EasyChair Preprint

№ 8087

Root Causing MPI Workloads Imbalance Issues via Scalable MPI Critical Path Analysis

Maksim Fatin, Artem Shatalin and Vitaly Slobodskoy

May 24, 2022

Root causing MPI workloads imbalance issues via scalable MPI Critical Path analysis

Maksim Fatin[1], Artem Shatalin[1] and Vitaly Slobodskoy[1]

[1] Huawei

Abstract. Analyzing performance of MPI application usually requires non-trivial approaches. Classical hotspot-based analysis is often misleading for such an applications because hotspots optimization might not actually cause any speedup, but just increase the time ranks spent on waiting for each other.
One of the solutions is representing MPI program as a graph (known as Program Activity Graph) and perform only analysis of activities on Critical Path of this graph (the longest path containing computation and communication, but not waiting). Reducing computing time on Critical Path obviously reduces elapsed time of the whole application. While there are many papers in this area, Critical Path analysis representation in well-known performance tools is still quite limited. One side of this is that real-life HPC applications running on large scale produce huge Program Activity Graphs and scalability of classical graph algorithms is quite poor to calculate Critical Path reasonably fast. Another one relates to the limited capabilities performance tools provide based on Critical Path using timing information only. This paper describes an algorithm of building Program Activity Graph and calculating Critical Path which naturally scales to the same amount of CPU cores as profiled MPI application uses. We also show how to combine Critical Path analysis with Performance Monitoring Unit (PMU) data to enable efficient root causing of MPI imbalance issues even on very high scale.

Keywords: MPI, Imbalance, Critical Path, PMU.

1 Introduction

Developing parallel scalable applications is a very challenging task and its complexity grows significantly with the growth of scale. One of the poor scaling reasons is imbalance of work assigned to the hardware resources. Using MPI, load imbalance can be detected by various tools measuring amount of time spent within MPI API calls, however it is very hard to root cause the reason of imbalance – for example, different MPI ranks might execute different amount of job, the slowdown might be caused by specific CPU Microarchitecture issues happening within dedicated ranks only, the communication path between particular ranks takes longer than for others, etc. All these cases cause increase the time spent within MPI API functions waiting on implicit barriers.

The traditional performance analysis methodology is based on statistical sampling – an interrupt is triggered every N events (or just time – milliseconds), instruction pointer (address) is captured for the interrupted thread, N is attached for this sample as performance data (event count). Finally it can be represented as a list of hotspot functions (modules, threads, etc) aggregating event counters across all the samples and grouping by various things (e.g. function name, source line number, etc). This analysis usually provides quite detailed and precise information about places in the code where CPU spent its time, however, it might not be a surprise to see spin locks within MPI runtime as top hotspots quite often.

MPI program can be represented as a graph (known as Program Activity Graph) to analyze only activities on Critical Path of this graph (the longest path containing computation and communication, but not waiting). Reducing computing time on Critical Path obviously reduces elapsed time of the whole application.

In order to simplify the analysis of MPI workloads, we propose to combine PMU-based data with Critical Path analysis to show only hotspots on Critical Path naturally representing the root causes of imbalance and minimize amount of useless data (e.g. there is no wait time on the Critical Path by definition). We show how to find Critical Path in a fast and reliable way performing scalable calculation right on the MPI cluster used for the application execution. We demonstrate the usefulness of data retrieved as result of this performance tuning approach.

2 Prior State of the Art

Algorithms for finding Critical Path in the Program Activity Graph ([1], [2]) related to the Parallel MPI applications have been described in [1]-[4]. Possible optimizations to reduce graph size on collection time are described in [1], however the overhead claimed exceeds 8%. It also proposes further reduction steps on post-processing stage, but scalability of this step is not clear.

Known solutions suggest to preprocess data in the runtime in order to reduce post-processing data time, however this increases collection overhead. The scalability of the post-processing step is quite poor or not specified.

In this paper we propose a solution for constructing Critical Path with less than 5% of collection overhead and less than 10% of application elapsed time spent on post-processing independently on the number of ranks maintaining a good scaling of the algorithm.

3 Root causing MPI Imbalance issues

There are many tools measuring MPI Imbalance Time for the application execution, however they usually do not provide any clues about the reasons of imbalance. Consider the following hotspots data:

```
Function                       Module                 CPU Time(s)
-------------------------------------------------------------------------
hmca_bcol_basesmuma_bcast_k...  hmca_bcol_basesmuma.so   2838.6540
```

uct_dc_mlx5_iface_progress_ll	libuct_ib.so.0.0.0	2795.7913
gomp_barrier_wait_end	libgomp.so.1.0.0	2396.5551
uct_mm_iface_progress	libuct.so.0.0.0	2327.4292
opal_progress	libopen-pal.so.40.30.1	2090.3369

There is a significant MPI Imbalance Time in the application causing spinning from within MPI runtime to take the leading position in the hotspots data. However, analyzing spinning functions is a wrong way since we have to determine why there is so much waiting in the ranks.

Grouping of PMU samples on critical path allows to naturally filter out all the spinning (since there is no wait time on critical path by definition) and get only meaningful information:

Function	Module	CPU Time(s)
__mapz_module_MOD_ppm2m	cesm.exe	1977.2391
__clubb_intr_MOD_clubb_tend...	cesm.exe	1969.3700
memcpy	libc-2.28.so	1342.6702
__physics_types_MOD_physics...	cesm.exe	1200.6958
__mo_nln_matrix_MOD_nlnmat	cesm.exe	859.0117

4 Finding Critical Path in Program Activity Graph

First of all, we have to trace all the MPI calls executed from within application in all the ranks. This is usually done via wrapping MPI function using LD_PRELOAD and PMPI interface in order to preserve the data about API name, begin/end timestamps, data size and other data required on post-processing step. There are few important things to note:

1. MPI functions are matched according to id of communicator provided within function argument.
2. After non-blocking MPI function PMPI_Test is executed to detect the situation of immediately execution (needed for optimization).
3. Communicator creation APIs are carefully handled as well in order to maintain communicator IDs.

Applying known approaches with construction of Program Activity Graph in one place and then apply standard Critical Path finding algorithms (e.g. Dijkstra) is not going to scale due to its synchronization and dependency issues. Instead, we don't actually construct the whole Program Activity Graph, instead we record all the MPI API called and replay all these calls in some dedicated form to find Critical Path.

Algorithm has 3 main steps:

- Replay Point-to-Point (P2P) calls to exchange information about both parties of P2P calls
- Create critical path edges for all the MPI calls

- Restore critical path by edge

Let introduce some terms:

- ICF – Immediately Call Function. This is either a blocking MPI function, non-blocking MPI function with further MPI_Wait function call or non-blocking MPI function that was processed immediately (further PMPI_Test returned true).
- WALLS – wait all strategy which allows to replay MPI_Waitall behavior optimally. The idea is to create an array of requests of all the non-blocking MPI functions that were not immediately completed (further PMPI_Test returned false) and also create an array MPI_Irecv results. Then call PMPI_Waitall and store requested data to use it in the next algorithm steps.

Step 1:

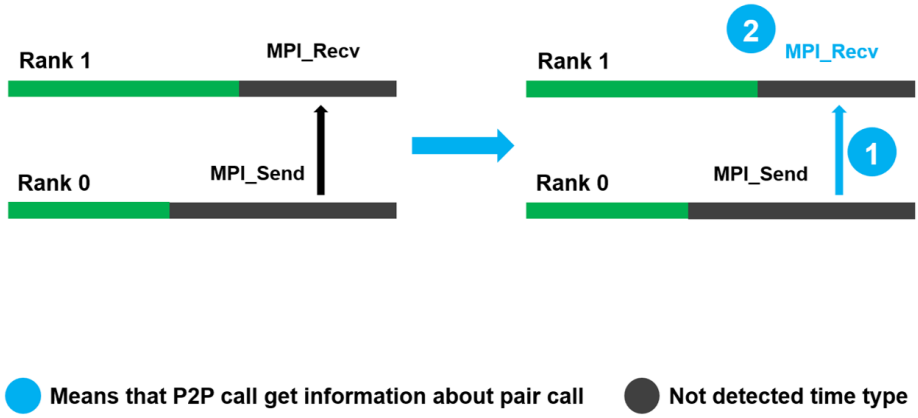

Fig. 1 – Step 1

Repeat ICF calls and use WALLS algorithm to repeat P2P calls and send data (Figure 1). Figure 2 illustrates how MPI_Recv gets information about pair call. Algorithm stores information about rank 0 (MPI_Send) in rank 1 and then sends information about MPI_Recv from rank 1 to rank 0.

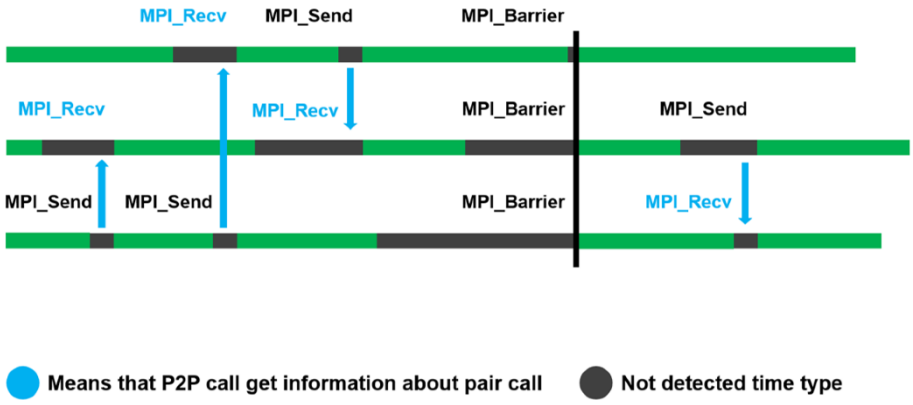

Fig. 2: result of the first step

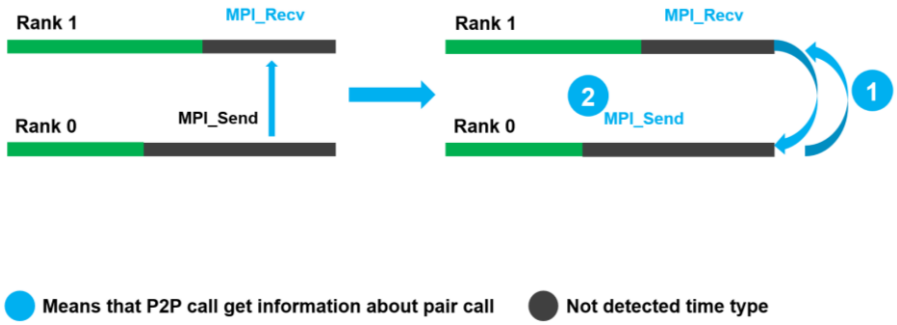

Figure 3

The second stage is transfer information about MPI_Recv function to the corresponding rank (Figure 3, step 1). After that we got information about pair call in MPI_Send function (Figure 3, step 2).

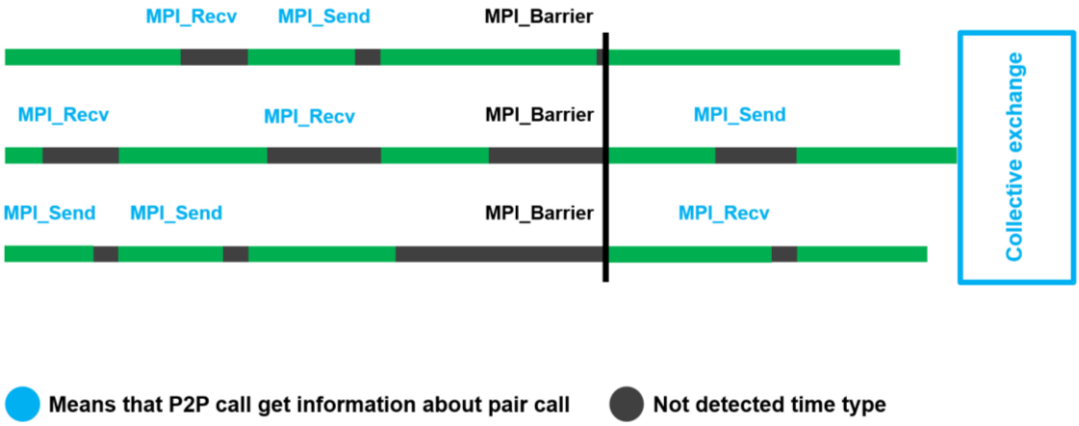

Fig. 4

At the end, all the P2P calls have information about their pair (Figure 4).

Step 2

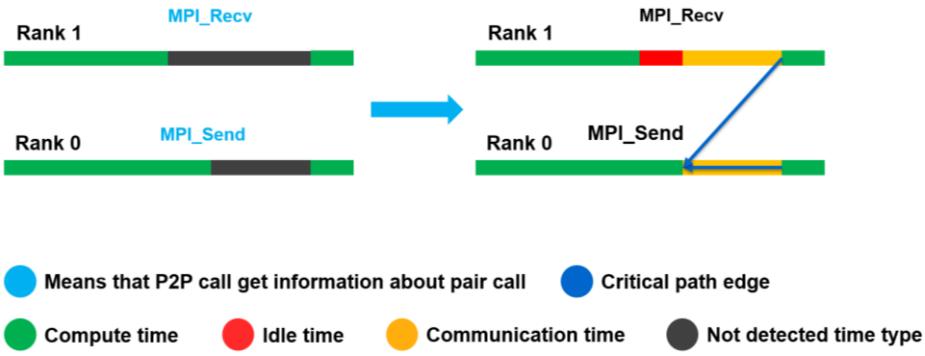

Fig. 5

Algorithm in this step does not call any P2P function, it only repeats collective call. When the program reads ICF from the file, it is decided in each rank to which rank edge has to be made (Figure 5).

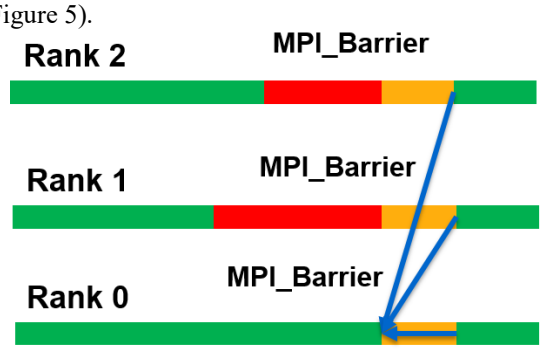

Fig. 6

When collective function is read from file, PMPI_Allreduce with MPI_MIN is executed to find out collective call segment with the lowest duration and ranks with the duration bigger than minimal mark time. Ranks create critical path edge to the rank that has a minimal length of MPI collective call (Figure 6).

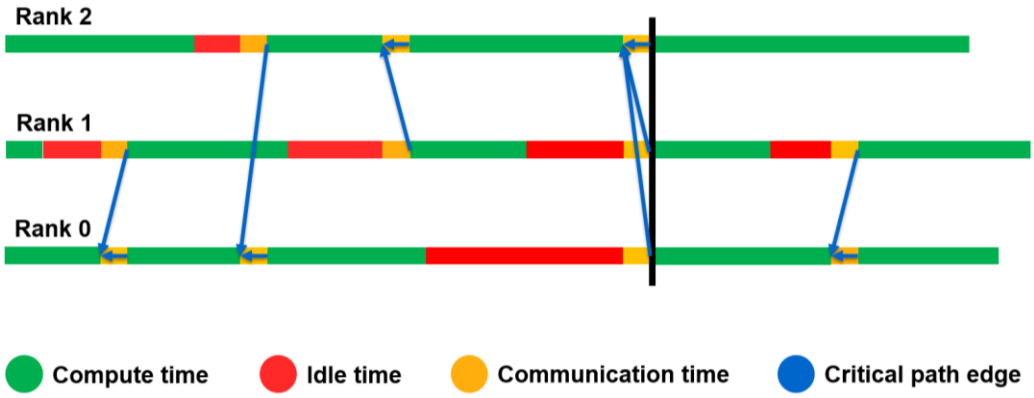

Fig. 7

The result for the whole graph is illustrated on Figure 7.

Step 3

In this step write critical path segments into each rank file locally.

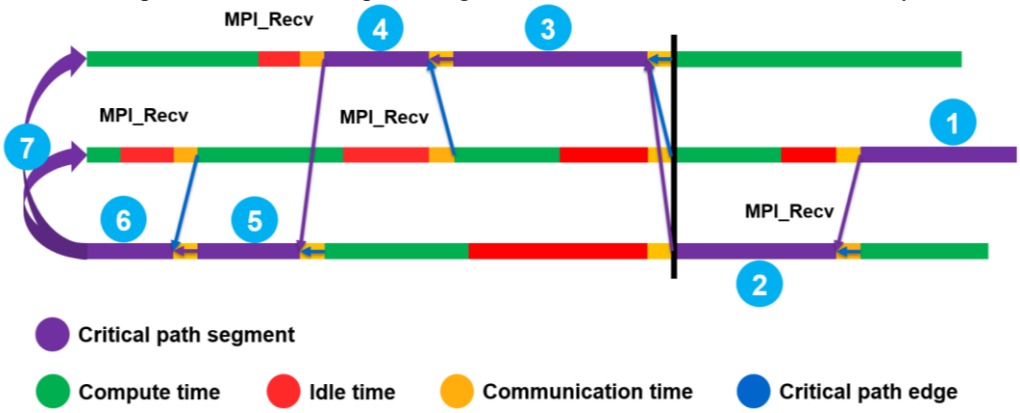

Fig. 8

Find out rank finished last and start restoring path from it (Figure 8, step 1).

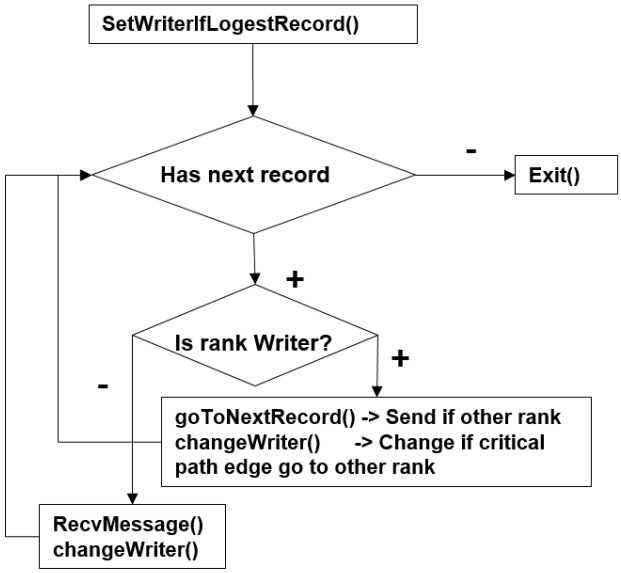

Fig. 9

Block scheme on Figure 9 represents an operation executed within all the ranks. The result about critical path segment is recorded in each rank. At the end, “writer rank” sends stop message (Figure 8, step 7) and all ranks call PMPI_Finalize.

5 PMU samples aggregation on Critical Path

In order to improve efficiency of root causing the reasons of imbalance, we suggest aggregating PMU samples on critical path because:

- Hotspots on Critical Path based on PMU samples naturally highlight activities on the critical path which have the most significant influence on application elapsed time. Optimization of the hotspots on critical path obviously leads to the reduction of application elapsed time. Various situations are easily handled with Hotspots on Critical Path:
  - Load imbalance due to difference in the code flows (e.g. amount of load) within ranks
  - CPU Microarchitecture Issues happening in particular ranks only
  - Communication problems (e.g. improper MPI stack configuration)
- It brings natural scaling capability for the performance analysis tool, because aggregation of PMU samples on Critical Path always limits the amount of aggregating samples to the number of samples collected from within just a single node:

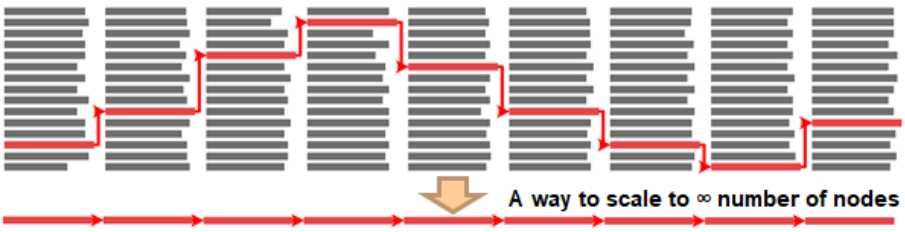

Fig. 10 – Scaling of PMU data analysis with critical path

6 Performance Evaluation

Scale (HW nodes * Ranks)	Elapsed time(s)	Elapsed time under collector(s)	Runtime overhead	Post-processing	
				s	%
64 (1x64)	735.2751	753.9689	2.54%	1.5102	0.20%
121 (1x121)	429.0090	444.5571	3.62%	1.7102	0.38%
256 (4x64)	290.4169	297.1639	2.32%	1.5312	0.52%
484 (4x121)	167.2342	172.6392	3.23%	1.7423	1.01%

Fig. 11. Collection overhead and data post-processing time for NASA Parallel benchmark BT class D

As you can see, the runtime overhead is less than 4%, post-processing time is almost stable and doesn't depend on the number of ranks taking just 1% of application elapsed time in the worst case.

7 Conclusion

In order to improve efficiency of performance analysis of MPI parallel applications, we developed a novel scalable approach for finding Critical Path in the Program Activity Graph of the application which scales well and doesn't require any complex operations on data collection. Combining critical path intervals with PMU data we can efficiently root cause MPI Imbalance issues.

References

1. M. Schulz, "Extracting Critical Path Graphs from MPI Applications," in Proceedings of the 7th IEEE International Conference on Cluster Computing, September 2005
2. D. Bohme, F. Wolf, and M. Geimer, "Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications," in Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, 2012, pp. 2538-2541
3. C. Herold, O. Krzikalla and A. Knüpfer, "Optimizing One-Sided Communication of Parallel Applications Using Critical Path Methods," 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 567-576, doi: 10.1109/IPDPSW.2017.64.
4. J. Chen and R. M. Clapp, "Critical-path candidates: scalable performance modeling for MPI workloads," 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 1-10, doi: 10.1109/ISPASS.2015.7095779.