Narrowing the Search Space of Applications Mapping on Hierarchical Topologies

Nicolas Denoyelle* Emmanuel Jeannot† Swann Perarnau* Brice Videau* Pete Beckman*
*Argonne National Laboratory {ndenoyelle, swann, bvideau}@anl.gov, beckman@mcs.anl.gov
†Inria Bordeaux Sud-Ouest emmanuel.jeannot@inria.fr

***Abstract*—Processor architectures at exascale and beyond are expected to continue to suffer from nonuniform access issues to in-die and node-wide shared resources. Mapping applications onto these resource hierarchies is an on-going performance concern, requiring specific care for increasing locality and resource sharing but also for ensuing contention. Application-agnostic approaches to search efficient mappings are based on heuristics. Indeed, the size of the search space makes it impractical to find optimal solutions nowadays and will only worsen as the complexity of computing systems increases over time. In this paper we leverage the hierarchical structure of modern compute nodes to reduce the size of this search space. As a result, we facilitate the search for optimal mappings and improve the ability to evaluate existing heuristics. Using widely known benchmarks, we show that permuting thread and process placement per node of a hierarchical topology leads to similar performances. As a result, the mapping search space can be narrowed down by several orders of magnitude when performing exhaustive search. This reduced search space will enable the design of new approaches, including exhaustive search or automatic exploration. Moreover, it provides new insights into heuristic-based approaches, including better upper bounds and smaller solution space.**

***Index Terms*—Multicore, NUMA, Threads, MPI, OpenMP, Mapping, Placement, Topology, Hierarchy**

I. Introduction

The U.S. Department of Energy report on extreme heterogeneity [1] established that high-performance computing (HPC) servers at exascale and beyond will exhibit a deep and large hierarchy of computing resources and suffer from nonuniform access issues. Therefore, assessing locality of data access is a growing issue when optimizing application performance. The task becomes increasingly difficult, however, because of the growing complexity of HPC systems.

HPC systems are composed of several nodes interconnected with a high-performance network. At the scale of a single node, several chips are interconnected with a coherent network. At the chip scale, AMD EPYC processors are divided into interconnected chiplets [2] while Intel KNL [3] and Skylake have a "Sub-NUMA Cluster" [4] mode exposing local areas of the chip with local memory. Each of these architectures has multiple layers of cache organized into a memory hierarchy, down to cores sharing several hyperthreads. Extreme-scale machines with multiple accelerators per node will exhibit similar properties and issues, for example, nonuniform interconnect performance between hosts and devices.

Distributed and multithreaded applications can respectively map their processes and threads on computing resources[1] of a cluster. Some applications may see significant performance [5]–[7] variations from one mapping to another. However, interactions between an application mapping and memory subsystems are hard to systematically evaluate in order to find the best mapping. High-performance scientific applications running on such architectures may hit different bottlenecks across the machine. For instance, in-die topology and network properties can be exploited to optimize the mapping of OpenMP reduction routines on a KNL chip [8]. The mapping of threads can also impact performance through cross-die network access and NUMA effects [9]. One can use the knowledge of the coherency protocol and runtime information about coherency messages [10] to capture more of the relevant data movements and take decisions accordingly with a mapping algorithm [11]. Similarly, lower in the hierarchy, sharing cache space with some threads may improve locality [12] of shared-memory accesses. Sharing cache with the wrong threads, however, can cause contention [13] and harm performance. This locality and balance trade-off exists also higher in the hierarchy at the memory level and may arise from different reasons [9]. Multiple other works have addressed the mapping problem through the prism of balancing data locality and resource contention [14]–[16]. This shows that automatically finding a good application mapping is a tedious optimization problem.

Given an application with n threads to map on n distinct compute resources, there are $n!$ possible mappings. It would take years for any contemporary HPC processor and with an application lasting about a second to try every configuration. Accounting for undersubscribing or oversubcribing the machine will further enlarge this number with additional mapping possibilities. Therefore, approaches to find a good application mapping on the machine are often based on heuristics [11], [12], [17] sometimes involving machine learning approaches [18], [19], simulations [20] or even combined with exploratory methods [20]. A complementary step to these approaches is to narrow the search space to remove mappings that would lead to similar performance. A smaller search space may allow for exhaustive search, better heuristic designs, or faster exploration or even finding more realistic performance

[1]The smallest computing resource on which instructions can be explicitly pinned is called an execution unit here.

bounds. To the best of our knowledge, the latter approach has yet to be addressed.

In this paper we propose to leverage the hierarchical structure of machine topology to narrow the mapping search space. The main idea is that machine hierarchies are essentially built from interconnecting clones of identical resources with—often—symmetric links. Therefore, many mappings consisting in exchanging resources of the machines will hit the same hardware limitations. According to this idea, a machine with 16 execution units (or hyperthreads) has 16! possible permutations of mapping when mapping one thread or process per unit. Of these, only $1/294912$ (see Section III for details on this number) may yield significantly different execution times if exchanging clone resources via the application mapping yields the same performance. Hence, accounting for similar resource permutations would effectively reduce the size of the search space by many orders of magnitude.

Prior works [21]–[24] suggest "topology-aware" mapping techniques leveraging this kind of information to optimize applications. These approaches are consistent with the idea developed in this paper, However, whereas they assume what we formalize and experimentally validate, our contribution is broader (as explained below). On the other hand, exploratory approaches [19], [25]–[27] to thread and process mapping usually consider only mapping policies, dramatically narrowing the search space but also potentially missing performance improvement opportunities.

Our contributions in this paper are as follow.

1) We define and describe a structure in the space of possible mappings on hierarchical machine topologies, and we validate the existence of such a structure experimentally.
2) We provide a methodology and algorithms to explore the space of possible mappings more efficiently, namely, by skipping structurally equivalent mappings. We also quantify the exploration space reduction.
3) We define an analytic classification methodology to label applications with respect to their mapping on the machine as to whether they behave according to the mapping space structure we define in this paper.

These contributions will lead to designing better mapping heuristics, to exploring the mapping space faster, creating new mapping policies, or even finding better upper and lower bound estimators of mapping performance.

The remainder of this paper is organized as follow. Section II motivates the characterization of the search space with an example and lists foreseeable caveats to the formulated hypothesis on equivalent mappings. Section III describes how to sample permutations inside and across mapping classes of equivalent performance. Section IV details the experimental setup and experimental results when assessing the existence of mapping performance classes.

II. Motivation

HPC node resources are organized in a hierarchy. In contemporary machines, resources at the same level are usually of the same type, from the silicon to the whole system. For instance, hardware threads and caches are homogeneous and multisocket systems use the same dies. In the case of a distributed HPC system, nodes interconnected through the network also have the same components. The link between the same type of resources is usually symmetric. Some clusters also interconnect their nodes with a (symmetric) fat-tree network topology [28]. Processes and threads mapping on such structures have been shown to significantly benefit some applications [6], [7], [29], [30]. Moreover, leveraging the machine topology has also been shown to improve the performance of some application.

Figure 1 depicts an instance of a system organized in a hierarchical manner. This figure is obtained with hwloc [31] and uses the same naming scheme, where processing units (`PUs`), also frequently named hyperthreads, are the smallest execution entities. This machine has sixteen PUs. Each core of the machine hosts two identical PUs, and cores are further grouped by four, sharing an L3 cache. The machine has two L3 caches. For the sake of the example, let us suppose a hypothetical application is running on this machine. The application is composed of two threads (T:0 and T:1) mapped on PU:10 and PU:11, respectively. We conjecture that T:0 and T:1 can be exchanged without impacting the execution time. Indeed, they share common parent resources, the link between them is likely symmetrical, and PUs executing them are identical. Likewise, T:0 could be mapped to any core from 0 to 3, and T:1 could be mapped to any distinct core under the first L3 cache without impacting execution time. Indeed, they share common parent resources, the link between them is likely symmetrical, and cores executing them are identical. The generalization of these assumptions is expressed in the following property. The validation this property is investigated in Subsection IV-C.

Property of Similar Mappings on Hierarchical Topologies *Two mappings of computations onto compute resources are equivalent in execution time if the data paths and system resources are symmetrical from one mapping to the other.*

In the next section we define mapping classes where all placements are expected to have similar performance. Then we take advantage of symmetries in hierarchical topologies to identify these classes and prune the space of placements to avoid the redundant ones. Not accounting for permutations in the same performance group can dramatically narrow the space to explore when looking for the best mapping. For instance, consider a 1:1 mapping of 1 thread per PU. For a machine with n PUs, a total of $n!$ permutations are possible. If we had a supercomputer with a million nodes like the one in Figure 1, it would take 1.3 years of uninterrupted parallel runs to try all the placements for an application with an average execution time of 2 seconds. Using the mappings property to avoid testing equivalent mappings on this hierarchical topology would make it possible to find an optimal mapping in a reasonable timeframe (see Section III), *e.g.* a few hours.

In practice, hierarchical topologies such as the one depicted in Figure 1 may have nonsymmetrical attributes. For instance,

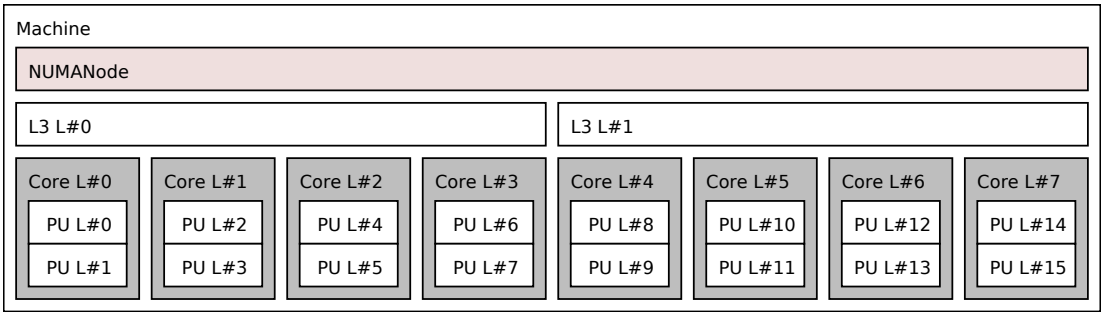


Fig. 1: hwloc hierarchical representation of a machine topology

the cross-chip network might not be symmetric [32] and so can be the routing algorithm [8]. However, the nonsymmetrical link will alter mapping performance only if the former is a bottleneck. Pinning data in memories of a NUMA system with local and remote memories can also break the symmetries. For instance, consider a NUMA system with 2 memories and 2 groups of PUs local to each memory and an application with two groups of threads. Let us consider the scenario where all of the application data is mapped in the first memory and where the group of threads close to this memory is the only one accessing the data memory. If the two groups of threads are swapped, the group making memory access will be far from the memory holding the data and will slow down the application. Hence, both thread mappings would not be equivalent. In practice these caveats have workarounds. Many applications and runtime systems will pin their threads statically for the span of a run [29]. Moreover, they will allocate their data with the default linux *firsttouch* allocation policy. As a result, from run to run, when changing thread placement, data will follow threads and preserve the symmetries. Additionally, hardware and runtime implementations balancing resources usage (e.g., *interleave* allocation policy) will—by definition—generate symmetry in the machine with respect to data mapping. Even in the case where some components do not have symmetrical links, some hierarchical subcomponents do, and the above property can be leveraged there to narrow down the size of the mapping space. Furthermore, dark silicon designs [33] may dynamically change on-chip data paths without the user and the operating system having knowledge of any change. Therefore, dark silicon may disrupt measurable performance at chip scale when exploring mappings.

III. Generating Permutations

By definition, a mapping is a function that specifies for each thread (or process) which processing unit (PU) is executing it. It can be expressed as a list of size n (n is the number of threads or processes) where element i of this list is the PU that executes thread or process i. We consider the case where threads are mapped to a single PU to avoid unsolicited migrations and likely performance degradation [29]. In the following, the number of threads or processes is often the same as the number of PUs. In this case we will also talk about a permutation to specify a mapping of n threads on n PUs. In this section we discuss how different mappings (*i.e.* , different permutations) are in the same class of performance because of the symmetries in the underlying topology.

TABLE I: Examples of thread index permutations on the 16 PUs (first line) of topology 1

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2.	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
3.	0	15	2	3	4	5	6	7	8	9	10	11	12	13	14	1

Let us consider a 1:1 mapping of 1 thread per PU on topology 1. Table I exhibits possible permutations (labeled with the first column) on the 16 PUs of the topology (labeled in the header). The second permutation *ii)* is obtained from the first permutation *i)* by interchanging L3 caches on the topology. Both mappings belong to the same permutation group, namely, the group with the base permutation *i)* and generated by enumerating all permutations of the topology nodes. However, permutation *iii)* cannot be obtained by permuting topology nodes of the same level. It belongs to another subgroup (or *class*) of permutations. This example exhibits equivalent thread permutations (with respect to execution time), obtained by swapping topology resources, and nonequivalent permutations where threads cross the boundaries of topology objects.

We use the term Σ_n to denote the symmetric group [34] of all permutations of n elements. Σ_n contains $n!$ elements, where the ith permutation is $\sigma_{n,i}$. Let τ_n be a rooted tree with a total of n leaves. Then $\sigma_{n,i}$ can be mapped on the leaves of τ_n by associations of the leaves with the elements of the permutation $\sigma_{n,i}$. According to our property on mappings, with τ_n a machine topology and $\sigma_{n,i}$ a permutation of threads mapped on τ_n, we can shuffle the children of an arbitrary node of the topology and yield an equivalent permutation at the leaves. The subgroup $\Sigma_{n,i}$ of equivalent permutations can be generated from $\sigma_{n,i}$ in a similar way, namely, by enumerating all possible shuffles of τ_n nodes children and observing resulting permutations at the leaves. Note that $\Sigma_{n,i} \subseteq \Sigma_n$ and that both sets are equal if τ_n has only one level.

A *tleaf* is a tree where all nodes at the same level have the same number of child nodes. We call the number of children of a node *arity*. Most compute node topologies (for instance,

TABLE II: Exploration space size for several hypothetical topologies. Topology structure is expressed as level of arity. $|\Sigma|$ is the total number of permutations. $|\Sigma_{n,i}|$ is the number of permutations inside a subgroup.

Topology	$\lvert\Sigma\rvert$	$\lvert\Sigma_{n,i}\rvert$	Number of Subgroups
2, 4, 2, 1, 1	2.1E+13	2.9E+05	7.1E+07
2, 14, 1, 2, 2	2.0E+182	2.9E+47	6.7E+134
4, 16, 1, 4, 1	8.6E+506	9.9E+143	8.7E+363

the one in Figure 1) exhibit such a property. In the case of a non-tleaf tree, nodes with a different subtree structure cannot be swapped because they are not symmetric. When subtrees do not have the same amount of leaves, the mapping consisting of exchanging child leaves would simply not fit all the threads on one side. When subtrees have different structures, they induce different interactions with the hardware. Therefore, exchanging them might result in a different execution time even if they have the same number of leaves. For instance, removing Core:0, PU:8, and PU:10 on topology 1 is a possible instance of an imbalanced topology. In this case, left and right resulting L3 subtrees have the same amount of leaves but different structures. Therefore, threads under the first and second L3 caches cannot be swapped under the property of similar mappings on hierarchical topologies.

$$|\Sigma_{n,i}| = \prod_{l \in Levels} (arity(l)!)^{|l|} \quad (1)$$

When the topology is a tleaf, the cardinality of subgroups $|\Sigma_{n,i}|$ generated from permutations $\sigma_{n,i}$ can be computed as in Equation 1, where $|l|$ is the number of nodes of a level l. Table II reports an estimate of the size of the search space for several tleaf topologies (in rows). The table is organized in four columns. The first column describes a hypothetical topology by enumerating the arity of the nodes for each of its levels, for example, sockets per motherboard, NUMA nodes per socket, L3 caches per NUMA node, cores per L3 cache, or PUs per core. The second column provides the number of possible mappings. The third one gives the size of a mapping class, namely, the number of mappings inside each class where all mappings yield similar execution times. The last column gives the number of mapping classes, namely, the size of the space to explore to cover all performance classes. The first line represents the topology in Figure 1. A subgroup of this topology contains $294\,912$ permutations. Therefore, the amount of trials to explore all relevant placements in this case is decreased by 5 orders of magnitude. The second machine represents a contemporary bisocket node. The last topology represents an hypothetical accelerator. The amount of possible permutations grows so fast that even a reduced space exploration is impractical to explore entirely. In either case, however, it remains relevant to assess similarities inside permutation subgroups to be used as a heuristic for direct or exploratory approaches.

To sample the space of distinct subgroups, one must identify them. We do so by identifying a representative permutation in the subgroup, using a bottom-up approach on the corresponding tree. Starting from the leaves, we can uniquely identify them by their index in the permutation $\sigma_{n,i}$. For each node above leaves, we can sort child leaves with a unique result since the indexes of leaves are distinct. We can also tag these nodes with the smallest child leaf index. Sorting the childen of parent nodes based on this index also yields a unique result since indexes are necessarily distinct. By recurrence, from the leaves to the root, the sort operation on τ_n yields a unique result. Using Algorithm 1 on any permutation results in a unique representative permutation of its subgroup. We call it the *canonical* permutation of the subgroup. In this algorithm, $makeIndex(\tau_n, \sigma_{n,i})$ is the procedure that labels leaves of τ_n with the indexes of $\sigma_{n,i}$. In a tleaf topology, one can uniformly sample the set of canonical permutations by generating random permutations and computing their canonical representation.

Algorithm 1 Canonical Permutation.

procedure CANONICAL(τ_n,$\sigma_{n,i}$)
  if $isRoot(\tau_n)$ **then**
    $makeIndex(leaves(\tau_n, \sigma_{n,i}))$
  end if
  if $isLeaf(\tau_n)$ **then**
    return
  end if
  for all $c \in children(\tau_n)$ **do**
    CANONICAL($c, \sigma_{n,i}$)
  end for
  $children(\tau_n) \leftarrow reorderByIndex(children(\tau_n))$
  $index(\tau_n) \leftarrow index(firstChild(\tau_n))$
end procedure $\rightarrow$ $\sigma_{n,i}$

In summary, this section described the tools we use in our experiments to sample the space of permutations, the space of permutation subgroups, and the space of intra-subgroup permutations to validate our property on mappings.

IV. Experimental Results

This section starts with a description of the experimental testbed. We further show a sample of execution times inside and across mapping classes for this testbed. With this sample, we visually assess that placement classes have a different performance profile whereas placements inside classes have a similar performance profile even when applications are sensitive to the mapping of their threads or processes. We define an analytic methodology to label applications whether they have significant differences between classes, and whether they have significant differences inside classes. This methodology can further be used to validate our property on mappings with another testbed.

A. Experimental Testbed

Experiments in this section have been carried out on the PlaFRIM computing platform. The web page[2] provides extensive details about the compute nodes. Most relevant details are

[2]https://www.plafrim.fr

summarized in Table III. We used five multisocket nodes with different hierarchies. They feature the two main HPC system chips providers.

Our experimental testbed includes some benchmarks from the NAS parallel benchmarks (version 3.4.1) suite and LULESH proxy application (version 2.0). In the remainder of the section we use *proxy applications*, *workloads*, *benchmarks*, or *applications* to refer to these. We compiled and ran all workloads using either plain (i.e., not MPI+X) MPI or OpenMP implementations. In both cases, the compiler was gcc (9.2.0), and the MPI implementation was OpenMPI (4.0.3).

Applications were mapped on a basis of one MPI process or OpenMP thread per core. On all systems of the platform, hyperthreading was disabled [3] leaving a single PU per core. Therefore, we use interchangeably the terms PUs and cores in the remainder of the section. The number of processes or threads was adjusted to fit the application constraints while spanning as many PUs as possible in the machine. In some cases the workloads required the number of processes to be a square number or a cubed number or a power of 2. Incomplete lists of processes or threads were mapped on the leftmost part of the machine topology and were completed with fake processes or threads to compute permutations based on a whole list of hardware resources. When computing intraclass or cross-class permutations with fake processes or threads, assumptions of the property (of similar mappings on hierarchical topologies) were trivially kept intact. All benchmarks were run with the Linux default *firsttouch* memory allocation policy. Therefore, data was allocated at the page granularity close to the threads touching them first and followed the latter when threads were mapped differently.

All the tools we used to cover experiment cases described in the paper are publicly available. We gathered machine topology using the *lstopo* utility from hwloc [31]. We developed the *tmap* software to collect tree representations of machine topology, generate permutations, and map permutations on trees. tmap can shuffle mapped permutations inside a class or across classes. It can also identify every permutation with a unique identifier and the permutation class they belong to with Algorithm 1. This software can be found online at the following url: https://github.com/NicolasDenoyelle/tmap. The binding of threads and processes was enforced and tested with *starbind* and *tmap*. We also published the former online at the following url: https://github.com/NicolasDenoyelle/starbind.

For all experiments, a *sample* consists of the combination of a machine, a benchmark, a benchmark input, a programming model (MPI or OpenMP), a number of processes or threads, and a mapping (also placement). In the case of the LULESH proxy application, the input is not mentioned in the figures for clarity. The MPI LULESH runs use the option `-i 100` while the OpenMP runs use the option `-i 300` to keep the execution time small and not overburden the platform. For the NAS benchmarks, the application name mentions the input class.

[3]Hyperthreading is disabled by the platform to ensure reproducibility.

B. Empirical Assessment of Mappings Similarities

For the first experiment, we drew twenty random classes and twenty random placements per class and ran them with several sets (*test cases*) of a machine, an application, and a programming model. Among the twenty classes, one class is the *canonical class* (1,2,3,...); and inside each class, one placement is the canonical class placement (see Section III). Each placement of a test case was run fifty times. When representing the execution time of a placement, we show the average value with an error bar representing the 95% confidence interval. The mean and confidence interval obtained with a one sided t–test require that the samples distribution of a placement be uniform. We performed a Shapiro–Wilk test [35] on placement distribution and found that only 35% of the placements have a normal distribution, namely, with a *p-value* > 0.05. In fact, only 30% of the test cases have more than half of their placements exhibiting a uniform distribution. We tried to troubleshoot this outcome with a system we administer. We ran all applications with a default round-robin mapping and collected 200 samples from each test case. We took care to minimize possible sources of noise, we made sure that the applications are deterministic, and we set the CPU frequency governor to *performance* mode in order to get more stable runs. Despite these efforts, some test cases still exhibit a multinomial distribution. Since the effect is observed on a same test case and placement, it is assumed to be independent from the experiment itself. According to the central limit theorem, the distribution obtained by combining (here in the execution time) a large enough number of independent variables is uniform. Therefore, we consider here that our sample distributions can be summarized accurately enough with their mean and confidence interval.

We represented the average execution time and 95% confidence interval (y-axis) of all twenty classes (x-axis) in Figure 2. Classes contain twenty different placements, and placement execution times contain fifty samples each. Each facet of the plot represents a different test case. For each test case, we sorted classes from the smallest class average time to the highest class average time. Note that we could have obtained smaller intervals by acquiring more samples. Because of the already heavy load on the experiment platform, however, we refrained from doing so to avoid disrupting other users. Figure 2 shows statistically significant differences between most pairs of classes in all test cases, even those with variations of only a few percents across classes. For most test cases where the best class clearly detaches itself from the others, the best class is often the canonical class. However, this class represents less than 50% of the test cases' best class. Among test cases with the most dramatic speedups, the canonical class is not necessarily the best class. For instance, for the case *cg.C - brise - 64 - OpenMP*, the canonical class is not the best class.

In a similar fashion, we show placement performance in Figure 3 to observe what is happening inside classes. For each test case we sorted the placements per class, from the smallest

Fig. 2: Comparison of average execution times of mapping classes for different workloads, machines, and programming models.

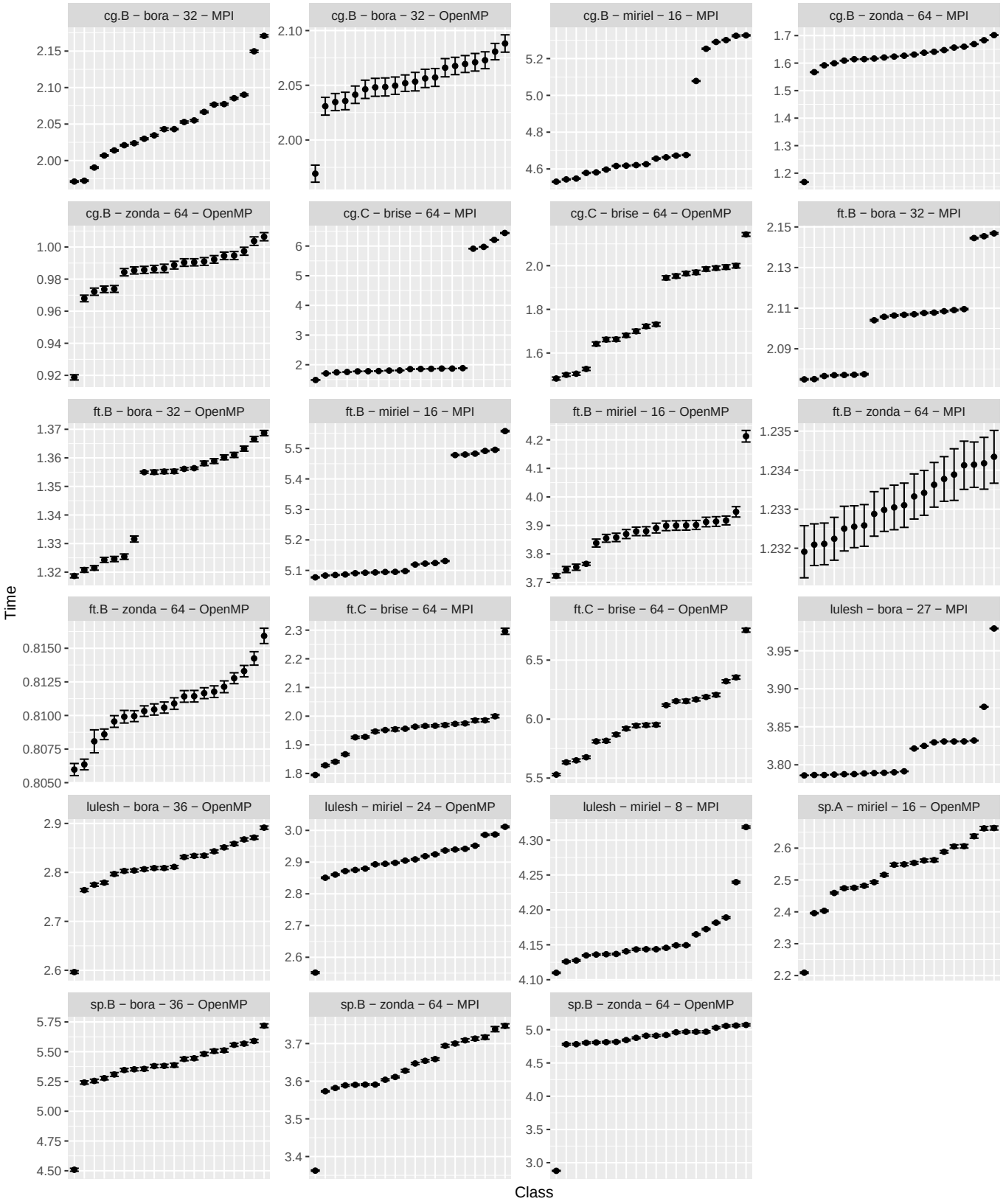


Fig. 3: Comparison of average execution times of mappings inside and across classes for different workloads, machines, and programming models.

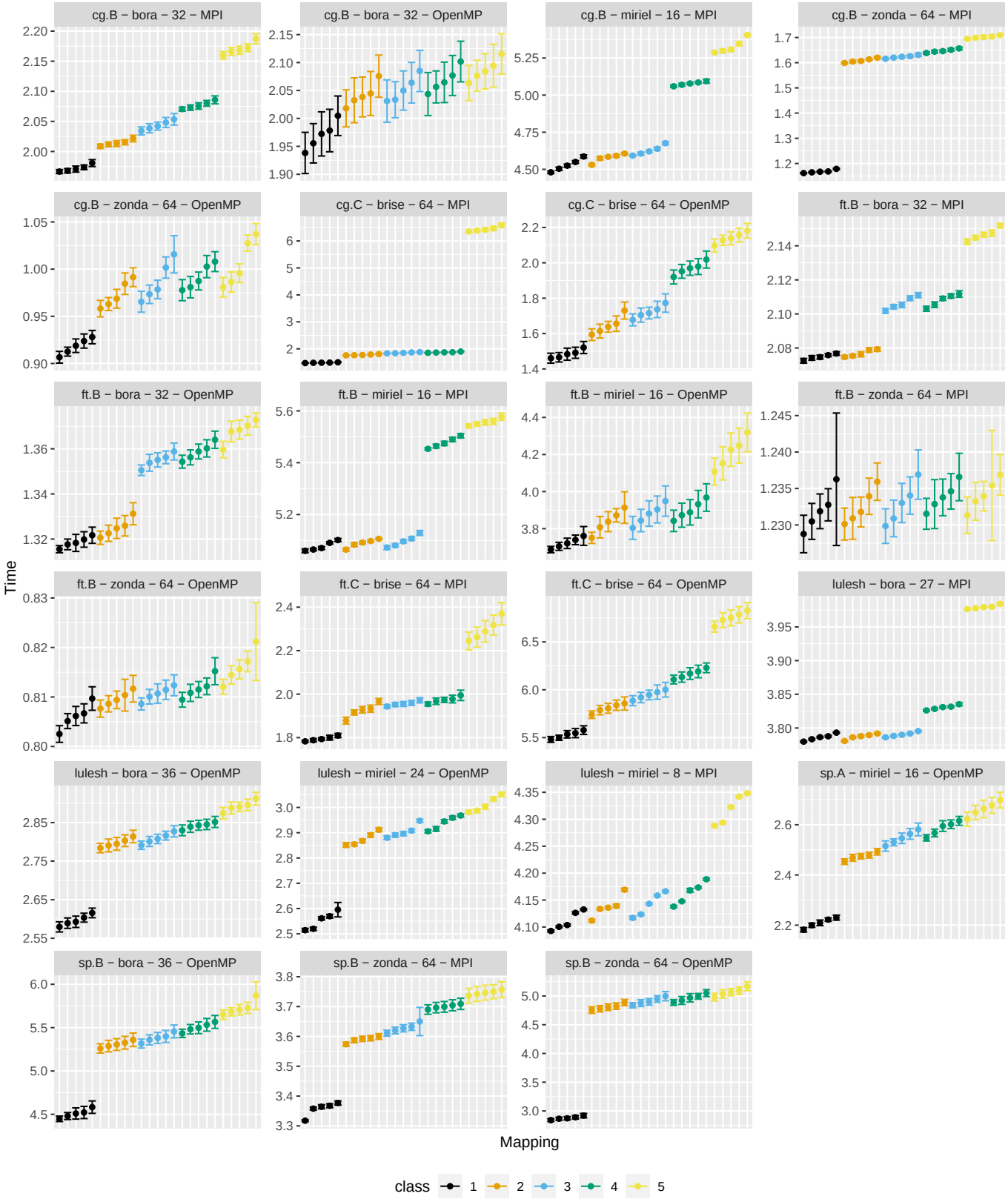


TABLE III: Topology description of testbed platforms.

Node	Processor	No. of Sockets	No. of L3 per Socket	No. of Cores per L3
brise	Xeon® E7-8890 v4	4	2	12
bora	Xeon® Gold 6240	2	1	18
zonda	AMD EPYC 7452	2	8	6
miriel	Xeon® E5-2680 v3	2	2	6

class average time to the highest class average time, and inside classes from the smallest placement average time to the highest placement average time. For the sake of clarity, we display only five classes and five placements per class, each uniformly spaced respectively across and inside the sorted classes. We list below our observations from Figure 3:

- One can visually identify cases supporting the property of similar mappings on hierarchical topologies. That is, cases where placements inside classes have similar execution time and overlapping confidence interval while some classes have significantly different overall execution time. For instance, *cg.C - brise - 64 - OpenMP* shows a clear distinction between classes and similar execution times inside classes.
- In many cases we can observe interclass and intraclass variations. Although intraclass error bars often overlap, one frequently sees at least two placements of the same class with a (statistically) significant difference. In this case, the difference between some classes is larger than the difference we observe inside classes.
- For some cases, all classes exhibit intraclass variations that are comparable to interclass variations. However, all these differences are small (i.e., a few percent). For instance, *ft.B - zonda - 64 - MPI* has a similar pattern across classes but also has less than 1% variation between its most extreme placements.

In Table IV we summarize for each test case the maximum speedup between pairs of placements from the same class (*Placement Speedup*) versus the maximum speedup between pairs of classes (*Class Speedup*), and compare both values to show that cross classes permutations carry a larger performance impact than inner-class permutations. The speedup is computed as the difference between the maximum and the minimum execution times, respectively among placements inside a class for Placement Speedup and among classes for Class Speedup, divided by the execution time. Execution time here means the average execution time, respectively of samples of a placement and of samples of all placements inside a class. We provide the maximum speedup of the class with the highest speedup for Placement Speedup. The table shows that the speedup inside classes never exceeds 9% and is always lower than the speedup between classes when the test case is sensitive to placement, that is when at least one speedup exceeds 2%. This result further confirms that differences inside a class are negligible while the class choice may have a significant impact on performance.

These preliminary observations establish that the proposed structure of the mapping space with mapping classes based on a hierarchical description of a machine topology is relevant. In all test cases that show significant variations across at least two mappings, even if we may observe smaller variations inside a same class, classes seem to delineate placement groups with a specific performance profile. This structure justifies a two-step exploration, that is, across classes first to find the best class and then eventually inside the best class to refine the result and find a near-optimal mapping.

TABLE IV: Comparison of maximum speedup between placements of the same class or between classes. The speedup here is defined as the difference of the most extreme execution times divided by the smallest execution time. Execution times are average values for either a placement or a whole class.

Test Case	Placement Speedup	Class Speedup
ft.B - zonda - 64 - MPI	0.01	0.00
ft.B - zonda - 64 - OpenMP	0.02	0.01
ft.B - bora - 32 - MPI	0.01	0.03
ft.B - bora - 32 - OpenMP	0.01	0.04
lulesh - miriel - 8 - MPI	0.01	0.05
lulesh - bora - 27 - MPI	0.00	0.05
cg.B - bora - 32 - OpenMP	0.04	0.06
ft.B - miriel - 16 - MPI	0.01	0.09
cg.B - zonda - 64 - OpenMP	0.06	0.10
cg.B - bora - 32 - MPI	0.01	0.10
lulesh - bora - 36 - OpenMP	0.02	0.11
sp.B - zonda - 64 - MPI	0.06	0.11
ft.B - miriel - 16 - OpenMP	0.07	0.13
cg.B - miriel - 16 - MPI	0.03	0.18
lulesh - miriel - 24 - OpenMP	0.03	0.18
sp.A - miriel - 16 - OpenMP	0.04	0.21
ft.C - brise - 64 - OpenMP	0.03	0.22
sp.B - bora - 36 - OpenMP	0.04	0.27
ft.C - brise - 64 - MPI	0.06	0.28
cg.C - brise - 64 - OpenMP	0.09	0.44
cg.B - zonda - 64 - MPI	0.02	0.46
sp.B - zonda - 64 - OpenMP	0.05	0.76
cg.C - brise - 64 - MPI	0.05	3.33

C. Statistical Assessment of Mappings and Classes Similarities

In this section we define an analytic methodology to label test cases as to whether they invalidate Section II property or not. To this end, we consider two scenarios labeled as follows:

- cross-class different: If at least two mapping classes are different, then the best mapping search should take into account classes. Therefore, we consider that test cases with this label do not invalidate our hypothesis. As a shorthand, we will say that it "validates" our hypothesis, even though one can only try to invalidate it.
- intra-class similar: If all placements inside a class are similar and this is true for all classes, then our hypothesis is also validated. Note that if all classes are also similar, then the application is not sensitive to the mapping choice.

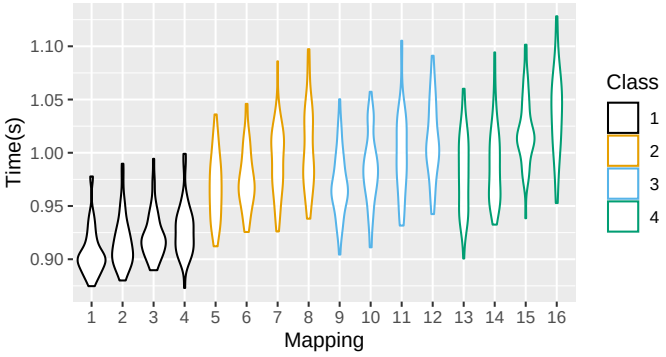
With these labels, the only configuration invalidating our hypothesis would be a test case for which placements inside a same class are different while classes are similar; in other words, classes do not bring structure to the exploration space. We define below what it means for two placements or two classes to be similar or different.

The Kolmogorov–Smirnov test [36], [37] is a statistical test taking two sets of samples as input and computing a distance between the observed cumulative distributions of the samples. This test does not require the sample sets to have a uniform distribution. The distance takes values between 0 (equal distributions) and 1 (totally different distributions). We use the Kolmogorov–Smirnov (KS) test to quantify the distance between two distributions and find a distance threshold that would accurately tag test cases as *cross-class different* or *intraclass similar* according to what we observed earlier. We define the value *threshold* such that a pair of distributions is tagged as similar or different depending on whether the (KS) test value (or distance) is respectively below or above this threshold. In Figure 4, we show an example of placement distributions in a violin plot (Figure 4a) and the matching (KS) test value for each pair of placements (Figure 4b). This figure helps gauge the range of values to expect from the test for similar placement inside the same class (above the diagonal in Figure 4b) versus placements from different classes (below the diagonal). In this example, the test value never exceeds 0.56 when placements are from the same class whereas the most different pair gets a (KS) test value of 0.96 across most extreme classes.

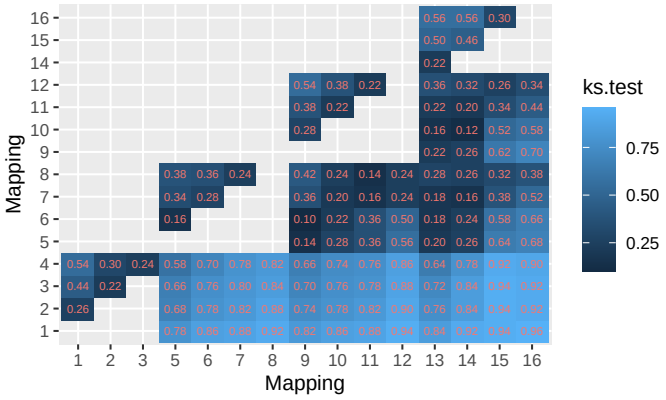
We computed the distance between every pair of placements belonging to the same class. In the bottom facet of Figure 5, we show the percentage of placement pairs with a distance below the threshold (y-axis) for a range of thresholds between 0 and 0.99 (x-axis). Each line of the plot represents a different test case. The bottom line test case, *lulesh - miriel - 8 - MPI*, has many placements with different distributions even though they are from the same class. However, we can also see even more significant differences between best and worst classes in Figure 2. In the upper facet of Figure 5 we compare the distributions of classes and report the percentage of class pairs for which the distance (y-axis) is less than the chosen threshold (x-axis). Classes exhibit longer plateaus than do mappings, characterizing groups of classes with close but different performances. This result can typically be observed for the test case *ft.C - brise - 64 - OpenMP* in Figure 2. Only few largely different groups of class with such a difference may exist. Therefore, we use a very high threshold value, 0.99, to tag test cases as *cross-class different*. As a result, if not all the pairs of class are tagged as similar, then there is at least one pair of classes with a significant difference in their average execution time. Using this methodology, we are able to identify sixteen test cases that we can tag as cross-class different. The other test cases also validate our hypothesis if they are labeled as *intraclass similar*. The latter depends on the choice of the threshold value. We report the classification of test cases for *cross-class different* labels and the percentage

Fig. 4: Comparison of placement distributions for *cg.B - zonda - 64 - OpenMP*.

(a) Placements Distribution



(b) Placement Pairs (KS) value



of *intraclass similar* mappings for threshold values of 0.50 and 0.90 in Table V. In this table we observe, for instance, that *cg.B - zonda - 64 - MPI* is tagged as *cross-class different*, which reflects accurately what can be observed on Figure 2, and validates our hypothesis. The test case *ft.B - zonda - 64 - MPI* seem also to be correctly classified as having similar classes but also similar intraclass placements distribution with a 0.90 and even 0.50 thresholds making all pairs similar. We can see in Figure 2 that this test case is not really sensitive to mappings, a result that is consistent with its associated labels, that is, neither different across classes nor different inside classes.

At the top of the table we find the most "troublesome" cases. *cg.B - zonda - 64 - OpenMP* is a case with no dramatically different pair of classes and some noticeable differences inside classes. However, only a small proportion of the placements can be considered as such. Moreover, Table IV indicates that this test case has a low sensitivity to placement across classes and is even less sensitive when looking only inside classes. This last case shows the limit of the methodology when cases are on the fence for the cross-class different label and intraclass similar label with a low threshold. In our experiments, however, test cases have a low sensitivity to placement. Overall, our methodology accurately detects test

Fig. 5: Percentage of similar class/mapping pairs for varying threshold levels when using the Kolmogorov–Smirnov test value.

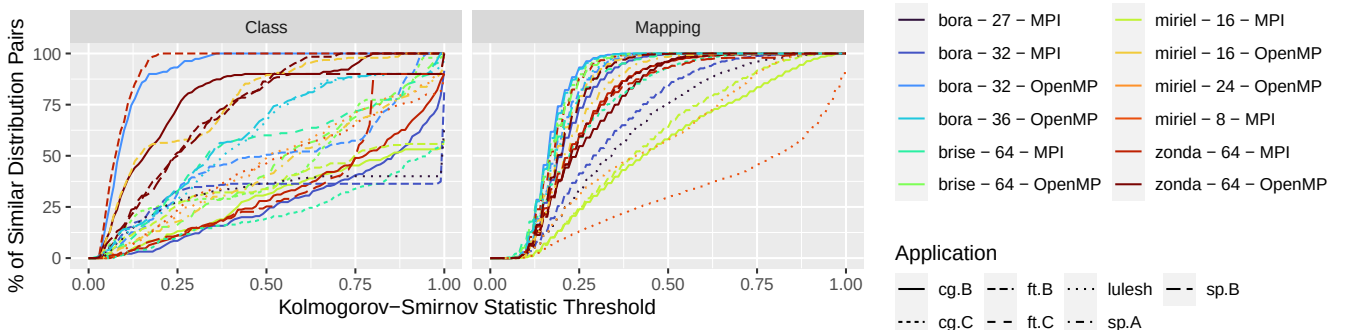


TABLE V: Classification of test cases for *cross-class different* and *intraclass similar* labels. The cross-class column states whether a test case is labeled as *cross-class different* or not. Columns 0.50 and 0.9 respectively give the percentage of intraclass placement pairs with a distance below 0.50 and 0.9 thresholds for each test case.

Test Case	Cross-Class $\neq$	0.50	0.90
cg.B - zonda - 64 - OpenMP	No	95.89	100.00
ft.B - zonda - 64 - OpenMP	No	96.08	99.68
ft.B - miriel - 16 - OpenMP	No	99.84	100.00
ft.B - bora - 32 - OpenMP	No	99.89	100.00
cg.B - bora - 32 - OpenMP	No	100.00	100.00
ft.B - zonda - 64 - MPI	No	100.00	100.00
sp.B - bora - 36 - OpenMP	No	100.00	100.00
lulesh - miriel - 8 - MPI	Yes	29.13	61.55
lulesh - miriel - 24 - OpenMP	Yes	57.32	99.79
cg.B - miriel - 16 - MPI	Yes	58.03	96.29
ft.B - miriel - 16 - MPI	Yes	64.00	99.16
lulesh - bora - 27 - MPI	Yes	75.05	99.45
ft.B - bora - 32 - MPI	Yes	83.89	99.82
cg.C - brise - 64 - MPI	Yes	92.52	100.00
sp.B - zonda - 64 - MPI	Yes	92.95	99.95
cg.B - zonda - 64 - MPI	Yes	95.71	100.00
sp.A - miriel - 16 - OpenMP	Yes	98.32	100.00
ft.C - brise - 64 - MPI	Yes	99.45	100.00
cg.B - bora - 32 - MPI	Yes	99.66	100.00
sp.B - zonda - 64 - OpenMP	Yes	99.79	100.00
cg.C - brise - 64 - OpenMP	Yes	100.00	100.00
ft.C - brise - 64 - OpenMP	Yes	100.00	100.00
lulesh - bora - 36 - OpenMP	Yes	100.00	100.00

cases, validating our property and edge cases.

V. Conclusion

Next-generation HPC systems are poised to exhibit greater NUMA effects as well as a larger and a deeper memory hierarchy. Therefore, mapping of threads and processes in-die, nodewide, and across nodes of distributed-computing systems is an on-going matter because it improves locality. Some approaches tackle the problem using heuristics—for instance, MPI communications volume, cache misses, remote memory access—as input for a placement strategy. These approaches may use topology structural information, eventually annotated with (measured) performance attributes. Few approaches combine it with an exploratory-based approach to better cover the overwhelmingly large search space.

This paper is complementary to these approaches. It formalized the structure of the mapping space on hierarchical topologies and breaks down steps to reach closer to the best and worst mapping solutions via exploration-based approaches. We defined the concept of mapping classes based on the hierarchical structure of machine topology. Mapping classes are classes of performance and can be used for a coarse exploration of the mappings space, with further refinement of the solutions inside a class. We implemented and published online the methodology to identify classes and to generate them. To the best of our knowledge, such a description of the mapping space and exploration strategy did not exist before. Throughout our experiments, we showed cases where mapping can impact application execution time by more than 50%. With four types of processors and as many distinct topologies, for OpenMP and MPI applications, and more than 424 node-hours of run, we were able to exhibit in nearly all cases the existence of such classes. We also presented an analytic methodology to measure differences between placements and performance classes and determine whether the property of similar mappings on hierarchical topologies defined in this paper holds true. Furthermore, this methodology can be used to validate the paper assumptions on a different testbed.

Future work includes the following. First, we want to extend the results of this paper to multinode systems where we expect to see an even larger impact of classes since moving processes across nodes significantly changes data paths in the machine. We also plan to study the impact of individual levels of the topology on the placement choice. If some levels have negligible impact on performance with respect to application placement, then they could be removed from the topology descriptions and further reduce the size of the space to explore. Our exploration method helps find near-upper and lower bounds of possible execution times. We would like to compare the solution with the state-of-the-art static mapping decisions. Furthermore, based on the structure of the mapping space, we want to design a mapping cost model correlating with the execution time. Inverting such a model would effectively provide a near-optimal mapping solution.

ACKNOWLEDGMENTS

This research is supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. Argonne's work is also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine.

REFERENCES

[1] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity."

[2] K. Lepak, G. Talbot, S. White, N. Beck, S. Naffziger *et al.*, "The next generation amd enterprise server product architecture," *IEEE hot chips*, 2017.

[3] A. Sodani, "Knights landing (knl): 2nd generation intel® xeon phi processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*.

[4] J. Hofmann, G. Hager, G. Wellein, and D. Fey, "An analysis of core- and chip-level architectural features in four generations of intel server processors," in *International Supercomputing Conference*, 2017.

[5] T. Hoefler, E. Jeannot, and G. Mercier, "An overview of process mapping techniques and algorithms in high-performance computing," *High Performance Computing on Complex Environments*, 2014.

[6] G. Mercier and J. Clet-Ortega, "Towards an efficient process placement policy for mpi applications in multicore environments," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[7] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss, "Evaluating thread placement based on memory access patterns for multi-core processors," in *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*.

[8] S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with xeon phi KNL," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*.

[9] M. Diener, E. H. Cruz, and P. O. Navaux, "Locality vs. balance: Exploring data mapping policies on numa systems," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.

[10] E. H. Cruz, M. Diener, M. A. Alves, and P. O. Navaux, "Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols," *Journal of Parallel and Distributed Computing*, 2014.

[11] E. H. Cruz, M. Diener, L. L. Pilla, and P. O. Navaux, "Eagermap: a task mapping algorithm to improve communication and load balancing in clusters of multicore systems," *ACM Transactions on Parallel Computing (TOPC)*, 2019.

[12] J. Meng, J. W. Sheaffer, and K. Skadron, "Exploiting inter-thread temporal locality for chip multithreading," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*.

[13] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS XV*, 2010.

[14] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," in *Proceedings of the international symposium on Memory management*, 2011.

[15] N. Denoyelle, B. Goglin, A. Ilic, E. Jeannot, and L. Sousa, "Modeling non-uniform memory access on large compute nodes with the cache-aware roofline model," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[16] L. Li, M. Fussenegger, and G. Cichon, "A data locality and memory contention analysis method in embedded numa multi-core systems," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*.

[17] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-scale Parallel Architectures," in *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*.

[18] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *ACM Sigplan notices*, 2009.

[19] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in numa architectures: A statistical learning approach," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019.

[20] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout, "A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms," in *Design, Automation and Test in Europe*, 2005.

[21] E. Jeannot and G. Mercier, "Near-optimal placement of mpi processes on hierarchical numa architectures," in *Europar*, 2010.

[22] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P.-A. Wacrenier, "An Efficient OpenMP Runtime System for Hierarchical Architectures," in *International Workshop on OpenMP (IWOMP)*, 2007.

[23] J. Gustedt, E. Jeannot, and F. Mansouri, "Automatic, abstracted and portable topology-aware thread placement," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[24] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß, "Communication-aware process and thread mapping using online communication detection," *Parallel Computing*, 2015.

[25] G. P. Berned, T. S. Medeiros, M. Serpa, F. D. Rossi, M. C. Luizelli, P. O. Navaux, A. C. S. Beck, and A. F. Lorenzon, "Combining thread throttling and mapping to optimize the edp of parallel applications," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*.

[26] J. Schwarzrock, H. M. G. de A. Rocha, A. C. S. Beck, and A. F. Lorenzon, "Effective exploration of thread throttling and thread/page mapping on numa systems," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*.

[27] M. Popov, A. Jimborean, and D. Black-Schaffer, "Efficient thread/page/parallelism autotuning for numa systems," in *Proceedings of the ACM International Conference on Supercomputing*, 2019.

[28] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, 1985.

[29] L. Lundberg, "Evaluating the performance implications of binding threads to processors," in *Proceedings Fourth International Conference on High-Performance Computing*, 1997.

[30] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[31] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a generic framework for managing hardware affinities in hpc applications," 2010.

[32] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on {NUMA} systems: Asymmetry matters," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015.

[33] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*.

[34] "Permutation group," 2020, *Checked on: 01/08/2020*.

[35] S. S. SHAPIRO and M. B. WILK, "An analysis of variance test for normality (complete samples)†," *Biometrika*, 1965.

[36] F. J. M. Jr., "The kolmogorov-smirnov test for goodness of fit," *Journal of the American Statistical Association*, 1951.

[37] "R man page for kolmogorov-smirnov test," 2020.