Scheduling Jobs Under a Variable Number of Processors

Joachim Cendrier, Anne Benoit, Frédéric Vivien

Scheduling Jobs Under a Variable Number of Processors

Joachim Cendrier, Anne Benoit, Frédéric Vivien

RESEARCH REPORT
N° 9582
March 2025
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9582--FR+ENG

Scheduling Jobs Under a Variable Number of Processors

Joachim Cendrier[*], Anne Benoit[†], Frédéric Vivien[‡]

Project-Team ROMA

Research Report n° 9582 — March 2025 — 59 pages

Abstract: Even though it is usually assumed that data centers can always operate at maximum capacity, there have been recent scenarios where the amount of electricity that can be used by data centers evolve over time. Hence, the number of available processors is not a constant anymore. In this work, we assume that jobs can be checkpointed before a resource change. Indeed, in the scenarios that we consider, the resource provider warns the user before a change in the number of processors. It is thus possible to anticipate and take checkpoints before the change happens, such that no work is ever lost. The goal is then to maximize the goodput and/or the minimum yield of jobs within the next period (time between two changes in the number of processors). We model the problem and design greedy solutions and sophisticated dynamic programming algorithms. A comprehensive set of simulations building on real-life job sets demonstrates the performance of the proposed algorithms. The best solution maximizes platform utilization while ensuring a high level of fairness.

Key-words: Scheduling, variable capacity, goodput, yield, dynamic programming.

Authors' emails: `{joachim.cendrier,anne.benoit,frederic.vivien}@inria.fr`

[*] ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
[†] ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, and Institut Universitaire de France, and Institute for Data Engineering and Science (IDEaS), Georgia Tech, USA
[‡] ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France

Ordonnancement de tâches sur une machine avec un nombre variable de processeurs

Résumé : Même si l'on part généralement du principe que les centres de calculs peuvent toujours fonctionner à leur capacité maximale, des scénarios récents ont montré que la quantité d'électricité pouvant être utilisée par les centres de calculs évolue au fil du temps. Par conséquent, le nombre de processeurs disponibles n'est plus une constante. Dans ce travail, nous supposons que les tâches peuvent être sauvegardées avant un changement de ressource. En effet, dans les scénarios que nous considérons, le fournisseur de ressources avertit l'utilisateur avant un changement du nombre de processeurs. Il est donc possible d'anticiper et de faire une sauvegarde avant que le changement ne se produise, de sorte qu'aucun travail ne soit jamais perdu. L'objectif est alors de maximiser l'utilisation de la plateforme et/ou l'avancement minimum des tâches au cours de la période suivante (temps entre deux changements du nombre de processeurs). Nous modélisons le problème et concevons des solutions gloutonnes et des algorithmes de programmation dynamique sophistiqués. Un ensemble complet de simulations basées sur des ensembles de tâches réelles démontre la performance des algorithmes proposés. La meilleure solution maximise l'utilisation de la plateforme tout en garantissant un haut niveau d'équité.

Mots-clés : Ordonnancement, capacité variable, rendement, programmation dynamique

Contents

1 Introduction and related work

Most (if not all) work targeting the optimization of data centers assume implicitly that data centers are always provided all the electricity they need to operate at their maximum capability. Hence, at any time, all their components (processors, etc.) are powered on and can operate at full power. There are, however, existing and emerging scenarios in which data centers sometimes only receive a fraction of the electricity they require to work at full power, and this fraction evolves with time. Therefore, in this work, we consider the problem of scheduling jobs in a data center whose processing capability evolves with time.

Some countries suffer from systemic energy under-production, such as the Republic of South Africa, which has experienced an energy crisis since 2007 [19]. Some other countries did experience temporary energy crises: in France, some industrial problems have led to risks of blackouts in 2022-2023 [1], while some blackouts happened in the USA due to meteorological events in 2021-2022 [18]. In such cases, energy distributors usually resort to rolling blackouts, that is, to "intentionally engineered electrical power shutdowns in which electricity delivery is stopped for non-overlapping periods of time over different parts of the distribution region" [18]. Even if a data center can be supplied all the energy it requires, data center managers may not want to operate their centers at full power. This can happen, for instance, if energy prices are high and data centers are run on a tight budget, or if the energy mix is highly carbonated and the data centers are required to achieve a low carbon footprint [17].

From the user point of view, a platform whose energy supply evolves with time is seen as a platform whose processing power evolves with time: its processing resources become volatile [20]. This problem is termed the *variable capacity scheduling* problem [21], and it has raised a lot of interest both from resource providers, with the aim of reducing their costs, and from researchers, with the goal of designing clever solutions to make the best usage of resources, as highlighted in a recent workshop report [4].

Typically, when a resource must be turned off, the jobs that were being executed on that resource have to stop their execution. If the user is not warned beforehand, this results in loosing the work that was on-going on the resource [16]. However, in the scenarios described above, the evolution of the energy supply can be predicted, at least to some extent: institutional rolling blackouts are announced in advance, weather forecasts enable to predict solar and wind energy production [5], and consumption patterns are well studied [6].

In this work, we focus on a setting where, when the energy supply decreases, there is no backup and some processors must be shut down. However, we go beyond the classical approach that terminates jobs, by allowing jobs to be checkpointed before a resource change, hence, avoiding loosing the work already done. Indeed, the checkpoint/restart mechanism has been widely used for platforms confronted to failures [8]: the whole state of a job is periodically checkpointed, so that it is possible to recover from the last checkpoint when the platform is subject to a failure. In the context of varying energy supply, a loss in power is equivalent to a failure of some of the processors.

The main difference with classical studies in resilience is that we can predict, with a good accuracy, the time at which a change in the number of processors will occur. Predictions turn out to be very accurate in the near future [7]. Therefore, we assume that we know the date of the next *event*, i.e., the next time at which we might loose or gain resources, even though we do not have an exact knowledge of the number of processors that will be available after the change. (This conservative assumption enables us to cover a wide range of realistic scenarios.) Hence, it is possible to anticipate and take checkpoints before the change happens. To the best of our knowledge, there is no related work that formalizes this problem and designs algorithms to solve it. Some scheduling algorithms have been designed to account for variations in green energy, such as GreenSlot [10] and GreenPar [11]. However, the setting is very different as they do not need to take checkpoints. Indeed, they assume that there is always some energy available from the grid, even though it might come at a higher cost.

The proposed model considers that when an event occurs at time T_i (with $1 \leq i \leq m$), the user is informed of the time of the next event T_{i+1}, but there is no global knowledge of all future events. An event may correspond to a loss or gain of processors, that may not exceed a total number δ. Furthermore, the exact variation in the number of processors for event T_i is known only at time T_i, together with the date of the next event. With this knowledge, we enforce that we checkpoint jobs so that at least δ processors can be released just before T_{i+1}. Therefore, we ensure that no work will be lost, even if δ processors must be shut off.

In this work, we initially assume that all jobs have an infinite execution time. We make this (potentially surprising) assumption because scheduling strategies that decide which applications are allocated processors and which are (temporarily) stopped influence the completion time of applications. If we were to consider, in the theoretical part of this work, jobs with finite execution times, we would have to study the interplay of these strategies and of the batch scheduling system. To avoid doing this, we initially assume that jobs have infinite execution times. Later, we will show how to extend the proposed algorithms to jobs with finite execution times, and we will show that the conclusions derived for infinite jobs also hold for finite ones.

Our main contributions are the following:

- We formalize the problem with a set of infinite parallel jobs, each of them requiring a fixed number of processors to be executed, and with the ability to take checkpoints and recover after an interruption (Section 2);
- We define two objective functions in this setting. The *goodput* is platform-centric and measures the platform utilization (without accounting for the time spent checkpointing or recovering some jobs). The *minimum yield* is user-centric and aims at ensuring some fairness between jobs, i.e., all jobs should get a chance to be executed approximately the same amount of time (Section 3);
- The core contribution is Section 4, with the design of several sophisticated algorithms using dynamic programming (DP), either to optimize one of the criteria, or to provide a bi-criteria solution; the solutions returned by the algorithms are close to the optimal, and we even prove the optimality for goodput maximization in the time interval before the next event, when all jobs take the same time to checkpoint;
- In Section 5, we extend the model to deal with jobs of finite duration. We then need to consider job arrivals and departures. Therefore, the objective functions are revisited, and we also explain how to adapt the algorithms of Section 4;
- Extensive simulations are performed in Section 6 on jobs extracted from the Parallel Workloads Archive [12], in order to assess the performance of the algorithms. The bi-criteria DP algorithm achieves impressive results, succeeding to reach high values of goodput while ensuring a fair treatment of all jobs through the yield.

Finally, we conclude and discuss the limitations of the hypotheses of this work, as well as future working directions, in Section 7.

2 Model

Application. We consider a set of m parallel (rigid) jobs $J_1, \ldots, J_m$, where job J_j uses exactly p_j processors, for $1 \leq j \leq m$. Each job J_j can be checkpointed in time C_j, and its recovery time is R_j. For consistency, job J_j needs to wait for the recovery time also the first time that it is executed (e.g., because of the time required to load the application code and input data).

Therefore, we do not differentiate jobs that have already been executed and checkpointed from new jobs.

Finally, let us denote $C_{\min} = \min_{1\leq j\leq m} C_j$, $C_{\max} = \max_{1\leq j\leq m} C_j$, $R_{\min} = \min_{1\leq j\leq m} R_j$, and $R_{\max} = \max_{1\leq j\leq m} R_j$. Note that if all jobs share the same checkpointing (resp. recovery) time, this time is denoted by C (resp. R).

Platform. The jobs are executed on a parallel platform with $P_{\max}$ processors, but these processors are not all available at all time. Indeed, the aim of this paper is to consider scenarios where the number of powered processors vary over time. From the user point of view, this is seen as events where the number of available processors changes. The execution starts at time T_1 with a given number of processors, and then events of changes in the number of available processors happen at dates $T_2, T_3, \dots, T_{n+1}$. We call a *section* the time interval defined by two of these consecutive events. There are n such sections, and the i-th section is $[T_i, T_{i+1})$. The number of processors is fixed during a section. Let P_i denote the number of processors available in the i-th section. P_i cannot exceed the maximum number of processors of the platform $P_{\max}$, and cannot go below $P_{\min}$, the number of processors that are always powered up; hence, $P_{\min} \leq P_i \leq P_{\max}$. Furthermore, the change at each event is bounded by δ, i.e., we may not loose or gain more than δ processors at each event: for $1 \leq i \leq n$, we have $|P_{i+1} - P_i| \leq \delta$. We can of course have $P_{\min} = 0$ and $\delta = P_{\max}$.

During section i, we can hence execute a set of jobs of indices $J^{(i)} \subseteq \{1, \dots, m\}$, such that $\sum_{j\in J^{(i)}} p_j \leq P_i$, by allocating exactly p_j processors to each job J_j with $j \in J^{(i)}$. Note that job J_j cannot run if allocated less than p_j processors, hence it is allocated either p_j or zero processors.

The knowledge of the scheduler is limited: we only know the date of the next event T_{i+1}, we only know it at time T_i, but we do not know the exact number of processors that will be available after the change. However, thanks to the bound δ, we can proactively checkpoint enough jobs to ensure that no work will be lost, even in the worst possible scenario where δ processors are shut off at the next section. We impose that enough jobs are checkpointed before each event, so that we are in a setting in which we are *never losing work*.

Example. Figure 1 presents a toy example to illustrate the model. In this example, we have five jobs taking 3, 3, 5, 7 and 8 processors respectively, and we have a total of 19 processors in section $i-1$, then 15 in section i, and finally 21 in section $i+1$. In blue, we have the useful work phases, in red the checkpoint phases, and in green the recovery phases. Finally, in yellow, the processors are switched on but not working (idle phase). At the end of each section, some jobs are checkpointed for a total of at least $\delta = 6$ in the example. At time T_i, four processors are powered off, so jobs J_1 and J_2 are stopped. Since there are two idle processors and no small job able to run on these two processors, we decide to checkpoint J_3 at time T_i, so that job J_4 can run for the rest of section i instead of J_3, using all available processors.

Checkpoints are also taken just before T_{i+1}, but since the number of processors finally increases, new jobs are recovered and started in section $i+1$.

3 Objective functions

We consider two optimization objectives: a platform-centric one, the goodput, and a user-centric one, the yield. In order to formally define these optimization objectives, we start by introducing the notion of effective working time of a job in Section 3.1. We then define both objective functions (Sections 3.2 and 3.3), and derive an upper bound for each of them in Section 3.4. From these bounds, we also define the notions of relative goodput and relative yield.

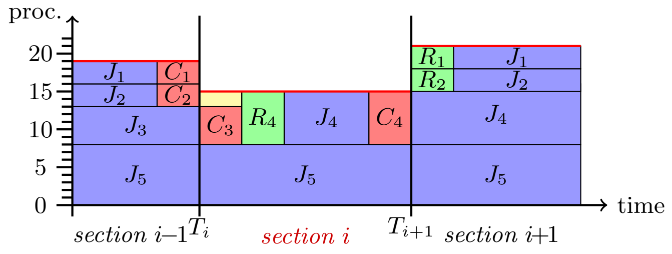


Figure 1: Toy example with $\delta = 6$.

3.1 Preliminaries

We denote by l_j^i the effective working time of job J_j during section i: this is the amount of time during which job J_j was allocated p_j processors but was not performing any checkpointing or recovery in section i. This time is equal to the length of the corresponding blue area on Figure 1. Note that if job J_j is not allocated any processor during section i, then $l_j^i = 0$. We also denote by $l_j(t)$ the amount of time from the beginning until time t during which job J_j was effectively working. Hence, for $t = T_i$, $l_j(T_i) = \sum_{k=1}^{i-1} l_j^k$.

A job is said to be *active* at a time t if it is allocated processors at that time (it is then either executing, checkpointing, or recovering).

3.2 Goodput

A classical platform-centric objectif is to optimize the platform utilization or, conversely, to minimize the platform idleness. However, in our context, even though the platform is used while checkpointing or recovering some jobs, there is no useful work done during checkpoints and recoveries. Hence, rather than considering the platform utilization, we focus on the goodput, which only accounts for the effective work done by jobs [2, 21]. The goodput, given a time interval, is the total effective work done by all jobs during that interval divided by the total processor time that was available during this interval. The effective work done by job J_j during section i is exactly $l_j^i p_j$. Then, the goodput for section i is:

$$Goodput^i = \frac{\sum\limits_{j=1}^{m} p_j l_j^i}{P_i \times (T_{i+1} - T_i)}.$$

Indeed, the total available processor time is equal to P_i times the section duration.

We can then write the expression for the optimization objective, that is, the goodput of the whole execution, from the beginning up to the end of section n:

$$Goodput = \frac{\sum\limits_{j=1}^{m} p_j l_j(T_n)}{\sum\limits_{i=1}^{n} P_i \times (T_{i+1} - T_i)}. \tag{1}$$

3.3 Yield

The different jobs compete for the processors. Optimizing the goodput may lead to job starvation, that is, a job J_j never being allocated p_j processors. Hence, we also want to consider a *fair* optimization objective. A possible candidate would be the *stretch* or *slowdown* [3]. By definition,

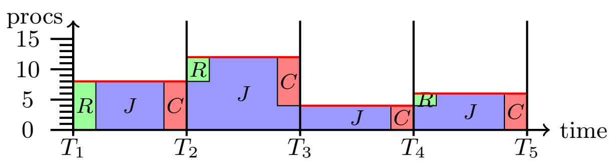


Figure 2: Upper bound on goodput when $\delta = 8$.

the stretch takes values between one and infinity. An infinite stretch happens when a job suffers from starvation, which can be the case in our context with some scheduling algorithms optimizing the goodput. Hence, we rather consider the inverse of the stretch, or yield. The *yield* of a job J_j at time t is the ratio of the total effective working time of J_j up to time t by the total effective working time it would have had if it was the only job using the platform. If there was no competition for resources, the job could have been executed during each section, unless there were not enough processors to execute it. Formally, the yield of job J_j at time t is then:

$$Yield_j(t) = \frac{l_j(t)}{\int_{T_1}^{t} \mathbb{1}_{P(u)\geq p_j} du}, \tag{2}$$

where the denominator computes the amount of time when there are at least p_j processors available (i.e., when job J_j could run if it was alone on the platform).

Then, we can compute the minimum yield among jobs at the end of section i:

$$minYield(T_{i+1}) = \min_{1\leq j\leq m} Yield_j(T_{i+1}).$$

Finally, the goal is to maximize the minimum yield among jobs at the end of section n:

$$minYield = minYield(T_{n+1}) = \min_{1\leq j\leq m} Yield_j(T_{n+1}). \tag{3}$$

Note that a job with a yield of zero, which corresponds to an infinite stretch, is a job that has never been executed during any section.

3.4 Upper bounds

We establish an upper bound for the goodput by considering that the entire workspace can be used for useful work, with the exception of the following areas:

- At the start of each section, the new processors (when there are new ones) cannot be used for useful work before some recovery takes place, which takes at least a time $R_{\min}$. This results in a non-useful work area of size $R_{\min} \times \max\{0,\ P_i - P_{i-1}\}$ for section i.
- At the end of section i, $\min\{\delta,\ P_i - P_{\min}\}$ processors must be checkpointed, and each checkpoint lasts at least $C_{\min}$. This results in a non-useful work area of size $C_{\min} \times \min\{\delta,\ P_i - P_{\min}\}$.

The maximum area of useful work is shown in blue on Figure 2. This gives us an upper bound on the *Goodput*.

For the yield, we consider the same workspace than for the goodput, i.e., the total blue area in Figure 2. We reshape this total area as a single rectangle, whose height is the sum of the number of processors required by all jobs, i.e., $\sum_{i=1}^{m} p_i$ (Figure 3). An upper bound on the minimum

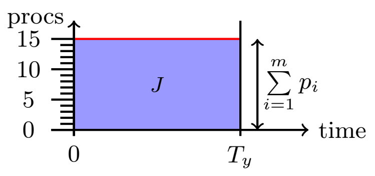


Figure 3: Upper bound on minimum yield.

yield is then obtained by dividing the length of this rectangle, T_y, by the total time available, $T_{n+1} - T_1$, which corresponds to the fairest scenario where each job ends up with the same yield. Indeed, within the rectangle, all jobs can be executed simultaneously and run during a time T_y, hence reaching a yield of $\frac{T_y}{T_{n+1}-T_1}$.

Note that it is unlikely that these bounds are achievable, since jobs need to be checkpointed and restarted to ensure that no processor is left idle, and that processor allocation to job is fair (for yield optimization). However, these bounds will allow us to assess the absolute performance of algorithms in terms of goodput and yield. Indeed, the distance of an algorithm performance to these upper bounds is an upper bound to its distance to the optimal performance. Hence, we define the relative goodput *Relative Goodput* (resp. relative yield *RelativeYield*) of a heuristic as the ratio of the goodput (resp. yield) it achieved, divided by the upper bound on the goodput (resp. yield). The relative goodput (resp. yield) takes values between 0 and 1, and the higher the better.

4 Algorithms

We start by introducing one greedy algorithm for each of the two considered objective functions (Section 4.1). Then, we design dynamic programming algorithms to optimize for a single objective function (Section 4.2) or for a combination of them (Section 4.3).

4.1 Greedy algorithms

4.1.1 Goodput optimization

We describe the greedy algorithm for goodput optimization, Algorithm GreedyGoodput, considering any section $[T_i, T_{i+1}]$. At time T_i, Algorithm GreedyGoodput restarts checkpointed jobs as long as there are idle processors and jobs that can be restarted. The jobs are considered by non-increasing values of the number of requested processors (p_j). Note that some processors may remain idle at the end of this step.

At the end of a section, recall that we impose that no work is ever lost, even in the case of the largest possible decrease in the number of available processors. Hence, we need to make sure that enough jobs are checkpointed. GreedyGoodput only checkpoints jobs at the end of a section, and does it in an as late as possible manner, that is, so that each checkpoint completes exactly at time T_{i+1}. In practice, GreedyGoodput selects jobs *that will not be* checkpointed. Jobs are considered by non-increasing values of the number of requested processors (p_j), and marked to be kept running if this does not lead to using more than $\max\{P_i - \delta, P_{\min}\}$ processors.

The pseudo-code for GreedyGoodput is available as Algorithm 1 in Section A of the Appendix.

4.1.2 Yield optimization

Algorithm GreedyYield is a greedy algorithm that aims at maximizing the minimum yield over all jobs. All jobs that were running during section $i-1$ are checkpointed, and this is done as late as possible so that their checkpoints all complete at time T_i. Then, all jobs are sorted by non-decreasing yield at time T_i (ties are broken by non-increasing values of p_j). Jobs are then restarted in this order while there remain enough available processors.

4.2 Mono-criteria DP algorithms

We design sophisticated dynamic programming (DP) algorithms that aim at optimizing a single criteria (*Goodput* or *minYield*) within a single section, since there is no a priori knowledge beyond the actual section. We start with the goodput in Section 4.2.1, and prove that this DP algorithm is optimal when all checkpoint times are identical in Section 4.2.2. We also propose a simpler and faster version of this DP algorithm in Section 4.2.3. We then focus on the minimum yield in Section 4.2.4.

4.2.1 Dynamic programming for Goodput optimization

We design a dynamic programming algorithm that aims at maximizing the goodput within section i. At time T_i, we know which jobs were running during the previous section, as well as which jobs were checkpointed prior to event T_i. We need to decide which jobs should be checkpointed at the beginning of the section, and then which ones should be recovered and executed for the remaining of the section, so that we can maximize the goodput during section i. The total length of the current section is denoted by $T = T_{i+1} - T_i$.

The section is divided in three phases. The first phase occurs between time T_i and a time $T_i + C_{nf}$ (to be defined) at which all the checkpoints complete. These checkpoints are done as late as possible to be completed at time $T_i + C_{nf}$, hence some work can still be done before the checkpoint by jobs with a time of checkpoint smaller than C_{nf}. Most of the work done during section i is done during the second phase, which lasts until time $T_{i+1} - C_{\max}$. Finally, the third phase is where proactive checkpoints are taken before the next section change (it will become clearer later why we do not need to be as precise for the start time of the third phase as we were for the ending time of the first phase, using C_{nf}). And at the beginning of both first and second phases, if some available processors are idle, we try to allocate jobs to them, and start their recovery as soon as possible. Since we also need to know the status of jobs during section $i-1$, we also consider a phase 0, which corresponds to the third phase of the previous section, section $i-1$.

In each phase, we distinguish the jobs depending on their status (active or idle, checkpointing or not checkpointing, under recovery, etc.), so that we will be able to handle all possible cases and maximize the goodput. Figure 4 illustrates the different job statuses and the possible transitions between statuses.

In phase 0 (resp. phase 3), a job is either active (it is allocated p_j processors), or idle (`Idle`). In this first case, it might be checkpointed (`Ckpt`) or not (`Run`); if it is checkpointed, this is done as late as possible so that its checkpoint ends exactly at time T_i (resp. T_{i+1}).

The situation is a bit more complicated in phase 1. First, note that some jobs might also be checkpointed during this phase, as explained before and, hence, we still have the statuses `Idle`, `Ckpt`, and `Run` for jobs, similarly to phase 0. It is also possible to resume some jobs that were previously idle, by paying a recovery cost, and then either to checkpoint them at the end of phase 1 (`Rec+Ckpt`) or not (`Rec`). Here again, checkpoints are taken at the end of the phase (as late as possible), while recoveries are done at the beginning of the phase (as soon as possible).

$$\begin{tikzcd}
 & \texttt{Run}^{(1)} \arrow[dr] & & \\
\texttt{Run}^{(0)} \arrow[ur] \arrow[r] & \texttt{Ckpt}^{(1)} \arrow[ddr] & \texttt{Run}^{(2)} \arrow[r] \arrow[dr] & \texttt{Run}^{(3)} \\
\texttt{Ckpt}^{(0)} \arrow[uur] \arrow[ur] \arrow[ddr] & \texttt{Rec}^{(1)} \arrow[ur] & \texttt{Rec}^{(2)} \arrow[ur] \arrow[r] & \texttt{Ckpt}^{(3)} \\
\texttt{Idle}^{(0)} \arrow[ur] \arrow[r] \arrow[dr] & \texttt{Rec+Ckpt}^{(1)} \arrow[r] & \texttt{Idle}^{(2)} \arrow[r] & \texttt{Idle}^{(3)} \\
 & \texttt{Idle}^{(1)} \arrow[uur] \arrow[ur] & &
\end{tikzcd}$$

Figure 4: The different job statuses in the different phases and their transitions.

Finally, in phase 2, no checkpoint is taken; a job may be running during the whole phase (`Run`), or after a recovery (`Rec`), or not be executed during this phase (`Idle`).

At the end of a phase, depending on its status, a job may gain a new status for the next phase. This leads to the 15 different cases listed below and to the 15 different paths from a source to a sink in the graph of Figure 4. The first three cases concern jobs that were executed in section $i-1$ and not checkpointed (jobs in $\texttt{Run}^{(0)}$). Hence, they must still be active in phase 1. Next, we have six cases for jobs that were checkpointed at the end of phase 0 (jobs in $\texttt{Ckpt}^{(0)}$), which can either pursue their execution or be stopped. Finally, there are six cases for jobs that were idle in section $i-1$ (jobs in $\texttt{Idle}^{(0)}$).

1. $J_j \in \texttt{Run}^{(0)} \cup \texttt{Run}^{(1)} \cup \texttt{Run}^{(2)} \cup \texttt{Run}^{(3)}$: The job is constantly executed and never checkpointed;

2. $J_j \in \texttt{Run}^{(0)} \cup \texttt{Run}^{(1)} \cup \texttt{Run}^{(2)} \cup \texttt{Ckpt}^{(3)}$: The job is constantly executed, and checkpointed at the end of phase 3;

3. $J_j \in \texttt{Run}^{(0)} \cup \texttt{Ckpt}^{(1)} \cup \texttt{Idle}^{(2)} \cup \texttt{Idle}^{(3)}$: The job, that was not checkpointed at phase 0, is checkpointed during phase 1 to become idle from phase 2 onwards;

4. $J_j \in \texttt{Ckpt}^{(0)} \cup \texttt{Run}^{(1)} \cup \texttt{Run}^{(2)} \cup \texttt{Run}^{(3)}$: The checkpointed job is executed in all phases, with no checkpoint at the end;

5. $J_j \in \texttt{Ckpt}^{(0)} \cup \texttt{Run}^{(1)} \cup \texttt{Run}^{(2)} \cup \texttt{Ckpt}^{(3)}$: Similar to case (4), but with a checkpoint taken at the end of phase 3;

6. $J_j \in \texttt{Ckpt}^{(0)} \cup \texttt{Ckpt}^{(1)} \cup \texttt{Idle}^{(2)} \cup \texttt{Idle}^{(3)}$: The job does a little work during phase 1 before taking another checkpoint, and finally becomes idle from phase 2 onwards;

7. $J_j \in \texttt{Ckpt}^{(0)} \cup \texttt{Idle}^{(1)} \cup \texttt{Rec}^{(2)} \cup \texttt{Run}^{(3)}$: The job becomes idle in phase 1, but is recovered in phase 2 and then is executed until the end, with no checkpoint taken;

8. $J_j \in \texttt{Ckpt}^{(0)} \cup \texttt{Idle}^{(1)} \cup \texttt{Rec}^{(2)} \cup \texttt{Ckpt}^{(3)}$: Similar to case (7), but with a checkpoint taken at the end of phase 3;

9. $J_j \in \texttt{Ckpt}^{(0)} \cup \texttt{Idle}^{(1)} \cup \texttt{Idle}^{(2)} \cup \texttt{Idle}^{(3)}$: The job becomes idle from phase 1 onwards;

10. $J_j \in \texttt{Idle}^{(0)} \cup \texttt{Rec}^{(1)} \cup \texttt{Run}^{(2)} \cup \texttt{Run}^{(3)}$: The job that was idle during phase 0 is recovered during phase 1 and then it is executed; no checkpoint is taken at the end;

11. $J_j \in \texttt{Idle}^{(0)} \cup \texttt{Rec}^{(1)} \cup \texttt{Run}^{(2)} \cup \texttt{Ckpt}^{(3)}$: Similar to case (10), but with a checkpoint at the end of phase 3;

12. $J_j \in \texttt{Idle}^{(0)} \cup \texttt{Rec+Ckpt}^{(1)} \cup \texttt{Idle}^{(2)} \cup \texttt{Idle}^{(3)}$: The job does a little work, but only during phase 1 (after a recovery and before a checkpoint), and then becomes idle again;

13. $J_j \in \texttt{Idle}^{(0)} \cup \texttt{Idle}^{(1)} \cup \texttt{Rec}^{(2)} \cup \texttt{Run}^{(3)}$: The job becomes active only during phase 2 (after a recovery); no checkpoint is taken at the end;

14. $J_j \in \texttt{Idle}^{(0)} \cup \texttt{Idle}^{(1)} \cup \texttt{Rec}^{(2)} \cup \texttt{Ckpt}^{(3)}$: Similar to case (13), but with a checkpoint at the end of phase 3;

15. $J_j \in \texttt{Idle}^{(0)} \cup \texttt{Idle}^{(1)} \cup \texttt{Idle}^{(2)} \cup \texttt{Idle}^{(3)}$: The job remains idle during the whole section.

Building upon these job statuses, the dynamic programming algorithm maximizes the amount of useful work, called *gain*, that is done during the section. We consider all possible decisions for job J_j, building upon decisions that can be taken for jobs $J_1, ..., J_{j-1}$ and, hence, obtaining the gain for scheduling jobs $J_1, \dots, J_j$. This gain function depends on the number of processors available at each phase, given the decisions already taken for jobs $J_{j+1}, \dots, J_m$:

- Π_1 is the number of processors available for phase 1. At time T_i, some of the P_i available processors are used by jobs that were active and running during phase 0. Hence, the initial value of Π_1 is: $\Pi_1 = P_i - \sum_{J_j \in \texttt{Run}^{(0)}} p_j$;

- Π_2 is the number of processors available for phase 2, initially $\Pi_2 = P_i$;

- Π_3 is the number of processors that can still be used to run jobs at the end of the section. Initially, $\Pi_3 = \max(P_i - \delta, P_{\min})$ since we need that at least $\min\{\delta, P_i - P_{\min}\}$ processors are not used right before T_{i+1} to ensure that no work will be lost at time T_{i+1}.

We add a fourth parameter, C_{nf}, which determines the length of phase 1. There is a trade-off between choosing a small C_{nf} that leaves little flexibility for which jobs can be checkpointed, but allows for a longer phase 2 with useful work done, versus having a large C_{nf} that gives more flexibility but a smaller phase 2.

We now write the gain function, $G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf})$, which decides how to handle jobs $J_1, \dots, J_j$ during section i. We introduce three sub-functions (with same parameters), which correspond to the possible statuses of J_j during phase 0: $G_j^{\texttt{Run}}$, $G_j^{\texttt{Ckpt}}$, and $G_j^{\texttt{Idle}}$. The function is expressed as:

$$G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}) = \begin{cases} G_j^{\texttt{Run}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{if } J_j \in \texttt{Run}^{(0)} \\ G_j^{\texttt{Ckpt}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{if } J_j \in \texttt{Ckpt}^{(0)} \\ G_j^{\texttt{Idle}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{if } J_j \in \texttt{Idle}^{(0)} \end{cases}$$

Hence, if job J_j was active in phase 0 and not checkpointed, $G_j^{\texttt{Run}}$ decides what is its best scenario. There are three cases for $G_j^{\texttt{Run}}$, as described above, which leads to the formula below:

$$G_j^{\texttt{Run}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) =$$
$$\max \begin{cases} p_j T + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3 - p_j, C_{nf}) & (1) \\ p_j(T - C_j) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3, C_{nf}) & (2) \\ \begin{cases} -\infty & \text{if } C_j > C_{nf} \\ p_j(C_{nf} - C_j) + G_{j-1}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{otherwise} \end{cases} & (3) \end{cases}$$

Case (1) corresponds to a job running during the whole section; hence, a gain of $p_j T$ (the job is executed on p_j processors for a duration T). We then add the gain that can be achieved by jobs $J_1, \dots, J_{j-1}$, with a recursive call to the gain function. Note that the values of Π_2 and Π_3 decrease by p_j, since the job uses p_j processors during phase 2 and is not checkpointed in phase 3. However, since the job was not checkpointed during phase 0, the value of Π_1 remains unchanged.

Case (2) is similar, except that the job is checkpointed at the end. Hence, the value of Π_3 remains unchanged in the recursive call, and the gain is slightly lower because the job only works during a total duration of $T - C_j$.

Finally, in case (3), the job is only executed in phase 1 for a duration $C_{nf} - C_j$, before becoming idle. This is possible only if there is enough time to perform the checkpoint, i.e., if $C_j \leq C_{nf}$.

We now explicit the expression for $G_j^{\texttt{Ckpt}}$, which considers the six cases when J_j has been checkpointed during phase 0:

$$G_j^{\texttt{Ckpt}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) =$$

$$\max \begin{cases} p_j T + G_{j-1}(\Pi_1 - p_j, \Pi_2 - p_j, \Pi_3 - p_j, C_{nf}) & (4) \\ p_j(T - C_j) + G_{j-1}(\Pi_1 - p_j, \Pi_2 - p_j, \Pi_3, C_{nf}) & (5) \\ \begin{cases} -\infty & \text{if } C_j > C_{nf} \\ p_j(C_{nf} - C_j) + G_{j-1}(\Pi_1 - p_j, \Pi_2, \Pi_3, C_{nf}) & \text{otherwise} \end{cases} & (6) \\ p_j(T - C_{nf} - R_j) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3 - p_j, C_{nf}) & (7) \\ p_j(T - C_{nf} - R_j - C_j) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3, C_{nf}) & (8) \\ 0 + G_{j-1}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & (9) \end{cases}$$

Cases (4), (5) and (6) are similar to cases (1), (2) and (3) respectively, except that the job was checkpointed during phase 0 and, because it is running during phase 1, it uses p_j processors during that phase. Therefore, Π_1 must be decreased by p_j. For cases (7) and (8), the job is idle during phase 1; hence, we do not gain any goodput for a duration of C_{nf}. Then, a recovery must be paid to restart the job in phase 2, and during phase 3, the job can be checkpointed or not. Finally, there is no gain in case (9) since the job is idle throughout section i.

Next, we consider the last six cases, which correspond to J_j being idle during phase 0. The expression for $G_j^{\texttt{Idle}}$ is:

$$G_j^{\texttt{Idle}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) =$$

$$\max \begin{cases} p_j(T - R_j) + G_{j-1}(\Pi_1 - p_j, \Pi_2 - p_j, \Pi_3 - p_j, C_{nf}) & (10) \\ p_j(T - R_j - C_j) + G_{j-1}(\Pi_1 - p_j, \Pi_2 - p_j, \Pi_3, C_{nf}) & (11) \\ \begin{cases} -\infty & \text{if } C_j + R_j > C_{nf} \\ p_j(C_{nf} - C_j - R_j) + G_{j-1}(\Pi_1 - p_j, \Pi_2, \Pi_3, C_{nf}) & \text{otherwise} \end{cases} & (12) \\ p_j(T - C_{nf} - R_j) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3 - p_j, C_{nf}) & (13) \\ p_j(T - C_{nf} - R_j - C_j) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3, C_{nf}) & (14) \\ 0 + G_{j-1}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & (15) \end{cases}$$

Here, cases (10), (11) and (12) are similar to cases (4), (5) and (6), except that a recovery must be paid (and, hence, in case (12), we must have time to perform a recovery and then a checkpoint during phase 1). Cases (13), (14) and (15) are similar to cases (7), (8) and (9).

The dynamic programming algorithm is initialized as follows:

- $G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}) = -\infty$ if $\Pi_1 < 0$, if $\Pi_2 < 0$, or if $\Pi_3 < 0$; this corresponds to cases where we do not have enough processors in one of the phases;
- $G_0(\Pi_1, \Pi_2, \Pi_3, C_{nf}) = 0$ if $\Pi_1 \geq 0$, $\Pi_2 \geq 0$, and $\Pi_3 \geq 0$; this corresponds to cases where we have correctly scheduled all jobs without running out of processors; hence, a gain of 0 (with 0 jobs to be scheduled).

Finally, in order to maximize the goodput within the section, we try all possible values of C_{nf}, knowing that C_{nf} must be equal to the checkpointing time of a job running but not checkpointed during phase 0.

$$\max_{C_{nf} \in \{C_k | J_k \in \texttt{Run}^{(0)}\}} \left\{ G_m(P_i - \sum_{J_j \in \texttt{Run}^{(0)}} p_j, P_i, \max(P_i - \delta, P_{\min}), C_{nf}) \right\}$$

The time complexity of this dynamic programming algorithm is in $O(m^2 P_{\max}^3)$. Indeed, there are m jobs, up to m different values of C_{nf}, and we need to explore all values for the number of processors Π_1, Π_2, Π_3, which can each be as large as P_i and, hence, are bounded by $P_{\max}$. The spatial complexity is in $O(mP_{\max}^3)$, as it is not necessary to maintain the whole gain table when changing the value of C_{nf}. This dynamic programming algorithm explores all possible cases; hence, it has a large complexity. We denote it as DPGoodputSlow-Best-C_{nf}. In the next section, we prove its optimality when checkpoint times are all equal, and introduce a faster version for that particular case.

It may happen that there exist several different optimal solutions. For such a situation, we have added two rules to attempt to choose a more favorable situation for the following sections. When the same number of processors is used, it seems to us that the more jobs are running (and therefore the smaller the jobs), the less expensive (in *Goodput*) it is to reorganize the processor usage for the following section (smaller jobs means more flexibility on average). We therefore chose, among the optimal solutions, the one with the highest number of active jobs during phase 2 and, if there are still equalities, the one with the highest number of checkpointed jobs at the end of phase 3.

4.2.2 Goodput optimization with identical checkpoint times

In this section, we assume that all jobs have the same checkpointing time ($C_j = C$ for $1 \leq j \leq m$). In this case, we can prove that the shape of an optimal solution actually follows the shape of solutions returned by the DP algorithm: checkpoints are either taken at time T_i, hence in phase 1, or at time $T_{i+1} - C$, hence in phase 3.

Theorem 1. *For any i, with $1 \leq i \leq n$, let $\mathcal{O}^i$ be a schedule maximizing the goodput $Goodput^i$ during section i, $[T_i, T_{i+1}]$. If all checkpoints last the same duration C, then under schedule $\mathcal{O}^i$, checkpoints only start at time T_i (to complete at time $T_i + C$) or at time $T_{i+1} - C$ (to complete at time T_{i+1}).*

Due to lack of space, the proof is only available in the Appendix (Section B). The core idea is that any solution not following this form can be transformed into a better one. An immediate corollary is the optimality of DPGoodputSlow-Best-C_{nf}; indeed, DPGoodputSlow-Best-C_{nf} builds a schedule that follows the shape described in Theorem 1, while choosing the best from all possibilities for job statuses in the different phases.

Corollary 1. *If all checkpoints last the same duration C, then* DPGoodputSlow-Best-C_{nf} *builds an optimal schedule for the maximization of the goodput of the current section.*

Since checkpointing times are all equal (to C), DPGoodputSlow-Best-C_{nf} can be simplified. First, there is no need to consider C_{nf}, as phase 1 necessarily lasts for a time C, as well as phase 3. Also, cases (6) and (12) never happen since any checkpoint lasts for the whole duration of phase 1.

When checkpoints have different durations, there exist scenarios where checkpoints (respectively recoveries) in the first phase should not be done at the end (resp. beginning) of the phase. For instance, we could allow a job to start slightly before the beginning of phase 2 (see Section C of the Appendix for an example). Therefore, DPGoodputSlow-Best-C_{nf} is no longer optimal when checkpoints have different durations. On the one hand, an optimal algorithm would have a prohibitive complexity (we would need to consider all possible decisions at all time instants before $C_{\max}$). On the other hand, the minimal solution produced by DPGoodputSlow-Best-C_{nf} would have the same shape as the optimal solution for phases 2 and 3, and might only differ in phase 1. Because phase 1 is never longer than $C_{\max}$, if the duration of the section is large with respect to $C_{\max}$, DPGoodputSlow-Best-C_{nf} should achieve near optimal performance.

4.2.3 Faster dynamic programming algorithm for goodput optimization

We propose in this section a new dynamic programming algorithm with smaller time and space complexity than DPGoodputSlow-Best-C_{nf}. A first simplification consists in only computing the solution for $C_{nf} = C_{\max}$, assuming that the improvement obtained by using a smaller value of C_{nf} is not cost-effective in terms of additional computational time. This simplification reduces the time complexity by a factor of m and reduces the achieved goodput by at most $(C_{\max} - C_{\min})P_{\max}$. We call this variant DPGoodputSlow.

In the following, we propose to further accelerate this dynamic programming algorithm with some simplifications. The idea is to separate the choices made for the different phases and split the algorithm into two disjoint dynamic programming algorithms. The first algorithm, DP1, only considers phases 1 and 2 (and thus ignores phase 3), using the same job statuses as in Figure 4. It achieves a gain denoted by G'. The second algorithm, DP2, decides which jobs to checkpoint during phase 3; its gain function is denoted H.

Since DP1 only considers phases 1 and 2, there remain ten different cases, and we only need to keep parameters Π_1 and Π_2. The reasoning is then similar to that for DPGoodputSlow-Best-C_{nf} G. All cases are detailed in Section D.2 of the Appendix. The time complexity drops from $O(m^2 P_{\max}^3)$ to $O(mP_{\max}^2)$. The space complexity also becomes $O(mP_{\max}^2)$. The gain of DP1 during phase 2 is at least as large as that of DPGoodputSlow-Best-C_{nf} for that phase (any solution of the later algorithm is a solution of the former).

DP2 will be called after the completion of DP1, that is, once decisions have been taken for jobs in phases 1 and 2. Then, in order to maximize G', there only remains to decide which jobs to checkpoint during phase 3. Only jobs active during phase 2 may be checkpointed; hence, only jobs in the set $Ja = \texttt{Run}^{(2)} \cup \texttt{Rec}^{(2)}$. Once again, we use parameter Π_3 to count the number of processors used by non checkpointed jobs. There are only two cases to consider: a job is either checkpointed or it is not. The expression then becomes:

$$H_j(\Pi_3) = \begin{cases} H_{j-1}(\Pi_3) & \text{if } J_j \notin Ja \\ \max \begin{cases} p_j C_{\max} + H_{j-1}(\Pi_3 - p_j) \\ p_j(C_{\max} - C_j) + H_{j-1}(\Pi_3) \end{cases} & \text{otherwise} \end{cases}$$

Indeed, if job $J_j \in Ja$ is not checkpointed, Π_3 decreases by p_j, and the job works for the whole duration of phase 3, i.e., $p_j C_{\max}$. Hence, there is an increase of gain of $p_j C_{\max}$. However, if the job is checkpointed, Π_3 remains the same and the gain is only $p_j(C_{\max} - C_j)$, since the job must be checkpointed. We initialize DP2 with:

- $H_j(\Pi_3) = -\infty$ if $\Pi_3 < 0$ (not enough processors available); and
- $H_0(\Pi_3) = 0$ if $\Pi_3 \geq 0$ (successful decisions taken for all jobs).

Finally, the objective is to maximize the gain $H_m(\max(P_i - \delta, P_{\min}))$. The time and space complexity of DP2 is only $O(mP_{\max})$. The loss in gain during phase 3 with respect to DPGoodputSlow-Best-C_{nf} is at most $C_{\max}P_{\max}$ (the maximum gain of DPGoodputSlow-Best-C_{nf} during that phase).

By running successively DP1 and then DP2, we obtain the DPGoodput and DPGoodput-Best-C_{nf} algorithms depending on whether we only compute the solution for $C_{nf} = C_{\max}$ or not. These algorithms have a time and space complexity $O(mP_{\max}^2)$ and $O(m^2 P_{\max}^2)$ respectively. The difference of the gain achieved by the two algorithms is at most $(2C_{\max} - C_{\min})P_{\max}$.

4.2.4 Dynamic programming for Yield optimization

We now design a dynamic programming algorithm whose goal is to maximize the minimum yield $\mathit{minYield}_i$ achieved at the end of the current section, section i. Although our main goal is to maximize the minimum yield at the end of the execution ($\mathit{minYield}$, cf. Eq. (2)), because we have no global knowledge of events, we design an algorithm that works section by section.

To ensure that each job makes some progress during the current section, rather than having some jobs remaining idle for the whole section, it might be worth to checkpoint jobs in the "middle" of the section. However, this would induce many checkpoints and recoveries during each section and, hence, loosing useful working time. Since the ultimate goal is to maximize *minYield* at the end of the execution, we decide to restrict to solutions where checkpoints are taken only at the beginning and at the end of sections (similarly to the dynamic programming algorithms for goodput maximization), but where jobs with a small yield are executed with a higher priority.

We follow an approach very similar to that of DPGoodput, and design two dynamic programming algorithms, one for phases 1 and 2, and the one for phase 3. The resulting algorithm, DPYield, is very similar to DPGoodput and can be found in Section D.3 of the Appendix. Its time and space complexity are in $O(mP_{\max}^2)$.

4.3 Bi-criteria dynamic programming algorithms

We have designed three bi-criteria algorithms, building on the mono-criterion dynamic programming algorithms. The first two algorithms enforce a target on one objective while optimizing the other criteria. Finally, we modify DPGoodput-Best-C_{nf} to combine the yield and goodput within a single gain function, hence, obtaining algorithm DPBiC–Best-C_{nf}.

4.3.1 TargetYield

This algorithm checks, at each section, whether the target bound on the *minYield* is achieved by all jobs. At the end of the first section, for instance, the bound is probably not achieved since some jobs had no opportunity to be executed yet. Then, when reaching section i, either the yield of all jobs is above the target value *minYield* at time T_i and we run DPGoodput during section i to improve the goodput as much as possible, or we run DPYield.

4.3.2 TargetGoodput

For the second algorithm, we rather consider a target bound on the goodput, and our goal is to maximize the minimum yield while keeping the goodput above the bound. We then design a new dynamic programming algorithm. This algorithm is similar to DPYield, but it introduces a new variable, *una* (standing for *UNused Area*), which accounts for the amount of potential work that is unused because of checkpoints, recoveries, or idle processors. Given the target goodput and the execution so far, one can derive a threshold value for *una*: as long as *una* is below this threshold, we are assured that the target bound on the goodput will be reached. The details of this algorithm are available in Section D.4 of the Appendix. The time and space complexity of this algorithm is $O(mP_{\max}^3 T)$ (the increase of complexity is due to the *una* parameter).

4.3.3 Bi-criteria dynamic programming

In this algorithm, the goal is to maximize the two objective functions at the same time. To do this, we slightly modify DPGoodput-Best-C_{nf}. We want to take into account the current yield of each job in the gain function, so that a job with a high yield will see its gain reduced relatively to a job with a low yield, which will see its gain increased. To do this, we use the formula $2 - \mathit{Yield}_j(T_i)$, which takes values in the interval $[1, 2]$. Then, to increase or decrease the weight of the yield in the gain function, we add a multiplicative parameter Y.

DP1: G'. Each equation for the gain function G' of DPGoodput-Best-C_{nf} of the shape:

$$G_j^{'X}(\Pi_1, \Pi_2, C_{nf}) = p_j Ewt_j(X, \alpha, \beta, C_{nf}) + G'_{j-1}(\Pi_1 - \alpha p_j, \Pi_2 - \beta p_j, C_{nf})$$

is replaced by the equation:

$$G_j^{'X}(\Pi_1, \Pi_2, C_{nf}) = p_j Ewt_j(X, \alpha, \beta, C_{nf})(2 - \mathit{Yield}_j(T_i))^Y + G'_{j-1}(\Pi_1 - \alpha p_j, \Pi_2 - \beta p_j, C_{nf}),$$

where $X \in \{\texttt{Run}, \texttt{Ckpt}, \texttt{Idle}\}$, $\alpha, \beta \in \{0, 1\}$ and Ewt_j is the potential Effective Working Time of the job J_j in the condition determined by X, α, β and C_{nf} for this section.

Hence, we are multiplying the goodput contribution of the job in each scenario, denoted here by Ewt_j, by the value $(2 - \mathit{Yield}_j(T_i))^Y$, which is directly impacted by the yield of the job. Finally, we are computing the first part of DPBiC–Best-C_{nf}, with the same initialization and parameters as DPGoodput-Best-C_{nf}.

DP2: H. For this part, we make the same modification, with the following factor: $(2 - \mathit{Yield}_j(T_{i+1} - C_{\max}))^Y$, since the algorithm works on phase 3 and accounts for the yield achieved at the end of phase 2, at time $T_{i+1} - C_{\max}$.

Since the whole algorithm is similar to DPGoodput-Best-C_{nf}, we obtain the same time and space complexity as the DPGoodput-Best-C_{nf} algorithm, i.e., the time and space complexity are in $O(m^2 P_{\max}^2)$. We also introduce a variant DPBiC of DPBiC–Best-C_{nf}, that we make explicit in the Appendix in Section D.5.

5 Adaptation to finite duration jobs

In this section, we adapt the algorithms to work in the more realistic framework of jobs with finite durations. The model described in Section 2 must be slightly refined. Each job J_j now has a release date r_j, which is the time at which the job is submitted to the system and, hence, is the earliest time at which the job can start to be executed. J_j now also has an execution length L_j, which is the time it takes to complete its execution when it is always allocated p_j processors. Finally, we denote by e_j the completion time of J_j, corresponding to the time at which its effective working time becomes equal to its length: $l_j(e_j) = L_j$. Recall that we have assumed that a job pays a recovery the very first time it is executed (to simulate the job startup cost). We similarly assume that a job has to take a final checkpoint right after its completion (to simulate sending back its results). We say that job J_j is *alive* between its release date and its completion time; only alive jobs can be allocated processors.

In Section 5.1, we explain how to derive a new upper bound on the maximum minimum yield in the presence of release dates. In Section 5.2, we present an adaptation of the algorithms for jobs of finite durations.

5.1 New upper bound on the maximum minimum yield

As defined in Section 3.3, the yield of a job J_j corresponds to the time the job has been running in the system, divided by the time it would have run if it were alone in the system. Therefore, we need to rewrite Equation (2) in order to take into account the release date r_j of J_j, and its completion time e_j if it has already been reached. If J_j was not completed by the considered time t, then $e_j = +\infty$. Thus, the yield of job J_j at time t is now:

$$Yield^j(t) = \frac{l^j(t)}{\int_{r_j}^{\min\{t,e_j\}} \mathbb{1}_{P(u)\geq p_j} du}.$$

The set of alive jobs changes over time because new jobs arrive and some jobs complete. This forbids us to reuse the technique of Section 3.3 to derive a lower bound on the *minYield*. We proceed indirectly with a linear program (in rational), which checks whether a given target yield Y can be achieved by all jobs, and with a binary search to identify the maximum achievable such target yield.

Let Y be a given target yield. If the completion time of job J_j is such that $e_j \leq r_j + \frac{L_j}{Y}$, then the yield of J_j is at least equal to Y. Therefore, we define for each J_j a deadline $d_j = \min\left\{r_j + \frac{L_j}{Y}, T_{n+1}\right\}$. If $d_j = T_{n+1}$, the job reaches the end of execution and we decrease its length to $L_j = (T_{n+1} - r_j)Y$ to account for the fact that the work will stop at time T_{n+1}. Hence, if each job can be completed by its own deadline, the minimum yield is at least equal to Y. We now write a linear program (in rational) to check whether each job can be completed by its deadline. Using the set of job release dates and deadlines, we subdivide the sections in a set of subsections $\mathcal{S}$ such that release dates and deadlines only happen at subsection boundaries. For any subsection $\phi \in \mathcal{S}$, we denote by $S(\phi)$ the section that includes ϕ ($\phi \subset [T_{S(\phi)}, T_{S(\phi)+1}]$), by $L(\phi)$ its duration, by $M(\phi)$ its middle point, and by $x_{j,\phi}$ the working time of job J_j during ϕ. The middle point is used to characterize when a job cannot be executed during a whole subsection ϕ, in which case the work must be set to $x_{j,\phi} = 0$: $(r_j \geq M(\phi))$ *or* $(d_j \leq M(\phi)) \Rightarrow x_{j,\phi} = 0$.

The following linear program, together with the above constraint, enables us to check whether the target yield Y is achievable:

$$\begin{cases} \forall \phi \in \mathcal{S},\ \sum_{j=1}^{m} p_j x_{j,\phi} \leq L(\phi) P_{S(\phi)} & (4) \\ \forall 1 \leq j \leq m, \sum_{\phi \in \mathcal{S}} x_{j,\phi} = L_j & (5) \\ \forall 1 \leq j \leq m, \forall \phi \in \mathcal{S},\ 0 \leq x_{j,\phi} \leq L(\phi) & (6) \end{cases}$$

Equation (4) states that the total work done by the jobs in a subsection cannot exceed the work capacity of the subsection. Equation (5) states that each job must be completed. Equation (6) states that the working time of a job during a subsection cannot exceed the duration of that subsection.

Then, either this linear program admits a solution and the yield Y is achievable, or no solution exists and the yield Y is not achievable.

The bound obtained this way will not be achievable in practice for several reasons. First, the linear program approach considers neither checkpoints nor recoveries. Furthermore, in practice, a new job may be released in the middle of a section, and to guarantee the minimum yield in such a case, we would have to checkpoint active jobs to be able to start right away the new released jobs. When adapting previous algorithms for finite duration jobs, we are still not allowing checkpoints in the middle of a section, in order to avoid paying excessive checkpointing and recovery costs.

5.2 Processor reallocation at completion time

When a job J_j completes its execution during a section, the processors it was using become unallocated. To avoid wasting computational capabilities, we reallocate them immediately. If the algorithm considered is a greedy algorithm, we greedily reallocate these processors (potentially along with some other idle processors) to idle jobs. If the algorithm considered is a dynamic programming algorithm, we do not run the dynamic programming algorithm, which could lead to additional checkpoints and recoveries and, hence, negatively impact the goodput. Instead, we allocate the idle processors using an exact subset-sum algorithm [9], which has a pseudo-polynomial cost, to maximize the number of processors reallocated. Finally, if J_j was supposed to be checkpointed at the end of the section, the jobs that are now running on its original processors are checkpointed at the end of the section.

6 Simulations

We begin this section with a few remarks on the metrics used, and we present the workloads and the simulation parameters in Section 6.1. Then, in Section 6.2, we present simulation results for the case with infinite jobs, before moving to the case of finite duration jobs in Section 6.3. For reproducibility purpose, all the equations, the codes and the data used in the simulations are available in the Appendix and at `https://doi.org/10.5281/zenodo.14917803`.

6.1 Methodology

For the simulations, algorithms are run on a large number of sections. We report on the goodput and the yield achieved at the end of the last section. We have established in Section 3 an upper bound for each of the two objective functions, and introduced *Relative Goodput* (resp. *Relative-Yield*), which is the ratio of the achieved absolute *Goodput* (resp. *Yield*) divided by the upper bound of Section 3. The *Relative Goodput* (resp. *Relative Yield*) is always a value between 0 and 1, the higher the better. This relative value is an indicator of the absolute quality of the performance: a *Relative Goodput* (resp. *Relative Yield*) of 0.8 means that the studied algorithm achieves at least 80% of the performance of an optimal solution. Thanks to the use of relative performance indicators, the evaluation depends less on the difficulty of the different instances, and hence we can better highlight the differences between the algorithms.

6.1.1 Workloads

We generate workloads from traces taken from the Parallel Workloads Archive [12]. We have chosen three workloads with significantly different distributions of the number of processors used by jobs. The first workload (*KIT*) comes from a year and a half recording from the ForHLR II system located at the Karlsruhe Institute of Technology in Germany [13]. The second workload (*Thunder*) comes from several months of recordings from a large Linux cluster called Thunder installed at Lawrence Livermore National Laboratory [14]. The third workload (*UniLu*) comes from three months of data from the Gaia cluster at the University of Luxembourg [15]. In the first set of simulations, we consider jobs of infinite duration and the only data we are interested in is the number of processors required by each job, whose distribution is presented on Figure 5. The distribution of job execution times (with respect to the required number of processors) is presented in the Appendix (Figure 24).

6.1.2 Parameters

In the simulations, we have independently varied almost all parameters. We list below either the values they can take or their default values (their variations will then be made explicit in each particular study):

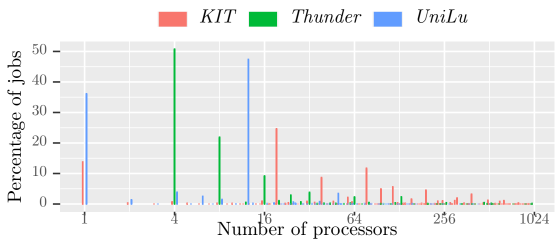


Figure 5: Number of processors required by each job.

- the number of sections is $n = 400$;
- the average duration of a section is set by default to $avgT = 100$; then, the duration of each section is drawn uniformly between $0.8avgT$ and $1.2avgT$; furthermore, the cost of checkpoints C_j are drawn uniformly in $[5, 20]$, and the cost of the recovery R_j is equal to C_j; hence, varying $avgT$ amounts at varying the relative cost of checkpoints (and of recoveries) with respect to the duration of sections;
- the maximum number of processors is, by default, $P_{\max} = 400$;
- the minimum number of processors is $P_{\min} = 0.2P_{\max}$ (to keep *Yield* calculations simple, we do not allow jobs to use more than $P_{\min}$ processors);
- the bound on the maximum variation in the number of processors between two consecutive sections is $\delta = 0.1P_{\max}$;
- the *Load* of the system, that is the ratio of the total required number of processors ($\sum_{j=1}^{m} p_j$) to $P_{\max}$, is set to $Load = 2P_{\max}$.

Finally, as some of the algorithms (or versions of them) may be too computationally and memory intensive, we only run them on smaller instances (called *light simulation*), setting $P_{\max} = 100$, while the other parameters keep their default values.

6.2 Results

For each combination of parameters, we randomly generate ten instances and we report the geometric mean of the achieved performance. The choice of the workload does not have a significant impact on the performance of the different algorithms. Therefore, we report the average performance over the three workloads. The performance detailed by workloads can be found in Section E of the Appendix.

6.2.1 DPGoodput versus DPGoodputSlow

In this section, we compare the different dynamic programming algorithms defined in Section 4, that is, the original version DPGoodputSlow-Best-C_{nf}, and its variants that have a phase 1 of fixed length $C_{\max}$ (variants without the -Best-C_{nf} suffix), or that optimize separately phases 1 and 2, and phase 3 (variants without the Slow suffix). We compare their performance using the *light simulation* parameters, since the *slow* DP algorithms have a high complexity.

Figure 6 presents the *Relative Goodput* achieved by the four algorithms for each of the three traces. First, all algorithms achieve very similar performance and the quicker variant, DPGoodput, can be safely used. In each case, the variant with -Best-C_{nf} achieves slightly better goodput

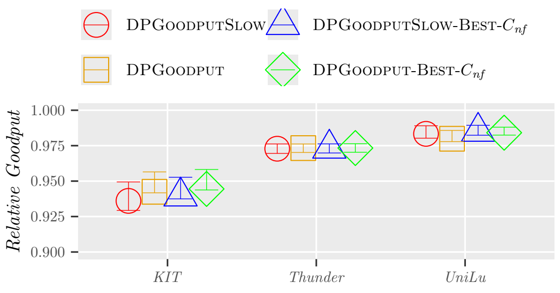


Figure 6: Goodput achieved by the variants of DPGoodput.

than its counterpart. For the *KIT* trace, DPGoodput achieves slightly better performance than DPGoodputSlow, which illustrates the fact that, although DPGoodputSlow delivers a solution very close to the optimal one section by section, this does not guarantee that it achieves an overall optimal. For the *UniLu* trace, the situation is reversed and DPGoodputSlow achieves better performance. With the *Thunder* trace, the goodputs are undistinguishable.

Similarly, further simulations show that the goodputs achieved by DPBiC–Best-C_{nf} and DPBiC are very close (see Section E.1 of the Appendix), and hence we only consider DPBiC in what follows.

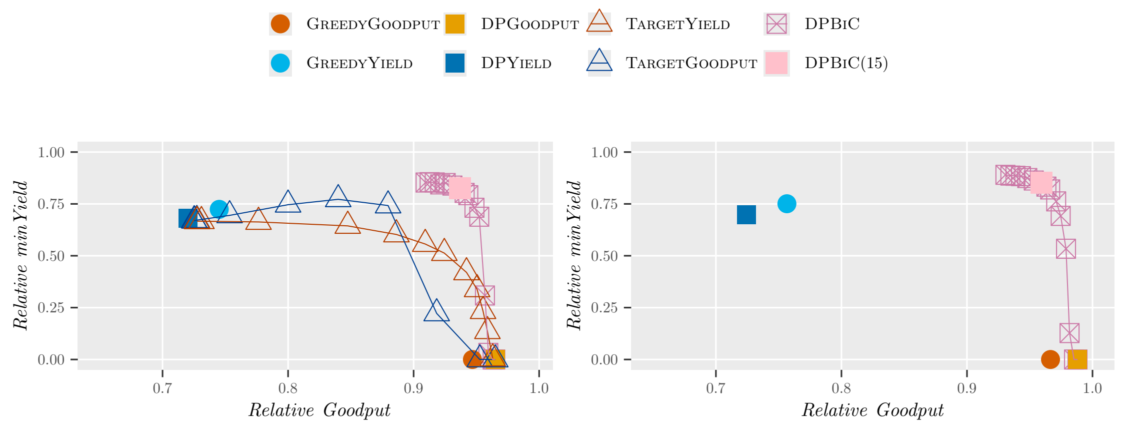
6.2.2 Pareto comparison of all algorithms

We now compare on Figure 7a all the algorithms with the *light simulation* parameters. We use a smaller setting in order to be able to run the most expensive algorithms, including TargetGoodput. We run TargetYield($\mathcal{Y}$) with $\mathcal{Y} \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9\}$, TargetGoodput($\mathcal{G}$) with $\mathcal{G} \in \{0.6, 0.65, 0.7, 0.75, 0.8, 0.84, 0.88, 0.92, 0.96, 1\}$, and DPBiC($\mathcal{X}$) with $\mathcal{X} \in \{0, 1, 2, 3, 5, 7, 10, 12, 15, 20, 25, 30, 35, 40, 45\}$, as well as the four single-criteria algorithms, GreedyGoodput, GreedyYield, DPGoodput and DPYield. We also compare these algorithms, except TargetGoodput and TargetYield, under the *default simulation* settings on Figure 7b. We highlight in both figures the point corresponding to DPBiC(15), which is a middle value of the heuristic parameter offering an interesting trade-off.

In these figures, the highest *Relative Goodput* is achieved by DPGoodput, whose performance is (slightly) better than that of GreedyGoodput. For the *Relative Yield*, GreedyYield achieves a slightly better performance than DPYield.

Then, the TargetYield and TargetGoodput algorithms, with all their possible parameters, propose some trade-offs between the two objectives. TargetYield performs similarly to the *Yield*-oriented algorithms when its *Yield* target is large. Its *Relative Goodput* quickly improves when this constraint is relaxed, allowing it to compete with DPGoodput. TargetGoodput, if very lightly constrained, matches the performance of *Yield*-oriented algorithms, as it then aims to maximize the *minYield*. When it is constrained to have a slightly higher *Goodput*, its *Relative Yield* improves and even dominates that of TargetYield. This is because, to achieve a higher *Goodput*, it makes less drastic changes to the set of active jobs which, in the long run, pays off because less time is wasted in checkpoints and recoveries.

This tendency reaches its peak with the DPBiC algorithm. This algorithm succeeds in achieving a very high *Relative Yield* for a very small degradation of the *Relative Goodput*. DPBiC almost succeeds in avoiding the *Goodput* versus *minYield* trade-off as it defines alone the Pareto front. In the figures, we highlight the performance of DPBiC(15), which offers an excellent trade-



(a) *Light simulation.* (b) *Default simulation.*

Figure 7: Pareto comparison of the performance of the different algorithms.

off, achieving close to best relative min yield and goodput. In the remainder of the evaluations, we will always consider DPBiC with this parameter value of 15.

6.2.3 Parameter variations

To check the stability of algorithm performance, we vary the five parameters we thought would be the most impactful: *avgT*, $P_{\max}$, $P_{\min}$, δ, and *Load*.

Figure 8 reports algorithm performance when parameter *avgT* varies, Figure 9 when $P_{\max}$ varies, Figure 10 when $P_{\min}$ varies, Figure 11 when δ varies, and Figure 12 when *Load* varies. We now discuss these results, starting with general comments on *Relative Goodput* and *Relative - minYield*.

Relative Goodput In all configurations, GreedyGoodput, DPGoodput, and DPBiC(15) achieve extremely close and near optimal *Relative Goodput*. DPGoodput achieves the best performance in each case.

GreedyYield and DPYield achieve worse performance (with GreedyYield being slightly better). This is easily explained. Because these heuristics target the optimization of the minimum yield, they guarantee that each job has access to processors. Roughly speaking, they all implement a kind of weighted round-robin. Hence, they perform more checkpoints and recoveries than the heuristics designed for goodput optimization, which leads to worse goodputs. DPYield achieves even more checkpoints and recoveries because it can recover a job and checkpoint it both during phase 1, something GreedyYield would never do. These heuristics nevertheless always achieve at least 68% of the optimal relative goodput.

As a conclusion, all heuristics achieve very good *Relative Goodput* and the three best ones have near-optimal performance. This is not very surprising. Indeed, any reasonable heuristic, in the worst case, would spend the first $C_{\max}$ units of time of a section checkpointing all jobs, then the next $R_{\max}$ units of times recovering jobs before using as many processors as possible until the end of the section where it will once again spend $C_{\max}$ units of time checkpointing all jobs. Therefore, the goodput of any reasonable heuristic (using all processors) on section i is therefore

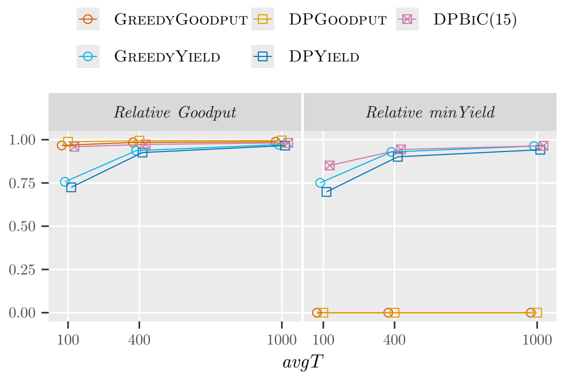


Figure 8: Impact of the average section length.

not smaller than $\frac{T_{i+1}-T_i-2C_{\max}-R_{\max}}{T_{i+1}-T_i}$. Hence, if checkpointing and recovery costs are rather small with respect to section durations, the achieved *Goodput* will necessarily be high. Because the *Relative Goodput* takes mandatory checkpoints and recoveries into account, the achieved *Relative Goodput* should look even better.

Relative minYield Behaviors are more diverse for *Relative minYield*. GreedyGoodput and DPGoodput achieve a *Relative minYield* of 0, on all instances. This means that at least one job is victim of starvation, that is, is never executed, as already discussed in Section 3.

DPBiC(15) always achieves the best performance. Its *Relative minYield* is in most cases above 0.8, meaning that it achieves a minimum yield that is at most 20% away from an upper bound on the optimal, and thus at most 20% away from the optimal. This performance is due to the high *Goodput* maintained by this algorithm, and thus to the greater amount of work performed, which, when evenly distributed, leads to a higher *minYield*. GreedyYield and DPYield achieve significantly worse but similar performance, with a *Relative minYield* usually around 0.75.

The overall conclusion is that some heuristics achieve high-quality *Relative minYield*, but that a heuristic that only targets *Goodput* optimization can achieve an abysmal *Relative minYield*. On the contrary, all heuristics that target *minYield* optimization achieve good *Relative Goodput*.

Influence of *avgT* (Figure 8) When the average section duration increases, the *Relative - Goodput* of all algorithms increases but this is more significant for the two worse algorithms, GreedyYield and DPYield, whose performance becomes comparable to that of the best three heuristics. This is because the total cost of checkpoints and recoveries in a section is bounded and becomes negligible when the duration of the section increases. When the section duration is small, decisions have a greater impact and differences between heuristics performance are more pronounced.

For the same reason, the *Relative Yield* of GreedyYield and DPYield also increases with the average section duration. This phenomenon also applies to DPBiC(15), although to a lesser extent, as the number of checkpoints and recoveries performed by this algorithm is already limited.

Influence of $P_{\max}$ (Figure 9) $P_{\min}$, δ and *Load* are always defined as fixed-ration fractions of $P_{\max}$. When $P_{\max}$ increases, there is a slight increase in *Relative Goodput* and *Relative Yield*.

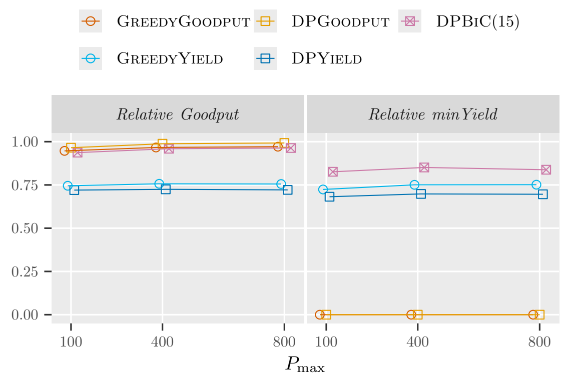


Figure 9: Impact of the maximum number of processors.

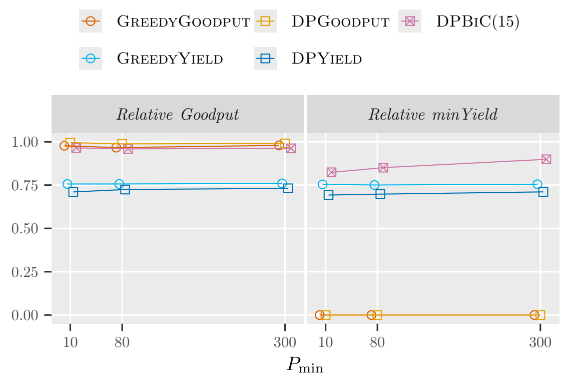


Figure 10: Impact of the minimum number of processors.

This is because when $P_{\max}$ increases, the number of jobs in the system increases and, therefore, the possibility to find a better solution.

Influence of $P_{\min}$ (Figure 10) All algorithms have very stable performance, except for DPBiC(15), which suffers from a slight decrease of its *Yield* when $P_{\min}$ is very small. This seems to be a consequence of the experimental constraint to only have jobs using at most $P_{\min}$ processors. Therefore, for a same value of $P_{\max}$, there are more jobs when $P_{\min} = 10$ than when $P_{\min} = 300$, and therefore more flexibility in obtaining a good *minYield*. This effect is not noticeable on GreedyYield and DPYield.

Influence of δ (Figure 11) The computations of the upper bounds of the objective functions take into account the fact that at each section start, the number of available processors varies by an amount $\Delta \leq \delta$: they account for a loss of working time equal to $\delta C_{\min}$ (or slightly less if the actual number of processors is close to $P_{\min}$), plus $\Delta R_{\min}$ when the number of processors increases ($\Delta > 0$). *Yield*-oriented algorithms tend to change the majority of their active jobs at each start of section. When δ increases, a higher fraction of the cost of this change of active jobs is "covered" by the computation of the upper bounds. This explains that the performance of *Yield*-oriented algorithms improve for both objectives when δ increases.

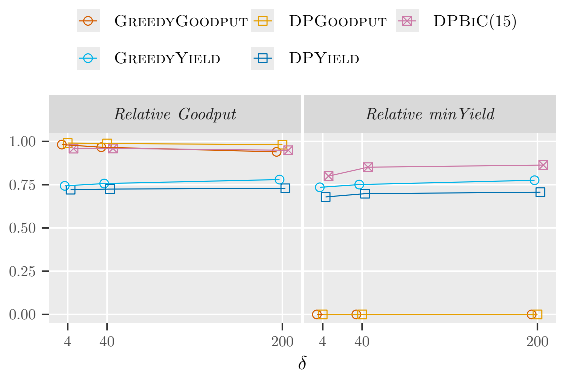


Figure 11: Impact of variability on number of processors.

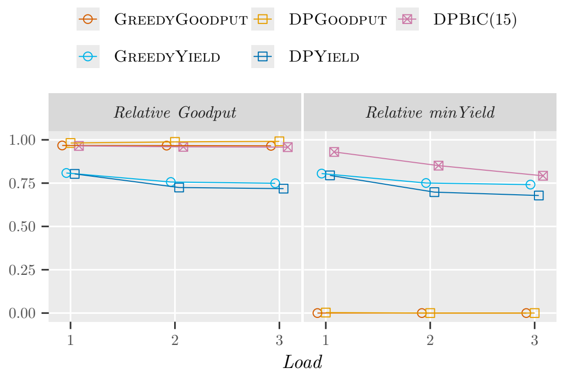


Figure 12: Impact of the overall *Load*.

On the contrary, the way upper bounds are computed leads to a decrease in the *Relative Goodput* for *Goodput*-oriented algorithms and DPBiC. Indeed, upper-bound assumes (conservatively) that each checkpoint (resp. recovery) lasts for the minimum duration $C_{\min}$ ($R_{\min}$). When δ increases, more jobs are checkpointed (resp. recovered) and the maximum checkpointing (resp. recovery) cost of checkpointed (resp. recovered) jobs increases. Hence, there is an increased gap between the bound and the actual performance of these algorithms. This is also true for DPBiC(15) after the first phase where it benefits from a larger value of δ for the same reason than the *Yield*-oriented algorithms.

As in the previous cases, an increase in *Relative Goodput* implies an increase in *Relative minYield* for the algorithms aiming to improve it.

Influence of *Load* (Figure 12) The *Load* is computed in relation to $P_{\max}$. Hence, when *Load* $= 1$, this means that if the number of processors available is $P_{\max}$, then all jobs can be active at the same time. When *Load* increases, *Yield*-oriented heuristics have more jobs to circle through. Therefore, they pay more checkpoints and recoveries and both their *Relative Goodput* and *Relative Yield* decrease. The *Goodput*-oriented heuristics achieve near-optimal *Goodput*: this is already the case when *Load* $= 1$, and a larger *Load* means a larger number of jobs to choose from and, thus, a higher probability to be able to use all processors.

DPBiC(15) maintains its goodput when *Load* increases. In this case, the number of jobs in the system also increases. However, DPBiC(15) does not aim to systematically execute the jobs with the lowest yields, but it rather gives the priority to jobs with a low yield. When there are more jobs, there might be more jobs with a low yield, in which case DPBiC(15) may be less likely to launch at each section the job with the lowest yield, hence a decrease in *RelativeYield*.

6.3 Finite duration jobs

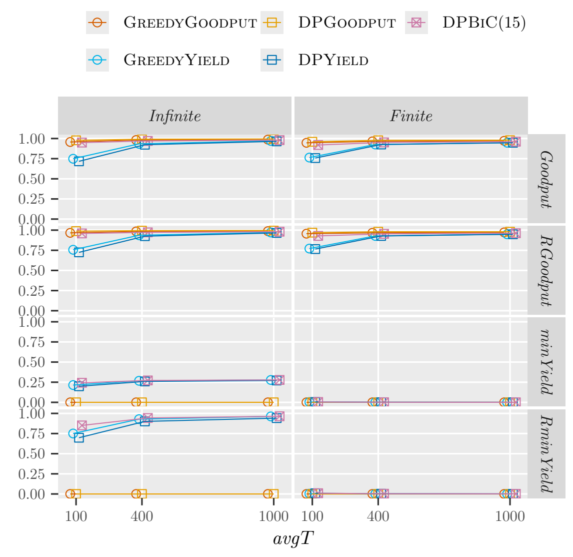
We now want to assess whether the conclusions drawn with infinite jobs still hold in the more realistic setting of jobs with a finite duration. The algorithms are adapted to this finite jobs context as described in Section 5. The workloads are generated from the same traces than previously, with the additional information of the job durations (which is included in the traces). When generating the checkpoint and recovery costs of a job, we upper bound these values to be at most one-tenth of the job execution time if they exceed 5 time units, and we set them at this lower-bound of 5 time units otherwise. Mathematically speaking, if x is the randomly generated checkpoint cost, we have $C_j = \max\{5, \min\{x, L_j/10\}\}$, hence having a checkpointing cost of at least 5, and such that the cost is not too high for small jobs. Finally, we need to define release dates for the jobs in order for the system load to reach the target *Load*. We consider three variants, called Accumulation, FIFO and Smallest, whose characteristics are compared in Section E.5 of the Appendix. We select the FIFO method for the rest of this section, as it gives us the most stable load for all algorithms, ensuring that the load is achieved with jobs scheduled in a FIFO-like order (see the Appendix for details).

6.3.1 Comparison between finite and infinite settings

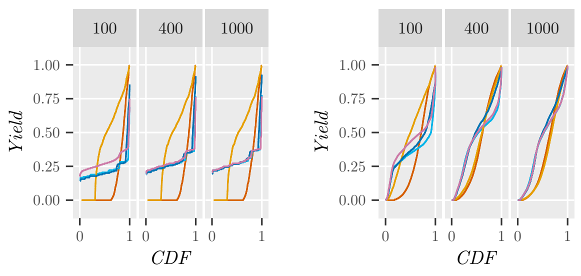
On Figure 13a, we compare the algorithms while varying the average length of a section. For both the *Goodput* and the *Relative Goodput*, there is no visible difference between the performance achieved for the finite case and the infinite one. This is true for the average length of a section (Figure 13a), but also for all other parameters (all figures for the other parameters can be found in Section E.6 of the Appendix). In particular, jobs that finish in the middle of a section are correctly replaced without noticeable impact on the performance.

On the contrary, the achieved *minYield* and *Relative minYield* are very different, and close to 0 with finite jobs. This is due to the fact that a job released during a section will not be allocated processors before the start of the next section (except if some jobs complete before the end of the section while releasing enough processors). Hence, the *Yield* of that job – and the *minYield* of the instance – will be zero at the end of the section. If this section is the last of the whole simulation, the overall *minYield* will be null. Furthermore, if the released job is short, the time it spends waiting for the beginning of the next section and the availability of processors will significantly impact its yield, whatever its future allocation of processors. Hence, with finite jobs, to be able to compare the performance of the different algorithms, we cannot just rely on the *minYield*, and we rather consider the (cumulative) distribution of the yields, as presented on Figure 13b.

We should be careful when comparing the cumulative distribution of the yields in the finite and infinite case because, in the infinite case, the algorithms deal with the exact same set of jobs throughout the simulation, which is obviously not the case with finite jobs. The increase of the average section length mainly impacts the performance with finite jobs. There, the longer the sections, the closer the performance of (similar) heuristics. This is, once again, because decisions taken at the beginning and/or at the end of a section have less impact when the size of that section is larger.



(a) Impact of the average section length.



(b) Cumulative distribution function for the *Yield* for the different average section lengths.

Figure 13: Comparison of the performance achieved when jobs are infinite (on the left) or finite (on the right).

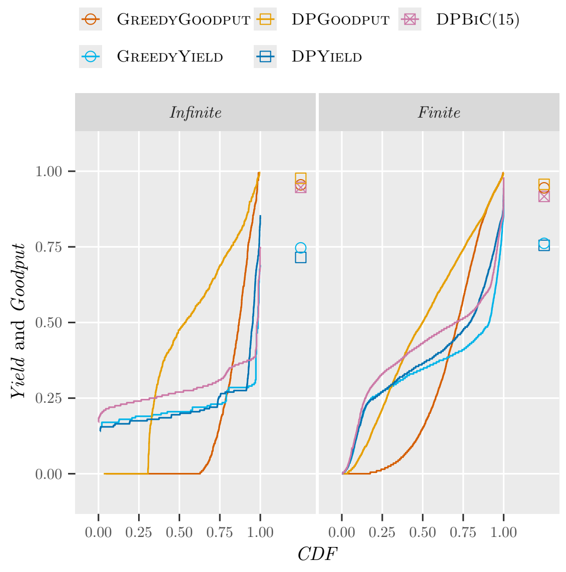


Figure 14: Cumulative distribution function for the *Yield* and value of the *Goodput* achieved by the different algorithms when jobs are infinite (on the left) or finite (on the right).

6.3.2 Yield evolution

Figures 14 and 15 present the cumulative distribution functions (CDF) for the yield of the different algorithms, as well as their *Goodput* (the points on the right of each subgraph) under the *default simulation* parameters. Because we want the smallest yields to be as large as possible, the higher and the leftmost the CDF, the better. The leftmost point of a CDF is the *minYield* of the considered algorithm.

Figure 14 shows that, for the infinite jobs case, more than 25% of the jobs are never executed with DPGoodput, and this value increases to over 60% for GreedyGoodput. On the contrary, for the three other algorithms, 95% of the jobs have roughly similar yields (comprised between 0.15 and 0.30 for DPYield and GreedyYield, and between 0.20 and 0.35 for DPBiC). Hence, the algorithms targeting the optimization of the *minYield* happen to be fair by achieving comparable performance for most of the jobs. The conclusions for finite jobs are rather similar. The main difference is that DPGoodput is far less unfair.

Finally, Figure 15 shows the influence of the $\mathcal{X}$ parameter of DPBiC($\mathcal{X}$). DPBiC(0) is exactly DPGoodput. The larger the value of $\mathcal{X}$, the more fair the algorithm in both the infinite- and finite-jobs cases, and the lower the *Goodput*. In the infinite case, we only wanted to maximize the *minYield*, which led us to choose $\mathcal{X} = 15$. In the finite case, the *minYield* can no longer be guaranteed. If we only want to maximize the *Yield* of 90% of the jobs, we can achieve that by setting $\mathcal{X} = 2$. With DPBiC(2) we have the guarantee that all jobs will eventually be executed but we are not over-optimizing the *minYield* at the detriment of the *Goodput*.

Overall, the behaviors of the different algorithms under the finite-jobs settings are quite similar to their behaviors under the infinite-jobs one. This validates our approach. The main difference is that the *minYield* metric is not pertinent in the case of finite jobs and one should rather consider the distribution of the yields.

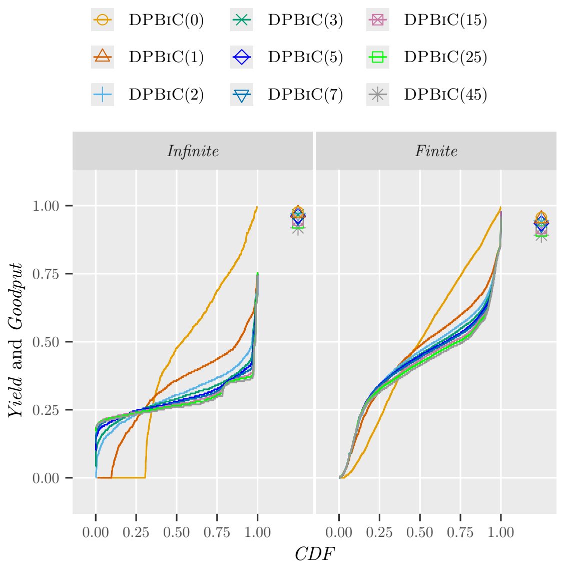


Figure 15: Comparison of CDF of *Yield* of various DPBɪC.

7 Conclusion

In this paper, we have addressed the problem of scheduling parallel rigid jobs onto a set of processors subject to variations in their number, with the objective of optimizing both the goodput and the minimum yield. The goal is to maximize platform utilization while keeping some fairness between jobs. Jobs are allowed to take checkpoints before a change in the number of processors; hence, we never loose work that was previously done.

We have formalized the problem, provided upper bounds to the objective functions, and designed sophisticated dynamic programming algorithms, as well as greedy solutions, aiming at optimizing both of the objectives in a setting with jobs of infinite duration. The simulation results with a wide range of parameters, including both infinite and finite duration jobs, demonstrate that the bi-criteria dynamic programming algorithm DPBɪC achieves impressive results both in terms of yield and goodput, in a great variety of scenarios. In particular, DPBɪC(15) achieves a great trade-off between the two objective functions.

This work provides a first step towards designing solutions with guaranteed performance, by thoroughly analyzing the case of infinite jobs. The proposed algorithms have been shown to be adaptable in models with release dates, and they could be further extended to jobs with deadlines or priorities, building on the yield and release dates. Now that the performance of the strategies has been demonstrated, another interesting research direction would be to relax some of the hypotheses, for instance by considering moldable jobs, and accounting for non-exact predictions, i.e., variation in the time of the next event. This initial study opens an avenue of challenging problems in the context of data centers subject to variable capacity.

References

[1] L. Alderman. French Nuclear Power Crisis Frustrates Europe's Push to Quit Russian Energy. *The New York Times*, June 2022. `https://www.nytimes.com/2022/06/18/business/`

`france-nuclear-power-russia.html`.

[2] J. Basney and M. Livny. Improving Goodput by Coscheduling CPU and Network Capacity. *IJHPCA*, 13(3):220–230, 1999.

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

[4] A. Benoit, A. A. Chien, and Y. Robert. Scheduling variable capacity resources for sustainability workshop. Technical report, INRIA; University of Chicago., 2023. `https://inria.hal.science/hal-04159509`.

[5] P. Blanc, J. Remund, and L. Vallance. Short-term solar power forecasting based on satellite images. In G. Kariniotakis, editor, *Renewable Energy Forecasting*, Woodhead Publishing Series in Energy, pages 179–198. Woodhead Publishing, 2017.

[6] California ISO. Today's outlook. `https://www.caiso.com/todays-outlook`.

[7] B. Camus, F. Dufossé, and A.-C. Orgerie. The SAGITTA approach for optimizing solar energy consumption in distributed clouds with stochastic modeling. In *Smart Cities, Green Technologies, and Intelligent Transport Systems*, pages 1–25. Dec. 2018.

[8] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2001.

[10] I. n. Goiri, K. Le, M. E. Haque, R. Beauchea, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenSlot: scheduling energy consumption in green datacenters. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[11] M. E. Haque, I. n. Goiri, R. Bianchini, and T. D. Nguyen. GreenPar: Scheduling Parallel High Performance Applications in Green Datacenters. In *Proc. of the 29th ACM Int. Conf. on Supercomputing*, ICS '15, page 217–227, 2015.

[12] Parallel workloads archive. `https://www.cs.huji.ac.il/labs/parallel/workload/logs.html`.

[13] KIT ForHLR II log, September 2019. `https://www.cs.huji.ac.il/labs/parallel/workload/l_kit_fh2/index.html`.

[14] LLNL Thunder log, March 2008. `https://www.cs.huji.ac.il/labs/parallel/workload/l_llnl_thunder/index.html`.

[15] University of Luxemburg Gaia Cluster log, March 2015. `https://www.cs.huji.ac.il/labs/parallel/workload/l_unilu_gaia/index.html`.

[16] L. Perotin, C. Zhang, R. Wijayawardana, A. Benoit, Y. Robert, and A. Chien. Risk-aware scheduling algorithms for variable capacity resources. In *Proc. of SC '23 Workshops (PMBS)*, page 1306–1315, 2023.

[17] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2023.

[18] Wikipedia. Rolling blackout. `https://en.wikipedia.org/wiki/Rolling_blackout#USA`.

[19] Wikipedia. South African energy crisis. `https://en.wikipedia.org/wiki/South_African_energy_crisis`.

[20] F. Yang and A. A. Chien. ZCCloud: Exploring wasted green power for high-performance computing. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1051–1060, 2016.

[21] C. Zhang and A. A. Chien. Scheduling challenges for variable capacity resources. In *Job Scheduling Strategies for Parallel Processing*, pages 190–209. Springer, 2021.

A Pseudo-code for algorithm GreedyGoodput

We detail here the pseudo-code for algorithm GreedyGoodput (Algorithm 1). This algorithm is designed to schedule jobs $J_1, \ldots, J_m$ on the n sections that form the problem, with the aim of maximizing goodput, in a greedy way. At the beginning of each section s, as long as there are processors available and not in use, it considers the inactive jobs (in non-increasing order of the number of processors required) and launches the recovery and then the execution of each job as long as this is possible without exceeding the number of available processors. Then, starting from line 9, it determines which active jobs will be checkpointed at the end of section s, to keep at most only $\max\{P_s - \delta, P_{\min}\}$ processors used at the end of the section.

Algorithm 1: GreedyGoodput

```
1  Rename jobs $J_1, \ldots, J_j, \ldots, J_m$ by non-increasing number of required processors, $p_j$
2  $used \leftarrow 0$                              /* Number of processors used */
3  for $s = 1$ to $n$ do
       /* Starting and recovering jobs at time $T_s$ */
4      $j \leftarrow 1$
5      while $used < P_s$ and $j \leq m$ do
6          if $J_j$ is not active and $used + p_j \leq P_s$ then
               /* Job $J_j$ will be recovered (and then executed) at $T_s$ */
7              $used \leftarrow used + p_j$
8          $j \leftarrow j + 1$
       /* Checkpointing jobs before time $T_{s+1}$ */
9      $used \leftarrow 0$     /* Number of processors remaining in use during the next section */
10     $j \leftarrow 1$
11     while $used < \max\{P_s - \delta, P_{\min}\}$ and $j \leq m$ do
12         if $J_j$ is active then
13             if $used + p_j \leq \max\{P_s - \delta, P_{\min}\}$ then
                   /* Job $J_j$ will remain active at the end of the section */
14                 $used \leftarrow used + p_j$
15             else
                   /* Job $J_j$ will be checkpointed at time $T_{s+1} - C_j$ */
16         $j \leftarrow j + 1$
```

B Proof of Theorem 1

In this section, we prove Theorem 1, which establishes the optimality of algorithm DPGoodput for the maximization of goodput in a single section when all checkpoints have the same duration.

Theorem 2. *Let i be any section, with $1 \leq i \leq n$, and assume that there exists a schedule for sections 1 through $i-1$ (if $i > 1$). Then, let $\mathcal{O}^i$ be a schedule maximizing the goodput Goodput^i during section i, $[T_i, T_{i+1}]$. If all checkpoints last the same duration C, then under schedule $\mathcal{O}^i$, checkpoints only start at time T_i (to complete at time $T_i + C$) or at time $T_{i+1} - C$ (to complete at time T_{i+1}).*

Proof. Let $\mathcal{J}^s_{\mathcal{O}}$ be the set of jobs that are allocated processors at the start of the section, that is, right after time T_i. Let $\mathcal{J}^e_{\mathcal{O}}$ be the set of jobs that are allocated processors at the end of the section, that is, right before time T_{i+1}. Let us denote by $\mathcal{J}^{ckpt}_{\mathcal{O}} \subseteq \mathcal{J}^e_{\mathcal{O}}$ the set of jobs that are checkpointed by $\mathcal{O}$ at the end of the section. Finally, let q be the maximum number of processors used during (T_i, T_{i+1}). We have two cases to consider depending on the relationship between q and $\sum_{j\ \in \mathcal{J}^s_{\mathcal{O}}} p_j$.

Case 1: $q = \sum_{j \in \mathcal{J}^s_{\mathcal{O}}} p_j$. We now build a new schedule $\mathcal{S}$ as follows: $\mathcal{J}^s_{\mathcal{S}} = \mathcal{J}^e_{\mathcal{S}} = \mathcal{J}^s_{\mathcal{O}}$, that is, $\mathcal{S}$ does not checkpoint any job at the start of the section, then it executes throughout the section the jobs that where initially active under $\mathcal{O}$. And finally, $\mathcal{S}$ checkpoints at the end of the section the jobs that were active under $\mathcal{O}$ at the beginning of the section and that are either checkpointed by $\mathcal{O}$ or no longer active at the end of the section: $\mathcal{J}^{ckpt}_{\mathcal{S}} = (\mathcal{J}^s_{\mathcal{O}} \setminus \mathcal{J}^e_{\mathcal{O}}) \cup \left(\mathcal{J}^s_{\mathcal{O}} \cap \mathcal{J}^{ckpt}_{\mathcal{O}}\right)$. Then, we check that there are at least as many free processors under $\mathcal{S}$ than under $\mathcal{O}$ at time T_{i+1}:

$$\begin{aligned}
\sum_{j \in \mathcal{J}^{ckpt}_{\mathcal{S}}} p_j &= \sum_{j \in \mathcal{J}^s_{\mathcal{O}} \setminus \mathcal{J}^e_{\mathcal{O}}} p_j + \sum_{j \in \mathcal{J}^s_{\mathcal{O}} \cap \mathcal{J}^{ckpt}_{\mathcal{O}}} p_j \\
&\geq \sum_{j \in \mathcal{J}^e_{\mathcal{O}} \setminus \mathcal{J}^s_{\mathcal{O}}} p_j + \sum_{j \in \mathcal{J}^s_{\mathcal{O}} \cap \mathcal{J}^{ckpt}_{\mathcal{O}}} p_j \\
&\geq \sum_{j \in \mathcal{J}^{ckpt}_{\mathcal{O}} \setminus \mathcal{J}^s_{\mathcal{O}}} p_j + \sum_{j \in \mathcal{J}^s_{\mathcal{O}} \cap \mathcal{J}^{ckpt}_{\mathcal{O}}} p_j = \sum_{j \in \mathcal{J}^{ckpt}_{\mathcal{O}}} p_j
\end{aligned}$$

because, by hypothesis $q = \sum_{j \in \mathcal{J}^s_{\mathcal{O}}} p_j$, implies $\sum_{j \in \mathcal{J}^s_{\mathcal{O}}} p_j \geq \sum_{j \in \mathcal{J}^e_{\mathcal{O}}} p_j$, which implies that $\sum_{j \in \mathcal{J}^s_{\mathcal{O}} \setminus \mathcal{J}^e_{\mathcal{O}}} p_j \geq \sum_{j \in \mathcal{J}^e_{\mathcal{O}} \setminus \mathcal{J}^s_{\mathcal{O}}} p_j$.

$\mathcal{S}$ uses q processors throughout the section, the maximum used by $\mathcal{O}$ during the section. Each job checkpointed by $\mathcal{S}$ during the section is only checkpointed once by $\mathcal{S}$ and is checkpointed at least once by $\mathcal{O}$. Finally, $\mathcal{S}$ only performs recoveries for jobs belonging to $\mathcal{J}^s_{\mathcal{S}} = \mathcal{J}^s_{\mathcal{O}}$ but not active right before time T_i. $\mathcal{O}$ also must perform recoveries for these jobs. Therefore, the goodput of $\mathcal{S}$ during this section is not smaller than that of $\mathcal{O}$ and $\mathcal{S}$ is thus optimal. It is then easy to see that if $\mathcal{S}$ started a checkpoint earlier than at time $T_{i+1} - C$, it would be suboptimal.

Case 2: $q > \sum_{j \in \mathcal{J}^s_{\mathcal{O}}} p_j$. Let τ be the first instant under $\mathcal{O}$ at which q processors are used and let $\mathcal{J}^{max}_{\mathcal{O}}$ be the set of simultaneously active jobs at that time, $\sum_{j \in \mathcal{J}^{max}_{\mathcal{O}}} p_j = q$, and hence $\mathcal{J}^{max}_{\mathcal{O}} \neq \mathcal{J}^s_{\mathcal{O}}$. We now build a new schedule $\mathcal{S}$ as follows: $\mathcal{S}$ starts with jobs in $\mathcal{J}^s_{\mathcal{O}}$ active, it checkpoints jobs in $\mathcal{J}^s_{\mathcal{O}} \setminus \mathcal{J}^{max}_{\mathcal{O}}$ and these checkpoints end at time $T_i + C$; thereafter $\mathcal{S}$ executes until time T_{i+1} the set of jobs $\mathcal{J}^{max}_{\mathcal{O}}$. Hence, $\mathcal{J}^s_{\mathcal{S}} = \mathcal{J}^s_{\mathcal{O}}$, $\mathcal{J}^e_{\mathcal{S}} = \mathcal{J}^{max}_{\mathcal{O}}$. All checkpoints and recoveries executed between T_i and T_{i+1} in $\mathcal{S}$ are executed at least once in $\mathcal{O}$ between T_i and T_{i+1} because $\mathcal{O}$ switches from the set $\mathcal{J}^s_{\mathcal{O}}$ to the set $\mathcal{J}^{max}_{\mathcal{O}}$. Then,

$T_i + C \leq \tau$ because $\mathcal{J}_{\mathcal{O}}^{max} \neq \mathcal{J}_{\mathcal{O}}^{s}$ and so there is at least one job in $\mathcal{O}$ that needs to be checkpointed (because $\mathcal{O}$ is optimal, if it does not use q processors from time T_i, this is because this is impossible as long as some job has not been checkpointed). However, in $\mathcal{S}$, all the required checkpoints take place from the start, so $\mathcal{S}$ uses q processors from $T_i + C$ to T_{i+1}, whereas $\mathcal{O}$ does so only starting at time τ. Finally, the set of checkpointed jobs at time T_{i+1} is:

$$\mathcal{J}_{\mathcal{S}}^{ckpt} = \left(\mathcal{J}_{\mathcal{O}}^{max} \cap \mathcal{J}_{\mathcal{O}}^{ckpt}\right) \cup \left(\mathcal{J}_{\mathcal{O}}^{max} \setminus \mathcal{J}_{\mathcal{O}}^{e}\right).$$

We need to check that there are at least as many free processors under $\mathcal{S}$ than under $\mathcal{O}$ at time T_{i+1}. Let us consider any processor that is free under $\mathcal{O}$ at time T_{i+1}. If this processor was unused at time τ under $\mathcal{O}$, it is unused at time T_{i+1} under $\mathcal{S}$. If it was used at time τ under $\mathcal{O}$, either it was used by a job that is checkpointed at the end of the section by $\mathcal{O}$ or it was used by a job no longer active at time T_{i+1} by $\mathcal{O}$. In both cases, this job starts to be checkpointed at time $T_{i+1} - C$ by $\mathcal{S}$.

To conclude, note that $\mathcal{S}$ is optimal for goodput optimization, because its goodput is not smaller than that of $\mathcal{O}$: its total idle time is not larger (it is equal to the idle time under $\mathcal{O}$ up to time $T_i + C \leq \tau$, because $\mathcal{O}$ cannot use more than $\sum_{j \in \mathcal{J}_{\mathcal{O}}^{s}} p_j$ processors before $T_i + C$, and it is not larger afterwards), and it only performs checkpoints and recoveries that $\mathcal{O}$ performs.

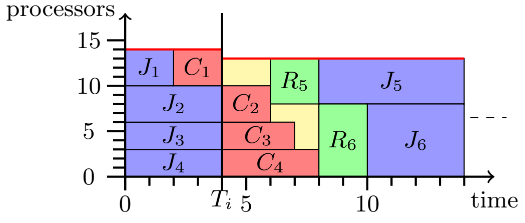
In both cases, schedule $\mathcal{S}$ has a *Goodput* no smaller than the one of $\mathcal{O}$ and it only preform checkpoints, if any, at the start and at the end of the section. $\square$

C Checkpoints in phase 1

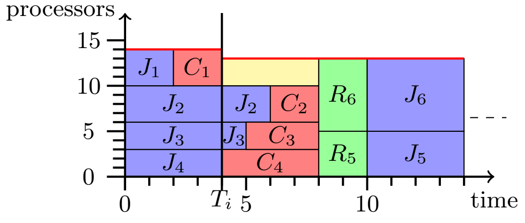
In this section, we focus on the case where checkpoints do not all have the same duration. We know that when all checkpoints last the same duration, DPGoodputSlow-Best-C_{nf} is optimal. The question is whether DPGoodputSlow-Best-C_{nf} is optimal in the general case and, if not, whether it could have been possible to design an optimal algorithm.

Considering algorithms organized in three phases like DPGoodputSlow-Best-C_{nf}, we show on an example that the time at which checkpoints are taken during the first phase influences the goodput achieved, that there are cases where checkpoints should not all be taken at the earliest, or all completed at the latest.

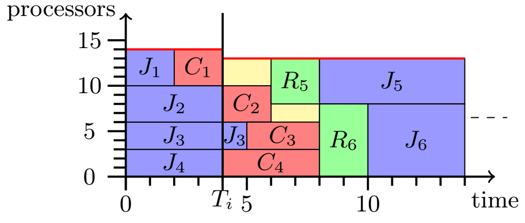
To compare different checkpoint configurations, we count the amount of idle time during phase 1. There are two obvious policies: either all jobs in $\mathcal{J}^s_{\mathcal{S}} \setminus \mathcal{J}^e_{\mathcal{S}}$ start their checkpoints at the beginning of the section and we launch a job from the set $\mathcal{J}^e_{\mathcal{S}}$ each time enough processors become available (see Figure 16a for an example), or jobs in $\mathcal{J}^s_{\mathcal{S}} \setminus \mathcal{J}^e_{\mathcal{S}}$ all complete their checkpoints at time $T_i + C_{nf}$. In the latter case, jobs with shorter checkpoint times continue their execution at the beginning of the section (see Figure 16b for an example). However, neither of these policies guarantees an optimal solution. Figure 16c presents another schedule for the same problem, which achieves a smaller amount of idle time. We conjecture that the problem of when to take checkpoints is NP-complete in the general case. Hence, for the dynamic programming algorithms, we have chosen the second policy, which we believe to be the most stable (for minimizing idle time) and easiest to implement.



(a) All checkpoints are taken starting at time 0. The total idle time is: $1\times2+2\times4+3\times1 = 13$.



(b) All checkpoints are completed at the latest (at time 4). The total idle time is: $3 \times 4 = 12$.



(c) Checkpoints are taken so as to minimize the total idle time which is equal to: $1\times2+2\times4 = 10$.

Figure 16: Checkpoints taken at the beginning (top left), at the end (top right), and in the middle (bottom) of phase 1.

D Dynamic programming formulas

We present in this section the different formulas used by the dynamic programming algorithms DPGoodputSlow-Best-C_{nf} (section D.1), DPGoodput (section D.2), DPYield (section D.3), TargetGoodput (section D.4) and DPBiC (section D.5). We reuse the notations defined in the Section 4.2 of the main article.

Each of these dynamic programming algorithms aims at optimizing one objective function (goodput, yield, etc.). However, there is no reason there exists a unique solution to their optimization problem. Therefore, when several solutions achieve the same performance for the target objective function, we pick a solution which we believe will enable to achieve better performance in practice. Among solutions achieving the same performance, we prefer solutions handling smaller jobs (on average) because we believe smaller jobs gives us more flexibility when building sets of jobs having a target total processor usage. Therefore, among solutions achieving the same performance, we prefer first solutions using a larger number of active jobs at the end of phase 3, and then solutions having a larger number of checkpointed jobs at the end of phase 3. Therefore, the equation below do not define scalar values but vectors of 2 or 3 elements. These vectors are ordered lexicographically in order to define our preferred solution achieving the optimal performance.

D.1 Equations for DPGoodputSlow-Best-C_{nf}

We write the gain function, $G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}) = (w, a, c)$, which decides how to handle jobs $J_1, \dots, J_j$ during section i. It is expressed as a triplet, the first value $G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}).w$ is the quantity of useful work, the second value $G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}).a$ is the number of active jobs at the end of phase 3, and the third value $G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}).c$ is the number of checkpointed jobs at the end of phase 3.

We denote by $\preceq_{lex}$ the lexicographic order on the triplets, and $\max_{lex}$ is the related maximum operation.

We introduce three sub-functions (with same parameters), which correspond to the possible statuses of J_j during phase 0: $G_j^{\texttt{Run}}$, $G_j^{\texttt{Ckpt}}$, and $G_i^{\texttt{Idle}}$. The gain function is expressed as:

$$G_j(\Pi_1, \Pi_2, \Pi_3, C_{nf}) = \begin{cases} G_j^{\texttt{Run}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{if } J_j \in \texttt{Run}^{(0)} \\ G_j^{\texttt{Ckpt}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{if } J_j \in \texttt{Ckpt}^{(0)} \\ G_j^{\texttt{Idle}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{if } J_j \in \texttt{Idle}^{(0)} \end{cases}$$

Hence, if job J_j was active during phase 0 and not checkpointed, $G_j^{\texttt{Run}}$ decides what is its best scenario. There are three cases for $G_j^{\texttt{Run}}$, as described in Section 4.2 of the paper, which leads to the following formula:

$$G_j^{\texttt{Run}}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) =$$
$$\max_{lex} \begin{cases} (p_j T, 1, 0) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3 - p_j, C_{nf}) & (1) \\ (p_j(T - C_j), 1, 1) + G_{j-1}(\Pi_1, \Pi_2 - p_j, \Pi_3, C_{nf}) & (2) \\ \begin{cases} (-\infty, 0, 0) & \text{if } C_j > C_{nf} \\ (p_j(C_{nf} - C_j), 0, 0) + G_{j-1}(\Pi_1, \Pi_2, \Pi_3, C_{nf}) & \text{otherwise} \end{cases} & (3) \end{cases}$$

Then, we now explicit the expression for $G_j^{\texttt{Ckpt}}$, which comprises the six cases when J_j has been checkpointed during phase 0:

$$G_j^{\texttt{Ckpt}}(\Pi_1,\Pi_2,\Pi_3,C_{nf}) =$$
$$\max_{lex}\begin{cases}(p_jT,1,0)+G_{j-1}(\Pi_1-p_j,\Pi_2-p_j,\Pi_3-p_j,C_{nf}) & (4)\\ (p_j(T-C_j),1,1)+G_{j-1}(\Pi_1-p_j,\Pi_2-p_j,\Pi_3,C_{nf}) & (5)\\ \begin{cases}(-\infty,0,0) & \text{if } C_j>C_{nf}\\ (p_j(C_{nf}-C_j),0,0)+G_{j-1}(\Pi_1-p_j,\Pi_2,\Pi_3,C_{nf}) & \text{otherwise}\end{cases} & (6)\\ (p_j(T-C_{nf}-R_j),1,0)+G_{j-1}(\Pi_1,\Pi_2-p_j,\Pi_3-p_j,C_{nf}) & (7)\\ (p_j(T-C_{nf}-R_j-C_j),1,1)+G_{j-1}(\Pi_1,\Pi_2-p_j,\Pi_3,C_{nf}) & (8)\\ (0,0,0)+G_{j-1}(\Pi_1,\Pi_2,\Pi_3,C_{nf}) & (9)\end{cases}$$

Next, we consider the last six cases, which correspond to J_j being idle during phase 0. The expression for $G_j^{\texttt{Idle}}$ is:

$$G_j^{\texttt{Idle}}(\Pi_1,\Pi_2,\Pi_3,C_{nf}) =$$
$$\max_{lex}\begin{cases}(p_j(T-R_j),1,0)+G_{j-1}(\Pi_1-p_j,\Pi_2-p_j,\Pi_3-p_j,C_{nf}) & (10)\\ (p_j(T-R_j-C_j),1,1)+G_{j-1}(\Pi_1-p_j,\Pi_2-p_j,\Pi_3,C_{nf}) & (11)\\ \begin{cases}(-\infty,0,0) & \text{if } C_j+R_j>C_{nf}\\ (p_j(C_{nf}-C_j-R_j),0,0)+G_{j-1}(\Pi_1-p_j,\Pi_2,\Pi_3,C_{nf}) & \text{otherwise}\end{cases} & (12)\\ (p_j(T-C_{nf}-R_j),1,0)+G_{j-1}(\Pi_1,\Pi_2-p_j,\Pi_3-p_j,C_{nf}) & (13)\\ (p_j(T-C_{nf}-R_j-C_j),1,1)+G_{j-1}(\Pi_1,\Pi_2-p_j,\Pi_3,C_{nf}) & (14)\\ (0,0,0)+G_{j-1}(\Pi_1,\Pi_2,\Pi_3,C_{nf}) & (15)\end{cases}$$

Finally, the dynamic programming algorithm is initialized as follows:

- $G_j(\Pi_1,\Pi_2,\Pi_3,C_{nf}) = (-\infty,0,0)$ if $\Pi_1<0$, $\Pi_2<0$, or $\Pi_3<0$; this corresponds to cases where we do not have enough processors in one of the phases;
- $G_0(\Pi_1,\Pi_2,\Pi_3,C_{nf}) = (0,0,0)$ if $\Pi_1\geq 0$, $\Pi_2\geq 0$, and $\Pi_3\geq 0$; this corresponds to cases where we have 0 jobs to be scheduled; hence, a gain of 0.

Finally, in order to maximize the goodput within the section, we try all possible values of C_{nf}, knowing that C_{nf} must be equal to the checkpointing time of a job running but not checkpointed during phase 0.

$$\max_{lex\atop C_{nf}\in\{C_k\,|\,J_k\in\texttt{Run}^{(0)}\}}\left(G_m\left(P_i-\sum_{J_j\in\texttt{Run}^{(0)}}p_j,P_i,\max(P_i-\delta,P_{\min}),C_{nf}\right)\right)$$

The time complexity is in $O(m^2P_{\max}^3)$ and the spatial complexity is in $O(mP_{\max}^3)$.

D.2 Equations for DPGoodput in two parts

From this section and onwards, all dynamic programming algorithms will be in two parts, as explained in Section 4.2.3 of the article. Gain function G' (respectively H) will take as its value a couple where the first element always corresponds to that of our objective function and the second element is the number of jobs active during phase 2 (respectively, the number of checkpointed jobs during phase 3). Again, we apply a lexicographical order to these couples.

Gain function G'.

$$G'_j(\Pi_1,\Pi_2) = \begin{cases} G_j^{'\texttt{Run}}(\Pi_1,\Pi_2) & \text{if } J_j \in \texttt{Run}^{(0)} \\ G_j^{'\texttt{Ckpt}}(\Pi_1,\Pi_2) & \text{if } J_j \in \texttt{Ckpt}^{(0)} \\ G_j^{'\texttt{Idle}}(\Pi_1,\Pi_2) & \text{if } J_j \in \texttt{Idle}^{(0)} \end{cases}$$

$G_j^{'\texttt{Run}}(\Pi_1,\Pi_2) =$

$$\max_{lex}\begin{cases} (p_j(T-C_{\max}),1) + G'_{j-1}(\Pi_1,\Pi_2-p_j) & (1) \\ \begin{cases} (-\infty,0) & \text{if } C_j > C_{\max} \\ (p_j(C_{\max}-C_j),0) + G'_{j-1}(\Pi_1,\Pi_2) & \text{otherwise} \end{cases} & (2) \end{cases}$$

$G_j^{'\texttt{Ckpt}}(\Pi_1,\Pi_2) =$

$$\max_{lex}\begin{cases} (p_j(T-C_{\max}),1) + G'_{j-1}(\Pi_1-p_j,\Pi_2-p_j) & (3) \\ \begin{cases} (-\infty,0) & \text{if } C_j > C_{\max} \\ (p_j(C_{\max}-C_j),0) + G'_{j-1}(\Pi_1-p_j,\Pi_2) & \text{otherwise} \end{cases} & (4) \\ (p_j(T-2C_{\max}-R_j),1) + G'_{j-1}(\Pi_1,\Pi_2-p_j) & (5) \\ (0,0) + G'_{j-1}(\Pi_1,\Pi_2) & (6) \end{cases}$$

$G_j^{'\texttt{Idle}}(\Pi_1,\Pi_2) =$

$$\max_{lex}\begin{cases} (p_j(T-C_{\max}-R_j),1) + G'_{j-1}(\Pi_1-p_j,\Pi_2-p_j) & (7) \\ \begin{cases} (-\infty,0) & \text{if } C_j + R_j > C_{\max} \\ (p_j(C_{\max}-C_j-R_j),0) + G'_{j-1}(\Pi_1-p_j,\Pi_2) & \text{otherwise} \end{cases} & (8) \\ (p_j(T-2C_{\max}-R_j),1)G'_{j-1}(\Pi_1,\Pi_2-p_j) & (9) \\ (0,0) + G'_{j-1}(\Pi_1,\Pi_2) & (10) \end{cases}$$

The initializations are:

- $G'_j(\Pi_1,\Pi_2) = (-\infty,0)$ if $\Pi_2 < 0$ or $\Pi_1 < 0$
- $G'_0(\Pi_1,\Pi_2) = (0,0)$ if $\Pi_2 \geq 0$, $\Pi_1 \geq 0$

The solution for section i is determined by:

$$G'_m\left(P_i - \sum_{J_j \in \texttt{Run}^{(0)}} p_j, P_i\right).$$

The time and space complexity are in $O(mP_{\max}^2)$.

Gain function H.

$$H_j(\Pi_3) = \begin{cases} H_{j-1}(\Pi_3) & \text{if } J_j \notin Ja \\ \max_{lex}\begin{cases} H_{j-1}(\Pi_3-p_j) + (p_jC_{\max},0) \\ H_{j-1}(\Pi_3) + (p_j(C_{\max}-C_j),1) \end{cases} & \text{otherwise} \end{cases}$$

The initializations are:

- $H_j(\Pi_3) = (-\infty,0)$ if $\Pi_3 < 0$

- $H_0(\Pi_3) = (0, 0)$ if $\Pi_3 \geq 0$

The solution for section i is determined by:

$$H_m(\max(P_i - \delta, P_{\min})).$$

The time and space complexity are in $O(mP_{\max})$.

D.3 Equations for DPYield

Gain function G'.

The main difference with the dynamic program for goodput optimization is that the gain function is now looking for the minimum yield. Rather than computing the yield, we actually consider the time spent by each job to do useful work, since the minimum yield is defined by the job that has spent the less time working in the case of infinite job. As a reminder, the total time that job j has been doing useful work up to time T_i is denoted by $l_j(T_i)$. Here, for the sake of readability, we write it l_j. Also, let $l_{\min} = \min_{j\leq m}(l_j(T_i))$ be the minimum amount of useful work done by a job up to the beginning of section i. The dynamic program computes (according to each considered case), the minimum over all jobs of the amount of useful work done by a job at the end of the section. The aim is then to maximize this minimum. However, it can happen that the minimum yield cannot evolve during a section (this is the case, for instance, if the total processor requirement of jobs whose yield is minimum exceeds the total number of available processors). Hence, we are looking to maximize first the platform utilization with jobs whose *Yield* is the current *minYield*, and then we try to do that while using the highest number of processors. So, once again, we have a triplet of values for the gain G, comprising the minimum time spent doing useful work (among all jobs), the number of processors used for jobs whose current yield is the *minYield*, and, finally, the number of processors used. To combine two triplets together, we use the operator $\bigoplus$, which is defined as follows:

$$\forall (a_1, b_1, c_1), (a_2, b_2, c_2), \qquad \bigoplus((a_1, b_1, c_1), (a_2, b_2, c_2)) = (\min(a_1, a_2), b_1 + b_2, c_1 + c_2).$$

Then the equations for the dynamic program for *Yield* optimization are as follows:

$$G'_j(\Pi_1, \Pi_2) = \begin{cases} G_j'^{\texttt{Run}}(\Pi_1, \Pi_2) & \text{if } J_j \in \texttt{Run}^{(0)} \\ G_j'^{\texttt{Ckpt}}(\Pi_1, \Pi_2) & \text{if } J_j \in \texttt{Ckpt}^{(0)} \\ G_j'^{\texttt{Idle}}(\Pi_1, \Pi_2) & \text{if } J_j \in \texttt{Idle}^{(0)} \end{cases}$$

$$G_j'^{\texttt{Run}}(\Pi_1, \Pi_2) =$$

$$\max_{lex} \begin{cases} \bigoplus\Big((l_j + (T - C_{\max}), p_j \mathbb{1}_{l_j = l_{\min}}, p_j), G'_{j-1}(\Pi_1, \Pi_2 - p_j)\Big) & (1) \\ \begin{cases} (-\infty, 0, 0) & \text{if } C_j > C_{\max} \\ \bigoplus\Big((l_j + (C_{\max} - C_j), 0, 0), G'_{j-1}(\Pi_1, \Pi_2)\Big) & \text{otherwise} \end{cases} & (2) \end{cases}$$

$$G_j^{'\texttt{Ckpt}}(\Pi_1,\Pi_2) =$$

$$\max{}_{lex}\begin{cases} \bigoplus\Big((l_j+(T-C_{\max}),p_j\mathbb{1}_{l_j=l_{\min}},p_j),G'_{j-1}(\Pi_1-p_j,\Pi_2-p_j)\Big) & (3)\\ \begin{cases}(-\infty,0,0) & \text{if } C_j > C_{\max}\\ \bigoplus\Big((l_j+(C_{\max}-C_j),0,0),G'_{j-1}(\Pi_1-p_j,\Pi_2)\Big) & \text{otherwise}\end{cases} & (4)\\ \bigoplus\Big((l_j+(T-2C_{\max}-R_j),p_j\mathbb{1}_{l_j=l_{\min}},p_j),G'_{j-1}(\Pi_1,\Pi_2-p_j)\Big) & (5)\\ \bigoplus\Big((l_j,0,0),G'_{j-1}(\Pi_1,\Pi_2)\Big) & (6)\end{cases}$$

$$G_j^{'\texttt{Idle}}(\Pi_1,\Pi_2) =$$

$$\max{}_{lex}\begin{cases} \bigoplus\Big((l_j+(T-C_{\max}-R_j),p_j\mathbb{1}_{l_j=l_{\min}},p_j),G'_{j-1}(\Pi_1-p_j,\Pi_2-p_j)\Big) & (7)\\ \begin{cases}(-\infty,0,0) & \text{if } C_j+R_j > C_{\max}\\ \bigoplus\Big((l_j+(C_{\max}-C_j-R_j),0,0),G'_{j-1}(\Pi_1-p_j,\Pi_2)\Big) & \text{otherwise}\end{cases} & (8)\\ \bigoplus\Big((l_j+(T-2C_{\max}-R_j),p_j\mathbb{1}_{l_j=l_{\min}},p_j),G'_{j-1}(\Pi_1,\Pi_2-p_j)\Big) & (9)\\ \bigoplus\Big((l_j,0,0),G'_{j-1}(\Pi_1,\Pi_2)\Big) & (10)\end{cases}$$

The initializations are:

- $G'_j(\Pi_1,\Pi_2) = (-\infty,0,0)$ if $\Pi_2 < 0$ or if $\Pi_1 < 0$
- $G'_0(\Pi_1,\Pi_2) = (T_{m+1},0,0)$ if $\Pi_2 \geq 0$ and $\Pi_1 \geq 0$, i.e., we successfully scheduled all jobs and, hence, we set a larger time so that the minimum will be determined by the time spent working by one of the j first jobs.

The solution for section i is determined by:

$$G'_m(P_i - \sum_{J_j\in\texttt{Run}^{(0)}} p_j, P_i).$$

The time and space complexity are in $O(mP_{\max}^2)$.

Gain function H.

Here, we use the same operator $\bigoplus$ but with pairs instead of triplets:

$$\forall (a_1,b_1),(a_2,b_2), \bigoplus((a_1,b_1),(a_2,b_2)) = (\min(a_1,a_2),b_1+b_2).$$

$$H_j(\Pi_3) = \begin{cases} H_{j-1}(\Pi_3) & \text{if } J_j \notin Ja\\ \max{}_{lex}\begin{cases}\bigoplus\Big((l_j(T_{i+1}-C_{\max})+C_{\max},0),H_{j-1}(\Pi_3-p_j)\Big)\\ \bigoplus\Big((l_j(T_{i+1}-C_{\max})+(C_{\max}-C_j),1),H_{j-1}(\Pi_3)\Big)\end{cases} & \text{otherwise}\end{cases}$$

The initializations are:

- $H_j(\Pi_3) = (-\infty,0)$ if $\Pi_3 < 0$
- $H_0(\Pi_3) = (T_{m+1},0)$ if $\Pi_3 \geq 0$

The solution for section i is determined by:

$$H_m(\max(P_i-\delta,P_{\min})).$$

The time and space complexity are in $O(mP_{\max})$.

D.4 Equations for TargetGoodput

For this dynamic program, we use the gain function $G'_j(\Pi_1, \Pi_2, una)$, where una denotes the allowed amount of *UNused Area* (see Section 4.3.2 of the article). una evolves throughout the computation of G'. In cases (2) and (4), we remove p_jC_j since the job is checkpointed and, thus, this induces an *UNused Area* of size p_jC_j. In cases (5), (7) and (9), we remove p_jR_j since there is a recovery. Finally, in case (8), we remove $p_j(C_j + R_j)$ since there is both a checkpoint and a recovery. Let:

$$MW_i = P_i(T_{i+1} - T_i) - (\max(0, P_i - P_{i-1})R_{\min} + \min(\delta, P_i - P_{\min})C_{\min}),$$

with $P_0 = 0$, be the maximum amount of useful work that can be done during section i as presented in Section 3.4 of the article. With a target goodput equal to tgp, the initial value of una for section i is computed recursively with:

$$\begin{cases} una_1 = (1 - tgp)MW_1 \\ una_i = (1 - tgp)MW_i + una_{i-1} - (1 - RG_{i-1})MW_{i-1} \end{cases}$$

where RG_i (*Relative Goodput*) is the amount of useful work done during section i divided by MW_i. In other words, for the first section, to reach a target goodput no smaller than tgp, we should not have more than $(1 - tgp)MW_1$ of unused area. For the next sections, we account for the non-used area that might have been consumed in the previous section, hence, accounting for the surplus $(una_{i-1} < (1 - RG_{i-1})MW_{i-1})$ or deficit $(una_{i-1} > (1 - RG_{i-1})MW_{i-1})$ of unused area during the preceding sections. The equations are then:

$$G'_j(\Pi_1, \Pi_2, una) = \begin{cases} G_j'^{\texttt{Run}}(\Pi_1, \Pi_2, una) & \text{if } J_j \in \texttt{Run}^{(0)} \\ G_j'^{\texttt{Ckpt}}(\Pi_1, \Pi_2, una) & \text{if } J_j \in \texttt{Ckpt}^{(0)} \\ G_j'^{\texttt{Idle}}(\Pi_1, \Pi_2, una) & \text{if } J_j \in \texttt{Idle}^{(0)} \end{cases}$$

$$G_j'^{\texttt{Run}}(\Pi_1, \Pi_2, una) =$$
$$\max_{lex} \begin{cases} \bigoplus \Big((l_j + (T - C_{\max}), p_j \mathbb{1}_{l_j = l_{\min}}, p_j), G'_{j-1}(\Pi_1, \Pi_2 - p_j, una) \Big) & (1) \\ \begin{cases} (-\infty, 0, 0) & \text{if } C_j > C_{\max} \\ \bigoplus \Big((l_j + (C_{\max} - C_j), 0, 0), G'_{j-1}(\Pi_1, \Pi_2, una - p_jC_j) \Big) & \text{otherwise} \end{cases} & (2) \end{cases}$$

$$G_j'^{\texttt{Ckpt}}(\Pi_1, \Pi_2, una) =$$
$$\max_{lex} \begin{cases} \bigoplus \Big((l_j + (T - C_{\max}), p_j \mathbb{1}_{l_j = l_{\min}}, p_j), G'_{j-1}(\Pi_1 - p_j, \Pi_2 - p_j, una) \Big) & (3) \\ \begin{cases} (-\infty, 0, 0) & \text{if } C_j > C_{\max} \\ \bigoplus \Big((l_j + (C_{\max} - C_j), 0, 0), G'_{j-1}(\Pi_1 - p_j, \Pi_2, una - p_jC_j) \Big) & \text{otherwise} \end{cases} & (4) \\ \bigoplus \Big((l_j + (T - 2C_{\max} - R_j), p_j \mathbb{1}_{l_j = l_{\min}}, p_j), G'_{j-1}(\Pi_1, \Pi_2 - p_j, una - p_jR_j) \Big) & (5) \\ \bigoplus \Big((l_j, 0, 0), G'_{j-1}(\Pi_1, \Pi_2, una) \Big) & (6) \end{cases}$$

$$G_j^{'\texttt{Idle}}(\Pi_1, \Pi_2, una) =$$

$$\max_{lex}\begin{cases} \bigoplus \Big((l_j + (T - C_{\max} - R_j), p_j \mathbb{1}_{l_j = l_{\min}}, p_j), G'_{j-1}(\Pi_1 - p_j, \Pi_2 - p_j, una - p_j R_j) \Big) & (7) \\ \begin{cases} (-\infty, 0, 0) & \text{if } C_j + R_j > C_{\max} \\ \bigoplus \Big((l_j + (C_{\max} - C_j - R_j), 0, 0), G'_{j-1}(\Pi_1 - p_j, \Pi_2, una - p_j(C_j + R_j)) \Big) & \text{o/w} \end{cases} & (8) \\ \bigoplus \Big((l_j + (T - 2C_{\max} - R_j), p_j \mathbb{1}_{l_j = l_{\min}}, p_j), G'_{j-1}(\Pi_1, \Pi_2 - p_j, una - p_j R_j) \Big) & (9) \\ \bigoplus \Big((l_j, 0, 0), G'_{j-1}(\Pi_1, \Pi_2, una) \Big) & (10) \end{cases}$$

The initializations are:

- $G'_j(\Pi_1, \Pi_2, una) = (-\infty, 0, 0)$ if $\Pi_2 < 0$ or if $\Pi_1 < 0$ (same as for DPYIELD)
- $G'_0(\Pi_1, \Pi_2, una) = (T_{m+1}, 0, 0)$ if $\Pi_2 \geq 0$ and $\Pi_1 \geq 0$ (same as for DPYIELD)
- $G'_0(\Pi_1, \Pi_2, una) = (-\infty, 0, 0)$ if $una < \Pi_2 T + \Pi_1 C_{nf}$ (we take into account the number of unused processors during phases 1 and 2 and the durations of those phases, and we check that the remaining authorized una is large enough to compensate for these unused processors).

The solution for section i is determined by:

$$G'_m\left(P_i - \sum_{J_j \in \texttt{Run}^{(0)}} p_j, P_i, una_i\right).$$

Since una can take values in the interval $[0, P_{\max}T]$, there is an additional $P_{\max}T$ factor in the time and space complexity of TARGETGOODPUT when compared to those of DPYIELD.

The second part of this algorithm is exactly the same as the second part of DPYIELD, with the same gain function H. Therefore, overall, the time and space complexity of TARGETGOODPUT is $O(mP_{\max}^3 T)$ (as it is still dictated by G').

D.5 Equations for DPBIC

Gain function G'. For the sake of readability, let $Yield_j = Yield_j(T_i)$.

$$G'_j(\Pi_1, \Pi_2) = \begin{cases} G_j^{'\texttt{Run}}(\Pi_1, \Pi_2) & \text{if } J_j \in \texttt{Run}^{(0)} \\ G_j^{'\texttt{Ckpt}}(\Pi_1, \Pi_2) & \text{if } J_j \in \texttt{Ckpt}^{(0)} \\ G_j^{'\texttt{Idle}}(\Pi_1, \Pi_2) & \text{if } J_j \in \texttt{Idle}^{(0)} \end{cases}$$

$$G_j^{'\texttt{Run}}(\Pi_1, \Pi_2) =$$

$$\max_{lex}\begin{cases} (p_j(T - C_{\max})(2 - Yield_j)^Y, 1) + G'_{j-1}(\Pi_1, \Pi_2 - p_j) & (1) \\ \begin{cases} (-\infty, 0) & \text{if } C_j > C_{\max} \\ (p_j(C_{\max} - C_j)(2 - Yield_j)^Y, 0) + G'_{j-1}(\Pi_1, \Pi_2) & \text{otherwise} \end{cases} & (2) \end{cases}$$

$$G_j^{\prime\texttt{Ckpt}}(\Pi_1,\Pi_2) =$$
$$\max_{lex}\begin{cases}(p_j(T-C_{\max})(2-\mathit{Yield}_j)^Y,1)+G'_{j-1}(\Pi_1-p_j,\Pi_2-p_j) & (3)\\ \begin{cases}(-\infty,0) & \text{if } C_j > C_{\max}\\ (p_j(C_{\max}-C_j)(2-\mathit{Yield}_j)^Y,0)+G'_{j-1}(\Pi_1-p_j,\Pi_2) & \text{otherwise}\end{cases} & (4)\\ (p_j(T-2C_{\max}-R_j)(2-\mathit{Yield}_j)^Y,1)+G'_{j-1}(\Pi_1,\Pi_2-p_j) & (5)\\ (0,0)+G'_{j-1}(\Pi_1,\Pi_2) & (6)\end{cases}$$

$$G_j^{\prime\texttt{Idle}}(\Pi_1,\Pi_2) =$$
$$\max_{lex}\begin{cases}(p_j(T-C_{\max}-R_j)(2-\mathit{Yield}_j)^Y,1)+G'_{j-1}(\Pi_1-p_j,\Pi_2-p_j) & (7)\\ \begin{cases}(-\infty,0) & \text{if } C_j+R_j > C_{\max}\\ (p_j(C_{\max}-C_j-R_j)(2-\mathit{Yield}_j)^Y,0)+G'_{j-1}(\Pi_1-p_j,\Pi_2) & \text{otherwise}\end{cases} & (8)\\ (p_j(T-2C_{\max}-R_j)(2-\mathit{Yield}_j)^Y,1)G'_{j-1}(\Pi_1,\Pi_2-p_j) & (9)\\ (0,0)+G'_{j-1}(\Pi_1,\Pi_2) & (10)\end{cases}$$

The initializations are:

- $G'_j(\Pi_1,\Pi_2) = (-\infty,0)$ if $\Pi_2 < 0$ or $\Pi_1 < 0$
- $G'_0(\Pi_1,\Pi_2) = (0,0)$ if $\Pi_2 \geq 0$, $\Pi_1 \geq 0$

The solution for section i is determined by:

$$G'_m\left(P_i - \sum_{J_j\in\texttt{Run}^{(0)}} p_j, P_i\right).$$

The time and space complexity are in $O(mP_{\max}^2)$.

Gain function H.

$$H_j(\Pi_3) = \begin{cases} H_{j-1}(\Pi_3) & \text{if } J_j \notin Ja\\ \max_{lex}\begin{cases}(p_jC_{\max}(2-\mathit{Yield}_j(T_{i+1}-C_{\max}))^Y,0)+H_{j-1}(\Pi_3-p_j)\\ (p_j(C_{\max}-C_j)(2-\mathit{Yield}_j(T_{i+1}-C_{\max}))^Y,1)+H_{j-1}(\Pi_3)\end{cases} & \text{otherwise}\end{cases}$$

The initializations are:

- $H_j(\Pi_3) = (-\infty,0)$ if $\Pi_3 < 0$
- $H_0(\Pi_3) = (0,0)$ if $\Pi_3 \geq 0$

The solution for section i is determined by:

$$H_m(\max(P_i-\delta,P_{\min})).$$

The time and space complexity are in $O(mP_{\max})$.

E Simulation results

In this section, we present numerous figures to complement the results presented in Section 6 of the main article. We first compare the DPBIC and DPBIC–BEST-C_{nf} algorithms in Section E.1. Then, we present detailed versions of the figures included in Sections 6.2.2 and 6.2.3: we present separate figures for different traces in Sections E.2 and E.3. In Section E.4, we compare the computation of upper bounds of the maximum minimum yield presented in Sections 3.4 and 5.1 of the main article. Next, we present in Section E.5 the traces used for the finite jobs along with the techniques used to extract data. Finally, we study in Section E.6 the impact of parameters by comparing versions with infinite and finite jobs, as we did in Section 6.3.1 of the main article.

All figures are produced using the same methodology as described in Section 6.1 of the main article for infinite jobs, and described in Section 6.3 of the main article for finite jobs.

E.1 Impact of the optimization of phase 1 for DPBIC

In this section, we compare the algorithm DPBIC($\mathcal{X}$) to the algorithm DPBIC–BEST-C_{nf}($\mathcal{X}$) with the parameter $\mathcal{X} \in \{2, 5, 7, 10, 15, 25, 45\}$, in order to study the impact of the optimization of the length of phase 1.

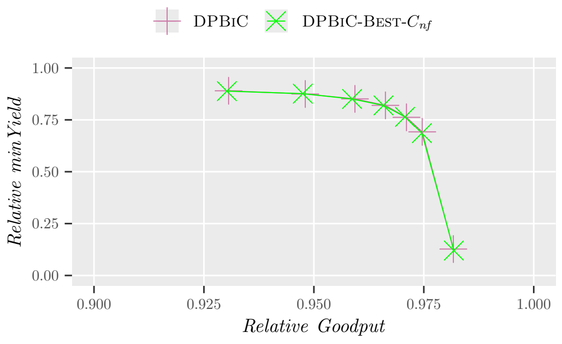


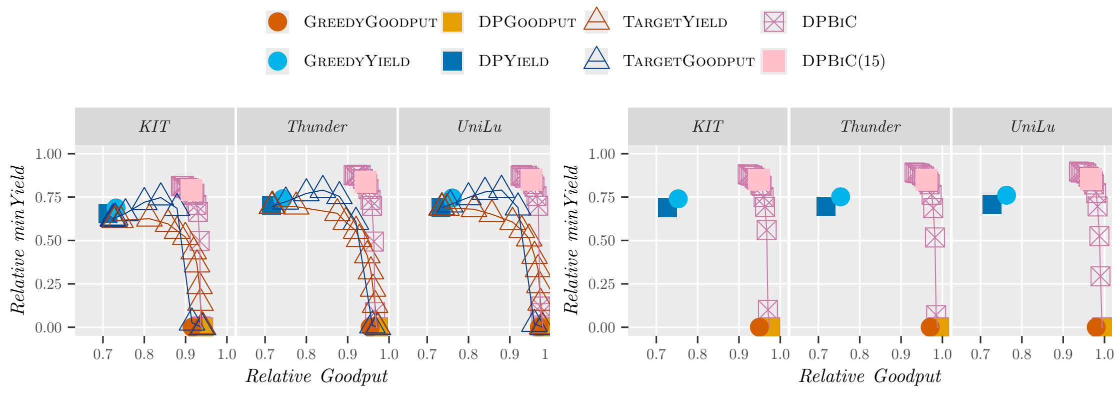
Figure 17: Pareto comparison of the two DPBIC variants.

Figure 17 presents a Pareto comparison with both objective functions. We can see that all the dots are extremely close, as was the case in Figure 6 of the main article. Therefore, in the following, we will only show the versions with a fixed-length first phase, that is, DPBIC.

E.2 Pareto comparison

In this section, we present the same Pareto comparisons than presented in Section 6.2.2 of the main article, except that we separate the results of the different traces.

Figure 18 presents two configurations of three Pareto comparisons. We can see approximately the same dots or at least the same disposition of curves in relation to each other, in the subfigures corresponding to the same parameter settings. Therefore, the origin of the jobs has no visible impact on the conclusions taken from the simulations.



(a) *Light simulation*. (b) *Default simulation*.

Figure 18: Pareto comparison of the performance of the different algorithms in *light simulation* (left) and in *default simulation* (right).

Mean	*Min*	*First decile*	*First quartile*	*Mediane*	*Last quartile*	*Last decile*	*Max*
$2.09e-04$	$3e-06$	$5.8e-05$	$1.45e-04$	$2.86e-04$	$3.85e-04$	$4.89e-04$	$7.42e-04$

Table 1: Statistics on a set of 330 instances on the difference between the bounds derived by the two methods to derive an upper bound on the maximum *minYield*.

E.3 Impact of parameter variations for the different traces, with infinite jobs

In this section, we present the same variations of each parameter as described in Section 6.2.3 of the main article, but we separate the performance achieved for each trace.

Figure 19, 20, 21, 22 and 23 present the performance as a function of the parameter variation for each of the traces and then the mean performance over all traces. Once again, there is no significant difference between the different traces in the position of dots and even less in the relative position of curves.

E.4 Comparison of the two upper bounds of the maximum *minYield*

In this section, we compare the values of the two upper bounds for maximum *minYield* derived in Sections 3.4 and 5.1 of the main article. The first method is simpler. However, it cannot be applied to finite duration jobs, whereas the second method can. We compare the results achieved by the two computation methods with various parameters in the infinite setting.

In Table 1, we report statistics on the difference of the bounds obtained by the two methods to compute an upper bound of the maximum *minYield*. These statistics are obtained on a set of 330 instances obtained by varying the value of each of the parameters. We can see that the maximum difference observed is very small (with respect to the value of the bound itself, which varies between 0.1 and 0.9) and that, therefore, both bounds can be used for this purpose in the case of infinite jobs. In the case of finite jobs, only the second method can be used. However, as observed and explained in Section 6.3.1 of the main article, the *minYield* is generally close to 0 in the case of finite jobs and, hence, this calculation has not been used.

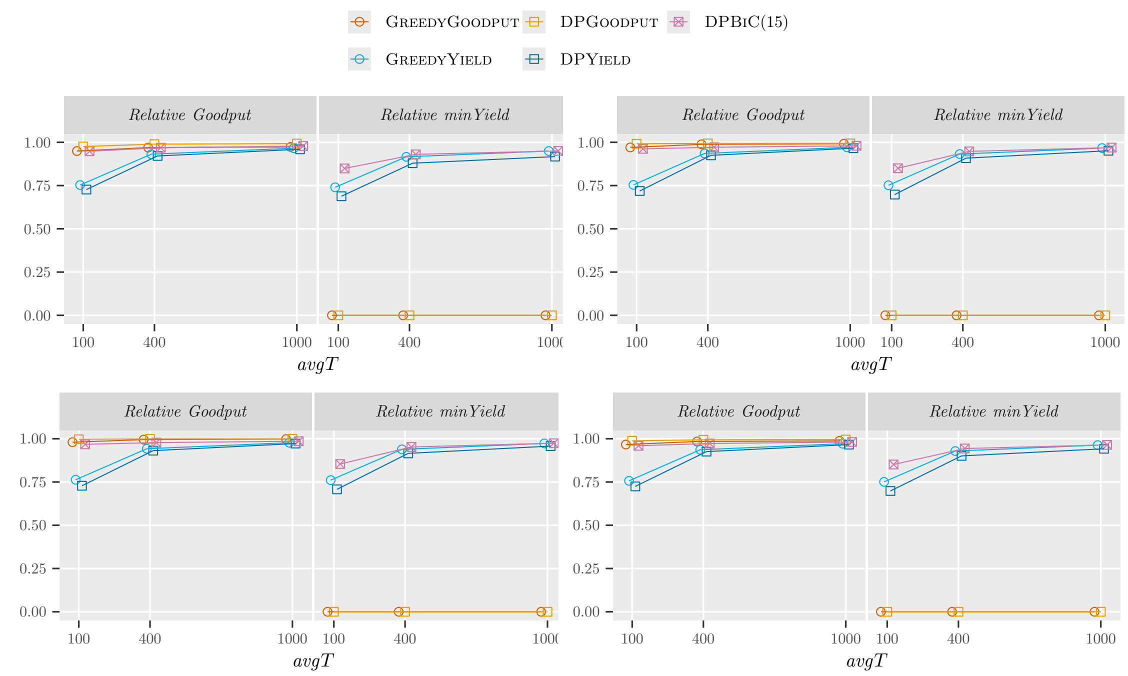


Figure 19: Variation of *avgT* in trace *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left) and mean of all traces (bottom right), infinite.

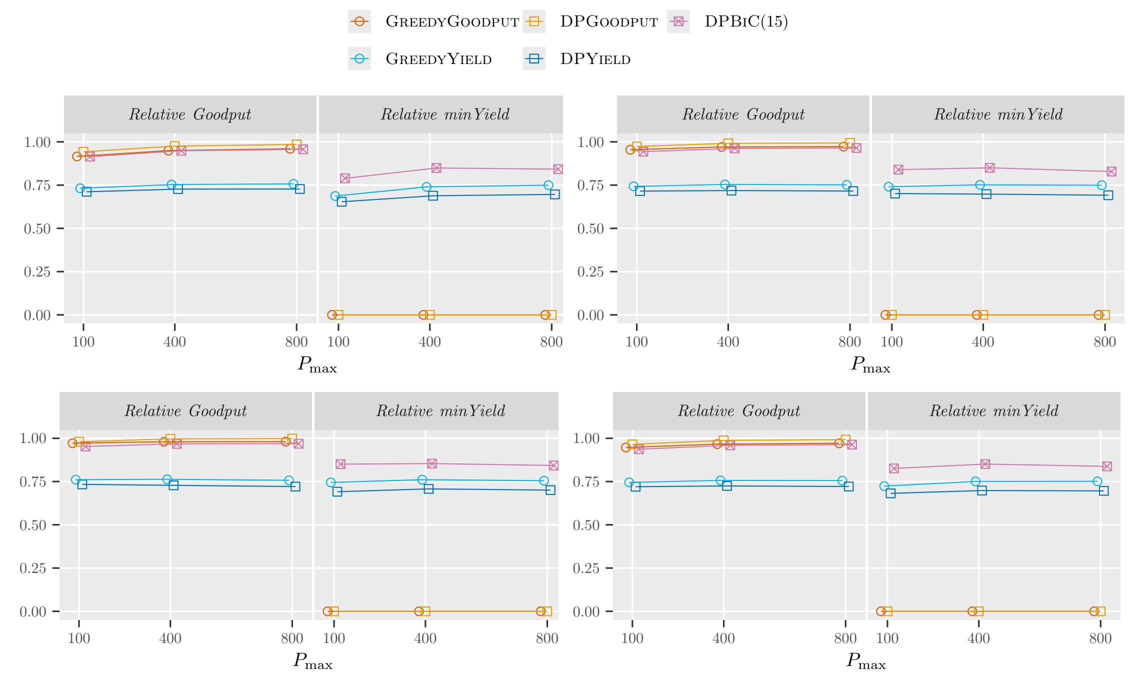


Figure 20: Variation of $P_{\max}$ in trace *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left) and mean of all traces (bottom right), infinite.

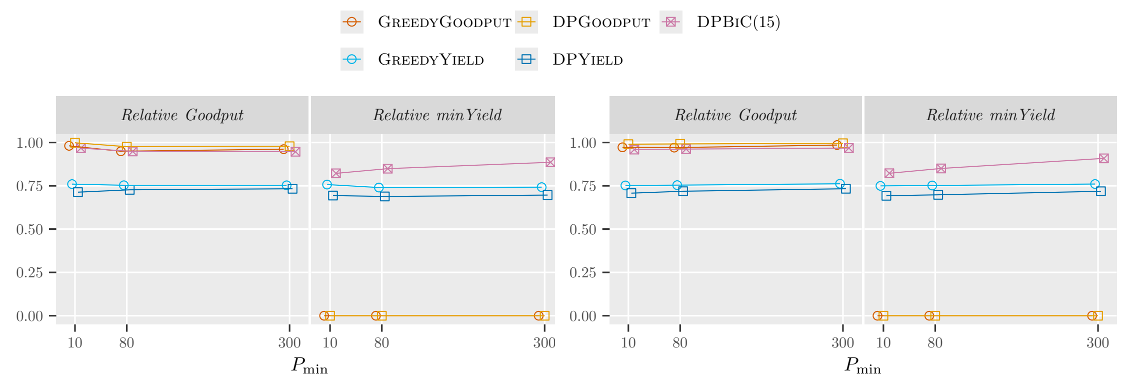


Figure 21: Variation of $P_{\min}$ in trace *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left) and mean of all traces (bottom right), infinite.

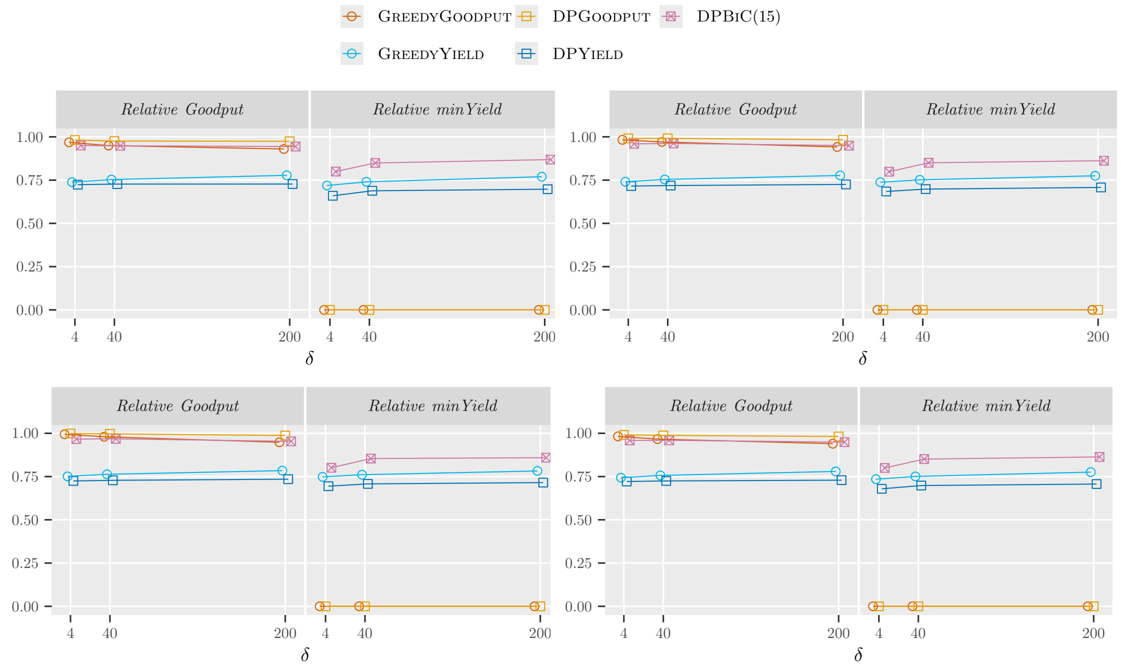


Figure 22: Variation of δ in trace *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left) and mean of all traces (bottom right), infinite.

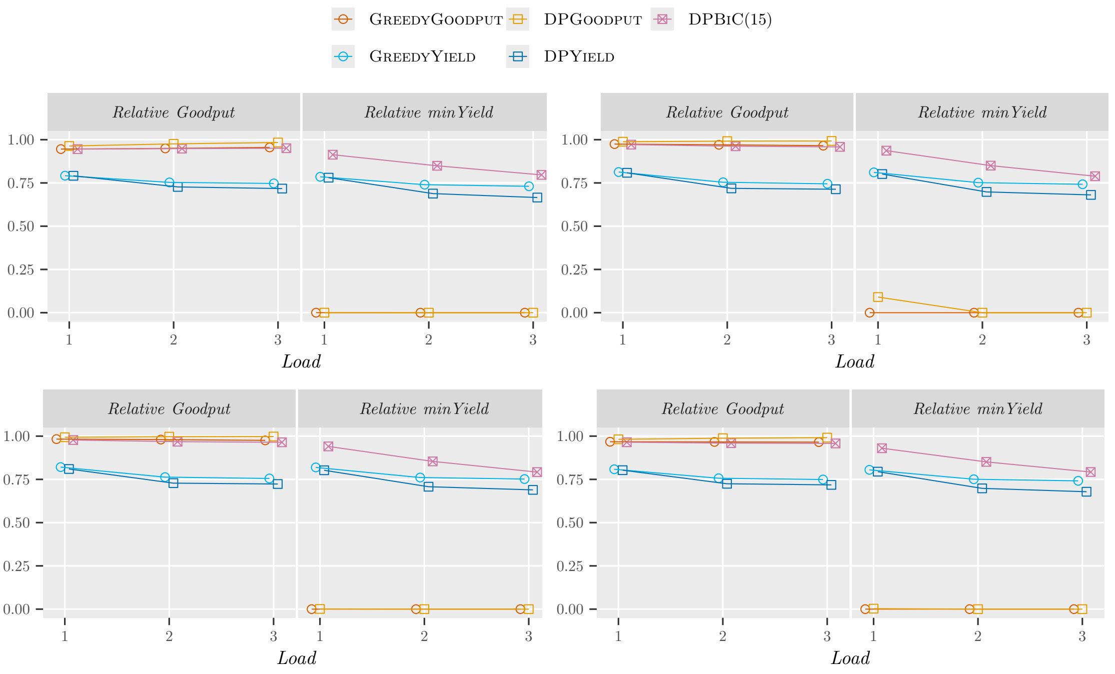


Figure 23: Variation of *Load* in trace *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left) and mean of all traces (bottom right), infinite.

E.5 Workloads for finite duration jobs

In this section, we present how we generate instances for the case with finite duration jobs. First, we extract jobs from the same traces than previously. Then, we define different methods for generating the required release dates of jobs. Finally, based on some observations, we choose which method to use for the rest of the simulations.

E.5.1 Traces

We use the same traces than for infinite jobs to define the execution length and number of processors of jobs.

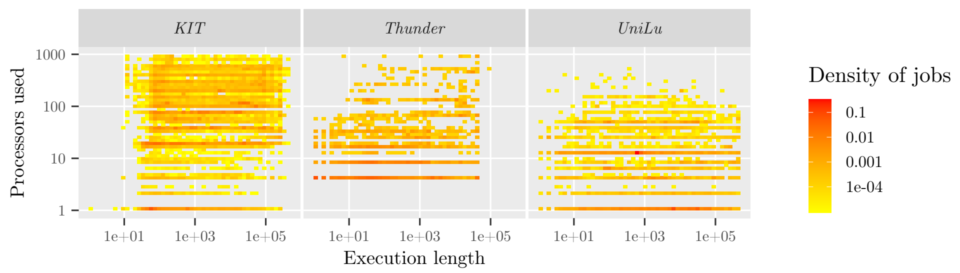


Figure 24: Heatmap of execution length in relation to processor usage of jobs for each of the three traces.

Figure 24 is the heatmap of job execution length in relation to the number of processors used, for each trace, with logarithmic scales. We can see that in *KIT*, most jobs take more than 20 seconds (which is the time unit used) and up to more than 24 hours (whereas the complete size of a simulation corresponds on average to just over 11 hours), with jobs requiring from 1 to 1,000 processors. In *Thunder*, there are many very short jobs using 4 processors (and none using fewer processors) and relatively few jobs using more than a hundred processors. Finally, in *UniLu*, most jobs last more than 5 minutes and use more than a dozen processors, and almost none use more than 100 processors. So the three traces have very distinct profiles.

E.5.2 Generation of release dates in function of a target load

We define the load of the system at a given time as the total number of processors required by the alive jobs (that is, those released and not yet completed) divided by the number of available processors. We want to define the release dates of jobs in such a way as to maintain a stable load over time, regardless of the algorithm in use. However, since different algorithms do not execute the same jobs at the same time, the set of alive jobs in a given section depends on the algorithm used which, in turn, impacts the load.

In addition, the release dates of jobs depend on the target load factor and of the method used to maintain this target load. We have experimented with three methods for generating the release dates of jobs, using the following general logic: at each section i, we have the amount of potential useful work that can be done during the section, MW_i (as previously defined in Section 3.4 of the main article), and we want to update the work done by the set of yet-uncompleted jobs in order to estimate which jobs may be completed during the section and, therefore, need to be replaced by new jobs in order to maintain the desired load:

- Accumulation: In each section i, we compute ω_i, the total work that all alive jobs could do if each of them was allocated for the whole section its required number of processors. If $\omega_i < MW_i$, we generate new jobs to be alive during this section and add the work they could do during that section to ω_i, as long as $\omega_i < MW_i$. We then check whether the total number of required processors of alive jobs corresponds to the desired load. If this is not the case, we generate new jobs to be alive during this section (and add their work to ω_i). When moving from one section to the next, we initialize ω_{i+1} with $\omega_i - MW_i$.
- FIFO: For each section, we consider the jobs in their release date order, and we allocate them the maximum possible work in the section as long as the total limit on the work that can be done, MW_i, is not exhausted. If there are not enough jobs to exhaust MW_i, we release new jobs for this section. We then check that the total number of required processors of the alive jobs corresponds to the desired load and, if not, we release new jobs for this section.
- Smallest: Same principle as for FIFO, except that jobs are sorted in non-decreasing order of the amount of remaining work (rather than by release dates) for the work allocation.

Once we have defined the job requirement for each section, we generate release dates by drawing them uniformly in the preceding section.

E.5.3 Evaluation

In this section, we compare the different release date generation methods with the load obtained afterwards with each of the algorithms.

Figure 25 shows the evolution of the number of available processors over time used for this simulation.

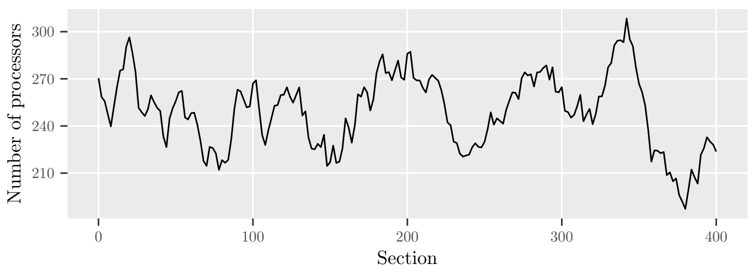


Figure 25: Evolution of the number of available processors.

Figure 26 shows the evolution of the current load, that is the total number of processors required by the jobs in the system divided by the current number of available processors, for each of the three different traces and then aggregated for all the traces. We note that the actual load is almost never smaller than the target load.

Figure 27 shows the evolution of the maximum load, that is the total number of processors required by the jobs in the system divided by the maximum number of available processors, for each of the three different traces and then aggregated for all the traces. We remark that the system appears to work as it should.

From these set of figures, it appears that the FIFO method is the most stable for achieving a target load, even after many sections. This is therefore the method that we have been using in the simulations.

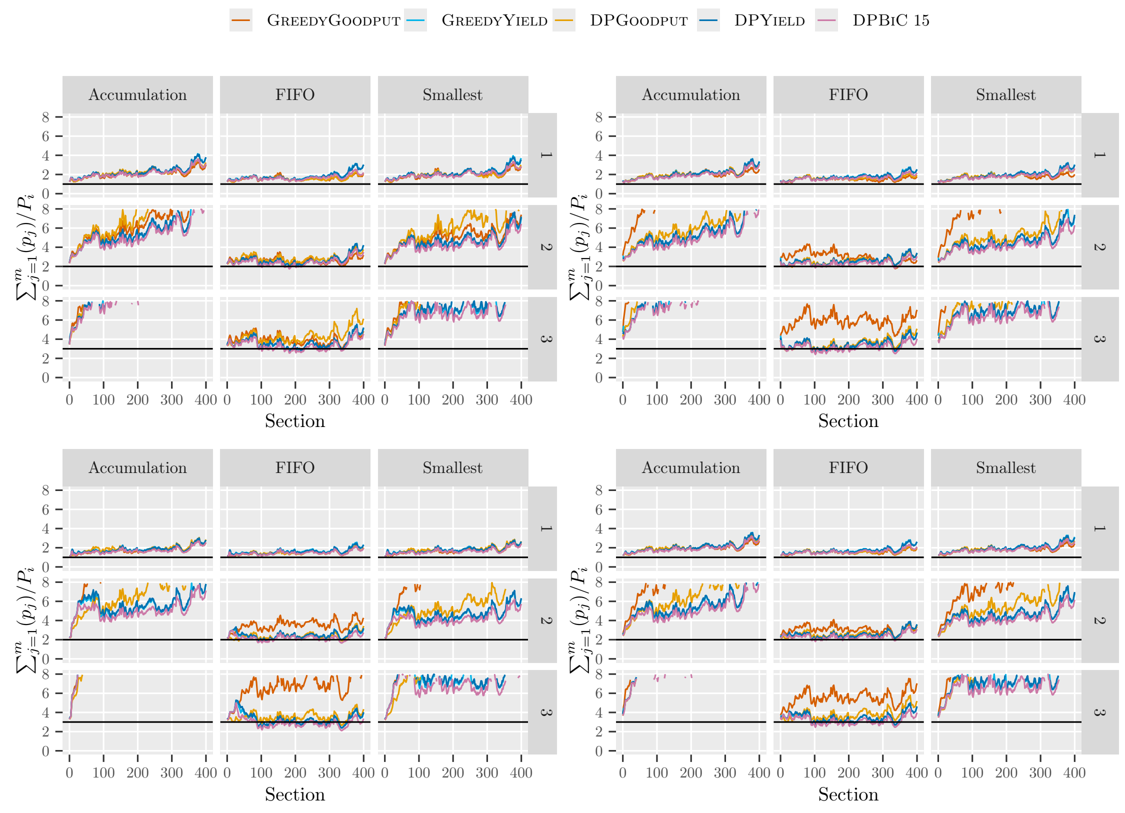


Figure 26: Evolution of the load with time for the traces *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left), and for all three traces (bottom right). Each row of each graph corresponds to a target load, indicated on the right of each row and by the black line.

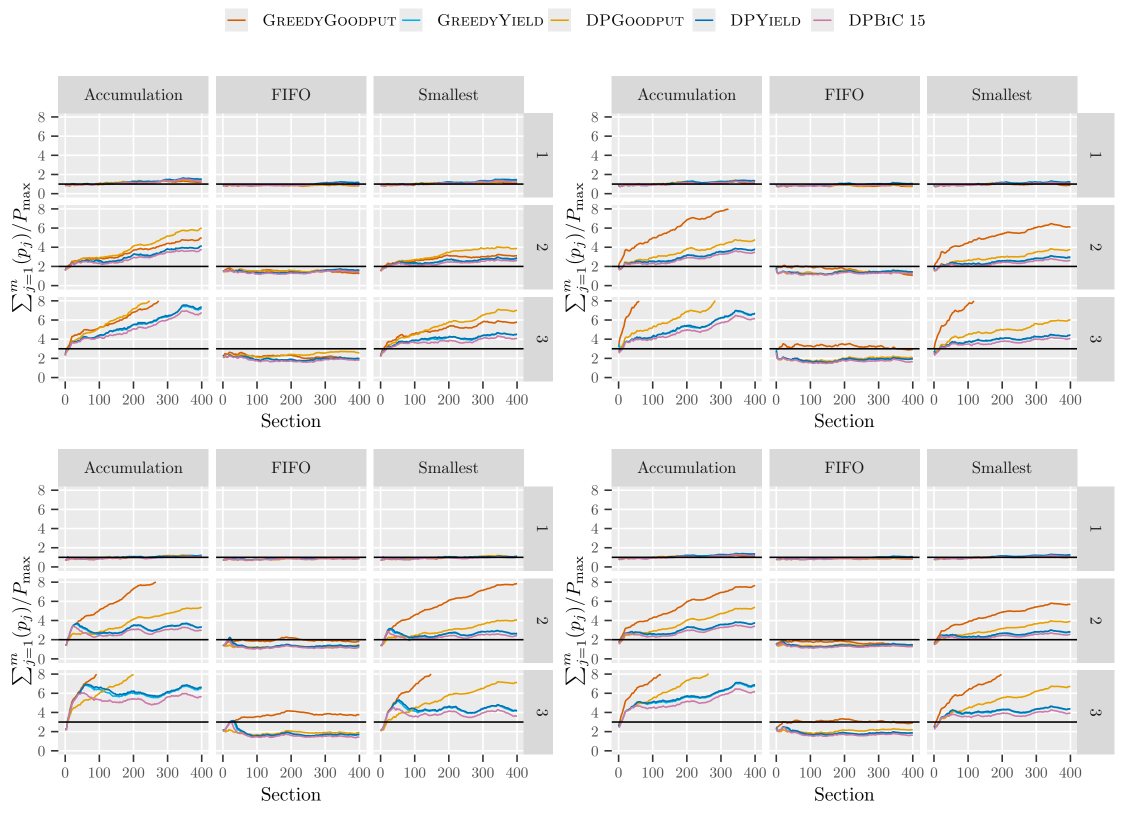


Figure 27: Evolution of the maximum load with time for the traces *KIT* (top left), *Thunder* (top right), *UniLu* (bottom left), and for all three traces (bottom right). Each row of each graph corresponds to a target load, indicated on the right of each row and by the black line.

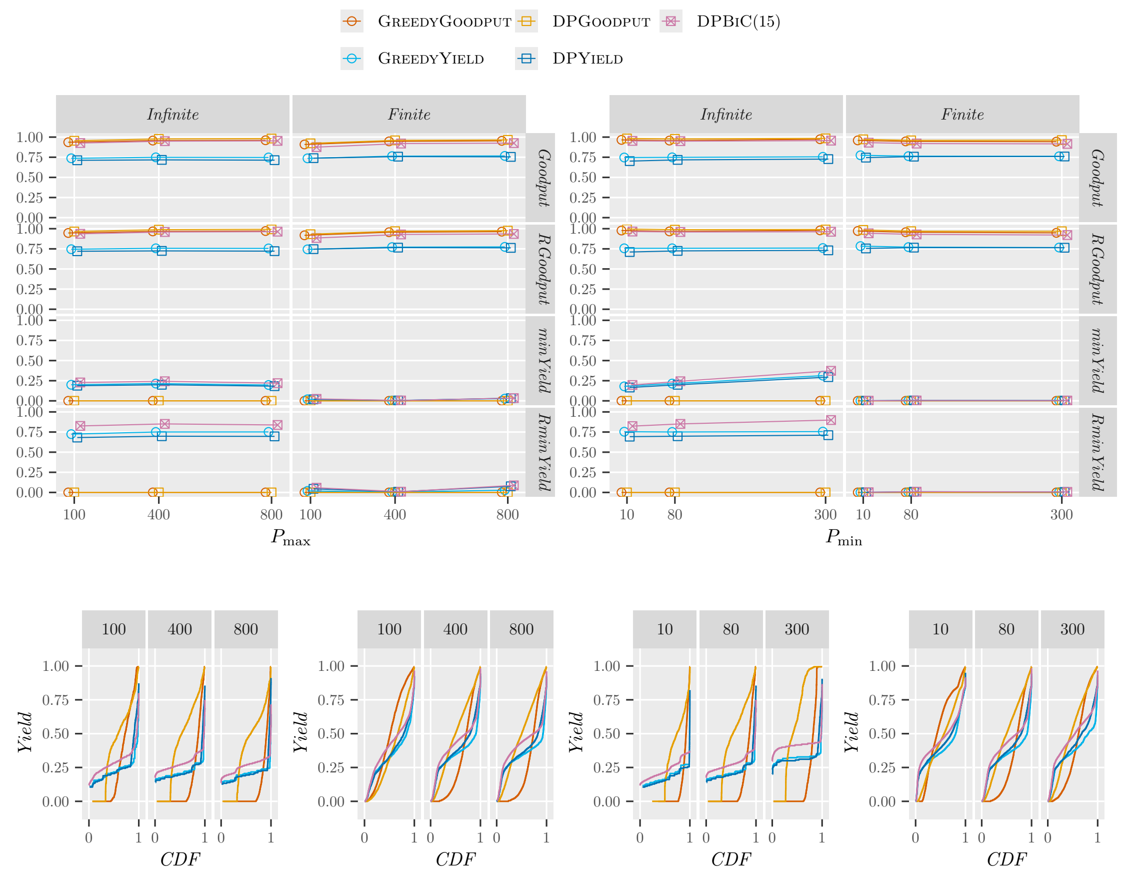


Figure 28: Impact of the variation of $P_{\max}$ (left) and $P_{\min}$ (right) for the comparison between infinite settings (far left and center right) and finite settings (centerleft and far right)

E.6 Comparison of the impact of the variation of parameters for finite and infinite jobs

In this section, we present a comparison between the perfomance achieved with infinite and finite jobs when the different parameters vary, as was done in Section 6.2.3 of the main article, for all main algorithms, and then only for DPBiC($\mathcal{X}$) with the parameter $\mathcal{X} \in \{0, 1, 2, 3, 5, 7, 15, 25, 45\}$.

Figures 28 and 29 present the comparison for the main algorithms. On the one hand, we observe very little differences between the behavior for infinite and finite jobs for algorithms *Goodput* and *RGoodput*. On the other hand, the observation made in Section 6.3.1 of the main article remains valid: the *minYield* obtained with finite jobs is always close to 0. Then, if we examine the cumulative distribution functions (CDF) of the *Yield*, the curves have a similar profile between finite and infinite jobs. DPBiC achieves slightly better performance than *Yield*-oriented algorithms (its curve is always above those of *Yield*-oriented algorithms). One can also remark a greater fairness of the *Yield* for the *Yield*-oriented algorithms (most jobs achieve similar yields), whatever the parameters.

Figures 30, 31, and 32 presents the performance of of DPBiC($\mathcal{X}$) for different values of $\mathcal{X}$.

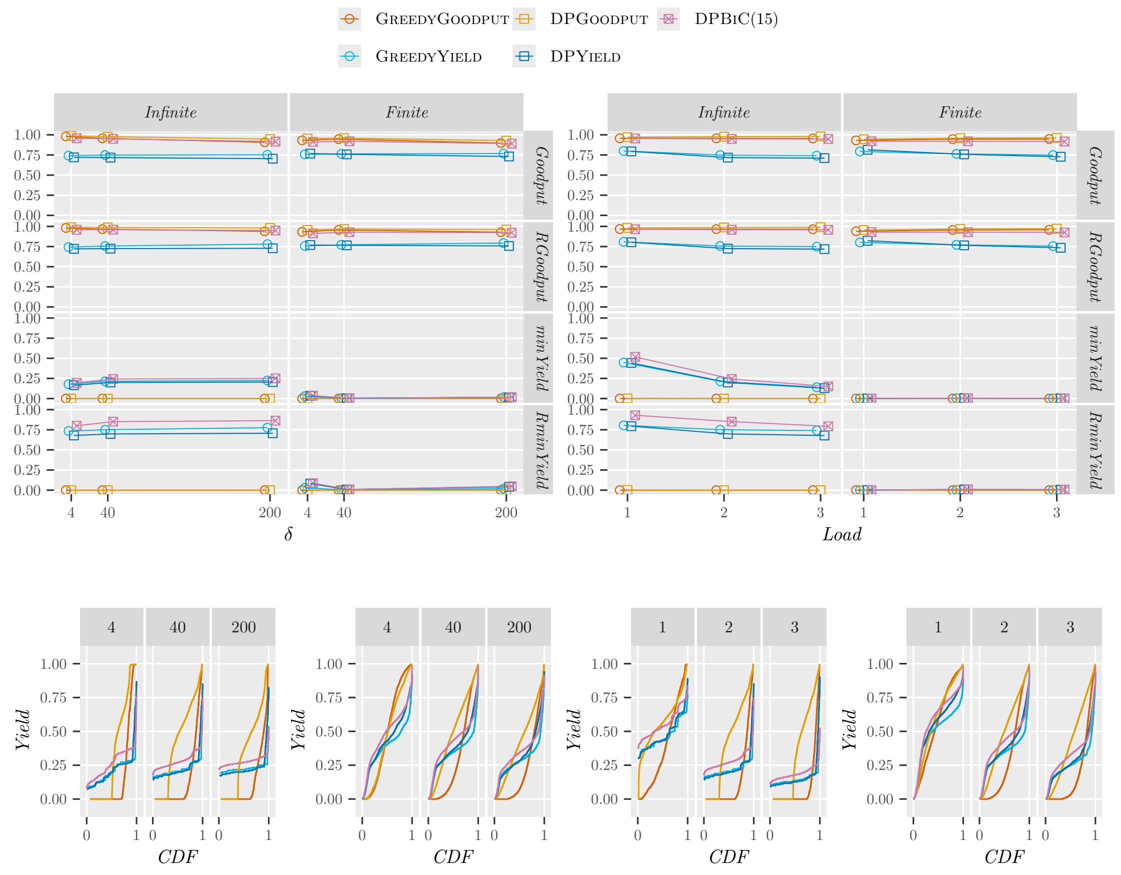
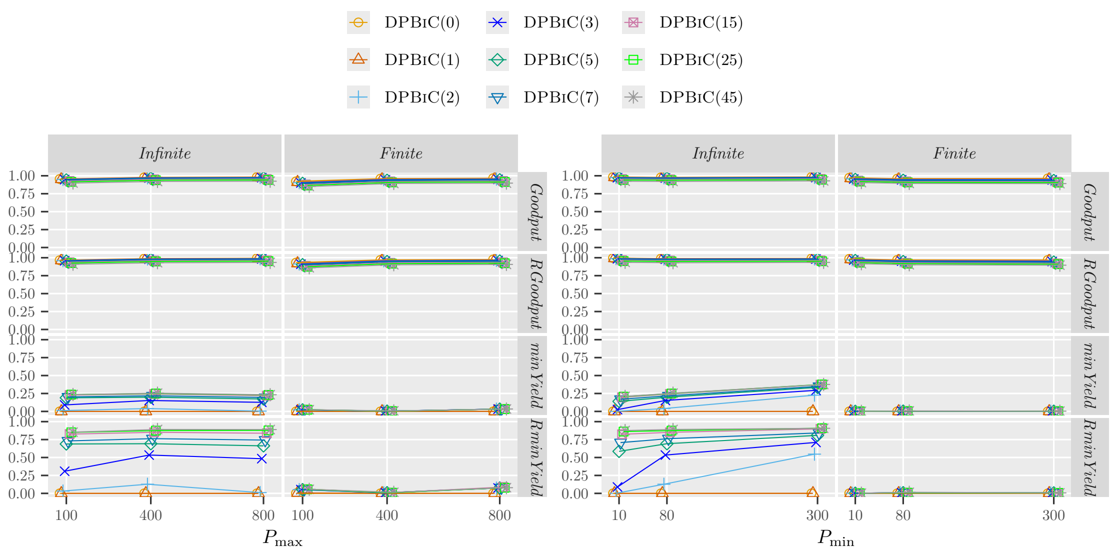


Figure 29: Impact of the variation of δ (left) and *Load* (right) for the comparison between infinite settings (far left and center right) and finite settings (centerleft and far right)



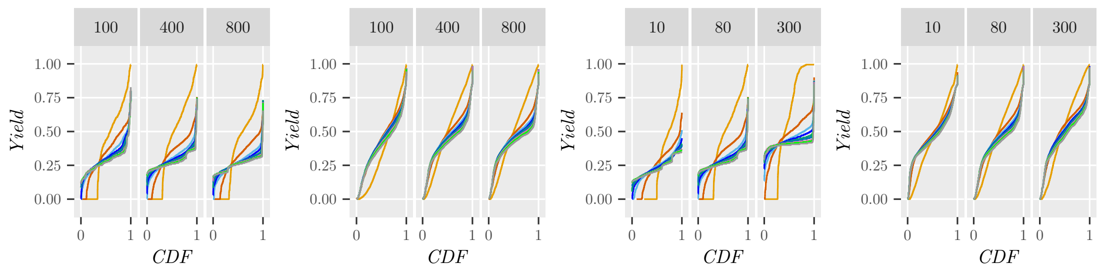


Figure 30: Impact of the variation of $P_{\max}$ (left) and $P_{\min}$ (right) with DPBIC algorithms for the comparison between infinite settings (far left and center right) and finite settings (centerleft and far right)

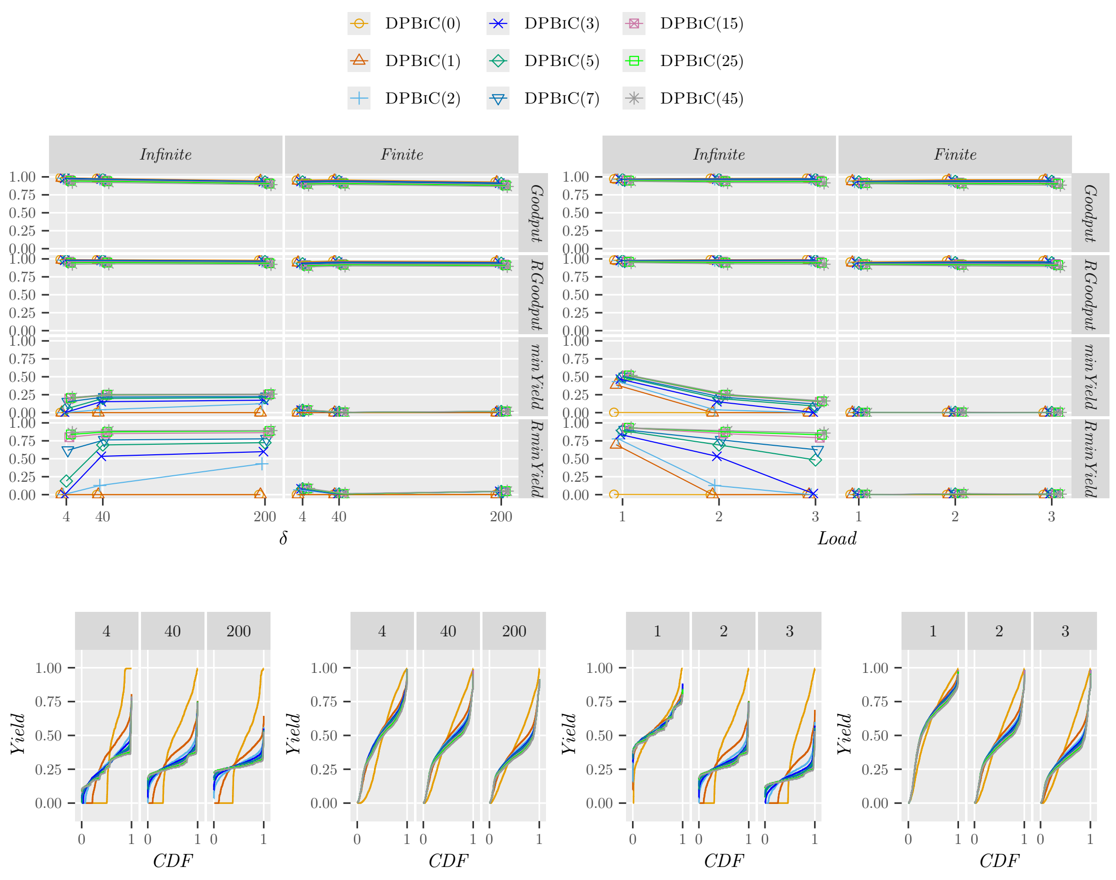


Figure 31: Impact of the variation of δ (left) and *Load* (right) with DPBIC algorithms for the comparison between infinite settings (far left and center right) and finite settings (centerleft and far right)

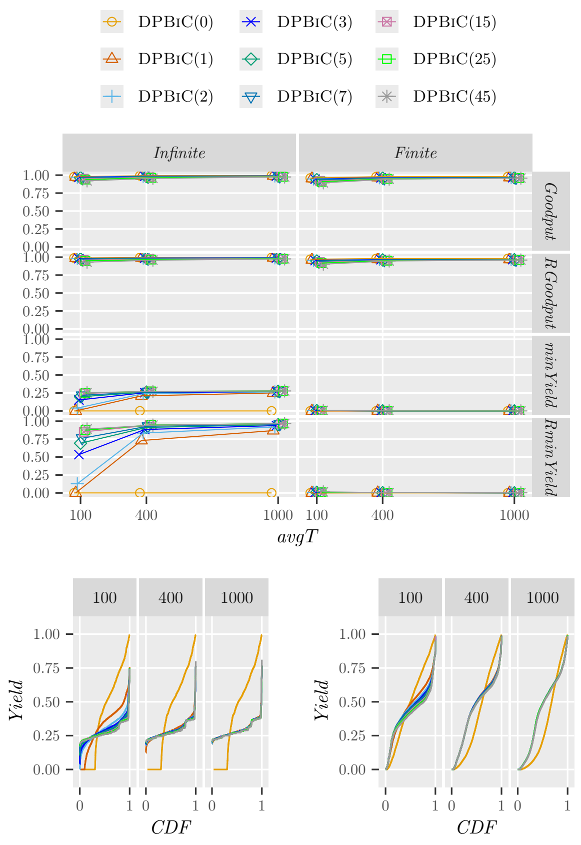


Figure 32: Impact of the variation of *avgT* with DPBɪC algorithms.

In general, as $\mathcal{X}$ increases, the *Goodput* decreases and the *minYield* increases. But in the case of finite jobs, as the *minYield* is always close to 0, the statistic of interest for the *Yield* is once again its cumulative distribution function. One can remark that, in each figure, the curves are initially very similar (except when $\mathcal{X} = 0$) and then the algorithms with a smaller value for $\mathcal{X}$ dominate, as observed in the Section 6.3.2 of the main article. And so, once again, we can conclude that choosing $\mathcal{X} = 2$ could be a good trade-off between *Goodput* and *Yield* for finite duration jobs.

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

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

ISSN 0249-6399