HAL
open science

Dynamic Placement of Progress Thread for Overlapping MPI Non-Blocking Collectives on Manycore Processor

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Marc Pérache, Hugo Taboada

To cite this version:

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Marc Pérache, Hugo Taboada. Dynamic Placement of Progress Thread for Overlapping MPI Non-Blocking Collectives on Manycore Processor. EURO-PAR 2018 : 24th International European Conferenceon Parallel and Distributed Computing, Aldinucci, Marco, Padovani, Luca, Torquati, Massimo (Eds.), Aug 2018, Turin, Italy. hal-01888255

HAL Id: hal-01888255

https://hal.science/hal-01888255v1

Submitted on 4 Oct 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.

Dynamic Placement of Progress Thread for Overlapping MPI Non-Blocking Collectives on Manycore Processor

Alexandre Denis[1], Julien Jaeger[2], Emmanuel Jeannot[1], Marc Pérache[2], and Hugo Taboada[1,2]

[1] Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, France
{alexandre.denis,emmanuel.jeannot,hugo.taboada}@inria.fr
[2] CEA, DAM, DIF, F-91297 Arpajon, France
{julien.jaeger,marc.perache}@cea.fr

Abstract. To amortize the cost of MPI collective operations, non-blocking collectives have been proposed so as to allow communications to be overlapped with computation. Unfortunately, collective communications are more CPU-hungry than point-to-point communications and running them in a communication thread on a dedicated CPU core makes them slow. On the other hand, running collective communications on the application cores leads to no overlap. To address these issues, we propose an algorithm for tree-based collective operations that splits the tree between communication cores and application cores. To get the best of both worlds, the algorithm runs the short but heavy part of the tree on application cores, and the long but narrow part of the tree on one or several communication cores, so as to get a trade-off between overlap and absolute performance. We provide a model to study and predict its behavior and to tune its parameters. We implemented it in the MPC framework, which is a thread-based MPI implementation. We have run benchmarks on manycore processors such as the KNL and Skylake and get good results for both performance and overlap.

1 Introduction

MPI is the standard interface for communications in HPC applications. It is used by applications for inter-node (i.e. network) and intra-node (processes on the same node) communications. The cost of communications is one of the main obstacles to get a good speedup for parallel applications. To amortize the cost of MPI communications, application programmers try to overlap communications with computation by using non-blocking communication primitives, and let them progress in background while keeping the CPU busy with computation.

Initially the non-blocking communications were only available for point-to-point communications. The extension of the non-blocking communications to collective operations (i.e. primitives that involve more than two nodes, such as broadcast, reduce, scatter, gather, ...) is an addition of the latest major MPI

version [1]. It opens the door to communication/computation overlap for collective operations too. However, collective communications are more CPU-hungry than point-to-point communications, and are therefore it is harder to make them progress in background.

In this paper, we tackle the problem of overlapping communication and computation for non-blocking collectives on manycore processors. We study the case of MPI tasks spread on a manycore processor, with one task per core, and how to improve overlap with cores dedicated to communications. We explore the trade-off between executing collective communication on dedicated CPU cores versus using application cores. We restrict ourselves to the case of tree-based collective operations (broadcast, reduce, scatter, gather, allreduce) because they are the one where this trade-off has the most impact on the performance as we are able to tune it dynamically.

In short, this paper makes the following contributions:

- we propose an *algorithm* that splits the tree of the collective operation, running parts of the tree on cores dedicated to communication, and parts of the tree on the application core;
- we propose a *model* for the above algorithm, so as to demonstrate the improvement of global performance when overlapping communication and computations, and to tune its parameters;
- we *implemented* the algorithm in the MPC MPI implementation [2].

The rest of the paper is organized as follows. Section 2 presents related works about communication/computation overlap in general, and for collective communication in particular. Section 3 presents our split-tree algorithm for tree-based collective communications. In Section 4, we present a model of the algorithm and how to tune it for optimal performance. Section 5 describes how the algorithm is implemented in the MPC software. Section 6 reports experimental results, and Section 7 concludes.

2 Related Works

The topic of communication progression has already been studied for some aspects in the literature. Several strategies do exist for background progression of point-to-point communications, such as offloading the communication to hardware [3, 4] and let the hardware do the progression; use of a thread [5] or process [6] dedicated to communication progression; opportunistic scheduling of communication tasks [7, 8].

MPI non-blocking collective communications are more difficult to make progress in the background, since not only the data transfer but the collective algorithm too needs to progress, which makes it harder to rely on hardware. There is specific work [9] for hardware-assisted progression on Blue Gene, or offloading shared memory collectives to a kernel module [10] (although authors only address performance of blocking collectives, not progression of non-blocking collectives). The reference NBC implementation [11] relies on a progression thread,

with some tricks [12] to improve overlap on InfiniBand. This approach is quite different from ours since it leads to one progression thread per MPI task, while our approach runs multiple MPI ranks in the same process and the algorithm for the collectives is shared across all MPI ranks in the same process.

3 A split-tree algorithm for MPI collective operations

In this Section, we propose a split tree algorithm for MPI collective communications which improves communication/computation overlap.

In this paper, we focus on intra-node communications on a manycore machine, with one MPI task per core. To obtain a good overlap for communications and computation, they have to run in parallel. On a manycore machine, the straightforward way to get background progression of communication is to dedicate some cores to communications, thus some cores host an MPI rank, we call them *application cores*; the remaining cores (one or several) host communication progression threads, we call them *communication cores*.

However, collective communication algorithms involve a huge amount of point-to-point communications, and thus a lot of communication tasks. When communication cores perform all communications on behalf of all application cores, the algorithm is *folded* and communications from a given step of the collective algorithm may be serialized. As a consequence, when folded on few communication cores, collective communications get much slower than when executed as a blocking call on all application cores simultaneously.

There are multiple topologies for collective communications. We restrict ourselves to *tree-based* algorithms (reduce, broadcast, gather, scatter, allreduce). The time steps of such a tree-based collective is depicted in Figure 1: each level of the tree is a step in the algorithm, from the root to the leaves. The rank of MPI tasks participating to each step is represented in the vertices. The left child of a vertex is the same MPI task; only the right child involves a communication. When represented as time steps of the algorithm, it is a binary tree, although when considering the data flow by deduplicating vertices which are the same task, the algorithm is really a binomial tree.

On such tree-based algorithms, we observe that the amount of work is very unbalanced in time and space. On the example depicted in Figure 1 for 16 MPI

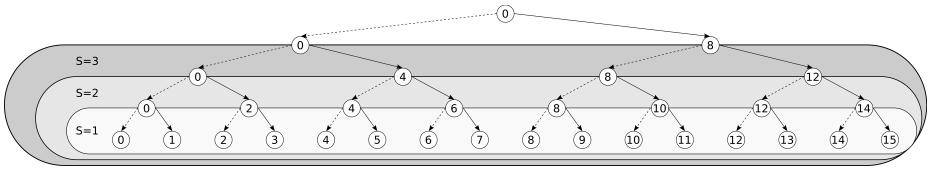

Fig. 1. Communication tree for a broadcast collective with 16 MPI tasks. S is the number of steps (tree levels) running on application cores. Plain edges are communications. Vertices are the MPI tasks.

tasks, there are 15 communication tasks and the algorithm needs 4 steps. If we fold these communications on a single communication core, it would need 15 steps which is 4 times slower. Since half of the work is in the last step, represented as $S = 1$ with levels numbered from the leaves, we can trade some performance against some overlap by executing different parts of the tree on different cores. If only the upper part of the tree is executed on the communication cores, and the last step $S = 1$ is executed on the application cores, then the total is twice as fast as running everything on communication cores, while only a single step cannot be overlapped with computation.

Our proposed algorithm is a generalization of this principle for a *trade-off between communication performance and overlap*: split the communication tree with the upper part running on communication cores, so as to have full overlap, and the lower part running on all application cores. Let S the number of steps (tree levels) running on application cores. $S = 0$ is equivalent to running all the communication on communication cores. The algorithm runs S steps of the tree on application cores as depicted in Figure 1. When $S = 1$, the algorithm runs the short but heavy part of the tree on application cores whereas the long but narrow part of the tree is running on one or several communication cores. All the communications running on application cores cannot be overlapped by computation because they are running on the same cores. However, this part of the tree is the heaviest and running these communications on few communication cores would jeopardize communication performance. The part of the tree running on communication cores benefits of total overlapping of its communications.

If S is increased, the algorithm loses a bit of its ability of being overlapped but can increase its absolute performance depending on the communication/computation ratio. We have to get a trade-off between overlap and absolute performance.

4 Modeling and tuning

In this Section, we propose a performance model of the algorithm described in Section 3, so as to show its relevance and to tune its S parameter.

Model for collective operations. Let N_{proc} the total number of cores, and N the number of cores for the application (i.e. number of MPI ranks), then the number of dedicated cores for communication is $P(N) = N_{proc} - N$.

We consider collective operations as binomial trees only. The proposed model could be easily extended to N-nomial trees if needed. It applies to operations such as: reduce, broadcast, gather, scatter; scan and alltoall, not based on a tree topology, are out of scope. We model communication cost as linear, neglecting latency and cache effects. We take as unit the point-to-point transfer time of one buffer of the size of the considered collective operation. We study first operations with a constant buffer size across the whole tree (reduce, broadcast). We will extend it to variable-buffer size operations (scatter, gather) in a second step.

The height of the tree[3] is $H(N) = \lceil \log_2(N) \rceil$. In the case of a blocking operation where communication is performed simultaneously by all application cores, we get the following execution time:

$$T_{blocking}(N) = H(N) = \lceil \log_2(N) \rceil \tag{1}$$

Let $C(N)$ the computation time on N nodes. To model computation and communication overlap, we consider the application programmer tried to reach perfect overlap and sized computation to have the same duration on all cores as the blocking collective operation, i.e. $C(N_{proc}) = T_{blocking}(N_{proc})$. If we assume computation scales linearly, we have the following time for computation on N nodes:

$$C(N) = \frac{N}{N_{proc}} \times C(N_{proc}) \tag{2}$$

Model for the proposed algorithm. We now model the split tree algorithm itself. As defined in Section 3, S is the number of steps running on application cores; the time to run these steps is the depth of the sub-trees, namely S, unless the tree height is smaller than S. The algorithm schedules operations from the upper $H(N) - S$ levels on communication cores, folded on $P(N)$ cores. Let $R(N) = N - 2^{\lfloor \log_2(N) \rfloor}$ the number of leaves that are not on the largest complete binary sub-tree of the tree. Let $F(N, i)$ the number of communications for N MPI tasks in the level i:

$$F(N, i) = 2^{\lfloor log_2(N) \rfloor - (H(N) - i + 1)} + \left\lfloor \frac{R(N) + 2^{(H(N)-i)}}{2^{(H(N)-i+1)}} \right\rfloor \tag{3}$$

Since each level of the tree contains $F(N, i)$ communications for level i numbered from 1 for the root, it takes a time of $\lceil F(N, i)/P(N) \rceil$ once folded on $P(N)$ communication cores, assuming each level is run in sequence because of communication dependencies. As a result, the time for a non-blocking collective with split steps algorithm is Equation 4 as below:

$$T_{non-blocking}(S, N) = \underbrace{\min(S, H(N))}_{\text{last } S \text{ steps from leafs}} + \underbrace{\sum_{i=1}^{\max(0, H(N)-S)} \left\lceil \frac{F(N, i)}{P(N)} \right\rceil}_{\text{upper levels of tree, up to } S} \tag{4}$$

With communication and computation overlap with the same collective operation, given that the part running on application cores cannot be overlapped and the part running on communication cores is fully overlapped, we get the result in Equation 5 as time for overlapped computation and non-blocking collective with split tree:

[3] We use a binomial tree where the N MPI tasks are leaves. In case of a binary tree, we will have N vertices and $H(N) = \lceil \log_2(N + 1) \rceil - 1$

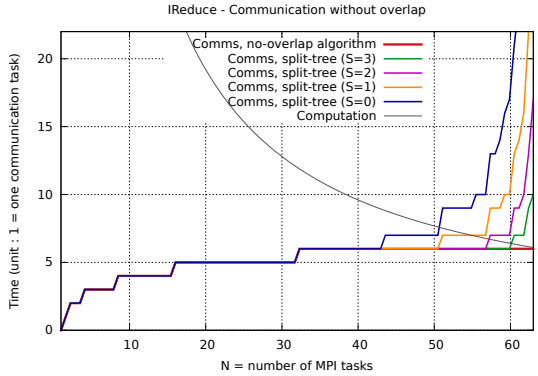

Fig. 2. Model of communication cost for operations with constant-size buffer (broadcast, reduce) on 64 cores.

$$T_{overlapped}(S,N) = \underbrace{\min(S,H(N))}_{\text{non-overlapable comms}} + \underbrace{\max\left(C(N), \sum_{i=1}^{\max(0,H(N)-S)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil\right)}_{\text{overlapable communications}} \tag{5}$$

The graph $C(N)$, $T_{blocking}$, and $T_{non-blocking}(N,S)$ for increasing values for S and $N_{proc} = 64$ is depicted in Figure 2. We observe that for large values of N (i.e. small number of communication cores), the communication is huge for $S = 0$ (all communication on communication cores). The cost decreases when S increases.

Figure 3 represents the total time of computation overlapped with communications when using blocking communications (computation and communication run in sequence) and when using non-blocking communications with split tree algorithm. We observe that increasing values for S increases the cost for small values of N (reduces overlap), but this cost is amortized for large values of N where the total time is dominated by the cost of the communication folded on few communication cores.

Discussion and tuning. From observation of Figure 3, the absolute minimum time is reached for $S = 0$ and $N = 51$. However, it means that 13 cores are dedicated to communications, which may not be desirable for the user since it would degrade performance of parts of the application without communication. With 7 cores dedicated to communications ($N = 57$), the optimal is $S = 1$; for 4 cores dedicated to communication ($N = 60$), the optimal is $S = 2$; and finally $S = 3$ for $N = 62$ (2 communication cores).

As a general case, for a given value of N, it is enough to compute the predicted performance with the model for a few values of S to find the optimal

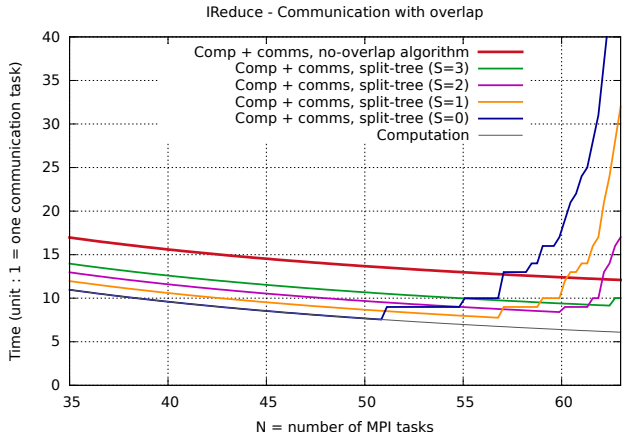

Fig. 3. Model of communication/computation overlap for operations constant-size buffer (broadcast, reduce) on 64 cores.

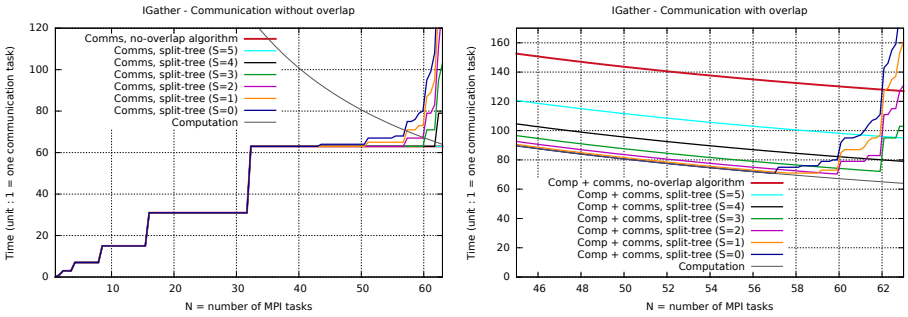

Fig. 4. Model of communication cost (left) and communication/computation overlap (right) for operations with increasing buffer size (scatter, gather) on 64 cores.

value. However finding N for the best overall performance depends on application scalability and communication/computation ratio and is out of scope for this paper. We can extend the proposed model for collective operations where not all tree edges have the same weight, such as scatter and gather; when going from leaves to root, data size doubles at each level of the tree. If we modify the model for such operations, we get the graphs for communication cost and overlapped time as depicted in Figure 4, which exhibits a behavior similar to the previous one.

5 Implementation

In this Section, we present the implementation of the algorithm in MPC [2], our thread based MPI implementation.

In MPC, MPI tasks are implemented with threads. MPC also implements POSIX threads and an OpenMP runtime system. MPC has its own scheduler allowing a fine-grained scheduling of all these threads. Thus, we bypass the system scheduler. MPC uses a tuned version of libNBC [11] to implement MPI 3 Non-Blocking Collectives. One progress thread is created for each MPI task. These threads can be bound through different algorithms. In the default behavior used in our experiments, MPI tasks are bound with a scatter policy and progress threads are bound to the closest idle cores.

In this implementation, a MPI non-blocking collective is decomposed in MPI point-to-point non-blocking calls fulfilling the collective algorithm. When a MPI non-blocking collective is called, each MPI task creates a *schedule* containing requests for the point-to point non-blocking calls corresponding to its part of the collective algorithm, and attach it to its associated progress thread. Thus, the progress threads handle the communication described by the schedules while MPI tasks continue to execute computation.

To implement our algorithms, we define the parameter S to be the number of steps (tree levels) that we want to run on application cores. For *all-to-one* algorithms (reduce, gather), we run the S steps on MPI tasks using MPI point-to-point blocking communication before creating the NBC schedule of $H(N) - S$ steps. Then, we attach it to its associated progress thread. Thus, the first part of the algorithm is running on application cores whereas the last part is running on the cores dedicated to the progress threads. For *one-to-all* algorithms (broadcast, scatter), we define the requests of $H(N) - S$ steps and create the NBC schedule first. We attach it in its associated progress thread. Then we implement the S steps in the MPI_Wait function executed by the MPI tasks. Hence, the first part is running on the cores dedicated to the progress threads whereas the last part is running on application cores.

6 Experimental Results

In this Section, we present experimental results of our algorithm implemented within MPC.

We implemented our own micro-benchmarking tool to evaluate the performance of our algorithm. This tool works similarly to the Intel MPI Benchmarks [13] but with fixed problem size allowing us to have the same computation workload for different number of MPI tasks. We arbitrary set the buffer size to 2MB and sized the computation workload to reach perfect overlap Then, we reduce the number of MPI tasks while keeping the same global computation workload. Thus, when we have less MPI tasks, the duration of computation increases and more idle cores are available for progress threads. This contributes to decreasing the time of communications and maximize the overlap. When all cores are used by the MPI tasks, they are no cores left for progress threads. In this case, the algorithm is the same as for the blocking call. Thus we do not show these points in the following performance figures.

We ran our benchmark on two different manycore architectures: a 1.4GHz Intel Xeon Phi Knights Landing with 64 cores (KNL) and a 2.7GHz bi-socket Xeon Platinum Skylake with a total of 48 cores (SKL).

Comparing split-tree algorithm to default setup. In our first experiments, we tested the interest of the split-tree algorithm. As described in Section 5, MPC already provides progress threads for communication collectives. The progress threads are gathered on the available cores. This mapping brings good performances when the number of available cores is high. However, performances collapse when too many progress threads are gathered on the same core. The blue lines labeled "Comp + comms, split-tree (S=0)" show this behavior on KNL for collective Ibcast (Figure 5) and for collective Ireduce on KNL and Skylake (Figure 6). The label "Comp + comms, split-tree (S=0)" means that no level of the communication tree is done on the MPI tasks, thus all communications are realized on the progress threads.

Thanks to the split tree algorithm, we were able to balance more efficiently communications between the MPI tasks and the progress threads. The orange line labeled "Comp + comms, split-tree (S=1)" (resp. purple line labeled "Comp + comms, split-tree (S=2)" and green line labeled "Comp + comms, split-tree (S=3)") shows the performance of the same algorithms when 1 (resp. 2 and 3) levels of the communication tree remains on the MPI tasks. If enough cores are available to correctly handle the progress threads, the split-tree version is less performant. However, when the number of available cores is shrinking, the split-tree version is more stable. For each additional level attached to the MPI tasks, the sudden performance drop is observed with fewer available cores, until S=3 allows to maintain better performances than the blocking call even in the least favorable case (only one core available for all progress threads). Hence, it is possible to select the best split-tree value S depending on the algorithm and the number of cores hosting progress threads.

Comparing performance results to model. To help select the number of tree levels to leave on the MPI tasks, we proposed a model in Section 4. The model projection for Ireduce collective on 64 cores is shown in Figure 3. Comparing this projection to the result of Ireduce on the 64 cores KNL displayed in Figure 6, we can see that the model is really close to the results.

Moreover, the values for switching from a value S in the split-tree to the next one are the same between the prediction and the measured performance. This allows us to select the correct number of levels to leave on the MPI tasks by implementing this model in the MPI runtime system.

Comparing MPI implementations. We also compare our algorithm with other MPI implementation such as Intel-MPI and OpenMPI. We ran OpenMPI and Intel-MPI tests with the same compute workload as for our previous experiments. We compare these results to our split-tree algorithm with the S value chosen accordingly to our model. Hence, when the model predicts that an S value is better than another one, this value is automatically applied. For example, on

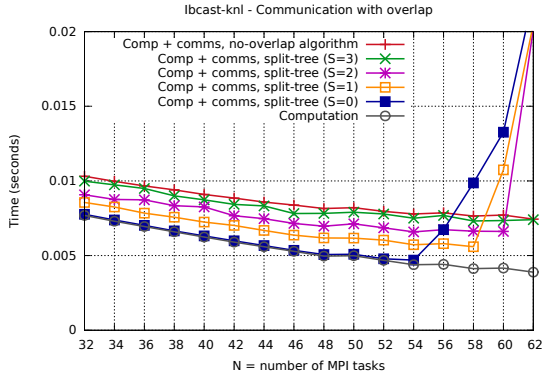

Fig. 5. Result of split-tree algorithm with different values of S, for MPI_Ibcast with constant-size buffer of 2MB on 64 cores (KNL).

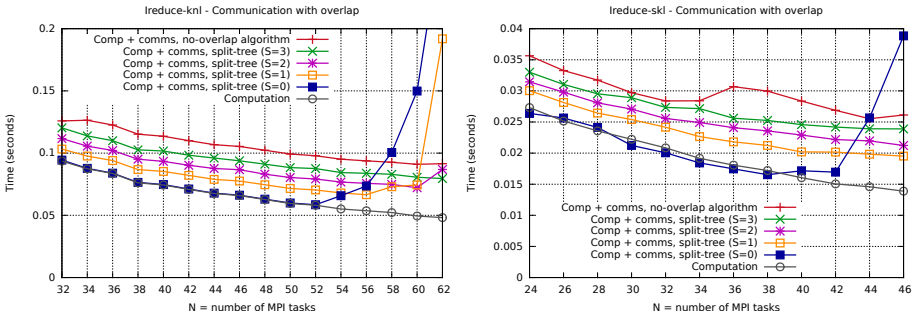

Fig. 6. Result of split-tree algorithm with different values of S, for MPI_Ireduce with constant-size buffer of 2MB on KNL (left) and Skylake (right) processors.

KNL, we switch from S=0 to S=1 for 52 MPI tasks, from S=1 to S=2 for 58 MPI tasks, and from S=2 to S=3 for 62 MPI tasks.

The results for all tested MPI implementation, including our MPC model-based results, are depicted in Figure 7 for MPI_Ireduce.

We observe that our split-tree algorithm, with the selection of the number of levels left on the MPI tasks based on our model (MPC model-based – green), performs well on KNL and Skylake. On KNL, MPC model-based (green lines) is always better than OpenMPI (purple) and IntelMPI (royalblue). To be fair, we activated for IntelMPI the flags allowing asynchronous progression (*I_MPI_ASYNC_PROGRESS* and and *I_MPI_ASYNC_PROGRESS_PIN*), but these flags reduced the performances (skyblue and blue lines) instead of improving them. On Skylake, OpenMPI performs better than on KNL. However, except for last number of MPI tasks, MPC model-based managed to have better performance thanks to the split-tree algorithm.

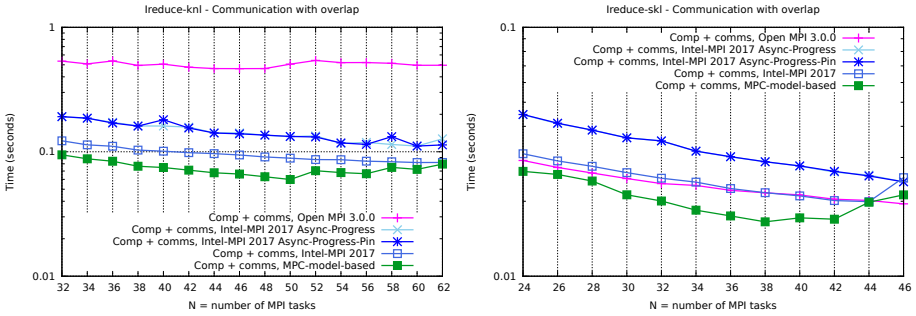

Fig. 7. Result of multiple MPI implementation for MPI_Ireduce with constant-size buffer of 2MB on KNL (left) and Skylake (right) processors (Y-axis in log scale).

Very interestingly, we also see that in this case, the best performance is obtained with 50 cores for the KNL and 38 cores for the SKL, meaning that the best trade-off is far from using all the available cores.

7 Conclusion and Future Work

Overlapping communications with computation is the key to amortize the cost of communications, especially for collective communications which are heavier than point-to-point communications. Approaches for progression relying on a progression thread per task suffer from competition between communication and computation, and approaches relying on a pool of cores dedicated to communication exhibit a slowdown in pure communication time when the collective is folded on few cores.

In this paper, we have proposed a novel algorithm that combines the best of both worlds. It splits the communication tree so as to execute the narrow part of the tree, representing most of its depth, on dedicated communication cores; this part may be fully overlapped with computation. It places the widest part of the tree, which represents a small part of its depth but a large part of the total work, on all applications cores to benefit from parallelism.

We have modeled the algorithm to demonstrate its relevance and to tune its parameter. We have implemented the algorithm in the MPC software and evaluated its performance on manycore processors (Intel KNL and Skylake). Thanks to the excellent accuracy of the model we are able to almost always find the best trade-off between using dedicated CPU cores or application cores and hence exceed the performance of state-of-the-art competitors. Moreover, it is important to notice that our solution is not bound to the MPC runtime system but can be implemented in any MPI library featuring progress threads for communication.

As future work, we plan to extend the approach of our algorithm to inter-node communications, which have a different behavior than intra-node communications considered in this paper. Moreover, we also plan to extend auto-tuning to

choose the number of MPI tasks (parameter N) to optimize the overall performance and not only sections with non-blocking collectives.

References

1. MPI Forum: MPI: A Message-Passing Interface Standard Version 3.0 (September 2012)
2. Pérache, M., Jourdren, H., Namyst, R.: MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In Springer, ed.: the 14th International Euro-Par Conference. Volume 5168 of LNCS., Las Palmas de Gran Canaria, Spain (August 2008) 78–88
3. Sur, S., Jin, H., Chai, L., Panda, D.: RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM New York, NY, USA (2006) 32–39
4. Rashti, M.J., Afsahi, A.: Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects. In: High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on, IEEE (2008) 95–101
5. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? In: Proceedings of the 2008 IEEE International Conference on Cluster Computing, IEEE Computer Society (Oct. 2008)
6. Lai, P., Balaji, P., Thakur, R., Panda, D.: ProOnE: A General Purpose Protocol Onload Engine for Multi- and Many-Core Architectures. (June 2009)
7. Denis, A.: pioman: a pthread-based Multithreaded Communication Engine. In: Euromicro International Conference on Parallel, Distributed and Network-based Processing, Turku, Finland (March 2015)
8. Si, M., Peña, A., Balaji, P., Takagi, M., Ishikawa, Y.: MT-MPI: multithreaded MPI for many-core environments. In: Proceedings of the International Conference on Supercomputing. (06 2014)
9. Almási, G., Heidelberger, P., Archer, C.J., Martorell, X., Erway, C.C., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of MPI Collective Communication on BlueGene/L Systems. In: Proceedings of the 19th Annual International Conference on Supercomputing. ICS '05, New York, NY, USA, ACM (2005) 253–262
10. Ma, T., Bosilca, G., Bouteiller, A., Goglin, B., Squyres, J.M., Dongarra, J.J.: Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs. In IEEE, ed.: 40th International Conference on Parallel Processing (ICPP-2011), Taipei, Taiwan (September 2011)
11. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07, IEEE Computer Society/ACM (Nov. 2007)
12. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop. (Apr. 2008)
13. : IMB-NBC benchmarks. https://software.intel.com/fr-fr/node/561946 Accessed: 2018-05-10.