Optimal Scheduling of Periodic Gang Tasks

Joël Goossens[1] **and Pascal Richard**[2]

1 **Université Libre de Bruxelles (ULB)**
50 av. F.D. Roosevelt 1050 Brussels, Belgium
joel.goossens@ulb.ac.be

2 **LIAS/Isae-Ensma-Université de Poitiers**
1 av. Clément Ader, BP 40109, 86961 Chasseneuil du Poitou, France
pascal.richard@univ-poitiers.fr

Abstract

The gang scheduling of parallel implicit-deadline periodic task systems upon identical multiprocessor platforms is considered. In this scheduling problem, parallel tasks use several processors simultaneously. We propose two DP-Fair (deadline partitioning) algorithms that schedule all jobs in every interval of time delimited by two subsequent deadlines. These algorithms define a static schedule pattern that is stretched at run-time in every interval of the DP-Fair schedule. The first algorithm is based on linear programming and is the first one to be proved optimal for the considered gang scheduling problem. Furthermore, it runs in polynomial time for a fixed number m of processors and an efficient implementation is fully detailed. The second algorithm is an approximation algorithm based on a fixed-priority rule that is competitive under resource augmentation analysis in order to compute an optimal schedule pattern. Precisely, its speedup factor is bounded by $(2 - 1/m)$. Both algorithms are also evaluated through intensive numerical experiments.

2012 ACM Subject Classification Computer systems organization, Real-time systems, Embedded and cyber-physical systems, Software and its engineering, Process management, Scheduling, Multithreading, Theory of computation, Design and analysis of algorithms, Scheduling algorithms

Keywords and phrases Real-time systems, scheduling, parallel tasks

Digital Object Identifier 10.4230/LITES-v003-i001-a004

Received 2015-05-21 **Accepted** 2016-05-05 **Published** 2016-06-29

1 Introduction

We consider the preemptive scheduling of real-time tasks on identical multiprocessor platforms (see [13]). We deal with *parallel* real-time tasks, the case where each job may be executed on different processors *simultaneously*, i.e., we have *job parallelism*. Nowadays, the design of parallel programs is common thanks to parallel programming paradigms like Message Passing Interface (MPI [26, 28]) or Parallel Virtual Machine (PVM [40, 23]). Even better, sequential programs can be parallelized using standards like OpenMP application programming interface (see [8] for details).

Contributions. We define and prove correct a technique to schedule *optimally* periodic implicit deadline rigid gang tasks (see Definition 1 for details) upon multiprocessors. The algorithm is based on linear programming (LP) and runs in polynomial time for a fixed number m of processors. The second proposed method is a fixed-task priority rule with a performance guarantee of $(2-\frac{1}{m})$ under resource augmentation analysis. These algorithms are compared through numerical experiments.

Organization. Section 2 presents the studied scheduling problem and the related work. Section 3 presents basic results about DP-Fair scheduling of parallel tasks. Section 4 presents the LP-based optimal method and its implementation. Section 5 presents a gang heuristic and its worst-case

performance analysis under resource augmentation. Section 6 presents numerical results comparing both methods on randomly generated task systems. Then, Section 7 concludes the paper and presents some future work.

2 Model, Problem and Related Work

2.1 Parallel Task & Job Models

We deal with jobs and tasks which may be executed on different processors at the very same instant, in which case we say that *job (or task) parallelism* is allowed. Various kinds of parallel task models exist. Goossens et al. [25] adapted parallel computing terminology [7] to *recurrent* (real-time) tasks and jobs as follows.

▶ **Definition 1** ([25] Rigid, Moldable and Malleable Job)**.** A *job* is said to be (i) *rigid* if the number of processors assigned to this job is specified externally to the scheduler a priori, and does not change throughout its execution; (ii) *moldable* if the number of processors assigned to this job is determined by the scheduler, and does not change throughout its execution; (iii) *malleable* if the number of processors assigned to this job can be changed by the scheduler during the job's execution.

A recurrent task is said to be *rigid* if all its jobs are rigid, and the number of processors assigned to the jobs is specified externally to the scheduler; a recurrent task is said to be *moldable* if all its jobs are moldable; malleable if all its jobs are malleable.

Additionally at task level the literature distinguishes between at least three kinds of parallelism:

- *Multithread [12, 32, 39, 38, 1].* Each task is sequence of phases (multiphase in the following), each phase is composed of several threads, each thread requires a single processor for execution and threads *can* be scheduled simultaneously [38]. A particular case is the *Fork-Join* (see e.g. [36]) task model where the phases are an alternate sequence of sequential and parallel segments; task begins as a single master thread that executes sequentially until it encounters the first fork construct, where it splits into multiple parallel threads which synchronize/join on their terminaison and so on.
- *Dag task model [2, 6, 34].* The model generalizes the fork-join model, each task is represented as a directed acyclic graph, which is a set of precedence-constrained sequential jobs. Any group of jobs that are not constrained *may* execute in parallel.
- *Gang [11, 4, 25, 30].* Each task corresponds to $e \times k$ rectangle where e is the execution time requirement and k the number of required processors with the restriction: the k processors *must* execute task in *unison* (i.e., at the exact same time). This model is very representative to real world parallel applications where, at the submission time, users or scheduler select the number of processors for the task [14, 16] and consequently the number of generated threads corresponds to the number of used processors like MPI [41] and OpenMP [9] tools do. The threads communicate each other, they must be ready to communicate at the same time which imposes the synchronous threads execution.

2.2 Our Task/Job Model and Scheduling Problem

At job-level we consider the preemptive scheduling of parallel jobs on a multiprocessor platform upon m identical processors. We will focus on the problem of scheduling a set of rigid gang parallel jobs, each job $J_j \stackrel{\text{def}}{=} (r_j, v_j, e_j, d_j)$ is characterized by a release time r_j, a required number of processors v_j, an execution requirement e_j and an absolute deadline d_j. The job J_j must execute

for e_j time units over the interval $[r_j, d_j)$ on v_j processors. We consider the scheduling of *rigid* jobs since v_j is fixed externally to the scheduler.

Our main scheduling problem concerns *periodic* (and sporadic —see discussion Section 5.3) preemptive *hard* real-time systems. Let $\tau \stackrel{\text{def}}{=} \{\tau_1, \ldots, \tau_n\}$ denote a set of n periodic *implicit-deadline* rigid gang parallel tasks. Each task $\tau_i \stackrel{\text{def}}{=} (v_i, C_i, T_i)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is $((k-1) \cdot T_i, v_i, C_i, k \cdot T_i)$. In the following $u_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$ denotes the utilization factor of task τ_i. This utilization is not related to the number of required processors (v_i) this is an *horizontal* notion. The execution requirement of a job of τ_i corresponds to a $C_i \times v_i$ *rectangle*.

This Research. We study the scheduling of preemptive periodic implicit-deadline rigid *gang* real-time tasks. We address the feasibility and the schedulability questions by designing an optimal scheduler. The proposed approach exploits the deadline partitioning of the schedule. Precisely, in every slice delimited by two subsequent deadlines in the schedule, a static schedule pattern is stretched according to the slice length. Two algorithms are proposed to define such a pattern. The first algorithm is based on linear programming and is the first one to be proved optimal for the considered gang scheduling problem. Furthermore, it runs in polynomial time for a fixed number m of processors and an efficient implementation is fully detailed. The second algorithm is an approximation algorithm based on a fixed-priority rule that is competitive under resource augmentation analysis for computing a static schedule pattern. Precisely, its speedup factor is bounded by $(2 - 1/m)$. Both algorithms are also evaluated through intensive numerical experiments.

For the sake of simplicity we consider periodic tasks and we assume that C_i is the *exact* duration of the task τ_i. Consequently, we consider an offline scheduling problem. The discussion of Section 5.3 extends the scope of our result to the scheduling of *sporadic* tasks with C_i as the *worst-case* execution time.

2.3 Related Work

Optimal solutions. To the best of our knowledge there is a single work which provides an *optimal* solution for the scheduling of recurrent *hard* real-time *parallel* computing: the work of Collette et al. [11] which considers sporadic implicit-deadline *malleable* gang scheduling. The authors provide an optimal scheduler and an exact feasibility/schedulability test. The work we report in this document provides an optimal scheduler and an exact feasibility/schedulability test as well, except we consider a more realistic parallel task model since our task are *rigid* jobs—the degree of parallelism is specified at design time and does not change at run-time—while the authors of [11] consider *malleable* jobs.

Non optimal solutions. The other contributions to the scheduling of recurrent *hard* real-time *parallel* computing consider *non* optimal schedulers and schedulability tests (sufficient or exact). [30] considers the EDF (Earliest Deadline First [35]) scheduler for rigid gang sporadic tasks and proposes a sufficient schedulability test. [25, 4] consider FTP (Fixed Task Priority, such as Rate Monotonic [35]) periodic gang scheduling and provide an exact/sufficient schedulability tests. [12] considers FTP and periodic multithread tasks and proposes an exact schedulability test. [32] consider Deadline Monotonic scheduling of fork-join tasks, they provide a competitive analysis for that suboptimal scheduler. [39] considers EDF and FTP scheduling for multiphase multithread periodic tasks and provides a task decomposition technique and a competitive analysis. [38] considers optimal schedulers for sequential tasks (e.g., DP-Fair described Section 3.4), implicit

deadline multiphase multithread recurrent tasks, for which the authors propose a decomposition technique and a competitive analysis. More recent works [6, 34] consider DAG tasks model (which generalizes fork-join model). The authors study the competitiveness of global EDF and global RM.

3 Basic Results

3.1 Hardness for arbitrary number of processors

Assuming that the number of processors is an input in the problem model, it is easy to show that the preemptive Gang scheduling problem is equivalent to the Bin Packing problem. Such a result was observed (without proof) for the non preemptive scheduling problem of parallel (Gang) tasks having unit processing times [5]. Hereafter, we provide a basic proof sketch to clearly exhibit the reducibility among both problems.

▶ **Theorem 2.** *Preemptive Gang scheduling is* NP*-hard in the strong sense for problem instances with an arbitrary number of processors.*

Proof. (Sketch) We transform from Bin Packing [22]: Finite set A of items, a rational size $s(a) \in \mathbb{Q}^+$ for each $a \in A$, a positive integer bin capacity B, and a positive integer K; is there a partition A into disjoint $A_1, A_2, \ldots, A_K$ such that the sum of the sizes of the items in each A_i is no more than B?

Without loss of generality we assume that all $s(a)$ have a common denominator and B is scaled accordingly, we define a Gang scheduling instance as follow:

- $m = B$ processors,
- each item $a \in A$ is model as a unit-length task of period K, $\tau_a = (v_a, e_a, K)$, such that $e_a = 1$ and $v_a = s(a)$.

The Bin Packing decision problem is equivalent to determine if there is a feasible schedule of period K? (i.e., a schedule with no more than m processors are simultaneously used at any time). ◀

The previous transformation shows that preemptive Gang Scheduling with an arbitrary number of processors contains the bin packing problem as a particular case (i.e., with all task execution times equal to 1). It is also known that the Bin Packing problems can be solved in polynomial time for any fixed B by exhaustive search [22]. Such a property will be also exploited for defining our optimal polynomial time algorithm for scheduling periodic Gang real-time tasks.

3.2 Maximum Rectangle Utilization Bound

In our task model (see Section 2.2), the task utilization $u_i \stackrel{\text{def}}{=} C_i/T_i$ represents an *horizontal* utilization. In this section we introduce the notion of total *rectangle* task set utilization which is by definition $\sum_{i=1}^{n} \frac{C_i \times v_i}{T_i}$.

The maximum rectangle utilization bound U_b of a scheduler guarantees that every system of tasks whose total utilization is smaller than or equal to U_b will be correctly scheduled. Beyond this utilization limit, and if the bound is said to be tight, then there exist systems of tasks which are not schedulable.

First notice that, since our scheduling problem of parallel tasks is a generalization of the popular scheduling problem of periodic *sequential* tasks, we have to report a negative result: the maximum rectangle utilization bound is $1/m$.

▶ **Theorem 3.** *The maximum rectangle utilization bound for the scheduling of periodic Gang tasks is* $1/m$.

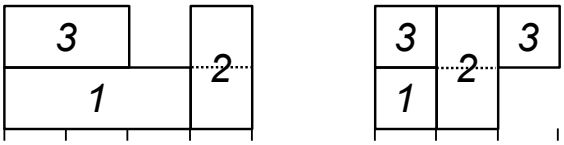


Figure 1 Non-predictability of gang FJP schedulers. Job 1 has the highest priority, job 3 has the lowest one and job 2 in the middle, and they all arrive at time 0.

Proof. The result will be established using a simple parallel task set with two tasks[1]: $\tau_1(1,1,1)$ and $\tau_2(m,\epsilon,1)$, where ϵ is an arbitrary small positive infinitesimal number. This system is trivially infeasible and the total task rectangle utilization is $1/m$. ◀

3.3 Scheduling Anomalies

We have to report a second negative result concerning our scheduling problem: FJP (Fixed Job Priority, such as Earliest Deadline First [35]) and consequently FTP gang scheduling are not predictable [25] [2]. Here is an example task system, on 2 processors and three jobs[3] (see Figure 1):

$$J_1 = (0,1,3,3), J_2 = (0,2,1,4), J_3 = (0,1,2,2) \ .$$

Using the priority assignment $J_1 > J_2 > J_3$, Gang FJP schedules the set of jobs (J_3 completes at time-instant 2). Unfortunately, if the actual duration of J_1 is 1, J_2 will preempt J_3 at time $t = 1$ and J_3 will complete *later*, at time-instant 3. Then, J_3 does not miss its deadline in the "worst case" scenario, but misses it if J_1 uses less than its worst case execution time C_1. Thus, reducing an execution time can delay the completion of another job.

3.4 DP-Fair Scheduling for Sequential Tasks

While this research concerns *parallel* tasks we will introduce a scheduling technique defined for *sequential* tasks (in the next section —Section 3.5— we will show how to apply the very same technique to gang tasks). Consequently, we assume in this section the scheduling of n *sequential* and implicit-deadline tasks upon m identical processors. Each task $\tau_i \stackrel{\text{def}}{=} (C_i, T_i)$ is characterized by a worst-case execution duration C_i and a period T_i. Seminal optimal multiprocessor scheduling techniques were based on the notion of *proportionate fairness*, it is the case for instance of the PF (Proportionate Fairness) scheduler [3]. This type of algorithm assumes that time is discrete.

The quantum-by-quantum construction of the scheduling *is not necessary* in order to define an optimal algorithm [42, 20, 24]. DP (Deadline Partitioning) scheduling techniques do not decompose the tasks into single-time unit (sub)-tasks. The construction of the scheduling is done over time intervals delimited by two *consecutive deadlines* called blocks. In each block, every task receive a workload that is proportional to its utilization so that the fairness property is satisfied at each deadline in the schedule.

Let L_j be the length of the block j delimited by two subsequent task deadlines, every implicit deadline periodic task τ_i receives an amount of processor equal to $L_j \times u_i$. Consequently, a task τ_i has received an execution times equal to $u_i \times T_i = C_i$ for each of its deadlines.

---

[1] (v_i, C_i, T_i)

[2] a scheduling algorithm is predictable if reducing an execution requirement cannot increase the completion of tasks.

[3] (r_j, v_j, e_j, d_j)

The previous algorithms assume that time is by its nature discrete: the times at which the scheduler can be activated are integers (in other words correspond to the clock ticks of the real-time operating system). Discrete time is by its nature a source of complexity in multiprocessor scheduling and for this reason, algorithms which exploit the continuous nature of time have been defined.

We will now detail the simplest algorithm in this category: the DP-Wrap algorithm.

The DP-Wrap algorithm is a very simple deadline fair algorithm which is optimal for tasks with implicit deadlines [20]. Contrarily to the previous algorithms, DP-Wrap considers that the time is continuous. The scheduling is broken down into blocks delimited by deadlines/periods. The distribution of the tasks into each interval is equal to the length of the interval multiplied by the utilization of the task, i.e., $u_i \stackrel{\text{def}}{=} C_i/T_i$. Thus, in the interval $[s_j, s_{j+1})$, each task τ_i is given $C_i(j) \stackrel{\text{def}}{=} u_i \times (s_{j+1} - s_j)$. Consequently, at each deadline, the tasks have received an execution time equals to $u_i \times T_i = C_i$. The scheduling (distribution) in each block is done by McNaughton's algorithm, which has been proposed in 1959 [37].

In the next section, we show how to reuse the deadline partitioning technique in order to define an optimal static gang scheduling algorithm.

3.5 DP-Fair and Gang Scheduling

The next result (Theorem 4) shows that concerning our parallel scheduling problem (gang scheduling defined Section 2.2) we can, without loss of generality, consider DP-Fair scheduling, i.e., schedule where the same pattern is replicated (and stretched) in each interval delimited by deadline/period. In Section 4 we will define a technique to build such pattern optimally.

▶ **Theorem 4.** *Every feasible parallel task set is schedulable with a DP-Fair schedule.*

Proof. Assume we have a feasible schedule, then we show how to define a feasible DP-Fair schedule. Since parallel tasks are strictly periodic and are simultaneously released, the whole schedule is periodic and has a period equal to the hyperperiod. Within the hyperperiod H, every task is executed for $\frac{H}{T_i}C_i = Hu_i$. To define a DP-Fair schedule:

- Schedule pattern: stretch the complete schedule within unit time slots.
- Stretch the pattern accordingly in every block of the schedule.

The corresponding schedule is DP-Fair. At every block boundary t, every task $\tau_i, 1 \leq i \leq n$ receives exactly $t \cdot u_i$. Hence, for every time instant t corresponding to a deadline of task τ_i (i.e., $t = kT_i$) the task τ_i receives exactly $kT_iu_i = kC_i$ and thus has been executed to completion by its deadline. ◀

4 Optimal Pattern Definition

4.1 Research Method

Firstly, we will revisit a non real-time scheduling problem and its solution (the work of Błazewicz et al. [5]) where the main goal is to *minimize* the schedule length of non recurrent rigid gang jobs. We will show how that technique can be *optimally* adapted to our hard real-time scheduling problem.

4.2 The work of Błazewicz et al. revisited

Principles. In [5] the authors consider the scheduling of n rigid gang jobs, each job J_i is characterized by the couple (u_i, v_i), i.e., a duration u_i and a required number of processors v_j, all

these jobs are released simultaneously at time origin. Upon an identical multiprocessor platform the scheduling problem is to find a schedule which minimizes the schedule length, the first instant where the jobs are completed or equivalently to minimize the makespan.

The authors present a polynomial time algorithm for a fixed number of processors m, based on linear programming, for computing an optimal schedule in the general case. Particular cases are also considered but not useful in our framework. Notice that the problem is NP-hard for arbitrary number of processors (i.e., m is an input of the problem) [15].

The method decomposes the schedule as a sequence of slices. Remember, a feasible allocation of jobs is the one that uses no more than m processors. Each slice σ_i is characterized by the set S_i of feasible jobs and the duration x_i of their execution. The algorithm computes the length for every feasible allocation of jobs. As a result, the slices having a positive length are sequenced in arbitrary order. Moreover the method minimizes the $\sum_{i=1}^{M} x_i$, i.e., the makespan to define an optimal schedule.

▶ **Definition 5** (Feasible allocation)**.** A *feasible allocation* of jobs is a subset s of job indexes that can be processed *simultaneously* on the platform: $\sum_{i \in s} v_i \leq m$. By definition we have a *finite* number of different feasible allocation sets. In the following M is the number of different feasible allocation sets. Thus the set of all feasible allocation subsets is denoted $S = \{S_1, \ldots, S_M\}$.

▶ **Example 6.** For instance, consider three jobs J_1, J_2, J_3 with $v_1 = 1, v_2 = 2, v_3 = 1, u_1 = 3, u_2 = 1$ and $u_3 = 2$. For $m = 2$ the feasible allocations are $S_1 = \{1\}, S_2 = \{2\}, S_3 = \{3\}$ (jobs can be executed alone); $S_4 = \{1, 3\}$ (J_1 and J_3 can be executed in parallel). Remark that J_2 cannot be executed with J_1 nor J_2.

Notice that S has a cardinality of $M \stackrel{\text{def}}{=} |S| \leq \sum_{k=1}^{m} \binom{n}{k} \leq (n)^m$. It is important to notice that the number of subsets M (i.e., number of variables in the linear program) is in $O(n^m)$ that is *polynomial* for fixed values of m.

Let Q_j be the set of those subset indexes which contain job J_j. Let x_i be the processing time of the subset S_i in the schedule and used to define variables in the linear program. The linear program computes the schedule as a set of slices of length x_i. Every value x_i such that $x_i > 0$ defines a slice in the schedule in which the jobs of S_i are executed. Slices are executed in an arbitrary order without any inserted delays between them. Hence, the computed makespan is $\sum_{i=1}^{M} x_i$.

The objective function is to minimize the makespan: $\sum_{i=1}^{M} x_i$; and the linear program must enforce that all jobs are completed. The corresponding constraint is: $\sum_{i \in Q_j} x_i = u_j, j = 1 \ldots n$. Hence, the schedule with the smallest length (i.e., makespan) is defined as follows:

Algorithm 1: Optimal Schedule Pattern construction by Linear Programming.

$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{M} x_i \\ \text{subject to} & \sum_{i \in Q_j} x_i = u_j \qquad j = 1 \ldots n \end{array}$$

A solution for Example 6 is $x_1 = 1$ (duration of the execution of J_1 only), $x_2 = 1$ (duration of the execution of J_2 only), $x_3 = 0$ and $x_4 = 2$ (duration of the joint execution of J_1 and J_3) which corresponds to the schedule of Figure 2.

In the previous linear program, there are: M variables and n constraints. It can be solved in polynomial time using for instance Khachiyan's algorithm [31]. This is pretty much what Błazewicz et al. [5] did to solve optimally and polynomially their scheduling problem. We first show how

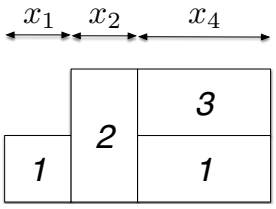


Figure 2 A solution for Example 6.

that technique can be used to solve our hard real-time scheduling problem (Section 4.3). Then, we will show how to speedup the resolution time by defining an efficient problem construction before calling the LP solver (Section 4.4).

4.3 Minimizing the Makespan vs. Meeting Hard Real-time Deadline of Recurrent Tasks

The next property establishes an equivalence between the Błazewicz et al. optimal solution and an optimal scheduler for our real-time scheduling problem thanks to the DP-Fair scheduling theory [33].

▶ **Theorem 7.** *A periodic implicit deadline rigid gang scheduling system* $(v_i, C_i, T_i)_{i=1\cdots n}$ *is feasible* $\Leftrightarrow$ *the set of synchronous jobs* $(u_i, v_i)_{i=1\cdots n}$ *has a minimum makespan not larger than the unity.*

Proof. $\Leftarrow$ Assuming the makespan of the set of synchronous jobs $(u_i, v_i)_{i=1\cdots n}$ is not larger than the unity we have a schedule *pattern* which executes each task for a duration of u_i in a unit-length interval. Using deadline partitioning fairness theory (DP-Fairness, see [33, 20, 10, 18, 21, 19]) we can schedule our original *periodic* task set. The main idea of that kind of schedule is the deadline partitioning of the timeline: the time is divided in *slices* bounded by two successive job deadlines [33, 20]. All tasks are assigned a local execution time which is the length of the current slice times the task utilization u_i. As basically DP-Wrap [20] we use an identical pattern (the solution of the LP) in each time slice, i.e., the unitary pattern is *stretched* according to the length of the slice. Thanks to the DP-Fair theory, we know that for each time interval $[kT_i, (k+1)T_i)$ the task τ_i executes during $u_i T_i = C_i$ time units on v_i processors simultaneously. Consequently, the periodic gang task is feasibly scheduled.

$\Rightarrow$ We will show the contra-positive, i.e., assuming that *all* schedules of the synchronous job set have a length larger than one. Since DP-Fair is optimal for periodic implicit-deadline systems and since the technique of Błazewicz et al. determines the minimal makespan we can conclude that it is *necessary* for the schedulability of periodic implicit-deadline tasks that in each slice the active task τ_i executes for u_i times the slice length. Hence, slices where all tasks are active (like the first one in the synchronous case) cannot execute u_i times the slice length since the solution of the LP is larger than one, consequently (because DP-Fair is optimal), the periodic system is not schedulable on m unit-speed processors. ◀

4.4 LP implementation issues

Efficient generation of the set of all feasible allocations S (Definition 5) is the main combinatorial problem in the linear program construction (Algorithm 1) in order to setup the linear program that will compute the optimal schedule pattern. Even if the size of the problem is polynomially bounded for fixed values of m, the brute force definition of the set of all feasible allocations S

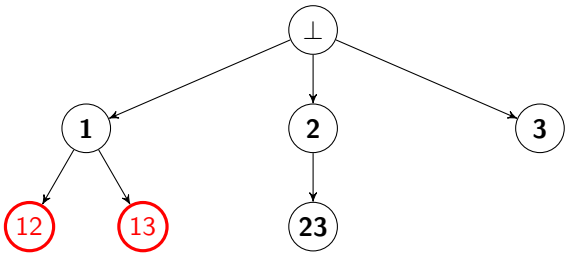


Figure 3 Search tree with nodes defined by task indexes in subsets; black nodes are feasible subsets whereas red nodes are infeasible subsets.

requires a huge amount of time for $n > 20$. This stage is the bottleneck of the approach since the linear program is solved quickly as it will be shown in the experimental section.

A simple way to implement a Brute Force generation of all subsets of tasks is to represent every feasible allocation by an integer in which the binary encoding represents the tasks selected in a subset. There are 2^n subsets of a set with n elements, exactly as there are 2^n different ways to write numbers with n bits. Let s denote such an integer, if i^{th} binary digit is 1 in s, then it indicates that task τ_i is in the feasible allocation represented by s. With such a binary encoding, the brute force enumeration of all feasible allocations of n tasks simply consists in counting from 1 to $2^n - 1$ and defining subsets from the binary encoding. Subsets corresponding to feasible allocations are those that do not use more than m processors.

Using the same binary encoding principles for feasible allocations, we define an efficient generation using Depth First Search with lexicographic ordering of enumerated subsets. Tasks are sorted in non increasing order of v_i. As a consequence, vertices corresponding to infeasible allocations are efficiently pruned. A branch in the search tree is pruned if the current vertex in the tree corresponds to subset of tasks that use more than m processors. The search tree is illustrated in Figure 3 for three tasks: $\{\tau_1, \tau_2, \tau_3\}$ which respectively use 3, 2 and 1 processors. The considered platform has 3 processors. Nodes define indexes of tasks in a subset. Black nodes are feasible subsets whereas red ones are infeasible (i.e., requires more than 3 processors). During the search, the node $\{123\}$ is not defined since $\{12\}$ is already infeasible (i.e., the branch is pruned or fathomed). Using a Depth First Search, the search tree simply consists in the list of unexplored subsets (i.e., encoded as one integer each) and its size is upper-bounded by $\mathcal{O}(n^2)$ subset entries. Our Matlab implementation of this algorithm, denoted DFSLex hereafter, is limited to 64 tasks (i.e., due to Matlab 64-bit integers). Brute Force and DFSLex methods will be compared in the section dedicated to numerical experiments. The performance of the LP for optimally solving Gang scheduling problems will be also presented in Section 6.

In the next section, we propose an heuristic that avoids the previous combinatorial problem. This heuristic has a performance guarantee in terms of resource augmentation (i.e., speedup factor).

5 Gang heuristic

This section presents a scheduling heuristic for defining the schedule pattern. As for the optimal solution presented in the previous section, we consider a DP-Fair schedule in which the pattern will be stretched in every block delimited by two subsequent deadlines of tasks.

The heuristic algorithm is a fixed-task priority scheduling rule that runs in $\mathcal{O}(n \log n)$ for an arbitrary number of processors. We provide a resource augmentation analysis to compare its worst-case performance against the optimal method.

5.1 Fixed-Task Priority Scheduling gang-h

[29] presents an heuristic for minimizing the makespan of preemptive parallel jobs. We shall reuse the basis of the algorithm for defining a fixed-task priority algorithm, denoted gang-h hereafter. As in the previous section, the algorithm is used to define the pattern of tasks to schedule in every time slot (i.e., block) of the DP-Fair schedule. As in the optimal method, the pattern is defined as a unit-length schedule in which utilization factors of tasks play the role of execution requirements. For each block in the schedule, the rule is simultaneously used to:

- allocate the portion of each task to processors.
- sequence tasks within the block.

gang-h is a fixed-task priority scheduling rule that works as follows [29]:

1. Priorities are assigned in non-increasing order of the number of requested processors (i.e., non-increasing order of v_i); ties are broken arbitrarily.
2. Scheduling decisions are taken every time a job is released or completed. At such event, all tasks are preempted and the priority list is used to allocate ready tasks to the processors greedily as feasibly while the current job can be scheduled on the remaining available processors.

The complete algorithm is described in Algorithm 2. This algorithm considers synchronous jobs and will be used to define the pattern of jobs to be scheduled in every block of the schedule (as in the optimal algorithm).

5.2 Optimality under resource augmentation

The scheduling rule gang-h is obviously not optimal for minimizing the pattern makespan in our DP-Fair approach. Nevertheless, we next prove that it is as powerful as an optimal algorithm if it is allowed to use a faster processor than the optimal algorithm executed upon a unit-speed processor. Such a performance guarantee quantifies the price being paid for using gang-h. Precisely, we establish that the speedup factor is bounded by $2 - 1/m$. We first recall the speedup factor metric.

▶ **Definition 8.** (Speedup factor) A scheduling algorithm A has a speedup factor f, $f \geq 1$, if it can schedule any task set that can be scheduled on a given platform by an optimal algorithm, provided that A is able to schedule the same task set upon a platform in which each processor is f times as fast as the processors available to the optimal algorithm.

For proving the resource augmentation performance guarantee for our real-time scheduling problem, we reuse the following results that establish the competitive ratio for the makespan minimization problem.

▶ **Lemma 9.** *(Theorem 3.1 in [29]) Let w_L be the makespan computed by gang-h and w_0 be the optimal makespan, then:*

$$w_L \leq (2 - \frac{1}{m})w_0 \tag{1}$$

The approximation bound of $(2 - 1/m)$ can be easily proved to be tight by using the same task set that has been proposed in [27]: $m^2 - m + 1$ jobs. This job set is defined by $(m^2 - m)$ unit-length jobs and one job of length m. Every task uses exactly one processor. Since ties are broken arbitrarily, assume that gang-h assigns the lowest priority to the job of length m. Consequently, gang-h defines a schedule that first allocates all unit-length jobs to the m processors and lastly the last job. This schedule is of length $m - 1 + m = 2m - 1$, whereas the optimal makespan is m by allocating the long job to a dedicated processor and by scheduling the unit-length jobs on $m - 1$ remaining processors.

Algorithm 2: gang-h (synchronous jobs).

input :
n jobs $J_j(u_j, v_j), 1 \leq j \leq n$;
m: number of processors;
output:
Slice lengths $S(s), s = 1, 2, \ldots$;
Scheduled jobs j in slice s: Sched(s,j) $\in \{0, 1\}, 1 \leq j \leq n$;

```
List=Sort(J_1, ..., J_n) ;        /* Jobs are sorted in non increasing order of v_j */
s = 0 ;                                                    /* Number of slices */
r_j := u_j   ∀j = 1 ··· n ;                  /* Job remaining execution times r_j */
while ∃j, r_j > 0, 1 ≤ j ≤ n do
    /* create a new slice                                                      */
    s = s + 1 ;                                            /* Number of slices */
    K = m ;                                          /* Remaining processors */
    ℓ = ∞ ;                                        /* Slice length upper bound */
    Sched(s,j)=0   1 ≤ j ≤ n ;                                  /* Empty Slice */
    foreach j ∈ List do
        /* For each job j in priority List                                     */
        if v_j ≤ K then
            /* the job j is schedulable in current slice                       */
            Sched(s,j)=1;
            ℓ = min(ℓ, r_j);                              /* update slice length */
            K = K − v_j ;                                /* remaining processors */
        end
    end
    S(s)=l;                                                /* Slice length */
    for j = 1 ... n do
        /* Update remaining execution times                                    */
        if Sched(s, j) then
            r_j = r_j − ℓ;
        end
    end
end
```

The following lemma states that gang-h always produces the same distribution of tasks among identical processors whatever the platform speed.

▶ **Lemma 10.** *Let P be the schedule pattern computed by gang-h on m unit-speed processors and w be its makespan, then gang-h produces a pattern P' of length w/s if a platform is of speed $s > 0$.*

Proof. All jobs in a pattern are simultaneously available at the beginning of the schedule. Thus, scheduling decisions are only taken by gang-h when a job is completed. Using m speed-s processors will stretch execution times to the value C_i/s for every task τ_i. Hence, the pattern is of length w/s. ◀

▶ **Theorem 11.** *gang-h has a speedup factor not exceeding* $(2 - \frac{1}{m})$.

Proof. Consider a task set τ on a platform Π defined by m unit-speed processors. Assume that τ is feasible upon Π and let w_0 be the length of the pattern computed by an optimal algorithm, then

by Lemma 9, gang-h produces a pattern of length not exceeding $(2 - \frac{1}{m})w_0$ on Π. Now consider a platform Π' where each processor is of speed $s = 2 - \frac{1}{m}$. The task set corresponding to τ, denoted τ', has an execution requirement defined by C_i/s for every tasks. By Lemma 10, the length of the schedule defined by gang-h upon Π' is not exceeding $(2 - \frac{1}{m})w_0/s = w_0$. Hence, the pattern defined by gang-h is feasible. ◀

Thus, a processor speedup of $2 - 1/m$ is an upper bound on the price being paid for using the presented gang heuristic for defining the schedule pattern to be stretched in every block of the DP-Fair schedule.

5.3 Extending the scope of the results

For the sake of simplicity we considered implicit-deadline periodic tasks and we assumed that C_i is the *exact* duration of the task τ_i. In this section we discuss straightforward extensions (constrained-deadlines, asynchronous systems and C_i as the worst-case execution requirement) and possible extensions which are left as future work (sporadic and arbitrary deadlines).

Constrained-deadlines. We considered in this work *implicit*-deadline rigid gang parallel tasks. Constrained-deadline tasks are characterized by an additional parameters $D_i \leq T_i$ the relative deadline. Each constrained-deadline task $\tau_i \stackrel{\text{def}}{=} (v_i, C_i, T_i, D_i)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is $((k-1) \cdot T_i, v_i, C_i, (k-1) \cdot T_i + D_i)$. DP-Fair techniques can be obviously extended for constrained-deadlines by considering the task *density* $\delta_i \stackrel{\text{def}}{=} C_i/D_i$ instead of task utilization. DP-Fair is not longer optimal for constrained-deadline and sequential tasks, but if the makespan of gang jobs $\{(\delta_i, v_i) \mid i = 1, \ldots, n\}$ is not larger than the unity our method schedule feasibly constrained-deadline gang tasks. Funk et al. extended for instance DP-Wrap for constrained- deadlines [20].

Asynchronous periodic tasks. Asynchronous periodic tasks are characterized by an additional parameters O_i the release time of the first job of τ_i. Each task $\tau_i \stackrel{\text{def}}{=} (v_i, C_i, T_i, O_i)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is $(O_i + (k-1) \cdot T_i, v_i, C_i, O_i + k \cdot T_i)$. Once again the technique can be used for that asynchronous system: we define the pattern for the synchronous job scenario and we apply the deadline partitioning method and stretching accordingly that pattern.

Early completion. We assumed in this work that C_i is the *exact* duration of the task τ_i. Meanwhile, from applicative perspective this is incorrect, at design time we determine the worst-case execution time (C_i is the WCET) for each task. At run-time the actual duration of any job of τ_i can be smaller than C_i. Again DP-Fair techniques can be obviously extended for that case, a task might not use all the capacity reserved for it, but because of scheduling anomalies reported Section 3.3 we have to respect the stretched pattern, in other words it is forbidden to schedule another task earlier.

Sporadic tasks. Sporadic tasks are quite similar to periodic tasks, the only difference being that the period of a sporadic task denotes the *minimum* inter-arrival time instead of the *exact* one. While Funk et al. show that handle arrivals within a time slice is fairly straightforward (see [20], Section 6.) for sequential tasks we consider that extension to parallel gang tasks is no direct and that extension is left as future work.

6 Numerical experiments

Intensive numerical experiments have been performed using Matlab. The used LP solver is an interior point method (i.e., solver linprog included in Matlab). Source codes of all algorithms and experimental results are available at the project page[4] including a wiki page for detailing the file organization. We next detail the task set synthesis and the numerical results.

6.1 Task set synthesis

Input parameters for the task set synthesis are m, i.e., the number of processors in the platform, its total utilization U, and the number of tasks n. Stafford's algorithm is used for generating utilization factors u_i of gang tasks $\tau_1, \dots \tau_n$ to meet a total utilization of $m \times U$. As shown in [17], the method is suitable for task set synthesis for multiprocessor systems. The utilization factors of tasks are picked up by Stafford's algorithm in the interval $[0.02, m]$. The number of used processors for every gang task is generated using uniformly distributed pseudo random integers in the interval $[1, m)$. We do not allow a task to simultaneously use m processors since such a situation is not interesting from scheduling perspectives. Precisely, such tasks can be removed from the optimization problem in order to compute an optimal pattern, and added afterwards in the previously computed optimal pattern.

In DP-Fair scheduling, task individual periods and execution requirements are not useful since between two subsequent deadlines, d_j and d_{j+1}, the execution requirement to be scheduled is exactly $(d_{j+1} - d_j) \times u_i$ for every task τ_i, $1 \leq i \leq n$. Furthermore, the presented algorithms build up the pattern that will be stretched in every block in the DP-Fair schedule. The length of the interval is basically set to one hundred to avoid small decimal numbers that can lead to numerical problems while using a LP solver.

6.2 LP-based method evaluation

As previously mentioned, the optimal LP-based algorithm must handle two different combinatorial problems: the feasible subsets construction stage and the optimization stage. From the computational time point of view, the optimization stage is quite fast in comparison with the construction of all feasible subsets (i.e., setting up the matrix of constraints).

Figure 4 presents the computation times of Brute Force v.s. Depth First Search with Lexicographic order (DFSLex) for this problem construction for $m = 16$. In the following plots, every point corresponds to 1000 runs (i.e., simulation with replication factor equal to 1000). As commonly observed, Brute Force is still manageable until $n = 20$, but cannot be used beyond whereas the DFSLex algorithm runs quite efficiently. The drawback of the DFSLex algorithm is that it is limited to 64 gang tasks due to the binary encoding of feasible subsets as 64-bit integers. The DFSLex results for larger task sets are depicted in Figure 5. We also implement a similar version of that algorithm relaxing the 64-bit constraint by using variable integer arithmetic routines but it runs quite slowly in our Matlab implementation (e.g., it is as slow as the Brute Force algorithm for small task sets). All these implementations are available in the project page.

Figure 6 depicts the resolution times of the optimal algorithm (with both stages) for several numbers of processors and for global utilization equal to 50% and 90%. As depicted, the utilization factor has a moderate influence on average resolution times. Problems become harder to solve when first, the number of gang tasks increases, and second, when the number of processors increases;

[4] http://www.lias-lab.fr/forge/projects/multiprocessorgangscheduling

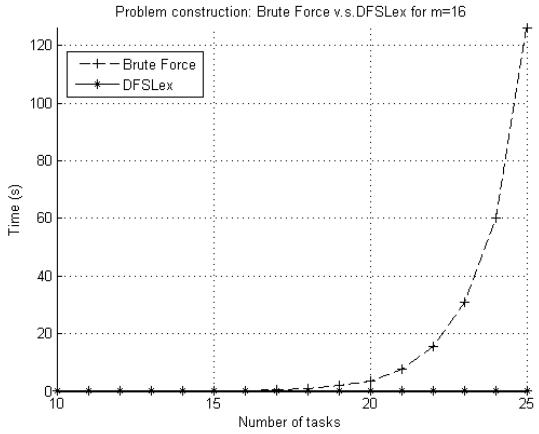


Figure 4 Brute Force v.s Depth First Search enumeration of feasible allocations in the linear program OPT.

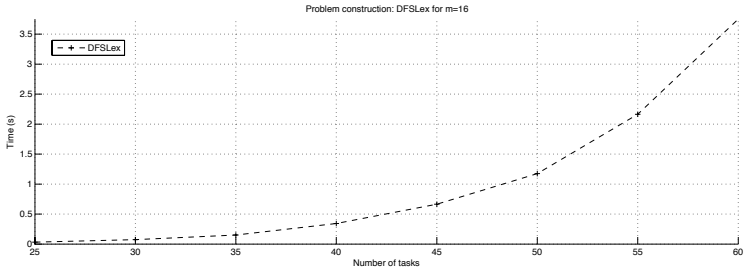


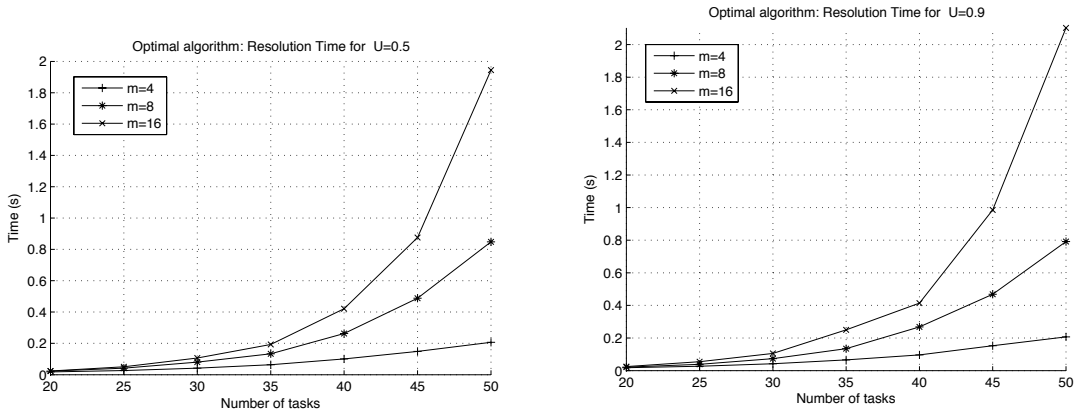
Figure 5 Depth First Search enumeration of feasible allocations for larger task sets.

but require few seconds in the average. In both cases, the feasible subsets construction requires an important amount of computations when the problem size increases.

6.3 Acceptance ratio

We compare the optimal algorithm (OPT) and the heuristic (gang-h) for computing a schedule pattern according to the average acceptance ratio for a platform with 16 processors and varying utilization factors. The used schedulability tests are Theorem 7 for the optimal algorithm and its sufficient version for the heuristic (i.e., if (gang-h) generates a schedule pattern of length not larger that 1, then the task set is schedulable). Figure 7 depicts the results for 20 and 40 tasks, respectively. The replication factor during the simulation is set to 10000 (i.e., every point in graphs is the average of 10000 results).

For the optimal algorithm, when the number of tasks increases in the experiment for $m = 16$ processors, then every task has relatively smaller individual utilization due to the task set synthesis method. As a consequence, there are more feasible subsets and consequently more feasible schedules. As depicted in Figure 7, the acceptance ratio for the optimal algorithm doubles for $U = 0.95$ when the number of tasks doubles. Such a benefit is not observed for the gang heuristic that achieves quite poor results when the total utilization becomes high.



(a) Resolution time for $U = 0.5$

(b) Resolution time for $U = 0.9$

Figure 6 Resolution times of the LP-based optimal algorithm.

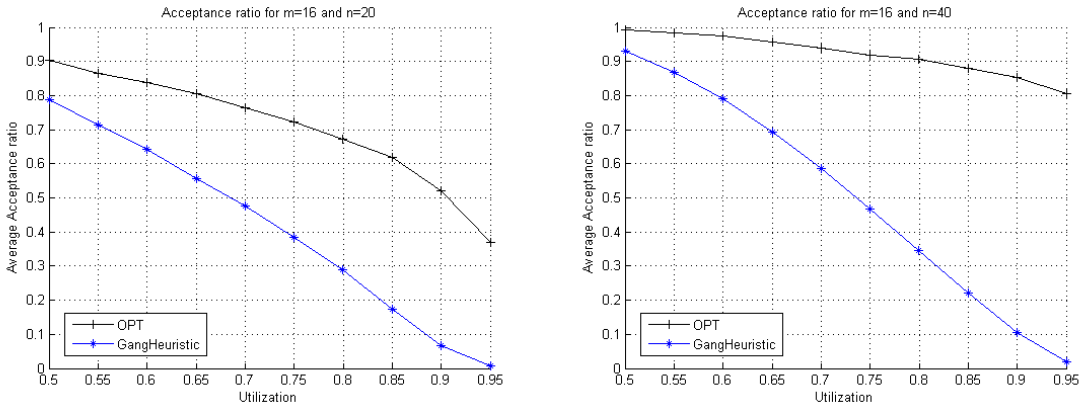
6.4 Average error

We also compare the optimal and heuristic algorithms according to the average error in comparison with the length of the schedule pattern. Let OPT be the length of the pattern computed by the optimal algorithm and UB be the corresponding upper bound computed by the gang heuristic, the error is defined by: $\text{err} = (\text{UB} - \text{OPT})/\text{OPT}$. Due to Theorem 11, we verify that: $\text{OPT} \leq \text{UB} \leq (2 - \frac{1}{m})\text{OPT}$. Hence, the error is bounded by $\text{err} \leq (1 - 1/m)$.

We perform comparison of algorithms for several numbers of tasks and processors that are depicted in Figure 8. As for the acceptance ratio, simulations for computing the average error have been replicated 10000 times. In these graphs, the y-axis is delimited by the worst-case error of $(1 - 1/m)$. First, the average error is not sensitive to the utilization factor but only to the number of tasks. Precisely, the average ratio compare the schedule lengths of the patterns computed by OPT and Gang-h. Varying utilization factors of synthetic task sets will define quite similar pattern shapes that lead to quite similar average error. Second, when the number of tasks increases, the average error also increases and but is still under half of the worst-case error.

7 Conclusion

In this paper we considered the preemptive scheduling of implicit-deadline periodic gang task systems upon identical multiprocessors. We proposed two algorithms which define static patterns that are stretched at run-time in a DP-Fair way. The first one is optimal and runs in polynomial time for a fixed number of processors; the second one is a sub-optimal fixed-priority rule but it is competitive under resource augmentation analysis. Precisely, the speedup factor of the heuristic is bounded by $(2 - \frac{1}{m})$. Our numerical experiments show that the optimal pattern can be computed efficiently up to 60 tasks and ensures a high acceptance ratio when the number of tasks is not too small. For larger systems ($m >> 16$ or $n > 64$), computing an optimal pattern becomes a hard combinatorial problems. In these cases as for most hard combinatorial problems, we think that heuristics (e.g., gang-h) must be used rather than an optimal algorithm. Concerning the proposed gang heuristic, the experiments show that the acceptance ratio decreases quasi-linearly according to the platform utilization factor, but the average error with respect to the optimal pattern length is less than 40%.



(a) Acceptance ratio for $n = 20$

(b) Acceptance ratio for $n = 40$

Figure 7 Acceptance ratio of the LP-based optimal algorithm and Gang Heuristic.

Future Work. Future work will concern the definition of a pattern schedule that aims to reduce the number of preemptions. This latter problem seems to be hard to cope with but still significant for allowing practical applications of real-time scheduling methods. As we said in the discussion Section 5.3 the case of sporadic task is left as future work.

References

1 Björn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In Roberto Baldoni, Paola Flocchini, and Binoy Ravindran, editors, *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, volume 7702 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2012. doi:10.1007/978-3-642-35476-2_2.

2 Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012*, pages 63–72. IEEE Computer Society, 2012. doi:10.1109/RTSS.2012.59.

3 Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. doi:10.1007/BF01940883.

4 V. Berten, P. Courbin, and J. Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *5th Junior Researcher Workshop on Real-Time Computing*, pages 9–12, 2011. URL: http://arxiv.org/abs/0805.3237.

5 Jacek Blazewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Computers*, 35(5):389–393, 1986. doi:10.1109/TC.1986.1676781.

6 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 225–233. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.32.

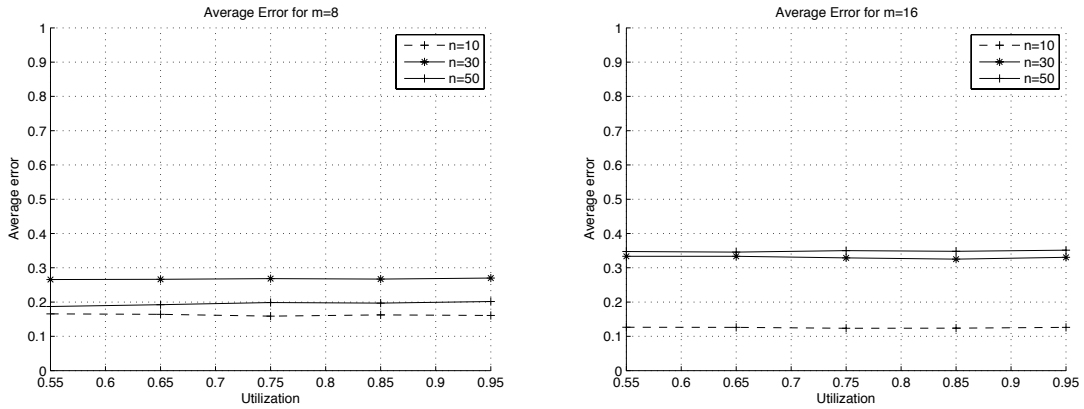
7 Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, chapter Scheduling Parallel Jobs on Clusters, pages 519–533. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

8 Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. URL: http://dl.acm.org/citation.cfm?id=355074.

9 Rohit Chandra, D Leonaldo, Kohr Dave, Maydan Dror, M Jeff, and Menon Ramesh. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001. URL: http://lib.mdp.ac.id/ebook/Karya%20Umum/Parallel_Programming_in_OpenMP.pdf.

10 Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 101–110. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.10.

11 Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time



(a) Average error for $m = 8$

(b) Average error for $m = 16$

Figure 8 Average error on the schedule pattern length for the LP-based optimal algorithm and gang heuristic.

scheduling theory. *Inf. Process. Lett.*, 106(5):180–187, 2008. doi:10.1016/j.ipl.2007.11.014.

12 Pierre Courbin, Irina Iulia Lupu, and Joël Goossens. Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks. *Real-Time Systems*, 49(2):239–266, 2013. doi:10.1007/s11241-012-9173-x.

13 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

14 M. Drozdowski. *Handbook of Scheduling Algorithms, Models and Performance Analysis*, chapter Scheduling Parallel Tasks — Algorithms and Complexity. Chapman & Hall/CRC, 2004.

15 Maciej Drozdowsli. On the complexity of multiprocessor task scheduling. *Bulletin of the polish academy of sciences*, 43(3):381–392, 1995. URL: http://www.cs.put.poznan.pl/mdrozdowski/txt/BullPAS95.pdf.

16 P.-F. Dutot, G. Mounie, and Denis Trystram. *Handbook of Scheduling Algorithms, Models and Performance Analysis*, chapter Scheduling Parallel Tasks — Approximation Algorithms. Chapman & Hall/CRC, 2004.

17 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor task sets. In *In proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010. URL: http://retis.sssup.it/waters2010/waters2010.pdf.

18 Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 13–22. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.15.

19 Shelby Funk. LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Systems*, 46(3):332–359, 2010. doi:10.1007/s11241-010-9109-2.

20 Shelby Funk, Greg Levin, Caitlin Sadowski, Ian Pye, and Scott A. Brandt. Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, 47(5):389–429, 2011. doi:10.1007/s11241-011-9130-0.

21 Shelby Funk and Vijaykant Nanadur. LRE-TL: An optimal multiprocessor scheduling algorithm for sporadic task sets. In *17th International Conference on Real-Time and Network Systems*, pages 159–168, 2009. URL: https://hal.inria.fr/inria-00442002/document.

22 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

23 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. URL: http://www.netlib.org/pvm3/book/pvm-book.html.

24 Joël Goossens and Pascal Richard. *Real-time Systems Scheduling*, volume Fundamentals, chapter Multiprocessor Architecture Solutions. Wiley-ISTE, September 2014. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1848216653.html.

25 Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. *CoRR*, abs/1006.2617, 2010. URL: http://arxiv.org/abs/1006.2617.

26 Sergei Gorlatch and Holger Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998. doi:10.1142/S0129626498000456.

27 R L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:416–426, 1966. URL: http://www.jstor.org/stable/2099572.

28 William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MIT Press, 2nd edition, 1999. URL: http://mitpress.mit.edu/books/using-mpi.

29 Berit Johannes. Scheduling parallel jobs to minimize the makespan. *J. Scheduling*, 9(5):433–452, 2006. doi:10.1007/s10951-006-8497-6.

30 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In Theodore P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 459–468. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.42.

31 Leonid G Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980. URL: http://www.sciencedirect.com/science/article/pii/0041555380900610.

32 Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, pages 259–268. IEEE Computer Society, 2010. doi:10.1109/RTSS.2010.42.

33 Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott A. Brandt. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 3–13. IEEE Computer Society, 2010. doi:10.1109/ECRTS.2010.34.

34 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Outstanding paper award: Analysis of global EDF for parallel tasks. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 3–13. IEEE Computer Society, 2013. doi:10.1109/ECRTS.2013.12.

35 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

36 Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luís Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 3. ACM, 2014. doi:10.1145/2659787.2659815.

37 R McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1), 1959. URL: http://www.jstor.org/stable/2627472.

38 Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 321–330. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.37.

39 Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pages 217–226. IEEE Computer Society, 2011. doi:10.1109/RTSS.2011.27.

40 Vaidy S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency - Practice and Experience*, 2(4):315–339, 1990. doi:10.1002/cpe.4330020404.

41 David W Walker and Jack J Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

42 Dakai Zhu, Daniel Mossé, and Rami G. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*, pages 142–151. IEEE Computer Society, 2003. doi:10.1109/REAL.2003.1253262.