# Profiles of upcoming HPC Applications and their Impact on Reservation Strategies

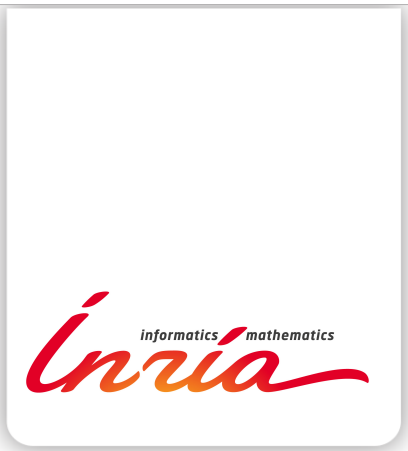
Ana Gainaru, Brice Goglin, Valentin Honoré, Guillaume Pallez


# Profiles of upcoming HPC Applications and their Impact on Reservation Strategies

**Ana Gainaru, Brice Goglin, Valentin Honoré, Guillaume Pallez**

**RESEARCH REPORT**
**N° 9359**
Août 2020
Project-Teams TADaaM

ISSN 0249-6399 ISRN INRIA/RR--9359--FR+ENG




# Profiles of upcoming HPC Applications and their Impact on Reservation Strategies

Ana Gainaru*, Brice Goglin†, Valentin Honoré†, Guillaume Pallez†

Project-Teams TADaaM

Research Report n° 9359 — Août 2020 — 30 pages

**Abstract:** With the expected convergence between HPC, BigData and AI, new applications with different profiles are coming to HPC infrastructures. We aim at better understanding the features and needs of these applications in order to be able to run them efficiently on HPC platforms.
The approach followed is bottom-up: we study thoroughly an emerging application, *Spatially Localized Atlas Network Tiles* (SLANT, originating from the neuroscience community) to understand its behavior. Based on these observations, we derive a generic, yet simple, application model (namely, a linear sequence of stochastic jobs). We expect this model to be representative for a large set of upcoming applications that require the computational power of HPC clusters without fitting the typical behavior of large-scale traditional applications.
In a second step, we show how one can manipulate this generic model in a scheduling framework. Specifically we consider the problem of making reservations (both time and memory) for an execution on an HPC platform. We derive solutions using the model of the first step of this work. We experimentally show the robustness of the model, even with very few data or with another application, to generate the model, and provide performance gains with regards to standard and more recent approaches used in the neuroscience community.

**Key-words:** stochastic application, execution time, memory footprint, scheduling, checkpointing, reservation-based platform, reservation sequence, neuroscience application

* Department of EECS, Vanderbilt University, Nashville, TN, USA
† Inria, LaBRI, Université de Bordeaux, France

**RESEARCH CENTRE**
**BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour
33405 Talence Cedex

# Étude d'applications émergentes en HPC et leurs impacts sur des stratégies d'ordonnancement

**Résumé :**

La convergence entre les domaines du calcul haute-performance, du *BigData* et de l'intelligence artificielle fait émerger de nouveaux profils d'application sur les infrastructures HPC.

Dans ce travail, nous proposons une étude de ces nouvelles applications afin de mieux comprendre leurs caractériques et besoins dans le but d'optimiser leur exécution sur des plateformes HPC.

Pour ce faire, nous adoptons une démarche ascendante. Premièrement, nous étudions en détail une application émergente, SLANT, provenant du domaine des neurosciences. Par un profilage détaillé de l'application, nous exposons ses principales caractéristiques ainsi que ses besoins en terme de ressources de calcul. À partir de ces observations, nous proposons un modèle d'application générique, pour le moment simple, composé d'une séquence linéaire de tâches stochastiques. Ce modèle devrait, selon nous, être adapté à une grande variété de ces applications émergentes qui requièrent la puissance de calcul des clusters HPC sans présenter le comportement typique des applications qui s'exécutent sur des machines à grande-échelle.

Deuxièmement, nous montrons comment utiliser le modèle d'application générique dans le cadre du développement de stratégies d'ordonnancement. Plus précisément, nous nous intéressons à la conception de stratégies de réservations (à la fois en terme de temps de calcul et de mémoire). Nous proposons de telles solutions utilisant le modèle d'application générique exprimé dans la première étape de ce travail. Enfin, nous montrons la robustesse du modèle d'application et de nos stratégies d'ordonnancement au travers d'évaluations expérimentales de nos stratégies. Notamment, nous démontrons que nos solutions surpassent les approches standards de la communauté des neurosciences, même en cas de données partielles ou d'extension à d'autres applications que SLANT.

**Mots-clés :** ordonnancement, coût stochastique, plateformes de calcul, empreinte mémoire, stratégies de réservations, point de sauvegarde, applications de neurosciences

# Contents

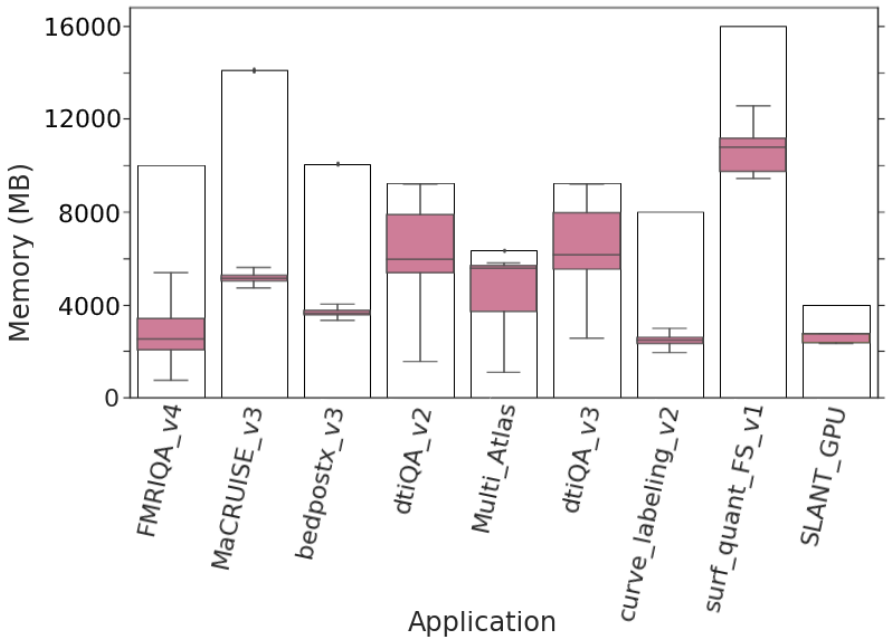



Figure 1: Memory requests during submission and memory usage variation for nine representative medical and neuroscience applications.

# 1 Introduction

High performance computing platforms are amongst the most powerful structures to perform heavy-load critical computations. A typical HPC application is a massively parallel code that requires an important number of computing resources to satisfy its requirement in terms of memory and computation. Fields such as astronomy and cosmology, computational chemistry, earth, particle physics and climate science have evolved together with the advance of platform architecture and software stack in order to leverage massive levels of parallel processing. Newly emerging applications move beyond large monolithic codes that use tightly-coupled, compute-centric algorithms. Fields such as neuroscience, bioinformatics, genome research, computational biology are doing exploratory research that embrace more dynamic, heterogeneous multi-phase workflows using ad-hoc computations and methodologies. New Machine Learning (ML) and AI frameworks have become important tools in exploratory domains. While progresses have been made over past years to improve these ML techniques, this progress has induced high requirements in terms of computations. As instance, Deep Learning techniques require an important training part where the quality of the model increases with the dataset size.

Hence, such workflows involving ML techniques will soon target HPC infrastructures that offer high computation support, as well as high memory and network performance. However, their profiles differ from classic HPC applications. Often, the duration of these applications is difficult to estimate because they are input-independent. It is common for such an application to have walltimes between several hours to days. This characteristic is a real limitation for users for which requesting the maximum possible walltime often induces an overestimation that penalizes the total cost of the request. In addition, the stochastic memory utilization often requires users to request only high memory nodes for their execution.

Figure 1 presents the memory requirements and requests for nine exploratory applications

from the medical and neuroscience department at the Vanderbilt University [31]. The logs are generated for a 6-month period in 2018 running on their in-house cluster. Users often utilize only fractions of the requested memory (e.g. MaCRUISE_v3, bedpostx_v2 in Fig. 1) or end up with their application killed due to memory underestimation (e.g. dtiQA_v2 and dtiQA_v3). Users tend to overestimate their resource requirements in both time and memory, which leads to these application typically waiting in the scheduler queue for days before eventually running.

In this work, we study the profile of an exploratory application from the neuroscience domain with the goal of understanding the properties and characteristics of these new frameworks. We are interested in behavior that is non-biased by intereference due to the system or other applications (e.g. congestion due to shared resources). We focus on the *Spatially Localized Atlas Network Tiles* (SLANT) [25] application. This code follows the typical behavior of the upcoming stochastic applications: 1) its workflow consists of multiple stages and a walltime between tens of minutes to hours depending on hidden characteristics of the input MRI; 2) while its peak memory requirement is predictable,the memory footprint can have variations of tens of GBs within one execution; 3) its code is dynamic, in continuous development depending on the needs of each study. SLANT has an easy to understand workflow whose input data are simply MRI images, which makes it ideal for study, but it is representative for many of this new type of HPC applications. For example the RADICAL-Pilot job system to develop bioinformatics workflows is often used to create workflows that spawn large numbers of short-running processes that can exhibit highly irregular I/O and computation patterns [35]. Similarly, applications using Adaptive Mesh Refinement (AMR) methods have been shown to have high unpredictable performance variations based on characteristics of the input data [47].

Based on our observations of SLANT, we propose a generic application model where an application is described as a chain of tasks whose walltimes follow probability distributions. We use this model to estimate the resource request for SLANT when deployed on an HPC system. We show that our resource estimator needs only a few runs to learn the model and to optimize the submission and execution of these types of applications without any modification to the batch scheduler or HPC middleware. This is essential for productivity focused applications since their codes are in continuous change based on the requirements of each study. Performance prediction methods can be used by scientific applications to adjust their resource requirements during submission. However they tend to work well only on well known codes that can provide a rich history of past runs. Our study aims to bridge the gap between the specific characteristics of exploratory applications and the strict requirements of HPC batch schedulers that hinder productivity and innovation for new computational methods.

The rest of the paper is organized as follows. Section 2 presents the study of the SLANT application and highlights key characteristics of the behavior of each stage in the application. Following these observations, Section 3 derives a new computational model that is used to generate reservation strategies that can be used for deployment on HPC systems. Section 4 presents an extensive study on the impact of the new strategies on application and system level metrics when running on large-scale systems. Finally we present related works in Section 5 and conclude.

## 2 Case study of a Neuroscience Application

In this section, we study thoroughly the performance of an upcoming HPC application from neuroscience: SLANT, introduced in Section 2.1. First we make high-level observations in Section 2.2, then we explain them with lower-level performance analysis in Section 2.3.

## 2.1 Spatially Localized Atlas Network Tiles (SLANT)

The study of this work is centered around a specific representative neuroscience application: SLANT [24, 25]. This application performs multiple independent 3D fully convolutional network (FCN) for high-resolution whole brain segmentation. It takes as input an MRI image obtained by measuring spin–lattice relaxation times of tissues. We use a CPU version of the application whose code is freely available at `https://github.com/MASILab/SLANTbrainSeg`. There exists different version of SLANT depending on whether the network tiles are overlapped or not. Here, we consider the overlapped version (SLANT-27 [25]) in which the target space is covered by $3 \times 3 \times 3 = 27$ 3D FCN. The application is divided into three main phases: i) a preprocessing phase that performs transformations on the target image (MRI is a non-scaled imaging technique) ii) a deep-learning phase iii) a post-processing phase doing label fusion to generate the final application result. Each of the tasks may present run-to-run variations in their walltime.

## 2.2 High-level observations

In recent work [15], observations showed large variations in execution time of neuroscience applications, complicating their execution on HPC platforms. We are interested in verifying and studying this. To do so, we run SLANT on 312 different inputs. These inputs are extracted from OASIS-3 [30][1]and *Dartmouth Raiders Dataset* (DRD)[2] [20] datasets. We run the application on a *Haswell* platform composed of a server with two Intel Xeon E5-2680v3 processors (12 core @ 2,5 GHz). We run the docker image presented in the Git repository of SLANT-27 using the Singularity container runtime.

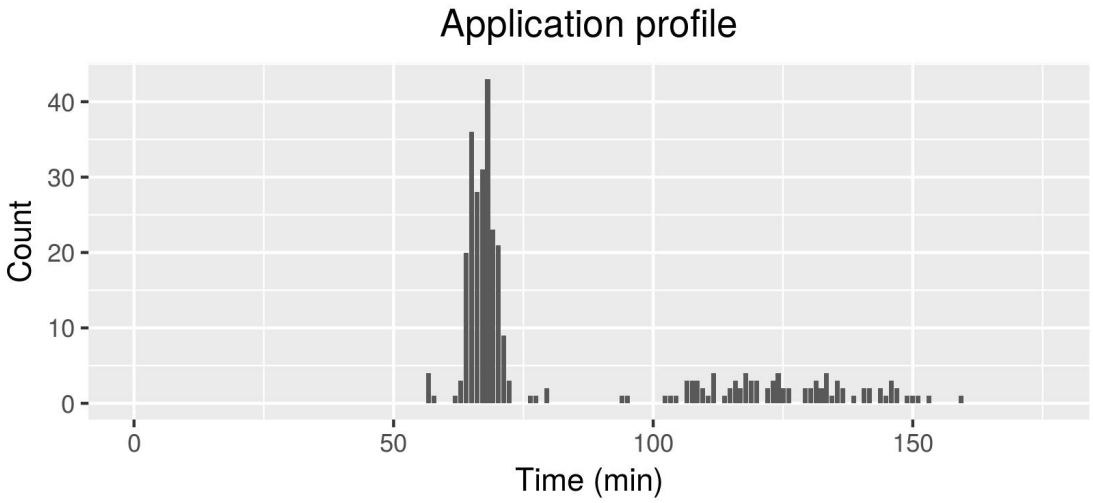



Figure 2: SLANT application walltime variation for various inputs.

In Figure 2, we confirm the observations about the large walltime variations. Specifically we can see two categories of walltimes which correspond to the two datasets: OASIS inputs have a walltime of 70min±15% and DRD inputs have a walltime of 125min±30%. The natural questions that arise are the following:

[1]For this very large dataset, we only used a subset of available data.
[2]Available at `http://datasets-dev.datalad.org/?dir=/labs/haxby/raiders`

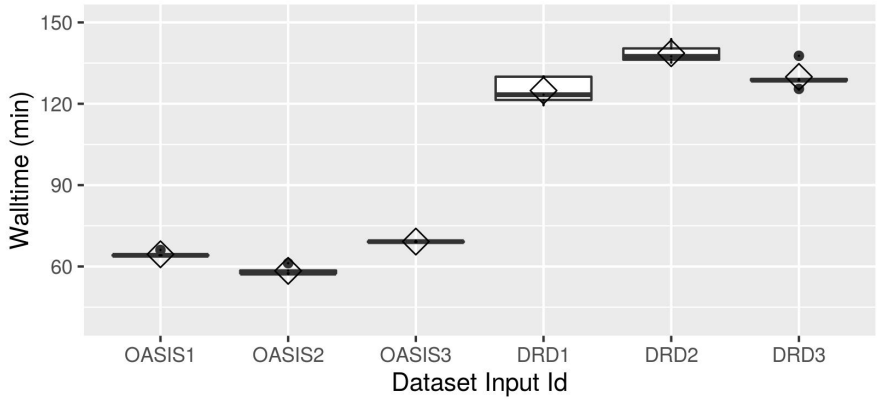



Figure 3: Performance variability on identical inputs. Variability is studied over five runs.

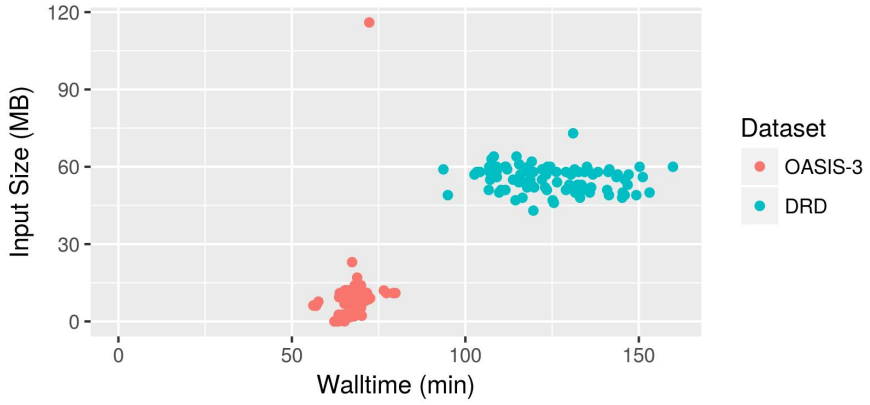
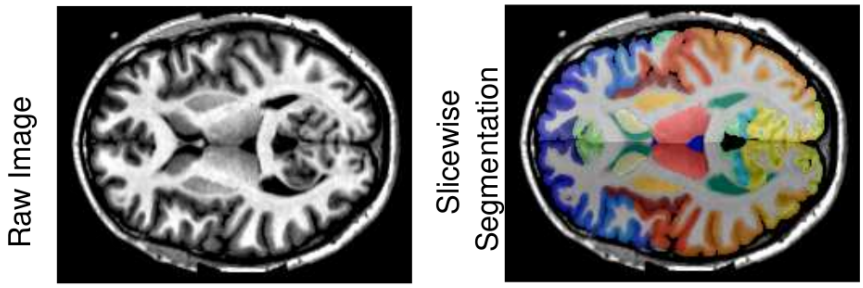



Figure 4: Correlation between the size of the input and the walltime over the 312 runs.

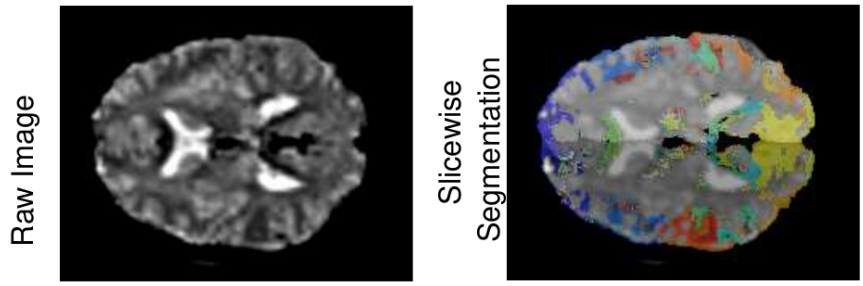
- Is the walltime variation due to a machine artifact (or is it due to the *quality* of the input)?
- Is the walltime variation due to the input size (and can it be predicted using this information)?

We study these questions in the following experiments. First we randomly select three inputs of both datasets and execute them five times each. We present the results in Figure 3. We see that the behavior for each input is quite robust. There are slight variations for DRD inputs, but nothing of the order of magnitude observed over all inputs. Hence, it seems that the duration of the execution is mainly linked to the input.

We then study the variation of walltime as a function of the input size in Figure 4. We can see that for a given dataset, the walltime does not seem correlated to the input size. The corresponding Pearson correlation factors are 0.30 (OASIS) and $-0.15$ (DRD). The datasets however seem to have different input types: except for the outlier at 120 MB, the input sizes of OASIS vary from 0 to 30MB while those from DRD vary from 45 to 75MB. We present visually the type of inputs for the two databases in Figure 5. Intuitively, the performance difference on OASIS versus DRD inputs is probably due to the resolution quality.




(a) Segmentation for OASIS.




(b) Segmentation for DRD.

Figure 5: Typical inputs and outputs based on the dataset.

Altogether, we believe we can give these preliminary observations on these new applications:

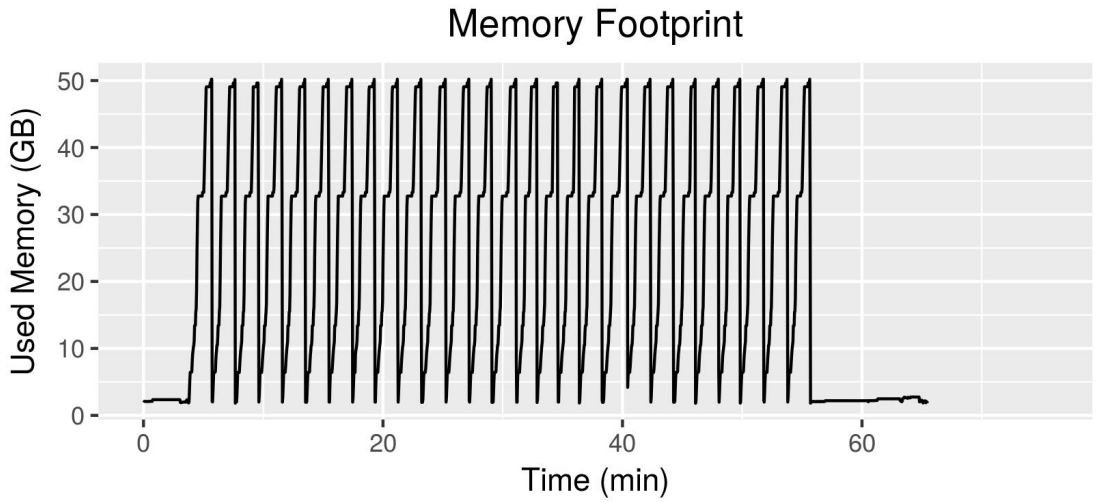
1. We confirm the observations of significant variations in their walltime.
2. These variations are mostly determined by elements from the input, but are not correlated to the size of the input (*quality* and not quantity).

## 2.3 Task-level observations

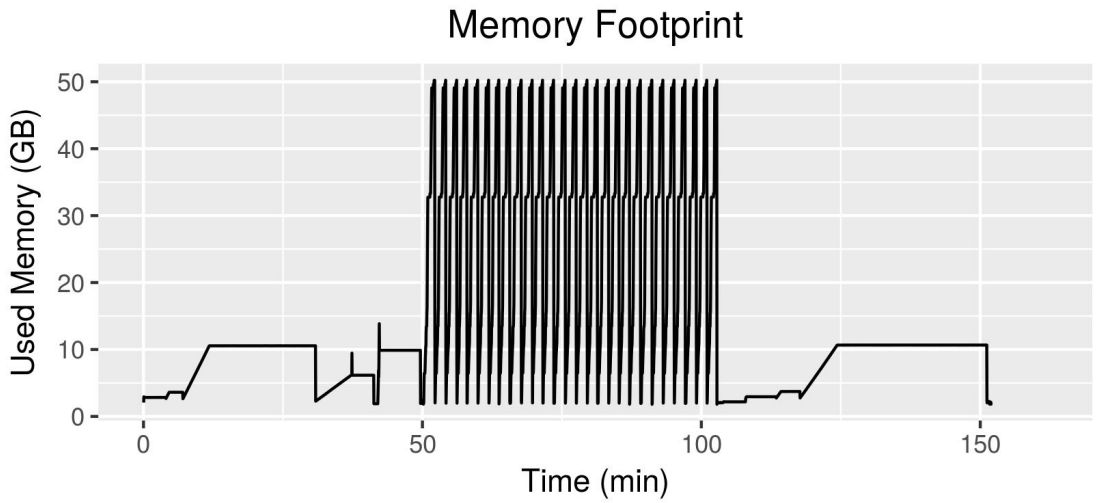
Studies using machine learning methods to estimate the future resource consumption of an application assume a constant peak memory footprint (e.g. [46]). In this section, we study more closely the memory behavior of these new HPC applications.

Figure 6 presents the memory footprint of two runs of the SLANT application, one for each of the input categories. Note that all other runs follow similar trends, specifically **the peak memory usage is not dependent on the input**, only the time depends (and hence the average memory utilization). For both profiles, we can see clearly the three phases of the application (pre-processing, deep-learning, post-processing). Note that these traces hint at the fact that the difference in executed time is more linked to a quality element since there is fewer pre/post-processing time for OASIS input.

In the following, we focus our discussions on the runs obtained from the 88 DRD inputs (Figure 6b) because their pre/post processing steps are more interesting, although the same




(a) Typical memory profile with OASIS input.




(b) Typical memory profile with DRD input.

Figure 6: Examples of memory footprints of the SLANT application with inputs from each considered dataset. Memory consumption is measured every 2 seconds with the *used memory* field of the `vmstat` command.

study could be done for the OASIS inputs.

These memory footprints show that the runs can be divided into roughly seven different tasks of "constant" memory usage:

- *pre-processing* phase: This phase includes the four first tasks. The 1st task shows a memory consumption peak of around 3.5GB for the few first minutes of the application execution. The $2^{nd}$, $3^{rd}$ and $4^{th}$ tasks have respectively a peak of about 10GB, 6GB and 10GB.
- *deep-learning* phase: The $5^{th}$ task, represents the deep-learning phase. This task presents

a periodic pattern with memory consumption peaks going up to 50GB. Each pattern is repeated 27 times, corresponding to the parameterization of the network tiles in SLANT-27 version.

- *post-processing* phase: The $6^{\text{th}}$ and $7^{\text{th}}$ tasks model the last phase of the application, with a memory peak to respectively 3.5GB and 10GB.

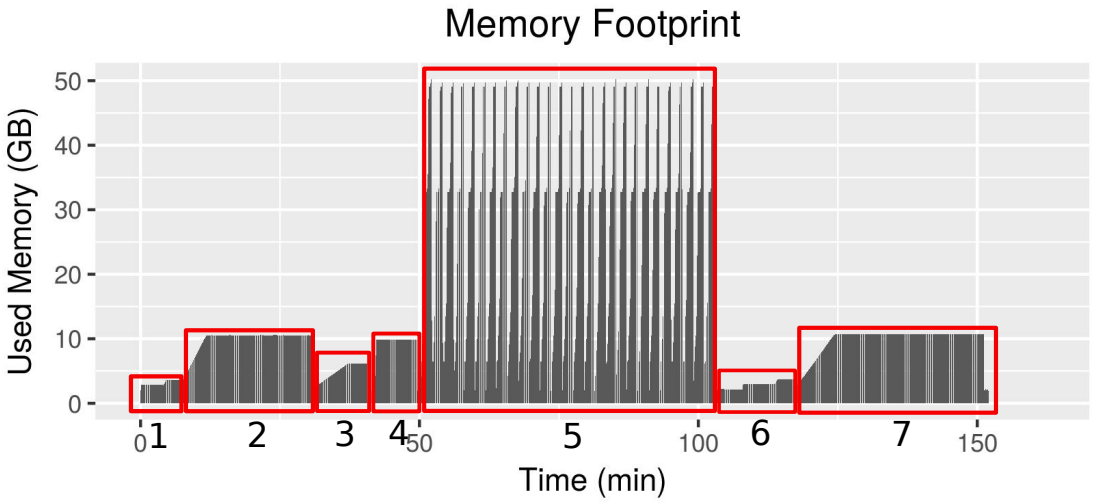



Figure 7: Job decomposition in tasks based on raw data of a memory footprint.

In the second step of this analysis we are interested in the behavior of the job at the task level. We decompose the job into tasks based on the memory characteristics by using a simple parser (see Figure 7). This parser returns the duration of each task within each run based on their memory footprint. Note that this decomposition can be incorrect, we discuss this and its implications later.

Using the decomposition in tasks, we can plot the individual variation of each task execution time (for simplicity, we only considered execution time at the minute level) in Figure 8.

We make the following observations. First, all tasks show variation in their walltime based on the input run. This variation differs from task to task. For instance, task #7 has variations up to 25 minutes while tasks #3 and #4 have less than 5 minutes difference between runs.

Another observation from the raw data on Figure 8, is that some tasks present several peaks (tasks #5 and #7). There may be several explanations to this, from actual task profile (for instance a condition that adds a lot of work if it is met), lack of sufficient data for a complete profile, or finally a bad choice in our task decomposition. Going further, one may be interested in generating a finer grain parsing of the application profile to separate these peaks into individual tasks, based on more parameters than only the memory consumption. We choose not to do this to preserve some simplicity to our model. In the following, we denote by $X_1, \ldots, X_7$ the random variable that represents the execution times of the seven tasks.

An important next question is whether they show correlation in their variation. Indeed, given that they are based on the same input, one may assume that they vary similarly. To study this, we present in Table 1 their Pearson Correlation coefficients. We see that only tasks #1 and #2 present a very high correlation (meaning that their execution times are proportional), while

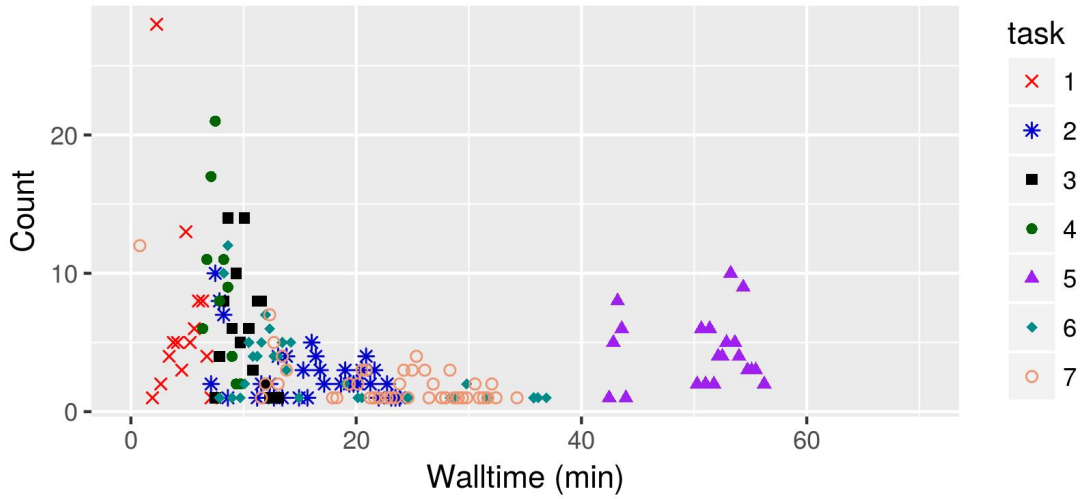



Figure 8: Analysis of the task walltime for all jobs (raw data).

Table 1: Pearson Correlation matrix of the walltimes of the different tasks.

| Task Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1.000 | 0.998 | -0.308 | -0.261 | -0.114 | -0.039 | 0.139 |
| 2 | | 1.000 | -0.293 | -0.277 | 0.142 | -0.058 | 0.159 |
| 3 | | | 1.000 | 0.076 | 0.547 | -0.283 | 0.223 |
| 4 | | | | 1.000 | -0.361 | 0.296 | -0.308 |
| 5 | | | | | 1.000 | -0.568 | 0.574 |
| 6 | | | | | | 1.000 | -0.475 |
| 7 | | | | | | | 1.000 |

others have meaningless correlation. This measure is important as it hints at the independence of the different execution time variables.

Finally, to investigate the distribution of memory usage overtime, we study the task status at all time (at time $t$, which task is being executed). To do so, given $X_i$ $(i = 1 \dots 7)$ the execution time of task $i$, we represent in Figure 9 the functions $y_i(t) = \mathbb{P}\left(\sum_{j \leq i} X_j < t\right)$. Essentially, it means that $y_i$ is the probability that task $i$ is finished.

Figure 9 is read this way: the probability that task $i$ is running at time $t$ corresponds to the distance between the plots corresponding to task $i-1$ and task $i$. For instance, at time $t = 0$ task #1 is running with probability 1. At time 100, tasks #5 to #7 are running (roughly) with respective probability 0.06, 0.5, 0.38. In addition, with probability 0.06 the job has finished its execution.

This figure is interesting in the sense that it gives task properties as a function of time. For instance, given the memory footprint of each task, one can estimate the probability of the different memory needs.

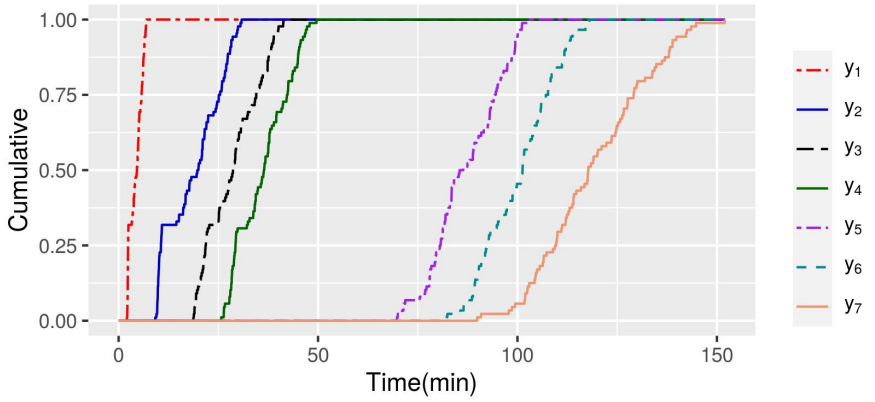



Figure 9: $y_i(t) = \mathbb{P}\left(\sum_{j \leq i} X_j < t\right)$ is the probability that task $i$ is finished at time $t$ (raw data).

# 3 From observations to a theoretical model

Using the observations from Section 2, we now derive a new computational model. We discuss the advantages and limitations of this model in Section 3.2.

## 3.1 Job model

We model an application $A$ as a chain of $n$ tasks:

$$A = j_1 \to j_2 \to \cdots \to j_n,$$

such that $j_i$ cannot be executed until $j_{i-1}$ is finished. Each task $j_i$ is defined by two parameters: an execution time and a peak memory footprint. The peak memory footprint of each task does not depend on the input, and hence can be written as $M_i$. The execution time of each task is however input dependent, and we denote by $X_i$ the random variable that represents the execution time of task $j_i$. $X_i$ follows a probability distribution of density (PDF) $f_i$. We also assume that the $X_i$ are independent.

Finally, the compact way to represent an application is

$$\{(f_1, M_1), \ldots, (f_n, M_n)\}. \tag{1}$$

## 3.2 Discussion

To discuss the model, we propose to interpolate the data from our application with Normal Distributions[3]. We present such an interpolation on Figure 10 (data in Table 2). Fitting to continuous distributions is interesting in terms of data representation, and offers more flexibility to study the properties of the application. As we have seen earlier, Normal Distributions may not be the best candidate for those jobs (for examples jobs with multiple peaks), but they have the advantage of being simpler to manipulate. This is also a good element to discuss the limitations of our model.

Using the interpolations, one can then compute several quantities related to the problem with more or less precision. We show how one would proceed in the following.

[3]We write that $X$ follows a normal distribution $\mathcal{N}(\mu, \sigma)$.

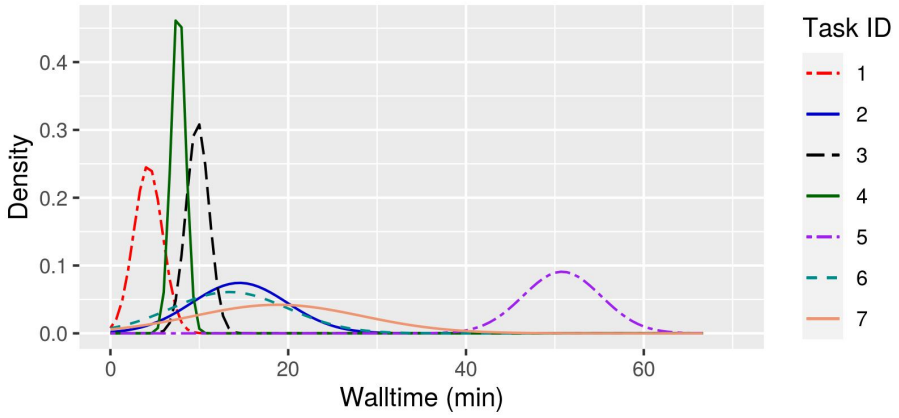



Figure 10: Interpolation of data from Figure 8 with Normal Distributions.

Table 2: Parameters $(\mu, \sigma)$ of the Normal Distributions interpolated in Figure 10.

| Task ID | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Mean $\mu$ (in sec) | 255 | 871 | 588 | 459 | 3050 | 804 | 1130 |
| Std $\sigma$ (in sec) | 96.7 | 322 | 76.8 | 48.1 | 263 | 393 | 568 |

### 3.2.1 Task status with respect to time

We can estimate the functions $\mathbb{P}\left(\sum_{j\leq i} X_j < t\right)$ represented in Figure 9, which later helps to guess the task status with respect to time. Indeed, if $X_1, \ldots, X_i$ are independent normal distributions of parameters $\mathcal{N}(\mu_1, \sigma_1), \ldots, \mathcal{N}(\mu_i, \sigma_i)$, then $Y_i = \sum_{j\leq i} X_j$ follows $\mathcal{N}(\sum_{j\leq i} \mu_j, \sqrt{\sum_{j\leq i} \sigma_j^2})$. We plot in Figure 11 the functions $f_i = \mathbb{P}(Y_i < t)$.

An important observation from this figure is that even if the interpolations per task are not perfect, the sum of their model gets *closer* with time to actual data. This is further discussed in Section 4. Obviously this may not be true for all applications and is subject to caution, however the fact that initially all models seemed far off on a per task basis but converged well is positive.

### 3.2.2 Memory specific quantities

Using this data, one should be able to compute different grandeurs needed for an evaluation, such as:

- The average memory needed for a run $\bar{M} = \sum_{i=1}^{n} M_i \mathbb{E}[X_i] / \sum_{i=1}^{n} \mathbb{E}[X_i]$. This quantity may be useful for co-scheduling schemes in the case of shared/overprovisionned resources [7,40];
- Or even arbitrary values such as, the "likely" maximum memory needed as a function of time.

$$M_\tau(t) = \max\left\{ M_i | \mathbb{P}\left( \sum_{j<i} X_i < t \leq \sum_{j\leq i} X_i \right) > \tau \right\} \tag{2}$$

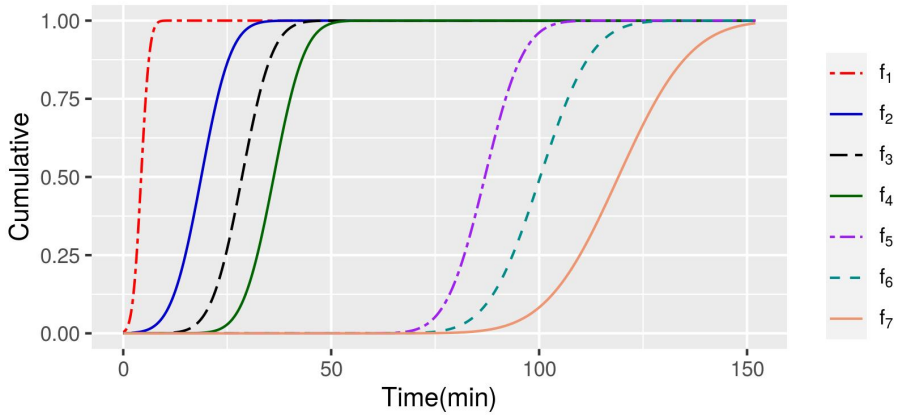



Figure 11: Representation of the cumulative distribution of the termination time of the 7 tasks over time from raw data.

We introduce this value as it will be used in Section 4.1.

In addition, the data for the values of $M_i$ can be obtained with traces of very few executions (since it is not input dependent).

The $f_i$ can also be interpolated from very few executions with more or less precision. We evaluate this precision here with the following experiment, presented in Figure 12. We interpolate from 5, 10, 20, 50 randomly selected (with replacement) runs the functions $f_i$ and compare (i) the evolution of $\bar{M}$; and (ii) the maximum memory need $t \mapsto M_{0.1}(t)$. Each experiment is repeated 10 times to study the variations.

We observe from Figure 12a that with respect to the average memory need, increasing the number of data elements does not improve the precision significantly. This was expected since the only information needed is the expectation of the random variables, which is a lot easier to obtain than the distribution.

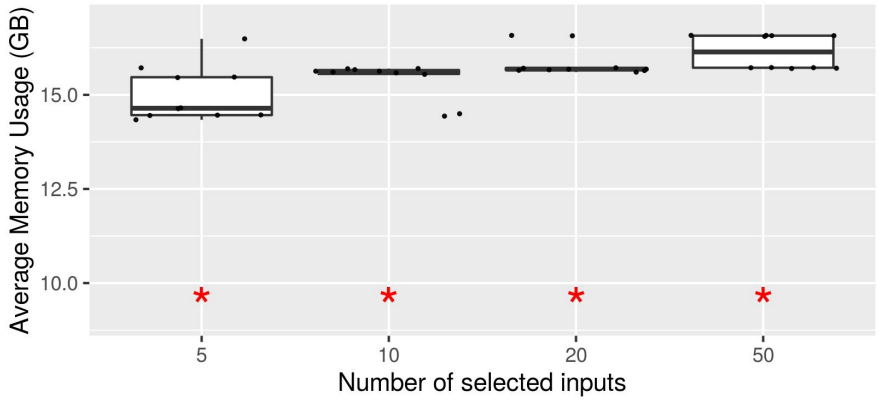
With respect to the maximum memory requirements (Figure 12b), it seems that very few runs (5 runs) already give good performance. This could also be predicted due to the *Maximum* function which gives more weight to any single run.

Obviously this modelization is not perfect and can be improved depending on the level of precision one needs, specifically we can see the following caveats:

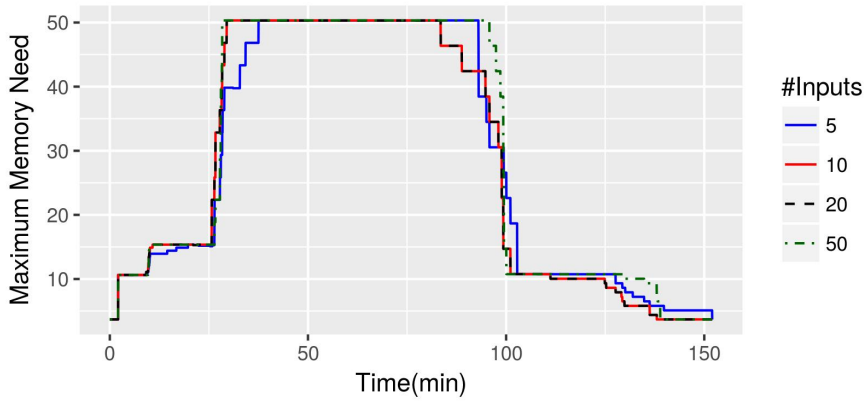
- The peak memory is different from the average memory usage (see for instance task #5 in Figure 7), where the job varies between high-memory needs and low-memory needs. Hence using peak memory to guess the average memory may lead to an overestimation of the average memory (as shown in Figure 12a). To mitigate this, one may add as a variable the average memory per task.

- The model assumes that the lengths of the tasks are independent. However this may not be true as we have seen in Table 1 where the lengths of tasks #1 and #2 are highly correlated. In our case, a simple way to fix this would have been to merge them into a single meta task. We chose not to do this to study the limits of the model.

- This model is based on the information available today. Specifically, the jobs here are sequentialized (the dependencies are represented by a chain of tasks). However we can




(a) Average memory $\bar{M}$ for different number of inputs over 10 experiments. Red star is $\bar{M}$ of the original 88 runs.




(b) $M_{0.1}$ for different number of inputs (avg of 10 experiments).

Figure 12: The model can help interpolate different quantities such as average memory (top) or peak memory (bottom).

expect a more general formulation where the dependencies are more parallel (and hence represented by a Directed Acyclic Graph instead of a linear chain).

To conclude this section, we have presented a model for the novel HPC applications that is easy to manipulate but still seems close to the actual performance. We discussed possible limitations to this model. In the remainder of the paper, we present an algorithmic use-case where one may use this model, and show on experiments that solutions derived from this model are efficient.

# 4 Impact of Stochastic Memory Model on Reservation Strategies

In this section, we now discuss how our model may be used to inform on reservation strategies for HPC schedulers.

Reservation strategies were discussed and studied in a couple of papers to deal with stochastic applications [13, 34]. Essentially, for an application of unknown execution time, the strategies provided users with increasingly-long reservations to use for submission until one was sufficient to execute the whole job. Gainaru et al. [13] included also the optional use of checkpointing in order not to waste what was previously computed. In this work we focus on reservations where a checkpoint is saved after each reservation.

## 4.1 Algorithmic Framework

A reservation strategy is presented under the form

$$\mathcal{S} = ((R_1, T_1, C_1), (R_2, T_2, C_2), \ldots, (R_n, T_n, C_n)) .$$

The strategy would then be executed as follows: initially, the user asks to the system a reservation of length $R_1 + T_1 + C_1$ (time to restart from previous checkpoint, the estimated walltime and the time to checkpoint at the end of the reservation). During the initial $R_1$ units of time, the application gathers the data needed for its computation. Then, during a time $T_1$ it executes. If the walltime is smaller than $T_1$, then the user saves the output data and the run ends. Otherwise, at the end of these $T_1$ units of time, the application checkpoints its current state during the $C_1$ units of time.

- If $C_1$ is enough to perform the checkpoint, then the user repeats the previous step with a reservation of length $R_2 + T_2 + C_2$.
- If $C_1$ is not enough to perform the checkpoint, then the user repeats the previous step with a reservation of length $R_1 + T_1 + T_2 + C_2$.

Finally, we associate to each $(R_i, T_i, C_i)$ in $\mathcal{S}$ a memory request $M_i$ that corresponds to an estimation of the minimum amount of memory for the application not the fail during this reservation. Typically, this value is the maximum peak of the reservation during its computation of $T_i$ units of time. This can be obtained by tracking the progress of the application over reservations. Then, using the likely maximum memory needed as presented in Fig 12b, one is able to estimate the maximum memory need of the application.

### 4.1.1 Evaluated algorithms

In this work we compare three algorithms to compute the reservation strategies. All these strategies are based from the same input: $k$ previous runs of the application (in practice we use $k = 5,\ 10,\ 20,\ 50$).

- All-Ckpt [13, III.D]: This computes the optimal solution to minimize the expected total reservation time when all reservations are checkpointed and when the checkpoint cost is constant. We take the maximum memory footprint over the execution as the basis for the checkpoint cost.
- Mem-All-Ckpt: it is an extension of All-Ckpt based on Section 3.1. Specifically it uses $M_{0.1}$ (defined in Eq. (2)) as the basis for the checkpoint cost function. The complete procedure of this extension is described below.
- Neuro [15, 31]: This is the algorithm used by the neuroscience department at Vanderbilt University. In their algorithm, they use the maximum length of the last $k$ runs as their first reservation. If it is not enough they multiply it by 1.5 and repeat the procedure. To

be fair with the other strategies, we added a checkpoint to this strategy. Hence the length of the second reservation ($T_2$) is only 50% of the first one ($T_1$), so that $T_1 + T_2 = 1.5T_1$. We use the maximum size of a checkpoint as checkpoint cost. For completeness, we have also added a strategy that uses average length instead of maximum length. We denote it by NEURO-AVG.

The strategies of both ALL-CKPT and MEM-ALL-CKPT assume that we have a discrete distribution of execution time for the application. Hence they start by a *modeling phase* using the $k$ inputs. In order to do so, we fit the walltime of the $k$ runs to a normal distribution. We then discretize it into 1000 equally spaced values on the truncated domain $[0, Q(10^{-7})]$ (where $Q(\varepsilon)$ is the $\varepsilon$ quantile of the distribution). In addition we then model a checkpoint cost via a simple latency/bandwidth model, where given a latency $l$ and a bandwidth $b$, the checkpoint time for a volume of data $V$ is $C(V) = l + V/b$.

After discretization we obtain a random variable $Y \sim (v_i, C_i, f_i)_{1 \leq i \leq n}$, such that for $1 \leq i \leq n$, $\mathbb{P}(Y = v_i) = f_i$. The cost to perform a checkpoint at time $v_i$ is $C_i = C(M_{0.1}(v_i))$ for MEM-ALL-CKPT. We assume the cost to restart is constant $R$. Finally, we apply the following dynamic programming procedure to $Y$ ($v_0 = 0$):

$$\begin{cases} S_{\text{MAC}}(n) = 0 \\ S_{\text{MAC}}(i) = \min\limits_{i+1 \leq j \leq n} \left( S_{\text{MAC}}(j) + (R + (v_j - v_i) + C_i) \cdot \sum\limits_{k=i+1}^{n} f_k \right) \end{cases}$$

MEM-ALL-CKPT and ALL-CKPT are then the associated solutions to $S_{\text{MAC}}(0)$ (depending on the checkpoint function). They can be computed in $\mathcal{O}(n^2)$ time.

## 4.2 Experimental Setup

*All code and data for this Section are available for reproducibility at* `https://github.com/anagainaru/ReproducibilityInitiative/tree/master/2020_tpds`. The execution of the application is performed on the Haswell platform. The $k$ inputs chosen for the modeling phase used to derive the algorithms are picked uniformly at random with replacement in the DRD set. The evaluation is performed on the set of 88 inputs from DRD. All evaluations are repeated 10 times.

### 4.2.1 Checkpointing

SLANT is currently available within a Docker image. We used the CRIU external library [42] to perform system level checkpointing of the Docker container without changing the code of SLANT. With each execution of SLANT we are running a daemon in charge of triggering checkpoints at the times given by our strategy.

Actual checkpointing could not be used on the Haswell platform because Docker is not available there and we also do not have the required credentials require by CRIU. Hence we also used the *KNL* platform composed of a 256-thread Intel Knights Landing processor (Xeon Phi 7230, 1.30GHz, Quadrant/Cache mode) with 96GB of main memory. This KNL platform is too slow to perform thorough experiments but Docker checkpointing is supported. Hence experiments on KNL were performed using the checkpoint times (corresponding to the right memory footprint) from that platform and simulated checkpoints (based on the KNL checkpoints) for the Haswell machine. Before doing so, we verified that the memory footprint was identical over the different phases between the two platforms (Figure 17). To evaluate the latency and bandwidth we use

the `dd` unix command with characteristics typical for the CRIU library (multiple image files in Google protocol buffer format [2]).

### 4.2.2 Performance Evaluation

Given a reservation strategy consisting of two reservations $(R_1, T_1, C_1), (R_2, T_2, C_2)$ and an application of walltime $t$, s.t. $T_1 < t \leq T_2$, we define:

1. Its total reservation time: $(R_1 + T_1 + C_1) + (R_2 + (T_2 - T_1) + C_2)$. That is:

$$R_1 + R_2 + T_2 + C_1 + C_2;$$

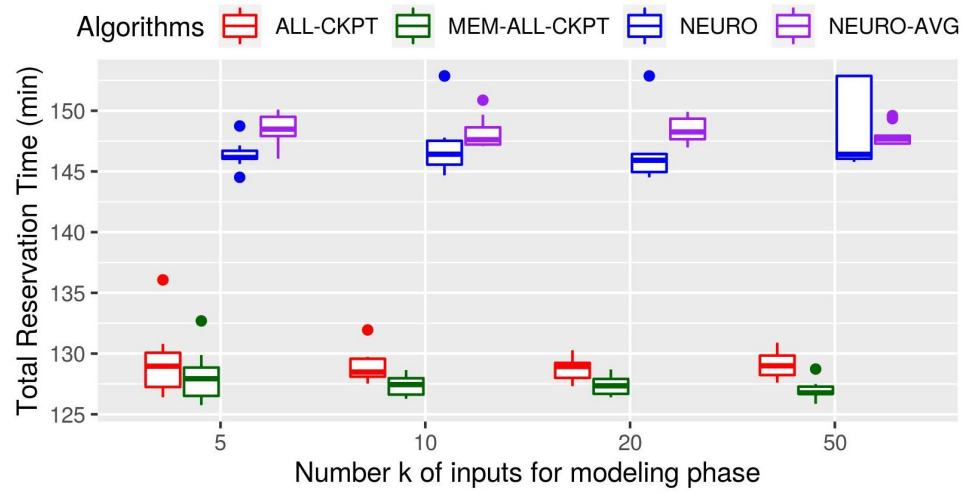
2. Its system utilization, i.e. its walltime divided by its reservation time:

$$\frac{t}{R_1 + R_2 + T_2 + C_1 + C_2};$$

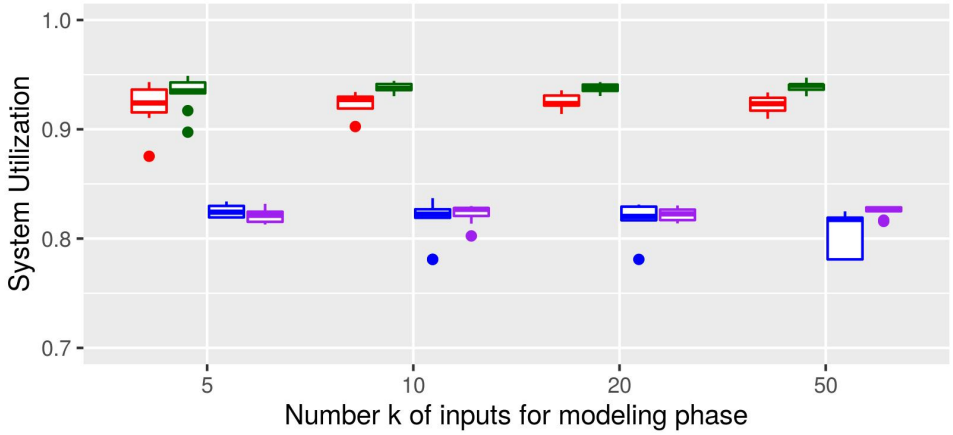
3. In addition, if we define $M_1$ and $M_2$ the memory requested for the reservations, we can define the weighted requested memory as:

$$\frac{(R_1 + T_1 + C_1) \cdot M_1 + (R_2 + T_2 - T_1 + C_2) \cdot M_2}{R_1 + R_2 + T_2 + C_1 + C_2}.$$

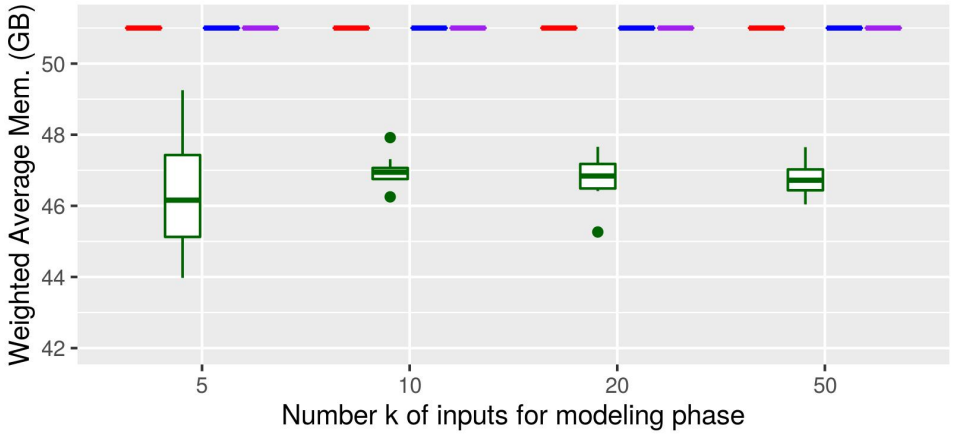
Intuitively this is the total memory used by the different reservations normalized by time.




(a) Average reservation time.




(b) Average utilization.




(c) Weighted average memory.

Figure 13: Performance of the different algorithms for various criteria.

We present in Figure 13 several performance criteria to compare the different algorithms. We first discuss from a high level before entering specifics. Overall using the improved model from Section 3 to design the reservation algorithm allows to improve performances on all fronts. In addition, this model does not use much data, since performance with $k = 5$ are almost as good as performance with $k = 50$. This is an important result which shows the robustness of the model designed to the various approximations that are made (independence of variables etc).

Figure 13a presents the results for the total reservation time metric. NEURO and NEURO-AVG have an higher reservation time, which can be expected because they are naive strategies. An interesting observation is that more data does not help it (on the contrary). This is due to the fact that with more data the strategy includes more outliers, and since the initial reservation uses the maximum length, it guarantees an overestimation every time. MEM-ALL-CKPT performs better than all ALL-CKPT, but the difference is not large. This is probably due to a better estimation of the reservation time for the checkpoint. The observations are similar for the utilization (Figure 13b), for similar reasons.

Finally, Figure 13c plots the weighted average requested memory. ALL-CKPT and NEURO are not memory-aware, and hence assume a constant memory footprint of 51GB throughout execution. In this figure we are more interested by the performance of MEM-ALL-CKPT. The gain is $\sim 8\%$ and corresponds to the runs that needed to use a second reservation (the first one always cover task #5 and hence also has a peak memory of 51GB).

To finish this Section, we would like to point out what we believe is an essential point on the robustness of our model. One can notice that ALL-CKPT is actually the solution MEM-ALL-CKPT when the number of tasks is severely underestimated (essentially estimated to a single task). This is actually a strong argument to make for our solution where, even an unprecise/wrong estimation is robust enough (loss within 4% in reservation time /utilization and 10% in memory). Of course, the more precise the model, the better the results are as is shown by the performance of MEM-ALL-CKPT. A better, task-level, estimation can also lead to other benefits. We discuss them in the next section.

#### 4.2.3 Going further

The next step would be to see how one could deduce a new and improved algorithm by using the task-level information. Specifically, looking at Figure 12a, the natural intuition is to make a first reservation of length 25 min (guaranteed to finish before the memory intensive task #5), allowing it to be a cheaper solution memory-wise.

We study the new version of MEM-ALL-CKPT: MEM-ALL-CKPTV2 that incorporate this additional reservation. In this solution, if task #4 finishes before these 25 minutes, we cannot start task #5 since we do not have enough memory available, hence we checkpoint the output and waste the remaining time. We plot in Figure 14 the total reservation time and weighted average requested memory for ALL-CKPT and MEM-ALL-CKPTV2.

We see that now that MEM-ALL-CKPTV2 can gain $\sim 25\%$ of memory in average in comparison with ALL-CKPT, at no cost reservation-wise. This shows that an application model can offer an optimized strategy when applied to scheduling strategies. In addition, by leveraging the knowledge that task #5 has a huge memory peak in comparison with the other, we are able to optimize the memory usage of reservations for which the probability of running task #5 is unlikely.

### 4.3 Extension to other applications

In this section, we verify that the model constructed based on SLANT also works easily on another application. We used the same evaluation pipeline using a second CPU application,

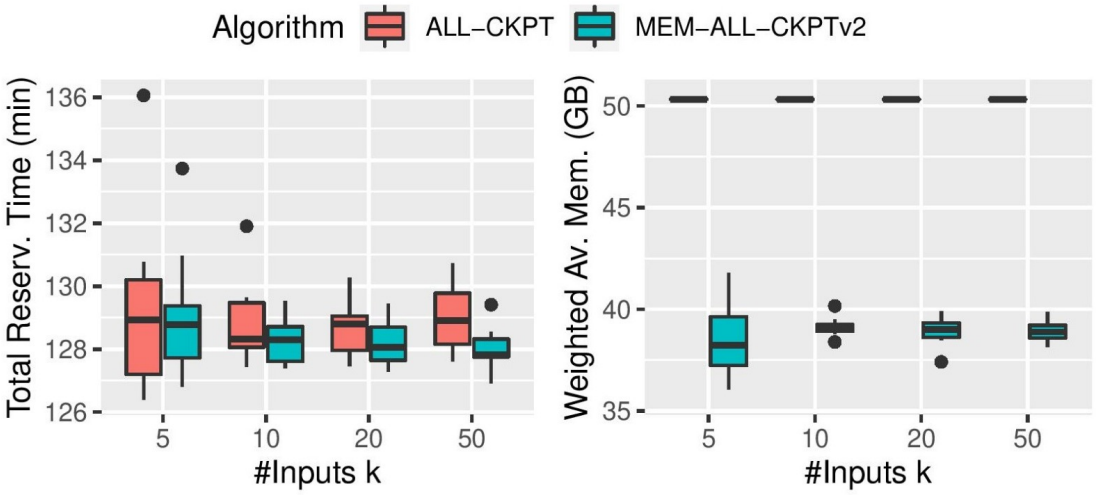



Figure 14: Weighted average requested memory for All-Ckpt and Mem-All-Ckptv2

*MaCRUISE* [22, 23], freely available at `https://github.com/MASILab/MaCRUISE`. Note that this application takes two inputs, the OASIS input and the output of the SLANT application previously studied. The pipeline consists in:

- Running the application $k$ times on the Haswell platform;
- Generating a task model and distribution based on those $k$ inputs;
- Generating different solutions via the algorithmic framework, and evaluating those on the Haswell platform (the evaluation is performed on 46 inputs).

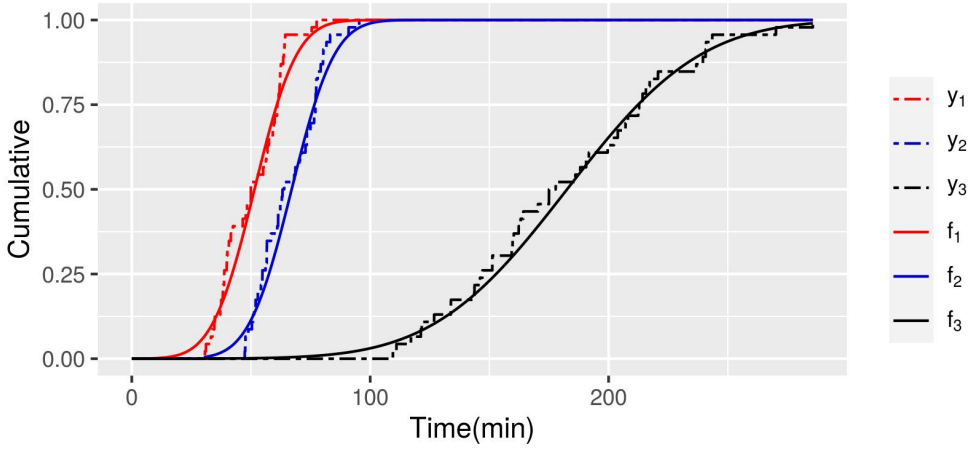



Figure 15: Representation of the cumulative (full) and estimated cumulative (dashed) distribution of the termination time of the 3 tasks over time from raw data.

Table 3: Mean time and Peak Memory of the different tasks of MaCRUISE application.

| Task ID | 1 | 2 | 3 |
|---|---|---|---|
| Mean time (in min) | 50 | 66 | 181 |
| Std (in min) | 12 | 12.5 | 43 |
| Peak Memory (GB) | 7.5 | 5.5 | 2.75 |

Figure 15 presents the cumulative distribution of the termination of the three tasks (the notations used are the same as in the previous section). It allows to evaluate the wall time variation of MaCruise (given by the plot $y_3$): it fluctuates between 2h and 5h. Obviously, the memory footprint and tasks are different than SLANT, we summarize for each task their attribute in Table 3.

Figure 16 presents experimental results performed in the same framework as the ones presented in Fig. 13. (NEURO-AVG is not presented: it gives similar results to NEURO). Overall the results are very positive as they confirm the trend observed in the previous analysis: the solution using the modelization is efficient even with very few information (5 inputs). The performance of MEM-ALL-CKPT compared to ALL-CKPT is not significantly better due to the very low memory cost of all the tasks.

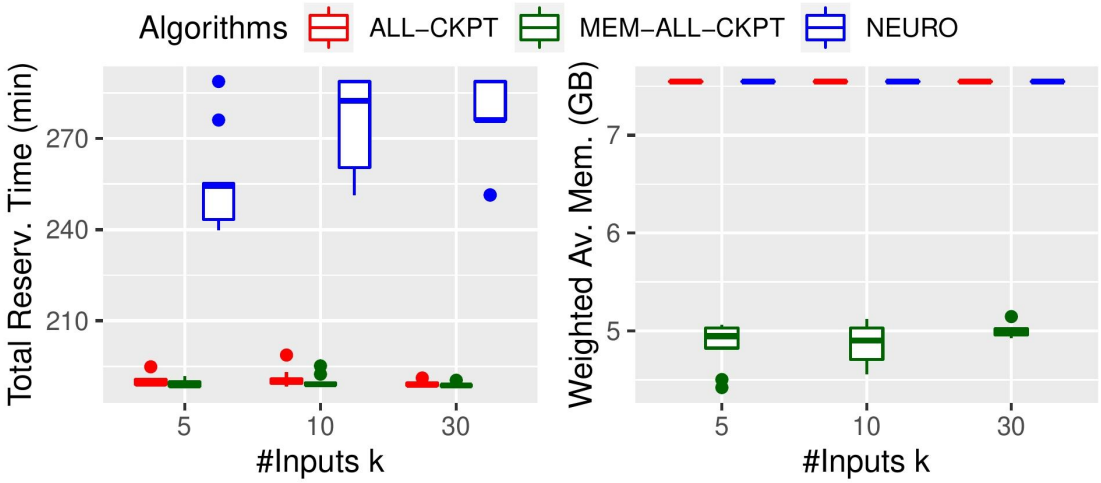
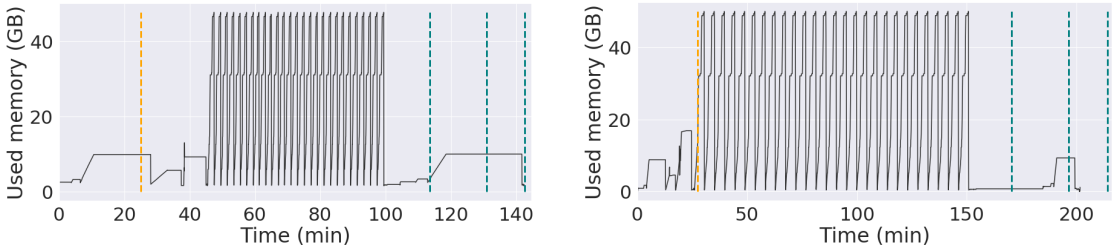



Figure 16: Performance of the different algorithms for various criteria for MaCRUISE application.

## 4.4 Transfers to other architectures

We showed that the MEM-ALL-CKPT reservation strategy may guide users in requesting resources (time and memory) using as little as 5 training data containing detailed information about memory utilization. However, it is often the case that users are only able to extract aggregated information about their application's memory utilization when using a cluster. Users [31] shared that their common practice is to develop and test an application on a local server before deploying it to larger systems. In this section, we are interested to study how well the

strategy transfers between different architectures. For this purpose, we consider we are training the MEM-ALL-CKPT strategy using the Haswell platform presented in Section 2.2 and use the generated reservations to submit the same applications on the KNL platform presented in Section 4.2.1 where we are only allowed to submit an application and read its output. The output of the SLANT application, in addition to information about the brain segmentation, also includes execution time information for the pre-processing, deep-learning and post-processing stages. The deep-learning phase is computation intensive and thus has a relatively constant slowdown (2.3x-2.5x) on the KNL machine. The pre-processing and post-processing phases depend heavily on the quality of the input MRI and have smaller and variable slowdowns (1x-1.6x). Based on these number we use a simple strategy that scales the reservations given by the algorithms by 1.7 (corresponding to the average slowdown in the total execution time). In addition, it scales the initial reservation of MEM-ALL-CKPTV2 by 1.1 (to guarantee that it happens in the pre-process step). Figure 17 presents such a translation.




(a) Execution on the Haswell platform.

(b) Execution on the KNL platform.

Figure 17: Memory footprint of SLANT on the platforms. Vertical lines indicate the reservations given by the MEM-ALL-CKPTV2 using the Haswell platform and scaled for the KNL platform.

We made experiments on 10 randomly chosen input datasets and observed that the memory footprint executions on the two machines typically have different properties (the 7 tasks start and different moments of time, have different durations and different memory consumption as seen in Figure 17). However, the generated reservations have similar properties (e.g. in Figure 17, reservation #1 requires 13 GB of memory for 18% of the total walltime for the Haswell platform and 14 GB for 13% of the total walltime for the KNL; #2 needs 50GB for around 65% of the walltime for both platforms; and #3 and #4 require a little over 10GB for the remaining walltime).

We made experiments on multiple applications using different number of inputs for the training. The experimental workflow consists of 3 steps: (i) we run applications on the Haswell platform and use 5, 10, 20, 50 inputs to compute the reservations based on MEM-ALL-CKPTV2 and ALL-CKPT; (ii) we make 5 runs on random inputs on the KNL and gather the walltime for each stage in order to compute the scaling factor; (iii) we submit new runs on the KNL using the scaled reservations and record the walltime and requested memory for each reservation (10 runs for each experiment). For each reservation we are requesting an upper-bound on the expected memory (15GB for reservations that include the pre-process and post-process and 51GB for the ones including the segmentation stage). Figure 18 presents the weighted average memory requests for the MEM-ALL-CKPTV2 and ALL-CKPT for the original runs on the Haswell platform and for the runs on the KNL when using the scaled reservations on the same set of applications.

The runs on the KNL platform have an overall higher memory footprint (5% in the worse case) than the runs on the Haswell platform since the scaling factors are chosen so that they do not overlap the segmentation phase. We expect other opportunistic strategies based on scaling

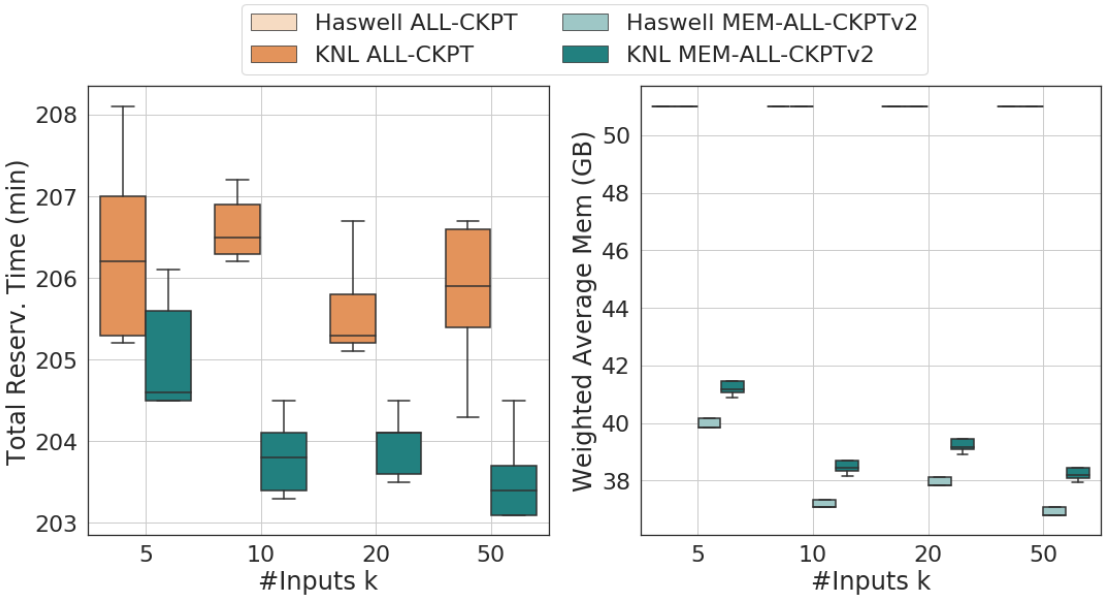



Figure 18: Weighted average memory request for the Mem-All-Ckpt and All-Ckpt for the original runs on the Haswell platform and on the KNL when using the scaled reservations.

factors for each task to give even better results. The total reservation time difference is only of a few minutes to a walltime of more than 3 hours caused by the unnecessary scaling of the checkpoint/restart times. Overall, the average requested memory is 20%-25% smaller when using our strategy even when using a simple scaling strategy. More complex solutions based on current research in cross platform execution transfers [50] can be investigated in the future.

# 5 Related Work

Variation in resource requirements is a known fact for HPC even for existing traditional applications. It can be attributed to several factors: randomized algorithms, inherent job variability (e.g. depending on input data), resource sharing and interferences, OS jitter, etc. Inherent job variability is the topic of this work and includes iterative methods that work towards convergence [47] through discrete steps or studies that trigger an in-depth analysis of subproblems based on certain observations. Those will experience variability in both execution time and memory consumption. It has also been recently observed in machine learning framework on GPUs [32]. Other system constraints such as I/O interference [12] or including consideration of network traffic, power limits or concurrency tuning in the HPC middleware [41], can also become a significant reason for performance variability. Although we could include them in our model, we chose to focus on application-specific variations, a new trend in HPC, and separate their impact from the hardware constraints.

Resource overestimation during submission is a typical strategies for HPC applications since the cost of getting your application killed due to underestimation is very high. This overestimation directly impacts the performance of batch schedulers. To deal with this, typical batch schedulers such as Slurm, Torque or Moab combine simple resource reservation schemes with backfilling [33, 38, 45]. Users are expected to provide the resource requirements when submitting

jobs (most typically walltime and node characteristics, like memory, GPU type, etc). However, as was recently empirically showed by Gainaru et al. [14], the runtime overestimation due to the inherent structure of stochastic jobs can impact both system utilization and user response time by 25-30%. Several authors aim at improving the use of batch scheduler in the presence of uncertainty on the runtimes. Zrigui et al. [52] discussed using online learning to improve the performance of batch schedulers by a simple classification of jobs into two categories, small and large. Big-data frameworks such as MapReduce [11] and Dryad [27] rely on schedulers (e.g. YARN [48] and Mesos [21]) with distinct features such as fairness or resource negotiation to manage the workload. However accurate application needs must be known to the scheduler. The presented strategies aim at providing hints to the user so they can optimize their submissions, but also to these communities since their schedulers may use user-given execution-time distributions of tasks to implement their own sequence of reservation with checkpointing.

To provide solutions in the presence on uncertain execution time, some work focus on optimizing the expected response time of applications by performing distribution fitting [8,18,28,37,39]. They assume a well-known probability distribution of the job execution time. These ideas were extended to provided near-optimal reservation strategies in both HPC and cloud systems [5] for a set of stochastic jobs with backfilling [14], and later with optional checkpointing [13]. These work do not consider a task model for the stochasticity of the application because they simply focused on the execution time (flat memory model). Our work extends the ideas from these papers in Section 4.1 by developing a stochastic task model which allows to study a memory footprint model.

While our work focuses on working *with* the uncertainty of execution time, another complementary direction is to try to remove this uncertainty by predicting the execution time. The predictive methods based on machine learning, often rely on supervised inductive learning over historical log files on large-scale compute clusters, using either predicted memory usage of the jobs or predicted the execution time of the jobs and assume a large set of training data. Tanash et al. [46] use five types of regression algorithms on a large dataset (millions of entries) containing past executions of applications on their internal cluster and predict both the memory and the processing time of future runs. Andresen et al. [3] combines CPU and GPU execution historic logs and generate observations that help users or administrators to classify jobs into equivalence classes by likelihood of failure. Kumar et al. [29] use a predictive scheme for identifying small walltime jobs. In a similar approach, Gaussier et al. [17] introduced several machine learning methods for predicting the class of execution (small/large) for HPC application with the goal of improving scheduling and backfilling algorithms. Closer to our study, Matsunaga and Fortes [34] focus on two bioinformatics applications. Their method is capable of increasing the accuracy of predicting the job execution time, memory and space usage, but requires a large training set. Unlike these studies, our applications are extremely dynamic with their codes in continuous change. Thus they require a strategy that not only is capable of dealing with stochasticity in memory and execution time, but can learn the behavioral pattern of the application fast.

Checkpoint-restart is an obvious way to deal with stochastic applications and/or platform unavailability [26,49]. Insufficient reservations or failures are mitigated by recovering a checkpoint that was periodically saved. Computing the optimal checkpointing interval was the target of a lot of work [10, 26, 51] to ensure a good probability of application success without spending to much time/resources for checkpointing.

Checkpointing may be performed either by the application itself explicitly modifying the code to work with a user level checkpoint library (like FTI [6]) or by linking an external library. We focus on this latter case because it generally does not require to modify the application. BCLR [19] was a popular solution but it does not seem to be maintained anymore and does not support containers as far as we know. DMTCP [4] is a more recent alternative that has

good support for parallel HPC jobs and may be integrated with Slurm [44]. However it lacks container supports. Hence we rather used CRIU [1] which is well supported in the upstream Linux kernel and has support for checkpointing containers [36, 42]. However Docker container support [9] seems still experimental. Our work is actually not strongly tied with CRIU. Hence we may revise our choice in the future if target applications require MPI support or do not need containers.

Checkpointing GPU-enabled applications is difficult without a way to save the internal GPU state. Although some proxy-based approaches have been proposed [16], most actual implementations still rely on application-specific modification [43] which is not applicable to our study. Moreover using GPUs in Docker requires adhoc solutions such as NVIDIA-Docker that do not support checkpointing currently.

# 6 Conclusion

The new wave of HPC applications are using exploratory codes designed with a focus on productivity and not performance and do not fit the traditional cloud/HPC models. We propose a novel approach to extract a generic model of their runtime behavior with stochastic execution times and memory footprints. The model may be used to optimize their execution on large-scale clusters by guiding the resource reservation and checkpointing strategies. We chose in this paper a neuroscience application to demonstrate the benefits of such a method, but we believe our approach is general and can be applied to a large set of applications including the ones using AMR based methods or from fields using highly dynamic and complex workflows, like bioinformatics or phylogeny. We then demonstrated the robustness of the model that can be generated even from very few inputs (five previous runs!). This good performance may be a good indicator that the learning could be done "on-the-fly", for applications whose code may be dynamic. Further investigations into this need to be done to verify that this robustness holds for a wider class of applications.

Using this model, we have demonstrated how one could optimize the utilization of current HPC schedulers in order to minimize the total reservation time and requested memory. These are important metrics in HPC: the wait time to be scheduled for execution in a cluster for neuroscience applications can typically reach days. Obviously, to further demonstrate this, one would need to account for other sources of variability in a more complete execution model such as shared resources (shared nodes, I/O congestion, etc). We plan to further investigate more complex methods of optimization in the future. Finally, we also believe the application behavioral model can be beneficial in understanding the needs of these applications and can guide the design of future middleware for HPC systems (including the I/O and memory management frameworks).

**Acknowledgments** Executions of SLANT application on Haswell platform presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see `https://www.plafrim.fr/en/home/`).

# References

[1] Checkpoint restart in userspace (criu). `https://criu.org`.

[2] Google protocol buffer format. `https://developers.google.com/protocol-buffers`.

[3] D. Andresen, W. Hsu, H. Yang, and A. Okanlawon. Machine learning for predictive analytics of compute cluster jobs. *CoRR*, abs/1806.01116, 2018.

[4] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–12, Rome, Italy, 2009. IEEE.

[5] G. Aupy, A. Gainaru, V. Honoré, P. Raghavan, Y. Robert, and H. Sun. Reservation Strategies for Stochastic Jobs. In *IPDPS 2019 - 33rd IEEE International Parallel and Distributed Processing Symposium*, pages 166–175, Rio de Janeiro, Brazil, May 2019. IEEE.

[6] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[7] J. Breitbart, S. Pickartz, S. Lankes, J. Weidendorfer, and A. Monti. Dynamic co-scheduling driven by main memory bandwidth utilization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, 2017.

[8] J. Bruno, P. Downey, and G. N. Frederickson. Sequencing tasks with exponential service times to minimize the expected flow time or makespan. *Journal of the ACM*, 28(1):100–113, 1981.

[9] Y. Chen. Checkpoint and Restore of Micro-service in Docker Containers. In *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*. Atlantis Press, 2015/10.

[10] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. Scheduling the i/o of hpc applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[13] A. Gainaru, B. Goglin, V. Honoré, G. Pallez, P. Raghavan, Y. Robert, and H. Sun. Reservation and Checkpointing Strategies for Stochastic Jobs. In *IPDPS 2020 - 34th IEEE International Parallel and Distributed Processing Symposium*, New Orleans, United States, May 2020.

[14] A. Gainaru, G. Pallez, H. Sun, and P. Raghavan. Speculative scheduling for stochastic HPC applications. In *ICPP*, 2019.

[15] A. Gainaru, H. Sun, G. Aupy, Y. Huo, B. A. Landman, and P. Raghavan. On-the-fly scheduling versus reservation-based scheduling for unpredictable workflows. *Int. J. High Perf. Computing Applications*, 2019.

[16] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman. Crum: Checkpoint-restart support for cuda's unified memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 302–313, 2018.

[17] E. Gaussier, J. Lelong, V. Reis, and D. Trystram. Online tuning of easy-backfilling using queue reordering policies. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2304–2316, 2018.

[18] A. Goel and P. Indyk. Stochastic load balancing and related problems. In *FOCS*, pages 579–586. ACM, 1999.

[19] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics. Conference Series*, 46, 9 2006.

[20] J. Haxby, J. S. Guntupalli, A. Connolly, Y. Halchenko, B. Conroy, M. Gobbini, M. Hanke, and P. Ramadge. A common, high-dimensional model of the representational space in human ventral temporal cortex. *Neuron*, 72:404–16, 10 2011.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Conf. Networked Systems Design and Implementation*, pages 295–308, 2011.

[22] Y. Huo, A. Carass, S. M. Resnick, D. L. Pham, J. L. Prince, and B. A. Landman. Combining multi-atlas segmentation with brain surface estimation. In *Medical Imaging 2016: Image Processing*, volume 9784, page 97840E. International Society for Optics and Photonics, 2016.

[23] Y. Huo, A. J. Plassard, A. Carass, S. M. Resnick, D. L. Pham, J. L. Prince, and B. A. Landman. Consistent cortical reconstruction and multi-atlas brain segmentation. *NeuroImage*, 138:197–210, 2016.

[24] Y. Huo, Z. Xu, K. Aboud, P. Parvathaneni, S. Bao, C. Bermudez, S. M. Resnick, L. E. Cutting, and B. A. Landman. Spatially localized atlas network tiles enables 3d whole brain segmentation from limited data. In A. F. Frangi, J. A. Schnabel, C. Davatzikos, C. Alberola-López, and G. Fichtinger, editors, *Medical Image Computing and Computer Assisted Intervention – MICCAI 2018*, pages 698–705, Cham, 2018. Springer International Publishing.

[25] Y. Huo, Z. Xu, Y. Xiong, K. Aboud, P. Parvathaneni, S. Bao, C. Bermudez, S. M. Resnick, L. E. Cutting, and B. A. Landman. 3d whole brain segmentation using spatially localized atlas network tiles. *NeuroImage*, 194:105 – 119, 2019.

[26] T. Hérault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*. Springer Verlag, 2015.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *2nd ACM SIGOPS/EuroSys European Conf. Computer Systems*, 2007.

[28] J. Kleinberg, Y. Rabani, and E. Tardos. Allocating bandwidth for bursty connections. In *STOC*, pages 664–673, 1997.

[29] R. Kumar and S. Vadhiyar. Identifying quick starters: Towards an integrated framework for efficient predictions of queue waiting times of batch parallel jobs. In W. Cirne, N. Desai, E. Frachtenberg, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 196–215, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] P. J. LaMontagne, T. L. Benzinger, J. C. Morris, S. Keefe, R. Hornbeck, C. Xiong, E. Grant, J. Hassenstab, K. Moulder, A. Vlassenko, M. E. Raichle, C. Cruchaga, and D. Marcus. Oasis-3: Longitudinal neuroimaging, clinical, and cognitive dataset for normal aging and alzheimer disease. *medRxiv*, 2019.

[31] B. Landman. Medical-image Analysis and Statistical Interpretation (MASI) Lab. `https://my.vanderbilt.edu/masi/`.

[32] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–61, 2020.

[33] D. A. Lifka. The ANL/IBM SP Scheduling System. In *JSSPP*, pages 295–303, 1995.

[34] A. Matsunaga and J. A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504, 2010.

[35] A. Merzky, M. Santcroos, M. Turilli, and S. Jha. Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers. *CoRR*, abs/1512.08194, 2015.

[36] A. Mirkin, A. Kuznetsov, and K. Kolyshkin. Containers checkpointing and live migration. In *In Ottawa Linux Symposium*, 2008.

[37] R. H. Möhring, A. S. Schulz, and M. Uetz. Approximation in stochastic scheduling: The power of LP-based priority policies. *Journal of the ACM*, 46(6):924–942, 1999.

[38] A. W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[39] J. Niño Mora. Stochastic scheduling. *Encyclopedia of Optimization*, pages 3818–3824, 2009.

[40] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. De Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 173–182, 2013.

[41] T. Patki, J. J. Thiagarajan, A. Ayala, and T. Z. Islam. Performance optimality or reproducibility: That is the question. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[42] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti. Migrating linux containers using criu. In M. Taufer, B. Mohr, and J. M. Kunkel, editors, *High Performance Computing*, pages 674–684, Cham, 2016. Springer International Publishing.

[43] B. Pourghassemi and A. Chandramowlishwaran. cudacr: An in-kernel application-level checkpoint/restart scheme for cuda-enabled gpus. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 725–732, 2017.

[44] M. Rodríguez, J. Moríñigo, and R. Mayo-García. When you have a hammer, everything looks like a nail - Checkpoint/restart in Slurm. SLURM User Group 2017.

[45] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The EASY - LoadLeveler API Project. In *JSSPP*, pages 41–47, 1996.

[46] M. Tanash, B. Dunn, D. Andresen, W. Hsu, H. Yang, and A. Okanlawon. Improving hpc system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[47] C. T. Vaughan and S. D. Hammond. Evaluating production load balancing functions for adaptive mesh schemes using mini-applications. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

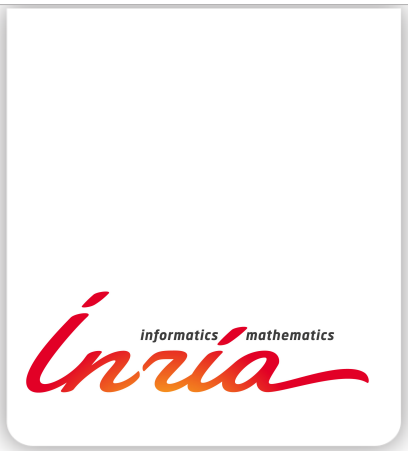
[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *the 4th Annual Symposium on Cloud Computing*, pages 5:1–5:16, 2013.

[49] K. Wolter, editor. *Stochastic Models for Fault Tolerance, Restart, Rejuvenation, and Checkpointing*. Springer Verlag, 2010.

[50] L. T. Yang, Xiaosong Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 40–40, 2005.

[51] J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. ACM*, 17(9):530–531, 1974.

[52] S. Zrigui, R. de Camargo, D. Trystram, and A. Legrand. Improving the performance of batch schedulers using online job size classification. 2019.




**RESEARCH CENTRE**
**BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour
33405 Talence Cedex

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

ISSN 0249-6399