Exploring Mapping Strategies for Co-allocated HPC Applications

Ioannis Vardas(✉), Sascha Hunold, Philippe Swartvagher, and Jesper Larsson Träff

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

Abstract. In modern HPC systems with deep hierarchical architectures, large-scale applications often struggle to efficiently utilize the abundant cores due to the saturation of resources such as memory. 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 topology-aware mappings can improve the performance of parallel applications that do not share resources. In this work, we implement application-oblivious, topology-aware process-to-core mappings via different core enumerations that support the co-allocation of parallel applications. We show that these mappings have a significant impact on the available memory bandwidth. We explore how these process-to-core mappings can affect the individual application duration as well as the makespan of job schedules 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 currently common in HPC 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 that 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 become 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: (1) the makespan which is the time difference between the start and finish of a sequence of jobs and, (2) the individual

The original version of this chapter was previously published without open access. A correction to this chapter is available at
https://doi.org/10.1007/978-3-031-48803-0_41

job duration. Co-allocating multiple applications to share the compute nodes can reduce the makespan of a job schedule. Even though previous research has shown promising results for co-scheduling [2,3], it is rarely used in production systems for multi-node CPU applications. 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.

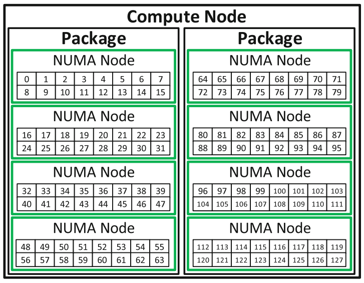
To address this issue, applying efficient process mappings can improve the performance of applications by reducing their communication time [4]. Mapping is the assignment of processes of a parallel application to the processing units (cores) of the system. Due to the deeper memory hierarchies, the higher core-density nodes, and the increased number of compute nodes of HPC systems, the mapping of parallel applications can significantly affect their performance. Most works that improve the mapping of applications are not concerned with co-allocated applications, and they often require 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 method for co-allocating 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 go beyond job striping by devising 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 of a parallel application. In our evaluation, we employ typical HPC applications to explore the impact of mapping and co-allocation. We compare our strategies combined with an allocation policy that is common in HPC systems.

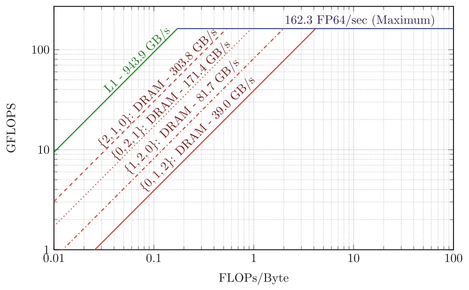
2 Experimental Environment and Methods

As an example of an HPC system with a deep resource hierarchy, we use the Vienna Scientific Cluster 5 (VSC-5). Each of its 770 compute nodes consists of two packages (sockets), and each socket has an AMD EPYC$^{\text{TM}}$ 7713 processor with 64 cores. Figure 1a depicts the hierarchical topology of one 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 each NUMA node comprises 16 cores. Each compute node has 512 GiB of RAM and eight memory channels per socket (2 per NUMA node). This deep and complex hierarchy motivates us to explore various mapping strategies that leverage the resources of such hierarchies differently. Our strategies take into account three levels of the node's hierarchy: the package, the NUMA node, and the core. In multi-node scenarios, we make three assumptions: (1) the scheduler distributes an equal number of processes to each node; (2) each compute node is shared between applications; and (3) the mapping is replicated to each node.

We produce different process-to-core mappings by varying core enumerations. With $n!$ possible enumerations of n elements, directly exploring all of them is



(a) Hierarchical view of the architecture of a VSC-5 compute node



(b) Roofline models: DRAM lines show the available bandwidth with different mappings

Fig. 1. Overview of a single VSC-5 compute node: (a) Hierarchical view of the architecture, and (b) the Roofline models of eight processes.

impractical. Our mappings leverage the machine's hierarchical topology to narrow this search space. We represent core enumerations in a mixed-radix numerical system, based on an ordered set of the hierarchy $h = \{2, 4, 16\}$, which denotes two sockets, four NUMA nodes per socket, and 16 cores per NUMA node. First, we decompose the core IDs into digit sets using Algorithm 1. These sets of digits are the indices to the different levels defined by the hierarchy. Second, we compute the new core IDs via Algorithm 2, employing the decomposed digits from Algorithm 1, hierarchy h, and order o of hierarchy. The order o denotes the sequence of hierarchy levels considered by Algorithm 2. Using order $o = \{0, 1, 2\}$, we produce the enumeration in Fig. 1a. By permuting o, we produce different mappings, e.g., $o = \{2, 1, 0\}$ enumerates the core IDs as $\{0, 64, 16, 80, ...\}$, that is, we first cyclically assign processes between packages and then cyclically assign processes between NUMA nodes. Since this hierarchy has three levels, o is a set of three elements, therefore, six permutations and thus six different mappings are possible. We focus on four out of six mappings, which differ significantly in terms of bandwidth.

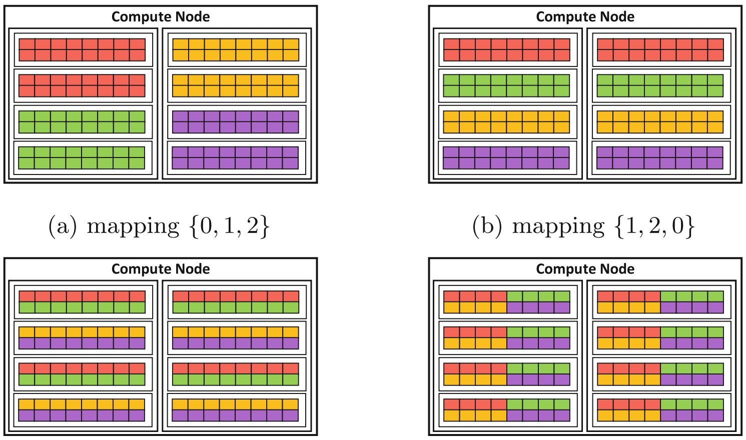
Algorithm 1. Decompose core ID

```
    Input: $h$: hierarchy, $id$: core id
    Output: $d$: decomposed digits
1: $d \leftarrow []$
2: for $i \leftarrow 0$ to length($h$) $- 1$ do
3:     $d[i] \leftarrow id \mod h[i]$
4:     $id \leftarrow id \mathbin{//} h[i]$   ▷ Integer division
5: end for
```

Algorithm 2. Compute core ID

```
    Input: $h$: hierarchy, $d$: digits, $o$: order
    Output: $nid$: new core id
1: $nid \leftarrow 0, s \leftarrow 1$
2: for $i \leftarrow 0$ to length($h$) $- 1$ do
3:     $nid \leftarrow nid + d[o[i]] \times s$
4:     $s \leftarrow s \times h[o[i]]$
5: end for
```

Figure 2 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 cores in a VSC-5 compute node. We name the mappings after the orders that they are derived from. From Fig. 2, we notice that different applications use different resources: For example, when applications are mapped with {1, 2, 0} the processes are placed in one NUMA node per package, whereas with {0, 2, 1} they are placed in two NUMA nodes per package.



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

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

Fig. 2. Process-to-core mappings of four co-allocated MPI applications each with 32 processes sharing one compute node. Different colors denote cores that are allotted to different applications.

Figure 1b illustrates the distinct memory bandwidths offered by the four mappings, each with eight processes. The arithmetic intensity is on the x-axis and the performance on the y-axis. We notice that mapping {2, 1, 0} yields a maximum bandwidth of 304 GB/s, in contrast to {0, 1, 2} at 39 GB/s. Consequently, an application with 1 FLOP/Byte arithmetic intensity, for example, can attain 162 GFLOPS and 39 GFLOPS at the respective bandwidths. Moreover, our mappings influence the resource contention of co-allocated applications by affecting their shared resources. The least amount of shared resources between applications is achieved by the {0, 1, 2} mapping in Fig. 2a, whereas Fig. 2d shows that more resources are shared with {2, 1, 0} mapping. We categorize 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

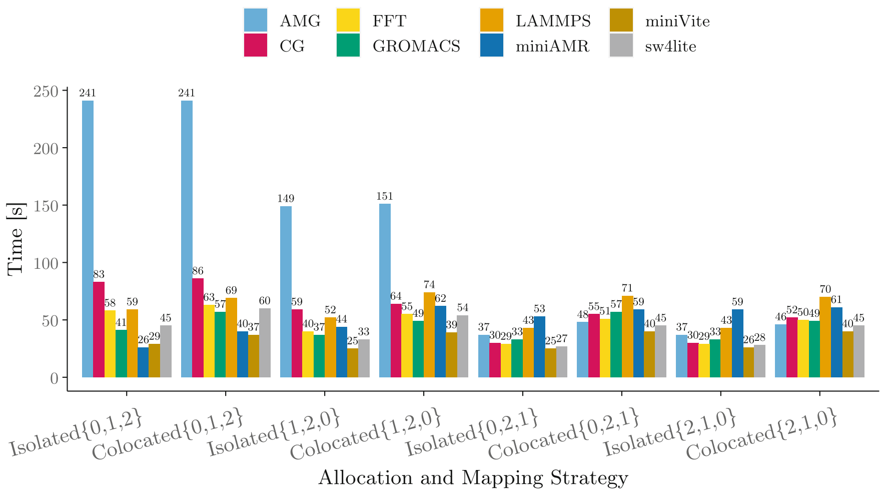


Fig. 3. The impact of mapping and co-allocation on performance, applications run with 8 × 16 processes, either in isolation or co-allocation with four mappings.

on the application performance when run either in isolation or co-allocation. Figure 3 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 part of the nodes, similar to Fig. 2. 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, which can increase 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 the sum of job durations. We compare our mapping methods with co-allocation against the common allocation and mapping policy of Slurm on 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 these results in Fig. 4, where we 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. Strategy `colocated{2,1,0}` offers an improvement of 2.4× and 1.4× over `exclusive.RR` in terms of makespan and the sum of job durations, respectively.

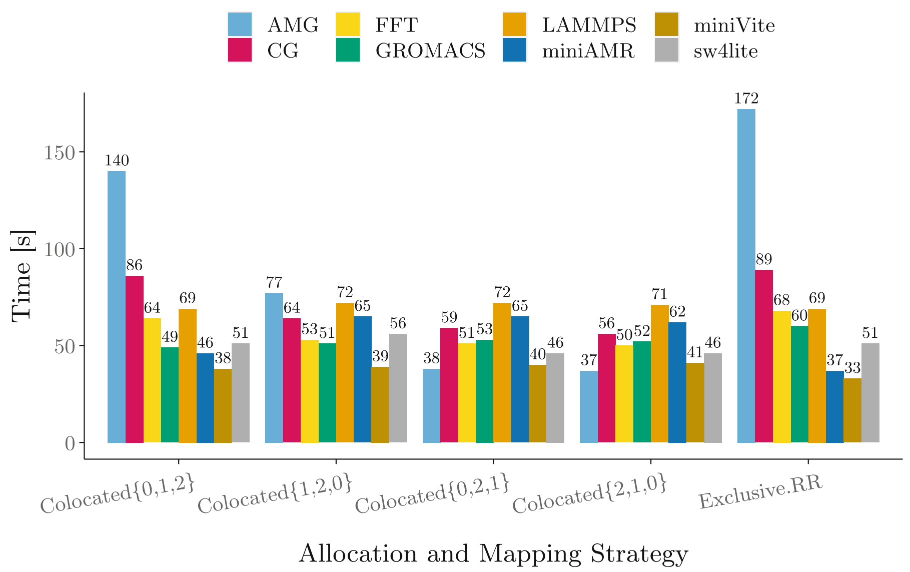


Fig. 4. Comparison between our mapping strategies with co-allocation against `exclusive.RR` where each application runs in one node exclusively.

4 Conclusion

We have explored the effects and benefits of different process-to-core mappings coupled with co-allocation. We have devised application-oblivious and topology-aware process-to-core mapping strategies using different core enumerations. Our preliminary results show that co-allocation coupled with *spread* mappings can improve both individual job performance and makespan for workloads consisting of eight HPC applications compared to the exclusive allocation. We plan to implement more dynamic mappings using additional HPC applications and scenarios with diverse numbers of processes, and perform experiments on additional HPC systems with different architectures.

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

References

1. de Blanche, A., 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., et al.: The case for colocation of high performance computing workloads. Concurr. Comput.: Pract. Exper. 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)