Netloc: a Tool for Topology-Aware Process Mapping

Cyril Bordage Clément Foyer
Brice Goglin

Inria Bordeaux – Sud-Ouest – LaBRI – Univ. Bordeaux – France
firstname.lastname@inria.fr

October 11, 2017

Abstract

Interconnection networks in parallel platforms can be made of thousands of nodes and hundreds of switches. The communication cost between tasks of a parallel application varies significantly with their actual location in such platforms. Topology-aware process mapping consists in matching the application communication pattern with the network topology to improve the communication cost by placing related tasks close on the hardware.

We show that our Netloc tool for gathering network topology in a generic way can be combined with the state-of-the-art Scotch partitioner for computing topology-aware MPI process placement. Our experiments with a stencil application on a fat-tree machine show that we are able to significantly improve the runtime in the vast majority of cases.

1 Introduction

Parallel platforms are increasingly complex. They feature deep memory hierarchies as well as multi-level interconnection networks that cause locality to severely impact application performance. Topology-aware process placement is a widely used optimization technique for improving the overall application time by reducing communication overheads. It requires knowledge of the hardware organization, of the application needs, and algorithmics to make them match.

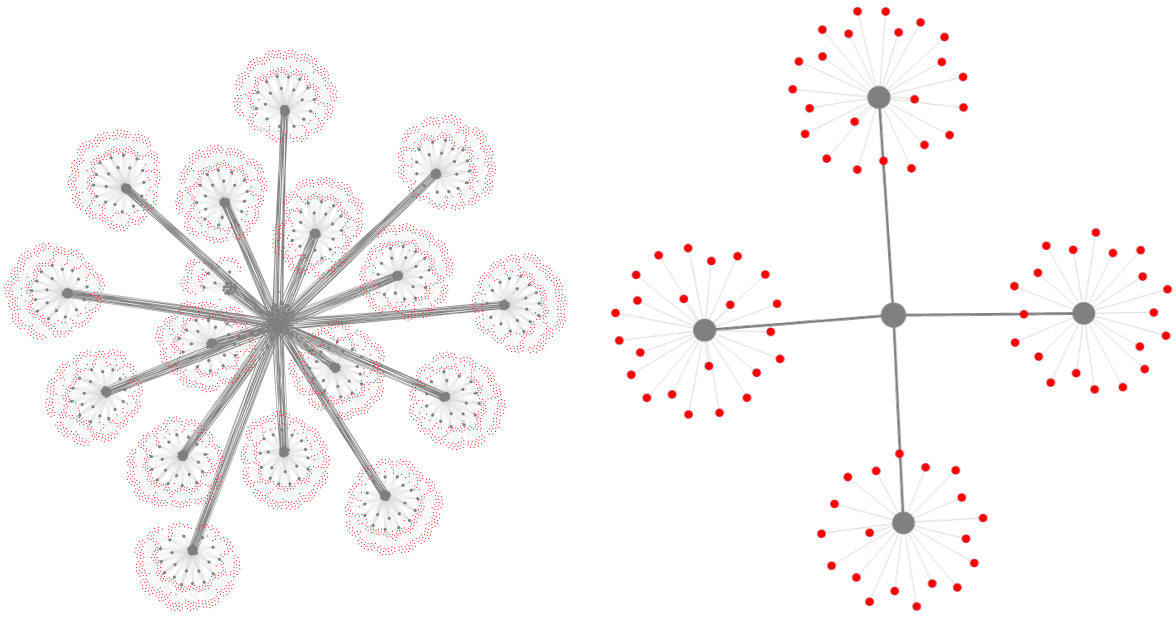
The internals of nodes are nowadays well modeled thanks to software tools such as hwloc [5]. However, there is still a need for a generic way to model networks and map processes to different compute nodes. We present in this paper the implementation of topology-aware process placement using the Netloc tool and the Scotch graph partitioner.

2 Generic Network Topology Discovery

On the road to exascale, parallel platforms are growing. Computing nodes now contain tens of cores. There is also an increasing number of nodes interconnected with various network technologies and fabric topologies, such as InfiniBand or Intel Omni-Path fat-trees, or Cray Aries dragonfly. Understanding the organization of these networks is required for administration, debugging, and performance optimization.

The topology inside nodes is well understood problem. It was solved by using software such as hwloc [5] which builds a hierarchical model of each host by organizing processors, cores, caches, NUMA nodes and hardware threads as a tree. Although this strategy is only a structural model of the hardware without performance information such as physical distances between components, it still provides valuable information for describing hardware affinities and easing locality-aware task placement [8, 9, 11].

On the network side, each technology has its own custom command-line administration tools (`ibnetdiscover`, `opareport`, `xtprocadmin`, *etc.*) with different outputs or even different notions of nodes, ports and links.



(a) InfiniBand fabric with 6000 nodes.

(b) Plafrim 2 cluster with 88 nodes.

Figure 1: Network topologies of two machines. Nodes in red and switches in grey.

Some technology-specific tools have been proposed [12] for exposing the network topology in a convenient manner. However, they cannot be extended to other technologies because of these different management tools, query APIs, and fabric topologies.

The Netloc project was designed to manage network topologies in a generic manner [6]. It uses hardware-specific tools to discover the entire fabric topology (nodes, switches, and links) and exposes it to HPC applications and runtimes through an abstracted API. This enables visualization of the topology as depicted on Fig. 1. However, Netloc was mostly developed for improving the performance of applications by taking the network locality into account in HPC runtimes. Possible optimizations include selecting the best route between peers or building neighborhoods for mapping hierarchical algorithms to the hierarchy of nodes and switches. We discuss in the next sections the use of Netloc for topology-aware process placement.

3 Topology-aware Mapping in Fat-Trees

In an MPI application, the communication cost between two ranks depends on where the processes are mapped on the machine. If a task sends a message to a task on the same processor, it will be faster than if they are on two different nodes connected to different switches. Hence, it is important to consider the topology of the machine when placing tasks on a parallel platform. In order to reduce the communication time, it seems obvious to map processes on close cores if they communicate a lot. Some applications may also have been optimized for other criteria (e.g. placing specific tasks near GPUs or I/Os), which can fortunately usually be combined with our policy.

The vast majority of message-passing applications can potentially benefit from communication-aware task placement. This has been the subject of several research projects [7, 8, 12]. Nevertheless, none of these approaches provide state-of-the-art network topology discovery tools that can be used on a variety of hardware, neither state-of-the-art graph partitioning techniques for computing a good mapping. We propose in this paper to combine Netloc and Scotch to address this problem.

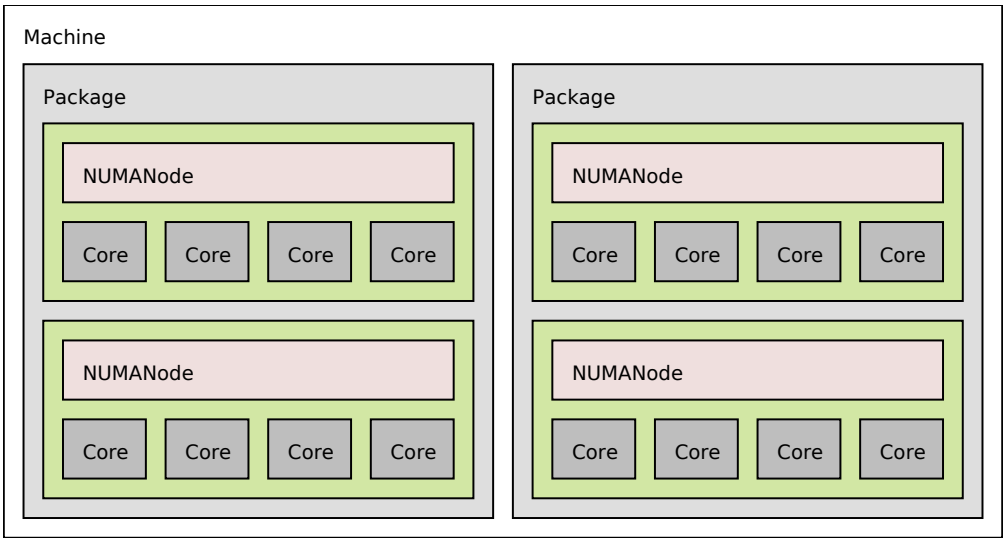


Figure 2: Example of the topology of a node, given by hwloc.

3.1 Scotch

Scotch [10] is a widely-used library for graph partitioning and mapping. It can operate on very large graphs (up to billions of vertices) using thousands of cores in parallel.

In our work, we use Scotch *Graphs* to model applications (communication patterns) and Scotch *Architecture Graphs* to represent platforms. Scotch can deal with random architectures but works best with regular structures such as fat-trees, torus, meshes, hypercubes, etc. Scotch is also able to deal with partial architectures, i.e. a restriction of a regular architecture, for instance to represent the subset of cluster nodes that were actually allocated by the resource manager.

3.2 Using Scotch from Netloc

The whole topology of a machine is given by the network topology but also by the node topology. The latter is generally a tree as depicted in the example in Fig 2.

Consequently, the whole topology of a platform interconnected through a fat-tree fabric is a tree as shown in Fig. 3. Thus, we only need to give the right tree to Scotch.

The architecture graph needed by Scotch to compute the mapping is then created from Netloc by exporting the network topology as well as the node topology provided by hwloc into the Scotch format. Moreover, Netloc is able to find the currently allocated resources and export them into a Scotch partial architecture.

We have integrated a tool in Netloc that builds a Scotch architecture representing the topology of the machine and uses Scotch to generate a good process mapping. Then, it converts the mapping returned by Scotch into a rank file for MPI. Thus, if a user provides a communication matrix between the MPI ranks, we can give him a rank file that will be used to launch the MPI application in a topology-aware-optimized way. Instead of manually building a communication matrix, the user can generate it using existing monitoring tools [4].

4 Evaluation

To see the relevance of our topology-aware mapping, we have conducted tests and compared our mapping to the default mapping of MPI that is round-robin.

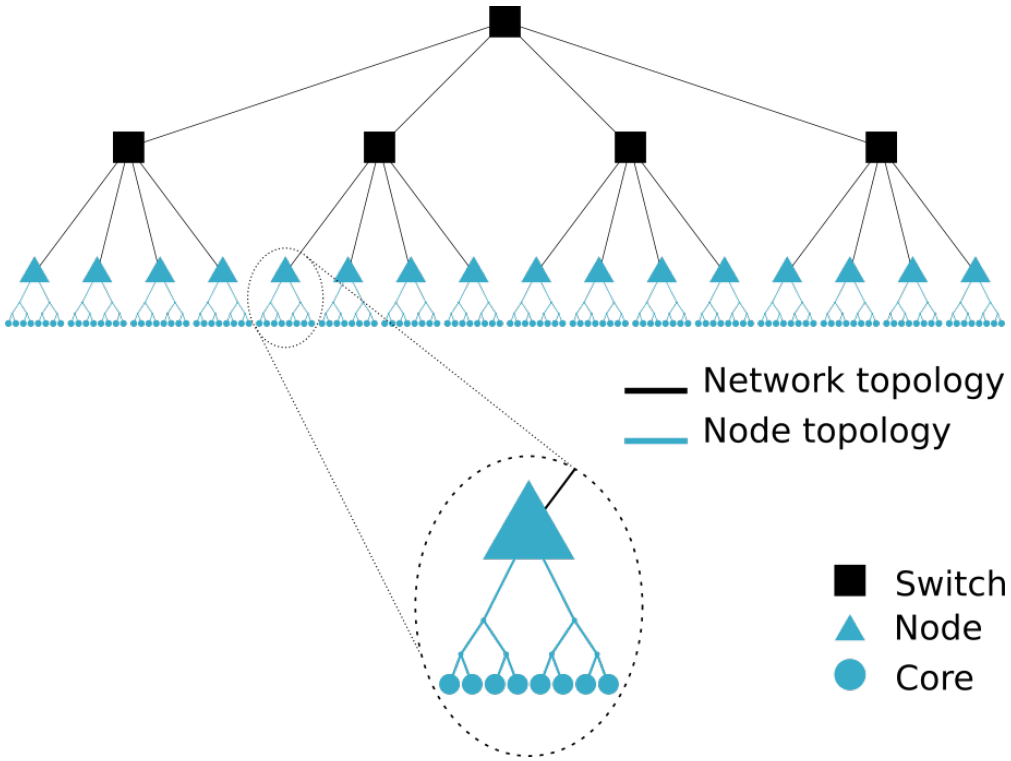


Figure 3: Example of a fat-tree with node topologies nodes.

4.1 Setup

The experiments were carried out using Plafrim 2, an 88 node machine with a fat-tree network, as shown in Fig. 1b. It is an InfiniBand QDR fat-tree network made of four leaf switches with 22 nodes each. There are 2 spine switches although they were combined into a single virtual switch on the figure. Each node contains two Intel Xeon E5-2680 v3 processor (24 cores total, split in 4 NUMA nodes with 6 cores each).

The tested application was miniGhost [3], a miniapp doing stencil computations. It allows to set a lot of different parameters that will change the communication pattern.

We use Open MPI with monitoring [4] to build our communication matrices. Each value in the matrices corresponds to the volume of communication between a pair of ranks during the overall execution.

4.2 Results

We have run miniGhost with more than 300 parameter sets to see the influence of our mapping in different configurations. We have changed grid sizes, stencil patterns, schemes of communication, number of variables, *etc.* The gain is measured when using the mapping given by Netloc with Scotch compared to the default mapping of the MPI implementation. To summarize the gains and make that more readable, we have computed the percentage of test cases that have a gain at least greater than a specific value. These results are presented in Fig. 4.

In 92.5% of the test cases, our mapping is at least as good as round-robin mapping. It means that we lose performance in only 7.5% of the cases. The largest loss is 3.6%. It can be due, for instance, to cache pollution from more shared memory communication. Our mapping shows a gain at least equal to 10.1% for 50% of the tested cases, at least 20% for 24.9% of the cases, and at least 30% for 5.7% of cases. The maximum gain is 33.6%.

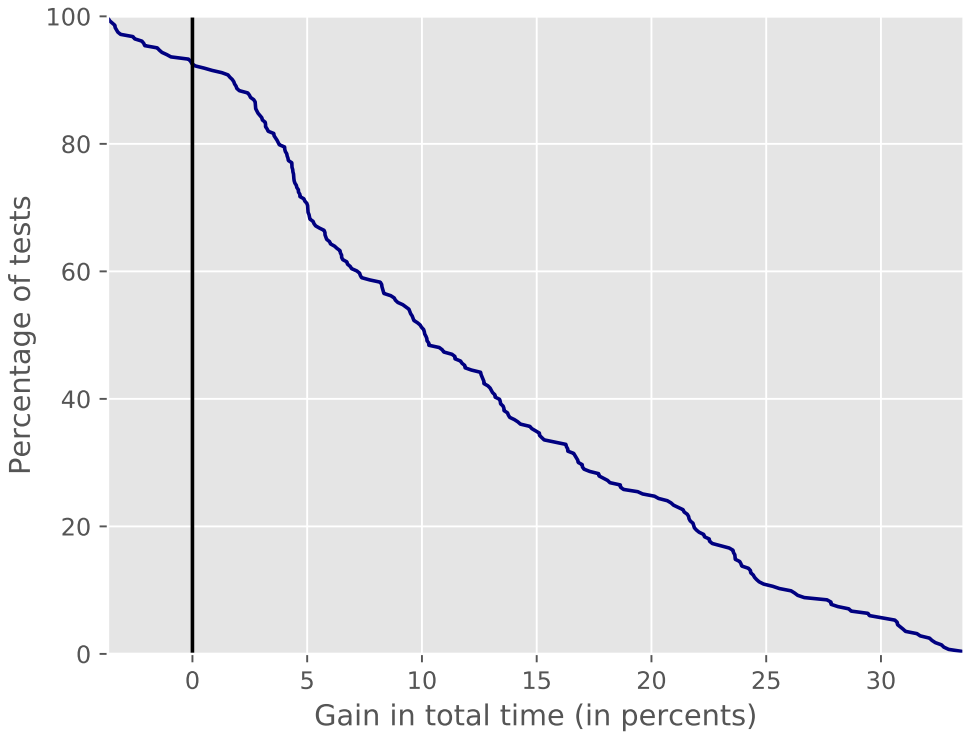


Figure 4: Percentages of test cases with gain greater than x-value.

5 Mapping on complex Topologies

As we use Scotch to do the mapping, the architecture of the machine must be a topology handled by Scotch. Scotch can handle a lot of different structures, as shown in Section 3.1. Nevertheless, the whole topology must be a single topology and not an aggregation of different kinds. However, as we saw in Fig. 3, the whole topology is composed of the network topology and the node topologies. Since the node topology is generally a tree, to have a topology handled by Scotch, it imposes to the network topology to be a fat-tree.

To tackle this problem, we propose a method to handle architectures composed of different topologies.

5.1 Expression of the topology

The mapping is done on the cores allocated by the resource manager. Therefore, to build the mapping, we need to get only the part of the topology with the current resources. Thus, our description of the topology will contain the lists of available nodes. For that we chose a hierarchical format, with one line for one level of the global topology:

```
<type> <number of dimensions> <dimension1 size> <dimension1 speed>
... <dimensionD size> <dimensionD speed> <number of nodes> <node1
index>...<nodeN index>
```

The field called `type` designates the type of topology. For now, it can only be `tree` or `torus`. With that we can handle fat-trees and Cray XE [2]. However, we will extend it with composition of simple topologies such as `alltoall` or `ring` to handle more topologies like, for instance, Dragonfly with Cray XC [1]. The node indices represent the available elements in the topology. The index is a way to identify the position of the element in the topology. Moreover, if the line describes a compute node, we prefix it with:

`node <hostname>`

To make it clearer, we consider an example with a 3d torus of size 4 for each dimension. On each point of the torus, we have a router that connects 6 other routers (two neighbors in each dimension) and 4 local nodes together. This topology is quite similar to Cray XE but with 4 nodes in a router instead of 2. A node contains two processors split in 2 NUMA nodes with 4 cores each.

Assuming the resource manager gives us 3 nodes, two on the same router and the third on another one, this network topology is represented in Fig 5.

This topology will be described by the following lines:

```
torus 3 4 1600 4 1600 4 1600 2 32 52
tree 1 4 800 1 0
node n129 tree 4 2 80 2 40 4 20 2 10 -1
tree 1 4 800 2 0 1
node n208 tree 4 2 80 2 40 4 20 2 10 -1
node n210 tree 4 2 80 2 40 4 20 2 10 -1
```

The first line (`torus`) indicates the 3D torus with 4 nodes in each dimension. 32 and 52 indicate the indices of the two used routers in our allocation.

Each line starting with `tree` describes what is connected to each coordinate of the torus. Since we have a router with 4 nodes, it is modeled as a tree with 4 leaves. For the first occurrence of the `tree` line we have `1 0` to indicate that we have a single node on this router at the index 0. For the other occurrence, `2 0 1` shows that the router has two nodes of indices 0 and 1.

The lines beginning with `node` give us the hostname of the nodes followed by their structures that are trees with 4 levels: 2 processors, with 2 NUMA nodes each, 4 cores each. The value -1 is to avoid putting the complete list of the cores, it simply means that the node is fully available.

The values 10, 20, 40, 80, 800 and 1600 represent the costs of communication of their corresponding links.

5.2 Handling Hierarchical Topology

There are two ways to process complex topologies. The first strategy that we propose is to compute the mapping recursively. We do not create the general graph of the platform but rather keep the information that it is composed of a torus of trees on our example. First, we compute a mapping for the top topology. Usually this is network fabric. On our example, this step consists in mapping processes to routers on the torus (torus partitioning). Then, we recursively compute the mapping inside sub-topologies. In our example, this consists in considering all ranks selected for a router actually mapping them to individual cores on nodes connected to that router (tree partitioning). This approach has the advantage of using optimized partitioning technique for each step since both torus and trees are regular and well-known Scotch architectures.

Unfortunately, it is not possible in Scotch to put weight on nodes on the predefined architectures and it can lead to unbalanced mapping. In our torus example, if you use this technique without any weight on the routers, both routers will have the same amount of processes when the one with only one node needs one third of the processes, not half.

The other strategy is not concerned with this issue. The principle is to describe the entire topology, including switches, torus, routers, nodes and cores as a single graph. Scotch can handle such graphs using the *deco* architecture but it is more complex to compute than for regular topologies such as fat-trees or torus. Nonetheless, the time of computation is very low, since the complexity depends only on the number of used cores, and the quality of the mapping is preserved.

6 Conclusion

We showed how to perform a topology-aware mapping using state-of-the-art topology discovery and graph partioning tools, Netloc with Scotch, and a communication matrix. Tests on a stencil application showed

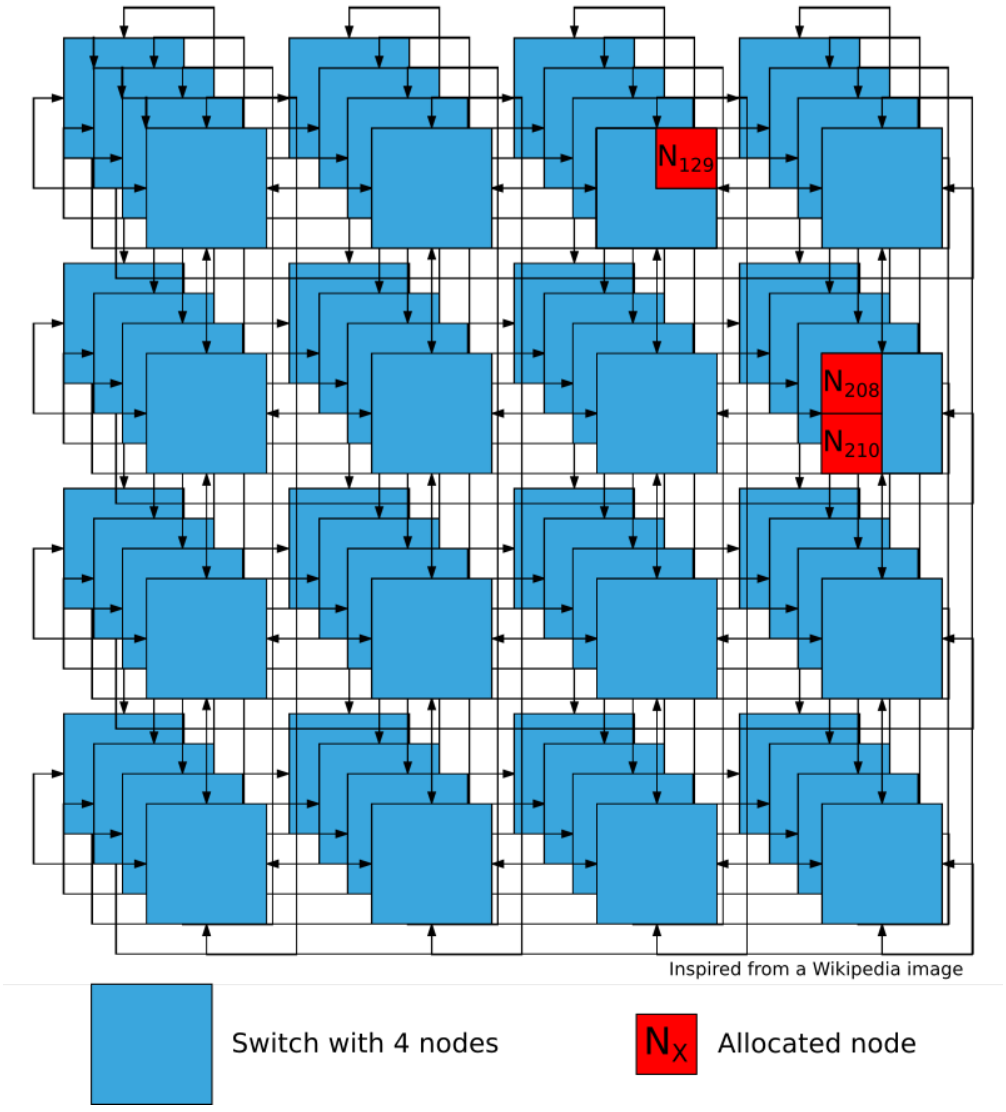


Figure 5: Example of a 3d torus machine with 3 allocated nodes.

the relevance of this approach on this kind of applications since we are able to significantly decrease the runtime in the vast majority of cases.

We are now running more tests with different applications and network technologies to further show the genericity and usefulness of our proposal. As we have proven the utility of our model, we are extending Netloc to handle more process placement algorithms. In the meantime, the user can get the discovered topology by exporting into a Scotch architecture and converting it into his desired format.

Our implementation is freely available in the current Netloc development code that can be found in the hwloc git repository at `https://github.com/open-mpi/hwloc`. It will be officially released in the upcoming hwloc 2.0 release in the next months.

Acknowledgements

Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS and ANR in accordance to the Programme d'Investissements d'Avenir (see `https://www.plafrim.fr/`).

This work is partially funded under the ITEA3 COLOC project #13024.

References

[1] Alverson, B., Froese, E., Kaplan, L., Roweth, D.: Cray xc series network. Cray Inc., White Paper WP-Aries01-1112 (2012)

[2] Alverson, R., Roweth, D., Kaplan, L.: The gemini system interconnect. In: 2010 18th IEEE Symposium on High Performance Interconnects. pp. 83–87 (Aug 2010)

[3] Barrett, R.F., Vaughan, C.T., Heroux, M.A.: Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Sandia National Laboratories, Tech. Rep. SAND 5294832 (2011)

[4] Bosilca, G., Foyer, C., Jeannot, E., Mercier, G., Papauré, G.: Online dynamic monitoring of mpi communications. In: Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 – September 1, Proceedings. pp. 49–62. Springer, extended version in `https://hal.inria.fr/hal-01485243`

[5] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010). pp. 180–186. IEEE Computer Society Press, Pisa, Italia (Feb 2010), `http://hal.inria.fr/inria-00429889`

[6] Goglin, B., Hursey, J., Squyres, J.M.: netloc: Towards a Comprehensive View of the HPC System Topology. In: Proceedings of the fifth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2014), held in conjunction with ICPP-2014. pp. 216–225. Minneapolis, MN (Sep 2014), `http://hal.inria.fr/hal-01010599`

[7] Hoefler, T., Snir, M.: Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In: Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11). pp. 75–85. ACM (Jun 2011)

[8] Jeannot, E., Mercier, G., Tessier, F.: Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. IEEE Transactions on Parallel and Distributed Systems 25(4), 993–1002 (4 2014)

[9] Li, S., Hoefler, T., Snir, M.: NUMA-aware Shared-memory Collective Communication for MPI. In: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing. pp. 85–96. HPDC '13, ACM (2013)

[10] Pellegrini, F., Roman, J.: Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: High-Performance Computing and Networking. pp. 493–498. Springer (1996)

[11] Solernou, A., Thiyagalingam, J., Duta, M.C., Trefethen, A.E.: The Effect of Topology-Aware Process and Thread Placement on Performance and Energy, pp. 357–371. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[12] Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., Panda, D.K.: Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes. In: Proceedings of the 2012 ACM/IEEE conference on Supercomputing. Salt Lake City, UT (Nov 2012)