Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc)

Brice Goglin

Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc)

Brice Goglin
Inria Bordeaux - Sud-ouest – University of Bordeaux – 33405 Talence cedex – France
Brice.Goglin@inria.fr

***Abstract*—Modern computing platforms are increasingly complex, with multiple cores, shared caches, and NUMA architectures. 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 and localities of the CPU and memory resources. It is already widely used for affinity-based task placement in high performance computing.**

In this article we present how hwloc is extended to describe more than computing and memory resources. Indeed, I/O device locality is becoming another important aspect of locality since high performance GPUs, network or InfiniBand interfaces possess privileged access to some of the cores and memory banks. hwloc integrates this knowledge into its topology representation and offers an interoperability API to extend existing libraries such as CUDA with locality information. We also describe how hwloc now helps process managers and batch schedulers to deal with the topology of multiple cluster nodes, together with compression for better scalability up to thousands of nodes.

***Keywords*—topology; locality; affinities; I/O devices; clusters; hwloc**

I. 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.

Performance optimization of parallel applications require a thorough knowledge of the hardware, and many research projects aim to model the platform to tackle this challenge. Besides analytical performance models, one solution consists in static 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 [1].

However, the locality importance has grown and it now applies to high-performance I/O devices such as accelerators or network interfaces. Moreover, several 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.

We present how hwloc has evolved into a central place for gathering locality information about all hardware subsystems in HPC servers. It achieves this goal by combining topology information from many sources, including operating systems, domain-specific libraries and platform-specific instructions. It interoperates with these sources by extending their interfaces with locality information about the devices they manage. hwloc also offers ways to manipulate multiple nodes topologies with the ability to avoid duplication in case of nearly-identical cluster nodes.

The remainder of this paper is organized as follows: Section II introduces the challenges and use cases for providing topology information. Section III then summarizes the hwloc model and describes how it manages all sources of information. I/O device locality within heterogeneous nodes is then presented in Section IV while the management of multiple nodes topologies is described in Section V.

II. Context and State of the Art

A. Why Locality matters and Where

Locality has been cited as a critical aspect of performance of parallel applications for a long time, from distributed computing [2] to single servers [3]. The complexity of modern computing platforms is increasing, even inside commodity nodes. Figure 1 shows the hierarchical organization of resources within a widespread type of servers where some physical devices have affinities for some cores and memory banks. Developers and users have to take the hardware topology into account when trying to optimize their codes.

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 from one core to another (or worse, from one NUMA node to another) usually causes the execution to slow down because of cache affinities. Thus, it is well-known that computing tasks should be bound to a single core to avoid such migration. Migration can also cause the performance to vary depending on the cores' locality with regard to the I/O devices used by the task [4].

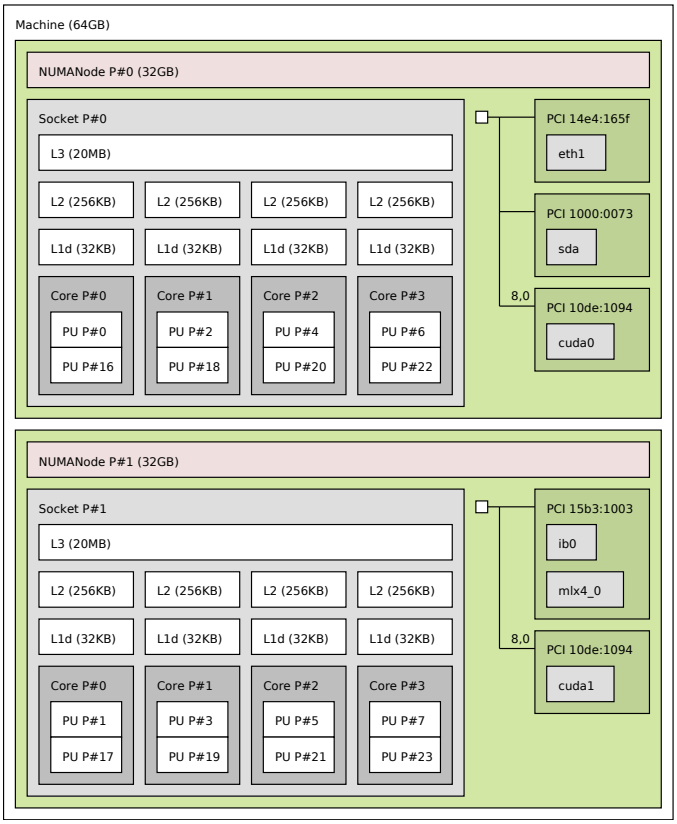


Figure 1. Topology of a dual-Xeon E5 host with GPUs (cuda0, cuda1), network (eth1), InfiniBand (mlx4_0) and disk (sda) connected to different sockets, simplified and reported by hwloc's `lstopo` tool.

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].

Moreover, some energy-based affinities may also be involved. Technologies such as Intel TurboBoost can improve sequential performance on partially-idle multicore processors, while some processors can be shutdown completely when entirely idle.

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].

B. Many Sources of Hardware Information

Tackling locality issues within parallel applications actually involves three steps: gathering the hardware topology, expressing the software affinities, and matching one with another. We focus on the former in this article: how to gather, abstract and expose useful hardware 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 HPC. 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.

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. Tools such as the Intel compiler and LIKWID[1] use this idea and end up failing to discover the right topology on some custom platforms. 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.

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[2] possesses knowledge of NUMA, CPU and I/O locality but lacks caches. Moreover, its programming interface is unstable, and it was designed for binding tasks only: it cannot be used for querying details about hardware characteristics.

As shown on Figure 2, many libraries exist for querying the topology of specific subsystems, for instance pciutils for PCI[3], libibverbs for InfiniBand, CUDA for NVIDIA GPUs, etc. Unfortunately, there is almost no interoperability between these libraries and other topology-related tools. Therefore, it is for instance necessary to query sysfs, pciutils and CUDA

[1] http://code.google.com/p/likwid/
[2] http://oss.sgi.com/projects/libnuma/
[3] http://mj.ucw.cz/sw/pciutils/

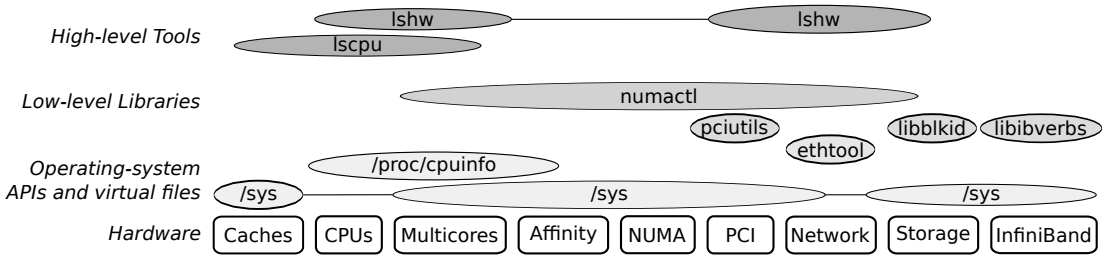


Figure 2. Overview of existing sources of locality information on Linux.

when looking for the locality of a NVIDIA GPU with regard to host CPUs.

Some higher level tools such as lscpu or lshw[4] merge the information from several sources but they lack a programming interface. In brief, all these sources of information still have to be used concurrently for a developer to gather locality information about all hardware resources. Some non-Linux operating systems may have better interfaces but they lack part of the information. For instance, Solaris does not report cache information. There is therefore a need for a portable, system-wide topology discovery tool.

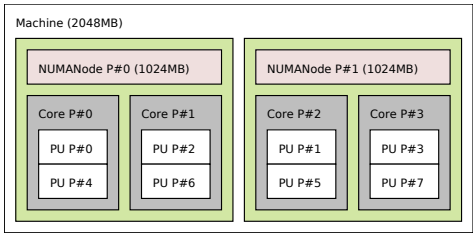
C. Execution and Memory Binding

Besides consulting topology information, the other important technical requirement for tackling locality is binding. Applications need ways to specify that a task or memory buffer should be allocated to one (or some) hardware resources. Many command-line binding tools exist, including numactl, taskset and schedtool on Linux. But most of them may bind tasks only. Moreover, they only operate on sets of logical processors without any knowledge of processor sockets, caches, etc.

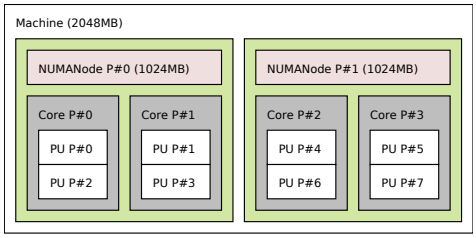
Manipulating sets of logical processors unfortunately raises the issue of resource numbering. The BIOS and operating system are indeed free to renumber hardware devices, especially processor cores, based on their expectations of what the best numbering is. Memory bandwidth needs usually lead to numbering by NUMA node first, while sequential performance would likely number by hyperthread first. It leads to cases where a standard dual-socket platform can have up to 8 different numbering schemes depending on the vendor or BIOS version (see examples on Figure 3). Applications cannot be portable anymore if they rely on physical resource numbers.

A higher-level approach, based on the hierarchical organization, is required to keep affinity information: two cores sharing a L2 cache are considered neighbors even if their OS indexes are 0 and 4 respectively. Discovery and binding interfaces must therefore be integrated so that the same objects are manipulated for querying information about the platform hierarchy and resource characteristics, and for binding on these resources, without exposing numbering issues. We now present the Hardware Locality project that was notably designed to solve this particular problem.

[4]http://ezix.org/project/wiki/HardwareLiSter



(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.

Figure 3. Numbering of the processing units (PU) on dual-socket dual-core hyperthreaded 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.

III. HWLOC'S VIEW OF THE HARDWARE

The Hardware Locality project was announced in 2009 as the replacement and merger of former Open MPI PLPA[5] and Inria libtopology[6] projects. It quickly raised attention of HPC runtime developers as an easy way to discover the topology of servers and to bind tasks. hwloc is now used by most MPI implementations, many batch schedulers and parallel libraries[7]. We summarize in this section the early design choices that led to hwloc success before explaining its evolution into the central place for information and interoperability about the topology of multiple hardware subsystems.

[5]http://www.open-mpi.org/projects/plpa/

[6]http://runtime.bordeaux.inria.fr/hwloc/

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

A. Organizing the Information

hwloc resource organization is based on the natural inclusive order of computing resources: every machine contains one or several sockets, that contain one or several cores. hwloc builds a *Tree* of *Objects* describing these computing resources organized just like they are physically packaged. hwloc cores can actually contain multiple *Processing Unit* objects (PU), defined as the smallest resource that can execute a thread or a process. PUs correspond to logical processors or hardware threads as found in technologies such as simultaneous multi-threading or Intel hyperthreading.

Each hwloc object is characterized by a type, some hardware characteristics such as a socket number, and some optional parameters such as local cache or memory sizes. The inclusion-ordering is extended to memory objects by considering that cores sharing a cache or near a NUMA memory node are included in it. Thus, the tree is made of a mix of levels made of computing and memory resources, ordered by locality without depending on actual physical numbering.

hwloc does not enforce the vertical ordering between these levels in the tree because some AMD platforms have two NUMA nodes per socket (see Figure 4) while some Itanium machines have multiple sockets per NUMA node. hwloc just moves larger objects above smaller ones depending on the architecture inclusion characteristics. Sections IV and V explain how hwloc was recently extended to more than computing and memory resources inside nodes.

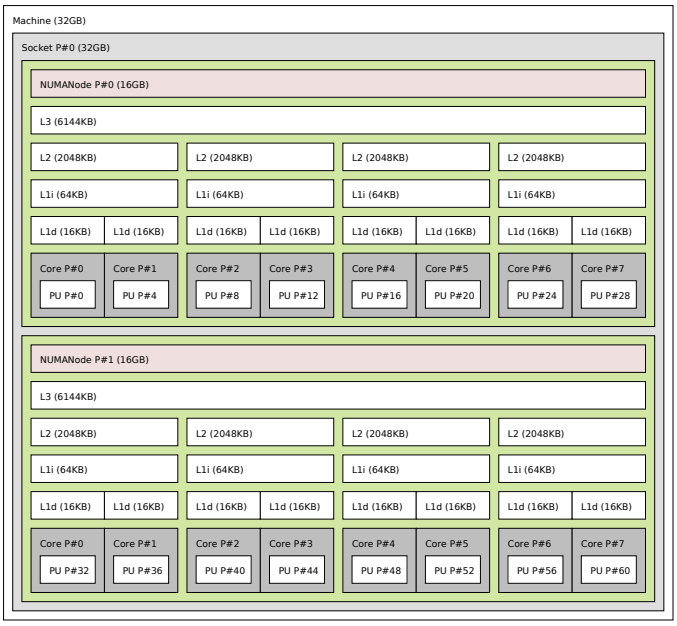


Figure 4. AMD platform containing a single Opteron 6272 socket. hwloc's inclusion ordering is machine, socket, NUMA node, L3, L2, L1i, L1d, core, PU.

The hwloc programming interface allows walking the tree edges to find ancestors or children 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 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 [1]. In the rest of the paper, we focus on major improvements in later releases.

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 assemble cores on a ring, and the NUMA memory interconnect is not always a complete graph. Both are ignored when objects are represented as an array of children. To workaround this constraint, hwloc now annotates the tree with distance matrices and creates additional hierarchical *Groups* of object close to each-other. Large SGI Altix UV platforms are therefore represented with multiple levels of Groups between the machine and NUMA nodes so that the physical organization as racks and blades is exposed.

B. Orchestrating multiple Sources of Information

1) Combining multiple Sources: As explained in Section II-B, multiple sources have to be used to gather all topology information about the machine. On Linux, virtual files under both /sys and /proc have to be used. x86-specific instruction may also bring more precise information about the CPU type, especially for non-Linux operating systems. I/O information involves several specific libraries as well as other virtual files under /sys. The hwloc library is therefore organized as several backend components.

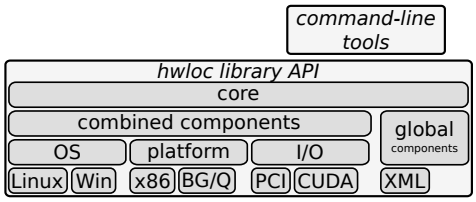


Figure 5. hwloc's component-based organization.

Discovery usually uses several components to match the aforementioned combination of sources as show on Figure 5[8]. Information is first gathered from *operating system* components (most Unix systems, Windows and Mac OS) that use OS interfaces. It is then extended by *platform-specific* components (BlueGene/Q, x86, Xen). I/O discovery is finally performed using specific libraries such as pciutils and CUDA. Each component can specify conflicts with others, and priorities can be changed to avoid a component that would return wrong information on a given platform. Moreover, some inter-component callbacks can be specified so that PCI discovery immediately checks whether a new PCI device corresponds to one of the CUDA devices.

[8]The XML import backend cannot be combined, it is a global component that manages all objects, as explained in Section V-A).

2) Interoperating with external Libraries: hwloc components can be built either statically inside the main hwloc library or as separate plugins. This is necessary for a convenient distribution of binary packages to avoid strong dependencies on external libraries. Indeed, binary packages should support as many cases as possible, which means hwloc should be built using all aforementioned I/O libraries such as pciutils and CUDA, but such dependencies are not acceptable for administrators that do not have GPUs on their platform. Building as plugins is an easy way to make these dependencies optional: all plugins are installed by binary packages but hwloc only loads plugins whose dependency libraries are available on the system. Obviously, it is still possible to build a custom hwloc library from source and embed all components that are useful to a given platform.

hwloc uses external libraries to gather topology information. However, it was not designed to *replace* these libraries that offer a lot more of features unrelated to topology. hwloc was rather designed as a central place of topology details that *interoperates* with existing libraries and extend them with locality information. It therefore offers several interoperability headers that let developers translate between external library objects and hwloc data structures. For instance, an application using CUDA or InfiniBand verbs can retrieve the locality of `CUdevice` or `struct ibv_device`. Applications can therefore keep using existing specific libraries for non-topology-related information and switch to hwloc for topology-related queries.

IV. Managing Heterogeneous Nodes

We explained in the previous section how the hwloc library combines multiple sources of topology information from all subsystems in the machine. We now detail how it actually manages heterogeneous servers combining CPUs, accelerators such as GPUs or Xeon Phi, and network or InfiniBand interfaces.

A. I/O Discovery

I/O controllers are often placed near one of the processor socket within servers. They are even integrated inside sockets on recent Intel processors. Hence, devices connected to these controllers have a privileged access to the local memory and cores. These I/O affinities actually matter to latency or throughput sensitive applications that use high performance GPUs or network interfaces. Thus, it is important to offer I/O locality information to applications so as to optimize their placement and use of I/O devices [12], [13].

We added the ability to expose I/O device affinity in hwloc. The inclusion-based tree has been extended to attach new I/O objects under hwloc computing and memory resources they are close to (usually a NUMA node). Since high performance I/O is only significant for PCI devices, PCI is the only I/O hierarchy that is currently discovered by hwloc, using either an external PCI library such as pciutils or Linux sysfs files. PCI bridges are also discovered (see Figure 6) in case applications need to know which devices share PCI links and the speed of these links. But the tree may also be simplified to only retain the actual locality of PCI devices.

B. Identifying Objects from Applications

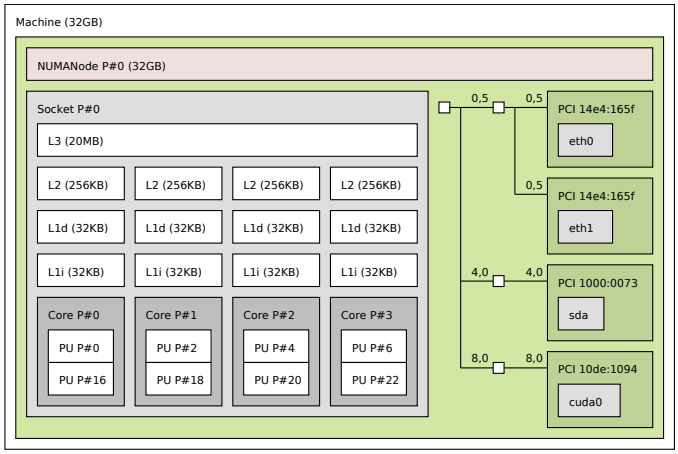


Figure 6. I/O device hierarchy connected to a NUMA node. Grey boxes inside dark green boxes on the right are OS device objects inside PCI devices. Small squares represent bridges, and decimal values are PCI link speeds in GB/s.

The main issue with I/O discovery is that applications do not manipulate PCI devices, they operate on *software handles* instead, such as network socket file descriptor, InfiniBand `ibv_device`, CUDA `CUdevice`, etc. When multiple similar devices are available in the system, finding which hardware device corresponds to the application software handle can be difficult.

hwloc solves this issue by inserting *OS device* objects describing these software handles inside *PCI device* objects, and adding human readable types and names. For instance, a *cuda0* device is inserted so that the locality of CUDA device #0 can be retrieved by walking up the tree across the PCI hierarchy up to NUMA node #0 on Figure 6.

One immediate advantage of this feature is for binding microbenchmarks. Instead of manually binding a network ping-pong to a core near the InfiniBand interface `mlx4_0`, binding can be performed automatically near a specific OS device with the `hwloc-bind` tool:

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

C. Identifying Objects from outside the Host

We described in the previous section how to identify I/O devices from host applications. We now look at identifying them from other points of view. The first use case is for matching Xeon Phi boards as viewed from the host and from inside the board. Indeed, one way to use Xeon Phi is

to mix MPI ranks on the host CPUs and on the Phi. MPI communication have to be implemented depending on whether the Phi and the CPU are located inside the same server or not. One requirement from MPI implementers is therefore to identify which MPI ranks are inside the same server. We solved this issue by extracting the Phi serial number and making it available in the hwloc topologies of the host (in the Phi OS device) and of the Phi itself (in the root object). This is already in use in Open MPI 1.7.

Another case where devices have to be identified from outside the host is network and InfiniBand interfaces. MAC/IP addresses and InfiniBand GUID/LIDs respectively are the only way to refer to a specific remote host, especially when multiple interfaces or ports exist. We therefore added to hwloc I/O objects several additional attributes enabling such identification. This feature is already used by the netloc project as discussed in Section V-C.

Examples of InfiniBand and Xeon Phi I/O-specific attributes added to hwloc OS devices are presented on Figure 7.

```
Co-Processor L#5 (CoProcType=MIC MICFamily=x100
  MICSerialNumber=ADKC32800176 MICActiveCores=61
  MICMemorySize=16252928 ...) "mic0"
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 7. Textual dump of some attributes gathered by hwloc for OS devices describing a Xeon Phi (mic0) and a InfiniBand HCA (mlx4_0).

D. Being Generic enough

hwloc represents CPU and memory objects using an exhaustive set of widespread resource types (PU, core, cache, socket, NUMA node, machine) as well as additional generic objects (such as Groups for describing intermediate affinity neighborhoods in the tree). I/O objects raised the need to provide even more convenient types. Unfortunately, there are many different types of I/O devices and hwloc cannot list all of them explicitly. Moreover, many of these objects have specific attributes such as the memory size, the cache type, the network address, etc. Therefore, there is a need for a generic way to annotate hwloc objects with custom attributes instead of adding many hardwired type-specific structures of attributes.

hwloc usually gathers about one hundred object attributes for an entire server. They are attached to the relevant objects within the tree or to the root object when the attribute applies to the entire topology. These generic attributes are stored as a pair of *key* and *value* strings, such as *Address=00:11:22:33:44:55*. This presents the drawback of requiring string manipulations for applications. As a consequence, widely-used attributes (such as cache sizes) are stored using explicit fields within the object structure instead of generic key/value string pairs.

Each object needs a way to store these key/value pairs. The number may vary but it always remains small, usually lower than 10. So there is no need to optimize lookup with advanced data structures such as hash tables. hwloc just uses an array of key/value pairs in each object. This mechanism is already widely used in hwloc since many users requested new attributes to be added. For instance, Intel and Oracle-specific drivers consult hwloc CPU attributes to dynamically optimize the Open MPI implementations on the corresponding platforms.

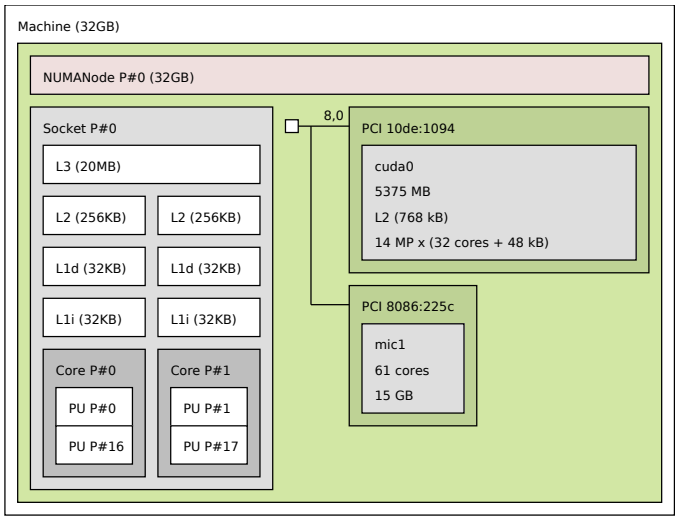


Figure 8. Object attributes include cache types (L1i, L3, etc), memory sizes, object numbers, PCI device and vendor numbers, PCI link speed, Xeon Phi memory and cores, CUDA memory and multiprocessors, as well as CPU vendor and model.

V. Managing 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 [14]. They have to know how many cores each node features before deciding how many processes should run. Moreover knowing topology details let them allocate resources better. The topology of multiple nodes can then be combined to build a global cluster-wide topology, where placement algorithms can perform both inter-node and intra-node management simultaneously [8], [9].

A. Remote Node Topology

Managing the topology of multiple nodes requires a way to manipulate remote node topologies. Since most topology information pieces are read from virtual files on Linux, hwloc gained the ability to change the filesystem root path in order to use copies of the /sys and /proc files from another node. This feature is convenient for debugging the Linux discovery code without immediate access to a remote node but hwloc offers a more convenient solution: an API to import/export entire topologies to XML, either as a file, or as a memory buffer that can be transferred on the network.

This is useful for developing topology-aware algorithms and testing on a variety of different platform topologies. But it is also already widely used by MPI process launchers: each compute node sends a XML copy of its local topology to the frontend node which implements the process placement algorithms cluster-wide, before actually starting processes on the compute nodes.

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 II-B. Discovery the topology natively on Linux indeed reads 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 accesses by all cores simultaneously. Table I shows that the native Linux discovery does not scale well with the number of cores working in parallel (contention in the Linux kernel filesystem locking code) while XML import scales well. It also shows that very large machines may benefit from always loading from XML (up to 70x faster) even when a single discovery is performed simultaneously.

TABLE I
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 AND FOR ALL CORES DISCOVERING SIMULTANEOUSLY.

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

B. Cluster Nodes are (almost) Identical

Once compute node topology has been retrieved on a master node, one may wonder if storing all of them locally scales to a high number of nodes. Moreover, cluster nodes are usually similar: clusters are made of a single (or few) types of nodes. Why storing the topologies of all nodes if most of them are identical? We identified three actual possible differences between cluster node topologies:

- **different kinds of nodes** (e.g. compute node vs fat node): topologies are very different;
- **modified nodes** (BIOS upgrade, software update, or hardware replacement): topologies can be different;
- **similar nodes** with different identification numbers such as MAC address, InfiniBand GUID, etc.

In the *similar* case, only for key/value pairs 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 key/value pairs have been modified. This loss-less 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 project.

TABLE II
MEMORY OCCUPANCY OF HWLOC TOPOLOGIES FOR 2 CLUSTERS WHEN 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 = 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 = 264 compute + 2 phi + 4 fat + 4 visio + 2 ssh			
Uncompressed	110 MB	276 × 402 kB	N/A
Compressed	6.9 MB	12 × 539 kB	264 × 1.63 kB

Table II presents the memory occupancy improvement based on the compression of the topologies of two clusters of the University of Bordeaux[9]. 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 in memory. Full topologies seem bigger in the compressed case because the share of fat nodes among reference topologies is higher.

Each difference is actually made of about 10 key/value pair 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).

C. Multiple Node Topology

Finally, we look at how to manage a full, cluster-wide topology. hwloc offers an API to assemble the topologies of multiple nodes into a global single one. However, the resulting topology must respect hwloc's tree model while networks interconnecting nodes can be random graphs. We explained in Section III-A that distance 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 socket rings.

Moreover, cluster nodes are interconnected by NICs or InfiniBand HCAs at the bottom of the tree, not by the hwloc tree roots (the entire machine object). Therefore, assembling multiple nodes into a global hwloc topology does not seem convenient. That is why there is an ongoing work to develop a hwloc-companion called netloc[10] to combine hwloc node topologies with network graphs without enforcing a tree model [15].

[9]The hwloc-compress-dir utility was used.

[10]Available under the BSD license at http://www.open-mpi.org/projects/netloc/.

VI. 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 the most popular software for exposing a static view of the topology of CPUs and memory.

We presented in this article how we have extended hwloc to more than these computing resources by also incorporating the topology of I/O devices and offering ways to manage multiple nodes. hwloc now integrates locality information from many sources and offers APIs to interoperate with these libraries and operating systems without replacing them. I/O locality information has been added to the hwloc tree representing the hardware as well as many attributes to help applications identify the resources they use, place tasks near them or adapt to their locality. An API to manipulate the topologies of remote hosts with compression for better scalability was also recently added.

All features listed in this paper are available in hwloc v1.9 (released in Spring 2014)[11]. On-going work is now focusing on improving topology detection on emerging ARM architectures for high-performance computing as well as automatic management of conflicts between redundant sources of information.

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 LABRI and IMB and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS.

References

[1] 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," in *Proceedings of the 18th Euromicro Intl Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia, Feb. 2010, pp. 180–186.

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

[3] M. Steckermeier and F. Bellosa, "Using locality information in userlevel scheduling," University of Erlangen-Nürnberg – Computer Science Department – Operating Systems – IMMD IV, Tech. Rep. TR-95-14, Dec. 1995.

[4] S. Moreaud and 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)*, Cambridge, Massachussetts, Nov. 2007, pp. 24–29.

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

[5] F. Song, S. Moore, and J. Dongarra, "Feedback-Directed Thread Scheduling with Memory Considerations," in *Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC07)*, Monterey Bay, CA, Jun. 2007.

[6] S. Kim, D. Chandra, and 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)*, Washington, DC, USA, 2004, pp. 111–122.

[7] 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)*. ACM, Jun. 2011, pp. 75–85.

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

[9] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2013.

[10] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis," in *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*, Vienna, Austria, Sep. 2009, pp. 462–469.

[11] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Locality and Topology aware Intra-node Communication Among Multicore CPUs," in *Proceedings of the 17th European MPI Users Group Conference (EuroMPI 2010)*, ser. Lecture Notes in Computer Science, vol. 6305. Stuttgart, Germany: Springer, Sep. 2010.

[12] S. Moreaud, B. Goglin, and R. Namyst, "Adaptive MPI Multirail Tuning for Non-Uniform Input/Output Access," in *Proceedings of the 17th European MPI Users Group Conference (EuroMPI 2010)*, ser. Lecture Notes in Computer Science, vol. 6305. Stuttgart, Germany: Springer, Sep. 2010.

[13] B. Goglin and 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*, Anchorage, AK, May 2011.

[14] J. Hursey and J. M. Squyres, "Advancing Application Process Affinity Experimentation: Open MPI's LAMA-Based Affinity Interface," in *Proceedings of the 20th European MPI Users Group Conference (EuroMPI 2013)*. Madrid, Spain: ACM, Sep. 2013, pp. 163–168.

[15] B. Goglin, J. Hursey, and J. M. Squyres, "netloc: Towards a Comprehensive View of the HPC System Topology," in *Submitted to the 43rd International Conference on Parallel Processing (ICPP-2014)*, Minneapolis, MN, Sep. 2014.