2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC) | 979-8-3315-0909-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/HIPC62374.2024.00010

Allocation Strategies for Disaggregated Memory in HPC Systems

Robin Boëzennec
University of Rennes, Inria, CNRS, IRISA
Rennes, France
robin.boezennec@inria.fr

Danilo Carastan-Santos
Univ. Grenoble Alpes, Grenoble INP, Inria, CNRS, LIG
Grenoble, France
danilo.carastan-dos-santos@inria.fr

Fanny Dufossé
Univ. Grenoble Alpes, Grenoble INP, Inria, CNRS, LIG
Grenoble, France
fanny.dufosse@inria.fr

Guillaume Pallez
INRIA
Rennes, France
guillaume.pallez@inria.fr

***Abstract*—In this work we consider scheduling strategies to deal with disaggregated memory for HPC systems. Disaggregated memory is an implementation of storage management that provides flexibility by giving the option to allocate storage based on system-defined parameters. In this case, we consider a memory hierarchy that allows to partition the memory resources arbitrarily amongst several nodes depending on the need. This memory can be dynamically reconfigured at a cost. We provide algorithms that pre-allocate or reconfigure dynamically the disaggregated memory based on estimated needs. We provide theoretical performance results for these algorithms. An important contribution of our work is that it shows that the system can design allocation algorithms even if user memory estimates are not accurate, and for dynamic memory patterns. These algorithms rely on statistical behavior of applications. We observe the impact on the performance of parameters of interest such as the reconfiguration cost.**

***Index Terms*—Memory Disaggregation, High performance computing, Scheduling, Stochastic model.**

I. Introduction

To reach peak performance, node memory on High-Performance Computing systems (HPC) is usually high, designed to be able to cope with most applications. Yet, data shows that HPC systems generally underutilize memory. For instance, Peng et al. [1] observed on a large scale study of four HPC clusters at Lawrence Livermore National Laboratory, that 90% of the time a node utilizes less than 35% of its memory capacity. By studying traces of the Marconi-100 supercomputer [2], we observed a higher memory utilization. Still, Marconi100 uses up to 50% of its total memory for half of its operation time.

This underutilization of resources has a high cost in terms of machine construction. To give orders of magnitude, today 256GB of RAM costs about 2000USD [3]. The Marconi100 supercomputer [4] had 920 nodes, each with 256GB of RAM. This leads to an estimated cost of RAM in the order of 2 million euros. Wahlgren et al. [3] estimate the node memory cost (DDR4+HBM2e) on Frontier to $170 million out a global cost of $600 million. All these motivate memory size reduction.

Recent architectural advances have included the use of disaggregated memory (such as CXL [5]). The idea behind it is to semi-dynamically (provided various costs such as a reconfiguration cost) share and allocate memory for the compute nodes, adjusting the allocated memory to the actual needs of the applications. Disaggregated memory would therefore help to reduce the total volume of memory consumed by the HPC resource.

In this work we discuss algorithmic solutions for efficient disaggregated memory usage. We consider an architecture with multiple tiers of memory/storage, such that when an application does not have enough *first tier* memory available (which we call in the following *node memory*), it needs to access a much slower memory which slows down its performance. We aim to answer the following question: given an implementation of disaggregated memory on a system with multiple memory-tiers, how do we allocate the first-tier of memory between competing applications? To be able to design these algorithmic solutions, we assume that we have access to some knowledge on application behavior. This knowledge can be precise (the memory footprint for the next x units of time), or statistic, based on historical data. For instance, we plot in Figure 1 the distribution of memory utilization per node in May 2022 on Marconi100. Similar statistical studies have been performed on other systems such as Quartz and Lassen at LLNL [6].

We consider the problem of disaggregated memory in a much larger scheme of multidimensional HPC Resource Management, where the resource manager has to allocate the applications on the compute nodes, and partition the extra memory amongst the running applications.

More specifically, this work presents the following contributions:

- We present two novel algorithmic strategies with guaranteed performance when not considering the reconfiguration cost, one of which is using as input a statistical description of resource usage;
- We show that these solutions can help reduce largely the memory needed by an HPC machine while keeping the

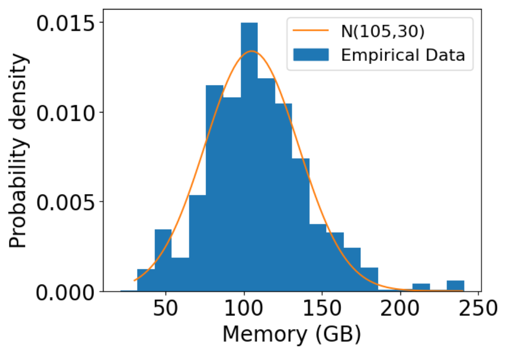


Figure 1: Distribution of the per node memory utilization in Marconi100 dataset 8 (May 2022) [7]. Marconi100 has 256GB of RAM available per node (242 usable) [4].

same performance, including in cases where the memory pattern of an application is unknown;
- We demonstrate their performance via a thorough evaluation, including in limit cases.

Our work focuses on the robustness and performance of the proposed algorithmic strategies and is complementary to those focusing on practical/technical implementations [3], [8]. We observe that an implementation of disaggregated memory with our algorithms could theoretically halve the memory usage of HPC machines with insignificant losses of performances.

We organize the rest of this paper as follows: we provide some related work in Section II. In Section III, we propose a mathematical formulation of the scheduling problem, and propose algorithmic solutions along with proofs of their optimality for certain objectives in Section IV. We design an evaluation framework based on real traces in Section V and evaluate our solutions compared to baselines in Section VI. Finally, we conclude and open discussion on the next steps.

II. Related Work

The need for memory heterogeneity or disaggregated memory on HPC resource has been demonstrated by many work that study various workload. Peng et al. [6] have studied HPC systems workloads. They have shown the heterogeneity of workloads where 80% of the workloads of both the Lassen and Quartz systems use less than 25% of the available memory. In parallel, they have observed the emergence of memory-intensive workloads. By showing a correlation between jobs with high memory consumption and large jobs on a system with low memory per node, and showing the absence of this correlation on a system with high memory per node, they have intuited that some applications reserve more nodes to have more available memory, and hence wasting compute node power.

a) Implementation of Disaggregated Memory: Implementations of disaggregated memories belong into several categories: hardware based such as CXL [5], [9], RDMA [10], [11], Infiniband [12] or RoCE [12], and software based (or logical disaggregation).

In addition to technological implementation, there are several logical disaggregation modes (i.e. software based) that have been implemented. Recently, Copik et al. [8] have proposed a software-based implementation for HPC systems based on Function-as-a-Service paradigm to utilize idle resources while retaining near-native performance.

Several works have discussed what would be expected from a fully functional disaggregation system [13]. Other works focus on the practical challenge such an implementation could face [3]. This paper approaches resource disaggregation under an algorithmic point of view. We assume that fully functional disaggregation systems are theoretically available.

Our work focuses on cluster-wide memory disaggregation. Cluster-wide memory disaggregation adds the challenge of keeping track of the location of data [13]. In practice, this is done by updating a memory map (which implies a reconfiguration cost). To deal with the scalability challenge of maintaining this map (and keep this cost to a minimum), solutions for large size cluster include hierarchical constructions, by constructing smaller groups of nodes that share the resources [14].

b) Allocation algorithms for competing applications: Most of the allocation algorithms for disaggregated memory consider the case where memory is sufficiently available for all applications running in the system. This is particularly true for single node applications/OS level scheduling [15]. In such a case, the scheduling problem consists of deciding which data goes where, depending on factors such as the frequency [15] or the proximity [16] of the data accesses.

These node-level memory disaggregation solutions are out of the scope of our study. Here we are interested in memory disaggregation at the cluster level. With respect to cluster-based scheduling algorithms, we do not know related work for the HPC decision problem.

The algorithmic solutions that we are looking for are closer to those from the Cloud Computing community, with the main difference that in cloud system, the applications cannot be slowed down because they have a quality of service to match. Applications are looked at independently of the rest of system, for instance Rzadca et al. propose Autopilot, a ML-based solution to predict how much memory to allocate dynamically to each job [17]. The goal is then to minimize the cost associated to adding more resources (optimization problem), and the solutions in elastic memory/shared memory are often more technical (extra memory is available, how do we get access to it), rather than decisive (who gets most memory).

In our case, we are interested in a problem where resources are bounded, and need to be shared with competing applications (decision problem). To the best of our knowledge, we did not find explicit algorithmic solutions to solve this decision problem.

III. Preliminary Concepts

A. Disaggregated memory model

We consider an architecture with P compute nodes and a two tier memory. Typically, this corresponds to an archi-

Symbol	Meaning
P	Number of nodes
α	Bandwith ratio between the two tiers of memory. $\alpha \in]0,1[$
M	Size of the disaggregated memory
$\tau_{\texttt{alloc}}$	Reconfiguration time of the disaggregated memory
N	Number of applications
A_i	i^{th} application
γ_j	Ratio of completion of the application at which the phase j ends
m_j	Memory need during phase j of the application
T_i^{opt}	Completion time of A_i if it has all the memory it needs
$M_i(t)$	Memory allocated to A_i as a function of the time
$SL_i(M_i, j)$	Slowdown ratio of A_i, when it is in phase j and has M_i memory allocated
$U(S)$	Utilization (or mean throughput) of a schedule
$\rho(t)$	Throughput at time t

Table I: This table summarizes the symbols and notations which are used in the paper.

tecture with a shared NVME and I/O storage. The first tier (disaggregated memory) is shared between the nodes and faster than the second tier. α ($\in]0,1[$) represents the ratio of bandwidth between these two tiers of memory. $\alpha \approx 0$ corresponds to the second tier being extremely slow, while $\alpha = 1$ corresponds to both memory tiers having the same bandwidth.. The shared disaggregated memory of size M can be partitioned between the different nodes. There is one *reconfigure* operation which can change how much memory each node is provided. This reconfiguration has a cost; during a time $\tau_{\texttt{alloc}}$ it is impossible to use the memory that is allocated and/or desallocated. This model represents an hypothetical HPC machine using disaggregated memory. Such machines do not exist yet.

B. Application model

We consider N parallel applications $\{A_1, \dots, A_N\}$. Each application can run using one of the three processing modes: (1) with no access to the disaggregated memory at limited speed, (2) at full speed with complete access to disaggregated memory, or (3) a trade-off with limited access to disaggregated memory. All applications have access to the slower memory. For an application A_i, we characterize its memory profile as a function of the requested memory: $\gamma \mapsto m_i(\gamma)$ with $\gamma \in [0,1]$ the proportion of the application completed. Note that in general this profile is an unknown variable that can only be traced after execution (see Section III-D).

To model the application memory consumption, we first consider the memory profile as piecewise constant. We divide the memory profile into successive phases, with constant memory profile during each phase. A memory profile is therefore denoted as a set $\{(\gamma_j, m_j), 0 \le j < J\}$ where J is the number of phases of the application, m_j is the memory request on phase j, that start when a proportion γ_{j-1} of the application is executed, and finishes at ratio γ_j of the application ($\gamma_0 = 0$ and $\gamma_{J-1} = 1$).

Performance model: The performance of application A_i is computed based on the allocation M_i of memory allocated to A_i: If the application runs with requested memory at each phase, then the runtime of phase j will be $T_i^{\text{opt}}(\gamma_j - \gamma_{j-1})$ and the total runtime of the application will be T_i^{opt}.

With a fixed memory $M_i \le M$, a slowdown affects the runtime of each phase. In general this slowdown is very application dependent, but some data [13], [14] show that a linear slowdown on memory accesses may be a good approximation. Particularly, Liu et al. [13, Figure 7] have shown it using several memory swapping systems (Linux, Infiniswap, Fastswap). We can observe that the main difference in performance lies in an architecture-dependent growth factor. This is coherent with the roofline model [18], [3]. Hence, we compute the slowdown on phase j as follows:

$$SL_i(M_i, j) = \left(\alpha + (1-\alpha)\min\left(1, \frac{M_i}{m_j}\right)\right).$$

Discussion: the underlying hypothesis behind this model is that all memory accesses are accessed with the same frequency. This is the case for some applications such as HPL/SuperLU [3]. For some applications where the inbalance between the memory blocks that are accessed is extremely important (NekRS, BFS, XSBench [3]), this model could also work by considering as main *memory footprint, only the blocks that correspond to 90% of the memory accesses and considering a first order approximation.*

We only model a slowdown due to not having enough memory available: in line with recent literature on disaggregated memory systems in HPC [3], we consider that access to this shared memory does not impact the bandwidth in general.

With this formula, an application that has all the memory it needs (or more) will have a slowdown of 1 (i.e., no slowdown). An application with no memory will have a slowdown of α (the bandwidth ratio between fast and slow storage). An application with βm_j memory ($\beta \in [0,1]$) will have a slowdown of $\alpha + (1-\alpha)\beta$, which is the linear interpolation between the slowdown with no memory and the slowdown with all the memory.

We obtain a runtime for phase j:

$$r_i(M_i, j) = \frac{T_i^{\text{opt}}(\gamma_j - \gamma_{j-1})}{SL_i(M_i, j)}. \quad (1)$$

More generally, an application can be run with successive memory allocations $\{(\gamma'_k, M_k), 1 \le k \le K\}$ during phase j with $M_k \le M$ and $\gamma'_K = \gamma_j$ and $\gamma'_0 = \gamma_{j-1}$, where memory M_k is allocated to the application when the application is between γ'_{k-1} and γ'_k of its execution. Then, the execution time of the phase will be

$$\sum_{k=1}^{K} \frac{T_i^{\text{opt}}(\gamma'_k - \gamma'_{k-1})}{SL_i(M_k, j)}.$$

The execution time of the application will thus be the sum of execution of each phase. For better clarity, we formulate in the following the memory allocation as a function of the time.

C. Scheduling problem

Given a number of nodes P, a fast, disaggregated memory of size M, a relative bandwidth ratio α between fast and slow memory, and a set $\{A_1, \cdots, A_N\}$ of applications defined by: their memory profile $m_i(\gamma)$, their node request c_i, a minimal execution time T_i^{opt} with unlimited memory.

A schedule consists in allocating applications to the P nodes, and partitioning the memory between applications. The memory allocation of a schedule can be described as a time vector $t \mapsto \pi(t) = (M_1(t), \ldots, M_N(t))$ s.t. for all t, $\sum_{i \leq N} M_i(t) \leq M$.

Optimization objective: In line with the literature [19], we first define the throughput or instantaneous utilization of the system. More specifically, given a set of applications $S_t = \{A_1, \cdots, A_k\}$, if at time t applications A_i uses c_i cores, then the throughput ρ is:

$$\rho(t) = \sum_{A_i \in S_t} c_i$$

Due to low memory allocation, an application can be slowed down, hence we use the or 'useful' throughout. It consists in giving to each node a weight proportional to the quantity of work it is effectively processing. We use this modified version of the metric to account for the fact that if an application is executed on a node with a slowdown of 0.5, then the node run at half of its maximal capacity, hence, only half of the node counts as producing 'useful' work. If application A_i is running phase j_i with memory M_i, then the *useful* throughput is:

$$\rho(t) = \sum_{A_i \in S_t} c_i \cdot SL_i(M_i(t), j_i).$$

Finally, we want to maximize the useful utilization of the system, defined as the mean useful throughput over time. The useful utilization U of the schedule S is therefore given by the formula:

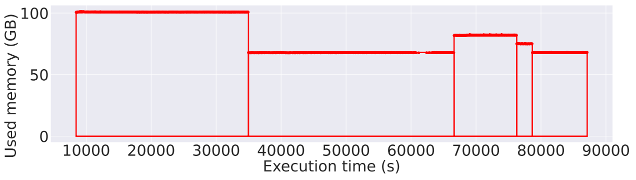
$$U(S) = \frac{1}{T_S} \sum_t \sum_{A_i \in S_t} c_i \cdot SL_i(M_i(t))$$

where T_S is the makespan of the schedule. For simplicity, in the rest of this work we use *utilization* when we talk about *useful utilization* and *throughput* when we talk about *useful throughput*.

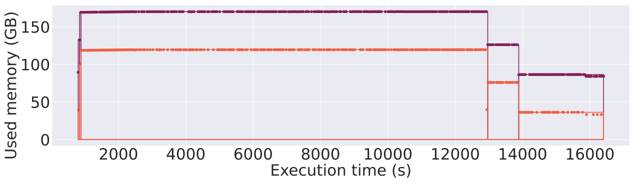
D. Limits of Memory Models

While an application can be described by a memory profile $t \mapsto M(t)$, it may not always be reasonable to consider that we can accurately obtain this profile in practice [20]. There are too many variables that impact the accuracy of this profile: the performance of the system (the CPU may be slower at some time which translates the memory function in time), the shape of the data etc.

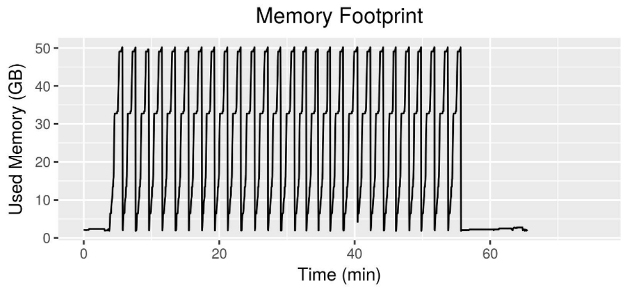
Storing an entire memory profile for applications running on a large scale machine may be extremely costly. For instance, if we collect the node memory consumption every second (stored on 4B), that is roughly 350KB/day/node. On a machine like Frontier with more than 9400 nodes, this comes to 3.2GB/day,



(a) Memory footprint of a Marconi100 Job with phased memory pattern. The lines represent the interpolation and the points are the data.



(b) Memory footprint of a Marconi100 Job with various footprints per phase, one footprint per computing node. The lines represent the interpolation and the points are the data.



(c) Memory footprint of a Neuroscience job with a dynamic memory pattern [21].

Figure 2: Different flavors of memory behavior

more than 1.2TB/year of memory profile. Because storing fine-grained memory profiles can be prohibitive in terms of data footprint, we need to consider lower data footprint representations of a memory profile.

In comparison, over five years of Mira (Jan 2014 to Dec 2018), 330k jobs were executed [19]. If we ran aggregated data per jobs (for instance the percentage of time spent using 0-8GB, 8-16GB, 16-32GB, 32-64GB, 64-128GB [1]), that would amount to 6.6GB. Of course, this comes with a loss of information.

In addition to the inaccuracy of the measurement, Peng et al. [6] identified four types of memory temporal patterns to describe applications:

- Constant pattern: the memory utilization has minimal changes throughout the execution;
- Phased pattern: the memory consumption is constant by segments;
- Dynamic patterns: a pattern with frequent and substantial changes in memory usage over time;
- Sporadic patterns: low memory utilization most of the time with spiked memory usage for short periods.

In the evaluation we consider two flavors of input that the algorithm can use, that would represent various means of collecting memory information:

1) Constant memory profiles: Our observation from Marconi100's traces [7] is that some applications have very structured behavior (see Figure 2a). In cases where phases with constant memory usage are quite long in front of the reconfiguration cost, this cost can usually be neglected.

2) Stochastic memory profiles: Finally, there are jobs whose memory behavior varies dynamically during its execution (see for instance Fig. 2c). This is typical of jobs relying on Deep-Learning phases. This variability can be random or deterministic but with variations that are too fast in front of the time to reconfigure the machine. For these jobs, we propose to use a stochastic model to describe the memory consumption. This model can be obtained based on historical data.

Definition 1 (Stochastic Memory Profile)**.** We say that an application has a stochastic memory profile $(m_j, p_j)_j$ on time interval $[l, l+1]$, with $\sum_j p_j = 1$ and $m_{j_0-1} < m_{j_0}$ for all $j_0 > 0$, if for $t \in [l, l+1]$, $t \mapsto m(t)$ is equal to X where X is a discrete random variable of distribution $(m_j, p_j)_j$ (i.e. $\mathbb{P}(X = m_j) = p_j$)).

Some comments: we focus on a simple definition, but it could be generalized trivially to a model where the random variable is time dependent (phase behavior). Similarly, the duration of the time interval where the memory is constant could be non-deterministic.

One can verify that, for application A with node request c and memory profile $(m_j, p_j)_j$, if $m_{j_0} \le M_0 < m_{j_0+1}$, then we have the following properties:

$$\mathbb{E}\left(\rho_A(M_0)\right) = c\alpha + c(1-\alpha)\sum_{j \le j_0} p_j + c(1-\alpha)M_0 \sum_{j > j_0} \frac{p_j}{m_j}$$

$$\begin{aligned}\mathbb{E}\left(\rho_A(M_0)\right) &- \mathbb{E}\left(\rho_A(m_{j_0})\right) \\ &= c(1-\alpha)(M_0 - m_{j_0})\sum_{j > j_0}\frac{p_j}{m_j} \\ &= (1-\alpha)\cdot(M_0 - m_{j_0})\cdot g(m_{j_0}) \qquad (2)\\ \mathbb{E}\left(\rho_A(m_{j_0+1})\right) &- \mathbb{E}\left(\rho_A(M_0)\right) \\ &= c(1-\alpha)(m_{j_0+1} - M_0)\sum_{j > j_0}\frac{p_j}{m_j} \\ &= (1-\alpha)\cdot(m_{j_0+1} - M_0)\cdot g(m_{j_0}) \qquad (3)\end{aligned}$$

where, for $m_{j_0} \le M_0 < m_{j_0+1}$,

$$g(M_0) = c\sum_{j > j_0}\frac{p_j}{m_j} = c\mathbb{E}\left(\frac{1}{X} \middle| \, X > M_0\right)\mathbb{P}\left(X > M_0\right) \quad (4)$$

IV. Algorithms

In this section we present various algorithmic strategies to solve the problem described in Section III-C. We propose two algorithms: the `Priority` algorithm, which is suited for application scenarios with known memory behavior, and the `Stochastic` algorithm, which is suited for application scenarios with dynamic memory requirements. These algorithms are compared to baseline algorithms presented in Section IV-C. Finally, we discuss how these algorithms impact the batch scheduling mechanism in Section IV-D.

A. `Priority`

The first algorithm is priority-based: at all time t_0, `Priority` sorts all running applications by decreasing $c_i/m_i(t_0)$. Then while there is memory available, it allocates it (up to the memory requirement) to the first application of the list that needs additional memory.

Theorem 1. *Given a set of running jobs $\{A_1, \cdots, A_N\}$, with respective constant memory profile $t \mapsto m_i(t) = m_i$. `Priority` is optimal with respect to the system throughput $\rho(t)$.*

Proof. We show the result by contradiction.

Assume an optimal solution $\pi = (M_1, \ldots, M_N)$, such that there exists two jobs A_1 and A_2, such that $c_1/m_1 > c_2/m_2$, and such that $M_1 < m_1$ and $0 < M_2$ $(\le m_2)$, then we show that there exists $\varepsilon > 0$ such that the allocation π' with $M'_1 = M_1 + \varepsilon$, $M'_2 = M_2 - \varepsilon$ and $M'_i = M_i$ for $i > 2$ has better performance.

Because the throughput is additive, one can notice that

$$\rho(\pi') - \rho(\pi) = (1-\alpha)\frac{c_1}{m_1}\varepsilon - (1-\alpha)\frac{c_2}{m_2}\varepsilon > 0$$

which contradicts the optimality of π. □

B. `Stochastic`

Based on Theorem 1, we expect `Priority` to work well when the memory profile of an application stays steady, close to the predicted peak usage as can be observed on traces on the Marconi100 supercomputer (see Section III-D1). For applications where the memory consumption is more dynamic within the execution [21], or where the predicted peak consumption can be far-off, there can be an important reconfiguration cost.

We provide another approach based on a stochastic memory profile (Definition 1). In the following we assume that the memory profile of applications are described by random variables X whose discrete distributions are known $(m_j, p_j)_j$ (i.e. for all j, $P(X = m_j) = p_j$).

In this case, we derive `Stochastic`, a strategy with loglinear complexity that minimizes the expected throughput when applications follow a stochastic memory model. In general this can be obtained with historic data or known behavior for recurrent jobs (see Section III-D).

In the following and for simplicity when the job index i is omitted when it is obvious.

Lemma 1. *Let X a random variable that follows the discrete distribution $(m_j, p_j)_j$, then for all j:*

$$g(m_j) \ge g(m_{j+1})$$

This lemma is trivial with Equation (4).

We now prove the main theorem that we use to define `Stochastic`. Intuitively Theorem 2 along with Lemma 1 state that we can sort applications by decreasing values $g(m_j)$, and allocate memory greedily up to the next value m_j.

Theorem 2. *Given a set of running applications $\{A_1, \ldots, A_N\}$ with stochastic memory profile. Given an optimal schedule $\pi = (M_1, \cdots, M_n)$ for the problem of maximizing the expected throughput, then this schedule satisfies the following property.*

For all i_1, i_2, let j_1 (resp. j_2) s.t. $m_{j_1}^{(i_1)} \le M_{i_1} < m_{j_1+1}^{(i_1)}$ (resp. $m_{j_2}^{(i_2)} \le M_{i_2} < m_{j_2+1}^{(i_2)}$), then:

$$g_{i_1}\left(m_{j_1}^{(i_1)}\right) < g_{i_2}\left(m_{j_2-1}^{(i_2)}\right).$$

Proof of Theorem 2. We show the result by contradiction. Given $\pi = (M_1, \cdots, M_n)$ an optimal schedule, assume that there exists

$$\begin{array}{ll} m_{j_1}^{(1)} \text{ such that} & m_{j_1}^{(1)} \le M_1 < m_{j_1+1}^{(1)} \\ m_{j_2}^{(2)} \text{ such that} & m_{j_2}^{(2)} \le M_2 < m_{j_2+1}^{(2)} \end{array}$$

and $g_1(m_{j_1}^{(1)}) > g_2(m_{j_2-1}^{(2)}) \ge g_2(m_{j_2}^{(2)})$.

We show that for

$$0 < \varepsilon \le \begin{cases} \min\left(m_{j_1+1}^{(1)} - M_1, M_2 - m_{j_2}^{(2)}\right) & \text{if } M_2 > m_{j_2}^{(2)} \\ \min\left(m_{j_1+1}^{(1)} - M_1, M_2 - m_{j_2-1}^{(2)}\right) & \text{if } M_2 = m_{j_2}^{(2)} \end{cases}$$

the schedule $\pi' = (M_1 + \varepsilon, M_2 - \varepsilon, \cdots, M_n)$ has a better expected throughput than π.

$$\begin{aligned} \mathbb{E}(\rho(\pi)) - \mathbb{E}(\rho(\pi')) = {} & \mathbb{E}(\rho_{A_1}(M_1)) - \mathbb{E}(\rho_{A_1}(M_1+\varepsilon)) \\ & + \mathbb{E}(\rho_{A_2}(M_2)) - \mathbb{E}(\rho_{A_2}(M_2-\varepsilon)) \end{aligned}$$

Using Equation (2) and (3):

$$\begin{aligned} & \mathbb{E}(\rho_{A_1}(M_1)) - \mathbb{E}(\rho_{A_1}(M_1+\varepsilon)) = -(1-\alpha)\cdot\varepsilon\cdot g_1(m_{j_1}^{(1)}) \\ & \mathbb{E}(\rho_{A_2}(M_2)) - \mathbb{E}(\rho_{A_2}(M_2-\varepsilon)) \\ & \qquad = \begin{cases} (1-\alpha)\cdot\varepsilon\cdot g_2(m_{j_2}^{(2)}) & \text{if } M_2 > m_{j_2}^{(2)} \\ (1-\alpha)\cdot\varepsilon\cdot g_2(m_{j_2-1}^{(2)}) & \text{if } M_2 = m_{j_2}^{(2)} \end{cases} \\ & \qquad \le (1-\alpha)\cdot\varepsilon\cdot g_2(m_{j_2-1}^{(2)}) \qquad \text{(Lemma 1)} \end{aligned}$$

Hence

$$\begin{aligned} \mathbb{E}(\rho(\pi)) - \mathbb{E}(\rho(\pi')) & \le (1-\alpha)\cdot\varepsilon\left(g_2(m_{j_2-1}^{(2)}) - g_1(m_{j_1}^{(1)})\right) \\ & < 0 \end{aligned}$$

Contradicting the optimality hypothesis. □

Algorithm 1 is derived from Theorem 2. It initially allocates 0 memory to each application. Then, in increasing order of $g_i(m_j^{(i)})$, it increases the memory allocation of each application up to the next value of the memory distribution until the memory limit is reached or until all application met their maximum memory value.

Algorithm 1 `Stochastic` $(A_1, \ldots, A_N, M)$

```
1: Avail_M ← $M$
2: $G$ table of size $N$, initialized with $G[i] = c_i \cdot \sum_{j\ge1} p_j^{(i)}/m_j^{(i)}$.
3: $V$ table of size $N$, initialized at 0.
4: $H$ heap of applications sorted by decreasing value of $G$
5: $\pi$ table of size $N$, initialized at 0.
6: while Avail_M $> 0$ OR $H$ is empty do
7:     $i \leftarrow \text{pop}(H)$
8:     $j \leftarrow V[i]$
9:     $M_t \leftarrow \pi[i]$
10:    $\pi[i] \leftarrow \min\left(m_{j+1}^{(i)}, M_t + \texttt{Avail\_M}\right)$
11:    Avail_M ← Avail_M $- (\pi[i] - M_t)$
12:    $G[i] \leftarrow G[i] - c_i \cdot p_{j+1}^{(i)}/m_{j+1}^{(i)}$
13:    $V[i] \leftarrow V[i] + 1$
14:    if $G[i] > 0$ then insert$(H, i)$ end if
15: end while
16: return $\pi$
```

C. Others heuristics

We compared `Priority` and `Stochastic` with several heuristics:

- `Aggregated`, the baseline heuristic, behaves like a solution would on a machine with aggregated memory, i.e. the memory allocation is proportional to the number of compute resources used by the application.
- We also compare to other priority-based heuristics (i.e., heuristics that sort jobs by a priority function and allocates the maximum between the available memory and what the job requires), namely:
  - `Oldest-First`: sorts jobs by increasing arrival time (i.e. the date in which the job entered the system);
  - `Largest-First`: sorts jobs by decreasing size.

D. Batch Scheduler Integration

In our work we considered a batch-scheduler relying on EASY-BF [22]. At submission time, users are expected to provide an expected walltime to the batch scheduler. The usual impact of this expected walltime is that if a job lasts longer than this value, it is killed by the resource manager. In our implementation, we chose to separate the memory partitioning from the node allocation, in order to simplify the overhead. This means that if the expected walltime is shorter than the worse case (T^{opt}/α), the job is at the risk of being killed. Hence, a batch scheduler needs to use the worse possible walltime as the expected walltime.

The focus of this work was to compare different memory allocation strategies, and determining the impact of disaggregated memory. In future work, it may be interesting to study the co-design of Batch-Scheduling strategies with memory partitioning algorithms, this co-design could use better runtime estimates.

V. Evaluation Methodology

In order to generate traces for the evaluation, we rely on real behavior. We use application traces from the Marconi100 supercomputer [2]. The Marconi100 supercomputer consisted of 980 computing nodes, each of which having 2x IBM POWER9 AC922 (32 cores), 4x NVIDIA Volta V100 GPUs, and 256GB of RAM. In Section V-A, we explain how we extract stochastic profiles. These profiles are then used to generate synthetic traces as described in Section V-B.

For the evaluation we designed an event-based simulator based on the model designed in Section III-A. This simulator is available freely at https://doi.org/10.5281/zenodo.13981594. This simulator takes in entry the memory profiles of the applications, the platform parameters (number of nodes and quantity of disaggregated memory) and a memory allocation strategy. It then performs the simulation and return some parameters of interests such as the quantity of memory used at each event or the completion time of each application.

A. Extracting stochastic memory profiles

To extract stochastic memory from real workload data, we sampled 400 jobs from dataset 8 (2022-05) of Marconi100 [2]. The sampling method used mainly two criteria to select jobs: select jobs that (i) run for more than one minute, and (ii) had exclusive access to the computing nodes.

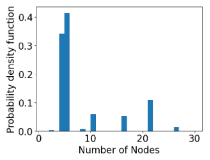
For each sampled job, its memory footprint consists of a time series of measurements of the nodes' memory consumption. From this time series, we define the phases frontiers in three steps. First, we calculate the distribution of memory consumption differences between subsequent measurements in the memory consumption time series. Second, we use this distribution to calculate the z-scores of the memory consumption differences. Third, from these z-scores, we define the phase frontiers as the timestamps where the z-scores of the memory consumption differences were larger than a threshold Δ, expressed as a ratio of the standard deviation.

Put another way, we set the phases frontiers where the memory consumption significantly went up or down, where "how much significantly" is determined by Δ. Finally, for each phase determined by two subsequent frontiers, we determine its memory consumption as being the maximum memory consumption of the initial memory consumption time series during the duration of the phase.

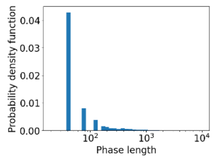
In practical terms, we reduce a fine-grained time series of memory consumption into a coarse-grained sequence of memory consumption phases, where we only store new information when the memory consumption significantly changed. Figures 2a, and 2b illustrate some outputs of the above procedure. In these figures, the points represent the data and the lines the output.

Based on the obtained phases, we generated four distributions of behaviors on Marconi100 that we use for synthetic trace generation:

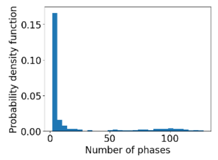
- a node distribution C_m (see Figure 3a). We have $\mathbb{E}(C_m) = 7$.



(a) Distribution of number of nodes per applications on Marconi100.



(b) Distribution of phase length distribution on Marconi100.



(c) Distribution of number of phases per applications on Marconi100.

Figure 3: Job parameters on the Marconi100 dataset that we studied.

- a phase length distribution L_m (see Figure 3b which is showed truncated). We have $\mathbb{E}(L_m) = 1000$s.
- a distribution of number of phases N_p (see Figure 3c). We have $\mathbb{E}(N_p) = 17$.
- a memory usage distribution X_m (see Figure 1). We have $\mathbb{E}(X_m) = 105$GB.

B. Generating synthetic traces

In all the experiments we consider that there are $P = 54$ nodes. Unless specified otherwise, we used as machine parameters: $\tau_{\texttt{alloc}} = 1$s and $\alpha = 0.03$. We chose $\alpha = 0.03$ as the ratio between the speed of a SSD at 300MB/s versus a RAM at 10GB/s. The available memory M depends on the experiment, but we call $\bar{M} = \mathbb{E}(X_m) = 105GB$ the average memory usage of Marconi100's traces, and $M_{\text{mar}} = 256GB$ the memory per node on Marconi100.

Unless specified otherwise, the applications are generated as follows: there are 30 batches of $N = 1000$ applications. For each application A_j:

- Its number of nodes is selected following C_m
- The number of phases is selected following N_p (where we have bounded the number of phases at 45), and for each phase:
  - Its memory is selected following X_m
  - Its length is selected following L_m

The release time of each job is 0 for the first 10% of jobs, then each job is released $0.9 \cdot 10\mathbb{E}(L_m)\frac{C_m}{P}$ units of time after the previous one. This release ratio is to guarantee that there is always enough work to be executed. Intuitively, if there was no scheduling constraints, the jobs released after $t = 0$ could allow a theoretical utilization of 90%.

C. Measuring Performance

When measuring the performance of an algorithm, we consider an interval of time included in the workload generation interval. That is, if the last application is submitted at time T_{last}, we measure the system utilization on the interval $[0, T_{\text{last}}]$. In practice this means that the executed workload is not the same for all solutions, but when we take T_{last} to be large enough, we reach a steady state which makes the solution trustable [19].

VI. Experiments

We first evaluated the impact of disaggregated memory allocation solution in *simple* cases where the memory is constant by phases (Section VI-A). Next, we evaluated the limits of the online strategy `Priority` when the memory pattern changes frequently. Finally, we evaluated the importance of `Stochastic` (Section VI-B) to alleviate the cost of reconfiguration. In this section we refer to the term *baseline performance* as the performance on an architecture where memory is not limited (in this case this is satisfied by 256GB per node).

A. Evaluation on Phased patterns

Using the nomenclature proposed by Peng et al. [6] we evaluated our algorithms on *Phased* patterns, that is patterns where memory is constant by segments. The number of segments can be equal to one which corresponds to the *constant* patterns. We first provided a general evaluation based on Marconi100 job profile, then we studied the limits when the architecture parameters vary. In this section, we slightly modified the trace generation by drawing the number of nodes per applications uniformly between 1 and 23. This provides more variability in the applications to observe more differences in the results. However our take-aways hold with the original Marconi100 node distribution.

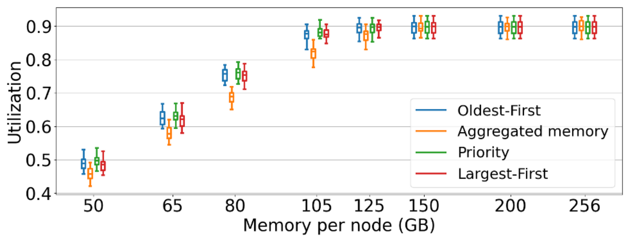


Figure 4: Utilization of different algorithms.

1) General evaluation: In Figure 4, we measured the mean *useful* utilization (i.e. equivalent to the number of Flops) when the average memory per node varies. Since all applications use less than 256GB of memory per node, all allocation policies give similar results with 256GB (or any volume of memory larger, what we consider in the following *baseline*/unlimited memory case). The difference in behavior between `Aggregated` and disaggregated heuristics is that in our implementation, disaggregated heuristics still release unused memory and reallocate it later when needed.

When the memory becomes more limited, disaggregated heuristics behave better than the `Aggregated` that uses the same volume of memory. Our result showed that using 150% of the average occupied memory (150GB), then using disaggregated heuristics allow to have roughly the same performance to that of a machine where memory is not limited (loss of 0.2%). In this scenario, heuristics that do fewer reconfiguration (`Oldest-First`, `Largest-First`) perform slightly better than `Priority`.

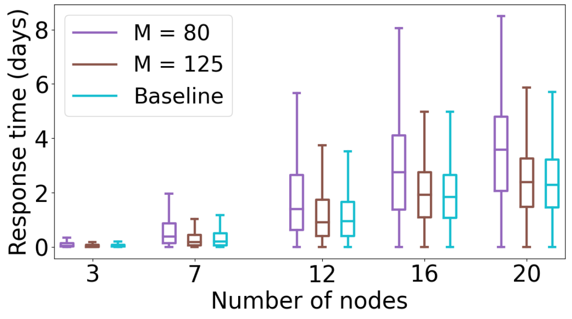


Figure 5: Impact of the response time of `Priority` as a function of the memory available.

When the average memory per node gets closer to the average memory usage (125GB), then as expected by our theoretical results `Priority` outperforms other heuristics. The overhead compared to the *baseline* solution is still minimal with 125GB (i.e. the machine has 25% more memory than the average memory usage): there is a 0.7% utilization loss compared to the baseline on average. Finally, the `Priority` heuristic really shows its performance with very limited memory on the machine compared to the memory needed by applications. In these case it allows to gain up to 2% of utilization over `Oldest-First` and nearly 4% over `Largest-First`.

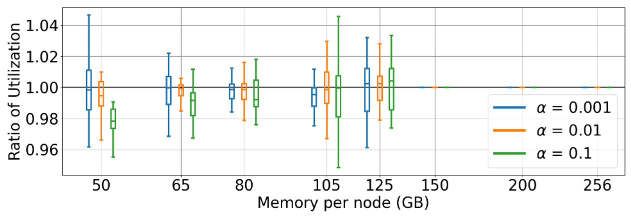
The response time is the average duration between a job submission and its completion. In Figure 5, we showed the response time of `Priority` for various job sizes, when the available memory varies compared to the baseline strategy. This allows to confirm that no jobs are arbitrarily hurt by our scheduling heuristic, even when the memory is small.

Take-aways:
1) Disaggregated heuristics allow to considerably reduce the memory needed by the machine.
2) When the memory available is small, then a more precise heuristic like `Priority` is important, however otherwise, simpler heuristics like `Oldest-First` are sufficient and do not require precise memory requirements.

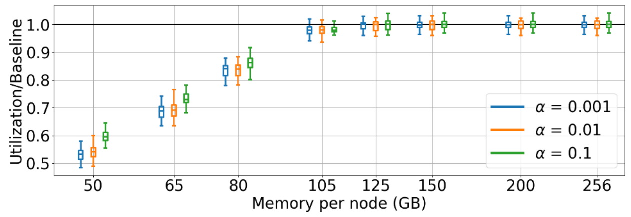
2) Impact of machine parameters: In this section we are interested by how architectural parameters impact the performance of disagregated algorithms. Specifically we evaluated the impact of α, the ratio between memory bandwidth and out of node storage bandwidth, and $\tau_{\texttt{alloc}}$, the reconfiguration time. In the following of this section, instead of an absolute value of the utilization, we study relative values, that is, the difference with the baseline utilization when memory is considered as unlimited (i.e., `Aggregated` with 256GB).

In Figure 6, we plotted the mean utilization as a function of the memory for various values of α ($\alpha = 0.1$ means that memory bandwidth is 10 times faster than external storage).

We perform two evaluations: in Figure 6a, we eval-

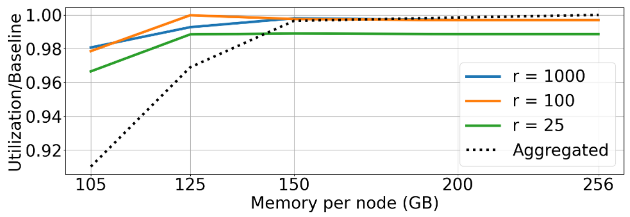


(a) Relative performance of `Oldest-First` and `Priority`. A performance higher than 1 means that `Oldest-First` performs better, lower than 1 than `Priority` performs better.



(b) Performance of `Priority` compared to baseline with 256GB of memory per node.

Figure 6: Machine utilization as a function of α.



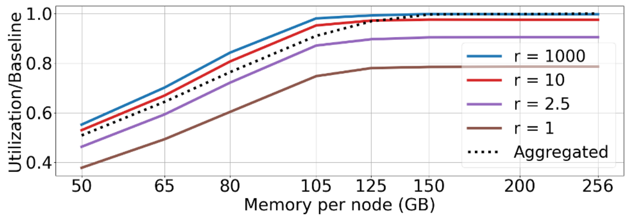


Figure 7: Average utilization of `Priority` (normalized by baseline utilization) when $r = \mathbb{E}(L_m)/\tau_{\texttt{alloc}}$ varies.

uated the relative performance between `Priority` and `Oldest-First` when α varies. This evaluation confirms our previous findings that when there is enough memory, a solution that minimizes the number of memory reallocation by giving priority to longer running jobs is better, however, as memory becomes limited, `Priority` improves this performance. It should be noted that in both cases, the relative gains are negligible (about 1%).

In Figure 6b, we evaluated the relative performance of `Priority` compared to the baseline architecture (256GB memory per node). Naturally, the higher α is, the better performance disaggregated solutions obtain. From the results in Figure 6, we observed a robustness of disaggregated heuristics.

Take-aways:

3) Even in limit cases where the cost of *overflowing* the memory is high, previous take-aways hold.

Our next step is to evaluate the impact of $\tau_{\texttt{alloc}}$. In practice, this characteristic time of the system is interesting compared to the characteristic time of applications (i.e., the average phase length). This is what we showed in Figure 7.

We increased the value of $\tau_{\texttt{alloc}}$ gradually, and plotted as a function of the ratio $r = \mathbb{E}(L_m)/\tau_{\texttt{alloc}}$, for $r \in \{1, 2.5, 10, 25, 100, 1000\}$ (hence, $\tau_{\texttt{alloc}} \in \{1000, 400, 100, 40, 10, 1\}$).

We observed that `Priority` stays efficient until $\mathbb{E}(L_m)/\tau_{\texttt{alloc}} = 100$. The difference between $r = 100$ and $r = 1000$ are negligible. As this ratio decreases we observed the following:

- When $r = 25$, the reconfiguration cost becomes so important that when memory is not limited (memory per node greater than 200GB). In this case, an aggregated strategy is better;
- When the reconfiguration cost has the same order of magnitude than the phase length, even with small volumes of memory it is better to use an aggregated storage.

These observations hint that an online reconfiguration solution may not be adapted to a scenario with dynamic memory pattern which is what we study in the next section.

B. Evaluation on dynamic patterns

In this section we are now interested by dynamic memory patterns, such as in Figure 2c. In order to do so, we update the generation protocol as follows:

- The number of phases is drawn uniformly between 50 and 149, but all phases have length $\tau_{\texttt{alloc}}$;
- For each application, we draw uniformly at random one variable out of 4 to describe their memory patterns in the following truncations of the normal law of mean 105 and scale 30:
  - A truncation between 30 and 80;
  - A truncation between 80 and 130;
  - A truncation between 130 and 180;
  - A truncation between 180 and 240.

  This statistical behavior is the only memory information available to the scheduler ahead of time.
- All phases memory from one application are drawn according to the selected distribution (this information is used for the evaluation, but not for the scheduling decision).

1) General evaluation: The results are presented in Figure 8. We compared `Priority`, `Stochastic` and `Aggregated`, and we normalized their performance to the baseline architecture with 256GB memory per node. As expected, `Priority` is outmatched by the two others policies. Indeed, as the length of a phase is equal to the reconfiguration time, it can no longer keep up with the fast-changing memory profile. `Stochastic` is able to correctly balance the memory

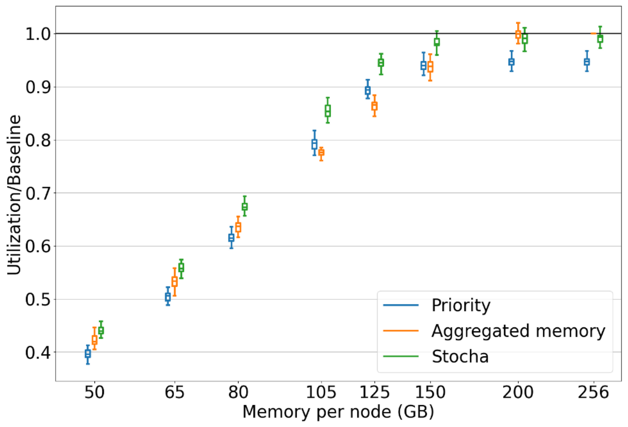


Figure 8: Utilization (normalized by baseline) of different algorithms when memory per node varies for dynamic patterns.

allocation between the application which allows it to perform better than `Aggregated` when the memory is constrained.

For each quantity of memory per node, the `Aggregated` policy performed as well in this case as in the first experiment. This is not surprising as this policy is independent of the variations speed of memory consumption of the application. Unsurprisingly and due to the dynamicity of patterns, the best performance cannot reach the same level of utilization that the `Priority` algorithm was able to reach in the first experiment (80% of utilization for `Stochastic` with 105 GB of memory versus 85% for `Priority` in the first scenario). Yet, those performance are still impressive given the dynamicity and overhead dut to reallocation, where the `Stochastic` algorithm allows to reduce the memory per node by more than 40% (to 150GB) with only 2% loss in performance.

A small comments on the results: in very rare cases we can observe that `Stochastic` of `Aggregated` with less memory behaves better than the Baseline that uses 256GB (see for instance 200GB on Figure 8). After checking the Gantt chart and the results, this is an artifact of the global scheduling strategy and a consequence of their heuristical nature: slowing down jobs can reorder the order in which the jobs are executed, which in turns can provide improvement on the global performance.

Finally, by studying the response time as a function of the size of the jobs, we confirm in Figure 9, that in this case as well, no jobs are arbitrarily hurt by our scheduling heuristic even when the memory is small.

Take-aways:

4) Even with dynamic patterns, when the overhead of an online algorithm would be too high, the static algorithm `Stochastic` is able to provide important gains, simply by using the statistical memory behavior. For our traces, it brought memory consumption reductions by more than 40% while losing less than 2% of performance.

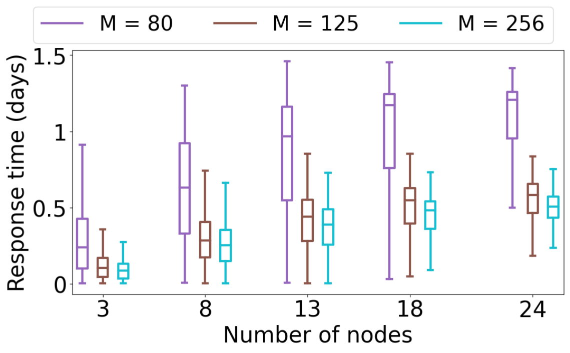
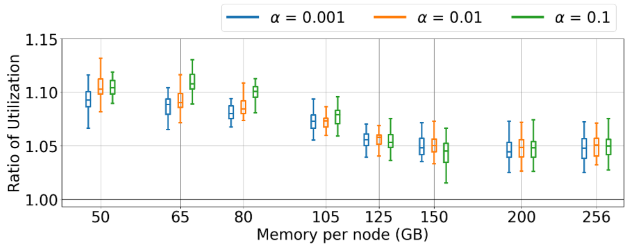
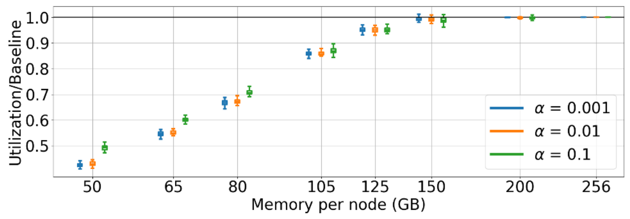


Figure 9: Impact of the response time of `Stochastic` as a function of the memory available.



(a) Relative performance of `Stochastic` over `Priority`. A performance higher than 1 means that `Stochastic` performs better, lower than 1 than `Priority` performs better.



(b) Performance of `Stochastic` compared to the *baseline*.

Figure 10: Relative performance for several values of α.

2) Impact of machine parameters: In Figure 10, we plot the results of the simulations for others values of α. In Figure 10a, we evaluate the relative performance between `Stochastic` and `Priority` when α and M vary. A performance higher than 1 means that `Stochastic` performs better, lower than 1 than `Priority` performs better. In Figure 10b we compared the performance of `Stochastic` compared to the *baseline*. Just like in the phased pattern use-case, these two experiments confirm that the performance of `Stochastic` are robust to others machine parameters.

Take-aways:

5) The results for `Stochastic` are robust to various access cost to the second tier of memory.

VII. Conclusion

Memory capacity is a critical point of HPC architecture. To cope with most applications, HPC systems classically over-supply this resource with high financial cost. Disaggregated memory has been proposed as a solution to provide shared memory to multiple nodes.

We present in this work a model of HPC with disaggregated memory and different strategies for memory allocation. Each proposed strategy is validated by theoretical results.

`Priority` strategy is designed for memory profiles constant by part, with a reconfiguration time at least one order of magnitude lower than the length of a phase. It outperforms `Aggregated` as soon as the memory starts to be constrained. It allows reducing by 50% the memory usage while only losing 0.7% performance.

The second proposed strategy `Stochastic` is designed for dynamic memory patterns. It allocates memory based on statistical data of applications. It outperforms `Priority` for these applications with dynamic memory and allows reducing memory usage by 40%, while only losing 2% of performance.

In future work, it may be interesting to study the co-design of Batch-Scheduling strategies with memory partitioning algorithms, this co-design could use better runtime estimates. If still using EASY-BF, it would also be interesting to study what happens when using runtimes estimates better than the worst-case scenario.

VIII. Acknowledgements

We want to thank the reviewers for their thorough evaluation. Parts of this work have been supported by the Inria Exploratory Project REPAS, by the EuroHPC EU Regale project (g.a. 956560) and by the Exa-DoST project (PEPR NumPEx), reference ANR-22-EXNU-0004.

References

[1] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 183–190.

[2] A. Borghesi, C. Di Santi, M. Molan, M. S. Ardebili, A. Mauri, M. Guarrasi, D. Galetti, M. Cestari, F. Barchi, L. Benini *et al.*, "M100 exadata: a data collection campaign on the cineca's marconi100 tier-0 supercomputer," *Scientific Data*, vol. 10, no. 1, p. 288, 2023.

[3] J. Wahlgren, G. Schieffer, M. Gokhale, and I. Peng, "A quantitative approach for adopting disaggregated memory in hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[4] Cineca, "Ug3.2: Marconi100 userguide," 2023. [Online]. Available: https://wiki.u-gov.it/confluence/pages/viewpage.action?pageId=336727645

[5] Y. Fridman, S. Mutalik Desai, N. Singh, T. Willhalm, and G. Oren, "Cxl memory as persistent memory for disaggregated hpc: A practical approach," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 983–994.

[6] I. Peng, I. Karlin, M. Gokhale, K. Shoga, M. Legendre, and T. Gamblin, "A holistic view of memory utilization on hpc systems: Current and future trends," in *Proceedings of the International Symposium on Memory Systems*, 2021, pp. 1–11.

[7] A. Borghesi, C. Di Santi, M. Molan, M. S. Ardebili, A. Mauri, M. Guarrasi, D. Galetti, M. Cestari, F. Barchi, L. Benini, F. Beneventi, and A. Bartolini, "M100 dataset 1: from 20-03 to 20-12," 2023. [Online]. Available: https://zenodo.org/record/7588814

[8] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler, "Software resource disaggregation for hpc with serverless computing," *arXiv preprint arXiv:2401.10852*, 2024.

[9] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access,{High-Performance} memory disaggregation with {DirectCXL}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[10] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 249–264.

[11] Z. Wang, Y. Guo, K. Lu, J. Wan, D. Wang, T. Yao, and H. Wu, "Rcmp: Reconstructing rdma-based memory disaggregation via cxl," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 1, pp. 1–26, 2024.

[12] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda, "Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems," in *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. IEEE, 2012, pp. 48–55.

[13] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, "Memory disaggregation: Research problems and opportunities," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1664–1673.

[14] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 844–855, 2020.

[15] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.

[16] R. Bleuse, K. Dogeas, G. Lucarelli, G. Mounié, and D. Trystram, "Interference-aware scheduling using geometric constraints," in *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24*. Springer, 2018, pp. 205–217.

[17] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[18] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[19] R. Boëzennec, F. Dufossé, and G. Pallez, "Qualitatively analyzing optimization objectives in the design of hpc resource manager," 2023.

[20] G. Pallez, "Model design and accuracy for resource management in hpc," Ph.D. dissertation, Université de Bordeaux, 2023.

[21] A. Gainaru, B. Goglin, V. Honoré, and G. Pallez, "Profiles of upcoming hpc applications and their impact on reservation strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1178–1190, 2020.

[22] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.