Noname manuscript No.
(will be inserted by the editor)

Multi-core Real-Time Scheduling for Generalized Parallel Task Models

Abusayeed Saifullah · Jing Li · Kunal Agrawal · Chenyang Lu · Christopher Gill

Received: date / Accepted: date

Abstract Multi-core processors offer a significant performance increase over single-core processors. They have the potential to enable computation-intensive real-time applications with stringent timing constraints that cannot be met on traditional single-core processors. However, most results in traditional multiprocessor real-time scheduling are limited to sequential programming models and ignore intra-task parallelism. In this paper, we address the problem of scheduling periodic parallel tasks with implicit deadlines on multi-core processors. We first consider a synchronous task model where each task consists of segments, each segment having an arbitrary number of parallel threads that synchronize at the end of the segment. We propose a new task decomposition method that decomposes each parallel task into a set of sequential tasks. We prove that our task decomposition achieves a resource augmentation bound of 4 and 5 when the decomposed tasks are scheduled using global EDF and partitioned deadline monotonic scheduling, respectively. Finally, we extend our analysis to a directed acyclic graph (DAG) task model where each node in the DAG has a unit execution requirement. We show how these tasks can be converted into synchronous tasks such that the same decomposition can be applied and the same augmentation bounds hold. Simulations based on synthetic workload demonstrate that the derived resource augmentation bounds are safe and sufficient.

Keywords parallel task · multi-core processor · real-time scheduling · resource augmentation bound

Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, Christopher Gill
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box. 1045
St Louis, MO 63130, USA
E-mail: {saifullah, li.jing, kunal, lu, cdgill}@wustl.edu

1 Introduction

In recent years, multi-core processor technology has improved dramatically as chip manufacturers try to boost performance while minimizing power consumption. This development has shifted the scaling trends from increasing processor clock frequencies to increasing the number of cores per processor. For example, Intel has recently put 80 cores in a Teraflops Research Chip (Intel, 2007) with a view to making it generally available, and ClearSpeed has developed a 96-core processor (ClearSpeed, 2008). While hardware technology is moving at a rapid pace, software and programming models have failed to keep pace. For example, Intel (2007) has set a time frame of 5 years to make their 80-core processor generally available due to the inability of current operating systems and software to exploit the benefits of multi-core processors.

As multi-core processors continue to scale, they provide an opportunity for performing more complex and computation-intensive tasks in real-time. However, to take full advantage of multi-core processing, these systems must exploit intra-task parallelism, where parallelizable real-time tasks can utilize multiple cores at the same time. By exploiting intra-task parallelism, multi-core processors can achieve significant real-time performance improvement over traditional single-core processors for many computation-intensive real-time applications such as video surveillance, radar tracking, and hybrid real-time structural testing (Huang et al., 2010) where the performance limitations of traditional single-core processors have been a major hurdle.

The growing importance of parallel task models for real-time applications poses new challenges to real-time scheduling theory that had previously mostly focused on sequential task models. The state-of-the-art work (Lakshmanan et al., 2010) on parallel scheduling for real-time tasks with intra-task parallelism analyzes the *resource augmentation bound* using partitioned Deadline Monotonic (DM) scheduling. A *resource augmentation* under a scheduling policy quantifies processor-speed up factor (how much we have to increase the processor speed) with respect to an optimal algorithm to guarantee the schedulability of a task set under that policy. The state-of-the-art work (Lakshmanan et al., 2010) considers a synchronous task model, where each parallel task consists of a series of sequential or parallel segments. We call this model *synchronous*, since all the threads of a parallel segment must finish before the next segment starts, creating a synchronization point. However, that task model is *restrictive* in that, for every task, all the segments have an *equal* number of parallel threads, and the execution requirements of all threads in a segment are equal. Most importantly, in that task model, the number of threads in every segment is *no greater* than the total number of processor cores.

While the work presented by Lakshmanan et al. (2010) represents a promising step towards parallel real-time scheduling on multi-core processors, the restrictions on the task model make the solutions unsuitable for many real-time applications that often employ different numbers of threads in different segments of computation. In addition, it analyzes the resource augmentation

bound under partitioned DM scheduling only, and does not consider other scheduling policies such as global EDF. In this work, we consider real-time scheduling on multi-core processors for a more general synchronous task model. Our tasks still contain segments where the threads of each segment synchronize at its end. However, in contrast to the restrictive task model addressed in Lakshmanan et al. (2010), for any task in our model, each segment can contain an *arbitrary* number of parallel threads. That is, different segments of the same parallel task can contain different numbers of threads, and segments can contain more threads than the number of processor cores. Furthermore, the execution requirements of the threads in any segment can vary. This model is more portable, since the same task can be executed on machines with small as well as large numbers of cores. Specifically, our work makes the following new contributions to real-time scheduling for periodic parallel tasks.

- For the general synchronous task model, we propose a task decomposition algorithm that converts each implicit deadline parallel task into a set of constrained deadline sequential tasks.
- We derive a resource augmentation bound of 4 when these decomposed tasks are scheduled using global EDF scheduling. To our knowledge, this is the first resource augmentation bound for global EDF scheduling of parallel tasks.
- Using the proposed task decomposition, we also derive a resource augmentation bound of 5 for our more general task model under partitioned DM scheduling.
- Finally, we extend our analyses for a Directed Acyclic Graph (DAG) task model where each node in a DAG has a unit execution requirement. This is an even more general model for parallel tasks. Namely, we show that we can transform unit-node DAG tasks into synchronous tasks, and then use our proposed decomposition to get the same resource augmentation bounds for the former.

We evaluate the performance of the proposed decomposition through simulations based on synthetic workloads. The results indicate that the derived bounds are safe and sufficient. In particular, the resource augmentations required to schedule the decomposed tasks in our simulations are at most 2.4 and 3.4 for global EDF and partitioned DM scheduling, respectively, which are significantly smaller than the corresponding theoretical bounds.

In the rest of the paper, Section 2 describes the parallel synchronous task model. Section 3 presents the proposed task decomposition. Section 4 presents the analysis for global EDF scheduling. Section 5 presents the analysis for partitioned DM scheduling. Section 6 extends our results and analyses for unit-node DAG task models. Section 7 presents the simulation results. Section 8 reviews related work. Finally, we conclude in Section 9.

2 Parallel Synchronous Task Model

We primarily consider a synchronous parallel task model, where each task consists of a sequence of computation segments, each segment having an arbitrary number of parallel threads with arbitrary execution requirements that synchronize at the end of the segment. Such tasks are generated by parallel *for* loops, a construct common to many parallel languages such as OpenMP (OpenMP, 2011) and CilkPlus (Intel, 2010).

We consider n periodic synchronous parallel tasks with implicit deadlines (i.e. deadlines are equal to periods). Each task τ_i, $1 \leq i \leq n$, is a sequence of s_i segments, where the j-th segment, $1 \leq j \leq s_i$, consists of $m_{i,j}$ parallel threads. First we consider the case when, for any segment of τ_i, all parallel threads in the segment have equal execution requirements. For such τ_i, the j-th segment, $1 \leq j \leq s_i$, is represented by $\langle e_{i,j}, m_{i,j} \rangle$, with $e_{i,j}$ being the worst case execution requirement of each of its threads. When $m_{i,j} > 1$, the threads in the j-th segment can be executed in parallel on different cores. The j-th segment starts only after all threads of the $(j-1)$-th segment have completed. Thus, a parallel task τ_i in which a segment consists of equal-length threads is shown in Figure 1, and is represented as $\tau_i : (\langle e_{i,1}, m_{i,1} \rangle, \langle e_{i,2}, m_{i,2} \rangle, \cdots, \langle e_{i,s_i}, m_{i,s_i} \rangle)$ where

- s_i is the total number of segments in task τ_i.
- In a segment $\langle e_{i,j}, m_{i,j} \rangle$, $1 \leq j \leq s_i$, $e_{i,j}$ is the worst case execution requirement of each thread, and $m_{i,j}$ is the number of threads. Therefore, any segment $\langle e_{i,j}, m_{i,j} \rangle$ with $m_{i,j} > 1$ is a *parallel segment* with a total of $m_{i,j}$ parallel threads, and any segment $\langle e_{i,j}, m_{i,j} \rangle$ with $m_{i,j} = 1$ is a *sequential segment* since it has only one thread. A task τ_i with $s_i = 1$ and $m_{i,s_i} = 1$ is a sequential task.

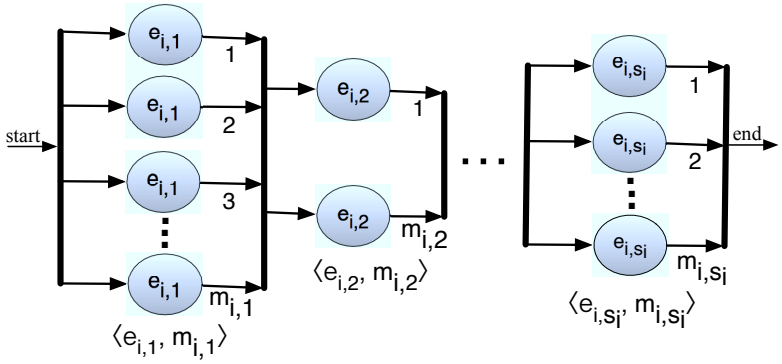
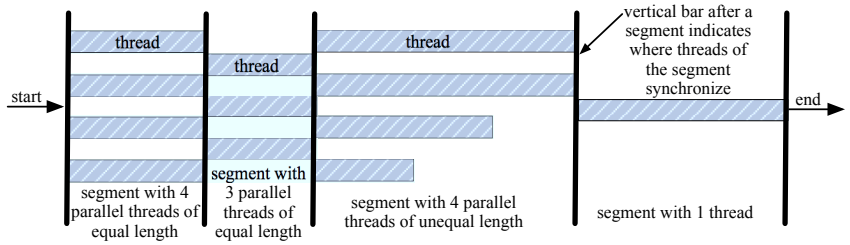


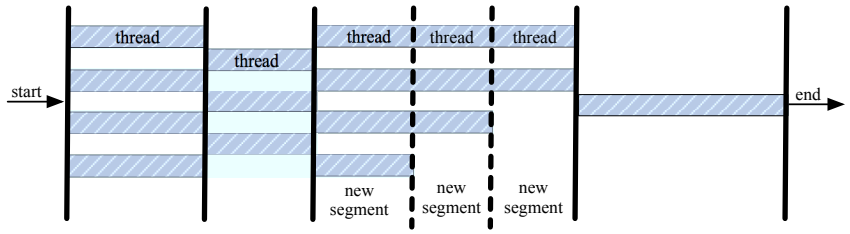
Fig. 1 A parallel synchronous task τ_i

Now, we consider the case when the execution requirements of parallel threads in a segment of τ_i may differ from each other. An example of such a

task is shown in Figure 2(a), where each horizontal bar indicates the length of the execution requirement of a thread. As the figure shows, the parallel threads in the third segment have unequal execution requirements. By adding a new synchronization point at the end of each thread in a segment, any segment consisting of threads of unequal length can be converted to several segments each consisting of threads of equal length as shown in Figure 2(b). Specifically, for the task with unequal-length threads in a segment shown in Figure 2(a), Figure 2(b) shows the corresponding task in which each segment consists of equal-length threads. Thus, in any synchronous parallel task, any segment consisting of threads of different execution requirements can be converted to several segments each consisting of threads of an equal execution requirement without changing any task parameter such as period, deadline, or execution requirement. It is worth noting that such a conversion is not entirely loss-less since it adds additional synchronization points. It is entirely possible that some task system that would be schedulable with another transformation might not be schedulable with our proposed one. However, since the execution requirements do not change, it has no effect on the utilization of the system, and our bounds depend on the utilization of the system. Hence, we concentrate only to the task model where each segment in a task consists of equal-length threads (such as the one shown in Figure 1).



(a) A synchronous task with unequal-length threads in a segment



(b) The corresponding synchronous task with equal-length threads in each segment (each dotted vertical line indicates a newly added synchronization point at the end of a thread)

Fig. 2 Conversion of a segment with unequal-length threads to segments with equal-length threads in a synchronous parallel task

Therefore, considering a multi-core platform consisting of m processor cores, we focus on scheduling n parallel tasks denoted by $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where each τ_i is represented as $\tau_i : (\langle e_{i,1}, m_{i,1}\rangle, \langle e_{i,2}, m_{i,2}\rangle, \cdots, \langle e_{i,s_i}, m_{i,s_i}\rangle)$ (as the one shown in Figure 1). The period of task τ_i is denoted by T_i. The deadline D_i of τ_i is equal to its period T_i. Each task τ_i generates (potentially) an infinite sequence of jobs, with arrival times of successive jobs separated by T_i time units. Jobs are fully independent and preemptive: any job can be suspended (preempted) at any time instant, and is later resumed with no cost or penalty. The task set is said to be *schedulable* when all tasks meet their deadlines.

3 Task Decomposition

In this section, we present a decomposition of the parallel tasks into a set of sequential tasks. In particular, we propose a strategy that decomposes each implicit deadline parallel task (like the one shown in Figure 1) into a set of constrained deadline (i.e. deadlines are no greater than periods) sequential tasks by converting each thread of the parallel task into its own sequential task and assigning appropriate deadlines to these tasks. This strategy allows us to use existing schedulability analysis for multiprocessor scheduling (both global and partitioned) to prove the resource augmentation bounds for parallel tasks (to be discussed in Sections 4 and 5). Here, we first present some useful terminology. We then present our decomposition and a density analysis for it.

3.1 Terminology

Definition 1 The *minimum execution time* (i.e. the *critical path length*) P_i of task τ_i on a multi-core platform where each processor core has unit speed is defined as

$$P_i = \sum_{j=1}^{s_i} e_{i,j}$$

Observation 1 *On a unit-speed multi-core platform, any task τ_i requires at least P_i units of time even when the number of cores m is infinite.*

On a multi-core platform where each processor core has speed ν, the critical path length of task τ_i is denoted by $P_{i,\nu}$ and is expressed as follows.

$$P_{i,\nu} = \frac{1}{\nu}\sum_{j=1}^{s_i} e_{i,j} = \frac{P_i}{\nu}$$

Definition 2 The *maximum execution time* (i.e. the *work*) C_i of task τ_i on a multi-core platform where each processor core has unit speed is defined as

$$C_i = \sum_{j=1}^{s_i} m_{i,j}.e_{i,j}$$

That is, C_i is the execution time of τ_i on a unit-speed single core processor if it is never preempted. On a multi-core platform where each processor core has speed ν, the maximum execution time of task τ_i is denoted by $C_{i,\nu}$ and is expressed as follows.

$$C_{i,\nu} = \frac{1}{\nu}\sum_{j=1}^{s_i} m_{i,j}.e_{i,j} = \frac{C_i}{\nu} \tag{1}$$

Definition 3 The *utilization* u_i of task τ_i, and the *total utilization* $u_{\text{sum}}(\tau)$ for the set of n tasks τ on a unit-speed multi-core platform are defined as

$$u_i = \frac{C_i}{T_i}; \qquad u_{\text{sum}}(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Observation 2 *If the total utilization u_{sum} is greater than m, then no algorithm can schedule τ on m identical unit speed processor cores.*

Definition 4 The *density* δ_i of task τ_i, the *maximum density* $\delta_{\max}(\tau)$ and the *total density* $\delta_{\text{sum}}(\tau)$ of the set of n tasks τ on a unit-speed multi-core platform are defined as follows:

$$\delta_i = \frac{C_i}{D_i}; \quad \delta_{\text{sum}}(\tau) = \sum_{i=1}^{n} \delta_i; \quad \delta_{\max}(\tau) = \max\{\delta_i | 1 \leq i \leq n\}$$

For an implicit deadline task τ_i, $\delta_i = u_i$.

3.2 Decomposition

Following is the high-level idea of the decomposition of a parallel task τ_i.

1. In our decomposition, each thread of the task becomes its own sequential subtask. These individual subtasks are assigned release times and deadlines. Since each thread of a segment is identical (with respect to its execution time), we consider each segment one at a time, and assign the same release times and deadlines to all subtasks generated from threads of the same segment.
2. Since a segment $\langle e_{i,j}, m_{i,j} \rangle$ has to complete before segment $\langle e_{i,j+1}, m_{i,j+1} \rangle$ can start, the release time of the subtasks of segment $\langle e_{i,j+1}, m_{i,j+1} \rangle$ is equal to the absolute deadline of the subtasks of segment $\langle e_{i,j}, m_{i,j} \rangle$.
3. As stated before, we analyze the schedulability of the decomposed tasks based on their densities. The analysis is largely dependent on the total density (δ_{sum}) and the maximum density ($\delta_{\max}$) of the decomposed tasks. Therefore, we want to keep both δ_{sum} and $\delta_{\max}$ bounded and as small as possible. In particular, we need $\delta_{\max}$ to be at most 1, and we want δ_{sum} over all tasks to be at most u_{sum} after decomposition. To do so, usually we need to assign enough slack among the segments. It turns out to be difficult to assign enough slack to each segment if we consider unit-speed processors.

However, we can increase the available slack by considering higher speed processors. In particular, we find that decomposing on speed 2 processors allows us enough slack to decompose effectively, keeping both δ_{sum} and $\delta_{\max}$ at desired levels. Decomposing on processors of higher speed is also possible but leads to lower efficiency. On the other hand, decomposing on speed 1 processors would be clearly non-optimal (in terms of the availability of slack). In particular, if we do the decomposition with α-speed processor, where $\alpha \geq 1$, then the best value of α to minimize both δ_{sum} and $\delta_{\max}$ becomes 2. Therefore, in the remainder of the paper we restrict ourselves to decomposition on 2-speed processors. Thus, the *slack* for task τ_i, denoted by L_i, that is distributed among the segments is its slack on speed 2 processors, given by

$$L_i = T_i - P_{i,2} = T_i - \frac{P_i}{2} \tag{2}$$

This slack is distributed among the segments according to a principle of "equitable density" meaning that we try to keep the density of each segment approximately rather than exactly equal by maintaining a uniform upper bound on the densities. To do this, we take both the number of threads in each segment and the computation requirement of the threads in each segment into consideration while distributing the slack.

In order to take the computation requirement of the threads in each segment into consideration, we assign proportional slack fractions instead of absolute slack. We now formalize the notion of *slack fraction*, $f_{i,j}$, for the j-th segment (i.e. segment $\langle e_{i,j}, m_{i,j} \rangle$) of task τ_i. *Slack fraction* $f_{i,j}$ is the fraction of L_i (i.e. the total slack) to be allotted to segment $\langle e_{i,j}, m_{i,j} \rangle$ proportionally to its minimum computation requirement. Each thread in segment $\langle e_{i,j}, m_{i,j} \rangle$ has a minimum execution time of $\frac{e_{i,j}}{2}$ on 2-speed processor cores, and is assigned a slack value of $f_{i,j}\frac{e_{i,j}}{2}$. Each thread gets this "extra time" beyond its execution requirement on 2-speed processor cores. Thus, for each thread in segment $\langle e_{i,j}, m_{i,j} \rangle$, the relative deadline is assigned as

$$d_{i,j} = \frac{e_{i,j}}{2} + f_{i,j}.\frac{e_{i,j}}{2} = \frac{e_{i,j}}{2}(1 + f_{i,j}) \tag{3}$$

Equation 3 shows how a thread's deadline $d_{i,j}$ is calculated based on its assigned slack fraction $f_{i,j}$. For example, if a thread has $e_{i,j} = 4$ and it is assigned a slack fraction of $f_{i,j} = 1.5$, then its relative deadline is $2(1+1.5) = 5$. That is, the thread has been assigned a slack value of $d_{i,j} - \frac{e_{i,j}}{2} = 5 - \frac{4}{2} = 3$ (or, equivalently $f_{i,j}\frac{e_{i,j}}{2} = 3$) on 2-speed cores. Similarly, the value of 0 of $f_{i,j}$ implies that the thread has been assigned no slack on 2-speed processor cores. Note that since the slack fraction and hence the slack can not be negative, the density of a segment is at least 1. Therefore, in order to satisfy our maximum density requirement — that $\delta_{\max}$ is at most 1 after decomposition on speed-2 processors — we must ensure that the slack fraction is never negative.

Since a segment cannot start before all previous segments complete, the release offset of a segment $\langle e_{i,j}, m_{i,j} \rangle$ is assigned as

$$\phi_{i,j} = \sum_{k=1}^{j-1} d_{i,k} \tag{4}$$

Thus, the density of each thread in segment $\langle e_{i,j}, m_{i,j} \rangle$ on 2-speed cores is

$$\frac{\frac{e_{i,j}}{2}}{d_{i,j}} = \frac{\frac{e_{i,j}}{2}}{\frac{e_{i,j}}{2}(1+f_{i,j})} = \frac{1}{1+f_{i,j}}$$

Since a segment $\langle e_{i,j}, m_{i,j} \rangle$ consists of $m_{i,j}$ threads, the segment's density on 2-speed processor cores is

$$\frac{m_{i,j}}{1+f_{i,j}} \tag{5}$$

Note that to meet the deadline of the parallel task on 2-speed processor cores, the segment slack should be assigned so that

$$f_{i,1}.\frac{e_{i,1}}{2} + f_{i,2}.\frac{e_{i,2}}{2} + f_{i,3}.\frac{e_{i,3}}{2} + \cdots + f_{i,s_i}.\frac{e_{i,s_i}}{2} \leq L_i.$$

In our decomposition, we always assign the maximum possible segment slack on 2-speed processor cores and, therefore, for our decomposition, the above inequality is in fact an equality.

Since after assigning slack, we want to keep the density of each segment about equal, we must take the number of threads of the segment into consideration while assigning slack fractions. As stated before, we need to keep the sum of densities bounded on speed-2 processors after decomposition. To determine whether slack assignment to a segment is critical or not, we calculate a threshold based on task parameters. The segments whose number of threads is greater than this threshold are computation intensive, and hence assigning slack to such segments is critical. The remaining segments are deemed to be less computation intensive, and hence assigning slack to such segments is less critical. Hence, to calculate segment slack according to equitable density, we classify segments into two categories based on their computation requirements and slack demand:

- *Heavy segments* are those which have $m_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$. That is, they have many parallel threads, and hence are computation intensive.
- *Light segments* are those which have $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$. That is, these segments are less computation intensive.

The threshold $\frac{C_{i,2}}{T_i - P_{i,2}}$ is chosen to ensure that no thread is assigned any negative slack on 2-speed processor cores. We later (in Subsection 3.2.1) prove that every thread is indeed assigned a non-negative slack which, in fact, guarantees that its density is at most 1 on 2-speed processor cores.

Using the above categorization, we also classify parallel tasks into two categories: tasks that have some or all heavy segments versus tasks that have only light segments, and analyze them separately as follows.

3.2.1 Tasks with some (or all) heavy segments

For the tasks which have some heavy segments, we treat heavy and light segments differently while assigning slack. In particular, we assign no slack to the light segments; that is, segments with $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$ of τ_i are assigned $f_{i,j} = 0$. The total available slack L_i is distributed among the heavy segments (segments with $m_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$) in such a way that each of these segments has the same density.

For simplicity of presentation, we first distinguish notations between the heavy and light segments. Let the heavy segments of τ_i be represented as $\{\langle e^h_{i,1}, m^h_{i,1}\rangle, \langle e^h_{i,2}, m^h_{i,2}\rangle, \cdots, \langle e^h_{i,s^h_i}, m^h_{i,s^h_i}\rangle\}$, where $s^h_i \leq s_i$ (superscript h standing for 'heavy'). Then, let

$$P^h_{i,2} = \frac{1}{2}\sum_{j=1}^{s^h_i} e^h_{i,j}; \qquad C^h_{i,2} = \frac{1}{2}\sum_{j=1}^{s^h_i} m^h_{i,j}.e^h_{i,j} \tag{6}$$

The light segments are denoted as $\{\langle e^\ell_{i,1}, m^\ell_{i,1}\rangle, \langle e^\ell_{i,2}, m^\ell_{i,2}\rangle, \cdots, \langle e^\ell_{i,s^\ell_i}, m^\ell_{i,s^\ell_i}\rangle\}$, where $s^\ell_i = s_i - s^h_i$ (superscript ℓ standing for 'light'). Then, let

$$P^\ell_{i,2} = \frac{1}{2}\sum_{j=1}^{s^\ell_i} e^\ell_{i,j}; \qquad C^\ell_{i,2} = \frac{1}{2}\sum_{j=1}^{s^\ell_i} m^\ell_{i,j}.e^\ell_{i,j} \tag{7}$$

Now, the following equalities must hold for task τ_i.

$$P_{i,2} = \frac{P_i}{2} = P^h_{i,2} + P^\ell_{i,2}; \qquad C_{i,2} = \frac{C_i}{2} = C^h_{i,2} + C^\ell_{i,2} \tag{8}$$

Now we calculate slack fraction $f^h_{i,j}$ for all heavy segments (i.e. segments $\langle e^h_{i,j}, m^h_{i,j}\rangle$, where $1 \leq j \leq s^h_i$ and $m^h_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$) so that they all have equal density on 2-speed processor cores. That is,

$$\frac{m^h_{i,1}}{1 + f^h_{i,1}} = \frac{m^h_{i,2}}{1 + f^h_{i,2}} = \frac{m^h_{i,3}}{1 + f^h_{i,3}} = \cdots = \frac{m^h_{i,s^h_i}}{1 + f^h_{i,s^h_i}} \tag{9}$$

In addition, since all the slack is distributed among the heavy segments, the following equality must hold.

$$f^h_{i,1}.e^h_{i,1} + f^h_{i,2}.e^h_{i,2} + f^h_{i,3}.e^h_{i,3} + \cdots + f^h_{i,s^h_i}.e^h_{i,s^h_i} = 2.L_i \tag{10}$$

It follows that the value of each $f^h_{i,j}$, $1 \leq j \leq s^h_i$, can be determined by solving Equations 9 and 10 as shown below. From Equation 9, the value of $f^h_{i,j}$ for each j, $2 \leq j \leq s^h_i$, can be expressed in terms of $f^h_{i,1}$ as follows.

$$f^h_{i,j} = (1 + f^h_{i,1})\frac{m^h_{i,j}}{m^h_{i,1}} - 1 \tag{11}$$

Putting the value of each $f^h_{i,j}$, $2 \leq j \leq s^h_i$, from Equation 11 into Equation 10:

$$\begin{aligned}
2L_i &= f^h_{i,1}e^h_{i,1} + \sum_{j=2}^{s^h_i}\left(\Big((1+f^h_{i,1})\frac{m^h_{i,j}}{m^h_{i,1}} - 1\Big)e^h_{i,j}\right) \\
&= f^h_{i,1}e^h_{i,1} + \sum_{j=2}^{s^h_i}\left(\frac{m^h_{i,j}}{m^h_{i,1}}e^h_{i,j} + f^h_{i,1}\frac{m^h_{i,j}}{m^h_{i,1}}e^h_{i,j} - e^h_{i,j}\right) \\
&= f^h_{i,1}e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j} + \frac{f^h_{i,1}}{m^h_{i,1}}\sum_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j} - \sum_{j=2}^{s^h_i}e^h_{i,j}
\end{aligned}$$

From the above equation, we can determine the value of $f^h_{i,1}$ as follows.

$$\begin{aligned}
f^h_{i,1} &= \frac{2L_i + \sum\limits_{j=2}^{s^h_i}e^h_{i,j} - \frac{1}{m^h_{i,1}}\sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j}}{e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j}} \\
&= \frac{2L_i + (\sum\limits_{j=2}^{s^h_i}e^h_{i,j} + e^h_{i,1}) - (e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j})}{e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j}} \\
&= \frac{2L_i + \sum\limits_{j=1}^{s^h_i}e^h_{i,j}}{e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j}} - 1
\end{aligned}$$

In the above equation, replacing $\sum_{j=1}^{s^h_i} e^h_{i,j}$ with $2P^h_{i,2}$ from Equation 6, we get

$$f^h_{i,1} = \frac{2L_i + 2P^h_{i,2}}{e^h_{i,1} + \frac{1}{m^h_{i,1}}\sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j}} - 1 = \frac{m^h_{i,1}(2L_i + 2P^h_{i,2})}{m^h_{i,1}e^h_{i,1} + \sum\limits_{j=2}^{s^h_i}m^h_{i,j}e^h_{i,j}} - 1$$

Similarly, in the above equation, replacing $(m^h_{i,1}e^h_{i,1} + \sum_{j=2}^{s^h_i} m^h_{i,j}e^h_{i,j})$ with $2C^h_{i,2}$ from Equation 6, the value of $f^h_{i,1}$ can be written as follows.

$$
\begin{aligned}
f^h_{i,1} &= \frac{m^h_{i,1}(2L_i + 2P^h_{i,2})}{2C^h_{i,2}} - 1 = \frac{m^h_{i,1}(L_i + P^h_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 && \text{(From 8)} \\
&= \frac{m^h_{i,1}((T_i - P_{i,2}) + P^h_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 && \text{(From 2)} \\
&= \frac{m^h_{i,1}(T_i - (P^h_{i,2} + P^\ell_{i,2}) + P^h_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 && \text{(From 8)} \\
&= \frac{m^h_{i,1}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1
\end{aligned}
$$

Now putting the above value of $f^h_{i,1}$ in Equation 11, for any heavy segment $\langle e^h_{i,j}, m^h_{i,j}\rangle$, we get

$$f^h_{i,j} = \frac{m^h_{i,j}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1 \tag{12}$$

Intuitively, the slack never should be negative, since the deadline should be no less than the computation requirement of the thread. Since $m^h_{i,j} > \frac{C_{i,2}}{T_i - P_{i,2}}$, according to Equation 12, the quantity $\frac{m^h_{i,j}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} > 1$. This implies that $f^h_{i,j} > 0$. Now, using Equation 5, the density of every segment $\langle e^h_{i,j}, m^h_{i,j}\rangle$ is

$$\frac{m^h_{i,j}}{1 + f^h_{i,j}} = \frac{m^h_{i,j}}{1 + \frac{m^h_{i,j}(T_i - P^\ell_{i,2})}{C_{i,2} - C^\ell_{i,2}} - 1} = \frac{C_{i,2} - C^\ell_{i,2}}{T_i - P^\ell_{i,2}} \tag{13}$$

Figure 3 shows a simple example of decomposition for a task τ_i consisting of 3 segments.

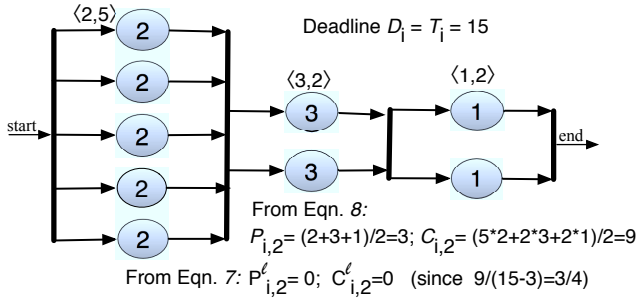
3.2.2 Tasks with no heavy segments

When the parallel task does not contain any heavy segments, we just assign the slack proportionally (according to the length of $e_{i,j}$) among all segments. That is,

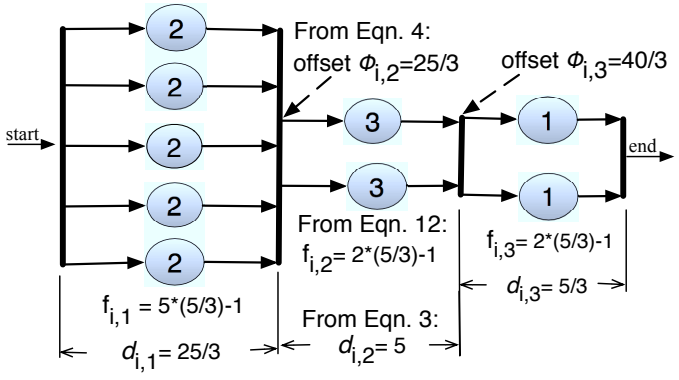
$$f_{i,j} = \frac{L_i}{P_{i,2}} \tag{14}$$

By Equation 5, the density of each segment $\langle e_{i,j}, m_{i,j}\rangle$ is

$$\frac{m_{i,j}}{1 + f_{i,j}} = \frac{m_{i,j}}{1 + \frac{L_i}{P_{i,2}}} = m_{i,j}\frac{P_{i,2}}{L_i + P_{i,2}} = m_{i,j}\frac{P_{i,2}}{T_i} \tag{15}$$



(a) A parallel synchronous task τ_i



(b) Decomposed task τ_i^{decom}

Fig. 3 An example of decomposition

3.3 Density Analysis

Once the above decomposition is done on task τ_i: $(\langle e_{i,1}, m_{i,1}\rangle, \cdots, \langle e_{i,s_i}, m_{i,s_i}\rangle)$, each thread of each segment $\langle e_{i,j}, m_{i,j}\rangle$, $1 \leq j \leq s_i$, is considered as a sequential multiprocessor subtask. We use τ_i^{decom} to denote task τ_i after decomposition. That is, τ_i^{decom} denotes the set of subtasks generated from τ_i through decomposition. Similarly, we use τ^{decom} to denote the entire task set τ after decomposition. That is, τ^{decom} is the set of all subtasks that our decomposition generates. Since $f_{i,j} \geq 0$, $\forall 1 \leq j \leq s_i$, $\forall 1 \leq i \leq n$, the maximum density $\delta_{\max,2}$ of any subtask (thread) among τ^{decom} on 2-speed processor core is

$$\delta_{\max,2} = \max\{\frac{1}{1+f_{i,j}}\} \leq 1 \tag{16}$$

We must now bound δ_{sum}. Lemma 1 shows that the density of every segment is at most $\frac{C_i/2}{T_i - P_i/2}$ for any task with or without heavy segments.

Lemma 1 *After the decomposition, the density of every segment* $\langle e_{i,j}, m_{i,j}\rangle$*, where* $1 \leq j \leq s_i$*, of every task* τ_i *on* 2*-speed processor cores is upper bounded by* $\frac{C_i/2}{T_i - P_i/2}$.

Proof First, we analyze the case when the task contains some heavy segments. According to Equation 13, for every heavy segment $\langle e_{i,j}, m_{i,j}\rangle$, the density is

$$\begin{aligned}
\frac{C_{i,2} - C^{\ell}_{i,2}}{T_i - P^{\ell}_{i,2}} &\leq \frac{C_{i,2}}{T_i - P^{\ell}_{i,2}} && \text{(since } C^{\ell}_{i,2} \geq 0\text{)} \\
&\leq \frac{C_{i,2}}{T_i - P_{i,2}} && \text{(since } P_{i,2} \geq P^{\ell}_{i,2}\text{)} \\
&= \frac{C_i/2}{T_i - P_i/2}
\end{aligned}$$

For every light segment $\langle e_{i,j}, m_{i,j}\rangle$ (i.e., a segment with $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$), the slack fraction $f_{i,j} = 0$. That is, its deadline is equal to its computation requirement $\frac{e_{i,j}}{2}$ on 2-speed processor cores. Therefore, its density, by definition, is

$$\frac{m_{i,j}}{1 + f_{i,j}} = m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}} = \frac{C_i/2}{T_i - P_i/2}$$

For the case when there are no heavy segments in τ_i, for every segment $\langle e_{i,j}, m_{i,j}\rangle$ of τ_i, $m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}}$. Since $T_i \geq P_{i,2}$ (Observation 1), the density of each segment $\langle e_{i,j}, m_{i,j}\rangle$ (Equation 15) of τ_i:

$$m_{i,j}\frac{P_{i,2}}{T_i} \leq m_{i,j} \leq \frac{C_{i,2}}{T_i - P_{i,2}} = \frac{C_i/2}{T_i - P_i/2}$$

Hence, follows the lemma. □

Thus, our decomposition distributes the slack so that each segment has a density that is bounded above. Theorem 2 establishes an upper bound on the density of every task after decomposition.

Theorem 2 *The density* $\delta_{i,2}$ *of every* τ^{decom}_i*,* $1 \leq i \leq n$*, (i.e. the density of every task* τ_i *after decomposition) on* 2*-speed processor cores is upper bounded by* $\frac{C_i/2}{T_i - P_i/2}$.

Proof After the decomposition, the densities of all segments of τ_i comprise the density of τ^{decom}_i. However, no two segments are simultaneous active, and each segment occurs exactly once during the activation time of task τ_i. Therefore, we can replace each segment with the segment that has the maximum density. Thus, task τ^{decom}_i can be considered as s_i occurences of the segment that has the maximum density, and therefore, the density of the entire task set τ^{decom}_i is equal to that of the segment having the maximum density which is at most $\frac{C_i/2}{T_i - P_i/2}$ (Lemma 1). Therefore, $\delta_{i,2} \leq \frac{C_i/2}{T_i - P_i/2}$. □

Lemma 3 *If* τ^{decom} *is schedulable, then* τ *is also schedulable.*

Proof For each τ_i^{decom}, $1 \leq i \leq n$, its deadline and execution requirement are the same as those of original task τ_i. Besides, in each τ_i^{decom}, a subtask is released only after all its preceding segments are complete. Hence, the precedence relations in original task τ_i are retained in τ_i^{decom}. Therefore, if τ^{decom} is schedulable, then a schedule must exist for τ where each task in τ can meet its deadline. □

4 Global EDF Scheduling

After our proposed decomposition, we consider the scheduling of synchronous parallel tasks. Lakshmanan et al. (2010) show that there exist task sets with total utilization slightly greater than (and arbitrarily close to) 1 that are unschedulable with m processor cores. Since our model is a generalization of theirs, this lower bound still holds for our tasks, and conventional utilization bound approaches are not useful for schedulability analysis of parallel tasks. Hence, like Lakshmanan et al. (2010), we use the *resource augmentation bound approach*, originally introduced by Phillips et al. (1997). We first consider *global scheduling* where tasks are allowed to migrate among processor cores. We then analyze schedulability in terms of a resource augmentation bound. Since the synchronous parallel tasks are now split into individual sequential subtasks, we can use global Earliest Deadline First (EDF) scheduling for them. The global EDF policy for subtask scheduling is basically the same as the traditional global EDF where jobs with earlier deadlines are assigned higher priorities.

Under global EDF scheduling, we now present a schedulability analysis in terms of a resource augmentation bound for our decomposed tasks. For any task set, the *resource augmentation bound* ν of a scheduling policy $\mathbb{A}$ on a multi-core processor with m cores represents a processor speedup factor. That is, if there exists any (optimal) algorithm under which a task set is feasible on m identical unit-speed processor cores, then $\mathbb{A}$ is guaranteed to successfully schedule this task set on a m-core processor, where each processor core is ν times as fast as the original. In Theorem 5, we show that our decomposition needs a resource augmentation bound of 4 under global EDF scheduling.

Our analysis uses a result for constrained deadline sporadic sequential tasks on m processor cores proposed by Baruah (2007) as re-stated here in Theorem 4. This result is a generalization of the result for implicit deadline sporadic tasks (Goossens et al., 2003).

Theorem 4 *(Baruah, 2007) Any constrained deadline sporadic sequential task set π with total density $\delta_{\text{sum}}(\pi)$ and maximum density $\delta_{\max}(\pi)$ is schedulable using global EDF strategy on m unit-speed processor cores if*

$$\delta_{\text{sum}}(\pi) \leq m - (m-1)\delta_{\max}(\pi)$$

Since we decompose our synchronous parallel tasks into sequential tasks with constrained deadlines, this result applies to our decomposed task set τ^{decom}. If we can schedule τ^{decom}, then we can schedule τ (Lemma 3).

Theorem 5 *If there exists any way to schedule a synchronous parallel task set τ on m unit-speed processor cores, then the decomposed task set τ^{decom} is schedulable using global EDF on m processor cores each of speed* 4.

Proof Let there exist some algorithm $\mathbb{A}$ under which the original task set τ is feasible on m identical unit-speed processor cores. If τ is schedulable under $\mathbb{A}$, the following condition must hold (by Observation 2).

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \tag{17}$$

We decompose tasks considering that each processor core has speed 2. To be able to schedule the decomposed tasks τ^{decom}, suppose we need to increase the speed of each processor core ν times further. That is, we need each processor core to be of speed 2ν.

According to the definition of density, if a task has density x on a processor system S, then its density will be $\frac{x}{\nu}$ on a processor system that is ν times as fast as S. On an m-core platform where each processor core has speed 2ν, let the total density and the maximum density of task set τ^{decom} be denoted by $\delta_{\text{sum},2\nu}$ and $\delta_{\text{max},2\nu}$, respectively. From 16, we have

$$\delta_{\text{max},2\nu} = \frac{\delta_{\text{max},2}}{\nu} \leq \frac{1}{\nu} \tag{18}$$

The value $\delta_{\text{sum},2\nu}$ turns out to be the total density of all decomposed tasks. A task has a density of at most $\frac{C_i/2}{T_i - P_i/2}$ on 2-speed processor cores after decomposition. Therefore, its density on 2ν-speed cores is at most $\frac{\frac{C_i/2}{T_i - P_i/2}}{\nu} = \frac{C_i/2\nu}{T_i - P_i/2}$, where we cannot replace $P_i/2$ with $P_i/2\nu$. Although the critical path length on 2ν-speed cores is $P_i/2\nu$, our decomposition was performed on 2-speed cores where critical path length was $P_i/2$. By Theorem 2 and Equation 1, the density of every task τ_i^{decom} on m identical processors each of speed 2ν is

$$\begin{aligned} \delta_{i,2\nu} &\leq \frac{\frac{C_i}{2\nu}}{T_i - \frac{P_i}{2}} \leq \frac{\frac{C_i}{2\nu}}{T_i - \frac{T_i}{2}} && \text{(since } P_i \leq T_i\text{)} \\ &= \frac{\frac{C_i}{2\nu}}{\frac{T_i}{2}} = \frac{1}{\nu}.\frac{C_i}{T_i} \end{aligned}$$

Thus, from 17,

$$\delta_{\text{sum},2\nu} = \sum_{i=1}^{n} \delta_{i,2\nu} \leq \frac{1}{\nu} \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{m}{\nu} \tag{19}$$

Note that, in the decomposed task set, every thread of the original task is a sequential task on a multiprocessor platform. Therefore, $\delta_{\text{sum},2\nu}$ is the total density of all threads (i.e. subtasks), and $\delta_{\text{max},2\nu}$ is the maximum density

among all threads. Thus, by Theorem 4, the decomposed task set τ^{decom} is schedulable under global EDF on m processor cores each of speed 2ν if

$$\delta_{\text{sum},2\nu} \leq m - (m-1)\delta_{\text{max},2\nu} \tag{20}$$

Now using the values of $\delta_{\text{sum},2\nu}$ (Equation 19) and $\delta_{\text{max},2\nu}$ (Equation 18) into Condition (20), task set τ^{decom} is schedulable if

$$\begin{aligned} & \frac{m}{\nu} \leq m - (m-1)\frac{1}{\nu} \\ \Leftrightarrow \quad & \frac{1}{\nu} + \frac{1}{\nu} - \frac{1}{m\nu} \leq 1 \quad \Leftrightarrow \quad \frac{2}{\nu} - \frac{1}{m\nu} \leq 1 \end{aligned}$$

From the above condition, τ^{decom} must be schedulable if

$$\frac{2}{\nu} \leq 1 \quad \Leftrightarrow \quad \nu \geq 2 \quad \Leftrightarrow \quad 2\nu \geq 4$$

Hence follows the theorem. □

5 Partitioned Deadline Monotonic Scheduling

Using the same decomposition described in Section 3, we now derive a resource augmentation bound required to schedule task sets under partitioned Deadline Monotonic (DM) scheduling. Unlike global scheduling, in *partitioned scheduling*, each task is assigned to a processor core. Tasks are executed only on their assigned processor cores, and are not allowed to migrate among cores. We consider the FBB-FFD (Fisher Baruah Baker - First-Fit Decreasing) partitioned DM scheduling proposed by Fisher et al. (2006) which Lakshmanan et al. (2010) also uses as the scheduling strategy for parallel tasks in a more restricted model. In fact, the FBB-FFD Algorithm was developed for periodic tasks without release offsets while our decomposed subtasks have offsets. Therefore, first we present how the FBB-FFD Algorithm should be adapted to partition our subtasks with offsets, and then we analyze the resource augmentation bound.

5.1 FBB-FFD based Partitioned DM Algorithm for Decomposed Tasks

The original FBB-FFD Algorithm by Fisher et al. (2006) is a variant of the first-fit decreasing bin-packing heuristic, and hinges on the notion of a request-bound function for constrained deadline periodic sequential tasks. For a sequential task π_i with execution requirement e_i, utilization u_i, and deadline d_i, its *request-bound function* $\text{RBF}(\pi_i, t)$ for any time interval of length t is the largest cumulative execution requirement of all jobs that can be generated by π_i to have their arrival times within a contiguous interval of length t. In the FBB-FFD Algorithm, $\text{RBF}(\pi_i, t)$ is approximated as

$$\text{RBF}^*(\pi_i, t) = e_i + u_i t \tag{21}$$

Let the processor cores be indexed as $1, 2, \cdots, m$, and Π_q be the set of tasks already assigned to processor core q, where $1 \leq q \leq m$. Considering the tasks in decreasing DM-priority order and starting from the highest priority task, the FBB-FFD algorithm assigns a task π_i to the first processor core q, $1 \leq q \leq m$, that satisfies the following condition

$$d_i - \sum_{\pi_j \in \Pi_q} \text{RBF}^*(\pi_j, d_i) \geq e_i \tag{22}$$

If no processor core satisfies the above condition for some task, then the task set is decided to be infeasible for partitioning.

To partition our decomposed subtasks based on the FBB-FFD algorithm, we need to adopt Condition (22) for decomposed subtasks. In τ_i^{decom}, let any subtask that belongs to the k-th segment in τ_i be denoted by $\tau_{i,k}$. Let the deadline and the worst case execution requirement of $\tau_{i,k}$ be denoted by $d_{i,k}$ and $e_{i,k}$, respectively. We need to update the expression of RBF* in Condition (22) by taking into account the release offsets. The RBF* of the subtasks of τ_j^{decom} that are assigned to processor core q for any interval of length t is denoted by $\text{RBF}_q^*(\tau_j^{\text{decom}}, t)$. Note that any two subtasks of τ_j^{decom} having different offsets belong to different segments in original task τ_j and hence are never active simultaneously. Therefore, the calculation of $\text{RBF}_q^*(\tau_j^{\text{decom}}, d_{i,k})$ in Condition (22) when $j \neq i$ is different from that when $j = i$.

To calculate $\text{RBF}_q^*(\tau_j^{\text{decom}}, d_{i,k})$ when $j = i$, we only need to calculate RBF* of all subtasks in the k-th segment that are assigned to processor core q, and call it RBF* of that segment (of τ_i) on core q. Since the subtasks of τ_i^{decom} that are in different segments will never be released and executed simultaneously, the sum of RBF* among all subtasks of the k-th segment assigned to core q is the $\text{RBF}_q^*(\tau_i^{\text{decom}}, d_{i,k})$. Let $m_{i,k,q}$ be the number of subtasks of τ_i^{decom} that belong to the k-th segment in τ_i and have already been assigned to processor core q. Thus,

$$\text{RBF}_q^*(\tau_i^{\text{decom}}, d_{i,k}) = \text{RBF}^*(\tau_{i,k}, d_{i,k}).m_{i,k,q} \tag{23}$$

To calculate $\text{RBF}_q^*(\tau_j^{\text{decom}}, d_{i,k})$ when $j \neq i$, we need to consider the worst case RBF* of a segment execution sequence starting from subtasks of any segment of task τ_j that are assigned to processor core q. Let $\text{RBF}_q^*(\tau_{j,l}^+, d_{i,k})$ be the RBF* of the segment execution sequence $\tau_{j,l}^+$ starting from subtasks of the l-th segment (of task τ_j) on core q. Since subtasks of τ_j^{decom} that are from different segments are never simultaneously active, we need to consider this in calculating $\text{RBF}_q^*(\tau_{j,l}^+, d_{i,k})$, where $j \neq i$. To do so, the approximation of $u_i d_{i,k}$ (considering $t = d_{i,k}$) in Equation (21) is modified to the sum of utilization of all the segments that have been assigned to core q, while the approximation of e_i is limited to the real possible interference starting from the l-th segment to the following segments that could be released before time $(d_{i,k} \bmod T_j)$. Let $r_{j,p}$ be the release time of the p-th segment of task τ_j, determined by our decomposition. Note that the first segment will be executed after the last segment in the same task, for at least one period after its last release time. So

the relative release time of the p-th segment, given that the l-th segment starts at relative time zero, will be $(r_{j,p} + T_j - r_{j,l}) \bmod T_j$. As long as the relative release time is no greater than deadline $d_{i,k}$, the segment will be executed before the execution of the k-th segment, which is released at relative time zero. Hence, $\text{RBF}^*_q(\tau^+_{j,l}, d_{i,k})$ is expressed as follows.

$$\text{RBF}^*_q(\tau^+_{j,l}, d_{i,k}) = \sum_{\tau_{j,p} \in \Pi_q, (r_{j,p}+T_j-r_{j,l}) \bmod T_j \leq d_{i,k}} e_{j,p}.m_{j,p,q} + \sum_{\tau_{j,p} \in \Pi_q} u_{j,p}.m_{j,p,q}.d_{i,k} \tag{24}$$

Considering all possible segment execution sequences of task τ_j, the maximum RBF* on core q is an upper bound of $\text{RBF}^*_q(\tau^{\text{decom}}_j, d_{i,k})$. We simply use this upper bound for the value of $\text{RBF}^*_q(\tau^{\text{decom}}_j, d_{i,k})$, where $j \neq i$ in the FBB-FFD Algorithm. Thus,

$$\text{RBF}^*_q(\tau^{\text{decom}}_j, d_{i,k}) = \max\left\{\text{RBF}^*_q(\tau^+_{j,l}, d_{i,k}) \middle| 1 \leq l \leq s_i\right\} \tag{25}$$

Now the decomposed subtasks are partitioned based on the FBB-FFD Algorithm in the following way. Recall that $d_{i,j}$ is the deadline and $e_{i,j}$ is the execution requirement of subtask $\tau_{i,j}$ (a subtask from the j-th segment of task τ_i). Let us consider Π_q to be the set of decomposed tasks τ^{decom}_i whose (one or more) subtasks have been assigned to processor core q. To assign a subtask to a processor core, the FBB-FFD based partitioned DM Algorithm sorts the unassigned subtasks in non-increasing DM-priority order. Let at any time $\tau_{i,j}$ be the highest priority subtask among the unassigned ones. Then the algorithm performs the following assignments. Subtask $\tau_{i,j}$ is assigned to the first processor core q, $1 \leq q \leq m$, that satisfies the following condition

$$d_{i,j} - \sum_{\tau^{\text{decom}}_k \in \Pi_q} \text{RBF}^*_q(\tau^{\text{decom}}_k, d_{i,j}) \geq e_{i,j} \tag{26}$$

If no such q, $1 \leq q \leq m$, exists that satisfies the above condition for $\tau_{i,j}$, then the task set is determined to be infeasible for partitioning. Note that RBF* is calculated using Equation (23) if $k = i$, and using Equation (25), otherwise.

From our above discussions, the value of $\sum_{\tau^{\text{decom}}_k \in \Pi_q} \text{RBF}^*_q(\tau^{\text{decom}}_k, d_{i,j})$ used in Condition (26) is no less than the total RBF* of all subtasks assigned to processor core q. Since any task for which processor core q satisfies Condition (22) is DM schedulable on q (according to the FBB-FFD Algorithm), any subtask $\tau_{i,j}$ for which processor core q satisfies Condition (26) must be DM schedulable on q.

5.2 Analysis for the FBB-FFD based Partitioned DM Algorithm

We use an analysis similar to the one used by Lakshmanan et al. (2010) to derive the resource augmentation bound as shown in Theorem 6. The analysis is based on the demand bound function of the tasks after decomposition.

Definition 5 The *demand bound function* (DBF), originally introduced by Baruah et al. (1990), of a task τ_i is the largest cumulative execution requirement of all jobs generated by τ_i that have both their arrival times and their deadlines within a contiguous interval of t time units. For a task τ_i with a maximum computation requirement of C_i, a period of T_i, and a deadline of D_i, its DBF is given by

$$\mathrm{DBF}(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right)$$

Definition 6 Based upon the DBF function, the *load* of task system τ, denoted by $\lambda(\tau)$, is defined as follows.

$$\lambda(\tau) = \max_{t>0} \left(\frac{\sum_{i=1}^{n} \mathrm{DBF}(\tau_i, t)}{t} \right)$$

From Definition 5, for a constrained deadline task τ_i:

$$\begin{aligned} \mathrm{DBF}(\tau_i, t) &\leq \max\left(0, \left(\left\lfloor \frac{t - D_i}{D_i} \right\rfloor + 1\right) C_i\right) \\ &\leq \left\lfloor \frac{t}{D_i} \right\rfloor C_i \ \leq \frac{t}{D_i}.C_i \ = \delta_i.t \end{aligned}$$

Based on the above analysis, we now derive an upper bound of DBF for every task after decomposition. Every segment of task τ_i consists of a set of constrained deadline subtasks after decomposition and, by Lemma 1, the total density of all subtasks in a segment is at most $\frac{C_i/2}{T_i - P_i/2}$. The constrained deadline subtasks are offset to ensure that those belonging to different segments of the same task are never simultaneously active. That is, for each task τ_i, each segment (and each of its subtasks) happens only once during the activation time of τ_i. Therefore, for decomposed task τ_i^{decom}, considering the segment having the maximum density in place of every segment gives an upper bound on the total density of all subtasks of τ_i^{decom}. Since, the density $\delta_{i,j}$ of any j-th segment of τ_i^{decom} is at most $\frac{C_i/2}{T_i - P_i/2}$, the DBF of τ_i^{decom} over any interval of length t is

$$\mathrm{DBF}(\tau_i^{\mathrm{decom}}, t) \leq \frac{C_i/2}{T_i - P_i/2}.t$$

The load of the decomposed task system τ^{decom} is

$$\lambda(\tau^{\mathrm{decom}}) = \max_{t>0} \left(\frac{\sum_{i=1}^{n} \mathrm{DBF}(\tau_i^{\mathrm{decom}}, t)}{t} \right) \leq \sum_{i=1}^{n} \frac{C_i/2}{T_i - P_i/2} \tag{27}$$

Theorem 6 *If there exists any (optimal) algorithm under which a synchronous parallel task set τ is schedulable on m unit-speed processor cores, then its decomposed task set τ^{decom} is schedulable using the FBB-FDD based partitioned DM Algorithm on m identical processor cores each of speed 5.*

Proof Fisher et al. (2006) proves that any constrained deadline sporadic task set π with total utilization $u_{\text{sum}}(\pi)$, maximum density $\delta_{\max}(\pi)$, and load $\lambda(\pi)$ is schedulable by the FBB-FFD Algorithm on m unit-speed processor cores if

$$m \geq \frac{\lambda(\pi) + u_{\text{sum}}(\pi) - \delta_{\max}(\pi)}{1 - \delta_{\max}(\pi)}$$

Using the same method used by Fisher et al. (2006) for proving the above sufficient schedulability condition, it can be shown that our decomposed (sub)tasks τ^{decom} are schedulable by the FBB-FFD based partitioned DM scheduling (presented in Subsection 5.1) on m unit-speed processor cores if

$$m \geq \frac{\lambda(\tau^{\text{decom}}) + u_{\text{sum}}(\tau^{\text{decom}}) - \delta_{\max}(\tau^{\text{decom}})}{1 - \delta_{\max}(\tau^{\text{decom}})} \tag{28}$$

where $\delta_{\max}(\tau^{\text{decom}})$, $u_{\text{sum}}(\tau^{\text{decom}})$, and $\lambda(\tau^{\text{decom}})$ denote the maximum density, total utilization, and load, respectively, of τ^{decom} on unit-speed processor cores.

We decompose tasks considering that each processor core has speed 2. To be able to schedule the decomposed tasks τ^{decom}, suppose we need to increase the speed of each processor core ν times further. That is, we need each processor core to be of speed 2ν. Let the maximum density, total utilization, and load of task set τ^{decom} be denoted by $\delta_{\max,2\nu}$, $u_{\text{sum},2\nu}$, and $\lambda_{2\nu}$ respectively, when each processor core has speed 2ν. Using these notations in Condition (28), task set τ^{decom} is schedulable by the FBB-FFD based partitioned DM Algorithm on m identical processor cores each of speed 2ν if

$$m \geq \frac{\lambda_{2\nu} + u_{\text{sum},2\nu} - \delta_{\max,2\nu}}{1 - \delta_{\max,2\nu}} \tag{29}$$

From Equation 1:

$$u_{\text{sum},2\nu} = \sum_{i=1}^{n} \frac{\frac{C_i}{2\nu}}{T_i} = \frac{1}{2\nu} \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{u_{\text{sum}}}{2\nu} \tag{30}$$

From Equations 1 and 27:

$$\lambda_{2\nu} \leq \sum_{i=1}^{n} \frac{\frac{C_i}{2\nu}}{T_i - \frac{P_i}{2}} \leq \sum_{i=1}^{n} \frac{\frac{C_i}{2\nu}}{T_i - \frac{T_i}{2}} = \frac{1}{\nu} \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{u_{sum}}{\nu} \tag{31}$$

Using Equations 31, 30, 18 in Condition (29), task set τ^{decom} is schedulable if

$$m \geq \frac{\frac{u_{\text{sum}}}{\nu} + \frac{u_{\text{sum}}}{2\nu} - \frac{1}{\nu}}{1 - \frac{1}{\nu}}$$

If the original parallel task set τ is schedulable by any algorithm on m unit-speed processor cores, then $u_{\text{sum}} \leq m$. Therefore, τ^{decom} is schedulable if

$$m \geq \frac{\frac{m}{\nu} + \frac{m}{2\nu} - \frac{1}{\nu}}{1 - \frac{1}{\nu}} \quad \Leftarrow \quad 2\nu - 2 \geq 3 \quad \Leftrightarrow \quad 2\nu \geq 5$$

Hence follows the theorem. □

6 Generalizing to a Unit-node DAG Task Model

In the analysis presented so far, we have focused on synchronous parallel tasks. That is, there is a synchronization point at the end of each segment, and the next segment starts only after all the threads of the previous segment have completed. In this section, we show that even more general parallel tasks that can be represented as directed acyclic graphs (DAGs) with unit time nodes can be easily converted into synchronous tasks. Therefore, the above analysis holds for these tasks as well.

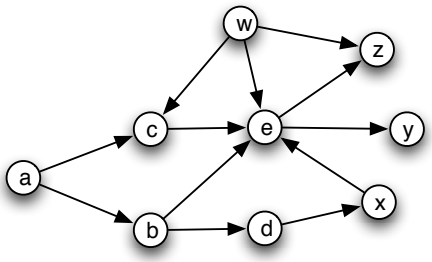
In the unit-node DAG model of tasks, each job is made up of nodes that represent work, and edges that represent dependencies between nodes. Therefore, a node can execute only after all of its predecessors have been executed. We consider the case where each node represents unit-time work. Therefore, a unit-node DAG can be converted into a synchronous task by simply adding new dependence edges as explained below.

If there is an edge from node u to node v, we say that u is the *parent* of v. Then we calculate the depth, denoted by $h(v)$, of each node v. If v has no parents, then it is assigned depth 1. Otherwise, we calculate the depth of v as

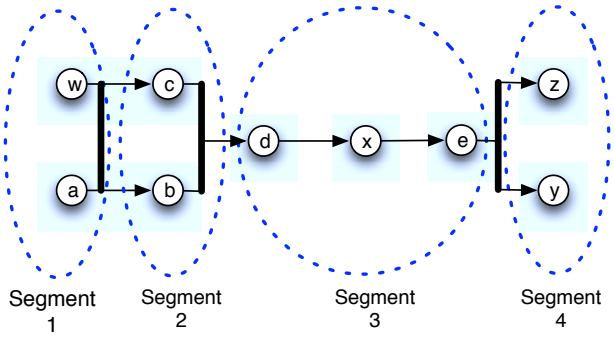
$$h(v) = \max_{u \text{ parent of } v} h(u) + 1$$

Each node with depth j is assigned to segment j. Then every node of the DAG is considered as a *thread* in the corresponding segment. The threads in the same segment can happen in parallel, and the segment is considered as a parallel segment of a synchronous task. If there are $k > 1$ consecutive segments each consisting of just one thread, then all these k segments are considered as one sequential segment of execution requirement k (by preserving the sequence). Figure 4 shows an example, where a DAG in Figure 4(a) is represented as a synchronous task in Figure 4(b). This transformation is valid since it preserves all dependencies in the DAG, and in fact only adds extra dependencies .

Upon representing a unit-node DAG task as a synchronous task, we perform the same decomposition proposed in Section 3. The decomposed task set can be scheduled using either global EDF or partitioned DM scheduling. Note that the transformation from a DAG task τ_i to a synchronous task preserves the work C_i of τ_i. Hence, the condition $\sum C_i/T_i \leq m$ used in our analysis still holds. Besides, the transformation preserves the critical path length P_i of τ_i and, hence, the rest of the analysis also holds. Therefore, a set of unit-node DAG tasks can be scheduled with a resource augmentation bound of 4 under global EDF scheduling, and of 5 under partitioned DM scheduling.



(a) Unit-node DAG



(b) Parallel synchronous model

Fig. 4 Unit-node DAG to parallel synchronous model

7 Evaluation

In this section, we evaluate the proposed decomposition through simulations. We randomly generate synchronous parallel tasks, decompose them, and simulate their schedules under global EDF and partitioned DM policies considering multi-core processors with different number of cores. We validate the derived resource augmentation bounds by considering different speeds of the processor cores.

7.1 Task Generation

In our simulation studies, parallel synchronous task sets are generated in the following way. The number of segments of each task is randomly selected from the range $[10, 30]$. The number of threads in each segment is randomly selected from the range $[1, 90]$. The execution requirements of the threads in a segment are selected randomly from the range $[5, 35]$. Each task is assigned a valid harmonic period (i.e. period is no less than its critical path length) of the form 2^k, where k is chosen from the range $[6, 13]$. We generate task sets considering

$m = 20, 40$, and 80 (i.e. for 20-core, 40-core, and 80-core processors). For each value of m, we generate 1000 task sets.

We want to evaluate our scheduler on task sets that an optimal scheduler could schedule on 1-speed processors. However, as we cannot compute this ideal scheduler, we assume that an ideal scheduler can schedule any task set that satisfies two conditions: (1) total utilization is no greater than m, and (2) each individual task is schedulable in isolation, that is, the deadline is no smaller than the critical path length. The second condition is implicitly satisfied by the way we assign period (which is the same as the deadline for our tasks) to tasks. Therefore, in our experiments, to generate a task set for each value of m (m being the number of processor cores), we keep adding tasks to the set as long as their total utilization does not exceed m but is at least 98% of m, thereby (almost) fully loading a machine of speed 1 processors. Since a system with a larger value of m is able to schedule a task set with higher utilization, the task sets generated for different values of m are different. According to above parameters, the generated task sets for 20-core, 40-core, and 80-core processors have the average number of tasks per task set of 4.893, 6.061, and 8.791, respectively, and the average ratios for total utilization to the number of cores are 99.3%, 99.2%, and 99.1% respectively.

7.2 Simulation Setup

The tasks generated for a particular value of m are decomposed using our proposed decomposition technique. The decomposed tasks are then scheduled by varying the speed of the cores on a simulated multi-core platform. We evaluate the performance in terms of *failure ratio* defined as the proportion of the number of unschedulable task sets to the total number of task sets attempted. We consider partitioned DM and two versions of global EDF scheduling policies. Note that under partitioned DM (P-DM), to make the analysis work, segments cannot be released before the decomposed release offsets, while under global EDF a segment can either wait till its relative release time, or start as soon as its preceding segments complete. Our analysis holds for both versions of the global EDF policies, and here we evaluate both of them to understand if they perform differently in simulations. The policy where subtasks wait until their release offset is called standard global EDF (G-EDF) and the policy where tasks are released as soon their dependences are satisfied is called greedily synchronized global EDF (GSG-EDF).

For all three methods, two kinds of failure ratios are compared. For the first kind marked with "simu", we measure the actual failure ratio in that a task-set is considered unschedulable if any task in the task set misses its deadline. For the other kind marked with "test", a task set is considered unschedulable if any subtask (some thread in a decomposed segment) misses its deadline. Note that the overall task may still be scheduled successfully if a subtask misses deadlines. Therefore, "test" failure ratios are always no better (and generally worse than) than "simu" ones. Since subtask deadlines are assigned

by the system and do not represent real constraints, a subtask deadline miss is not important to a real system. However, we include this result in order to throughly investigate the safety of our analytical bounds.

Note that only the "simu" results reflect the job deadline misses, which can be compared with other methods without any decomposition. Besides all the simulation results, for P-DM, we also include the failure ratio of the analysis. The failure ratio marked by "P-DM-analysis" indicates the ratio of the number of task sets that the offset-aware FBB-FFD Algorithm failed to partition with guaranteed schedulability to the total number of task sets attempted. Note that since this partitioning algorithm is also pessimistic, there are some task sets that the algorithm can not guarantee, but in simulation all their deadlines are met anyway. Therefore, this failure ratio is generally worse than P-DM-test.

In Figures 5, 6, and 7, the failure ratios under G-EDF, GSG-EDF, and P-DM are compared on 20, 40 and 80 core machines, respectively. All cores always have the same speed, and we increase the speed gradually until all task sets are schedulable. In particular, we start by setting a speed of 1 (i.e. unit-speed) at every processor core. Then we keep increasing the speed of each core by 0.2 in every step, and schedule the same set of tasks using the increased speed.

7.3 Simulation Results

In the first set of simulations, we evaluate the schedulability of 1000 task sets generated for a 20-core processor. According to Figure 5, when each core has speed 1, the failure ratio under G-EDF-test is 0.988 and under G-EDF-simu is 0.27. That is, out of 1000 test cases, 988 cases of decomposed task sets are not schedulable (having at least one segment deadline miss in simulation), and 270 cases have at least one job deadline miss. As we gradually increase the speed of each core, the failure ratios for G-EDF-test decrease slowly and are much higher than those of G-EDF-simu. For example, when each core has speed of 1.2, 1.4, 1.6, and 1.8, the failure ratios for G-EDF-test are 0.969, 0.935, 0.9, and 0.849, respectively, while for G-EDF-simu the ratios are 0.169, 0.119, 0.097, and 0.084, respectively. When each core has speed 2.4 or more, all task sets are schedulable. Thus, the resource augmentation required for the tasks we have evaluated under G-EDF is only 2.4 for this simulation setting.

Note one interesting fact about these figures. When speed is increased from 1.8 to 2, we observe a very sharp decrease in failure ratios of G-EDF-test. Specifically, at speeds 2 and 2.2, only one task set is unschedulable. This sharp decrease happens due to the following reason. In our method, the original decomposition occurs on speed-2 processors. Therefore, some (sub)tasks may have density greater than 1 on processors of speeds smaller than 2, meaning that these subtasks can never meet their deadlines at speed lower than 2. Since the decomposition guarantees that the maximum density among all (sub)tasks

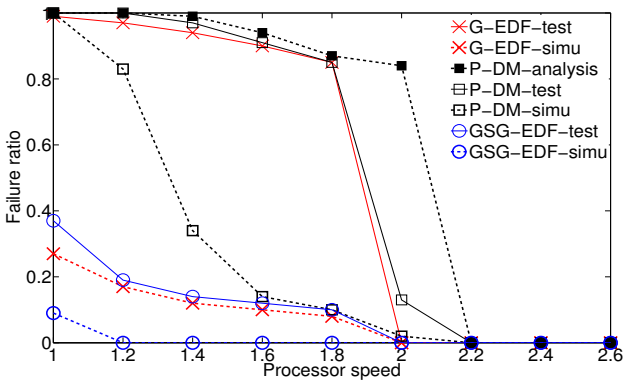


Fig. 5 Schedulability on a 20-core processor.

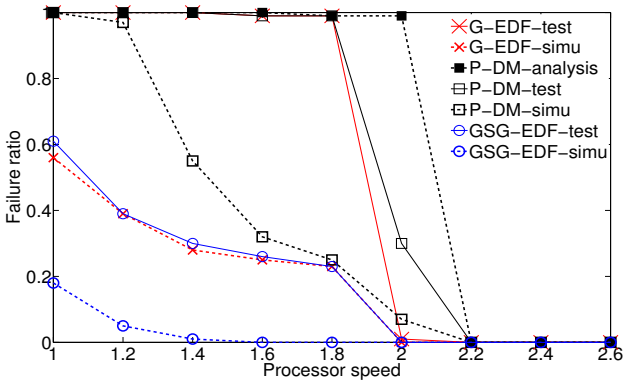


Fig. 6 Schedulability on a 40-core processor

is at most 1 at speed 2, many task sets that were unschedulable at speed 1.8 become schedulable at speed 2.

Unlike G-EDF, GSG-EDF with greedy synchronization does not wait for any segment release time. It can utilize the slack from the decomposed task sets and hence has better performance than G-EDF. The failure ratios in GSG-EDF-test are 0.37, 0.193, 0.135, 0.117 and 0.101 at speed 1, 1.2, 1.4, 1.6, and 1.8, respectively. It is slightly worse than that of G-EDF-simu, and is much better than those of G-EDF-test. Out of 1000 cases in GSG-EDF-simu, there are only 86 and 4 job deadline misses at speed 1 and 1.2, respectively, and no deadline misses when speed is greater than 1.2. Note that "test" of GSG-EDF also has a decrease at speed 2 for the same reason as G-EDF. However, GSG-EDF-simu (unlike G-EDF-test, G-EDF-simu, or GSG-EDF-test) does not experience this sharp decrease caused by decomposition process considering speed 2 processors. We speculate the reason is the following. The "simu" results basically measure if the last segment of the task misses a deadline. In

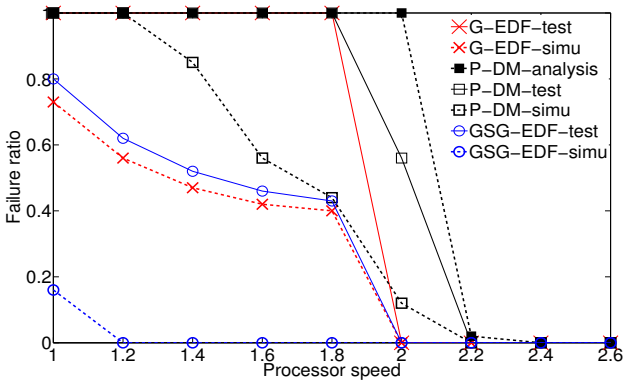


Fig. 7 Schedulability on a 80-core processor

the case of GSG-EDF, by the time the last segment comes around, the task has accumulated enough slack by starting the prior segments early. Therefore, even though the last segment's density may be larger than 1 (if it was released at its decomposed release time), it is released early enough that it can still generally meet its deadline.

Note that there are no theoretical guarantees about the comparison of job deadline misses between greedily synchronized global EDF for decomposed task sets and global EDF for original task sets. However, as we observe in practice, the resource augmentation required for the tasks we have evaluated under GSG-EDF is only 2.2, compared with 2.4 of G-EDF for the same 1000 task sets, suggesting that greedy synchronization provides some advantage over the standard version.

Besides G-EDF, Figure 5 also plots the failure ratios under P-DM (based on the offset-aware FBB-FFD Algorithm) on a 20-core processor. When each core has speed 1, all the 1000 test cases are unschedulable in both analysis and simulation under P-DM. With the increase in speed to 1.8, the failure ratios of P-DM-analysis and P-DM-test decrease slowly, while P-DM-simu decreases sharply. However, the result of P-DM-simu is not stable. A task set, which has no job deadline miss at certain speed, may result in job deadline misses even when the speed increases. This is because the partitioning results for the same task set under different processor speed are different. Since for P-DM, the segment release time must be followed, the job deadlines are just part of the overall segment deadlines. With different partitioning results, different parts of segment deadlines may be missed. Therefore, the job deadline misses of the same task set under different processor speed are not stable. The failure ratio of P-DM-test decreases sharply when reaching the speed 1.8 and the P-DM-analysis decreases sharply when reaching 2. At speed 2.4, only one task set cannot be scheduled according to the analysis while all can be scheduled in simulation, demonstrating a resource augmentation of 2.4 for this specific simulation setting which is smaller than our theoretical bound of 5 for P-DM.

These results seem to indicate that the global EDF with greedy synchronization is slightly better than standard global EDF which is slightly better than P-DM. Note that we should only care about the "simu" results when comparing across policies.

Similar experiments for 40 and 80 cores are shown in Figures 6 and 7 respectively. Note that these task sets are different from those generated for the previous set of simulations, and have higher total utilization (as these are generated for a higher number of cores). The curves follow similar trends. Namely, here also the "test" results are generally worse than the corresponding "simu" results, the P-DM-analysis results are slightly pessimistic and GSG-EDF is slightly better than G-EDF which is better than P-DM. On 40-core machines, the resource augmentation required by GSG-EDF, G-EDF and P-DM is 2.2, 2.6 and 2.6 respectively for these experiments. For 80 core machines, the respective resource augmentation requirements are 2.0, 2.4 and 3 respectively.

It is difficult to compare the results of the 20-core experiment with those of the 40-core and 80-core experiments since the task sets used in different experiments are different. There is no way of knowing if the differences we observe are due to the increase in the number of cores or because we happened to generate task sets of more or less inherent difficulty. However, the general trend (with the exception of GSG-EDF-simu when we go from 40 to 80 cores) seems to be that it is more difficult to schedule on a larger number of cores. This may be due to the fact that as the number of cores increases, we have a larger number of tasks and subtasks, increasing the number of deadlines, thereby increasing the chance of the event that a segment or job deadline miss occurs in a task set.

These results indicate that our theoretical bounds may be pessimistic for the particular cases we have experimented, since global EDF needs augmentation of at most 2.6 (in contrast to the analytical bound of 4) and P-DM needs augmentation of at most 3 (in contrast to the analytical bound of 5). Since the analytical bounds are guaranteed for any task set and for any number of processor cores, the results from our experiments indeed do not guarantee whether these bounds are very loose or tight in general, since the worst case task sets might have not appeared in our experiments. In addition, we have observed that the resource augmentation requirement slightly increases with the increase in the number of cores (and keeping the system almost fully loaded). Hence, it is conceivable that for thousands of processor cores and for very adversarial task sets, one might need an augmentation of 4 and 5 for global EDF and partitioned DM policies, respectively. This suggests some potential research direction towards exploring both better (smaller) augmentation bounds and tighter lower bounds.

8 Related Work

There has been extensive work on traditional multiprocessor real-time scheduling (Davis and Burns, 2011). Most of this work focuses on scheduling sequential

tasks on multiprocessor or multi-core systems. There has also been extensive work on scheduling of one or more parallel jobs on multiprocessors (Polychronopoulos and Kuck, 1987; Drozdowski, 1996; Deng et al., 1996; Arora et al., 1998; Bansal et al., 2004; Edmonds et al., 2003; Agrawal et al., 2006; Calandrino and Anderson, 2008; Calandrino et al., 2007a,b). However, the work in Polychronopoulos and Kuck (1987); Drozdowski (1996); Deng et al. (1996); Arora et al. (1998); Bansal et al. (2004); Edmonds et al. (2003); Agrawal et al. (2006) does not consider task deadlines, and that in Calandrino and Anderson (2008); Calandrino et al. (2007a,b) considers soft real-time scheduling. In contrast to the goal of a hard real-time system (i.e. to meet all task deadlines), in a *soft real-time system* the goal is to meet a certain subset of deadlines based on some application specific criteria.

There has been little work on hard real-time scheduling of parallel tasks. Anderson and Calandrino (2006) proposes the concept of *megatask* as a way to reduce miss rates in shared caches on multi-core platforms, and consider Pfair scheduling by inflating the weights of a megatask's component tasks. Preemptive fixed-priority scheduling of parallel tasks is shown to be NP-hard by Han and Lin (1989). Kwon and Chwa (1999) explores preemptive EDF scheduling of parallel task systems with linear-speedup parallelism. Wang and Cheng (1992) considers a heuristic for nonpreemptive scheduling. However, this work focuses on metrics like makespan (Wang and Cheng, 1992) or total work that meets deadline (Kwon and Chwa, 1999), and considers simple task models where a task is executed on up to a given number of processors.

Most of the other work on real time scheduling of parallel tasks also address simplistic task models. Jansen (2004), Lee and Lee (2006), and Collette et al. (2008) study the scheduling of *malleable tasks*, where each task is assumed to execute on a given number of cores or processors and this number may change during execution. Manimaran et al. (1998) studies non-preemptive EDF scheduling for *moldable tasks*, where the actual number of used processors is determined before starting the system and remains unchanged. Kato and Ishikawa (2009) addresses Gang EDF scheduling of moldable parallel task systems. They require the users to select at submission time the number of processors upon which a parallel task will run. They further assume that a parallel task generates the same number of threads as processors selected before the execution. In contrast, the parallel task model addressed in this paper allows tasks to have different numbers of threads in different stages, which makes our solution applicable to a much broader range of applications. For parallel tasks, Nelissen et al. (2012) has studied real-time scheduling to minimize the required number of processors. In contrast, we study schedulability of parallel tasks in terms of processor speed-up factor. Recently, processor speed up factor for a task represented as a DAG has been addressed by Baruah et al. (2012). But this work considers scheduling of a single task. In contrast, we consider scheduling a set of parallel tasks on multi-core platform.

Our work is most related to, and is inspired by, the recent work of Lakshmanan et al. (2010) on real-time scheduling for a restrictive synchronous parallel task model. In their model, every task is an alternate sequence of

parallel and sequential segments. All the parallel segments in a task have an *equal* number of threads, and that number *cannot exceed* the total number of processor cores. They also convert each parallel task into a set of sequential tasks, and then analyze the resource augmentation bound for partitioned DM scheduling. However, their strategy of decomposition is different from ours. They use a *stretch transformation* that makes a master thread of execution requirement equal to the task period, and assign one processor core exclusively to it. The remaining threads are scheduled using the FBB-FDD algorithm. Unlike ours, their results do not hold if, in a task, the number of threads in different segments vary, or exceed the number of cores. In addition, tasks that can be expressed as a DAG of unit time nodes cannot be converted to their task model in a work and critical path length conserving manner. Therefore, unlike ours, their analysis does not directly apply to these more general task models.

Our work in this paper has two major extensions beyond our previous work that appears as a conference version (Saifullah et al., 2011). First, we have added a detailed description of how the FBB-FFD partitioned deadline monotonic scheduling policy should be adopted for decomposed tasks with offsets. The conference version of the paper provided a resource augmentation bound for the FBB-FFD algorithm, and did not provide the detailed procedure for partitioning of the tasks with offsets. Note that the FBB-FFD algorithm, by default, is not offset-aware. Hence, we have provided a modified version of this algorithm to make it offset-aware. Second, we have provided evaluation results based on simulations. The conference version of the paper did not provide any evaluation result. In addition, we have presented how our results will hold for task models where a synchronous task may have some segment containing threads of different execution requirements. The conference version of the paper considered the synchronous task model where each segment in a task consists of equal-length threads.

9 Conclusion

With the advent of the era of multi-core computing, real-time scheduling of parallel tasks is crucial for real-time applications to exploit the power of multi-core processors. While recent research on real-time scheduling of parallel tasks has shown promise, the efficacy of existing approaches is limited by their restrictive parallel task models. To overcome these limitations, in this paper we have presented new results on real-time scheduling for generalized parallel task models. First, we have considered a general synchronous parallel task model where each task consists of segments, each having an arbitrary number of parallel threads. Then we have proposed a novel task decomposition algorithm that decomposes each parallel task into a set of sequential tasks. We have derived a resource augmentation bound of 4 under global EDF scheduling, which to our knowledge is the first resource augmentation bound for global EDF scheduling of parallel tasks. We have also derived a resource augmenta-

tion bound of 5 for partitioned DM scheduling. Finally, we have shown how to convert a task represented as a Directed Acyclic Graph (DAG) with unit time nodes into a synchronous task, thereby holding our results for this more general task model.

Through simulations, we have observed that bounds in practice are significantly smaller than the theoretical bounds. These results suggest many directions of future work. First, we can try to provide better bounds and/or provide lower bound arguments that suggest that the bounds are in fact tight. In the future, we also plan to consider even more general DAG tasks where nodes have arbitrary execution requirements, and to provide analysis requiring no transformation to synchronous model. We also plan to address system issues such as cache effects, preemption penalties, and resource contention.

Acknowledgements This research was supported by NSF under grants CNS-0448554 (CAREER) and CCF-1136073.

References

Agrawal K, He Y, Hsu WJ, Leiserson CE (2006) Adaptive task scheduling with parallelism feedback. In: PPoPP '06: Proceedings of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pp 100–109

Anderson JH, Calandrino JM (2006) Parallel real-time task scheduling on multicore platforms. In: RTSS '06: Proceedings of the 27th IEEE Real-Time Systems Symposium, pp 89–100

Arora NS, Blumofe RD, Plaxton CG (1998) Thread scheduling for multiprogrammed multiprocessors. In: SPAA '98: Proceedings of the 10th annual ACM Symposium on Parallel Algorithms and Architectures, pp 119–129

Bansal N, Dhamdhere K, Konemann J, Sinha A (2004) Non-clairvoyant scheduling for minimizing mean slowdown. Algorithmica 40(4):305–318

Baruah S (2007) Techniques for multiprocessor global schedulability analysis. In: RTSS '07: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp 119–128

Baruah S, Mok A, Rosier L (1990) Preemptively scheduling hard-real-time sporadic tasks on one processor. In: RTSS '90: Proceedings of the 11th IEEE Real-Time Systems Symposium, pp 182–190

Baruah S, Bonifaci V, Marchetti-Spaccamela A, Stougie L, Wiese A (2012) A generalized parallel task model for recurrent real-time processes. In: RTSS '12: Proceedings of the 33rd IEEE Real-Time Systems Symposium

Calandrino JM, Anderson JH (2008) Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In: ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems, pp 299–308

Calandrino JM, Anderson JH, Baumberger DP (2007a) A hybrid real-time scheduling approach for large-scale multicore platforms. In: ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems, pp 247–258

Calandrino JM, Baumberger D, Li T, Hahn S, Anderson JH (2007b) Soft real-time scheduling on performance asymmetric multicore platforms. In: RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium, pp 101–112

ClearSpeed (2008) CoSy compiler for 96-core multi-threaded array processor. `http://www.clearspeed.com/newsevents/news/ClearSpeed_Ace_011708.php`

Collette S, Cucu L, Goossens J (2008) Integrating job parallelism in real-time scheduling theory. Information Processing Letter 106(5):180–187

Davis RI, Burns A (2011) A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. ACM Computing Survey 43(4):35:1–44

Deng X, Gu N, Brecht T, Lu K (1996) Preemptive scheduling of parallel jobs on multiprocessors. In: SODA '96: Proceedings of the 7th annual ACM-SIAM Symposium on Discrete Algorithms, pp 159–167

Drozdowski M (1996) Real-time scheduling of linear speedup parallel tasks. Information Processing Letter 57(1):35–40

Edmonds J, Chinn DD, Brecht T, Deng X (2003) Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. Journal of Scheduling 6(3):231–250

Fisher N, Baruah S, Baker TP (2006) The partitioned scheduling of sporadic tasks according to static-priorities. In: ECRTS '06: Proceedings of the18th Euromicro Conference on Real-Time Systems, pp 118–127

Goossens J, Funk S, Baruah S (2003) Priority-driven scheduling of periodic task systems on multiprocessors. Real-Time Systems 25(2-3):187–205

Han CC, Lin KJ (1989) Scheduling parallelizable jobs on multiprocessors. In: RTSS '89: Proceedings of the 10th IEEE Real-Time Systems Symposium, pp 59–67

Huang HM, Tidwell T, Gill C, Lu C, Gao X, Dyke S (2010) Cyber-physical systems for real-time hybrid structural testing: a case study. In: ICCPS '10: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems, pp 69–78

Intel (2007) Teraflops research chip. `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`

Intel (2010) Cilk Plus. `http://software.intel.com/en-us/articles/intel-cilk-plus`

Jansen K (2004) Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. Algorithmica 39(1):59–81

Kato S, Ishikawa Y (2009) Gang EDF scheduling of parallel task systems. In: RTSS '09: Proceedings of the 30th IEEE Real-Time Systems Symposium, pp 459–468

Kwon OH, Chwa KY (1999) Scheduling parallel tasks with individual deadlines. Theoretical Computer Science 215(1-2):209–223

Lakshmanan K, Kato S, Rajkumar RR (2010) Scheduling parallel real-time tasks on multi-core processors. In: RTSS '10: Proceedings of the 30th IEEE Real-Time Systems Symposium, pp 259–268

Lee WY, Lee H (2006) Optimal scheduling for real-time parallel tasks. IEICE Transactions on Information and Systems E89-D(6):1962–1966

Manimaran G, Murthy CSR, Ramamritham K (1998) A new approach for scheduling of parallelizable tasks inreal-time multiprocessor systems. Real-Time Systems 15(1):39–60

Nelissen G, Berten V, Goossens J, Milojevic D (2012) Techniques optimizing the number of processors to schedule multi-threaded tasks. In: ECRTS '12: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 321–330

OpenMP (2011) OpenMP: Open multi-processing. `http://openmp.org`

Phillips CA, Stein C, Torng E, Wein J (1997) Optimal time-critical scheduling via resource augmentation (extended abstract). In: STOC '97: Proceedings of the 29th annual ACM symposium on Theory of Computing, pp 140–149

Polychronopoulos CD, Kuck DJ (1987) Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Transactions on Computers C-36(12):1425–1439

Saifullah A, Agrawal K, Lu C, Gill C (2011) Multi-core real-time scheduling for generalized parallel task models. In: RTSS '11: Proceedings of the 32nd IEEE Real-Time Systems Symposium, pp 217–226

Wang Q, Cheng KH (1992) A heuristic of scheduling parallel tasks and its analysis. SIAM Journal of Computing 21(2):281–294