Pure: Evolving Message Passing To Better Leverage Shared Memory Within Nodes

James Psota
Armando Solar-Lezama
{jim,asolar}@csail.mit.edu
MIT CSAIL
Cambridge, MA, USA

ABSTRACT

Pure is a new programming model and runtime system explicitly designed to take advantage of shared memory within nodes in the context of a *mostly* message passing interface enhanced with the ability to use tasks to make use of idle cores. We use microbenchmarks to evaluate Pure's key messaging and collective features and also show application speedups up to 2.1× on the CoMD molecular dynamics application. Overall, Pure offers improved performance by aggressively leveraging modern shared memory nodes with a programming model that will be familiar to MPI programmers.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Multiprocessing / multiprogramming / multitasking**.

1 INTRODUCTION

Pure is a C++17 library inspired by MPI; its programming model is essentially message passing with the optional use of tasks. However, Pure breaks the constraints of using process-level ranks and requiring support for older languages. In particular, Pure uses threads as ranks instead of processes, allowing it to use lightweight, lock-free synchronization to coordinate between threads running within the same node efficiently. Pure builds on the thread-based ranks to implement highly efficient collective operations within nodes leveraging efficient lock-free algorithms. Finally, Pure optionally allows pieces of the application to run in standard C++ lambda expressions that can be executed concurrently by the owning rank and any other ranks that are idle, all coordinated automatically by the Pure Runtime System. Expanding the responsibility of the runtime system to include optional concurrent task execution is valuable, as it allows the runtime to efficiently and automatically overlap communication and computation without orchestration by the programmer.

We outline several performance challenges in implementing our parallel runtime system and describe our optimizations, including a lock-free messaging approach for both small and large messages; lock-free collective data structures which we compose to implement collective algorithms; a lock-free task scheduler that allows idle threads to efficiently steal work from other threads. We use standard C++ libraries for widespread compatibility, and show substantial performance improvement over a heavily optimized MPI baseline.

2 RELATED WORK

While Pure uses threads and shared memory in its runtime system, it is unlike programming using MPI+OpenMP [11], which requires the programmer to manually orchestrate two distinct programming models. Pure's runtime unifies the within-node message and collective optimizations and optional task execution. Thus, Pure allows all cores to be efficiently utilized with less programmer effort compared to MPI+OpenMP.

MPI offers "one-sided" message APIs [7], which decouple data movement with process synchronization. Ranks are able to read and write part of other ranks' memories directly, and later synchronize. However, Pure provides a higher level mechanism for overlapping communication and computation. MPI Fine-points [6] and MPI Endpoints [3] outline independent approaches to introducing the concept of threading and "MPI+X" directly into MPI. Fine-points introduces a new MPI calls that express how multiple threads can concurrently work to enact single larger communication operations; Endpoints allows threads within an MPI process to concurrently send and receive their own small messages. As with Pure, MPI Endpoints allows threads to be addressed and for each thread to have its own rank. However, both Fine-points and Endpoints introduce additional complexity to the application code. Unlike Pure, both of these models are fundamentally hybrid, treating threads and processes hierarchically.

Parallel frameworks are libraries that provide a layer of abstraction between the application and the machine. They include Kokkos [4], STAPL [5] and BCL [2]. Like Pure, these libraries often do leverage modern C++ features, but require major rewrites of existing applications in order to use. AMPI [8] is an MPI-compliant library that builds on top of

```
void rand_stencil_mpi(double* const a, size_t arr_sz, size_t iters, int my_rank,
                      int n_ranks) {
    double temp[arr_sz];
    for (auto it = 0; it < iters; ++it) {
        for (auto i = 0; i < arr_sz; ++i) {
            temp[i] = random_work(a[i]);
        }
        for (auto i = 1; i < arr_sz - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) {
            MPI_Send(&temp[0], 1, MPI_DOUBLE, my_rank - 1, 0, MPI_COMM_WORLD);
            double neighbor_hi_val;
            MPI_Recv(&neighbor_hi_val, 1, MPI_DOUBLE, my_rank - 1, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        } // ends if not first rank
        if (my_rank < n_ranks - 1) {
            MPI_Send(&temp[arr_sz - 1], 1, MPI_DOUBLE, my_rank + 1, 0,
                     MPI_COMM_WORLD);
            double neighbor_lo_val;
            MPI_Recv(&neighbor_lo_val, 1, MPI_DOUBLE, my_rank + 1, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            a[arr_sz - 1] =
                    (temp[arr_sz - 2] + temp[arr_sz - 1] + neighbor_lo_val) /
                    3.0;
        } // ends if not last rank
    }     // ends for all iterations
}
```

Listing 1: 1-D Stencil with Random Work, MPI Version

```
void rand_stencil_pure(double* const a, size_t arr_sz, size_t iters,
                       int my_rank, int n_ranks) {
    double   temp[arr_sz];
    PureTask rand_work_task = [a, temp, arr_sz,
                               my_rank](chunk_id_t           start_chunk,
                                        chunk_id_t           end_chunk,
                                        std::optional<void*> cont_params) {
        auto [min_idx, max_idx] =
                pure_aligned_idx_range<double>(arr_sz, start_chunk, end_chunk);
        for (auto i = min_idx; i <= max_idx; ++i) {
            temp[i] = random_work(a[i]);
        }
    }; // ends defining the Pure Task rand_work_task
    for (auto it = 0; it < iters; ++it) {
        rand_work_task.execute(); // execute all chunks of rand_work_task
        for (auto i = 1; i < arr_sz - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) {
            pure_send_msg(&temp[0], 1, PURE_DOUBLE, my_rank - 1, 0,
                          PURE_COMM_WORLD);
            double neighbor_hi_val;
            pure_recv_msg(&neighbor_hi_val, 1, PURE_DOUBLE, my_rank - 1, 0,
                          PURE_COMM_WORLD);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        } // ends if not first rank
        if (my_rank < n_ranks - 1) {
            pure_send_msg(&temp[arr_sz - 1], 1, PURE_DOUBLE, my_rank + 1, 0,
                          PURE_COMM_WORLD);
            double neighbor_lo_val;
            pure_recv_msg(&neighbor_lo_val, 1, PURE_DOUBLE, my_rank + 1, 0,
                          PURE_COMM_WORLD);
            a[arr_sz - 1] =
                    (temp[arr_sz - 2] + temp[arr_sz - 1] + neighbor_lo_val) /
                    3.0;
        } // ends if not last rank
    }     // ends for all iterations
}
```

Listing 2: 1-D Stencil with Random Work, Pure Version.

Charm++ [9]. AMPI over-decomposes an MPI program, running more "virtual processes" than physical cores. It then efficiently schedules execution of virtual processes, possibly migrating or creating new ones to improve load balance. Like Pure, AMPI offers performance gains for MPI programs with minimal source code changes. Pure takes a different approach to improve performance. Firstly, as shown in our microbenchmarks, Pure achieves significant performance improvement due solely to its optimized messaging and collective routines. This first contribution improves performance even when there is no load imbalance. Pure's second approach to improve performance does mitigate load imbalance but in a more fine-grained manner than AMPI's virtual process migration approach. Pure's approach breaks up work into relatively tiny chunks using standard C++ lambdas that are stolen with low overhead.

3 EXAMPLE

We now illustrate how to use Pure using a simple example program. While the application is a trivial 1-D stencil-like algorithm, the fundamentals of Pure, and its commonalities with MPI, are demonstrated so more complex programs can be written. Listing 1 shows the MPI code. The bulk of the computation occurs in function `random_work`, on each element of array `a`.

Briefly, the `rand_stencil` function enters a loop of `iters` iterations and computes `random_work` on each element of `a`. Importantly, `random_work` takes a variable, unknown amount of time (and, therefore, introduces load imbalance), and it does not modify `a`. Then, `a` is updated by averaging adjacent elements of `temp`. Finally, we use `MPI_Send` and `MPI_Recv` to exchange the low and high elements of `temp` so that the low and high elements of `a` can be computed. Because `random_work` takes a variable amount of time, some ranks will finish their work early and sometimes block on the `MPI_Recv` call on a slow sender.

Listing 2 shows a Pure version of the same function with some key differences. First, the message calls are different, and use the analogous Pure messaging functions, `pure_send_msg` and `pure_recv_msg`, instead of the MPI calls. Note that the arguments are essentially the same as the MPI analogs; we used the included MPI-to-Pure source translator to automatically write the Pure message code. Pure messaging works like MPI messaging; message send-receive pairs are matched by the Pure runtime system and the payload is copied from the source buffer to the receiver buffer. Note that Pure uses a lightweight messaging approach within nodes to achieve lower latency than an optimized MPI baseline.

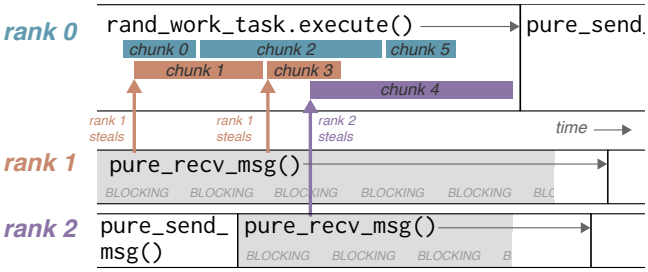


Figure 1: Timeline snippet of example Pure code

The more substantial change exists on lines 4–13 and 15, which are highlighted and define and execute a `PureTask` called `rand_work_task`. Pure Tasks are implemented as C++ lambdas with a particular set of arguments. We leverage the lambda's *capture* arguments, which allow variables outside the lambda's body to be *captured* by both value and reference, despite being out of scope. Pure Tasks can be thought of as snippets of code that the Pure runtime system is responsible for executing, possibly concurrently using multiple threads. Therefore, Pure Tasks should, if possible, be structured in such a way that parts of the work can be run concurrently. In other words, the body of a Pure Task is like an island of concurrent code that the programmer ensures is thread-safe.

In this example, we structure the computation to run on subranges of the work by running loop iterations in parallel. However, the programmer is free to use the chunk ranges to describe concurrency in a different way. The sub ranges, or chunks, are specified using the `start_chunk` and `end_chunk` arguments that are passed to the Pure Task by the runtime. The runtime system is responsible for ensuring all of the work is done, perhaps by different threads, by keeping track of which chunks have been allocated and completed.

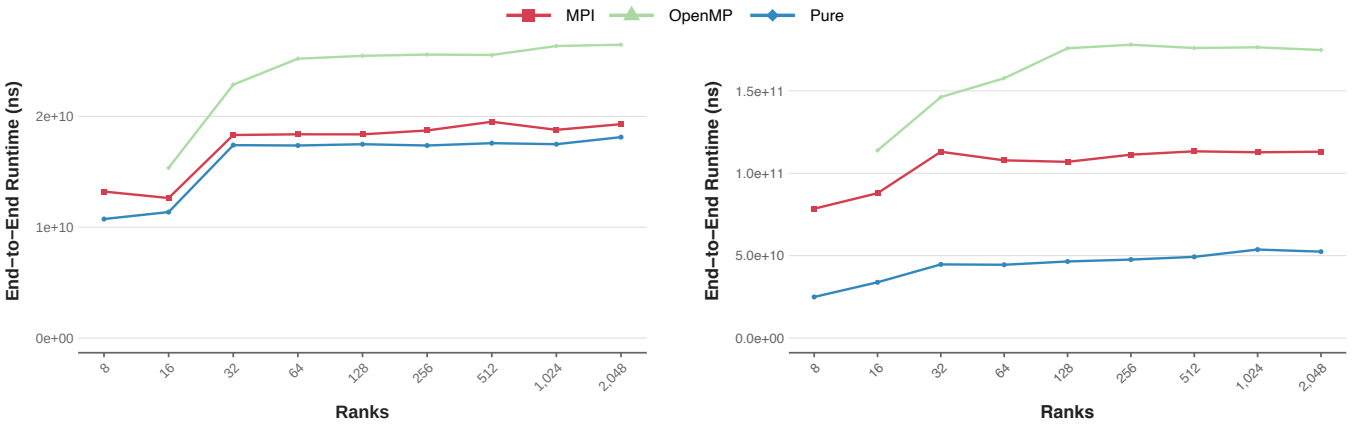
The programmer is responsible for mapping the `start_chunk` and `end_chunk` arguments, given by the Pure Runtime, to something relevant to the application's computation. In this case, we convert them to loop index subranges using the provided `pure_aligned_idx_range` helper function. This helper is aware of cache lines and should be used if possible to prevent false sharing, but an unaligned version is also available for other use cases. Lines 10–12 simply iterate through the computed range and calls the `random_work` function on the subrange of the array. On line 15, which is inside the outer loop, we call `execute` on the Pure Task. This call passes responsibility to the Pure runtime system to execute task and only returns when it is complete.

In this example, `random_work` introduces load imbalance, so some ranks will inevitably be waiting on other ranks to send their message. The Pure Task Scheduler automatically leverages these idle ranks to execute Pure Tasks that may be awaiting execution within the same node (i.e., shared memory region). This is illustrated in Figure 1, where we see three ranks coresident on a single node: rank 0 is executing a Pure Task that is broken into 6 chunks, and ranks 1 and 2 are blocking on `pure_msg_recv` (as shown by the gray shading). Note that time flows to the right in this diagram. Pure ranks are implemented as threads that efficiently share memory, and the Pure Runtime System is aware of both the blocking communication as well as the executing task, so ranks 1 and 2 can attempt to steal chunks of work from rank 0.

We can see that rank 0 starts off executing chunk 0, then rank 1 steals chunk 1, which is run in parallel to rank 0's execution. The Pure Task Scheduler then allocates chunk 2 to rank 0, and chunk 3 to rank 1. Then rank 2 tries to steal some work and is given chunk 4, which runs in parallel. Note that chunks 2 and 4 turn out to be long-running chunks of work due to the random nature of `random_work`. Chunk 5 is given to rank 1, which ends up being a small amount of work and finishes before rank 2 has finished chunk 4. Of course, the Scheduler does not let rank 0 return from the task until all chunks are done; here it completes when chunk 4 is done. Ranks 1 and 2 keep trying to steal more chunks while they are still blocking. Eventually, rank 1 and rank 2's messages arrive. Physically, ranks 1 and 2 run the chunks they steal on their own hardware threads but "peek" into rank 0's context given how Pure Tasks capture relevant application scope.

In our experiments with this simple example, the Pure version running on a single node with 32 ranks (threads) achieved about a 10% speed up over the MPI version from the faster messaging layer and achieved over 200% speed up from using Pure Tasks. These speedups, of course, are a function of the amount of load imbalance, which we chose somewhat arbitrarily. However, in Section 4 we show that for real applications, Pure is able to achieve speedups of 25% to over 200%. Again, these speedups are due to how the Pure runtime system is able to automatically "soak up" idle compute and redirect it to useful work when it exists.

More broadly, the Pure programming model can be summarized as "message passing with optional tasks." Pure messaging and collectives are semantically equivalent to MPI, with minor syntax differences. The Pure rank namespace is "flat" (nonhierarchical) across all nodes despite using threads within nodes and the number of ranks is fixed throughout a Pure program. Pure applications are written in C++ in an SPMD fashion and are internally multithreaded; the ranks within a node (i.e., a shared memory region) are implemented using kernel threads. This means that for applications originally written in MPI, any process-global variables must be made `thread_local` to preserve their semantics.



(a) CoMD: End-to-end runtime

(b) Imbalanced CoMD: End-to-end runtime

Figure 2: CoMD and Imbalanced CoMD End-to-End Performance

4 EVALUATION

We evaluated the performance and scalability of Pure on the NERSC Cori supercomputer (a Cray XC40) on microbenchmarks and three applications, comparing to a highly optimized MPI implementation. In this paper, we outline early results on the CoMD benchmark on Cori on up to 2048 cores. Additionally, we have demonstrated significant performance improvement of Pure over MPI, OpenMP (on a single node), and AMPI on messaging and collective microbenchmarks, other miniapps such as NAS DT and miniAMR. Pure's messaging and collectives speedups range from a few percent to 17×. We have also demonstrated that our results are portable to the NERSC Perlmutter supercomputer, which was launched in 2023.

CoMD [1] is a classical molecular dynamics application. The MPI version of CoMD contains about 3,000 source lines of code, containing both message and collective calls. We converted these MPI calls to Pure calls using the MPI-to-Pure translator. Profiling the MPI CoMD showed no significant load imbalance, so we did not introduce Pure Tasks. We also evaluated the provided MPI+OpenMP version of CoMD. In all cases, we scaled the input sizes weakly, retaining 32,000 elements per rank for MPI and Pure and otherwise ran with application defaults. We ran with 4 OpenMP threads per MPI process and 16 MPI ranks per node, as this yielded the best results after trying multiple configurations. As Figure 2a shows, Pure consistently outperformed, yielding speedups ranging from 7% to 25% relative to MPI and 35% to 50% relative to MPI+OpenMP. Pure's speedups were due to its reduced message and collective latencies.

We also implemented an imbalanced version of CoMD, inspired by [10]. The modified application dynamically adjusts the mesh, thereby varying the amount of force calculation work that each rank must perform each iteration. By profiling the MPI baseline with this change, we found that the majority of the time spent in the original CoMD application was in the `eamForce` function. After introducing load imbalance, the ranks that did have significant work spent a significant amount of time waiting on incoming messages during the halo exchange step. We extracted the core computation code in the `eamForce` function into a Pure Task, breaking two main `for` loops into chunks, as shown previously. Figure 2b shows the end-to-end runtimes. Pure outperforms MPI on all sizes ranging from 8 to 2,048 cores, with speedups ranging from 1.6× to 2.1×, largely due to how ranks stole chunks of the force calculations while waiting on communication. MPI+OpenMP significantly underperformed. Pure is able to efficiently handle dynamic load imbalance across the entire set of cores on each node, which allows it to improve the runtime by over 2× beyond the MPI version and over 3× more the MPI+OpenMP implementation.

5 CONCLUSION

We demonstrated that for representative distributed benchmarks, the Pure versions were faster than a highly optimized MPI implementation. The approaches we outlined should have an increasing impact as nodes continue to add more cores.

REFERENCES

[1] [n. d.]. CoMD Proxy Application. https://proxyapps.exascaleproject.org/app/comd/

[2] Benjamin Brock, Aydın Buluç, and Katherine Yelick. 2019. BCL: A cross-platform distributed data structures library. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[3] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2014. Enabling communication

concurrency through flexible MPI endpoints. *The International Journal of High Performance Computing Applications* 28, 4 (2014), 390–405.

[4] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.

[5] Buss et al. 2010. STAPL: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. 1–10.

[6] Ryan Grant et al. 2019. Finepoints: Partitioned multithreaded MPI communication. In *High Performance Computing: 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20, 2019, Proceedings 34*. Springer, 330–350.

[7] Message Passing Interface Forum. 2012. *MPI: A Message-Passing Interface Standard Version 3.0*. Technical Report. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[8] Chao Huang, Orion Lawlor, and Laxmikant V Kale. 2004. Adaptive mpi. In *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16*. Springer, 306–322.

[9] Laxmikant V Kale and Sanjeev Krishnan. 1993. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 91–108.

[10] Olga Pearce, Hadia Ahmed, Rasmus W. Larsen, Peter Pirkelbauer, and David F. Richards. 2019. Exploring dynamic load imbalance solutions with the CoMD proxy application. *Future Generation Computer Systems* 92 (2019), 920–932. https://doi.org/10.1016/j.future.2017.12.010

[11] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. 2009. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*. IEEE, 427–436.

A ARTIFACT DESCRIPTION

Summary of the Experiments Reported

While Pure has been tested on laptops, clusters, and supercomputers, and is supported on OSX, Linux, and Windows, we focused our experiments on the most representative system we could access: the NERSC Cori Supercomputer. Cori is a Cray XC40. We evaluated Pure on microbenchmarks and three applications, primarily comparing it to a highly optimized MPI implementation. While Pure's primary performance improvements are within a node, the Pure programming model targets problems of any size and any number of nodes. Our experiments used open-source MPI benchmarks, which allowed us to assess the programmability challenge of migrating to Pure.

A.1 Artifact Availability

Software Artifact Availability: All open-source appliations (e.g., CoMD) are publicly available. The Pure source code is not currently open but we are considering publishing the code after we clean it up and improve documentation.

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

Data Artifact Availability: All input data sets that are used by open-source applications are also available as open data. For CoMD, we used the Cu_u6.eam input, provided with the source.

The evaluation of our approach used 4,900 Cori node-hours. Across all our experiments, we collected 10.2M data points which were aggregated into 934 graph data points.

A.2 Baseline Experimental Setup, and Modifications Made for the Paper

Each Cori node contains Intel Xeon "Haswell" E5-2698 v3 processors, which each have 32 cores per node across two NUMA nodes, with two hardware threads per core. Granular details regarding the CPUs of the compute nodes on which we ran our experiments can be see in listing 3. Notably, these CPUs have a 32 kB L1 data- and instruction-cache and a 40 MB L3 cache, the latter of which is shared across all cores within each NUMA node. We enabled Hyperthreading and mostly ran our experiments with 64 hardware threads (and application ranks) per node. All threads within a node are children of the same parent process; that process is an "MPI process" in a multinode Pure application. In single-node Pure applications, MPI is not used at all. We pin ranks to cores for the duration of each application using `pthread_setaffinity_np`.

The nodes are connected via the Cray Aries interconnection network. The Aries network consists of interconnected local routers and high-speed global links that enable fast data transfer between router groups with only one route. Cori's MPI implementation uses the Aries network for its message passing and collective operations, building on the ugni communication layer.

Our primary baseline for comparison was the highly-optimized Cray MPICH MPI (version 7.7.19), the recommended and default MPI implementation on Cori. We also enabled all recommended modules and settings, enumerated in the paper "Preparing NERSC users for Cori, a Cray XC40 system with Intel many integrated cores": XPMEM v2.2.27, which allows processes on the same node to communicate efficiently through shared memory; DMAPP v7.1.1, which speeds up some MPI collectives. The system also uses the Cray-specific `ugni` communications layer. Also, we compiled with `-O3`, `-march=native`, and `-std=c++17`. We also enabled 2MB Linux Hugepages and used the default system compiler (icc 19.1.2.254). The specific software modules we used for experiments are shown in listing 4.

Ranks were mapped to cores identically for the Pure and MPI runs. We profiled our MPI baseline applications with Cray's CrayPAT profiler, which recommends an improved rank-to-node mapping and used the recommended mapping for both the MPI and Pure implementations. We measured clock cycles using `rdtscp`, taking the median result across 10 runs in most cases (although for some situations with very large node counts, we ran as few as one run due to limited machine time, potentially contributing to some noise in the results). We typically measured scaling by increasing the problem size as we increased the number of ranks (i.e., weak scaling).

We show our results for end-to-end runtime in nanoseconds, with a processor clock frequency of 2.3 GHz. Within a single node (i.e., 2-64 threads), we also compared against OpenMP when possible. We used the default and recommended OpenMP implementation and settings on Cori. Note that OpenMP only supports single node runs (i.e., using shared memory), so results only scale up to 64 threads in the results in this chapter.

Generally we used best practices and recommendations to optimize the MPI Baseline and the same settings for both the MPI baseline and Pure whenever possible, as specified in the "Preparing" paper mentioned above. MPI implementations have different levels of thread support, ranging from no thread support to arbitrary thread support. The default, `MPI_THREAD_SINGLE`, provides no thread safety and is the best-performing MPI thread mode. We used this mode for our baseline experiments.

```
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):              64
On-line CPU(s) list: 0-63
Thread(s) per core:  2
Core(s) per socket:  16
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               63
Model name:          Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
Stepping:            2
CPU MHz:             3141.651
CPU max MHz:         2301.0000
CPU min MHz:         1200.0000
BogoMIPS:            4600.31
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            40960K
NUMA node0 CPU(s):   0-15,32-47
NUMA node1 CPU(s):   16-31,48-63
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 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 cpuid_fault epb invpcid_single pti
                     intel_ppin ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept
                     vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
                     cqm xsaveopt cqm_llc cqm_occup_llc dtherm ida arat pln pts
                     md_clear flush_l1d
```

Listing 3: Cori Compute Node CPU Details via `lscpu`

Our multinode Pure experiments used `MPI_THREAD_MULTIPLE`, which Pure requires as any thread can make MPI calls. This mode is considered the slowest MPI mode due to synchronization overhead in the MPI runtime system; we did notice some performance impact in the Pure applications. For single node runs, Pure does not use MPI at all.

A.3 Compute Node Details

Listings 3 to 5 show details of the Cori compute nodes on which we ran our experiments.

```
Currently Loaded Modulefiles:
  1) modules/3.2.11.4
  2) craype-network-aries
  3) intel/19.1.2.254
  4) craype/2.7.10
  5) cray-libsci/20.09.1
  6) udreg/2.3.2-7.0.3.1_3.41__g5f0d670.ari
  7) ugni/6.0.14.0-7.0.3.1_6.23__g8101a58.ari
  8) pmi/5.0.17
  9) dmapp/7.1.1-7.0.3.1_3.42__g93a7e9f.ari
 10) gni-headers/5.0.12.0-7.0.3.1_3.25__gd0d73fe.ari
 11) xpmem/2.2.27-7.0.3.1_3.26__gada73ac.ari
 12) job/2.2.4-7.0.3.1_3.33__g36b56f4.ari
 13) dvs/2.12_2.2.224-7.0.3.1_3.32__gc77db2af
 14) alps/6.6.67-7.0.3.1_3.41__gb91cd181.ari
 15) rca/2.2.20-7.0.3.1_3.44__g8e3fb5b.ari
 16) atp/3.14.9
 17) perftools-base/21.12.0
 18) PrgEnv-intel/6.0.10
 19) craype-haswell
 20) cray-mpich/7.7.19
 21) craype-hugepages2M
```

Listing 4: Cori Loaded Modules for Experiments

```
lsb_release -a. # 20220816
LSB Version: n/a
Distributor ID: SUSE
Description: SUSE Linux Enterprise Server 15 SP2
Release: 15.2
Codename: n/a
```

Listing 5: Cori CPU info