hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications

François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie
Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, Raymond
Namyst

hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications

François Broquedis[1], Jérôme Clet-Ortega[1], Stéphanie Moreaud[2], Nathalie Furmento[3]
Brice Goglin[2], Guillaume Mercier[4], Samuel Thibault[1], Raymond Namyst[1]

[1] University of Bordeaux, [2] INRIA, [3] CNRS, [4] ENSEIRB

LaBRI – 351 cours de la Libération, F-33405 Talence – France

{broquedi,jcletort,smoreaud,furmento,goglin,mercier,thibault,namyst}@labri.fr

Abstract—**The increasing numbers of cores, shared caches and memory nodes within machines introduces a complex hardware topology. High-performance computing applications now have to carefully adapt their placement and behavior according to the underlying hierarchy of hardware resources and their software affinities.**

We introduce the *Hardware Locality* (`hwloc`) software which gathers hardware information about processors, caches, memory nodes and more, and exposes it to applications and runtime systems in a abstracted and portable hierarchical manner. `hwloc` may significantly help performance by having runtime systems place their tasks or adapt their communication strategies depending on hardware affinities.

We show that `hwloc` can already be used by popular high-performance OpenMP or MPI software. Indeed, scheduling OpenMP threads according to their affinities or placing MPI processes according to their communication patterns shows interesting performance improvement thanks to `hwloc`. An optimized MPI communication strategy may also be dynamically chosen according to the location of the communicating processes in the machine and its hardware characteristics.

I. Introduction

The democratization of multicore processors leads to a significant increase in the internal complexity of machines. Several levels of caches are now shared between cores, making the hardware topology hierarchical. Meanwhile, the centralized memory *Front-Side Bus* is being abandoned and replaced with distributed memory architectures such as AMD HyperTransport and Intel QPI architectures. Such *Non Uniform Memory Access* (NUMA) hardware increases even more the hierarchical aspects of modern machines.

The increasing complexity and level of parallelism inside the computing nodes raises the question of how to schedule work so as to minimize the impact of this complexity. Indeed, achieving high-performance with *e.g.* OpenMP or MPI requires careful placement of tasks and their data buffers according to affinities [17]. Shared-memory or synchronization between tasks benefits from shared caches, while intensive memory access benefits from local memory allocations.

Exploiting modern architectures thus requires an in-depth knowledge of the underlying architecture, but also of the application behavior. We present in this paper the `hwloc` software which aims at exposing a portable abstracted view of the hardware topology to the developer. While being accessible to end-user applications, `hwloc` was designed to target high-performance runtime systems such as OpenMP or MPI libraries, so as to help them exploit the hardware thanks to a detailed knowledge of its characteristics.

The article is organized as follows. Section II presents the context and explains why affinities are important in modern HPC hardware and applications. We then introduce the `hwloc` software in Section III. Several use cases with MPI and OpenMP libraries are then presented in Section IV so as to show how `hwloc` helps them achieve better performance. Before concluding, related work is presented in Section V.

II. Affinities in HPC Applications

A. Modern Architecture Topology

Initially simple and one-leveled like Intel's Pentium series up to Pentium 3, CPU architectures became more and more complex, multi-leveled with specific caches (one per core), a global cache (one for all the cores), or even further, some partially shared caches.

For instance, Figure 1 shows how L2 caches are shared in a machine based on quad-core Xeon E5345 processor. Depending on the behavior of two threads, the best binding strategy will vary. If they share data or communicate, it may be better to bind them together on cores 0 and 4 so that they share the L2 cache. On the contrary, it may be better to bind them to cores 0 and 2 so that each of them has full access to its own L2 cache. If the threads need a lot of memory bandwidth, it may even be better to bind them to separate sockets, *e.g.* to cores 0 and 1, so that the concurrency of memory accesses can be resolved at the memory controller level, which generally permits a better aggregated bandwidth.
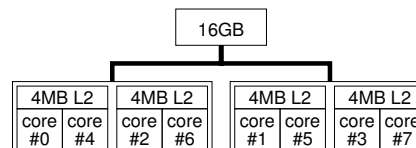

Fig. 1. Schematic view of a 2-socket quad-core Xeon E5345 host (L2 caches shared by pairs of cores).

Additionaly, *Symmetric Multithreaded* processors (also known as *HyperThreading* in Intel hardware) have a very

particular level of hierarchy, as it means sharing computation resources of a core between logical processors. For some applications with *e.g.* a lot of cache misses, this can potentially be very beneficial [4]. But for highly optimized computation kernels, this very often brings actually worse performance. In such case it is better to run only one thread on each multithreaded core.
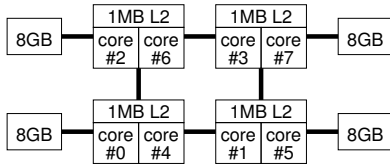

Fig. 2. Schematic view of a 4-socket dual-core OPTERON 8218 host (each socket is a NUMA node).

Moreover, to avoid memory access contention in the centralized memory *Front-Side Bus*, both AMD and now INTEL allow to transparently distribute memory among processors connected by HYPERTRANSPORT [13] or QPI links, resulting in a NUMA machine, as shown on Figure 2. The ratio between local memory accesses and remote memory accesses (the *NUMA factor*) is generally not very high (typically between 1.1 and 1.5, sometimes 3), but memory bandwidth-bound applications can get poor performance when not taking care of contention of memory accesses through the NUMA interconnection network.

In the past, cluster nodes used to have only one single-core socket, or several single-core sockets sharing central memory, or at worse several single-core sockets each having its own memory. Nowadays cluster nodes contain a complex hierarchy of cores, caches, sockets, and memory nodes. This hierarchy may vary a lot from one cluster to another due to different processor types, different numbers of cores in processors and numbers of sockets on motherboards on the market. In addition to that, while the processor numbers used by Operating Systems (as inherited from the machine's BIOS configuration) are sometimes linear in terms of proximity, they are also often interleaved as shown on Figures 1 and 2, and in such cases trivial binding strategies get miserable performance.

B. Software Affinities

Not only machine architectures have become deeply hierarchical, but applications and algorithms are also more and more complex. Some applications are actually a coupling of several very different simulation codes, *e.g.* coupling ocean and atmosphere simulation codes. Simulation codes also often use irregular hierarchical decomposition which can even be dynamic. For instance, it is a common practice to reduce the computation time *vs.* simulation accuracy dilemma by dynamically refining the simulation space only in the parts of the domain where accuracy is needed [3].

Scheduling such applications on the machines described above thus represents a real challenge. Affinities between tasks have to be carefully taken into account when scheduling them: whether they synchronize, exchange or share data, related tasks should be scheduled together, using the same caches and NUMA nodes, to re-use cached data and avoid inter-cache bounces and the NUMA factor penalty. Nevertheless, load balancing is still essential to actually take benefit of parallel machines. Combining both proximity of related tasks and distribution over the machine is thus a tricky compromise, which requires precise knowledge of the hierarchical architecture of the machine.

III. GENERIC AND PORTABLE HARDWARE TOPOLOGY ABSTRACTION

We now introduce the design and interface of `hwloc`. It aims at abstracting topology information in a portable manner so as to export it to applications and runtime systems in a convenient way.

A. Abstracting the Hardware Topology

Hardware Locality (`hwloc`) was designed from the idea that nowadays and next-generation architectures are hierarchical. Indeed, current machines consist of several processor sockets containing multiple cores composed of one or several threads. This led to representing the hardware architecture as a tree of resources. Figure 3 depicts the corresponding hierarchical view (including the knowledge of shared caches) for a dual-socket quad-core host.
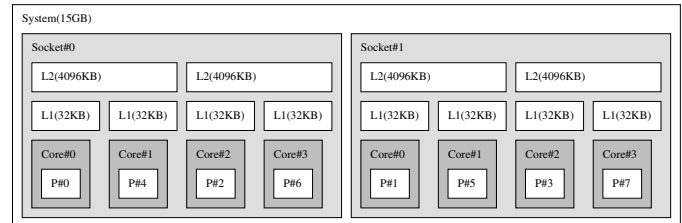

Fig. 3. Graphical output of the `lstopo` tool describing the topology of the host from Figure 1.

Moreover, the democratization of *Non Uniform Memory Access* (NUMA) raises the need to take memory placement into account when scheduling tasks. Therefore `hwloc` also includes NUMA memory nodes in its resource tree as depicted on Figure 4. In case of NUMA machines with dozens of memory nodes such as SGI ALTIX systems [2], `hwloc` can also parse the matrix of distances between nodes (reported by the operating system) so as to exhibit the hierarchical organization of these memory nodes.

Although all machines that we have seen so far are symmetric, `hwloc` was also designed with the idea that future architectures may be asymmetric (less cores in some sockets) or even heterogeneous (different processor types). Thus, the hierarchical tree is composed of generic objects containing a type (among *Node*, *Socket*, *Cache*, *Core*, and more) and various attributes such as the cache type and size, or the socket number. This design enables easy porting on future architectures thanks to no assumption being made on the presence of currently-existing object types (such as sockets or cores) or their relative depth in the tree. Indeed, there is

```
System(63GB)
  Node#0(15GB) + Socket#0
    L2(1024KB) + L1(64KB) + Core#0 + P#0
    L2(1024KB) + L1(64KB) + Core#1 + P#4
  Node#1(16GB) + Socket#1
    L2(1024KB) + L1(64KB) + Core#0 + P#1
    L2(1024KB) + L1(64KB) + Core#1 + P#5
  Node#2(16GB) + Socket#2
    L2(1024KB) + L1(64KB) + Core#0 + P#2
    L2(1024KB) + L1(64KB) + Core#1 + P#6
  Node#3(16GB) + Socket#3
    L2(1024KB) + L1(64KB) + Core#0 + P#3
    L2(1024KB) + L1(64KB) + Core#1 + P#7
```

Fig. 4. Text output of the `lstopo` tool describing the topology of the host from Figure 2.

no guarantee that computer designers will not add some new types of resources (for instance between sockets and cores), move some caches outside of processors, or change the relative depth of some components (NUMA nodes inside sockets).

B. Exposing the Topology to Applications

`hwloc` gathers information about the underlying hardware at startup. It uses operating system-specific strategies to do so: reading the `sysfs` pseudo-filesystem on LINUX, or calling some specific low-level library on AIX, DARWIN, OSF, SOLARIS or WINDOWS. It can then display to the user a graphical or textual output as depicted on Figures 3 and 4. It can also save it to an XML file so as to reload it later instead of re-gathering it from scratch, for instance if both a launcher and the actual process uses it (see Section IV-B).

The most interesting way to use `hwloc` is through its C-programming interface. All supported operating systems provide a specific API. However, these APIs are not only non-portable, they also vary significantly in their concepts: some use iterators while some do not, some manage all objects in a uniform manner while others do not, ... The `hwloc` interface not only abstracts these OS-specific interfaces into a portable API. It also tries to leverage all their advantages through both a low-level detailed interface and a high-level conceptual interface. The former lets an advanced programmer directly traverse the object tree, following pointers to parents, children, siblings, ... so as to find the relevant resource information using topology attributes such as their depth or index (see Figure 5). The latter API provides generic and higher-level helpers to find resources matching some properties (see Figure 6).

Once the application or runtime system has found the interesting objects in the topology tree, it can then retrieve information from its attributes to adapt its behavior to the underlying hardware characteristics (a cache-related example is given in Section IV-C). It is also possible to bind threads or processes to any object using the `hwloc` API. To do so, `hwloc` uses its own *Cpuset* structure containing the bitmask of allowed logical processors. Each object in the tree contains its own *Cpuset* which can be modified or combined with other objects through an extensive set of operations. Once ready for binding, a *Cpuset* can be given to a thread, process, or memory

```
/* Get depth of socket objects in the tree */
unsigned depth
  = hwloc_get_type_depth(topology, HWLOC_OBJ_SOCKET);

/* Get object #2 within depth */
hwloc_obj_t obj
  = hwloc_get_obj_by_depth(topology, depth, 2);

/* Return parent of object (could be e.g. a NUMA
   node object) */
return obj->father;
```

Fig. 5. Example of low-level interface usage to get the parent object of socket #2.

```
/* Get cores #2 and #3 */
hwloc_obj_t core2
  = hwloc_get_obj(topology, HWLOC_OBJ_CORE, 2);
hwloc_obj_t core3
  = hwloc_get_obj(topology, HWLOC_OBJ_CORE, 3);

/* Get lowest common ancestor */
hwloc_obj_t ancestor
  = hwloc_get_common_ancestor_obj(core2, core3);

/* Get first shared cache above ancestor */
hwloc_obj_t cache
  = hwloc_get_shared_cache_covering_obj(topology,
                                        ancestor);

/* Return cache size */
return cache->attr.cache.memory_kB;
```

Fig. 6. Example of high-level interface usage to retrieve the size of first cache shared between cores #2 and #3 and some other objects.

binding routine (see Figure 7). It can also be converted from/to the major existing binding libraries such as LINUX `libnuma` or the GLIBC `sched_affinity` routines to help interaction between them and `hwloc`.

IV. APPLICATIONS AND PERFORMANCE

We detail in this section how `hwloc` can be used by some existing OPENMP and MPI runtime systems. We first look at scheduling OPENMP threads and placing MPI processes depending on their software affinities and on the hardware hierarchy. Then, we show how a predefined process placement can benefit from topology information by adapting its communication strategy to the hardware affinities between processes.

```
/* Allocate an empty temporary cpuset */
hwloc_cpuset_t cpuset = hwloc_cpuset_alloc();

/* Combine the cpuset of cores #2 and #3 in cpuset */
hwloc_cpuset_orset(cpuset, core2->cpuset);
hwloc_cpuset_orset(cpuset, core3->cpuset);

/* Bind current thread */
hwloc_set_cpubind(topology, cpuset,
                  HWLOC_CPUBIND_THREAD);

/* Free the temporary cpuset */
hwloc_cpuset_free(cpuset);
```

Fig. 7. Example of Cpuset and binding interface usage, moving current thread to cores #2 or #3.

A. Affinity-aware Thread Scheduling

The OPENMP language consists of a set of compiler directives, library routines and environment variables that help the programmer with designing parallel applications. It has been originally designed for SMP architectures, and OPENMP runtime systems now have to evolve to deal with affinities on hierarchical NUMA machines.

FORESTGOMP [5] is an extension of the GCC GNU OPENMP runtime system (GOMP) that takes benefit from `hwloc` to be efficient on any kind of shared-memory architecture. It relies on the BUBBLESCHED scheduling framework to group related threads together into recursive *Bubble Structures* every time the application enters a parallel section, thus generating a tree of threads out of OPENMP applications.

BUBBLESCHED also decorates the topology provided by `hwloc` with thread queues called *Runqueues*. Each runqueue is thus attached to a different object of the architecture topology. This way, the computer architecture is modeled by a tree of runqueues on which a tree of threads can be scheduled. For instance, scheduling a thread on a socket-level runqueue means that this thread can only be executed by the corresponding cores. And each core can run any thread that is placed on the runqueue of an object containing this core.

So the problem of scheduling is only a matter of mapping a dynamic tree of threads onto a tree of runqueues. FOREST-GOMP provides several scheduling policies to fit different situations. One of them, called *Cache*, takes the topology into account to perform a thread distribution accounting for cache memory affinities. Its main goal is to schedule related threads together in a portable way, consulting the topology to determine which processing units share cache memory. It also keeps track of the last runqueue a thread was scheduled on to be able to move it back there during a new thread distribution, to benefit from cache memory reuse. When a processor idles, the *Cache* scheduler browses the topology to steal work from the most local cores to benefit from shared cache memory.

We experimented *Cache* on an implicit surface reconstruction application called MPU on a quad-socket quad-core OPTERON host (twice as many cores as depicted by Figure 2). The parallelism of this application is highly irregular and leads to the creation of a tree of more than 100,000 threads. Table I shows the results obtained by both the GOMP and the FORESTGOMP runtime systems.

TABLE I
PERFORMANCE OF BOTH GOMP AND FORESTGOMP RUNTIME SYSTEMS
EXECUTING THE MPU APPLICATION ON A QUAD-SOCKET QUAD-CORE
OPTERON COMPUTER.

Runtime	Execution time (s)	Speed-up
GOMP	5.71	4.18
FORESTGOMP (no topology)	2.79	8.52
FORESTGOMP (*Cache*)	1.71	14

We also slightly modified FORESTGOMP to ignore the architecture topology for comparison. It behaves better than the GOMP runtime system thanks to the cheap user-level thread management in BUBBLESCHED. As re-using cache memory is crucial for this kind of divide-and-conquer application, the topology-aware *Cache* scheduling policy behaves much better here. The OPENMP parallelization on this 16-core host achieves a speedup of 14 over the sequential code thanks to proper hardware affinity knowledge, while GOMP and the non-topology aware FORESTGOMP only reach 4.18 and 8.52 speedups.

B. Topology-aware Process Placement in MPI

`hwloc` can also be employed to gather hardware information in order to improve the performance of an MPI application. Indeed, the MPI processes of the application could be placed onto the various processors of the target machine according to the application's communication pattern. For instance, two processes that exchange a lot of data could be placed on two processors *close* to each other, thus sharing more cache levels and more generally, better exploiting the memory hierarchy. For two given processors this closeness can easily be measured with `hwloc` as the depth of their deepest common ancestor in the hierarchical tree data-structure representing the machine. Actually, the topology information supplied by `hwloc` can be used in two ways. The first one is to pass this information to the MPI implementation's process manager without any prior knowledge of the application's communication pattern. The second one is to use this information in order to create an adequate matching between the MPI processes locations on the architecture and the application's communication pattern.

1) Process manager support: In MPI implementations, it is the task of the *process manager* to dispatch the various MPI processes on the machine nodes of the system. When several MPI processes are located on the same node, they will most likely use shared-memory as their communication channel. Without knowledge of the application to launch, it is fairly difficult for the process manager to make an efficient dispatch within a node. The difficulty is even higher in the case of an hybrid application (for instance one using both MPI and OPENMP) because threads can be dynamically created and their placement raises lots of issues. Since `hwloc` passes information regarding memory hierarchy, we expect process managers to be able to use it in order to optimize process placement for hybrid applications. With our `hwloc` software we enhanced MPICH2's new process manager Hydra [11], so that it can bind MPI processes to specific processors based on `hwloc`-supplied information. Eventually, we plan to implement sophisticated binding schemes into `hwloc` so that Hydra can fully rely on them.

2) Communication pattern-aware placement: Another way to exploit the information supplied by `hwloc` is to compute an optimized placement of the MPI processes on the architecture's cores. Indeed any application using message-passing as programming paradigm possesses a communication pattern that can be characterized by the global amount of data exchanged between each pair of processes. This pattern can be represented by a graph with weighted edges where vertices are MPI processes ranks. For instance, Figure 8 shows the communication pattern for the LU NAS parallel benchmark.

In this example, we chose a class B, 8-processes version of this LU benchmark.
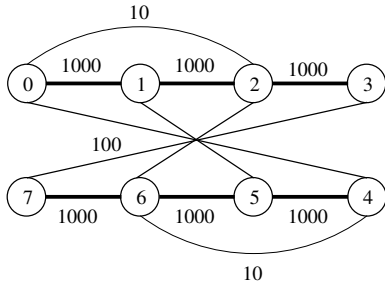

Fig. 8. NAS LU (class B, 8 processes) communication pattern representation. The coefficients on the edges represent the magnitudes in the amounts of data exchanged between processes.

Similarly, from the hierarchical tree data-structure that `hwloc` uses to represent the underlying hardware architecture, another graph can be derived. Each vertex represents a processor (or core) and the weight on the edges grows as more elements of the memory hierarchy are shared between cores. That is, the *closer* (as defined in this section's introduction) the cores are, the higher the weight shall be.

The process placement is computed from both the communication pattern's graph and the architecture's graph. We determined this *static mapping* using the SCOTCH [8] software. SCOTCH uses a dual recursive Bi-partitioning approach to solve this NP problem. For instance, Table II shows the resulting mapping between the NAS LU (class B, 8 processes) application and the OPTERON compute node architecture depicted by Figure 2.

TABLE II
RESULTING MAPPING FOR APPLICATION NAS LU.B.8 ON THE OPTERON COMPUTE NODE.

MPI_COMM_WORLD Rank	0	1	2	3	4	5	6	7
Core Number	3	7	4	0	6	2	5	1

Detailed experimental results and an analysis of several placement policies can be found in [14]. We used some of the NAS parallel benchmarks to assess the performance improvements induced by our placement method when compared to a simple one such as a *round-robin* policy. In particular, we showed that for the CG kernel (with 64 processes) running on a 8-nodes OPTERON cluster with node similar as Figure 2, we had 26% and 8% execution time improvements for classes C and D respectively as shown by Table III.

Execution time improvements have also been noticed through BT, LU, MG and SP kernels. Together with CG, they are the only ones which have an irregular communication pattern. So that they are likely to be the most significantly influenced by the placement of processes in a multicore environment. CG being the most representative of these tests, it's the only one for which we expose results.

This mapping mechanism does not rely on the MPI topologies but `hwloc` could also be used in this context in order to

TABLE III
EXECUTION TIMES (IN SECONDS) FOR NAS CG KERNEL (64 PROCESSES).

	Round-Robin	Placed	Improvement
CG (Class C)	21.16	15.6	**26%**
CG (Class D)	920.6	848.4	**8%**

improve its efficiency since MPI implementations are usually weak in this department [12].

C. Adapting the Intra-node MPI Communication to Hardware Characteristics

MPICH2 is a widely portable high-performance implementation of the Message Passing Interface (MPI) standard (version 2.1). Its communication subsystem NEMESIS [7] relies on shared memory for intra-node communications, and networking for inter-node communications. It offers highly optimized strategies to minimize small message latency. Large messages are managed by a dedicated internal API, the *Large Message Transfer* (LMT) interface, which is enabled for messages above 64 KB, and was designed to support various communication mechanisms.

KNEM, the latest LMT mechanism developed for intra-node communication, is based on a custom LINUX kernel module that was designed for large message MPI communications. It provides optimized single-copy transfers between two local processes. Memory copies can optionally be offloaded on INTEL I/OAT hardware [10], which is available on most modern INTEL servers. I/OAT offers the ability to perform efficient memory copies in the background without cache pollution thanks to a dedicated device called *DMA Engine*. It suffers from a high initialization overhead but improves performance for very large messages ($> 1\,\text{MB}$). Moreover, this copy does not involve any processor cache and thus does not pollute caches as regular memory copies do, and its performance does not depend on whether processes share caches.

Using the KNEM LMT backend raises the question of when to switch from a regular copy to an I/OAT offloaded copy. Performing some tests on the machine described by Figure 1 (4 MB L2 cache shared by 2 cores), we observed that KNEM should offload copies to I/OAT hardware when the size exceeds 1 MB. However, as shown in Figure 9, this threshold jumps to 2 MB when processes are running on two cores not sharing any cache. Running the experiment on another host with 6 MB L2 caches increased the threshold by 50%. These results led us to correlate the cache size and number of processes using it with the observed threshold:

$$\text{DMA}_{\text{min}} = \frac{\text{Cache Size}}{2 \times \text{Processes Using The Cache}}$$

Indeed, the copying process first fetches the data in its cache when reading from the sender pages into its processor registers. Then it stores the data back in its receive buffer and thus fills the cache again. Therefore, to avoid many cacheline flushes, the cache must be at least twice as large as the message being received. Larger messages should preferably be

transferred with I/OAT copy offload since a process copying data with I/OAT copy offload does not consume any cache line.
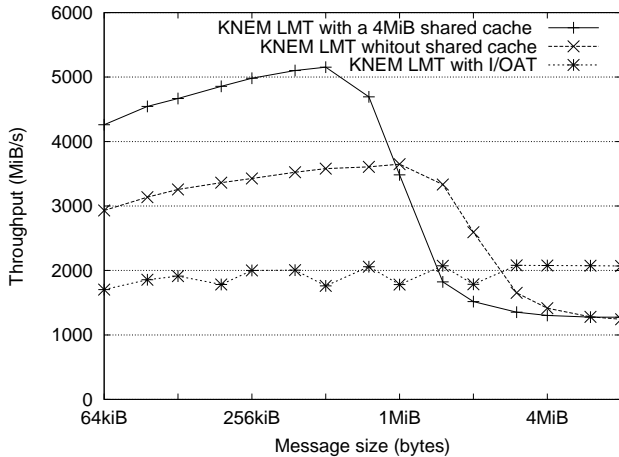

Fig. 9. IMB Pingpong throughput between 2 processes using LMT strategies.

Real applications using large intra-node MPI messages benefit from the combined use of KNEM with or without I/OAT-offload. Indeed, the NAS Parallel Benchmarks IS and FT, known to use large messages, show 25% and 10% improvements over the usual two-copy based user-space shared-memory implementation [6].

Achieving optimal performance requires the dynamic configuration of MPICH2 and KNEM I/OAT-offload threshold depending on the location of processes in the hardware topology, and the sharing and size of caches between these locations. Once the processes have been placed by the process manager (for instance using a strategy explained in Section IV-B), the KNEM LMT gathers the corresponding topology objects and retrieves the size of shared caches between them (as shown in Figure 6). Thanks to this knowledge, `hwloc` enables KNEM adapting the communication strategy for each transfer thanks to dynamically computed I/OAT-offload threshold depending on shared cache characteristics and process binding.

V. RELATED WORK

Binding each computing task to its own dedicated processor is fairly common nowadays. However, the reason mainly invoked for doing so is not related to hardware topology. Indeed, if a task is not bound, the operating system may migrate it to another CPU whenever a daemon wakes up and causes a load imbalance. Binding each task prevents migration from happening and thus keeps processor caches hot. MPI libraries such as OPEN MPI [9] and MPICH2 often run a single thread or process per core, binding them improves performance and reproducibility. OPENMP runtime systems such as GOMP [1] or the INTEL compiler [18] also offer the optional ability to bind threads. However, these high-performance computing software do not actually look at the application behavior before deciding where to bind each task.

Indeed, neither OPENMP nor MPI currently offers any way to specify affinities between tasks or between tasks and data.

The PLPA library (*Portable Linux Processor Affinity*, [16]) brought some knowledge of hardware topology to the OPEN MPI project. It offers the ability to bind MPI processes on specific sockets, cores, or hardware threads. OPEN MPI, MVAPICH2 [15] and MPICH2 (through Hydra) use these features to scatter processes across the available logical computing resources, or let the user manually bind if needed. However, we feel that the offered interface is too specific. For instance, shared caches are ignored while some sockets have shared caches between only some of their cores (see Section II-A). Such a piece of information may dramatically improve performance since tasks that share memory or synchronize often will be significantly faster if bound to different cores inside the same cache. PLPA also does not offer any memory node abstraction while most modern servers are NUMA. `hwloc` also offers a more generic representation of the hierarchic topology as explained at the end of Section III-A. It does not focus on the current concepts of sockets, cores, and threads since these might not be enough or might even become obsolete in next generation architectures. Finally, `hwloc` also has the advantage of working on multiple operating systems and not only on LINUX as PLPA does.

Still, PLPA objectives were similar to ours. For this reason, a collaboration between the research teams was started so as to merge the efforts into a single project[1] To help the port of existing PLPA users, the PLPA interface will soon be re-implemented on top of `hwloc`. However, it is expected that these users end up switching to the native `hwloc` interface since it offers more powerful features and knowledge of the hardware.

VI. CONCLUSION AND FUTURE WORK

The emergence of multicore processors with shared caches and non uniform memory access causes the hardware topology to become increasingly complex. Applications have to be carefully placed on these machines so that affinities are efficiently handled by the hardware, while still maintaining the ability to load-balance tasks across the machine.

We introduced the design of *Hardware Locality* (`hwloc`)[2] which aims at gathering a detailed knowledge of the hardware topology, including threads, cores, shared caches, sockets and NUMA nodes, and at exposing it in a generic and portable manner. It abstracts the machine characteristics as a hierarchical tree of resources that applications and runtime systems can traverse to retrieve hardware information.

By using `hwloc`, high-performance computing software are now able to carefully place tasks according to hardware affinities. We presented an example of OPENMP thread scheduling that relies on cache information to improve the overall efficiency thanks to better data locality. We then detailed a MPI placement strategy that combines `hwloc` abstractions and

---

[1]Before merging with PLPA, our early work was actually known as `libtopology`.

[2]Available for download at http://runtime.bordeaux.inria.fr/hwloc.

the knowledge of communication patterns to place processes according to their affinities. Finally, we also showed that `hwloc` may be used at runtime to adapt the intra-node MPI communication strategy depending on process placement and hardware cache characteristics.

We are now looking at improving other HPC codepaths such as OPENMP barriers that could map their hierarchical behavior to the hierarchical hardware topology. The MPI process launching strategy is also being improved so as to stop assuming that all nodes are identical and that all cores are available.

Then we envision the addition of I/O device knowledge to `hwloc` so as to expose accelerators or GPGPUs as new heterogeneous computing resources in the topology tree. The idea of combining multiple machines inside a single topology tree will also be studied since it may simplify the overall placement of MPI processes within a cluster.

REFERENCES

[1] GOMP - An OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp/.
[2] SGI Altix 4700: Delivering New Levels of Performance and Flexibility. http://www.sgi.com/products/servers/altix/4000/.
[3] Marsha J. Berger and Joseph E. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, pages 484–512, 1984.
[4] Lars Ailo Bongo, Brian Vinter, Otto J. Anshus, Tore Larsen, and John Markus Bjørndalen. Using overdecomposition to overlap communication latencies with computation and take advantage of smt processors. In *Proceedings of the 2006 International Conference on Parallel Processing Workshops*, pages 239–247, Columbus, OH, August 2006.
[5] François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *OpenMP in a New Era of Parallelism, 4th International Workshop on OpenMP, IWOMP 2008*, volume 5004 of *Lecture Notes in Computer Science*, pages 170–180, West Lafayette, IN, May 2008. Springer.
[6] Darius Buntinas, Brice Goglin, Dave Goodell, Guillaume Mercier, and Stephanie 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, September 2009. IEEE Computer Society Press. To appear.
[7] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and Evaluation of Nemesis: a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *Proc. 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, Singapore, May 2006. Held in conjunction with IEEE Computer Society and ACM.
[8] François Pellegrini. SCOTCH and LIBSCOTCH 5.1 *User's Guide*. ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, August 2008. http://www.labri.fr/perso/pelegrin/scotch/.
[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
[10] Andrew Grover and Christopher Leech. Accelerating Network Receive Processing (Intel I/O Acceleration Technology). In *Proceedings of the Linux Symposium*, pages 281–288, Ottawa, Canada, July 2005.
[11] The Hydra process manager. http://wiki.mcs.anl.gov/mpich2/index.php/Hydra_Process_Management_Framework.
[12] Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing 02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
[13] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March 2003.
[14] Guillaume Mercier and Jérôme Clet-Ortega. Towards an efficient process placement policy for mpi applications in multicore environments. In *EuroPVM/MPI 2009*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115, Espoo, Finland, September 2009. Springer.
[15] Network-Based Computing Lab, The Ohio State University. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.
[16] Portable Linux Processor Affinity (PLPA). http://www.open-mpi.org/projects/plpa.
[17] Rolf Rabenseifner, Georg Hager and Gabriele Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, pages 427–436, Weimar, Germany, February 2009.
[18] Xinmin Tian, Milind Girkar, Sanjiv Shah, Douglas Armstrong, Ernesto Su, and Paul Petersen. Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures. In *Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 47–55, April 2003.