Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?

Sascha Hunold
TU Wien, Faculty of Informatics
Vienna, Austria
hunold@par.tuwien.ac.at

Sebastian Steiner
TU Wien, Faculty of Informatics
Vienna, Austria
steiner@par.tuwien.ac.at

***Abstract*—Julia has quickly become one of the main programming languages for computational sciences, mainly due to its speed and flexibility. The speed and efficiency of Julia are the main reasons why researchers in the field of High Performance Computing have started porting their applications to Julia.**

Since Julia has a very small binding-overhead to C, many efficient computational kernels can be integrated into Julia without any noticeable performance drop. For that reason, highly tuned libraries, such as the Intel MKL or OpenBLAS, will allow Julia applications to achieve similar computational performance as their C counterparts. Yet, two questions remain: 1) How fast is Julia for memory-bound applications? 2) How efficient can MPI functions be called from a Julia application?

In this paper, we will assess the performance of Julia with respect to HPC. To that end, we examine the raw throughput achievable with Julia using a new Julia port of the well-known STREAM benchmark. We also compare the running times of the most commonly used MPI collective operations (e.g., `MPI_Allreduce`) with their C counterparts. Our analysis shows that HPC performance of Julia is on-par with C in the majority of cases.

***Index Terms*—Julia, HPC, MPI, OpenMP, STREAM**

I. Introduction

Since its first release in 2012, Julia has become a very popular programming language for scientific computing. The goal of Julia, according to their creators, is to attain "machine performance without sacrificing human convenience" [1]. The authors also showed that Julia does achieve similar performance numbers as C for simple tasks, such as computing Fibonacci numbers. However, to be considered a programming language that can be used in High Performance Computing (HPC) is a more complicated task, at which many other languages like Python or Java have failed [2].

Motivated by the success story of Regier et al. [3], who showed that a large Bayesian inference code written in Julia can achieve petascale performance, we set out to evaluate the speed of Julia in terms of low-level communication performance. For HPC, two types of communication benchmarks are important, which are 1) a node-level benchmark that analyzes the intra-node communication performance and 2) an inter-node communication benchmark that measures the attainable throughput between distributed compute nodes.

In order to evaluate the intra-node communication performance attainable with Julia, we resort to the well-known STREAM benchmark [4], which we have ported to Julia. For analyzing the inter-node communication performance, we first select the Message Passing Interface (MPI) as the communication interface to be investigated, as MPI has been and still remains the de-facto standard for data communication in large-scale machines. To assess the MPI performance of Julia, we have ported a subset of the ReproMPI benchmark suite to Julia. We perform a series of experiments on different shared- and distributed memory machines and compare the results obtained with the C and the Julia versions of the benchmarks.

This paper makes the following contributions:

- We present two novel Julia benchmarks that can be used to assess the performance of HPC machines.
- We carefully analyze the intra- and inter-node communication performance attainable with Julia, showing that Julia is indeed a serious alternative to C and Fortran in the HPC domain.

In the remainder of the article, we discuss related work in Section II and introduce the challenges that Julia faces in the HPC domain in Section III. We present our experimental setup in Section IV and the performance results in Section V. We conclude the article in Section VI.

II. Related Work

In contrast to other scientific domains, where many different programming languages have gained popularity, HPC has always been dominated by C/C++ and Fortran for one good reason: attainable performance. Achieving the peak performance of a supercomputer is very hard and complicated for the regular programmer [5], and performance tuning often requires tweaking low-level optimization knobs. Languages like Python or Java have never been able to deliver the same performance as C/C++ and Fortran, as their higher level of expressiveness lacks capabilities to perform low-level code optimizations. Even efficient multi-threading in Python is not supported by default [6]. Nevertheless, many efforts have been made to make Python and Java more efficient and attractive to HPC programmers, e.g., the Java Grande Forum has made significant contributions to the efficiency of Java [7]. In order to make a language applicable for HPC tasks, it needs an MPI binding. Thus, several MPI bindings have been presented for Python and Java, some of which use wrappers to the actual C MPI library [8] and others re-implement the MPI standard [9].

The Julia language [1] is one of the latest competitors to be used for HPC programming. Since Julia offers a very light-weight API to call C routines, using high-performance libraries that are written in C/C++, such as the Intel MKL, entails a very small overhead. It has also been shown that, in cases where calls to C functions are very short-lived, novel methods to reduce the overhead of calling these functions (e.g. BLAS routines) from Julia are possible and available [10].

To the best of our knowledge, there is no in-depth study on how Julia performs for intra- and inter-node communication operations. For that reason, the present paper attempts to close this gap.

III. HPC Performance Challenges for Julia

Our goal is to examine the communication performance achievable with Julia on HPC systems.

a) Challenge 1: STREAM: Our first challenge for Julia is to compete with the original STREAM benchmark [4]. STREAM is one of the widest known tools to experimentally determine the DRAM memory bandwidth of compute nodes. To that end, the STREAM benchmark uses four different kernels that read and write large arrays from and to DRAM. The kernels differ in the number of arrays accessed and also in the computations performed on the streamed data items, and thus, each kernel possesses a specific computation-to-communication ratio. The experimental memory bandwidth, determined with the STREAM benchmark, is a key metric for several performance-analysis methods in HPC, e.g., the Roofline model [11].

b) Challenge 2: ReproMPI: Our second challenge for Julia is to test its performance on communicating data between compute nodes using MPI. In a set of experiments, we measure the running time of blocking collective communication operations. We test three different representative MPI collective communication operations: `MPI_Bcast`, `MPI_Allreduce`, and `MPI_Alltoall`. The `MPI_Bcast` and `MPI_Allreduce` operations are the most frequently used collectives in large-scale application codes according to Chunduri et al. [12]. In addition, all three have distinct characteristics. `MPI_Bcast` is a rooted collective, where data is pushed into the network from a single source node. `MPI_Allreduce` and `MPI_Alltoall` are non-rooted collectives, as all processes contribute data to the final result. In an `MPI_Allreduce` operation, processes need to perform a reduction operation on the local data, which is not required in `MPI_Alltoall`. However, `MPI_Alltoall` is the heaviest collective operation in terms of the amount of data transferred, as all processes send and receive from all other processes.

IV. Experimental Setup

We briefly describe our experimental setup by summarizing our hardware and then by reporting on the software used.

a) Hardware Setup: Since we perform intra- and inter-node experiments, we divide the systems shown in Table I into two categories. For evaluating the intra-node communication performance, we use a single node of the *Hydra* cluster, which comprises two Intel Xeon Gold 6130 processors, with 16 cores each. We also conduct node-level experiments on a machine called *Nebula*, which comprises two AMD EPYC 7551 CPUs. Since this specific AMD EPYC processor contains 32 cores, *Nebula* has a total of 64 cores.

TABLE I: Hardware overview.

Machine	n	ppn	Processor	Interconnect
Hydra	36	32	2 × Intel Xeon Gold 6130	Intel OmniPath
Jupiter	35	16	2 × AMD Opteron 6134	Mellanox InfiniBand (QDR)
Nebula	1	64	2 × AMD EPYC 7551	none

For conducting the MPI experiments, we use the *Hydra* and *Jupiter* systems. Both are smaller cluster installations that can give us hints on how Julia's MPI performance compares to C with up to $36 \times 32 = 1152$ processes.

b) Software Setup: We ported the STREAM and the ReproMPI benchmark to Julia, which are referenced as STREAM.jl[1] and ReproMPI.jl[2] in the remainder, respectively. The Julia port of ReproMPI only contains a subset of all features available in ReproMPI [13], [14]. For example, when measuring the running time of the blocking collectives, processes are synchronized with an `MPI_Barrier` call, as done by the benchmark suites OSU Micro-Benchmarks [15] and Intel MPI Benchmarks [16]. Although this method might not be precise enough to compare algorithmic variants from different MPI libraries, it is sufficient in this work, as we always use the same MPI library. On *Hydra* and *Jupiter*, we used Open MPI 3.1.3 and Open MPI 4.0.3, respectively. We also tested the running times on *Hydra* with Open MPI 4.0.3, but the results were similar. We would like to note that the goal of our research was to quantify the overhead of using MPI from Julia, as opposed to finding the best possible running time of any MPI collective on our cluster system [17].

The C versions of the STREAM benchmark were compiled with gcc 8.3.0. We have also experimented with clang 7.0.1 and with pgcc 20.7, but the performance results were consistent with the results measured with the gcc-compiled benchmarks.

We measured the running times with multiple versions of Julia, ranging from version 0.7.0 to 1.4.0. Since Julia is LLVM-based, we employed LLVM 7.0.1 on *Hydra* as well as *Nebula* and LLVM 3.4.2 on *Jupiter*.

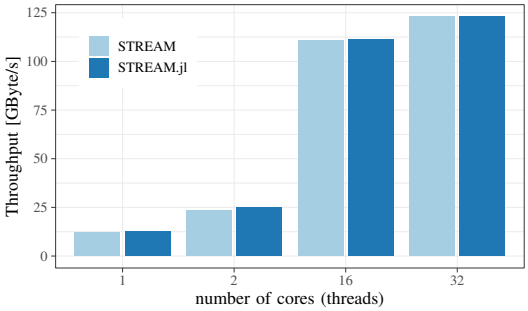
V. Experimental Results

Now, we discuss the results obtained in our experiments. First, we compare the throughput values measured on two different shared-memory systems. Second, we present and discuss results from the MPI experiments.

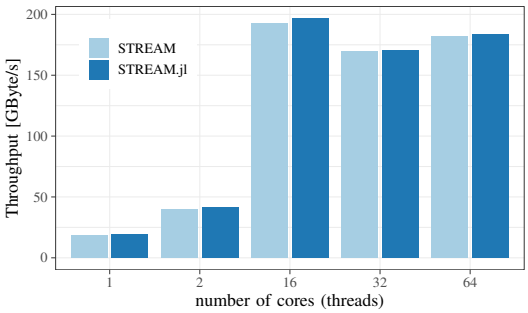
A. STREAM.jl

Figure 1 presents the throughput results measured on *Hydra* and *Nebula* with the C and the Julia version of the STREAM benchmark. Since both systems are NUMA machines that have

[1]https://github.com/sebastian-steiner/STREAM.jl
[2]https://github.com/sebastian-steiner/reproMPI.jl



(a) *Hydra* (32 cores in total)



(b) *Nebula* (64 cores in total)

Fig. 1: Throughput measured with STREAM and STREAM.jl.

several memory controllers, pinning threads to specific cores has a huge performance impact. Therefore, we use `numactl` to pin threads in a round-robin fashion to either the sockets (*Hydra*) or the NUMA packages (*Nebula*).

We ran each STREAM experiment three times and report the maximum value achieved by any of the four STREAM kernels. The figures show that there is no performance difference between the two STREAM versions. However, in the case of running on 16 cores of *Nebula*, it is interesting to note that the throughput obtained with either C or Julia was visibly higher than what was measured with 32 or 64 cores. Considering that each socket has eight memory channels, the results suggest that 16 cores are enough to saturate the available bandwidth.

B. ReproMPI

We now discuss the experimental results for each MPI collective operation separately. Notice that we also conducted an extensive set of experiments on *Jupiter*. On this machine, the performance results were almost identical for the C and the Julia version. For space limitations, we omit most of these results. For *Hydra*, we only show results for 36×32 processes, as performance differences are more pronounced for this process configuration. For a smaller number of processes, the performance differences between C and Julia were most often found to be negligible. We also include Julia performance data for several versions of Julia and the MPI.jl package. We present two views on the measurements for each experiment. On the left-hand side, we show the absolute running times (mean and 95% confidence interval), and on the right-hand side, we show the relative running times, which were normalized to the mean running times of the C versions.

a) `MPI_Bcast`: The running times of `MPI_Bcast` measured for the C and Julia version of ReproMPI are shown in Figure 2. These results show that the MPI binding in Julia does not lead to a considerable overhead. We also see that Julia 0.7.0 and MPI.jl 0.8.0 are outperformed by their successors.

b) `MPI_Alltoall`: The performance comparison for `MPI_Alltoall` is presented in Figure 3. Our findings are similar to the ones for `MPI_Bcast`: there is no significant performance loss when using Julia.

c) `MPI_Allreduce`: Figure 4 compares the running times of the C MPI benchmark to the Julia version. For larger message sizes ($> 10\,\mathrm{kB}$), we can observe a performance degradation when using Julia, which is not visible in the plot showing the absolute running times. Yet, we can see the difference in the figure presenting the relative performance results. For `MPI_Allreduce`, we noticed again that newer versions of Julia and the MPI.jl package improve the running time significantly. Nonetheless, for the largest message sizes in our experiment, the mean running times of the Julia benchmark were about twice as long as the ones with C.

As we could not explain this performance loss, we analyzed the individual measurements. First, we compared the minimum running times observed for the various message sizes in Figure 6. We can see that these values are identical for C and Julia, which means that Julia MPI bindings do not entail a systematic overhead for each MPI call. Second, we analyzed the running time distribution of `MPI_Allreduce` for these message sizes that showed performance degradation in Julia. These histograms are presented in Figure 7, each of which compares the running time distribution obtained from the C version of ReproMPI (top row) with the Julia version (bottom row). The figures clearly show that the message size has a significant impact on the shape of the distribution. Figure 7a shows that the distributions for the smallest messages size look very similar, while the opposite can be seen in Figure 7c, where the distributions have clearly diverged. For $1\,024\,000\,\mathrm{B}$, the distribution of running times measured with the Julia benchmark is much wider, and the mean running time increased substantially. Finding the actual root cause of this distribution shift could be the subject of future work. The data suggest that the communication performance can be improved by adjusting Julia internals (e.g., the influence of the garbage collector). Interestingly, we did not see such performance drops with `MPI_Allreduce` on *Jupiter*, as can be seen in Figure 5.

VI. Conclusions

Our study of Julia's communication performance closes a missing gap in the performance landscape of Julia. We conducted DRAM throughput experiments on individual compute nodes using the STREAM benchmark to evaluate the intra-node performance attainable with Julia. We also measured the running times of three different, blocking MPI collective operations to assess the overhead introduced by Julia.

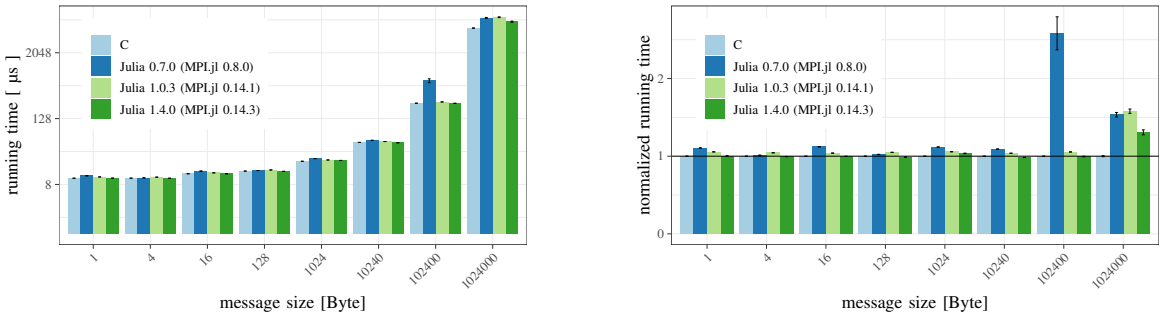


Fig. 2: Running times of `MPI_Bcast`, absolute (left), relative (right), 36×32 processes, *Hydra*.

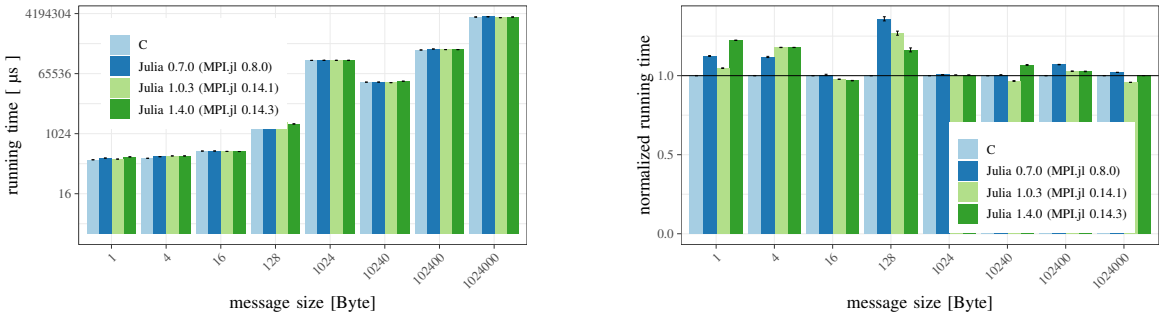


Fig. 3: Running times of `MPI_Alltoall`, absolute (left), relative (right), 36×32 processes, *Hydra*.

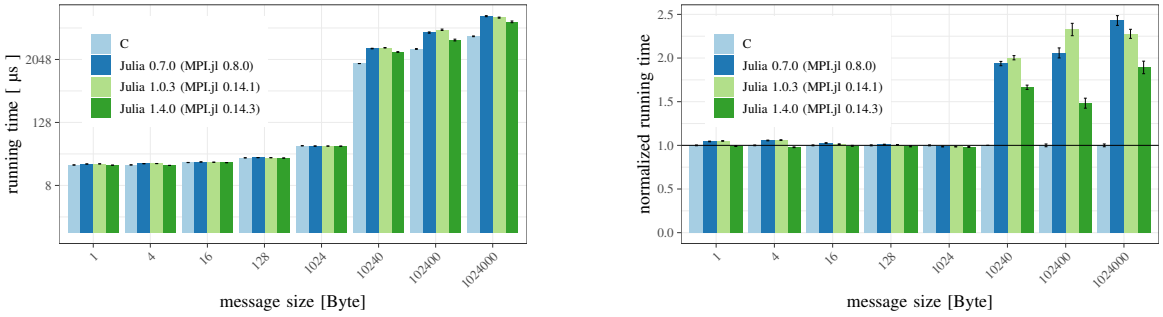


Fig. 4: Running times of `MPI_Allreduce`, absolute (left), relative (right), 36×32 processes, *Hydra*.

The experimental results show that Julia entails almost no overhead compared to the C benchmarks. Solely in the case of `MPI_Allreduce`, we observed a performance loss with the Julia MPI benchmark for larger message sizes. Therefore, our Julia performance numbers allow us to suggest that Julia is indeed a serious competitor for developing HPC codes, foremost since Julia requires a significantly smaller amount of code to achieve similar results as C.

This is only the first step in assessing Julia's ability to conduct HPC tasks as efficiently as C or Fortran. In future work, we will conduct an in-depth analysis of the performance drop for large message sizes in `MPI_Allreduce` calls.

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[2] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner *et al.*, "Programming languages for data-intensive HPC applications: A systematic mapping study," *Parallel Comput.*, vol. 91, 2020.

[3] J. Regier, K. Fischer, K. Pamnany, A. Noack, J. Revels *et al.*, "Cataloging the visible universe through bayesian inference in Julia at petascale," *J. Parallel Distributed Comput.*, vol. 127, pp. 89–104, 2019.

[4] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.

[5] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer *et al.*, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" *Commun. ACM*, vol. 58, no. 5, pp. 77–86, 2015.

[6] R. Meier and T. R. Gross, "Reflections on the compatibility, performance, and scalability of parallel Python," in *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*, S. Marr and J. Fumero, Eds. ACM, 2019, pp. 91–103.

[7] C. M. Pancake and C. Lengauer, "High-performance Java - introduction," *Commun. ACM*, vol. 44, no. 10, pp. 98–101, 2001.

[8] L. Dalcín, R. Paz, M. A. Storti, and J. D'Elía, "MPI for Python: Performance improvements and MPI-2 extensions," *J. Parallel Distributed Comput.*, vol. 68, no. 5, pp. 655–662, 2008.

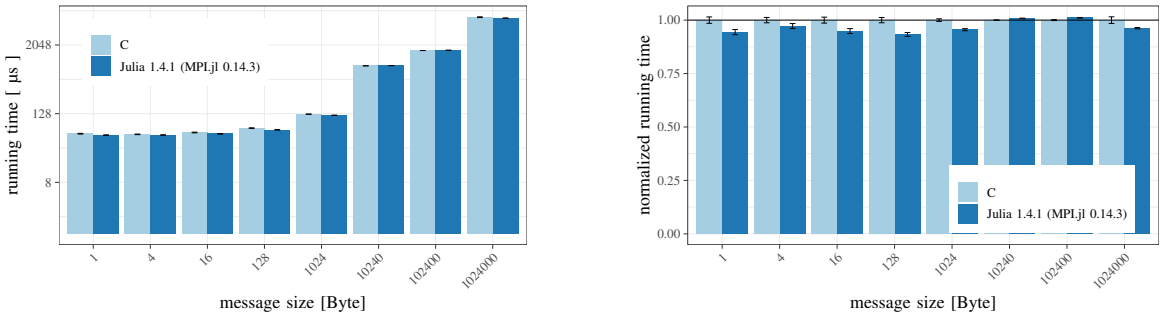


Fig. 5: Running times of `MPI_Allreduce`, absolute (left), relative (right), 32×16 processes, *Jupiter*.

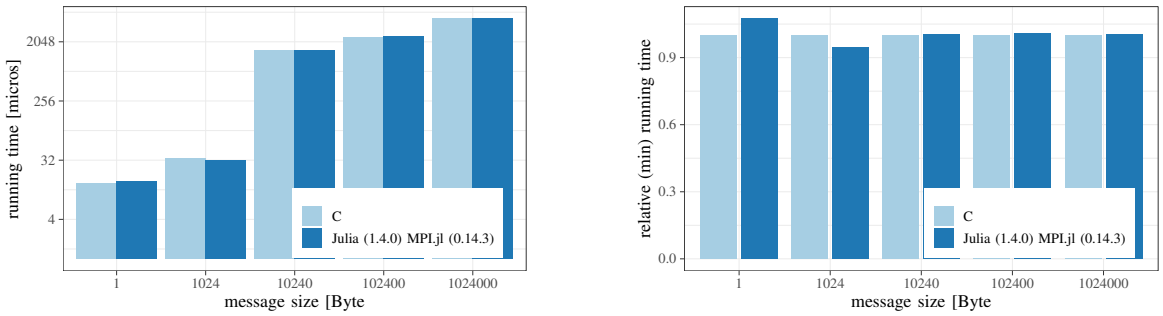
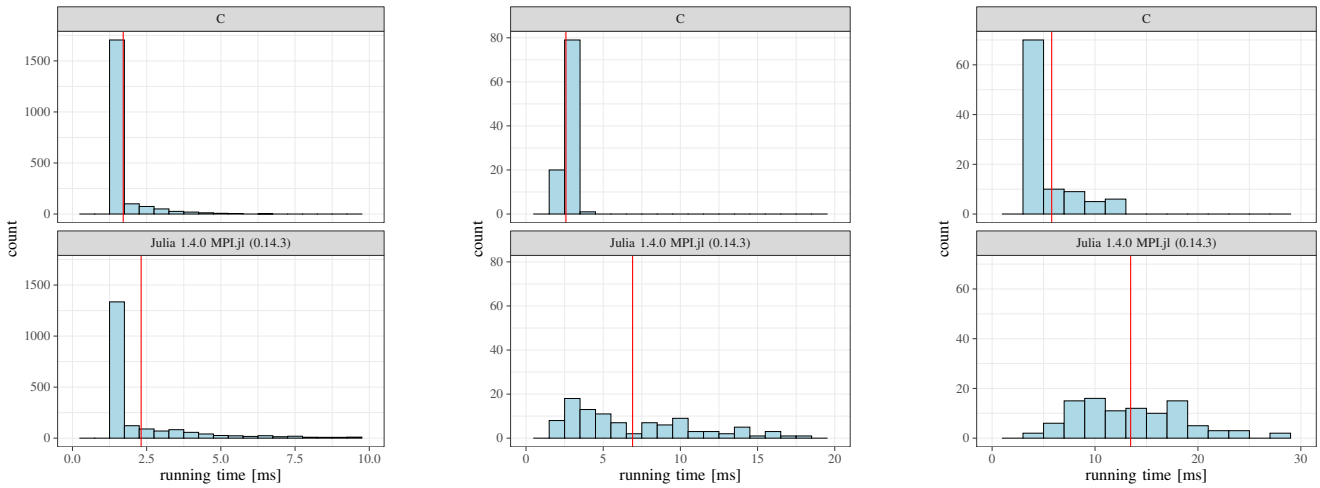


Fig. 6: Minimum running time of `MPI_Allreduce`, absolute (left), relative (right), 36×32 processes, *Hydra*.



(a) 10 240 B (b) 102 400 B (c) 1 024 000 B

Fig. 7: Histograms of running times for `MPI_Allreduce` with different message sizes, mean values are marked in red, 36×32 processes, *Hydra*.

[9] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the high performance computing arena: Research, practice and experience," *Sci. Comput. Program.*, vol. 78, no. 5, pp. 425–444, 2013.

[10] G. Frison, T. Sartor, A. Zanelli, and M. Diehl, "The BLAS API of BLASFEO: optimizing performance for small matrices," *ACM Trans. Math. Softw.*, vol. 46, no. 2, pp. 15:1–15:36, 2020.

[11] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[12] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *Proceedings of the Supercomputing*. IEEE / ACM, 2018, pp. 30:1–30:15.

[13] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[14] S. Hunold and A. Carpen-Amarie, "Hierarchical clock synchronization in MPI," in *IEEE CLUSTER*, 2018, pp. 325–336.

[15] "OSU Micro-Benchmarks." [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[16] "Intel MPI Benchmarks." [Online]. Available: https://github.com/intel/mpi-benchmarks

[17] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *IEEE CLUSTER*, 2020.

APPENDIX
ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

A. Summary of Experiments Reported

We report results of benchmarking MPI collectives using the ReproMPI benchmark and its Julia version called ReproMPI.jl. The benchmarks measure the running time of individual blocking MPI collectives implemented in Open MPI. For the Julia version of the benchmark, we rely not only on Open MPI but also on the Julia bindings for MPI (MPI.jl).

B. Artifact Availability

Software Artifact Availability: All benchmark codes are available on github (see list below). Some scripts to execute the benchmarks are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: There are no author-created proprietary artifacts.

List of URLs and/or DOIs where artifacts are available:

- https://github.com/sebastian-steiner/STREAM.jl
- https://github.com/sebastian-steiner/reproMPI.jl
- https://github.com/hunsa/reprompi

C. Baseline Experimental Setup, and Modifications Made for the Paper

Paper Modifications: See artifact description and details in the paper.

Output from scripts that gathers execution environment information:

Hydra*:*

```
uname -a
Linux hydra 4.19.0-9-amd64 #1 SMP Debian 4.19.118-2+deb10u1 (2020-06-07) x86_64 GNU
       /Linux

lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       46 bits physical, 48 bits virtual
CPU(s):              32
On-line CPU(s) list: 0-31
Thread(s) per core:  1
Core(s) per socket:  16
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Gold 6130F CPU @ 2.10GHz
Stepping:            4
CPU MHz:             1118.831
CPU max MHz:         2100.0000
CPU min MHz:         1000.0000
BogoMIPS:            4200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            22528K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
       pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
       rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
       nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
       tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
       tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
       cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd mba ibrs
       ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust
       bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq
       rdseed adx smap clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt
       xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local
       dtherm arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req pku ospke md_clear
        flush_l1d
```

Jupiter*:*

```
uname -a
Linux jupiter 3.10.0-1127.el7.x86_64 #1 SMP Tue Mar 31 23:36:51 UTC 2020 x86_64
       x86_64 x86_64 GNU/Linux

lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          4
Vendor ID:             AuthenticAMD
CPU family:            16
Model:                 9
Model name:            AMD Opteron(tm) Processor 6134
Stepping:              1
CPU MHz:               800.000
CPU max MHz:           2300.0000
CPU min MHz:           800.0000
BogoMIPS:              4600.10
Virtualization:        AMD-V
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              5118K
NUMA node0 CPU(s):     0-3
NUMA node1 CPU(s):     4-7
NUMA node2 CPU(s):     12-15
NUMA node3 CPU(s):     8-11
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
       cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt
       pdpe1gb rdtscp lm 3dnowext 3dnow constant_tsc art rep_good nopl nonstop_tsc
       extd_apicid amd_dcm pni monitor cx16 popcnt lahf_lm cmp_legacy svm extapic
       cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt nodeid_msr
        hw_pstate retpoline_amd ibp_disable vmmcall npt lbrv svm_lock nrip_save
       pausefilter
```

Nebula*:*

```
uname -a
Linux nebula 4.19.0-10-amd64 #1 SMP Debian 4.19.132-1 (2020-07-24) x86_64 GNU/Linux

lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       43 bits physical, 48 bits virtual
CPU(s):              64
On-line CPU(s) list: 0-63
Thread(s) per core:  1
Core(s) per socket:  32
Socket(s):           2
NUMA node(s):        8
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               1
Model name:          AMD EPYC 7551 32-Core Processor
Stepping:            2
CPU MHz:             2379.320
CPU max MHz:         2000.0000
CPU min MHz:         1200.0000
BogoMIPS:            3992.24
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           64K
L2 cache:            512K
L3 cache:            8192K
NUMA node0 CPU(s):   0,8,16,24,32,40,48,56
NUMA node1 CPU(s):   2,10,18,26,34,42,50,58
NUMA node2 CPU(s):   4,12,20,28,36,44,52,60
NUMA node3 CPU(s):   6,14,22,30,38,46,54,62
NUMA node4 CPU(s):   1,9,17,25,33,41,49,57
NUMA node5 CPU(s):   3,11,19,27,35,43,51,59
NUMA node6 CPU(s):   5,13,21,29,37,45,53,61
NUMA node7 CPU(s):   7,15,23,31,39,47,55,63
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
       pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb
       rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid amd_dcm
       aperfmperf pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt
       aes xsave avx f16c rdrand lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a
        misalignsse 3dnowprefetch osvw skinit wdt tce topoext perfctr_core
       perfctr_nb bpext perfctr_llc mwaitx cpb hw_pstate sme ssbd sev ibpb vmmcall
       fsgsbase bmi1 avx2 smep bmi2 rdseed adx smap clflushopt sha_ni xsaveopt
       xsavec xgetbv1 xsaves clzero irperf xsaveerptr arat npt lbrv svm_lock
       nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter
       pfthreshold avic v_vmsave_vmload vgif overflow_recov succor smca
```