On the Overhead of Topology Discovery for Locality-aware Scheduling in HPC

Brice Goglin

On the Overhead of Topology Discovery for Locality-aware Scheduling in HPC

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

***Abstract*—The increasing complexity of parallel computing platforms requires a deep knowledge of the hardware and of the application needs. Locality a key criteria for performance optimization. It involves software tools to expose information about the hardware topology to high performance runtime libraries.**

We show that the overhead of gathering such information from the operating system is significant on large computing nodes that run Linux. This overhead also increases more than linearly with the number of processes that perform it simultaneously.

We then study the actual needs of the HPC software ecosystem in terms of topology information. We propose some ways to avoid multiple expensive topology discovery and to share topology information between components such as the resource manager or the runtime libraries.

***Keywords*—topology; locality; discovery; overhead; operating system; Linux; file-system; XML**

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. It had a critical impact on the performance of parallel applications for a long time, from distributed computing [1] to single servers [2].

Tasks may 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 [3]. Another kind of affinity exists 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 [4]. 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 [5].

While understanding application needs is important, performance optimization of parallel applications also requires a thorough knowledge of the hardware. On the road to exascale, such criteria become critical to performance because the computing nodes are increasing large. As an example, latest Intel Knights Landing Xeon Phi contains between up to 72 cores, with 4 hyper-threads each. Many research projects aim to model the platform to tackle this challenge. Structural models as a hierarchy of processor packages, NUMA nodes, caches, cores and hardware threads is a convenient way to expose topology information to HPC runtime libraries [6]. It requires to query information from the operating system and assemble the output in a hierarchy manner.

The hwloc software project has evolved into the *de facto* central place for gathering locality information about all hardware subsystems in parallel platforms. This paper discusses the overhead of this topology discovery process. After presenting why modeling the structure of the hardware is important in Section II, we show in Section III how software tools can actually query the operating system about hardware resources and topology. We then show that this discovery process has an important overhead on Linux when performed on large nodes such as a Intel Knights Landing processor and a SGI Altix UV. This overhead does not scale when the discovery is performed simultaneously by multiple processes, for instance when multiple MPI ranks run on a large node. We then discuss in Section IV whether multiple software components actually need to perform this discovery multiple times and/or simultaneously and what kind of topology information they actually need. This leads us to propose ways to improve the topology management in Section V by avoiding multiple expensive discovery through the operating system. We also enable the sharing and compression of topology information between software components such as the resource managers and the HPC runtime libraries.

II. Modeling the Structure of Computing Platforms

The complexity of modern computing platforms makes them increasingly harder to use, causing the gap between peak performance and application performance to widen. We explain in this section why locality is critical to HPC application performance, why we should provide a model of the platform

to the runtime libraries, and why hwloc's structural model in a convenient solution.

A. On the Importance of Locality

Modern processors are deeply hierarchical. As depicted on Figure 1, they contain many cores[1], several levels of caches, either shared or private, and possibly multiple NUMA nodes[2]. Running HPC tasks on such architectures requires careful placement since their performance depends on locality. There are many possible reasons for applying specific placements. For instance, memory-bound tasks should likely run close to the NUMA node that contain their data buffers. Tasks sharing data buffers may run faster if their cores share a cache. Tasks that communicate or synchronize a lot with each other may perform better if the physical distance between them is small.

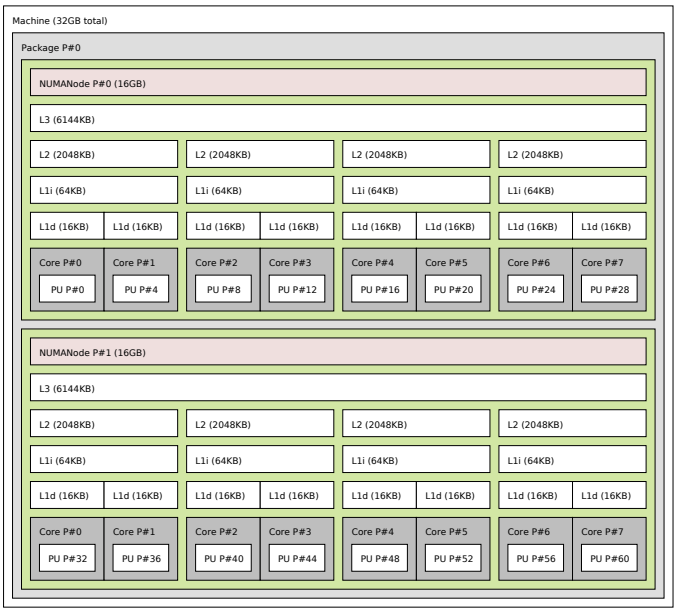

Figure 1. Hierarchy of resources inside a AMD Opteron 6272 processor as reported by hwloc's `lstopo` tool. This processor package is made of two parts containing one NUMA node and one L3 cache each. L2 and L1i caches are then shared by *Compute Units* pairs of cores, while the L1d is private to each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

These criteria are now widely understood by HPC runtime developers since users often want to manually place their tasks. Most MPI runtimes let users specify how MPI ranks are distributed between nodes and ordered on their cores [7]. The actual implementation of MPI communication strategies may also be adapted to the locality between cores [8], or between cores and network interfaces [9]. For instance, a shared cache between two cores makes double-copy strategies perform better. Similar placement strategies may also be applied in OpenMP runtimes by having the application use *Places* that correspond to the underlying hardware organization [10].

[1]Latest Intel E5v4 Xeon *Broadwell* features up to 22 cores.

[2]Intel Xeon E5 since v3, AMD Opteron since 6100, IBM POWER8 and Fujitsu Sparc XIfx may all contain 2 NUMA nodes per processor.

B. Modeling Platform for Performance Analysis

Understanding the platform behavior under different kinds of load is critical to performance optimization and proper task placement. Performance counters is a convenient way to retrieve information about bottlenecks for instance in the memory hierarchy [11] and apply feedback to better schedule the next runs [12]. The raw performance of a server may also be measured through different memory access workloads to predict the behavior of kernels [13]. 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.

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 [14] may be used to describe the network performance and build a hierarchy of processors based on this experimental distance for better process placement [15]. Improved performance models have been proposed since then to offer realistic simulation on larger platforms [16]. 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 [17]. Such an approach may actually also help experimentally rebuilding the entire topology of the clusters for better task placement [18].

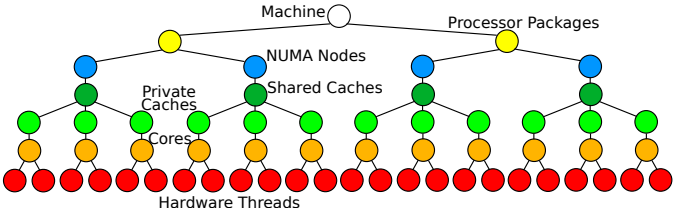

Figure 2. Structural Modeling of a dual-processor host. Each processor contains two dies that each contain one NUMA node and a shared cache, 3 cores. Each core has its a own private cache and 2 hardware threads.

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 different ways, and the impact 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 [19], MPI process placement [20] and task-based programming languages [21] are examples of scheduling opportunities that can benefit from deep platform topology knowledge through the structural modeling of hardware resources offered by the hwloc software [6]. Figure 2

shows an example of such modeling for a dual-processor server.

III. Discovering and Organizing Hardware Information

We describe in this Section how software may discover the platform topology. It means finding out the available computing, memory and I/O resources, as well as their locality, so as to build a structural model of the hardware platform.

A. 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 systems 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 pseudo-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. Other operating systems such as Solaris or Windows have dedicated system call interfaces for retrieving similar information (`kstat`, `GetLogicalProcessorInformation`, etc.).

Many processors also 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.

When it comes to I/O devices such as network interfaces or GPUs, finding their locality is even more tricky. First, specific tools (such as the CUDA SDK or the InfiniBand Verbs API) should be used to find the corresponding PCI devices. Then, operating system APIs have to be used to find the actual cores and NUMA nodes that are close to these PCI devices.

Therefore gathering information about the hardware topology and about the locality of all computing, memory and I/O resources is a tedious work. Numerous non-portable and hardware-specific programming interfaces must be combined in order to get a view of the entire platform. Many HPC libraries are already able to gather some of these pieces, either directly or through dedicated tools. We are now going to look at how this discovery is actually performed in our dedicated library hwloc [6].

B. Topology Discovery on Linux

We now focus on the specific-case of topology discovery on Linux. Gathering information about the available hardware resources on this operating system may be done by reading files under the sysfs pseudo-file-system. Each logical processor (hardware thread) is described by its own directory `/sys/devices/system/cpu/cpuX` which contains numerous files. The `topology` subdirectory contains several files indicating which processor package and core contain this thread, and which other threads are its siblings. Then, the `cache` subdirectory contains one directory per cache placed between this hardware thread and the memory. Overall, the locality information about each hardware thread is scattered among about 30 different files on modern platforms with 4 levels of caches (L1i, L1d, L2 and L3). There are also specific files for each NUMA node under directory `/sys/devices/system/node/nodeX`.

The reason for using that many files is that parsing simple files containing a single piece of information is much easier for software tools. Indeed parsing one file that contains a bitmask and another file that contains a integer lets software easily read what it actually needs. As an example, on our **KNL** platform (64-core 256-thread Intel *Knights Landing* Xeon Phi 7210), hwloc has to parse about 7400 files to get full topology information. Contrariwise, the single `/proc/cpuinfo` file contains only some information about hardware threads, cores and packages, but it is not portable and much harder to parse.

Some of these files are duplicates. For instance two threads of the same core report the same list of siblings in their `topology/thread_siblings` file and the same core ID in the `topology/core_id` file. However, avoiding reading both files makes the code much more complex, and requires to trust both the hardware and the operating system. Indeed, locality information may be wrongly reported on prototypes or when the operating system does not support some new hardware architecture yet[3].

C. Overhead of Topology Discovery on Linux

We explained in the previous sections that discovering the topology is a complicated task and that it requires to read hundreds of files on Linux. We are now going to actually study the performance overhead of this step in our implementation in the latest hwloc release 1.11.5. The process of discovering the topology first consists in reading all useful files under sysfs. Then we build a structural model of the platform by assembling the contents of these files in a hierarchical tree of processor packages, cores, hardware threads, NUMA nodes and caches.

[3]https://bugzilla.kernel.org/show_bug.cgi?id=42607 shows an example of AMD processor topology fix in Linux.

On our Xeon Phi 7210 with 64 1.3GHz cores with 4 hyperthreads each, this work takes about 750ms. It is supposedly performed only once per process, during initialization. However, such an overhead is not acceptable for short processes. As a comparison, the initialization of the CUDA toolkit is often of a similar duration, because it involves PCI hardware manipulation.

Since 7400 files have to be treated on this platform, it means that reading one file takes about 100μs on average, which appears huge for such very small files (less than 100 bytes). A micro-benchmark reveals that reading sysfs files without doing anything with their content actually takes 69μs on average. The 31μs difference comes from our hwloc code listing directory contents, generating target file names, converting file contents into into its own data format, and inserting the result in the hierarchical tree of resources.

As a comparison, reading such files on a normal laptop is about 30× faster. One obvious reason for KNL being slow is that the processor frequency is 2-3×s lower than a usual processor. However the main reason lies in the Linux virtual file system layer. Manipulating files implies some global synchronization that can hardly be negligible for small files when the machine is made of 256 threads.

In the end, there are several main causes for the large overhead:

- Files under sysfs pseudo-file-system are slower than normal files (opening and closing a normal file cached in memory is 5× faster on KNL). This pseudo-file-system was not designed for performance.[4]
- Manipulating small files on large platforms seems to suffer from limited scalability. This may have slightly improved on recent kernels.[5]
- Using raw syscalls such as `open` and `read` is less convenient but slightly faster than the `fopen` and `fread` functions that hwloc currently uses.

Hence there is room for improvement, but we show in the next section that this overhead is actually more severe than it seems.

D. Overhead of Parallel Topology Discoveries

A common way to use Knights Landing is to run one MPI rank per core (64 processes on our KNL platform), each process being 4-threaded. Each of these processes may have to perform its own topology discovery, for instance for binding its threads on individual hardware threads. When 64 processes perform our topology discovery simultaneously, the overhead jumps by a factor of 41×, from 750ms up to 31s. This is a very surprising result since these processes are totally independent and the discovery is a read-only task. It means that there is a significant bottleneck in the Linux kernel implementation of concurrent reads from files in sysfs.

We ran the micro-benchmark from previous section in multiple processes (using different files for each process). Reading a single sysfs file jumps by a factor of 27× between a single process (69μs) to 64 processes simultaneously (up to 1899μs). This confirms a bottleneck in the kernel implementation. Unfortunately, the sysfs pseudo-file-system has not been designed for performance.

TABLE I
OVERHEAD OF READING A SINGLE FILE AND OF DISCOVERING THE ENTIRE MACHINE TOPOLOGY. ON EACH HOST, WE MEASURE THE TIME FOR A SINGLE OPERATION ON ONE CORE, AND FOR ALL CORES PERFORMING THE SAME OPERATION SIMULTANEOUSLY.

	64-core KNL		96-core UV	
#processes	1	64	1	96
Reading a sysfs file (microseconds)	69	1899 (27×)	8.9	4054 (456×)
Topology discovery (milliseconds)	750	31439 (41×)	145	71173 (491×)

To better understand whether the issue is KNL-specific, we ran similar experiment on our **UV** platform (SGI Altix UV with 96 cores, 12 Xeon 2.6GHz E5-4620v2 processors with 8 cores each, with a single thread per core).[6] Table I shows that individual topology discovery is not as slow as on KNL (145ms instead of 750ms), possibly because there are 2.5× less files to parse (96 hardware threads instead of 256) and because the core frequency is twice higher. However, the UV also exhibits parallel discovery non-scalability since it jumps by a factor of 60× (8.7s) when running one process per processor (12 total), and by a factor of 491× (71s) when running one process per core (96 total). This non-scalability looks worse than on KNL because they are more processes (96 instead of 64) and the UV machine is fully-loaded (one process per single-thread core instead of one process per 4-thread core on KNL).

When the overhead of topology discovery on large nodes was first noticed, the idea of parallelizing the internal of a single discovery was raised: having one thread on each core discover its local resources before the hwloc library merges their outputs as a global topology. Our above study severely challenges this idea since it looks like parallel discovery would actually be slower because of contention in sysfs reads, even before synchronization between threads is added. A better solution would rather be to have the Linux kernel expose topology information in only few larger files, but this is unlikely to happen. We will see in Section V that hwloc now actually takes care of generating one single file containing all topology information so as to make discovery much faster.

[4]The sysfs pseudo-file-system is documented at http://lxr.free-electrons.com/source/Documentation/filesystems/sysfs.txt

[5]Our KNL runs a CentOS 7.2 distribution with a 3.10.0-327.el7 kernel.

[6]It runs a RHEL 7.1 distribution with a 3.10.0-327.10.1.el7 kernel.

IV. Study of the Lifetime of Topologies

We showed in the previous section that topology discovery on Linux is expensive on large nodes and that it has a strong scalability issue when performed simultaneously by multiple processes. We will discuss in Section V how to avoid these critical scalability issue. First, we take a look here at how topology and locality information is actually used by software tools and whether the topology discovery overhead should actually impact the HPC software ecosystem.

A. Different Kinds of Reuse

A single HPC process may use the same topology multiple times, for instance when using multiple programming models. Indeed, an hybrid MPI+OpenMP application will have both the OpenMP runtime and the MPI library use topology information. The MPI initialization usually binds the entire process while the OpenMP runtime creates and binds one thread per core. Unfortunately these software layers do not currently share topology information, they will perform redundant topology discovery. We call this case **Temporal Reuse** because the exact same topology information is required by different components of the same application.

The compute node topology is also required for HPC job management when the resource manager (such as SLURM) allocates cores and memory. However, this Temporal Reuse is different. The resource manager must have the knowledge of the entire computing nodes to actually allocate some cores and memory nodes to jobs. But individual jobs may get allocated only part of a node. Their actually available resources are a restriction of the full node topology.[7] Also different jobs may have different parts of the same node. Therefore, the same node topology is reused multiple times but it is restricted when used inside actual jobs (**Restricted Temporal Reuse**).

Besides Temporal Reuse, we also introduce the concept of **Spatial Reuse** since multiple processes may have to manipulate the topology simultaneously. For instance each MPI process within a single job requires the node topology for locality-aware placement. This topology is clearly a duplicate, and we showed in Section III-D that this is an issue on large nodes. Within a cluster, there is also the need to gather the topologies of different nodes. Those nodes are different but their topologies are usually very similar since clusters are homogeneous. This is a case of **Partial Spatial Reuse** which will be discussed further in Section V-B.

B. Different Needs

Now that we have identified where and when topology is needed, we look at what topology information is actually needed. We distinguish the following possible needs in topology-aware HPC components:

- **Number of Cores:** A basic batch scheduler does not need any knowledge of compute nodes as long as jobs request resources in terms of entire 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, and OpenMP runtimes need to start one thread per core. This already raises the question of defining a *Core*: does the application want a real core or just a hardware thread? Some platforms such as Intel Xeon Phis require the use of multiple hardware threads per core for best performance. However the vast majority of users rather use a single thread per core on common platforms. Getting the knowledge of cores requires more topology information (which hardware threads are in the same core?).
- **Hierarchy of Resources:** Advanced resource allocation policies also try 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 to know the full hierarchy of resources within each compute node.
- **Full Topology Details:** Some resource attributes are needed if the application can request specific kinds of CPUs or accelerators. Attributes such as indexes or memory size 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 [7], or placement algorithms such as TreeMatch [20] that map tasks to hardware resources. When job allocation or task placement is performed using I/O locality, or when runtime libraries adapt their decisions to specific object information such as NIC addresses or cache sharing, additional details are also required, i.e. the full topology.

C. Reusing Topologies in an Optimized HPC Ecosystem

Topology information is used by multiple software components, in multiple processes on the same node, and on different nodes. Since native topology discovery on Linux does not scale well, there is a need to improve its performance. These different software components do not always need to same levels of details about the hardware topology. If topology discovery is to be factorized for performance reason, it may also be simplified when some details are unneeded.

We envision many possibilities to improve the use of topology information. Figure 3 presents a possible optimized HPC ecosystem where each compute node topology is loaded only once. The administrator would save the topology to a XML file during the boot of each compute node. Then the resource manager would retrieve all of them during its startup on the

[7] Resource managers use mechanisms such as Linux cgroups to restrict the available cores and memory to specific processes.

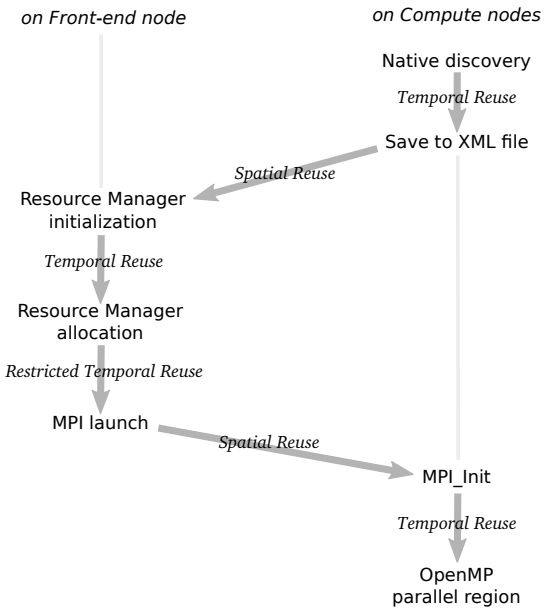

Figure 3. Reusing the output of a single native topology discovery in the resource manager on the front-end node, the MPI implementation and a runtime such as OpenMP on the compute nodes.

front-end node. If nodes are similar, a Partial Spatial Reuse can be used to avoid storing too many identical topologies.

When a new job is submitted, the resource manager looks at compute node topologies to allocate cores and memory to the job. For each compute node in the allocation, the topology is then restricted according to the allocation and passed to the MPI process launcher. The MPI implementation then launches processes on the compute nodes and passes the restricted topology so that the MPI library can bind the processes to some cores. Finally the topology is given to the OpenMP runtime so that it creates and binds one thread per core or hardware thread. This last Spatial Reuse could even be trivially implemented since the MPI and OpenMP libraries run in the same process address space.

The level of precision needed for all these steps depends on what the following users actually require. If the OpenMP runtime needs lots of details, a Full Topology should be used everywhere in the ecosystem. If the resource manager, the MPI implementation and the OpenMP runtime only need the number of cores, the Hierarchy of Resources may be enough.

V. Improving Topology Discovery Overhead

We now presents several features that help working around the overhead and non-scalability of the Linux topology discovery process.

A. XML Topologies

Our library hwloc has long had a way to export full topology information to XML and reload it later. This was initially developed as a way to manipulate the topology of remote nodes [22]. The master node of a cluster queries compute nodes for their topology (it is retrieved as XML on the network) before allocating resources and launching jobs. This is indeed already used in some resource managers and MPI implementations.

We now revisit this feature from the overhead point of view. XML has the advantage of being very easy to load since it consists of a single file (or memory buffer). Table II shows that the overhead of loading from XML barely increases with the number of simultaneous discoveries. Indeed, there is no concurrent accesses to the Linux kernel anymore besides reading the same 200kB file.

TABLE II
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.

	64-core KNL		96-core UV	
#processes	1	64	1	96
Linux native	750ms	31439ms	145ms	71173ms
XML import	8.1ms	18.1ms	3.4ms	7.4ms
XML size	167kB		220kB	

There is still a slowdown by a factor of 2.2×. We assume it is caused by some sub-optimal code in our library which causes some contention in the cache or NUMA subsystem when the entire machine is loaded. In the end, this small slowdown can be considered negligible since this topology discovery should only be performed during the software stack initialization.

In fact, loading from XML is always faster than native Linux discovery, even for a single process because there is no need to parse sysfs files anymore. XML is also useful for avoiding other slow discovery operations such as CUDA device locality probing which may take up to seconds because it involves PCI hardware queries.

XML export/import applies to the **Temporal Reuse** and **Spatial Reuse** cases described in Section IV-A. However, it fails to address other cases. For **Restricted Temporal Reuse**, we developed the ability to apply the restrictions of the current process to a topology that was loaded from XML. Therefore, as envisioned in Section IV-C, the administrator can export the full topology as XML during the boot, before each job imports and restricts it according to the resources it was actually allocated.

B. Compressing for Managing Thousands of Nodes

Managing clusters of thousands of nodes requires the front-end node to retrieve the topologies of each compute node. This is problematic for scalable resource managers that targets exascale because they try to avoid putting pressure on the network (when transferring many topologies as XML) or

on the front-end (for storing topologies in memory during allocations and/or process launch). Indeed the size of XML exported by hwloc scales in $O(P \log P)$ with P the number of cores[8]. Deeper resource 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 in future hardware.

Table II shows that current computing nodes generate XML files whose size can already reach hundreds of kilobytes. It is hard to predict whether the actual transfer of tens of thousands of such files on a future supercomputer of even larger nodes 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.

First, our tool offers the ability to store differences between topologies [22]. This sort of *Lossless Compression* is useful for clusters since most compute nodes are very similar by default. The only difference between nodes lies in network addresses, etc. Therefore a single topology can be used to represent many similar nodes. This addresses the **Partial Spatial Reuse** case in Section IV-A.

TABLE III
HWLOC TOPOLOGY DISCOVERY TIME DEPENDING ON THE SOURCE, EITHER XML IMPORT OR SYNTHETIC DESCRIPTION. 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. SYNTHETIC SIZES VARY WITH THE PRECISION OF THE DESCRIPTION (ONLY THE HIERARCHY OF RESOURCES, OR THE HIERARCHY WITH SPECIFIC DETAILS SUCH AS MEMORY SIZES AND RESOURCE INDEXES).

	64-core KNL		96-core UV	
#processes	1	64	1	96
XML	8.1ms	18.1ms	3.4ms	7.4ms
XML size	167kB		220kB	
Synthetic	3.4ms	8.1ms	1.3ms	2.6ms
Synthetic size	from 39 to 119B		from 45 to 126B	

The last useful feature in this regard is the concept of *Synthetic Topologies*: a string describing the hierarchy of computing and memory resources. Each element of the string describes the children of each resource specified by the previous element (which type and how many of them below). Table III shows that loading a topology from such a description string is even 2× faster than XML since parsing the input is much easier, as does not either suffer from strong scalability issues either when performed in parallel. Our KNL and UV platforms are respectively described as:

```
[KNL]
 Package:1 NUMA:4 L2:8 L1d:2 Core:1 PU:4

[UV]
 NUMA:12 Package:1 L3:1 L2:8 L1d:1 Core:1 PU:1
```

[8]The XML file contains one line per resource, and those lines contain bitmasks (representing the locality) whose sizes are proportional to the number of hardware threads.

This approach is a *Lossy Compression* of the topology since it removes some details about resources (for instance the processor model). However it is sufficient for the **Hierarchy of Resources** case in Section IV-B as well as **Partial Spatial Reuse** since similar node are described the same. The resource manager on the front-end may therefore describe many nodes with the same string (usually less than one hundred characters).

The synthetic description may even be further simplified to only report certain types of resources by ignoring others 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 in Section IV-B):

```
[KNL]
 NUMA:4 Core:16 PU:4

[UV]
 NUMA:12 Core:8 PU:1
```

Contrariwise, the description may also be enhanced with attributes specifying the memory sizes, cache sizes, processor indexes, etc. This may be useful to rather approach the **Full Topology Details** case:

```
[KNL]
 Package:1 NUMANode:4(memory=24GB indexes=0,1,3,2)
  L2Cache:8(size=1M) L1dCache:2(size=32K)
  Core:1 PU:4(indexes=4*64:1*4)

[UV]
 NUMANode:12(memory=256G) Package:1 L3Cache:1(size=20M)
  L2Cache:8(size=256K) L1dCache:1(size=32K)
  Core:1 PU:2(indexes=2*96:1*2)
```

Depending on the software component needs, this feature offers the ability to describe the topology of thousands of nodes with one hundred characters, with none or few details, while XML give full topology details by using much more memory.

VI. CONCLUSION AND FUTURE WORKS

The increasing complexity of modern parallel computing platforms causes the gap between theoretical and application performance to widen. Exploiting the hardware capabilities require a deep knowledge of its internals. Locality-awareness and structural modeling of the platform are keys to performance. They requires appropriate topology information to cleverly allocate resources and place tasks and data buffers according to their affinities. However, we showed in this article that discovering the hardware topology is a complicated task that has a non-negligible overhead on large nodes such as an Intel Knights Landing processor or a SGI Altix UV. Moreover it does not scale when performed by multiple processes simultaneously since it becomes up to hundreds times slower.

We explained that this performance bottleneck is actually caused by the Linux topology discovery requiring the reading of thousands of small files in the sysfs pseudo-file-system, which was not designed for performance. Then we listed the actual requirements of applications in terms of topology information, reuse and levels of precision to better understand how these limits should actually impact the software ecosystem.

Finally we showed that most cases where topology is currently discovered multiple times can actually be avoided by sharing the information between software components such as the resource manager, the MPI library or the OpenMP runtime. Our topology management library hwloc will offer ways to enable that sharing by exporting/importing topologies as XML or as synthetic description of the hierarchy of resources. These features reduce the overhead of topology management by reusing multiple times the result of the non-scalable native Linux discovery and by compressing it. The latest hwloc release 1.11.5[9] implements the basics of our proposal while new features such as restricting a XML topology to the available resources will be available in its next major release 2.0.

On-going work is now focusing on support for modeling next-generation memory architectures with different kinds of memory (high-bandwidth, non-volatile, etc.) [23]. We are also working at porting existing topology users to this new model of sharing topology information between software layers as envisioned in Section IV-C since it will be increasingly needed as computing nodes become larger on the road to exascale.

ACKNOWLEDGMENTS

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/).

REFERENCES

[1] 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.

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

[3] 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: ACTA Press, Nov. 2007, pp. 24–29.

[4] F. Song, S. Moore, and 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, Jun. 2007, pp. 97–106.

[9]hwloc 1.11.5 is available for download at http://www.open-mpi.org/projects/hwloc/

[5] 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: IEEE Computer Society, 2004, pp. 111–122.

[6] 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 International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*. Pisa, Italia: IEEE Computer Society Press, Feb. 2010, pp. 180–186.

[7] J. Hursey and 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)*. Madrid, Spain: ACM, Sep. 2013, pp. 163–168.

[8] 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*, ser. Lecture Notes in Computer Science, no. 6305. Stuttgart, Germany: Springer, Sep. 2010, pp. 265–274.

[9] S. Moreaud, B. Goglin, and R. Namyst, "Adaptive MPI Multirail Tuning for Non-Uniform Input/Output Access," in *Recent Advances in the Message Passing Interface. The 17th European MPI User's Group Meeting (EuroMPI 2010)*, ser. Lecture Notes in Computer Science, E. G. Rainer Keller and J. Dongarra, Eds., vol. 6305. Stuttgart, Germany: Springer-Verlag, Sep. 2010, pp. 239–248, best paper award.

[10] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey, "The Design of OpenMP Thread Affinity," in *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP (IWOMP 2012)*. Rome, Italy: Springer, Jun. 2012.

[11] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, Sept 2010, pp. 207–216.

[12] F. Song, S. Moore, and 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, Aug. 2009.

[13] B. Putigny, B. Goglin, and 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, Jul. 2014, pp. 943–950.

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

[15] H. Chen, W. Chen, J. Huang, B. Robert, and 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*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 353–360.

[16] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.

[17] E. Rodrigues, F. Madruga, P. Navaux, and 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*, July 2009, pp. 811–817.

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

[19] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and 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*, vol. 38, no. 5, pp. 418–439, 2010.

[20] 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. 25, no. 4, pp. 993–1002, 4 2014.

[21] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed {DAG} engine for high performance computing," *Parallel Computing*, vol. 38, no. 12, pp. 37 – 51, 2012, extensions for Next-Generation Parallel Programming Models.

[22] B. Goglin, "Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc)," in *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, Jul. 2014, pp. 74–81.

[23] B. Goglin, "Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications," in *The Second International Symposium on Memory Systems Proceedings (MEMSYS16)*. Washington, DC: ACM, Oct. 2016, pp. 30–39.