Minimizing System Noise Effects For Extreme-Scale Scientific Simulation Through Function Delegation

U.S. Department of Energy Grant DE-FG02-08ER25845
Final Report

Jack Dongarra and George Bosilca
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN 37919, USA
{dongarra,bosilca}@icl.utk.edu

While the insatiable demand of computing power from the domain sciences motivates the deployment of ever powerful High Performance Computing (HPC) systems, thermal and power consumption concerns have curbed the growth of both node count and processor frequency. As an alternate source of processing power, multicore clusters have become the most prominent form of HPC systems, and exhibit a rapid increase in the number of cores per node. The top ranking machine in the latest TOP500 list, the ORNL-based Titan computer, uses more than half a million cores¹. Processors with 12 cores are available from major commodity vendors, and it is common to have these deployed in multiple socket boards featuring 8 to 48 cores, with network-style interconnection between caches or to the memory banks (e.g., Intel QPI or AMD Hyper-transport). Unfortunately, this new hardware trend challenges the assumptions made by most current HPC programming models, which threatens the performance efficiency of the machines. Namely, within nodes, non uniform memory accesses (NUMA), memory and shared cache hierarchies, dismiss the assumptions of regular load balance and even link bandwidth and latency.

To meet the needs of scientific research, supercomputers continue to grow at an unrelenting rate. As supercomputers grow in size from mere thousands to hundreds of thousands of processor cores, important new challenges to performance and scalability are emerging. In the past, performance tuning of a parallel application could be accomplished fairly reliably by separately optimizing its algorithms, communication, and computational aspects. However, at the scales of today's and tomorrow's larger machines, these issues become co-mingled and must be addressed comprehensively.

The primary goal of the Minimizing System Noise Effects For Extreme-Scale Scientific Simulation through Function Delegation project is to eliminate or at best strongly minimize the impact of the noise introduced by the operating system, during large scale parallel applications runs. Collective communication operations are a basic building block for parallel programming models and scientific applications. These operations often dominate execution time of applications and tend to limit their scalability. In order to address this challenge, we evaluated different strategies to adapt the collective communications underlying topologies to the hardware architecture in order to provide increased levels of performance to the parallel applications. A software package (HirKNEM), several publications, and one award have sanctioned our realization during the course of this project.

This project is a joint effort between *Oak Ridge National Laboratory, Indiana University* and the *University of Tennessee*. The sub-projects undertaken at each location are closely linked and address complementary aspects of the global project. This report presents the work completed during the entire project period at the University of Tennessee.

¹<http://www.top500.org/list/2012/11/>

1 Report

The team at the University of Tennessee, Knoxville has focused their efforts on two aspects. First, we designed a delegation framework and API, a set of building blocks for more complex algorithms. While this delegation framework is flexible enough for several usages, our work in the context of this project only focused on collective communications. Second, we investigate how to schedule the resulting basic point-to-point communications and the required operations between buffers in order to take advantage of architectural features of the parallel machines, such as memory and processor affinity or hardware accelerators.

1.1 Kernel Assisted Operations

With the increase in the number of cores per node deeper level of memory hierarchies, it becomes necessary to adapt the point-to-point communications parameters, not only to the capabilities offered by the hardware capability, but also to the architectural distance between the source and the destination in order to allow the communication library to maximize its communication performance (decrease the latency and increase the bandwidth) and to decrease the noise imposed on the application via cache effects (cache appropriation).

The legacy approach for transferring data between processes running on a shared memory platform – but who do not share a common memory space – has been to establish a common memory segment between the two processes. The sender copies a message into the shared memory zone and the receiver copies it out to the target buffer. Such an approach suffers from several drawbacks, one of the most prominent being that it requires two copies for every data transfer (to and from the shared memory region). Kernel-assisted memory copy alleviates this issue by using system calls to offload the copy to the kernel (SMARTMAP, XPMEM, KNEM, CMA). Because the kernel has complete access to the memory space of both processes, it can perform the copy from the source buffer in the sender address space directly to the target buffer in the receiver address space without the need for an intermediate buffer.

Figure 1: Point-to-Point communication bandwidth with KNEM on Intel Tigerton and AMD Istanbul platforms.

1.2 Delegation Framework

The delegation framework will be used to describe what a collective communication is doing during all the intermediary steps (a set of atomic point-to-point communications) in an upper-level language and then delegate the outcome to an external entity to be realized. The goal is to take the communications outside of the user level in the operating system in order to minimize the system noise, and take them at a level immune to system noise. Such a level might be in the kernel or preferably on the network cards.

The API for the delegation framework is straightforward, with several similarities to task dependencies declaration. There are several main tasks, some related to point-to-point messages such as send and receive, and some related to local operations that have to be executed on array-like local data such as arithmetical operations (*sum*, *diff*, and *multiply*), logical operations (*and*, *or*, *xor*, and *not*), and some bitwise operations (*and*, *or*, *negate*, and *xor*). One can notice that most of

these operations are closely related to the operations exposed by the MPI Standard for the global reduction. Once tasks are created, special functions can be used to declare the dependencies between these tasks, based on the data flow between the tasks. The resulting dependencies graph can be visualized as a Directed acyclic graph (DAG), where the nodes are tasks to be executed and the edges are data flows between these tasks.

Using these building blocks, more complex algorithms can be described, and the scheduling engine will be responsible for unrolling the dependencies DAG and scheduling the tasks based on data and resource availability. Moreover, using the underlying data representation, several such algorithms can be composed together, allowing the construction of hierarchical algorithms where each level of the logical hierarchy is a particular level on the architectural hierarchy.

1.3 Scheduling policies

Once the tasks and their dependencies are declared, the resulting representation is packed in a special format and can be exported into the scheduler. Based on this information, the scheduler can start executing the ready tasks, i.e., the tasks that have all of their input dependencies satisfied. In case several tasks are ready at the same time, additional information can be provided to the scheduler in order to allow it to make the best choice about which task will be executed first. One piece of information is the task priority or task type. As an example, tasks that depend or generate remote dependencies, in other words tasks that exchange data with peers, will have a higher priority as they will have to be started as soon as possible in order to allow the scheduler to hide the overhead related to the network transfers.

Simultaneously, the scheduler is aware of the local architecture, depending on the level in the software stack where the scheduler resides. In the case where the scheduler is situated in the firmware of the network card, the topology knowledge is limited to the number of cores available. If the scheduler is situated at the kernel level it knows about both the processors' architecture (such as number of processors and cores, as well as their topology) and the memory cost in cases of *NUMA* architectures. Based on such information, the scheduler can schedule the tasks in order to minimize the memory transit, and therefore take advantage of the data locality. Obviously, such information can only benefit tasks that implement computations, as they are usually the ones that can benefit from data locality.

1.4 Exploiting hardware capabilities

A lot of research has been done to adjust the process layout based on an application's communication pattern and underlying hardware architecture. The driving idea is to build a communication graph where each edge is weighted by the amount of data exchanged between the two nodes. This weight is then used (either statically or during the parallel application execution) to compute an optimal process placement in order to map the processes on the resources attributed by the batch scheduler to the parallel job. Although these intelligent process placements have the potential to significantly decrease the overall communication time, their methodology is based on a pure point-to-point communication pattern, ignoring the different communication topologies used by MPI collective communications. As most of the collective communications exhibit specific communication patterns which are allowed to adapt to the underlying architectural features (split binary tree, binomial tree, chain, etc.), there is a mismatch between the MPI internal collective topology (which is usually created based on the processes' MPI ranks) and the external process placement decision.

Our collective delegation framework is capable of taking advantage of the process placement information and adapting the algorithms used for collective communications to maximize the benefit from the processes’ placement by combining process distance (in terms of NUMA nodes), underlying hardware architecture, and runtime information. Taking advantage of this framework, we developed several distance-aware collective communication algorithms, a hierarchical span over the physical machine. We have demonstrated that our distance-aware collective component provides stable and optimal performance, regardless of process placement, significantly outpacing the state-of-the-art collective algorithms, in both Open MPI and MPICH2 libraries. As an example, Figure 2 highlights the sensitivity of the hierarchical approaches to variation in the process placement, and shows the impact of two typical process placements on the performance of the Broadcast and Allgather operations.

As such, more than raw performance, it is the difference between the same algorithm on different mappings that is of interest here. Considering the Broadcast (Figure 2(a)), one can see that hierarchical approaches (HierKNEM and MVAPICH2 both feature a hierarchical algorithm) achieve more stable performance. The default Open MPI collective implementation (Tuned) exhibits very unstable performance trends; for some message sizes the by-node binding reaches better performance, while it is the contrary for larger messages. Figure 2(b) further displays the importance of considering hierarchical features to enable portability of performance across varied process mappings. This clearly illustrates the penalty suffered by *topology-unaware* algorithms when considering irregular process-per-core bindings, and highlights the fact that topology aware algorithms can drastically improve the performance of collective algorithms independently of process placement.

2 Summary

This report outlined the project accomplishments at the University of Tennessee, Knoxville during the 3 years of this project period.

Figure 2: Impact of process mapping: aggregate Broadcast and Allgather bandwidth of the collective modules for two different process-core bindings: by core and by compute node (Paraplue cluster, IB20G, 768 processes, 24 cores/ compute node).