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

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. Pure leverages shared memory in two ways: (a) by allowing cores to steal work from each other while waiting on messages to arrive, and, (b) by leveraging efficient lock-free data structures in shared memory to achieve high-performance messaging and collective operations between the ranks within nodes. 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 and the miniAMR adaptive mesh refinement applications scaling up to 4,096 cores.

CCS Concepts: • **Computing methodologies** → *Parallel programming languages*; *Shared memory algorithms*; *Massively parallel algorithms*; *Distributed programming languages.*

Keywords: parallel programming models; distributed runtime systems; task-based parallelism; concurrent data structures; lock-free data structures

ACM Reference Format:
James Psota and Armando Solar-Lezama. 2024. Pure: Evolving Message Passing To Better Leverage Shared Memory Within Nodes. In *The 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24), March 2–6, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3627535.3638503

1 Introduction

In the late nineties, high performance computing shifted from using large vector machines to clusters with lots of single processor machines connected by a network. MPI quickly became the standard parallel programming approach for these distributed memory machines, enabling ranks to communicate using message passing. Since then, programmers put significant effort into creating MPI applications, partitioning computation carefully to garner the benefits of locality on their clusters.

Over the last decade, hardware has shifted from uniprocessor clusters to multicore clusters, with cores on nodes sharing memory and nodes communicating through a network. The community has been on the lookout for new paradigms to more fully take advantage of modern clusters. The first approach, and perhaps still the most popular, is to retain the unified MPI programming approach across all cores, but to attempt to make MPI runtime system implementations better leverage shared memory. A huge advantage to keeping the status quo is that it doesn't requires any changes to applications. However, this approach potentially leaves performance on the table because of the strictures that the MPI standard places on the behavior of its interface.

A second common approach is the so-called MPI+X, or "hybrid" programming, which still uses MPI across nodes, but uses shared memory parallelism within nodes. With this approach, programmers aim to more closely match the programming model to the hardware in an attempt to gain improved performance. Yet, this approach introduces a significant programming challenge, as the programmer must now manage and optimize two programming models and carefully partition the application hierarchically. Furthermore, because of Amdahl's Law, the programmer must aggressively use, say, OpenMP throughout the code to leverage available cores. This is one reason why both our own experience and others in the literature have shown that getting hybrid schemes to even match, let alone surpass, the performance of MPI-everywhere often requires significant effort.

The community has experimented with many other approaches, including PGAS, which provides the illusion of shared memory across the entire cluster, and also implicit parallel programming languages like Legion, Chapel, and X10, which give the programmer higher level abstractions and attempt to efficiently orchestrate the application automatically. Despite the progress and proliferation of new approaches, a significant portion of modern HPC applications still using MPI [50]. MPC [43] and AMPI [33] also use threads as MPI

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03.
https://doi.org/10.1145/3627535.3638503

ranks and strive to leverage shared memory internally to improve performance. For example, AMPI, which we analyze in Section 5.2, has an SMP mode that improves messaging performance and also virtualizes application ranks and migrates them to mitigate load imbalance.

Surprisingly, the MPI-only approach often outperforms the hybrid applications. Yet, we believe the narrowness of the interface and inability to fully leverage shared memory within nodes causes MPI to leave a lot of performance unrealized. What we propose in this paper builds upon the first, MPI-everywhere approach, and breaks some of MPI's assumptions to more efficiently leverage shared memory while not requiring a significant rewrite of the program. MPI application developers have already developed effective approaches to cleanly partitioning applications that achieve strong locality and performance. Our proposed parallel programming system, Pure, offers a programming model that is similar to MPI, thereby leveraging the HPC community's existing MPI knowledge and extant application base.

Pure is inspired by MPI; the 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.

We also show that the Pure programming model is semantically similar to MPI, making learning it and migrating from existing applications straightforward. Furthermore, we show that we can easily migrate most of existing MPI programs with a provided source-to-source translation tool. In summary, this paper makes the following contributions:

```
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

1. We introduce a programming model and runtime system that efficiently integrates message passing and task parallelism using standard C++ features. By widening the runtime system interface, we show that the runtime system is able to efficiently and automatically overlap communication and task execution.
2. We show how modern C++ can help support a more flexible application interface for parallel runtime systems.
3. We describe the design of a lock-free, multithreaded and distributed runtime system which achieves significant within-node speedups over MPI.
4. We show that with only minimal source code changes to existing MPI applications, we can get substantial performance improvements over a state-of-the-art MPI implementation on both microbenchmarks and three applications.

2 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. In the MPI version in Listing 1, the bulk of the computation occurs in function `random_work`.

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

```
1  void rand_stencil_pure(double* const a, size_t arr_sz, size_t iters,
2                         int my_rank, int n_ranks) {
3      double   temp[arr_sz];
4      PureTask rand_work_task = [a, temp, arr_sz,
5                                 my_rank](chunk_id_t           start_chunk,
6                                          chunk_id_t           end_chunk,
7                                          std::optional<void*> cont_params) {
8          auto [min_idx, max_idx] =
9                  pure_aligned_idx_range<double>(arr_sz, start_chunk, end_chunk);
10         for (auto i = min_idx; i <= max_idx; ++i) {
11             temp[i] = random_work(a[i]);
12         }
13     }; // ends defining the Pure Task rand_work_task
14     for (auto it = 0; it < iters; ++it) {
15         rand_work_task.execute(); // execute all chunks of rand_work_task
16         for (auto i = 1; i < arr_sz - 1; ++i) {
17             a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
18         }
19         if (my_rank > 0) {
20             pure_send_msg(&temp[0], 1, PURE_DOUBLE, my_rank - 1, 0,
21                           PURE_COMM_WORLD);
22             double neighbor_hi_val;
23             pure_recv_msg(&neighbor_hi_val, 1, PURE_DOUBLE, my_rank - 1, 0,
24                           PURE_COMM_WORLD);
25             a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
26         } // ends if not first rank
27         if (my_rank < n_ranks - 1) {
28             pure_send_msg(&temp[arr_sz - 1], 1, PURE_DOUBLE, my_rank + 1, 0,
29                           PURE_COMM_WORLD);
30             double neighbor_lo_val;
31             pure_recv_msg(&neighbor_lo_val, 1, PURE_DOUBLE, my_rank + 1, 0,
32                           PURE_COMM_WORLD);
33             a[arr_sz - 1] =
34                     (temp[arr_sz - 2] + temp[arr_sz - 1] + neighbor_lo_val) /
35                     3.0;
36         } // ends if not last rank
37     }     // ends for all iterations
38 }
```

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

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 our MPI-to-Pure source translator to automatically write the Pure message code. Pure messaging semantics are like MPI's: the sender buffer is copied to the receiver buffer. Note that Pure uses a lightweight messaging approach within nodes to achieve lower latency than an optimized MPI baseline.

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++

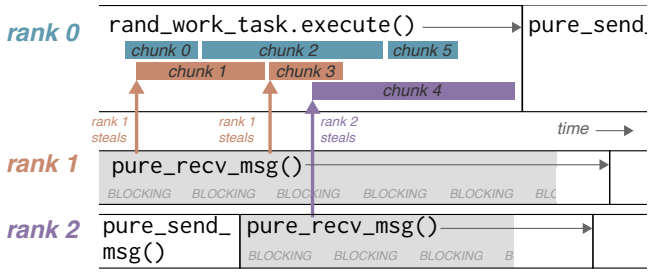


Figure 1. Timeline snippet of example Pure code

lambdas [1] 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 either value or reference and used when the lambda is executed. Pure Tasks can be thought of as snippets of application code that the Pure runtime system is responsible for executing, possibly concurrently using multiple threads. Therefore, Pure Tasks should 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 a small island of concurrent code that the programmer must ensure is thread-safe.

In this example, we structure the computation to run on subranges of the work by running independent 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. 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 in 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 chunks of Pure Tasks that are awaiting execution within the same node (i.e., shared memory region). Figure 1 illustrates this, 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` (see gray shading). Note

that time flows to the right in this figure. 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. 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. 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 (from any other rank) 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 achieved a 10% speedup over the MPI version from Pure's faster messaging, and achieved over 200% speedup from using Pure Tasks. These speedups, of course, are a function of the amount of load imbalance, which we chose arbitrarily. However, in Section 5 we show that for real applications, Pure is able to achieve similar speedups. 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.

3 Programming Model

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" (non-hierarchical) across all nodes, despite using threads within nodes. 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. Process-global variables in Pure applications must be removed or made `thread_local` to preserve their semantics and prevent race conditions. Note that this currently presents a challenge for applications that use closed-source libraries that make use of global variables.

For applications that contain load imbalance, programmers may use Pure Tasks in parts of the application that: (1) are a computational hotspot; (2) can be structured to execute concurrently. Appendix D offers recommendations on writing Pure programs.

3.1 Messaging and Collectives

Pure messaging calls, `pure_send_msg` and `pure_recv_msg`, are similar to `MPI_Send` and `MPI_Recv`, as seen in Section 2. Non-blocking calls are also available. The Pure runtime system guarantees that the message will eventually be delivered when the message calls return and messages are guaranteed to be delivered in send-order for each sender-receiver pair. Once the send call returns, the application can safely reuse or free the message buffer.

Pure also implements the following collectives, which are semantically equivalent to MPI's: reduce; all-reduce; barrier; broadcast. Pure also contains communicators, like MPI, which are created with `pure_comm_split`.

Pure applications should be written using modern C++, and must be compiled with the `std=c++11` flag (or newer). The Pure distribution is packaged with a Make-based build system, which automatically sets appropriate flags and links the Pure Runtime Library, `libpure`, and defines a number of useful debugging and profiling targets. See the Appendix for details.

3.2 Pure Tasks

Pure Tasks allow the programmer to describe how some part of the application's computation can be broken into "chunks" of work that can be executed concurrently by the Pure Runtime System. Use of Pure Tasks is optional; programmers should selectively add tasks when the work is efficiently partitionable into chunks and when it contributes to load imbalance. Anecdotally, we found that in our experiments we added Pure Tasks to fewer than 10% of the lines of code.

Pure Tasks are implemented using C++ lambdas and are executed synchronously when the owning rank calls `execute` on it. A given rank will only have at most one task executing at a time. C++ lambdas support *variable capture*, which conveniently allows context from one rank to be efficiently shared with other ranks that are helping execute chunks of a task. Typically the same task is defined once and executed many times over the course of an application, such as once per timestep in a scientific application.

The programmer is responsible for writing the task in a way that uses captured arguments, as well as the arguments passed by the runtime (i.e., chunk range and extra arguments from the application), to execute a non-overlapping portion, of the work upon each invocation. Tasks must also avoid dependencies with each other, but because they are executed completely during the `execute` call, their execution will not race with code that exists outside the task.

Tasks have a single method, `execute`, which is called by the application code and takes an `optional<void*> per_exe_args` argument that the runtime passes to each invocation of the task. We found this feature useful when the task body required values that changed upon successive task executions, and therefore could not be captured upon task definition. For

example, the programmer may define a local `struct` on the stack and pass a pointer to it to `execute`.

As shown in the example, the first two arguments to PureTasks are unsigned integers `start_chunk` and `end_chunk` that specify the range of chunks to be executed. The chunk arguments are assigned by the Pure Runtime to ensure that all chunks are executed exactly once, possibly concurrently and in an arbitrary order. Pure uses a range of chunks to allow the scheduler the flexibility to assign multiple chunks at a time (e.g., it assigns more at the beginning of a task's execution and to the owning rank). The number of chunks that the runtime breaks a given Task into is determined by the Pure Task Scheduler, but will never exceed a fixed maximum value specified by the `PURE_MAX_TASK_CHUNKS` Makefile variable.

The current interface requires the programmer to manually convert chunk numbers to array indicies. This is even more burdensome for multi-dimensional arrays. We aim to extend the current interface with cleaner, higher-level interfaces more akin to TBB's `parallel_for` [44].

The Pure programmer is responsible for ensuring thread-safety within the Pure Task definition so that multiple chunks of the same task that are executing concurrently do not race with each other. In just one task in one of the benchmarks shown in this paper (CoMD) we had to deal with multiple threads concurrently writing to the same memory location. We addressed this by changing an `int` array to a `std::atomic<int>` array. All other task definitions were embarrassingly parallel as concurrently executing threads wrote to different addresses. Also, by default Pure creates chunks that are cacheline-aligned to prevent false sharing.

4 Pure Runtime System

The Pure runtime system is implemented as a multithreaded and distributed runtime system library. Pure application developers include `pure.h`, build with a C++17 compiler, and link with `libpure`. The Pure Runtime automatically looks for opportunities to overlap computation and communication transparently to the programmer, typically during high-latency communication events.

Overall, Pure is responsible for efficiently: (1) creating and pinning the necessary processes and threads, and launching the application; (2) managing communication and collective operations between ranks; (3) managing internal memory buffers and data structures; (4) scheduling and executing Pure Tasks, if any are used in the application.

4.0.1 Rank Initialization and Mapping. Pure ranks are logically like MPI ranks, serving as a named thread of execution that can explicitly communicate with other ranks. Unlike in MPI, where ranks are implemented as operating system processes, ranks in Pure are implemented as kernel threads [3], which are children of MPI processes. Internally, Pure runs MPI for cross-node communication on multinode applications, and does not use MPI at all for single node runs, but Pure applications can not directly make MPI calls. Pure programs can be configured via Makefile flag to run either one MPI process per node (or NUMA node), and run as many threads as cores per node (or NUMA node). The application programmer only is aware of a flat rank namespace; notions of different nodes, threads, MPI processes, variable latencies, etc. are all abstracted away from the programmer.

Like MPI, Pure supports mapping ranks to nodes in an arbitrary fashion. By default, Pure allocates ranks using SMP-style placement and pins ranks to cores, but it supports arbitrary rank-to-node-to-core mappings. Pure also supports CrayPAT [18] rank reordering files. While these hierarchical hardware details are abstracted from the programmer, Pure internally uses this information to optimize key functionality.

When a Pure application starts, the application's original `main` function is not called directly. The underlying MPI program with one MPI rank per node contains a `main` function that is defined in the Pure Runtime. That function initializes key Pure data structures, forks and pins the threads, and those threads each run an `__original_main` function which is a renamed version of the original `main` function from the application code. Upon the application's completion, the application's `__original_main` returns back to the Pure runtime, which then finalizes MPI, cleans up, and returns. Pure includes special debugging and profiling modes to assist in application development.

4.0.2 The Spin-Steal Waiting Loop (SSW-Loop). When a Pure rank encounters a blocking event, such as waiting for a message to arrive, it must wait. Instead of yielding or idly waiting, Pure ranks perform the "spin-steal wait loop," or SSW-Loop. This loop simply checks the blocking condition (i.e., "has message arrived?") and, if not, tries to steal work. If the blocking rank is able to help another thread in its process that is coincidentally executing a concurrently-executable Pure Task, it does so. We use this approach in dozens of places in the Pure runtime, increasing the chances that Pure Tasks can be concurrently executed. Given that we are pinning threads to CPUs and only running one application on the node, we chose to actively spin instead of yielding. The SSW-Loop allows ranks to act "polymorphically," both as first-class application ranks and also helper threads that assist other ranks execute their work. Stealing threads, or "thieves," do just one chunk of stolen work before checking on their blocking event again; Pure prioritizes work owned by each first class rank, taking a *work-first* scheduling policy [5]. Pure's approach is unlike systems that use helper threads to implement work stealing or communication, as application ranks directly do the stealing.

4.0.3 Implementation Complexity. Pure was written using C++17 and the C++ Standard Library. The Pure runtime contains 21k source lines of code (SLOC) and Pure Tools

another 14k SLOC. Pure has been tested on laptops and clusters, and only requires a C++17 compiler, a Unix-like OS, and MPI. The Pure source code is available at https://github.com/psota/pure.

4.1 Point-to-Point Messaging in Pure

Pure implements blocking and non-blocking point-to-point messaging that is semantically equivalent to MPI's messaging. While the programmer's view of sending and receiving messages are the same regardless of where the sending and receiving ranks are physically located, Pure uses three different messaging strategies internally. The specific approach used depends on the size of the message and depending on if the sender and receiver rank are co-located on the same node or not. Pure supports both blocking and non-blocking sends and receives, and, like MPI, supports multiple non-blocking calls in flight before the corresponding wait call is made. For all message types described below, we allocate a persistent "channel" object that is stored in the runtime system and is reused throughout the program; the internal Channel Manager maps message arguments (e.g., ranks, tags, datatypes, etc.) to the appropriate data structure, creating it on-demand if needed.

4.1.1 Intra-node Short Messages. For short messages (e.g., <8kB, configurable) between two ranks on the same node, we implemented a lock-free circular queue with acquire-release memory semantics that buffers a fixed number of messages. The sender thread copies the message into the PureBufferQueue (PBQ) when space becomes available, and the receiver thread copies the message out when available. Note that this two-copy scheme is inspired by MPI's buffered send approach [22], and is typically used in messaging systems for short messages because the copy overhead is relatively small, and this allows the sending rank to return from the "send" call to [hopefully] proceed with other useful work.

Both threads employ the SSW-Loop to wait, automatically overlapping computation with communication when possible. We use a single contiguous buffer that stores all message "slots", and we use simple pointer arithmetic to align each slot to cacheline boundaries to avoid false-sharing between the writing sender thread and reading receiver thread. We experimented with different approaches while implementing fast messaging between threads and found that the key drivers of performance were: (1) reusing buffers and channel data structures; (2) using atomics instead of locks; (3) avoiding false sharing. The configurable number of slots within the PBQ was not a material performance driver in our experiments.

4.1.2 Intra-Node Large Messages. For large messages (e.g., ≥8kB) where the sender and receiver ranks are on the same node, we take a similar strategy to the PBQ, but with a single memory copy from sender to receiver, inspired by MPI's rendezvous mode [26]. We use a lock-free fixed-size circular buffer to store the receiver's receive call arguments (i.e., destination buffer, size, etc.). The sender rank waits via SSW-Loop on the metadata queue entry, and then copies the message payload directly into the receiver's desired buffer. The sender signals completion to the receiver using a different lock-free queue by inserting the number of bytes transferred.

4.1.3 Inter-Node Messages. For messages between ranks on different nodes, we transparently leverage `MPI_Send` and `MPI_Recv`. Using an internal thread-rank-process-node mapping data structure created during Pure initialization using a distributed consensus algorithm, we translate Pure ranks to MPI ranks within the given communicator. The runtime must also ensure that the proper receiver thread on the receiving node gets the correct message destined for it, and currently there is no native MPI mechanism to do this thread-level routing. Given that we run MPI in MPI_THREAD_MULTIPLE mode, any thread can make MPI calls and could receive a message destined for another thread on the same node. We resolve this challenge by encoding the sender thread number and receiver thread number (within their respective processes) into upper bits of the MPI tag argument. In our experiments, using the upper 6 bits of the tag was adequate for the 64 threads we used (at most) and the applications we ran. However, enabling a large tag MPI mode would also mitigate most tag overflow issues.

4.2 Collective Communications in Pure

Pure's collective communication operations are semantically equivalent to MPI's and are implemented using data structures within nodes built from the ground up. As with point-to-point messages, we use MPI's collectives across nodes. Despite using MPI across nodes, we achieved significant speedups over MPI on single and multi-node benchmarks.

We experimented significantly with different collective designs, starting with simple lock-based approaches and atomic counters across all threads. After significant exploration, we landed on new lock-free data structures that generally have a leader thread that orchestrates the collective process, leverages other threads for help as long as they don't false-share, and makes MPI collective calls as needed. We use a simple static leader election process, which outperformed a compare-and-swap based "first thread in" process. We walk through our two designs for all-reduce, one for each small and large data, and then generalize these approaches to other collectives.

4.2.1 All-Reduce on Small Data. The all-reduce operation requires each participating rank to contribute an input array and receive an element-wise reduction of all ranks' inputs in an output array. Figure 2 shows a simplified version of the concurrent data structure we created to execute the all-reduce for small data, called the Sequenced Per-Thread

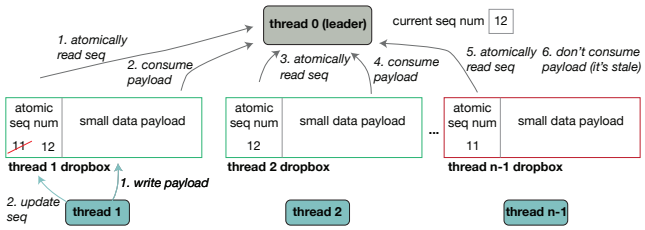


Figure 2. Sequenced Per-Thread Dropbox (SPTD)

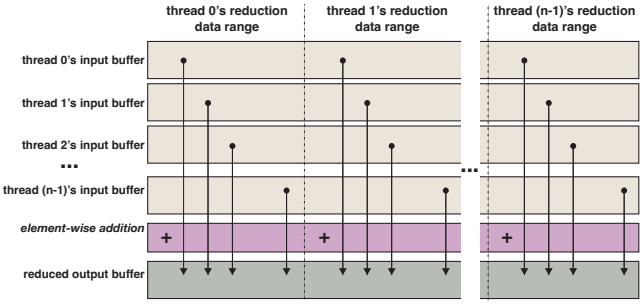


Figure 3. Partitioned Reducer on Large Data

Dropbox (SPTD). It provides an efficient lock-free mechanism for synchronizing and optionally sharing data between the leader thread and each non-leader thread in a pairwise fashion. We use a technique inspired by *flat-combining* [28] to execute the reduction, assigning thread 0 of the communicator as the leader. With this approach, which we used for arrays up to 2kB, each non-leader thread first copies its data into the SPTD and then pairwise-synchronizes with the leader thread to indicate that its input values are available. Then, the leader does the element-wise reduction computation on all of the input arrays. We use cacheline aligned buffers to achieve vectorized computation and pad buffers to avoid false sharing.

The leader threads on each node use `MPI_Allreduce` to reduce each node's reduction. After the leader has completed the reduction, it synchronizes and the non-leader threads each copy out the final reduced value into their respective private result buffers. While our approach serializes all computation through the leader thread, in practice we found this approach to work better than trying to have too many threads collaborating on too little work. We use atomic sequence numbers, with C++ acquire-release semantics, to indicate that a payload is ready and also when the final reduction is done. We found the pairwise synchronization offered by this approach vastly outperformed a shared atomic counter approach. Threads use the SSW-Loop to wait.

4.2.2 All-Reduce on Large Data. For reductions of arrays larger (e.g., ≥2kB), the actual reduction computation starts to become the driver of performance, so we take a different strategy. Here we employ as many as all the threads to concurrently execute the reduction operation by pulling directly from each threads' input data buffers using shared memory and writing the reduced result directly to the output buffer. The reduction work is divided into roughly equal chunks (in multiples of aligned cachelines to avoid false sharing and achieve vectorization), as seen in Figure 3. For example, a reduction of 4kB input arrays on a machine with 64B cachelines will be broken into 64 chunks, so up to 64 threads will concurrently execute the reduction operation. Some threads may not have any work to do if the number of threads is larger than the number of cachelines in the input buffers.

Threads alert other threads that they have "arrived" using a SPTD, but instead of copying in their data, they just set a pointer to their buffer (which is directly readable in shared memory) before incrementing their sequence number. After each thread executes their chunk of work by reducing all T threads' buffers, they signal that computation is done using a sequenced atomic counter with the leader thread. The leader thread executes the cross-node all-reduce using `MPI_Allreduce` and propagates the final reduced value using another atomic sequence number.

Pure implements other collectives (e.g., broadcast; barrier; reduce) using the same techniques outlined above. Pure's collectives significantly outperform their MPI and OpenMP analogs, as shown in Section 5.5 and Appendix A.

4.3 The Pure Task Scheduler

The Pure runtime maintains an `active_tasks` array in shared memory containing atomic pointers to running tasks, with one entry per rank (thread) on a node, each initialized to `nullptr`. When a task is executed, the runtime initializes relevant state and atomically updates the `active_tasks` entry for the owning rank. When `active_tasks` contains a non-null pointer, it indicates to other threads that this task is "open for stealing." Note that the rank (thread) that owns the task serves as the leader of the concurrent execution process; there are no other special threads running within the Pure runtime. After the task is initialized, the rank that owns the task starts executing chunks, as described below.

Other threads, during their SSL-Loop, probe `active_tasks`, atomically loading values looking for a non-null entry. Thief threads probe `active_tasks` randomly, as in Cilk [5] and other work-stealing schedulers. If they find a non-null task, they attempt to steal a chunk.

The chunks of a task are always executed by the owning rank and possibly other stealing ranks. Two atomic integer values drive the concurrent execution, `curr_chunk` and `chunks_done`. The owner rank and thief ranks run the same concurrent execution function, although the thieves execute just one chunk and then return, while the owning rank executes until all chunks are done. Threads use `fetch_add` to determine which chunk to execute, although they return if the value is already greater than the total number of chunks.

Threads also atomically increment chunks_done if they successfully finished any chunks; the owner only stores this locally to avoid a cache miss. Finally, the owning (leader) rank waits until all chunks have been executed, which, as illustrated in Figure 1. When all chunks have been executed, the owning rank resets their active_tasks entry to null and returns to the application.

Note that task chunks are executed on the same hardware thread as the application rank; every hardware thread is allocated to Pure application ranks. Pure currently does not leverage hardware accelerator hardware (e.g., GPUs) to accelerate task execution, but we believe the Pure architecture would support this.

The Pure scheduler has different chunk execution modes and stealing algorithms. For example, we implemented both a single chunk mode and a guided-self scheduling mode [46], which is a work partitioning algorithm whereby larger chunks of work are allocated (stolen) at first, and smaller chunks are allocated (stolen) later. The scheduler also has a NUMA-aware stealing mode (which prefers steals from victim threads on the same NUMA node) as well as a "sticky" stealing mode where thieves return to their most recently stolen task which may still be active. For simplicity in our evaluations, we only used "single chunk" and random steal mode in our evaluations, and did not find significant performance differences from these enhancements.

5 Evaluation

We evaluated the performance and scalability of Pure on the NERSC Cori supercomputer [27] (a Cray XC40) on microbenchmarks and three applications, comparing to a highly optimized MPI implementation. Our experiments used open-source MPI benchmarks, which allowed us to assess the programmability challenge of migrating to Pure. Our experiments varied the number of ranks from 2 to 65,536, which used up to 1,024 Cori nodes. Each Cori node shares memory across two Intel Xeon "Haswell" E5-2698 v3 processors, each of which have 32 hardware threads and constitute NUMA regions, which can communicate with each other via shared memory. The Cori nodes are connected via the Cray Aries network [6]. We enabled Hyperthreading and mostly ran our experiments with 64 hardware threads (and application ranks) per Cori node. We pinned ranks to cores for the duration of each application. Note that this paper's Appendix has additional results.

Our baseline for comparison was the highly-optimized Cray MPICH MPI (version 7.7.19), the recommended and default MPI on Cori. We also enabled all recommended modules /settings [27]: XPMEM v2.2.27 [30], which allows processes on the same node to communicate efficiently through shared memory; DMAPP v7.1.1, which speeds up some MPI collectives. Also, we compiled with -O3 and enabled 2MB Linux Hugepages and used the default system compiler (icc 19.1.2.254). Ranks were mapped to cores identically for the Pure and MPI runs. We profiled our MPI baseline applications with Cray's CrayPAT profiler [18], which recommends an improved rank-to-node mapping and used the recommended mapping for both the MPI and Pure implementations. Our baseline MPI experiments used MPI in MPI_THREAD_SINGLE, the fastest mode, and Pure with MPI_THREAD_MULTIPLE, considered the slowest MPI mode due to synchronization overhead in the MPI runtime system. We measured clock cycles using rdtscp, taking the median result across 10 runs.

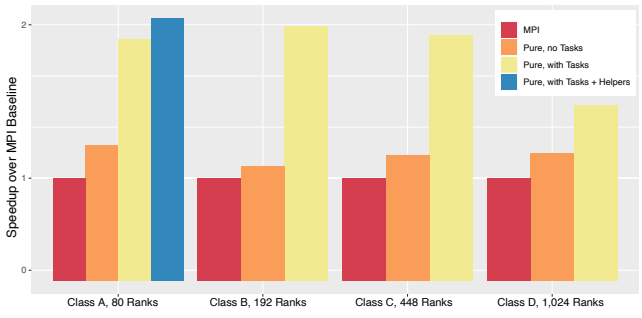
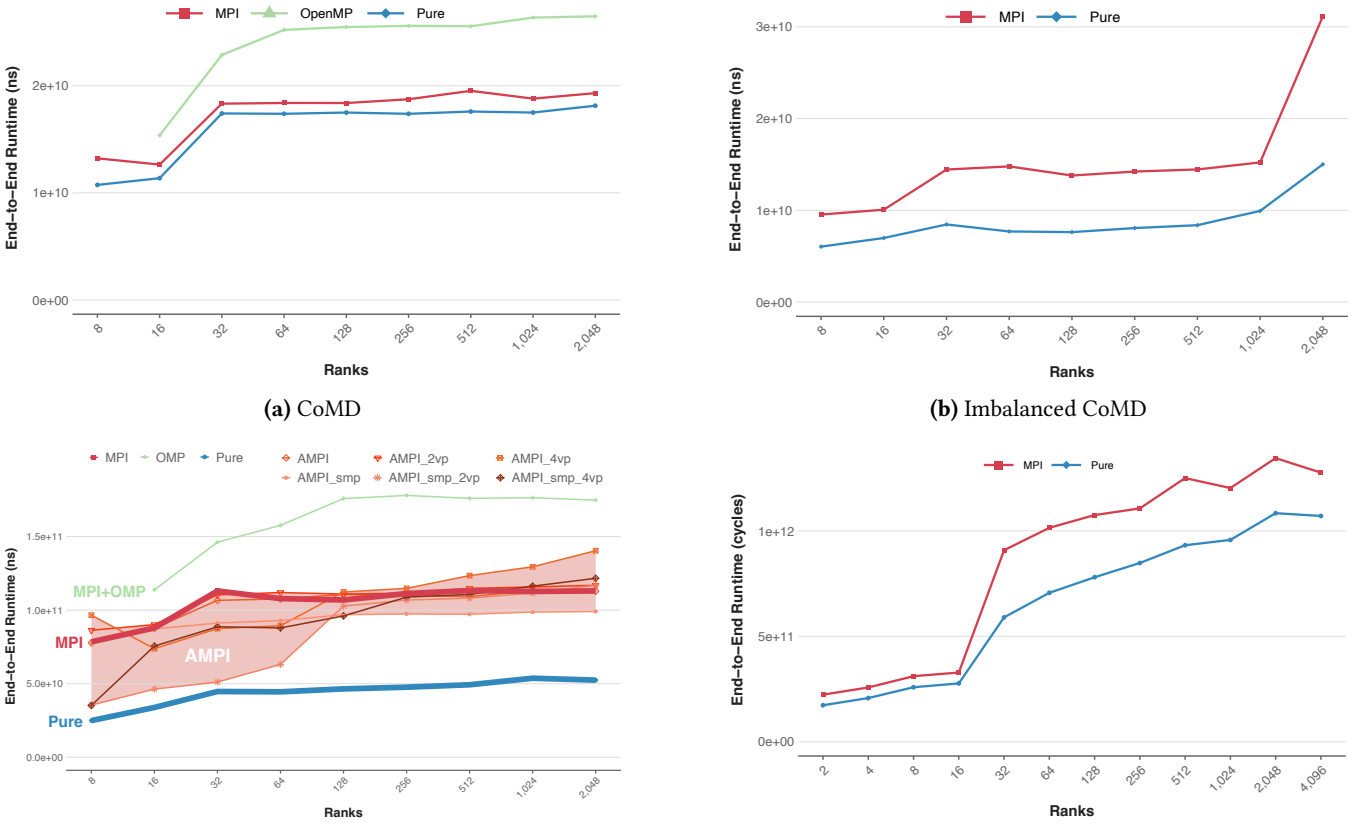


Figure 4. DT: Pure speedup over MPI

5.1 NAS DT Benchmark

NAS DT is a "data traffic" MPI benchmark that implements communication graphs, contains communication bottlenecks and load imbalance. We ran the SH, or "shuffle" graph topology, which has particularly unwieldy load imbalance. The original C code has 900 lines and uses just 8 MPI_Send and MPI_Recv calls. We converted the MPI calls to Pure automatically using our MPI-to-Pure source translator. For size D, we ran with 16 ranks per node due to memory limitations. On size A, we used 40 ranks per node and on sizes B and C we ran with 64 ranks per node.

Figure 4 shows speedups over MPI for three configurations, showing the impact of different Pure features. The orange bars show speedups due to Pure messages, ranging from 11% to 25%. Then, we added Pure Tasks to three sections of code due to significant application load imbalance; this was a two-day programming effort. A significant number of ranks block on incoming communication while upstream (sending) ranks are busy computing. The green bars show the Pure speedup due to both messaging and Pure Tasks, ranging from 1.7× to 2.5×. Finally, the blue bar in the figure for size A shows further benefit when Pure helper threads are enabled on unused cores. This was possible because for size A, which required 80 ranks, we had 24 unused cores on each node. Pure helper threads are simply extra threads that continuously try to steal work. For DT size A, this increases the speedup over MPI from 2.3× to 2.6×. Note that DT size A was the only benchmark in this paper for which we used helper threads; other benchmarks were able to scale up ranks to use all of the cores on a node.



(a) CoMD

(b) Imbalanced CoMD

(c) Dynamic imbalanced CoMD

(d) miniAMR

Figure 5. miniApp End-to-End runtimes

5.2 CoMD Benchmark

CoMD [2] is an open-source classical molecular dynamics application. CoMD's MPI version contains about 3,000 source lines of code, containing both message and collective calls. We used the MPI-to-Pure translator to create the Pure version. Profiling the default application did not show significant load imbalance, so we did not introduce Pure Tasks. We also evaluated the provided MPI+OpenMP version. We used all 64 hardware cores per node and we scaled the input sizes weakly. We ran the simulations for 150 iterations 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.

Figure 5a shows the end-to-end runtimes for all three configurations from 8 to 2,048 cores. Pure consistently performed the best across all problem sizes, 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 latency. The MPI+OpenMP version underperformed the MPI version.

5.2.1 Imbalanced CoMD Benchmark. We also implemented a statically imbalanced version of CoMD, inspired by [42]. The modified application does this by eliding atoms that exist within spheres upon initialization of the mesh, thereby varying the amount of force calculation work that each rank must perform. 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, this remained the case for the ranks that didn't have atoms removed from their part of the mesh, but the ranks that did have atoms removed 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 5b 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.

5.2.2 Dynamic Load Imbalance with AMPI Comparison. AMPI [33], described more in Section 6, is an MPI-compliant runtime system built on top of Charm++ [32]. It can reduce load imbalance between ranks by over-decomposing an MPI program (i.e., running more application ranks than processors) and trying to efficiently migrate and

schedule these virtualized ranks to reduce load imbalance. AMPI has multiple modes (i.e., "non-SMP" and "SMP" modes) and programmers can manually vary the number of "virtual ranks" per core.

Figure 5c shows the results of CoMD with dynamic load imbalance, where the work allocated to each rank varies throughout the simulation. Here we compare Pure to MPI and MPI+OpenMP, plus six different AMPI variations. For the "SMP AMPI" version, we allocated one non-rank worker thread per NUMA node, and so the AMPI SMP configuration got extra hardware (e.g., for a 64 ranks, we ran with one node for MPI and Pure, but 2 nodes for AMPI SMP). The fastest AMPI variation outperformed MPI version for all sizes. The SMP AMPI version with two virtual ranks per physical core performed best up to 64 cores (i.e., within one node) and the SMP AMPI version with no overdecomposition performed best on multiple nodes. Pure outperforms all alternatives. Compared to AMPI, Pure has a speedup over the fastest AMPI version of 25% on a single node and 2× the fastest version on multiple nodes. We attribute Pure's outperformance to its efficient and fine-grain work stealing scheduler, relative to AMPI's more coarse-grain virtual rank migration strategy. Pure can steal a small or large amount of work naturally during communication latency, and efficiently adapts to both transient and persistent load imbalance.

5.3 miniAMR Benchmark

miniAMR is an open-source MPI benchmark that applies a stencil calculation on a unit cube computational domain and is a compact proxy for octree-based AMR, which is used by many larger AMR applications. miniAMR consists of 10,000 lines of code, making over 100 MPI calls throughout the application. It primarily uses nonblocking point-to-point messages and all-reduce, both with small and large payload sizes. It also uses communicators other than the default "world" one. Migrating the messaging and collective calls to Pure was mostly automatic, and took less than a day of programmer effort. Profiling revealed no significant load imbalance for our configuration, which was the default, running for 10,000 iterations and using weak scaling from 2 to 4,096 ranks with 64 ranks per node. Figure 5d shows the end-to-end results.

5.4 Intra-Node Message Microbenchmark

Figure 6 shows the Pure Speedup for intra-node point-to-point messaging for various core placements (10M iterations). We pinned the ranks to share the same hardware core, same NUMA node, and different NUMA nodes, and varied payloads from 4B to 16MB, for which Pure used a PureBufferQueue and PureEnvelopeQueue (Section 4). Pure achieves a speedup over MPI ranging from a few percent to over 17×, depending on payload size and rank placement. Pure showed the greatest speedup with a small message sent from two threads on the same core.

5.5 Collective Benchmarks

Figure 7 shows that Pure outperformed MPI and MPI DMAPP's 8B all-reduce up to 16k cores, with speedups ranging from 11% to over 3.5× (Figure 7a). The Pure barrier outperforms all other barriers, ranging up to 16k cores with speedups from 2.4× to over 5× over MPI and up to 8× speedup over OpenMP. See Appendix A for more collective results.

6 Related Work

6.0.1 MPI. Many MPI projects have strived to leverage shared memory within multicore nodes to improve performance [12, 36, 50]. XPMEM [30], enabled in our MPI baseline experiments, significantly improves performance within nodes and is used in Cray MPICH. MPI's ch4 network library has improved MPI's shared memory performance [48] and [10] explored other approaches to fast intranode communication. [35] shows how modern MPI implementations use shared memory improve performance. Much work has also been done to optimize MPI collectives [34], including the MPI DMAPP library [27]. However, DMAPP only supports a subset of MPI collectives and only for 8B payloads, unlike Pure's which are fast for all collectives and sizes. [17] optimized large scale all-to-all collectives with large task counts per node using the matrix block all-to-all algorithm.

MPI's "one-sided" message APIs [19, 22] decouple data movement with process synchronization. Ranks read and write part of other ranks' memories directly, and later synchronize. Pure, though, provides a higher level mechanism for overlapping communication and computation.

MPI has long supported multithreading within its ranks via the `MPI_THREAD_MULTIPLE` mode [22, 49], which allows any thread of a parent MPI process to make MPI calls at any time. Because MPI was designed as a process-level interface, most MPI implementations support thread-safety using a global lock, effectively serializing the threads within the MPI runtime. Despite these challenges, most HPC programmers surveyed in 2022 [29] indicated it was important to make MPI calls within multithreaded code. The MPI 4.0 Standard [23] has an enhanced focus on support for threading within MPI through a hierarchical, MPI+X approach [40]. `MPI_COMM_TYPE_SHARED` splits an MPI communicator into subcommunicators, each of which can create an MPI shared memory region. This mechanism offers extended programming model but unclear if this helps performance and programmability relative to other hybrid programming approaches.

MPI Fine-points [25] and MPI Endpoints [20] 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

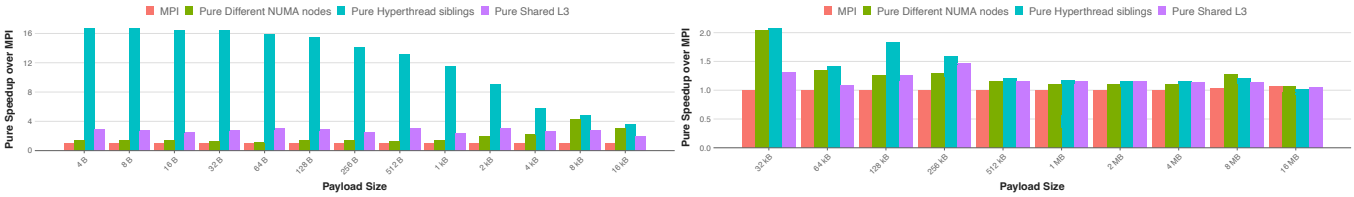
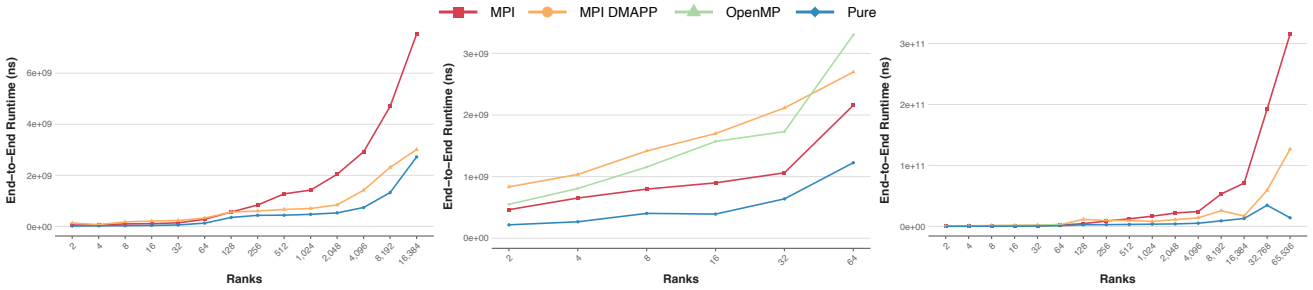


Figure 6. Pure speedup for intra-node point-to-point messaging, with payloads from 4 B–16 MB



(a) All-Reduce: 8B payload, 1–16k Ranks **(b)** Barrier, 1–64 Ranks (single node) **(c)** Barrier, 1–65k Ranks

Figure 7. Collective End-to-End Performance

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. [20] rightly pointed out the tradeoff between utilization of the network hardware interfaces (e.g., NICs) and using many threads per MPI process. To mitigate this, Pure provides a Makefile setting to adjust the number of processes per node (and, inversely, threads).

Also, while Pure uses `MPI_THREAD_MULTIPLE` mode, as any Pure rank can make MPI calls at any time, in many situations (e.g., Pure collectives) only a single leader thread is actively making an MPI call at a time and therefore there is no lock contention. Pure's application-level performance outperformed MPI running in single-threaded mode, its best performing communication mode, in spite of the performance issues with its multithreaded mode.

We sincerely hope our work inspires future versions of the MPI standard. There are, however, some constraints on MPI that may make it difficult to directly incorporate some of these ideas. MPI is fairly wedded to the idea of being multi-language (C, FORTRAN), and that makes it difficult to take advantage of modern C++. Pure's use of modern C++ is important to cleanly allow its runtime system to efficiently execute chunks of application computation. Relatedly, MPI currently focuses on communication, not execution of computation. Secondly, there is an expectation that every MPI rank operates in its own address space, typically implemented with OS processes. However, inter-process communication (even XPMEM) typically entails more overhead than two threads communicating via shared memory.

6.0.2 MPI+X. Many other projects [47, 51] have explored the MPI+X approach, usually using a shared memory programming model such as OpenMP [14], TBB [45], or CUDA [39] within nodes and MPI across nodes. These hybrid approaches require the programmer to manually orchestrate two distinct programming models. Moreover, it is both our experience and that of others in the literature that getting hybrid schemes to even match, let alone surpass, the performance of MPI-everywhere often requires significant effort [47, 51]. This is because on an n-core node, the programmer specifies how many threads k to run in OMP regions, and how many processes p, with $n = p * k$. In non-OMP sections of the application, only $p = n/k$ threads are active. Because of Amdahl's Law, the programmer must aggressively use OMP throughout the code to leverage available cores.

In contrast, Pure's ranks are non-hierarchical and one rank is allocated per core. This allows more application ranks to fit on a node, which benefits from improved intra-node communication. Programmers can run with no tasks and get a performance win; they can also annotate just a small section of the code with tasks (e.g., a load-imbalanced loop). Furthermore, Pure's runtime has access to *all* threads running on that node (not just k) and can "recruit" them to do work when they are idle. This ability to access all tasks is enhanced by the efficiency of switching between progressing intra-node communication and stealing work, which takes only a handful of assembly instructions and 1-3 cache misses in Pure's runtime. Note that for the CoMD benchmark, we used the unmodified MPI+OpenMP source code from the Mantevo project and presented results for the best-performing `OMP_NUM_THREADS` value. Further, note that in our experience, improving performance by adding optional Pure Tasks to

an existing MPI application is simpler than converting an entire application to hybrid MPI+OpenMP or a higher level programming model. In our Pure CoMD port, for example, we added 3 Pure Tasks in total, whereas the MPI+OpenMP version had 15 OMP blocks. In summary and in contrast to MPI+OpenMP, Pure lets the programmer work with a flat unified programming model, where tasks can be introduced where they are needed. With Pure there is no penalty for not using tasks, the way there is in MPI+OpenMP for not using threads in every loop.

[37] proposes a new level of MPI thread support for user-level threads to better overlap communication and computation, while adding programming complexity. [16] integrates the Habanero-C dynamic task-parallel programming model with MPI via a new data-flow programming model, and uses dedicated worker threads for the runtime system. Most of these systems use custom syntax and tasking constructs, whereas Pure uses standard C++ lambdas [1], benefiting from their standardized syntax and future improvements. [8] also supports shared memory communication with a different programming model.

6.0.3 AMPI. Charm++ [32] is a popular C++ framework offering higher-level parallel programming abstractions; migrating from MPI to it, though, is often a significant undertaking. As mentioned in Section 5.2, AMPI [33] is an MPI-compliant library that builds on top of Charm++. Pure outperformed AMPI in our experiments. We speculate that this is because of Pure's optimized messaging and collectives, as well as Pure's load balancing strategy, which occurs at a finer granularity and with less overhead than AMPI.

Like Pure, AMPI offers performance gains for MPI programs with no or 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, as shown in Section 5.2. Pure breaks up work into tiny chunks using standard C++ lambdas that are stolen with low overhead. AMPI SMP is also thread-based but, unlike Pure, requires at least one worker thread per node. As many applications *and* cores-per-node naturally come in powers of two, we found it challenging to efficiently map some applications to nodes. AMPI users must experimentally determine which version, SMP or non-SMP, performs best for their application and experimentally determine the best-performing number of virtual ranks for the application and input.

6.0.4 HPC Languages and Parallel Frameworks. PGAS languages constitute an alternative programming model where the runtime system creates the illusion of a global memory address space that is logically partitioned. PGAS models are similar to Pure in that they also employ the SPMD programming style, provide a unified programming model both within and across nodes, and aim to improve performance via locality of reference. Examples include Coarray Fortran, UPC/UPC++ [52], Coarray C++ [38], DASH [24], and SHMEM [41]. Chapel [13] and X10 [15] extend the PGAS approach with local and remote asynchronous task creation, centered on forking and joining tasks and less so on PGAS-sytle RMA operations. Communication libraries such as MPI and GASNet-EX [7] are often used for the transport layers of PGAS languages. HPX [31] is another parallel runtime system offering a global address space abstraction that extends the modern C++ standard to facilitate distributed operations. Some parallel programming systems offer implicit parallelism, where the programmer is responsible for partitioning the program into units of work and data. Legion [4] is a data-centric parallel programming system targeted at distributed heterogeneous architectures.

Frameworks, like Kokkos [21], STAPL [11] and BCL [9], provide a layer of abstraction between the application and the machine. Like Pure, these libraries often leverage modern C++ features, but using them usually requires major rewrites of existing applications.

7 Conclusion

Message passing has remained the de facto standard parallel programming model for decades because of its relative simplicity and performance. Nevertheless, as shown in this paper, message passing is not incompatible with shared memory. In fact, with suitably designed libraries, one can exploit shared memory without giving up on most of the benefits of message passing.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We are also grateful for NERSC's significant assistance with using Cori. We are grateful to David Richards for his advice and for his suggestion of using the imbalanced CoMD benchmark. Thank you also to Vlad Kiriansky and Nir Shavit for their feedback and wisdom during the course of this project.

References

[1] [n. d.]. C++ Lambda Expressions. https://en.cppreference.com/w/cpp/language/lambda

[2] [n. d.]. CoMD – ECP Proxy Applications. https://proxyapps.exascaleproject.org/app/comd/

[3] [n. d.]. pthreads(7) — Linux manual page. https://man7.org/linux/man-pages/man7/pthreads.7.html

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[5] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[6] Larry Kaplan Bob Alverson, Edwin Froese and Duncan Roweth. [n. d.]. Cray® XC Series Network. https://www.alcf.anl.gov/files/CrayXCNetwork.pdf

[7] Dan Bonachea and Paul H Hargrove. 2019. GASNet-EX: A high-performance, portable communication library for exascale. In *Languages and Compilers for Parallel Computing: 31st International Workshop, LCPC 2018, Salt Lake City, UT, USA, October 9–11, 2018, Revised Selected Papers 31*. Springer, 138–158.

[8] Mikhail Brinskiy and Mark Lubin. [n. d.]. An Introduction to MPI-3 Shared Memory Programming. https://www.intel.com/content/dam/develop/external/us/en/documents/an-introduction-to-mpi-3-597891.pdf

[9] 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.

[10] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. 2009. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. In *2009 International Conference on Parallel Processing*. 462–469. https://doi.org/10.1109/ICPP.2009.22

[11] Antal Buss, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M Amato, and Lawrence Rauchwerger. 2010. STAPL: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. 1–10.

[12] L. Chai, Q. Gao, and D.K. Panda. [n. d.]. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Seventh IEEE International Symposium on Cluster Computing and the Grid - Table of Contents* (2007).

[13] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[14] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan Kaufmann.

[15] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices* 40, 10 (2005), 519–538.

[16] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating Asynchronous Task Parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 712–725. https://doi.org/10.1109/IPDPS.2013.78

[17] George Chochia, David Solt, and Joshua Hursey. 2022. Applying on Node Aggregation Methods to MPI Alltoall Collectives: Matrix Block Aggregation Algorithm. In *Proceedings of the 29th European MPI Users' Group Meeting* (Chattanooga, TN, USA) *(EuroMPI/USA'22)*. Association for Computing Machinery, New York, NY, USA, 11–17. https://doi.org/10.1145/3555819.3555821

[18] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. 2008. Cray performance analysis tools. In *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*. Springer, 191–199.

[19] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2016. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience* 28, 17 (2016), 4385–4404.

[20] 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.

[21] 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.

[22] 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

[23] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0*. Technical Report. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[24] Karl Fürlinger, Colin Glass, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. 2014. DASH: Data Structures and Algorithms with Support for Hierarchical Locality. In *Euro-Par 2014 Workshops (Porto, Portugal)*. https://doi.org/10.1007/978-3-319-14313-2_46

[25] Ryan E Grant, Matthew GF Dosanjh, Michael J Levenhagen, Ron Brightwell, and Anthony Skjellum. 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.

[26] William Gropp. [n. d.]. Lecture 24: Buffering and Message Protocols. https://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture24.pdf

[27] Yun He, Brandon Cook, Jack Deslippe, Brian Friesen, Richard Gerber, Rebecca Hartman-Baker, Alice Koniges, Thorsten Kurth, Stephen Leak, Woo-Sun Yang, et al. 2018. Preparing NERSC users for Cori, a Cray XC40 system with Intel many integrated cores. *Concurrency and Computation: Practice and Experience* 30, 1 (2018), e4291.

[28] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 355–364.

[29] Michael A Heroux, Lois McInnes, Xiaoye Sherry Li, James Ahrens, Todd Munson, Kathryn Mohror, Terece Turton, Jeffrey Vetter, and Rajeev Thakur. 2022. *ECP software technology capability assessment report*. Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[30] N. Hjelm. [n. d.]. Linux Cross-Memory Attach. https://github.com/hjelmn/xpmem

[31] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (Eugene, OR, USA) *(PGAS '14)*. Association for Computing Machinery, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2676870.2676883

[32] 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.

[33] Laxmikant V Kale and Gengbin Zheng. 2009. Charm++ and AMPI: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications* (2009), 265–282.

[34] K Kandalla, David Knaak, K McMahon, N Radcliffe, and M Pagel. 2015. Optimizing Cray MPI and Cray SHMEM for Current and Next Generation Cray-XC Supercomputers. *Cray User Group (CUG)* 2015 (2015).

[35] Krishna Chaitanya Kandalla, Peter Mendygral, Nick Radcliffe, Bob Cernohous, David Knaak, Kim H. McMahon, and M. Pagel Cray. 2016. Optimizing Cray MPI and SHMEM Software Stacks for Cray-XC Supercomputers based on Intel KNL Processors.

[36] A. Kayi, T. El-Ghazawi, and G.B. Newby. [n. d.]. Performance issues in emerging homogeneous multi-core architectures. *Simulation Modelling Practice and Theory* 17, 9 ([n. d.]), 1485 – 1499,.

[37] Huiwei Lu, Sangmin Seo, and Pavan Balaji. 2015. MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 444–454. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.82

[38] Felix Mößbauer, Roger Kowalewski, Tobias Fuchs, and Karl Fürlinger. 2018. A Portable Multidimensional Coarray for C++. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 18–25. https://doi.org/10.1109/PDP2018.2018.00012

[39] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[40] Message Passing Interface Forum Working Group on Hybrid Issues. 2023. MPI 4.0 Working Group on Hybrid Issues. https://github.com/mpiwg-hybrid/hybrid-issues/wiki

[41] Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A Kendall. 2000. *A generalized portable SHMEM library for high performance computing*. Technical Report. Ames Lab., Ames, IA (US).

[42] 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

[43] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing* (Las Palmas de Gran Canaria, Spain) *(Euro-Par 2008)*. Springer-Verlag, Berlin, Heidelberg, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9

[44] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[45] Chuck Pheatt. 2008. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.* 23, 4 (apr 2008), 298.

[46] Constantine D. Polychronopoulos and David J. Kuck. 1987. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.* C-36, 12 (1987), 1425–1439. https://doi.org/10.1109/TC.1987.5009495

[47] 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.

[48] Ken Raffenetti, Abdelhalim Amer, Lena Oden, Charles Archer, Wesley Bland, Hajime Fujita, Yanfei Guo, Tomislav Janjusic, Dmitry Durnov, Michael Blocksome, et al. 2017. Why is MPI so slow? analyzing the fundamental limits in implementing MPI-3.1. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[49] Rajeev Thakur and William Gropp. 2007. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_-MULTIPLE. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting, Paris, France, September 30-October 3, 2007. Proceedings 14*. Springer, 46–55.

[50] Hui Zhou Yanfei Guo, Ken Raffenetti. [n. d.]. MPICH for Exascale: Supporting MPI-4 and ECP. https://www.exascaleproject.org/wp-content/uploads/2021/01/2021-ECP-BOF-days-master-deck.pdf

[51] E Yilmaz, RU Payli, HU Akay, and A Ecer. 2009. Hybrid parallelism for cfd simulations: Combining mpi with openmp. *Lecture Notes in Computational Science and Engineering, Springer Verlag, Antalya, Turkey* (2009), 401–408.

[52] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1105–1114. https://doi.org/10.1109/IPDPS.2014.115