HAL
open science

# Semi-matching algorithms for scheduling parallel tasks under resource constraints

Anne Benoit, Johannes Langguth, Bora Uçar

## To cite this version:

Anne Benoit, Johannes Langguth, Bora Uçar. Semi-matching algorithms for scheduling parallel tasks under resource constraints. [Research Report] RR-8089, 2012, pp.30. hal-00738393v1

**HAL Id: hal-00738393**

**https://inria.hal.science/hal-00738393v1**

Submitted on 4 Oct 2012 (v1), last revised 2 Jan 2014 (v5)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Semi-matching algorithms for scheduling parallel tasks under resource constraints

Anne Benoit, Johannes Langguth, Bora Uçar

RESEARCH REPORT
N° 8089
October 2012
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8089--FR+ENG

# Semi-matching algorithms for scheduling parallel tasks under resource constraints

Anne Benoit, Johannes Langguth, Bora Uçar

Project-Team ROMA

Research Report n° 8089 — October 2012 — 27 pages

**Abstract:** We study the problem of minimum makespan scheduling when tasks are restricted to subsets of the processors (resource constraints), and require either one or multiple distinct processors to be executed (parallel tasks). This problem is related to the minimum makespan scheduling problem on unrelated machines, as well as to the concurrent job shop problem, and it amounts to finding a semi-matching in bipartite graphs or hypergraphs. While the problem was known to be NP-complete for bipartite graphs, but solvable in polynomial time for unweighted graphs (i.e., unit tasks), we prove that the problem is NP-complete for hypergraphs even in the unweighted case. We design several greedy algorithms of low complexity to solve two versions of the problem, and assess their performance through a set of exhaustive simulations. Even though there is no approximation guarantee on these linear algorithms, they return solutions close to the optimal (or a known lower bound) in average.

**Key-words:** semi-matching, bipartite graphs, hypergraphs, scheduling, parallel tasks, resource constraints.

**RESEARCH CENTRE**
**GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

# Algorithmes de couplage partiel pour l'ordonnancement de tâches parallèles sous contraintes de ressources

**Résumé :** On étudie le problème d'ordonnancement visant à minimiser le temps d'exécution lorsque les tâches peuvent être exécutées soit sur un seul processeur, soit sur plusieurs processeurs en parallèle, et sont restreintes à certains sous-ensembles de processeurs. Le problème se ramène à devoir trouver un couplage partiel dans un graphe biparti ou dans un hypergraphe. Sur les graphes bipartis, le problème général est NP-complet, mais il est possible de résoudre en temps polynomial les instances de problèmes où toutes les tâches sont de poids unitaire. Nous montrons dans ce rapport que le problème devient NP-complet pour les hypergraphes même dans le cas unitaire. Nous proposons plusieurs algorithmes gloutons pour résoudre deux versions du problème, et nous étudions leur performance à travers des simulations. Bien qu'il n'y ait aucune garantie d'approximation sur ces algorithmes, ils retournent en moyenne des solutions proches de l'optimal (ou d'une borne inférieure connue), avec un temps d'exécution très rapide.

**Mots-clés :** couplage partiel, graphes bipartis, hypergraphes, ordonnancement, tâches parallèles, contraintes de ressources.

# 1 Introduction

The *Minimum Makespan Scheduling Problem on Unrelated Machines* is a classical topic in scheduling [7]. It can be described as follows: given a set of tasks and a set of processors, assign the tasks to the processors such that the load among the processors is balanced, i.e., the maximum load of a processor is minimized. The tasks usually differ in their processing time, i.e., in the load that they create on the processor they are assigned to.

For today's high performance computing environment dominated by server virtualization, cloud computing, application accelerators and emerging architectures, we need to refine the problem formulation. Indeed, in classical scheduling, while the need for different computational resources can be expressed through the difference in processing time, it does not express the fact that a single task may have a choice among combinations of multiple computational resources. We therefore consider the MultiProc problem where (i) tasks are parallel, i.e., a task can be split in several identical parts and computed simultaneously on several processors; (ii) tasks are subject to resource constraints, i.e., several configurations of processors, leading to different execution times, are proposed. The goal is to find one configuration for each task, in order to minimize the makespan.

MultiProc is related to the concurrent job shop problem [1]. In this problem, a job consists of multiple different components, each of which is to be processed on a specific dedicated processor. Components of the same job can be processed in parallel on their respective processors. A job is completed once all of its components are completed. According to Roemer [14], the problem was introduced by Ahmadi and Bagchi [1]. It has been studied widely [11, 12, 17] and was proved to be strongly NP-complete [14]. The main difference between MultiProc and the concurrent job shop problem is the resource constraints, i.e., whether components are restricted to a specific processor or not. In MultiProc, each task has the choice among multiple different sets of processors (different configurations). Those sets can differ in size (i.e., number of processors), but processing times are equal for all processors in each set. Most of the time, if there are more processors in a set, then the execution time becomes smaller on each processor of the set. Similarly to the concurrent job shop problem, the components of the tasks (i.e., the elements of the processor sets for each task) are independent: they do not require execution at the same time, and no order of execution is specified. Note that we only consider the overall makespan as objective function.

In graph theoretical terms, an instance of the MultiProc problem can be modeled as a hypergraph, and finding a schedule of minimum makespan amounts to finding a semi-matching in the hypergraph, where the matching hyperedges are to be disjoint when restricted to a subset of vertices. As far as we know, this problem has not been tackled before.

We also consider a variant of the problem with sequential tasks (and resource constraints), SingleProc. Here, each task is scheduled only on a single processor, chosen among a set of possible processors (and corresponding execution times). In this case, the hypergraph is in fact a bipartite graph, and the problem consists therefore of finding a semi-matching in a bipartite graph. This has been studied intensively in the case of unweighted bipartite graphs, i.e., unit weight tasks SingleProc-Unit: several polynomial time algorithms were

proposed [4, 8]. The weighted version turns out to be NP-complete, for which a 3/2-approximation algorithm has been recently proposed [13]. However, the running time of this approximation algorithm is quadratic in the total number of tasks (with the average degree as a multiplicative factor), and hence not very practical for large problem instances.

Our contribution is twofold, and stands at the crossroad between scheduling and graph theory.

1. For the SinglеProc-Unit problem, we design a set of linear time heuristics, and we investigate the quality of their approximations. The results indicate that even though they have no approximation guarantee (i.e., we can build examples in which they are as far as possible from the optimal), their average behavior is very good when compared to the optimal solution, and they are suitable for practical application, e.g., load balancing in parallel computing.

2. For the MultiProc problem, we prove that it is NP-complete even in the unweighted case (while SinglеProc-Unit was solvable in polynomial time). Moreover, for all $\epsilon > 0$, there is no $(2-\epsilon)$-approximation algorithm unless P=NP. Building on the concepts used in the previous heuristics, we design a new set of heuristics for the most general problem instance. Since it is not possible to compute easily the optimal solution, we rely on a lower bound to assess the performance of the heuristics, and present exhaustive simulation results.

The remainder of this paper is organized as follows. We start with a formal description of the optimization problems in Section 2. We then prove in Section 3 that the general problem MultiProc is NP-complete, even with unit weight tasks. We design several linear time algorithms for two variants of the problem in Section 4, and then assess their performance in Section 5. Finally, we conclude in Section 6 with a summary and plans on further investigations of the addressed problems.

## 2 Framework

We consider the problem of scheduling $n$ independent tasks onto a set of $p$ processors, with the objective of minimizing the makespan, i.e., the maximum load of a processor. Let $T_1, \ldots, T_n$ be the set of tasks, and let $P_1, \ldots, P_p$ be the set of processors.

For $1 \leq i \leq n$, task $T_i$ is subject to *resource constraints*: it can be executed only on some of the processors, and possibly in parallel on several processors (*parallel* task). There is therefore a set of possible configurations for each task, e.g., task $T_1$ can be processed either on processor $P_1$, or concurrently on processors $P_2$ and $P_3$. Let $\mathcal{S}_i$ be the different configurations for $T_i$, i.e., the collection of sets of processors on which $T_i$ can be executed. Back to our example, $\mathcal{S}_1 = \{\{P_1\}, \{P_2, P_3\}\}$. Task $T_i$ is executed on a set of processors $alloc(i) \in \mathcal{S}_i$, and it takes a time $w_i^{alloc(i)}$ on each of the processors $P_u \in alloc(i)$. The processing can be done at different time steps on the processors of $alloc(i)$, since the task is executed in parallel, and we assume that the different parts of the

task are independent, similarly to the concurrent job shop problem. The goal is to find a mapping of tasks to processors, i.e., decide the set $alloc(i)$ of processors on which $T_i$ is executed, for $1 \leq i \leq n$.

We define the load $l(u)$ of processor $P_u$ as its execution time:

$$l(u) = \sum_{i \mid P_u \in alloc(i)} w_i^{alloc(i)}.$$

The goal is to complete all tasks as soon as possible, i.e., minimize the *makespan* $M = \max_{1 \leq u \leq p} l(u)$. We consider several problem variants in the following, and describe the problems in the graph and hypergraph formalisms, so that the scheduling problem amounts to finding a semi-matching.

## 2.1 With a single processor

In some cases, tasks cannot be executed in parallel, and $\mathcal{S}_i$ is just a set of processors on which $T_i$ can be executed, instead of a *set of sets*. This problem is called SingleProc. Note that it amounts to finding a semi-matching in a bipartite graph. We recall some graph definitions below to ease the description.

In a bipartite graph $G = (V_1 \cup V_2, E)$, the vertex sets $V_1$ and $V_2$ are disjoint and for all edges in $E$, one of the endpoints belongs to $V_1$ and the other belongs to $V_2$. In our problem, $V_1$ is the set of tasks, $V_2$ is the set of processors, and an edge $e = (T_i, P_u) \in E$ between a task $T_i \in V_1$ and a processor $P_u \in V_2$ means that $P_u$ is in the set $\mathcal{S}_i$ (see Fig. 1, where $S_1 = \{P_1, P_2\}$ and $S_2 = \{P_1\}$). We use $d_v$ to refer to the number of neighbors of a vertex $v \in V_1 \cup V_2$. Moreover, we can add weights to the edges, that correspond to execution times: $w(e) = w(T_i, P_u) = w_i^{P_u}$.

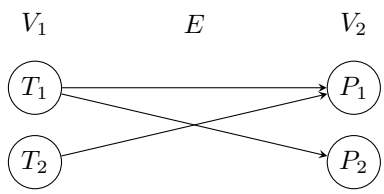


Figure 1: A sample bipartite graph for SingleProc-Unit.

Given a bipartite graph $G = (V_1 \cup V_2, E)$, a *semi-matching* $\mathcal{M}$ in $G$ is a set of edges $\mathcal{M} \subseteq E$ such that each vertex $v \in V_1$ is incident to exactly one edge in $\mathcal{M}$, i.e., it corresponds to the allocation function $alloc(i)$. Given a semi-matching $\mathcal{M}$, the load $l(u)$ of $u \in V_2$ is the sum of the weights of the edges in $\mathcal{M}$ incident on $u$. The objective is to find a semi-matching $\mathcal{M}$ such that $\max_{u \in V_2} l(u)$ is minimized.

This SingleProc problem was shown to be NP-complete [13] by reduction from the *Minimum Makespan Scheduling Problem on Identical Machines*, which differs from SingleProc in the fact that the tasks can be run on any machine (i.e., no resource constraints). It was also noted there that SingleProc can be reduced to the *Minimum Makespan Scheduling Problem on Unrelated Machines*, a more general formulation where the tasks can vary in execution time on different processors. For this problem, a 2-approximation algorithm was given by

Graham et al. [7], which was subsequently improved to $2 - \frac{1}{p}$ by Shchepin and Vakhania [16].

We consider also the unweighted version, that amounts to having $w_i^{P_u} = 1$ for $1 \leq i \leq n$ and $P_u \in \mathcal{S}_i$. This corresponds to unit tasks, and the problem is called SingleProc-Unit. This simpler instance of the problem can be solved via bipartite graph matching algorithms in polynomial time, as shown in [8]. In Section 4, we design several linear time heuristics for this SingleProc-Unit problem, since this allows us to compare the heuristics with the optimal solution in a reasonable time. Heuristics are then extended to solve the most general problem, that we detail below.

## 2.2 With multiple processors

The general problem is called MultiProc, and $\mathcal{S}_i$ is now a set of *sets of processors*. Back to the graph theory, this problem can then be seen as a matching problem in hypergraphs. A hypergraph $\mathcal{H} = (V, \mathcal{N})$ consists of a set of vertices $V$ and a set of hyperedges $\mathcal{N}$. Each hyperedge is a subset of vertices.

A MultiProc problem instance can be modeled as a bipartite hypergraph $\mathcal{H} = (V, \mathcal{N})$ of the following form: the vertex set is bipartite ($V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset$), and each hyperedge $h \in \mathcal{N}$ satisfies $|V_1 \cap h| = 1$, i.e., one single task $T_i \in V_1$ is associated to a set of processors in $V_2$ through an hyperedge. In the example of Fig. 2, tasks $T_3$ and $T_4$ have only one configuration (they are in a single hyperedge), and must therefore be executed on $P_3$. Tasks $T_1$ and $T_2$ can be executed in parallel and have the choice between several configurations. For example, $T_1$ can be executed by $P_1$ sequentially or by $P_2$ and $P_3$ collectively.

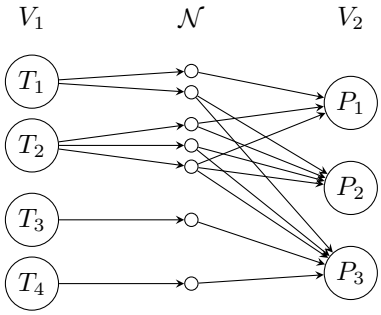


Figure 2: A sample hypergraph for MultiProc.

The problem now amounts to finding a semi-matching in a hypergraph, i.e., a set of hyperedges $\mathcal{M} \subseteq \mathcal{N}$ where the hyperedges in $\mathcal{M}$ are disjoint on the vertices in $V_1$. Thus, for all $T_i \in V_1$, there must be exactly one hyperedge $h_i$ in $\mathcal{M}$ such that $h_i \cap V_1 = T_i$. We then define $alloc(i) = h_i \cap V_2$. The processor load $l(u)$ of $u \in V_2$ is equal to the sum of the weights of hyperedges incident on $u$ in $\mathcal{M}$. For the ease of notations, we let $w_h = w_{h \cap V_1}^{h \cap V_2}$ be the weight associated to hyperedge $h \in \mathcal{N}$. We use $d_v$ to denote the number of hyperedges containing the vertex $v \in V_1$.

The unweighted version of this problem, where all weights are 1, is called MultiProc-Unit. While it is possible to solve SingleProc-Unit in poly-

nomial time, MultiProc-Unit turns out to be NP-complete, even with unit weights (see Section 3).

# 3 NP-completeness of MultiProc-Unit

It was shown that SingleProc is NP-complete [13], while SingleProc-Unit is solvable in polynomial time [8]. We show that MultiProc is NP-complete even for the unweighted version MultiProc-Unit. Moreover, the reduction implies approximation hardness.

**Theorem 1.** *The problem* MultiProc-Unit *is NP-complete, and for all* $\epsilon > 0$*, there is no* $(2-\epsilon)$*-approximation algorithm unless P=NP.*

**Proof**. We consider the associated decision problem: given an instance of MultiProc-Unit and a bound on the makespan $D$, is there a solution of makespan not larger than $D$? This problem is obviously in NP, since the makespan can be computed in linear time, given an assignment of tasks to processors.

To establish the completeness, we use a reduction from Exact Cover by 3-Sets (X3C) [5, p. 53]. We consider an instance $\mathcal{I}_1$ of X3C: given a finite set $X$ of elements where $|X| = 3q$ and a collection $C$ of 3-element subsets of $X$, does $C$ contain an exact cover $C' \subseteq C$ such that every element of $X$ occurs in one member of $C'$.

We build an instance $\mathcal{I}_2$ of MultiProc-Unit: the set of elements of $\mathcal{I}_1$ are the processors, i.e., the vertex set $V_2$ in the hypergraph formulation. There are $q$ tasks to be mapped on these $3q$ processors, i.e., $|V_1| = q$. Each of these tasks can be mapped onto the sets of processors corresponding to the collection $C$, i.e., $\mathcal{S}_i = C$ for $1 \leq i \leq n$. Moreover, we set the deadline $D = 1$.

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

If $\mathcal{I}_1$ has a solution, i.e., there is an exact cover, then we assign each task to the set of processors corresponding to one member of $C'$. Each processor is therefore processing exactly one task, and the makespan is $1 \leq D$, therefore $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. Since the makespan is at most 1, each processor can process at most one task, and since each task is executed on three distinct processors by construction of $\mathcal{S}_i$, the allocation of $\mathcal{I}_2$ forms a cover for $\mathcal{I}_1$, and hence the result.

There remains to prove the inapproximability result. Let us assume that there is a $(2-\epsilon)$-approximation algorithm of MultiProc-Unit, with $\epsilon > 0$. Then, we use this algorithm to solve instance $\mathcal{I}_2$, hence obtaining a makespan $M \leq (2-\epsilon)M_{opt}$, where $M_{opt}$ is the optimal makespan. Since $M_{opt} = 1$, we obtain $M < 2$, and hence $M = 1$ since all weights are unit. The algorithm has therefore found an optimal solution, that corresponds to a cover, in polynomial time. This cannot hold unless P=NP. □

Since the MultiProc problem is NP-complete even with unit weights (see MultiProc-Unit), we derive efficient algorithms to solve it. We design algorithms also for the simpler version of the problem SingleProc-Unit, since we are then able to compare heuristics to the optimal solution.

# 4 Algorithms

In this section, we propose heuristics for SingleProc-Unit and MultiProc, as well as a polynomial time exact algorithm for SingleProc-Unit and a lower bound for MultiProc, which we use for experimental comparison. The exact algorithm for SingleProc-Unit and the lower bound for MultiProc are developed to obtain baseline values for evaluating the proposed heuristics. The exact algorithm for SingleProc-Unit can be expensive for large problem instances, and therefore the heuristics for SingleProc-Unit are designed to be of linear time. Even though we focus on the unweighted case SingleProc-Unit, the heuristics can be easily extended to weighted instances without adversely affecting their running time. The heuristics for the MultiProc problem are based on the heuristics for SingleProc-Unit.

## 4.1 Exact Algorithm for SingleProc-Unit

As shown by Harvey et al. [8], SingleProc-Unit can be solved using some modified versions of the standard matching algorithms. We propose a conceptually simpler algorithm by making use of the standard matching algorithms (see a relatively recent survey [3] on augmenting-path based ones, and other studies,[6, 9] on push-relabel based ones). Assume a deadline $D = 1$ and run the push-relabel algorithm on $G$. If a perfect matching is found, we have found a schedule of makespan 1, hence an optimal schedule. Otherwise, increase $D$ by 1 and run the push-relabel algorithm on $G_D$, where $G_D$ is identical to $G$ except that it contains a total of $D$ copies of each vertex in $u \in V_2$, each having the same neighborhood as the original vertex $u$. Repeat this process until a matching covering all the task vertices is found, at which time $D$ is equivalent to the optimal makespan.

At step $D$, the algorithm has a complexity of $O(\sqrt{|V_1|}|E|D)$, and there are $M_{opt}$ steps, where $M_{opt}$ is the optimal makespan. Hence the algorithm has a running time complexity of $O(\sqrt{|V_1|}|E||M_{opt}^2)$. Note that $M_{opt} \leq |V_1|$, since the worst case is when all tasks are mapped on the same machine.

## 4.2 Greedy Algorithms for SingleProc-Unit

### 4.2.1 Basic-greedy

The basic greedy algorithm is straightforward (see Algorithm 1). It loops through the tasks in $V_1$ and assigns each task $v \in V_1$ to a processor in the neighborhood of $v$ that has the smallest current load. The running time is $O(|E|)$. Even though this algorithm often performs reasonably well, there are some instances in which it performs poorly. A toy example with two tasks (on the left) and two processors is shown in Fig. 1. If $T_1$ is mapped to $P_1$, with $T_2$ having a single choice, the *basic-greedy* algorithm can assign the two tasks to processor $P_1$ and reach a makespan of 2 (versus 1 for the optimal solution). We show below that the basic greedy algorithm does not have any approximation guarantee.

**Input:** A bipartite graph $G = (V_1 \cup V_2, E)$
**Output:** A matching $\mathcal{M}$ in $G$
1: **for all** $v \in V_1$ **do**
2: find an edge $e = \{v, u\} \in E$ for which $l(u)$ is minimum
3: $\mathcal{M} \leftarrow \mathcal{M} + e$
4: $l(u) \leftarrow l(u) + 1$
5: **return** $\mathcal{M}$

**Algorithm 1:** *basic-greedy*

#### 4.2.2 Sorted-greedy

We improve *basic-greedy* by sorting tasks by non-decreasing out-degrees. The idea is to schedule the tasks that have less freedom first, e.g., task $T_2$ in the example of Fig. 1. The only modification to Algorithm 1 consists in visiting the tasks according to a non-decreasing order of degrees ($d_v$) at line 1. Unfortunately, this *sorted-greedy* algorithm may also take wrong decisions. We show here an example where it is at a factor $k$ from the optimal solution, for any $k$ (this is an example also showing that *basic-greedy* can be arbitrarily far from the optimal).

Consider that there are $2^k - 1$ tasks to be mapped onto $2^k$ processors. For the ease of reading, tasks are named $T_i^{(\ell)}$, with $0 \leq \ell \leq k-1$, and $1 \leq i \leq 2^{k-1-\ell}$. Task $T_i^{(\ell)}$ can be placed either on processor $P_i$, or on processor $P_{i+2^{k-1-\ell}}$ (see Fig. 3, for $k = 3$).

The optimal solution places $T_i^{(\ell)}$ on $P_{i+2^{k-1-\ell}}$, for $0 \leq \ell \leq k-1$, and $1 \leq i \leq 2^{k-1-\ell}$. There is only one task per processor, hence an optimal schedule has a makespan of 1. However, the *sorted-greedy* algorithm starts by placing tasks $T_i^{(0)}$ on processors $P_1$ through $P_{2^{k-1}}$, and then all processors that can be used for tasks $T_i^{(1)}$ have already a makespan of 1, it places them on processors $P_1$ through $P_{2^{k-2}}$, and so on. Finally, task $T_1^{(k-1)}$ is also mapped on $P_1$, and processor $P_1$ achieves a makespan of $k$.

#### 4.2.3 Double-sorted

From the example above, it seems better to also sort processors by increasing in-degrees, when there is a tie (i.e., edges leading to identical loads). This *double-sorted* algorithm is detailed as Algorithm 2.

This algorithm may also take wrong decisions. We can for instance generalize the example of Fig. 3 by adding extra tasks with higher degree and processors, so that all processors from the previous example have the same degree and the newcomers have a smaller degree. We illustrate this counter-example only for the case $k = 3$, see Fig. 4. Tasks $T_9$ to $T_{12}$ have a larger out-degree, so they are considered last. Then, since processors $P_1$ to $P_8$ have an identical in-degree of 3, *double-sorted* may take the same wrong decisions as *sorted-greedy* does, and obtain a makespan of 3 (while the optimal makespan is 1).

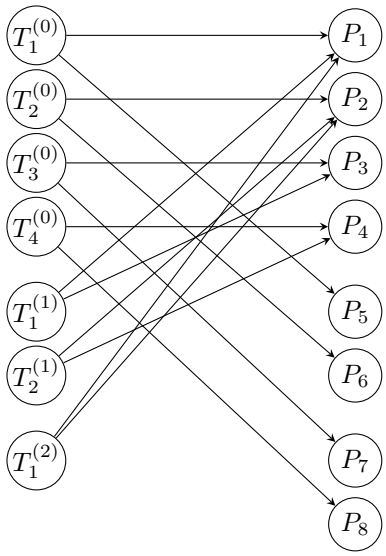


Figure 3: Example where *basic-greedy* and *sorted-greedy* obtain a makespan of $k = 3$, while the optimal makespan is 1.

**Input:** A bipartite graph $G = (V_1 \cup V_2, E)$
**Output:** A matching $\mathcal{M}$ in $G$

```
for all v ∈ V1, sorted by non-decreasing out-degree do
    minl ← n
    mind ← n
    for all e = {v, u} ∈ E do
        if l(u) < minl or (l(u) = minl and d_u ≤ mind) then
            minl ← l(u)
            mind ← d_u
            mine ← e
    M ← M+mine
    l(u) ← l(u) + 1
return M
```

**Algorithm 2:** *double-sorted*

**Input:** A bipartite graph $G = (V_1 \cup V_2, E)$
**Output:** A matching $\mathcal{M}$ in $G$

```
for all u ∈ V_2 do
    o(u) ← 0
for all v ∈ V_1 do
    for all {v, u} ∈ E do
        o(u) ← o(u) + 1/d_v
for all v ∈ V_1, sorted by non-decreasing out-degree do
    find an edge e = {v, u} ∈ E for which o(u) is minimum
    M ← M + e
    o(u) ← o(u) + 1 − 1/d_v
    for all {v, u} ∈ E \ {e} do
        o(u) ← o(u) − 1/d_v
return M
```

**Algorithm 3:** *expected-greedy*

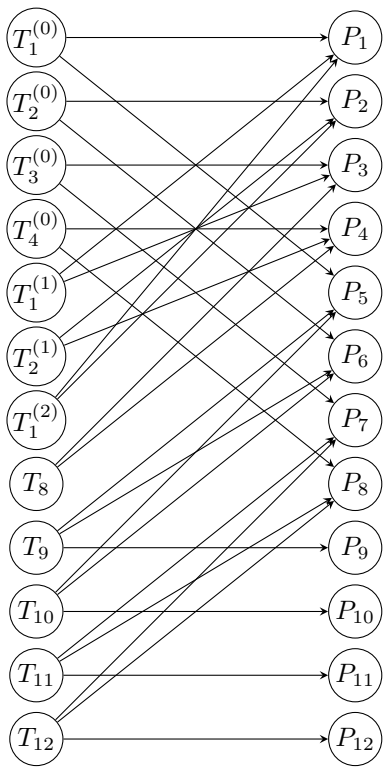


Figure 4: Example where *double-sorted* obtains a makespan of $k = 3$, while the optimal solution is 1.

#### 4.2.4 Expected-greedy

As we have seen above, a weakness of the greedy algorithms is their inability to predict load that will arrive at a given vertex later during the execution of the algorithm. In this last greedy algorithm, we add a simple load prediction technique to *sorted-greedy* and adapt the strategy for the assignment of matching vertices. The resulting algorithm shown in Algorithm 3 is referred to as *expected-greedy*.

In this algorithm, $o(u)$ represents the expected load of a vertex in $V_2$ (or processor). The values $o(u)$ can be interpreted as the expected load a vertex $u$ would have if the remaining matchings were performed uniformly at random. Actually, matching $v$ to $u$ can be seen as the collapse of the probability function. Consequently $u$, i.e., the possibility that was realized is assigned a probability of 1 and all other possibilities (i.e., neighbors of $v$) are assigned a probability of 0. The values of $o$ are updated accordingly. When the algorithm terminates, the values $o(u)$ are equivalent to actual loads $l(u)$, and their maximum is equal to the makespan. One immediately verifies that the running time remains $O(|E|)$.

On the example of Fig. 4, the values of $o(u)$ differ since tasks $T_9$ to $T_{12}$ are of degree 3, while the others are of degree 2. Therefore, *expected-greedy* places $T_1^{(0)}$ on $P_5$, as in the optimal solution, and reaches the optimal makespan of 1. However, it is possible to modify the example so that *expected-greedy* also takes the wrong decisions, by having 16 tasks and 16 processors, and all tasks of degree 2, as in Fig. 5. Tasks $T_9$ to $T_{16}$ can be assigned to their own processor, and processors $P_1$ to $P_8$ have a degree 3. Therefore, the same wrong decisions will be taken by *expected greedy* and by *double-sorted* (i.e., tasks $T_i^{(0)}$ will be mapped on $P_i$, for $1 \leq i \leq 4$, and so on).

Note that those worst cases are however extremely unlikely in practice. Therefore, we study the quality offered by the heuristics and contrast it with the optimum values obtained from the exact algorithm. Results are discussed in Section 5.

### 4.3 Lower bound for MultiProc

Since we cannot compute the optimal solution for MultiProc, we derive a lower bound so that we will be able to assess the performance of the heuristics, later in Section 5. For each task $T_i \in V_1$, we find a hyperedge $h_i \in \mathcal{N}$ such that $T_i \in h_i$, and $w_{h_i} \times |h_i \cap V_2|$ is minimum. We then define

$$time_i = \min_{h_i \in \mathcal{N}: T_i \in h_i} w_{h_i} \times |h_i \cap V_2| \ .$$

Since all tasks must be executed on the $p$ processors, the ideal case is when all processors achieve an identical load, equal to the makespan. Therefore, an obvious lower bound is one in which each task is in the best configuration in terms of the global load, leading to a total execution time of $time_i$, and where the load is equally shared between processors:

$$LB = \frac{1}{p} \sum_{1 \leq i \leq n} time_i \ .$$

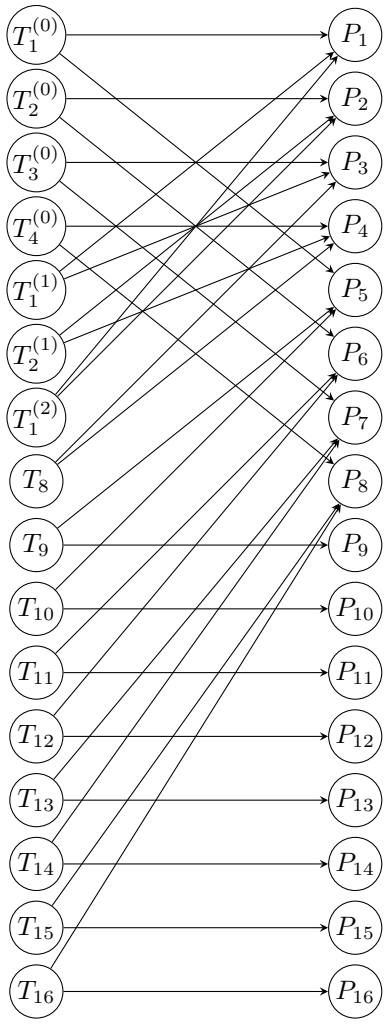


Figure 5: Example where *expected-greedy* obtains a makespan of $k = 3$, while the optimal solution is 1. Note that all tasks are of degree 2, processors $P_1$ to $P_8$ are of degree 3, and processors $P_8$ to $P_{16}$ are of degree 1.

## 4.4 Greedy Algorithms for MultiProc

In this section, we aim at adapting the previous greedy algorithms for the MultiProc problem. On one hand, we need to account for the fact that a task may be executed on several processors. On the other hand, we need to account for task incurring different weights in different configurations, since we considered only the unweighted case previously.

We consider four heuristics for MultiProc, mainly based on the *sorted-greedy* algorithm for SingleProc-Unit. We exploit the hypergraph structure by introducing a new way of deciding which set of processors to choose (i.e., which hyperedge) for a given task (see the *vector* heuristics).

### 4.4.1 Sorted-greedy-hyp

Adapting *sorted-greedy* for MultiProc requires only minimal effort. Instead of choosing a neighbor of $v$ having minimum current load, we chose a hyperedge $h$ that minimizes $\max_{u \in h} l(u)$, among all hyperedges incident to $v$ (see Algorithm 4). We also consider the weights in the new version of the algorithm, when computing the load $l(u)$.

**Input:** A hypergraph $\mathcal{H} = (V_1, V_2, \mathcal{N})$
**Output:** A matching $\mathcal{M}$
1: **for all** $v \in V_1$, sorted by non-decreasing out-degree **do**
2: find a hyperedge $h : v \in h$ for which $\max_{u \in h} l(u)$ is minimum.
3: $\mathcal{M} \leftarrow \mathcal{M} + h$
4: **return** $\mathcal{M}$

**Algorithm 4:** *sorted-greedy-hyp*

The running time now depends on the number of $V_2$ vertices in the hyperedges being inspected. In the worst case, the running time becomes $O(\sum_{h \in \mathcal{N}} |h|)$. Since bipartite graph semi-matching is a special case of hypergraph semi-matching, this algorithm also does not have an approximation guarantee. However, approximation of hypergraph semi-matching faces the additional difficulty that a single task can increase the load on multiple processors.

### 4.4.2 Expected-greedy-hyp

The *expected-greedy* algorithm can also be naturally extended to hypergraphs. In this case, when computing the values $o(u)$, a hyperedge $h$ containing $v$ assigns its value of $w_h/d_v$ to all vertices of $V_2$ contained in $h$, where $w_h$ is the weight associated to the hyperedge, i.e., the execution time of task corresponding to $v$ on each processor of the hyperedge. Other computations of $o(\cdot)$ are performed accordingly, as shown in Algorithm 5.

The running time of this algorithm is $O\left(\sum_{h \in \mathcal{N}} |h|\right)$. Due to the updates of $o$, its running time will generally be in $O\left(\sum_{h \in \mathcal{N}} |h|\right)$ since hyperedges cannot be skipped during the updates. As discussed above, in the hypergraph case the additional information provided by the $o$ values is far more important, since the potential mistakes the basic greedy algorithm can make are far greater.

**Input:** A hypergraph $\mathcal{H}=(V_1, V_2, \mathcal{N})$
**Output:** A matching $\mathcal{M}$
1: **for all** $u \in V_2$ **do**
2: $\quad o(u) \leftarrow 0$
3: **for all** $v \in V_1$ **do**
4: $\quad$ **for all** $h \in \mathcal{N} : v \in h$ **do**
5: $\quad\quad$ **for all** $u \in V_2 : u \in h$ **do**
6: $\quad\quad\quad o(u) \leftarrow o(u) + w_h/d_v$
7: **for all** $v \in V_1$, sorted by non-decreasing out-degree **do**
8: $\quad$ find a hyperedge $h : v \in h$ for which $\max_{u \in h} o(u)$ is minimum.
9: $\quad \mathcal{M} \leftarrow \mathcal{M} + h$
10: $\quad$ **for all** $u \in V_2 : u \in h$ **do**
11: $\quad\quad o(u) \leftarrow o(u) + w_h - w_h/d_v$
12: $\quad$ **for all** $h' \in \mathcal{N} \setminus \{h\} : v \in h'$ **do**
13: $\quad\quad$ **for all** $u \in V_2 : u \in h'$ **do**
14: $\quad\quad\quad o(u) \leftarrow o(u) - w_{h'}/d_v$
15: **return** $\mathcal{M}$

**Algorithm 5:** *expected-greedy-hyp*

#### 4.4.3 Vector-greedy-hyp

Consider the *sorted-greedy-hyp* algorithm. At line 2, instead of looking at the loads increased by the hyperedge, we can look at the current bottleneck value, e.g., $\max_{u \in V_2} l(u)$. Clearly there will be many ties. In these case, we can favor the hyperedge that has the smaller second largest load. This tie breaking mechanism can be extended to check the load vectors sorted in descending order lexicographically. That is, among the hyperedges, choose the ones that yield the smallest largest $l(\cdot)$ value; among the alternatives choose the ones that yield the smallest second largest $l(\cdot)$ value and so on.

The worst case running time complexity of this heuristic can be shown to be $O\left(\sum_{v \in V_1} d_v |V_2| \log |V_2| + \sum_{h \in \mathcal{N}} |h|\right)$, as checking the sorted load vectors lexicographically requires a sort operation (on $|V_2|$ items) for each hyperedge. Two improvements of this running time are immediate. First, if the hyperedges are unit weighted, then the sort operation can be done in linear time using bucket sort (or counting sort). Furthermore, one can keep the current load vector sorted as a list and then obtain the sorted load vector of an hyperedge by modifying the positions of modified loads. Here one can take advantage of the list already being sorted to reduce the sort operation as the merge of two lists (one of them is the processors in the hyperedge, the other is the remaining ones). This variant has the worst case time complexity of $O\left(\sum_{v \in V_1} d_v |V_2| + \sum_{h \in \mathcal{N}} |h|\right)$.

#### 4.4.4 Expected-vector-greedy-hyp

This last heuristic is a combination of the *expected* and *vector* greedy heuristics on hypergraphs. There is one difficulty though. The current expected load vector of the processors contains contributions from each hyperedge associated with the task to be assigned. In order to differentiate between the hyperedges, one of them should be tentatively realized, and the others should

be tentatively discarded so that the effect of each hyperedge can be measured. For a vertex with $d_v$ hyperedges, this requires $O(d_v \sum_{v \in h} |h|)$ operations. The overall complexity of the algorithm with the list representation would be $O\left(\sum_{v \in V_1} d_v |V_2| + \sum_v d_v \sum_{v \in h} |h|\right)$. The first term would likely be the dominant one in reasonable settings but the overhead due to the second term (with respect to the *vector-greedy-hyp*) can be significant.

# 5 Experiments

We have implemented the proposed heuristics in Matlab and run the codes on a MacBook Pro equipped with a 2.7 GHz Intel Core i7 processor and 8GBytes of 1333 MHz DDR3 ram. We used an implementation of the push-relabel algorithm [9] provided in MatchMaker suit [3]. The reported running times of algorithms are in seconds and obtained by `tic-toc` routines of Matlab.

Below, we first explain how the data has been generated. Then we detail experimental results for SingleProc-Unit and MultiProc.

## 5.1 Data set

We have simulated the algorithms in settings where the number of tasks $n$ is in $\{1280, 5120, 20480\}$, and the number of processors $p$ is in $256, 1024, 4096$; we did not test the cases where $n < 5 \times p$. The size of the problem instances is comparable to the numbers in recent studies [15]. We implemented, in Matlab, two random bipartite graph generators [2] to create the structure of bipartite graphs and the hypergraphs used in the experiments. These generators are widely used in testing matching [3, 10] and semi-matching algorithms [8]. The generators take a number of parameters and create an instance of the problems at hand. In order to remove statistical bias, we create 10 instances with a given parameter set and report the median of measurements in those 10 random instances for the given parameter set.

### 5.1.1 Bipartite graphs and SingleProc-Unit instances

The *HiLo* generator has been used in the cited resources to create bipartite graphs with $|V_1| = |V_2|$ where the resulting bipartite graph has a unique maximum matching with cardinality $|V_1|$. The associated task-processor bipartite graphs admit therefore a trivial makespan of one. We use this generator to create task-processor graphs with many more tasks than processors, hence possibly having many maximum matchings (with cardinality $|V_2|$). A little precision is necessary to describe the resulting random bipartite graphs resulting from this generator for the case $|V_1| \neq |V_2|$. There are four parameters to the *HiLo* bipartite graph generator: $n$, the number of vertices in $V_1$; $p$, the number of vertices in $V_2$; $g$ the number of groups in which the vertices of $V_1$ and $V_2$ are divided; and $d$, a parameter used in defining the neighbors of a vertex in $V_1$. Let $x_i^j$ be the $i$th vertex in the $j$th vertex group of $V_1$ and $y_k^j$ be the $k$th vertex in the $j$th vertex group of $V_2$. The vertex $x_i^j$ is connected to all vertices $y_k^j$ for $k = \max(1, \min(i, p/g) - d), \ldots, \min(i, p/g)$ and also if $j < g$ to those $y_k^{j+1}$ for

$k = \max(1, \min(i, p/g) - d), \ldots, \min(i, p/g)$. We use *HiLo*$(n, p, g, d)$ to denote a generic instance from this family created according to the four parameters.

The *FewgManyg* generator [2] also has four parameters: $n$, the number of vertices in $V_1$; $p$, the number of vertices in $V_2$; $g$ the number of groups in which the vertices of $V_1$ and $V_2$ are divided; and $d$, the average degree of a vertex in $V_1$. First, the number of neighbors $d_i$ of each vertex $x_i^j \in V_1$ is determined by sampling from a binomial distribution with mean $d$. Then for a vertex $x_i^j \in V_1$, $d_i$ vertices are randomly chosen (without replacement) among the $V_2$ vertices in the $j-1$st to $j+1$st groups with wrap-around. In cases where $d_i$ is bigger than $3p/g$, vertices are chosen with replacement. In the original description of the *FewgManyg* generator, $g = 32$ was used to refer to bipartite graph instances with few groups, and $g = 256$ was used to refer to bipartite graph instances with few groups. We use *FewgManyg*$(n, p, g, d)$ to denote a generic instance from this family created according to the four parameters.

In our study, we use all combinations of $d \in \{2, 5, 10\}$ and $g \in \{32, 128\}$ for the two generators to create instances of the problem SingleProc-Unit. We present detailed results for only $d = 10$, as this choice of $d$ is more common for these generators [2, 8], and give a short summary of the results for other combinations in the appendix.

### 5.1.2 Hypergraphs and MultiProc instances

The hypergraph corresponding to the MultiProc instances can conveniently be represented by two bipartite graphs. The first one represents the connections between $V_1$ and $\mathcal{N}$, the second one represents the connections between $\mathcal{N}$ and $V_2$. By exploiting this fact, we create the instances for the problem MultiProc in two steps using five parameters: $n$, the number of tasks; $p$, the number of processors; $d_v$, the average degree of a task; $d_h$, a parameter used in defining the processor vertices in an hyperedge; and $g$ the number of groups in which the processors and the hyperedges are divided.

In the first step, we choose the degrees of vertices in $V_1$ by random sampling a binomial distribution with mean $d_v$. Since the set of hyperedges of each vertex in $V_1$ are disjoint from the others, the degrees of vertices is enough to form the set of hyperedges. That is, we create $|\mathcal{N}| \approx |V_1| d_v$ hyperedges, each containing a unique vertex from $V_1$. Then, in the second step, given the total number of hyperedges from the first step, we call *HiLo*$(|\mathcal{N}|, p, g, d_h)$ or *FewgManyg*$(|\mathcal{N}|, p, g, d_h)$ to add the processor vertices to each hyperedge.

In our study, we use all combinations of $d_v, d_h \in \{2, 5, 10\}$ and $g \in \{32, 128\}$ for the two generators to create instances of the problem MultiProc. In one set of data created with these parameters, we used unit hyperedge weights, essentially creating instances of MultiProc-Unit. In the second set of experiments, we deterministically assigned the weight $w_h$ to an hyperedge $h$ as follows. Let $s_h = |h \cap V_2|$; then $w_h = \left\lceil \frac{\min_{j \in \mathcal{N}}\{s_j\} \times \max_{j \in \mathcal{N}}\{s_j\}}{s_h} \right\rceil$. Note that this results in almost linear speed up for all configurations represented by hyperedges. In all combinations of $d_v, d_h$, the ranking of the heuristics according to the mean average quality were the same. We present detailed results for only $d_v = 5$ and $d_h = 10$, as this choice of $d_h$ is more common with the use of the generators and $d_v = 5$ seems reasonable with respect to $d_h$. We give a short summary of the results for other combinations of $d_v$ and $d_h$ in the appendix.

Table 1: Random bipartite graph instances constructed by the *FewgManyg* and *HiLo* generators.

| Instance | $\|V_1\|$ | $\|V_2\|$ | $\|E\|$ |
|---|---|---|---|
| MG-5-1-SP | 1280 | 256 | 12446 |
| FG-5-1-SP | 1280 | 256 | 5552 |
| MG-20-1-SP | 5120 | 256 | 49775 |
| FG-20-1-SP | 5120 | 256 | 22165 |
| MG-20-4-SP | 20480 | 256 | 198779 |
| FG-20-4-SP | 20480 | 256 | 88705 |
| MG-80-1-SP | 5120 | 1024 | 51280 |
| FG-80-1-SP | 5120 | 1024 | 49817 |
| MG-80-4-SP | 20480 | 1024 | 204874 |
| FG-80-4-SP | 20480 | 1024 | 198561 |
| MG-80-16-SP | 20480 | 4096 | 204325 |
| FG-80-16-SP | 20480 | 4096 | 204853 |
| HLM-5-1-SP | 1280 | 256 | 18396 |
| HLF-5-1-SP | 1280 | 256 | 4845 |
| HLM-20-1-SP | 5120 | 256 | 78876 |
| HLF-20-1-SP | 5120 | 256 | 20145 |
| HLM-20-4-SP | 20480 | 256 | 320796 |
| HLF-20-4-SP | 20480 | 256 | 81345 |
| HLM-80-1-SP | 5120 | 1024 | 107415 |
| HLF-80-1-SP | 5120 | 1024 | 74460 |
| HLM-80-4-SP | 20480 | 1024 | 440055 |
| HLF-80-4-SP | 20480 | 1024 | 319260 |
| HLM-80-16-SP | 20480 | 4096 | 440055 |
| HLF-80-16-SP | 20480 | 4096 | 434775 |

## 5.2 Experimental results for SingleProc

We now present our experimental results. First, an overview over the test instances is shown in Table 1. Instances from the *FewgManyg* generator are named for number of groups with FG for few groups ($g = 32$) and MG for many groups ($g = 128$). Instances from the *HiLo* generator are named HLF for $g = 32$ and HLM for $g = 128$. The following two numbers denote the number of tasks and the number of processors, both in multiples of 256. Finally, the appended -SP denotes a SingleProc problem. In this table, $|E|$ is the median value of the number of edges in 10 random instances for a given parameter set.

Results are shown in Table 2. The tables report the optimal makespan $M$ computed with the exact algorithm of Section 4.1, and the ratio of each heuristic solution compared to the optimal makespan. The algorithms are those of Section 4.2: *basic-greedy* (BG), *sorted-greedy* (SG) *double-sorted greedy* (DG), and *expected-greedy* (EG).

For the *FewgManyg* generator, *basic-greedy* is the fastest algorithm but offers the lowest quality. *Sorted-greedy* significantly improves upon it while taking only marginally longer running time. The same is true for *double-sorted greedy*, but it offers no benefit in comparison to the standard sorted variant. Finally

Table 2: Performance of the greedy algorithms for the *FewgManyg* and *HiLo* random bipartite graph instances with respect to the optimal value $M$. BG: *basic-greedy*; SG: *sorted-greedy* DG: *double-sorted greedy*; EG: *expected-greedy*.

| Instance | $M$ | BG | SG | DG | EG |
|---|---|---|---|---|---|
| MG-5-1-SP | 5 | 1.20 | 1.20 | 1.20 | 1.20 |
| FG-5-1-SP | 5 | 1.20 | 1.20 | 1.20 | 1.20 |
| MG-20-1-SP | 20 | 1.25 | 1.05 | 1.05 | 1.05 |
| FG-20-1-SP | 20 | 1.25 | 1.15 | 1.15 | 1.05 |
| MG-20-4-SP | 80 | 1.25 | 1.02 | 1.04 | 1.01 |
| FG-20-4-SP | 80 | 1.26 | 1.14 | 1.14 | 1.01 |
| MG-80-1-SP | 5 | 1.40 | 1.20 | 1.20 | 1.20 |
| FG-80-1-SP | 5 | 1.20 | 1.20 | 1.20 | 1.20 |
| MG-80-4-SP | 20 | 1.30 | 1.05 | 1.05 | 1.05 |
| FG-80-4-SP | 20 | 1.25 | 1.05 | 1.05 | 1.05 |
| MG-80-16-SP | 5 | 1.40 | 1.20 | 1.20 | 1.20 |
| FG-80-16-SP | 5 | 1.40 | 1.20 | 1.20 | 1.20 |
| Average quality | 1 | 1.28 | 1.14 | 1.14 | 1.12 |
| Average time (.s) | 2.725 | 0.067 | 0.073 | 0.077 | 1.236 |
| HLM-5-1-SP | 5 | 1.80 | 1.40 | 1.40 | 1.20 |
| HLF-5-1-SP | 5 | 1.80 | 1.40 | 1.40 | 1.20 |
| HLM-20-1-SP | 20 | 1.95 | 1.50 | 1.50 | 1.25 |
| HLF-20-1-SP | 20 | 1.95 | 1.50 | 1.50 | 1.25 |
| HLM-20-4-SP | 80 | 1.99 | 1.50 | 1.50 | 1.25 |
| HLF-20-4-SP | 80 | 1.99 | 1.50 | 1.50 | 1.25 |
| HLM-80-1-SP | 12 | 2.00 | 1.58 | 1.58 | 1.25 |
| HLF-80-1-SP | 5 | 1.80 | 1.40 | 1.40 | 1.20 |
| HLM-80-4-SP | 56 | 2.00 | 1.50 | 1.50 | 1.25 |
| HLF-80-4-SP | 20 | 1.95 | 1.50 | 1.50 | 1.25 |
| HLM-80-16-SP | 47 | 2.00 | 1.51 | 1.51 | 1.26 |
| HLF-80-16-SP | 12 | 2.00 | 1.58 | 1.58 | 1.25 |
| Average quality | 1 | 1.94 | 1.49 | 1.49 | 1.24 |
| Average time (.s) | 6.842 | 0.058 | 0.067 | 0.071 | 1.083 |

we see that *expected-greedy* offers the best approximation but does so at the cost of an immense increase in running time, and its running time is closer to the exact algorithm's running time (which is implemented in C [9]) than to that of *sorted-greedy*. We note however that this is due to Matlab being an interpreted language, which cannot do all optimizations to each code. Thus, we can conclude that in this experiment, *sorted-greedy* offers the best combination of speed and quality with the current implementation. *Expected-greedy* should not be too slow with respect to *sorted-greedy* in an implementation using an imperative language such as C.

For the *HiLo* generator, the overall picture is similar to that of the *Fewg-Manyg* instances (see the lower part of Table 2). The difference in approximation quality is more pronounced here. *Expected-greedy* now offers a significantly better approximation and because values of $M$ are higher, its running time is much

Table 3: Random hypergraph instances constructed by the *FewgManyg* and *HiLo* generators.

| Instance | $\|V_1\|$ | $\|V_2\|$ | $\|\mathcal{N}\|$ | $\sum_{h\in\mathcal{N}} \|h \cap V_2\|$ |
|---|---|---|---|---|
| MG-5-1-MP | 1280 | 256 | 6368 | 61643 |
| FG-5-1-MP | 1280 | 256 | 6400 | 27705 |
| MG-20-1-MP | 5120 | 256 | 25504 | 248683 |
| FG-20-1-MP | 5120 | 256 | 25600 | 110817 |
| MG-20-4-MP | 5120 | 1024 | 25632 | 256459 |
| FG-20-4-MP | 5120 | 1024 | 25728 | 249483 |
| MG-80-1-MP | 20480 | 256 | 102336 | 993764 |
| FG-80-1-MP | 20480 | 256 | 102016 | 441810 |
| MG-80-4-MP | 20480 | 1024 | 102112 | 1021574 |
| FG-80-4-MP | 20480 | 1024 | 101888 | 994256 |
| MG-80-16-MP | 20480 | 4096 | 102176 | 1022141 |
| FG-80-16-MP | 20480 | 4096 | 102144 | 1027001 |
| HLM-5-1-MP | 1280 | 256 | 6368 | 99036 |
| HLF-5-1-MP | 1280 | 256 | 6400 | 25245 |
| HLM-20-1-MP | 5120 | 256 | 25472 | 400428 |
| HLF-20-1-MP | 5120 | 256 | 25600 | 101745 |
| HLM-20-4-MP | 5120 | 1024 | 26016 | 556479 |
| HLF-20-4-MP | 5120 | 1024 | 25600 | 400860 |
| HLM-80-1-MP | 20480 | 256 | 102752 | 1612548 |
| HLF-80-1-MP | 20480 | 256 | 102528 | 407235 |
| HLM-80-4-MP | 20480 | 1024 | 102848 | 2219679 |
| HLF-80-4-MP | 20480 | 1024 | 102656 | 1626900 |
| HLM-80-16-MP | 20480 | 4096 | 102592 | 2218293 |
| HLF-80-16-MP | 20480 | 4096 | 101888 | 2235585 |

faster compared to the exact algorithm. However, *sorted-greedy* still offers the best tradeoff between running time and approximation quality.

We can conclude that *sorted-greedy* is essentially superior to *basic-greedy* and *double-sorted greedy*. Since the problem can be solved exactly in polynomial time, these algorithms are mostly useful in situations where running times is crucial. Therefore, for *expected-greedy* to be useful, it must be implemented in another programming language.

We remind the reader that we have tested with $d \in \{2, 5, 10\}$, and presented only results for $d = 10$. However, the ranking of the heuristics for the SingleProc-Unit problem were always the same as in Table 2, for the two families of the random bipartite graphs.

## 5.3 Experimental results for MultiProc

For MultiProc, we study weighted as well as unweighted instances for both types of random hypergraphs. Here, the appended -MP denotes a MultiProc problem. The instances otherwise follow the same naming conventions as the SingleProc instances. Weighted instances are denoted by an appended -W,

but are otherwise identical to their unweighted counterparts. The instances are listed in Table 3. Instead of indicating the number of edges, we now report the number of hyperedges $|\mathcal{N}|$ and the total number of vertices of $V_2$ contained in the hyperedges $\sum_{h \in \mathcal{N}} |h \cap V_2|$, where these two last values are the median of the ten random instances generated with the given parameter settings.

Table 4: Performance of the greedy algorithms for the unweighted *FewgManyg* and *HiLo* random hypergraphs with respect to the lower bound LB. SGH: *sorted-greedy-hyp*; VGH: *vector-greedy-hyp*; EGH: *expected-greedy-hyp*; EVG: *expected-vector-greedy.*

| Instance | LB | SGH | VGH | EGH | EVG |
|---|---|---|---|---|---|
| MG-5-1-MP | 34 | 1.43 | 1.33 | 1.39 | 1.37 |
| FG-5-1-MP | 17 | 1.43 | 1.32 | 1.43 | 1.38 |
| MG-20-1-MP | 135 | 1.34 | 1.24 | 1.32 | 1.30 |
| FG-20-1-MP | 70 | 1.40 | 1.27 | 1.38 | 1.38 |
| MG-20-4-MP | 34 | 1.41 | 1.30 | 1.39 | 1.37 |
| FG-20-4-MP | 34 | 1.45 | 1.34 | 1.39 | 1.39 |
| MG-80-1-MP | 539 | 1.30 | 1.22 | 1.27 | 1.27 |
| FG-80-1-MP | 280 | 1.39 | 1.26 | 1.37 | 1.36 |
| MG-80-4-MP | 136 | 1.35 | 1.24 | 1.32 | 1.32 |
| FG-80-4-MP | 135 | 1.34 | 1.25 | 1.31 | 1.31 |
| MG-80-16-MP | 34 | 1.42 | 1.30 | 1.39 | 1.39 |
| FG-80-16-MP | 34 | 1.42 | 1.30 | 1.39 | 1.39 |
| Average quality | | 1.39 | 1.28 | 1.36 | 1.35 |
| Average time (.s) | | 0.717 | 5.355 | 0.732 | 9.819 |
| HLM-5-1-MP | 68 | 1.18 | 1.17 | 1.17 | 1.18 |
| HLF-5-1-MP | 19 | 1.12 | 1.12 | 1.12 | 1.12 |
| HLM-20-1-MP | 291 | 1.1 | 1.1 | 1.1 | 1.1 |
| HLF-20-1-MP | 78 | 1.04 | 1.04 | 1.04 | 1.04 |
| HLM-20-4-MP | 99 | 2.84 | 2.84 | 2.84 | 2.84 |
| HLF-20-4-MP | 72 | 1.12 | 1.12 | 1.12 | 1.12 |
| HLM-80-1-MP | 1182 | 1.08 | 1.08 | 1.08 | 1.08 |
| HLF-80-1-MP | 313 | 1.03 | 1.03 | 1.03 | 1.03 |
| HLM-80-4-MP | 405 | 3.06 | 3.06 | 3.06 | 3.06 |
| HLF-80-4-MP | 307 | 1.05 | 1.05 | 1.05 | 1.05 |
| HLM-80-16-MP | 101 | 10.54 | 10.54 | 10.54 | 10.54 |
| HLF-80-16-MP | 105 | 2.7 | 2.69 | 2.69 | 2.69 |
| Average quality | | 2.29 | 2.29 | 2.29 | 2.29 |
| Average time (.s) | | 0.758 | 4.944 | 0.810 | 9.479 |

Since $M$ is infeasible to compute via exact algorithm in this setting, the lower bound (LB) described in Section 4.3 is given for comparison. The algorithms are *sorted-greedy-hyp* (SGH), *vector-greedy-hyp* (VGH), *expected-greedy-hyp* (EGH) for hypergraphs, and *expected-vector-greedy-hyp* (EVG). The variants using lexicographic ordering of load vectors (VGH and EVG) are not implemented using the asymptotically faster algorithms discussed in the end of Secion 4.4.3. The quality of the four greedy algorithms is given as the ratio of the achieved

makespan to the lower bound, where the ratio is taken as the median of the ten random instances. Note that the lower bound is very optimistic and may be far from the optimal solution.

Results for the unweighted instances are reported in Table 4. For the *FewgManyg* generator, we immediately notice that *vector-greedy-hyp* provides better quality than the alternatives, but also takes significantly more time while *sorted-greedy-hyp* and *expected-greedy-hyp* are rather close. Interestingly, *expected vector-greedy-hyp* does not attain the good approximation quality of *vector-greedy-hyp*. For the unweighted *HiLo* instances, we again observe similar running times. However, all algorithms attain the same approximation quality which means that neither the *expected* nor the *vector* strategy work here.

Table 5: Performance of the greedy algorithms for the weighted *FewgManyg* and *HiLo* random hypergraphs with respect to the lower bound LB. SGH: *sorted-greedy-hyp*; VGH: *vector-greedy-hyp*; EGH: *expected-greedy-hyp*; EVG: *expected-vector-greedy*.

| Instance | LB | SGH | VGH | EGH | EVG |
|---|---|---|---|---|---|
| MG-5-1-MP-W | 87 | 1.34 | 1.3 | 1.27 | 1.25 |
| FG-5-1-MP-W | 26 | 1.63 | 1.59 | 1.51 | 1.32 |
| MG-20-1-MP-W | 335 | 1.25 | 1.24 | 1.19 | 1.19 |
| FG-20-1-MP-W | 103 | 1.55 | 1.55 | 1.43 | 1.28 |
| MG-20-4-MP-W | 123 | 1.35 | 1.35 | 1.26 | 1.17 |
| FG-20-4-MP-W | 84 | 1.41 | 1.36 | 1.31 | 1.26 |
| MG-80-1-MP-W | 1406 | 1.19 | 1.18 | 1.15 | 1.15 |
| FG-80-1-MP-W | 413 | 1.54 | 1.54 | 1.43 | 1.27 |
| MG-80-4-MP-W | 549 | 1.24 | 1.24 | 1.12 | 1.11 |
| FG-80-4-MP-W | 381 | 1.22 | 1.21 | 1.17 | 1.15 |
| MG-80-16-MP-W | 141 | 1.36 | 1.35 | 1.24 | 1.17 |
| FG-80-16-MP-W | 141 | 1.35 | 1.37 | 1.29 | 1.17 |
| Average quality | | 1.37 | 1.36 | 1.28 | 1.21 |
| Average time (.s) | | 0.717 | 6.213 | 0.730 | 9.816 |
| HLM-5-1-MP-W | 80 | 1.25 | 1.24 | 1.12 | 1.02 |
| HLF-5-1-MP-W | 20 | 1.15 | 1.15 | 1.05 | 1.05 |
| HLM-20-1-MP-W | 320 | 1.17 | 1.17 | 1.05 | 1.02 |
| HLF-20-1-MP-W | 80 | 1.06 | 1.06 | 1.03 | 1.01 |
| HLM-20-4-MP-W | 110 | 2.93 | 2.93 | 2.61 | 2.60 |
| HLF-20-4-MP-W | 80 | 1.18 | 1.18 | 1.16 | 1.02 |
| HLM-80-1-MP-W | 1280 | 1.15 | 1.15 | 1.03 | 1.02 |
| HLF-80-1-MP-W | 320 | 1.04 | 1.04 | 1.01 | 1.01 |
| HLM-80-4-MP-W | 440 | 3.22 | 3.23 | 2.87 | 2.86 |
| HLF-80-4-MP-W | 320 | 1.07 | 1.06 | 1.03 | 1.01 |
| HLM-80-16-MP-W | 110 | 11.07 | 11.06 | 9.89 | 9.85 |
| HLF-80-16-MP-W | 110 | 2.66 | 2.66 | 2.57 | 2.57 |
| Average quality | | 2.41 | 2.41 | 2.20 | 2.17 |
| Average time (.s) | | 0.733 | 4.921 | 0.780 | 9.134 |

The weighted results are then reported in Table 5. Interestingly, the weighted

*FewgManyg* results show a very different picture than the unweighted ones. Running times remain similar, but here the *expected-greedy-hyp* algorithm shows much better quality, while *vector-greedy-hyp* cannot improve upon *sorted-greedy-hyp*. Interestingly, *expected-vector-greedy-hyp* does improve upon the quality of *expected-greedy-hyp*, although at a steep cost in running time. For the weighted *HiLo* instances, similarly to the unweighted *HiLo* case, *vector-greedy-hyp* is at the same level as *sorted-greedy-hyp*, while the *expected* greedy algorithms show better approximation. This is consistent with their behavior in the weighted *FewgManyg* case.

From the above observation, we can conclude that the *expected* strategy is helpful in weighted instances. *Expected-greedy-hyp* showed better quality than *sorted-greedy-hyp* at the cost of only marginally higher running time. On the other hand, *vector-greedy-hyp* performed better only for unweighted *FewgManyg* and was significantly faster. Thus, the usefulness of the *vector* strategy is somewhat limited, although considering that these problems are NP-complete, using it to improve upon the quality of *expected-greedy-hyp* with *expected-vector-greedy-hyp* might still be worthwhile in order to obtain the best performance.

We remind the reader that we have tested with all combinations of $d_v, d_h \in \{2, 5, 10\}$. The ranking of the heuristics for the MultiProc-Unit and MultiProc problem were always the same as in the Tables 4 and 5, for the two families of the random hypergraphs.

# 6 Conclusion

We have studied the problem of scheduling parallel tasks under resource constraints to minimize the makespan. We have formulated the problem in terms of bipartite graphs and hypergraphs, and shown that the scheduling problem amounts to finding semi-matchings in the corresponding graph theoretical formulation. In the case of hypergraphs (i.e., parallel tasks), we have proved that the problem is NP-complete and that for all $\epsilon > 0$, there is no $(2 - \epsilon)$-approximation algorithm unless P=NP.

For the simplest problem instance corresponding to semi-matchings in unweighted bipartite graphs, we have designed several linear time greedy algorithms, and from the simulation results, it turns out that performing a simple sort on the out-degree of the tasks (*sorted-greedy* algorithm) is very efficient and the execution is much faster than for the optimal algorithm. In addition, *expected-greedy*, which incorporates expected loads of processors, is shown to be more effective, albeit with an increase in the running time, in the current test platform.

We have extended the heuristics proposed for the bipartite graphs to the general case of weighted hypergraphs. While the adaptation of *sorted-greedy* still performs quite well in this case, the one with the load prediction technique (*expected-greedy-hyp*), obtains better results at the price of only a small increase in the execution time. We have also introduced two new heuristics based on a lexicographic ordering of load vectors instead of just minimizing the maximum load at each step, and the obtained solution is then shown to be even better.

As future work, we plan to further investigate the hypergraph problem, and to design new algorithms with guarantees. Indeed, even if the greedy algorithms perform quite well, their solution may be arbitrarily far from the optimal. There-

fore, it seems challenging to obtain approximation algorithms for this problem. We also plan to implement the proposed heuristics in an imperative programming language to perform further tests.

## Acknowledgment

A. Benoit is with the Institut Universitaire de France. This work was supported in part by the ANR *Rescue* project.

# A Further experimental results

## A.1 SingleProc-Unit

In Section 5.2, we have presented some detailed results for the SingleProc-Unit problem. We have seen results with random bipartite graphs created using generators *FewgManyg*$(n, p, g, d)$ and *HiLo*$(n, p, g, d)$, where $n$, the number of tasks, is in $\{1280, 5120, 20480\}$; $p$, the number of processors, is in $256, 1024, 4096$ (we did not test the cases where $n < 5 \times p$); $d = 10$; and the number of groups $g \in \{32, 128\}$. In Table 6, we give average quality results for the heuristics with $d \in \{2, 5, 10\}$. In this table, a row shows the average performance of the heuristic with respect to the optimal value. In a sense each row corresponds to the row "Average quality" given in Table 2, which are repeated here for convenience. As seen from Table 6, whereas the quality of each heuristic changes with respect to the optimal value, the ranking of them remains the same: *expected-greedy* is the best, *basic-greedy* is the worst, and the two sorted variants are in the middle, without any difference between them.

Table 6: Performance of the greedy algorithms for the SingleProc-Unit problem with *FewgManyg* and *HiLo* random bipartite graph instances, reported as average quality with respect to the optimal value for the random graph families *FewgManyg*$(\cdot, \cdot, \cdot, d)$ and *HiLo*$(\cdot, \cdot, \cdot, d)$ for differing $d$. BG: *basic-greedy*; SG: *sorted-greedy* DG: *double-sorted greedy*; EG: *expected-greedy*.

| | BG | SG | DG | EG |
|---|---|---|---|---|
| $d$ | *FewgManyg* instances | | | |
| 2 | 1.31 | 1.02 | 1.02 | 1.02 |
| 5 | 1.32 | 1.24 | 1.24 | 1.12 |
| 10 | 1.28 | 1.14 | 1.14 | 1.12 |
| $d$ | *HiLo* instances | | | |
| 2 | 1.98 | 1.50 | 1.50 | 1.25 |
| 5 | 1.98 | 1.49 | 1.49 | 1.23 |
| 10 | 1.94 | 1.49 | 1.49 | 1.24 |

## A.2 MultiProc

In Section 5.3, we have presented some detailed results for the MultiProc-Unit and MultiProc problems. We have seen results with random hypergraphs created using generators *FewgManyg*$(|\mathcal{N}|, p, g, d_h)$ and *HiLo*$(|\mathcal{N}|, p, g, d_h)$, where $|\mathcal{N}|$ is the number of hyperedges determined according to a random sampling of the degree of task vertices, $n$, under a binomial distribution with mean $d_v$. In the presented detailed experiments (see Tables 4 and 5), we had $d_v = 5$ and $d_h = 10$. In Table 7, we give average quality results for the heuristics with $d_v, d_h \in \{2, 5, 10\}$. In this table, a row shows the average performance of the heuristic with respect to the lower bound. In a sense each row corresponds to the row "Average quality" given in Tables 4 and 5, which are repeated here for convenience. As seen in Table 7, the average quality of the heuristics changes, but their ranking remain the same in different problems and families.

Table 7: Performance of the greedy algorithms for the MultiProc-Unit (on the left) and MultiProc (on the right) problems with *FewgManyg* and *HiLo* random hypergraph instances, reported as average quality with respect to the lower bound value for differing $d_v$ and $d_h$. SGH: *sorted-greedy-hyp*; VGH: *vector-greedy-hyp*; EGH: *expected-greedy-hyp*; EVG: *expected-vector-greedy*.

| | | MultiProc-Unit problem | | | | MultiProc problem | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SGH | VGH | EGH | EVG | SGH | VGH | EGH | EVG |
| $d_v$ | $d_h$ | *FewgManyg* instances | | | | *FewgManyg* instances | | | |
| 2 | 2 | 1.39 | 1.37 | 1.40 | 1.40 | 1.29 | 1.30 | 1.25 | 1.25 |
| 2 | 5 | 1.28 | 1.25 | 1.27 | 1.27 | 1.29 | 1.28 | 1.22 | 1.16 |
| 2 | 10 | 1.31 | 1.27 | 1.30 | 1.30 | 1.32 | 1.30 | 1.29 | 1.25 |
| 5 | 2 | 1.47 | 1.26 | 1.40 | 1.40 | 1.43 | 1.43 | 1.13 | 1.13 |
| 5 | 5 | 1.49 | 1.30 | 1.43 | 1.43 | 1.39 | 1.39 | 1.19 | 1.14 |
| 5 | 10 | 1.39 | 1.28 | 1.36 | 1.35 | 1.37 | 1.36 | 1.28 | 1.21 |
| 10 | 2 | 1.51 | 1.16 | 1.34 | 1.33 | 1.56 | 1.55 | 1.12 | 1.16 |
| 10 | 5 | 1.74 | 1.37 | 1.61 | 1.61 | 1.47 | 1.47 | 1.19 | 1.15 |
| 10 | 10 | 1.55 | 1.32 | 1.48 | 1.48 | 1.45 | 1.44 | 1.31 | 1.21 |
| $d_v$ | $d_h$ | *HiLo* instances | | | | *HiLo* instances | | | |
| 2 | 2 | 6.78 | 6.78 | 6.78 | 6.78 | 6.92 | 6.91 | 6.68 | 6.66 |
| 2 | 5 | 3.59 | 3.59 | 3.59 | 3.59 | 3.65 | 3.65 | 3.52 | 3.51 |
| 2 | 10 | 2.25 | 2.25 | 2.24 | 2.24 | 2.30 | 2.30 | 2.23 | 2.19 |
| 5 | 2 | 7.03 | 7.02 | 7.02 | 7.02 | 7.36 | 7.36 | 6.67 | 6.65 |
| 5 | 5 | 3.72 | 3.72 | 3.71 | 3.71 | 3.87 | 3.87 | 3.52 | 3.50 |
| 5 | 10 | 2.29 | 2.29 | 2.29 | 2.29 | 2.41 | 2.41 | 2.20 | 2.17 |
| 10 | 2 | 7.45 | 7.45 | 7.45 | 7.45 | 8.05 | 8.05 | 6.67 | 6.65 |
| 10 | 5 | 3.93 | 3.93 | 3.93 | 3.93 | 4.22 | 4.22 | 3.51 | 3.50 |
| 10 | 10 | 2.45 | 2.45 | 2.45 | 2.45 | 2.61 | 2.61 | 2.19 | 2.17 |

# Bibliography

[1] R. H. Ahmadi and U. Bagchi. Scheduling of multi-job customer orders in multi-machine environments. *ORSA/TIMS*, 1990.

[2] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *Journal of Experimental Algorithmics*, 3:8, 1998.

[3] I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38:13:1–13:31, 2011.

[4] J. Fakcharoenphol, B. Laekhanukit, and D. Nanongkai. Faster algorithms for semi-matching problems. *CoRR*, abs/1004.3363, 2010.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[6] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the Association for Computing Machinery*, 35:921–940, 1988.

[7] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[8] N. J. A. Harvey, R. E. Ladner, L. Lovász, and T. Tamir. Semi-matchings for bipartite graphs and load balancing. *Journal Algorithms*, 59(1):53–78, 2006.

[9] K. Kaya, J. Langguth, F. Manne, and B. Uçar. Experiments on push-relabel-based maximum cardinality matching algorithms for bipartite graphs. Technical Report TR/PA/11/33, CERFACS, Toulouse, France, 2011.

[10] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *Journal of Experimental Algorithmics*, 15:1.1–1.22, 2010.

[11] J. Y.-T. Leung, H. Li, M. Pinedo, and C. Sriskandarajah. Open shops with jobs overlap—revisited. *European Journal of Operational Research*, 163(2): 569–571, 2005. ISSN 0377-2217.

[12] J. Y.-T. Leung, H. Li, and M. Pinedo. Scheduling orders for multiple product types to minimize total weighted completion time. *Discrete Applied Mathematics*, 155(8):945–970, 2007.

[13] C. P. Low. An approximation algorithm for the load-balanced semi-matching problem in weighted bipartite graphs. *Information Processing Letters*, 100(4):154–161, 2006.

[14] T. Roemer. A note on the complexity of the concurrent open shop problem. *Journal of Scheduling*, 9:389–396, 2006.

[15] E. Saule, D. Bozdağ, and Ü. V. Çatalyürek. Optimizing the stretch of independent tasks on a cluster: From sequential tasks to moldable tasks. *Journal of Parallel and Distributed Computing*, 72(4):489–503, 2012.

[16] E. V. Shchepin and N. Vakhania. Task distributions on multiprocessor systems. In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, TCS '00, pages 112–125, London, UK, 2000. Springer-Verlag.

[17] E. Wagneur and C. Sriskandarajah. Open shops with jobs overlap. *European Journal of Operational Research*, 71:366–378, 1993.

**RESEARCH CENTRE**
**GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399