Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 2403–2407

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

High Performance and Enhanced Scalability for Parallel Applications using MPI-3's non-blocking Collectives

Surendra Varma Pericherla and Sathish Vadhiyar

Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
surriennddraah@gmail.com,vss@cds.iisc.ac.in

Abstract

Collective communications occupy 20-90% of total execution times in many MPI applications. In this paper, we propose strategies for automatically identifying the most time-consuming collective operations that also act as scalability bottlenecks. We then explore the use of MPI-3's non-blocking collectives for these communications. We also rearrange the codes to adequately overlap the independent computations with the non-blocking collective communications. Applying these strategies for different graph and machine learning applications, we obtained up to 33% performance improvements for large-scale runs on a Cray supercomputer.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: MPI Collectives, non-blocking collectives, communication-computation overlap

1 Introduction

Collective communications consume significant percentage of execution times in many applications also influence the overall scalability trend of the applications. In this paper, we explore the use of non-blocking collective communications in improving the performance and scalability of different applications. Table 1 shows the different parallel applications considered for our study. As shown, collective communications occupy a major percentage of time in many of these applications.

We first identify the scalability and performance bottlenecks in the applications using HPCToolkit [12]. We use the work by Coarfa et al. [6] that uses scalability metrics in HPCToolkit for identifying the bottlenecks. We modified the collective causing bottleneck to use MPI-3's non-blocking collective communication, and followed a simple approach of overlapping all non-dependent computations and communications between the post and the complete operations. Our studies with large scale runs on a Cray supercomputer show that applying this approach gives significant performance improvement of upto 33% for the applications.

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.199

SNo.	Application	Most time-consuming Collective(s)	% Time consumed
1	Graph 500 [1]	MPI_Allreduce, MPI_Isend + wait	97.38
2	Triangle counting (based on [3])	MPI_Bcast	68.49
3	Traveling salesman problem [15]	MPI_Bsend, MPI_Barrier	86.94
4	Cycle detection in graphs (based on [4])	MPI_Bcast, MPI_Gather	96.72
5	Degree centrality (based on [7])	MPI_Allreduce, MPI_Bcast	78.3
6	Diameter and radius (based on [5])	MPI_Allgather, MPI_Bcast	16.77
7	Quick sort [13]	MPI_Gather	99.64
8	K-means [10]	MPI_Allreduce	44.3
9	piSVM [2]	MPI_Bcast	85.3

Table 1: Applications and Bottleneck due to Collective Communications

2 Related Work

Non-blocking point-to-point communication calls have been widely used for overlapping computation and communications. Bamboo [14] is a custom source-to-source translator that automatically overlaps communication with computation by using split-phase coding. Kandalla et al. [9] proposed scalable designs for non-blocking neighborhood collective operations based on InfiniBand's network-offload feature and applied in 2D BFS algorithm in CombBLAS to reduce communication overheads. Recently, MPI-3 standards introduced non-blocking collective communication calls. The non-blocking collective operations achieve efficient usage of CPU clock cycles by overlapping of communication and computation [8]. Our implementation performs a transformation which makes use of MPI-3 non-blocking collective communication calls.

3 Methodology

Identifying Scalability Bottlenecks: In this work, we have used HPCToolkit to identify scalability bottlenecks. HPCToolkit is more powerful than other tools due to its code centric and data centric capabilities [11] which helps to pinpoint scalability bottlenecks with low time and space overhead. However, HPC Toolkit does not directly help identify collective communications that act as scalability bottlenecks. We achieve this by collecting multiple call path profiling results from the toolkit for different number of processors.

We have used the value computed in in Equation 1, explained in [6], as a derived metric to compute the percentage of scaling losses.

$$X_s(C, n_q) = qC(n_q)pC(M_{qp}(n_q))/qT_q \tag{1}$$

where, C(n) denotes the cost incurred for a node n in a CCT (Calling Context Tree), $M_{qp}(n)$ is the mapping between corresponding nodes n_q in the CCT on q processors and n_p in the CCT on p processors, and qT_q is the total work performed in the experiment R_q.

By this work, experiments are performed for an application on p and q processors respectively, $p < q$. Using this, we were able to find scalability bottlenecks corresponding to these two processor profiles. However, there are cases where performing experiments on only two different number of processors are not sufficient in order to compute all possible metrics influencing scalability. Hence, we have have extended the work in [6] to identify all possible scalability impacting parameters for different ranges of processor cores. We have performed our experiments on 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 cores and have considered strong scaling. We have divided these cores into small cores

($sc = 32$ to 256 cores) and large cores ($lc = 512$ to 8192 cores). Our strategy is as follows: We first choose p and q from sc and use the above mentioned formula to find the set of scalability impacting parameters, B. We then repeat with p from sc and r from lc to find the scalability impacting parameters for this $< p, r >$ pair, and union them with the set B to find all the scalability impacting parameters.

MPI-3 Non-blocking Collectives using Code Rearrangements: Similar to non-blocking point-to-point communications, MPI-3 provides a post and completion operations for collective communications. For example, the non-blocking version of MPI_Bcast is:
$MPI\_Ibcast(buffer, count, datatype, root, comm, request)$ for posting the collective, and a
$MPI\_Wait(request, status)$ for completing the broadcast.

We organize program statements into five types w.r.t collective communication that results in scalability bottleneck. They are as follows: 1. Pre-collective communication independent statements: These are statements before the collective communication on which the collective is not dependent. 2. Pre-collective communication dependent statements: These are statements before the collective communication on which the collective is dependent. 3. MPI Blocking collective communication call. 4. Post-collective communication dependent statements: These are statements after the collective communication on which the collective is dependent. 5. Post-collective communication independent statements: These are statements after the collective communication on which the collective is independent.

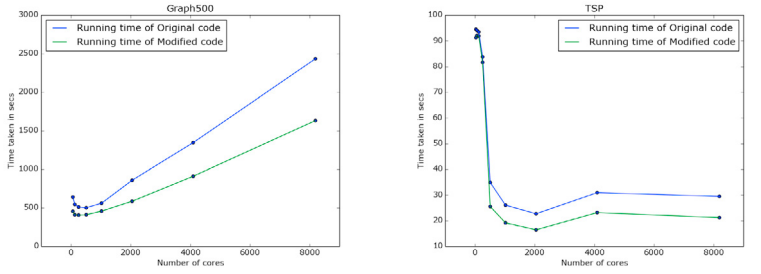
We replace the blocking collective communication with the posting of non-blocking collective call by code rearrangements. The independent statements in 1 and 5 are considered for overlapping with the collective communication.

4 Experiments and Results

We performed our experiments in our Institute's Cray XC40 cluster located in and maintained by Supercomputer Education and Research Centre (SERC). It has three kinds of nodes. It has 1468 CPU-only nodes with each node consisting of dual Intel Xeon E5-2680 v3 (Haswell) twelve-core processor at 2.5 GHz for a total of 35232 CPU cores. Each node has 128 GB memory with Cray Linux environment as the OS. The nodes are connected by Cray Aries interconnect using DragonFly topology. We used the nine applications mentioned in Table 1. In addition to measuring performance improvements due to non-blocking collectives, we also measure the overlap of the computations with the non-blocking communications due to our code restructuring. We find the time taken for the collective communication from the time the non-blocking collective is posted till the time the wait completes. We denote this time as t_{comm}. We also find the time for the independent computations that are overlapped with the communications, t_{comp}, by measuring the time from the return of the non-blocking post operation till before the wait is called. The computation-communication overlap percentage is then calculated as t_{comp}/t_{comm}.

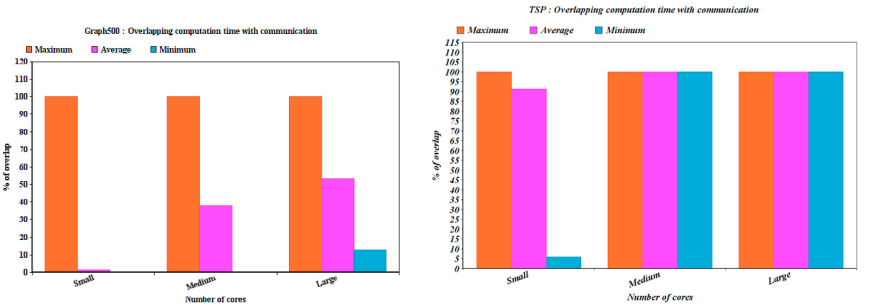
Graph500: Graph500 benchmark [1] includes a scalable data generator which produces edge tuples containing the start vertex and end vertex for each edge. The first kernel constructs an undirected graph and the second kernel performs a breadth-first search of the graph. Both kernels are timed. Graph500 uses two parameters to define the problem size, namely, $SCALE$ and $Edgefactor$. The graph size is such that the number of vertices is 2^{SCALE} and the number of edges is $Edgefactor * 2^{SCALE}$. In our experiments, we used $SCALE = 20$ and $Edgefactor = 16$ corresponding to graphs of 1 million vertices and 16 million edges.

Using our scalability analysis, we found that $MPI\_Allreduce$ is the most consuming collective, occupying 16.11-93.85% of the overall execution time. This $MPI\_Allreduce$ is mainly used to test globally whether all the queues are empty for performing BFS. Non-blocking point to point communication primitives such as MPI_Isend, MPI_Irecv, MPI_Wait and MPI_Test account for major amount of time for larger number of cores. They occupy only 2.88-31.17% of the total time for up to 1024 cores,



(a) Graph500 (b) TSP

Figure 1: Performance Improvement due to Non-Blocking Collectives



(a) Graph500 (b) TSP

Figure 2: Percentage of Communication Overlapped with Computation

but occupy 52.70-82.73% for more than 1024 cores. Interestingly, we find that $MPI\_Allreduce$ or non-blocking point-to-point communications individually does not capture the scalability trend adequately. Our strategy for finding scalability bottlenecks identifies (MPI_Allreduce+non-blocking point-to-point communications) combination as impacting scalability.

We then applied our non-blocking collective based code rearrangement technique on $MPI\_Allreduce$. We obtained up to 32.93% improvement as shown in Figure 1(a). In Figure 2(a), we can observe that the average percentage overlap increases, up to 53.46%, with increasing number of cores, resulting in increase in performance improvement with increasing number of cores.

Traveling Salesman Problem: The parallel version of TSP [15] partitions the search tree using breadth-first search. Then each process searches its assigned subtree and reuses the deallocated tours. Finally, the best tour structure is computed and broadcasted. The input is the number of cities, and the cost of traveling between the cities organized as a matrix.

For our experiments, we used as input a cost matrix of size 20X20 and is randomly generated within the code. We found using our scalability analysis that MPI_Barrier is the most time consuming collective, occupying about 41.67-63.45% of the overall execution time. This barrier is used to synchronize processes during parallel tree search. MPI_Bsend occupied only 0-1.84% of the total time for up to 1024 cores, but occupies 16.59-23.52% for more than 1024 cores. Hence, our strategy for finding scalability bottlenecks identifies ($MPI\_Barrier + MPI\_Bsend$) combination as impacting scalability. They ac-

count for up to 86.94% of the total time. The combination also shows the same scalability trend as the overall application, as shown in the figure.

After applying our proposed non-blocking collective based code rearrangement technique on MPI_Barrier, we obtained percentage of performance improvement up to 26.47% as depicted in Figure 1(b). The MPI_Bsend is not replaced by its non-blocking collective call as there are no pre and post independent statements for MPI_Bsend. In Figure 2(b), we can observe that the average percentage of overlap of computations with communications is high for medium and large number of cores than the small number of cores. Hence, we obtained better performance improvement at large and medium number of cores than at small number of cores as shown in Figure 1(b). In general, the communication-computation overlap is 6.00-100%.

We performed similar experiments for other applications, and obtained performance improvement of 18-33% due to our scalability analysis and use of non-blocking collectives, with 10-100% average computation-communication overlap. In the future, we plan to explore different ways of reducing the times due to collectives, including developing hierarchical collectives for the Cray's Dragonfly network topology.

References

[1] *Graph500*. Available at `http://www.graph500.org/`.

[2] *piSVM*. Available at `http://pisvm.sourceforge.net`.

[3] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. In *European Symposium on Algorithms*, pages 354–364, 1994.

[4] Noga Alon and Uri Zwick. Finding simple paths and cycles in graphs. 2002.

[5] Ishwar Baidari, Ravi Roogi, and Shridevi Shinde. Algorithmic approach to eccentricities, diameters and radii of graphs using dfs. *International Journal of Computer Applications*, 54(18), 2012.

[6] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. Scalability analysis of spmd codes using expectations. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 13–22, 2007.

[7] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[8] Torsten Hoefler, Jeffrey M Squyres, Wolfgang Rehm, and Andrew Lumsdaine. A case for non-blocking collective operations. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 155–164, 2006.

[9] K Kandalla, Aydin Buluç, Hari Subramoni, Karen Tomko, Jérôme Vienne, Leonid Oliker, and Dhabaleswar K Panda. Can network-offload based non-blocking neighborhood mpi collectives improve communication overheads of irregular graph algorithms? In *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pages 222–230. IEEE, 2012.

[10] Wei keng Liao. *Parallel K-Means Data Clustering*. Available at `http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html`.

[11] Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2013.

[12] John Mellor-Crummey. *HPC Toolkit*. Available at `http://hpctoolkit.org/`.

[13] Monismith. *Parallel Quick sort Implementation*. Available at `http://monismith.info/cs599/examples.html`.

[14] Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan, and Scott B Baden. Bamboo: translating mpi applications to a latency-tolerant, data-driven form. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 39, 2012.

[15] Peter S Pacheco. *Parallel programming with MPI*. 1997.