# Exploring Mapping Strategies for Co-allocated HPC Applications

Ioannis Vardas[0000−0001−5461−556X], Sascha Hunold[0000−0002−5280−3855], Philippe Swartvagher[0000−0003−3786−7364], and Jesper Larsson Träff[0000−0002−4864−9226]

TU Wien, 1040 Vienna, Austria
{vardas,hunold,swartvagher,traff}@par.tuwien.ac.at

**Abstract.** Due to contention for resources like memory bandwidth, large-scale applications are not always able to efficiently utilize the high number of cores available in modern, deeply hierarchical HPC systems. Co-allocating multiple applications to share compute nodes can mitigate these issues and increase system throughput. However, co-allocation may harm the performance of individual applications due to resource contention. Past research suggests that good mappings, which take the system topology into account, can improve the performance of parallel applications that do not share resources. In this work, we implement application-oblivious, topology-aware mappings via different core enumerations that produce and support the co-allocation of parallel applications. We show that these mappings have a significant impact on the memory bandwidth. We explore how these process-to-core mappings can affect the individual application duration as well as system throughput when they are combined with co-allocation. Our main objective is to assess whether co-allocation with a topology-aware mapping can be a viable alternative to the exclusive node allocation policies that are common in current, real-world clusters.

**Keywords:** High Performance Computing · Parallel Computing · Performance Optimization · Process Mapping · Co-allocation.

## 1 Introduction

HPC systems are typical multi-user systems, where users submit batch jobs to request compute resources for a specified amount of time. Many CPU-based supercomputers are composed of compute nodes that feature a large number of cores. Parallel applications that run on these compute nodes cannot always efficiently use all allocated cores as some resources are saturated at high core counts, such as the memory or I/O bandwidth [3]. Under these circumstances, HPC systems strive to maintain a high job throughput and low makespan while also keeping the job duration short to meet users' needs. Therefore, two important metrics for HPC systems are the makespan which is the time difference between the start and finish of a sequence of jobs, and the total work which is the sum of the jobs' durations. Co-allocating multiple applications to share the compute

nodes, can increase the job throughput or, equivalently, reduce the job makespan. Even though previous research has shown promising results for co-scheduling [2,3], it is rarely used in production systems for multi-node jobs. A drawback of co-allocation is that it can increase the duration of jobs [1] if jobs conflict over shared resources such as the L3 cache, memory controllers, or the network interface.

To tackle this issue, applying an efficient process-mapping strategy can improve the performance of applications by reducing their communication time [4]. Mapping is the assignment of processes of a parallel application to the processing units (cores) of the system to improve metrics such as communication time. Due to the deeper memory hierarchies, the higher core-density nodes, and the increased number of compute nodes of HPC systems, the mapping of parallel applications plays a critical role in performance. Most works that improve the mapping of applications are not concerned with co-allocated applications, and they often require an application profile which, in turn, requires an extra profiling run, which renders them impractical for production systems [5].

In the present work, we explore different process-to-core mappings for co-allocated applications similar to the work of Breslow et al. [2], which proposes a process-to-core mapping for colocating applications called job striping. With job striping, two jobs share a set of nodes where half of the cores of each node run one job and the other half run the other. In our work, we devise and implement several topology-aware and application-oblivious process-to-core mappings using different enumerations of the compute cores for co-allocated applications. We analyze the effects of our mapping strategies and show that they affect the available memory bandwidth to a parallel application. In our evaluation, we employ typical HPC applications to explore the impact of mapping and co-allocation. We compare our strategies with an exclusive allocation policy that is common in HPC systems.

## 2 Experimental Environment and Methods

Vienna Scientific Cluster 5 (VSC-5) consists of 770 compute nodes. This system is an example of a modern hierarchical system where each compute node consists of two packages (sockets), and each socket has an AMD EPYC™ 7713 processor totaling 128 cores. Figure 1 depicts the hierarchical topology of a VSC-5 compute node, showing the four levels of hierarchy, which are: (1) compute node, (2) package, (3) NUMA node and, (4) the core. Each package has four NUMA nodes and comprises 16 cores. Each compute node has 512 GiB of RAM and 8 memory channels per socket (2 per NUMA). This deep and complex hierarchy, motivates us to explore various mapping strategies that leverage the resources of such hierarchies differently. We implement process-to-core mapping strategies that take into account three levels of the node's hierarchy: the package, the NUMA node, and the core. In multi-node scenarios, we assume that the scheduler distributes an equal number of processes to each node and that each compute node is shared between applications and the mapping is replicated to each node.

We produce different mappings by enumerating the cores differently. Since there are $n!$ ways of enumerating or ordering $n$ elements, trying all enumerations

is impossible for practical use. To narrow the search space, we leverage the hierarchical structure of the machine's topology. We use a mixed-radix numerical system to represent different core enumerations. The base of our mixed-radix system is the hierarchy of a compute node, therefore we have $h = \{2, 4, 16\}$ (i.e., two sockets, four NUMA nodes per socket, 16 cores per NUMA node). To produce different enumerations of the cores, we first decompose the core ids into sets of digits using Algorithm 1. The default enumeration scheme is shown in Fig. 1. Then, we compute the new ids with Algorithm 2, which uses the decomposed digits from Algorithm 1, the hierarchy $h$, and the order $o$ of hierarchy. The order $o$ is the order in which the hierarchy levels are considered by Algorithm 2. To produce the enumeration shown in Fig. 1, we use the order $o = \{0, 1, 2\}$. By using different permutations of $o$, we produce different mappings.

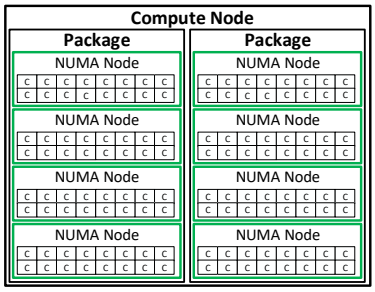



Fig. 1: Hierarchical view of the architecture of a VSC-5 compute node.

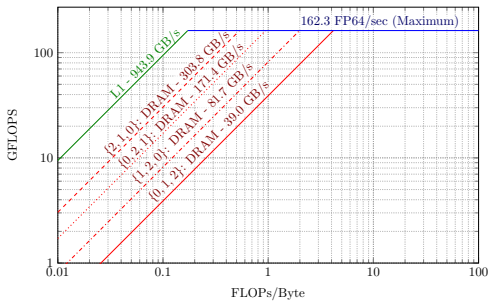



Fig. 2: Roofline models of eight processes running on a VSC-5 compute node. Each DRAM line shows the bandwidth with a different mapping.

**Algorithm 1** Decompose core id

**Input:** $h$: hierarchy, $id$: core id
**Output:** $d$: decomposed digits

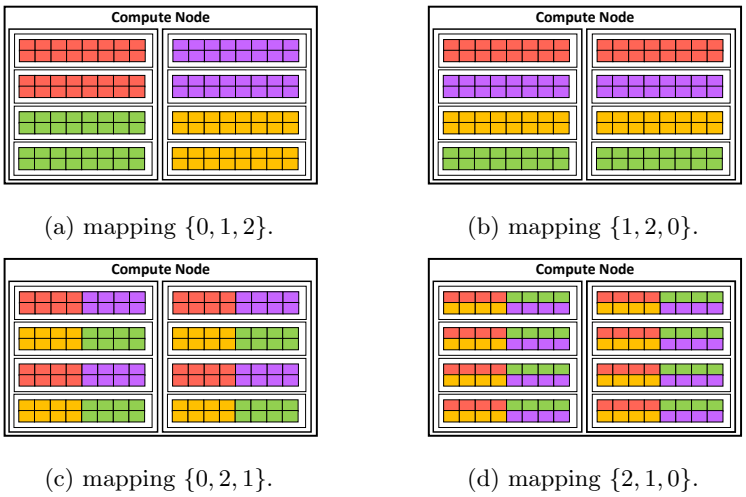
$d \leftarrow []$
**for** $i \leftarrow 0$ to length$(h) - 1$ **do**
  $d[i] \leftarrow id \mod h[i]$
  $id \leftarrow id \,//\, h[i]$
**end for**

**Algorithm 2** Compute core id

**Input:** $h$: hierarchy, $d$: digits, $o$: order
**Output:** $nid$: new core id

$nid \leftarrow 0, s \leftarrow 1$
**for** $i \leftarrow 0$ to length$(h) - 1$ **do**
  $nid \leftarrow nid + d[o[i]] \times s$
  $s \leftarrow s \times h[o[i]]$
**end for**

Since $o$ is a set of three elements, it has $3! = 6$ possible permutations, resulting in 6 different mappings. We focus on four out of six different mappings which, as we show later, differ significantly in terms of bandwidth. Figure 3 shows the mappings of four co-allocated MPI applications using orders $\{0, 1, 2\}, \{1, 2, 0\}, \{2, 1, 0\}$ and $\{0, 2, 1\}$, where each application is allotted 32 processes. From Fig. 3, we notice that different orders use different resources: order {0,1,2} maps all processes within the same NUMA node, whereas {2,1,0} maps each process to a different




(a) mapping $\{0, 1, 2\}$. (b) mapping $\{1, 2, 0\}$.

(c) mapping $\{0, 2, 1\}$. (d) mapping $\{2, 1, 0\}$.

Fig. 3: Process-to-core mappings of four co-allocated MPI applications each with 32 processes sharing one compute node. Different colors denote processes that belong to different applications.

NUMA node. Order {1,2,0} maps the processes in two NUMA nodes, one NUMA node per package whereas {0,2,1} uses two NUMA nodes per package.

These mappings offer different memory bandwidths, which we demonstrate in Fig. 2, which shows the roofline models for each of the four mappings with eight processes. Mapping {2,1,0} offers the highest memory (304 GB/s) bandwidth whereas {1,2,0} offers the lowest (39 GB/s). Moreover, our mappings influence the resource contention of co-allocated applications by affecting their shared resources. This is illustrated in Fig. 3, which shows an example of how these four mapping strategies map four co-allocated applications, each with 32 processes, in VSC-5 compute node. The least amount of shared resource between applications is achieved by the {0,1,2} mapping in Fig. 3a, whereas Fig. 3d, shows that more resources are shared with {2,1,0} mapping. We refer to mappings {0,1,2} and {1,2,0} as *compact*, whereas, {2,1,0} and {0,2,1} are categorized as *spread*.

## 3 Evaluation and Results

We conducted our experiments on the VSC-5 using a typical HPC workload of eight MPI applications: LAMMPS, CG from NAS Parallel Benchmarks, GROMACS, FFT, and four ECP Proxy applications, all compiled with Open MPI 4.1.3 In the first scenario, we measure the impact of the mappings on the performance of applications when run in both isolation and co-allocation. Figure 4 shows the duration of each application running with 8 nodes and 16 processes per node in either co-allocation or isolation mode using all four different mappings. In co-allocated runs, all eight applications run concurrently and share a different

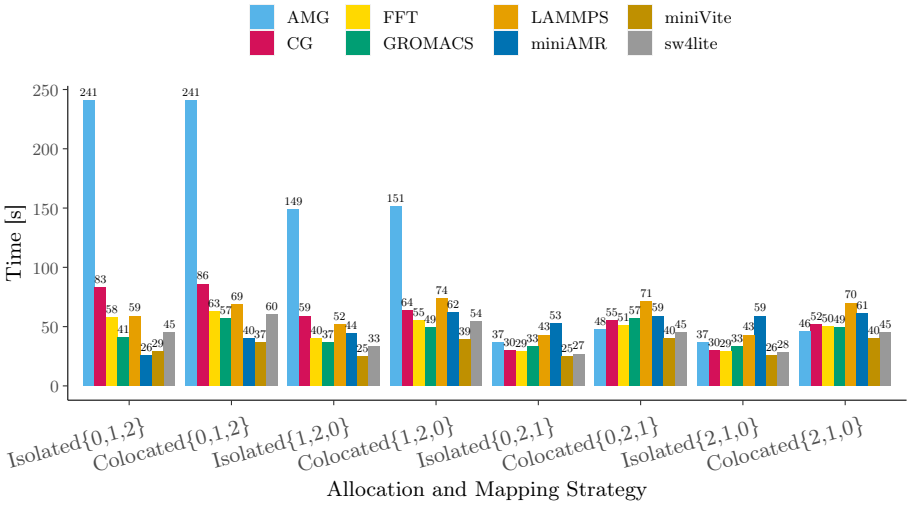



Fig. 4: The impact of mapping and co-allocation on performance, applications run with $8 \times 16$ processes, either in isolation or co-allocation with four mappings.

part of the nodes, similar to Fig. 3. In isolated runs, applications run exclusively on compute nodes while using the same mapping as with co-allocation. We notice that *spread* mappings improve the performance of most applications in this set. However, there is no mapping that benefits every application. We also notice that the negative impact of co-allocation on applications with *compact* mappings is lower than that of *spread*. This is because with *spread* mapping more resources are shared, causing more resource contention. Finally, *spread* mappings, show better performance, outweighing the negative effect of co-allocation in this application set. In the second scenario, we focus on the makespan and total work. We compare our mapping methods with co-allocation against the common allocation and mapping policy of Slurm in VSC-5, which performs an exclusive allocation with round-robin mapping, denoted as `exclusive.RR`. When applying the `exclusive.RR` strategy, one node is exclusively allotted to each application, where it runs with 128 cores with a round-robin mapping. We show the results in Fig. 5. We can observe that our mapping strategies outperform the `exclusive.RR` for most applications. Mappings {2,1,0} and {0,2,1} show the best performance in terms of makespan. `colocated{2,1,0}` offers an improvement of 2.4× and 1.4× over `exclusive.RR` in terms of makespan and total work, respectively.

## 4 Conclusion

We have explored the effects and benefits of different mappings coupled with co-allocation. We have devised application-oblivious and topology-aware process-to-core mapping strategies using different core enumerations. Our preliminary

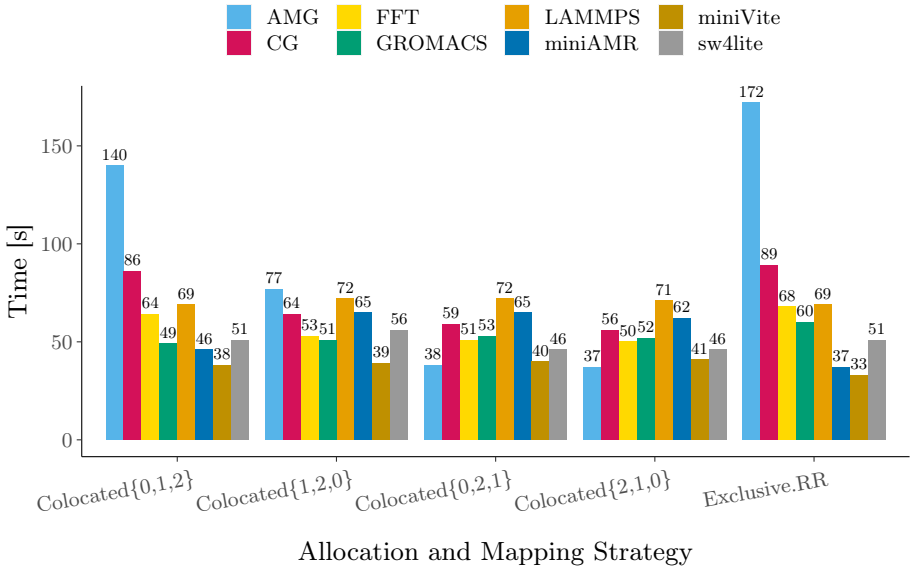



Fig. 5: Comparison between our mapping strategies with co-allocation against `exclusive.RR` where each application runs in one node exclusively.

results show that co-allocation coupled with *spread* mappings can improve both the total work and makespan for workloads consisting of up to eight HPC applications compared to the exclusive allocation. We are implementing more dynamic mappings using additional HPC applications and scenarios with diverse numbers of processes and performing experiments on additional systems.

**Acknowledgements** This work was partially supported by the Austrian Science Fund (FWF): project P 31763-N31 and project P 33884-N.

## References

1. Blanche, A.d., Lundqvist, T.: Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedules. In: Proceedings of the 1st COSH Workshop. pp. 25–30 (2016)
2. Breslow, A.D., Porter, L., Tiwari, A., Laurenzano, M., Carrington, L., Tullsen, D.M., Snavely, A.E.: The Case for Colocation of High Performance Computing Workloads. Concurr. Comput.: Pract. Exper. p. 232–251 (2016)
3. Frank, A., Süß, T., Brinkmann, A.: Effects and Benefits of Node Sharing Strategies in HPC Batch Systems. In: IEEE IPDPS. pp. 43–53 (2019)
4. von Kirchbach, K., Lehr, M., Hunold, S., Schulz, C., Träff, J.L.: Efficient Process-to-Node Mapping Algorithms for Stencil Computations. In: CLUSTER (2020)
5. Vardas, I., Hunold, S., Ajanohoun, J.I., Träff, J.L.: mpisee: MPI Profiling for Communication and Communicator Structure. In: IEEE IPDPSW. pp. 520–529 (2022)