Using Mixed-Radix Decomposition to Enumerate Computational Resources of Deeply Hierarchical Architectures

Philippe Swartvagher
Sascha Hunold
Jesper Larsson Träff
Ioannis Vardas
{swartvagher,hunold,traff,vardas}@par.tuwien.ac.at
TU Wien
Vienna, Austria

ABSTRACT

Current HPC architectures are deeply hierarchical (racks, nodes, sockets, NUMA domains, caches, ...), and the mapping of MPI processes to cores can significantly influence application performance. To study hierarchy effects on MPI application performance, we propose a procedure for expressing mappings by enumerating cores in the hierarchy in different orders. We explore two use cases: MPI rank reordering for applications using subcommunicators, and core selection for applications not using all cores on a node.

Results of micro-benchmarks executing collective operations in subcommunicators show a performance difference up to a factor 4 between the best and the worst rank orderings. By changing the rank orders, we observe a performance impact for the Splatt application. The evaluation of the strong scalability of a conjugate gradient benchmark shows that considering all hierarchy levels in the core selection policy can give better performance than using only options available with common MPI application launchers.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; **Massively parallel algorithms**; **Parallel programming languages**.

KEYWORDS

MPI, collective communication operations, process mapping, rank reordering, hierarchical topology

ACM Reference Format:
Philippe Swartvagher, Sascha Hunold, Jesper Larsson Träff, and Ioannis Vardas. 2023. Using Mixed-Radix Decomposition to Enumerate Computational Resources of Deeply Hierarchical Architectures. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3624062.3624109

1 INTRODUCTION

Modern parallel systems to run High-Performance Computing applications are built very hierarchically: compute nodes are grouped into racks, cabinets, or islands, they feature multiple processors, which are organized in different Non-Uniform Memory Access (NUMA) domains and shared caches. Carefully mapping application processes to these resources is crucial for performance. Indeed, links between resources on each level of the hierarchy have different capacities and can handle more or less contention. Each application (or even each application input) has its own optimal mapping which depends on its computation and communication pattern as well as on system characteristics. These system characteristics include the hierarchy (e.g., number of cores per NUMA domain), the communication performance between two MPI processes, depending on where they are mapped in the system, or the performance under contention, etc. The data access pattern and the communication scheme of an application, e.g., whether it is more compute-bound, latency-bound, or bandwidth-bound are properties that influence the choice of the best mapping. Given all the constraints to consider, while taking the topology of the system into account, the task of mapping application processes to cores to reach the best possible performance is very complex.

Taking into consideration the increasing number of hierarchy levels in HPC systems and current numbers of cores per processor, we propose in this paper a technique to express different process mappings by enumerating the compute cores in different orders. Especially, we consider the case of MPI applications executing simultaneously collective operations in different subcommunicators. To generate different enumeration orders, we apply mixed-radix decomposition to `MPI_COMM_WORLD` ranks. This technique has several interesting properties: it is simple to implement, it is application-oblivious while taking into account all the levels of the system hierarchy, and it allows more mapping schemes than options available with regular MPI job launchers, such as `mpirun` or Slurm's `srun`. We apply this algorithm to two use cases: MPI rank reordering and core selection for Slurm jobs. Our algorithm gives control over how MPI processes belonging to a communicator are spread over the resources composing the system. To characterize and distinguish the possible rank orders and the resulting mappings of communicators, we propose two metrics: the *ring cost* and the *percentages of process pairs per level*. The former reflects the order of enumerated processes inside a communicator and the latter shows how much processes belonging to a communicator are spread over the available resources. The experimental evaluations of different rank orders generated by our algorithm show that collective operations executed in subcommunicators can be very sensitive to the chosen mapping. We study the impact of rank order *inside* a

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624109

communicator on the collective application performance and we compare two scenarios: One where a single communicator performs collective operations to another where multiple communicators perform collective operations concurrently. As a real-world use case, our technique allows to apply a rank reordering that improves performance of the Splatt application by 32 % compared to the default mapping Slurm would set up. Finally, we evaluate the strong scaling of a conjugate gradient benchmark, using the cores selected by our algorithm.

Our algorithm requires as input a permutation of the different hierarchy levels to generate the new mapping. For a system hierarchy featuring h levels, there are $h!$ different permutations. We do not recommend to evaluate all $h!$ permutations to find the best one. The goal of this paper is to present an efficient algorithm allowing to express specific mappings and explore possible use cases.

The rest of the paper is organized as follows: Section 2 presents related work on rank reordering. Section 3 explains the enumeration algorithm that we propose and two possible use cases. The impact of our algorithm is evaluated in Section 4, and Section 5 concludes.

2 RELATED WORK

Rank reordering is closely related to the process-to-core mapping problem, which consists in finding an efficient placement of MPI processes on available computing units. The goal is usually to minimize communication costs by mapping MPI processes that communicate together close to each other (for instance, inside the same socket). This problem is NP-hard [10] and an extensive work already covers this topic [9]. A naive solution is to test all possible permutations and to select the most efficient one. However, if there are p processes to map on n computing units (with $p \leq n$), there are $\frac{n!}{(n-p)!}$ (or $n!$ if $p = n$) mapping combinations to evaluate, which is usually not feasible in a reasonable time. To tackle this issue, Denoyelle et al. [5] propose a method to reduce the space of candidate process mappings to evaluate. The algorithm we present in this paper also reduces the search space, since the number of mappings it can generate depends on the depth of the hierarchy h (always lower than 6 in our experiments) and not on the number of computing units n (up to 2048 in our experiments). Hence, we reduce the size of the search space from $n!$ to $h!$. This reduction is possible as we assume that permutations of MPI processes inside the same lowest level of the hierarchy (e.g., inside a NUMA node) have no impact on performance.

Another common technique is to provide the communication matrix of a program and the description of the system to a process mapping tool, which will return a process mapping minimizing communication costs for this program. The communication matrix can be built beforehand or at runtime [11]. Once an efficient mapping is known, the rank reordering step changes ranks of MPI processes to match the mapping. The work we present in this paper is oblivious to the communication matrix (for now). Communication matrices can help to determine a better mapping, while our technique can help to set up this mapping.

Kwon et al. [16] enumerate computing cores using a space-filling curve to map MPI processes to cores while preserving locality for communications. Li et al. [17] use a Morton space-filling curve to improve cache locality of all-to-all MPI communication patterns. Our technique is similar to a space-filling curve, because it enumerates all computing units in a specific and regular order, yet the main differences are that the construction is not iterative and our technique enumerates all computing cores in a hierarchical level before going to the next level.

The MPI standard offers also several routines to assist the mapping step [24]: Cartesian [7] or distributed graph [18] topologies define communication relationships between processes. When creating such *virtual* topologies, it is possible to request a rank reordering to better match the system topology. The creation of subcommunicators can also take into account the hierarchy of the machine, with the *guided* and *unguided* modes of `MPI_Comm_split_type` introduced in Version 4 of the MPI standard [6].

To improve performance of collective operations, the underlying algorithms can be improved by taking into account the hierarchy of the system [3, 12, 13]. To consider both collective and point-to-point communication in the mapping decision, collective operations can be decomposed into point-to-point communications [26].

For collectives in subcommunicators, Mirsadeghi and Afsahi build a communicator dedicated to `MPI_Allgather` operations, with reordered ranks according to an algorithm to minimize distance between MPI processes of the communicator [19]. Bhatele et al. propose a tool to efficiently map on a torus network applications that do collective operations in subcommunicators [2].

In contrast, our work stands apart by proposing a generic algorithm to enumerate computing units in different orders, taking the system hierarchy into account.

3 ENUMERATION ALGORITHM

This section presents our enumeration algorithm based on a mixed-radix decomposition, its use in MPI, how we characterize the different mappings obtained with this algorithm, and how we use it to generalize Slurm's mapping functionality.

3.1 Mixed-radix Decomposition

A mixed-radix numeral system is a way of expressing numbers with a numerical base that varies from position to position. Given a sequence of integers $(h_i)_{i\geq 1}$, all strictly greater than 1, any positive integer r can be decomposed into a unique sequence of coordinates $(c_i)_{i\geq 0}$ following Equation (1):

$$r = c_0 + \sum_{i=1}^{|h|-1} c_i \prod_{j=0}^{i-1} h_j \quad . \tag{1}$$

In everyday life, the most common mixed system is the one used to measure time: the quantity *3 weeks, 2 days, 9 hours, 22 minutes and 32 seconds* is equal to $32 + 22 \times 60 + 9 \times 60 \times 60 + 2 \times 24 \times 60 \times 60 + 3 \times 7 \times 24 \times 60 \times 60 = 2\,020\,952$ seconds. Thus, the number 2 020 952 can be decomposed into the coordinates $[32, 22, 9, 2, 3]$ in the mixed-radix system $[\![\mathbb{60}, \mathbb{60}, \mathbb{24}, \mathbb{7}]\!]$. This example was given by Knuth, who discusses mixed-radix systems in detail [15, p. 208-209].

Programmers also manipulate mixed-radix systems when they need to index in one dimension a multidimensional array, e.g., an image with dimensions $w \times h$, where the color of each pixel is encoded using three values. In this example, the third color value of a pixel located at $(12, 20)$ can be stored in the cell numbered $2 + 12 \times 3 + 20 \times w \times 3$ of a one-dimensional array, if the programmer

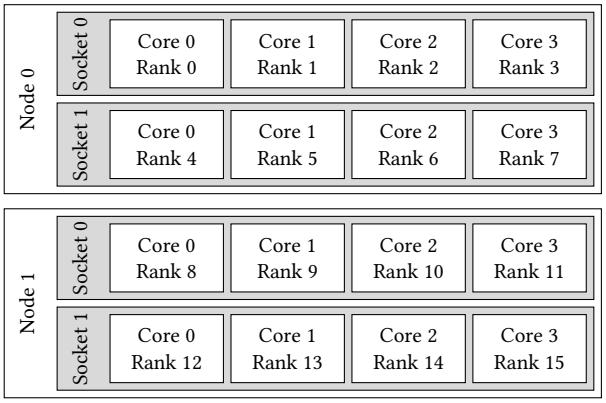


Figure 1: Example of a system with hierarchy $[\![2, 2, 4]\!]$, with one MPI process per core.

Algorithm 1 Get coordinates of a rank number

Inputs: h : hierarchy, r : rank
Output: c : coordinates

$c \leftarrow []$
for $i = |h| - 1$ **to** 0 **do**
  $c[i] = r \,\%\, h[i]$ ▷ Modulo operation
  $r = r \,//\, h[i]$ ▷ Integer division
end for

chooses to enumerate the image by line, pixel, and color value. However, the programmer can also choose another enumeration of the values forming the image: different orders may change the 1-dimensional coordinate of each value of the image.

Our rank reordering algorithm relies on this mixed-radix decomposition. Decomposed integers are MPI ranks, radixes are the number of elements in each level of the system hierarchy and changing the order of these radixes gives a new rank value.

In current hierarchical HPC systems, each component features a constant number of sub-components: e.g., a compute node has several processors, a processor has several NUMA domains, and a NUMA domain encompasses several cores. These numbers of sub-components are the mixed-radix base for the decomposition of MPI ranks. Figure 1 presents a system featuring two compute nodes, each with two sockets, where each socket comprises four cores. This hierarchy is denoted as $h = [\![2, 2, 4]\!]$. Given this hierarchy h and an initial rank numbering, each rank r can be decomposed into a set of coordinates c with Algorithm 1. These coordinates can be seen as the coordinates of the core in the multi-dimensional space defined by the hierarchy levels; for instance, rank 10 on Figure 1 is decomposed into coordinates $[1, 0, 2]$ because it is on the node 1 (node dimension), on the socket 0 (socket dimension) and on the core 2 (core dimension). If the initial rank numbering does not enumerate all cores before going to the next socket of the same node and then the next node (like on Figure 1), the obtained coordinates will not correspond to those of the core and will break the remaining of the reordering algorithm.

With the hierarchy h, the coordinates c of a rank value, and a permutation function σ, we can get a new rank value r with

Algorithm 2 Compute a rank from coordinates

Inputs: h : hierarchy, c : coordinates, σ : permutation
Output: r : rank

$r \leftarrow 0$
$f \leftarrow 1$
for $i = 0$ **to** $|h| - 1$ **do**
  $r = r + c[\sigma[i]] \times f$
  $f = f \times h[\sigma[i]]$
end for

Table 1: Examples of different orders, applied to rank 10 (decomposed into coordinates $[1, 0, 2]$), on a hierarchy $[\![2, 2, 4]\!]$.

Order	Permuted coordinates	Permuted hierarchy	New rank
$[0, 1, 2]$	$[1, 0, 2]$	$[\![2, 2, 4]\!]$	9
$[0, 2, 1]$	$[1, 2, 0]$	$[\![2, 4, 2]\!]$	5
$[1, 0, 2]$	$[0, 1, 2]$	$[\![2, 2, 4]\!]$	10
$[1, 2, 0]$	$[0, 2, 1]$	$[\![2, 4, 2]\!]$	12
$[2, 0, 1]$	$[2, 1, 0]$	$[\![4, 2, 2]\!]$	6
$[2, 1, 0]$	$[2, 0, 1]$	$[\![4, 2, 2]\!]$	10

Equation (2):

$$r = c_{\sigma(0)} + \sum_{i=1}^{|h|-1} c_{\sigma(i)} \prod_{j=0}^{i-1} h_{\sigma(j)} \quad . \tag{2}$$

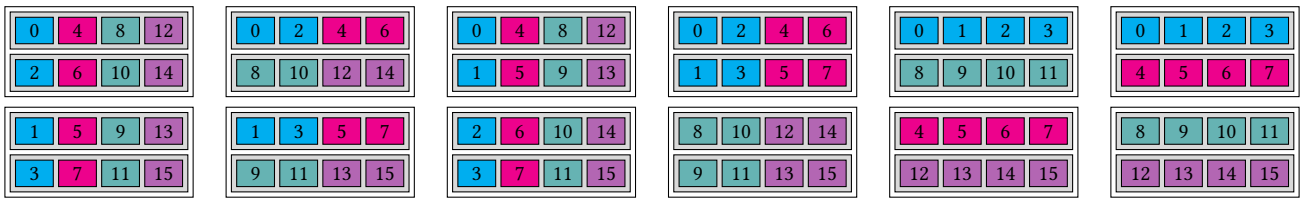
The permutation function $\sigma : [\![0, |h|]\!] \mapsto [\![0, |h|]\!]$ defines in which order the different levels of the hierarchy will be enumerated. Algorithm 2 computes the rank value following Equation (2).

Permutations (also called *orders* in this paper) are noted like $[2, 0, 1]$, which means $\sigma(0) = 2$, $\sigma(1) = 0$ and $\sigma(2) = 1$. The inverse of Algorithm 1 is Algorithm 2 applied with the order $[2, 1, 0]$ (for a level hierarchy with 3 levels). For a hierarchy with n levels (i.e., $|h| = n$), there are $n!$ different orders. Table 1 provides an example to show the impact of different orders applied to the initial rank 10: as described in Algorithm 2, the permutation is used to change the order of both coordinates and hierarchy levels. Figure 2 illustrates the different orders on the same hierarchy and initial ranks than on Figure 1 (only reordered ranks are represented for more readability). The order producing the original enumeration is $[2, 1, 0]$ (cf. Figure 2f), as previously mentioned.

3.2 First Use Case: Rank Reordering in MPI

This mixed-radix decomposition can be used to reorder ranks of MPI processes in the `MPI_COMM_WORLD` communicator. It is more difficult to reorder smaller communicators, since they may contain only a subset of processes and not be spread on the whole system, making some hierarchy levels in the mixed-radix base useless, and there is no guarantee the initial rank numbering in any other communicator than `MPI_COMM_WORLD` respects the conditions of Algorithm 1.

Reordering `MPI_COMM_WORLD` can be done in two ways. The first method creates a new communicator containing all processes (i.e., all processes have the same `color`) by calling `MPI_Comm_split` with the reordered rank as the `key`, to specify for each process its



(a) Order $[0, 1, 2]$ `cyclic:cyclic` **(b) Order** $[0, 2, 1]$ `cyclic:block` **(c) Order** $[1, 0, 2]$ ***Not possible*** **(d) Order** $[1, 2, 0]$ `block:cyclic` **(e) Order** $[2, 0, 1]$ `plane=4` **(f) Order** $[2, 1, 0]$ `block:block`

Figure 2: All possible orders for a $[\![2, 2, 4]\!]$ hierarchy. Numbers represent reordered ranks of processes in `MPI_COMM_WORLD`; processes with the same color belong to the same subcommunicator. Below each order is the value to pass to the Slurm `--distribution` option to achieve the same rank mapping; order $[1, 0, 2]$ cannot be achieved with the use of this option.

new rank in this communicator. The reordered rank is computed by applying Algorithm 2 on the coordinates of the process rank in `MPI_COMM_WORLD`, obtained with Algorithm 1. Then, the application can use this new communicator instead of `MPI_COMM_WORLD`. Thus, this method may require modifications in existing applications. The second method is to use a *rankfile*, a file describing on which core should be mapped which process with the given `MPI_COMM_WORLD` rank. This method is transparent for applications, since the rank of each MPI process in the `MPI_COMM_WORLD` has already its reordered value. However, rankfiles are not available with all MPI launchers.

Our reordering algorithm needs a hierarchy description. The hierarchy information passed to the decomposition algorithm should reflect the actual hardware hierarchy of the system executing the MPI application. Thus, the number of levels in the hierarchy and the size of each level can be gathered with tools such as `hwloc` [4] or directly with MPI routines, by splitting communicators according to the hardware hierarchy [6]. However, it can also be interesting to manually provide a hierarchy that can include more or less levels than in the reality. Adding a *fake* level (e.g., a socket containing 16 cores can be faked as containing 2 components with 8 cores each) allows to explore more ordering possibilities. Although it is not exploited in the present work, the hierarchy can also include levels outside of nodes, like cabinets or the topology of the network, especially if it is a tree. The two constraints regarding the hierarchy are (1) the product of all numbers describing the hierarchy must be equal to the number of MPI processes, and (2) this hierarchy of the system must be homogeneous: it cannot be used for MPI applications running on a mix of compute nodes with different topologies, e.g., a set of compute nodes with 32 cores and others with 48. Hence, including the network hierarchy in the hierarchy description requires more constraints: the allocated compute nodes must be contiguous leaves of the network tree and their number has to be equal to the total number of nodes plugged to the selected switches (i.e., if the full hierarchy description is $[\![2, 3, 16, 2, 2, 8]\!]$ with the first three numbers describing the network hierarchy, the number of compute nodes has to be $2 \times 3 \times 16 = 96$) and the compute nodes have to entirely fill all selected switches (i.e., selected switches cannot contain compute nodes that are not part of our job).

Once `MPI_COMM_WORLD` has reordered ranks, subcommunicators can be created, for instance by taking as `color` the quotient of the division of the reordered rank in `MPI_COMM_WORLD` by the size of child communicators. According to the order used to reorder ranks, processes belonging to same new subcommunicators will be more or less spread across all compute nodes or packed inside a single level of the hierarchy. The different colors in Figure 2 represent four different subcommunicators of size 4, where the numbers correspond to the reordered ranks of each MPI process in `MPI_COMM_WORLD`. One can notice how changing the order can map the same subcommunicator to different cores, because of the different rank numbering of the global communicator.

3.3 Characterizing Orderings

As illustrated by Figure 2, different orders can give similar mappings of communicators. For instance, with orders $[2, 0, 1]$ and $[2, 1, 0]$ communicators containing ranks 0, 1, 2, 3 and 12, 13, 14, 15 are mapped onto the same cores, and communicators containing ranks 4, 5, 6, 7 and 8, 9, 10, 11 only exchange the sockets they are mapped to. If there is no inter-communicator communications, exchanging the mapping of communicators in these two configurations will not change performance of communications: being mapped on a socket of the first or the second compute node does not change anything to communications done inside a socket. Thus, orders $[2, 0, 1]$ and $[2, 1, 0]$ can be considered as *similar* and evaluating performance with both of them can be redundant since they will exhibit the same performance (as long as there is no inter-communicator communications). Orders $[0, 1, 2]$ and $[1, 0, 2]$ look also similar, but the order of ranks inside subcommunicators is different. This difference can impact performance of collective operations, as it will be seen further in Section 4.1.

Intuitively visualizing how an order maps subcommunicators can be difficult only from the order. We propose two metrics to characterize the obtained mapping: the *ring cost* and the *percentages of process pairs per level*.

Ring cost. This metric helps to understand how rank numbers are assigned to processes belonging to a subcommunicator. It is computed as the cost of a ring communication pattern. For example, in a communicator with four processes, the ring cost is the total duration of data transmission from rank 0 to rank 1, then from rank 1 to rank 2, and so on until rank 4 receives the data. The communication cost between two processes depends on their relative position in the hierarchy: if they are inside the same lowest hierarchy level (e.g., a NUMA node), the communication cost is 1. However, if the

communication has to cross several hierarchy level (e.g., one level when the sender and receiver processes are on different NUMA nodes, but still inside the same socket), the communication cost increases by 1 for each level it has to cross. When the ring cost is low, it means ranks are assigned sequentially (processes inside the same hierarchy levels get contiguous rank numbers) and the ring communication pattern benefits from locality. On the contrary, if the ring cost is high, ranks are assigned in a more round-robin fashion and communications in the ring pattern always cross several levels of the hierarchy. For example, in Figure 2, considering the orders $[0, 1, 2]$ and $[1, 0, 2]$, the communicator containing ranks 0, 1, 2, 3 is mapped with both orders on the same set of cores, but the cores have different ranks inside this subcommunicator: the order $[0, 1, 2]$ has a ring cost of 9 and the order $[1, 0, 2]$ has a ring cost of 7.

Percentages of process pairs per level. This metric represents the percentage of process pairs of a communicator that communicate inside each level of the hierarchy. For each level, process pairs communicating inside the level are counted, excluding process pairs fitting into a smaller level of the hierarchy. For example, on Figure 2, for the order $[2, 1, 0]$, the percentages are $[100, 0, 0]$: all pairs of processes in a subcommunicator can communicate inside the lowest level of the hierarchy. For the order $[1, 0, 2]$, the percentages are $[0, 33.3, 66.7]$: none of the processes pairs belonging to a subcommunicator can communicate by staying inside the lowest level, one third of the pairs have to cross only one level and two thirds have to cross two levels. This set of percentages helps to quantify how communicators are mapped on available resources: if percentages for the lower levels of the hierarchy are high (like for the order $[2, 1, 0]$), it means the mapping is *packed*: processes belonging to the same communicator cover a minimum of higher level resources (e.g., a minimum of compute nodes) and feature high locality; on the contrary, if percentages for the higher levels are high (like for order $[1, 0, 2]$), it means the mapping is *spread*: the set of processes belonging to a communicator covers the maximum of resources (e.g., one process per compute node).

Both metrics are independent: the ring cost can distinguish two orders with the same percentages of process pairs per level.

3.4 Second Use Case: Add More Distributions for Slurm

Commands of the Slurm job scheduler have an option `--distribution` to specify which cores should be selected to execute jobs. It is possible to change the distribution policy of two hierarchy levels: at the compute node level or socket level. The two main distribution policies are *block*: all cores of a compute node (or a socket) are filled with job processes before going to the next compute node (or a socket); and *cyclic*: cores are selected in a round-robin fashion. Captions of rank mappings in Figure 2 show the policies that can be used to achieve presented rank mappings.

The main limitation of this `--distribution` option is the possible levels of hierarchy to apply a distribution policy: only the compute node and the socket levels are available. It is not possible to change process distribution in other hierarchy levels, such as NUMA node, L3 cache or even *fake* levels like explained previously.

When a Slurm job uses all cores of granted compute nodes, it is possible to use rank reordering inside the MPI application, such as explained in Section 3.2. However, when the number of MPI processes is lower than the total number of cores available on granted compute nodes (e.g., the job uses only half of the core per compute node), rank reordering can still be done in the MPI application, but the cores where processes are mapped are already chosen by Slurm.

It is possible to instruct Slurm which cores to use on each compute node with the option `--cpu-bind=map_cpu:list`, where `list` is an explicit comma-separated list of core physical IDs. This option applies to all compute nodes. The order of core IDs in the list has an impact on the MPI rank mapping. The list can be generated with Algorithm 3: after computing the number of cores per compute nodes (Lines 2 to 5), it iterates over all cores and keeps cores selected by the permutation σ, by applying Algorithms 1 and 2. By manually providing the hierarchy h to Algorithm 3, we are able to consider any level of the hierarchy, extending Slurm's capabilities.

Algorithm 3 Generate list of cores for `--cpu-bind=map_cpu`

Inputs: h : hierarchy of one compute node, σ : permutation, n : number of cores to use
Output: l : list of core physical IDs

1: $l \leftarrow []$
2: $N \leftarrow 1$ ▷ Will be the total number of cores on a node
3: **for** $i = 0$ **to** $|h| - 1$ **do**
4: $N = N \times h[i]$
5: **end for**
6: **for** $c = 0$ **to** $N - 1$ **do**
7: $r \leftarrow \text{ComputeNewRank}(h, c, \sigma)$ ▷ Apply Alg. 1 and 2
8: **if** $r < n$ **then**
9: $l[r] \leftarrow c$ ▷ Core c will be on r^{th} position in the array l
10: **end if**
11: **end for**

For applications that do not use all cores of compute nodes, the mixed-radix decomposition technique can be used for two distinct steps: (1) selecting cores and (2) reordering the MPI ranks. For the first step, Algorithm 3 produces, in some cases, the same set of cores in a different order, leading to a different MPI rank mapping. One could choose to keep only distinct sets of cores and then apply MPI rank reordering, as explained in Section 3.2. Yet, this would lead to more combinations to evaluate. The hierarchy used for the second step has also to match the hierarchy formed by the set of cores chosen in the first step. For example, with 2 compute nodes as on Figure 1, if the first step selects all cores of the first socket on both nodes, the hierarchy for the second step will be $[\![2, 4]\!]$; if the first step selects two cores per socket, the hierarchy will be $[\![2, 2, 2]\!]$. Since the depth of these two hierarchies is different, the number of possible orders for reordering MPI ranks is also different.

4 EVALUATION

We evaluated the impact of our enumeration algorithm with custom micro-benchmarks, a real-world application and a conjugate gradient benchmark. To generate all possible orders for a given hierarchy

(i.e., all permutations σ of $|h|$ elements), we use Heap's algorithm [8] and the Python function `itertools.permutations()`.

Machine descriptions. We conducted our experiments on two clusters: *Hydra* and *LUMI*. *Hydra* is our local cluster, consisting of 32 compute nodes with two 16-core Intel Xeon Gold 6130F CPUs (2.1 GHz), with 96 GB of RAM, which are interconnected with two Intel Omni-Path HFI Silicon 100 Series network interfaces. Unless stated otherwise, only one NIC is used to mimic a standard cluster setup. For *Hydra*, we describe its hierarchy as $[\![$number of compute nodes, 2, 2, 8$]\!]$: each compute node has two sockets and we insert a *fake* level to split the 16 cores per socket into two groups and thus explore more mapping possibilities (sub-NUMA clustering is disabled). *LUMI* is a Finnish HPE Cray supercomputer, where the nodes that we used have two AMD EPYC 7763 CPUs at 2.45 GHz with 64 cores each, 256 GB of RAM is partitioned in 8 NUMA nodes and each NUMA node has two L3 caches. The compute nodes are interconnected with a HPE Slingshot-11 200 Gbps network. *LUMI*'s hierarchy is noted $[\![$number of compute nodes, 2, 4, 2, 8$]\!]$. Hyperthreads are disabled on *Hydra* and are not used on *LUMI*. We used OpenMPI 4.1.4 with `gcc` 10.2 on *Hydra* and Cray flavors of MPICH 8.1.23 with `clang` 15 on *LUMI*.

4.1 Micro-benchmarks

4.1.1 Description. We evaluated the impact of different rank reorderings on the performance of collective operations executed in subcommunicators. The experimental protocol is the following:

1. Reorder ranks of `MPI_COMM_WORLD` in a new communicator.
2. Create several subcommunicators, all containing the same number of processes.
3. In the first subcommunicator only, measure the performance of the specified collective operation.
4. Finally, in all subcommunicators simultaneously, execute the specified collective operation and measure its performance.

Rank reordering and subcommunicator creation are performed as explained in Section 3.2: process ranks in `MPI_COMM_WORLD` are reordered according to the chosen order in a new communicator, then this communicator is split using the color `reordered_rank % subcomm_size`. We distinguish the case when either only one communicator or all communicators perform collective operations simultaneously to evaluate the performance difference according to the number of processes communicating simultaneously.

All kinds of collective operations can be evaluated with this protocol, but we choose to evaluate only non-rooted collectives (`MPI_Alltoall`, `MPI_Allreduce` and `MPI_Allgather`), since with rooted collectives (such as `MPI_Bcast` or `MPI_Reduce`), the choice of the root process can have an impact on performance and would add another dimension to the results. Moreover, non-rooted collective operations are usually more communication-intensive, thus they are good candidate to evaluate their performance according to MPI rank mapping. Due to lack of space, we present in this paper mostly results for `MPI_Alltoall` operation; other results are available online [23]. The choice of the algorithm to execute these collective operations (e.g., Bruck, pair-wise, etc.) is left free to the MPI implementation: we do not force a specific algorithm to be always used (results with a fixed algorithm show similar trends).

To measure performance of these collective operations, they are executed until a defined time period has elapsed (0.5 s for these experiments), to reach a steady state between all processes. The initial synchronization between processes to start the time window is done with a call to `MPI_Barrier`. Even if it is not the most precise way, we believe that the duration of the time window mitigates the possible extra delay introduced by this method. The number of iterations of the collective operations executed by each communicator, as well as the actual duration of the time window are used to compute the average duration of one collective call.

4.1.2 Results. In the following figures, the size indicated on the x-axis is the amount of data involved in the collective, i.e., size of the communicator $\times$ `count` $\times$ `sizeof(datatype)` (here the used MPI datatype is `MPI_BYTE`) for `MPI_Alltoall`, `MPI_Allreduce`, and `MPI_Allgather`. The bandwidth on the y-axis is computed as this amount of data divided by the average duration of one collective operation. In each figure, the left plot represents performance when only one subcommunicator (the 'first' one, containing the lowest process rank values of `MPI_COMM_WORLD`, i.e., blue communicators depicted on Figure 2) executes the benchmarked operation, the right one when all subcommunicators execute simultaneously the operation. The plotted curves show the average of bandwidths reported by all processes with rank 0 in each subcommunicators. Error areas are bounded by the first and last deciles among averages reported by all subcommunicators (hence, there is no error area on plots with only one communicator). To ease the reading of the figures, only a subset of possible orders are presented, other orders perform similarly to the orders shown here. Legend items show the permutation used to reorder ranks, as well as the ring cost and percentages of process pairs per topology level for one communicators in the specified order (cf. Section 3.3).

Figure 3 shows the performance of `MPI_Alltoall` operations on 16 *Hydra* nodes with 16 MPI processes per communicator. When there is only one subcommunicator executing the collective operation, the most spread order $[0, 1, 2, 3]$, mapping each process of the subcommunicator on a different compute node, gives the highest bandwidth of 7731 MB/s with a data amount of 3.8 MB (i.e., each process contributes 245 kB to the collective operation). When 32 subcommunicators simultaneously execute `MPI_Alltoall` operations, this order gives the worst performance (not exceeding 360 MB/s) and the best mapping is now $[3, 2, 1, 0]$, the most packed one (all 16 MPI processes belonging to the same subcommunicator are mapped inside the same socket), giving a maximum bandwidth of 3527 MB/s. The rank order inside subcommunicators has no impact on performance of `MPI_Alltoall`: orders $[1, 3, 2, 0]$ and $[3, 1, 0, 2]$ map processes on the same resources (percentages of process pairs per level are the same) but assign MPI ranks in different orders (ring costs are 45 and 17 respectively), yet they present the same performance. We observe similar behaviors on Figure 4, which uses only 4 subcommunicators, but with 128 processes each.

Similar observations can be made on Figure 5, depicting the performance of `MPI_Alltoall` operation on 16 *LUMI* nodes, with 16 processes per communicator. For data size larger than 8 MB, the best order is $[0, 1, 2, 3, 4]$, yet for data sizes between 2 and 8 MB, less spread orders $[1, 2, 3, 0, 4]$ and $[3, 2, 1, 4, 0]$ give superior bandwidth.

The performance results of `MPI_Allreduce` on 16 *Hydra* nodes (Figure 6) and `MPI_Allgather` on 16 *LUMI* nodes (Figure 7) show that both communicator mapping and rank order inside communicators impact the bandwidth of these collective operations.

4.1.3 Common Observations. Although the results on the evaluated platforms are all different and it is difficult to generalize, some common observations can be made.

Packed mappings have constant performance regardless the number of simultaneous communicators executing collective operations. Orders that benefit from locality by mapping processes of the same communicator on a low number of higher-level resources (such as $[3, 2, 1, 0]$ on figures 3 and 4) exhibit the same performance when only one communicator executes a `MPI_Alltoall` operation or when all communicators execute this operation simultaneously. This is expected: with a packed order, communicators use all resources of the lowest levels (e.g., all cores of a NUMA node), hence additional communicators will not share resources of lowest levels and will not disturb other communicators (i.e., the next communicator will use the next NUMA node and will not share the NUMA node already taken by another communicator).

Spread mappings are usually better for single small communicators. As observed in all left plots of previous figures, spread mappings (like $[0, 1, 2, 3]$ on *Hydra*, or $[0, 1, 2, 3, 4]$ on *LUMI*) are better when only one small communicator executes `MPI_Alltoall` operations. It may not reach the peak performance, but these mappings are less impacted by contention for larger messages, as shown on Figure 5.

Rank ordering inside communicators can have an impact, even for non-rooted collective operations. Especially for `MPI_Allgather` and `MPI_Allreduce`, the enumeration order inside a communicator to assign ranks can have an impact on the performance of the collective operation. See for instance Figure 7: orders $[0, 1, 2, 3, 4]$ and $[1, 2, 3, 0, 4]$ map communicators to the same set of cores (same percentages of process pairs per level), but rank orders inside communicators are different (different ring costs). We can observe the order $[1, 2, 3, 0, 4]$ allows higher performance for `MPI_Allgather` operations than $[0, 1, 2, 3, 4]$. The same happens with `MPI_Allreduce` on 16 *Hydra* nodes with 64 processes per communicators (Figure 6): orders $[0, 1, 2, 3]$ and $[2, 1, 0, 3]$ have similar communicator mappings but different rank ordering inside communications and they lead to different performance, mostly due to the collective algorithm.

4.2 A Real-world Application: Splatt

To test the impact of our rank reordering algorithm, we were looking for proxy applications that use non-rooted collective operations in subcommunicators that are smaller than `MPI_COMM_WORLD`. Possible candidates are listed in literature that studies the use of MPI features by proxy applications [14, 22]. Unfortunately, all of these applications either do not use small communicators, do not employ the collective operations we are interested in, or they do not perform enough communication to demonstrate significant performance differences based on the communicator mapping.

Splatt (*The Surprisingly ParalleL spArse Tensor Toolkit* [21]) is a C library and a set of command-line utilities to apply operations on tensors, using non-rooted collective operations on small subcommunicators. We measured the performance of the Canonical Polyadic Decomposition (CPD) operation according to the order used to reorder ranks of `MPI_COMM_WORLD`. Indeed, we follow a black-box approach: we do not modify the application, instead, we change the rank mapping of `MPI_COMM_WORLD` in a transparent way for the application. Splatt uses several communicators with different sizes and the characteristics of these communicators depend on several factors: the input tensor, the number of MPI processes, etc. Therefore, we evaluate the performance of each order.

Figure 8 shows durations of the CPD operation obtained with all possible rank reorderings on 32 *Hydra* nodes with one MPI process per core and `nell-1` as input tensor, from the Frostt collection [20]. The order $[1, 3, 2, 0]$ is the mapping that Slurm would setup by default on *Hydra*, identical to `--distribution=block:cyclic`. Bars represent the average of 3 runs and error bars highlight the best and the worst durations. In the configuration of Figure 8a, only one network interface is used. The Slurm default mapping is one of the inefficient rank mappings: the CPD operation lasts 32.58 s. The best mapping is $[0, 3, 1, 2]$ with 22.09 s, improving the default Slurm mapping by 32 %. With two network interfaces per compute node (Figure 8b), all orders are in average faster than with only one NIC (22.9 s vs. 27.4 s), the worst mapping is $[2, 3, 0, 1]$ (26.25 s) and the best one is still $[0, 3, 1, 2]$ (19.79 s), improving the worst mapping by 25 % and the Slurm one by 19 %. The general performance improvement when using two NICs instead of only one is expected: it allows higher network bandwidth and thus network communications are less a bottleneck for the application performance.

To confirm performance differences are actually coming from time saved on MPI operations, we profiled Splatt execution with `mpisee` [25], a profiling tool focusing on communicators. `mpisee` shows us that during an execution on 32 nodes with 1024 MPI processes and the tensor `nell-1` as input data, Splatt uses 3 communicators containing all 1024 MPI processes, 8 communicators with 256 processes each and 64 communicators containing 16 processes. Most time-consuming collective operations executed inside these communicators are `MPI_Alltoallv`, `MPI_Bcast`, `MPI_Allreduce`, `MPI_Reduce`, `MPI_Alltoall`, `MPI_Scan`, and `MPI_Gather`. The durations of the CPD operation are mainly correlated with the duration of `MPI_Alltoallv` operations executed on communicators with 16 processes, with a Pearson's correlation coefficient of 0.98 and 0.92 when respectively one and two network interfaces are used. This strong correlation shows the performance difference of CPD operations when different rank orders are used originates from the duration of `MPI_Alltoallv` operations, which are impacted by rank mapping, as the previous micro-benchmarks of the similar `MPI_Alltoall` operation showed.

4.3 Strong Scaling of Conjugate Gradient

In this experiment, we evaluate the strong scaling on one *LUMI* compute node of the conjugate gradient (CG) benchmark provided by the NAS Parallel Benchmark [1]. While keeping a constant problem size, we increase the number of MPI processes and change their mapping with the technique described in Section 3.4.

The results are depicted in Figure 9. For each tested number of MPI processes, all enumerations using a different set of cores or

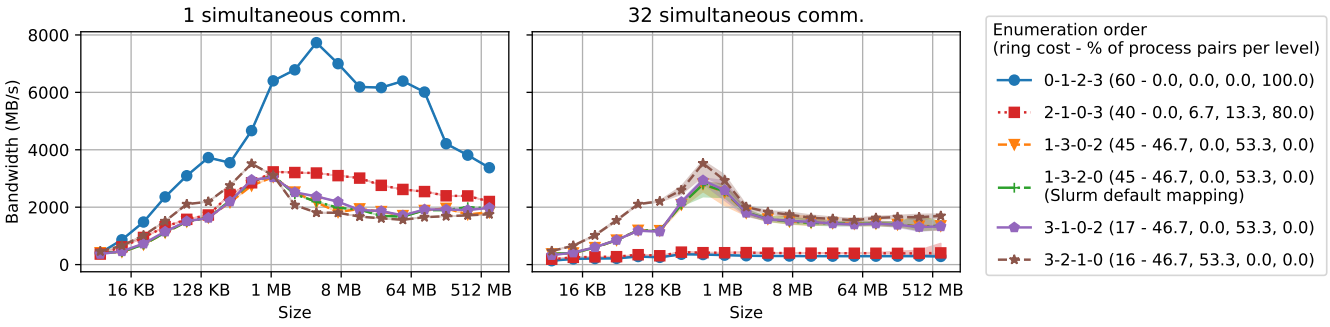


Figure 3: 16 *Hydra* nodes, 512 MPI processes, `MPI_Alltoall`, 16 processes per communicator, OpenMPI 4.1.4

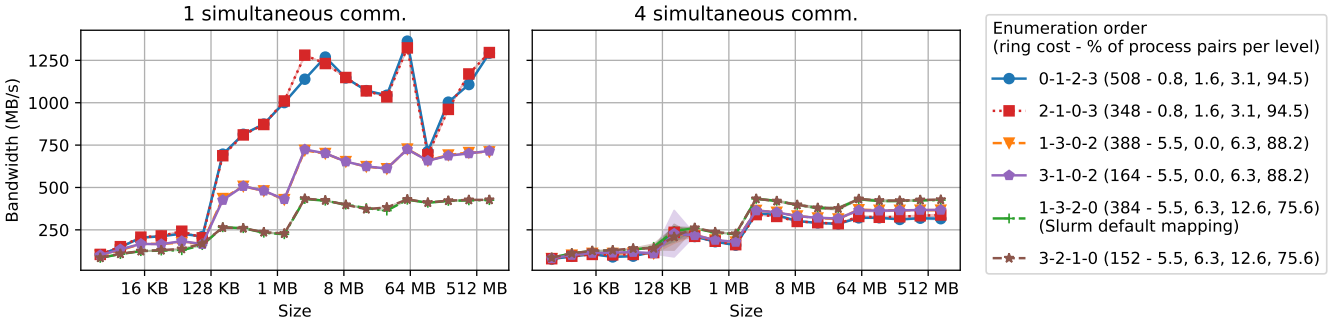


Figure 4: 16 *Hydra* nodes, 512 MPI processes, `MPI_Alltoall`, 128 processes per communicator, OpenMPI 4.1.4

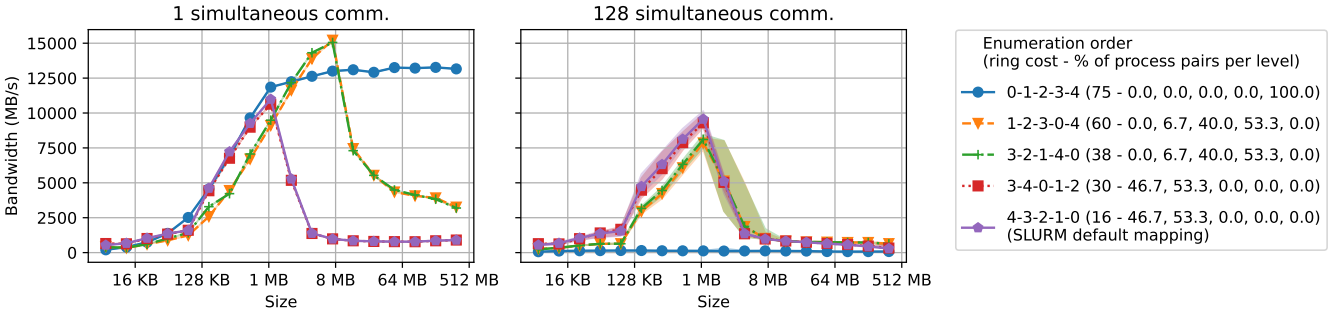


Figure 5: 16 *LUMI* nodes, 2048 MPI processes, `MPI_Alltoall`, 16 processes per communicator, CrayMPI 8.1.23

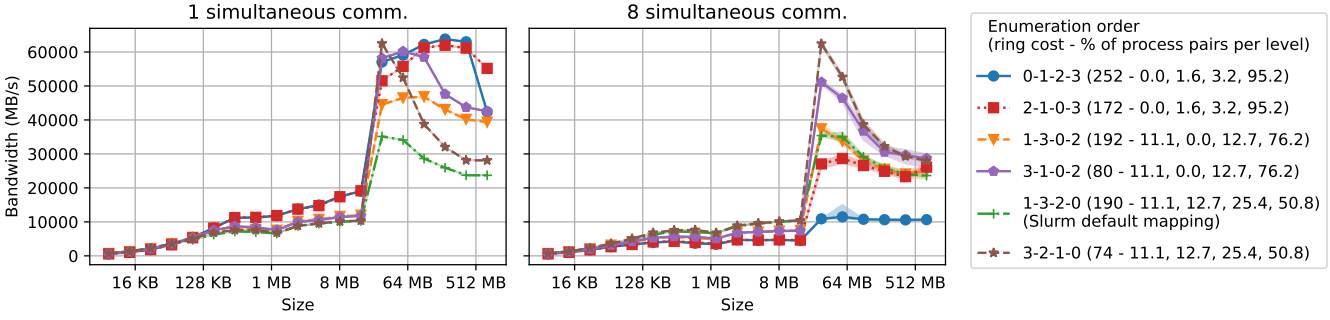


Figure 6: 16 *Hydra* nodes, 512 MPI processes, `MPI_Allreduce`, 64 processes per communicator, OpenMPI 4.1.4

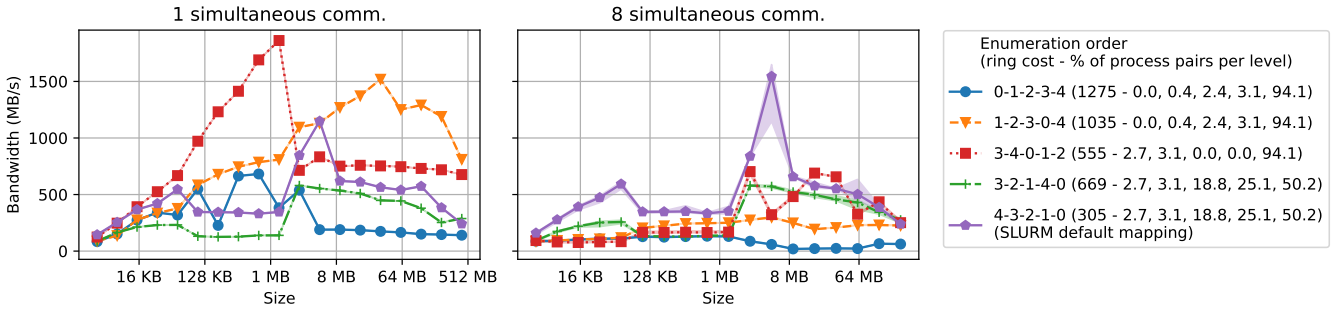
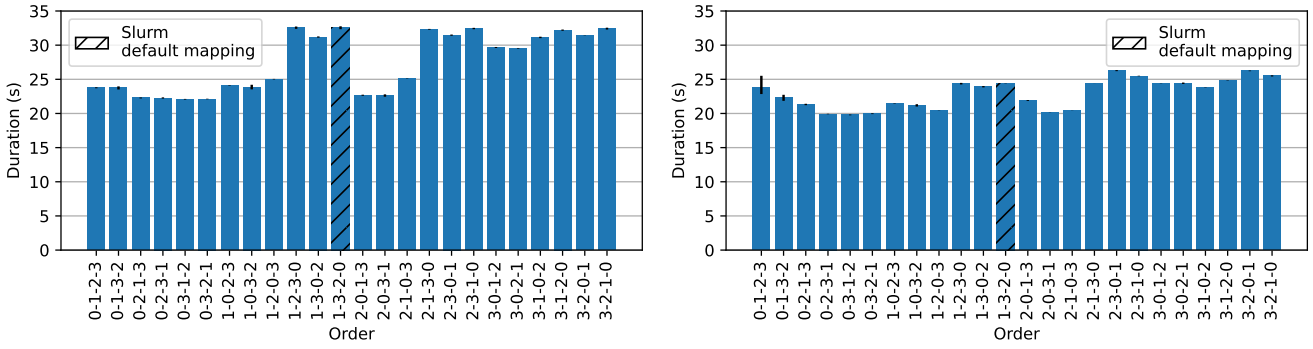


Figure 7: 16 *LUMI* nodes, 2048 MPI processes, `MPI_Allgather`, 256 processes per communicator, CrayMPI 8.1.23



(a) With 1 NIC per compute node

(b) With 2 NICs per compute node

Figure 8: Impact of process mapping of Splatt executions on 32 *Hydra* nodes (1024 MPI processes, OpenMPI 4.1.4)

in a different order are evaluated. Bars correspond to the median duration of 5 executions and error bars delimit the best and worst durations. For a given number of MPI processes, orders with bars of the same color use the same set of cores, but with a different MPI rank mapping. Next to each set of bars using the same set of cores, we put the core IDs, to better understand how the processes are mapped. For instance, with 2 processes, order $[0, 1, 2, 3]$ selects the first core of each socket; with 8 processes, $[0, 1, 2, 3]$ and $[1, 0, 2, 3]$ (the first blue ones) select the first core of each NUMA node. We also highlight the default mapping Slurm performs and display what should be the application runtime if perfect scaling was achieved.

With 4 processes, the best performance is reached with order $[2, 1, 0, 3]$, which uses one core per L3 cache, in the L3 caches of the two first NUMA nodes of the first socket. With 8 processes, the best performance is again achieved with orders using one core per L3 cache of the first socket: $[1, 2, 0, 3]$ and $[2, 1, 0, 3]$. With 16 processes, the best strategy is to use two cores per L3 cache of the first socket (orders $[1, 2, 3, 0]$ and $[2, 1, 3, 0]$). From 16 processes, CG does not scale anymore, the parallel efficiency decreases: the duration of the perfect scaling is never reached, regardless of the mapping. One can also notice that the default Slurm mapping exhibits almost always the longest duration. Especially, using more processes and an inefficient process mapping can degrade the performance compared to fewer processes and the best mapping: For instance, with 8 processes and order $[3, 0, 1, 2]$, CG lasts 22.5 s while with only 4 processes and order $[2, 1, 0, 3]$ it lasts 17.3 s; even worse, with 32 processes and the default Slurm mapping, it lasts 9.4 s while with only 8 processes, the best order $[1, 2, 0, 3]$ gives a duration of 8.1 s. This shows that CG can achieve better performance using only one fourth of the cores with a better mapping.

The set of selected cores has more impact on performance than the MPI rank ordering inside a set of cores: for example, with 32 cores, the duration with the first 6 blue orders, using the first two cores of each L3 cache, ranges between 3.2 s and 4.6 s while with the green orders, using the first 16 cores of both sockets, it ranges between 12.1 s and 14.4 s. Note that in this experiment, as explained in Section 3.4, we use mixed-radix decomposition only to select cores and keep orders with same sets of cores with different rank mappings. Keeping only distinct set of cores and then test different rank mappings would lead to more combinations to evaluate.

This experiment shows an application of our mixed-radix decomposition technique to select cores on a compute node, with more possible policies (in our case the policies are the orders) than Slurm permits.

5 CONCLUSION

As modern HPC systems feature deeper hierarchies, we propose a technique that enumerates the computing units, considers the hierarchical topology of the system and follows a specified order. We present two use cases: reordering MPI ranks of an application

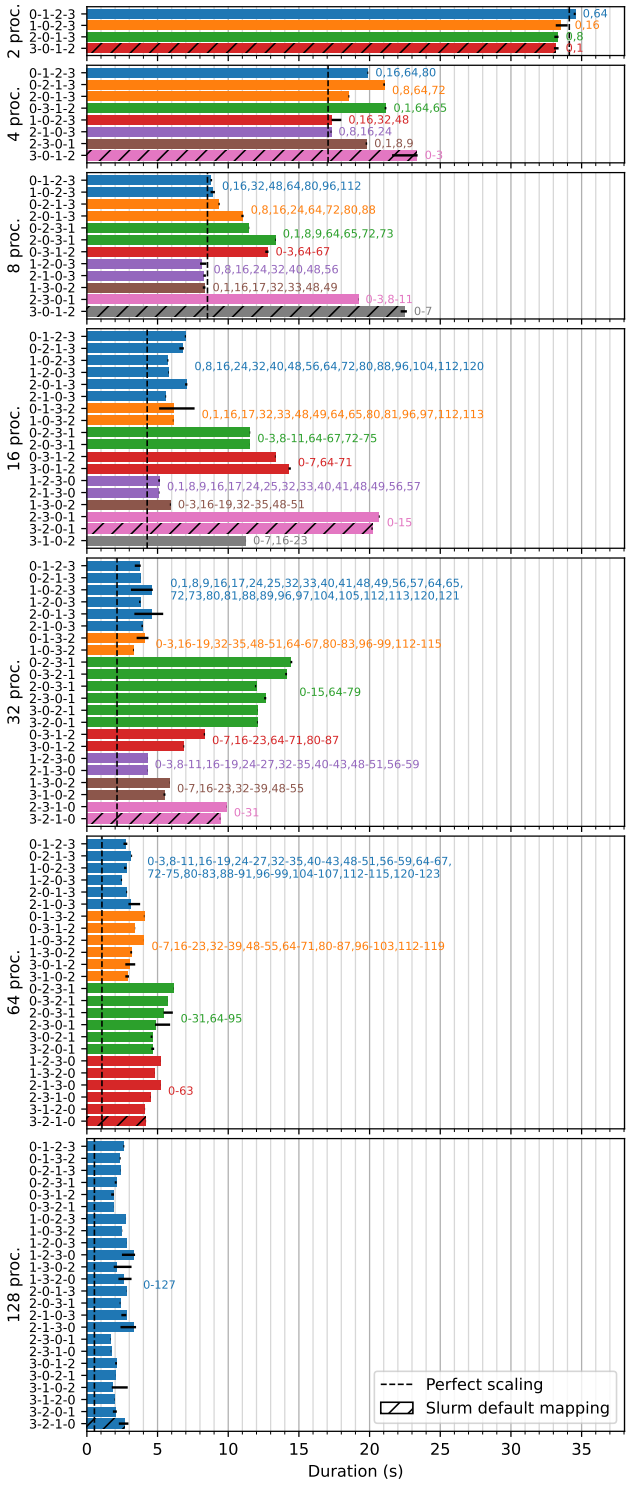


Figure 9: Strong scaling of the CG benchmark (class C) on one *LUMI* compute node (CrayMPI 8.1.23), with different mappings. Different color bars represent the different sets of used cores; numbers on the right of bars are used core IDs.

and selecting cores for Slurm jobs that do not use all cores per compute node. Both use cases allow to explore more mapping possibilities than the one provided by options of MPI job launchers (MPI implementation or Slurm), by taking into account all hierarchy levels the user provides.

Micro-benchmarks executing collective operations in subcommunicators show that packed mappings have constant performance regardless of the number of communicators simultaneously executing the collective operation, spread mappings are better for small communicators when only one communicator executes the collective operation and rank reordering inside communicators can have an impact on performance of non-rooted collectives. Regarding real-world applications, our rank-reordering algorithm allowed to find a mapping that outperforms by 32 % the default mapping setup by Slurm for the Splatt application. The evaluation of a conjugate gradient showed that carefully selecting a few cores can give better performance than inefficient mapping with more cores.

The goal of this paper was to highlight possible applications of mixed-radix decomposition to mapping problems in HPC. The presented use cases could be directly integrated into MPI implementations or job schedulers. MPI runtimes could offer the possible rank orderings as process sets available as MPI sessions, introduced in the Version 4 of the MPI standard. Slurm could provide options with a feature similar to `--distribution`, but which accepts a hierarchy and an order to select cores and/or attribute MPI ranks.

Since the goal of this work is not to try all possible enumerations, future directions regarding this work fall mainly into two categories: understanding performance and automatically applying the best order. Micro-benchmarks showed that performance with different orders can vary according to the evaluated cluster: we would like to better understand which application properties and cluster characteristics impact the performance obtained with different orders. This knowledge could help to predict which order is the most suitable for the used system and applications. We plan to implement strategies in MPI libraries to reorder ranks and create communicators (especially by splitting them) in a hierarchy-sensitive way, to achieve better parallel performance. Finally, we are interested in making our algorithm more general and dynamic: being able to follow an order for a set of communicators and another order for remaining communicators and to have subcommunicators with different sizes.

Software Availability. A public companion[1] contains the instructions to reproduce our study.

ACKNOWLEDGMENTS

This work was partially supported by the Austrian Science Fund (FWF): project P 31763-N31. We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Regular Access call.

[1] https://gitlab.tuwien.ac.at/philippe.swartvagher/paper-mpi-rank-reordering-r13y, archived on https://www.softwareheritage.org/ with the ID `swh:1:dir:a14ef5edfeeb677e396c9986dc90eae0042c03cd`

REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. 158–165. https://doi.org/10.1145/125826.125925

[2] Abhinav Bhatele, Todd Gamblin, Steven H. Langer, Peer-Timo Bremer, Erik W. Draeger, Bernd Hamann, Katherine E. Isaacs, Aaditya G. Landge, Joshua A. Levine, Valerio Pascucci, Martin Schulz, and Charles H. Still. 2012. Mapping Applications with Collectives over Sub-Communicators on Torus Networks. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 97, 11 pages.

[3] Amanda Bienz, Luke Olson, and William Gropp. 2019. Node-Aware Improvements to Allreduce. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. 19–28. https://doi.org/10.1109/ExaMPI49596.2019.00008

[4] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE (Ed.). https://doi.org/10.1109/PDP.2010.67

[5] Nicolas Denoyelle, Emmanuel Jeannot, Swann Perarnau, Brice Videau, and Pete Beckman. 2021. Narrowing the Search Space of Applications Mapping on Hierarchical Topologies. In *PMBS21 Workshop - 12th IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, In conjunction with SC21*. https://doi.org/10.1109/PMBS54543.2021.00018

[6] Brice Goglin, Emmanuel Jeannot, Farouk Mansouri, and Guillaume Mercier. 2018. Hardware topology management in MPI applications through hierarchical communicators. *Parallel Comput.* 76 (2018), 70–90. https://doi.org/10.1016/j.parco.2018.05.006

[7] William D. Gropp. 2019. Using node and socket information to implement MPI Cartesian topologies. *Parallel Comput.* 85 (2019), 98–108. https://doi.org/10.1016/j.parco.2019.01.001

[8] B. R. Heap. 1963. Permutations by Interchanges. *Comput. J.* 6, 3 (Nov. 1963), 293–298. https://doi.org/10.1093/comjnl/6.3.293

[9] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. 2014. An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing. In *High Performance Computing on Complex Environments*, Emmanuel Jeannot and Julius Zilinskas (Eds.). Wiley, 75–94. https://doi.org/10.1002/9781118711897.ch5

[10] Torsten Hoefler and Marc Snir. 2011. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. 75–84. https://doi.org/10.1145/1995896.1995909

[11] Emmanuel Jeannot and Richard Sartori. 2023. An introspection monitoring library to improve MPI communication time. *The Journal of Supercomputing* (16 Feb 2023). https://doi.org/10.1007/s11227-023-05084-8

[12] Krishna Kandalla, Hari Subramoni, Abhinav Vishnu, and Dhabaleswar K. Panda. 2010. Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8. https://doi.org/10.1109/IPDPSW.2010.5470853

[13] Nicholas Karonis, Bronis Supinski, Ian Foster, Ewing Lusk, and John Bresnahan. 2000. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. *Proceedings of the International Parallel Processing Symposium, IPPS* (03 2000). https://doi.org/10.1109/ipdps.2000.846009

[14] Benjamin Klenk and Holger Fröning. 2017. An Overview of MPI Characteristics of Exascale Proxy Applications. In *High Performance Computing: 32nd International Conference, ISC High Performance 2017*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Frankfurt, Germany, 217–236. https://doi.org/10.1007/978-3-319-58667-0_12

[15] D.E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley.

[16] Oh-Kyoung Kwon, Ji-hoon Kang, Seungchul Lee, Wonjung Kim, and Junehwa Song. 2023. Efficient Task-Mapping of Parallel Applications Using a Space-Filling Curve. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. 384–397. https://doi.org/10.1145/3559009.3569657

[17] Shigang Li, Yunquan Zhang, and Torsten Hoefler. 2018. Cache-Oblivious MPI All-to-All Communications Based on Morton Order. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 542–555. https://doi.org/10.1109/TPDS.2017.2768413

[18] Guillaume Mercier and Emmanuel Jeannot. 2011. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface (EuroMPI'11)*. 39–49.

[19] Seyed H. Mirsadeghi and Ahmad Afsahi. 2016. Topology-Aware Rank Reordering for MPI Collectives. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1759–1768. https://doi.org/10.1109/IPDPSW.2016.139

[20] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. http://frostt.io/

[21] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 61–70. https://doi.org/10.1109/IPDPS.2015.27

[22] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. 2021. Understanding the use of message passing interface in exascale proxy applications. *Concurrency and Computation: Practice and Experience* 33, 14 (2021). https://doi.org/10.1002/cpe.5901

[23] Philippe Swartvagher, Sascha Hunold, Jesper Larsson Träff, and Ioannis Vardas. 2023. Results of MPI mixed-radix decomposition reordering algorithm. https://doi.org/10.5281/zenodo.8272970

[24] J.L. Träff. 2002. Implementing the MPI Process Topology Mechanism. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 28–28. https://doi.org/10.1109/SC.2002.10045

[25] Ioannis Vardas, Sascha Hunold, Jordy I. Ajanohoun, and Jesper Larsson Träff. 2022. mpisee: MPI Profiling for Communication and Communicator Structure. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 520–529. https://doi.org/10.1109/IPDPSW55747.2022.00092

[26] Jin Zhang, Jidong Zhai, Wenguang Chen, and Weimin Zheng. 2009. Process Mapping for MPI Collective Communications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*. 81–92. https://doi.org/10.1007/978-3-642-03869-3_11