Scheduling Malleable Jobs Under Topological Constraints

Evripidis Bampis*, Konstantinos Dogeas*, Alexander Kononov†, Giorgio Lucarelli‡ and Fanny Pascual*
*Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
firstname.lastname@lip6.fr
†Novosibirsk State University, Sobolev Institute of Mathematics, Novosibirsk, Russia
alvenko@math.nsc.ru
‡University of Lorraine, LCOMS, Metz, France
giorgio.lucarelli@univ-lorraine.fr

***Abstract*—Bleuse et al. (EuroPar 2018) introduced a general model for interference-aware scheduling in large scale parallel platforms. They considered two different types of communications: the flows induced by data exchanges during computations and the flows related to Input/Output operations. Rather than taking into account these communications explicitly, they restrict the possible allocations of a job by external topological constraints. In their work, jobs are considered to be rigid: a job requires a specific number of machines in order to be executed. Here, we first adopt the same framework for the platform and the aforementioned topological constraints. We show that there is no polynomial time approximation algorithm under the rigid setting with ratio smaller than 3/2, unless P = NP. Then, we focus on the malleable setting. We show that in the proportional-malleable setting, where the work of every job remains constant independently of the number of machines on which it is executed, the scheduling problem remains NP-hard even in the uniform case, where the maximum number of machines is the same for all the jobs. Then, we propose a 2-approximation algorithm for this case. Furthermore, we present an approximation algorithm solving the more general case where the maximum number of machines is job-dependent and the work of the jobs is increasing with respect to the number of used machines, due to the communication overhead.**

***Keywords*-Approximation algorithms, communications, Input/Output, malleable jobs**

I. Introduction

High Performance Computers (HPCs) are widely used to run applications of great societal importance, due to their extreme computational power. Since their debut, the goal of engineers as well as of the scientific community is to increase their efficiency. For many years, this was achieved via the hardware of the systems: either by increasing the scale of the platform, or by introducing special purpose processors and heterogeneity on the machines (nodes), or by improving the interconnection network. However, when the energy consumption limit was met, HPCs' designers turned their focus on the scheduling algorithms.

Nowadays, real life HPCs consist of more than one type of nodes like computational accelerators (GPUs) as long as machines dedicated to the communication with the file system (Input/Output nodes). GPUs are used due to their computational efficiency in specific kind of operations while I/O nodes have a positive effect on reducing communication cost and they can prevent the network from acting as a bottleneck to the overall performance of the platform, since a single all-purpose interconnection network is usually implemented in a HPC platform.

As the complexity of platforms increases, the need for new, more precise, platform-oriented algorithms, which take into consideration the various features of HPCs, is crucial. In this work, we propose generic scheduling algorithms for HPC platforms taking into account communication issues as well as the existence of I/O nodes.

The idea of generic topology-oriented algorithms is not new. Bladek et al. introduced the notion of *contiguous allocations* and they theoretically proved that imposing this kind of allocations does not deteriorate too much the optimal schedule [1]. Extending this work, Lucarelli et al. studied the impact in backfilling scheduling of topological constraints like *contiguity* and *locality* in hierarchical platforms [2]. Bleuse et al. [3], [4] introduced a more general model for interference-aware scheduling in large scale parallel platforms by distinguishing between direct and indirect topologies. Moreover, they study the effect of a second type of nodes, the input/output nodes, in conjunction with standard computing nodes. This is motivated by the fact that in many current systems, network congestion is a major issue, due to the vast amount of data that are either needed for, or produced from the execution of an application.

In all these works, the communication costs are taken into account implicitly: appropriate allocations that reduce intra-job communications and data transferring from/to the file system are imposed to the scheduler, without measuring explicitly the congestion. Ideally, we would like to design algorithms with no interactions between any two applications, resulting in less congestion in the underlying network, and at the same time do not have a bad effect on the execution of an application (delays due to the imposed constraints) and on the overall performance of the system.

A. Model

We model the platform by distinguishing two kinds of nodes: a set $\mathcal{V}^{\mathrm{C}}$ of m^{C} nodes dedicated to computations,

and a set $\mathcal{V}^{I/O}$ of $m^{I/O}$ nodes that are entry points to a high performance file system. Let $\mathcal{V} = \mathcal{V}^{\mathrm{C}} \cup \mathcal{V}^{I/O}$ and $m = m^{I/O} + m^{\mathrm{C}}$. Usually, $m^{I/O} << m^{\mathrm{C}}$. We assume that each node has a specific functionality: it can either be a computing or an I/O node. Furthermore, we suppose that any computing or I/O node is dedicated to one application throughout its execution, meaning that two jobs cannot use the same node simultaneously.

The network topology considered in this work is the line. This is an interesting topology for the two following reasons. First of all, it is a basic case of higher dimensional topologies and as such it provides lower bounds for the more complex ones. In addition, it retains the attributes of real-life systems which use direct topologies and can be projected in a single dimension, like mesh or 3D-torus. In a line topology, all nodes (computing and I/O) form a single connected component, each one connected to two other nodes, except the two nodes in the extremities. We assume that the localisation of every node within the topology is known. In lines this can be very easily done, by numbering the nodes from left to right.

We see applications as jobs which are queued in a set $\mathcal{J}$. The total number of jobs is n. We distinguish two models with respect to the computing need of a job.

1) In the *rigid* model a job $j \in \mathcal{J}$ requires a fixed number of computing nodes $q_j \leqslant m^{\mathrm{C}}$. The processing time is also fixed, denoted by p_j.

2) In the *malleable* model a job $j \in \mathcal{J}$ asks for a number of computing nodes Q_j, and the scheduler can decide the number of computing nodes $q_j \leqslant Q_j$ to be used for its execution. Each job j has a required execution load, denoted by a_j. The exact processing time of the job j depends on the number of assigned computing nodes. Let $f : \mathbb{N} \to \mathbb{R}$ be a speed-up function. The processing time of j is defined as $p_j = a_j f(q_j)$. A common assumption in parallel computing is that the jobs are *monotonic* [5], that is their processing time is non-increasing when more computing nodes are used, while their total work (execution load plus communication overhead due to the parallelization) is non-decreasing. This is the case when the speed-up function is non-increasing and convex. In this paper, we consider two cases. In the *generalized-malleable model*, the function f is an arbitrary convex non-increasing function. In the *proportional-malleable model*, the total work does not depend on the number of computing nodes assigned to it: $f(q_j) = \frac{1}{q_j}$ and hence $p_j = \frac{a_j}{q_j}$.

In any of the above cases, let $\mathcal{V}^{\mathrm{C}}(j)$ be the set of computing nodes assigned to the job $j \in \mathcal{J}$ by the scheduler.

In addition, jobs have a demand for a specific I/O node, denoted as $\mathcal{V}^{I/O}(j)$. As mentioned before, applications need to read/write data to the disk. Usually, I/O nodes are the entry points to a high performance file system. Lustre, implemented in the BlueWaters platform, is an example of such a distributed file system. The total address range of the file system is divided in stripes and each I/O node is responsible for a stripe. As a result, applications which know where their data is stored, can ask for the specific I/O node.

Due to the parallelization of the HPC jobs, all the parts of a job need to communicate with each other to complete the execution. We refer to this kind of data flows in the network as *computational communications*. Furthermore, in general, jobs that need to run on HPC platforms are computationally demanding and one reason is the great volume of data they need to process. Specifically, each job needs to read the data from the disk when it starts its execution and write data to the disk once it finishes. We refer to this kind of data flows as *I/O communications*. Given the direct topology of the line, each machine is occupied when traffic needs to pass "through" itself in order to arrive to the destination. If this machine is allocated to a different job, then we have the undesirable effect of delaying the completion time of one job in order to handle the traffic from a different job. In order to avoid both the aforementioned data flows, we use the following definitions introduced in [3], [4] to restrict the number of possible allocations of a job.

Definition 1. *An allocation for a job j is said to be* contiguous *if and only if the nodes of the allocation form a contiguous range with respect to the nodes' ordering.*

Definition 2. *An allocation for a job j is said to be* local *if and only if the node $\mathcal{V}^{I/O}(j)$ is adjacent to the computing nodes in $\mathcal{V}^{C}(j)$, with respect to the underlying topology.*

Note that, Bleuse et al. [3] presented a 6-approximation algorithm for the problem of scheduling rigid jobs under the contiguity and the locality constraints. However, these constraints have not been studied in the context of scheduling malleable jobs, which is the main subject of our paper.

Given the overhead of distant communications, we may add a new kind of locality by introducing a limit on the number of machines that may be used for the execution of the jobs. This limitation is parameterized by a common resource requirement $Q = Q_j$ for all jobs in $\mathcal{J}$ in the malleable model. The value of Q is chosen based on the size and the structure of the platform. We call instances satisfying this kind of locality as *uniform* instances. Note that if $Q = m^{\mathrm{C}}$, then the scheduling problem is trivial in the proportional-malleable model, since the total work is constant and thus all jobs will be assigned the maximum number of computing resources. However, for smaller values of Q, the problem becomes $\mathcal{NP}$-hard (Section II).

Our objective is to minimize the maximum completion time among all jobs (i.e., the makespan of the schedule) while enforcing the *contiguity* and the *locality* constraints.

B. Related Work

Scheduling rigid jobs under the contiguity constraint is closely related to the *Strip Packing* problem [6], the *Dynamic*

Storage Allocation [7], as well as to the problem of scheduling multiprocessor jobs [8], [9]. Moreover, scheduling under both contiguity and locality constraints in the rigid model can be seen as a special case of the scheduling problem $P|set_j|C_{max}$, where, for each job $j \in \mathcal{J}$, the set set_j describes the different alternatives (subsets of simultaneously required processors) that can be used for the execution of j: it suffices to define the set_j so that it includes only allocations that satisfy our constraints. If the number of computing nodes is fixed, then a Polynomial Time Approximation Scheme (PTAS) for $P|set_j|C_{max}$ has been presented in [10]. However, if m is part of the instance, there is no polynomial time approximation algorithm with ratio smaller that n^{δ}, for any $\delta > 0$, as shown in [11]. Although this negative result for the more general $P|set_j|C_{max}$, a 6-approximation algorithm has been presented in [3] for scheduling rigid jobs under contiguity and locality constraints.

The problem of scheduling malleable jobs is shown to be *strongly-$\mathcal{NP}$*-hard by Du and Leung [12] in the case of non-monotonic jobs. A 2-factor approximation algorithm for this version has been given by Turek et al. in [13]. Jansen and Porkolab [14] gave a PTAS for instances with a constant number of machines, while in [15], Jansen and Thöle proposed a PTAS when the number of machines is polynomial in the number of jobs. In the case of monotonic jobs, Mounié et al. proposed a $\frac{3}{2}$-approximation algorithm [5]. More recently, Fotakis et al. studied the case of malleable job scheduling, where jobs can be executed simultaneously on multiple non-identical machines with the processing time depending on the number of allocated machines [16]. No results are known for the problem of scheduling malleable jobs under contiguity and locality constraints.

C. Our Contribution

In this paper, we study the problem of scheduling malleable jobs with respect to *contiguity* and *locality* constraints in a line topology, and we give the first complexity and approximability results for it. Our work extends the model proposed by Bleuse et al. [3] and introduces a new mechanism for addressing the energy/performance-tradeoff by using a malleable model. This is a first step towards a more realistic model and our results show that this approach needs to be further investigated in the future.

In Section II, we present complexity results for both the rigid and the proportional-malleable models, implying also the complexity of the generalized-malleable model. We show that the problems are $\mathcal{NP}$-hard even in very restricted cases. We also show that, for any $\epsilon > 0$, there is no approximation algorithm with ratio $\frac{3}{2} - \epsilon$ for the problem of scheduling rigid jobs with respect to contiguity and locality constraints, unless $\mathcal{P}$=$\mathcal{NP}$. This result reduces the approximability gap for the rigid model for which a 6-approximation algorithm is known [3].

In Section III, we first deal with the proportional-malleable model in uniform instances and we propose a novel polynomial-time 2-approximation algorithm. Then, we present an approximation algorithm for the generalized-malleable problem. This algorithm is analyzed in a computational way and it achieves an approximation ratio that depends on the function f.

II. Complexity

In this section, we are proposing reductions to classify special restrictive versions of our problem in complexity classes. In the following theorems, we are going to use the *Partition* problem which is defined as follows: given a finite set $S = \{w_1, w_2, \ldots, w_k\}$ of k positive integers, the objective is to decide if there is a subset $S' \subset S$ such that $\sum_{i \in S'} w_i = \sum_{i \in S \setminus S'} w_i$.

The problem of scheduling rigid jobs under contiguity and locality constraints is shown to be *strongly-$\mathcal{NP}$*-hard by Bleuse et al. in [3]. In the following theorem we provide an inapproximability result for this problem.

Theorem 1. *Unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial time approximation algorithm having a guarantee of $3/2 - \epsilon$ for the problem of scheduling rigid jobs with $p_j = 1$ under contiguity and locality constraints, for any $\epsilon > 0$.*

Proof: We will prove the inapproximability result by a reduction from a special case of the *Partition* problem. In the *Partition-Pairs* problem, we are given two sets $A = \{a_1, a_2, \ldots, a_k\}$ and $A' = \{a'_1, a'_2, \ldots, a'_k\}$, each one containing k elements. Let $S = A \cup A'$. Each element $a_i \in A$ (resp. $a'_i \in A'$) has weight $w_i \in \mathbb{Z}^+$ (resp. $w'_i \in \mathbb{Z}^+$). Let $B = \sum_{a_i \in A} w_i + \sum_{a'_i \in A'} w'_i$. The goal is to decide if there is a partition of S into two subsets S' and $S \setminus S'$ such that

- $\sum_{a_i \in S'} w_i + \sum_{a'_i \in S'} w'_i = \sum_{a_i \in (S \setminus S')} w_i + \sum_{a'_i \in (S \setminus S')} w'_i = \frac{B}{2}$, and
- for each $i \in \{1, \ldots, k\}$, the elements $a_i \in A$ and $a'_i \in A'$ are not assigned to the same set, i.e., if $a_i \in S'$ then $a'_i \in S \setminus S'$ and vice-versa.

Note that, if all the weights in A' are set to zero, we still need to find a solution for the *Partition* problem in the set A. Thus, the problem *Partition-Pairs* is $\mathcal{NP}$-complete.

We propose now a transformation from *Partition-Pairs* to our problem as follows:

- $m^C = k \cdot B + \frac{B}{2}$, $m^{I/O} = k$
- the topology is a line starting with B computing nodes followed by one I/O node. This pattern repeats for k times and after the k^{th} I/O node we have the last $\frac{B}{2}$ computing nodes. With respect to this ordering, we refer to the computing nodes as $1, 2, \ldots, m^C$ and to the I/O nodes as $1, 2, \ldots, m^{I/O}$.
- for each $a_i \in A$ (resp. $a'_i \in A'$)), we create a job j_i (resp. j'_i) with $q_{j_i} = B + w_i$ (resp. $q_{j'_i} = B + w'_i$). Both jobs

j_i and j'_i require the i^{th} I/O node. All jobs of the created instance have unit processing time. Note that $n = 2k$.

Note that this transformation can be done in polynomial time: it is sufficient to give the number of machines $m = kB + \frac{B}{2} + k$, to assume that the machines are numbered from left to right from 1 to m, and to indicate the k numbers which correspond to the I/O nodes. We will prove that a solution to *Partition-Pairs* exists if and only if there is a schedule that satisfies all constraints and has a makespan at most 2.

Assume that there is a solution $(S', S \setminus S')$ for *Partition-Pairs*. For each $i \in \{1, 2, \ldots, k\}$, let us denote by y_i (resp. z_i) the job corresponding to the element in $\{a_i, a'_i\}$ which belongs to S' (resp. $S \setminus S'$). Then, we create a schedule for our problem as follows: we schedule the jobs corresponding to elements in S' at time interval $(0, 1]$ and the jobs corresponding to elements in $S \setminus S'$ at time interval $(1, 2]$. Specifically, in the time interval $(0, 1]$, the job y_1 will use the computing nodes $1, 2, \ldots, B + q_{y_1}$, the job y_2 will use $B+q_{y_1}+1, B+q_{y_1}+2, \ldots, 2B+q_{y_1}+q_{y_2}$, and so on. In general, the job y_i, $1 \leqslant i \leqslant k$, will use the computing nodes $\sum_{\ell=1}^{i-1}(B + q_{y_\ell}) + 1, \ldots, \sum_{\ell=1}^{i}(B + q_{y_\ell})$. In a similar way, in the time interval $(1, 2]$, the job z_i, $1 \leqslant i \leqslant k$, will use the computing nodes $\sum_{\ell=1}^{i-1}(B + q_{z_\ell}) + 1, \ldots, \sum_{\ell=1}^{i}(B + q_{z_\ell})$.

The created schedule satisfies the contiguity constraint. By the construction of the solution $(S', S \setminus S')$, the jobs scheduled in the time interval $(0, 1]$ require in total $\sum_{\ell=1}^{k}(B + q_{y_\ell}) = Bk + \frac{B}{2}$ computing nodes. Similarly, for the jobs scheduled in the time interval $(1, 2]$. Hence, there are enough computing nodes in each time interval. Moreover, by the construction of the scheduling instance, the i^{th} I/O node is between the computing nodes iB and $iB + 1$, for each $1 \leqslant i \leqslant k$. Since for each job y_i, $1 \leqslant i \leqslant k$, it holds that $\sum_{\ell=1}^{i} q_{y_\ell} \leqslant \frac{B}{2}$, then for the leftmost and the rightmost computing nodes assigned to y_i we have $\sum_{\ell=1}^{i-1}(B + q_{y_\ell}) + 1 = (i-1)B + \sum_{\ell=1}^{i-1} q_{y_\ell} + 1 \leqslant (i-1)B + \frac{B}{2} + 1 < iB$ and $\sum_{\ell=1}^{i}(B + q_{y_\ell}) = iB + \sum_{\ell=1}^{i} q_{y_\ell} \geqslant iB + 1$. Thus, the allocation for y_i is local since it always contains the computing nodes iB and $iB + 1$. The same holds for each job z_i, $1 \leqslant i \leqslant k$. Finally, the length of the created schedule is equal to two.

Conversely, assume now that there is a schedule respecting contiguity and locality constraints of makespan at most 2. Due to the unit processing time of each job, each computing node has to execute exactly two jobs. Hence, the partition is directly derived by assigning jobs that are scheduled in the time interval $(0, 1]$ in the set S' and those scheduled in the time interval $(1, 2]$ in the set $S \setminus S'$. Since the scheduling solution respects the locality constraint, the two jobs j_i and j'_i asking for the i^{th} I/O node are scheduled in a different time interval. Therefore, the elements a_i and a'_i are not in the same subset of the solution of the *Partition-Pairs* problem, and hence this solution is feasible for it.

Note that, the above proof directly implies that there is no $3/2-\epsilon$-approximation algorithm for our scheduling problem, assuming that $\mathcal{P} \neq \mathcal{NP}$. ∎

We now focus on the malleable model and we show that the problem is $\mathcal{NP}$-hard even for the proportional-malleable model and uniform instances.

Theorem 2. *The problem of scheduling malleable jobs with respect to contiguity and locality constraints is $\mathcal{NP}$-hard even in the proportional-malleable model and uniform instances.*

Proof: We reduce *Partition* to our scheduling problem. We choose $Q \in \mathbb{Z}^+$ and $B > 0$ such that $2BQ = \sum_{i \in S} w_i$. We create a line which consists of $m^{\mathrm{C}} = 2Q$ computing nodes and $m^{I/O} = 3$ I/O nodes. The topology starts with an I/O node, followed by Q computing nodes, the second I/O node, the remaining Q computing nodes and the last I/O node.

For each $i \in S$, we create a "small" job i with $a_i = w_i$, all of them asking for the middle I/O node. Moreover, we create two "big" jobs with $a_j = BQ^2$, each one asking for a different extreme I/O node. All jobs require exactly Q computing nodes, i.e., $Q_j = Q$ for each $j \in \mathcal{J}$.

We will prove that a solution to *Partition* exists if and only if there is a schedule that satisfies all the constraints and has a makespan at most $B(Q + 1)$.

Assume that there is a solution $(S', S \setminus S')$ for *Partition*, i.e., $\sum_{i \in S'} w_i = \sum_{i \in S \setminus S'} w_i = BQ$. We create a schedule as follows:

- on the leftmost Q computing nodes, we schedule: (i) the "big" job targeting the left I/O node in the time interval $(0, BQ]$ using $q_j = Q$ and hence a processing time $p_j = \frac{a_j}{q_j} = BQ$, and (ii) the "small" jobs corresponding to the elements of the set S' (and targeting the middle I/O node) in the time interval $(BQ, BQ + B]$ using for each of them $q_j = Q$ and hence a processing time $p_j = \frac{a_j}{q_j} = \frac{w_j}{Q}$.
- on the rightmost Q computing nodes, we schedule: (i) the "small" jobs corresponding to the elements of the set $S \setminus S'$ (and targeting the middle I/O node) in the time interval $(0, B]$ using for each of them $q_j = Q$ and hence a processing time $p_j = \frac{a_j}{q_j} = \frac{w_j}{Q}$, and (ii) the "big" job targeting the right I/O node in the time interval $(B, BQ + B]$ using $q_j = Q$ and hence a processing time $p_j = \frac{a_j}{q_j} = BQ$.

By construction, the contiguity and the locality constraints are satisfied, while the makespan of the schedule is exactly $B(Q + 1)$. Due to the solution of *Partition*, we have that $\sum_{i \in S'} w_i = BQ$, and hence the total processing time of all jobs corresponding to the elements of S' is $\frac{BQ}{Q} = B$, fitting in the assigned time interval. The same argument holds for the jobs corresponding to the elements of $S \setminus S'$. Thus, the created schedule is feasible.

Conversely, given a feasible optimal schedule of makespan $B(Q + 1)$, we first need to show that both "big" jobs are necessarily executed using $q_j = Q$ computing

nodes. Suppose that a "big" job is executed using $q_j < Q$ computing nodes. Therefore, the load on each of these nodes is at least

$$\frac{BQ^2}{q_j} \geqslant \frac{BQ^2}{Q-1} = \frac{BQ^2 - B + B}{Q-1} = \frac{B(Q^2-1)+B}{Q-1} = \frac{B(Q-1)(Q+1)+B}{Q-1} = B(Q+1) + \frac{B}{Q-1}$$

which is strictly greater than $B(Q+1)$, and hence we have a contradiction to the feasibility of the schedule. Moreover, the total processing time of "small" jobs that are executed on the computing node just on the left of the middle I/O node is at most $B(Q+1) - BQ = B$; similarly, for the computing node which is just on the right of the middle I/O node. Furthermore, the "small" jobs have a total processing time at least $2B$; this happens if all of them use Q computing nodes. We conclude that the set of "small" jobs has been partitioned into two subsets of the same total processing time, and therefore we can construct a solution for the *Partition* problem. ∎

In the proof of the previous theorem, if $Q = 1$ then the constructed instance coincides with a very special case of the rigid model where the number of machines is fixed, and we get the following corollary.

Corollary 1. *The problem of scheduling rigid jobs with respect to contiguity and locality constraints is $\mathcal{NP}$-hard even if $m^C = 2$, $m^{I/O} = 3$ and $q_j = 1$ for each job $j \in \mathcal{J}$.*

Finally, we can extend the proof given in [3] for the rigid model, and get the following theorem, whose proof is omitted due to space limitations.

Theorem 3. *The problem of scheduling malleable jobs with respect to contiguity and locality constraints is* strongly-*$\mathcal{NP}$-hard even in the proportional-malleable model with $m^{I/O} = 3$ and $a_j = Q_j$, for each $j \in \mathcal{J}$.*

III. Approximation Algorithms

In this section, we propose approximation algorithms for the problem of scheduling malleable jobs with respect to *contiguity* and *locality* constraints.

A. Proportional-Malleable Model in Uniform Instances

In a *uniform instance* of the Proportional-Malleable Model, each job can be executed by at most Q computing nodes. We denote by $\mathcal{M}_j$ the set of computing nodes (machines) that can participate in the execution of a job j. Let $\mathcal{M}$ be an arbitrary set of machines. Then the average time $\mathcal{LB}_1$ for which a machine from $\mathcal{M}$ has to run is a lower bound on the length of an optimal schedule. That is:

$$\mathcal{LB}_1 = \max_{M \subseteq \mathcal{V}^C} \left\{ \frac{\sum_{j | \mathcal{M}_j \subseteq \mathcal{M}} a_j}{|\mathcal{M}|} \right\} \quad (1)$$

For each node $i \in \mathcal{V}^{I/O}$ we denote by $\mathcal{J}_i$ the set of jobs with $\mathcal{V}^{I/O}(j) = i$. Given that each job cannot be allocated on more than Q machines, we obtain a second lower bound.

$$\mathcal{LB}_2 = \max_{i \in \mathcal{V}^{I/O}} \left\{ \sum_{j \in \mathcal{J}_i} \frac{a_j}{Q} \right\} \quad (2)$$

In total, the lower bound is given by the biggest of these quantities, $\mathcal{LB} = \max\{\mathcal{LB}_1, \mathcal{LB}_2\}$.

In this section, we present an algorithm that finds a schedule whose makespan does not exceed $2\mathcal{LB}$. Firstly, we transform the instance in order to create a simplified jobset $\mathcal{J}'$. For each node $i \in \mathcal{V}^{I/O}$ we replace each set of jobs $\mathcal{J}_i$ with a single job j', the execution load of which is equal to $A_{j'} = \sum_{j \in \mathcal{J}_i} a_j$. After determining the allocation for a job j' in $\mathcal{J}'$, we will assign all jobs j in $\mathcal{J}$ corresponding to the same I/O node to the same set of computing nodes.

Proposition 1. *$\mathcal{J}$ and $\mathcal{J}'$ have the same lower bound $\mathcal{LB}$.*

We introduce at this point an *auxiliary problem* which will help us find a feasible schedule to our initial problem.

Auxiliary problem

The instance consists of a set of malleable rectangles $\mathcal{U} = \{U_1, U_2, \ldots, U_m\}$ and a strip of width W. Designate the bottom left corner of the strip as the origin of the xy-plane, letting the x-axis be the direction of the width of the strip, and the y-axis be the direction of the height. Each rectangle U_i has a fixed area A_i and a given access point (which corresponds to the input/output nodes of our original problem) τ_i, such that $0 \leqslant \tau_1 \leqslant \tau_2 \leqslant \cdots \leqslant \tau_m \leqslant W$. We represent the location of each rectangle U_i in the strip by the coordinate (s_i, y_i) of its bottom left corner and the coordinate (f_i, y_i) of its bottom right corner. We denote the width of a rectangle as $\lambda_i = f_i - s_i$. In this case, the height of the rectangle R_i is equal to $h_i = \frac{A_i}{\lambda_i}$.

We say that the location of the rectangles is valid if the following *conditions* hold:

1) $s_i, f_i \in \mathbb{N}$, $i = 1, \ldots, m$
2) $1 \leqslant \lambda_i \leqslant Q$, $i = 1, \ldots, m$
3) $s_i \leqslant \tau_i \leqslant f_i$, $i = 1, \ldots, m$
4) $s_i \leqslant s_j$, $\forall\ i < j$
5) $f_i \leqslant f_j$, $\forall\ i < j$

In correspondence with the initial scheduling problem, condition 1 means that the computing need of a job is integral, while in condition 2, a job must ask for at least one node with the upper limit being Q. Condition 3 guarantees locality for a job. Finally, conditions 4 and 5 ensures jobs are scheduled following the ordering of the I/O nodes.

The objective of the auxiliary problem is to place m rectangles into the strip without intersections, so as to minimize the height H of the strip.

Consider an arbitrary rectangle U_i. In order to satisfy conditions 2, 3, we define the interval $[r_i, d_i]$. Let $r_i =$

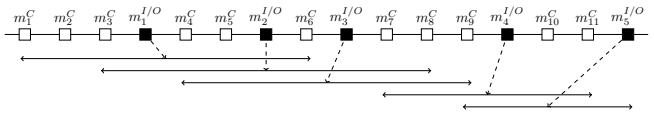


Figure 1. Intervals $[r_i, d_i]$ for jobs with for $Q = 3$

$\max\{0, \tau_i - Q\}$ and $d_i = \min\{W, \tau_i + Q\}$. We can see this interval as the set of computing nodes, where a job i can be scheduled both *locally* and *contiguously*. An example of these intervals is shown in Fig. 1. Conditions 2 and 3 also imply that in any feasible solution of the auxiliary problem we have $r_j \leqslant s_j < f_j \leqslant d_j$.

In terms of the auxiliary problem, we can re-write lower bounds (1) and (2) as follows.

$$\mathcal{LB}_1 = \max_{i,j|i<j} \left\{ \frac{\sum_{k|r_i \leqslant r_k \leqslant d_k \leqslant d_j} A_k}{d_j - r_i} \right\} \tag{3}$$

$$\mathcal{LB}_2 = \max_i \left\{ \frac{A_i}{Q} \right\} \tag{4}$$

Finally, we have that, $\mathcal{LB} = \max\{\mathcal{LB}_1, \mathcal{LB}_2\}$

Proposition 2. *If we do not impose integrality and locality (conditions 1 and 3), the auxiliary problem under conditions 2, 4, 5 has a solution of $H = \mathcal{LB}$.*

To create such a solution, we set $\lambda_i = \frac{A_i}{\mathcal{LB}}$ for all $U_i \in \mathcal{U}$. From (4) we have $\lambda_i \leqslant Q$. We then place each rectangle on the bottom line, so $y_i = 0$ for all $U_i \in \mathcal{U}$. We determine the x-coordinates of each rectangle according to the following rule. For the first rectangle we set $s_1 = 0$ and $f_1 = \lambda_1$. For the rest of the jobs we set $s_i = \max\{r_i, f_{i-1}\}$ and $f_i = s_i + \lambda_i$.

In the next proof we show that the solution described above and summarized in Algorithm 1, provide a feasible solution to the auxiliary problem with respect to condition 2, 4, 5.

Proof: We will prove by contradiction that $f_i \leqslant d_i$ for all i. Suppose there exists a rectangle U_i such that $f_i > d_i$. Let $j < i$ be the maximal index of a rectangle such that $s_j = r_j$. Then we have $s_k = f_{k-1}$ for all $k = j+1, \ldots, i$. Thus, we have that

$$\sum_{k=j}^{i} \lambda_k > d_i - r_j \tag{5}$$

Algorithm 1: Auxiliary solution

```
λ_1 = A_1/LB; y_1 = 0; s_1 = 0; f_1 = λ_1;
for i > 1 | U_i ∈ U do
    λ_i = A_i/LB; y_i = 0;
    s_i = max{r_i, f_{i-1}}; f_i = s_i + λ_i;
```

Moreover, for each rectangle U_k, where $k = j+1, \ldots, i$ we have $r_j \leqslant r_k$ and $d_k \leqslant d_i$. From $\mathcal{LB}$ and (3) we obtain

$$\mathcal{LB} \geqslant \frac{\sum_{k=j}^{i} A_k}{d_i - r_j} = \frac{\mathcal{LB} \sum_{k=j}^{i} A_k}{\mathcal{LB}(d_i - r_j)} = \frac{\mathcal{LB} \sum_{k=j}^{i} \lambda_k}{d_i - r_j} > \mathcal{LB}$$

where the last inequality follows from (5); contradiction. ■

Algorithm 1 places all the rectangles as close as possible to the left edge of the strip. We shift to the right those rectangles for which $f_i < \tau_i$, without changing the order or the location of the other rectangles. Thus, a rectangle can be moved either until it becomes *local*, or until it meets the starting node of the next job. This procedure, which returns a strip S', is described in Algorithm 2.

The solution created by Algorithm 2 may consist of several blocks of jobs. A *block* is a maximal set of rectangles that form a continuous strip of size $\mathcal{LB}$ and consists of:

- a set of *local* rectangles for which $\tau_i \in [s_i, f_i]$,
- a set of rectangles, $\mathcal{L}$ (left), for which $\tau_i > f_i$,
- a set of rectangles, $\mathcal{R}$ (right), for which $\tau_i < s_i$.

Moreover, all left rectangles precede local rectangles, and all right rectangles succeed local rectangles. Notice that sets $\mathcal{L}$ and $\mathcal{R}$ may be empty, while the set of local rectangles must have at least one rectangle. After applying Algorithm 2, it is the local rectangles which prevent other jobs from being local. Therefore, there is always a local rectangle in a block. The structure of a block can be seen in Fig. 3a.

Our main idea is to split the set of rectangles in $\mathcal{U}$ into two subsets, $\mathcal{U}_1$ and $\mathcal{U}_2$, and then pack each of them into a strip of height $\mathcal{LB}$. We start with local rectangles. We number the rectangles in the order as they are carried out in the strip S', resulted by Algorithm 2. Then, we reallocate each local rectangle i to the interval $[\lfloor s_i \rfloor, \lceil f_i \rceil]$. Subsequently, we set all odd local rectangles in the first subset $\mathcal{U}_1$ and all even rectangles on the second subset $\mathcal{U}_2$. Each rectangle i has a positive width, and therefore we get that $s_{i+2} \geqslant f_{i+1} > s_{i+1} = f_i$. Since each τ_i is integral, and for each local rectangle we have $\tau_i \in [s_i, f_i]$, we get that $\lfloor s_{i+2} \rfloor \geqslant \lceil f_i \rceil$, and as a result local rectangles from the same subset do not overlap.

For rectangles which are not local, we focus on each block separately. We make rectangles of a block to be *local*, starting by jobs in set $\mathcal{R}$. Rectangles in set $\mathcal{L}$ are handled similarly and therefore the procedure is not explicitly mentioned here.

Algorithm 2: Create Initial Blocks

```
Start with the solution S returned by Algorithm 1;
for (i = |J'|; i == 1; i--) do
    if τ_i > f_i then
        t = min{(s_{i+1} − f_i), (τ_i − f_i)};
        f_i += t; s_i += t;
```

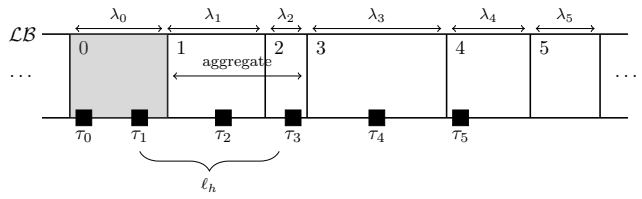


Figure 2. Example of creating the first aggregated job of a block in $\mathcal{R}$. The renumbering of nodes and jobs of this step is also depicted. Jobs correspond to the jobs $5-10$ shown in Fig. 3a.

We call the last local rectangle in the block as R_0. Consider the rectangles in $\mathcal{R} \cup \{R_0\}$. We refer to the access point of the rectangle R_0 as τ_0 and respecting the ordering from left to right we re-number the access points of the rectangles in $\mathcal{R}$. Rectangles' indices follow the same numbering. Fig. 2 illustrates this notation.

Let $r = |\mathcal{R}|$. We partition the set of rectangles $\mathcal{R}$ into disjoint subsets using the following procedure (see Algorithm 3). We consider each subset $\mathcal{R}_h$ as an aggregated rectangle R_h. For each aggregated rectangle R_h, we define two quantities. The first one is

$$dist^C = \sum_{U_j \in \mathcal{R}_h} \lambda_j \tag{6}$$

which corresponds to the total width of all the rectangles in $\mathcal{R}_h$. Let $\tau_h^{min} = \min_{U_j \in \mathcal{R}_h}\{\tau_j\}$ and $\tau_h^{max} = \max_{U_j \in \mathcal{R}_h}\{\tau_j\}$. The second one is

$$dist^{I/O} = \tau_h^{max} - \tau_h^{min} \tag{7}$$

and is related to the distance between the leftmost and the rightmost access points in $\mathcal{R}_h$. We then define the width of the aggregated rectangle R_h as

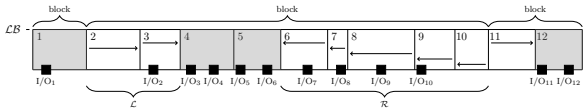
$$\ell_h = \max\left\{\left\lceil dist^C \right\rceil, dist^{I/O}\right\} \tag{8}$$

More precisely, we replace each rectangle $U_j \in \mathcal{R}_h$ of width λ_j and height $\mathcal{LB}$ with a rectangle of width ℓ_h and height $\lambda_j \mathcal{LB}/\ell_h$, and put them on top of each other. As a result, we get a rectangle of width ℓ_h and of height $dist^C \mathcal{LB}/\ell_h$, which is no more than Q.

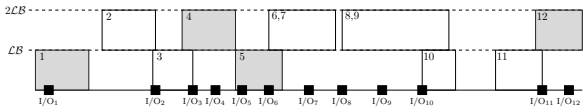
At this point, we are ready to make rectangles in $\mathcal{R}$ both *local* and *integral*. We place each aggregated rectangle R_h in the interval $[\tau_h^{min}, \tau_h^{min} + \ell_h]$. For each rectangle, we also need to choose one of the subsets $\mathcal{U}_1$ or $\mathcal{U}_2$.

Algorithm 3: Create Aggregated Rectangles

```
Put h = 0; k = 1;
while k ⩽ r do
    set i = k, h = h + 1, R_h := ∅;
    while Σ_{j=i}^{k} λ_j ⩽ Q and τ_k − τ_i ⩽ Q do
        set R_h = R_h ∪ {U_k}; k = k + 1;
```



(a) Schedule returned by Algorithm 2. Structure of a *block*.



(b) The final schedule after imposing locality and integrality

Figure 3. Example of the transformation from a partial solution return by Algorithm 2 to a *contiguous*, *local*, *integral* feasible schedule of our algorithm. Input/Output nodes are depicted by a black square. Computing nodes are not shown. Jobs and Input/Output nodes follow the same numbering. Jobs in grey are *local* while jobs in white need to be reallocated.

This choice is based on the placement of the previous rectangle. Rectangles R_0 and R_1 are always assigned to different sets based on the choice for R_0. If the maximum in ℓ_{i-1} is given by $\left\lceil dist^C \right\rceil$ and $R_{i-1} \in \mathcal{U}_1$, assign R_i to $\mathcal{U}_2$ (resp. if $R_{i-1} \in \mathcal{U}_2$, assign R_i to $\mathcal{U}_1$). If the maximum in ℓ_{i-1} is given by $dist^{I/O}$ and $R_{i-1} \in \mathcal{U}_1$ (resp. $\mathcal{U}_2$), assign R_i to $\mathcal{U}_1$ (resp. $\mathcal{U}_2$). Similarly, we impose locality in $\mathcal{L}$.

Lemma 4. *The aggregated rectangles do not overlap.*

Proof: Since all rectangles in $\mathcal{R}$ are shifted to the left and all rectangles in $\mathcal{L}$ are shifted to the right, the shifted rectangles from different blocks do not overlap. Let U_l be any left aggregated rectangle and U_r to be any right aggregated rectangle. We have $f_l = \tau_l^{max} \leqslant \tau_0 \leqslant \tau_r^{min} = s_r$. It follows that two rectangles U_l, U_r with $U_l \in \mathcal{L}$ and $U_r \in \mathcal{R}$ from the same block do not overlap.

Now we show that rectangles in $\mathcal{R}$ from one block do not overlap. Let $\ell_{h-1} = \tau_{h-1}^{max} - \tau_{h-1}^{min}$. Since $\tau_{h-1}^{max} < \tau_h^{min}$ the rectangles J_{h-1} and J_h do not overlap. Now, let $\ell_{h-1} = \sum_{j \in R_{h-1}} \lambda_j$. In this case, rectangles J_h and J_{h-1} belong to different sets and do not overlap. It remains for us to show that the rectangle J_h does not overlap with previous rectangles. Let U_j be the first rectangle in R_h. From the execution of Algorithm 3, we have $\ell_{h-1} + \lambda_j > Q$. Hence, $f_j - \min_{i \in R_{h-1}} s_i > Q$. Taking into account that $\tau_j = d_j - Q \geqslant f_j - Q$, we obtain

$$\tau_h^{min} = \tau_j \geqslant f_j - Q > \min_{i \in R_{h-1}} s_i = s_{h-1} \geqslant \tau_{h-1}^{min} > \tau_{h-2}^{max}$$

Similarly, rectangles in $\mathcal{L}$ from one block do not overlap. ■

Lemma 5. *The aggregated rectangles and the local rectangles do not overlap.*

Proof: We prove this result for jobs in $\mathcal{R}$. Let R_i be the first aggregated rectangle that belongs to the same set as R_0. It follows that $\ell_{i-1} = \sum_{j \in R_{i-1}} \lambda_j$. As shown in Lemma 4 we have $\tau_i^{min} \geqslant s_{i-1} \geqslant f_0$. Since τ_i^{min} is integer, we have $\tau_i^{min} \geqslant \lceil f_0 \rceil$ and rectangles R_0 and R_i do not overlap. Similarly, we can prove this result for jobs in $\mathcal{L}$. ■

Following the procedure above, we can find a solution of height $2\mathcal{LB}$ to the auxiliary problem which partitions the rectangles into two subsets, and packs them in two strips of height $\mathcal{LB}$ without intersections.

Now, consider the topology of an instance of the scheduling problem. We number the computing nodes in $\mathcal{V}^{\mathrm{C}}$ independently from the I/O nodes, respecting their ordering on the line. Furthermore, we set $W = m^{\mathrm{C}}$ and we associate the unit interval $[i-1, i]$ with the computing node i. For each I/O node $j \in \mathcal{V}^{I/O}$ we set $\tau_j = 0$ if the I/O node precedes all computing nodes, $\tau_j = m^{\mathrm{C}}$ if the I/O-node follows all computing nodes and $\tau_j = k$ if the I/O node is located between the computing nodes k and $k+1$. For the j-th I/O node we create a rectangle R_j such that its area is A_j.

Lemma 6. *Let S be a feasible solution of the auxiliary problem with height H. Then there exists a feasible solution of the scheduling problem with makespan H.*

Proof: Let the rectangle U_j has coordinates (s_j, y_j) and (f_j, y_i) for its bottom left corner and its bottom right corner, respectively. We assign the job j corresponding to the j-th I/O node on computing nodes $\{s_j + 1, \ldots, f_j\}$ in the time interval $(y_j, y_j + h_j]$, where $h_j = \frac{A_j}{\lambda_j}$. Let $\mathcal{V}_t^{I/O} = \{j \in \mathcal{V}^{I/O} | \tau_j = t\}$. If $s_j < t < f_j$ then job j occupies all I/O nodes from $\mathcal{V}_t^{I/O}$. If $s_j = t$ that job j occupies the node $\mathcal{V}^{I/O}(j)$ and all I/O nodes from $\mathcal{V}_t^{I/O}$ to the right of the node $\mathcal{V}^{I/O}(j)$. If $f_j = t$ that job j occupies the node $\mathcal{V}^{I/O}(j)$ and all I/O nodes from $\mathcal{V}_t^{I/O}$ to the left of the node $\mathcal{V}^{I/O}(j)$. Thus, the job is also local due to condition 3 of the auxiliary problem. Suppose that a job i and a job j overlap on some I/O node. Let $i < j$. Since the rectangles R_i and R_j do not overlap we have $\tau_i = f_i = s_j = \tau_j$ due to conditions 4 and 5. But in this case, the job i does not occupy the I/O-nodes to the right of the i-th I/O node and the job j does not occupy the I/O-nodes to the left of the j-th I/O node. Hence these jobs do not overlap. ∎

This directly yields the following result.

Theorem 7. *There exists a polynomial time factor 2 approximation algorithm for the problem of scheduling malleable jobs with respect to contiguity and locality constraints on line in the uniform proportional-malleable model.*

B. Generalized-Malleable model

Bleuse et al. [3] introduced an integer linear program for the rigid model with respect to contiguity and locality constraints in order to minimize the total load of each node; note that the maximum load over all nodes is a lower bound to the makespan of the schedule. By solving the relaxed version and rounding this solution, they obtain one allocation whose makespan is at most twice the maximum load in the solution returned by the linear program (LP). Having fixed a valid allocation for each job, the problem coincides with the already known Dynamic Storage Allocation problem for which they use an already known 3-approximation algorithm [17] to create a feasible schedule, getting a 6-approximation algorithm for the rigid model.

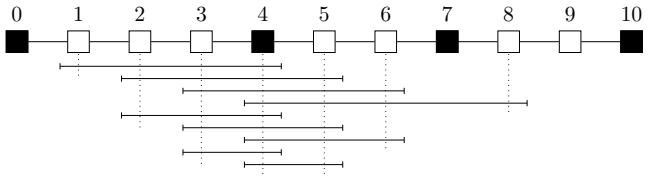


Figure 4. Possible valid allocations for a job j asking for 3 computing nodes ($Q_j = 3$) and the 4^{th} node as input/output.

We extend the integer linear program of [3] as follows. Let $\mathcal{A}_j$ be the set of all potential allocations for each job $j \in \mathcal{J}$. In the malleable model, the set $\mathcal{A}_j$ contains more allocations than the rigid model, as in the former one we have also to decide the number of computing nodes to be used for the execution. Due to the contiguity and the locality constraints, there are $Q_j + 1$ allocations using Q_j computing nodes, Q_j allocations using $Q_j - 1$ computing nodes, and so on. Hence, $|\mathcal{A}_j| \leqslant \sum_{i=1}^{Q_j}(i+1)$: the number of potential allocations for each job j remains polynomial.

Each allocation $\ell \in \mathcal{A}_j$ contains a number of computing nodes as well as the required I/O node. Note that, an allocation may include more I/O nodes that will not be used during the execution of j, neither by j nor by any other job due to the locality constraint. By slightly abusing the notation, given an allocation ℓ, we write $i \in \ell$ if the node i is included in ℓ, and we denote by $|\ell|$ the number of computing nodes included in ℓ. Moreover, given an allocation $\ell \in \mathcal{A}_j$ for a job $j \in \mathcal{J}$, we denote by $p_{j\ell} = a_j f(|\ell|)$ the processing time of j if it is executed according to ℓ. Note that the number of different $p_{j\ell}$ is also polynomial. An example of all possible allocations can be seen in Fig. 4. For each job $j \in \mathcal{J}$ and allocation $\ell \in \mathcal{A}_j$, we introduce a binary indicator variable $x_{j,\ell}$ which is equal to one if j is executed according to the allocation ℓ, and zero otherwise. Moreover, for each node $i \in \mathcal{V}$ (computing and I/O) we introduce a non-negative variable Λ_i which corresponds to the total load of jobs whose assigned allocation includes the node i. Let also Λ be a variable corresponding to the maximum load among all nodes. Then, we consider the following integer linear program which minimizes the maximum load.

$$\min \Lambda, \qquad \text{(ILP)}$$

$$\text{s.t.} \quad \Lambda \geqslant \Lambda_i \qquad \forall i \in \mathcal{V} \qquad (C_1)$$

$$\Lambda_i \geqslant \sum_{j \in \mathcal{J}} \sum_{\ell \in \mathcal{A}_j} \sum_{i \in \ell} x_{j\ell} p_{j\ell} \qquad \forall i \in \mathcal{V} \qquad (C_2)$$

$$\sum_{\ell \in \mathcal{A}_j} x_{j\ell} = 1 \qquad \forall j \in \mathcal{J} \qquad (C_3)$$

$$x_{j\ell} \in \{0, 1\} \qquad j \in \mathcal{J}, \ell \in \mathcal{A}_j \qquad (C_4)$$

Constraints (C_1) take the maximum load over all nodes.

Constraints (C_2) compute the total load for each node, while Constraints (C_3) ensure that each job is assigned to an allocation. By relaxing the integrity Constraints (C_4) to $x_{j\ell} \in [0,1]$ for each $j \in \mathcal{J}$ and $\ell \in \mathcal{A}_j$, we can solve the corresponding LP in polynomial time. An optimal solution to the relaxed linear program is a lower bound to the makespan of an optimal solution for our problem.

The main differences of the above integer linear program with respect to the one for rigid jobs is that: (i) there is a quadratic number to Q_j of potential allocations for each job j (instead of linear) and (ii) the processing time of j depends on the allocation and the number of computing nodes used by it (instead of a single p_j for all potential allocations).

Consider now an optimal solution of the relaxed linear program. In this solution, let $\tilde{x}_{j\ell}$ be the value of the indicator variable for each job $j \in \mathcal{J}$ and allocation $\ell \in \mathcal{A}_j$, and $\tilde{\Lambda}_i$ be the value of the variable corresponding to the load of node i. In the following, we explain how to round these indicator variables and get an integral allocation for each job $j \in \mathcal{J}$. Let $\bar{x}_{j\ell}$ be the integral value of the indicator variable for each job $j \in \mathcal{J}$ and allocation $\ell \in \mathcal{A}_j$ after the rounding and $\bar{\Lambda}_i$ the corresponding load of node i. We denote by $L_i(j)$ the contribution of job j to the load of node $i \in \mathcal{V}$ in solution $\tilde{\Lambda}_i$: $L_i(j) = \sum_{\ell: i\in\ell} \tilde{x}_{j\ell} p_{j\ell}$. Let $\tilde{\mathcal{V}}_j = \{i : L_i(j) > 0\}$ be the set of nodes having a positive fractional load for the job j.

Given an allocation $\ell \in \mathcal{A}_j$, we consider the worst case increase of the load of a machine if we decide to schedule j according to ℓ. Specifically, for each node $i \in \tilde{\mathcal{V}}_j$ in this allocation ℓ we compute the ratio $\frac{p_{j\ell}}{L_i(j)}$, while the worst case corresponds to the node for which this ratio is maximized. Finally, we decide to schedule j according to the allocation ℓ^* that minimizes this worst case ratio and we set $\bar{x}_{j\ell^*} = 1$. All the other variables for the job j are set to zero, i.e., $\bar{x}_{j\ell} = 0$ for each $\ell \neq \ell^*$. Intuitively, the above procedure aims to choose the allocation for each job $j \in \mathcal{J}$ that increases as little as possible the impact of j on the load of the nodes, without regarding the load of the other jobs.

When a single allocation has been selected for each job, our problem coincides with the Dynamic Storage Allocation problem and we can apply the 3-approximation algorithm proposed in [17]. Algorithm 4 summarizes this procedure.

In what follows, we bound the approximation ratio of Algorithm 4. We initially focus on the rounding procedure (Lines 2–7 of the algorithm). Our analysis is performed for each job $j \in \mathcal{J}$ separately and the approximation ratio of our algorithm depends on the function f and $Q_{\max} = \max\{Q_j, j \in \mathcal{J}\}$. However, the algorithm works for instances with different values of Q_j.

The key idea of our analysis is, given a job $j \in \mathcal{J}$, to find a worst-case assignment for the variables $x_{j\ell}$, $\ell \in \mathcal{A}_j$, that maximizes the quantity $\min_{\ell\in\mathcal{A}_j}\{ratio_j^\ell\}$: it will maximize the minimal increase of the contribution of task j to the load of a machine between $\tilde{\Lambda}_i$ and $\bar{\Lambda}_i$. Then, any other assignment for the variables $x_{j\ell}$, including the one obtained by solving

Algorithm 4:

1 Solve the relaxed version of (ILP)
2 **for** *each job* $j \in \mathcal{J}$ **do**
3 **for** *each node* $i \in \{1, \ldots, m\}$ **do**
4 $L_i(j) = \sum_{\ell: i\in\ell} \tilde{x}_{j\ell} p_{j\ell}$
5 **for** *each allocation* $\ell \in \mathcal{A}_j$ **do**
6 $ratio_j^\ell = \max_{i\in\ell, i\in\tilde{\mathcal{V}}_j} \left\{ \frac{p_{j\ell}}{L_i(j)} \right\}$
7 Choose the allocation $\ell^* = \text{argmin}_{\ell\in\mathcal{A}_j}\{ratio_j^\ell\}$
8 Create a feasible schedule by applying the algorithm proposed in [17] for the Dynamic Storage Allocation problem using the allocations determined by ℓ^*.

the relaxed (ILP), will lead to a smaller increase. In order to do this, we create the following *feasibility linear program* for the job $j \in \mathcal{J}$, where α is a constant which corresponds to the value of $\min_{\ell\in\mathcal{A}_j}\{ratio_j^\ell\}$ we are searching for.

$$p_{j\ell} \geqslant \alpha \sum_{\ell': i\in\ell'} x_{j\ell'} p_{j\ell'} \quad \forall \ell \in \mathcal{A}_j, i \in \ell \qquad (R_1)$$

$$\sum_{\ell\in\mathcal{A}_j} x_{j\ell} = 1 \qquad (R_2)$$

$$x_{j\ell} \geqslant 0 \qquad \forall j \in \mathcal{J}, \ell \in \mathcal{A}_j \qquad (R_3)$$

Constraints (R_1) express the ratio between the integral load if allocation ℓ is selected to execute j and the fractional load based on the obtained assignment for a node i, i.e., correspond to the quantity $\frac{p_{j\ell}}{L_i(j)}$. Constraints (R_2) ensure the j is assigned and guarantees the constraints (C_3) of ILP.

Observe that, the values of $p_{j\ell} = a_j f(|\ell|)$ and $p_{j\ell'} = a_j f(|\ell'|)$ in Constraint (R_1) depend on the same job j. Thus, a_j can be eliminated and the constraint depends only on the number of computing nodes of allocations ℓ, ℓ' (which take value in $\{1, 2, \ldots, Q_j\}$) and the function f. In other words, we obtain the same feasibility linear program for all jobs requiring the same number of computing nodes Q_j, and thus the value of α depends only on the function f and the Q_j, and not the specific job. In order to determine the value of α, we perform a binary search. An infeasible solution to the above linear program implies that it is not possible to have a gap of α and hence we need to choose a smaller value of α. At the end of the binary search procedure, we get the maximum value of α that makes the linear program (R_1)–(R_3) feasible. Then, the following lemma holds.

Lemma 8. *For a job $j \in \mathcal{J}$ and a node $i \in \mathcal{V}$, it holds that $p_{j\ell^*} \leqslant \alpha L_i(j)$, where α is the maximum value which makes the linear program (R_1)–(R_3) feasible.*

As an example, we calculated the value of α for different values of Q_j in the proportional-malleable model. Fig. 5 illustrates some of these values.

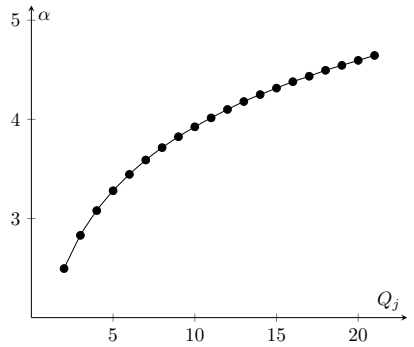


Figure 5. The value of α with respect to different Q_j's for the proportional-malleable model.

Theorem 9. *Let $\alpha_{\max}$ be the maximum value over all jobs which makes the linear program (R_1)–(R_3) feasible. Algorithm 4 achieves an approximation ratio of $3\alpha_{\max}$.*

Proof: For each job $j \in \mathcal{J}$, let ℓ_j^* be the allocation selected by Algorithm 4 to execute j. Then, for the integral load of the node $i \in \mathcal{V}$ we have

$$\begin{aligned}\bar{\Lambda}_i &= \sum_{j\in\mathcal{J}:i\in\ell_j^*} p_{j\ell_j^*} \leqslant \sum_{j\in\mathcal{J}:i\in\ell_j^*} \alpha_{\max} L_i(j) \\ &= \alpha_{\max} \sum_{j\in\mathcal{J}:i\in\ell_j^*} \sum_{\ell:i\in\ell} \tilde{x}_{j\ell} p_{j\ell} \\ &= \alpha_{\max} \sum_{j\in\mathcal{J}:i\in\ell_j^*} \sum_{\ell\in\mathcal{A}_j} \sum_{i\in\ell} \tilde{x}_{j\ell} p_{j\ell} = \alpha_{\max}\tilde{\Lambda}_i\end{aligned}$$

By construction, $\tilde{\Lambda} = \max_{i\in\mathcal{V}}\{\tilde{\Lambda}_i\}$ is a lower bound the makespan of an optimal schedule for our problem. Then, the theorem follows, since in Line 8 of Algorithm 4 the 3-approximation algorithm for the Dynamic Storage Allocation is used in order to create the final schedule. ■

IV. Conclusion

In this work we studied the makespan minimization problem on the malleable and the rigid models under contiguity and locality constraints. We give inapproximability results for the rigid model and complexity results for the malleable one. Focusing on the malleable model, we give approximation algorithms for the proportional uniform setting as well as the generalized one. As future work, it would be interesting to search for a constant factor approximation algorithm for the generalized-malleable problem and further close the approximability gap for both malleable and rigid settings. Furthermore, one can try to extend the topological constraints and therefore the algorithms mentioned in this work in more complex and differently structured topologies.

Acknowledgment

The work presented in this paper has been funded by the ANR project ENERGUMEN, ANR-18-CE25-0008 and the RFBR 20-07-00458.

References

[1] I. Bladek, M. Drozdowski, F. Guinand and X. Schepler, *On Contiguous and Non-contiguous Parallel Task Scheduling*, J. Scheduling vol. 18, no. 5, pp. 487–495, 2015.

[2] G. Lucarelli, F. M. Mendonca, D. Trystram and F. Wagner, *Contiguity and Locality in Backfilling Scheduling*, CCGrid'15 pp. 586–595, 2015.

[3] R. Bleuse, K. Dogeas, G. Lucarelli, G. Mounié and D. Trystram, *Interference-Aware Scheduling Using Geometric Constraints*, Euro-Par'18 pp. 205–217, 2018.

[4] R. Bleuse, G. Lucarelli and D. Trystram, *A Methodology for Handling Data Movements by Anticipation: Position paper*, COLOC'18 (Euro-Par Workshops) pp. 134–145, 2018.

[5] G. Mounié, C. Rapine and D. Trystram, *A 3/2-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks*, SIAM J. Comput. vol. 37, no. 2, pp. 401–412, 2007.

[6] C. Kenyon and E. Rémila, *Approximate Strip Packing*, FOCS'96 pp. 31–36, 1996.

[7] A. L. Buchsbaum, H. J. Karloff, C. Kenyon, N. Reingold and M. Thorup, *OPT Versus LOAD in Dynamic Storage Allocation*, SIAM J. Comput. vol. 33, no. 3, pp. 632–646, 2004.

[8] M. Drozdowski, *Scheduling multiprocessor tasks — An overview*, European Journal of Operational Research vol. 94, no. 2, pp. 215 - 230, 1996.

[9] A. K. Amoura, E. Bampis, C. Kenyon and Y. Manoussakis, *Scheduling Independent Multiprocessor Tasks*, Algorithmica vol. 32, no. 2, pp. 247–261, 2002.

[10] J. Chen and A. Miranda, *A Polynomial Time Approximation Scheme for General Multiprocessor Job Scheduling*, SIAM J. Comput. vol. 31, no. 1, pp. 1–17, 2001.

[11] A. Miranda, L. Torres and J. Chen, *On the Approximability of Multiprocessor Task Scheduling Problems*, ISAAC'02 pp. 403–415, 2002.

[12] J. Du and J. Y. Leung, *Complexity of Scheduling Parallel Task Systems*, SIAM J. Discrete Math. vol. 2, no. 4, pp. 473–487, 1989.

[13] J. Turek, J. L. Wolf and P. S. Yu, *Approximate Algorithms Scheduling Parallelizable Tasks*, SPAA'92 pp. 323–332, 1992.

[14] K. Jansen and L. Porkolab, *Linear-Time Approximation Schemes for Scheduling Malleable Parallel Tasks*, Algorithmica vol. 32, no. 3, pp. 507–520, 2002.

[15] K. Jansen and R. Thöle, *Approximation Algorithms for Scheduling Parallel Jobs*, SIAM J. Comput. vol. 39, no. 8, pp. 3571–3615, 2010.

[16] D. Fotakis, J. Matuschke and O. Papadigenopoulos, *Malleable Scheduling Beyond Identical Machines*, APPROX/RANDOM'19 pp. 17:1–17:14, 2019.

[17] J. Gergov, *Algorithms for Compile-Time Memory Optimization*, SODA'99 pp. 907–908, 1999.