Locality-Aware Parallel Process Mapping for Multi-Core HPC Systems

Joshua Hursey
Oak Ridge National Laboratory
Oak Ridge, TN USA 37831
Email: hurseyjj@ornl.gov

Jeffrey M. Squyres
Cisco Systems, Inc.
San Jose, CA 95134
Email: jsquyres@cisco.com

Terry Dontje
Oracle
Redwood Shores, CA 94065
Email: terry.dontje@oracle.com

***Abstract*—High Performance Computing (HPC) systems are composed of servers containing an ever-increasing number of cores. With such high processor core counts, non-uniform memory access (NUMA) architectures are almost universally used to reduce inter-processor and memory communication bottlenecks by distributing processors and memory throughout a server-internal networking topology. Application studies have shown that the tuning of processes placement in a server's NUMA networking topology to the application can have a dramatic impact on performance. The performance implications are magnified when running a parallel job across multiple server nodes, especially with large scale HPC applications.**

This paper presents the Locality-Aware Mapping Algorithm (LAMA) for distributing the individual processes of a parallel application across processing resources in an HPC system, paying particular attention to the internal server NUMA topologies. The algorithm is able to support both homogeneous and heterogeneous hardware systems, and dynamically adapts to the available hardware and user-specified process layout at run-time. As implemented in Open MPI, the LAMA provides 362,880 mapping permutations and is able to naturally scale out to additional hardware resources as they become available in future architectures.

***Keywords*-Process Affinity; Locality; NUMA; MPI; Resource Management**

I. Introduction

High Performance Computing (HPC) systems are commonly composed of server nodes containing many processing units (e.g., cores or hardware threads), often separated into non-uniform memory access (NUMA) domains. Announcements of 128 (or more) core machines indicate a continuing trend to increase processor core counts within individual nodes. Such large numbers of cores significantly increase the hardware complexity of nodes. For example, accessing memory – or other inter-process communication methods – may require traversing inter-NUMA-node network links, reminiscent of large processor-count symmetric multi-processor (SMP) machines.

Indeed, NUMA performance effects may no longer be as simple to model as "near" vs. "far" memory latency. Since modern inter-NUMA-node networking topologies are capable of routing, remote memory may be more than one hop away. Networking effects such as routing and congestion between processors in the same server can become performance bottlenecks. In addition to NUMA networking effects, sophisticated memory hardware subsystems involve multiple layers of cache that sometimes span multiple processor cores. For example, a process running on core A that thrashes its cache may also affect the performance of a process running on core B that shares the same cache.

All of these factors mean that, in today's "highly NUMA" server architectures, HPC applications must be aware of and react to the location of both the processors where it is running and its associated memory in the internal server topology. While operating systems do a reasonable job of trying to keep a process' memory physically located near the processor core(s) where it is running, they lack the application developer's knowledge of what the process will do, and what resources it will need to access in the future.

Accordingly, the HPC application community has begun experimenting with manual placement of the individual processes in a parallel job. Application studies have shown that application tuned placement of processes can lead to significant performance improvements. Commonly referred to as "process placement" or "process affinity," this type of optimization is typically divided into two steps:

1) *Mapping:* Create a layout of processes-to-resources.
2) *Binding:* Launch processes and enforce the layout.

When servers only had a few processor cores, simplistic schemes such as a round-robin distribution of the processes in a parallel job across server nodes or across processor cores were sufficient. However, with the growing "depth" of processors, memory, and complex NUMA networking interconnects, more fine-grained control of process placement is required to optimize application performance.

This paper presents the Locality-Aware Mapping Algorithm (LAMA) for distributing the individual processes of a parallel application across processing resources spanning one or more compute nodes. The LAMA is able to support both homogeneous and heterogeneous hardware systems. The algorithm dynamically adapts, at runtime, to the available hardware and user-specified process layout. The flexible process layout input specification enables the algorithm to naturally support a large number of mapping permutations.

Since process placement optimization is unique to individual applications, user-level input to the LAMA is critical. Domain-level experts need to be able to specify and experiment

978-0-7695-4516-5/11 $26.00 © 2011 IEEE
DOI 10.1109/CLUSTER.2011.59

with different placements to find an optimal configuration. A compact yet flexible user-level interface to the LAMA is presented in the context of an Message Passing Interface (MPI) [1] implementation, allowing the user to specify both regular and irregular process placement layouts across an HPC system.

II. Related Work

Application studies show significant performance improvements when non-traditional process placement is used to optimize communication patterns. For example, the Gyrokinetic Toroidal Code (GTC) application, used in fusion simulations, demonstrated that application-specific mapping of processes to nodes on a network can significantly improve both scalability and performance (improving performance up to 30% between 8K and 32K nodes) [2]. Other research has demonstrated the performance effects of specific process placement on the NAS parallel benchmarks for different systems [3].

To support the process placement of such applications, implementations of the MPI standard either provide their own run-time systems for launching and monitoring the individual processes in a parallel application (also commonly referred to as a parallel job), or use a back-end parallel run-time environment support for this functionality. Regardless of the underlying mechanism, two common regular mapping patterns that are almost uniformly provided by all MPI implementations are *by-node* (a.k.a., *scatter*, *cyclic*) and *by-slot* (a.k.a., *bunch*, *pack*, *block*). The *by-node* option places processes in a round-robin style at the node level; process N is placed on node M, process $(N+1)$ is placed on node $(M+1)$, and so on. The *by-slot* option places processes in a round-robin style at the slot level; process N is placed in slot M, process $(N+1)$ is placed in slot $(M+1)$, and so on. Unfortunately, the specific definition of *slot* varies between implementations and platforms, but it usually means either processor cores or hardware threads – a distinction that can significantly impact performance. Recently, options are starting to emerge for round-robin process distribution across processor sockets, caches, hardware threads, and network resources.

The Open MPI [4] and MPICH2 [5] projects are both popular open source implementations of the MPI specification. The Open MPI v1.5 series provides a large number of different `mpirun` command line options paired with numerous runtime configuration parameters that influence the mapping and binding of processes. Both regular and irregular process layouts are supported. MPICH2 v1.3.2p1 [6] supports irregular layouts (via a *hostfile*), as well as different topology-aware regular allocation strategies to allow for packing or scattering processes across any one level of hardware threads, cores, cache, and processor sockets.

MPI implementations sometimes rely on back-end parallel runtime environments provided by job schedulers and resource managers to provide process placement functionality for parallel jobs. SLURM [7] is a resource manager and scheduler for Linux clusters. As of v1.2, SLURM supports several explicit process affinity patterns and a set of regular distributions, including *block*, *cyclic*, and *plane*. Other options are provided to restrict the total number of processes for any particular resource, and a *hostfile* can also be used to specify irregular process layouts.

Most mapping algorithms view compute nodes as equidistant from one another. The MPI implementation on IBM BlueGene systems allow applications to map with respect to their position in the three-dimensional torus network [8], and, additionally in the BlueGene/P system, across the cores within a node [9, Appx. F]. The regular mapping pattern is expressed in terms of relative X, Y, Z coordinate ordering for the torus network, and an additional T parameter for cores. The order of these parameters (e.g., $XYZT$ vs. $YXTZ$ vs. $TZXY$) determines the order of mapping directions across the torus network and cores within a node. Using these options, literature has investigated various permutations of patterns to optimize application process layouts for performance [10]. Unfortunately, this mapping pattern does not account for internal server topologies that might affect application performance such as sockets and cache levels. A mapping file can be specified for irregular patterns not well supported via command line options.

Cray's Application Level Placement Scheduler (ALPS) mechanism is used to map processes in the Cray Linux Environment (CLE) [11]. As of version 3.1.UP03 of CLE, ALPS provides similar functionality as BlueGene to map processes with respect to the relative location in the torus network, though with a different command line interface. Irregular patterns can be similarly specified via a CPU placement file. ALPS has additional options to adjust the iteration order for each resource, and for restricting the total number of processes placed on any given resource.

The Locality-Aware Mapping Algorithm (LAMA) presented in this paper draws inspiration from all of the work cited above, but most notably by the mapping technique used by the BlueGene system. The LAMA extends the previously established capabilities by allowing for the expression of a relative mapping across any set of levels of the on-node hardware topology. By its nature, the LAMA is extensible to include additional hardware localities (such as additional caches, or other machine-internal groupings of processors or memory) without modifying the algorithm's core logic. Additionally, the algorithm can be applied to heterogeneous hardware systems (inclusive of scheduler and operating system (OS) restrictions) which have not been well-supported in some previous implementations. Finally, the algorithm nicely supports a command line interface (implemented in Open MPI) that enables complex regular pattern specification for both mapping and binding.

III. Launching Parallel Applications

The concepts presented in this paper generally apply to parallel applications in many kinds of distributed computing environments. However, for simplicity, this paper will present concepts in the context of a single use-case: launching a multi-process parallel MPI job in a typical HPC cluster consisting of commodity multi-core NUMA machines.

Launching multi-process, (possibly multi-threaded) parallel applications requires the support of a parallel run-time environment. Parallel run-time environments can launch and monitor groups of processes across nodes in an HPC system.

A. Mapping Processes

Among the first steps in launching a parallel job is obtaining computational resources on which to run. Modern HPC cluster resource managers can allocate compute resources at the granularity of individual processor cores, instead of only the more traditional node granularity. For example, the resource manager may (depending on site policy) allocate half the cores from node A and half the cores from node B to a single job.

In some clusters, all nodes contain identical hardware and internal topologies – a *homogeneous hardware* system. Other clusters are composed of different kinds of hardware (possibly collected over time), referred to as a *heterogeneous hardware* system. Some operating systems allow for individual computational domains within a node (e.g., processor socket, processor core, hardware thread) to be off-lined, often for maintenance or allocation reasons. Such restrictions in a system can make a *homogeneous hardware* system look like a *heterogeneous hardware* system.

Once resources have been allocated to the job, a mapping agent (provided by either the resource manager or the MPI implementation) must create a plan for launching the overall application – a map pairing processes-to-processors and memory. For optimal application performance, this map should be created with awareness of both the network and internal node topologies, and guided by the application.

Once the hardware topologies from all allocated nodes have been assembled by probing the environment, processes can be mapped to processors and memory. A variety of mapping strategies can be used to place MPI processes across the available resources, including the *by-slot* and *by-node* patterns described in Section II.

In most HPC environments, if a single processing unit (usually a processor core) is assigned more than one process, this is considered *oversubscribing* the hardware. Oversubscription is generally disallowed because most MPI / HPC applications are CPU-intensive; sharing processors between processes would cause starvation and severe performance degradation. Conversely, some applications may need *more* than one processing unit – the application may be multi-threaded, for example. In these cases, the mapping agent needs to be able to assign multiple processing resources to each process.

It is important to note that the mapping algorithm used by the mapping agent only *plans* what processing resources are assigned to processes. Specifically, the act of planning process-to-processor mappings is used to:

1) Specify which processes will be launched on each node.
2) Identify if any hardware resource will be oversubscribed.

Once the mapping is complete, the parallel run-time environment launches individual processes on their respective nodes. In the map, processes are addressed to the resolution of a single processing unit – whatever the smallest processing unit found on the cluster hardware (e.g., a hardware thread). This allows maximal resolution for the binding step to accurately determine the *binding width*, described in Section III-B. Though whether or not processes are *bound* to resources finer-grained than an entire node is a separate, optional step.

B. Binding Processes

The process-launching agent of the parallel run-time environment works with the OS to limit exactly where each process can run in one of several ways: *no restrictions*, *limited set restrictions*, or *specific resource restrictions*.

1) No restrictions: The OS scheduler has full autonomy to decide where the process runs and what resources it uses.

2) Limited set restrictions: The process-launching agent limits the job's individual processes to run on a common subset of processors on a node.

3) Specific resource restrictions: The process-launching agent assigns specific, unique processors for each individual process.

The lack of inter-processor migration in the *specific resource restrictions* case provides the best possibility for optimal execution. Note that a process may be bound exclusively to multiple processors; a multi-threaded application may benefit from being bound to an entire processor socket, for example.

The number of processors to which a process is bound is referred to as its *binding width*. For example, a process bound to an entire processor socket has a *binding width* of the N smallest processing units (e.g., hardware threads) in that socket.

IV. Locality-Aware Mapping Algorithm

Once the parallel run-time environment has access to the per-node hardware topology information, its mapping agent can start mapping processes to the available resources. The mapping algorithm used by the mapping agent must be able to effectively translate the mapping intent of the user (typically expressed through the environment or command line) to the hardware topologies allocated to the job. The mapping algorithm must be able to handle homogeneous and heterogeneous hardware systems, and be extensible enough to support the ever-advancing complexity of such systems and the demands of applications. The Locality-Aware Mapping Algorithm (LAMA) supports both hardware system types and is dynamically extensible to arbitrary levels of the on-node hardware topology.

A. Process Layout

The LAMA applies a user-specified mapping iteration ordering, called a *process layout*, to the per-node topology information gathered by the parallel run-time environment. The *process layout* is specified as a sequence of letters, such as those shown in Table I for the Open MPI implementation. The algorithm inspects the process layout from left-to-right to determine the order in which cluster resources are traversed when assigning processes. A nested set of loops operates at the

Resource	Abbreviation	Description
Node	n	Server node
Board	b	Motherboard
Processor Socket	s	Processor socket
Processor Core	c	Processor core (on a socket)
Hardware Thread	h	Hardware thread (e.g., hyperthread)
Optional Locality Parameters		
L1 Cache	L1	L1 Cache
L2 Cache	L2	L2 Cache
L3 Cache	L3	L3 Cache
NUMA Node	N	NUMA memory locality

TABLE I
RESOURCES THAT CAN BE SPECIFIED AND THE CORRESPONDING ABBREVIATIONS USED TO REPRESENT THEM IN THE PROCESS LAYOUT FOR THE OPEN MPI IMPLEMENTATION OF THE LAMA.

```
 1 // maximal_tree is represented as a 2D array of resources
 2 // Launch 24 processes 'by−socket'
 3 call mapper(``sbnch'', 5, 24, maximal_tree);
 4 mapper(layout, max_level, max_rank, mtree) {
 5   rank = 0;
 6   while (rank < max_rank) {
 7     // Traverse process layout right−to−left
 8     rank = inner_loop(layout, max_level−1,
 9                       rank, max_rank, mtree);
10   }
11 }
12 inner_loop(layout, level, rank, max_rank, mtree) {
13   foreach resource in mtree[level] {
14     if (level > 0) {
15       rank = inner_loop(layout, level−1,
16                         rank, max_rank, mtree);
17     } else if (resource_exists_and_is_available(...)) {
18       map(rank, resource);
19       ++rank;
20     }
21     if (rank == max_rank) return rank;
22   }
23   return rank;
24 }
```

Fig. 1. Pseudo code describing the recursive function at the heart of the LAMA. It is used to iterate over all resources levels specified in the process layout.

heart of the LAMA with the left-most resource in the process layout placed in the innermost nested loop, and the right-most resource placed in the outermost nested loop.

The LAMA will map the process to the smallest processing unit available. If, for example, cores are the smallest processor unit available (e.g., if hardware threads are disabled), then processes will be mapped to processor cores instead of hardware threads. Instead of using explicit nested loops, a recursive function (shown in Figure 1) is used so that the depth of the loop structure can easily be adjusted at runtime depending upon the resources available and the process layout specified by the user. By default, each resource level is iterated sequentially starting at the lowest logical resource number at that level to the highest (as shown on Line 13 of Figure 1). Other iteration patterns, such as custom versions provided by the end user, can also be supported by the LAMA.

It is possible for the LAMA to iterate onto a resource location that is *unavailable* for mapping. *Unavailable* resources are present in the hardware topology but have been disallowed by the scheduler and/or OS. If the resource is unavailable, the LAMA skips it and the iterations continue searching for the next acceptable resource (as seen on Line 17 of Figure 1).

B. Maximal Tree

Since the node resource (n) can be placed anywhere in the process layout, iterating across other resource levels is difficult in heterogeneous hardware systems since the widths of those levels (e.g., number of cores) might change between nodes. To address this issue, the LAMA creates an abstract maximal tree topology used solely for mapping purposes. The maximal tree topology is the union of all the different single-node hardware topologies in the HPC system.

It is possible that a resource is represented in the maximal tree topology (e.g., L2 cache), but is not specified by the user in the process layout. In this case, the resource level is *pruned* from the maximal tree. When a resource level is pruned from the maximal tree, its children become those of its parent. The parent provides a new logical numbering to the children since it is combining subtrees due to the elimination of an entire resource level. This maximal tree pruning technique enables the algorithm to be adaptive to changing user requirements and system information. This technique is also useful in resolving architectural differences (e.g., placement of caches) in heterogeneous hardware systems.

C. Example Mappings

Figure 2 provides an illustration of how the mapping pattern scbnh would map a 24 process job onto two nodes in a homogeneous hardware system. The scbnh process layout scatters processes across all sockets then all cores within a single node before moving on to the next node. Once all nodes have the first hardware thread on every core in every socket mapped, then the LAMA starts again from the second hardware thread on the first core of the first socket of the first node.

V. IMPLEMENTATION

The LAMA described in Section IV has been implemented as part of the Open MPI project's run-time environment, the Open MPI Runtime Environment (ORTE). The LAMA is implemented as the hwtopo component of the rmaps framework. Additionally, the rankfile rmaps component provides a file format for irregular mapping and binding patterns. The Portable Hardware Locality (hwloc) project is used to provide on-node hardware topology information and portable process binding abstractions for Open MPI [12].

The Open MPI project defines four levels of abstraction in the command line interface ranging from simplicity of

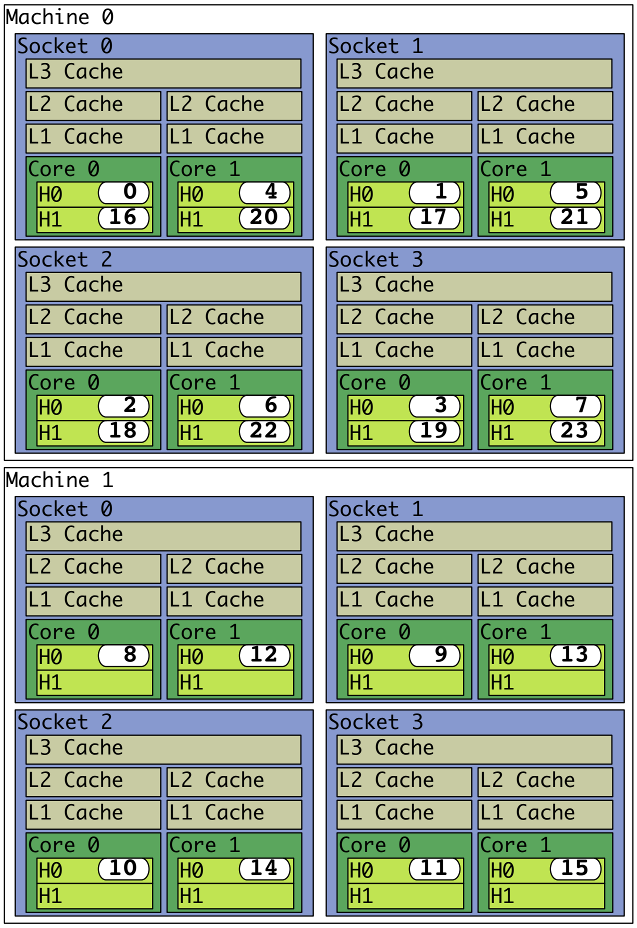

Fig. 2. Example mapping (not binding) of 24 processes using a `scbnh` process layout. This is a regular pattern that scatters processes across all sockets then all cores of a node. The nodes in this example only have one motherboard, and therefore the motherboard is not shown for simplicity.

expression to flexibility of specification. Levels 1 and 2 are shortcuts for Level 3 LAMA specifications.

- **Level 1**: No mapping or binding options specified.
- **Level 2**: Simple, common patterns.
- **Level 3**: LAMA process layout regular patterns.
- **Level 4**: Irregular patterns described in a *rankfile*.

VI. Conclusion

Commodity HPC systems are composed of server nodes containing many processing units (e.g., cores or hardware threads) often separated into separate NUMA domains. Applications have demonstrated significant performance benefits by tuning process placement in the overall HPC system to their application. As HPC systems continue grow more complex, a wider variety of process mapping and binding options will be needed to support these applications.

This paper introduced the Locality-Aware Mapping Algorithm (LAMA) for distributing individual processes of a parallel application across the processing resources in an HPC system paying particular attention to on-node hardware topologies and memory locales. The algorithm is able to support both homogeneous and heterogeneous hardware systems inclusive of scheduler and/or OS restrictions. The recursive nature of the LAMA allows it to respond dynamically, at runtime, to changing hardware topologies and user specified process layouts.

As implemented in Open MPI, the LAMA is used to provide a range of regular mapping pattern abstraction levels. As a result, Open MPI is able to provide up to 362,880 mapping permutations to the end user by using the LAMA.

Acknowledgments

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

[1] Message Passing Interface Forum, "MPI: A Message Passing Interface," in *Proceedings of Supercomputing '93*. IEEE Computer Society Press, November 1993, pp. 878–883.

[2] S. Ethier, W. M. Tang, R. Walkup, and L. Oliker, "Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas," *IBM Journal of Research and Development*, vol. 52, pp. 105–115, January 2008.

[3] E. Jeannot and G. Mercier, "Near-optimal placement of MPI processes on hierarchical NUMA architectures," in *Proceedings of the 16th international Euro-Par conference on Parallel Processing*, ser. Euro-Par'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 199–210.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[5] Argonne National Laboratory, "MPICH2," http://www.mcs.anl.gov/mpi/mpich2.

[6] ——, "Using the Hydra process manager," http://wiki.mcs.anl.gov/mpich2/index.php/Using_the_Hydra_Process_Manager.

[7] A. Yoo, M. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer Berlin / Heidelberg, 2003, vol. 2862, pp. 44–60.

[8] G. Almási, C. Archer, J. Castaños, C. Erway, P. Heidelberger, X. Martorell, J. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-burow, W. Gropp, and B. Toonen, "Implementing MPI on the BlueGene/L supercomputer," in *Euro-Par 2004 Parallel Processing*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds. Springer Berlin / Heidelberg, 2004, vol. 3149, pp. 833–845.

[9] C. Sosa and B. Knudson, "IBM System Blue Gene Solution: Blue Gene/P application development," IBM, Tech. Rep., September 2010. [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg247287.html

[10] H. Yu, I.-H. Chung, and J. Moreira, "Topology mapping for Blue Gene/L supercomputer," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[11] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, "The application level placement scheduler," in *Cray Users Group*, 2006.

[12] 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.