Moldable and Rigid : models and algorithms

February 12, 2025

1 Introduction

This work investigates the online scheduling of parallel task graphs on a set of identical processors, where each task in the graph is *moldable* or *rigid*. In the scheduling literature, a moldable task (or job) is a parallel task that can be executed on an arbitrary but fixed number of processors. The execution time of the task depends upon the number of processors used to execute it, which is chosen once and for all when the task starts its execution but cannot be modified later on during execution. This corresponds to a variable static resource allocation, as opposed to a fixed static allocation (*rigid* tasks) where both the exectution time and the number of processors are fixed [4].

Moldable tasks offer a nice trade-off between rigid and and malleable tasks: they easily adapt to the number of available resources, contrarily to rigid tasks, while being easy to design and implement, contrarily to malleable tasks. This explains that many computational kernels in scientific libraries for numerical linear algebra and tensor computations are provided as moldable tasks that can be deployed on a wide range of processor numbers. We assume that the scheduling of each task is non-preemptive and without restarts [5], which is a highly desirable approach to avoid high overheads incurred by checkpointing partial results, context switching, and task migration.

Because of the importance and wide availability of moldable tasks, scheduling algorithms for such tasks have received considerable attention. The scheduling problem comes in many flavors:

- **Offline Scheduling vs. Online Scheduling.** In the offline version of the scheduling problem, all tasks are known in advance, before the execution starts. The problem is $\mathcal{NP}$-complete, and the goal is to design good approximation algorithms. On the contrary, in the online version of the problem, tasks are released on the fly, and the objective is to derive competitive ratios [8] for the performance of a scheduling algorithm against an optimal offline scheduler, which knows in advance all the tasks and and their dependencies in the graph. The competitive ratio is established against all possible strategies devised by an adversary trying to force the online algorithm to take *bad* decisions.
- **Independent Tasks vs. Task Graphs.** There are two versions of the online problem, with independent tasks or with task graphs. For the version with independent tasks, the tasks are released on the fly and the scheduler discovers their characteristics only upon release. For the version with task graphs, the whole graph is released at the start, but the scheduler discovers a new task and its characteristics only when all of its predecessors have completed execution. In other words, the shape of the graph and the nature of the tasks are not known in advance and are revealed only as the execution progresses.

In this work, we investigate the most difficult instance of the problem, namely, the online scheduling of moldable tasks graphs, with the goal of minimizing the overall completion time, or the makespan.

2 Moldable Online : Problem Statement

In this section, we formally present the online scheduling model and the objective function.

2.1 Model and Objective

We consider the online scheduling of a ***Directed Acyclic Graph (DAG)*** of moldable tasks on a platform with P identical processors. Let $G = (V, E)$ denote the task graph, where $V = \{1, 2, \ldots, n\}$ represents a set of n tasks and $E \subseteq V \times V$ represents a set of precedence constraints (or dependencies) among the tasks. An edge $(i, j) \in E$ indicates that task j depends on task i, and therefore it cannot be executed before task i is completed. Task i is called the ***predecessor*** of task j, and task j is called the ***successor*** of task i. In this work, we do not consider the costs associated with the data transfers between dependent tasks.

Algorithm 1: Compute_Ratio(j)

```
Input: Task $j$
                                                  // Task to be processed
Output: Ratio $R_j$ and processor allocation $p_j$
                    // Optimal ratio and processor allocation for task $j$
// Compute Initial Allocation
1 Compute $p_j^{\max}$           // The number of processors that minimizes execution time
2 Compute $t_j^{\min} = t_j(p_j^{\max})$ and $a_j^{\min} = a_j(1)$      // Minimum execution time and area
3 Find $p_j \in [1, p_j^{\max}]$ minimizing $R_j \triangleq \max\left(\frac{t_j(p)}{t_j^{\min}}, \frac{a_j(p)}{a_j^{\min}}\right)$      // Balance time and area
4 return $R_j, p_j$                   // Return optimal ratio and processor allocation
```

The tasks are assumed to be ***moldable***, meaning that the number of processors allocated to a task can be determined by the scheduling algorithm at launch time, but once the task has started executing, its processor allocation cannot be changed. The execution time $t_j(p_j)$ of a task j is a function of the number p_j of processors allocated to it, and we assume that the processor allocation must be an integer between 1 and P.

From the execution time function of the task j, we can further define the ***area*** of the task as a function of the processor allocation as follows: $a_j(p_j) = p_j t_j(p_j)$. Intuitively, the area represents the total amount of processor resources utilized over the entire period of task execution.

Remark 1. *We don't assume anything more, the instance simply consist of a set of task defined by their time function and precedence constraints. For instance, we don't assume tasks have a request in terms of number of processors*

In this work, we consider the ***online scheduling*** model, where a task becomes available only when all of its predecessors have been completed. This represents a common scheduling model for ***dynamic task graphs***, whose dependencies are only revealed upon task completions [7, 5, 1, 3]. Furthermore, when a task j is available, it's exact time function $t(p)$ (i.e. the execution time for any number of processors in $[1, P]$) becomes known to the scheduling algorithm as well. The goal is to find a feasible schedule of the task graph that minimizes its overall completion time or ***makespan***, denoted by T. The performance of an online scheduling algorithm is measured by its competitive ratio: the algorithm is said to be c***-competitive*** if, for any task graph, its makespan T is at most c times the makespan T^{opt} produced by an optimal offline scheduler, i.e., $\frac{T}{T^{\text{opt}}} \leq c$. Note that the optimal offline scheduler knows in advance all the tasks and their speedup models, as well as all dependencies in the graph. The competitive ratio is established against all possible strategies by an adversary trying to force the online algorithm to take bad decisions.

Remark 2. *The main result of the last paper were that if we assume we know exactly the speedup model of tasks in advance, we could design an algorithm roughly optimal in the worst case, that only works on the specific model. Therefore, it was useless in practice. The main result of this work is to design a single algorithm that works for all models, and which gives guarantees almost as good for each model.*

3 Moldable Online Algorithm

In this section, we present Fair, an online scheduling algorithm.

3.1 Algorithm Description

The algorithm begins by computing for each task j a processor allocation p_j and its associated ratio R_j using the Compute_Ratio subroutine (Algorithm 1). First, we define the maximum number of processors $p_j^{\max}$ that task j can use as follows:

$$p_j^{\max} = \arg\min_p t_j(p), \tag{1}$$

where $t_j(p)$ is the execution time of task j on p processors. This represents the number of processors that minimize the execution time of task j.

Next, we define two normalized metrics for task j:

Algorithm 2: FAIR

```
initialize waiting queue $Q$                                        // Queue to hold available tasks
$R = 1$                                                             // Initialize maximum ratio to 1
when at time 0 or a task completes do
    // Update Task Metrics
    for each new task $j$ (that just became available/discovered) do
        $(R_j, p_j) \leftarrow$ Compute_Ratio($j$)                  // Compute ratio and allocation for task $j$
        $R \leftarrow \max(R, R_j)$                                 // Update maximum ratio if necessary
        insert $j$ into $Q$ with $p_j$                              // Add task to waiting queue
    end
    // List Scheduling with Dynamic Allocation
    for each task $j$ in $Q$ do
        $p'_j \leftarrow \min(p_j, \lceil \mu(R) \cdot P \rceil)$   // Scale processor allocation using $\mu(R)$
        if available processors $\geq p'_j$ then
            allocate $p'_j$ processors to $j$                       // Allocate processors to task $j$
            execute $j$ immediately                                 // Start execution of task $j$
        end
    end
end
```

- The normalized execution time:

$$f_j(p) = \frac{t_j(p)}{t_j^{\min}}, \tag{2}$$

  where $t_j^{\min} = t_j(p_j^{\max})$ is the minimum execution time of task j.
- The normalized area:

$$g_j(p) = \frac{a_j(p)}{a_j^{\min}}, \tag{3}$$

  where $a_j^{\min} = a_j(1)$ is the minimum area of task j (i.e., the area when using 1 processor).

The ratio R_j for task j is then defined as:

$$R_j = \min_{p \in [1, p_j^{\max}]} \max\left(f_j(p), g_j(p)\right). \tag{4}$$

This ratio balances the trade-off between execution time and area. Specifically, it finds the processor allocation p that minimizes the maximum of the normalized execution time and normalized area. Since $g_j(p)$ is non-decreasing with p (more processors generally increase the area) and $f_j(p)$ is non-increasing with p (more processors generally reduce execution time), the optimization problem in Equation (4) can be efficiently solved using binary search in $O(\log P)$ time. This ensures that the algorithm scales well even for large systems.

Algorithm 2 outlines the online scheduling process of FAIR. When a task arrives or completes, the algorithm updates the maximum observed ratio R. During scheduling, each task's processor allocation is reduced to $\lceil \mu(R) \cdot P \rceil$ if p_j exceeds this value, where $\mu(R)$ is defined as:

$$\mu(R) = \frac{2R + 1 - \sqrt{4R^2 + 1}}{2R} \tag{5}$$

where R represents the maximum ratio among all observed tasks. This function ensures that the total number of processors used for each task doesn't exceed a certain fraction of the platform, to optimize the competitive ratio in the worst case. Tasks are then scheduled greedily whenever sufficient processors are available for their scaled allocation. This approach balances parallelism and resource efficiency while maintaining provable competitiveness.

Remark 3. *The choice of ordering for Q has no impact in the worst case competitive ratio, we can do FIFO, FILO, longest task first, largest area first.... Some strategies are tested in experiments.*

Remark 4. *The point of the paper is to say that this algorithm, with this function of μ is almost optimal in the worst case (see paper)*

4 Moldable Online Experimental Setup

4.1 Tasks Generation

Our experimental study evaluates scheduling heuristics across five task graph models with distinct time functions:

- ***Roofline Model*** [9]:

$$t(p) = \frac{w}{\min(p, \bar{p})} \tag{6}$$

  where $\bar{p} \leq P$ is the task's maximum parallelism degree. Tasks exhibit linear speedup until $\bar{p}$ processors.

- ***Communication Model*** [6]:

$$t(p) = w\left(\frac{1}{p} + c(p-1)\right) \tag{7}$$

  Combines perfect parallelism with linear communication overhead proportional to allocated processors.

- ***Amdahl's Model*** [2]:

$$t(p) = w\left(\frac{1-d}{p} + d\right) \tag{8}$$

  Features a parallel fraction (w) and fixed sequential fraction (d), modeling inherent serialization.

- ***Power Communication Model***:

$$t(p) = w\left(\frac{1-c}{p} + cp^{\gamma}\right) \tag{9}$$

  Generalizes communication costs with exponent $\gamma \in [0, 1]$. Recovers Amdahl's model ($\gamma = 0$) and Communication Model ($\gamma = 1$).

- ***General Model***:

$$t(p) = w\left(\frac{1-d}{\min(p, \bar{p})} + d + c(p-1)\right) \tag{10}$$

  Hybrid model combining roofline constraints, sequential overhead, and linear communication costs.

For all instances, tasks are generated randomly using the following parameters

- **Sequential length of the task** : w is generated in the Task Generation Framework and corresponds to $t(1)$ for all models.
- **Parallelism bounds**: $p \in [1, 512]$ (uniform)
- **Sequential fraction**: $d \in [10^{-3}, 10^{-0.5}]$ (log-uniform via $10^{\mathrm{Unif}(-3,-0.5)}$)
- **Communication coefficient**: $c \in [10^{-6}, 10^{-2}]$ (log-uniform via $10^{\mathrm{Unif}(-6,-2)}$)
- **Power exponent**: $\gamma \in [0, 1]$ (uniform)

4.2 Graph Generation

Task instances are generated with daggen which generates random, synthetic task graphs for the purpose of simulation. (git). Graph are organised in layers, with precedence constraints from a layer to the next one. If the Jump parameter is more than 1 precedence constraints can jump a layer.

Topology parameters control graph structure:

- **Density** $\in [0, 1]$ (default : 0.5): Determines the numbers of dependencies between taks of two consecutive DAG levels.
- **Fat** $\in [0, 1]$ (default : 0.5) : Width of the DAG, that is maximum number of tasks that can be executed concurrently. 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
- **Regular** $\in [0, 1]$ (default : 0.5): Regularity of the distribution of tasks between the different levels of the DAG
- **Jump** $\in [1, 9]$ (default : 5): maximum number of levels spanned by inter-task communications. This allows to generate DAGs with execution paths of different lengths
- **Number of tasks:** $n \in [500, 5000]$ (default : 2500).
- **Number of processors:** $P \in [124, 1024]$ (default : 512).

4.3 Heuristics

For each model, we compare five scheduling strategies:

- **ICPP22** : Heuristic from ICPP paper. **The heuristic DEPENDS on the speedup model**
- **TOPC24** : Heuristic from TOPC paper. **The heuristic DEPENDS on the speedup model**
- **Fair** : Heuristic from this paper. **The heuristic DOESN'T DEPEND on the speedup model**
- **minTime** : Minimize tasks execution time
- **minArea** : Minimize tasks area

Each of the strategy only impact the processor allocation function, but the ordering of the queue doesn't impact the analysis, therefore we test various **priority rules**, for ordering the queue Q:

- FIFO (default method): First In First Out
- procs : Largest number of processors first
- area : largest area ($a(p) = t(p) \times p$) first
- length : largest execution time first (sort by decreasing $t(p)$)

4.4 Summary of experimental Parameters

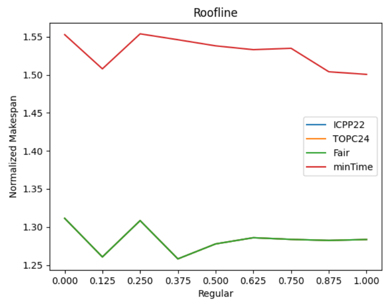
Param	**Default**	**Test Values**	**Type**	**Description**
n	2500	[500,...,5000]	Scale	Number of tasks
P	512	[124,...,1024]	System	Available processors
Priority	FIFO	[FIFO,...]	Policy	Queue ordering strategy
Density	0.5	[0,...,1]	Graph	Edge probability
Fat	0.5	[0,...,1]	Graph	Width/depth ratio
Regular	0.5	[0,...,1]	Graph	Degree uniformity
Jump	5	[1,...,9]	Graph	maximum jump distance between layers

4.5 Experimental Protocol

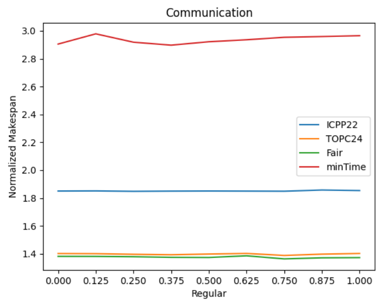
We will do one experiment per parameter. For each parameter, we generate one figure for each different speedup models for tasks. One subfigure = one couple (parameter, model). Each figure represent a parameter, and contains a subfigure for each model. We proceed as follow:

- Set values of all parameters to default value, except the one we experiment on, which will be tried for all its **Test values**.
- Once all parameters are fixed, generate 50 random instances, simulate them with all heuristics. The output function is the ratio $\frac{T}{L}$ where L is a lower bound on the optimal execution time
- Average the 50 values for each heuristic (if we chose lines figure), or make boxplots (if we chose boxplot figures)
- Generate all figures
- Generate tables : one table for the average values among all experiments for each couple (heuristic, speedup model)+overall average. One table generated similarly with maximum values.

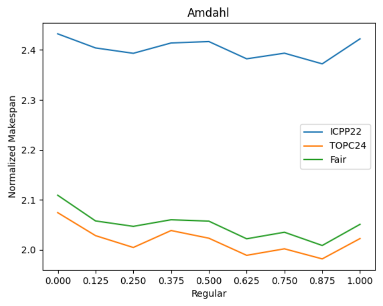
4.6 Lines Figures



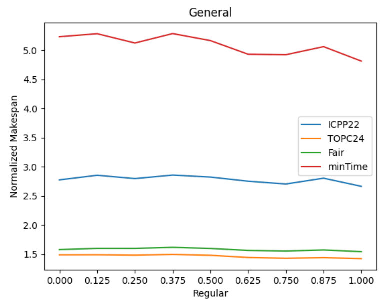
(a) Roofline



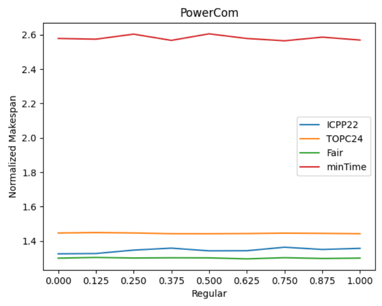
(b) Communication



(c) Amdahl

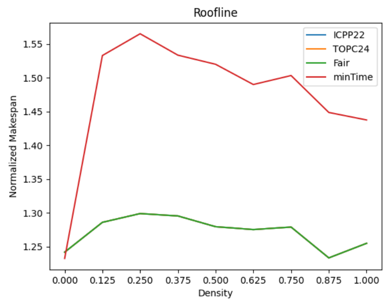


(d) General

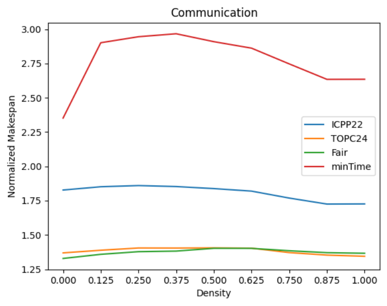


(e) PowerCom

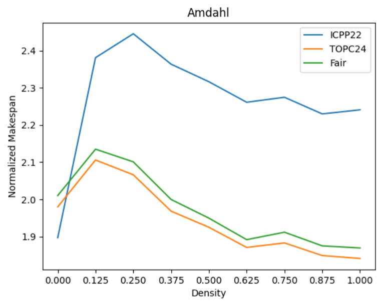
Figure 1: Lines Figure for Regular



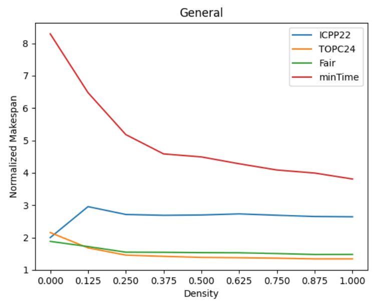
(a) Roofline



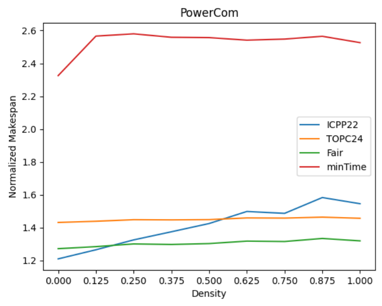
(b) Communication



(c) Amdahl

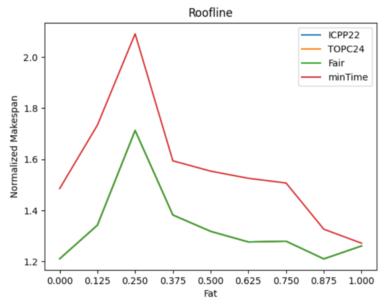


(d) General

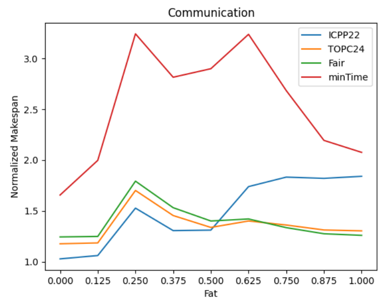


(e) PowerCom

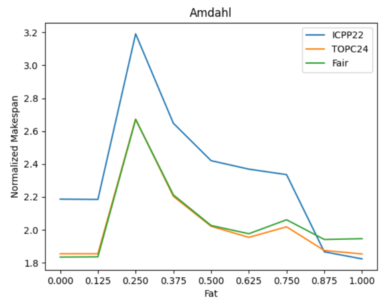
Figure 2: Lines Figure for Density



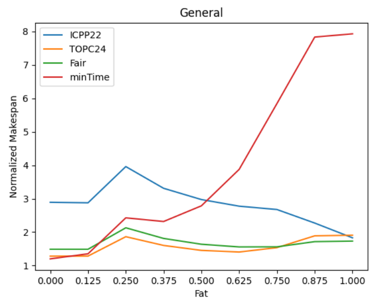
(a) Roofline



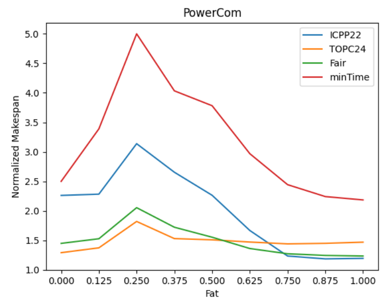
(b) Communication



(c) Amdahl

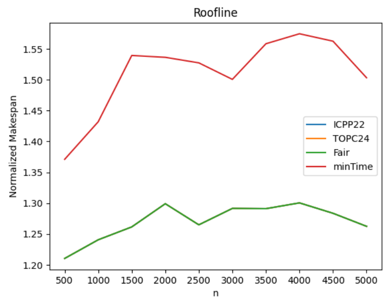


(d) General

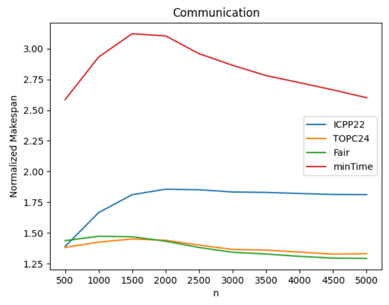


(e) PowerCom

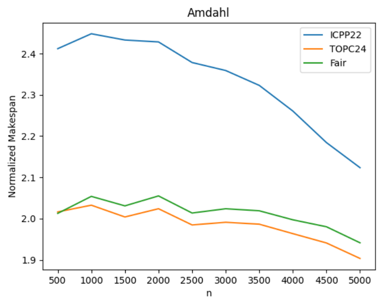
Figure 3: Lines Figure for Fat



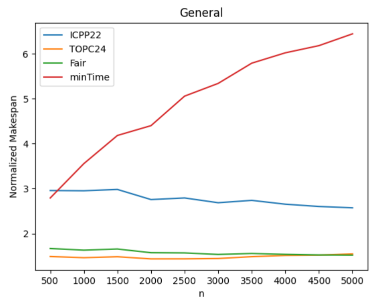
(a) Roofline



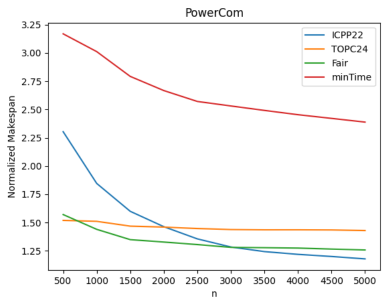
(b) Communication



(c) Amdahl

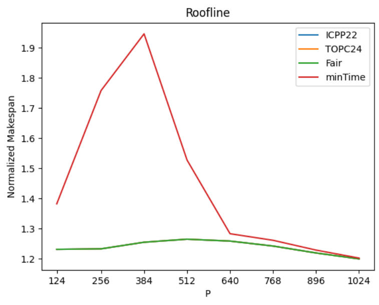


(d) General

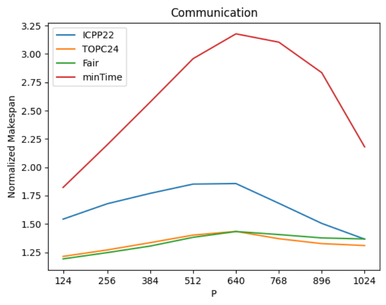


(e) PowerCom

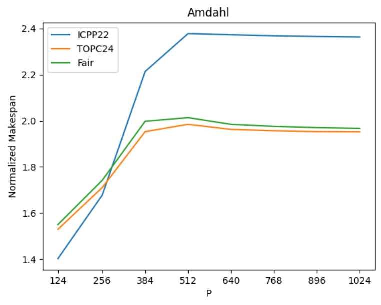
Figure 4: Lines Figure for n



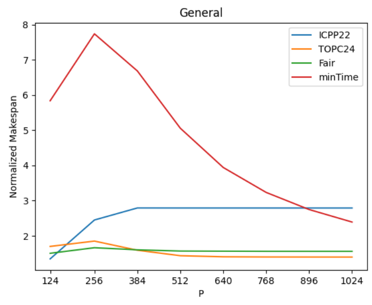
(a) Roofline



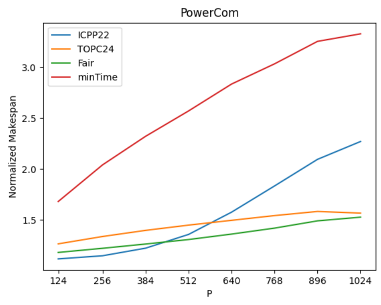
(b) Communication



(c) Amdahl

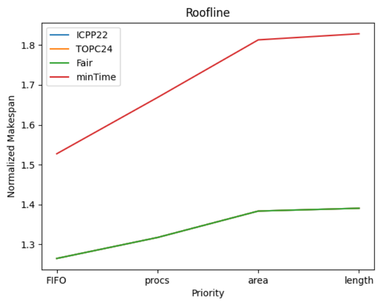


(d) General

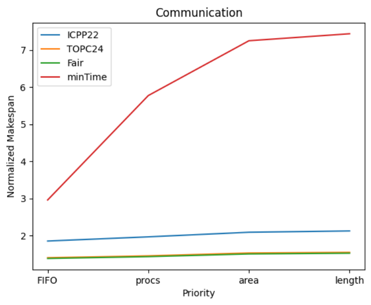


(e) PowerCom

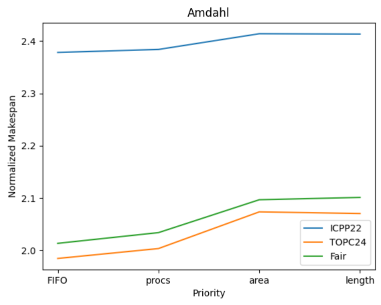
Figure 5: Lines Figure for P



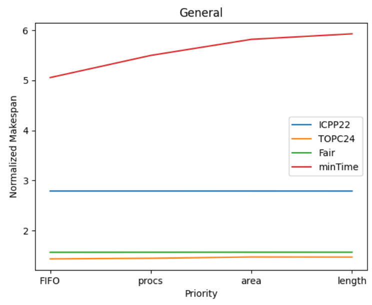
(a) Roofline



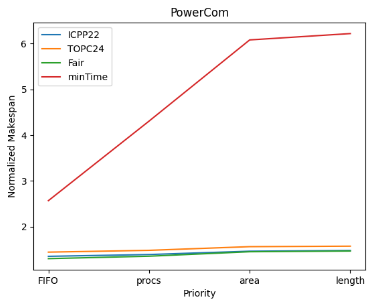
(b) Communication



(c) Amdahl

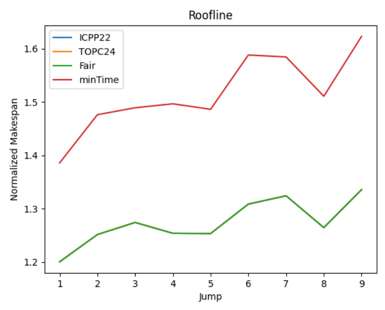


(d) General

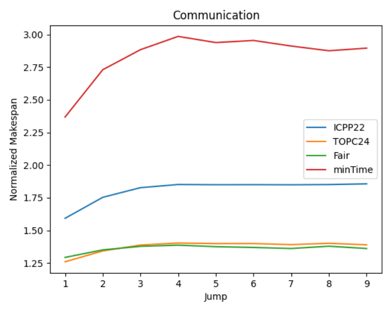


(e) PowerCom

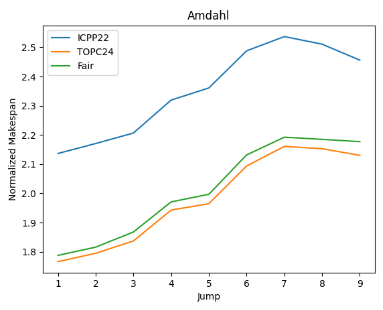
Figure 6: Lines Figure for Priority



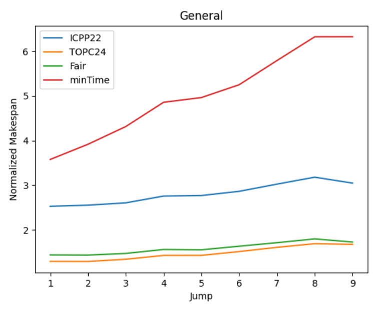
(a) Roofline



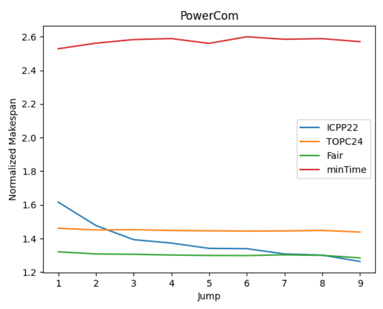
(b) Communication



(c) Amdahl



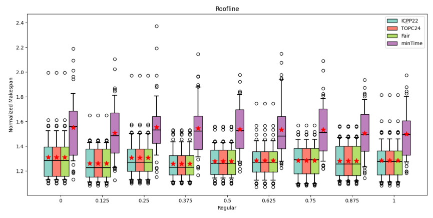
(d) General



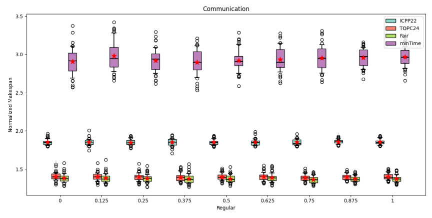
(e) PowerCom

Figure 7: Lines Figure for Jump

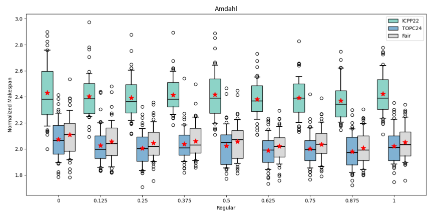
4.7 Boxplot Figures



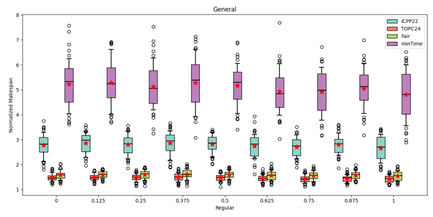
(a) Roofline



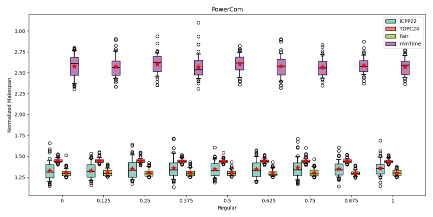
(b) Communication



(c) Amdahl

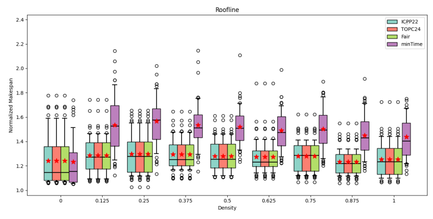


(d) General

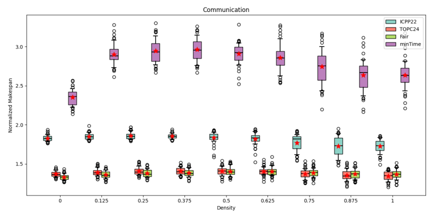


(e) PowerCom

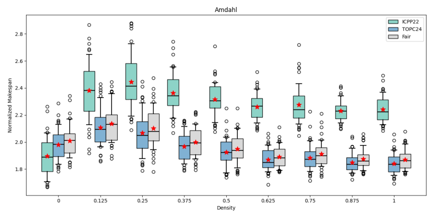
Figure 8: Boxplot Figure for Regular



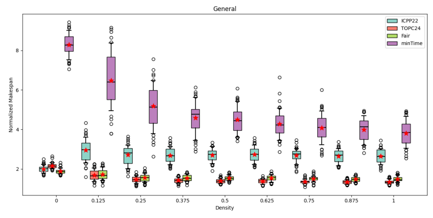
(a) Roofline



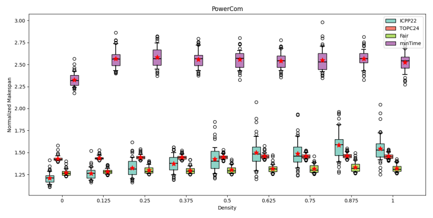
(b) Communication



(c) Amdahl

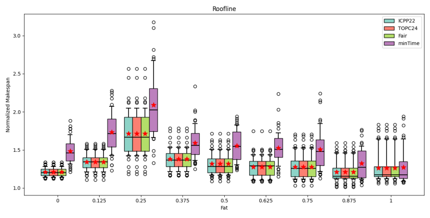


(d) General

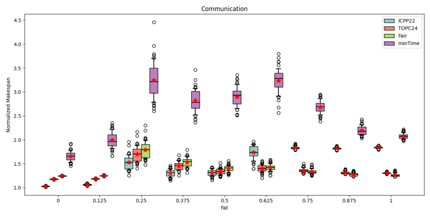


(e) PowerCom

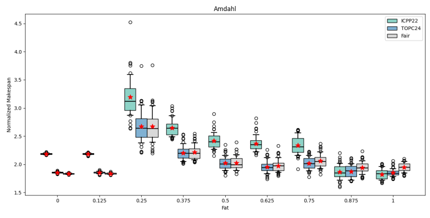
Figure 9: Boxplot Figure for Density



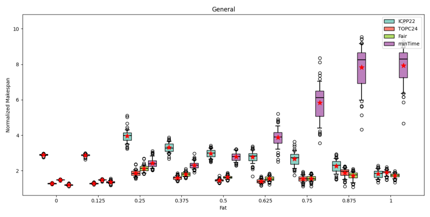
(a) Roofline



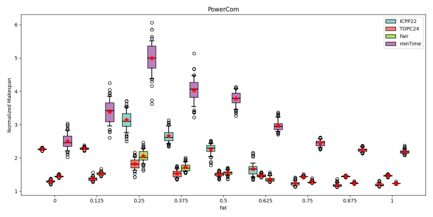
(b) Communication



(c) Amdahl

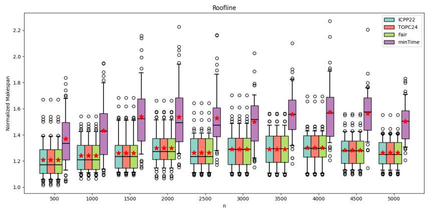


(d) General

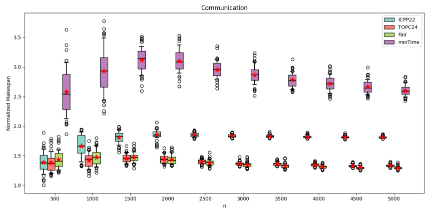


(e) PowerCom

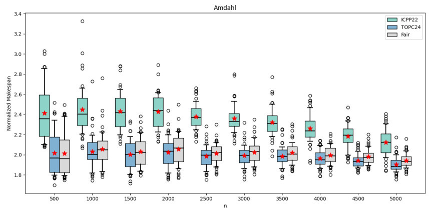
Figure 10: Boxplot Figure for Fat



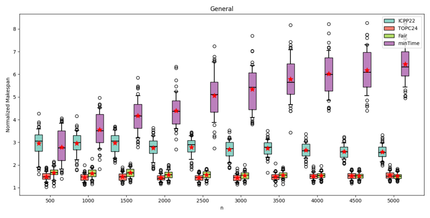
(a) Roofline



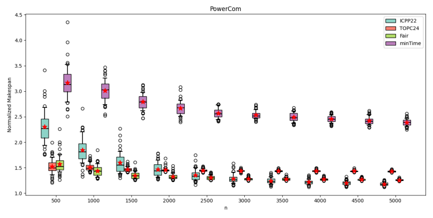
(b) Communication



(c) Amdahl

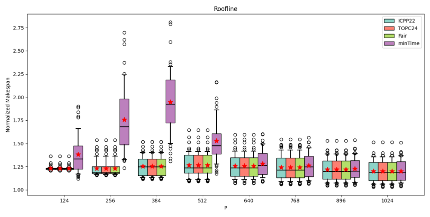


(d) General

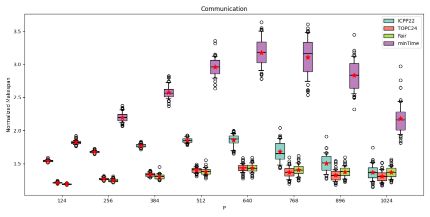


(e) PowerCom

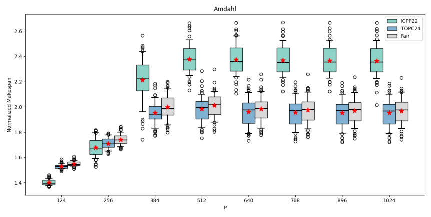
Figure 11: Boxplot Figure for n



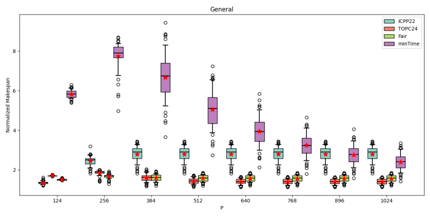
(a) Roofline



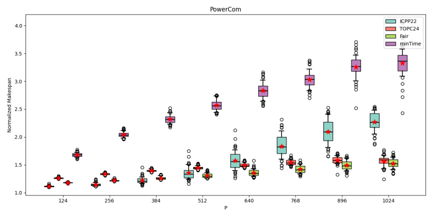
(b) Communication



(c) Amdahl

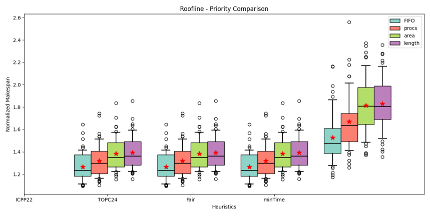


(d) General

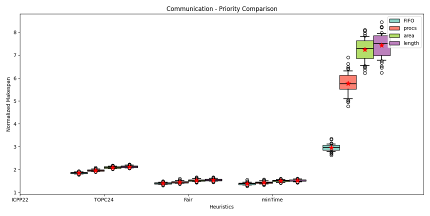


(e) PowerCom

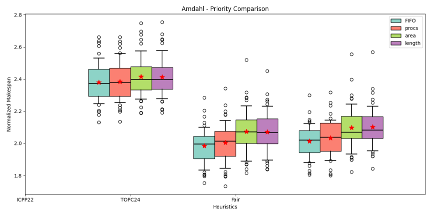
Figure 12: Boxplot Figure for P



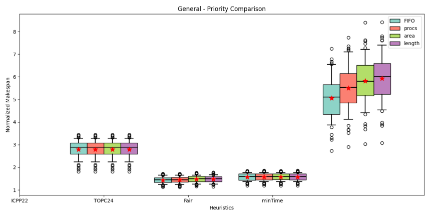
(a) Roofline



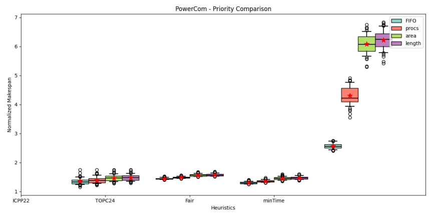
(b) Communication



(c) Amdahl

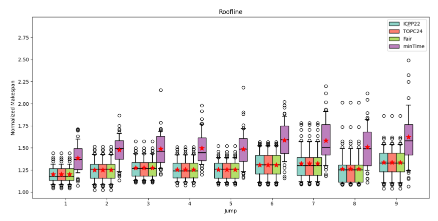


(d) General

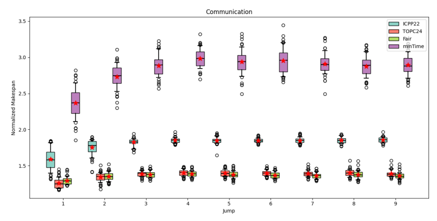


(e) PowerCom

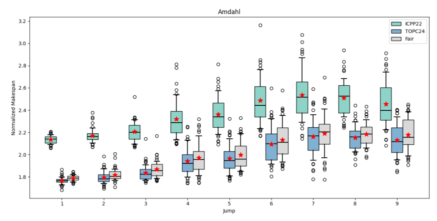
Figure 13: Boxplot Figure for Priority



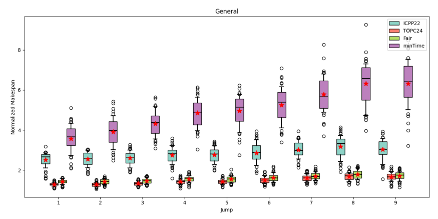
(a) Roofline



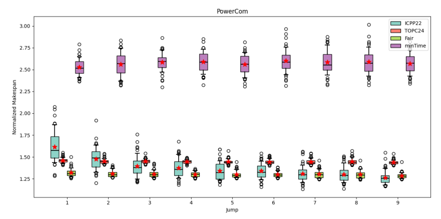
(b) Communication



(c) Amdahl



(d) General



(e) PowerCom

Figure 14: Boxplot Figure for Jump

4.8 Average Values

Table 1: Average Values for Each Model and Heuristic

Model	ICPP22	TOPC24	Fair	minTime	minArea
Roofline	1.28	1.28	1.28	1.52	11.28
Communication	1.75	1.38	1.38	2.97	10.22
Amdahl	2.32	1.98	2.01	21.31	7.88
General	2.74	1.50	1.60	4.86	5.37
PowerCom	1.52	1.46	1.35	2.84	12.76
Average	1.92	1.52	1.52	6.70	9.50

4.9 Maximum Values

Table 2: Maximum Values for Each Model and Heuristic

Model	ICPP22	TOPC24	Fair	minTime	minArea
Roofline	2.56	2.56	2.56	3.18	117.75
Communication	2.22	2.17	2.30	8.45	34.30
Amdahl	4.52	3.75	3.76	33.26	23.45
General	5.11	2.64	2.70	10.34	14.76
PowerCom	3.71	2.16	2.46	6.83	69.98
Maximum	5.11	3.75	3.76	33.26	117.75

Remark 5. *The point of this paper is that ICPP and TOPC had one different heuristic for EACH possible speedup model. And they couldn't deal with tasks that didn't respect the model. FAIR is a general heuristic that works for any kind of tasks, which is as good in practice and in theory, therefore the results are great.*

Algorithm 3: General Heuristics

```
initialize waiting queue $Q$                                        // Queue to hold available tasks
when at time 0 or a task completes do
    // Update Task Metrics
    for each new task $j$ (that just became available/discovered) do
        $\zeta_j \leftarrow$ Compute_Category($j$)                  // Compute category of task $j$
        insert $j$ into $Q$ with $\zeta_j$                          // Add task to waiting queue
    end
    Sort $Q$ by increasing $\zeta_j$ (ties broken arbitrarily) // List Scheduling
    for each task $j$ in $Q$ do
        if available processors $\geq p_j$ then
            execute $j$ immediately                                  // Start execution of task $j$
        end
    end
    if "bad condition" then
                                                                     // condition depend of heuristics
        terminate some tasks                                         // to define
        launch some tasks instead                                    // to define
    end
end
```

5 Online Rigid Scheduling

5.1 Model

Same model for the graph structure, but tasks have fixed processor allocation. Therefore a task j has 2 parameters : number of processors p_j and length (or estimated length) t_j.

5.2 Algorithm

The general scheduling heuristic (Algorithm 2) follows a priority-driven list scheduling approach. When processors become available, it first categorizes newly available tasks through a `Compute_Category()` function (detailed in Section 5.3.1). Tasks are then inserted into a waiting queue and sorted by their category values. The algorithm subsequently allocates processors using a greedy first-fit strategy, executing tasks whenever sufficient processors are available. An adaptive component (in blue) allows for dynamic reconfiguration—specific "bad condition" checks (heuristic-dependent) may trigger task preemptions and replacements to optimize resource utilization. The key differentiation between heuristic variants lies in their reconfiguration conditions.

Examples of conditions:

- **Strong condition (Heuristic A)**: Whenever the task j of smallest category in Q could be launched by preempting tasks of category strictly greater, terminate them to launch j.
- **Weak condition (Heuristic B)** : Whenever less than $\frac{P}{2}$ processors are used and the task j of smallest category in Q has a category strictly smaller than all the tasks running, terminate some tasks to launch j.
- **No conditions (Heuristic C)** : Never interrupt tasks.

Heuristic A would be nearly-optimal in the worst case but likely inefficient in practice since it has too much task termination. (*Main result of rigid paper : heuritic A is* $\theta(\log(n))$ *approx, and* $o(\log(n))$ *is impossible*). Heuristic C would be probably ok in most cases but has no theoretical guarantee (could be a factor $\theta(n)$ from the optimal).

5.3 What are categories?

5.3.1 Criticality and Category

Definition 1. *Given a task $\mathcal{T}$ of length t, we define its* ***earliest start time*** *as s^∞, which indicates the time the task would be launched in an ASAP schedule with unlimited number of processors. It also represents the longest path length from the start of the graph to this task. Similarly, we define the* ***earliest finish time*** *of the task as $f^\infty = s^\infty + t$, indicating the time in which the task could be completed in an ASAP schedule. We further refer to (s^∞, f^∞) as the* ***criticality****, indicating the interval in which the task will be executed in an ASAP schedule. As $\mathcal{C}(\mathcal{I})$ corresponds to the longest path in the graph, or the minimum completion time with unlimited number of processors of an ASAP schedule, we have $\mathcal{C}(\mathcal{I}) = \max_j f_j^\infty$.*

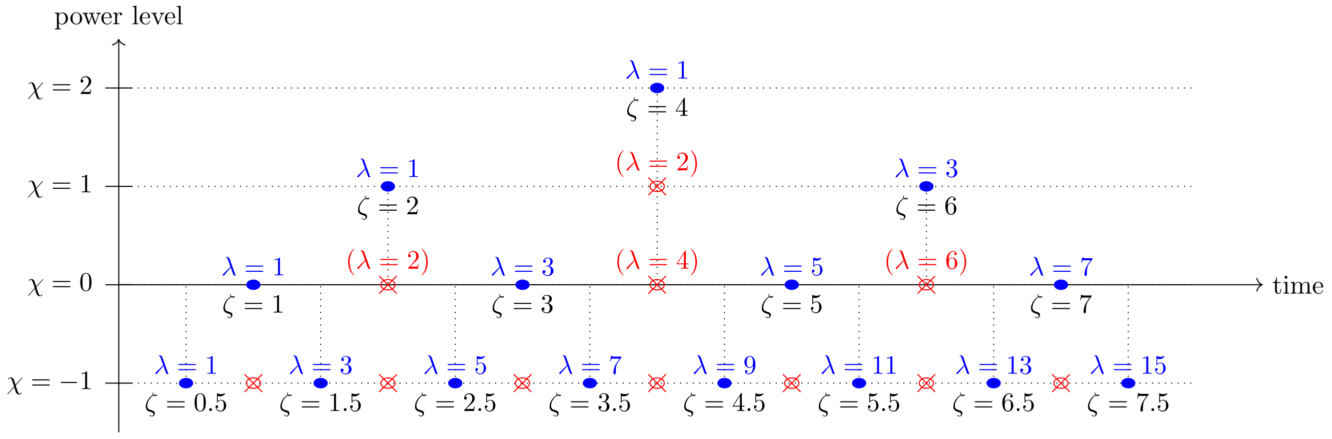


Figure 15: Graphical representation of some possible values of category ζ, power level χ and longitude λ.

Lemma 1. *Given a task $\mathcal{T}$ and its set of predecessors $\mathcal{P}(\mathcal{T})$, we have:*

$$s^\infty = \begin{cases} \max_{\mathcal{T}_j \in \mathcal{P}(\mathcal{T})} f_j^\infty, & \text{if } \mathcal{P}(\mathcal{T}) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

Definition 2. *Given a task $\mathcal{T}$ and its criticality (s^∞, f^∞), we define its **power level** χ as:*

$$\chi = \max\{\chi \in \mathbb{Z} : \exists \lambda \in \mathbb{N}, s^\infty < \lambda 2^\chi < f^\infty\} \tag{12}$$

*where λ is the associated **longitude**. Given the power level and longitude of the task, we further define $\zeta = \lambda 2^\chi$ as the task's **category**. Lemma 2 shows that all these values are unique and well-defined for any given task.*

Figure 15 illustrates some values of λ, χ and ζ. Blue points represent category values with odd λ's, whereas red points represent category values with even λ's. The category of a task corresponds to the highest point within (s^∞, f^∞). We can see that such a point always corresponds to an odd λ (since red points always have a blue point directly above) and is unique since if two consecutive λ's are acceptable, one will be red and therefore have another point directly above. This is formally stated in the next lemma.

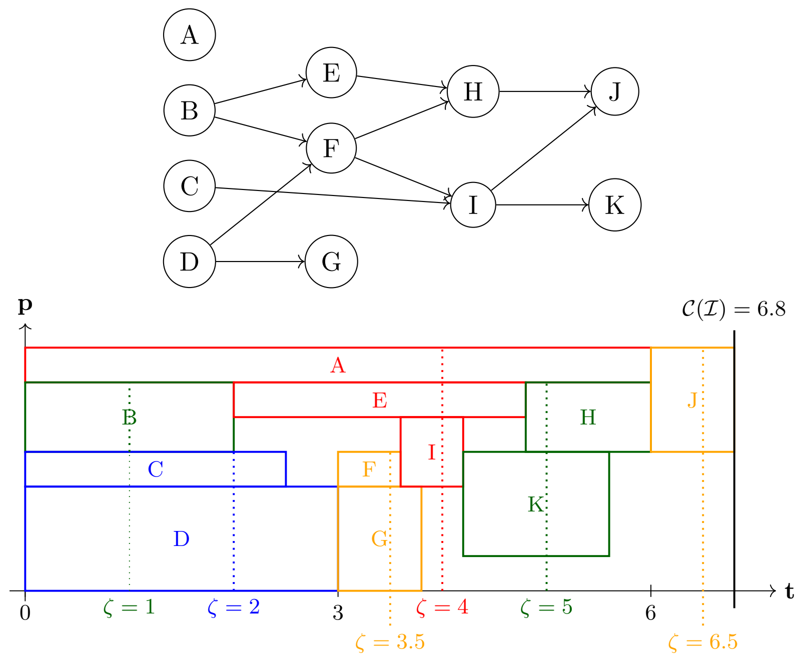
Lemma 2. *Given a task $\mathcal{T}$ of criticality (s^∞, f^∞), its power level χ, longitude λ and category ζ are all unique and well-defined. Furthermore, λ is odd and satisfies:*

$$(\lambda - 1)2^\chi \leq s^\infty < \zeta = \lambda 2^\chi < f^\infty \leq (\lambda + 1)2^\chi$$

Corollary 1. *All tasks in the same category share the same power level and longitude. Thus, in the following, we will refer to the power level χ and longitude λ as two attributes of a category ζ.*

Figure 16 illustrates all these definitions with an example. In the top left is a graph of tasks to be scheduled. For each task, its length t, processor allocation p and various attributes (i.e., criticality $[s^\infty, f^\infty]$, longitude λ, power level χ and category ζ) are given in the table on the right. The bottom left represents the tasks' criticalities, which can be viewed as an ASAP schedule with an unbounded number of processors. Categories are represented as vertical lines, and the color of each line (or category) represents its power level: by increasing order, yellow for $\chi = -1$, green for $\chi = 0$, blue for $\chi = 1$ and red for $\chi = 2$. Correspondingly, the color of each task also indicates its power level, and the category of the task is given by the vertical line of the highest power level that separates the task into two parts.

The algorithm to compute categories is as follows: (if tasks length are unknown categories are estimated).



Task	**t**	**p**	s^∞	f^∞	λ	χ	ζ
A	6	1	0	6	1	2	4
B	2	2	0	2	1	0	1
C	2.5	1	0	2.5	1	1	2
D	3	3	0	3	1	1	2
E	2.8	1	2	4.8	1	2	4
F	0.6	1	3	3.6	7	-1	3.5
G	0.8	3	3	3.8	7	-1	3.5
H	1.2	2	4.8	6	5	0	5
I	0.6	2	3.6	4.2	1	2	4
J	0.8	3	6	6.8	13	-1	6.5
K	1.4	3	4.2	5.6	5	0	5

Figure 16: Top-left: An example task graph consisting of 11 tasks; Right: The various attributes of each task in the task graph; Bottom-left: Graphical representation of the tasks' criticalities and categories, which can also be viewed as an ASAP schedule of the tasks with an unbounded number of processors.

Algorithm 4: Compute_Category($\mathcal{T}$)

```
1. If $\mathcal{P}(\mathcal{T}) = \emptyset$ (where $\mathcal{P}(\mathcal{T})$ contains the list of predecessors of task $\mathcal{T}$):
     • Set $s^\infty \leftarrow 0$.
2. Else:
     • Set $s^\infty \leftarrow \max_{\mathcal{T}_j \in \mathcal{P}(\mathcal{T})} f_j^\infty$ (start time in the ASAP schedule).
3. Compute $f^\infty \leftarrow s^\infty + t$ (finish time (or estimated finish time) in the ASAP schedule).
4. // (If task length are estimated, we will update $s^\infty$ after completion)
5. Find $\chi$ and $\lambda$ from $s^\infty$ and $f^\infty$ based on Definition 2.
6. Compute $\zeta \leftarrow \lambda 2^\chi$.
7. Return $\zeta$.
```

6 TODO

Do experiments, in priority for rigid scheduling. Build task and play with the heuristic in real HPC. I will provide details with interruption criteria later

7 Useful links

- First Rigid paper (task lengths considered known, paper finished) : (link)
- Moldable paper : (link)
- All rigid theory including when task lengths unknown, but it's a draft(link)
- Simulation code for moldable : (link)

References

[1] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, 2010.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'67*, pages 483–485, 1967.

[3] Louis-Claude Canon, Loris Marchal, Bertrand Simon, and Frédéric Vivien. Online scheduling of task graphs on heterogeneous platforms. *IEEE Trans. Parallel Distributed Syst.*, 31(3):721–732, 2020.

[4] Dror G. Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[5] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal on-line scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[6] Jessen T. Havill and Weizhen Mao. Competitive online scheduling of perfectly malleable jobs with setup times. *European Journal of Operational Research*, 187:1126–1142, 2008.

[7] Theodore Johnson, Timothy A. Davis, and Steven M. Hadfield. A concurrent dynamic task graph. *Parallel Computing*, 22(2):327–333, 1996.

[8] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[9] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.