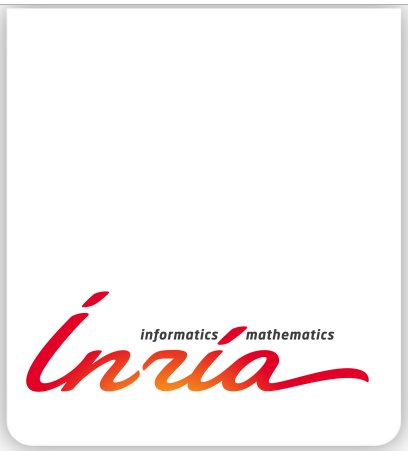
Max-stretch minimization on an edge-cloud platform

Anne Benoit, Redouane Elghazi, Yves Robert

RESEARCH REPORT
N° 9369
October 2020
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9369--FR+ENG



Max-stretch minimization on an edge-cloud platform

Anne Benoit, Redouane Elghazi, Yves Robert

Project-Team ROMA

Research Report n° 9369 — October 2020 — 37 pages

Abstract: We consider the problem of scheduling independent jobs that are generated by processing units at the edge of the network. These jobs can either be executed locally, or sent to a centralized cloud platform that can execute them at greater speed. Such edge-generated jobs may come from various applications, such as e-health, disaster recovery, autonomous vehicles or flying drones. The problem is to decide where and when to schedule each job, with the objective to minimize the maximum stretch incurred by any job. The stretch of a job is the ratio of the time spent by that job in the system, divided by the minimum time it could have taken if the job was alone in the system. We formalize the problem and explain the differences with other models that can be found in the literature. We prove that minimizing the max-stretch is NP-complete, even in the simpler instance with no release dates (all jobs are known in advance). This result comes from the proof that minimizing the max-stretch with homogeneous processors and without release dates is NP-complete, a complexity problem that was left open before this work. We design several algorithms to propose efficient solutions to the general problem, and we conduct simulations based on real platform parameters to evaluate the performance of these algorithms.

Key-words: Edge-cloud, maximum stretch, NP-completeness, algorithms

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Minimization du stretch maximum sur une plate-forme edge-cloud

Résumé : Nous considérons l'ordonnancement de jobs indépendants qui sont générés par des processeurs *edge*. Ces jobs peuvent soit être exécutés localement, soit envoyés à une plate-forme *cloud* plus puissante, mais il faudra alors prendre en compte les communications montantes et descendantes. Ce type de plate-forme à deux niveaux *edge-cloud* a de nombreuses applications. Le problème est de décider où et quand exécuter chaque job, avec pour objectif de minimiser le *stretch* maximal. Le stretch d'un job est le rapport du temps passé par le job dans le système sur le temps qu'il aurait passé s'il avait été seul dans ce système, et mesure donc un facteur de ralentissement dans le temps de réponse. Nous formalisons le problème en expliquant les différences avec des modèles classiques. Nous prouvons des résultats de NP-complétude sur la complexité du problème *offline*, à partir d'un nouveau résultats sur un problème classique et ouvert, celui de la minimisation du stretch avec des processeurs homogènes et sans dates d'arrivées (*release dates*) des jobs. Nous proposons plusieurs heuristiques pour le problème général *online*, que nous comparons à l'aide de simulations basées sur des paramètres de plates-formes existantes.

Mots-clés : Edge-cloud, stretch maximum, NP-complétude, algorithmes

1 Introduction

Edge-Cloud computing is a recent paradigm that is currently deployed by many vendors (see [19, 20, 21, 27] and many others). The idea is to execute some jobs in-situ, directly on the edge server where they originate from, thereby providing a flexible, cost-effective and decentralized solution that dramatically reduces the volume of data transfers. But some jobs may be too demanding in terms of computing power, or some edge servers may be overloaded because they launch too many jobs. This calls for coupling the edge server with a powerful cloud platform, where some jobs can be delegated whenever needed.

Deciding which jobs should be executed locally and which ones should be communicated (up and down) to the cloud platform is a challenging problem. The main focus of this paper is to lay the foundations for solving the instance of the problem dealing with response time as a single optimization criterion. Response time is a very important criterion, in particular for the users, and arguably the most important one in a real-time framework. But it is not the unique criterion to optimize on an edge-cloud platform: energy consumption and resource costs are important criteria too. However, we show that dealing with response-time is already an intricate problem, and we leave multi-objective optimization for further work.

The *response time* for a job [12, 28] (also called *flow time* in the scheduling literature [8]) is the time spent by that job in the system, starting from its release date and up to final completion (including output transfers if any). The standard scheduling objective is to minimize the maximum response time over all jobs[1]. However, maximum response time is not appropriate to ensure fairness, because giving the same response time for all jobs results in worst performance for short jobs, compared to the performance achieved for long ones. In order to provide a fair processing of jobs, job lengths should be taken into account. The *stretch* [11, 3] is the metric used to ensure fairness among jobs, and it is defined as the response time normalized by the job length. More precisely, the stretch is the response time divided by the best possible execution time for the job if it were alone on the platform. Therefore, a job stretch measures how the performance of a job is degraded compared to a system dedicated exclusively to this job. The *maximum stretch* is the maximum of the stretches of all jobs[2] and provides a measure of the responsiveness of the system: it quantifies the user expectation that the response time should be proportional to the load incurred by the execution of the job by the system. For an illustration, consider two jobs released at the same time, one lasting 1 hour and the other 10 hours. With a single processor, executing the long job first leads to a maximum stretch of 11, while executing the short job first leads to a maximum stretch of 1.1. Intuitively, the latter scheduling is more fair to users than the former. A word of caution: assume now that we have one edge processor and one cloud processor for the two jobs. For each of them, the time that it would spend in a dedicated system

[1] Minimizing the average response time of all jobs, or equivalently the sum of the response times of all jobs (also called *total flow time*), has been extensively studied too [8].

[2] Just as for response time, the average stretch has also been studied as a metric [5].

must be computed as the minimum of its execution times on the edge and on the cloud (including transfers), if it was the only job.

In this paper, we focus on the maximum stretch as the optimization metric, and we investigate how to minimize the maximum stretch of independent jobs executing on an edge-cloud platform. This is an online scheduling problem, as the jobs have release dates that are unknown until the jobs are submitted. The first task is to design a realistic model for such a platform. In a nutshell, we propose a model with preemption and possible re-execution, but no migration: jobs are either executed on the edge processor where they originate from, or on a cloud processor, and can be dynamically re-assigned to another resource. If this is case, the execution must then restart from scratch, thus the time spent executing the job until re-assignment is lost. Note that migration would require some kind of checkpoint mechanism to be able to restart the job on another resource from its current state.

If a job is executed on the cloud, up and down transfers are accounted for. We allow for computation and communication overlap, and many communications can occur in parallel across edge and cloud processors. However, we enforce the one-port full-duplex communication model that states that any resource cannot be involved either in two sending operations, or in two receiving operations, at the same-time step, but sending and receiving simultaneously is allowed. Independent sender/receiver pairs can communicate in parallel. A given message may be preempted (and resumed later) if the sender or the receiver has a more urgent message to process (because of the release of a new job). This communication model [16, 18, 32, 33] has been advocated as being much more accurate than the traditional *macro-dataflow* communication model of the scheduling literature, which allows a processor to, say, send 100 different messages in parallel without accounting for any bandwidth limitation.

After designing the model, we assess complexity results. The offline problem is shown to be NP-hard, even in the simpler instance with no release dates (all jobs are known in advance). This result comes from the proof that minimizing the max-stretch with homogeneous processors and without release dates is NP-complete, a problem whose complexity was left open before this work. Given these negative (but somewhat expected) results, we introduce several heuristics for the general online problem, and compare their performance through simulations. In summary, the main contributions of this paper are the following:

- A realistic model for scheduling jobs on edge-cloud platforms;
- Several complexity results for the offline problem, proving in particular the NP-hardness of an open problem;
- New heuristics and their experimental comparison for the online problem.

The rest of the paper is organized as follows. We first discuss related work in Section 2. The detailed model of the edge-cloud platform is provided in Section 3, along with the MinMaxStretch-EdgeCloud problem definition.

We then prove the NP-completeness of MinMaxStretch-EdgeCloud in Section 4, and we design several heuristic algorithms to solve this problem in Section 5. Finally, extensive simulations are presented in Section 6 to evaluate and compare the algorithms. Conclusions and directions for future work are presented in Section 7.

2 Related work

We discuss related work along three topics: edge-cloud platforms (Section 2.1), communication models (Section 2.2), and maximum stretch (Section 2.3).

2.1 Edge-cloud platforms

A range of applications of edge computing is given in [7], and these include e-health, disaster recovery, autonomous vehicles or flying drones. Although no specific model is given, a state of the art is established with a list of objectives and constraints that have been studied, such as the delay, the bandwidth, the energy, QoS-assurance, etc. Some challenges of edge computing are also discussed in [22].

Some papers influenced our choices related to the model. For example, we chose a model that allows preemption but not migration. This was influenced by papers such as [1], where it is shown that allowing migration does not affect the efficiency of the algorithms that minimize the average flow time, whereas allowing preemption does. This is also proven for the minimization of the average stretch in [2].

We also reviewed papers specifically to design our edge/cloud model. For example, in [25], even though the concern is not exactly the same as ours, a model is proposed for mobile-edge computing systems. Our model is simpler than theirs because we do not consider the power consumption, whereas they do. Existing models include those from [30] and [29], where the focus is on the placement of data stream processing applications. However, this latter approach is application-specific, which our model aspires to be agnostic of the nature of the application.

Finally, we decided to consider heterogeneous edge processors but kept homogeneity inside the cloud platform. This decision is motivated by the fact that users can always request identical virtual machines on cloud platforms. On the theoretical side, dealing with heterogeneous processors might be an obstacle to finding competitive algorithms: in some cases [15], having heterogeneous processors invalidates the theoretical bounds of some usual scheduling algorithms. However, it is not difficult to extend our model with heterogeneous cloud processors, and all the algorithms can be modified in a straightforward manner to handle a fully heterogeneous edge-cloud platform.

2.2 Communication model

Contention-aware task scheduling has been considered since many years in the scheduling literature [16]. In particular, [32, 33] showed through simulations that taking contention into account is essential for the generation of accurate schedules. They investigate both end-point and network contention. Here end-point contention refers to the bounded multi-port model [17]: the volume of messages that can be sent/received by a given processor is bounded by the limited capacity of its network card. Network contention refers to the one-port model, which has been advocated by [6] because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, all these operations "are eventually serialized by the single hardware port to the network." Experimental evidence of this fact has been reported by [31], which reports that asynchronous sends become serialized as soon as message sizes exceed a few megabytes. The one-port model fully accounts for end-point contention. Coupled with message preemption, it provides a very realistic model for communications.

2.3 Stretch

The stretch is a particular case of weighted response time [11] and has been studied extensively because of its fairness. Here is a quote from [5]: "Flow time measures the time that a job is in the system regardless of the service it requests; the stretch measure relies on the intuition that a job that requires a long service time must be prepared to wait longer than jobs that require small service times." An interesting analogy is made in [13] between maximum stretch and distributive justice, defined as the perceived fairness in the way costs and rewards are shared within a group of individuals.

Stretch optimization was studied for independent jobs without preemption [4], bag-of-tasks applications [24, 10], multiple parallel task graphs [9], and for sharing broadcast bandwidth between client requests [34].

On the algorithmic side, [26] proposes an algorithm called Shortest Remaining Processing Time (SRPT), that is an approximation for the online case with the average stretch as an objective. It is a greedy approach. Then [3] proposes an algorithm for minimizing the maximum stretch in the online case of only one processor. This algorithm is optimal for the offline problem, and is Δ-competitive for the online problem, where Δ is the ratio between the longest and the shortest job. Another version of this algorithm in given in [4], with a better time complexity but similar bounds on the resulting stretch.

3 Model

We introduce the framework in Section 3.1, we survey schedule constraints in Section 3.2, and we work out an example in Section 3.3.

3.1 Framework

We consider a two-level platform, with P^c homogeneous processors in a cloud, and P^e edge computing units. The cloud processors have a speed normalized to 1, while edge computing units run at a slower pace. Hence, the j-th edge computing unit, with $1 \leq j \leq P^e$, is operating at a speed $s_j \leq 1$.

The application consists in n independent jobs $J_1, \ldots, J_n$, where each job is issued from an edge computing unit. Job J_i is generated by the edge computing unit o_i (with $1 \leq o_i \leq P^e$), and this edge computing unit must obtain the result of job J_i, either by executing J_i locally, or by delegating the job to the cloud, and getting the result back. Parameters for job J_i are as follows:

- o_i is the origin processor on the edge ;
- w_i is the number of operations required by the job;
- r_i is the release date;
- up_i and dn_i are the communication times required to send the job to the cloud and get the result back (uplink/downlink communications).

The execution time of job J_i hence depends whether it is executed directly at the edge processing unit where the job originates from, or whether it is sent for an execution on the cloud. If the job is executed on the edge, it takes a time $t_i^e = \frac{w_i}{s_{o_i}}$, since the edge processing unit o_i operates at speed s_{o_i}. Computation may be faster on the cloud, but we must then account for communication time, hence an execution on the cloud takes a time $t_i^c = up_i + w_i + dn_i$ (send the job up to the cloud, compute at speed 1, and get the result back down). In both cases, the execution of the job can be preempted, which means that we can interrupt its execution (to schedule another job), and resume it at a later time. Once started on a given resource, the execution cannot migrate to another resource, but re-execution from scratch is possible (and the time spent up to re-execution is lost).

Let f_i denote the time at which the execution of J_i is completed. Then, the *stretch* S_i of job J_i is defined by:

$$S_i = \frac{f_i - r_i}{\min(t_i^e, t_i^c)}.$$

Indeed, job J_i is released at time r_i and spends a time $f_i - r_i$ in the system, while it could have taken a time $\min(t_i^e, t_i^c)$ in the best case of a dedicated system. Hence, the stretch of each job is equal to one if the job is executed with the minimum possible time, and we want the maximum stretch (over all jobs)

to be as close to one as possible. Overall, the objective function is to minimize the *maximum stretch*, which is defined as $\max_{1 \leq i \leq n} S_i$.

While several jobs may be competing for cloud resources, several constraints must be enforced within a schedule of jobs. We overlap computations and communications, and we consider communication channels to be full-duplex between an edge processor and a cloud processor, hence it is possible to perform in parallel a computation, an uplink communication, and a downlink communication. However, communications involving a common processor must be sequentialized: if two jobs originating from the same edge processing unit are delegated to the cloud, we cannot perform the two uplink (and then downlink) communications in parallel. Similarly, if two jobs (that may come from different edge processing units) are sent to the same cloud processor, their communications (uplink and then downlink) must be sequentialized.

Furthermore, communications, just as computations, are preemptive, which means that we can interrupt a communication (for instance, to schedule a communication for a smaller job), and resume the interrupted communication at a later time.

Optimization problem: The goal is to find a schedule that respects all constraints, with the aim of minimizing the maximum stretch. This problem is denoted as MinMaxStretch-EdgeCloud.

3.2 Schedules

For each job J_i, we first need to decide where it is executed: $alloc(i) = 0$ if the job is executed on its local edge processor o_i, otherwise we set $alloc(i) = k$, where k is the cloud processor on which J_i is executed ($1 \leq k \leq P^c$). Formally, a schedule consists in sets of disjoint execution intervals E_i for each job (recall that we allow preemption), and disjoint uplink/downlink communication intervals for jobs executed on the cloud ($alloc(i) \neq 0$), $U_i(o_i, alloc(i))$ and $D_i(alloc(i), o_i)$.

Several constraints must be ensured for the schedule to be valid. In particular, for any two jobs $J_i, J_{i'}$ (with $1 \leq i, i' \leq n$), executed on the same processor, i.e., $alloc(i) = alloc(i') \neq 0$ (cloud), or $alloc(i) = alloc(i') = 0$ and $o_i = o_{i'}$ (edge), all their execution intervals must be disjoint:

$$\forall I \in E_i, \forall I' \in E_{i'},\ I \cap I' = \emptyset.$$

We must also ensure that communications are serialized, i.e., for any two sets $U_i(j,k)$ and $U_{i'}(j',k')$ (resp. $D_i(k,j)$ and $D_{i'}(k',j')$), if $j = j'$ or $k = k'$, then the communication originates from or targets the same processor, and hence all intervals must be pairwise disjoint. Finally, we must ensure that all computations and communications are done, in the correct order. Given a set of intervals E, we denote by $\min(E)$ (resp. $\max(E)$) the smallest (resp. largest) extremity of all intervals in E. Hence, for job J_i ($1 \leq i \leq n$):

- If $alloc(i) = 0$, then $\sum_{I \in E_i} |I| \geq \frac{w_i}{s_{o_i}}$;
- Otherwise,

- $\sum_{I \in E_i} |I| \geq w_i$;
- $\sum_{I \in U_i(o_i, alloc(i))} |I| \geq up_i$;
- $\sum_{I \in D_i(alloc(i), o_i)} |I| \geq dn_i$;
- we must complete uplink communications before starting computation: $\max(U_i(o_i, alloc(i))) \leq \min(E_i)$;
- and we must complete computation before starting downlink communications: $\max(E_i) \leq \min(D_i(alloc(i), o_i))$.

3.3 Example

An example of a valid execution with one edge processor and one cloud processor is depicted in Figure 1, where J_1, J_4 and J_6 are executed on the edge, while J_2, J_3 and J_5 are sent to the cloud. Job parameters are as follows, where the speed of the edge processor is $\frac{1}{3}$:

- J_1: $r_1 = 0$, $w_1 = 1$, $up_1 = dn_1 = 5$;
- J_2: $r_2 = 0$, $w_2 = 4$, $up_2 = dn_2 = 2$;
- J_3: $r_3 = 3$, $w_3 = 2$, $up_3 = 2$, $dn_3 = 1$;
- J_4: $r_4 = 5$, $w_4 = 4/3$, $up_4 = dn_4 = 5$;
- J_5: $r_5 = 5$, $w_5 = 2$, $up_5 = 2$, $dn_5 = 1$;
- J_6: $r_6 = 6$, $w_6 = 1/3$, $up_6 = dn_6 = 5$.

The execution intervals, as well as uplink/downlink communication intervals, are depicted. At time-step 6, we compute at the same time on the cloud, on the edge, and we perform an uplink communication and a downlink communication. Also, this is the time when J_6 preempts job J_4, since it can be executed locally within one time unit, while J_4 is a longer job and will be delayed only by one time unit.

One can check, by an exhaustive look at all possible schedules, that this one is optimal. Indeed, jobs J_1 and J_6 run at their minimum possible time $w_i \times 3$ on the edge, while executing them on the cloud would have cost at least 10 units

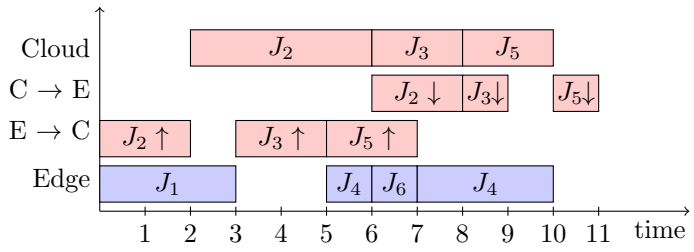


Figure 1: Example of a valid schedule, with $P^e = P^c = 1$.

of communication time, hence they have a stretch of 1. On the other hand, J_2 would take a time 12 on the edge, and it is sent to the cloud where the total time is $up_2 + w_2 + dn_2 = 8$, hence also a stretch of 1. Next, consider J_3 and J_5, which have the same characteristics: they can be executed in time 6 on the edge, but 5 on the cloud. Here, we execute them on the cloud, where they both experience one time-step of delay (waiting for the processor to be available), hence they have a stretch of $\frac{6}{5}$, and they could not have a better stretch on the edge. Finally, job J_4 takes a time 5 on the edge, while its minimum time is 4, because it is preempted at time-step 6 to execute the shorter job J_6 that would greatly impact the stretch if it was delayed. On the cloud, it would have taken a time at least $10 + \frac{4}{3}$, hence much greater. Its stretch is $\frac{5}{4}$, which determines the maximum stretch for this example.

Throughout this example, we see that decisions are more difficult to take when there is no knowledge about jobs that will be released in the future. For instance, one could schedule job J_3 either on the edge or on the cloud, since it would complete in both cases within time 9, but depending on the jobs that come next (computation-intensive vs communication-intensive), one decision would be better than the other. The *online* case is the problem where jobs are not known in advance, while in the *offline* case, all job parameters are known in advance. In the following, we prove that the problem is difficult even in the offline case, and then we derive heuristics to address the general online problem in Section 5.

4 Problem complexity

This section is devoted to assess the study of the complexity of MinMaxStretch-EdgeCloud in the offline case. To shorten notations, we write MMSECO instead of MinMaxStretch-EdgeCloud-Offline throughout this section, and MMSECO-Dec is the corresponding decision problem (is it possible to achieve a target maximum stretch?). First, we prove that MMSECO-Dec is in NP in Section 4.1. Then, we derive two main complexity results, that remain true even without release dates (consider that all jobs are released at time 0). The results are the following:

- For a fixed number of processors, MMSECO-Dec is NP-complete in the weak sense (Section 4.2).
- For a variable number of processors, MMSECO-Dec is NP-complete in the strong sense (Section 4.3).

4.1 NP membership

Lemma 1. MMSECO-Dec *is in NP.*

Proof. A solution to MMSECO-Dec is a sequence of events, where an event is defined with the following list:

- a job J_i being released;
- a job J_i finishing its uplink communication;
- a job J_i finishing its execution;
- a job J_i finishing its downlink communication.

In between two events, there is no reason to reconsider any decision taken, nor to preempt any computation or computation. There are only two events for the jobs executed on edge processors, and four for those executed on a cloud processor, so there are at most $4n$ events. Therefore, a schedule can be represented in polynomial space: at each of these $4n$ time-steps, there might be at most one event per processor. The validity of the solution can then be verified in polynomial time by checking each constraint for a valid schedule, as detailed in Section 3.2. Therefore, MMSECO-Dec is in NP. □

4.2 Weak NP-completeness with two processors

We are now ready to prove the NP-completeness of MMSECO-Dec. We first focus on minimizing the *maximum stretch* for a fixed number of homogeneous processors, and without release dates (hence forgetting about the cloud-edge system, called MMSH problem), and the associated decision problem MMSH-Dec. The execution of job J_i hence takes w_i time units, and there are no communications. To the best of our knowledge, the complexity of MMSH has not been established yet, and we prove its NP-completeness. In a second step, we show that it is easy to create an instance of MMSECO-Dec from a general instance of MMSH-Dec, hence proving the NP-completeness of MMSECO-Dec.

As a preliminary, we derive an interesting result about the optimal order of jobs in a schedule for MMSH: with a single processor, jobs should be ordered from the shortest to the longest (i.e., by non-decreasing w_i's).

Lemma 2. *For* MMSH *with a single processor, there always exists a schedule that minimizes the* maximum stretch *by ordering the jobs from the shortest to the longest, and without preemption.*

Proof. First, we observe that preemption is not useful when all jobs have the same release date (which is 0 here). To see this, assume we execute a sequence with preemption $(J_i^{(1)}, J_j^{(1)}, J_i^{(2)})$ of three job chunks, where J_i and J_j are two different jobs. Here, $J_i^{(1)}$ and $J_i^{(2)}$ are two chunks of job J_i, and $J_j^{(1)}$ is one chunk of job J_j. Swapping the first two chunks leads to the sequence $(J_j^{(1)}, J_i^{(1)}, J_i^{(2)})$. Because J_i and J_j have the same release date, this swap is valid: it does not change the stretch of J_i and can only decrease or leave unchanged the stretch of J_j. After a finite number of such swaps, we have a non-preemptive schedule whose stretch does not exceed that of the original schedule with preemption.

Then, to prove that the jobs can be ordered as stated, we consider a schedule that does not order the jobs from the shortest to the longest, and we construct

another schedule with fewer mis-orderings, and which has a maximum stretch that is smaller than or equal to the original maximum stretch. Let $(k_1, \ldots, k_n)$ be (the indices of) the ordering of the jobs in the initial schedule, meaning that we execute J_{k_1} first and J_{k_n} last. By assumption, there exists i such that $w_{k_i} > w_{k_{i+1}}$ (i.e., job J_{k_i} is longer than job $J_{k_{i+1}}$). We swap these two jobs, which is possible because all jobs have same release dates, and which does not change the stretch of the other jobs.

Since we consider the maximum stretch of the schedule, let us compare the maximum between the stretches of the two jobs before and after the swap. If X is the starting time of the earlier job, then the values that we consider are:

$$S_b = \max\left(\frac{X + w_{k_i}}{w_{k_i}}, \frac{X + w_{k_i} + w_{k_{i+1}}}{w_{k_{i+1}}}\right) \text{ before the swap;}$$

$$S_a = \max\left(\frac{X + w_{k_{i+1}}}{w_{k_{i+1}}}, \frac{X + w_{k_{i+1}} + w_{k_i}}{w_{k_i}}\right) \text{ after the swap.}$$

As we have $\frac{X+w_{k_{i+1}}}{w_{k_{i+1}}} < S_b$ and $\frac{X+w_{k_{i+1}}+w_{k_i}}{w_{k_i}} < S_b$, we indeed get $S_a < S_b$, so the stretch after the swap is at most the stretch before the swap. We can repeat the operation until there are no mis-orderings anymore, and we get a schedule that orders the jobs from the shortest to the longest and has a maximum stretch less than or equal to that of the initial schedule. □

Theorem 1. MMSH-Dec *with two homogeneous processors is NP-complete in the weak sense.*

Proof. It is easy to see that MMSH-Dec is in NP, since it is a simpler case of MMSECO-Dec (see Section 4.1). We now prove that it is NP-complete.

Let I_1 be an instance of 2-Partition-Eq [14], which is a special case of 2-Partition where the partitions must have equal cardinality: Given $2n$ integers $\{a_1, \ldots, a_{2n}\}$ with $2S = \sum_{i=1}^{2n} a_i$, does there exist a subset $I \in \{1, \ldots, 2n\}$ with $|I| = n$ and $\sum_{i \in I} a_i = S$?

We create an instance I_2 of MMSH-Dec with two processors and $2n + 2$ jobs, such that:

- for $1 \leq i \leq 2n$, the execution time of job J_i is $w_i = nS + a_i$;
- the execution time of the two additional jobs J_{2n+1} and J_{2n+2} is $w_{2n+1} = w_{2n+2} = (n + 1)S$.

We ask whether it is possible to achieve a stretch of $\frac{n^2+n+2}{n+1}$.

Note that, building upon Lemma 2, we know that each processor will sort its jobs by non-decreasing execution time in an optimal solution, so a schedule is characterized only by a partition P_1, P_2 of the jobs. We now show that I_1 has a solution if and only if I_2 has a solution.

- Consider first that I_1 has a solution, I. We consider the schedule with P_1 containing jobs J_i with $i \in I$ and job J_{2n+1}, while P_2 contains all other

jobs (J_i, $i \notin I$ and J_{2n+2}). We prove that this schedule has a max-stretch equal to $\frac{n^2+n+2}{n+1}$.

First, note that the sum of job weights in each set is:

$$\sum_{i \in I}(nS + a_i) + (n+1)S = n^2S + S + (n+1)S = (n^2 + n + 2)S,$$

since there are exactly n jobs in set $|I|$ (and the same number of jobs for P_2).

By Lemma 2, job J_{2n+1} (resp. J_{2n+2}) is executed last in P_1 (resp. P_2), since $w_{2n+1} = w_{2n+2} > w_i$ for all $1 \leq i \leq 2n$. Therefore, these additional jobs both complete at time $(n^2+n+2)S$, and they have a stretch $\frac{n^2+n+2}{n+1}$.

For any other job J_i, the stretch is $S_i = \frac{knS+X}{nS+a_i}$, where there are $k \leq n$ jobs before J_i (including J_i) with a total weight $knS + X$, where $X = \sum_{j \in \texttt{Pred}(i)} a_j$. Here, $\texttt{Pred}(i)$ denotes the indices of the k jobs that precede J_i, and we have $X \leq S$. We have the following inequalities:
(i) $\frac{knS+X}{nS+a_i} \leq \frac{(n^2+1)S}{nS+a_i} \leq \frac{n^2+1}{n}$ (constraints on k and X), and
(ii) $\frac{n^2+1}{n} \leq \frac{n^2+n+2}{n+1}$ (for any $n \geq 1$).

Therefore, job J_i has a stretch $S_i \leq \frac{n^2+n+2}{n+1}$, meaning that the max-stretch of the schedule is $\frac{n^2+n+2}{n+1}$, and hence I_2 has a solution.

- Consider now that I_2 has a solution: Let P_1, P_2 be a schedule with max-stretch $\frac{n^2+n+2}{n+1}$ or less. We prove that P_1, P_2 is a partition of the jobs into two sets of equal sum and equal size, with $J_{2n+1} \in P_1$ and $J_{2n+2} \in P_2$.

  We first prove that each processor finishes its whole execution at a time t_{finish} such that $t_{finish} \leq (n^2+n+2)S$. Let J_i be the last job to finish, its stretch is $S_i = \frac{t_{finish}}{w_i}$. Since $w_i \leq (n+1)S$ for all jobs, $S_i \geq \frac{t_{finish}}{(n+1)S}$. Hence, since P_1, P_2 is a solution to I_2, it means that $S_i \leq \frac{n^2+n+2}{n+1}$ and therefore $t_{finish} \leq (n^2+n+2)S$. Furthermore, since the sum of all execution times is $2(n^2+n+2)S$, this means that both processors finish at exactly time $t_{finish} = (n^2+n+2)S$.

  Now, we specifically look at the two jobs that finish their execution at time $(n^2+n+2)S$, one on each processor. We prove that these two jobs are J_{2n+1} and J_{2n+2}. For their stretch to be at most $\frac{n^2+n+2}{n+1}$, they need to have an execution time $w_i \geq (n+1)S$. The only jobs for which this constraint holds are J_{2n+1} and J_{2n+2}. So, we indeed have J_{2n+1} and J_{2n+2} that are scheduled on different processors, say J_{2n+1} in P_1 and J_{2n+2} in P_2 (the two jobs are identical).

  Let us now consider the jobs in P_1, excluding the large job J_{2n+1}. Their total weight is hence $t_{finish} - (n+1)S = n \times nS + S$, which means that there are exactly n jobs (the a_i's are small in comparison with nS). Since their sum is $n^2S + S$, we obtain that $\sum_{i \in P_1 \setminus \{2n+1\}} a_i = S$, and the set of jobs from P_1 is a solution to 2-PARTITION-EQ, and I_1 has a solution.

Overall, I_1 has a solution if and only if I_2 has a solution, and the construction of I_2 from I_1 can obviously be done in polynomial time, hence MMSH-DEC is NP-complete. This NP-completeness result is established in the weak sense since 2-PARTITION-EQ can be solved by a pseudo-polynomial algorithm. □

We can now derive the result for the original cloud-edge problem, by performing a simple reduction from MMSH-DEC.

Corollary 1. MMSECO-DEC *is NP-complete in the weak sense, even with one edge processor, one cloud processor, and no release dates.*

Proof. By Lemma 1, MMSECO-DEC is in NP. We perform a reduction from MMSH-DEC with two processors, where $P^e = P^c = 1$, the speed of the edge processor is set to 1, and all communication costs of jobs are set to $up_i = dn_i = 0$. The problem is then exactly equivalent to the original instance, which concludes the proof. □

4.3 Strong NP-completeness for a variable number of processors

Theorem 2. MMSH-DEC *with a variable number of homogeneous processors is NP-complete in the strong sense.*

Proof. We already showed that MMSH-DEC is in NP when proving Theorem 1. We now prove that it is strongly NP-hard, i.e., it remains NP-hard even if the input parameters are bounded above by a polynomial in p, where p is the number of processors.

Let I_1 be an instance of 3-PARTITION [14]: Given $3n$ integers $\{a_1, \ldots, a_{3n}\}$ whose sum is $nB = \sum_{i=1}^{3n} a_i$ and with $\frac{B}{4} < a_i < \frac{B}{2}$ for $1 \leq i \leq 3n$, does there exist a partition of $\{1, \ldots, 3n\}$ into n subsets $S_1, \ldots, S_n$ such that $\sum_{i \in S_i} a_i = B$ for $1 \leq i \leq n$? We create an instance I_2 of MMSH-DEC with n processors and $4n$ jobs as follows:

- For $1 \leq i \leq 3n$, the execution time of job J_i is $w_i = a_i$;
- For $3n + 1 \leq i \leq 4n$, the execution time of job J_i is $w_i = \frac{B}{2}$; these n additional jobs are hence larger than any other jobs;
- The bound on the max-stretch is set to 3.

We now prove that I_1 has a solution if and only if I_2 has a solution, and the solution of I_2 always maps exactly one of the additional large jobs on each processor.

- Consider first that I_1 has a solution, $\{S_1, \ldots, S_n\}$, with $\sum_{i \in S_i} a_i = B$. Given the constraints on the a_i's, there are exactly three elements per set S_i. We consider the schedule with jobs from S_i and one additional job on processor i, hence four jobs per processor. As already stated, jobs are scheduled from the shortest to the longest, and therefore, the k-th job

always has a stretch that is at most k. Hence, the first three jobs have a stretch at most 3 (equal to three if the three jobs have a weight $\frac{B}{3}$, and then the third job has a stretch $\frac{B}{B/3} = 3$). The last job finishes at time $3B/2$ and it has a length $B/2$, hence it has a stretch of 3. Finally, each processor has a stretch of 3, hence we have constructed a solution to I_2.

- Consider now that I_2 has a solution, that is a partition of jobs onto processors, $\{P_1, \ldots, P_n\}$, such that the max-stretch is at most 3. Since the max-stretch is 3 and all jobs are of length at most $\frac{B}{2}$, no job can finish after $\frac{3B}{2}$ and only a job of length $B/2$ can finish at $\frac{3B}{2}$ (a smaller job would lead to a larger stretch). Furthermore, the total work is $\frac{3nB}{2}$ and there are n processors, so each processor finishes exactly at time $\frac{3B}{2}$, with one of the n large jobs. Consider $S_i = P_i \setminus \{k \mid k > 3n\}$: S_i consists in three jobs of sum B (since all job weights are between $B/4$ and $B/2$), hence the S_i's are a solution to I_1.

To conclude, I_1 has a solution if and only if I_2 has a solution, and we have performed in polynomial time the reduction from 3-Partition that is NP-hard in the strong sense, therefore MMSH-Dec is NP-complete is the strong sense when considering a variable number of homogeneous processors. □

Corollary 2. MMSECO-Dec *with an arbitrary number of processors is strongly NP-complete.*

Proof. By Lemma 1, MMSECO-Dec is in NP. We perform a reduction from MMSH-Dec with p processors, where $P^e = 1$, $P^c = p-1$, the speed of the edge processor is set to 1, and all communication costs of jobs are set to $up_i = dn_i = 0$. The problem is then exactly equivalent to the original instance, since all jobs originate from the only edge processor, and they can be scheduled either on the edge processor or at one of the cloud processors, and each of these processors (cloud and edge) behave similarly; the platform is hence fully homogeneous. □

5 Algorithms

Since MinMaxStretch-EdgeCloud is NP-complete even in the offline case, we design general heuristic algorithms to address the problem in the most general online setting, with P^e different edge computing units generating jobs, that can be processed either locally, or on one of the P^c cloud processors.

We first propose a strategy that does not exploit the cloud platform, but executes all jobs locally following a competitive strategy (Section 5.1). We then design a simple greedy algorithm that greedily decides whether to execute a job locally or on the cloud (Section 5.2). Finally, we revisit efficient algorithms from the literature to adapt them to the edge-cloud setting, building upon the shortest remaining processing time of a job in Section 5.3, and on an earliest deadline first strategy in Section 5.4.

The algorithms are event-based, i.e., we reconsider decisions only when an event occurs, so that they all remain of polynomial cost. There are at most $4n$ events, corresponding to the time-steps at which job J_i, for $1 \leq i \leq n$,

1. is released at edge processor o_i (event $\mathscr{E}_r(i)$);
2. completes its execution on processor $alloc(i)$ (event $\mathscr{E}_f(i)$);
3. completes an uplink communication (event $\mathscr{E}_{up}(i)$);
4. completes a downlink communication (event $\mathscr{E}_{dn}(i)$).

5.1 Baseline: Edge-Only strategy

We first consider a simple heuristic, *Edge-Only*, that does not use the cloud platform, but where all jobs are executed locally on the edge. This might actually be a good strategy when edge processing units have a good processing speed and/or when communications are costly.

Since all edge processors are independent with this strategy, the problem consists in minimizing the max-stretch on one processor, and this is done independently at each edge processing unit. We use the Stretch-So-Far Earliest-Deadline-First algorithm, designed by Michael Bender et al. [3]. Indeed, this algorithm is Δ-competitive with a single processor, where Δ is the ratio between the longest and the shortest job.

Note that we have to modify the algorithm of [3] to account for the edge-cloud framework: when computing the stretch of a job, we consider its potential execution time on the cloud. We will compare heuristics exploiting the cloud with this edge-only strategy in Section 6.

5.2 Greedy

We design a greedy heuristic, called *Greedy*, that greedily schedules first the job that would currently achieve the highest stretch, as it is the job that might impact most the maximum stretch. Decisions are taken only when an event occurs (new job released, job completion, completion of an uplink or downlink communication).

The algorithm works as follows: at each event, as long as there are available resources, we compute for each job the minimum stretch that can be achieved using an available resource immediately. We select the job that maximizes this value, and execute the job on the resource (edge or cloud) on which it achieves the minimum stretch.

The complexity of the routine that runs at each event is $O(n(1 + P^c))$: compute the stretch for each job, for either a local execution, or the execution on one of the P^c cloud computing units. Since there are at most $4n$ events, the worst case complexity is $O(n^2P^c)$. Of course, in an online setting, we expect the number of jobs that are simultaneously processed in the system to be far less than n. Hence, the average complexity is expected to be much lower.

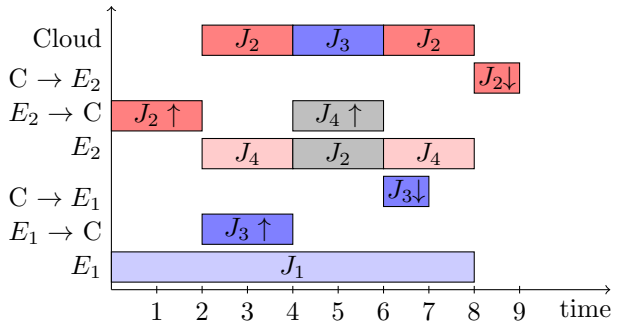


Figure 2: Example of a schedule obtained with *Greedy*.

We give an example with $P^c = 1$, $P^e = 2$, $n = 4$, and both edge speeds $s = 0.5$. Two jobs are released at time 0, J_1 on the first edge processing unit E_1, and J_2 on the second edge processing unit E_2. Both jobs have the same parameters: $w = 4$, $up = 2$, and $dn = 1$. A stretch of 1 can be obtained by scheduling these jobs on the cloud, but since there is only one cloud processing unit, one of the jobs is executed locally, say J_1, on the first edge processor E_1, with a stretch $\frac{8}{7}$. At time 2, J_2 completes its uplink communication (event $\mathscr{E}_{up}(2)$), and a new job spawns at each processor (J_3 on E_1 and J_4 on E_2), with $w = 2$, $up = 2$, $dn = 1$ (events $\mathscr{E}_r(3)$ and $\mathscr{E}_r(4)$). Both of these new jobs have a minimum stretch of 1 with a local execution, and J_2 still has a minimum stretch of 1 as well. However, J_1 has the maximum stretch and it pursues its execution on E_1, while J_3 is sent on the cloud. J_4 begins its execution on E_2 expecting a stretch of 1 if it is not preempted. At time 4, corresponding to event $\mathscr{E}_{up}(3)$, the maximum minimum stretch is obtained by J_3 ($\frac{5}{4}$), which preempts J_2 on the cloud. As the cloud is currently unavailable, J_2 takes the priority on E_2, thus preempting J_4. When the cloud is available at time 6, J_2 resumes its execution on the cloud, as it is expected to end faster on the cloud. It finishes at time 9 with a stretch of $\frac{9}{7}$. The max-stretch is finally reached by J_4, that can resume its execution on E_2 after being delayed by J_2, and that completes at time 8, with a stretch of $\frac{3}{2}$, see Figure 2. The parts of execution or communication that were not used because of preemption are in grey.

5.3 Shortest Remaining Processing Time

The *SRPT* heuristic builds on the classical Shortest Remaining Processing Time strategy, which assigns to a processing unit the job that it can finish the earliest, so that shortest jobs will be executed with little delay. Again, decisions are taken at each event.

The incentive for the use of this algorithm is that it is a $O(1)$-competitive algorithm for the average stretch problem, as shown in [26]. However, this is not the case for the maximum stretch: a long job could be blocked by short jobs for an arbitrary long duration. Note that migration is not allowed, but full re-execution is possible, thus a job that has been preempted by another job

might start again (from scratch) on another processor, if it becomes the job that will finish first on that other processor.

The algorithm takes new decisions at each event. As long as we still have a processor on which we can execute a job, we choose the job that can be completed the earliest and execute it on the processor that can complete it the earliest, and we remove this job and processor from the list of available jobs/processors. The algorithm is detailed in Algorithm 1, where we handle a priority queue with jobs ordered by release dates (*releases*), and we keep track of the next release event *nextRelease*, corresponding to an event $\mathscr{E}_r(i)$, with $1 \leq i \leq n$. Also, *nextFinish* corresponds to the next completion event, and it can be either the completion of a job, or the completion of a communication. Once the time of the next event is known (t_{new}), we compute the remaining work and communication at this time, given that processors have been working and communicating for a time $t_{new} - t$. These updates are done through the procedure **execute**, which is detailed in Algorithm 2, where initially $w^e_{i,k} = w^e_i = w_i$, $up^c_i = up_i$ and $dn^c_i = dn_i$, and then these values may decrease when some work has already been completed for job i. Note that at the first iteration of the loop, all the arrays *executing*, *upcom* and *downdom* are empty, hence **execute** does nothing. We will effectively start scheduling jobs once the first job release has been done, at the first time t_{new}. Then, we compute all possible finish times of jobs (on edge or cloud) and decide of the SRPT schedule by calling the **SRPT-schedule** routine, see Algorithm 3: jobs are sorted by non-increasing finish times, and we assign them in this order to processors (cloud or edge). The current assignment of jobs to processors and communications is then stored in the arrays *executing*, *upcom* and *downcom*, and *nextFinish* is the next event given this assignment (which communication or computation will complete first).

The complexity of the routine that runs at each event is in $O(n(P^e + P^c))$, and since there are at most $4n$ events, the worst case complexity is $O(n^2(P^e + P^c))$.

We consider the same example as in Section 5.2: $P^c = 1$, $P^e = 2$, $n = 4$, and both edge speeds $s = 0.5$. Jobs J_1 and J_2 are such that $r = 0$, $w = 4$, $up = 2$, $dn = 1$, and $o_1 = 1, o_2 = 2$. J_3 and J_4 are such that $r = 2$, $w = 2$, $up = 2$, $dn = 1$, and $o_3 = 1, o_4 = 2$. The schedule returned by *SRPT* is given by Figure 3, with max stretch $\frac{11}{7}$. When the two first jobs are released at time 0, their SRPT is reached by sending the jobs to the cloud. However, once J_1 has been sent to the cloud, the uplink communication channel for the cloud is busy and T_2 is hence executed locally at E_2. The next events occur at time 2, with the release of new jobs. Remaining processing times are updated with the work already done ($up^c_1 = 0$ since T_1 has completed its uplink communication). J_3 and J_4 have then the shortest remaining processing time for an execution on the edge, and J_4 preempts J_2. Since the uplink communication channel is now available, J_2 is sent to the cloud. At time 6, J_1 has a finish time of 7 and its downlink communication is first scheduled, while J_2 would finish the earliest by pursuing its execution on the cloud.

Algorithm 1: *SRPT* heuristic

Data: Job and platform parameters.

$releases \leftarrow PriorityQueue(i, r_i)$ // Jobs ordered by release dates;
$executing \leftarrow$ array$(None)$ // Job currently executed on each processor $1 \leq j \leq P^c + P^e$);
$upcom \leftarrow$ array$(None)$;
$downcom \leftarrow$ array$(None)$;
$Jobs = \{\}$;
$t \leftarrow 0$;
$nextFinish \leftarrow None$;
$i \leftarrow releases.pop()$;
$nextEvent = nextRelease \leftarrow (r_i, \mathscr{E}_r(i))$;
while $nextEvent \neq None$ **do**
    `/* Actualize state of jobs */`
    $(t_{new}, e) \leftarrow nextEvent$;
    if $e = \mathscr{E}_f(i)$ **then**
        $Jobs.remove(i)$;
    if $e = \mathscr{E}_r(i)$ **then**
        $Jobs.insert(i)$;
        $i' \leftarrow releases.pop()$;
        $nextRelease \leftarrow (r_{i'}, \mathscr{E}_r(i'))$;
    `/* Update remaining work */`
    execute$(t,\ t_{new},\ \{w^c_{i,k}, w^e_i, up^c_i, dn^c_i\}_{i \in Jobs, 1 \leq k \leq P^c})$;
    `/* Compute all possible finish times and SRPT schedule */`
    $finishingTimes \leftarrow List()$;
    $t \leftarrow t_{new}$;
    foreach $i \in Jobs$ **do**
        $t^e_i \leftarrow \frac{w^e_i}{s_{o_i}}$;
        $finishingTimes.insert(t + t^e_i, (i, o_i))$;
        foreach $p_k \in P^c$ **do**
            $t^c_{i,k} \leftarrow w^c_{i,k} + up^c_i + dn^c_i$;
            $finishingTimes.insert(t + t^c_i, (i, P^e + k))$;
    $executing, upcom, downcom, nextFinish \leftarrow$ **SRPT-schedule**$(finishingTimes)$;
    if $nextRelease \neq None$ **then**
        $nextEvent \leftarrow \min(nextRelease, nextFinish)$;
    else
        $nextEvent \leftarrow nextFinish$;

Algorithm 2: Update of remaining work

Data: Two timestamps t_1, t_2, remaining work and communication at time t_1
Result: Remaining work and communication at time t_2

execute$(t_1, t_2, \{w^c_{i,k}, w^e_i, up^c_i, dn^c_i\}_{i \in Jobs, 1 \leq k \leq P^c})$
begin
  foreach $j \in P^e$ **do**
    $i \leftarrow executing[j]$;
    $w^e_i \leftarrow w^e_i - s_{o_i}(t_2 - t_1)$;
  foreach $k \in P^c$ **do**
    $i \leftarrow executing[P^e + k]$;
    $w^c_{i,k} \leftarrow w^c_{i,k} - (t_2 - t_1)$;
  foreach $k \in P^c$ **do**
    $i \leftarrow upcom[k]$;
    $up^c_i \leftarrow up^c_i - (t_2 - t_1)$;
    $i \leftarrow downcom[k]$;
    $dn^c_i \leftarrow dn^c_i - (t_2 - t_1)$;

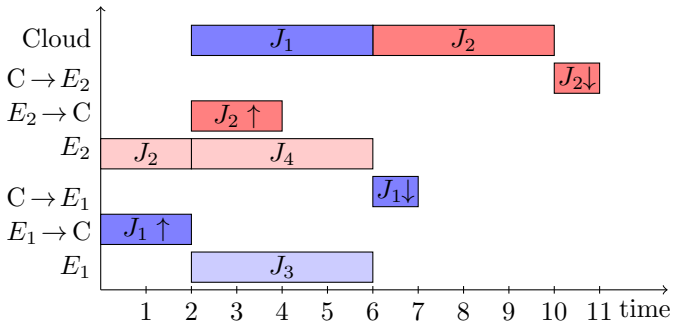


Figure 3: Example of a schedule obtained with *SRPT*.

Algorithm 3: Scheduling between two events using SRPT

Data: List of tuples $(t,(i,j))$: t is the remaining duration needed to finish executing job i on processor j
Result: Job allocations to processors and communication channels, and next completion event

SRPT-schedule($finishingTimes$)
begin
  $executing \leftarrow$ array($None$) // for processors $1 \leq j \leq P^e + P^c$;
  $UpCloud, DownCloud \leftarrow$ array($None$); // for $1 \leq k \leq P^c$;
  $UpEdge, DownEdge \leftarrow$ array($None$); // for $1 \leq j \leq P^e$;
  $nextFinish \leftarrow (\infty, \mathscr{E}_f(-1))$;
  sort($finishingTimes$);
  foreach $(t,(i,j)) \in finishingTimes$ **do**
    $assigned \leftarrow false$;
    if *executing*$[j] = None$ *and* i *is not in executing and* ($j \leq P^e$ *or* $up_i = 0$) **then**
      $executing[j] \leftarrow i$;
      $event = (t, \mathscr{E}_f(i))$;
      $assigned \leftarrow true$;
    else
      if $j > P^e$ *and* $UpCloud[j - P^e] = None$ *and* $UpEdge[o_i] = None$ *and* $up_i \neq 0$ **then**
        $UpCloud[j - P^e] = i$;
        $UpEdge[o_i] = i$;
        $event = (t, \mathscr{E}_{up}(i))$;
        $assigned \leftarrow true$;
      if $j > P^e$ *and* $DownCloud[j - P^e] = None$ *and* $DownEdge[o_i] = None$ *and* $up_i = 0$ *and* $w_i^c = 0$ **then**
        $DownCloud[j - P^e] = i$;
        $DownEdge[o_i] = i$;
        $event = (t, \mathscr{E}_{dn}(i))$;
        $assigned \leftarrow true$;
    if *assigned* **then**
      $nextFinish \leftarrow \min(nextFinish, event)$;
  if $nextFinish = (\infty, \mathscr{E}_f(-1))$ **then**
    $nextFinish \leftarrow None$;
  return $executing, UpCloud, DownCloud, nextFinish$

5.4 Stretch-so-far EDF

For one processor in the offline case, the algorithm in [4] finds the optimal maximum stretch in polynomial time. This algorithm was extended to the online case, and obtained an algorithm that is Δ-competitive, where Δ is the ratio between the longest and the shortest job. The core idea of this algorithm consists in giving deadlines to jobs, based on an estimate of the maximum stretch. This estimate can change over the course of the algorithm and is computed by finding the lowest stretch we can get, through a binary search on the stretch. More precisely, these deadlines are computed as $d_i = r_i + S_e \times t_i$, where S_e is the stretch estimate. This estimate is computed as $S_e = \alpha \times S^*_{\leq k}$, where $S^*_{\leq k}$ is the optimal stretch of the jobs that have already been released (computed through binary search), and α is a parameter of the algorithm. Then, the algorithm schedules the jobs by taking the earliest of these deadlines first. With a well chosen α, this algorithm has a competitive ratio that is a function of $\Delta = \frac{w_{max}}{w_{min}}$. More specifically, for $\alpha = 1$, it is Δ-competitive, which is optimal up to a constant factor. However, the result can be better if Δ is known in advance. This algorithm is called Stretch-so-far Earliest-Deadline-First, since it is the algorithm Earliest Deadline First, with, as input, deadlines derived from the optimal stretch so far.

In the original problem, this algorithm relies on the fact that *Earliest Deadline First* (*EDF*) is optimal with a single processor. However, this is not the case anymore when we consider the MinMaxStretch-EdgeCloud problem, since we need to account for uplink and downlink communications for each job. We provide an example of the non-optimality of *EDF* with two jobs and one cloud processor:

- J_1 with $up_1 = 3$, $w_1 = 1$, $dn_1 = 0$, and deadline $d_1 = 5$;
- J_2 with $up_1 = 1$, $w_2 = 3$, $dn_2 = 0$, and deadline $d_2 = 6$.

EDF establishes that the job with the highest priority is J_1, since $d_1 < d_2$. Therefore, it first sends J_1 to the cloud, and hence J_2 starts its uplink communication at time 3 and completes at time $3 + 1 + 3 = 7 > d_2$. However, by executing J_2 first, J_1 is able to also fit its deadline (uplink communication starting at time 1 while J_2 is executed on the cloud, and execution of J_1 at time 4).

Still, we propose an adaptation of this algorithm for MinMaxStretch-EdgeCloud. However, while *EDF* tells us which job to execute, we now also have to decide on which processor we execute the job with the highest priority. We decide to execute the job on the processor that minimizes the stretch of this job (or, equivalently, its finishing time). This algorithm is called *SSF-EDF*.

At each event corresponding to the release of a job (event $\mathscr{E}_r(i)$), we compute deadlines for each of the jobs currently on the platform, accounting for the remaining working time for the job if part of it has already been executed. These deadlines are computed using a target stretch and $\alpha = 1$, and a binary search is done to try different stretches. Indeed, the goal is to find the best

possible achievable stretch at the current time, but as already pointed out, the *EDF* strategy is no longer optimal and we may not get the optimal stretch. In some cases, a stretch might not be achievable, while a smaller stretch would have been possible with another scheduling strategy.

Given a set of deadlines (and hence a target stretch), we assign the job with the smallest deadline on the processor where it completes the earliest (either the origin edge processor of the job, or one of the P^c cloud processors), and then iterate on other jobs sorted by non-decreasing deadlines. If all deadlines can be respected, a smaller stretch might be considered, and when the binary search is done, we update remaining processing and communication times, similarly to *SRPT*, with the assignment returned by *SSF-EDF* until the next event.

The complexity of *SSF-EDF* algorithm is $O(n(1+P^c))$ at each event, as for the greedy (compute for each job its shortest execution time locally or on one of the cloud processors), but we should re-execute the routine as many times as needed by the binary search. So the actual worst case complexity that we get is $O(n^2 P^c \log(\frac{1}{\epsilon}))$, where ϵ is the relative precision we want to keep on the estimate of the stretch (the binary search takes a time $\log(\frac{1}{\epsilon})$).

If we consider the same example as in Section 5.2 and 5.3, then we find out that *SSF-EDF* outputs the same schedule as *SRPT* for this specific example, with the same maximum stretch of $\frac{11}{7}$ (see Figure 3).

6 Simulations

In order to test the efficiency of the proposed algorithms and to compare them with the *Edge-Only* strategy, we implemented and simulated them in different scenarios, using parameters from real edge-cloud platforms. All heuristics are implemented in C++ and the code can be downloaded at `github.com/Redouane-Elghazi/Max-Stretch-Minimization-on-an-Edge-Cloud-Platform.git` for reproducibility purpose.

6.1 Simulation setup

We consider two main problem instances: instances with random scenarios to experiment with the various parameters, and instances with parameters coming from [23], inspired by a real-life situation.

Random instances: We first consider a platform with:

- 20 cloud processors;
- 10 slow edge processors with speed $s = 0.1$;
- 10 fast edge processors with speed $s = 0.5$.

The jobs are generated using a uniform distribution for the execution and communication times, as well as the release date and the origin processor. Both execution and communication times follow the same distribution. The parameters of the distribution for communication are tied to the parameters of the

distribution for execution, through the notion of Communication/Computation-Ratio (CCR). More precisely, both distributions are chosen so that the ratio between their expected values is equal to some value determined in advance. We consider CCRs ranging from 0.1 (compute-intensive scenario) to 10 (communication-intensive scenario).

Kang instances: These instances are meant to be closer to a real life situation, inspired from [23]. They contain different types of edge processors depending on whether:

- their computational unit is a GPU or a CPU;
- their communication channel is 3G, LTE, or Wi-Fi.

Then, the jobs are created with an execution time and a communication time that is directly inferred from the type of computational unit and the type of communication channel. More precisely, the values are chosen as follows (expected values and speeds according to [23]):

- the execution time follows a normal distribution with mean 6 and relative standard deviation $\frac{1}{3}$;
- the uplink communication time follows a normal distribution with mean t and relative standard deviation $\frac{1}{3}$, where:
  - $t = 95$ if the communication technology is Wi-Fi;
  - $t = 180$ if the communication technology is LTE;
  - $t = 870$ if the communication technology is 3G.
- the downlink communication time is 0 for all jobs.

Also, the speed of an edge processor is:

- $\frac{6}{81}$ if the processor computes on a GPU;
- $\frac{6}{382}$ if the processor computes on a CPU.

Release dates and load: With both instance types, the distribution of the release dates is chosen to control the *load* on edge processors, i.e., the average number of jobs originating from the edge processor that are simultaneously in the system. Hence, for a load ℓ, the maximum release date is set to $\frac{\sum_{i=1}^{n} w_i}{\ell \sum_{j=1}^{P^e} s_j}$: the sum of the work over the aggregated speed is the average execution time using all processors; dividing this ratio by, say, $\ell = 0.1$, augments release times by a factor ten, thereby decreasing the load accordingly. By default, we consider a load of 5% ($\ell = 0.05$), hence the processors are able to cope with the jobs before new jobs arrive in the system, but we still have some interesting collisions on both the edge and the cloud.

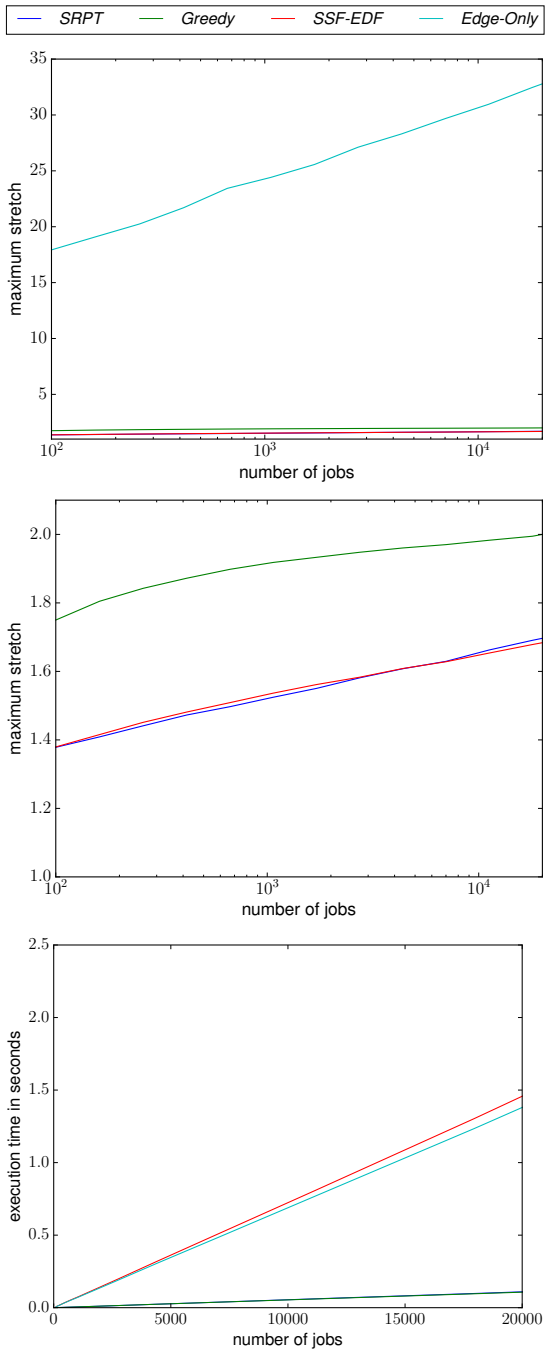


Figure 4: Maximum stretch and execution time with a Communication/Computation Ratio of 0.1 (Random instances).

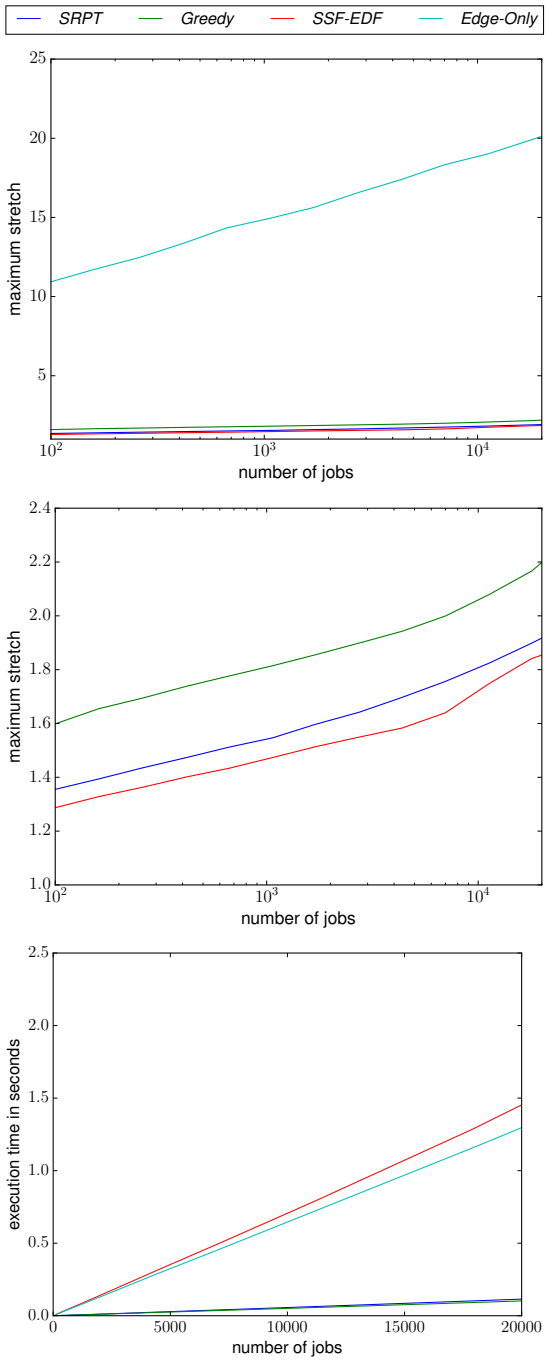


Figure 5: Maximum stretch and execution time with a Communication/Computation Ratio of 1 (Random instances).

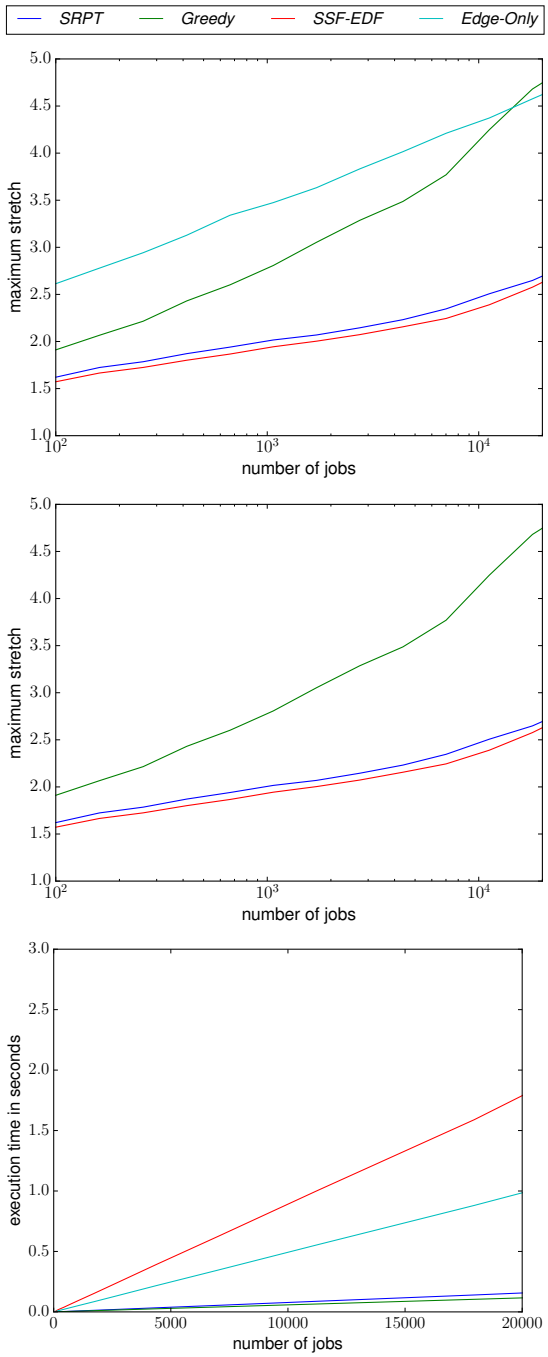


Figure 6: Maximum stretch and execution time with a Communication/Computation Ratio of 10 (Random instances).

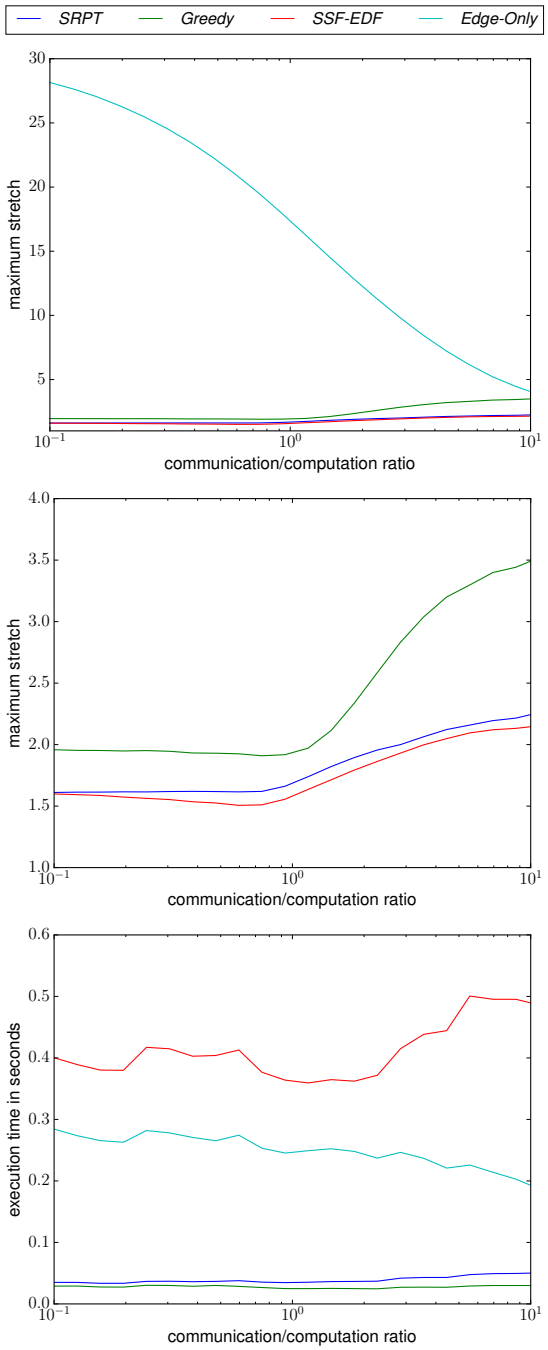


Figure 7: Maximum stretch and execution time as a function of the CCR, with $n = 4000$ jobs (Random instances).

6.2 Simulation results

For each experiment, we report the maximum stretch (with and without the *Edge-Only* heuristic, since it often leads to much higher and out-of-range stretch values), as well as the execution time of each algorithm (time to compute the schedule). Each point corresponds to the average of 1000 instances created with the same parameters.

Random instances: We first run the heuristics on random instances, with three different values of CCR (0.1, 1, and 10), and study the impact of the number of jobs ($100 \leq n \leq 20,000$). As seen on Figures 4, 5, and 6, the proposed heuristics lead to much better stretches than the *Edge-Only* strategy, in particular when communications are not too costly (CCR of 0.1 or 1). Indeed, when the communication cost increases, it is better to handle most jobs on the edge. *SSF-EDF* almost always gives the lowest stretch. *SRPT* is very close to *SSF-EDF*, it can even become better than *SSF-EDF* when there are cheap communications (CCR of 0.1). Even though *Greedy* was designed to minimize the maximum stretch, it always leads to higher stretches compared to the more sophisticated heuristics *SRPT* and *SSF-EDF*.

In terms of execution time, *SRPT* and *Greedy* have similar small execution times, while *SSF-EDF* and *Edge-Only* are significantly slower. Indeed, these latter two heuristics must run a binary search at each new job release to fix deadlines, and hence they take much longer to execute. Overall, all heuristics have an expected execution time that scales linearly and not quadratically as a function of n; only the constants vary.

We also study in more depth the impact of the CCR on Figure 7, where the number of jobs is set to $n = 4000$, and the CCR varies from 0.1 to 10. This confirms that the new heuristics gain more compared to *Edge-Only* for small values of CCR, because it is then very useful to send jobs to the cloud. When looking at the other algorithms, *SSF-EDF* is the best in all scenarios, with *SRPT* being very close. Their stretch exceeds two only for the largest values of CCR. *Greedy* is slightly behind, with stretches going up to 3.5. However, execution times are not impacted by the CCR, they remain roughly constant for each algorithm.

Finally, we investigate scenarios with a higher load, with a load ℓ up to 2, except for *Edge-Only*, which becomes too costly since all jobs compete on the edge. We focus on the instance with a CCR of 1 and $n = 1000$ jobs (Figure 8) or $n = 4000$ (Figure 9). Again, *SSF-EDF* is clearly the best, and this is even more striking when the load increases, where the stretch of *SRPT* and *Greedy* drastically increases while it stays under three for *SSF-EDF*. It is also interesting to note that *Greedy* becomes slightly better than *SRPT* with an increased load, in particular with $n = 1000$. Execution times follow the same trend as before, with an increase in execution time when the load increases, in particular for *SSF-EDF* and *Greedy*; in this case, *Greedy* becomes as costly as *SSF-EDF* (around 10 seconds) with a load $\ell = 2$ and $n = 4000$.

Kang instances: For the Kang instances, the CCR is dictated by the platform parameters, so we only vary the number of jobs, in two different scenarios

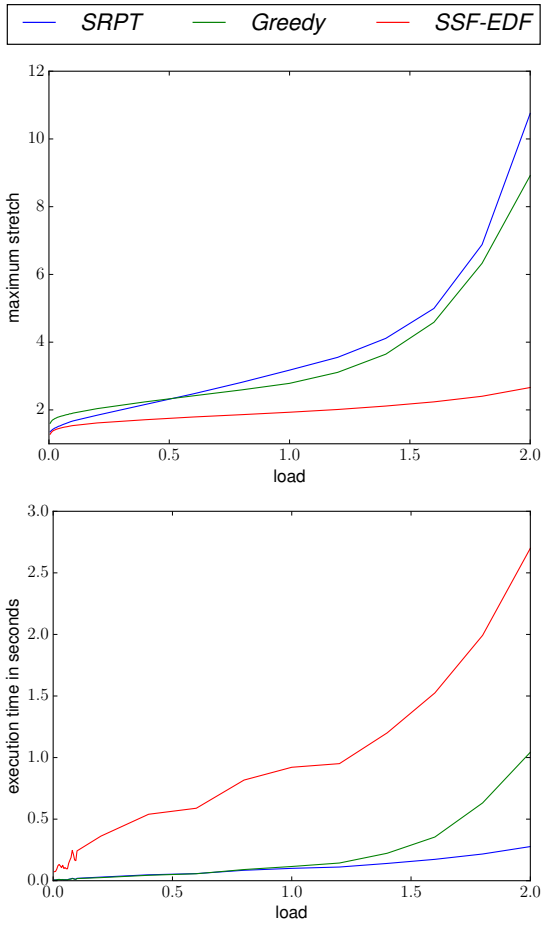


Figure 8: Maximum stretch and execution time with $n = 1000$ jobs and a CCR of 1 (Random instances).

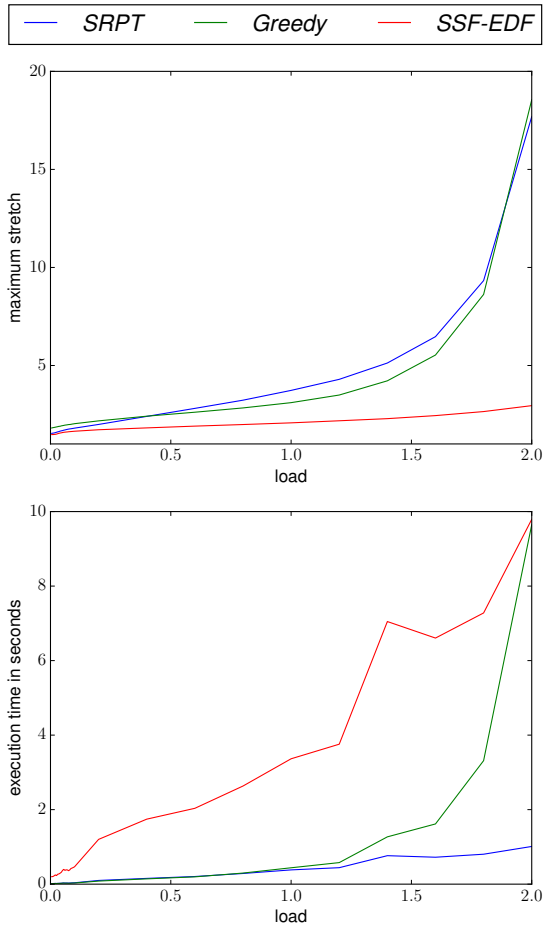


Figure 9: Maximum stretch and execution time with $n = 4000$ jobs and a CCR of 1 (Random instances).

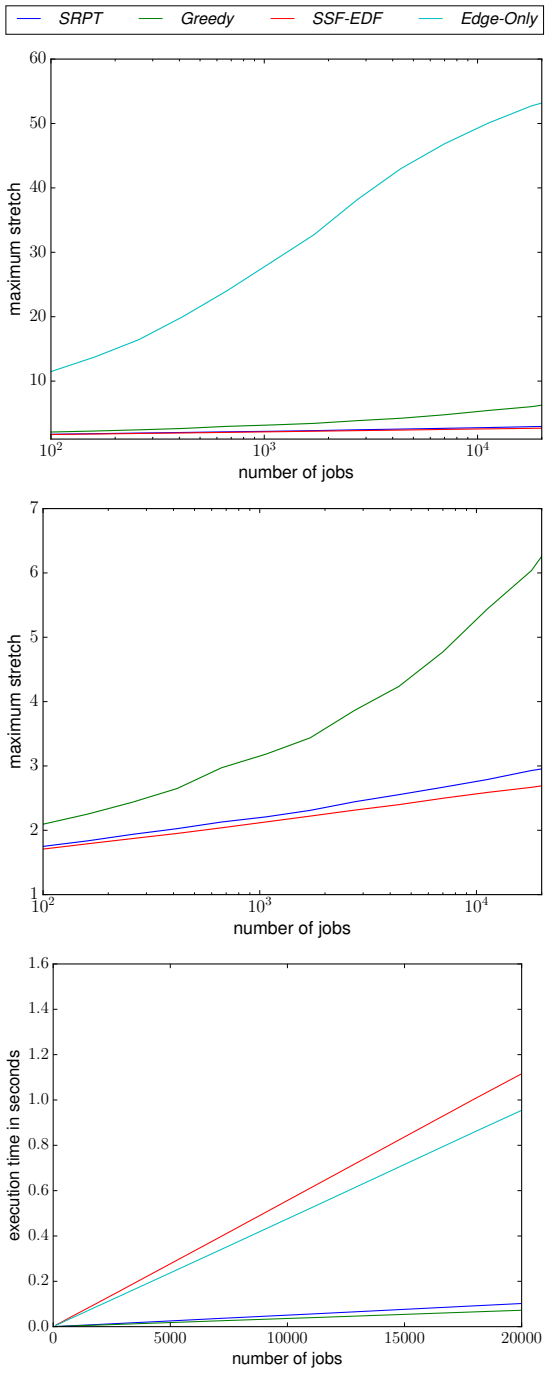


Figure 10: Maximum stretch and execution time: Kang instance with 20 mixed edge processors and 10 cloud processors.

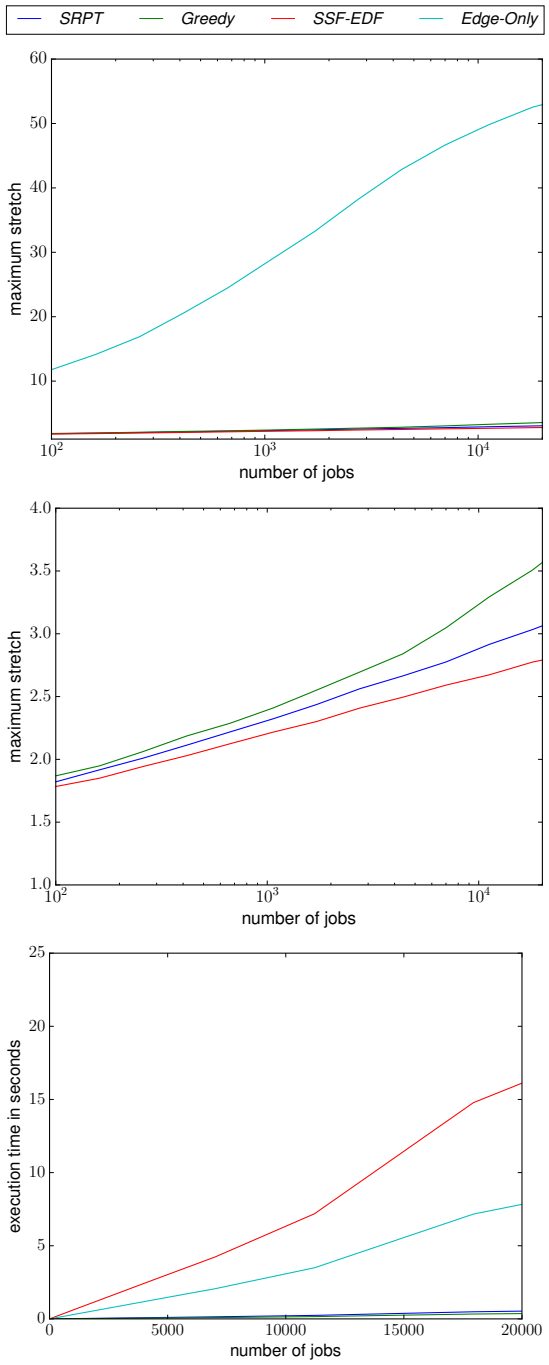


Figure 11: Maximum stretch and execution time: Kang instance with 100 mixed edge processors and 10 cloud processors.

with 20 or 100 edge processors, while keeping ten cloud processors (see Figures 10 and 11). Again, *SRPT* and *SSF-EDF* are clearly the best heuristics, and *Edge-Only* cannot keep up when the number of jobs increases. It is interesting to see that with more edge processors (Figure 11), and hence more competition for cloud resources, *Greedy* gets a stretch that is close to the ones achieved by *SRPT* and *SSF-EDF*.

The execution times are much higher in the scenario with 100 edge processors, and as before, *Greedy* and *SRPT* are much faster than *Edge-Only* and *SSF-EDF*.

Summary: Overall, we conclude that it is always a better option to choose *SRPT* over *Greedy* for lightly loaded systems, since it always leads to smaller stretches, and has approximately the same time complexity. *Greedy* however may outperform *SRPT* when the system is heavily loaded, but then also becomes more costly. Then, choosing between *SRPT* and *SSF-EDF* is a matter of trade-off between efficiency and execution time, since *SSF-EDF* is more costly than *SRPT*, but also more efficient. *Edge-Only* is a costly solution that does not exploit the cloud, and a comparison to this strategy highlights the importance of using cloud resources when available, in particular when communication costs are not too important.

7 Conclusion

We have tackled the problem of scheduling independent jobs on an edge-cloud platform, with the goal of minimizing the maximum stretch of jobs. Jobs are produced by edge computing units, and they can be processed either locally (at a slow speed), or delegated to a cloud (at the price of communications to pay). We have designed a general model, accounting for a realistic communication model, and formalizing the scheduling constraints in this particular setting. While establishing problem complexity, we have encountered the problem of minimizing the max-stretch without release dates on a homogeneous platform, whose complexity was left open in the literature; we have proven that it is NP-hard, filling a gap in fundamental scheduling complexity results. The NP-hardness of the problem in the edge-cloud setting is then a corollary of this result.

We designed heuristic algorithms to address the problem in the general online setting, and we assessed the performance of these algorithms through simulations, using parameters coming from real-world edge-cloud platforms. The algorithms delegating jobs to the cloud achieve much better stretches than a competitor *Edge-Only* approach, in particular when communication costs are not too high. The proposed *SSF-EDF* algorithm always achieves very good stretches. *SRPT* is also is very efficient, and computes its schedules more rapidly, hence it might be an interesting alternative to *SSF-EDF*.

As future work, it would be very interesting to derive theoretical bounds for the proposed online algorithms (competitive results), for instance for some specific job distributions. Furthermore, we would like to investigate a more

complicated framework where cloud processors are not available full-time: in practice, a realistic but intricate framework is to consider that cloud processors may be dynamically requested by other applications at certain time intervals.

Acknowledgments: We thank Denis Trystram for several fruitful discussions on edge computing.

References

[1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. *SIAM J. on Computing*, 31(5):1370–1382, 2002.

[2] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Average stretch without migration. *J. Comput. Syst. Sci.*, 68:80–95, 02 2004.

[3] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proc. of the 9th ACM-SIAM Symp. on Discrete Algorithms*, SODA'98, page 270–279, 1998.

[4] M. A. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, SODA'02, page 762–771, USA, 2002.

[5] M. A. Bender, S. Muthukrishnan, and R. Rajaraman. Approximation algorithms for average stretch scheduling. *Journal of Scheduling*, 7:195–222, 2004.

[6] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[7] A. Brogi, S. Forti, C. Guerrero, and I. Lera. How to place your apps in the fog: State of the art and open challenges. *CoRR*, 1901.05717, 2019.

[8] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 2004.

[9] H. Casanova, F. Desprez, and F. Suter. Minimizing stretch and makespan of multiple parallel task graphs via malleable allocations. In *39th International Conference on Parallel Processing*, pages 71–80, 2010.

[10] J. Celaya and L. Marchal. A Fair Decentralized Scheduler for Bag-of-Tasks Applications on Desktop Grids. In *10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 538–541, 2010.

[11] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 297–305, 2002.

[12] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[13] V. Gama Pinheiro. *The management of multiple submissions in parallel systems : the fair scheduling approach*. Theses, Université de Grenoble & Universidade de São Paulo, 2014. Available at `https://tel.archives-ouvertes.fr/tel-01677743`.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[15] A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *Proc. of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'12, page 1242–1253, USA, 2012.

[16] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen. Scheduling problems in a practical allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[17] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *Int. Parallel and Distributed Processing Symposium IPDPS'2004*, 2004.

[18] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[19] IBM. Cloud at the Edge, 2019. `https://www.ibm.com/cloud/blog/cloud-at-the-edge`.

[20] Infradata. What is Edge-Cloud Computing, 2020. `https://www.infradata.com/resources/what-is-edge-cloud/`.

[21] Intel. Edge-Cloud and Edge Servers, 2020. `https://www.intel.com/content/www/us/en/edge-computing/edge-cloud.html`.

[22] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin, and J. Wan. Energy aware edge computing: A survey. *Computer Communications*, 151, 02 2020.

[23] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *22nd Int. conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS)*, pages 615–629, 2017.

[24] A. Legrand, A. Su, and F. Vivien. Minimizing the stretch when scheduling flows of divisible requests. *Journal of Scheduling*, 11:381–404, 2008.

[25] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. *CoRR*, abs/1604.07525, 2016.

[26] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *40th Annual Symposium on Foundations of Computer Science*, pages 433–443, 1999.

[27] Nokia. Edge-Cloud, 2020. `https://www.nokia.com/networks/solutions/edge-cloud/`.

[28] Y. Robert and F. Vivien, editors. *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2009.

[29] F. Rodrigo De Souza, A. Da Silva Veith, M. Dias de Assuncao, and E. Caron. Scalable Joint Optimization of Placement and Parallelism of Data Stream Processing Applications on Cloud-Edge Infrastructure. In *18th Int. Conf. on Service Oriented Computing (ICSOC)*, Dubai, 2020.

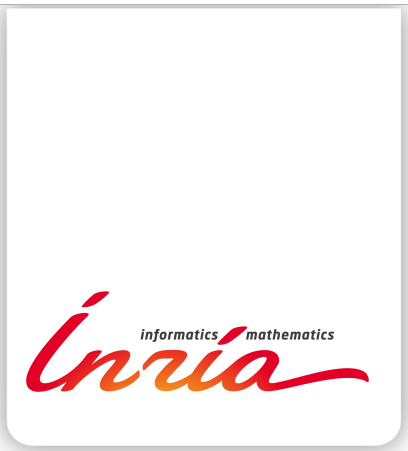
[30] F. Rodrigo De Souza, M. Dias de Assuncao, E. Caron, and A. da Silva Veith. An Optimal Model for Optimizing the Placement and Parallelism of Data Stream Processing Applications on Cloud-Edge Computing. In *IEEE 32nd Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Porto, Portugal, Sept. 2020.

[31] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182, 2004.

[32] O. Sinnen and L. Sousa. Experimental evaluation of task scheduling accuracy: Implications for the scheduling model. *IEICE Trans. on Information and Systems*, E86-D(9):1620–1627, 2003.

[33] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[34] Y. Wu and G. Cao. Stretch-optimal scheduling for on-demand data broadcasts. In *Proc. Tenth Int. Conf. on Computer Communications and Networks*, pages 500–504, 2001.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

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

ISSN 0249-0803