HAL
open science

Towards the Structural Modeling of the Topology of next-generation heterogeneous cluster Nodes with hwloc

Brice Goglin

To cite this version:

Brice Goglin. Towards the Structural Modeling of the Topology of next-generation heterogeneous cluster Nodes with hwloc. [Research Report] Inria. 2016. hal-01400264v1

HAL Id: hal-01400264

https://inria.hal.science/hal-01400264v1

Submitted on 21 Nov 2016 (v1), last revised 10 Nov 2017 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards the Structural Modeling of the Topology of next-generation heterogeneous cluster Nodes with hwloc

Brice Goglin

Inria Bordeaux – Sud-Ouest, Univ. Bordeaux
F-33400 Talence, France
Corresponding author email: Brice.Goglin@inria.fr – Phone: +33 5 24 57 40 91 – Fax: +33 5 24 57 40 41

Abstract

Parallel computing platforms are increasingly complex, with multiple cores, shared caches, and NUMA memory interconnects, as well as asymmetric I/O access. Upcoming architectures will add a heterogeneous memory subsystem with non-volatile and/or high-bandwidth memory banks.

Parallel applications developers have to take locality into account before they can expect good efficiency on these platforms. Thus there is a strong need for a portable tool gathering and exposing this information. The Hardware Locality project (hwloc) offers a tree representation of the hardware based on the inclusion of CPU resources and localities of memory and I/O devices. It is already widely used for affinity-based task placement in high performance computing.

We present how hwloc represents parallel computing nodes, from the hierarchy of computing and memory resources to I/O device locality. It builds a structural model of the hardware to help application find the best resources fitting their needs. hwloc also annotates objects to ease identification of resources from different programming points of view. We finally describe how it helps process managers and batch schedulers to deal with the topology of multiple cluster nodes, by offering different compression techniques for better management of thousands of nodes.

Keywords: Multi-Core Architectures and Support; Modeling, Simulation and Evaluation Techniques of HPC Systems

1. Introduction

High performance computing relies on powerful computing nodes made of tens of cores and accelerators such as GPUs or Xeon Phi. The architecture of these servers is increasingly complex because these resources are interconnected by multiple levels of hierarchical shared caches and a NUMA memory interconnect. Execution performance now significantly depends on locality, i.e. where a task runs with respect to its data allocation in memory, or with respect to the other tasks it communicates with. It had a critical impact on the performance of parallel applications for a long time, from distributed computing [1] to single servers [2].

Performance optimization of parallel applications requires a thorough knowledge of the hardware, and many research projects aim to model the platform to tackle this challenge. Besides analytic performance models, one solution consists in structural modeling of the hardware resource organization. Indeed, parallel developers need such information to properly use the platform. hwloc (Hardware Locality) is the *de facto* standard software for representing CPU and memory resources, and for binding software tasks in a portable and abstracted manner [3].

We identify two major types of affinity. First, tasks have affinities for hardware resources they use. This includes memory banks, caches and TLBs that contain some of their data as well as I/O devices such as accelerators and network interfaces. Moving a task away from one core can cause the performance to vary depending on the cores' locality with regard to the I/O devices used by the task [4]. The second kind of affinity is between tasks. Indeed, parallel applications often involve communication, synchronization and/or sharing between some of the processes or threads. It usually means that related tasks should be placed on neighbor cores to optimize the communication/synchronization performance between them [5]. However, the affinity can also be reversed when single tasks have strong needs. For instance, memory-intensive applications may want to avoid sharing memory links or caches with others [6].

Applications can have several of these types of affinities simultaneously, even with conflicting needs. We envision two ways to deal with these needs. First, tasks can be placed on the hardware resources according to their affinities. For instance, MPI process placement based on the communication scheme and on the platform topology is a very active area of research [7, 8, 9]. Then, the actual communication between tasks can be adapted to the existing placement. For instance, the existence and the size of a shared cache between processes can be a reason to switch from one communication strategy to another [10, 11]. The locality of I/O devices can also be used to better tune collective operations [12, 13]. Moreover, batch schedulers or process managers try to manage clusters of such heterogeneous nodes in a global manner, making locality an important aspect, outside of nodes as well.

This paper describes how hwloc has evolved into the *de facto* central place for gathering locality information about all hardware subsystems in parallel platforms. Our contributions include a structural model based on physical locality and inclusion, as discussed in Section 2. This strategy is already successfully used by many high performance computing software for querying locality information. Then Section 3 describes how hwloc exposes a portable and abstracted view of the hardware by combining topology information from many sources, including operating systems, domain-specific libraries and platform-specific instructions. We finally present in Section 4 a contribution to the way batch schedulers and process managers apply cluster-wide allocation or placement policies. hwloc lets them manipulate, summarize, compress and/or compare the topologies of multiple nodes from the front-end.

2. Modeling the Structure of Computing Platforms

hwloc is now used by most MPI implementations, many batch schedulers and parallel libraries[1]. We explain in this section why hwloc's structural model is a good solution for their needs.

2.1. Modeling Platform for Performance Analysis

The complexity of modern computing platforms makes them increasingly harder to use, causing the gap between peak performance and application performance to widen. Understanding the platform behavior under different kinds of load is critical to performance optimization. Performance counters is a convenient way to retrieve information about bottlenecks for instance in the memory hierarchy [14] and apply feedback to better schedule the next runs [15]. The raw performance of a server may also be measured through different memory access workloads to predict the behavior of kernels [16]. However these strategies remain difficult given the number of parameters that are involved (memory/cache replacement policy, prefetching, bandwidth at each hierarchy level, etc.), many of them being poorly documented.

[1] A non-exhaustive list of hwloc users is available on the project webpage `http://www.open-mpi.org/projects/hwloc/`.

At the scale of a cluster, performance evaluation has been a research topic much earlier because network communication caused slowdowns long before servers became hierarchical (when multicore and NUMA processors emerged). The LogP model [17] may be used to describe the network performance and build a hierarchy of processors based on this experimental distance for better process placement [18]. Improved performance models have been proposed since then to offer realistic simulation on larger platforms [19]. These approaches may also be combined for inter-node and intra-node communication so as to weight the communication performance of all combinations of cores before scheduling jobs [20]. Such an approach may actually also help experimentally rebuilding the entire topology of the clusters for better task placement [21].

These results however lack a precise description of the structural model of the machine. Experimental measurement cannot ensure the reliable detection of the hierarchy of computing and memory resources such as packages, cores, shared caches and NUMA nodes. Indeed, they impact performance in a different way, and it may vary significantly with the workload (memory footprint vs cache size, number of processes involved vs memory bandwidth, etc.). It explains why performance models only give hints about the impact of the platform on performance. On the other hand, the structural modeling of the platform gives precise performance reports. OpenMP thread scheduling [22] or MPI process placement [9] are examples of scheduling opportunities that can benefit from deep platform topology knowledge.

2.2. Structural Modeling of the Hardware as a Tree

The organization of computing resources in clusters is hierarchical: a cluster contain servers that contain processor packages, that contain cores, that optionally contain several hardware threads. Current AMD Opteron processors (63xx) also have an intermediate level called *Compute Unit* between packages and cores. Moreover, from a locality point of view, memory resources such as caches and NUMA nodes can be considered as embedded into such computing resources: indeed NUMA nodes are usually attached to sets of cores[2], while caches are usually placed between some cores and the main memory, Therefore we can sensibly extend a hierarchy of computing resources to a hierarchy of memory resources as depicted in Figure 1.

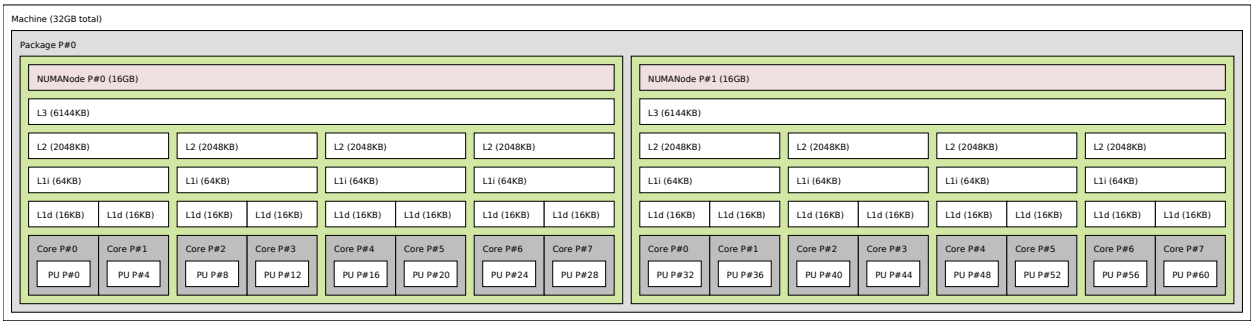


Figure 1: AMD platform containing Opteron 6272 processors, simplified to a single processor, and reported by hwloc's `lstopo` tool. This processor package is made of two parts containing one NUMA node and one L3 cache each. L2 and L1i caches are then shared by *Compute Units* pairs of cores, while the L1d is private to each single-thread core.

We represent the entire machine as a tree of resources organized by locality: the more the hardware threads share common ancestors (same NUMA node, shared caches, or even same core), the better locality between them is. This approach has several advantages:

[2]Old architectures such as Itanium even had each NUMA node attached to several processors.

- First the tree representation of the topology is very convenient because it exposes the natural inclusion-based organization of the platform resources. Indeed binding memory near a core or finding a shared cache between cores only requires to walk up the tree until we find the relevant ancestor object and/or walk down to iterate over children. We can easily iterate over cores and threads close to a given object in the tree, a very common operation in locality-aware runtimes.

  Another solution would be to represent the structure as the generic graph whose nodes are resources and edges are physical connections. However, the concepts of inclusion, container parent and contained children would not be obvious, making application queries less convenient.

- Secondly, this logical organization based on locality solves many portability issues by hiding platform specific parameters. Indeed vendor-specific configurations, BIOS or software upgrades may change the numbering of resources even if the processors are identical[3]. Therefore relying on hardware numbering of resources to perform explicit task placement makes programs non-portable, even to a similar platform with the same processors but a different BIOS (see Section 3.3).

  On the other hand, relying on hwloc's *logical* numbering of resources to perform explicit task placement makes programs more portable as it is based on the actual physical localities.

- Third, using a tree-structure is a good trade-off between performance and precision: while using a graph to represent the architecture would be more precise in some cases, tree algorithms are often much more efficient in both memory and time. Indeed, process placement techniques often rely on graph partitioning techniques (a communication graph must be mapped onto an architecture graph) which are much more efficient when generic graphs are replaced with trees. For instance, the Scotch partitioning software supports a *Tleaf* architecture definition for enabling specifically optimized algorithms on hierarchical platforms [23].

hwloc builds a *Tree* of *Objects*, from the root *Machine* (a single-image shared-memory system) to the leaves *Processing Units* (PU, that corresponds to hardware threads, logical processors, etc). It is based on the natural inclusive order of computing resources and the locality of memory resources. Each hwloc object is characterized by a type, some hardware characteristics such as a package number, and some optional parameters such as local cache or memory sizes. hwloc does not enforce the vertical ordering between these levels in the tree because some most modern processors may have two NUMA nodes per package (see Figure 1) while some Itanium machines have multiple packages per NUMA node. hwloc just moves larger objects above smaller ones depending on the architecture inclusion characteristics.

The hwloc model is further extended to include I/O devices by attaching I/O objects based on their locality. If a PCI bus is close to a NUMA node, the corresponding hwloc objects are attached as additional children to this NUMA node object, as depicted on Figure 2.

The hwloc programming interface allows walking the tree edges to find ancestors or child objects of a given type (e.g. when looking for the NUMA node close to a given core), iterate over objects of a same type (e.g. when binding processes on cores), etc. hwloc offers a convenient way to apply mapping policies [24] or partitioning algorithms by matching applications affinity graphs onto the hwloc tree of hardware resources [9]. More use cases and hwloc v1.0 early design details are presented in [3].

[3]For instance, Processing Unit (PU) #0 and #4 are close in Figure 1 while #1 is located in another processor (not displayed).

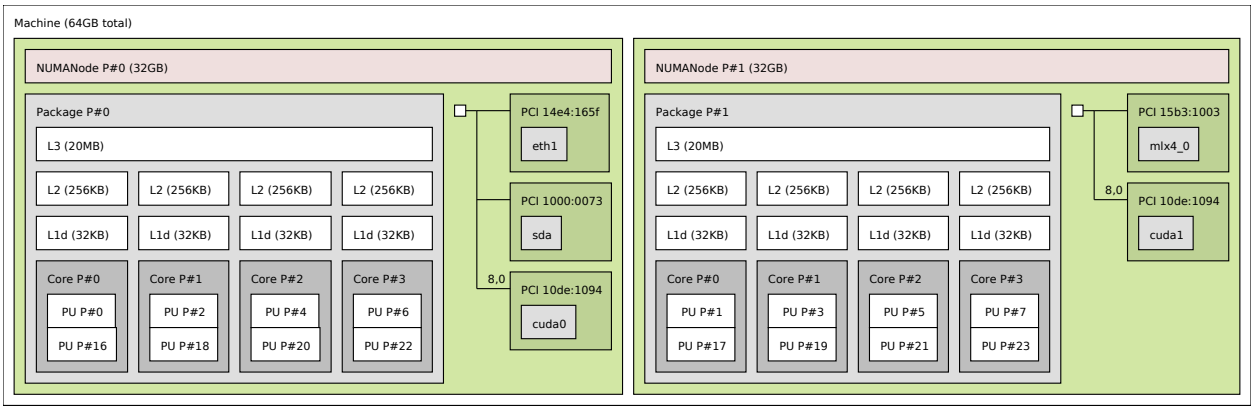


Figure 2: Topology of a dual-Xeon E5 host with GPUs (cuda0, cuda1), network (eth1), InfiniBand (mlx4_0) and disk (sda) connected to different packages, simplified and reported by hwloc's `lstopo` tool.

2.3. Limits to the Tree Model

We explained in the previous section why modeling the structure of a parallel platform as a tree is convenient, we now list several drawbacks of this approach.

One critique against the model is its lack of topology information within single levels of the tree. For instance, Xeon E5 and Xeon Phi processors are actually made of cores connected by an internal ring. hwloc only exposes them as an unordered list of cores, without any information about distances between specific cores. The upcoming Intel *Knights Landing* processor will even interconnect core tiles by a mesh [25]. Although this knowledge has been used to improve memory allocation in *Knights Corner* Xeon Phi [26], it remains hardly usable for applications in practice. Therefore, it is not clear yet if exposing such information would be of a lot of interest to hwloc users.

The same argument could apply to the NUMA memory interconnect since it is not always a complete graph. Indeed the performance of remote NUMA node memory access may vary with the physical distance[4]. This organization is available as a latency matrix (reported by the hardware ACPI tables, or provided by micro-benchmarks) and exposed by hwloc to annotate the set of NUMA node objects. Moreover, hwloc uses this information to create additional hierarchical *Group* objects that contain close NUMA nodes. Large NUMA platforms are therefore represented with a hierarchy of Groups between the *Machine* and NUMA node objects so that the physical organization as racks and blades is exposed[5].

Another issue with the tree model is upcoming architectures with different kinds of memory. For instance, the Intel *Knights Landing* processor will feature high-bandwidth memory (MCDRAM) inside the package while still using normal (DDR) external memory [25]. It means that each core may have two distinct local NUMA nodes, hence two parents: a normal one, and a faster but smaller one. Another constraint is that memory allocation should not go into the smaller and faster MCDRAM unless specifically requested. It means that the normal memory should appear closer to cores than the MCDRAM (even if the MCDRAM is physically closer). hwloc therefore currently exposes the MCDRAM as another NUMA node object on the side as shown on Figure 3. This work is part of a larger rework on the hwloc memory model. Indeed, upcoming non-volatile memory DIMMs will also be attached to processors and be byte-accessible. The

[4]On a 12-processor SGI Altix UV2000, the stream Triad benchmark bandwidth drops from 31GB/s locally, to 7.4GB/s when accessing memory from the neighbor NUMA nodes, and down to 5.3GB/s from the farthest NUMA node.

[5]Some examples of latency-based grouping are available at `https://www.open-mpi.org/projects/hwloc/lstopo/`.

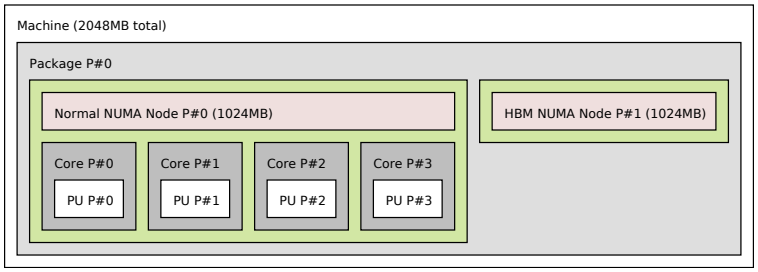


Figure 3: hwloc's view of a quad-core processor with 1GB of high-bandwidth memory (HBM) and 1GB of normal memory.

exact way these new memories will be exposed to the operating system is still under standardization. But we envision the idea of moving NUMA node objects to the side and rather attach them to processor like I/O devices.

Finally, considering the variety of network fabrics, the hwloc tree model does not work well for representing multiple machines and their interconnection network. This issue will be addressed specifically in Section 4.5.

3. Discovering and Organizing Hardware Information

We describe in this Section how hwloc discovers the existing computing, memory and I/O resources, as well as their locality before explaining how they are actually exposed within the tree in a abstracted and portable manner.

3.1. Where and How to Gather Topology Information

The importance of locality led many developers to retrieve topology information within their applications or libraries. Unfortunately, this work is difficult because of the amount and variety of the sources of locality information, ranging from operating system, to direct hardware query and high-level tools.

Linux is widely used in high performance computing. Unfortunately, its ability to report topology information was designed over more than ten years and therefore suffers from a partial and non-uniform interface. Many hardware details are available from the sysfs virtual file system (`/sys`) but it misses processor details (only available in `/proc/cpuinfo`) and I/O information such as network connectivity. Moreover, some of these files are in human-readable format, while some other pieces of information are split into many different machine-readable files. Extracting locality information from an application is therefore a lot of work.

Convenient topology discovery should be available in higher-level libraries that hide the difficulty of parsing low-level system files or architecture-specific registers. On Linux, numactl[6] possesses knowledge of NUMA, CPU and I/O localities but lacks caches. Moreover, its programming interface is unstable, and it was primarily designed for binding tasks: it cannot be used for querying details about hardware characteristics.

Some processors have dedicated instructions for retrieving topology information such as CPUID on x86. However, applications relying on this feature need to be updated for every new micro-architecture because special values with new meanings are often added and have to be supported. The operating system usually takes care of these cases, so these processor-specific instructions should not be needed in topology-aware applications, as long as the OS is recent enough.

[6] `http://oss.sgi.com/projects/libnuma/`

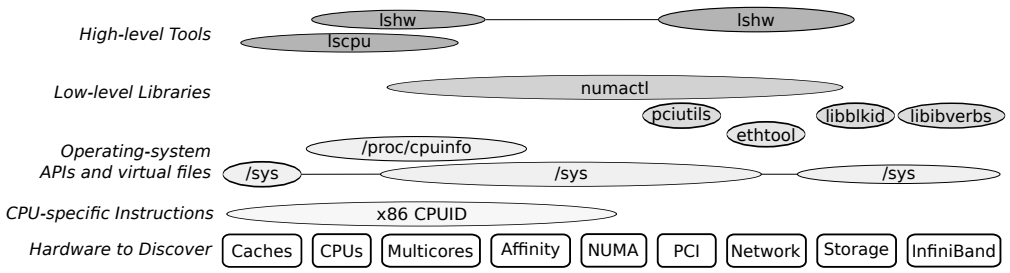


Figure 4: Overview of existing sources of locality information on Linux.

As shown on Figure 4, many libraries exist for querying the topology of specific subsystems, for instance pciutils for PCI[7], libibverbs for InfiniBand, CUDA for NVIDIA GPUs, etc. Unfortunately, there is almost no interoperability between these libraries and other topology-related tools. Therefore, we often have to combine information extracted from different sources, which can be a tricky process: for instance, it is necessary to query sysfs, pciutils and CUDA when looking for the locality of a NVIDIA GPU with regard to host CPUs.

Some higher level tools such as lscpu or lshw[8] merge the information from several sources but they were only designed for displaying the topology, without any programming interface for querying. In addition, some non-Linux operating systems may have better interfaces but they lack part of the information. For instance, Solaris does not report cache information, but the CPUID instruction may detect it on x86 platforms.

In conclusion, all these sources of information still have to be used concurrently for a developer to gather locality information about all hardware resources. There is therefore a need for a portable, system-wide topology discovery tool that combines all these pieces and exposes the information in a convenient and portable programming API. hwloc has been designed from the start to be this needed portability layer.

3.2. Combining Multiple Sources

hwloc's discovery mechanism is based on several *components*[9], each gathering information from some specific sources (operating system, hardware, low-level libraries, etc.), as described on Figure 5. These components are ordered by priority that we defined empirically based on the completeness and reliability of the information they provide. Each component also defines a list of conflicting components to avoid redundant topology information (that could even be different in case of bugs).

If any, the operating-system-specific component is loaded first (on Linux, it reads CPU, cache and NUMA information from `/proc` and `/sys` virtual files). Then, the CPU-specific component is invoked to add missing information (the x86 component detects additional CPU and cache characteristics). Then a PCI component takes care of gathering the locality of PCI bus, before I/O-specific components attach objects describing GPUs, NIC, accelerators, etc.

This component model lets user or administrator tune the information gathering process in case of buggy information. For instance, some Linux kernels report buggy cache information for AMD processors, that may be worked around by loading the x86 component first (the Linux component then only discovers what x86 did not). The user is advised when such a workaround is recommended but it should not be automatically

[7] `http://mj.ucw.cz/sw/pciutils/`

[8] `http://ezix.org/project/wiki/HardwareLiSter`

[9] hwloc v1.11 embeds 10 OS components, 6 I/O, 1 platform-specific, 1 CPU-specific, and 3 global components.

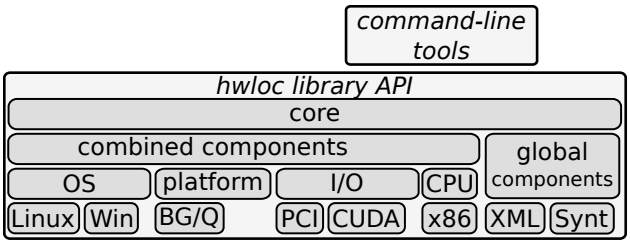


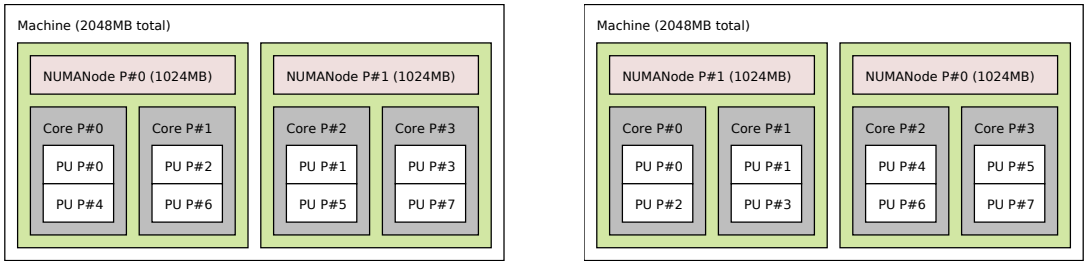
Figure 5: hwloc's component-based organization.

enabled. Indeed the exact list of affected platforms is hard to define and may even change in the future. The only exception is for platform-specific components (for BlueGene/Q and Fujitsu Sparc-based servers) which are automatically enabled because it is known that only one model of these exists and that their operating-system topology support will remain poor.

Finally some *global* components are standalone and cannot be combined because they already provide all topology information (XML and Synthetic topologies are described in Section 4).

3.3. Locating and Identifying Resources

Applications typically use the hwloc tree for locating where they are currently running, finding neighbor objects, consulting object attributes (see Section 3.4), and binding tasks or memory. There is a need for easy ways to locate and identify specific objects in a portable and abstracted manner.



(a) PU numbering by NUMA node first, then by core, then by PU.

(b) PU numbering by core first, then by PU, then by NUMA node; out-of-order NUMA node numbering.

Figure 6: Numbering of the processing units (PU) and NUMA nodes on dual-package dual-core hyper-threaded platforms. Two inter-dependent tasks running on logical processors 0 and 1 are actually not close to each other on these platforms. The binding cannot be portable unless it is specified as positions within the hierarchy of resources instead of as PU numbers. Memory binding is also non-portable because the relative ordering of NUMA nodes is different.

Computing and memory resources can be identified by indexes provided by the hardware and operating system (for instance the hardware thread that executes a given task). Unfortunately, these indexes are not portable from one machine to another because different vendors, BIOS versions or operating systems often use different ways to order CPU cores. It leads to cases where a standard dual-package platform can have up to 8 different numbering schemes in practice (see examples on Figure 6). hwloc therefore provides all these resources with a *logical* index that is portable because it is defined by the position in the hierarchical tree. Application may use such indexes to find out where they are running or where memory buffers are allocated, or bind new tasks to specific cores and memory buffers to specific NUMA nodes.

Managing I/O devices is more difficult because there are many kinds of these, and because applications do not actually natively use them like CPUs or memory. Indeed applications communicate on the network

using Sockets, write to files, or use CUDA or OpenCL structures for running kernels on GPUs. They use such *software handles* instead of using PCI devices explicitly, so offering the locality of PCI devices is not actually useful to end-users. hwloc therefore offers different ways for application to locate the I/O devices they want to use:

- hwloc inserts *OS device* objects in the topology tree to represent these software handles (CUDA device, network interface, etc.) inside the corresponding physical devices (PCI objects). They are annotated with a kind attribute and a name in particular that lets applications identify them (see *eth1* and *cuda0* among others on Figure 2, or *cuda0* and *mic1* on Figure 7). An application willing to use the first CUDA device just has to ask for the first *OS device* of kind *CUDA*, walk up the tree to find out which NUMA node is physically close to the corresponding GPU, and optionally walk down to find the local cores.

- hwloc provides interoperability helpers for the majority of user-space interfaces to high-performance devices (OpenCL, CUDA, Intel Xeon Phi co-processors, etc.). They translate application software handles into hwloc concepts and the corresponding locality information. For instance an application using InfiniBand will be able to retrieve the hwloc locality of the `struct ibv_device` that it already uses for posting RDMA requests.

Binding a process on cores close to a given device is such a common need (especially to perform device micro-benchmarking), that hwloc can do it automatically. For instance, the `hwloc-bind` command-line tool can be used as follows to execute the given program close to the OS device identified by its name with the `os=` prefix:

```
$ hwloc-bind os=mlx4_0 infiniband_benchmark
$ hwloc-bind os=cuda1 cuda_benchmark
$ hwloc-bind os=mic0 xeon_phi_benchmark
```

3.4. Exposing Hardware Characteristics

hwloc represents computing and memory objects using an exhaustive set of widespread resource types (PU, core, cache, package, NUMA node, machine, PCI, etc.). They may have type-specific attributes such as the memory size for NUMA nodes, the cache type and size, or bus ID for PCI devices. However, these static attributes cannot exhaustively cover the wide variety of hardware characteristics such as the CPU model and features, GPU memory and processor details, network address configuration, etc.

To be as descriptive as possible, hwloc even allows objects to be annotated with custom attributes. They are store as pairs of *key* and *value* strings, such as *Address=00:11:22:33:44:55*. To avoid requiring many string operations in applications, this mechanism is only used for uncommon attributes (widely-used attributes such as cache sizes are still stored as static fields within the object structure). This feature is notably used by Intel and Oracle-specific drivers for dynamically optimizing the Open MPI implementation on their platforms.

Each object usually contains at most a few key-value pairs, especially OS devices that describe GPU characteristics, NIC addresses, etc. (see Figures 7 and 8). These can be useful to distinguish between otherwise identical devices in a node or identical hosts in a network. Indeed, we need to identify network hosts and switches in order to manage clusters (as further discussed in Section 4.5). Similarly, we need to distinguish between processes executed on local or remote identical accelerators (e.g. Xeon Phi). MPI implementations such as Open MPI use hwloc *SerialNumber* attribute to detect whether two MPI processes are executed on the same machine, inside accelerators or not, and to choose the communication strategies accordingly.

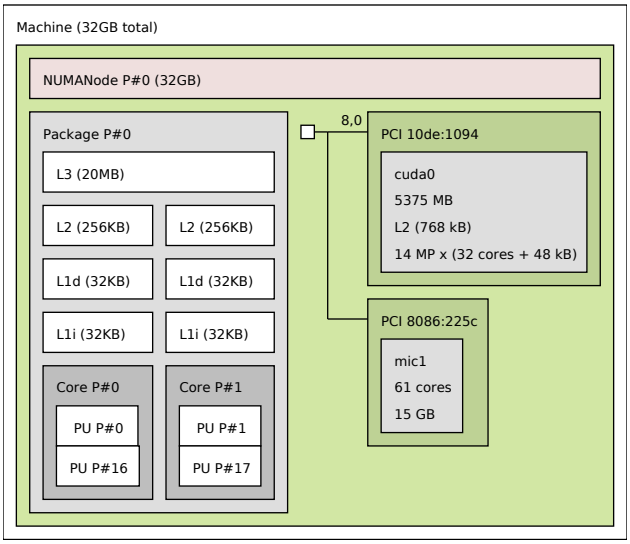


Figure 7: Object attributes include cache types (L1i, L3, etc), memory sizes, PCI device and vendor numbers, PCI link speed, Xeon Phi memory and cores, CUDA memory and multiprocessors, as well as CPU vendor and model (not displayed here).

```
Co-Processor (CoProcType=CUDA GPUVendor="NVIDIA Corporation" GPUModel="Tesla M2075"
  CUDAGlobalMemorySize=5504448 CUDAL2CacheSize=768 CUDAMultiProcessors=14 CUDACoresPerMP=32
  CUDASharedMemorySizePerMP=48) "cuda0"
Co-Processor (CoProcType=MIC MICFamily=x100 MICSerialNumber=ADKC32800176
  MICActiveCores=61 MICMemorySize=16252928 ...) "mic1"
OpenFabrics L#8 (NodeGUID=f452:1403:007a:7260 Port1GID0=fe80:0000:0000:0000:f452:1403:007a:7261
  Port1State=4 Port1LID=0x1 Port1LMC=0) "mlx4_0"
```

Figure 8: Textual dump of some attributes gathered by hwloc for OS devices describing a CUDA GPU (cuda0), a Xeon Phi co-processor (mic1) and a InfiniBand HCA (mlx4_0).

4. Managing Heterogeneous Clusters of Nodes

We now look at managing with hwloc the topologies of multiple nodes, such as a cluster. This is used for batch schedulers such as Slurm or Torque and process launchers found in most MPI implementations [24]. They gather computing node topologies on the front-end and apply cluster-wide locality-aware allocation or placement algorithms. We first describe the actual needs of these hwloc users.

4.1. Different Needs

We distinguish the following possible needs in topology-aware cluster-management tools:

Number of Cores: A batch scheduler does not need any knowledge of compute nodes as long as jobs request resources in terms of nodes instead of cores. However that is hardly the case, and the scheduler usually has to know at least the number of cores within each node. MPI process launchers also have this requirement for starting the right number of processes per node. This already raises the question of defining a *Core*: does the application want a real core or just a hardware thread? Giving real cores requires actual topology information.

Hierarchy of Resources: Clever resource allocation also tries to avoid breaking resource sets in pieces. For instance a scheduler processing a request for 6 cores among servers containing either two 6-core processors or two 4-core processors may want to allocate one entire 6-core processor (to avoid breaking one 4-core in two halves). Such strategies need the knowledge of the hierarchy of resources within each compute node.

Basic Attributes: Some object attributes are needed if the application can request specific kinds of CPUs or accelerators. Attributes such as indexes or memory size (see Section 3.3) are required once the batch scheduler reserves some processors and/or memory to isolate each job with mechanisms such as Linux cgroups. This information is also useful to runtimes such as MPI process launchers [24], or placement algorithms such as TreeMatch [9] that map tasks to hardware resources.

Full Attributes: When job allocation or task placement is performed using I/O locality, or when runtimes adapt their decisions to specific object information, the full topology of the machine is required.

We describe in the next sections how hwloc addresses these requirements.

4.2. Full XML Topologies

Managing the topology of multiple nodes requires a way to manipulate remote node topologies. We explained in the previous section that the full details of the topology may not be useful in all cases. However hwloc still supports that need anyway by allowing compute nodes to dump their entire topology to a XML file (or memory buffer) that can be transferred through a network and reloaded by the manager remotely. The loaded topology is strictly identical except that it cannot be used for actually binding task and memory since the local hardware is different.

This is useful for developing topology-aware algorithms and testing on a variety of different platform topologies without actually having access to these platforms. But it is also already widely used by batch schedulers and MPI process launchers: each compute node sends a XML copy of its local topology to the front-end node which implements the allocation/placement policy cluster-wide, before actually starting processes on the compute nodes. This strategy matches the **Full Attributes** case in Section 4.1.

XML also has the advantage of being very easy to load, much easier than rereading topology information from the different sources as explained in Section 3.2. Gathering the topology information natively on Linux indeed implies to read information from several hundreds of files under `/sys` and `/proc`. A naive MPI implementation running one process per core would load the topology once per core, causing all these files to be accessed by all cores simultaneously. Table 1 shows that the native Linux discovery does not scale well with the number of cores working in parallel (contention in the Linux kernel file-system locking code). On the other hand, XML import scales well. It also shows that very large machines may benefit from always loading from XML (up to 70x faster) even when not performing multiple discoveries simultaneously.

Table 1: hwloc topology discovery time depending on the source, either native Linux discovery, or XML import. On each host, we measure the time for a single discovery on one core, and for all cores discovering simultaneously their own copy of the topology.

Host	16 cores without I/O		16 cores with 3 GPUs		160 cores SGI Altix UV	
# Processes	1	16	1	16	1	160
Linux	26 ms	1 s	210 ms	6 s	390 ms	107 s
XML	3 ms	7 ms	3 ms	7 ms	12 ms	22 ms
XML buffer size	34 kB		39 kB		241kB	

However the XML export has the drawback of putting pressure on the network and front-end node by transferring lots of data from compute nodes. Indeed the size of hwloc XML exports scales in $O(P \log P)$ with P the number of cores[10]. Deeper hierarchies also generate slightly larger XMLs ($O(\log D)$ where D is the number of hierarchy levels in the machine) but we do not expect many new hierarchy levels to appear in hardware in the future.

Table 1 shows that current computing nodes generate XML files whose size varies from tens to hundreds of kilobytes. It is hard to predict whether the actual transfer of tens of thousands of such files on a supercomputer would be an important issue in term of performance. At least, the workload on the front-end for processing these XML topologies will likely be problematic. Therefore there is a need for alternative ways to describe the topologies of remote nodes.

4.3. Synthetic Topology Description

We explained in Section 4.1 that many use cases do not actually require the full details of the compute nodes. The entire topology may therefore be replaced with only the relevant topology information. hwloc provides the concept of *Synthetic Topologies* to tackle this need: the topology may be exported or created as a string describing the hierarchy of computing and memory resources. Each element of the string describes the children of each object specified by the previous element (which type and how many of them below). Our dual-processor 8-core hyper-threaded Intel Xeon and SGI Altix UV hosts are respectively described as:

```
NUMANode:2 Package:1 L3:1 L2:8 L1d:1 L1i:1 Core:1 PU:2
Group:10 NUMANode:2 Package:1 L3:1 L2:8 L1d:1 L1i:1 Core:1 PU:1
```

This approach is sufficient for the **Hierarchy of Resources** case in Section 4.1. Moreover this description is software-independent: it is identical for all cluster nodes of the same kind, and it only changes if hardware is modified. The manager on the front-end may therefore describe many nodes with the same string (usually less than one hundred characters). Such a factorization is not possible with XML exports because of several host-dependent attributes (network addresses, hostname, etc.). This will be further discussed in Section 4.4.

The synthetic description may even be further simplified to only report certain types of resources by ignoring some levels before exporting the synthetic description. For instance it may only report the number of NUMA nodes, cores and threads, which are the most widely used resources (**Number of Cores** case).

```
NUMANode:2 Core:8 PU:2
NUMANode:20 Core:8 PU:1
```

However this approach is not sufficient as soon as the manager requires **Basic Attributes** for actual process placement or processor/memory reservation. To address this need, the synthetic description may also include the most widely used attributes such as memory sizes and physical resource numbering:

```
NUMANode:2(memory=34330173440) Package:1 L3:1(size=20971520) L2:8(size=262144)
  L1d:1(size=32768) L1i:1(size=32768) Core:1 PU:2(indexes=2*16:1*2)
Group0:10 NUMANode:2(memory=34338770944) Package:1 L3:1(size=25165824)
  L2:8(size=262144) L1d:1(size=32768) L1i:1(size=32768) Core:1 PU:1
```

[10]The XML file contains one line per object, and those lines contain bitmasks (representing the object locality) whose sizes are proportional to processor numbers.

Resource numbering is only displayed if non linear. `2*16:2*1` is a hierarchical round-robin specification describing that physical PU indexes are ordered as 0, 16, 1, 17, ..., 14, 30, 15, 31 on our dual-Xeon host.

One drawback of synthetic topology description is that it does not currently support I/O devices. This would require to define which I/O devices are most-likely useful to add to the synthetic description (current servers contains several dozens of PCI devices, most of them being virtual and uninteresting for locality problems). This feature is currently under early design for future hwloc releases.

4.4. Differences Between (Almost) Identical Cluster Nodes

XML and synthetic topology descriptions have both advantages and drawbacks, but the gap between them is quite large. One way to bridge that gap is to factorize similarities between different nodes described as XML. It is an interesting approach in clusters to avoid having to manipulate a huge number of topologies at the same time on the front-end. We identified three actual possible differences between cluster node topologies:

- **different kinds of nodes** (e.g. compute node vs fat node vs GPU nodes): topologies are very different;
- **modified nodes** (BIOS upgrade, software update, or hardware replacement): topologies may be different;
- **similar nodes** with different identification numbers such as network addresses, hostname, etc.

In the *similar* case, only some key/value pair attributes are modified. In other cases, the tree structure can be different. Therefore, we added to hwloc the ability to compute the difference between 2 *similar* nodes by recording which attributes have been modified. This lossless compression consist in identifying a few *reference* nodes whose topologies will be entirely stored (uncompressed). All other nodes are then compressed by only storing the difference between their topology and one of the references. This feature is already used in the netloc submodule of hwloc (further described in Section 4.5).

Table 2: Memory consumed when loading the topologies of all nodes of a cluster within a single process. Topologies were either stored as full topologies (uncompressed), or as a few reference full topologies and many differences against one of these references.

	Total	Full topologies	Differences
Plafrim cluster = 21+65+16+9 compute nodes + 5 fat + 6 ssh			
Uncompressed	42 MB	122 × 345 kB	N/A
Compressed	11 MB	18 × 622 kB	104 × 2.03 kB
Avakas cluster = 264 compute + 2 phi + 4 fat + 4 visu + 2 ssh			
Uncompressed	110 MB	276 × 402 kB	N/A
Compressed	6.9 MB	12 × 539 kB	264 × 1.63 kB

Table 2 presents the memory occupancy improvement based on the compression of the topologies of two clusters[11]. Each cluster is made of different kinds of nodes (6 for Plafrim and 5 for Avakas), but we observe more reference topologies (respectively 18 and 12) because of the *modified* case above. However, many topologies can indeed be reduced from several hundreds of kilobytes down to 1 or 2 kB of differences in memory. Full topologies seem bigger in the compressed case because the share of fat nodes among reference topologies is higher.

[11] The `hwloc-compress-dir` utility was used.

Each difference is actually made of about 10 key/value pair attribute changes. We could even improve compression further by ignoring keys that are not needed by the target application (for instance the platform serial number, or the MAC addresses if only InfiniBand is used).

4.5. Multiple Node Topology

Finally, we look at how to manage a full, cluster-wide topology. In hwloc 1.4, we introduced a *Custom* API to assemble the topologies of multiple nodes into a global single one. It lets the application build a hierarchy of Groups (that represent switches) and attach node topologies as children.

However, the resulting topology must respect hwloc's tree model while networks interconnecting nodes may be a random graph. We explained in Section 2.3 that latency matrices can be used to annotate some levels of the tree but this idea is only satisfying for simple topologies such as NUMA interconnects or package rings. high performance fabrics can be made of dragonfly or torus topologies that cannot be represented as such an annotated tree.

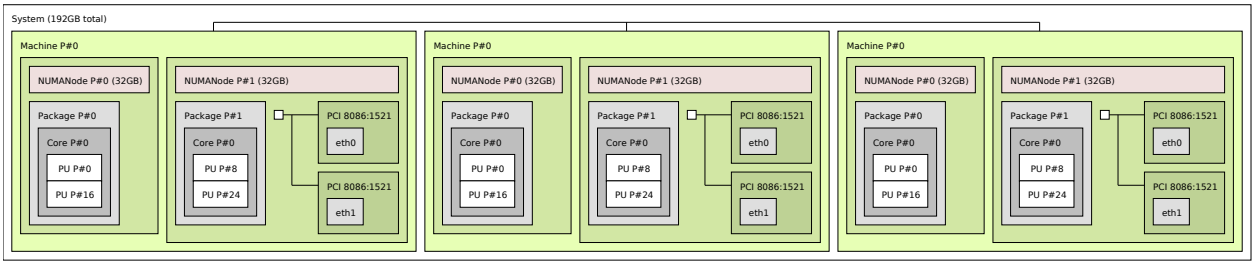


Figure 9: Custom topology made of 3 servers assembled by their *Machine* root object. They are actually physically connected to the network by their *eth0* and *eth1* interfaces that are close to their second NUMA node.

Moreover, the *Custom* API always attaches node topologies by the root of their tree (the entire *Machine* object, see Figure 9). This is inconsistent with the network locality inside the nodes because network interfaces (or InfiniBand HCAs) are often attached to a single NUMA node, and because there can be multiple interfaces per node.

Therefore, assembling multiple nodes into a global hwloc topology tree does not seem convenient. That is why the *Custom* API has been removed from the upcoming hwloc 2.0 release. There is an ongoing work to develop a hwloc submodule called netloc that will not enforce a tree model [27].

There is a netloc graph describing the network and there are hwloc topologies for each node. In addition netloc also offers ways to translate handles from one world (hwloc OS devices for NICs) to the other (NIC ports in the network graph). One drawback of this new approach is that there is no global structure describing the entire cluster. This would be an issue if batch schedulers or process managers had cluster-wide algorithms handling both the inter-node and intra-node cases simultaneously. Fortunately, there is no such need: the inter-node policy may be applied by weighting nodes based on their number of cores, while the intra-node policy is applied later separately.

5. Conclusion and Future Works

The increasing complexity of computing platforms raises the need for developers to understand the hardware organization and adapt their application layout. As part of the overall optimization process, there is a strong need for a tool modeling the platform, and hwloc is currently the most popular software for exposing a structural view of the topology of CPUs and memory. This approach is orthogonal to experimental performance models that give hints about the actual behavior of the resources exposed by hwloc.

We have presented in this article why and how the hwloc model has been designed as a hierarchical structural model, describing the locality of computing, memory and I/O resources. It combines locality information from many sources and offers APIs to interoperate with device-specific libraries. The hwloc tree also exposes many hardware attributes to help applications identify the resources they use, place tasks near them or adapt their behavior to their locality.

hwloc also offers ways to manipulate the topology of multiple nodes so that batch schedulers or MPI process managers may apply resource allocation and task placement policies between nodes and inside nodes. Multiple node topologies can be consulted offline, compared, compressed or synthetized depending on the actual topology information requirements.

All the features listed in this paper are actually implemented and ready to use in hwloc releases[12]. Ongoing work is now focusing on support for next-generation memory architectures with different kinds of memory (high-bandwidth, non-volatile) as discussed in Section 2.3.

Acknowledgments

Some of cluster node topologies used in this study were provided by the computing facilities MCIA (Mésocentre de Calcul Intensif Aquitain) of the Université de Bordeaux and of the Université de Pau et des Pays de l'Adour.

Some 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'Avenirs, see `https://www.plafrim.fr/`).

[1] A. Szalay, A. Bunn, J. Gray, I. Foster, I. Raicu, The importance of data locality in distributed computing applications, in: NSF Workflow Workshop, 2006.

[2] M. Steckermeier, F. Bellosa, Using locality information in userlevel scheduling, Tech. Rep. TR-95-14, University of Erlangen-Nürnberg – Computer Science Department – Operating Systems – IMMD IV, Martensstraße 1, 91058 Erlangen, Germany (Dec. 1995).

[3] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, 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), IEEE Computer Society Press, Pisa, Italia, 2010, pp. 180–186.
URL `http://hal.inria.fr/inria-00429889`

[4] S. Moreaud, B. Goglin, Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines, in: Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007), ACTA Press, Cambridge, Massachussetts, 2007, pp. 24–29.
URL `http://hal.inria.fr/inria-00175747`

[5] F. Song, S. Moore, J. Dongarra, Feedback-Directed Thread Scheduling with Memory Considerations, in: Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC07), Monterey Bay, CA, 2007, pp. 97–106.

[12] hwloc is available from `http://www.open-mpi.org/projects/hwloc/` under the BSD license.

[6] S. Kim, D. Chandra, Y. Solihin, Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, in: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 111–122.

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

[8] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, D. K. Panda, 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, 2012.

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

[10] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, S. Moreaud, Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis, in: Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009), IEEE Computer Society Press, Vienna, Austria, 2009, pp. 462–469.
URL `http://hal.inria.fr/inria-00390064`

[11] T. Ma, G. Bosilca, A. Bouteiller, J. J. Dongarra, Locality and Topology aware Intra-node Communication Among Multicore CPUs, in: Proceedings of the 17th European MPI Users Group Conference, no. 6305 in Lecture Notes in Computer Science, Springer, Stuttgart, Germany, 2010, pp. 265–274.

[12] S. Moreaud, B. Goglin, R. Namyst, Adaptive MPI Multirail Tuning for Non-Uniform Input/Output Access, in: E. G. Rainer Keller, J. Dongarra (Eds.), Recent Advances in the Message Passing Interface. The 17th European MPI User's Group Meeting (EuroMPI 2010), Vol. 6305 of Lecture Notes in Computer Science, Springer-Verlag, Stuttgart, Germany, 2010, pp. 239–248, best paper award.
URL `http://hal.inria.fr/inria-00486178`

[13] B. Goglin, S. Moreaud, Dodging Non-Uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters, in: CASS 2011: The 1st Workshop on Communication Architecture for Scalable Systems, held in conjunction with IPDPS 2011, IEEE Computer Society Press, Anchorage, AK, 2011, pp. 788–794.
URL `http://hal.inria.fr/inria-00566246`

[14] J. Treibig, G. Hager, G. Wellein, Likwid: A lightweight performance-oriented tool suite for x86 multicore environments, in: Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, 2010, pp. 207–216. doi:10.1109/ICPPW.2010.38.

[15] F. Song, S. Moore, J. Dongarra, Analytical Modeling and Optimization for Affinity Based Thread Scheduling on Multicore Systems, in: Proceedings of the 2009 IEEE International Conference on Cluster Computing, New Orleans, LA, 2009.

[16] B. Putigny, B. Goglin, D. Barthou, A Benchmark-based Performance Model for Memory-bound HPC Applications, in: Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014), Bologna, Italy, 2014, pp. 943–950. doi:10.1109/HPCSim.2014.6903790.
URL `http://hal.inria.fr/hal-00985598`

[17] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: Towards a Realistic Model of Parallel Computation, in: Principles Practice of Parallel Programming, 1993, pp. 1–12.

[18] H. Chen, W. Chen, J. Huang, B. Robert, H. Kuhn, MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters, in: Proceedings of the 20th Annual International Conference on Supercomputing, ICS '06, ACM, New York, NY, USA, 2006, pp. 353–360. doi:10.1145/1183401.1183451.

[19] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms, Journal of Parallel and Distributed Computing 74 (10) (2014) 2899–2917.

[20] E. Rodrigues, F. Madruga, P. Navaux, J. Panetta, Multi-core aware process mapping and its impact on communication overhead of parallel applications, in: Computers and Communications, 2009. ISCC 2009. IEEE Symposium on, 2009, pp. 811–817. doi:10.1109/ISCC.2009.5202271.

[21] J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, J. Touriño, Automatic Mapping of Parallel Applications on Multicore Architectures using the Servet Benchmark Suite, Computers and Electrical Engineering 38 (2012) 258–269.

[22] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, R. Namyst, ForestGOMP: an efficient OpenMP environment for NUMA architectures, International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Mller and Eduard Ayguad 38 (5) (2010) 418–439. doi:10.1007/s10766-010-0136-3.
URL `http://hal.inria.fr/inria-00496295`

[23] F. Pellegrini, Scotch and libScotch 5.1 User's Guide, `https://gforge.inria.fr/docman/view.php/248/7104/scotch_user5.1.pdf` (Dec. 2010).

[24] J. Hursey, J. M. Squyres, Advancing Application Process Affinity Experimentation: Open MPI's LAMA-Based Affinity Interface, in: Recent Advances in the Message Passing Interface. The 20th European MPI User's Group Meeting (EuroMPI 2013), ACM, Madrid, Spain, 2013, pp. 163–168.

[25] Intel Corporation, What public disclosures has Intel made about Knights Landing?, `https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing`.

[26] B. Gerofi, M. Takagi, Y. Ishikawa, Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II, Springer International Publishing, 2014, Ch. Exploiting Hidden Non-uniformity of Uniform Memory Access on Manycore CPUs, pp. 242–253.

[27] B. Goglin, J. Hursey, J. M. Squyres, 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, Minneapolis, MN, 2014, pp. 216–225.
URL `http://hal.inria.fr/hal-01010599`