Master Operations Research and Combinatorial Optimization
Master Informatique / Master Mathématiques & Applications

Modeling Scheduling Policies for Serverless Computing

Luc ANGELELLI

August 30^{th} 2021

Research project performed at LIG, Datamove INRIA Team

Under the supervision of:
Prof. Denis TRYSTRAM, Grenoble INP
Assoc. Prof. Grégory MOUNIÉ, Grenoble INP

Defended before a jury composed of:
Prof. Van-Dat CUNG
Prof. Matej STEHLIK
Prof. Nadia BRAUNER
Dr. Claude LE PAPE

[September] 2021

Abstract

With a growing demand for edge computing in applications like Internet-of-Things (IoT), comes a growing need for better and easier to use tools that take advantage of the specificities and avoid the weaknesses of this kind of platforms. One such tool is *Serverless Computing*. Already in use in cloud applications such as Amazon's AWS Lambda, it can be extremely useful in the context of edge computing, as the users do not have to know what the platform is to use it effectively.

Using the edge comes at a cost in terms of computing power, but with a clear benefit in terms of privacy, latency and bandwidth use. Scheduling, in this context, is also not straight-forward due to the highly heterogeneous and changing nature of the edge. So a serverless approach would need to take those parameters into account to be useful.

This work presents such an attempt to model scheduling policies for serverless computing, with an eye toward edge applications. The main result is the design and analysis of an approximation algorithm for scheduling on unrelated machines. We proved that the schedules produced by this algorithm are bounded in makespan with regard to the optimal solution. We also run some experiments to complement the theoretical worst case analysis.

Contents

— 1 —

The Scheduling Problem of Serverless Computing

Serverless Computing - also known as Function as a Service (Faas) - is a model in which users give the application they want to execute to a cloud provider, then the provider allocates as much resources as needed and manages the server for the user. Without Serverless Computing, users would usually reserve a set amount of resources for a set amount of time. This would be difficult and time-consuming to change depending on the actual load, so users would usually reserve more resources than needed to handle high loads easily, and the price would reflect this dynamic. By contrast, Serverless Computing can offer a pricing model based on actual resource usage instead of fixed reservations for a set amount of time. Serverless Computing is also easier to develop for, due to the provider taking responsibility for processes such as scaling, management, maintenance, etc. Serverless Computing is interesting for providers too, as they have control over the resources and it reduces wastes, more users can share the same amount of resources, increasing availability and profit.

However, there are drawbacks to switching to Serverless Computing. For users, it means giving up a lot of control over how, where and when their application is executed. The lack of control of the users makes it difficult for them to optimize their application. It also requires existing applications to be re-designed with Serverless Computing in mind, which might not be trivial. As for providers, they also need to re-design their infrastructure to accommodate Serverless Computing, which is not an easy task. Then they have to take responsibility for the settings and configuration of the platform. Serverless Computing tends to make better use of the resources available, and, even considering all the drawbacks, it can still be worth pursuing for both users and providers.

The goal of this work was to study the scheduling of microservices on serverless platform through edge computing. However since one way to deploy microservices on top of such platforms is using containers, our approach focused on scheduling containers on serverless platforms through edge computing. This problem is NP-Complete [13]. The remaining of this section will briefly present the concepts of microservices, containers and edge computing.

1.1 Microservices and Containers

Microservices are fine-grained and lightweight services. In microservice architecture, instead of having a monolithic application, we have a collection of loosely linked microservices deployed independently. This approach presents many advantages, such as modularity and

scalability. A developer can make changes to a microservice without touching, seeing or even knowing about the other microservices in the system. They can also add or remove clones of the services depending on the load they are experiencing. Another major advantage is the ability to deploy microservices on heterogeneous systems. But this all comes at a cost: the developer has to deal with a software which is more complicated to test and deploy while being slower than a monolith due to communication costs. Microservices are similar to virtual machines (VMs), except in 2 main ways. First, it does not contain an operating system, so it is much smaller and faster to launch than VMs. Then it can run on any machine, given that there is a container engine compatible with the machine.

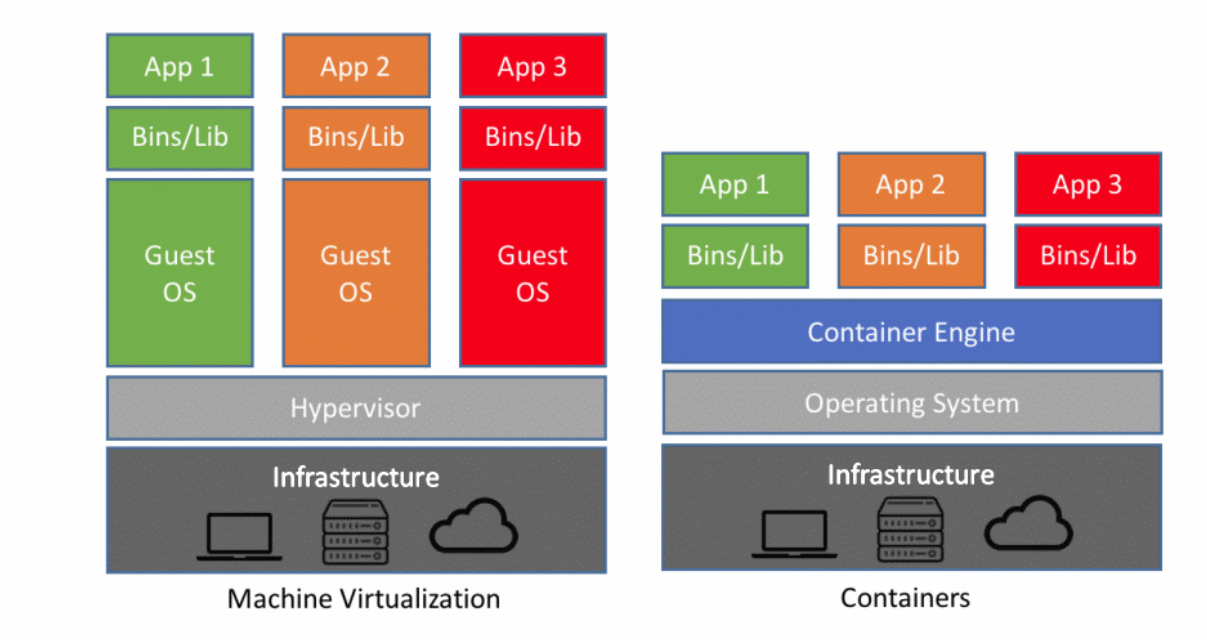


Figure 1.1 – Comparison of Virtual Machines and Containers

Figure 1.1 depicts the differences and similarities between VMs and containers. We see 3 apps and their libraries being deployed in both. In VMs, we also need an OS for each app and an hypervisor to manage communication between the OSs and the infrastructure. For containers, the apps interact with a unique container engine sitting on top of the infrastructure's OS.

The use of containers also comes with a few interesting characteristics such as the design of applications by layers and different start up mechanisms known as Cold/ Warm Start Up. In this work we are interested on the second one. It deals with the presence or the absence of a container in the platform. Indeed, if a container is already there, it is considered as *warm*, and whenever an application which depends on that container is triggered, its start up time is fast. The opposite happens when a needed container is not in the platform, which characterizes a *cold* start up. Figure 1.2 illustrates such behavior. We can see the phases in the squares a) Download the code; b) Container setup; c) Initialization and start of the code; d) Code Execution. The arrows represent the cold start time and the total time. The first one happens when an application needs to execute phases a), b) and c). A warm start time would execute just phase d), being faster than the cold start time. This gain of time is specifically illustrated on Figure 1.3, where we can compare the first call with the others. The first one has cold start

time because it considers the initialization phase, and for the other calls, it is no longer needed - the container is warm - then they are faster.

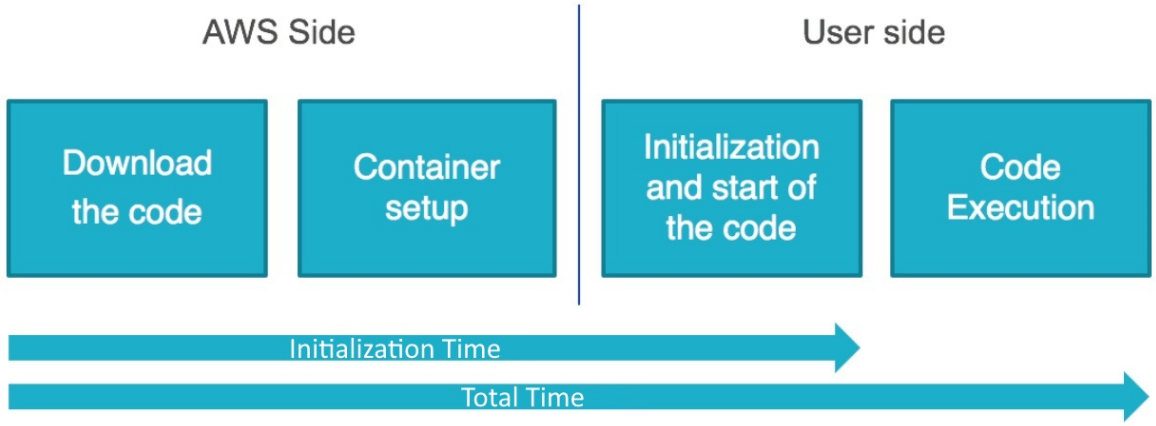


Figure 1.2 – Whole starting sequence of a container in AWS Lambda

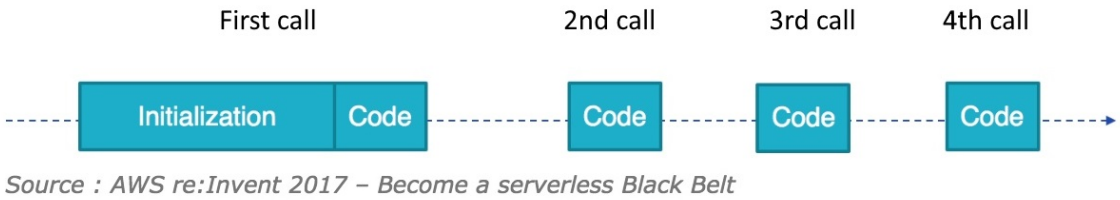


Figure 1.3 – Impact of Warm Start in AWS Lambda

1.2 Edge Computing

The edge is composed of several cloudlets - small cloud-like structures that sit in between the users and the Cloud. Using Edge Computing instead of Cloud Computing can result in lower latency and better bandwidth, as the cloudlets are usually physically much closer to the user than the datacenters hosting the cloud, and there is usually less competition when communicating with a cloudlet, as fewer people are connected to each cloudlet. Cloudlets can be composed of many different devices, for example smart objects, phones or network gateways. So the edge is heterogeneous in a lot of ways, processing power, memory, latency, etc. Incidentally, this heterogeneity makes the edge an interesting platform for microservices. And the complexity it introduces means that we cannot expect developers to know the system their code will run on, so a serverless approach to this problem is necessary.

1.3 Contributions and Remarks

Considering the context presented above, this work contributes with: i) a model for scheduling functions in a serverless architecture with heterogeneous machines, ii) an analysis of input

parameters of the model.

The rest of this report will focus on describing an approximation algorithm for scheduling tasks to heterogeneous machines. At first we formulated the problem like a multidimensional knapsack problem in section 3. Yet this method was too difficult to expand to containers in the way we wanted, so we tried instead to model the containers first, then create schedules with those. We will describe the algorithm that inspired us, and the modifications we made to fit our needs, detailing its proofs and particularities in section 4. Finally, we will end by showing some experimental results we obtained in section 5.

— 2 —
State of the art

As Edge computing and Serverless computing are relatively new, there is not a lot of references in operations research related to those areas. There is however some research related to cloud applications using bin-packing algorithms [14, 5], or knapsacks [3].

This does not mean that our problem has never been explored. A very similar problem to ours was studied by Guisheng Fan et al. [11], though our solutions differ greatly. Their focus is on network related optimization, with a goal to optimize network latency among microservices, reliability of microservice applications and load balancing of the cluster.

N. Dragoni et al. offer a good starting point to studying microservices and microservice architecture [9]. They define quickly the terms of "Monolith", "Microservice" and "Microservice architecture", then describe how it started, what it is now and what we should expect from the future of microservices.

P. Sharma et al. run a comparative study of containers and virtual machines in large data center environments [23]. Their focus is on comparing performance, manageability and software development.

W. Shi et al. define the concept of Edge computing [24]. They describe a few case studies for edge computing, and discuss several challenges and opportunities that awaits future developments for edge computing.

I. Baldini et al. survey existing serverless platforms used in industry, academia and open source projects [7]. This allows them to identify key characteristics and use cases, and describe technical challenges and open problems.

D. Shmoys and E. Tardos developed an approximation algorithm [25] to assign independent tasks to unrelated machines, and are able to bound their approximation to at most the cost and twice the makespan of the optimal solution. It uses a linear program to get an assignment of tasks to machines. But its solutions are not necessarily integer, so it then creates from the solution a bipartite graph in which we can find an integer matching. This matching is a feasible schedule with bounds in cost and makespan.

Approximation algorithms have been presented in great depths by D. Williamson and D. Shmoys [26]. They describe a number of algorithms, use cases, and techniques from greedy algorithms to primal-dual methods.

M. Khatiri's thesis [17], which focuses on work stealing algorithm at first, presents in the end ideas for microservices scheduling. They describe two models partly based on the article by R. L. Graham and M. R. Garey, [12]. The first model, called the static model, is used for an initial placement of containers on machines with resource constraints. In this case, the resources are CPU and Memory needed by the containers, and CPU and Memory present on

each machine. The second model, called dynamic model, is used to update the placement of containers by introducing a cost to add, a cost to delete and a cost to move a container. Its objectives are to reschedule the containers to minimize the number of machines used, while minimizing the cost of container management.

— 3 —
Scheduling with memory constraints

As discussed in section 2, there is not many operations research approaches already existing for edge computing and serverless computing. However, the problem looks like it would lend itself well to a bin packing or a knapsack modelisation. This is exactly what Khatiri [17] started and what we attempted to continue until we decided to focus on the Shmoys/Tardos algorithm [25] in section 4 because it seemed more promising.

This work aims to be a simplified model of a scheduler of microservices for edge platforms. As such, we will only consider edge platforms as "heterogeneous machines", forgoing any communication, topology or network-related problems. Microservices are also simplified as independent tasks for now. An advantage of this method over the ones presented in section 4 is that it allows online scheduling, that is to say we do not need to know the tasks in advance to produce a result.

In this chapter, we will present a model based on a resource constrained scheduling algorithm. We will start with a formulation of the problem, and follow with steps toward a solution. The first one, Initialization, acts on an empty schedule, and the second one, Update, acts on a non-empty schedule.

3.1 Modeling for resource constrained machines

Following the work of Khatiri [17], we have 2 resources we keep track of for each machines, CPU utilization and Memory Utilization. As we work with heterogeneous machines, the amount of each resource available in each machine can be different. So a machine can be represented by a box in 3 dimensions (CPU, MEM, Time).

Similarly, a task can be represented by a 3D box, where the dimensions show how much of each resource the task needs.

We can see a representation of machines and tasks in figure 3.1. For simplicity, we will represent from now on states of the system at a specific time, so that we only need to show 2 dimensions.

We start with a list of tasks we want to execute. From that list, we fill the machines as much as we can, while minimizing the number of machines used. Then, when a task has ended, we find a way to put the next task on a machine while still minimizing the number of machines used and the movement of tasks between machines.

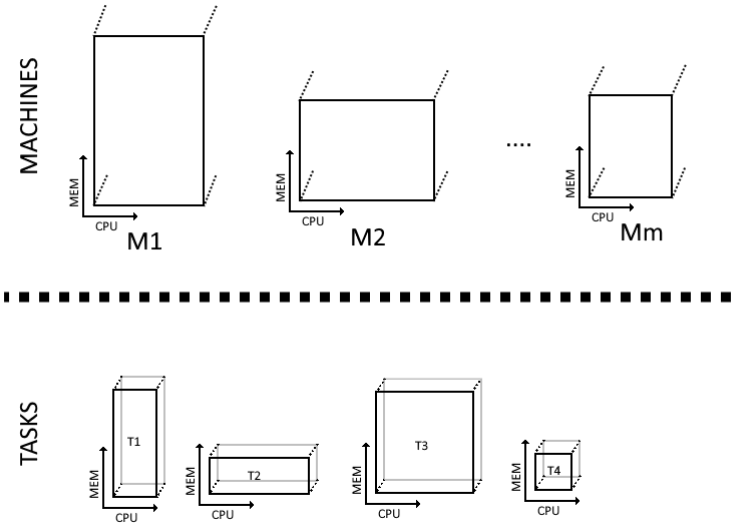


Figure 3.1 – Representation of machines and tasks

3.2 Initialization step: a static model

At first call, all machines are idle and there are tasks in a task_to_do list. The scheduler should try to place as many tasks as possible, while minimizing the number of machines used. Since resources are not shared between tasks, a valid scheduling state at time 0 might look like figure 3.2.

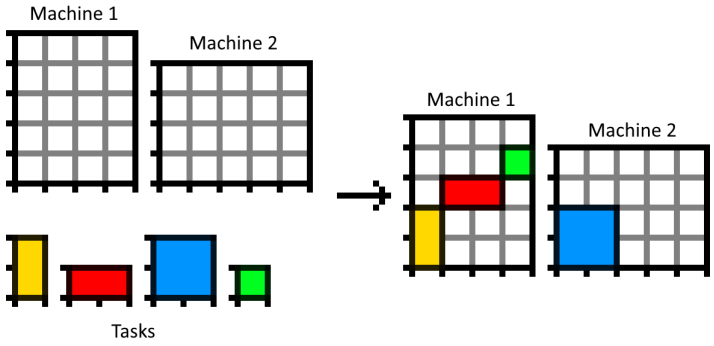


Figure 3.2 – A valid schedule at time 0

This scheduling policy can be done simply by taking the highest priority task, and placing it on the first machine that has enough available resources to accept it.

Availability of a resource is defined as:

$$res(m) - \sum_{t \in tasks(m)} res(t)$$

Moreover, when tasks are assigned to a machine, their order of arrival is not important, as any order should result in the same resource usage. We use in our model a boolean matrix that links tasks to machines. Such model follows as:

Variables:

int N;
int M;
int $QMEM[0..N]$;
int $QCPU[0..N]$;
int $TMEM[0..M]$;
int $TCPU[0..M]$;
dvar boolean $m[0..M]$;
dvar boolean $x[0..N][0..M]$;
dvar int $AMEM[0..M]$;
dvar int $ACPU[0..M]$;

minimize $\sum_j m[j]$

Constraints:

$$AMEM[j] = TMEM[j] - \sum_i x[i,j] * QMEM[i] \quad \forall j \leq M \quad (1)$$

$$ACPU[j] = TCPU[j] - \sum_i x[i,j] * QCPU[i] \quad \forall j \leq M \quad (2)$$

$$\sum_j x[i,j] = 1 \quad \forall i \leq N \quad (3)$$

$$\sum_i x[i,j] \leq m[j] * N \quad \forall j \leq M \quad (4)$$

$$AMEM[j] \geq 0 \quad \forall j \leq M \quad (5)$$

$$ACPU[j] \geq 0 \quad \forall j \leq M \quad (6)$$

Where each constraint means:

(1-2) For each resource, availability depends on the tasks assigned

(3) A task is assigned to exactly one machine

(4) A machine is used if there is at least one task assigned to it

(5-6) Cannot use more resources than available

(7) A task that was running does not change place

3.3 Updating a schedule: a dynamic model

With the initialization step done, the rest of the problem is unlocked. The initialization can only give us an assignment when there are no tasks being processed on any machines, usually at time 0. At any other time, we should update the current assignment. Such an update should be able to assign as many of the remaining tasks as possible, and be able to move them around as it sees fit in order to minimize the number of machines used.

The order of the actions MOVE and ASSIGN does not seem to have an impact on the number of machines used. However, moving a task can have a cost that we would want to minimize. In order to minimize that cost, we will first assign, then move.

In figure 3.3, we can see differences between these two orders. At time t, the scheduler has been called after some tasks ended, and a task is present in the list. If we choose to MOVE first, the running tasks can fit in one machine, at the cost of one move. Then we ASSIGN the remaining task, and we have to use the second machine anyway. If we ASSIGN first, the result is similar in terms of number of machines used, but the operation was cheaper as we did not have to MOVE any task.

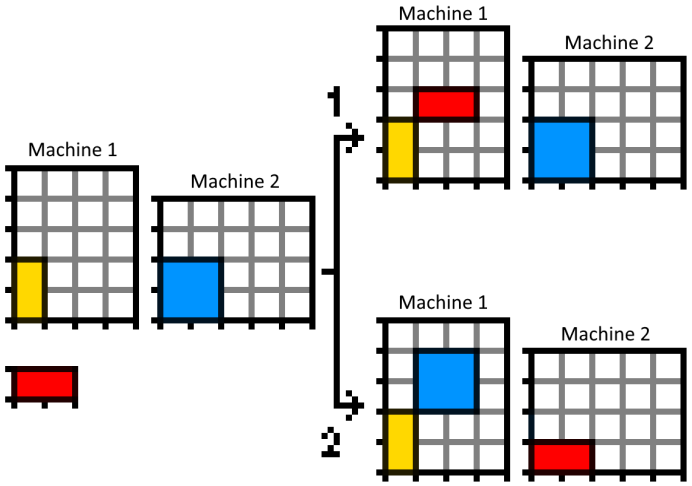


Figure 3.3 – At time t, (1) is ASSIGN then MOVE, (2) is MOVE then ASSIGN

The ASSIGN operation is very similar to the initialization. In fact, the initialization should be a special case of the ASSIGN operation. As such, we can start from the initialization, and add to it a constraint saying that a task that is already running must stay at the same place, as to not move or erase them unnecessarily.

The linear program for that step is then as follows:

Variables:

int N;
int M;
int $QMEM[0..N]$;
int $QCPU[0..N]$;
int $TMEM[0..M]$;
int $TCPU[0..M]$;
dvar boolean $m[0..M]$;
dvar boolean $x[0..N][0..M]$;
dvar int $AMEM[0..M]$;
dvar int $ACPU[0..M]$;

minimize $\sum_j m[j]$

Constraints:

$$AMEM[j] = TMEM[j] - \sum_i x[i,j] * QMEM[i] \qquad \forall j \leq M \quad (1)$$

$$ACPU[j] = TCPU[j] - \sum_i x[i,j] * QCPU[i] \qquad \forall j \leq M \quad (2)$$

$$\sum_j x[i,j] = 1 \qquad \forall i \leq N \quad (3)$$

$$\sum_i x[i,j] \leq m[j] * N \qquad \forall j \leq M \quad (4)$$

$$AMEM[j] \geq 0 \qquad \forall j \leq M \quad (5)$$

$$ACPU[j] \geq 0 \qquad \forall j \leq M \quad (6)$$

$$x[i,j] \geq prev\_x[i,j] \qquad \forall i \leq N, \forall j \leq M \quad (7)$$

Where each constraint means:

(1-2) For each resource, availability depends on the tasks assigned

(3) A task is assigned to exactly one machine

(4) A machine is used if there is at least one task assigned to it

(5-6) Cannot use more resources than available

(7) A task that was running does not change place

The MOVE operation was not completed during the internship, nevertheless we can give a few properties that it would need. The MOVE operation should take a schedule as input and find a minimal set of moves that transforms the input schedule into a schedule that minimizes the number of machines used.

— 4 —

From task scheduling to container scheduling

In this chapter we present how we model our scheduling policy for microservices based on a serverless architecture with heterogeneous machines. Since one way to deploy microservices on top of such platforms is using containers, our approach focused on scheduling of containers on serverless platforms through edge computing.

This chapter go as follows: a presentation of how we modeled containers to fit in our algorithms on Section 4.1; next an introduction of the algorithm basis used for this work on Section 4.2; then a description of our three approaches on top of the basis algorithm: a) a direct application of the algorithm base on Section 4.3, b) a modified version of the algorithm on Section 4.4, and c) another modification considering resources constraints on Section 4.5.

4.1 Modeling Containers

To start modeling the problem of scheduling containers, we first need to define what we want from the model. In order to keep the model light and understandable, we had to make some assumptions and simplifications regarding the containers and the edge platform. Those assumptions are:

- Tasks are independent and known in advance
- The cost and processing time of a task on a machine is known
- The environments needed for the tasks are independent and known in advance
- The cost and processing time of an environment initialization on a machine is known
- The dependency between tasks and environments is known
- Communication time and cost are ignored

For a task to be processed, we have to know both processing time and cost on every machine, and the environment it needs to run. Running a function on a machine means booting the environment on that machine, then running the function. But if the environment is already running, we do not need to reboot it. To mirror the notation used by Shmoys and Tardos [25], we define a function $f_j = \{p_{ij}, c_{ij}, env[j]\}$, where p is a processing time depending on the machine i and the function j where it is executed. c represents a cost that depends on the task and the machine, and e is an environment that the function i needs. To an environment $env[j]$, we can associate a boot time b_i and a cost d_i that depends on the machine i.

4.2 An Approximation Algorithm

Shmoys and Tardos [25] presented an algorithm that allows to get a schedule with bounds on cost and time. However, the algorithm is designed to work with normal tasks, but we want to take advantage of the warm/cold start properties of containers, so we need to adapt Shmoys/Tardos algorithm accordingly. Let us first describe in details how the Shmoys/Tardos algorithm works:

- **Step 1**: Run the linear program presented in Algorithm 4.1. This linear program assigns tasks to machines.
- **Step 2**: Create a bipartite graph that has an integer matching that matches exactly all job nodes.
- **Step 3**: Find a minimum cost integer matching that matches exactly all job nodes.
- **Step 4**: Convert to a schedule.

Variables:

int M;

int N;

int $c[0..M][0..N]$;

int $p[0..M][0..N]$;

dvar float $x[0..M][0..N]$;

Constraints:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij} * x_{ij} \leq C \quad (1)$$

$$\sum_{i=1}^{M} x_{ij} = 1 \qquad \forall j \leq N \quad (2)$$

$$\sum_{j=1}^{N} p_{ij} * x_{ij} \leq T \qquad \forall i \leq M \quad (3)$$

$$x_{ij} \geq 0 \qquad \forall i \leq M, \forall j \leq N \quad (4)$$

$$x_{ij} = 0 \text{ if } p_{ij} > t \qquad \forall i \leq M, \forall j \leq N \quad (5)$$

Figure 4.1 – Linear Program as described by Shmoys/Tardos [25]

Where each constraint means:

(1) The total cost of the assignment is less than C;

(2) For each task, the task is done completely;

(3) Machines are not filled more than T;

(4) All decision variables are non-negative;

(5) A task can only be assigned to a machine if it fits in the time limit;

We will describe Step 2 in more details now. An intuition can be drawn from the property that for a bipartite graph $G = (X + Y, E)$, the following are equivalent (see [22]):

- G admits an X-perfect fractional matching
- G admits an X-perfect integral matching

So creating a bipartite graph that admits an X-perfect fractional matching ensures the existence of an X-perfect integral matching. If we define X as corresponding to the jobs and Y as corresponding to the machines, an X-perfect integral matching would be assigning every job fully to a machine.

Using the notation of the original paper, one side of the bipartite graph represents job nodes $W = \{w_j,\ j \text{ in } 1..n\}$, and the other side contains machine nodes $V = \{v_{is},\ i \text{ in } 1..m \text{ and } s \text{ in } 1..k_i\}$, where $k_i = \lceil \sum_{j=1}^{n} x_{ij} \rceil$, with x a solution to the linear program described in figure 4.1.

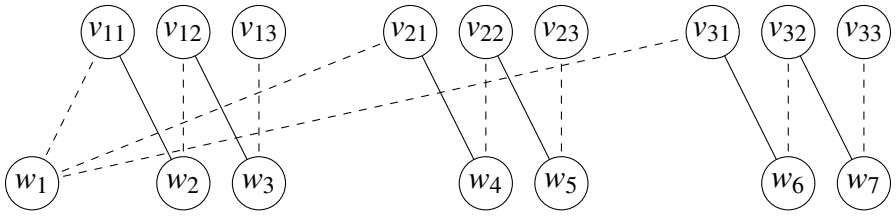
To generate the edges of the graph, we define a vector x' on the edges of the graph such that for each $i = 1..m, j = 1..n$, $x_{ij} = \sum_{x(v_{is},w_j)\in E} x'(v_{is}, w_j)$ For each machine i, we first sort the jobs by non-increasing processing time p_{ij}. Then we have 2 cases. If for a machine i, $\sum_{j=1}^{n} x_{ij} \leq 1$, then that machine has only one node in V, v_{i1}, and every $x_{ij} \geq 0$ corresponds to an edge (v_{i1}, w_j). We then set $x'(v_{i1}, w_j) := x_{ij}$. In the second case, $\sum_{j=1}^{n} x_{ij} > 1$, we can find a minimum index j_1 such that $\sum_{j=1}^{j_1} x_{ij} \leq 1$. We can add all the edges (v_{i1}, w_j) for $j = 1,..,j_1 - 1$ with $x_{ij} \geq 0$ to the graph. For all these, we set $x'(v_{i1}, w_j) := x_{ij}$. Then we add the edge (v_{i1}, w_{j_1}) with $x'(v_{i1}, w_{j_1}) := 1 - \sum_{j=1}^{j_1-1} x'(v_{i1}, w_j)$. This ensures that the sum of the components of x' for edges incident to v_{i1} is exactly 1. Then, if a fraction of x_{ij_1} is still unassigned, we create an edge (v_{i2}, w_j) with $x'(v_{i2}, w_j) := x_{ij_1} - x'(v_{i1}, w_j) = (\sum_{j=1}^{j_1} x_{ij}) - 1$. This operation is repeated for all remaining jobs. We find a minimum index j_s such that $\sum_{j=j_{(s-1)}+1}^{j_s} x_{ij} \leq 1$. We assign all the jobs from $j_{s-1} + 1$ to $j_s - 1$ to v_{is}. For the last j_s, we create an edge (v_{is}, w_{j_s}) with $x'(v_{is}, w_{j_s}) := 1 - \sum_{j=j_{(s-1)}+1}^{j_s-1} x_{ij}$, and another edge (v_{is+1}, w_{j_s}) with $x'(v_{is+1}, w_{j_s}) := x_{ij_s} - x'(v_{is}, w_{j_s}) = (\sum_{j=1}^{j_s} x_{ij}) - s$. Finally, as the last machine node v_{ik_i} might not be filled completely, for each $j > j_{k_i-1}$, we create an edge (v_{ik_i}, w_j) with $x'(v_{ik_i}, w_j) := x_{ij}$.

To better understand, let us consider the following example:

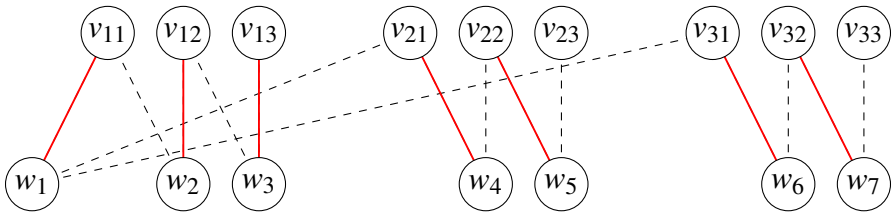
- $m = 3$
- $n = m(m-1) + 1$
- $p_{i1} = m, i = 1, ..., m$
- $p_{ij} = 1, i = 1, ..., m, j = 2, ..., n$
- $c_{ij} = 0, i = 1, ..., m, j = 1, ..., n$
- $C = 0$
- $T = m$

Then, the linear program gives us a solution X = $\begin{bmatrix} 1/3 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

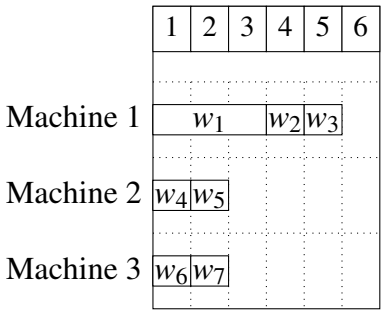
With X, we can construct the following bipartite graph, with doted lines weighing $1/3$ and full lines weighing $2/3$:



And finally, a minimum cost matching gives us:



Which can be translated to a schedule by putting a task j on a machine i if there exists an edge between any sub-machine of i and the task j:



This is obviously not an optimal schedule, or else it would have a makespan of 3 in this example. But as this is an approximation algorithm, we do not expect optimal solutions. And this approximation algorithm ensures bounds of C in cost and $2T$ in makespan for its solutions.

4.3 A First Approach: Direct Application of the Approximation Algorithm

To transition to a more "Serverless" view with environments, a first step can be to consider that every function needs to boot its environment regardless of previous boots. In that case, processing time of a function is $p_{ij} + b_{ij}$, the processing time of the function and the boot time of its environment on a machine. The same idea applies to the cost, with $c_{ij} + d_{ij}$. Formulating the problem this way means that we need no modifications to the approximation algorithm of Shmoys and Tardos, where they found a polynomial-time 2-approximation algorithm to

minimize a weighted sum of the costs and the makespan. However, this formulation is not good when it comes to grouping tasks by environment, or minimizing the number of environment boots, as it was not designed to care about these issues. This might lead to a much longer makespan than necessary, since grouping by environment would save some time with redundant boots. Figure 4.2 and 4.3 show what we could gain by being smarter in how we consider the environments.

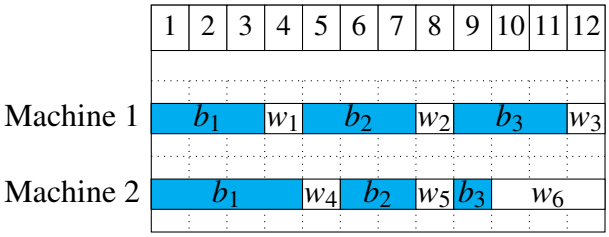


Figure 4.2 – If every function boots its environment

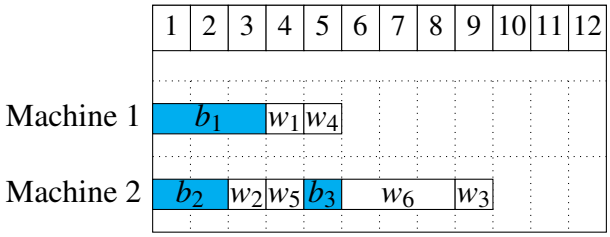


Figure 4.3 – If environments are considered separately

4.4 A Second Approach: Minimizing Startup Costs

A better approximation can be obtained by modifying the linear program given by [25] to include the environments as sorts of tasks. This can be achieved with the modified LP in figure 4.4. In this new LP, the combination of the constraints (6) and (7) ensures that the solution does not boot multiple times an environment on a machine, but does boot when it is needed. The rest of the modifications make sure that the tasks and their environment fit in the machines they are assigned to.

Variables:

int M;

int N;

int K;

int $c[0..M][0..N]$;

int $p[0..M][0..N]$;

int $d[0..M][0..K]$;

int $b[0..M][0..K]$;

int $env[0..N]$;

dvar float $x[0..M][0..N]$;

dvar float $e[0..M][0..K]$;

Constraints:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij} * x_{ij} + \sum_{i=1}^{M}\sum_{k=1}^{K} d_{ik} * e_{ik} \leq C \quad (1)$$

$$\sum_{i=1}^{M} x_{ij} = 1 \qquad \forall j \leq N \quad (2)$$

$$\sum_{j=1}^{N} p_{ij} * x_{ij} + \sum_{k=1}^{K} b_{ik} * e_{ik} \leq T \qquad \forall i \leq M \quad (3)$$

$$x_{ij} \geq 0 \qquad \forall i \leq M, \forall j \leq N \quad (4)$$

$$x_{ij} = 0 \text{ if } p_{ij} + b_{i,env[j]} > t \qquad \forall i \leq M, \forall j \leq N \quad (5)$$

$$x_{ij} \leq e_{i,env[j]} \qquad \forall i \leq M, \forall j \leq N \quad (6)$$

$$e_{ik} \leq 1 \qquad \forall i \leq M, \forall k \leq K \quad (7)$$

$$e_{ik} \geq 0 \qquad \forall i \leq M, \forall k \leq K \quad (8)$$

Figure 4.4 – Linear Program modified to include environment boots

Where each constraint means:

(1) The total cost of the assignment is less than C;

(2) For each task, the task is done completely;

(3) Machines are not filled more than T;

(4) All x decision variables are non-negative;

(5) A task can only be assigned to a machine if it fits in the time limit;

(6) A task needs its environment on the same machine;

(7) An environment is present at most once per machine;

(8) All e decision variables are non-negative;

Shmoys and Tardos managed to prove upper bounds for the cost and the makespan of the schedules. We will show later that we can have a bound on the makespan too, but we could not prove a bound for the cost.

By construction, x' is a fractional matching of cost at most C, which matches exactly all job nodes. For a bipartite graph $G = (X + Y, E)$, the following properties are equivalent (see [22]):

- G admits an X-perfect fractional matching
- G admits an X-perfect integral matching
- G satisfies Hall's marriage theorem's condition

This is the reason for C being the upper bound of the cost in the Shmoys/Tardos algorithm. However, it does not apply to our problem, as the cost is a function of the task and the environment. As this is not taken into account either by the bipartite graph or by the matching, we can say that C is not an upper bound for us. This result was shown experimentally in section 5.

Theorem 1 : If the LP has a feasible solution, then there exists a schedule that has a makespan at most T+2t.

Proof of theorem 1 : Then we will show that the makespan of the schedule is at most T+2t. For a machine i, there are k_i nodes corresponding to it in $B(x)$. For each of these, we will have at most one job scheduled on i corresponding to an edge. As the job nodes where ordered by non-increasing p_{ij} in the construction of $B(x)$, a machine will have at most $\sum_{s=1}^{k_i} p_{is}^{max}$, with p_{is}^{max} the largest p_{ij} assigned to node s. We have p_{is}^{min} defined similarly to p_{is}^{max}, giving us $p_{is}^{min} \geq p_{is+1}^{max}$. In a similar way we obtained $B(x)$, we can get a bipartite graph from e, an environment assignment solution of the LP, with on one side the environments and on the other the machines. This graph would have l_i nodes corresponding to machine i, with $l_i = \lceil \sum_k e_{ik} \rceil \leq k_i$. Then the edges would be placed like for $B(x)$, by first ordering the environments by boot time for each machine, then "filling" each submachine up to 1 with the e_{ik}. From this we define b_{ir}^{max} the largest b_{ik} assigned to node r of machine i and b_{ir}^{min} the smallest, with $b_{ir}^{min} \geq b_{ir+1}^{max}$. The last piece of the puzzle is to observe that $p_{i1}^{max} + b_{i1}^{max} \leq 2t$ because $p_{ij} + b_{i,env[j]} \leq t$.

The remaining terms for p and b give us:

$$
\begin{aligned}
&\sum_{s=2}^{k_i} p_{is}^{max} + \sum_{r=2}^{l_i} b_{ir}^{max} \\
&\leq \sum_{s=1}^{k_i-1} p_{is}^{min} + \sum_{r=1}^{l_i-1} b_{ir}^{min} \\
&\leq \sum_{s=1}^{k_i-1} \sum_{j:(v_{is},w_j)\in E} p_{ij} x'(v_{is},w_j) + \sum_{r=1}^{l_i-1} \sum_{j:(v_{ir},w_j)\in E} b_{ij} e'(v_{ir},w_j) \\
&\leq \sum_{s=1}^{k_i} \sum_{j:(v_{is},w_j)\in E} p_{ij} x'(v_{is},w_j) + \sum_{r=1}^{l_i} \sum_{j:(v_{ir},w_j)\in E} b_{ij} e'(v_{ir},w_j) \\
&= \sum_{j=1}^{n} p_{ij} x_{ij} + \sum_{k=1}^{K} b_{ik} e_{ik} \\
&\leq T
\end{aligned}
$$

Thus for a machine i, the makespan $T_i \leq \sum_{s=1}^{k_i} p_{is}^{max} + \sum_{r=1}^{l_i} b_{ir}^{max} \leq T + 2t$

4.5 A Third Approach: Adding Resources Constraints

The next step is to add an idea of resource constraint to the algorithm. Using what was done by Katiri [17] and Graham [12], we are able to modify further the LP to take a resource constraint. We decided to only use memory as a resource as it seems to be the most limiting factor in placing a task on a machine. Reusing the notation from [17], we add a memory amount $TMEM[i]$ on each machine i, an amount of memory $QMEM[j]$ required for each task j. Those give rise to a decision variable $AMEM[i]$ for each machine i that represents the amount of memory left on that machine, such that : $AMEM[i] = TMEM[i] - \sum_{j=1}^{N} x_{ij} * QMEM[j], \forall i$.

We can see a full implementation in figure 4.5. Note that adding those constraints does not affect the bounds that we proved previously, nor does it changes the behaviour of the algorithm to get a schedule from an LP solution.

Variables:

int M, N, K;
int $c[0..M][0..N]$;
int $p[0..M][0..N]$;
int $d[0..M][0..K]$;
int $b[0..M][0..K]$;
int $env[0..N]$;
int $TMEM[0..M]$;
int $QMEM[0..N]$;
dvar float $x[0..M][0..N]$;
dvar float $e[0..M][0..K]$;
dvar float $AMEM[0..M]$;

Constraints:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij} * x_{ij} + \sum_{i=1}^{M}\sum_{k=1}^{K} d_{ik} * e_{ik} \leq C \quad (1)$$

$$\sum_{i=1}^{M} x_{ij} = 1 \qquad \forall j \leq N \quad (2)$$

$$\sum_{j=1}^{N} p_{ij} * x_{ij} + \sum_{k=1}^{K} b_{ik} * e_{ik} \leq T \qquad \forall i \leq M \quad (3)$$

$$x_{ij} \geq 0 \qquad \forall i \leq M, \forall j \leq N \quad (4)$$

$$x_{ij} = 0 \text{ if } p_{ij} + b_{i,env[j]} > t \qquad \forall i \leq M, \forall j \leq N \quad (5)$$

$$x_{ij} \leq e_{i,env[j]} \qquad \forall i \leq M, \forall j \leq N \quad (6)$$

$$e_{ik} \leq 1 \qquad \forall i \leq M, \forall k \leq K \quad (7)$$

$$e_{ik} \geq 0 \qquad \forall i \leq M, \forall k \leq K \quad (8)$$

$$AMEM[j] = TMEM[j] - \sum_{i} x[i,j] * QMEM[i] \qquad \forall j \leq M \quad (9)$$

$$AMEM[j] \geq 0 \qquad \forall j \leq M \quad (10)$$

Figure 4.5 – Linear Program with environment boots and resource constraints

Where each constraint means:

- (1) The total cost of the assignment is less than C;
- (2) For each task, the task is done completely;
- (3) Machines are not filled more than T;
- (4) All x decision variables are non-negative;
- (5) A task can only be assigned to a machine if it fits in the time limit;
- (6) A task needs its environment on the same machine;
- (7) An environment is present at most once per machine;
- (8) All e decision variables are non-negative;
- (9) The amount of memory required on a machine is less than the amount available;
- (10) All $AMEM$ decision variables are non-negative.

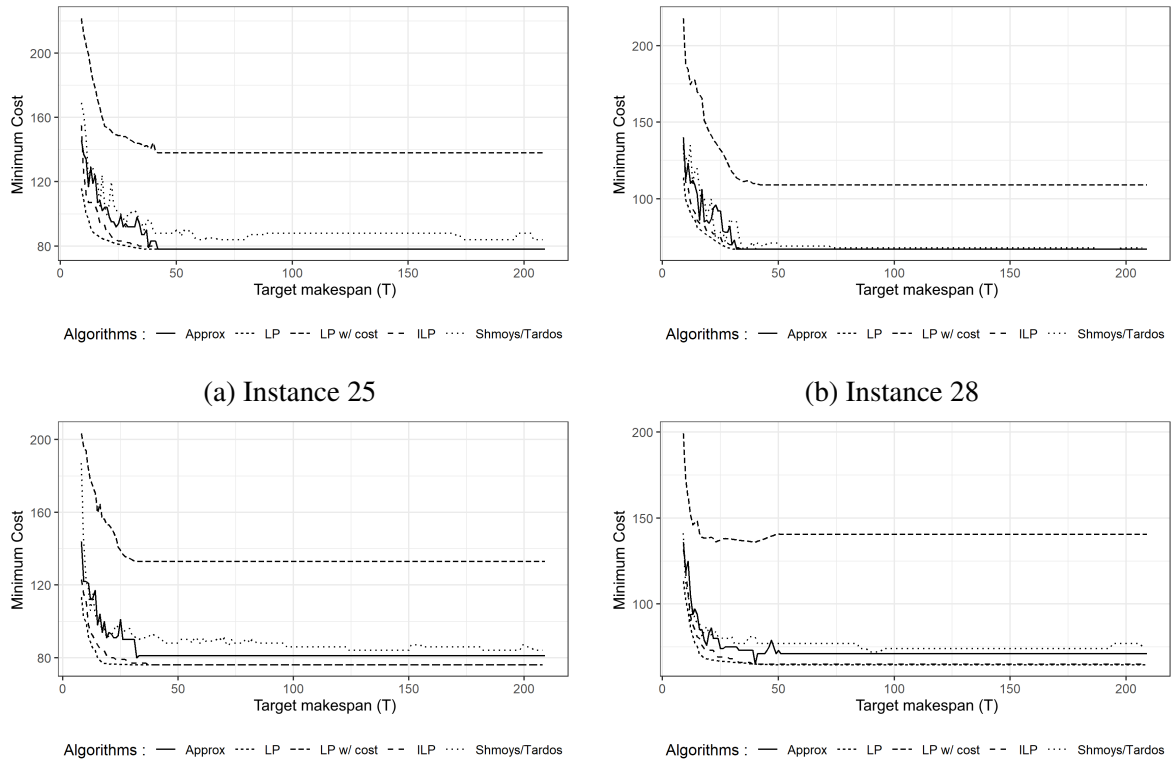
— 5 —
Choosing parameters: Hyper-parameters Tuning

In the end, an important question remains : How to choose the parameters C and T? As C represents a budget, in our case we were thinking of an energy budget, we would want to set C as small as possible. The same idea applies to T, which represents the maximum amount of time we allow a machine to run. However, those variables are not independent, so choosing too small values for both C and T might not give an answer. We can show this trade off easily with a pareto frontier.

The experiments presented in this chapter were produced in python with the library python-MIP, which uses by default the CBC solver [1, 20].

The results we will show in this part were generated by drawing uniformly in the interval [1, 10] the costs and times for each task/machine couple and each environment/machine couple. The task/environment couples are also drawn uniformly.

In figure 5.1, we see comparisons of the costs obtained by different algorithms for a given time T. The *Approx* curve represents the result for the approximation algorithm presented in section 4.2. We can observe that it tends to stay close to the results of the *LP* and the *ILP* alone. We also see that it is usually better than using *Shmoys/Tardos* with startup cost for every function like described in section 4.3. The *LP w/ cost* curve represents an attempt to find an upper bound for the cost. It shows the cost of the LP solution when every task starts its environment.



(a) Instance 25

(b) Instance 28

(c) Instance 34

(d) Instance 39

Figure 5.1 – Minimum Cost depending on Target Makespan (T) for each algorithm.
4 instances generated with 10 machines, 21 tasks and 4 environments.

Now, let us work with smaller examples now to show some properties. These small examples were generated with 2 machines, 4 tasks and 2 environments. The first observation is that the upper bound on cost that was proved by Shmoys and Tardos [25], an approximated schedule costs at most C, does not apply here. The presence of the environments in this problem, and the fact that we do not actually consider them correctly during the approximation step make it so that the selection made in the approximation can cost more than the LP result. When given the choice, the approximation will take the option that costs less overall when looking at $function\_cost + environment\_cost$. It does this without checking if the environment was already on the machine. Solving this requires to be able to create a minimum cost X-perfect matching using

$$environment\_cost = \begin{cases} 0, & \text{if already on the machine} \\ environment\_cost, & \text{otherwise} \end{cases}$$

when evaluating costs of edges in the bipartite graph. We can visualize this effect taking place in most tests, like in figure 5.2, where for some values of T, the approximation can cost about 20% more than the LP.

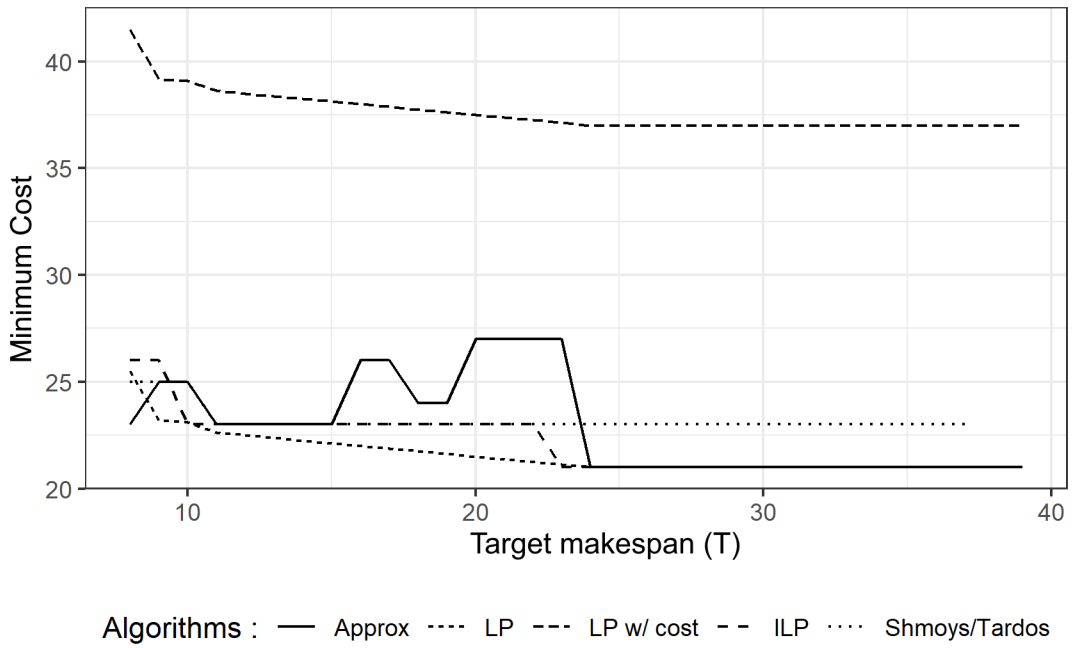


Figure 5.2 – The cost of the approximation is not bound by the cost of the LP

Next property is that the approximation can give a smaller cost than both the LP and the ILP. This might be surprising at first, but it makes sense if we think that it will try to minimize the cost, and is allowed to "trade" time for it. So if the environments are cooperating, we can see costs smaller than both the LP and the ILP. As illustrated in figure 5.3, we see that for $T = 20$, the approximation costs 22, while the LP and the ILP cost respectively 25 and 28.

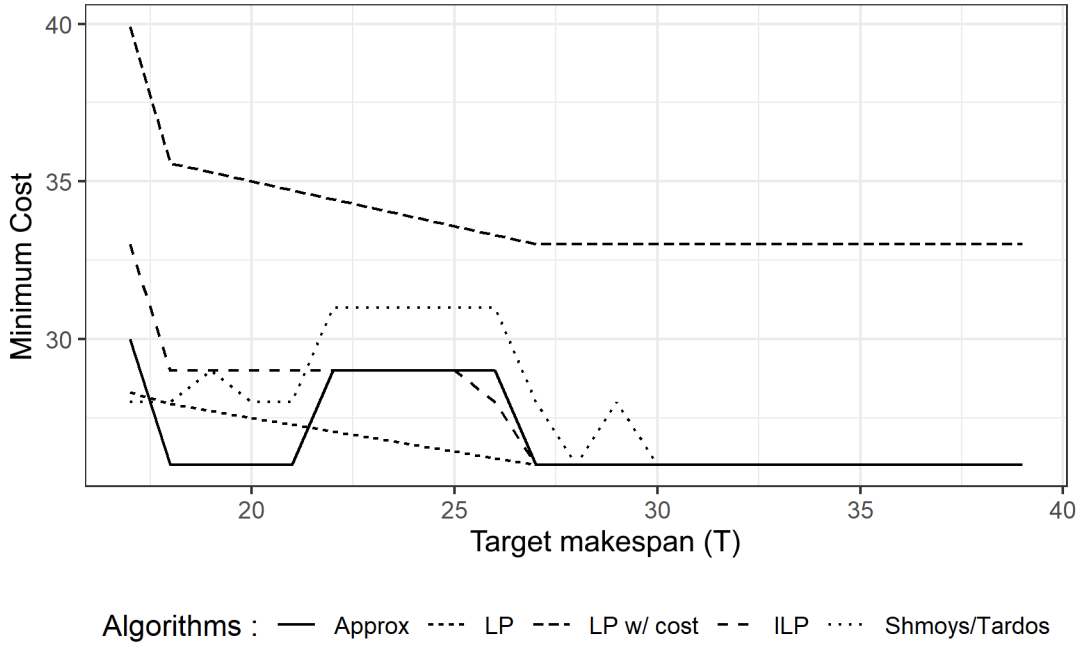


Figure 5.3 – The cost of the approximation can be lower than the LP and the ILP

Then we tried to evaluate what the upper bound would be. Shmoys/Tardos approximation with startup cost for every function seemed good at first, but it can happen that our approximation takes bad decisions where Shmoys/Tardos approximation makes better choices, resulting in better cost than our approximation. Another test was to take the solution of the LP, and calculate the cost as if every task needed to boot its environment. If there is 50% of a task on a machine, we assign 50% of the corresponding environment to the same machine. But, like for Shmoys/Tardos, some decisions can be very bad, and so this is not a bound either. As shown in figure 5.4, at $T = 21$, the approximation costs more than all the other algorithms tested.

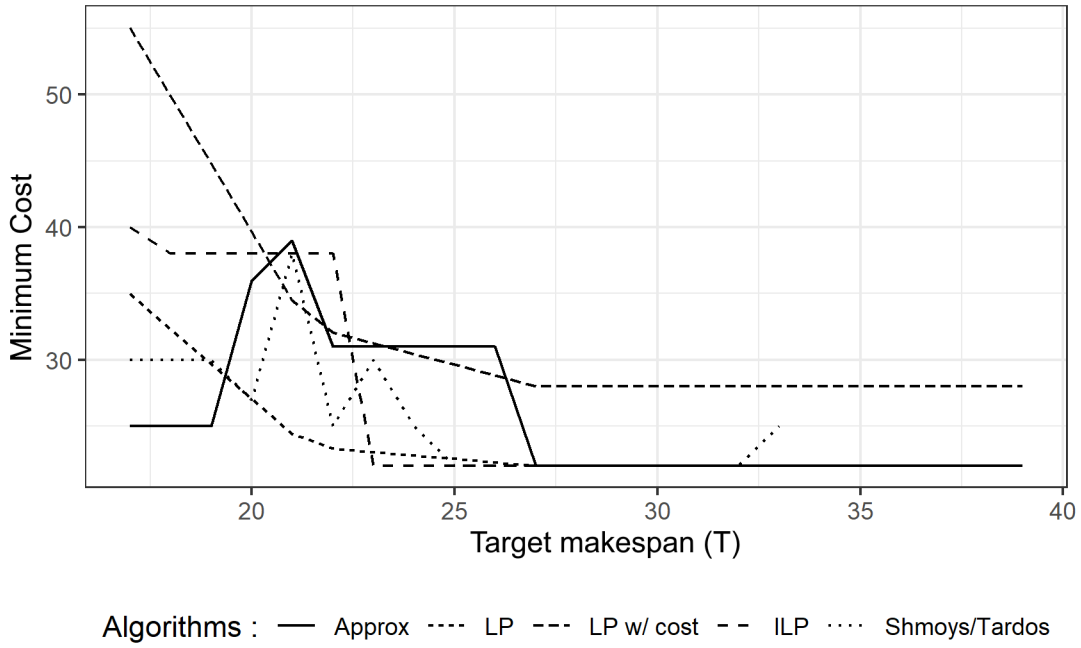


Figure 5.4 – The cost of the approximation can be both lower and greater than the other algorithms

Lastly, we measured the computation times of the different algorithms in figure 5.5. The figure shows the average time to solution for the LP described by Shmoys/Tardos (*LP S/T*), the same LP with the approximation steps added (*LP S/T approx*), our LP that includes environments (*LP*), that LP and the approximation steps (*LP approx*), and the ILP (*ILP*).

We observe that, as expected, the ILP is much slower than the LPs. We also see that the approximations do not cost a lot of time, being barely noticeable when compared to the LPs. It is not clear in this figure, but the approximation step is expected to take more time as the instances grow. Due to the matching algorithm we used in the implementation, it has a complexity $O(NMlog(M))$ (see [15]), with M the number of sub-machines in the bipartite graph and N the number of tasks.

Then, between Shmoys/Tardos algorithm and ours, we see a difference that grows with the size of the instances, which is explained simply by the larger number of variables and constraints in our linear program.

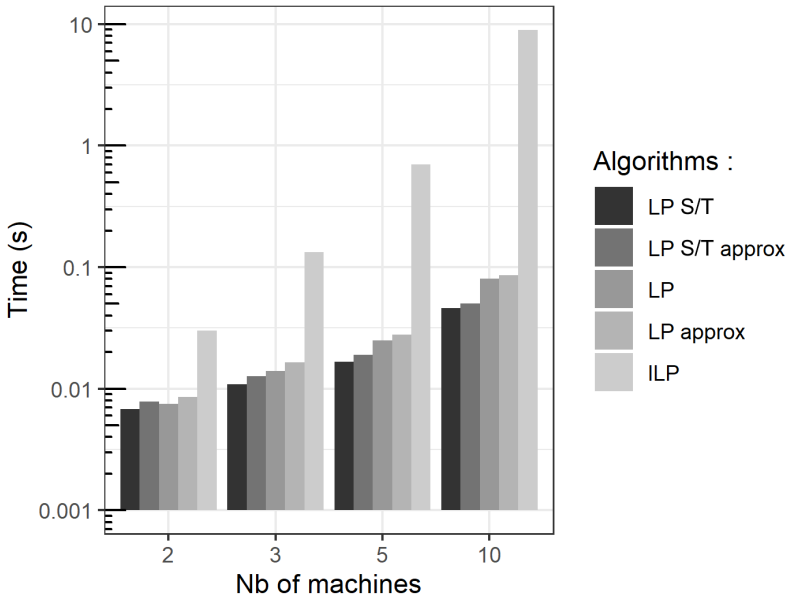


Figure 5.5 – Processing times of the different algorithms

— 6 —
Conclusion

With the increasing presence of IoT devices and the rising need for on-demand computing services, ways to process the huge amount of data created have to be discussed. Using cloud computing to answer this problem has limits and costs that could be negated by other solutions.

One solution, Edge computing, could reduce latency and bottlenecks due to bandwidth limitations by moving the computing power closer to where it is used. Rather than communicating with a cloud that can be thousands of kilometers away, we could use the computing power of local IoT devices, or small local clouds that are seeing much less traffic to provide for our computational needs.

Edge computing is an interesting idea, but is still in its infancy, so research is still needed to make it available and useful to consumers. One such research point asks how to orchestrate tasks between machines that are very different from each other. And another would be the quest to make the solution user-friendly. We started to elaborate on both those points in this internship through the application of serverless principles in the context of scheduling for unrelated machines.

To achieve this, we adapted an algorithm for scheduling tasks to unrelated machines to take into account tasks that need environments to run. This was done in an effort to mimic the use of containers, a popular tool for serverless applications.

This model, although basic, seems to be a good start for further applications. It was not tested in simulation due to time constraints and does not take into account some important parts, such as communication time between machines. This could actually be a good next development for this research. The analysis of the input parameters showed us that even if our algorithm is much faster than an integer linear program, it is still going to be too slow for larger applications. So the jump to simulation is not direct, as we will need to develop a heuristic to replace the linear program we are using now, before we can experiment on larger instances such as the usual instances from the real world. We might lose the theoretical bounds but we would gain a lot in speed, allowing us to actually use it in simulations.

Bibliography

[1] *Documentation of the Python-MIP library*. https://python-mip.readthedocs.io/en/latest/.

[2] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, 2015.

[3] Silvio Roberto Martins Amarante, Filipe Maciel Roberto, André Ribeiro Cardoso, and Joaquim Celestino. Using the multiple knapsack problem to model the problem of virtual machine allocation in cloud computing. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 476–483, 2013.

[4] Mohammad S. Aslanpour, Adel N. Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: Vision and challenges. In *2021 Australasian Computer Science Week Multiconference*, ACSW '21, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Nurşen Aydın, İbrahim Muter, and Ş. İlker Birbil. Multi-objective temporal bin packing problem: An application in cloud computing. *Computers & Operations Research*, 121:104959, 2020.

[6] Kapil Bakshi. Microservices-based software architecture and approaches. In *2017 IEEE Aerospace Conference*, pages 1–8, 2017.

[7] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178, 2017.

[8] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10, 2019.

[9] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[10] P.-F. Dutot, L. Eyraud, G. Mounie, and D. Trystram. Models for scheduling on large scale platforms: which policy for which application? In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 172–, 2004.

[11] Guisheng Fan, Liang Chen, Huiqun Yu, and Wei Qi. Multi-objective optimization of container-based microservice scheduling in edge computing. *Computer Science and Information Systems*, 18:41–41, 01 2020.

[12] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975.

[13] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[14] Shahin Kamali. Efficient bin packing algorithms for resource provisioning in the cloud. In *Revised Selected Papers of the First International Workshop on Algorithmic Aspects of Cloud Computing - Volume 9511*, ALGOCLOUD 2015, page 84–98, Berlin, Heidelberg, 2015. Springer-Verlag.

[15] Richard M. Karp. An algorithm to solve the m × n assignment problem in expected time o(mn log n). *Networks*, 10(2):143–152, 1980.

[16] Gabor Kecskemeti, Attila Kertesz, and Attila Csaba Marosi. Towards a methodology to form microservices from monolithic ones. In Frédéric Desprez, Pierre-François Dutot, Christos Kaklamanis, Loris Marchal, Korbinian Molitorisz, Laura Ricci, Vittorio Scarano, Miguel A. Vega-Rodríguez, Ana Lucia Varbanescu, Sascha Hunold, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer, editors, *Euro-Par 2016: Parallel Processing Workshops*, pages 284–295, Cham, 2017. Springer International Publishing.

[17] Mohammed Khatiri. *Task scheduling on heterogeneous multi-core*. Theses, Université Grenoble Alpes [2020-....] ; Université Mohammed Premier Oujda (Maroc), September 2020.

[18] K. Lange. What's serverless? pros, cons & how serverless computing works, 2021. https://www.bmc.com/blogs/serverless-computing/.

[19] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., USA, 2004.

[20] J. Forrest; R. Lougee-Heimer. *CBC User Guide*, 2014. https://doi.org/10.1287/educ.1053.0020.

[21] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Microservices and containers. In Michael Felderer, Wilhelm Hasselbring, Rick Rabiser, and Reiner Jung, editors, *Software Engineering 2020*, pages 115–116, Bonn, 2020. Gesellschaft für Informatik e.V.

[22] M.D. Plummer and L. Lovász. *Matching Theory*. ISSN. Elsevier Science, 1986.

[23] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[25] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62(1–3):461–474, February 1993.

[26] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, USA, 1st edition, 2011.