# Supporting Soft Real-Time Parallel Applications on Multicore Processors[*]

Cong Liu and James H. Anderson

Department of Computer Science, University of North Carolina at Chapel Hill

## Abstract

*The prevalence of multicore processors has resulted in the wider applicability of parallel programming models such as OpenMP and MapReduce. A common goal of running parallel applications implemented under such models is to guarantee bounded response times while maximizing system utilization. Unfortunately, little previous work has been done that can provide such performance guarantees. In this paper, this problem is addressed by applying soft real-time scheduling analysis techniques. Analysis and conditions are presented for guaranteeing bounded response times for parallel applications under global EDF multiprocessor scheduling.*

## 1   Introduction

The growing prevalence of multicore platforms has resulted in the wider applicability of parallel programming models such as OpenMP [4] and MapReduce [6]. Such models can be applied to parallelize certain segments of programs, thus better utilizing hardware resources and possibly shortening response times. Many applications implemented under such parallel programming models have soft real-time (SRT) constraints. Examples include real-time parallel video and image processing applications [1, 8] and computer vision applications such as colliding face detection [13]. In these applications, providing fast and bounded response times for individual video frames is important, to ensure smooth video output. However, achieving this at the expense of using conservative hard real-time (HRT) analysis is not warranted. In this paper, we consider how to schedule parallel task systems that require such SRT performance guarantees on multicore processors.

Parallel task models pose new challenges to scheduling since intra-task parallelism has to be specifically considered. Recent papers [14, 18] on scheduling real-time periodic parallel tasks have focused on providing HRT guarantees under global-earliest-deadline-first (GEDF) or partitioned deadline-monotonic (DM) scheduling. However, as discussed above, viewing parallel tasks as HRT may be overkill in many settings and furthermore may result in significant schedulability-related utilization loss. Thus, our focus is to instead ensure bounded response times in supporting parallel task systems by applying SRT scheduling analysis techniques. Specifically, we assign deadlines to parallel tasks and schedule them

using GEDF, but in contrast to previous work [14, 18], we allow deadlines to be missed provided such misses are bounded (hence response times are bounded as well). Moreover, we consider a generalized parallel task model that removes some of the restrictions seen in previous work (as discussed below).

Response time bounds have been studied extensively in the context of global real-time scheduling algorithms such as GEDF. It has been shown that a variety of such algorithms can ensure bounded response times in ordinary real-time sporadic task systems (i.e., without intra-task parallelism) with no utilization loss on multiprocessors [7, 15].[1] Motivated by these results, we consider whether it is possible to specify reasonable constraints under which bounded response times can be guaranteed using global real-time scheduling techniques, for sporadic parallel task systems that are not HRT in nature.

**Related work.** Scheduling non-real-time parallel applications is a deeply explored topic [5, 6, 10, 11, 19, 20]. However, in all of just-cited work, scheduling decisions are made on a best-effort basis, so none of these results can provide performance guarantees such as response time bounds.

Regarding scheduling HRT parallel task systems, Lakshmanan et al. proposed a scheduling technique for the *fork-join* model, where a parallel task is a sequence of segments, alternating between sequential and parallel phases [14]. A sequential phase contains only one thread while a parallel phase contains multiple threads that can be executed concurrently on different processors. In their model, all parallel phases are assumed to have the same number of parallel threads, which must be no greater than the number of processors. Also, all threads in any parallel segment must have the same execution cost. The authors derived a resource augmentation bound of 3.42 under partitioned DM scheduling.

In [18], Saifullah et al. extended the fork-join model so that each parallel phase can have a different number of threads, but their model is still limited, as the number of threads in any parallel phase must not exceed the number of processors and all such threads must have the same execution cost. The authors proposed an approach that transforms each implicit-deadline periodic parallel task into a number of ordinary constrained-deadline periodic tasks by creating per-segment intermediate deadlines. They also showed that resource augmentation bounds of 2.62 and 3.42 can be achieved under GEDF and partitioned DM scheduling, respectively.

---

[1]Technically, bounded response times can only be ensured for task systems that do not over-utilize the underlying platform. In all claims in this paper concerning bounded response times, a non-over-utilized system is assumed.

In contrast to this prior work, we seek to efficiently support parallel task systems on multiprocessors with bounded response times. We generalize the parallel task models considered in [14, 18] by allowing the number of threads in any parallel segment to exceed the number of processors as well as allowing threads belonging to the same segment to have different execution costs. Also, we consider sporadic parallel tasks instead of periodic ones. A more fundamental difference is that we propose a SRT schedulability analysis framework to derive conditions for guaranteeing bounded response times.

**Contributions.** In this paper, we show that by assigning deadlines to parallel task systems and scheduling them under GEDF, such systems can be supported on multiprocessors with bounded response times. Our analysis shows that on a two-processor platform, no utilization loss results for any parallel task system. Despite this special case, on a platform with more than two processors, utilization constraints are needed. To discern how severe such constraints must fundamentally be, we present a parallel task set with minimum utilization that is unschedulable on any number of processors. This task set violates our derived constraint and has unbounded response times. The impact of utilization constraints can be lessened by restructuring tasks to reduce intra-task parallelism. We propose optimization techniques that can be applied to determine such a restructuring. Finally, we conduct experiments to evaluate the applicability of the derived schedulability condition.

**Organization.** The rest of this paper is organized as follows. Sec. 2 describes our system model. In Sec. 3, we present our analytical results. In Sec. 4, we discuss the above mentioned optimization technique. In Sec. 5, we experimentally evaluate the proposed analysis. Sec. 6 concludes.

## 2 System Model

We consider the problem of scheduling a set of $\tau = \{\tau_1, ..., \tau_n\}$ of $n$ independent sporadic parallel tasks on $m$ processors. Each parallel task $\tau_i$ is a sequence of $s_i$ segments, where the $j^{th}$ segment $\tau_i^j$ contains a set of $v_i^j$ threads ($v_i^j > m$ is allowed). The $k^{th}$ ($1 \leq k \leq v_i^j$) thread $\tau_i^{j,k}$ in segment $\tau_i^j$ has a worst-case execution time of $e_i^{j,k}$. We assume that each thread $\tau_i^{j,k}$ executes for exactly $e_i^{j,k}$ time units. This assumption can be eased to treat $e_i^{j,k}$ as an upper bound, at the expense of more cumbersome notation. For notational convenience, we order the threads of each segment $\tau_i^j$ of each parallel task $\tau_i$ in largest-worst-case-execution-time-first order. Thus, thread $\tau_i^{j,1}$ has the largest worst-case execution time among all threads in any segment $\tau_i^j$. For any segment $\tau_i^j$, if $v_i^j > 1$, then the threads in this segment can be executed in parallel on different processors. The threads in the $j^{th}$ segment can execute only after all threads of $(j-1)^{th}$ segment (if any) have completed. We let $v_i^{max}$ denote the maximum number of threads in any segment of task $\tau_i$. We assume $v_i^{max} \geq 2$
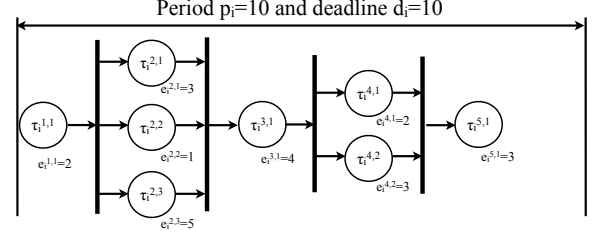


Figure 1: Example parallel task $\tau_i$. It has five segments where the second and fourth segments are parallel segments and contain three and two threads, respectively. This task has a worst-case execution cost of 23 time units, a period of 10 time units, and thus a utilization of 2.3.

holds for at least one task $\tau_i$; otherwise, the considered task system is simply an ordinary sporadic task system (without intra-task parallelism).

The worst-case execution time of any segment $\tau_i^j$ is defined as $e_i^j = \sum_{k=1}^{v_i^j} e_i^{j,k}$ (when all threads execute sequentially). The worst-case execution time of any parallel task $\tau_i$ is defined as $e_i = \sum_{j=1}^{s_i} e_i^j$ (when all threads in each segment of the task execute sequentially). We also need to determine the best-case execution time of $\tau_i$ on $m$ processors (when $\tau_i$ is the only task executing on $m$ processors), denoted $e_i^{min}$. In general, for any parallel task $\tau_i$, if we allow $v_i^{max} \geq m$ and threads in each segment have different execution costs, then the problem of calculating $e_i^{min}$ is equivalent to the problem of *minimum makespan scheduling* [12], where we treat each thread in a segment as an independent job and seek to obtain the minimum completion time for executing all such jobs on $m$ processors. This gives us per-segment best-case execution times, which can be summed to yield $e_i^{min}$. Unfortunately, this problem has been proved to be NP-hard [12]. This problem can be solved using a classical dynamic programming-based algorithm [12], which has exponential time complexity with respect to the per-segment thread count. However, for some special cases where certain restrictions on the task model apply, we can easily calculate $e_i^{min}$ in linear time. For example, when $v_i^{max} \leq m$ holds, $e_i^{min} = \sum_{j=1}^{s_i} e_i^{j,1}$ since in this case all threads of each segment of $\tau_i$ can be executed in parallel on $m$ processors and thread $\tau_i^{j,1}$ has the largest execution cost in each segment $\tau_i^j$. Moreover, when all threads in each segment have equal execution costs, $e_i^{min} = \sum_{j=1}^{s_i} \sum_{k=1}^{\lceil v_i^j/m \rceil} e_i^{j,1}$, because the execution of each segment $\tau_i^j$ can be viewed as the executions of $\lceil v_i^j/m \rceil$ sequential sub-segments, each with an equal execution cost of $e_i^{j,1}$.

We require $e_i^{min} \leq p_i$ for any task $\tau_i$; otherwise, response times (defined next) can grow unboundedly. An example parallel task is shown in Fig. 1.

Each parallel task is released repeatedly, with each such invocation called a *job*. The $k^{th}$ job of $\tau_i$, denoted $\tau_{i,k}$, is released at time $r_{i,k}$. Associated with each task $\tau_i$ is a period

2

Table 1: Summary of notation.

| | |
|---|---|
| $\tau_{i,h}^j$ | $j^{th}$ segment of the $h^{th}$ job of task $\tau_i$ |
| $\tau_{i,h}^{j,k}$ | $k^{th}$ thread of segment $\tau_i^j$ of the $h^{th}$ job of task $\tau_i$ |
| $s_i$ | Number of segments of task $\tau_i$ |
| $e_i^{j,k}$ | Worst-case execution cost of thread $\tau_i^{j,k}$ |
| $e_i^j$ | Worst-case execution cost of segment $\tau_i^j$ |
| $e_i$ | Worst-case execution cost of task $\tau_i$ |
| $e_i^{min}$ | Best-case execution cost of task $\tau_i$ |
| $v_i^{max}$ | Maximum number of threads in any segment of task $\tau_i$ |
| $v_{max_i}$ | Maximum number of threads of any segment of the task that has the $i^{th}$ maximum number of threads of any segment among all tasks |

$p_i$, which specifies the minimum time between two consecutive job releases of $\tau_i$. The utilization of a task $\tau_i$ is defined as $u_i = e_i/p_i$, and the utilization of the task system $\tau$ as $U_{sum} = \sum_{\tau_i \in \tau} u_i$. We require $U_{sum} \leq m$; otherwise, response times can grow unboundedly. For any job $\tau_{i,k}$ of task $\tau_i$, its $u^{th}$ segment is denoted $\tau_{i,k}^u$, and the $v^{th}$ thread of this segment is denoted $\tau_{i,k}^{u,v}$. For clarity, a summary of the important terms defined so far, as well as some additional terms defined later, is presented in Table 1.

Successive jobs of the same task are required to execute in sequence. If a job $\tau_{i,k}$ completes at time $t$, then its *response time* is $t - r_{i,k}$. A task's response time is the maximum response time of any of its jobs. Note that, when a job of a task completes after the release time of the next job of that task, this release time is not altered.

**Assigning deadlines.** For each parallel task $\tau_i$ with a period $p_i$, we assign it a relative deadline of $d_i = p_i$. Therefore, we can model each parallel task as a real-time sporadic parallel task with implicit deadlines. Any job $\tau_{i,k}$ of $\tau_i$ thus has an absolute deadline at $d_{i,k} = r_{i,k} + d_i$. Under GEDF, released jobs are prioritized by their absolute deadlines. We assume that ties are broken by task ID (lower IDs are favored).

## 3 Response Time Bound

We derive a response time bound for GEDF by comparing the allocations to a task system $\tau$ in a processor sharing (PS) schedule and an actual GEDF schedule of interest for $\tau$, both on $m$ processors, and quantifying the difference between the two. We analyze task allocations on a per-task basis.[2]

The time interval $[t_1, t_2)$, where $t_2 > t_1$, consists of all time instances $t$, where $t_1 \leq t < t_2$, and is of length $t_2 - t_1$. We assume time is discrete. For any time $t > 0$, the notation $t^-$ is used to denote the time $t - \varepsilon$ in the limit $\varepsilon \to 0+$, and the notation $t^+$ is used to denote the time $t + \varepsilon$ in the limit $\varepsilon \to 0+$.

---

[2]The SRT analysis framework used here has been adopted from a framework for ordinary sporadic task systems first proposed in [7], and subsequently used in several other papers [15–17].
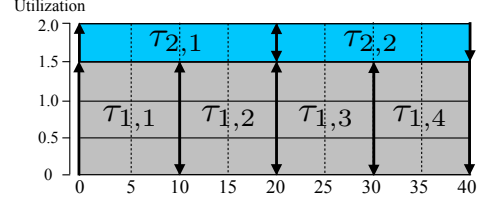


Figure 2: PS schedule for a task system containing two tasks. Task $\tau_1$ has a period of 10 time units and a utilization of 1.5. Task $\tau_2$ has a period of 20 time units and a utilization of 0.5. As seen in the PS schedule, intra-task parallelism is not considered and each job completes exactly at its deadline.

**Definition 1.** A task $\tau_i$ is *active* at time $t$ if there exists a job $\tau_{i,h}$ such that $r_{i,h} \leq t < d_{i,h}$.

**Definition 2.** Job $\tau_{i,h}$ is *pending* at time $t$ if $t \geq r_{i,h}$ and $\tau_{i,h}$ has not completed by $t$.

**Definition 3.** Job $\tau_{i,h}$ is *enabled* at $t$ if $t \geq r_{i,h}$, $\tau_{i,h}$ has not completed by $t$, and $\tau_{i,h-1}$ (if $h > 1$) has completed by $t$. Similarly, any thread in segment $\tau_{i,h}^k$ is *enabled* at $t$ if $t \geq r_{i,h}$, the thread has not completed by $t$, and all threads in segment $\tau_{i,h}^{k-1}$ (if any) have completed by $t$.

Let $A(\tau_{i,j}, t_1, t_2, S)$ denote the total allocation to the job $\tau_{i,j}$ in an arbitrary schedule $S$ in $[t_1, t_2)$. Then, the total time allocated to all jobs of $\tau_i$ in $[t_1, t_2)$ in $S$ is given by

$$A(\tau_i, t_1, t_2, S) = \sum_{j \geq 1} A(\tau_{i,j}, t_1, t_2, S).$$

Consider a PS schedule *PS*. In such a schedule, $\tau_i$ executes with the rate $u_i$ when it is active. (*Note that intra-task parallelism is not considered in the PS schedule.*) Thus, if $\tau_i$ is active throughout $[t_1, t_2)$, then

$$A(\tau_{i,j}, t_1, t_2, PS) = (t_2 - t_1)u_i. \quad (1)$$

Note that according to the parallel task model, the term $u_i$ in (1) could be greater than one. This is a key difference in comparison to most prior work where a PS schedule is considered. A PS schedule for an example task system is shown in Fig. 2.

The difference between the allocation to a job $\tau_{i,j}$ up to time $t$ in a PS schedule and an arbitrary schedule $S$, denoted *the lag of job $\tau_{i,j}$ at time $t$ in schedule $S$*, is defined by $lag(\tau_{i,j}, t, S) = A(\tau_{i,j}, 0, t, PS) - A(\tau_{i,j}, 0, t, S)$. The *lag of a task $\tau_i$ at time $t$ in schedule $S$* is given by

$$
\begin{aligned}
lag(\tau_i, t, S) &= \sum_{j \geq 1} lag(\tau_{i,j}, t, S) \\
&= A(\tau_i, 0, t, PS) - A(\tau_i, 0, t, S). \quad (2)
\end{aligned}
$$

The concept of lag is important because, if lags remain bounded, then response times are bounded as well. The *LAG*

for a finite job set $J$ at time $t$ in the schedule $S$ is defined as

$$LAG(J, t, S) = \sum_{\tau_{i,j} \in J} lag(\tau_{i,j}, t, S)$$
$$= \sum_{\tau_{i,j} \in J} (A(\tau_{i,j}, 0, t, PS) - A(\tau_{i,j}, 0, t, S)). \quad (3)$$

Our response time bound derivation focuses on a given task system $\tau$. We order jobs in $\tau$ by EDF, and break ties by task ID. Let $\tau_{l,j}$ be a job of a task $\tau_l$ in $\tau$, $t_d = d_{l,j}$, and $S$ be a GEDF schedule for $\tau$ with the following property.

**(P)** The response time of every job $\tau_{i,k}$ of higher priority than $\tau_{l,j}$ is at most $x + p_i + e_i$ in $S$, where $x \geq 0$.

Our objective is to determine the smallest $x$ such that the response time of $\tau_{l,j}$ is at most $x + p_l + e_l$. This would by induction imply a response time of at most $x + p_i + e_i$ for all jobs of every task $\tau_i$, where $\tau_i \in \tau$. We assume that $\tau_{l,j}$ finishes after $t_d$, for otherwise, its response time is trivially no greater than $p_l$. The steps for determining the value for $x$ are as follows.

1. Determine an upper bound on the work pending for tasks in $\tau$ that can compete with $\tau_{l,j}$ after $t_d$. This is dealt with in Lemmas 1 and 2 in Sec. 3.1.

2. Determine a lower bound on the amount of work pending for tasks in $\tau$ that can compete with $\tau_{l,j}$ after $t_d$, required for the response time of $\tau_{l,j}$ to exceed $x + p_l + e_l$. This is dealt with in Lemma 3 in Sec. 3.2.

3. Determine the smallest $x$ such that the response time of $\tau_{l,j}$ is at most $x + p_l + e_l$, using the above upper and lower bounds. This is dealt with in Theorem 1 in Sec. 3.3.

**Definition 4.** $\mathbf{d} = \{\tau_{i,h} : (d_{i,h} < t_d) \vee (d_{i,h} = t_d \wedge i \leq l)\}$.

$\mathbf{d}$ is the set of jobs with deadlines at most $t_d$ with priority at least that of $\tau_{l,j}$. These jobs do not execute beyond $t_d$ in the PS schedule. Note that $\tau_{l,j}$ is in $\mathbf{d}$. Also note that jobs not in $\mathbf{d}$ have lower priority than those in $\mathbf{d}$ and thus do not affect the scheduling of jobs in $\mathbf{d}$. For simplicity, we will henceforth assume that no job not in $\mathbf{d}$ executes in either the PS or GEDF schedule. To avoid distracting "boundary cases," we also assume that the schedule being analyzed is prepended with a schedule in which no deadlines are missed that is long enough to ensure that all previously released jobs referenced in the proof exist.

According to Property (P), job $\tau_{l,j-1}$ has a response time of at most $x + p_l + e_l$. Thus, the completion time of $\tau_{l,j-1}$, denoted $t_p$ ($p$ for predecessor), is given by

$$t_p \leq r_{l,j-1} + p_l + x + e_l \leq r_{l,j} + x + e_l = t_d - p_l + x + e_l. \quad (4)$$

**Definition 5.** A time instant $t$ is *busy* for a job set $J$ if all $m$ processors execute jobs in $J$ at $t$. A time interval is busy for $J$ if each instant within it is busy for $J$.

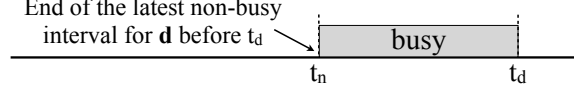The following claim follows from the definition of $LAG$.



Figure 3: Definition of $t_n$.

**Claim 1.** *If $LAG(\mathbf{d}, t_2, S) > LAG(\mathbf{d}, t_1, S)$, where $t_2 > t_1$, then $[t_1, t_2)$ is non-busy for $\mathbf{d}$. In other words, LAG for $\mathbf{d}$ can increase only throughout a non-busy interval.*

An interval could be non-busy for $\mathbf{d}$ only if there are not enough enabled jobs in $\mathbf{d}$ to occupy all available processors.

Since $\mathbf{d}$ includes all jobs of higher priority than $\tau_{l,j}$, the competing work for $\tau_{l,j}$ after time $t_d$ is given by the amount of work pending at $t_d$ for jobs in $\mathbf{d}$, which is given by $LAG(\mathbf{d}, t_d, S)$.

### 3.1 Upper Bound

In this section, we determine an upper bound on $LAG(\mathbf{d}, t_d, S)$. We first upper bound $lag(\tau_i, t, S)$ ($t \in [0, t_d]$) in Lemma 1 below. Then, in Lemma 2, we upper bound $LAG(\mathbf{d}, t_d, S)$ by summing individual task lags.

**Definition 6.** Let $t_n$ be the end of the latest non-busy interval for $\mathbf{d}$ before $t_d$, if any; otherwise, let $t_n = 0$ (see in Fig. 3).

By the above definition and Claim 1, we have

$$LAG(\mathbf{d}, t_d, S) \leq LAG(\mathbf{d}, t_n, S). \quad (5)$$

**Lemma 1.** $lag(\tau_i, t, S) \leq u_i \cdot x + (u_i + 1) \cdot e_i$ for any task $\tau_i$ and $t \in [0, t_d]$.

*Proof.* Let $d_{i,k}$ be the deadline of the earliest pending job of $\tau_i$, $\tau_{i,k}$, in the schedule $S$ at time $t$. If such a job does not exist, then $lag(\tau_i, t, S) = 0$, and the lemma holds trivially. Let $\gamma_i$ be the amount of work $\tau_{i,k}$ performs before $t$.

By the selection of $\tau_{i,k}$, we have

$$lag(\tau_i, t, S) = \sum_{h \geq k} lag(\tau_{i,h}, t, S)$$
$$= A(\tau_{i,k}, r_{i,k}, t, PS) - A(\tau_{i,k}, r_{i,k}, t, S)$$
$$+ \sum_{h > k} \big( A(\tau_{i,h}, r_{i,h}, t, PS)$$
$$- A(\tau_{i,h}, r_{i,h}, t, S) \big). \quad (6)$$

By the definition of $PS$, $A(\tau_{i,k}, r_{i,k}, t, PS) \leq e_i$, and $\sum_{h > k} A(\tau_{i,h}, r_{i,h}, t, PS) \leq u_i \cdot \max(0, t - d_{i,k})$ (the latter follows because each such job $\tau_{i,h}$ executes with rate $u_i$ in $PS$ while active, and the sum of the active intervals under consideration is at most $t - d_{i,k}$). By the selection of $\tau_{i,k}$, $A(\tau_{i,k}, r_{i,k}, t, S) = \gamma_i$, and $\sum_{h > k} A(\tau_{i,h}, r_{i,h}, t, S) = 0$. By setting these values into (6), we have

$$lag(\tau_i, t, S) \leq e_i - \gamma_i + u_i \cdot \max(0, t - d_{i,k}). \quad (7)$$

There are two cases to consider.

4

**Case 1.** $d_{i,k} \geq t$. In this case, (7) implies $lag(\tau_i, t, S) \leq e_i - \gamma_i \leq u_i \cdot x + (u_i + 1) \cdot e_i$.

**Case 2.** $d_{i,k} < t$. In this case, because $t \leq t_d$ and $d_{l,j} = t_d$, $\tau_{i,k}$ is not the job $\tau_{l,j}$. Thus, by Property (P), $\tau_{i,k}$ has a response time of at most $x + p_i + e_i$. Since $\tau_{i,k}$ is the earliest pending job of $\tau_i$ at time $t$, the earliest possible completion time of $\tau_{i,k}$ is at $t^+$. Thus, we have $t - r_{i,k} < t^+ - r_{i,k} \leq x + p_i + e_i$, which (because $d_{i,k} = r_{i,k} + p_i$) implies $t - d_{i,k} = t - r_{i,k} - p_i < x + e_i$.

Setting this value into (7), we have $lag(\tau_i, t, S) < e_i - \gamma_i + u_i \cdot (x + e_i) \leq u_i \cdot x + (u_i + 1) \cdot e_i$. $\square$

Lemma 2 below upper bounds $LAG(\mathbf{d}, t_d, S)$. We first define some needed terms.

**Definition 7.** Let $U$ be the sum of the $min(m - 1, n)$ largest task utilizations. Let $E$ be the largest value of the expression $\sum_{\tau_i \in \gamma} ((u_i + 1) \cdot e_i)$, where $\gamma$ denotes any set of $min(m - 1, n)$ tasks in $\tau$.

**Lemma 2.** $LAG(\mathbf{d}, t_d, S) \leq U \cdot x + E$.

*Proof.* By (5), we have $LAG(\mathbf{d}, t_d, S) \leq LAG(\mathbf{d}, t_n, S)$. By summing individual task lags at $t_n$, we can bound $LAG(\mathbf{d}, t_n, S)$. If $t_n = 0$, then $LAG(\mathbf{d}, t_n, S) = 0$, so assume $t_n > 0$. Consider the set of tasks $\beta = \{\tau_i : \exists \tau_{i,h}$ in $\mathbf{d}$ such that $\tau_{i,h}$ is enabled at $t_n^-\}$. Given that the instant $t_n^-$ is non-busy, there are not enough enabled jobs in $\mathbf{d}$ to occupy all $m$ processors. More precisely, there are not enough enabled threads belonging to jobs in $\mathbf{d}$ to occupy all $m$ processors. There could be at most $min(m - 1, n)$ parallel tasks that have enabled jobs at $t_n^-$ since each such parallel task has at least one enabled thread at $t_n^-$; that is, $|\beta| \leq min(m - 1, n)$.

If task $\tau_i$ does not have pending jobs at $t_n^-$, then $lag(\tau_i, t_n, S) \leq 0$. Therefore, we have $LAG(\mathbf{d}, t_d, S)$ $\overset{\{\text{by } (5)\}}{\leq}$ $LAG(\mathbf{d}, t_n, S)$ $\overset{\{\text{by } (3)\}}{=}$ $\sum_{\tau_i : \tau_{i,h}^w \in \mathbf{d}} lag(\tau_i, t_n, S) \leq$ $\sum_{\tau_i \in \beta} lag(\tau_i, t_n, S)$ $\overset{\{\text{by Lemma 1}\}}{\leq}$ $\sum_{\tau_i \in \beta} (u_i \cdot x + (u_i + 1) \cdot e_i)$. By Def. 7 and because $|\beta| \leq min(m - 1, n)$, we have $LAG(\mathbf{d}, t_d, S) \leq \sum_{\tau_i \in \beta} (u_i \cdot x + (u_i + 1) \cdot e_i) \leq U \cdot x + E$. $\square$

### 3.2 Lower Bound

In the following lemma, we determine a lower bound on $LAG(\mathbf{d}, t_d, S)$ that is necessary for the response time of $\tau_{l,j}$ to exceed $x + p_l + e_l$.

**Definition 8.** If any thread of any segment of job $\tau_{i,h}$ is enabled at time $t$ but does not execute at $t$, and at least one processor is executing some job other than $\tau_{i,h}$ at $t$, then $\tau_{i,h}$ is *preempted* at $t$ (see Fig. 4).

**Definition 9.** Let $v_{max_i}$ denote the maximum number of threads of any segment of the task that has the $i^{th}$ maximum number of threads of any segment among tasks in $\tau$.
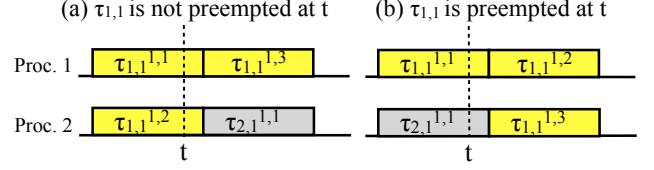


Figure 4: Illustration of a preemption. Job $\tau_{1,1}$ has one segment with three parallel threads, executed on two processors. In inset **(a)**, although $\tau_{1,1}^{1,3}$ is enabled but does not execute at time $t$, $\tau_{1,1}$ is not preempted at $t$ since both processors are executing threads of $\tau_{1,1}$. In inset **(b)**, $\tau_{1,1}$ is preempted by $\tau_{2,1}$ at $t$.

If $\sum_{i=1}^n v_{max_i} \leq m$, then each thread of each segment of each task in $\tau$ can be executed on a processor without being preempted, which implies that each task $\tau_k \in \tau$ has a bounded response time of $e_k^{min} < x + p_k + e_k$. Thus, we consider the other case, where $\sum_{i=1}^n v_{max_i} > m$. Moreover, since we assume that there exists at least one task $\tau_k \in \tau$ with $v_k^{max} \geq 2$ (as discussed in Sec. 2), we have $v_{max_1} \geq 2$. Thus, if $n > m$, then $\sum_{i=1}^m v_{max_i} > m$ holds. Therefore, we have

$$\sum_{i=1}^{min(m,n)} v_{max_i} > m. \tag{8}$$

**Definition 10.** Let

$$Q = \begin{cases} 2 & \text{if } v_{max_1} > m \\ \min\{k \mid k = \sum_{i=1}^k v_{max_i} > m\} & \text{if } v_{max_1} \leq m. \end{cases}$$

$Q$ is used in Lemma 3 below to obtain a lower bound on $LAG(\mathbf{d}, t_d, S)$; the two conditions in the definition of $Q$ arise because of different subcases considered in the proof of Lemma 3. Note that by the above definition and (8), we have

$$2 \leq Q \leq min(m, n) \leq m. \tag{9}$$

**Lemma 3.** *If the response time of $\tau_{l,j}$ exceeds $x + p_l + e_l$, then $LAG(\mathbf{d}, t_d, S) > Q \cdot x - (m - 1) \cdot e_l$.*

*Proof.* Throughout the proof of this lemma, we assume $\sum_{i=1}^n v_{max_i} > m$ and $v_{max_1} \geq 2$ both hold, for reasons discussed above. We prove the contrapositive: we assume that

$$LAG(\mathbf{d}, t_d, S) \leq Q \cdot x - (m - 1) \cdot e_l \tag{10}$$

holds and show that the response time of $\tau_{l,j}$ cannot exceed $x + p_l + e_l$. Let $\eta_l$ be the amount of work $\tau_{l,j}$ performs by time $t_d$ in $S$. Define $y$ as follows.

$$y = \frac{Q}{m} \cdot x + \frac{\eta_l}{m} \tag{11}$$

Let $W$ be the amount of work due to jobs in $\mathbf{d}$ that can compete with $\tau_{l,j}$ after $t_d + y$, including the work due for $\tau_{l,j}$. Let $t_f$ be the completion time of $\tau_{l,j}$. We consider two cases.
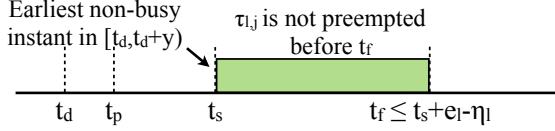
Figure 5: Subcase 2.1



Figure 6: Subcase 2.2

**Case 1.** $[t_d, t_d + y)$ *is a busy interval for* **d**. In this case,

$$W = LAG(\mathbf{d}, t_d, S) - my \overset{\{\text{by (10)}\}}{\leq} Q \cdot x - (m-1) \cdot e_l -$$
$$my \overset{\{\text{by (11)}\}}{=} Q \cdot x - (m-1) \cdot e_l - Q \cdot x - \eta_l = -(m-1) \cdot e_l -$$

$\eta_l < 0$. Because GEDF is work-conserving (i.e., GEDF idles a processor only when there is no enabled job), at least one processor is busy until $\tau_{l,j}$ completes. Thus, the amount of work performed by the system for jobs in **d** during the interval $[t_d + y, t_f)$ is at least $t_f - t_d - y$. Hence, $t_f - t_d - y \leq W < 0$. Therefore, the response time of $\tau_{l,j}$ is $t_f - r_{l,j} = t_f - t_d + p_l <$

$$y + p_l \overset{\{\text{by (11)}\}}{=} \frac{Q}{m} \cdot x + \frac{\eta_l}{m} + p_l \overset{\{\text{by (9)}\}}{\leq} x + e_l + p_l.$$

**Case 2.** $[t_d, t_d + y)$ *is a non-busy interval for* **d**. Let $t_s \geq t_d$ be the earliest non-busy instant in $[t_d, t_d + y)$. Recall (see (4)) that $t_p$ is the completion time of job $\tau_{l,j-1}$. We consider three subcases.

**Subcase 2.1.** $t_p \leq t_s$ *and* $\tau_{l,j}$ *is not preempted within* $[t_s, t_p)$. As illustrated in Fig. 5, in this case, $\tau_{l,j}$ can start execution at $t_s$ because $t_s$ is non-busy. Since $\tau_{l,j}$ is not preempted within $[t_s, t_p)$, $\tau_{l,j}$ completes by $t_s + e_l - \eta_l$. Thus, because $t_s < t_d + y$, $\tau_{l,j}$ finishes by time $t_s + e_l - \eta_l < t_d + y + e_l -$

$$\eta_l \overset{\{\text{by (11)}\}}{=} t_d + \frac{Q}{m} \cdot x + \frac{\eta_l}{m} + e_l - \eta_l \overset{\{\text{by (9)}\}}{\leq} r_{l,j} + p_l + x + e_l.$$

**Subcase 2.2** $t_p \leq t_s$ *and* $\tau_{l,j}$ *is preempted within* $[t_s, t_p)$. If $t_f \leq y + t_d$, then $t_f - r_{l,j} \leq y + t_d - r_{l,j} \overset{\{\text{by (11)}\}}{=} \frac{Q}{m} \cdot x +$
$\frac{\eta_l}{m} + p_l \overset{\{\text{by (9)}\}}{\leq} x + e_l + p_l$. So assume $t_f > y + t_d$. Let $t_1 > t_s$ be the earliest time when $\tau_{l,j}$ is preempted. As shown in Fig. 6, by the definition of $t_s$ and $t_1$, $\tau_{l,j}$ executes throughout $[t_s, t_1)$ without being preempted. Because $\tau_{l,j}$ is preempted at $t_1$, $t_1$ is busy with respect to **d**. Let $t_2$ be the last time $\tau_{l,j}$ resumes execution after being preempted if such a time exists; if such a time does not exist, which implies that $\tau_{l,j}$ is preempted until $t_f$, then let $t_2 = t_f$ (note that by Def. 8, some threads of $\tau_l^j$ can execute while $\tau_l^j$ is preempted). Within $[t_1, t_2)$, $\tau_{l,j}$ could be preempted multiple times. By Def. 8, all such intervals during which $\tau_{l,j}$ is preempted must be busy in order for the preemption to happen. Given that $t_f \leq t_2 + e_l - \eta_l$, if $t_2 \leq y + t_d$, then $t_f \leq y + t_d + e_l - \eta_l$, in which case, because $t_d - r_{l,j} = p_l$, the response time of $\tau_{l,j}$ is $t_f - r_{l,j} \leq$

$$y + p_l + e_l - \eta_l \overset{\{\text{by (11)}\}}{\leq} \frac{Q}{m} \cdot x + p_l + e_l \overset{\{\text{by (9)}\}}{\leq} x + p_l + e_l,$$

as required.

If $t_2 > t_d + y$, then the amount of work due to **d** performed within $[t_d, t_d + y)$ is at least $my - (m-1) \cdot \min(e_l, y)$ because
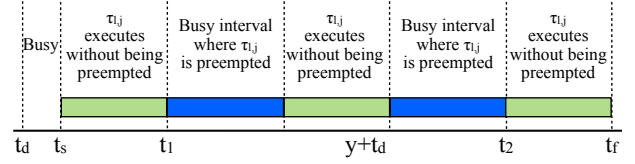
all intervals during which $\tau_{l,j}$ is preempted are busy, and $\tau_{l,j}$ can execute for at most $e_l$ time in $[t_d, y + t_d)$. (Within intervals in $[t_s, t_d + y)$ where $\tau_{l,j}$ is not preempted, at least one processor is occupied by $\tau_{l,j}$.) Thus, the amount of work that can compete with $\tau_{l,j}$ after $t_d + y$ is $W \leq LAG(\mathbf{d}, t_d, S) -$

$$(my - (m-1) \cdot \min(e_l, y)) \overset{\{\text{by (10)}\}}{\leq} Q \cdot x - (m-1) \cdot e_l -$$
$$(my - (m-1) \cdot \min(e_l, y)) \leq Q \cdot x - my \overset{\{\text{by (11)}\}}{=} -\eta_l \leq 0.$$

Since $W$ is defined to be the amount of work due to jobs in **d** that can compete with $\tau_{l,j}$ after $t_d + y$ and $W \leq 0$, the latest completion time of $\tau_{l,j}$ is at $t_d + y + e_l - \eta_l$. Therefore, the response time of $\tau_{l,j}$ is $t_f - r_{l,j} \leq t_d + y + e_l - \eta_l - r_{l,j} =$

$$y + e_l - \eta_l + (t_d - r_{l,j}) = y + e_l - \eta_l + p_l \overset{\{\text{by (11)}\}}{=}$$
$$\frac{Q}{m} \cdot x + \frac{\eta_l}{m} + e_l - \eta_l + p_l \overset{\{\text{by (9)}\}}{\leq} x + e_l + p_l.$$

**Subcase 2.3**: $t_p > t_s$. The earliest time $\tau_{l,j}$ can commence execution is $t_p$, as shown in Fig. 7. Let $S(\tau_{l,j})$ be the time when $\tau_{l,j}$ starts execution for the first time. If $\tau_{l,j}$ is not preempted after $t_p$, then $\tau_{l,j}$ starts execution at $t_p$ and completes no later than $t_p + e_l^{min}$. Thus, we have $t_f - r_{l,j} =$

$$t_p + e_l^{min} - r_{l,j} \overset{\{\text{by (4)}\}}{\leq} t_d - p_l + x + e_l + e_l^{min} - r_{l,j} =$$
$$x + e_l + e_l^{min} \leq x + e_l + p_l \text{ (because } e_l^{min} \leq p_l).$$

The other possibility is that $\tau_{l,j}$ gets preempted after $t_p$. Let $\lambda$ denote the set of tasks including $\tau_l$ that have ready jobs in **d** at any time instant within $[t_s, t_p)$.

We now prove that $|\lambda| \geq Q$ holds. By Def. 8, in order for $\tau_{l,j}$ to be preempted after $t_p$, the number of processors required by tasks in $\lambda$ (note that $\tau_l \in \lambda$) at some time instant after $t_p$ must exceed $m$. Thus, the maximum total number of threads of tasks in $\lambda$ that can execute in parallel at the same time must exceed $m$, which gives

$$\sum_{\tau_i \in \lambda} v_i^{max} > m. \tag{12}$$

Thus, by the definition of $v_{max_k}$, we have $\sum_{k=1}^{|\lambda|} v_{max_k} \geq$
$\sum_{\tau_i \in \lambda} v_i^{max} \overset{\{\text{by (12)}\}}{>} m$. By Def. 10, we consider two cases: $v_{max_1} \leq m$ and $v_{max_1} > m$. If $v_{max_1} \leq m$, then $|\lambda| \geq Q$ holds. On the other hand, if $v_{max_1} > m$, then although $\sum_{k=1}^{|\lambda|} v_{max_k} > m$ may hold when $|\lambda| = 1$, $\lambda$ clearly needs to contain at least two tasks in order for $\tau_{l,j}$ to be preempted (namely, $\tau_l$ and at least one other task). Thus, $|\lambda| \geq Q$ also holds in this case.

Because $|\lambda| \geq Q$, we know that at least $Q$ tasks have ready jobs in **d** at any time instant within $[t_s, t_p)$, which occupy
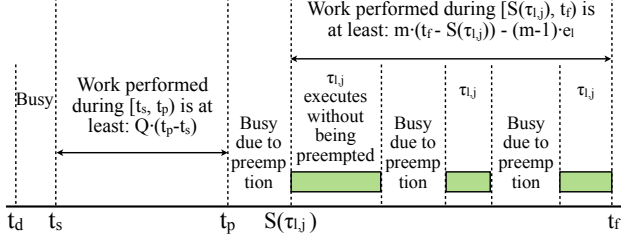
Figure 7: Subcase 2.3

at least $Q$ processors throughout the interval $[t_s, t_p)$. Thus, the amount of work due to **d** performed in $[t_s, t_p)$ is at least $Q \cdot (t_p - t_s)$. We now complete the proof of Subcase 3.2 (and thereby Lemma 3).

By the definitions of $t_s$ and $t_p$, $[t_d, t_s)$ and $[t_p, S(\tau_{l,j}))$ are busy for **d**. As discussed above, the amount of work due to **d** performed in $[t_s, t_p)$ is at least $Q \cdot (t_p - t_s)$. Moreover, the amount of work due to **d** performed in $[S(\tau_{l,j}), t_f)$ is at least $m \cdot (t_f - S(\tau_{l,j})) - (m-1) \cdot e_l$.[3] Thus, we have

$$
\begin{aligned}
LAG(\mathbf{d}, t_d, S) \geq\ & m \cdot (t_s - t_d) + Q \cdot (t_p - t_s) \\
& + m \cdot (S(\tau_{l,j}) - t_p) \\
& + m \cdot (t_f - S(\tau_{l,j})) - (m-1) \cdot e_l.
\end{aligned}
$$

By (10), we therefore have

$$
\begin{aligned}
& Q \cdot x - (m-1) \cdot e_l \\
\geq\ & m \cdot (t_s - t_d) + Q \cdot (t_p - t_s) \\
& + m \cdot (S(\tau_{l,j}) - t_p) \\
& + m \cdot (t_f - S(\tau_{l,j})) - (m-1) \cdot e_l,
\end{aligned}
$$

which gives,

$$
t_f - t_d \ \leq\ \frac{Q}{m} \cdot x + \left(1 - \frac{Q}{m}\right) \cdot (t_p - t_s). \quad (13)
$$

Also, we have $t_p - t_s \leq t_p - t_d \overset{\{\text{by (4)}\}}{\leq} t_d - p_l + x + e_l - t_d = x - p_l + e_l$. Therefore, $t_f - r_{l,j} = t_f - t_d + p_l \overset{\{\text{by (13)}\}}{\leq} \frac{Q}{m} \cdot x + \left(1 - \frac{Q}{m}\right) \cdot (x - p_l + e_l) + p_l \overset{\{\text{by (9)}\}}{\leq} x + p_l + e_l$. $\square$

## 3.3 Determining $x$

Setting the upper bound on $LAG(\mathbf{d}, t_d, S)$ in Lemma 2 to be at most the lower bound in Lemma 3 will ensure that the response time of $\tau_{l,j}$ is at most $x + p_l + e_l$. By solving for the minimum $x$ that satisfies the resulting inequality, we obtain a value of $x$ that is sufficient for ensuring a response time of at most $x + p_l + e_l$. By Lemmas 2 and 3, this inequality is

$$
\begin{aligned}
& U \cdot x + E \\
\leq\ & Q \cdot x - (m-1) \cdot e_l.
\end{aligned}
$$

---

[3] We apply the same reasoning as used in Subcase 2.2. All intervals in $[S(\tau_{l,j}), t_f)$ during which $\tau_{l,j}$ is preempted are busy, and $\tau_{l,j}$ can execute for at most $e_l$ time in $[S(\tau_{l,j}), t_f)$. (Within such intervals, at least one processor is occupied by $\tau_{l,j}$.)

Solving for $x$, we have

$$
x \geq \frac{E + (m-1) \cdot e_l}{Q - U}. \quad (14)
$$

If $x$ equals the right-hand side of (14), then the response time of $\tau_{l,j}$ will not exceed $x + p_l + e_l$. A value for $x$ that is independent of the parameters of $\tau_l$ can be obtained by replacing $(m-1) \cdot e_l$ with $max_l((m-1) \cdot e_l)$ in (14).

**Theorem 1.** *With $x$ as defined above, the response time for any task $\tau_l$ scheduled under GEDF is at most $x + p_l + e_l$, provided $U < Q$, where $U$ and $Q$ are defined in Def. 7 and Def. 10, respectively.*

## 3.4 A Case with No Utilization Loss

The following corollary shows that GEDF results in no utilization loss for scheduling any parallel task system on two processors.

**Corollary 1.** *For two-processor systems, the response time for any task $\tau_l$ scheduled under GEDF is at most $x + p_l + e_l$, where $x = \dfrac{E + (m-1) \cdot e_l}{Q - max_i(u_i)}$ and $max_i(u_i)$ is the maximum task utilization of tasks in $\tau$.*

*Proof.* If the system only contains one task, then clearly this task, denoted $\tau_1$, has bounded response time, which is given by $e_1^{min} \leq x + p_1 + e_1$. If the system contains more than one task, then by Defs. 7 and 10 and $m = 2$, we have $U = max_i(u_i)$ and $Q = 2 = m$. Thus, the utilization constraint in Theorem 1 becomes $max_i(u_i) < Q = m$, which always holds. $\square$

## 3.5 Cases with Utilization Loss

As shown in Theorem 1 and Corollary 1, the utilization constraint $U < Q$ is needed on $m \geq 3$ processors while no utilization constraint is needed on $m = 2$ processors. By Defs. 7 and 10, in the worst case, $U = U_{sum}$ and $Q = 2$. This implies that in some cases even when $m$ is arbitrarily large, $U_{sum} < 2$ is needed in our analysis. Since no utilization loss can be achieved on two processors as shown in Corollary 1, we can schedule any parallel task system with $U_{sum} = 2$ on only two processors (i.e., leave the other $m - 2$ processors idle if $m > 2$). Thus, in the worst case, $U_{sum} \leq 2$ (rather than $U_{sum} < 2$) is needed under our analysis for any parallel task system to have bounded response times for $m \geq 3$ processors. To discern how severe such constraints must fundamentally be, we next show that for any $m \geq 3$, there exists a parallel task system with a total utilization of $2 + \sigma$ that has unbounded response times, where $\sigma$ can be an arbitrarily small value. This proves that utilization constraints are fundamental for parallel task systems scheduled on $m \geq 3$ processors. (Note that this task set also violates our derived utilization constraint.)
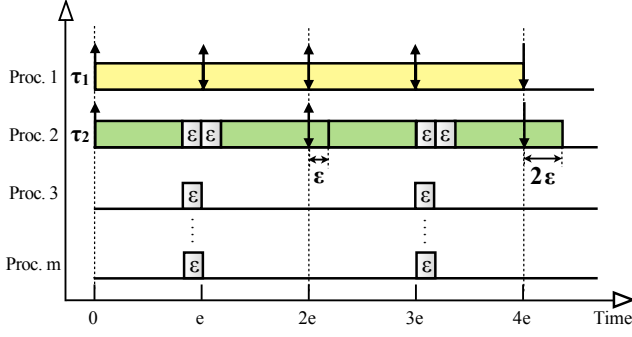
Figure 8: The worst-case parallel task set.



Figure 9: Illustration of the optimization algorithm.

**Worst-case parallel task set.** Consider a parallel task system containing two parallel tasks. Task $\tau_1$ has only one segment that contains one thread with an execution cost of $e$ time units, and $\tau_1$ has a period of $e$ time units. Thus, $\tau_1$ has a utilization of 1.0. Task $\tau_2$ has three segments, where the first segment contains one thread with an execution cost of $e - \varepsilon$ time units, where $\varepsilon$ can be an arbitrarily small value, the second segment contains $m$ parallel threads, each of which has an execution cost of $\varepsilon$ time units, and the third segment contains one thread with an execution cost of $e$ time units. $\tau_2$ has a period of $2e$ and a utilization of $\dfrac{e - \varepsilon + m \cdot \varepsilon + e}{2e} = 1 + \dfrac{(m-1)}{2e} \cdot \varepsilon$.

Thus, this task set has a total utilization of $2 + \dfrac{(m-1)}{2e} \cdot \varepsilon$, or

rather $2 + \sigma$, where $\sigma = \dfrac{(m-1)}{2e} \cdot \varepsilon$ can be arbitrarily small.

Fig. 8 shows the GEDF schedule of this parallel task system on any $m \geq 3$ processors. It is clearly seen that task $\tau_2$'s response time grows unboundedly regardless of $m$.

### 3.6 Potential Extension

For readability and conciseness, we limited attention to obtaining a basic response time bound under GEDF for sporadic parallel task systems. However, our techniques can also be applied to global first-in-first-out as well as many other global scheduling algorithms with minor modifications. Moreover, recent work [9] proposed a slightly different analysis framework, compliant vector analysis, that tightens the response time bound for ordinary sporadic task systems scheduled under GEDF compared to the framework we propose. This new analysis framework can be applied to provide tighter bounds for scheduling sporadic parallel task systems as well.

## 4 Optimization

The utilization loss seen in the utilization constraint $U < Q$ is mainly caused by a small value of $Q$. By Def. 10, $Q$ depends on $v_{max_i}$ $(1 \leq i \leq n)$. If the value of $v_{max_i}$ can be decreased, then the value of $Q$ is increased.

To decrease $v_{max_i}$ $(1 \leq i \leq n)$, we can seek to decrease $v_k^{max}$ (the maximum number of threads in any segment of $\tau_k$) for each task $\tau_k \in \tau$. This can be done by splitting any
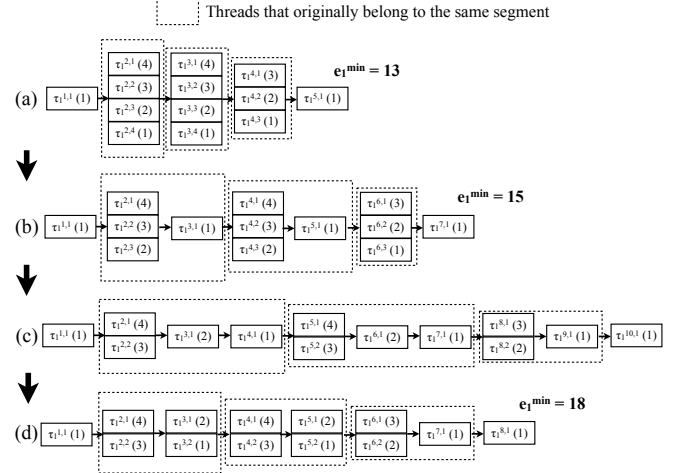
segment of $\tau_k$ with a large number of threads into multiple sequential sub-segments, each of which has a smaller number of threads, thus decreasing $v_k^{max}$. Notice that a critical constraint to enable such splittings is to ensure that $e_k^{min} \leq p_k$ still holds for any task $\tau_k$ after splitting; otherwise, response times may grow unboundedly. Thus, for each task, we need to determine the maximum degree to which its segments can be split. We propose an algorithm to solve this problem. Since the idea behind this algorithm is intuitive, we next present an example to illustrate it. The pseudo-code and a detailed description of the algorithm are given in Appendix A. Applying this algorithm can also reduce the response time bound, as seen in Sec. 5.

**Optimization example.** Since we seek to decrease $v_k^{max}$ for each task $\tau_k$ in any given task system using the same optimization algorithm, we use one example task $\tau_1$ to illustrate the idea. In this example, $m = 4$ and $\tau_1$ originally has five segments, as illustrated in Fig. 9(a). The notation $\tau_1^{i,j}(e)$ in Fig. 9 denotes that thread $\tau_1^{i,j}$ has an execution cost of $e$ time units. $\tau_1$ has a period of 18 time units, thus $p_1 = 18$.

Because we want to decrease $v_1^{max}$, we first try to decrease the number of threads of segments in $\tau_1$ that have the largest number of threads, which are $\tau_1^2$ and $\tau_1^3$. We make two important observations. *(i)* We desire to reduce the number of threads of $\tau_1^2$ and $\tau_1^3$ to no less than the number of threads of $\tau_1^4$, which contains the second largest number of threads. Further reductions do not reduce $v_1^{max}$. *(ii)* We desire to reduce the number of threads of segments in $\tau_1^2$ and $\tau_1^3$ by the same amount. Reducing any such segment's thread count by a greater amount than the others does not reduce $v_k^{max}$. Therefore, according to these observations, we split each of $\tau_1^2$ and $\tau_1^3$ into two sequential sub-segments, one with three threads and the other one with one thread, as shown in Fig. 9(b) (note that in the figure updated segment notations are used after each splitting). After this splitting, we obtain $e_1^{min} = 15 < p_1 = 18$ (we apply the same method discussed

in Sec. 2 to obtain $e_1^{min}$). Thus, this splitting is valid. Now we obtain a task $\tau_1$ in which segments $\tau_1^2$, $\tau_1^4$, and $\tau_1^6$ have the largest number of threads (three threads per segment), while segment $\tau_1^1$ has the second largest number of threads (one thread per segment). Therefore, we again try to reduce the number of threads of $\tau_1^2$, $\tau_1^4$, and $\tau_1^6$ to no less than the number of threads of $\tau_1^1$. This can be achieved by splitting each of these three segments into three sequential segments, each of which contains only one thread. However, after such a splitting, we have $e_1^{min} = 28 > p_1 = 18$. Thus, such a splitting is invalid.

Therefore, our goal now is trying to reduce the number of threads of $\tau_1^2$, $\tau_1^4$, and $\tau_1^6$ to a smallest possible number, which is two threads per segment in this case. As shown in Fig. 9(c), we split each of $\tau_1^2$, $\tau_1^4$, and $\tau_1^6$ into two sequential sub-segments, one with two threads and another one with one thread. Also notice that after this splitting, $\tau_1^3$ and $\tau_1^4$ originally belonged to the same segment, and $\tau_1^6$ and $\tau_1^7$ originally belonged to the same segment. Since combining $\tau_1^3$ and $\tau_1^4$ (as well as $\tau_1^6$ and $\tau_1^7$) into one sub-segment does not increase $v_1^{max}$, we combine them in such a way to decrease $e_1^{min}$, as illustrated in Fig. 9(d). After this splitting, we have $e_1^{min} = 18 = p_1$. Thus, we cannot split segments any further, and we successfully reduce $v_1^{max}$ from 4 to 2.

## 5  Experimental Evaluation

In this section, we describe experiments conducted using randomly-generated parallel task sets to evaluate the applicability of the response time bound in Theorem 1. Moreover, we evaluate whether the optimization algorithm can effectively improve schedulability and reduce the bound.

**Experimental setup.** In our experiments, parallel task sets were generated as follows. The number of segments of each task was uniformly distributed over [1, 30]. The number of threads of each segment was uniformly distributed over [1, $m$/2]. The execution cost of each thread was uniformly distributed over [1$ms$, 100$ms$]. The worst-case execution cost $e_i$ and the best-case execution cost $e_i^{min}$ of each parallel task $\tau_i$ were then calculated using the approach discussed in Sec. 2. Then, for each task $\tau_i$, its period was uniformly distributed over [$e_i^{min}$, $e_i^{min} + e_i$], and its utilization was calculated using $e_i$ and $p_i$. We also varied the system utilization $U_{sum}$ within {0.1, 0.2, ..., $m$ }. For each $U_{sum}$, 1,000 parallel task sets were generated for systems with four, six, and eight processors.[4] Each such parallel task set was generated by creating parallel tasks until total utilization exceeded $U_{sum}$, and by then reducing the last task's utilization so that the total system utilization equalled $U_{sum}$. For each generated system, we first checked schedulability (i.e., the ability to ensure bounded

response times) and the magnitude of response time bounds using Theorem 1. Then, for each such generated system, we applied the optimization algorithm and re-checked schedulability and response time bounds. In doing so, system overheads were ignored (factoring overheads into our analysis is beyond the scope of this paper). In all figures and tables presented in this section, we let "Original" and "Optimization" denote results under the original analysis and results after applying the optimization algorithm.

**Results.** The schedulability results that were obtained on four-, six-, and eight-processor systems are shown in insets (a)–(c) of Fig. 10, respectively. In these figures, the $x$-axis denotes $U_{sum}$ and the $y$-axis denotes the fraction of generated task sets that were successfully schedulable with bounded response times. Each curve plots the fraction of the generated parallel task sets the corresponding approach successfully scheduled, as a function of $U_{sum}$. As seen, our analysis can provide reasonable schedulability. For example, on four processors, all parallel task sets have bounded response times until $U_{sum}$ reaches 3.0 and more than 40% of the task sets still have bounded response times when $U_{sum}$ reaches 3.3. Moreover, the optimization algorithm is able to effectively improve schedulability, especially when the processor count is large. For example, on eight processors, the optimization algorithm can improve schedulability by more than 100% in many cases (e.g., when $U_{sum} = 4.0$). Such improvements tend to increase with increasing processor count. This is because when $m$ becomes larger, it is easier to increase $Q$ by applying the optimization algorithm, which is intuitive according to the definition of $Q$. Note that, when schedulability drops significantly, it does so at an integral values of $U_{sum}$. For example, as seen in Fig. 10(a), when $U_{sum}$ reaches 3.0, schedulability drops from 100% to less than 50% under Original. This is because when $U_{sum}$ reaches 3.0, by Def. 7, $U$ may also equal 3.0 since some parallel tasks very likely have utilization greater than 1.0. Thus, $Q$ has to be 4.0 instead of 3.0 (when the utilization is below 3.0) in order for the utilization constraint $Q > U$ to hold; this obviously makes this constraint much more severe.

Fig. 11 shows the computed response time bounds using Theorem 1 under Original and Optimization. To better illustrate the magnitude of the response time bounds, we plot *relative response time bounds*. A task's relative response time bound is given by the ratio of its response time bound divided by its period. The data in Fig. 11 shows average relative response time bounds obtained by considering all tasks in certain selected task sets. Such task sets were selected by considering values of $U_{sum}$ for which 100% schedulability can be ensured, which guarantees all such task sets valid response time bounds. For example, on four processors, we calculated the average relative response time bound over task sets whose utilizations are within [0.1, 3) (all such task sets are schedulable and thus have valid response time bounds). As seen in

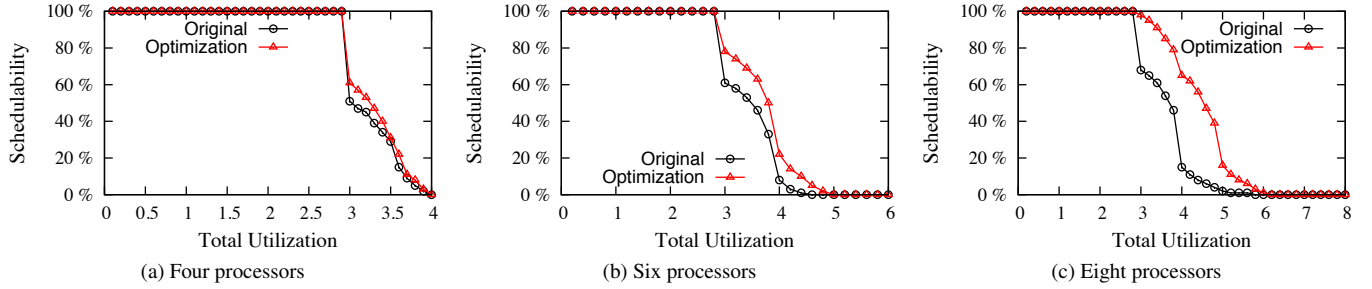(a) Four processors     (b) Six processors     (c) Eight processors
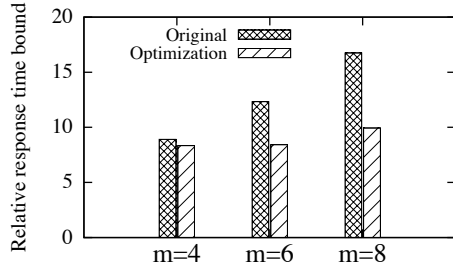
Figure 10: Schedulability results.



Figure 11: Response time bounds.

the figure, our analysis can achieve reasonable response time bounds. For example, on four processors, the average relative response time bound is around nine. The benefit of the optimization algorithm is apparent. For example, on eight processors, we can reduce the average relative response time bound from sixteen to less than ten. This is because applying the optimization algorithm only increases $Q$ and does not change other values in the response time bound expression shown in Theorem 1.

## 6 Conclusion

We have presented schedulability analysis for sporadic parallel task systems under GEDF scheduling. The proposed analysis shows that such systems can be efficiently supported on multiprocessors with bounded response times. In experiments presented herein, our analysis is proved to provide good performance w.r.t. both schedulability and response time bounds. In future work, it would be interesting to investigate more practical parallel task models where data communicate among segments within a parallel task. Moreover, allowing more general parallel execution patterns such as cycles could be a significant improvement.

## References

[1] D. Bailey. An optimal scheduling algorithm for parallel video processing. In *Proc. of the 5th IEEE Intl. Conf. on Multimedia Computing and Systems*, pp. 245-248, 1998.

[2] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *Proc. of the 31st Real-Time Sys. Symp.*, pp. 14-24, 2010.

[3] C. Liu and J. H. Anderson. Supporting soft real-time parallel applications on multicore processors (full version). Available at: http://www.cs.unc.edu/~anderson/papers.

[4] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. Parallel programming in OpenMP. In *Morgan Kaufmann*, 2000.

[5] S. Chen and S. Schlosser. MapReduce meets wider varieties of applications. In *Technical Report IRP-TR-08-05, Intel Labs Pittsburgh*, 2008.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th USENIX Conf. on Symp. on Operating System Design and Implementation*, pp. 137-150, 2004.

[7] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Int'l Real-Time Sys. Symp.*, pp. 330-341, 2005.

[8] E. Dougherty and P. Laplante. Introduction to real-time imaging. In *Wiley-IEEE Press*, 1995.

[9] J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global EDF. In *Proc. of the 22nd Euromicro Conf. in Real-Time Sys.*, pp. 14-23, 2010.

[10] D. Feitelson. Job scheduling in multiprogrammed parallel systems. In *Technical Report RC 19790 (87657)*, 1997.

[11] D. Feitelson and L. Rudolph. Parallel job scheduling: issues and approaches. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 1-18, 1995.

[12] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. In *Journal of ACM*, 23(2), 1976.

[13] Y. Kitamura, A. Smith, H. Takemura, and F. Kishino. Parallel algorithms for real-time colliding face detection. In *Proc. of the 4th IEEE Workshop on Robot and Human Communication*, pp. 211-218, 1995.

[14] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. of the 31st Real-Time Sys. Symp.*, pp. 259-268, 2010.

[15] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proc. of the 28th IEEE Real-TIme Systems*, pp. 413-422, 2007.

[16] C. Liu and J. Anderson. Supporting pipelines in soft real-time multiprocessor systems. In *Proc. of the 21st Euromicro Conf. on Real-Time Sys.*, pp. 269-278, 2009.

[17] C. Liu and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proc. of the 30th Real-Time Sys. Symp.*, pp. 425-436, 2009.

[18] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proc. of the 32nd Real-Time Sys. Symp.*, pp. 217-226, 2011.

[19] C. Volker, V. Hamscher, and R. Yahyapour. Economic scheduling in grid computing. In *Proc. of the Conf. on Scheduling Strategies for Parallel Processing*, pp. 128-152, 2002.

[20] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. In *Technical Report EECS-2009055, UC Berkeley*, 2009.

# Appendix A

We propose Algorithm *Q-Optimization* to increase $Q$ for any given parallel task system $\tau$ by decreasing $v_k^{max}$ for each task $\tau_k \in \tau$, as discussed in Sec. 4. The pseudo-code is given in Fig. 12.

**Algorithm description.** Algorithm *Q-Optimization* seeks to increase the value of $Q$ by decreasing the maximum number of threads in any segment of each task.

We first describe the function *SPLIT* used in the main algorithm. *SPLIT*($\tau_k$,$H$) splits the segments with the maximum number of threads into a number of sequential sub-segments, each with at most $H$ threads (Lines 1-3 in Function *SPLIT*). Threads are assigned to each of these sub-segments in smallest-thread-ID-first order, until either a sub-segment contains $H$ threads or all threads have been assigned. Then Function *COMBINE* (Line 4 in Function SPLIT) seeks to combine any two sub-segments that originally belong to the same segment into one segment if the sum of the number of threads in both sub-segments is no greater than the maximum number of threads of any segment. Finally, Function *SPLIT* calculates $e_k^{min}$ (Line 5 in Function *SPLIT*) using the method we discussed in Sec. 2.

Now we describe Algorithm *Q-Optimization* in detail. According to observations (i) and (ii) given in Sec. 4, the algorithm first executes *SPLIT*($\tau_k, v_k^{max}$), which splits each of the segments in $\tau_k$ that have the maximum number of threads into a sequential number of sub-segments, each with at most $v_k^{max}$ threads. After such a splitting, if $e_k^{min} < p_k$ and $v_k^{max} \neq 1$ (Lines 5-7 in Algorithm *Q-Optimization*), then we set the *further-split-flag* to be true, which implies that there is still room for us to split $\tau_k$ to further reduce $v_k^{max}$.

On the other hand, if $e_k^{min} > p_k$ after such a splitting (Line 8 in Algorithm *Q-Optimization*), then it implies that such a splitting causes $e_i^{min}$ to exceed $\tau_k$'s period (which causes the task to have unbounded response times) and is thus invalid. Since this splitting is invalid, we restore the task structure to the one before the splitting (Lines 10-11 in Algorithm *Q-Optimization*). Thus, we now know that it is impossible to split segments in $\tau_k$ to reduce $v_k^{max}$ to equal $v_k^{secmax}$. However, by splitting, we might still be able to reduce $v_k^{max}$ to some number between $v_k^{secmax}$ and $v_k^{max}$ (realized by Lines 12-19 in Algorithm *Q-Optimization*). Since the maximum value of such a number is given by $C_k$, starting from $C_k$, the algorithm uses the *SPLIT* function and compares the resulting $e_k^{min}$ with $p_k$ to determine whether any such splitting is valid (using the same logic as discussed above).

ALGORITHM: Q-OPTIMIZATION

*further-split-flag*: A BINARY VARIABLE
$v_k^{secmax}$: THE SECOND MAXIMUM NUMBER OF PER-SEGMENT THREADS IN $\tau_k$
$C_k$: $C_k := v_k^{secmax} + 1$

1   **for** each parallel task $\tau_k \in \tau$
2     *further-split-flag:=false*
3     **do**
4        $SPLIT(\tau_k, v_k^{secmax})$
5       **if** $e_k^{min} < p_k$
6         **if** $v_k^{max} \neq 1$
7           **then** *further-split-flag:=true*
8       **else if** $e_k^{min} > p_k$
9         Restore the structure of $\tau_k$ to the one before the last splitting
10       Update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$
11        **while** $C_k < v_k^{max}$
12          $SPLIT(\tau_k, C_k)$
13          **if** $e_k^{min} \leq p_k$
14            **break**
15          **else**
16            Restore the structure of $\tau_k$ to the one before the last splitting
17            Update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$
18            $C_k := C_k + 1$
19     **while** *further-split-flag=true*

FUNCTION: SPLIT($\tau_k$,$H$)

$A_k$: THE SET OF SEGMENTS IN $\tau_k$ THAT HAVE $v_k^{max}$ NUMBER OF THREADS

1   **for** each segment $\tau_k^j \in A_k$
2     Split $\tau_k^j$ into $\left\lceil \frac{v_k^j}{H} \right\rceil$ sequential sub-segments, each with at most $H$ threads, and
       assign threads to each sub-segment in smallest-thread-ID-first order
3     Update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$
4   $COMBINE(\tau_k)$
5   Calculate $e_k^{min}$

FUNCTION: COMBINE($\tau_k$)

$h$: A VARIABLE, INITIALLY $h := 1$

1   **while** $\tau_k^h$ exists
2     **if** $\tau_k^h$ and $\tau_k^{h+1}$ (if any) are sub-segments that originally belong to the same segment, and $v_k^h + v_k^{h+1} \leq v_k^{max}$
3       **then** combine $\tau_k^h$ and $\tau_k^{h+1}$ into one segment
4       Update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C$
5     **else**
6       $h := h + 1$

Figure 12: Algorithm *Q-Optimization*.