = w_1^{12} + w_1^{13} = 2$. Intuitively, node T_1^1 produces two data units on average on its outgoing edges per time unit, given its execution rate $(x_1^1, y_1^1) = (1, 4)$.

The proposed DAG assignment algorithm, denoted *ASSIGN*, is shown in Fig. 4.

Algorithm description. Algorithm *ASSIGN* assigns DAGs to clusters in two phases. In the first phase (lines 1-7), it assigns DAGs in largest-average-data-weight-first order to clusters in smallest-capacity-first order, which gives a higher possibility for DAGs with larger average data weight to be fully assigned to a cluster. DAGs that cannot be assigned to clusters in the first phase are considered in the second phase (lines 8-15). For each unassigned DAG in order, its tasks are ordered by depth and then tasks at the same depth are ordered by data weight, which gives tasks with larger data weight a greater chance to be assigned to the same cluster as their predecessor tasks (lines 8-9). Then each task in order is assigned to clusters in largest-capacity-first order (lines 10-15). Task T_i^k is assigned to cluster C_j if T_i^k can receive

its full share of its utilization from C_j (lines 13-14). If not, then C_j is excluded from being considered for any of the later tasks, and the next cluster in order will be considered for scheduling T_i^k (line 15).

Example. Consider the example DAG system in Fig. 2 to be partitioned under the proposed algorithm. By Defs. 1 and 2, T_1 has an average data weight of $3/4$ and T_2 has an average data weight of 1. Thus, T_2 is ordered before T_1. Since T_2 has a utilization of $7/6$ and T_1 has a utilization of $19/12$, the tasks of T_2 are assigned to the first cluster, and the tasks of T_1 are assigned to the second cluster, which leads to $\varpi_{sum} = 0$.

Time complexity. The time complexity of Phase 1 of *ASSIGN* depends on (*i*) the sorting process (lines 1-2), which is $O(n \cdot logn + \varphi \cdot log\varphi)$, and (*ii*) the two **for** loops (lines 3-4), which is $O(N \cdot \varphi)$, where N is the number of tasks in the system (each task corresponds to a node in a DAG). Thus, Phase 1 has a time complexity of $O(n \cdot logn + \varphi \cdot log\varphi + N \cdot \varphi)$. The time complexity of Phase 2 of *ASSIGN* depends on (*i*) the sorting process (lines 9-10), which is $O(n \cdot \mu \cdot log\mu + \varphi \cdot log\varphi)$, where μ is the maximum number of tasks per-DAG, and (*ii*) the two **for** loops (lines 11-12), which is $O(N \cdot \varphi)$. Thus, Phase 2 has a time complexity of $O(n \cdot \mu \cdot log\mu + \varphi \cdot log\varphi + N \cdot \varphi)$. The overall time complexity of *ASSIGN* is thus $O(n \cdot logn + \varphi \cdot log\varphi + n \cdot \mu \cdot log\mu + N \cdot \varphi)$.

Partitioning condition. The following theorem gives a condition for *ASSIGN* to successfully partition any given DAG-based task system onto clusters. For conciseness, let us denote tasks (each task corresponds to a node in a DAG) after ordering by $T_1, T_2, ..., T_N$ (note that tasks within DAGs that are fully assigned to clusters in the first phase are assumed to be ordered before all other tasks here). Let $u(T_i)$ denote the utilization of task T_i under this notation. Before stating the theorem, we first prove the following lemma.

Lemma 1. *Under Algorithm ASSIGN, if a task T_i is the first task that cannot be assigned to any cluster, then $\sum_{k=1}^{i} u(T_k) > m - u_{\varphi-1}$, where $u_{\varphi-1}$ is the sum of $\varphi - 1$ largest task utilizations.*

Proof. Due to the fact that some task T_i cannot be assigned to any cluster, the second phase of Algorithm *ASSIGN* is executed. In the second phase, if Algorithm *ASSIGN* fails to assign the ith task T_i to any cluster, then the last cluster C_φ does not have enough capacity to accommodate T_i. Moreover, for each previous cluster C_j, where $j \leq \varphi - 1$, there exists a task, denoted T^j, that could not be assigned to C_j, and thus the next cluster in order was considered to accommodate T^j and C_j was removed from being considered again for any of the later tasks (line 15). That is, for each such cluster C_j, its remaining capacity is strictly less than the utilization of T^j (for the last cluster, we know that T^φ is T_i). Thus, for any cluster C_j, its allocated capacity is strictly greater than $\lambda_j - u(T^j)$. Since tasks $\{T_1, T_2, ..., T_{i-1}\}$ have been successfully assigned, the total utilization of these

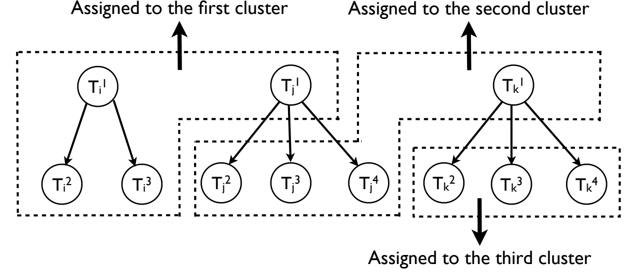

Figure 5: Example worst-case scenario where all edges of $\varphi - 1$ DAGs contribute to the total communication cost.

tasks is equal to the total allocated capacity of clusters, which is given by $\sum_{k=1}^{i-1} u(T_k)$. Hence, we have

$$\begin{aligned}
& \textstyle\sum_{k=1}^{i-1} u(T_k) > \sum_{j=1}^{\varphi}(\lambda_j - u(T^j)) \\
\Leftrightarrow\ & \{\text{adding } u(T_i) \text{ on both sides}\} \\
& \textstyle\sum_{k=1}^{i} u(T_k) > \sum_{j=1}^{\varphi}(\lambda_j - u(T^j)) + u(T_i) \\
\Leftrightarrow\ & \{\text{because } \textstyle\sum_{j=1}^{\varphi} \lambda_j = m \text{ and } u(T^\varphi) = u(T_i)\} \\
& \textstyle\sum_{k=1}^{i} u(T_k) > m - \sum_{j=1}^{\varphi-1} u(T^j) \\
\Rightarrow\ & \{\text{by the definition of } u_{\varphi-1}\} \\
& \textstyle\sum_{k=1}^{i} u(T_k) > m - u_{\varphi-1}.
\end{aligned}$$

□

Theorem 1. *Algorithm ASSIGN successfully partitions any DAG-based task system τ on φ clusters for which $u_{sum} \leq m - u_{\varphi-1}$.*

Proof. Let us suppose that Algorithm *ASSIGN* fails to assign the ith task T_i to any cluster. Then by Lemma 1, $\sum_{k=1}^{i} u(T_k) > m - u_{\varphi-1}$ holds. Therefore, we have

$$\begin{aligned}
& \textstyle\sum_{k=1}^{i} u(T_k) > m - u_{\varphi-1} \\
\Rightarrow\ & \textstyle\sum_{k=1}^{N} u(T_k) = u_{sum} > m - u_{\varphi-1}.
\end{aligned}$$

Hence, any system that Algorithm *ASSIGN* fails to partition must have $u_{sum} > m - u_{\varphi-1}$. □

If φ is much smaller than m, which will often be the case in practice, then the proposed assignment algorithm results in little utilization loss even in the worst case.

Bounding ϖ_{sum}. For any given DAG system, if all DAGs can be assigned in the first phase, then $\varpi_{sum} = 0$. In the second phase, each DAG is considered in order, and if a cluster fails to accommodate a task (line 15), then it will never be considered for later tasks. Thus, it immediately follows that at most $\varphi - 1$ DAGs can contribute to ϖ_{sum}. Due to the fact that in the worst case all edges of a DAG can cause inter-cluster communication (as illustrated by the example below), an upper-bound on ϖ_{sum} under Algorithm *ASSIGN* is given by the sum of $\varphi - 1$ largest DAG data weights.

Example. Consider a scenario where three DAGs T_i, T_j, and T_k are assigned to three clusters in a way as shown in Fig. 5. Note that all edges of $\varphi - 1 = 2$ DAGs T_j and T_k contribute to ϖ_{sum}.

3.2 Scheduling Phase

After executing the assignment phase, every task is mapped to a cluster. The scheduling phase of CDAG ensures that each task is scheduled with bounded tardiness. The scheduling phase consists of two steps: (*i*) transform each PGM graph into ordinary sporadic tasks by redefining job releases, and (*ii*) apply any window-constrained scheduling policy [14] such as GEDF to globally schedule the transformed tasks within each cluster.

Transforming PGM graphs into sporadic tasks. Our recent work has shown that on a multiprocessor, any PGM graph system can be transformed into a set of ordinary sporadic tasks without utilization loss (see [15] for details). The transformation process ensures that all precedence constraints in the original PGM graphs are met. This is done by redefining job releases properly. However, data communication delays (inter-cluster or intra-cluster) were not considered in this previous work. In this paper, for each cluster, we apply the same approach but redefine job releases in a way such that data communications are considered. Later we shall show that this process still ensures bounded tardiness for any graph.

Definition 5. Let $F_{max}(pred(T_{l,j}^h), \upsilon_{l,j}^h)$ denote the latest completion time plus the data communication time among all predecessor jobs of $T_{l,j}^h$, where $\upsilon_{l,j}^h$ denotes the time to transfer data from the corresponding predecessor job of $T_{l,j}^h$ to $T_{l,j}^h$. For any predecessor job $T_{l,i}^k$ of $T_{l,j}^h$, $\upsilon_{l,j}^h$ can be computed by dividing ρ_l^{kh} (the number of produced data units on the corresponding edge) by the corresponding network bandwidth (i.e., B for inter-cluster data communications and b for intra-cluster data communications).

The following equations can be applied to redefine job releases and deadlines in an iterative way (job $T_{l,j}^h$'s redefined release depends on the redefined release of $T_{l,j-1}^h$ where $j > 1$).

For any job $T_{l,j}^h$ where $j > 1$ and $h > 1$, its redefined release time, denoted $r(T_{l,j}^h)$, is given by

$$r(T_{l,j}^h) = max\big(r_{l,j}^h, r(T_{l,j-1}^h) + d_l^h, F_{max}(pred(T_{l,j}^h), \upsilon_{l,j}^h)\big). \quad (2)$$

Given that a source task has no predecessors, the redefined release of any job $T_{l,j}^1$ ($j > 1$) of such a task, $r(T_{l,j}^1)$, is given by

$$r(T_{l,j}^1) = max\big(r_{l,j}^1, r(T_{l,j-1}^1) + d_l^1\big). \quad (3)$$

For the first job $T_{l,1}^h$ ($h > 1$) of any non-source task, its redefined release, $r(T_{l,1}^h)$, is given by

$$r(T_{l,1}^h) = max\big(r_{l,1}^h, F_{max}(pred(T_{l,j}^h), \upsilon_{l,j}^h)\big). \quad (4)$$

Finally, for the first job $T_{l,1}^1$ of any source task, its release time is not altered. (When redefining job releases in our previous work, the term $\upsilon_{l,j}^h$ did not appear in Eqs. (2)-(4) since data communications were not considered.)

After redefining job releases according to (2)-(4), any job $T_{l,j}^h$'s redefined deadline, denoted $d(T_{l,j}^h)$, is given by

$$d(T_{l,j}^h) = r(T_{l,j}^h) + d_l^h.$$

Note that these definitions imply that each task's utilization remains unchanged. In particular, as shown in Sec. 3.3, bounded tardiness can be ensured for every transformed task in any cluster. Thus, Eqs. (2)-(4) delay any job release by a bounded amount, which implies that the execution rate and the relative deadline of each task is unchanged. Note also that the release time of any job $T_{l,j}^h$ with predecessor jobs is redefined to be at least $F_{max}(pred(T_{l,j}^h), \upsilon_{l,j}^h)$. Hence, the schedule preserves the precedence constraints enforced by the PGM model. Furthermore, since the release time of each $T_{l,j}^h$ ($j > 1$) is redefined to be at least that of $T_{l,j-1}^h$ plus d_l^h, T_l executes as a sporadic task with a period of d_l^h.

Example. Suppose that DAG T_2 in Fig. 2 is to be assigned to clusters in a way such that T_1^1 and T_1^2 are assigned to different clusters. For any job $T_{2,j}^2$ of T_2, its predecessor job is $T_{2,j}^1$. Thus, assuming $B = 2$ as in Fig. 1(c), by Def. 5, for any job $T_{2,j}^2$, we have $\upsilon_{2,j}^2 = \frac{\rho_2^{12}}{B} = \frac{4}{2} = 2$. Fig. 6(a) shows the original job releases for T_2^1 and T_2^2 and Fig. 6(b) shows the redefined job releases according to Eqs. (2)-(4) and the corresponding job executions. (Insets (c) and (d) are considered later.) Given that job $T_{2,1}^1$ completes at time 4, according to Eq. (4), the release of $T_{2,1}^2$ is redefined to be at time 6. According to Eq. (3), the release of $T_{2,2}^1$ is redefined to be at time 6. Then, $T_{2,2}^1$ completes at time 8. According to Eq. (2), the release of $T_{2,2}^2$ is redefined to be at time 10, which is the completion time of its predecessor job $T_{2,2}^1$ plus the data communication time. Similarly, releases of other jobs can be defined by Eqs. (2)-(4). Note that the redefined job releases are in accordance with the sporadic task model. Moreover, T_2^1 and T_2^2 execute as if they were ordinary sporadic tasks, and yet all precedence constraints are satisfied.

3.3 Tardiness Bound

Given a PGM-specified system, by applying the strategy presented above, we obtain a transformed task system τ containing only independent sporadic tasks. Then, we can use any window-constrained global scheduling algorithm [14] to schedule tasks within each cluster. It has been shown in [14] that any window-constrained global scheduling algorithm can ensure bounded tardiness for sporadic tasks on multiprocessors with no utilization loss.

In our previous work [15], we derived a tardiness bound for any PGM system scheduled on a multiprocessor (which

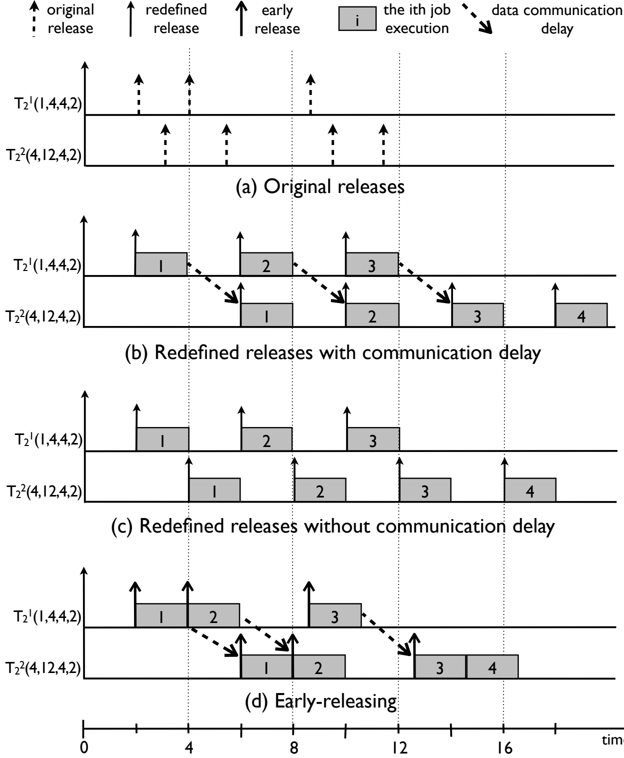

Figure 6: Illustrating various ideas on redefining job releases for DAG T_2 in Fig. 2.

can be considered as a single cluster, as a special case of our multi-cluster system) under GEDF, without considering the communication time, as stated in the following theorem.

Theorem 2. *[15] The tardiness of any job $T_{l,j}^h$ of any task T_l^h at depth k within a DAG T_l scheduled under GEDF on a multiprocessor is at most $(k+1)\cdot\Delta + 3(k+1)\cdot y_l^{max}$, where $y_l^{max} = max(y_l^1, y_l^2, ..., y_l^z)$ (z is the number of nodes within T_l) and Δ denotes the tardiness bound of T_l with respect to its redefined deadlines, as specified in [15] (omitted here due to space constraints).*

Fig. 6(b) shows the redefined releases and the job executions after considering data communication times (as covered earlier). Fig. 6(c) shows the redefined releases and the corresponding job executions assuming no data communication time for DAG T_2. As seen, the data communication further delays the redefined releases to later points of time. By bounding such data communication times and appropriately incorporating them into the prior tardiness bound (i.e., the one assuming no communication time), we are able to derive a final tardiness bound for every task in the given PGM system scheduled under CDAG, as stated in the following theorem. Due to space constraints, the full proof is given in a longer version of the paper available online [5].

Theorem 3. *The tardiness of any job $T_{l,j}^h$ of any task T_l^h at depth k within a DAG T_l scheduled under CDAG with respect to its original deadline is at most $(k+1)\cdot\Delta + 3(k+1)\cdot(y_l^{max} + max(v_{l,j}^h))$, where $max(v_{l,j}^h)$ denotes the maximum data communication time between any predecessor job of $T_{l,j}^h$ and $T_{l,j}^h$.*

Note that a per-task response time bound can be obtained from the above tardiness bound by adding the task's relative deadline. Such bounds are useful in settings where response time is used as the performance metric.[4] Note also that since no utilization loss occurs during the scheduling phase, any PGM system is *schedulable* with bounded response times as long as it can be partitioned onto clusters under CDAG.

3.4 Improving Job Response Times

According to Eqs.(2)-(4), we delay job releases to transform DAGs into sporadic tasks. However, excessive release delays are actually unnecessary and actual response times can be improved by applying a technique called "early-releasing," which allows jobs to execute before their specified release times [1]. The earliest time at which job $T_{l,j}^h$ may execute is defined by its *early-release time* $\varepsilon(T_{l,j}^h)$, where $\varepsilon(T_{l,j}^h) \leq r(T_{l,j}^h)$. For any job $T_{l,j}^h$, its early-releasing time can be defined as

$$\varepsilon(T_{l,j}^h) = \begin{cases} r_{l,j}^h & \text{if } h = 1 \\ F_{max}(pred(T_{l,j}^h), v_{l,j}^h) & \text{if } h > 1. \end{cases}$$

An unfinished job $T_{l,j}^h$ is *eligible* for execution at time t if $T_{l,j-1}^h$ has completed by t (if $j > 1$) and $t \geq \varepsilon(T_{l,j}^h)$. The tardiness bounds in Theorem 2 (as shown in [15]) and Theorem 3 continue to hold if early-releasing is allowed. Intuitively, this is reflective of the fact that schedulability mainly hinges on the proper spacing of consecutive job *deadlines* of a task, instead of its *releases*.

Example. Consider again the scheduling of T_2 as shown in Fig. 6(b). Fig. 6(d) shows early releases as defined above and the corresponding GEDF schedule. As seen, most jobs' response times are improved. For instance, $T_{2,2}^2$ now completes at time 10, two time units earlier than the case without early-releasing.

4 Experiments

In this section, we describe experiments conducted using randomly-generated DAG sets to evaluate the effectiveness of CDAG in minimizing utilization loss and total communication cost. We do this by comparing CDAG with the optimal ILP solution. The experiments focus on three performance metrics: (*i*) utilization loss, (*ii*) total communication cost, and (*iii*) each test's runtime performance.

In our experiments, we selected a random target size for DAGs, from at least one task to 100 per DAG. Then

[4] In some PGM-specified applications, deadlines are not specified but bounded response times are still required [11].

tasks within each DAG were generated based upon distributions proposed by Baker [2]. The source task of each DAG was assumed to be released sporadically, with a period uniformly distributed over $[10ms, 100ms]$. The produce amount of each edge was varied from 10 data units to 1000 data units. For every edge of each DAG, its produce amount, threshold, and consume amount were assumed to be the same. Valid execution rates were calculated for non-source tasks within each DAG using results from [11, 15]. Task utilizations were distributed using four uniform distributions, $[0.05, 0.2]$ (light), $[0.2, 0.5]$ (medium), $[0.5, 0.8]$ (heavy), and $[0.05, 0.8]$ (uniform). Task execution costs were calculated from execution rates and utilizations. We generated six clusters, each with a random processor count from 4 to 16, with a total processor count of 48. We assumed $B = 10$ and $b = 1000$. For each choice of utilization distribution, a cap on overall utilization was systematically varied within $[16, 48]$. For each combination of utilization cap and utilization distribution, we generated 100 DAG sets. Each such DAG set was generated by creating DAGs until total utilization exceeded the corresponding utilization cap, and by then reducing the last DAG's utilization so that the total utilization equalled the utilization cap.

The schedulability results that were obtained are shown in Fig. 7. In all insets of Fig. 7, "CDAG" denotes the schedulability results achieved by CDAG, "Thm. 1 Bound" denotes the worst-case utilization bound of CDAG as stated in Theorem 1, and "ILP" denotes the schedulability results achieved by ILP. Each curve in each figure plots the fraction of the generated DAG sets that the corresponding approach successfully scheduled, as a function of total utilization. (Note that the range of the x-axis in all insets is given by $[43, 48]$.) As Fig. 7 shows, under all four utilization distributions, CDAG yields schedulability results that are very close to that achieved by ILP. Moreover, the worst-case utilization bound in Theorem 1 is reasonable. For example, under the light per-task utilization distribution, the worst-case utilization bound of CDAG ensures that any DAG set with a total utilization up to 47 can be successfully scheduled in a distributed system containing 48 processors.

Table 1 shows the total communication cost achieved by both approaches categorized by the total utilization U_{sum} using the light per-task utilization distribution (we omit the results using other utilization distributions because they all show similar trends). In these experiments, all DAG sets were guaranteed to be schedulable since the total utilization of any DAG set (at most 44) is less than the worst-case utilization bound of CDAG, which is 47. In Table 1, for each U_{sum}, the total communication cost under ILP or CDAG was computed by taking the average of the total communication cost over the 100 generated DAG sets. The total communication cost for each generated DAG set is given by ϖ_{sum} as defined in Def. 2. The label "Total" represents the maximum communication cost of the DAG set, which is

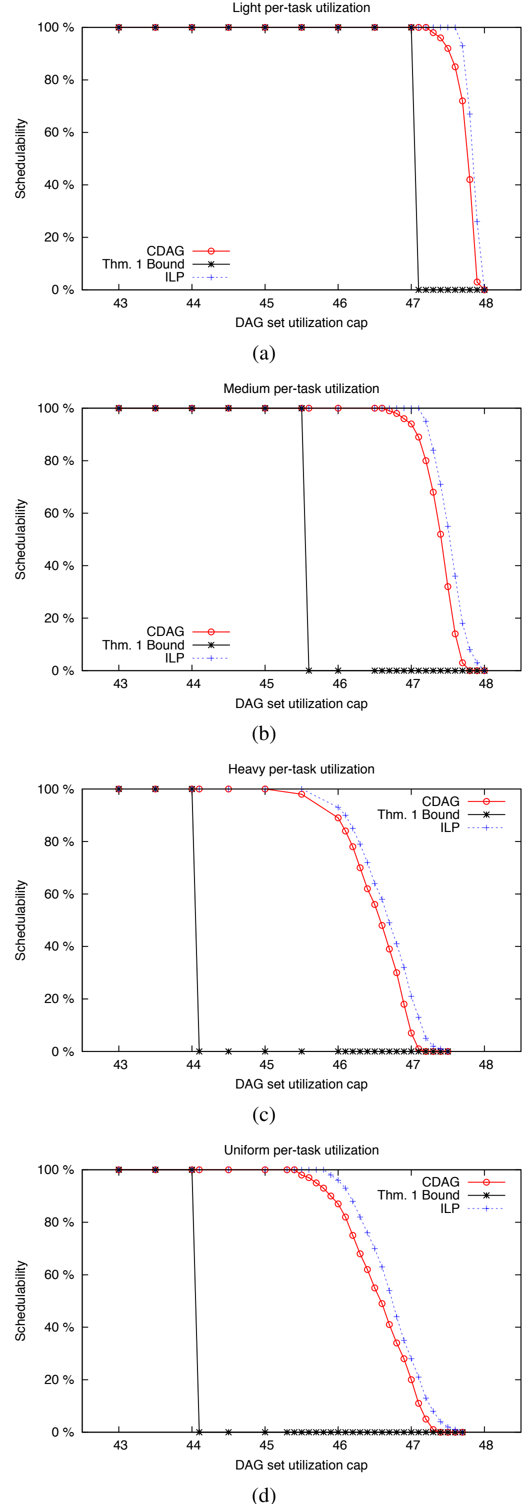

Figure 7: Schedulability results. **(a)** Light per-task utilization distribution. **(b)** Medium per-task utilization distribution. **(c)** Heavy per-task utilization distribution. **(d)** Uniform per-task utilization distribution.

given by $\sum_{T_i \in \zeta} \sum_{e_i^{jk} \in T_i} w_i^{jk}$ where (as noted earlier) ζ represents the corresponding DAG set. As seen, CDAG is effec-

Table 1: Total communication cost.

DAG set utilization / Method	U_{sum}=16	U_{sum}=20	U_{sum}=24	U_{sum}=28	U_{sum}=32	U_{sum}=36	U_{sum}=40	U_{sum}=44
ILP	6.4	7.5	10.6	17	22.5	26.2	29.8	32.6
CDAG	78.1	87.2	130.5	146.3	173.1	187.7	231.2	248.7
Total	34588.9	45048.8	52968.1	58895.3	69895.9	77125.5	86741.4	93775.9

Table 2: Runtime performance.

# of tasks / Method	N=100	N=200	N=300	N=400	N=500
ILP	51.9 (s)	165.9 (s)	412.2 (s)	2848.1 (s)	37791.8 (s)
CDAG	0.48 (ms)	0.58 (ms)	0.62 (ms)	0.56 (ms)	0.71 (ms)

tive in minimizing the total communication cost. The total communication costs achieved by CDAG are close to the optimal ones achieved by ILP and are significantly smaller than the maximum communication costs. For example, when $U_{sum} = 44$, CDAG achieves a total communication cost of 248.7 data units while ILP gives an optimal solution of 32.6 data units, both of which are almost negligible compared to the maximum communication cost, which is 93775.9 data units.

Regarding runtime performance, Table 2 shows the average time to run an experiment as a function of the number of tasks N using the light per-task utilization distribution (we again omit the results using other utilization distributions because they all show similar trends). For each N in the set $\{100, 200, 300, 400, 500\}$, we generated ten DAG sets and recorded the average running time of both ILP and CDAG. CDAG consistently took less than 1 ms to run while ILP ran for a significantly longer time, sometimes prohibitively so. For instance, when $N = 500$, ILP took more than 10 hours on average per generated DAG set.

5 Conclusion

In this paper, we have shown that DAG-based systems with sophisticated notions of acyclic precedence constraints can be efficiently supported under the proposed clustered scheduling algorithm CDAG in a distributed system, provided bounded deadline tardiness is acceptable. We assessed the effectiveness of CDAG in minimizing both utilization loss and total communication cost by comparing it with an optimal ILP solution. CDAG was analytically and experimentally shown to be effective in both respects and has low time complexity.

In future work, we would like to investigate more expressive resource models. For example, a distributed system may contain heterogeneous resources (such as CPUs and GPUs) and some tasks may need to access certain types of resources, which makes the task assignment problem more constrained. Moreover, it would be interesting to investigate other scheduling techniques such as semi-partitioned scheduling to achieve no utilization loss.

References

[1] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proc. of the 12th Euromicro Conf. on Real-Time Sys.*, pp. 35-43, 2000.

[2] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *Technical Report TR-051101, Florida State University*, 2005.

[3] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of real-time and embedded systems*, Chapman Hall/CRC, Boca Raton, Florida, 2007.

[4] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Algorithmica*, Vol.15, pp. 600-625, 1996.

[5] C. Liu and J. Anderson. Supporting graph-based real-time applications in distributed systems. Full Version. Available at:http://www.cs.unc.edu/ ∼anderson/papers.

[6] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore systems. In *Proc. of the 19th ECRTS*, pp. 247-256, 2007.

[7] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah. *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[8] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Lecture in Computer Science*, Vol. 2125, pp. 38-49, 2001.

[9] DARPA. Satellite System F6 Specification and Requirement. http://www.darpa.mil/Our_Work/TTO/Programs/Systemf6/System_F6.aspx, 2011.

[10] M. Delest, A. Don, and J. Benois-Pineau. DAG-based visual interfaces for navigation in indexed video content. In *Multimedia Tools and Applications*, Vol. 31, pp. 51-72, 2006.

[11] S. Goddard. *On the Management of Latency in the synthesis of real-time signal processing systems from processing graphs*. PhD thesis, The University of North Carolina at Chapel Hill, 1998.

[12] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proc. of the 20th RTSS*, pp. 304-314, 1999.

[13] Naval Research Laboratory. Processing graph method specification. In *prepared by the Naval Research Laboratory for use by the Navy Standard Signal Processing Program Office (PMS-412)*, 1987.

[14] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Real-Time Systems*, Vol. 44, pp. 26-71, 2010.

[15] C. Liu and J. Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proc. of the 31st RTSS*, pp. 3-13, 2010.

[16] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, 2007.