Profiles of upcoming HPC Applications and their Impact on Reservation Strategies
Ana Gainaru, Brice Goglin, Valentin Honoré, Guillaume Pallez

To cite this version:
Ana Gainaru, Brice Goglin, Valentin Honoré, Guillaume Pallez. Profiles of upcoming HPC Applications and their Impact on Reservation Strategies. IEEE Transactions on Parallel and Distributed Systems, 2021, 32 (5), pp.1178-1190. 10.1109/TPDS.2020.3039728 . hal-03010676

HAL Id: hal-03010676
https://inria.hal.science/hal-03010676
Submitted on 17 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Profiles of upcoming HPC Applications
and their Impact on Reservation Strategies

Ana Gainaru, Brice Goglin, Valentin Honoré and Guillaume Pallez (Aupy)

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 from emerging fields that start to 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 use 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 based on the application expected resource requirements. We derive solutions using the model provided by the first step of this work. We experimentally show the robustness of the model, even with very few data points or using another application, to generate the model, and provide performance gains with regards to standard and more recent approaches used in the neuroscience community.

Index Terms—stochastic application, execution time, memory footprint, scheduling, checkpointing

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. For 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.

Figure 1 presents the memory requirements and requests for nine exploratory applications from the medical and neuroscience department at the Vanderbilt University [29]. 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 Figure 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 a computational exploration application from the neuroscience domain with the goal of understanding the properties and characteristics of these new frameworks. We are interested in the pure application behavior that is non-biased by interference due to the system or other applications (e.g. congestion due to shared resources). We focus on the Spatially Localized Atlas Network Tiles (SLANT) 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 in 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 at the same time, 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 [33]. Similarly, applications using Adaptive Mesh Refinement (AMR) methods have been shown to have high unpredictable performance variations based on characteristics of the input data [45].

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 a 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 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 [22], [23]. 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 [27]. There exists different version of SLANT depending on whether the network tiles are overlapped or not. Here, we consider the overlapping version (SLANT-27 [23]) 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 [13], 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 [28] and Dartmouth Raiders Dataset (DRD) [18] 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.

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 $70\text{min} \pm 15\%$ and DRD inputs have a walltime of $125\text{min} \pm 30\%$. The natural questions that arise are the following:

- 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...

1. The code is freely available at [https://github.com/MASILab/SLANTbrainSeg](https://github.com/MASILab/SLANTbrainSeg).
2. For this very large dataset, we only used a subset of available data.
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.

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., [44]). 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 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 2nd, 3rd and 4th tasks have respectively a peak of about 10GB, 6GB and 10GB.
- deep-learning phase: The 5th 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 6th and 7th tasks model the
Fig. 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.

(a) Typical memory profile with OASIS input.

(b) Typical memory profile with DRD input.

last phase of the application, with a memory peak to respectively 3.5GB and 10GB.

Fig. 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 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 $X_i$ ($i = 1 \ldots 7$) the execution time of task $i$, we represent in Figure 9 the functions $y_i(t) = P\left(\sum_{j \leq i} X_j < t\right)$. Essentially, it means that $y_i$ is the probability that task $i$ is finished.

Fig. 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

![Fig. 8: Analysis of the task walltime for all jobs (raw data).](image)

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

<table>
<thead>
<tr>
<th>Task Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.99</td>
<td>-0.30</td>
<td>-0.26</td>
<td>-0.11</td>
<td>-0.04</td>
<td>0.14</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>-0.29</td>
<td>-0.27</td>
<td>0.14</td>
<td>-0.05</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.07</td>
<td>0.55</td>
<td>-0.28</td>
<td>0.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
<td>-0.36</td>
<td>0.29</td>
<td>-0.30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>-0.57</td>
<td>0.57</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>-0.47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.00</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


that represents the execution time of task $j$ written as $M_j$. Each task does not depend on the input, and hence can be a peak memory footprint. The peak memory footprint of each task, one can estimate the probability of the different memory needs.

### 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:

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

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. We present such an interpolation on Figure 10 (data in Table 2).

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

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Mean $\mu$ (in sec)</th>
<th>Std $\sigma$ (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>255</td>
<td>96.7</td>
</tr>
<tr>
<td>2</td>
<td>871</td>
<td>322</td>
</tr>
<tr>
<td>3</td>
<td>588</td>
<td>76.8</td>
</tr>
<tr>
<td>4</td>
<td>499</td>
<td>48.1</td>
</tr>
<tr>
<td>5</td>
<td>3050</td>
<td>263</td>
</tr>
<tr>
<td>6</td>
<td>804</td>
<td>393</td>
</tr>
<tr>
<td>7</td>
<td>1130</td>
<td>568</td>
</tr>
</tbody>
</table>

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.

#### 3.2.1 Task status with respect to time

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

![Fig. 11: Representation of the cumulative distribution of the termination time of the 7 tasks over time from raw data.](image)

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 $\bar{M} = \sum_{i=1}^{n} M_i E[X_i] / \sum_{i=1}^{n} E[X_i]$. This quantity may be useful for co-scheduling schemes in the case of shared/overprovisionned resources [5], [38].
- Or even arbitrary values such as, the “likely” maximum memory needed as a function of time.

$$M_f(t) = \max \left\{ M_i \mid P \left( \sum_{j \leq i} X_j < t \leq \sum_{j \leq i} X_j \right) > \tau \right\} \tag{2}$$
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 $M_i$; and (ii) the maximum memory need $t \mapsto M_{0,1}(t)$. Each experiment is repeated 10 times to study the variations.

![Fig. 12: The model can help interpolate different quantities such as average memory (top) or peak memory (bottom).](image)

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. The difference between $M$ as evaluated and the red star is because of the job modelisation. Indeed, we consider constant memory per task when it is not the case. For instance the memory of Task 5 is set at 50GB in the model (and in the computation of $M$), when in practice it fluctuates a lot (and in the measures taken for the red star).

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.

![Fig. 12: The model can help interpolate different quantities such as average memory (top) or peak memory (bottom).](image)

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 4 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 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 [11], [32]. 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. [11] 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, but the same type of analysis could be done without checkpointing.

4.1 Algorithmic Framework

A reservation strategy is presented under the form

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

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 $R_2 + T_2 + C_2$. 

In this work we compare three algorithms to compute the application over reservations. Then, using the likely maximum amount of memory for the application not to fail during the first reservation, the application can spend up to $T_i$ units of time, a checkpoint of the application progress is performed and takes $C_i$ units of time. If the actual execution time of SLANT is such that $t \leq 115$ min, then this first reservation was sufficient and we stop (and the total reservation is better than the first strategy). However, if $t > 115$ min, the second reservation is required. It starts by a restart overhead, which represents the time to read the checkpoint from disks. Then the application can be executed for 45 min more, which guarantees application termination. In this case, the total reservation time is worse than the first strategy. To determine the size of the reservations, we use the distribution of application walltime.

Finally, we associate to each $(R_i, T_i, C_i)$ in $S$ a memory request $M_i$ that corresponds to an estimation of the minimum amount of memory for the application not to 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.

### 4.2 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** [11, 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** [13, 29]: 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 $n$ equally spaced values (we use $n = 1000$ here) 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 $1 \leq i \leq n$, $P(Y = v_i) = f_i$. 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 $Y (v_0 = 0)$, which gives the following expected cost for all-checkpoint strategies:

$$
S_{\text{MAC}}(n) = \begin{cases} 0 & \text{(init)} \\
\min \sum_{i+1 \leq j \leq n} S_{\text{MAC}}(j) + (R + (v_j - v_{i+1}) + C_i) \cdot \sum_{k=i+1}^{n} f_k & \text{MEM-ALL-CKPT and ALL-CKPT are then the associated solutions to } S_{\text{MAC}}(0) \text{ (depending on the checkpoint function).} \\
\end{cases}
$$

The associated reservation strategies can be computed in $O(n^2)$ time.

### 4.3 Experimental Setup

All code and data for this Section are publicly available for reproducibility\footnote{https://github.com/anaginaru/ReproducibilityInitiative/tree/master/2020_tpsds}. 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.

#### 4.3.1 Checkpointing

SLANT is currently available within a Docker image. We used the CRIU external library\footnote{https://github.com/ProjectCRIU/criu} 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 required 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 18). 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)

4.3.2 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. \(T_1 < t < T_1 + T_2\), we define:

1) Its total reservation time: \(T_1 + T_2 + C_1 + C_2\).
2) Its system utilization, i.e. its walltime divided by its reservation time:

\[
\frac{t}{R_1 + R_2 + T_1 + T_2 + C_1 + C_2};
\]
3) In addition, if we define \(M_1\) and \(M_2\) the memory requested for the reservations, we can define the weight requested memory as:

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

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

We present in Figure 14 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). For completeness, it should be noted that in an HPC machine, utilization is a more global measure that is better than the one described here thanks to optimization strategies such as reshuffling or backfilling when a job terminates early. For this work we focus at the single application perspective in order to be able to interpret the performance precisely.

Figure 14a presents the results for the total reservation time metric. NEURO and NEURO-AVG have a 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 14b), for similar reasons.

Finally, Figure 14c 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

---

6. [https://developers.google.com/protocol-buffers](https://developers.google.com/protocol-buffers)
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.

4.3.3 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 15, the total reservation time and weighted average requested memory for ALL-CKPT and MEM-ALL-CKPTv2. Note that the better performance for 5 inputs in the right plot is not statistically significant due to small amount of selected inputs.

![Fig. 15: Weighted average requested memory for ALL-CKPT and MEM-ALL-CKPTv2](image)

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.

4.4 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, MaCRUISE [20], [21]. Note that this application takes two inputs, the OASIS input and the output of the SLANT application previously studied. The pipeline consists of:

- 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).

![Fig. 16: Representation of the cumulative (full) and estimated cumulative (dashed) distribution of the termination time of the 3 tasks over time from raw data.](image)

Figure 16 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 attributes in Table 3.

![Fig. 17: Performance of the different algorithms for various criteria for MaCRUISE application.](image)

Figure 17 presents experimental results performed in the same framework as the ones presented in Figure 14 (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.

4.5 Transfers to other architectures

We showed that the MEM-ALL-CKPT reservation strategy may guide users in requesting resources (time and mem-

<table>
<thead>
<tr>
<th>Task ID</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean time (in min)</td>
<td>50</td>
<td>66</td>
<td>181</td>
</tr>
<tr>
<td>Std (in min)</td>
<td>12</td>
<td>12.5</td>
<td>43</td>
</tr>
<tr>
<td>Peak Memory (GB)</td>
<td>7.5</td>
<td>5.5</td>
<td>2.75</td>
</tr>
</tbody>
</table>

Table 3: Mean time and Peak Memory of the different tasks of MaCRUISE application.
ory) 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 running on a cluster. The common practice for users [29] 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 4.3.1 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-Ckpt by 1.1 (to guarantee that it happens in the pre-process step). Figure 18 presents such a translation.

![Fig. 18: Memory footprint of SLANT on the platforms. Vertical lines indicate the reservations given by the MEM-ALL-Ckpt using the Haswell platform and scaled for the KNL platform.](image)

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 18). However, the generated reservations have similar properties (e.g. in Figure 18) 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 19 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 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. Note that these experiments are done on small systems where we control the interference of the system in order to better understand the connection between individual runs on each architecture without influence from system level variation. More experiments and more complex solutions based on current research in cross platform execution transfers [48] 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 [45] 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 [30]. Other system constraints such as I/O interference [10] or including consideration of network traffic, power limits or concurrency tuning in the HPC middleware [39], 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 strategy 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 [31], [35], [43]. 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. [12], 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. [50] 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 [9] and Dryad [25] rely on schedulers (e.g. YARN [46] and Mesos [19]) 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 [6], [16], [26], [35], [37]. 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 [3] for a set of stochastic jobs with backfilling [12], and later with optional checkpointing [14]. 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. [44] 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. [11] 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. [27] use a predictive scheme for identifying small walltime jobs. In a similar approach, Gaussier et al. [15] 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 [32] 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 and/or platform unavailability [24], [47]. 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 [8], [24], [49] 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 [4]) or by linking an external library. We focus on this latter case because it generally does not require to modify the application. BLCCR [17] was a popular solution but it does not seem to be maintained anymore and does not support containers as far as we know. DMTCP [2] is a more recent alternative that has good support for parallel HPC jobs and may be integrated with Slurm [42]. However it lacks container supports. Hence we rather used CRIU [7] which is well supported in the upstream Linux kernel and has support for checkpointing containers [34], [40]. However Docker container support [7] 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 [14], most actual implementations still rely on application-specific modification [41] 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 that do not fit the traditional cloud/HPC models. These applications present high variability in their resource utilization throughout time as well as between multiple runs on different input
files. These applications are currently penalized by current schedulers since they do not fit the normal tightly coupled scientific applications nor are they small enough to be used by the backfilling mechanism: the wait time to be scheduled for execution in a cluster for neuroscience applications can typically reach days.

We propose a novel approach to extract a generic model of their runtime behavior with stochastic execution times and memory footprints. In this work, we provided a first demonstration of what such an extraction would look like along with scheduling techniques to use this model. This work focused on a specific class of applications (linear dependencies between jobs, memory footprint independent of the input), run on dedicated HPC machines. We selected for this study 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.

In the framework of this study, 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.

We hope that this proof of concept can open larger studies to show whether this can be generalized to larger classes of application (such as non linear workflow models, varying level of parallelism). We believe that the model may be used to optimize their execution on large-scale clusters by guiding the resource reservation and checkpointing strategies. Ultimately, this could help to optimize the utilization of resources of current HPC schedulers.

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).

**References**


