Online scheduling of moldable tasks

Thomas Verrecchia

Under the supervision of : Hongyang Sun

from : The department of Electrical Engineering and Computer Science at the University of Kansas

Referent professor : Omar Hammami

1st year of ENSTA Paris

June 2022 - July 2022

This Document is not confidential

Acknowledgements

First of all I would like to express my gratitude to my supervisor professor Hongyang Sun. His guidance, encouragements and suggestions allowed me to focus on the most important points and to see more clearly the best paths to explore.

I would also like to thanks the teachers and administratives of both ENSTA Paris and Kansas University, for giving me the opportunity to do this internship in the best possible conditions.

Abstract

The problem of scheduling moldable tasks on multiprocessor systems has been widely studied, in particular when tasks have dependencies, or when tasks are released on-the-fly. However, few studies have focused on both. This report will have two functions. On one hand, we will assess experimentally some theoretical results proven in previous articles. On the other hand, we will study the impact of different parameters on the performances of allocation algorithms.

Key words : Online scheduling, moldable tasks, multiprocessor system,Python simulation.

Résumé

Le problème de l'attribution de tâche sur un système de processeur a été largement étudié, en particulier quand les tâches sont interdépendantes ou quand sont dévoilées au fur et à mesure. Cependant, peu d'études ce sont concentrés sur les deux à la fois. Ce rapport aura deux objectifs. D'une part reproduire expérimentalement des résultats théoriques prouvés dans de précédents articles. D'autre part étudier l'impact de différents paramètres sur les performances des algorithmes d'allocation.

Mots clés : Plannification en direct, taches, systeme de plusieurs processeurs, simulation Python.

Contents

List of Figures

1 Introduction

All the codes and experiments can be found on the following github :
github.com/thomasverrecchia/Internship_Kansas_University

This paper is to be seen as the report of a two-month internship realized in the Department of Electrical Engineering and Computer Science at the University of Kansas, USA. The subject of this internship is the study of the online scheduling of moldable task.

This is a particular instance of a problem wildly studied in the literature that could be reformulate as: How can we execute certain tasks on a certain number of processors in order to minimize the execution time. The tasks can need a fixed number of processors (allocation) in order to run (*rigid tasks*). The number of processors allocated to a task can be changed, even during the running of a task (*malleable tasks*). In our case we can allocate as many processors as we want to a task, but this number is fixed before we start computing the task and we can't change it afterward (*moldable tasks*). Instead of being independent from one an other (*independent tasks*), our tasks have dependencies in the form of a graph. In addition, we consider the *Online* instance of the problem stipulating that the task graph isn't known in advance by the algorithm. In order to discover a task the algorithm must first complete its predecessors in the graph. However, we made one important concession to avoid being stuck in a problem to complex to solve. We assume that a task can't fail. The objective of the scheduling problem is to find allocation algorithms [1] minimizing the overall completion time of the task graph.

Our first objective in this paper is to build a python simulation allowing us to run experiments on the problem specified earlier. An allocation algorithm that we will call "Paper Algorithm" has been created and studied in the literature [3] [5]. Our second objective is to assess experimentally theoretical results about this algorithm. Our last goal is to further the experiments by testing the effectiveness of different allocation algorithms under various conditions and ultimately conclude on the effectiveness of the Paper algorithm.

The rest of this paper is organized as follow, In section 2 we will define the model and we will give some keys to understand its implementation. In section 3 we will evaluate the impact of the task graph parameters on our allocation algorithms in order to choose the best parameters for the rest of the paper. The section 4 will be dedicated to the reproduction of the theoretical results and the section 5 to the improvement of the Paper algorithm and to further experiments. Finally in section 6 we will conclude and provide hints for future works.

[1] Algorithm that take a task in entry and return the number of processor we should allocate to it

2 Model and simulation

The function of this section will be twofold, first it will set the basic definitions and assumptions of our models. It will then explain its implementation, how we manage to generate task graphs and what allocation algorithms we will be using for our experiments.

2.1 Definitions

2.1.1 Task graphs

In this paper we will consider the online scheduling of a directed acyclic graph (DAG) of moldable tasks on a platform with P identical processors. In order to explain this sentence we must go step-by-step.

Let's start with a directed acyclic graph in which each node represents a task. This task can be computed in a certain amount of time depending on the number of processors we allocate to it and execution time function (or speedup model) we are using. In this graph each edge is directed from a predecessor to a successor. The "online" part of the first sentence means that in order to compute a task T_1 we must had computed every predecessors of T_1 before. An other important assumption of our problem is that the tasks are released on-the-fly, meaning that the algorithm doesn't know a task exists until every predecessor has been computed. Let's take for example the graph represented on figure 1, in order to see and to compute the task T_5 we must had computed the tasks T_3 and T_4 and thus the task T_1 before.

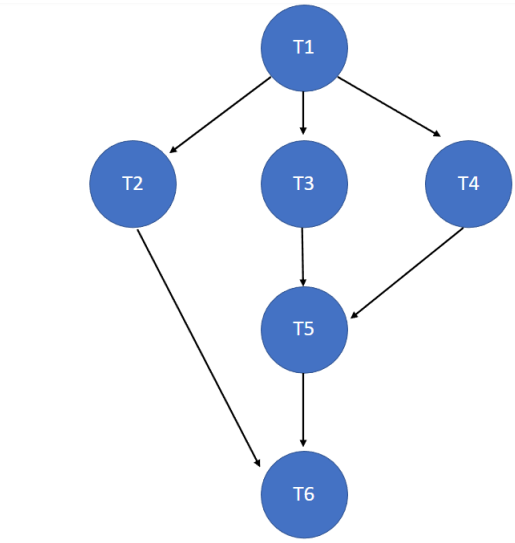


Figure 1: Representation of a directed acyclic task graph

Every task of the graph has a certain execution time depending on the number of processors we are allocating to it and the speedup model we are using. In order to present the different models we need to define the task parameters. Every task is characterized by the four following parameters:

- w_j : The total parallelizable work of the task.
- $\bar{p}_j$: The maximum degree of parallelism of a task.

- γ_j : The sequential fraction of the job.
- c_j : The communication overhead when more than one processor is used.

We will discuss in following sections how we choose these parameters, but for now, let's just assume that they differentiate one task from an other and have an impact on the execution time of a task.

Let j be a task and $t_j(p_j)$ its execution time. Let's also assume that the processor allocation p_j is an integer between 1 and P. In this paper we will study four different execution time functions, each one depend on the number of processors p_j and the task parameters.

- **Roofline Model :** $t_j(p_j) = \frac{w_j}{min(p_j, \bar{p}_j)}$
  When we increase the number of processor the execution time decrease linearly until a maximum degree of parallelism $\bar{p}_j \leq P$.
- **Communication Model :** $t_j(p_j) = \frac{w_j}{p_j} + c_j(p_j - 1)$
  In this model the work of a task can be perfectly parallelized but the communication overhead when more than one processor is allocated increase linearly with the number of processor allocated.
- **Amdahl's model :** $t_j(p_j) = w_j \left(\frac{1-\gamma_j}{p_j} + \gamma_j\right)$
  The decreasing in execution time is limited by the fraction of the task that can't be parallelized.
- **The general model :** $t_j(p_j) = w_j \left(\frac{1-\gamma_j}{min(p_j, \bar{p}_j} + \gamma_j\right) + c_j(p_j - 1)$
  This model is a combination of all the previous models.

In addition the tasks are moldable, meaning that the number of processors allocated to a task is determined by the allocation algorithm only one time at the beginning and cannot be changed during the computation of the task.

2.1.2 Competitive ratio

Our goal is to minimize the execution time of a task graph using different allocation algorithms. The only action of the algorithm is to decide, whenever a task is released, how many processor it will allocate to it. The performance of such an algorithm is measured by its competitive ratio which is given by:

$$\text{Competitive ratio } = \frac{T_{algo}}{T_{opt}}$$

With T_{algo} and T_{opt} respectively the execution time of our algorithm and the optimal execution time for a given task graph. The optimal offline scheduler knows all the tasks and dependencies in advance, however, a careful reader may have noticed that the problem is $NP - complete$ and thus we can't find the optimal time. Nevertheless we can determine a lower bound. In order to do so let's place ourselves in the general speedup model [2].

Let's define the area of a task as $a_j(p_j) = p_j \times t_j(p_j)$. The area can be pictured as the "space" a task is taking on the processor platform in term of number of processors used and execution

[2]The same reasoning is valid for every other speedup model.

time. Let $s_j = \sqrt{\frac{w_j}{c_j}}$. The maximum number of processors that should be allocated to a task is given by :

$$p_j^{max} = min(P, \bar{p}_j, \tilde{p}_j)$$

Where $\tilde{p}_j = \begin{cases} \lfloor s_j \rfloor \text{ if } t_j(\lfloor s_j \rfloor) \leq t_j(\lceil s_j \rceil) \\ \lceil s_j \rceil \text{ otherwise} \end{cases}$

Allocating more than p_j^{max} processors to a task is not decreasing the execution time anymore while increasing the number of processors being used and thus the area of the task, which is rather a bad strategy.

In the range $p_j \in [1, p_j^{max}]$ we can observe two behaviors,

- First of all, for $p, q \in [1, p_j^{max}]$, if $p \leq q$ then $t_j(p) \geq t_j(q)$. It implies that the minimum execution time of a task is given by $t_j^{min} = t_j(p_j^{max})$.
- Additionaly, for $p, q \in [1, p_j^{max}]$, if $p \leq q$ then $a_j(p) \leq a_j(q)$. It implies that the minimum area of a task is given by $a_j^{min} = a_j(1)$.

Let's call $A_{min} = \Sigma_{j=1}^{n} a_j^{min}$ the minimum total area of a graph. The optimal execution time of the graph can't be lower than $\frac{A_{min}}{P}$.

We now define $L_{min}(f)$, the minimum length of a path f in the graph. A path is defined as a sequence of tasks with dependencies where the first task has no predecessor and the last task has no successors. In this context the minimum length of a path is the sum of the minimum execution time of all the tasks along the path: $L_{min} = \Sigma_{j \in f} t_j^{min}$.

The last definition we need is the minimum critical path length $C_{min} = max_f L_{min}(f)$ which is the longest minimum length considering every path in the graph. To give you an example, if all the tasks of the figure 1 had the same minimum execution time, the minimum critical path would be T_1->T_3->T_5->T_6 or equivalently T_1->T_4->T_5->T_6. The optimal execution time can't be lower than C_{min} because in order to compute all the tasks in the graph we need to, at least, compute the minimum critical path.

We end up with the following result : $T_{opt} \geq max\left(\frac{A_{min}}{P}, C_{min}\right)$.
We can't compute the exact optimal time, so for the rest of this paper we will consider that :

$$T_{opt} \approx max\left(\frac{A_{min}}{P}, C_{min}\right)$$

If T_{opt} is in fact greater than this value the consequence will be us overestimating the competitive ratio. Thereby the competitive ratios we will be providing in this paper can be seen as the worst-case scenario [3]. The computing of the competitive ratio will be necessary to assess the performance of our algorithms and it will be detailed in the next section.

2.2 Building the model

In order to run experiments with different task graphs and speedup models we need to implement the tasks, graphs and processor platform. We choose to use python for its simplicity and

[3] The case where the optimal execution time is the lowest it can possibly be.

because of our familiarity with this language. We will not detailed every step of the implementation because there is little benefit to that and it would take too long. However, we will focus on some points of importance. We used object-oriented programming and designed specific class and interactions for Tasks, Graphs and Processors. The codes and experiments can be found on the following github project :
github.com/thomasverrecchia/Internship_Kansas_University

2.2.1 Processor platform

The processor platform can be represented as a table with the processors in ordinate and the time in abscissa. The figure 2 show an hypothetical representation of the processor table for the graph from on figure 1.

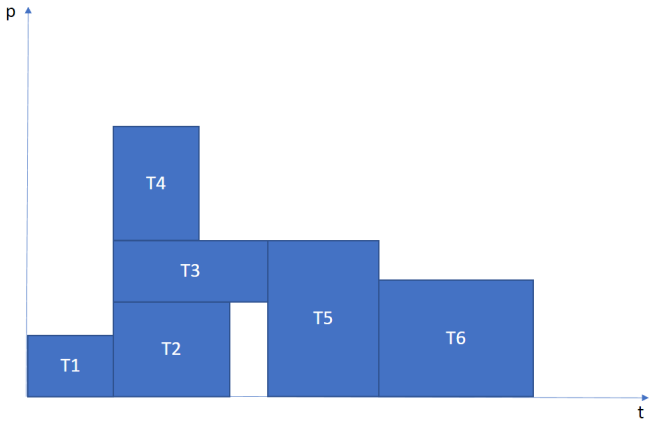


Figure 2: A representation of the processor table for the graph shown on figure 1

We can see that we first need to compute the task T_1 that has no predecessors, then, once it's completed we can see and compute the tasks T_2, T_3 and T_4. We need to wait until the task T_3 is completed before computing T_5 because T_3 is a predecessor to T_5. It goes the same way with T_6 and T_2. We can already see two different naive strategies to minimize the execution time. The first one would be to allocate t_j^{max} processors to every tasks in order to compute them quickly even if it means that we can't compute much tasks at the same time [4]. The other strategy would be to allocate less processors to every task, they will take longer to run but it will allow us to run more tasks in parallel. Those two strategies will give birth to two algorithms, respectively the Min Time algorithm and the Min Area Algorithm which will both be explained in the section **2.4**.

Let's refocus on the implementation of the processor platform. In order to implement it we are building two lists, a *Waiting Queue* containing the tasks that we can compute and a *Computing Queue* containing the tasks that are being computed. Every time a task T is computed we have to update the waiting queue by adding the successors of T. For every task we are adding to the queue list we have to make sure that every predecessors has been computed. We then need to allocate processors to these new tasks and see if we have enough processors available to fit a task from the waiting queue in the processing queue. We then repeat until every task of the graph is computed. This process is summarized in the following algorithm.
In order to allocate processors we will be using three different algorithms detailed in section **2.4**. We present on figure 3 the evolution of the waiting and processing queues when the

[4]The number of processors being used at the same time must be inferior or equal to P

Algorithm 1 Online Scheduling Algorithm

```
1: initialize a waiting queue Q
2: while Every task isn't computed do
3:     if at time 0 or a running task completes execution then        ▷ Processor Allocation
4:         for each new task j that becomes available do
5:             Allocate processor
6:             Insert task j into the waiting queue Q
7:         end for                                                    ▷ List Scheduling
8:         for each task j in the waiting queue Q  do
9:             if there are enough processors to execute the task then
10:                execute task j now
11:            end if
12:        end for
13:    end if
14: end while
```

online scheduling algorithm is working on the graph from the figure 1 and 2 so you can better understand its operation.

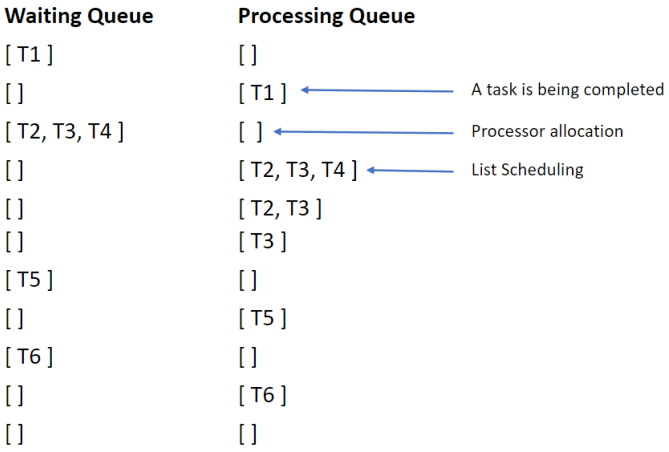


Figure 3: Running display of the algorithm 1 on the graph from the figure 1 with the processor table from the figure 2

2.3 Generating task graphs

In order to generate the task graph we used DAGGEN Algorithm [4] which generate random, synthetic task graphs based on four mains parameters:

- The width of the DAG, fat. This parameter can variate between 0 and 1. A small value will lead to a thin DAG (e.g., chain) with a low task parallelism, while a large value induces a fat DAG (e.g., fork-join) with a high degree of parallelism.
- The *density* that determines the numbers of dependencies between tasks of two consecutive DAG levels.
- The *regular* parameter that determine the regularity of the distribution of tasks between the different levels of the DAG.

- The maximum number of levels spanned by inter-task communications, $jump$.

In the section **3** we will variate these parameters and assess their impacts on the performances of our allocations algorithms.

Concerning the tasks, we will draw random values between the following bounds [5] for each parameters and each task:

- The sequential execution time of a task, $w_j \in [5000, 4000000]$
- the maximum degree of parallelism of a task, $\bar{p}_j \in [100, 800]$
- the communication overhead is set as $c_j = \alpha 2r$, where r is an integer uniformly chosen in $[0, 3]$ and α is drawn uniformly in $[1, 2]$.
- the sequential fraction of the job is set as $\gamma_j = \frac{\alpha}{10^r}$, where r is an integer uniformly chosen in $[2, 7]$ and α is drawn uniformly in $[0, 10]$.

2.4 Allocation algorithms

After postponing multiple times we will now present the allocation algorithms. Each algorithm take a task in input [5] and return the number of processor we should allocate to it (allocation).

2.4.1 Min Time Algorithm

This first algorithm is very simple and similar to a brute force method. For a given task, it compute the number of processor minimizing the execution time p_j^{max} and allocate it to the task.

Algorithm 2 Min Time

1: allocate p_j^{max} processors to the task.

We can easily see the limitations of such an algorithm. However, before experimenting anything we can also make the hypothesis that this algorithm may be efficient with some particular type of graph. For example, if we take a graph with low fat [6] we will only have one task to compute at a time. In such case allocating the number of processor minimizing the execution time may be the best idea. On the contrary if we take a graph with a fat close to 1 we will have a lot of task to compute at the same time due to the high degree of parallelism. By allocating the number of processors minimizing the execution time to each task we may not be able to process all the available tasks at the same time thus slowing down the process. We fall back to one of the brute force method limitations, depending on the problem we may not reach a solution close to the optimal by taking optimal decision at every step.

2.4.2 Min Area Algorithm

The Min Area algorithm follow the same principle as the previous one, but instead of minimizing the execution time we minimize the area of the task. This can be done by allocating one processor to the task.

[5] In the form of its parameters

[6] i.e. a thin graph close to a chain with low to no task parallelism

Algorithm 3 Min Time

1: allocate the number of processors minimizing the area to the task.

The Min Area algorithm has the same limitations as the Min Time algorithm, but the other way around. Indeed, it will perform very badly with a low fat graph because it will allocate one processor to each task and will be able to compute only one task at a time. However, it may be more efficient with a high fat graph because it will be able to process all the available tasks at the same time.

The experiments will show that this algorithm is in fact way worse than the two other in every cases we came upon, so we will not display it for clarity purposes. However, we used a relatively small number of tasks in our experiments so this algorithm may shine with a much higher number of task.

2.4.3 Paper Algorithm

This third algorithm is more complex in its construction and we hope that it will achieve better and more stable results than the last two ones.

The first step is to perform an initial allocation inspired by the Local Processor Allocation (LPA) [5] [6]. Let $\alpha_p = \frac{a_j(p)}{a_j^{min}}$ be the ratio between the area of the task and the minimum area and $\beta_p = \frac{t_j(p)}{t_j^{min}}$ be the ratio between the execution time of the task and the minimum execution time. We then create a constant μ which value will variate depending on the speedup model we are using [7]. This constant is called the processor adjustment. In order to complete the initial allocation we want to minimize α_p subject to the constraint $\beta_p \leq \frac{1-2\mu}{\mu(1-\mu)}$. This optimization problem can be solved in linear time.

The second step reduces the allocation to $\lceil \mu P \rceil$ if the first step had put it over this value, otherwise the allocation remain the same. This approach is used in the literature [7] [8] [9] and its purpose is to force the algorithm to allocate less processors to a task than it would normally do, allowing the processor platform to run more tasks simultaneously.

Algorithm 4 Paper allocation algorithm

1: Compute p_j^{max} ▷ Step 1 : Initial Allocation
2: Compute $t_j^{min} = t_j(p_j^{max})$ and $a_j^{min} = a_j(1)$
3: Find an allocation $p_j \in [1, p_j^{max}]$ by solving the following optimization problem:
4:
$$min_p \alpha_p = \frac{a_j(p)}{a_j^{min}} s.t. \beta_p = \frac{t_j(p)}{t_j^{min}} \leq \frac{1-2\mu}{\mu(1-\mu)}$$
5: **if** $p_j < \lceil \mu P \rceil$ **then** ▷ Step 2 : Allocation adjustment
6: $p'_j \leftarrow \lceil \mu P \rceil$
7: **else**
8: $p'_j \leftarrow p_j$
9: **end if**
10: Allocate p'_j processors to the task

[7] This constant is a central element of the Paper algorithm and we will delve into it in the sections **4.1** and **5**

We can see that this algorithm focus on two points at the same time, by minimizing the area of the task under an execution time condition, then reducing the allocation if its too important . Reducing the execution time and keeping an allocation low enough to allow the computation of multiple tasks in parallel.

We now need to compute the competitive ratio in order to compare our allocations algorithms in various situations.

2.4.4 Computing the competitive ratio

As explained earlier the competitive ratio is a measure of the effectiveness of a given allocation algorithm, in order to calculate it we need to find T_{opt} (i.e. $max\left(\frac{A_{min}}{P}, C_{min}\right)$). Finding A_{min} is rather easy we just have to compute the minimum area for each task than add them together. In order to do it we only need to calculate the execution time of the task with 1 processor. Concerning C_{min}, its calculation is a little less trivial so we went with the algorithm **2**. We started by giving each task an additional parameter called *weight*. The weight of a task represent the minimum critical path length from the top of the graph to this task. We will update the weight of each task while browsing the graph and at the end, the maximum weight will be C_{min}. In order to do it rigorously we start at the top of the graph with every tasks without predecessors and add them to a list called $offspring$. Every time a task is explored we put all his successors in offspring so we can update their weight.

This approach may not be the more efficient one because it may implies revisiting certain tasks multiple times [8]. Regarding the number of task we are considering we found this algorithm effective enough to not slow down the calculations.

Let n be the number of tasks in the graph and t_j^{min} be the minimum execution time for each task.

Algorithm 5 Calculate C_{min}

```
Create an empty list called offspring with every task without predecessors.
Set the weight of every task at 0
while offspring not empty do
    Pick the first task of offspring
    Calculate the maximum weight of its predecessors if possible. Let's call it p_weight
    weight ← t_j^min + p_weight
    Remove the task from offspring
    Add every successors to the offspring list and delete the duplicates
end while
Return the maximum weight of the graph
```

In this section we have set all the definition needed to understand the problem. We have given a quick look at the implementation, presented the allocation algorithms and found a way to evaluate them. Till the end of this paper we will now use those tools to assess theoretical results and to look at the impact of different parameters on the performance of our algorithms.

[8] Indeed, some tasks may have multiple predecessor so they every time a predecessor is checked they will be checked again.

As said previously we will exclusively be comparing the Min Time algorithm and the Paper Algorithm.

3 Impact of graph parameters

The first set of experiment we are going to conduct concern the graph parameters. As we explained earlier two graph with the same number of task n can have widely different structures depending on the density, fatness, jump and regular we chose before generating them. Our goal in this section is, of course, to compare the performance of our algorithms working with different types of graph. We will also be selecting the particular values of these parameters we will be using in the following experiments. We choose to display only the variation of density and fat because we found out that the two other parameters practically doesn't influence the performance of our algorithms.

The modus operandi of our experiments will remain the same along this paper so we will take the time to explain it here. For each experiment we modify only one parameter at a time. We then generate 30 graphs (we will call them instances) for each variation of this parameter and we display the mean value. When we generate graphs there is some randomness involved. It's due mainly to the fact that we choose the task parameters randomly between two bounds. We hope to reduce the randomness by working with 30 instances instead of one. It's important to note that both algorithms are tested on the same 30 instances. The next step is to run our algorithms through these instances for each of the four speedup models introduced earlier. We finally display the competitive ratio (or Normalized makespan) depending on the algorithm, speedup model and parameter variation.

Along with the task parameter bounds introduced earlier we will work with $n = 500$ tasks and $P = 1500$ processors, these numbers are based on previous experiments [5]. The value of μ for the Paper algorithm depend on the speedup model and we will start with the values from the literature [3] :

$$\mu = \begin{cases} 0.271 \text{ For the Amdhal speedup model} \\ 0.324 \text{ For the Communication speedup model} \\ 0.211 \text{ For the General speedup model} \\ 0.382 \text{ For the Roofline speedup model} \end{cases}$$

These values of μ have a theoretical interest that we will discuss in section **4.1** but may not be the best choices to optimize the performances of the Paper algorithm. We will further this study in section **5**.

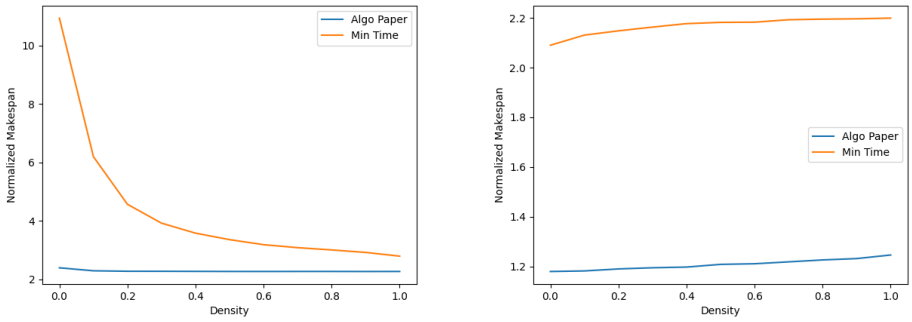
3.1 density

The first thing we can note on figure 4 is that the variation of density doesn't really impact the Paper algorithm, for all speedup models [9]. The Paper algorithm also give better results than the Min Time algorithm when working with the Amdahl's and Communication speedup model. The results are relatively close for the Roofline speedup model, this can be explained by the simplicity of this model which doesn't allow one strategy to really differentiate itself from the other. The increasing performance of the Min Time algorithm in the Amdahl's speedup model

[9] As a reminder the density determines the number of dependencies between tasks of two consecutive levels

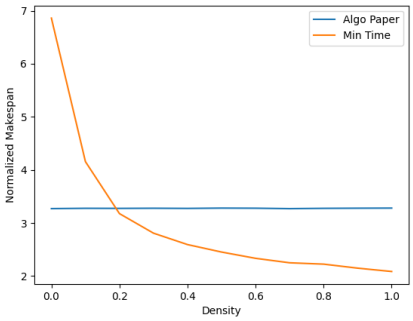
is due to the fact that the execution time of the Min Time algorithm is staying the same while the optimal execution increase thus diminishing the normalized makespan.

Concerning the General speedup model the Min Time algorithm is less consistent, but gives better results on a larger scale. It's a recurrent pattern that we will find again through our experiments. It can be explained by the fact that the choice of μ we did for the General model seems to be far from the best. We will sweep this problem under the carpet until section **4.1** where we will explain why we choose these particular values.

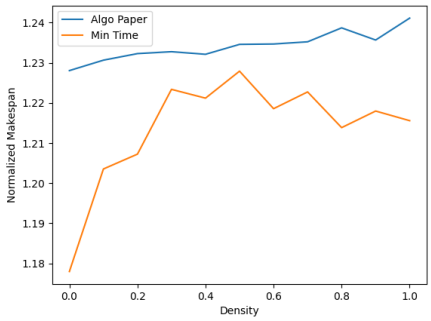


(a) Amdahl's

(b) Communication



(c) General



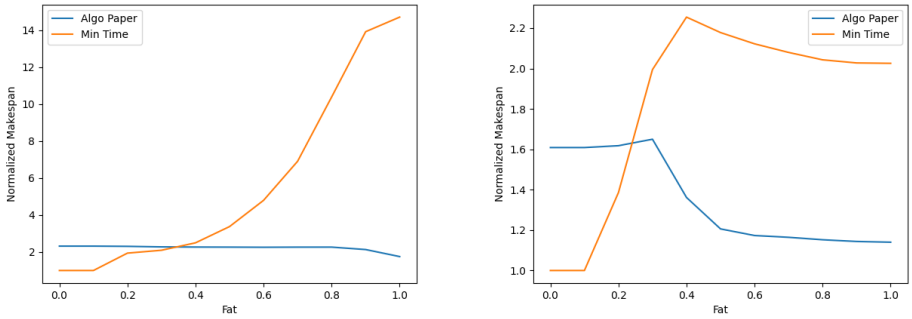
(d) Roofline

Figure 4: Variation of the normalized makespan depending on the algorithm, the density and the speedup model.

3.2 fatness

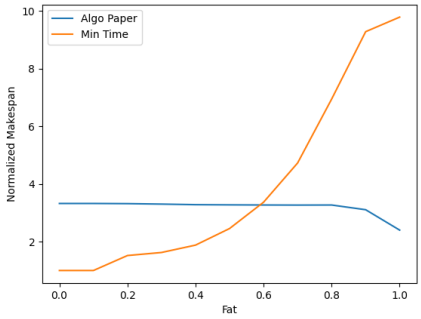
The results displayed on figure 5 are quite simple to explain and fit the hypothesis we made in section **2.4**. Except for the Roofline model we can see the performances of the Min Time Algorithm falling drastically when we increase the fatness of the graph. A fat closer to 1 means more tasks to compute at the same time and thus if the algorithm allocate too much processor to every single tasks it won't be able to do it [10] . The Min Time algorithm needs more steps to compute every levels of the graph and this is slowing down the process even if it computes every task faster than the Paper algorithm.

[10] The number of processor needed to compute all the tasks from the waiting queue at the same time will exceed P.

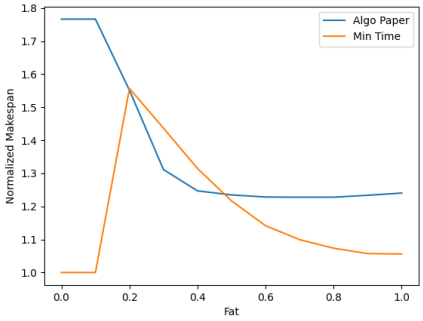


(a) Amdahl's

(b) Communication



(c) General



(d) Roofline

Figure 5: Variation of the normalized makespan depending on the algorithm, the fatness and the speedup model.

The conclusion we can make after this first set of experiment is that while not being better in every situation, the Paper algorithm seems to be more consistent than the Min Time algorithm, furthermore in order not to boost one algorithm instead of the other we will pick the following values for the rest of this paper:

- $jump = 1$
- $fat = 0.5$
- $density = 0.5$
- $regular = 0.5$

4 Impact of system parameters

Now that we have chosen the graph parameters we can variate the system parameters, P the number of processors and n the number of tasks. We will also take this experiment as an opportunity to assess some theoretical results from the literature [3].

4.1 Assessing the theoretical results

The paper "Online Scheduling of Moldable Task Graphs under Common Speedup Models" from A. Bennoit and al. gives us 4 theorems. They introduce a higher bound for the competitive ratio of the Paper algorithm for each speedup model. The interesting part is that this higher bound

is independent of the number of processor P but depend on μ. Meaning that if we pick the μ value from these theorem we are ensured that the competitive ratio of the Paper algorithm will be below a certain level no matter the number of processors available.

Theorem 1 (**Amdahl's**). *The Online algorithm along with the Paper allocation algorithm has a competitive ratio of 4.74 or below for any graph of tasks that follow the Amdahl's speedup model. This is achieved with mu* ≈ 0.271.

Theorem 2 (**Communication**). *The Online algorithm along with the Paper allocation algorithm has a competitive ratio of 3.61 or below for any graph of tasks that follow the Communication speedup model. This is achieved with mu* ≈ 0.324.

Theorem 3 (**General**). *The Online algorithm along with the Paper allocation algorithm has a competitive ratio of 5.72 or below for any graph of tasks that follow the General speedup model. This is achieved with mu* ≈ 0.211.

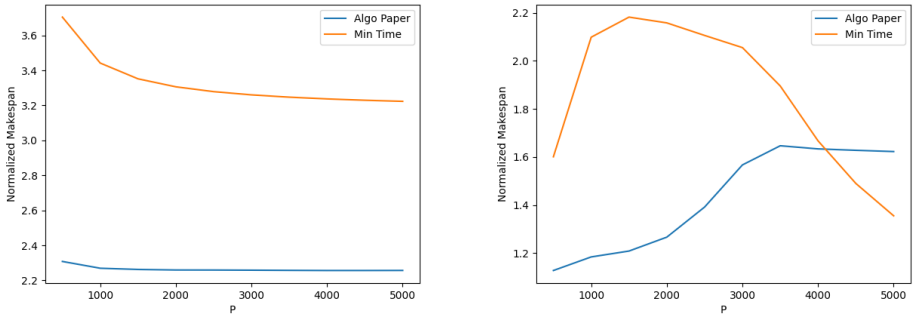
Theorem 4 (**Roofline**). *The Online algorithm along with the Paper allocation algorithm has a competitive ratio of 2.62 or below for any graph of tasks that follow the Roofline speedup model. This is achieved with mu* $= \frac{3-\sqrt{5}}{2}$.

We would like to emphasize the fact that the theorems don't present these values of μ as the best choice for optimizing the performance of the Paper algorithm. They state that if you choose these values you're sure that the competitive ratio of the Paper algorithm will be below a certain point. We leave to the reader the pleasure to discover the demonstrations of these theorems by reading the paper previously named.

In order to assess the theorems experimentally we variate the number of processor P, the results are shown on the figure 6. The results show that the normalized makespan of the Paper Algorithm stay way below the theoretical bound for every speedup model.

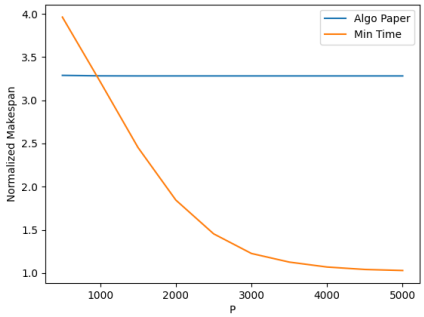
Furthermore, except again for the Roofline speedup model, we can see the performance of the Min Time algorithm increasing when we increase the number of processor. This phenomenon has the same explanation as in the last section. Indeed, when the number of processors is not high enough the Min Time algorithm can't compute all the tasks available at the same time hence increasing the total execution time.

The Paper algorithm seems to be better overall for all the speedup models except General. In addition we observe an odd behavior of the Paper algorithm under the general speedup model. This straight line can be explained by the choice of μ. If μ is too low, the Paper algorithm will tend to allocate a lower number of processor to every task and will not use all the processors available. We observe the same straight line at different makespan value for every value of μ below a certain threshold, when we pass this threshold and keep increasing μ the behavior change as we will see on section **5**.

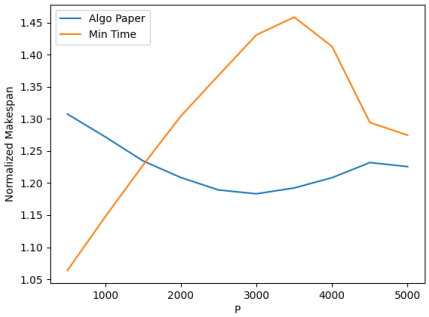


(a) Amdahl's

(b) Communication



(c) General

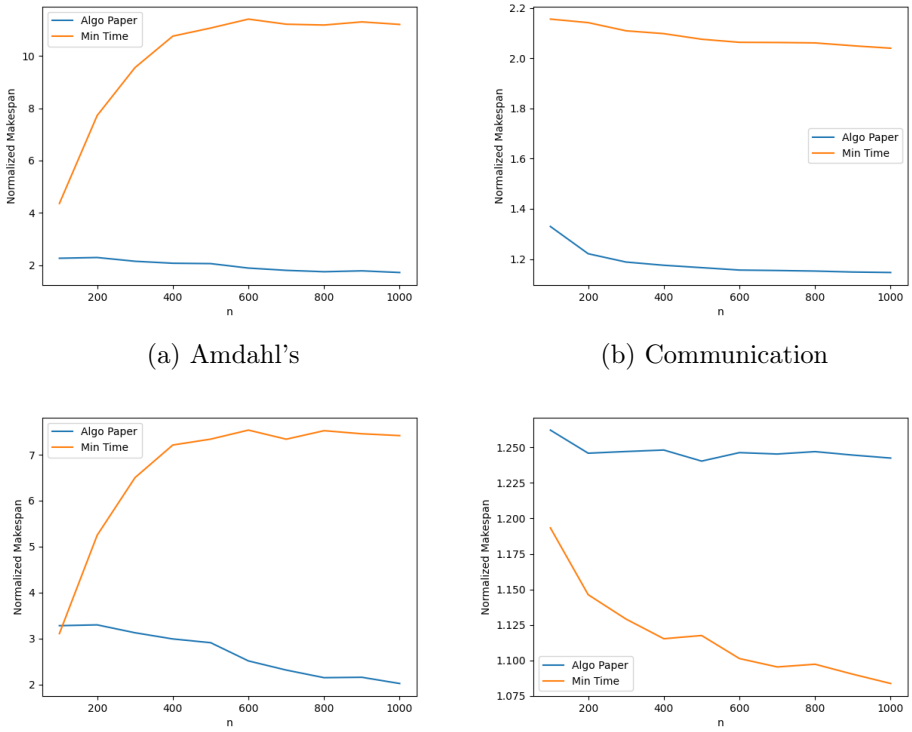


(d) Roofline

Figure 6: Variation of the normalized makespan depending on the algorithm, the number of processors and the speedup model.

4.2 Variation of the number of tasks

The second system parameter we can variate is the number of tasks (figure 7). The Paper algorithm seems consistently better than the Min Time algorithm except for the Roofline model where the results are close. You may note that the normalized makespan of the Paper Algorithm under the General speedup model is decreasing when the number of task is increasing, which may seem counter-intuitive. In fact, the execution time of the Paper algorithm is increasing but slower than the optimal execution time resulting in a decreasing normalized makespan.



(a) Amdahl's

(b) Communication

(c) General

(d) Roofline

Figure 7: Variation of the normalized makespan depending on the algorithm, the number of tasks and the speedup model.

This second set of experiment has allowed us to assess theoretical results on the higher bound of the competitive ratio for the Paper Algorithm. We also confirmed our statement that the Paper algorithm may not be better in all situations but is more stable than the Min Time algorithm. We are not entirely satisfied by how the Paper algorithm perform especially under the General speedup model. In the next section we will try to find a more advantageous value of μ for each speedup model. We will lose the higher bound, but we hope to gain some performances on the range of parameters we are looking at.

5 Impact of processor adjustment μ

We are looking for a better value of μ, but first of all we need to find in what range we can take this value. In order to allocate processors we have to satisfy the following condition :

$$\frac{t_j(p)}{t_j^{min}} \leq \frac{1-2\mu}{\mu(1-\mu)}$$

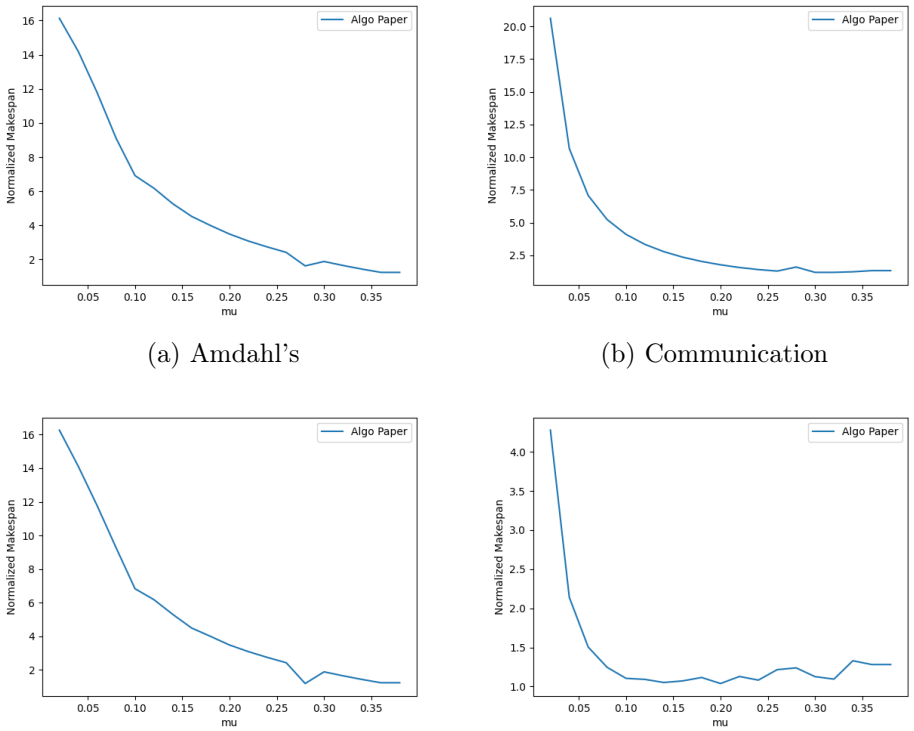
In other world, for every task j we have to found at least one value for p that satisfy this condition. If μ is to high such value can't always be found. Considering that $t_j(p) \geq t_j^{min} \quad \forall p \in [1, P]$ and that $t_j(p_j^{max}) = t_j^{min}$ by definition, we can deduce that the lower value for $\frac{t_j(p)}{t_j^{min}}$ is 1 and it can always be obtain by taking $p = p_j^{max}$. The maximum value of μ that satisfy, $1 \leq \frac{1-2\mu}{\mu(1-\mu)}$,

is $\mu = \frac{3-\sqrt{5}}{2}$. Finally μ has to be in the range :

$$0 < \mu \leq \frac{3-\sqrt{5}}{2} = \mu_{max}$$

5.1 Finding an alternate μ

We now variate μ with the standard parameters of $P = 1500$ and $n = 500$, we didn't bother to display the Min Time algorithm because it doesn't depend on μ.



(a) Amdahl's

(b) Communication

(c) General

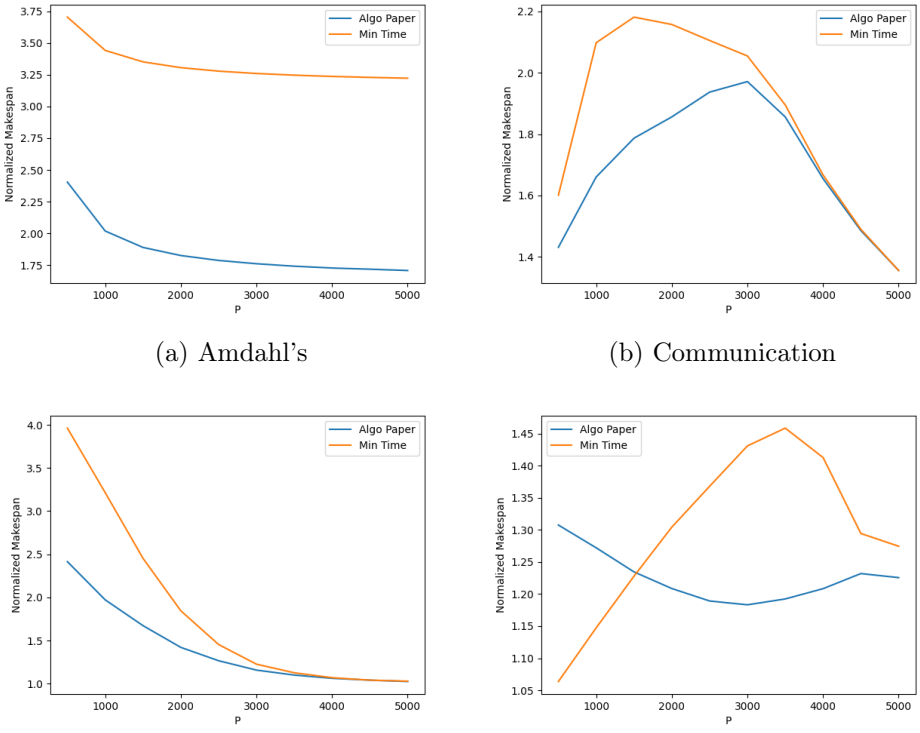
(d) Roofline

Figure 8: Variation of the normalized makespan of the Paper algorithm depending on μ and the speedup model.

The results on figure 8 are pretty self explanatory, the normalized makespan of the Paper Algorithm is decreasing when μ is increasing. We could take the value of μ with which the normalized makespan is minimal, but for simplicity purposes we will take the maximum value of $\mu = \mu_{max}$ for every speedup models. With this new value of μ we expected to get better results for the Paper Algorithms than then previous ones. In order to ascertain that fact we will repeat the last set of experiment by changing P and n again, but with the new value of μ.

5.2 Variation of the number of processors with $\mu = \mu_{max}$

As we can see on the figure 9, the Paper algorithm is now more efficient than the Min Time algorithm for every speedup model, we can also note the different behavior under the General speedup model which is now more acceptable.



(a) Amdahl's

(b) Communication

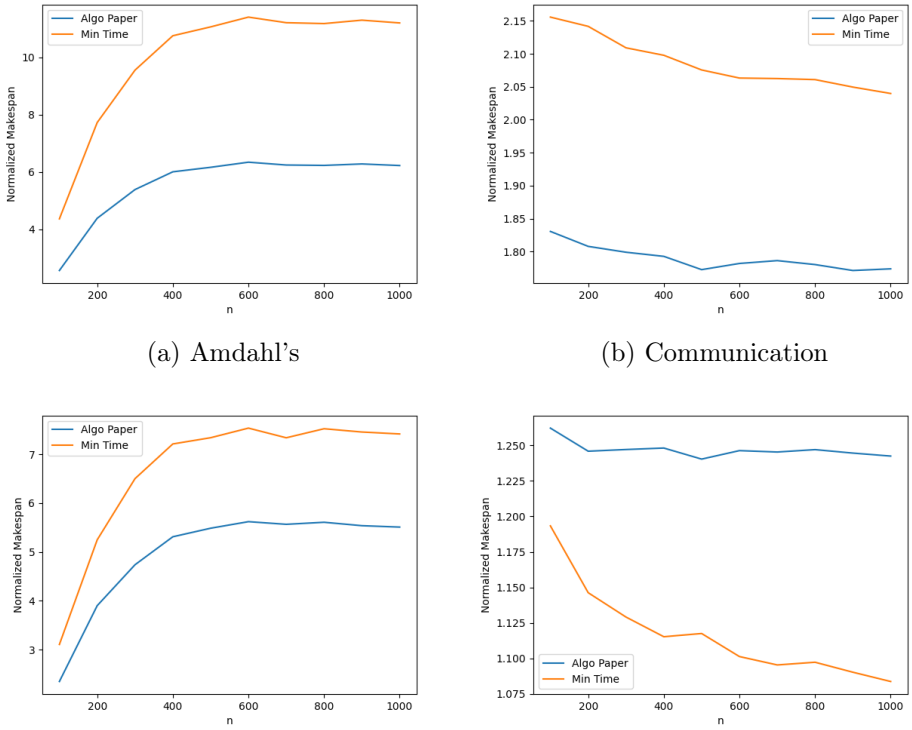
(c) General

(d) Roofline

Figure 9: Variation of the normalized makespan depending on the algorithm, the number of processors and the speedup model with $\mu = \mu_{max}$.

5.3 Variation of the number of tasks with $\mu = \mu_{max}$

When we variate n (figure 10), the Paper algorithm is still better than the Min Time algorithm but on some conditions it gives worst results than with the previous value of μ. It's not surprising at all because we look at the influence of μ with specific P and n values. In addition, we didn't took the best value but one that was close to it. Our approach is purely experimental and its only goal was to show that the Paper algorithm could perform better under most circumstances with an other choice for μ.



(a) Amdahl's (b) Communication

(c) General (d) Roofline

Figure 10: Variation of the normalized makespan depending on the algorithm, the number of tasks and the speedup model with $\mu = \mu_{max}$.

6 Conclusion and future work

Our first objective was the build a functioning simulation to run experiment. The code presented on github [11] fulfill this function. Some improvements in term of complexity should be consider if someone would like to run experiments on wider task graphs. We choose to code the graph with an adjacency matrix allowing use to consult it with a $O(1)$ complexity at the cost of a $O(n^2)$ complexity when creating it. When working with a high number of task, this $O(n^2)$ complexity could become problematic.

Our second goal was to assess the results on a higher bound for the competitive ratio of the Paper algorithm. This part was conclusive, but we also discover that the Paper algorithm wasn't performing as well as it could, especially under the General speedup model. This was mainly due to the choice of the processor adjustment μ, when choosing an alternate value of μ the Paper algorithm show better results than the Min Time algorithm under almost all circumstances. The cost of this increase in performance is the loss of the theoretical upped bound, meaning that for a high value a n and $fatness$, the competitive ratio could explode. On the range experimented this new μ seems to be a better fit.

There is still some other point to explore. First we didn't display the Min Area algorithm because it was consistently far worse than the two other. Its normalized makespan seemed to decrease when n and the $fatness$ increase, meaning that for a higher value of n, it could be a viable option. This leads us to the first point, it could be interesting to compare the performance of the algorithms with much wider graph. The second point would be to introduce failure into the model, it shouldn't complicate too much the implementation and offer a new range of experiments. The third point would be to find an optimal value for μ depending on the system and graph parameters. We have done it experimentally, but it would be a great addition to prove it theoretically even if it seems rather complicated.

[11] *github.com/thomasverrecchia/Internship_Kansas_University*

References

[1] S. Hunold (2014) "Scheduling Moldable Tasks with Precedence Constraints and Arbitrary Speedup Functions on Multiprocessors"

[2] G. Demirci and al. (2018) "A Divide and Conquer Algorithm for DAG Scheduling under Power Constraints"

[3] A. Bennoit and al. (2019) "Online Scheduling of Moldable Task Graphs under Common Speedup Models"

[4] frs69wq(2013) DAGGEN https://github.com/frs69wq/daggen

[5] A. Bennoit and al. (2020) "Resilient Scheduling of Moldable Jobs on Failure-Prone Platforms"

[6] A. Benoit and al. (2021) "Resilient scheduling of moldable parallel jobs to cope with silent errors. IEEE Transactions on Computers".

[7] R. Lepere, D. Trystram, and G. J. Woeginger. (2001) "Approximation algorithms for scheduling malleable tasks under precedence constraints. In ESA, pages 146–157"

[8] (2005) K. Jansen and H. Zhang."Scheduling malleable tasks with precedence constraints. In SPAA, page 86–95"

[9] n and H. Zhang. (2006) "An approximation algorithm for scheduling malleable tasks under general precedence constraints. ACM Trans. Algorithms, 2(3):416–434".