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. IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, May 2013, Cambridge, MA, United States. pp.1744-1753, 10.1109/IPDPSW.2013.30. hal-00738393v5

HAL Id: hal-00738393
https://inria.hal.science/hal-00738393v5
Submitted on 2 Jan 2014

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  
LIP, ENS Lyon, France  
UMR CNRS-ENS-INRIA-UCBL  
46, allée d’Italie, Lyon F-69364  
Email:anne.benoit@ens-lyon.fr

Johannes Langguth  
Simula Research Laboratory,  
Fornebu, Norway  
Email:langguth@simula.no

Bora Uçar  
CNRS and LIP, ENS Lyon, France  
UMR CNRS-ENS-INRIA-UCBL  
46, allée d’Italie, Lyon F-69364  
Email:bora.ucar@ens-lyon.fr

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. The problem is known to be NP-complete for bipartite graphs with general vertex (task) weights, and solvable in polynomial time for unweighted graphs (i.e., unit-weight 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 for these low-complexity algorithms, they return solutions close to the optimal (or a known lower bound) in average.

Keywords-semi-matching, bipartite graphs, hypergraphs, scheduling, parallel tasks, resource constraints.

I. INTRODUCTION

The Minimum Makespan Scheduling Problem on Unrelated Machines is a classical topic in scheduling [13]. 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, each part being computed independently on a different processor; (ii) tasks are subject to resource constraints, i.e., several configurations of processors, leading to different execution times, are possible for each task. 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 [25], the problem was introduced by Ahmadi and Bagchi [1]. It has been studied widely [20], [21], [28] and was proved to be strongly NP-complete [25]. 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. In this setting, the overall makespan is the 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 considered before by the graph theory community.

We also consider a simplified variant of the problem, SingleProc. Here, each task is sequential, and it can be scheduled on a single processor among a set of possible ones (with 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., the problem SingleProc-Unit with unit-weight tasks: several polynomial-time algorithms were proposed [10], [14], [22], [23]. The weighted version is NP-complete [24]. Our intention in studying SingleProc at the beginning was to gauge the heuristics for MultiProc in a...
simplified setting by comparing their results with an exact solution for SINGLEPROC-UNIT.

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

1) For the SINGLEPROC-UNIT problem, we propose an exact polynomial-time algorithm that is efficient in practice and conceptually very simple. We describe heuristics for this problem that will then be extended for MULTIProc. We examine those heuristics theoretically (we build examples in which the heuristics are as far as possible from the optimal), and practically (by comparing them to the optimal solution). Practical experiments are summarized to validate the interest of the ideas used in these heuristics for the MULTIProc problem.

2) The core of the paper resides in the study of the MULTIProc problem. First, we prove that it is NP-complete even in the unweighted case (while SINGLEPROC-UNIT was solvable in polynomial time). Moreover, we show that for all $\epsilon > 0$, there is no $(2 - \epsilon)$-approximation algorithm unless P=NP. We design a set of heuristics, building upon the SINGLEPROC-UNIT heuristics. We describe a lower bound to assess the performance of the heuristics for MULTIProc, and present exhaustive simulation results.

The remainder of this paper is organized as follows. We start with a formal description of the optimization problems and a summary of some related work in Section II. We then prove in Section III that the general problem MULTIProc is NP-complete even with unit-weight tasks. We design an optimal algorithm for SINGLEPROC-UNIT and several greedy algorithms of low complexity for the two variants of the problem in Section IV, and then assess the performance of these heuristics in Section V. Finally, we conclude in Section VI with a summary and plans on further investigations of the addressed problems.

II. 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 $S_i$ 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, $S_1 = \{\{P_1\}, \{P_2, P_3\}\}$. Task $T_i$ is executed on a set of processors $\text{alloc}(i) \in S_i$, and it takes a time $w^\text{alloc}(i)$ on each of the processors $P_u \in \text{alloc}(i)$. The processing can be done at different time steps on the processors of $\text{alloc}(i)$, 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 $\text{alloc}(i)$ 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:

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

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.

A. With a single processor

In some cases, tasks cannot be executed in parallel, and $S_i$ is just a set of processors on which $T_i$ can be executed, instead of a set of sets. This problem is called SINGLEPROC. This 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 $S_i$ (see Fig. 1, where $S_1 = \{P_1, P_2\}$ and $S_2 = \{P_1\}$). We use $d_e$ 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: $w(e) = w(T_i, P_u) = w^{P_u}_{T_i}$.

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

This SINGLEPROC problem was shown to be NP-complete [24] (see also a related note [17]) 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. [13]
which was subsequently improved to $2 - \frac{1}{p}$ by Shchepin and Vakhania [27].

We consider also the unweighted version, that amounts to having $w^P_h = 1$ for $1 \leq i \leq n$ and $P_u \in S_i$. 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 [14]. In Section IV, we design several heuristics of low complexity 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.

B. With multiple processors

The general problem is called MULTIPROC, and $S_i$ 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 $H = (V, \mathcal{N})$ 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 $H = (V, \mathcal{N})$ 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.

![Fig. 2. A sample hypergraph for MULTIPROC.](image)

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 $\text{alloc}(i) = h_i \cap V_2$. 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 $w^P_h = w^{V_2 \cap V_1}_h$ 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 polynomial time, MULTIPROC-UNIT turns out to be NP-complete (see Section III).

C. Some related work

The problem SINGLEPROC is well known and widely studied (see a recent survey [19], the references therein, and a more recent study [18] dealing with online and offline algorithms for different processor set restrictions).

The problem MULTIPROC is a variant of scheduling with processing set restrictions [19] which is referred to as $P|\text{set}_{ij}|C_{\text{max}}$ by Drozdzowski [8]. In heterogeneous computer systems, restrictions apply to task to processing unit assignments, as not all processors could be capable of meeting the task requirements. Furthermore, each processing unit could be a set of (sometimes hierarchical [19, p.260]) machines that collectively execute the job.

A more immediate problem for MULTIPROC is also known as scheduling independent tasks with multiple modes. In this setting, each task has a possible set of modes, where each mode is a set of resources simultaneously required by the task for a period of time [5]. The aim is to find a non-preemptive schedule of task executions in the decided modes so that the resource requirements at a time period do not exceed the available ones, and the makespan is minimized. A set of reasonable heuristics for this problem proceeds in two steps (see the discussions in a recent paper [6]), where the first step’s goal can be formulated as MULTIPROC problem. In other words, the MULTIPROC problem arises as a subproblem in multimode scheduling problems.

The two problems are also related to scheduling problems with setup times/costs, where the setup cost occurs before a job is processed at a machine [2]. Our formulation in MULTIPROC asks for the minimization of the maximum setup cost across machines, where the tasks are splittable (Aubry et al. [3] discuss such a setting and propose a mixed integer linear program formulation for minimizing the total setup cost under load balance constraints with preemption) and the setup costs are sequence-independent [2].

III. NP-COMPLETENESS OF MULTIPROC-UNIT

It was shown that SINGLEPROC is NP-complete [24], while SINGLEPROC-UNIT is solvable in polynomial time [14]. 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 polynomial time, given an assignment of tasks to processors.
To establish the completeness, we use a reduction from Exact Cover by 3-Sets (X3C) [11, p. 53]. We consider an instance \( I_1 \) 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 \( I_2 \) of MultiProc-Unit: the set of elements of \( I_2 \) 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., \( S_i = C \) for \( 1 \leq i \leq n \). Moreover, we set the deadline \( D = 1 \).

Clearly, the size of \( I_2 \) is polynomial in the size of \( I_1 \). We show that \( I_1 \) has a solution if and only if \( I_2 \) does.

If \( I_1 \) 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 \( I_2 \) has a solution. Suppose now that \( I_2 \) 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 \( S_i \), the allocation of \( I_2 \) forms a cover for \( I_1 \), 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 \( I_2 \), hence obtaining a makespan \( M \leq (2 - \epsilon)M_{\text{opt}} \), where \( M_{\text{opt}} \) is the optimal makespan. Since \( M_{\text{opt}} = 1 \), 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 \( \mathsf{P} = \mathsf{NP} \).

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

IV. ALGORITHMS

The organization of this section is from the simpler problem SingleProc-Unit to our main problem MultiProc. We propose an exact polynomial-time algorithm for SingleProc-Unit first. We then discuss heuristics for SingleProc-Unit as they are simplified versions of the proposed heuristics for MultiProc. We also discuss a lower bound for MultiProc. The exact algorithm for SingleProc-Unit and the lower bound for MultiProc are developed to obtain baseline values for evaluating the proposed heuristics.

A. Exact algorithm for SingleProc-Unit

As shown by Harvey et al. [14], SingleProc-Unit can be solved using some modified versions of the standard matching algorithms in \( O(|V_1| |E|) \) time. We propose a conceptually simpler algorithm by making use of the standard matching algorithms as a black box (see a relatively recent survey [9] on augmenting-path based ones, and two other studies [12], [15] 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 \( O(\sqrt{|V_1| |E|}) \), and there are \( M_{\text{opt}} \) steps, where \( M_{\text{opt}} \) is the optimal makespan. Hence the algorithm has a running time complexity of \( O(\sqrt{|V_1| |E| M_{\text{opt}}}) \). Note that \( M_{\text{opt}} \leq |V_1| \), since the worst case is when all tasks are mapped on the same machine. Note that a bisection search for optimal makespan would yield improved worst-case running time complexity (in practice the algorithm is very fast, see the accompanying technical report [4]).

B. Greedy algorithms for SingleProc-Unit

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.

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 optimum solution, for any \( k \) (this is an example also showing that basic-greedy can be arbitrarily far from the optimal).

---

**Algorithm 1 basic-greedy**

**Input:** A bipartite graph \( G = (V_1 \cup V_2, E) \)

**Output:** A matching \( M \) 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: \( M \leftarrow M + e \)
4: \( l(u) \leftarrow l(u) + 1 \)
5: return \( M \)
Consider that there are $2^k - 1$ tasks to be mapped onto $2^k$ processors. For the ease of reading, tasks are named $T_i^{(k)}$, with $0 \leq \ell \leq k - 1$, and $1 \leq i \leq 2^{k-1-\ell}$. Task $T_i^{(k)}$ 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 $T_i^{(k)}$ 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 $T_i^{(k-1)}$ is also mapped on $P_1$, and processor $P_1$ achieves a makespan of $k$.

![Fig. 3. Example where basic-greedy and sorted-greedy obtain a makespan of $k = 3$, while the optimal makespan is 1.](image)

**Algorithm 2 double-sorted**

**Input:** A bipartite graph $G = (V_1 \cup V_2, E)$

**Output:** A matching $\mathcal{M}$ in $G$

1: for all $v \in V_1$, sorted by non-decreasing out-degree do
2: min$\leftarrow n$
3: mind$\leftarrow n$
4: for all $e = \{v, u\} \in E$ do
5: if $l(u) < \text{minl}$ or ($l(u) = \text{minl}$ and $d_u \leq \text{mind}$) then
6: min$\leftarrow l(u)$
7: mind$\leftarrow d_u$
8: mine$\leftarrow e$
9: $\mathcal{M} \leftarrow \mathcal{M} + \text{mine}$
10: $l(u) \leftarrow l(u) + 1$
11: return $\mathcal{M}$

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 as follows: we add a task $T_k$, which can be mapped on $P_3$ or $P_4$, hence the first eight tasks have an out-degree 2 and the first four processors have an in-degree 3. Then we add four tasks of out-degree 3 (that will therefore be considered last), and four processors of in-degree 1, so that $P_3$ to $P_k$ also have an in-degree 3, and tasks $T_0$ to $T_{12}$ can be mapped respectively on $P_0$ to $P_{12}$ (see [4, Figure 4]). Therefore, since processors $P_1$ to $P_{k}$ 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).

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 described for double-sorted, the values of $o(u)$ differ since tasks $T_0$ to $T_{12}$ are of degree 3, while the others are of degree 2. Therefore, expected-greedy places $T_i^{(0)}$ on $P_0$, as in the optimal solution, and reaches the optimal makespan of 1.

**Algorithm 3 expected-greedy**

**Input:** A bipartite graph $G = (V_1 \cup V_2, E)$

**Output:** A matching $\mathcal{M}$ in $G$

1: for all $u \in V_2$ do
2: $o(u) \leftarrow 0$
3: for all $v \in V_1$ do
4: for all $\{v, u\} \in E$ do
5: $o(u) \leftarrow o(u) + 1/d_v$
6: for all $v \in V_1$, sorted by non-decreasing out-degree do
7: find an edge $e = \{v, u\} \in E$ for which $o(u)$ is minimum
8: $\mathcal{M} \leftarrow \mathcal{M} + e$
9: $o(u) \leftarrow o(u) + 1 - 1/d_v$
10: for all $\{v, u\} \in E \setminus \{e\}$ do
11: $o(u) \leftarrow o(u) - 1/d_v$
12: return $\mathcal{M}$
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 out-degree 2, see [4, Figure 5]. Tasks $T_9$ to $T_{16}$ can be assigned either to their own processor (of in-degree 1), or to one of processors $P_1$ to $P_8$, so that these processors have each an in-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 these worst cases are however extremely unlikely in practical scenarios. 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 V.

C. Lower bound for MULTIProc

Since we cannot compute the optimal solution for MULTIProc as in the SINGLEProc-UNIT case, we derive a lower bound so that we will be able to assess the performance of the heuristics, later in Section V. 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:

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

D. 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).

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 $l(u)$.

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.

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 $w_{h,v}/d_v$ to all vertices of $V_2$ contained in $h$, where $w_{h,v}$ 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.

Similarly to sorted-greedy-hyp, this algorithm has a running time of $O(\sum_{h \in \mathcal{N}} |h|)$. Due to the updates of $o$, its running time will generally be in $O(\sum_{h \in \mathcal{N}} |h|)$ 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.

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 cases, we can favor the hyperedge that has the smaller value, e.g., $\max_{u \in V_2} l(u)$.

\begin{algorithm}[H]
\caption{sorted-greedy-hyp}
\textbf{Input:} A hypergraph $\mathcal{H}=(V_1,V_2,\mathcal{N})$
\textbf{Output:} A matching $\mathcal{M}$
\begin{algorithmic}[1]
\State \textbf{for all} $v \in V_1$, sorted by non-decreasing out-degree \textbf{do}
\State \quad find a hyperedge $h : v \in h$ for which $\max_{u \in h} l(u)$ is minimum.
\State \textbf{end for}
\State $\mathcal{M} \leftarrow \mathcal{M} + h$
\State \textbf{return} $\mathcal{M}$
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[H]
\caption{expected-greedy-hyp}
\textbf{Input:} A hypergraph $\mathcal{H}=(V_1,V_2,\mathcal{N})$
\textbf{Output:} A matching $\mathcal{M}$
\begin{algorithmic}[1]
\State \textbf{for all} $u \in V_2$ \textbf{do}
\State \quad $o(u) \leftarrow 0$
\State \textbf{end for}
\State \textbf{for all} $v \in V_1$ \textbf{do}
\State \quad $o(v) \leftarrow 0$
\State \textbf{end for}
\State \textbf{for all} $h \in \mathcal{N} : v \in h$ \textbf{do}
\State \quad $o(u) \leftarrow o(u) + w_{h,v}/d_v$
\State \textbf{end for}
\State \textbf{for all} $v \in V_1$, sorted by non-decreasing out-degree \textbf{do}
\State \quad find a hyperedge $h : v \in h$ for which $\max_{u \in h} o(u)$ is minimum.
\State \textbf{end for}
\State $\mathcal{M} \leftarrow \mathcal{M} + h$
\State \textbf{for all} $u \in V_2 : u \in h$ \textbf{do}
\State \quad $o(u) \leftarrow o(u) + w_{h,v}/d_v$
\State \textbf{end for}
\State \textbf{for all} $h' \in \mathcal{N} \setminus \{h\} : v \in h'$ \textbf{do}
\State \quad $o(u) \leftarrow o(u) - w_{h,v}/d_v$
\State \textbf{end for}
\State \textbf{for all} $u \in V_2 : u \in h'$ \textbf{do}
\State \textbf{return} $\mathcal{M}$
\end{algorithmic}
\end{algorithm}
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 is \( O \left( \sum_{v \in V_1} d_v |V_2| \log |V_2| + \sum_{h \in \mathcal{N}} |h| \right) \). 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) 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 discarded so that the effect of each hyperedge can be measured. For a vertex with \( d_v \) hyperedges, this requires \( O(d_v \sum_{v \in \mathcal{N}} |h|) \) operations. The overall complexity of the algorithm with the list representation would be \( O \left( \sum_{v \in V_1} d_v |V_2| + \sum_{v} d_v \sum_{v \in \mathcal{N}} |h| \right) \). 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.

V. 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 8GB of 1333 MHz DDR3 ram. We used an implementation of the push-relabel algorithm [15] provided in MatchMaker suite [9]. 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 summarize experimental results for SingleProc-Unit and detail those for MultiProc.

A. 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 simulation studies [26]. We implemented, in Matlab, two random bipartite graph generators [7] to create the structure of bipartite graphs and the hypergraphs used in the experiments. These generators are widely used in testing matching [9], [16] and semi-matching algorithms [14]. 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.

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 neighborhoods 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 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 \( j \)-1st 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 [7], [14] (the results for other combinations are in our technical report [4, Appendix]).

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 \( |N| \approx |V_1|d_v \) 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\((|N|, p, g, d_h)\) or FewgManyh\((|N|, p, g, d_h)\) 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 MULTI\textsc{Proc}. In one set of data created with these parameters, we used unit hyperedge weights, essentially creating instances of MULTI\textsc{Proc-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 \( w_h = \left\lfloor \min_{s \in N} (s) \times \max_{s \in N} (s) \right\rfloor \). These weights are related: if a task is assigned to more processors, its computation time gets smaller (as would be the case in most realistic settings). Note that the problem with related weights is also NP-complete, since all hyperedges have the same degree in the proof of Theorem 1. In the third data set, weights are random so that we can double check the results; those experiments are reported in the accompanying technical report [4]. 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 our technical report [4, Appendix].

B. Experimental results for SINGLE\textsc{Proc}

We provide only a summary of the experimental results, since the goal of this section is to assess the validity of the concepts used in the MULTI\textsc{Proc} heuristics. Detailed results can be found in the accompanying technical report [4].

For the FewgManyh 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, it is observed 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 [15]) 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 quality and speed with the current implementation. Expected-greedy should not be too slow with respect to sorted-greedy in an implementation using an imperative language. Hence it is also as preferable as the sorted-greedy.

For the HiLo generator, the overall picture is similar to that of the FewgManyh instances. 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 faster compared to the exact algorithm. If the running time is of significant importance, sorted-greedy can be considered as the method that offers the best tradeoff between the quality and running time.

We can conclude that sorted-greedy is essentially superior to basic-greedy and double-sorted greedy and that expected-greedy is the best. The ranking of the heuristics for the SINGLE\textsc{Proc-Unit} problem were always the same, for the two families of the random bipartite graphs with all \( d \in \{2, 5, 10\} \) [4].

C. Experimental results for MULTI\textsc{Proc}

For MULTI\textsc{Proc}, we study weighted as well as unweighted instances for both types of random hypergraphs. Here, the appended -MP denotes a MULTI\textsc{Proc} problem. The instances otherwise follow the same naming conventions as the SINGLE\textsc{Proc} instances. Weighted instances are denoted by an appended -W, but are otherwise identical to their unweighted counterparts. The instances are listed in Table I. Instead of indicating the number of edges, we now report the number of hyperedges \(|N|\) and the total number of vertices of \( V_2 \) contained in the hyperedges \( \sum_{h \in N} |h \cap V_2| \), where these two last values are the median of the ten random instances generated with the given parameter settings.

Since \( M \) is infeasible to compute via exact algorithm in this setting, the lower bound LB (1) described in Section IV-C 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 Section IV-D3. 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 II. For the FewgManyh 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.
The weighted results are then reported in Table III. 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 vector strategy is useful in weighted instances and is of limited use in unweighted instances. Using it to improve upon the quality of expected-greedy-hyp with expected-vector-greedy-hyp is worthwhile in order to obtain the best performance (see also the experiments with random hyperedge weights [4, Table 8], where expected-vector-greedy-hyp gives much better results). The ranking of the heuristics for the MultiProcUnit and MultiProc problem were always the same as in Tables II and III, for the two families of the random hypergraphs with other combinations of \( d_v, d_h \in \{2, 5, 10\} \), and with random weights instead of related weights [4].

### Table I

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

### Table II

<table>
<thead>
<tr>
<th>Instance</th>
<th>LB</th>
<th>SGH</th>
<th>VGH</th>
<th>EGH</th>
<th>EVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG-5-1-MP</td>
<td>34</td>
<td>1.43</td>
<td>1.33</td>
<td>1.39</td>
<td>1.37</td>
</tr>
<tr>
<td>MG-5-1-MP</td>
<td>17</td>
<td>1.43</td>
<td>1.32</td>
<td>1.43</td>
<td>1.38</td>
</tr>
<tr>
<td>FG-20-1-MP</td>
<td>135</td>
<td>1.34</td>
<td>1.24</td>
<td>1.32</td>
<td>1.30</td>
</tr>
<tr>
<td>MG-20-1-MP</td>
<td>70</td>
<td>1.40</td>
<td>1.27</td>
<td>1.38</td>
<td>1.38</td>
</tr>
<tr>
<td>FG-20-4-MP</td>
<td>34</td>
<td>1.41</td>
<td>1.30</td>
<td>1.39</td>
<td>1.37</td>
</tr>
<tr>
<td>MG-20-4-MP</td>
<td>34</td>
<td>1.45</td>
<td>1.34</td>
<td>1.39</td>
<td>1.39</td>
</tr>
<tr>
<td>FG-80-1-MP</td>
<td>539</td>
<td>1.30</td>
<td>1.22</td>
<td>1.27</td>
<td>1.27</td>
</tr>
<tr>
<td>MG-80-1-MP</td>
<td>280</td>
<td>1.39</td>
<td>1.26</td>
<td>1.37</td>
<td>1.36</td>
</tr>
<tr>
<td>FG-80-4-MP</td>
<td>136</td>
<td>1.35</td>
<td>1.24</td>
<td>1.32</td>
<td>1.32</td>
</tr>
<tr>
<td>MG-80-4-MP</td>
<td>135</td>
<td>1.34</td>
<td>1.25</td>
<td>1.31</td>
<td>1.31</td>
</tr>
<tr>
<td>FG-80-16-MP</td>
<td>34</td>
<td>1.42</td>
<td>1.30</td>
<td>1.39</td>
<td>1.39</td>
</tr>
<tr>
<td>MG-80-16-MP</td>
<td>34</td>
<td>1.42</td>
<td>1.30</td>
<td>1.39</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Average quality 1.39 1.28 1.36 1.35
Average time (.s) 0.717 5.355 0.732 9.819

VI. 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, 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 greedy algorithms of low complexity, 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. Therefore, 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.

REFERENCES