Optimizing Collective Operations in Hybrid Applications

Aurèle Mahéo
Exascale Computing Research
Versailles, France
aurele.maheo@exascale-computing.eu

Patrick Carribault
CEA, DAM, DIF
F-91297 Arpajon, France
patrick.carribault@cea.fr

Marc Pérache
CEA, DAM, DIF
F-91297 Arpajon, France
marc.perache@cea.fr

William Jalby
Exascale Computing Research
Versailles, France
william.jalby@uvsq.fr

ABSTRACT

The advent of multicore and manycore processors in clusters advocates for combining MPI with a shared memory model like OpenMP in high-performance parallel applications. But exploiting hardware resources with such models can be sub optimal. Thus, one approach is to use the hybrid context to perform MPI communications. In this paper, we address this issue with a concept of hybrid collective communications, which consists in using OpenMP threads to parallelize MPI collectives. We validate our approach on several MPI libraries (IntelMPI and MPC), improving the overall time up to a factor of 5.29×, in a real world application.

Keywords

MPI, Collective Communications, OpenMP

1. INTRODUCTION

Multicore processors are nowadays the building blocks of current supercomputers. The main trend is to increase the number of available resources within a computational node: cores and memory. But, even though the main memory becomes larger, the amount of memory per core is decreasing. This evolution may have a negative impact on the MPI based applications that exploit those clusters. For a better use of memory, one possible solution is to mix the MPI programming model with a thread-based shared-memory approach like OpenMP. However, there are multiple directions to include a thread-based model within an existing MPI program. The *master* approach [3] progressively augments the application with loop-level OpenMP directives. Within those regions, all available cores are exploited, but outside the OpenMP parts, only a subset of cores are used. We propose a solution to this lack of parallelism by focusing on MPI collectives. Indeed, such constructs are widely encountered in MPI applications and represent a non negligible percentage of total execution time. For example, the time spent in MPI_Allreduce construct may account for more than 50% of global execution time [2]. Several papers investigate the problem of idle resources in a hybrid context, and optimizations of reduction collective. [5] targets the problem of idle OpenMP threads in hybrid MPI + OpenMP applications, identifies these idle threads, and exploits them by several ways. In [4], authors present two variants to parallelize the computational part in MPI_Allreduce through shared memory. Each process divides its own block of elements into several sub blocks and reduction is performed. But the second flavor optimizes the memory traffic by shifting the work on each process on a cyclical way. Our contribution adapts the approach of [4] to optimize hybrid programs performing MPI collective communications outside OpenMP regions, and thus automatically exploit the idle cores.

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642791.

2. USING OPENMP IN MPI ALLREDUCE

Our approach consists in splitting the vectors of an MPI Allreduce operation in multiple chunks such as each available core will be in charge to reduce one chunk. Thus, each OpenMP thread would process a subset of the input vector by performing a smaller MPI reduction, leading to the parallelization of the computational and communication parts of the whole operation with independent operations.

Figure 1 depicts an example with an application running on a computational node containing four 4-core processors for a total of 16 cores. Let us consider now a parallel MPI + OpenMP application running with 4 MPI processes. The regular approach to place the MPI processes would be one per multicore processor, letting 4 cores per MPI rank to launch an OpenMP parallel region. The top part of Figure 1 illustrates the behavior of the application when performing an MPI_Allreduce communication outside any OpenMP region. Because this operation is not done inside a parallel block, only 4 cores will help processing this operation. It leads to a total of 12 idle cores. Our solution is described on the bottom part of this figure: it shows how the spare cores are used when performing an hybrid MPI_Allreduce operation. We can see that for each MPI rank, 4 OpenMP threads are launched, each working on its subset of the initial vector,

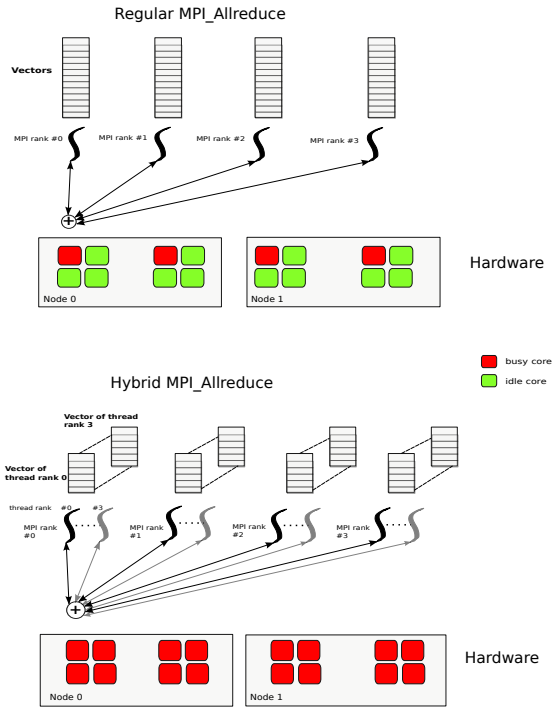

Figure 1: Hybrid MPI_Allreduce

and thus all cores of the node are active.

3. EXPERIMENTAL RESULTS

We implemented our concept in a wrapper, which captures calls to MPI_Allreduce collectives using the MPI Profiling Interface (PMPI). Our approach works with MPI runtimes supporting `MPI_THREAD_MULTIPLE` level and all OpenMP implementations. Moreover our design guarantees a full support of regular types, derived data types, and communicators passed to the MPI_Allreduce collective.

We evaluated our concept on 4 nodes of Curie supercomputer, containing 16 8-core CPU Nehalem-EX@2.27GHZ. Experiments were performed on a real-world MPI+OpenMP application (MC [2]) simulating particle interactions using Monte Carlo methods. It performs an MPI_Allreduce collective to update the number of available particles at each time step. We tested our hybrid approach on this 128KB operation with IntelMPI 4.1.3.048, MPC 2.4.1 [1] and BullxMPI 1.1.16.6. We fully populated the nodes with OpenMP threads, and tuned the number of threads for the hybrid MPI_Allreduce collective.

Figure 2 depicts comparative execution time of the MC application, on four 128-core nodes, with 1 MPI task per node. With MPC, best hybridization of MPI_Allreduce is obtained with 8 threads, giving 3.14× speedup. For IntelMPI, the best combination is obtained with 9 threads, leading to a speedup of 3.54×. At this point, we perform `numthreads` similar calls to MPI_Allreduce, `numthreads` being the number of used OpenMP threads. Since each thread has a copy of the same input communicator, they will all perform their own reduction following the same communication pattern. This scheme is likely to generate memory

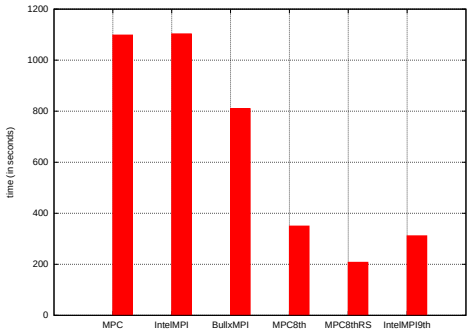

Figure 2: MC - Comparison between MPC, IntelMPI, BullxMPI, and best hybrid combination for MPC and IntelMPI (4 128-core nodes, 1 task per node)

contention. So we propose a variant of our approach: we give a different role to each thread, by shifting their rank in sub-communicators. It leads to an additional improvement of 68% with MPC (MPC8thRS on Figure 2), with a total speedup of 5.29×.

4. CONCLUSION AND FUTURE WORK

In this paper, we proposed a way to optimize MPI+OpenMP applications by parallelizing the collective operations using spare cores. Our approach is portable to any MPI implementation with `MPI_THREAD_MULTIPLE` support and any OpenMP runtime. Experimental results showed a speedup of 5.29× on a real world application. For future work, we plan to extend this work with heuristics to predict the best hybrid combination based on criteria such as vector length and hardware topology. Rank shifting should be improved, dealing with network topology and NUMA effects. Finally, this method can be applied and tuned to more scientific applications and clusters based on manycore processors such as Intel Xeon Phi.

5. REFERENCES

[1] P. Carribault, M. Pérache, and H. Jourdren. Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In *Proceedings of IWOMP'10*, 2010.

[2] D. Dureau and G. Poëtte. Hybrid parallelism models for neutron monte-carlo solver in an AMR framework. In *Proceedings SNA + MC 2013*, 2013.

[3] G. Hager, G. Jost, and R. Rabenseifner. Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of Cray User Group*, 2009.

[4] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *Proceedings of CCGRID'08*, 2008.

[5] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. MT-MPI: Multithreaded MPI for many-core environments. In *Proceedings of ICS'14*, 2014.