Memory Footprint of Locality Information on Many-Core Platforms

Brice Goglin

▶ **To cite this version:**

Brice Goglin. Memory Footprint of Locality Information on Many-Core Platforms. 6th Workshop on Runtime and Operating Systems for the Many-core Era (ROME 2018), held in conjunction with IPDPS, May 2018, Vancouver, BC, Canada. pp.10, 10.1109/IPDPSW.2018.00201 . hal-01644087

HAL Id: hal-01644087
https://inria.hal.science/hal-01644087

Submitted on 13 Mar 2018

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

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

Memory Footprint of Locality Information on Many-Core Platforms

Brice Goglin
Inria Bordeaux – Sud-Ouest, LaBRI, Univ. Bordeaux, France
Email: Brice.Goglin@inria.fr

***Abstract*—Exploiting the power of HPC platforms requires knowledge of their increasingly complex hardware topologies. Multiple components of the software stack, for instance MPI implementations or OpenMP runtimes, now perform their own topology discovery to find out the available cores and memory, and to better place tasks based on their affinities.**

We study in this article the impact of this topology discovery in terms of memory footprint. Storing locality information wastes an amount of physical memory that is becoming an issue on many-core platforms on the road to exascale.

We demonstrate that this information may be factorized between processes by using a shared-memory region. Our analysis of the physical and virtual memories in supercomputing architectures shows that this shared region can be mapped at the same virtual address in all processes, hence dramatically simplifying the software implementation.

Our implementation in hwloc and Open MPI shows a memory footprint that does not increase with the number of MPI ranks per node anymore. Moreover the job launch time is decreased by more than a factor of 2 on an Intel Knights Landing Xeon Phi and on a 96-core NUMA platform.

I. Introduction

Future exascale platforms are expected to be made of tens or hundreds of thousands of nodes. The internals of these nodes are increasingly complex, with tens of cores and deep memory hierarchies. Many modern HPC software projects, from resource managers to runtimes, rely on knowledge of the hardware topology for allocating resources and placing tasks and data buffers based on affinity. They often use third-party tools such as hwloc [1] which take care of gathering hardware information, exposing it in an portable way, and applying binding policies.

With nodes containing tens of resources (cores, hardware threads, packages, *etc.*) and deep memory hierarchies (levels of cache, NUMA nodes, heterogeneous and/or non-volatile memories, *etc.*), the amount of locality information is growing fast. Multiple components of HPC software stacks store similar information without sharing it with others. And multiple processes also duplicate it, causing the memory footprint of locality information to become significant. However, the amount of physical memory per core in HPC platform is not huge. It is expected to remain around one gigabyte per core for exascale platforms. Wasting megabytes of memory for locality information in several components of the HPC stacks therefore becomes an issue.

We propose to share locality information between MPI processes on each node by exposing the hwloc topology in a shared-memory region. The need for integration in the hwloc API without breaking existing codes leads us to enforce the mapping of this region at the same virtual address in all processes. We perform an in-depth analysis of the physical and virtual memories of current supercomputing architectures. It shows that most of the virtual memory is available in HPC processes, and that our proposed simple heuristic for finding an appropriate virtual address has a high probability of success. Experimentation on an 64-core Xeon Phi and on a 96-core NUMA host confirms that the memory footprint is significantly reduced, while the launch time is decreased by more than a factor of 2.

The remaining of this paper is organized as follows: Section II explains why locality is used in HPC software and how. Section III describes the way locality information is managed and how it may be exchanged between components. Our proposal to store topology in a shared-memory region is then detailed in Section IV. Finally Section V presents the evaluation of the implementation inside Open MPI.

II. Locality Information

We present in this section how locality information is used by high-performance computing software.

A. Why Locality is Important

Locality became a key criteria for performance optimization since the advent of NUMA architectures and multicore processors more than ten years ago. The physical distance between hardware components, cores and memory banks, varies significantly because HPC platforms are made of multiple multicore processor packages with their own local memory. Moreover a hierarchy of private and shared caches inside processors increase the observed performance difference since communicating through a shared cache is usually faster.

Modern HPC runtimes therefore gather topology information to find-out how cores are organized in hardware: inside processors, with respect to caches and memory banks, and with respect to I/O devices such as GPUs or InfiniBand HCAs. This information is used for placing tasks (usually threads or processes) in an affinity-aware way: matching inter-task affinities (tasks that synchronize/communicate a lot benefit from shorter distance between them) [2], [3]; placing tasks and their target data buffers together (on NUMA nodes close to cores and/or I/O devices that access them) [4].

Topology knowledge is often delegated to third party libraries such as hwloc [1] which take care of exposing topology information in a portable and abstracted manner.

B. How Locality Information is used

The first component that usually looks at hardware topology on a parallel platform is the resource manager (*e.g.* Slurm). It must be aware of the number of cores and the amount of memory on each node for allocating resources to jobs.

User processes are actually started on compute nodes either through resource manager tools (*e.g.* `srun`) or a specific process launcher (*e.g.* `mpiexec`). For MPI jobs, a daemon is first created on each node (*e.g.* `slurmd` or `orted`). It takes care of launching a process for each MPI rank. Both the daemon and MPI library in rank processes must have knowledge of the hardware topology since they bind processes to specific cores.

Inside processes, either MPI ranks or not, runtimes (such as OpenMP) or task-based schedulers (such as StarPU [5]) require knowledge of the hardware for selecting which core or accelerator to execute a task on, and which NUMA node to allocate memory on.

Finally the computing codes executed by these tasks or threads involve kernels that may be tuned based on cache or micro-architecture characteristics. Similarly, communication libraries may look at I/O locality for selecting which NIC to use and which communication strategy to select.

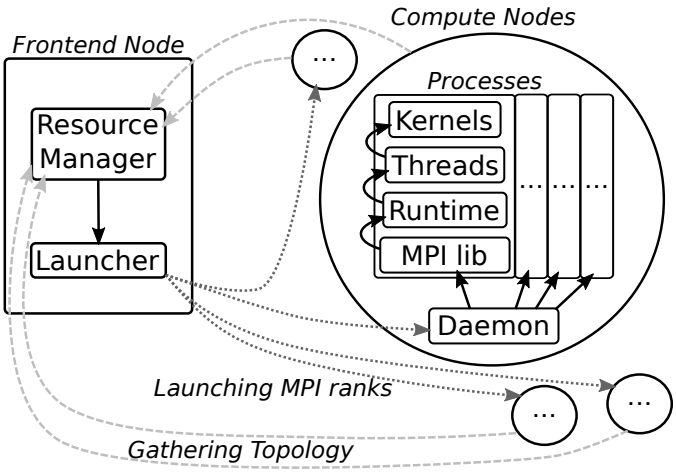

Fig. 1. Example of topology information exchange in a HPC software stack. The topology of compute nodes is gathered by the resource manager (light grey dashed arrows). After allocating nodes and cores using this information, it passes it to the job launcher on the frontend node which invokes daemons on compute nodes for spawning actual user processes (dark grey dotted arrows). Several components inside each process may then use topology information for binding the process, launching and binding some threads, tuning kernels, *etc.*

Hence, as summarized in Figure 1, many components in the HPC software stack require information about the hardware topology. In current implementations, most components in each process gather topology information on their own. We proposed some ideas to exchange this information between them to avoid multiple expensive discoveries [6]. However, it requires deep changes before these standards and implementations can actually interoperate. Moreover, it does not address to duplicated memory occupancy since each component still has its own copy of the topology information. We are now going to focus on this issue.

III. Managing Locality Information at Scale

Future Exascale platforms are expected to contain hundreds or thousands of cores per node. However, the memory may not be able to sustain this growth rate. The amount of memory per core is indeed not going to increase according to exascale reports [7], [8].

The Top500 list[1] reveals that the current most powerful supercomputers have only about 1GB of physical memory per CPU core. For instance Piz Daint (ranked #3 in November 2017) contains 340TB of memory for 361 760 cores.[2] Sunway TaihuLight (ranked #1) even offers only 32GB for each 260-core SW26010 processor.[3]

This means that developers of parallel codes for exascale platforms must be careful not only about scaling algorithms to millions of cores, but also about making sure each task can operate on a small dataset. Using large datasets would cause either memory exhaustion requiring I/Os to the storage subsystem, or exchanges between cores on the network. Such accesses severely limit the overall performance of parallel applications. It is therefore necessary to properly adapt the application datasets to the available physical memory. Hence reducing the memory footprint of other components is important.

A. Memory Footprint

The limited amount of physical memory per core puts a severe pressure on application datasets. Hence libraries used by the processes should try to reduce their own memory footprint so as to let as much memory available to the actual application as possible. We are going to discuss the memory footprint of locality information in the *de facto* standard topology management library, hwloc. [1]

1) hwloc Object Footprint: Figure 2 shows that hwloc models the platform as a hierarchical tree of objects based on inclusion: a Machine contains some processor Packages, which contain Cores, and hardware threads. Caches and NUMA nodes also appear in that hierarchy. Each of these hwloc objects weights about 1kB: it contains several generic attributes (kind, index, *etc.*), several pointers (to parents, children and neighbors), some type-specific attributes (cache associativity, memory size, *etc.*), and several bitmaps listing its local hardware threads and local NUMA nodes (the bitmap size increases with the platform size).

There is some ongoing work towards reducing the per-object memory footprint in the upcoming hwloc 2.0. However, we

[1] https://www.top500.org/lists/2017/11/

[2] We only consider host CPU memory. GPUs do not have the same notion of core and they do not have large amount of memory.

[3] The processor contains 4 general purpose cores and 256 specialized cores.

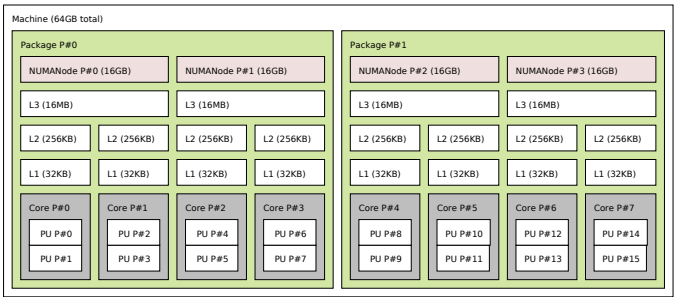

Fig. 2. Representation of the hwloc hierarchical tree with the lstopo tool. This fictive host is made of 2 processor packages. Each half of package contains one NUMA node, one shared L3 cache, and 2 cores. Each core contains two hyper-threads (PUs) and private L1 and L2 caches.

can expect neither large nor scalable improvements since the overall memory footprint is linear with the number of objects.[4] More intrusive micro-optimizations (such as removing generic attributes when they are irrelevant, for instance the index of processor Packages) are possible but they would make the hwloc programming interface much harder to use due to many specific cases. Also changing the hwloc programming interface would cause hundreds of existing user codes to require updates.

2) Number of objects: The number of objects used by hwloc to describe a node basically corresponds to the number of hardware resources. On a 64-core Intel Knights Landing Xeon Phi (later called KNL), there are 256 objects for hardware threads (4 per core), 64 for cores, 64 for L1 data caches, 32 for L2 caches (shared by dual-core tiles) and 1 for processor package. Depending on the KNL configuration (MCDRAM in cache, flat or hybrid mode, and mesh in quadrant, alltoall or SubNUMA-cluster mode), there may also exist up to 8 NUMA node objects and/or 4 additional caches. This means at least 400 objects (and even up to 500 if instruction caches or I/O locality are also needed by applications).

With some additional topology-wide information, we observe between 600kB and 700kB of memory to store each hwloc topology on KNL.

Several users already reported that this is an issue on their platform because they use one process per core (up to 72 on KNL), or even one process per hardware thread (up to 288 on KNL). Although using that many MPI processes per node may seem sub-optimal, it is actually required for legacy applications that could not be ported yet to a more modern programming model such as MPI + OpenMP. This issue will get more severe on larger many-core nodes in upcoming exascale platforms.

B. Filtering Topology Information

We now explain how horizontal or vertical filtering might help reduce the number of objects.

[4]It is expected that hwloc 2.0 per-object footprint will be about 30% lower than in hwloc 1.11.

1) Horizontal Filtering of Available Resources: The resource manager may allocate only part of a node to a job. Resources outside of the allocation should not be needed by the application. This *Horizontal Filtering* would almost divide the number of object by 4 if only a quarter of the node is allocated, as depicted on Figure 3.

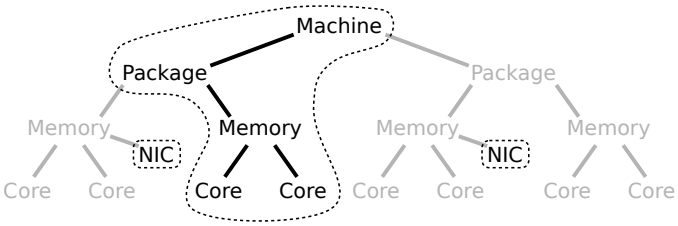

Fig. 3. Horizontal filtering of a topology. If only 2 cores of a 8-core machine were allocated to a job, most objects can be removed from the topology. However, if the process uses network interfaces to communicate, it may want to decide which NIC to use based on their locality. Hence removing Package and Memory must be done without hiding the fact that one NIC is closer to the available cores than the other NIC.

However, this filtering is not trivial because some non-allocated resources might still be useful when taking locality-aware decision. For instance, selecting which NIC to use for MPI communication involves looking at distances between NUMA memory banks and NICs, even if some of these banks are not available to the current process.

Anyway we focus on exascale platforms where large jobs will get allocated on lots of entire nodes. Allocating only part of nodes is usually reserved to very small jobs. Hence we do not expect much improvement of the memory footprint at scale thanks to horizontal filtering.

2) Vertical Filtering of Useful Resources: It is also possible to perform *Vertical Filtering* by asking hwloc to ignore some kinds of objects. If the application does not do any kind of tuning based on cache size, cache objects could be removed as depicted on Figure 4.

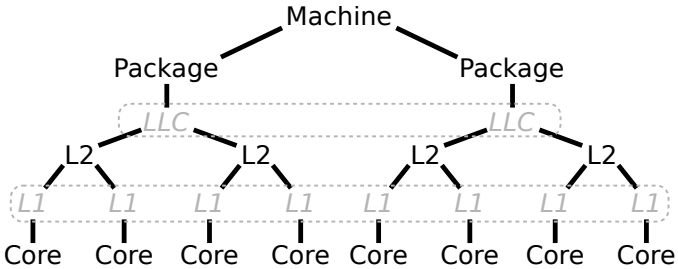

Fig. 4. Vertical filtering of a topology. If cache information is not needed by the application, L1 and LLC objects can be removed from the tree. However, L2 cannot be removed because they bring information about the hierarchy of cores: cores are organized by pairs, instead of being 4 random cores in each package.

However, aside of the cache-specific characteristics, some cache objects also bring information about the hierarchy of cores within processor packages. On KNL, cores are organized as dual-core tiles with a shared L2. Hence communication between cores inside a tile is likely faster than communication

between tiles. This locality information requires to keep L2 objects in the hierarchy for better affinity knowledge (*there are 32 dual-core tiles* instead of *there are of 64 cores*). Contrariwise, KNL L1 caches are private per core. Hence ignoring them does not remove any information about the hierarchy of cores (this removes about 15% of objects).

A way to perform this filtering, with or without ignoring the hierarchy information brought by some objects, is already offered by hwloc. In the end, one has to enable such filtering depending on his needs for information about the hierarchy of cores, memory, caches, *etc.* A single component may easily filter-out what it does not need. However, sharing topology information between multiple components makes filtering difficult: the component that is responsible for discovering and sharing topology information, should know what other components need (*e.g.* a parallel BLAS may need cache information). In practice, they keep everything enabled in case anybody ever needs it.

To summarize, there are ways to reduce the topology footprint by filtering unneeded objects out. However, this reduction hinders the ability to share topology information between components inside the same process, or between processes on the same node, because they may have different needs. We expect much larger improvements thanks to rather sharing the topology between them without filtering.

C. Opportunities for Sharing

As explained in Section II-B, many components in the HPC software stack may need topology information: the resource manager for allocating cores and memory, the launcher and MPI library for binding processes, the runtime for spawning and binding threads, and kernels for tuning implementations for the micro-architecture.

There are several ways to implement this:

Native Discovery: Each component may perform its own discovery of the hardware. One advantage is that they precisely filter what they need. However, the major drawback is the overhead of discovery which does not scale to large nodes [6]. Hence it is better to have a single topology discovery and exchange the result between components.

Exchanging through XML: hwloc offers easy ways to exchange topology information through XML files or buffers. It avoids the overhead of native discovery, but it still requires to instantiate the topology tree in memory, which causes a large memory footprint as explained in Section III-A. Therefore it is better to only instantiate the tree once.

Centralizing Topology Queries: If the topology information is centralized, all components could issues queries to a dedicated server. Inside a single process, this can still be performed using the existing hwloc API. However, exposing the long list of features from the hwloc API to other processes requires the design of a forwarding layer. Users would have to switch from the hwloc API to this new forwarding API, which would also be slower than existing function calls.

Topology in Shared-Memory: Instead of centralizing queries, the topology can be centralized in physical memory and shared in virtual memory. Once the topology is mapped in the virtual memory of all processes, they can use the existing hwloc API without having to instantiate multiple topologies in physical memory.

The main drawback of using shared-memory topology information is that none of the users can modify this information. Modifying would for instance include attaching custom information to hwloc objects. Fortunately, this may still be replaced with a lookup table for finding that information based on the object type and number (*e.g.* Core #5).

IV. Sharing Locality Information at Runtime

We now explain how to expose the hwloc topology information in a shared-memory region between a *master* process (*e.g.* the resource manager or MPI launcher daemon on the compute node) and many *slaves* (*e.g.* compute processes running MPI ranks).

A. Storing Topologies in Shared Memory

We implemented in hwloc a new API for storing a topology in a shared-memory region and memory-mapping it other processes, as depicted in Figure 5.

On the master side, `hwloc_shmem_topology_write()` uses a callback to replace `malloc()` with allocations in the target shared memory region (a memory-mapped shared file). Instead of modifying all allocations in the hwloc library, this callback may only be enabled when *duplicating* a topology. Hence the master process first creates a normal topology and then duplicates it as a *shmem clone* in a shared-memory region.

This *shmem clone* on Figure 5 is self-contained, except for function pointers. Indeed each topology contains pointers to operating-system-specific binding functions in the hwloc library. Given that the location of libraries in virtual memory varies from one process to another[5], each process still uses its own private function pointers.

On the slave side, `hwloc_shmem_topology_adopt()` gets the *shmem clone* and lets the local hwloc library use it. The main `hwloc_topology` structure (about 700 bytes) is duplicated locally so as to update function pointers to the local hwloc library as explained above. However, all objects (the remaining 600-700kB footprint on KNL) remain in the shared memory, they do not have to be duplicated locally.

One way to avoid duplication of this 700-byte structure would be to map the *shmem clone* with `MAP_PRIVATE` so that local changes are not propagated to other processes (*Copy-on-Write*). However, slaves are not supposed to modify anything but that structure. Hence we would have to keep the main `hwloc_topology` structure in a private writable mapping while everything else would be in a public read-only mapping. Moreover, each slave would allocate one physical page (4 kB on x86) to store its locally modified private copy. This is higher than our current 700-byte duplicate which only requires a new

[5]On Linux, the interpreter `ld.so` and the kernel are in charge of selecting library locations in virtual memory. They apply some randomization for security reasons.

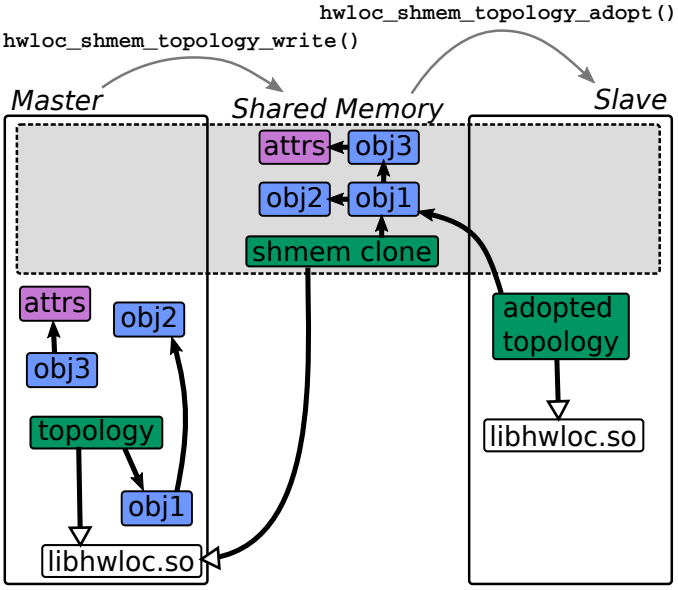

Fig. 5. Duplicating and adopting a hwloc topology across a shared memory region. Arrows ending with an empty triangle are function pointers while other arrows are normal pointers. The slave process cannot use the *shmem clone* topology because it points to functions in the hwloc library (`libhwloc.so`) in the other process. Hence the need for a new *adopted* topology that points to *shmem clone* objects and to functions in the local copy of the hwloc library.

physical page in 700/4k=17% of the cases. Anyway, this waste is negligible against the problem tackled by our work, but the underlying implementation can still be modified later if ever needed.

B. Partial Sharing

Sharing topology information between processes on the same node obviously exposes the exact same information. However, there is one point that may differ from one process to another: the list of available resources. Indeed the batch scheduler may allocate different cores or memories to different jobs. Technologies such as Linux cgroups are used to enforce this partitioning.

hwloc already supports filtering-out unavailable resources as explained in Section III-B1. However, a single server cannot expose the topology to different slaves in different cgroups using the same shared memory region. Hence this partial sharing requires slaves to manually ignore resources that are not available to their jobs when consulting the shared topology. Another solution consists in having one master per job, exposing the filtered topology only to processes in that job, instead of one master per node.

We do not expect the memory footprint to be critical on small jobs that do not use entire nodes. Hence both solutions look acceptable when really needed.

C. Sharing hwloc Pointers

One major hurdle in this implementation comes with pointers inside the shared-memory region (black arrows on Figure 5). The hwloc API was designed with a tree made of many public pointers between objects (parent, children, siblings, cousins, *etc.*) and between objects and attributes (strings, dynamically-extendable bitmaps, *etc.*). These pointers must be valid in all slaves that consult the topology.

One way to solve this would be to remember the *offset* that was added to the address when mapping in each process, and add that offset of each pointer before dereferencing it. Unfortunately this would require to change the existing hwloc API and break many existing applications.[6] Hence the master and the slaves must map the shared region at the same virtual address in their own address spaces so that pointers remain valid in every context.

As explained in Section IV-A, this does not apply to function pointers which are still mapped at different addresses in all slaves. We could have mapped yet another instance of the hwloc library at the same address in each process but it is not worth the 700-byte duplication avoidance.

The major argument in favor of our proposal to enforce mapping at the same address is the vast amount of virtual memory available on current architectures. While physical memory is not expected to increase beyond gigabytes per core in exascale platforms (see Section III), most HPC architectures already support about 47 bits of virtual address[7], which means 128TB of virtual memory per process.[8] This limit will even be bumped to 64PB on future x86 processors thanks to the *la57* processor extension.

TABLE I
PERCENTAGE OF FREE VIRTUAL MEMORY INSIDE PROCESS ADDRESS SPACE DEPENDING ON THE NODE CONFIGURATION AND ON HOW MANY PROCESSES ARE SPAWNED PER NODE.

Configuration	Physical Memory per Process	Virtual Memory per Process	% of Free Virtual Mem.
Trinity node (64-core KNL, 96GB)			
1 process/core	1.41GB	128TB	99.9988%
1 process/node	96GB	128TB	99.925%
KNL worst case (72-core KNL, 384GB)			
1 process/core	6GB	128TB	99.9953%
1 process/node	384GB	128TB	99.7%
Summit node (2× 22-core POWER9, 512GB)			
1 process/core	11.6GB	64TB	99.981%
1 process/node	512GB	64TB	99.2%

Table I reports some examples for the KNL nodes of the Trinity supercomputer (ranked #7 in Top500), fictive nodes corresponding to the worst KNL case (fewer cores but more physical memory), and nodes of the upcoming Summit pre-exascale platform (fat nodes with lots of memory to serve 6 GPUs). *1 process/core* is our ideal case where physical memory is split between processes, hence reducing the amount available to each process. *1 process/node* is our worst case where all physical memory is used by a single process. In practice, the number of processes is often between these

[6]The hwloc API was designed in 2010 without such issues in mind. Making pointers public simplified the API but caused this shared-memory issue years later.

[7]47 for Intel and AMD; 48 for ARMv8; only 46 on POWER because of the current Linux kernels.

[8]x86 architectures actually supports 256TB (48 bits virtual addresses) but the Linux kernel reserves half for kernel space.

cases, and some memory is shared between processes (at least programs and binaries).

In all cases considered here, virtual memory is more than 100× larger than physical memory. If one process was launched on each hardware thread (4× more), this percentage would be even higher. In the end, it shows that the vast majority of the virtual address space is free (far more than 99%), which should give a high probability of finding a virtual region that is available.

D. Mapping at the same Virtual Address

Mapping the shared-memory region at the same virtual address in the master and in the slave processes requires to find a virtual area that is free in all their address spaces. Unfortunately the layout of the processes when they will map that region is necessarily not known when the master process creates the region. Even worse, MPI dynamic spawning of process[9] prevents us from knowing the exact list of processes that will ever map the region during the lifetime of the job. Hence we use a heuristic for finding a good virtual memory area that *should* be available in most processes. If one process fails, it will fallback to the existing XML-base import (this case never happened in our experiments).

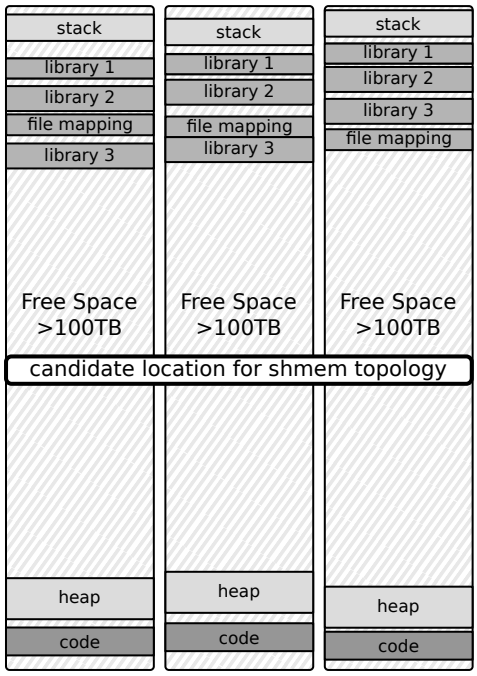

Fig. 6. Virtual address space layout of different processes on Linux on 64bits x86 architectures. Address space randomization makes actual locations vary slightly between processes but the general layout is the same.

Fortunately, the way virtual memory areas are organized on Linux is somehow predictable as depicted by Figure 6. On x86, the code is placed at the bottom of the address space, followed by the heap which grows up. The stack starts from the top and grows down. In the middle, there is a giant free space where custom memory mappings are added: shared libraries, file mappings and anonymous mappings. The Linux kernel places these mappings close to each other, either from top to bottom or from bottom to top depending on the architecture.

The exact layout may vary from one process to another because of address space randomization [9] but the general organization is the same. Hence all processes running on the same node have a large contiguous free area located at almost the exact same location. We verified this organization on multiple Linux kernels and distributions on different platforms. For instance on a KNL platform (64bits x86) running RHEL7[10] with randomization enabled[11], the location only varies by 1GB for the stack and 1TB for the libraries. However, the entire address space is 128TB wide and the free hole is in the order of 120TB and randomization is negligible.

Processes could use this free space by allocating large buffers or mapping large files. Fortunately, as explained in Section IV-C, the virtual memory per process is much larger than the physical memory per core in HPC platforms (about 1GB). Hence HPC applications will not be able to actually load terabytes of data in their virtual memory, leaving the 120TB free hole almost untouched.

Therefore we use a very simple heuristic that looks at the current process address space[12], finds the larger hole (usually larger than 100TB), and uses its middle for mapping the shared-memory region.[13] Since our mapping is small (about 700kB on KNL) and since most processes have a similar 100TB free hole, this mapping should fall in the free hole of all slave processes, as shown on Figure 6.

E. Implementation in Open MPI

After implementing the new hwloc API for sharing and adopting topologies between processes, we implemented the above heuristic in Open MPI.

Open MPI processes are launched on compute nodes using the `orted` daemon. This daemon is the *master* in our model: it is in charge of finding the appropriate shared-memory virtual address for the current platform using our heuristic. It creates a shared-memory file mapped at that address and duplicates the hwloc topology in it using `hwloc_shmem_topology_write()`.

Compute processes, *i.e.* MPI ranks, are *slaves* in our model. They retrieve the name of the file and the target virtual address from the daemon as part of the existing PMIx protocol [10]. Finally, they just map that file using `hwloc_shmem_topology_adopt()`.

If this mapping fails (for instance because the selected virtual address is not free in the current process), an error is returned. The Open MPI implementation then falls back to the old strategy which consists in requesting a XML dump of

[9] with `MPI_Comm_spawn()`.

[10] Linux kernel `3.10.0-327.36.3.el7.x86_64`.

[11] `/proc/sys/kernel/randomize_va_space` set to 2 to randomize library and stack location.

[12] by reading `/proc/self/maps`.

[13] We align the middle address to a multiple of the second page-table level for optimizing TLB use.

the topology to the daemon through PMIx. If this fails as well, it performs an expensive native topology discovery.

We also added several options for changing the heuristic in case there multiple large holes in virtual memory, so that we can force a placement before the heap or at a specific address. However, we have not met yet any case where this could be useful.

V. Evaluation

A. Experimental Setup

We tested our implementation on 3 platforms:

- **KNL64** is a Dell PowerEdge C6320p featuring a Intel Knights Landing Xeon Phi 7230 (64 cores, 1.3GHz). It is configured in SNC-4 and Flat modes. Its hwloc topology contains 430 objects (without counting I/O objects and L1i caches).
- **NUMA96** is a Dell R940 with 4 Intel Xeon E7-8890v4 (24 cores each, 2.2GHz). Hyper-threading is disabled, Cluster-on-Die is enabled. Its hwloc topology is made of 405 objects.
- **Normal24** is a Dell R640 with 2 Intel Xeon E5-2680v3 (12 cores each, 2.5GHz). Hyper-threading is disabled, Cluster-on-Die is enabled. Its hwloc topology contains 97 objects.

We use an Open MPI nightly snapshot from the master branch which contains our implementation.[14]

We measure the memory footprint and launch time with the ORTE `hello` test[15]. It calls `MPI_Init()` and then forces Open MPI internals to load the topology for checking the binding the process.

B. Memory Footprint

We measure the memory footprint difference between implementations using the `mallinfo()` function which returns the total allocated space with `malloc()` in the current process.[16]

TABLE II
Comparison of the memory footprint in Open MPI processes when retrieving the hwloc topology from the native operating system, through a XML buffer, through a shared-memory mapping, or when not retrieving the topology.

	Native Discovery	XML	Shared-Memory	No Topology
KNL64	2.21MiB	2.35MiB	1.614MiB	1.613MiB
NUMA96	1.82MiB	1.94MiB	1.230MiB	1.229MiB
Normal24	1.74MiB	1.78MiB	1.535MiB	1.534MiB

Table II reveals that using our shared-memory topology implies almost no memory footprint on slaves (Open MPI ranks) compared to when no topology information is retrieved at all. The 1kB difference comes from the local duplication of a structure required for function pointers as explained in Section IV-A. However, there is still 700kB memory footprint for creating the shared region on the master side (not shown here), *i.e.* only once per node.

In comparison, instantiating a local topology from native discovery requires more than 600kB on KNL64 and NUMA96 nodes, and 200kB on the Normal24 node.

When importing the topology from the local daemon through XML, the memory footprint is even higher because the XML buffer is kept in memory as long as the actual topology is used. This has been fixed in hwloc but not yet integrated in Open MPI.

TABLE III
Memory saving per node in the shared-memory case, depending on the number of processes. N slave-side topologies are replaced with a single master-side, hence saving N-1 topology footprints in physical memory.

		Memory Saving per Node
KNL64	1 process	0
	2 processes	500kB
	64 processes (1 per core)	31MiB
	256 processes (1 per hardware thread)	127MiB
NUMA96	1 process	0
	2 processes	590kiB
	96 processes (1 per core)	56MiB
Normal24	1 process	0
	2 processes	200kiB
	24 processes (1 per core)	4.6MiB

Table III shows that tens of megabytes are saved per node thanks to our strategy when launching one process per core.

C. MPI Launch Time

We now look at the impact of our change on the launch time of MPI jobs. Table IV shows times for our platforms with one MPI process (one slave) per core, or one per hardware thread on KNL64.

TABLE IV
Comparison of the launch-time of a Open MPI job depending on the machine and on the number of processes.

	Native Discovery	XML	Shared-Memory	vs. XML
KNL64 – 64 processes (1 per core)	9.69s	4.16s	1.68s	× 2.48
KNL64 – 256 processes (1 per hardware thread)	47.20s	18.45s	7.02s	× 2.63
NUMA96 – 96 processes (1 per core)	7.29s	1.17s	0.56s	× 2.10
Normal24 – 24 processes (1 per core)	0.84s	0.53s	0.47s	× 1.13

We observe more than 2× launch-time improvement from our shared-memory implementation over the standard XML one (which was already 2× faster than native topology discovery from the operating system [6]). On the slave-side, each process does not have to parse the XML topology dump anymore, it just has to memory-map the shared-memory region. However, the actual improvement comes from the master-side which does not have to send the XML to many processes anymore. It only has to create a single shared memory region.

[14]Git commit `6d7a780` from October 15th 2017 on Open MPI master branch contains an embedded copy of hwloc master branch at Git commit `65cb1de` with the new shared-memory API.

[15]Available under `orte/test/mpi/hello` in the Open MPI source.

[16]in the `uordblks` of the returned `mallinfo` structure.

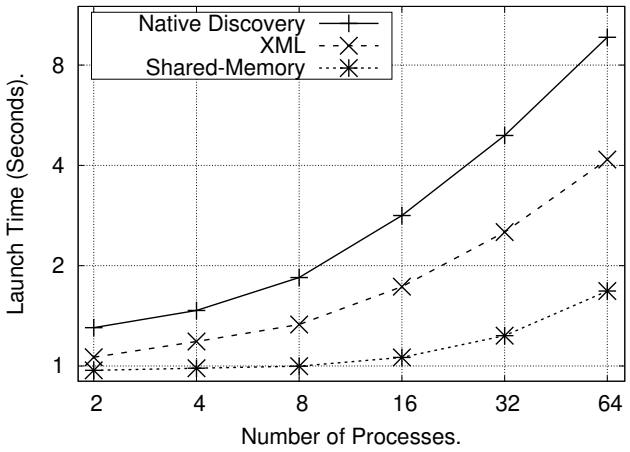

Fig. 7. Comparison of the launch-time of a Open MPI job depending on the number of process on the KNL64 platform. Processes are spawned on cores in a round-robin-by-core manner[18].

Figure 7 details the speedup on KNL64 when increasing from one to 64 processes. It shows that the speedup between shared-memory and XML increases with the number of processes. Indeed the main bottleneck is on the master-side when distributing the XML.

The launch-time cannot be constant with the number of processes even in shared-memory because processes are created as a hierarchy of children from the master. Also propagating the shared-memory topology across many cores of the platforms comes with the cost of transmitting information between the large cache hierarchy. The cost of the actual launch and of topology gathering cannot be easily separated here because the latter is performed early and deep inside the initialization procedure of the Open MPI processes, at least for binding reasons.

The launch-time improvement is the same in multi-node MPI jobs because the ORTE launcher uses one daemon per node for launching local MPI processes. Hence every node will get the same local launch-time improvement simultaneously.

VI. Related Works

A. Topology of Multiple Nodes

Large numbers of compute nodes will be used in exascale platforms. These nodes are managed by centralized batch schedulers which need information about resources in these nodes. Maintaining one topology per compute node is not possible. However, most of these nodes are often identical or similar, making synthesizing and/or factorizing of information possible [6], [11]. This approach focused on the memory footprint when looking at the topologies of multiple compute nodes from a master node, while this paper targets topology management between processes inside a single compute node.

[18] `--map-by core` binding argument on the `mpiexec` command-line.

B. Third-Party vs. Custom Topology Management

Although topology-awareness has been an important topic for more than a decade, hwloc is the only software that was widely used in this area. Other solutions consisted in having custom topology management embedded in runtimes. It enables adapting to the exact application needs, which may reduce the overhead for discovery time and memory footprint. For instance, sharing the locality of cores and NUMA nodes between processes can be implemented as sharing two arrays of integers. However, such approaches are hardly portable to new hardware technologies (new levels of caches, heterogeneous memories, *etc.*) and to new operating systems.

Delegating the topology discovery to third-party tools such as hwloc comes with advantages in terms of portability and flexibility, but with an overhead as discussed in this paper. A general purpose topology management library does not always know which information its users may actually need. The API must be generic enough for offering easy ways to consult different kinds of information (hierarchy of objects, list of similar resources, object attributes, *etc.*) and this API cannot be changed without breaking existing applications.

C. Memory Footprint vs. Processes

The reason memory footprint became an issue for hwloc is that many legacy applications (for instance from DOD and DOE labs) are still programmed in pure-MPI. They must run with one process per core (or even per hardware thread). Porting these applications to modern hybrid programming models would largely avoid the issue by dramatically reducing the number of processes. However, some of these legacy applications cannot be modified anymore, even if "*the potential gains in scalability and absolute performance may be worth the significant coding effort*" [12]. Besides, some programming models were designed to avoid the memory footprint of multiple processes. The concept of process virtualization in Charm++ and Adaptive MPI lets the programmer divide the work in many small chunks while the runtime takes care of scheduling them on the physical processors. [13]

D. Memory Footprint of MPI implementations

The memory footprint of libraries has been an important topic for MPI implementations. When InfiniBand networks became widespread, lots of memory was used for network buffers, preventing applications from using it for their datasets. Using unreliable datagrams instead of reliable connections was a first idea towards reducing the per-connection footprint [14]. Then hardware features such as Shared Receive Queues (SRQ) and eXtended Reliable Connection (XRC) enabled factorizing between connections [15], [16]. Now that these former heavy consumers have been fixed, other libraries in the HPC stack are being looked at, hence our work on hwloc.

Another approach for reducing the footprint of MPI jobs consist in adding some sharing between independent processes. MPC is an MPI implementation that uses threads instead of processes for running MPI ranks [17]. At the operating systems level, partial sharing of page-tables has been

proposed as a way to factorize the memory consumption when some libraries are shared [18].

Beside sharing of data structures inside nodes when appropriate, some compression techniques have also been proposed to reduce the memory footprint of MPICH for managing thousands of network addresses [19].

E. Mapping at the same Virtual Address

Mapping a shared-memory region at the same virtual address in different processes has been used several times in the past. PM2 [20] and Adaptive MPI [21] used *isomalloc* for placing thread stacks at the same location when threads migrated between processes. The runtime in charge of managing these processes and threads took care of pre-allocating the virtual region during startup. Virtual memory was not larger than physical memory at that time (about 32bits for both), making an approach like ours impossible. However, the runtime had full knowledge of which processes will map the region and when, making sure the target virtual address will always be free.

MPC [22] performed a reduce operation for intersecting the virtual memory holes of all processes to find a region available in all of them. This strategy is not applicable to our case because our daemon may create the shared region before some processes are created, at least in case of dynamically-spawned MPI processes. Our heuristic works in the general case where we do not know which processes will ever try to map the shared region. However, it requires a large portion of virtual memory to be free in order to get a good probability of success. This heuristic may be Linux specific. Fortunately 100% of the Top500 supercomputers currently run Linux and we do not expect things to change for many-core platforms in the near future.

VII. Conclusion

As nodes become more complex, with tens of cores and deep memory hierarchy, there is a need for making topology information available to all components of the software stack. The memory footprint of all these libraries is, however, severely constrained by the small amount of physical memory available per core on HPC platforms as well as the need to make most of it available to the actual application.

We discussed in this article the memory footprint of locality information implied by the hwloc library, the *de facto* standard tool for exposing topology. It already raises issues on many-core platforms running many MPI processes that used to instantiate their own copy of the topology information. It will be even more problematic on upcoming larger many-cores for exascale super-computers.

We presented a way to factorize this information in a shared memory region. Given that this information is already available to all processes in less efficient ways, our proposal raises no security issue. Based on our in-depth analysis of virtual and physical memories in HPC platforms, we proposed a heuristic that enables the mapping of this shared memory at the same address in all processes. Hence they can still use the existing hwloc API for consulting topology information, while dramatically reducing the memory footprint to a single topology per node.

We demonstrated that the footprint in the Open MPI implementation is indeed reduced. This enables significant memory savings on many-core platforms, which was the primary motivation for this work. Our work is therefore already used in production in DOE labs for pure MPI applications running on Intel Xeon Phi nodes.

Moreover the launch time is also improved by more than a factor of 2 on two many-core platforms, a 64-core Intel Knights Landing Xeon Phi and a 96-core NUMA host. However, this additional improvement, currently in the order of seconds, is currently negligible on long-running jobs.

This work is available in the recently released hwloc 2.0[19] and in the upcoming Open MPI 3.1[20]. It is also under investigation for integration in Slurm (whose compute node daemon can be the master when Slurm is used as a process launcher) as well as Intel MPI and MPICH (which already use hwloc for launching their jobs).

Future works include working on interoperability between HPC software layers: now that processes may share locality information at the MPI level, it should also be shared with parallel libraries or runtimes running inside these MPI processes. A working-group is already looking at using PMIx for OpenMP, MPI and resource manager coordination[21]. We envision sharing the hwloc topology within that scheme. We are also looking at managing resource allocation in that global context. Multiple software layers may want to use some cores concurrently for different reasons (parallel computation with processes and/or threads, offloading work, partitioning memory or caches, *etc.*). This requires to share topology information between them, and to design a generic way to request resources and to consult what other layers are doing.

We are also looking at the impact of non-volatile memory on our model. Terabyte-class NVDIMMs will be available in the future, making the physical memory per node possibly much larger. Depending on whether this new memory is used as storage or mapped as virtual memory, and how it is mapped, we may have to revisit our heuristic for finding the shared region location.

Acknowledgment

We would like to thank Ralph Castain from Intel for discussing the design and integrating our work in Open MPI.

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

[19] Available from https://www.open-mpi.org/projects/hwloc/.
[20] Available from https://www.open-mpi.org.
[21] https://github.com/pmix/RFCs/blob/master/RFC0017.md

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 International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*. Pisa, Italia: IEEE Computer Society Press, Feb. 2010, pp. 180–186. [Online]. Available: http://hal.inria.fr/inria-00429889

[2] 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.

[3] R. Yang, J. Antony, P. P. Janes, and A. P. Rendell, "Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2," in *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008)*, 2008, pp. 31–36.

[4] P. Micikevicius, "Multi-GPU Programming," GPU Technology Conference, 2012, http://on-demand.gputechconf.com/gtc/2012/presentations/S0515-GTC2012-Multi-GPU-Programming.pdf.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[6] B. Goglin, "On the Overhead of Topology Discovery for Locality-aware Scheduling in HPC," in *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2017)*. St. Petersburg, Russia: IEEE Computer Society Press, Mar. 2017. [Online]. Available: http://hal.inria.fr/hal-01402755

[7] S. Hemmert, J. Ang, B. Carnes, P. Chiang, D. Doerfler, S. Dosanjh, P. Fields, K. Koch, J. Laros, M. Leininger, J. Noe, T. Quinn, J. Torrellas, J. Vetter, C. Wampler, and A. White, "Exascale Hardware Architectures Working Group," Mar. 2011, NNSA Workshop: From Petascale to Exascale: R&D Challenges for HPC Simulation Environments.

[8] European Exascale Software Initiative 2, "D4.3 Final report on enabling technologies," Jul. 2015.

[9] Linux Weekly News, "Address space randomization in 2.6," 2005, https://lwn.net/Articles/121845/.

[10] R. H. Castain, D. Solt, J. Hursey, and A. Bouteiller, "Pmix: Process management for exascale environments," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 14:1–14:10. [Online]. Available: http://doi.acm.org/10.1145/3127024.3127027

[11] 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. [Online]. Available: http://hal.inria.fr/hal-00985096

[12] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009, pp. 427–436.

[13] L. V. Kalé, "The virtualization model of parallel programming : Runtime optimizations and the state of art," in *LACSI 2002*, Albuquerque, October 2002.

[14] A. Friedley, T. Hoefler, M. Leininger, and A. Lumsdaine, "Scalable High Performance Message Passing over InfiniBand for Open MPI," in *Proceedings of 3rd KiCC Workshop 2007*. RWTH Aachen, Dec. 2007.

[15] S. Sur, M. J. Koop, and D. K. Panda, "High-performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-depth Performance Analysis," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188565

[16] M. J. Koop, J. K. Sridhar, and D. K. Panda, "Scalable MPI design over InfiniBand using eXtended Reliable Connection," *IEEE International Conference on Cluster Computing*, pp. 203–212, 2008.

[17] M. Pérache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption." *PVM/MPI*, vol. 9, pp. 94–103, 2009.

[18] B. Gerofi, A. Shimada, A. Hori, and Y. Ishikawa, "Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures," in *Proceedings of ACM/IEEE International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Delft, Netherlands, 2013, pp. 360–368.

[19] Y. Guo, C. J. Archer, M. Blocksome, S. Parker, W. Bland, K. Raffenetti, and P. Balaji, "Memory Compression Techniques for Network Address Management in MPI," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1008–1017.

[20] G. Antoniu, L. Bougé, and R. Namyst, "An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System," in *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. Po Rico, Puerto Rico: Springer-Verlag, Apr. 1999, pp. 496–510. [Online]. Available: https://hal.inria.fr/inria-00565361

[21] C. Huang, O. Lawlor, and L. V. Kalé, *Adaptive MPI*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322. [Online]. Available: https://doi.org/10.1007/978-3-540-24644-2_20

[22] M. Pérache, H. Jourdren, and R. Namyst, "Mpc: A unified parallel runtime for clusters of numa machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85451-7_9