# Scaling Collectives on Large Clusters of Intel(R) Architecture Processors

Masashi Horikoshi
Software and Solutions Group, Intel K.K.
masashi.horikoshi@intel.com

Larry Meadows
Data Center Group, Intel Corporation, USA
lawrence.f.meadows@intel.com

Tom Elken
Data Center Group, Intel Corporation, USA
tom.elken@intel.com

Pradeep Sivakumar
Data Center Group, Intel Corporation, USA
pradeep.sivakumar@intel.com

Edward Mascarenhas
Data Center Group, Intel Corporation, USA
edward.mascarenhas@intel.com

James Erwin
Data Center Group, Intel Corporation, USA
james.erwin@intel.com

Dmitry Durnov
Software and Solutions Group, Intel Russia
dmitry.durnov@intel.com

Alexander Sannikov
Software and Solutions Group, Intel Russia
alexander.sannikov@intel.com

Toshihiro Hanawa
Information Technology Center, The University of Tokyo
hanawa@cc.u-tokyo.ac.jp

Taisuke Boku
Center for Computational Sciences, University of Tsukuba
taisuke@cs.tsukuba.ac.jp

## ABSTRACT

This paper provides results on scaling Barrier and Allreduce to 8192 nodes on an cluster of Intel® Xeon Phi™ processors installed at the University of Tokyo and the University of Tsukuba. We will describe the effects of OS and platform noise on the performance of these collectives, and provide ways to minimize the noise as well as isolate it to specific cores. We will provide results showing that Barrier and Allreduce scale well when noise is reduced. We were able to achieve a latency of 94 usec (7.1x speedup from baseline) or 1 rank per node Barrier and 145 usec (3.3x speedup) for Allreduce at the 16 byte (16B) message size at 4096 nodes.

## CCS CONCEPTS

• **Networks** → **Network performance analysis**; • **Software and its engineering** → *Scheduling*;

## KEYWORDS

MPI, Collectives, Network performance

**ACM Reference Format:**
Masashi Horikoshi, Larry Meadows, Tom Elken, Pradeep Sivakumar, Edward Mascarenhas, James Erwin, Dmitry Durnov, Alexander Sannikov, Toshihiro Hanawa, and Taisuke Boku. 2018. Scaling Collectives on Large Clusters of Intel(R) Architecture Processors. In *Proceedings of IXPUG Workshop HPC Asia (IXPUG'18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


*IXPUG'18, January 2018, Tokyo, Japan*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Oakforest-PACS (OFP): OFP is a supercomputer system introduced by Fujitsu and the Joint Center for Advanced HPC[4, 5], which was established by the University of Tokyo and the University of Tsukuba. The system comprises 8,208 nodes with a 68-core Intel® Xeon Phi™ 7250 (previously code-named Knights Landing, and referred to as KNL) CPU, 96 GB of DDR4 RAM, and 16 GB of stacked 3D MCDRAM. Intel® Omni-Path Architecture (OPA) fabric provides a 100-Gbps interconnection between nodes in a full bisection bandwidth fat tree. OFP is the largest KNL and OPA fabric cluster that has been installed in the field and ranked 9th in the Top 500 list (November 2017) with peak performance of 24.9 PFLOPS. The computational nodes are using CentOS 7.2 as an operating system. KNL has four hardware threads, known as Intel® Hyper-Threading (HT) Technology. In daily operation, OFP enables HT in the system BIOS. OFP also has a Lustre file system provided by DataDirect Networks (DDN) SFA14KE and burst buffer by DDN IME14K with Intel® Xeon® E5-2600 v4 CPUs powering the Object Storage Server (OSS), Meta-data Server (MDS), and IME (Infinite Memory Engine, aka burst-buffer) nodes.

One important performance metric for OFP is the performance of Message Passing Interface (MPI) Barrier and small-data MPI Allreduce at large scale. When we measured the performance of these collectives, we observed two problems:

(1) Performance of Barrier and 16B Allreduce showed run to run variability at large node counts
(2) These collectives did not scale well to 4096 and 8192 nodes

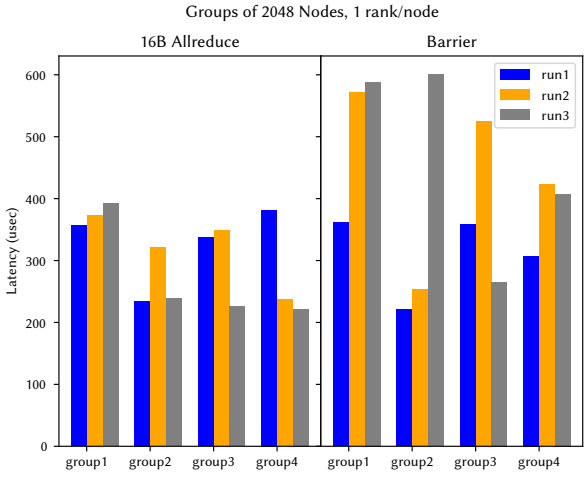


**Figure 1: Run-to-run variability.**

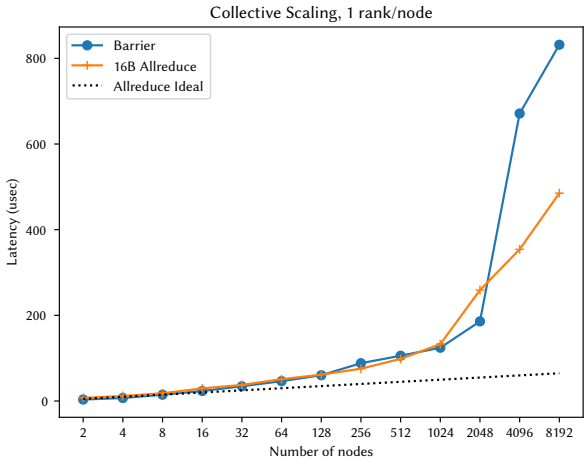


**Figure 2: Collective scaling.**

In this paper we show the initial performance and performance variability, describe system settings and software changes to improve performance, and show the current performance results. Though there are many works historically to observe and analyses MPI collective operations, and propose ways on large supercomputers in order to scale at large node counts, those works[2, 3, 6] were done on multi-core processor systems connected by InfiniBand or proprietary interconnect. This work reports an initial work on latest modern many core processor and interconnect. Finally, we describe ongoing work to further improve performance.

## 2 INITIAL RESULTS

This section shows baseline measurements on OFP prior to the tuning exercise. We used three different versions of Intel® MPI Library, depending on what was available at the time the experiments were run: 2017 Update 3 pre-release, 2017 Update 3 production, and 2019 Update 1 technical preview.[1] Figure 1 shows the run to run variability. The 8192 nodes were split into 4 groups of 2048 nodes[2], and the collective benchmarks run three times on each. We would expect each run in each group to show approximately the same latency, but they are very different. The group-to-group variance is probably insignificant, we expect that it is simply due to insufficient sample size. We hypothesize that the wide variance is due to OS noise.

Figure 2 [3] shows the scaling curve as we increase the number of nodes, while maintaining one rank per node. The ideal scaling is $log_2(N)$, where $N$ is the number of ranks/nodes. Here the hypothesis is that the OS noise increases non-linearly as the number of nodes increases.

[1]We hope to be able to rerun experiments with a single MPI library.
[2]There is no special significance to the grouping
[3]We have no good explanation for the substantially worse Barrier scaling at large node counts

## 3 INITIAL INVESTIGATION

We began by running the MPI collectives in a loop and recording the time for each iteration on each rank using the fine-grained RDTSC timer available on the OFP nodes, as shown here:

```
MPI_Barrier(MPI_COMM_WORLD);
tscs[0] = _rdtsc();
for (int i = 0; i < ntimes; ++i) {
    MPI_Barrier(MPI_COMM_WORLD);
    tscs[i+1] = _rdtsc();
}
report(rank, nranks, ntimes, tscs, benchmark, 0);
```

This method allows us to see the variance for each call to the collective on each rank with very little overhead. The `report` function computes descriptive statistics and optionally saves the time for each collective call on each rank for later analysis.

Figure 3 is a histogram of the barrier time in cycles from a single node for a run on 24 nodes for 100,000 iterations. The mean time is 29,621 cycles, the minimum time is 19,670 cycles, and the maximum time is 2,474,458 cycles. We expect that the extreme excursions from the mean are due to OS noise [4]

## 4 CAUSES OF VARIANCE

We used tools such as `ps`, `top` and kernel `ftrace` to monitor frequency using the KNL hardware counters, and detailed plotting to look for patterns in the iteration-to-iteration variance, and determined several factors that were causing some iterations to take much longer than others. There are three main sources of variance:

**Frequency Transition**: The processors on OFP compute nodes run at nominal 1.4GHz frequency. There are two possible turbo frequencies: multi-tile turbo can increase to 1.5GHz, and single-tile turbo can increase to 1.6GHz. Frequency transitions can cause the processor to stall for many microseconds. We determined that transitions between single- and multi-tile turbo were occurring frequently enough to result in significant delays in some iterations.

[4]Extensive investigation using kernel tracing and correlation with per-iteration timestamps confirm this; unfortunately the margin is too narrow to contain the details.

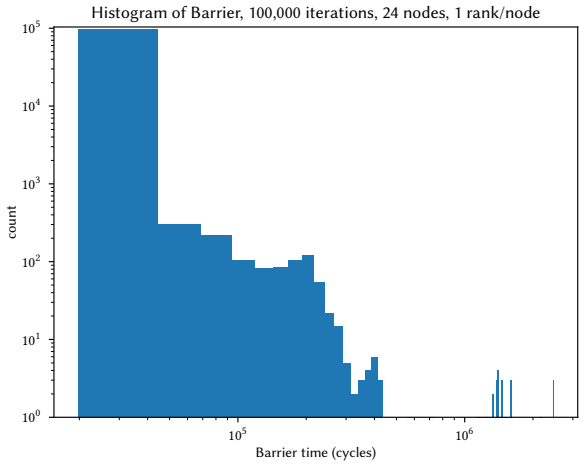


**Figure 3: Barrier Histogram.**

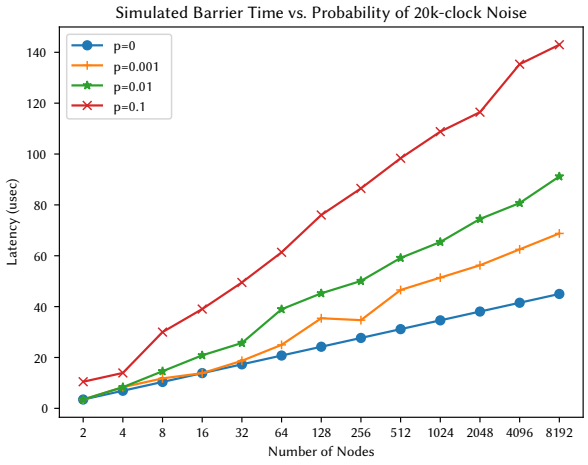


**Figure 4: Simulated Barrier Latency.**

**Periodic MWAIT Wake-up**: Linux uses the MONITOR and MWAIT instructions on idle hardware threads. MONITOR arms a cache line, and MWAIT causes the processor to enter a deep sleep state until that cache line is touched by another thread. KNL forces a periodic wake-up of hardware threads in an MWAIT state 10 times per second. When this wake-up occurs, the OS runs the scheduler to see if there is any work for the thread. Again, this wake-up can take many microseconds, and can additionally cause frequency transitions on the entire processor.

**OS Work**: The OS is constantly running daemons, taking hardware interrupts, and performing other tasks that are unrelated to the running application. When a thread performing application work is interrupted to perform other work, or when another otherwise idle thread on the same core or tile is awakened to perform OS work, the application thread will be delayed.

It is useful to consider the impact of unexpected delays on MPI collectives at scale. In this case we are timing collectives with minimal data transfer, so the total time is completely determined by latency. We can model the collective operation as a series of $log_2(N)$ steps, where $N$ is the number of nodes (and ranks since we are using one rank per node). Each step on each rank is a small-data exchange between the local rank and some remote rank (recursive doubling, as described in Thakur and Gropp's book [8]). Therefore theoretical performance is $log_2(N) * L$ where $L$ is the latency for a small-data exchange between two nodes. There is also an addition of switch hop latency per reduction step, but this is algorithm dependent and a secondary effect which is beyond the scope of this paper.

Figure 4 shows the results of a simulation of recursive doubling by using the Bernoulli case described in Agarwal et al.'s work. We use the measured value of 3.45 usec (for two-node barrier) for the latency $L$, and inject simulated noise of 14 usec (20,000 clocks [5] at 1.4GHz) at each step with probability $p$. The line for $p = 0$ is the ideal performance. It is clear that even a small amount of OS noise has substantial impact.

[5] A value chosen based on non-scientific sampling of many kernel traces

## 5 REMEDIES

Periodic MWAIT wake-up was removed by adding the kernel boot parameter `idle=halt`. This causes idle hardware threads to execute a HLT instruction rather than using MONITOR/MWAIT. This also prevents idle cores from entering deep sleep states, which in turn has the effect of preventing single-tile turbo and thus eliminates most of the frequency transitions. This does have two drawbacks: completely idle power consumption is increased because the package cannot enter deep sleep, and the time to start an idle hardware thread is slightly increased.

One large source of OS noise was removed by adding the kernel boot parameter `nohz_full=2-67,70-135,138-203,206-271` (so-called tickless mode). This disables the OS periodic timer interrupt (normally 1000 Hz) on all but the first tile. Applications must additionally use some mechanism to avoid running on tile 0 (e.g. taskset, numactl and so on). Using the Intel® MPI Library on OFP, set:

```
I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0,1,68,69,136,137,204,205
```

An additional OS noise source turned out to be periodic wake-ups by Lustre filesystem daemons. We avoid this by restricting Lustre daemons to tile 0 by modifying `/etc/modprobe.d/lustre.conf` to include the lines:

```
options libcfs cpu_pattern="0[0,68,136,204],1[1,69,137,204]"
options libcfs cpu_npartitions=2
```

This is not an ideal solution because it restricts the number of Lustre threads and many threads are needed to saturate bandwidth. Future versions of Lustre will not require this parameter[7].

We added the boot parameter `intel_pstate=disable` to use the older `acpi-cpufreq` driver. The default `intel_pstate` driver schedules periodic timer interrupts to determine the current state of the hardware threads in the system; we do not require this functionality. Newer versions of `intel_pstate` have the ability to disable this periodic checking.

The MPI and PSM2 runtimes poll for message completions. Normally they will poll for a short time and then make a system call to `sched_yield`. If the message becomes available while the kernel

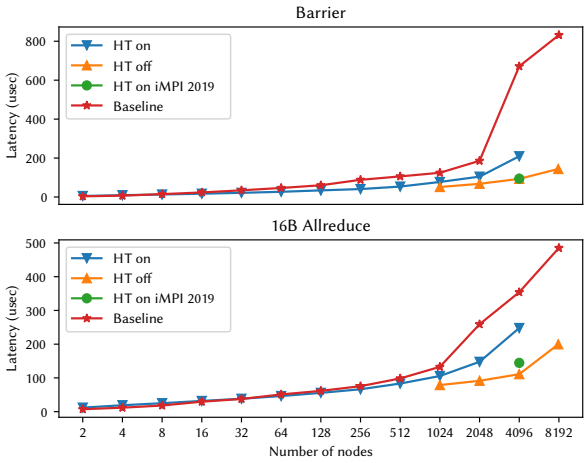


**Figure 5: Final Results.**

is processing the system call then additional latency is introduced. We reduce the effect of these items with the following environment variables:

```
PSM2_YIELD_SPIN_COUNT=10000
I_MPI_COLL_SHM_PROGRESS_SPIN_COUNT=100000
```

## 6 RESULTS

Our results are summarized in Figure 5. We include the baseline data from Figure 2, and three different runs after applying the tunings mentioned in the previous section. The three runs were with HT enabled, HT disabled, and HT enabled with the Intel MPI 2019 tech preview release.

Note that these runs were performed at different times with slightly different versions of the MPI library, as we were able to obtain dedicated access to all or part of OFP. This is also the reason that not all node counts are available for all runs.

Initially we booted with HT disabled, which prevents any other activity on the core running the collective process (unless the process itself is interrupted). However, production use of OFP requires that HT be enabled. The initial runs with HT on were much better than the baseline, but still not as good as the runs with HT off. The 8192 node allreduce result is still less than ideal. We suspect there may be some remaining noise sources that become important during larger scale runs.

After much tuning by the Intel MPI developers, we were able to meet (for barrier) or almost meet (for allreduce) the performance of the runs with HT off. Unfortunately we were able to get only one 4096-node run with the tuned MPI library.

The MPI tuning was related to reducing instructions executed in the collective code itself, not to OS or platform noise. We therefore expect that a run with HT off using the improved MPI implementation would give even better results.

## 7 CONCLUSION

We have demonstrated that OS and platform noise can have very large effects on small-data MPI collectives as the number of nodes increases. The initial estimate for collective performance did not account for this factor. After the efforts described in this paper we were able to exceed the initial estimates for 4096 nodes, even with HT enabled, as summarized in this table:

| Collective | Estimate(us) | Baseline(us) | Optimized(us) |
|---|---|---|---|
| barrier | 105 | 671 | 94 |
| 16B allreduce | 160 | 485 | 145 |

Work is ongoing and it is expected that we will get access to run these benchmarks at 8192 nodes with the important `idle=halt` and HT disabled settings using the latest optimized runtime libraries. The data for allreduce shows that there is still some noise with HT enabled. Since progressively smaller OS noise has substantial impact with larger scaling, it is possible we will need to find and remove other areas of noise. In the case of Multi ranks per node will also be a future work.

## ACKNOWLEDGMENTS

Part of the computational resource of the Oakforest-PACS was awarded by the "Large-scale HPC Challenge" Project, JCAHPC (Joint Center for Advanced High Performance Computing). We thank the faculty and staff of JCAHPC, as well as the engineers at Fujitsu (Computational Science and Engineering Solution DIV., Technical Computing Solutions Unit) and Intel, particularly Professor Kengo Nakajima (The University of Tokyo), Yoshio Sakaguchi (Fujitsu), Kohta Nakashima (Fujitsu Laboratories), Alexey Malhanov (Intel) and John Pennycook (Intel)

## REFERENCES

[1] Saurabh Agarwal, Rahul Garg, and Nisheeth K. Vishnoi. 2005. *The Impact of Noise on the Scaling of Collectives: A Theoretical Approach.* Springer Berlin Heidelberg, Berlin, Heidelberg, 280–289. https://doi.org/10.1007/11602569_31

[2] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10).* IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.12

[3] Masashi Horikoshi, Yutaka Ueshima, Keiji Kubo, Daisuke Wakabayashi, and Katsunobu Nishihara. 2005. Performance Evaluation of HP AlphaServer SC system. In *Journal of IPSJ HPCS 2005 (HPCS '05).* IPSJ, Japan, 65–72.

[4] JCAHPC. 2016. JCAHPC, Online. (2016). http://jcahpc.jp/eng/index.html.

[5] Oakforest-PACS. 2016. Oakforest-PACS Supercomputer System, Online. (2016). http://www.cc.u-tokyo.ac.jp/system/ofp/index-e.html.

[6] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03).* ACM, New York, NY, USA, 55–. https://doi.org/10.1145/1048935.1050204

[7] Intel HPDD Code Review site. 2017. LU-9441 pltrpc: don't wakeup on 1 second intervals, Online. (2017). https://review.whamcloud.com/#/c/28496.

[8] Rajeev Thakur and William D. Gropp. 2003. *Improving the Performance of Collective Operations in MPICH.* Springer Berlin Heidelberg, Berlin, Heidelberg, 257–267. https://doi.org/10.1007/978-3-540-39924-7_38