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

James Psota, Armando Solar-Lezama
MIT CSAIL

PPoPP’ 24

Frustrated with MPI+Threads? Try MPIxThreads!

Hui Zhou, Ken Raffenetti, Junchao Zhang, Yanfei Guo, Rajeev Thakur
Argonne National Laboratory

EUROMPI’23

梁恒中
2024.11.6

Background

MPI

- Multi-process parallelism
- Need to launch multiple ranks on multi-core processors
- Combination with other models, MPI+X: MPI+CUDA, MPI+OpenMP...

MPI+OpenMP/multithreading

- Better leverage shared memory on a node than MPI-only solution
- MPI lacks the ability to manage threads
- New challenges: performance, algorithms

Solution:

Extend MPI so that it is thread-aware.

Frustrated with MPI+Threads? Try MPIxThreads

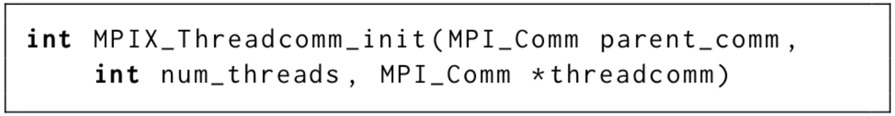
Proposal: MPIX thread communicator

An explicit, hierachical solution

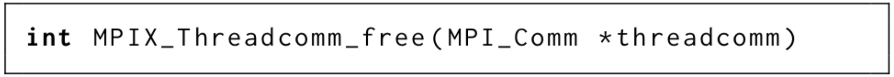
- Introduces new API initializing thread contexts
- Introduces parallel regions which are managed by MPI runtime and where threads run.
- Introduces thread comm to manage threads

Inside a parallel region, every thread is identified by a rank, and calls to MPI are using thread comm.

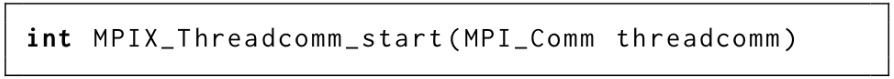
MPIxThread:
N MPI ranks, each MPI rank launches M thread, thus NxM threads.



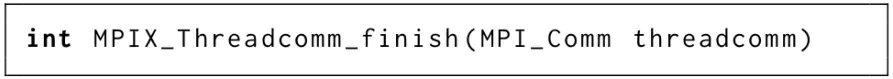
```
int MPIX_Threadcomm_init(MPI_Comm parent_comm,
    int num_threads, MPI_Comm *threadcomm)
```



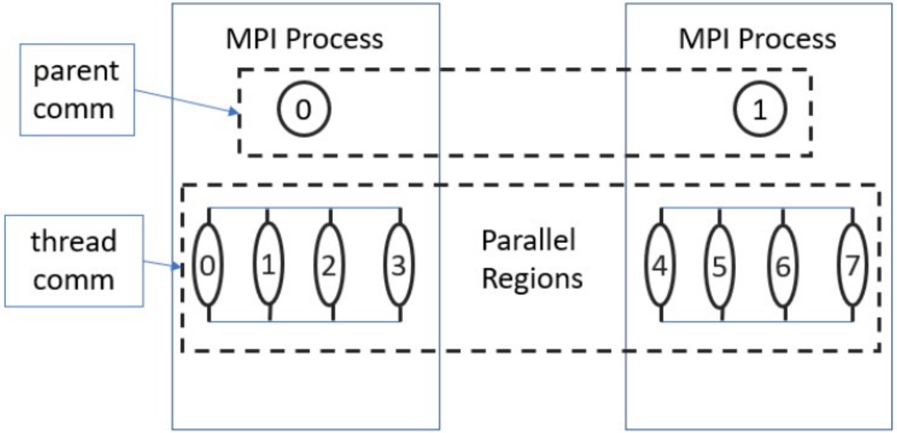
```
int MPIX_Threadcomm_free(MPI_Comm *threadcomm)
```



```
int MPIX_Threadcomm_start(MPI_Comm threadcomm)
```



```
int MPIX_Threadcomm_finish(MPI_Comm threadcomm)
```



Proposal: MPIX thread communicator

The initialization of thread context is heavy-weight, thus thread comm is designed to be reusable across multiple parallel regions.

Common resources for all ranks in the communicator

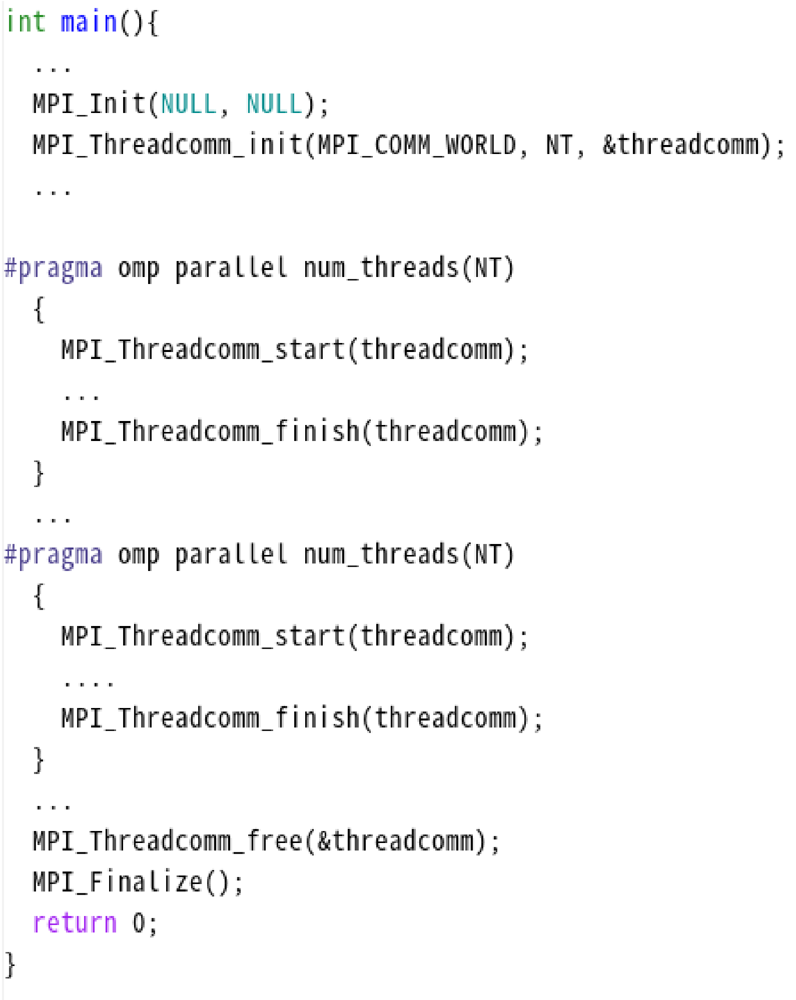
- Rank table, etc;
- Initialized on MPIX_Threadcomm_init().

Shared-memory communication devices

- Mailbox, etc;
- Initialized on MPIX_Threadcomm_init().

Rank-specific resources

- Thread-local data, etc;
- Initialized on MPIX_Threadcomm_start().



```
int main(){
  ...
  MPI_Init(NULL, NULL);
  MPI_Threadcomm_init(MPI_COMM_WORLD, NT, &threadcomm);
  ...

#pragma omp parallel num_threads(NT)
  {
    MPI_Threadcomm_start(threadcomm);
    ...
    MPI_Threadcomm_finish(threadcomm);
  }
  ...
#pragma omp parallel num_threads(NT)
  {
    MPI_Threadcomm_start(threadcomm);
    ....
    MPI_Threadcomm_finish(threadcomm);
  }
  ...
  MPI_Threadcomm_free(&threadcomm);
  MPI_Finalize();
  return 0;
}
```

Communication Implementation

Most collective algorithms consist of internal point-to-point communications.

Interprocess thread-to-thread communications are handled as regular MPI point-to-point communications. Internal tags are modified to identify source and destination ranks.

Shared-memory communication

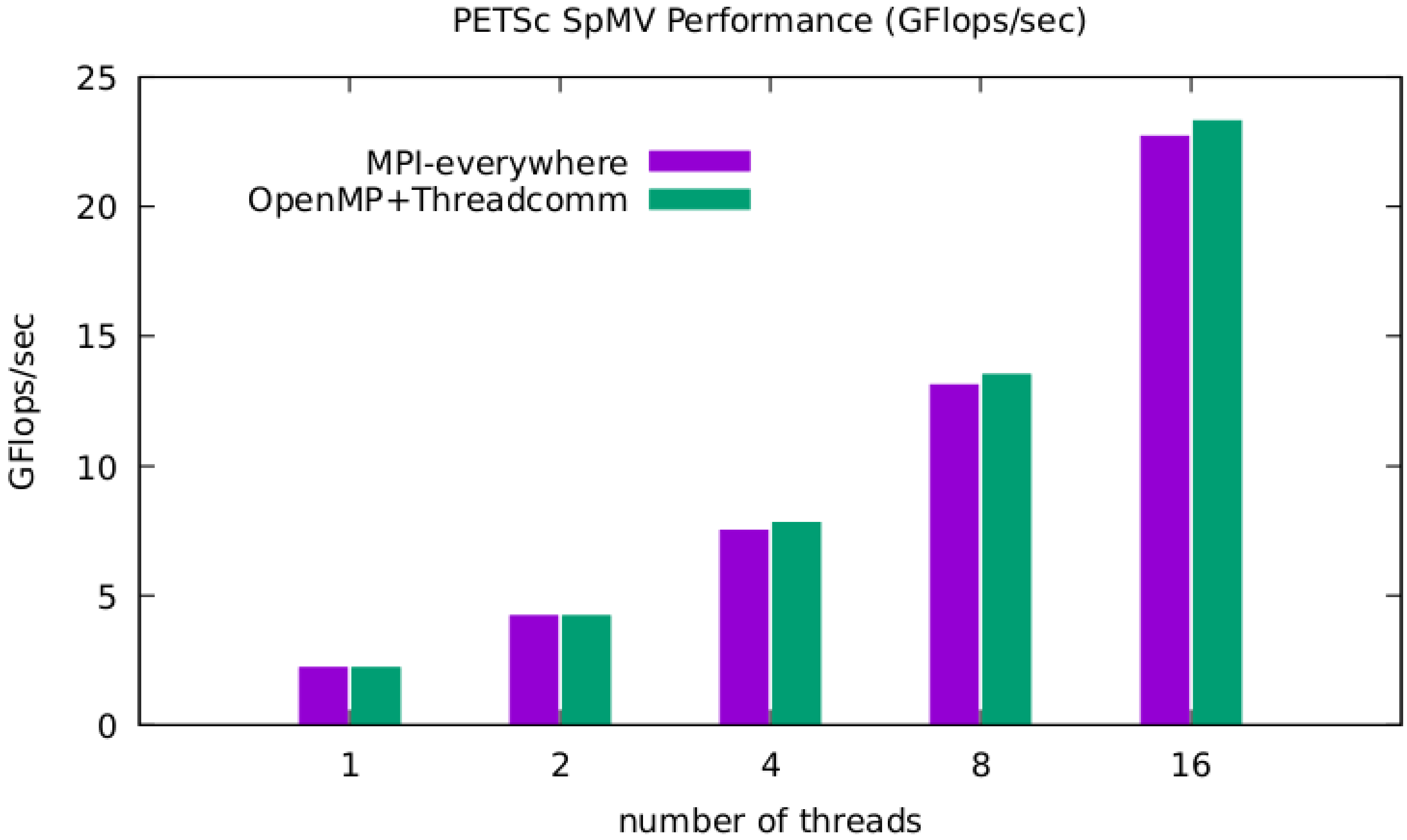
- lockless multiple-producer-single-consumer queue
- shared memory pools divided into cells and shared by all threads in a process

For small messages, messages are directly copied into cells and sent to receivers.
For large messages, headers are put into cells and an acknowledgement is required to start copy.

Evaluation

CPU: Xeon 5317 x2, 24 cores totally

With 4 threads per process, maybe.



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

Proposal

An implicit, flat solution

- Ranks are no longer processes, but threads instead.
- Based on MPI though, no direct MPI calls, and is process-unaware.
- Global variables are not allowed.

Collective communications implemented:

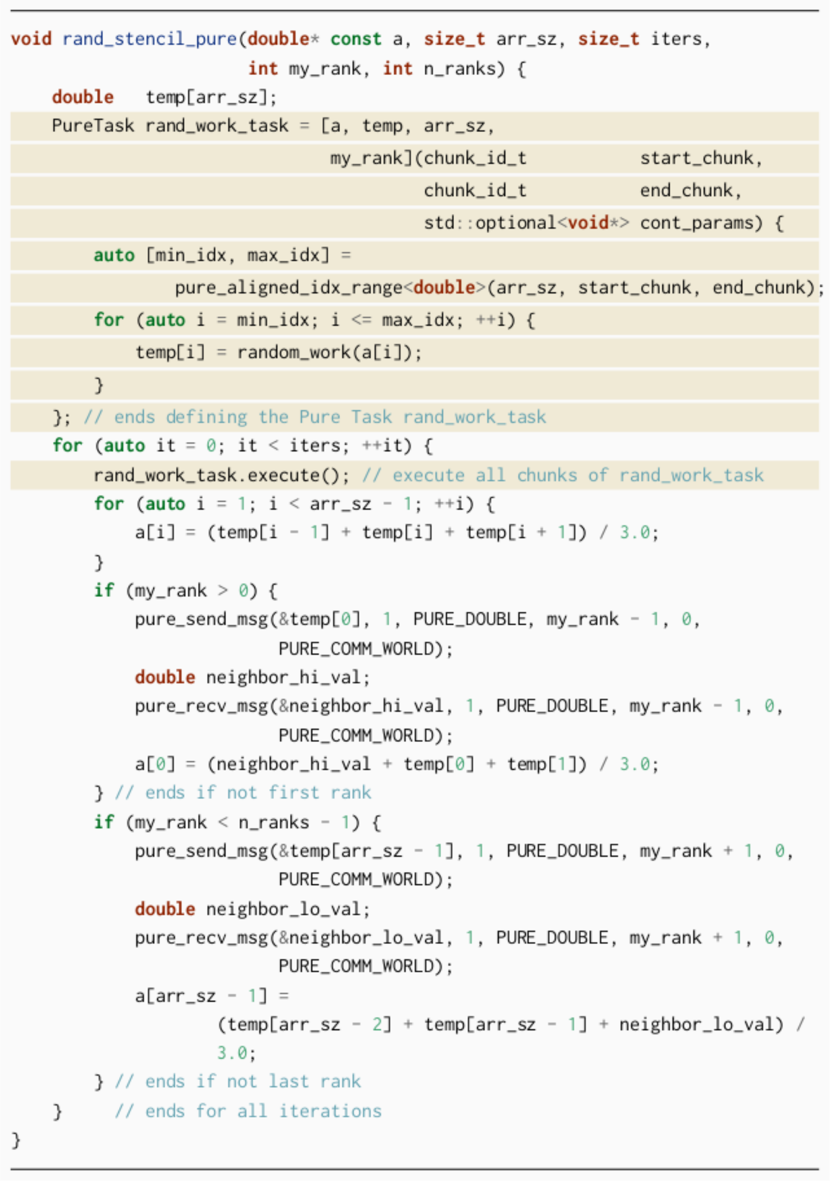
- Reduce
- Allreduce
- Barrier
- Broadcast

Task-based Load Balance

Pure tasks are C++ lambda that could execute in parallel.

Computation is segmented into chunks. Programmers should make sure that portions of the work do not rely on each other.

On execution, chunk arguments of a task is given by the runtime in an arbitrary order.



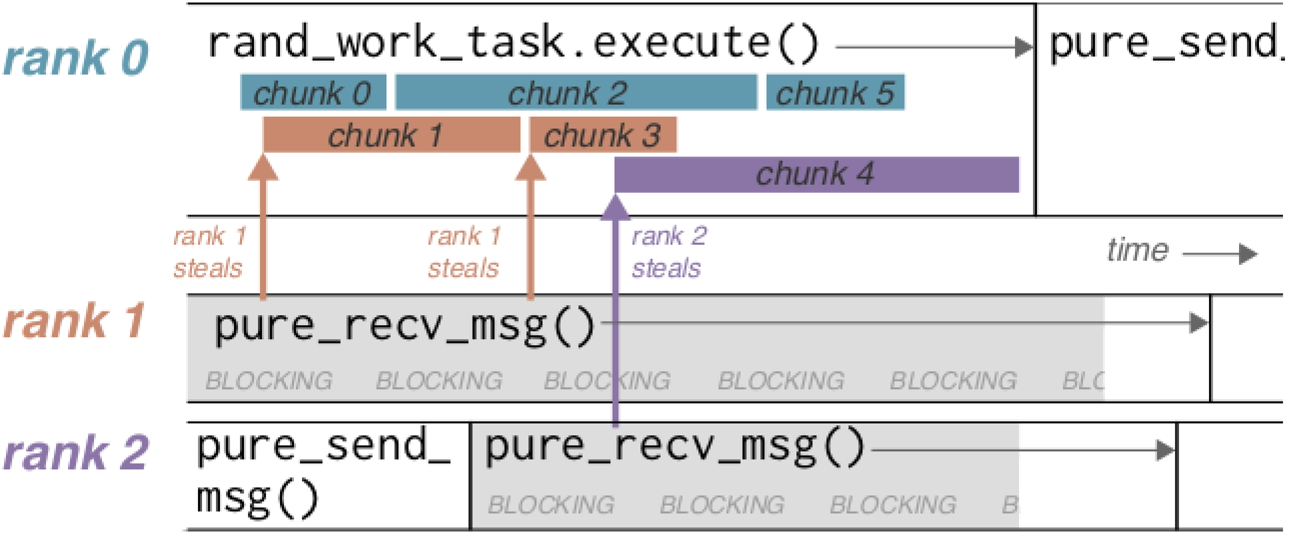
```
void rand_stencil_pure(double* const a, size_t arr_sz, size_t iters,
                       int my_rank, int n_ranks) {
    double   temp[arr_sz];
    PureTask rand_work_task = [a, temp, arr_sz,
                               my_rank](chunk_id_t           start_chunk,
                                        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
}
```

Spin-steal Waiting Loop

On a waiting, Pure ranks(help threads) try to steal work from busy ranks(leader threads).

The runtime maintains an active_tasks array to record remaining tasks for each rank on a node. When the array contains a non-null pointer, associated rank is busy.

Two integers, curr_chunk and chunks_done, indicate task status. Helper threads atomically modify their values when stealing tasks.



Communication Implementation

point-to-point communication

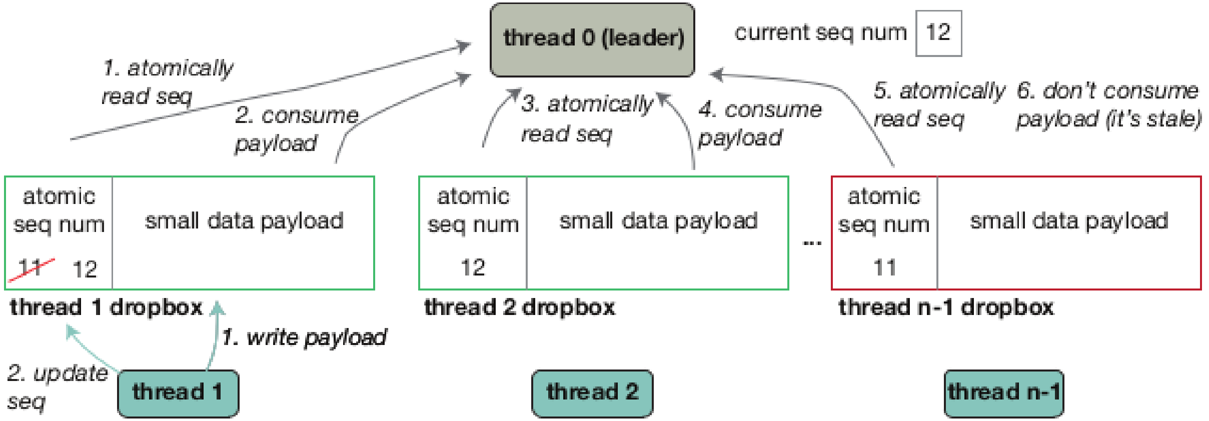
- Use lockless circular queue/ring.
- For intra-node small messages, the sender copies the message to a slot of the ring buffer, and the receiver copies the message out when available.
- For intra-node large messages, the receiver puts meta data to the ring buffer, and the sender copies data to destination directly.
- For inter-node messages, they are handled using MPI_Send and MPI_Recv implicitly. Pure ranks are translated to MPI ranks, and thread ids are encoded in MPI tags.

Communication Implementation

collective communication

For small message:

- Non-leader threads copy their data to leader thread buffer.
- Leader thread performs reduce locally.
- Leader threads on different nodes invoke MPI_Allreduce.
- Leader thread broadcasts result to non-leader threads.

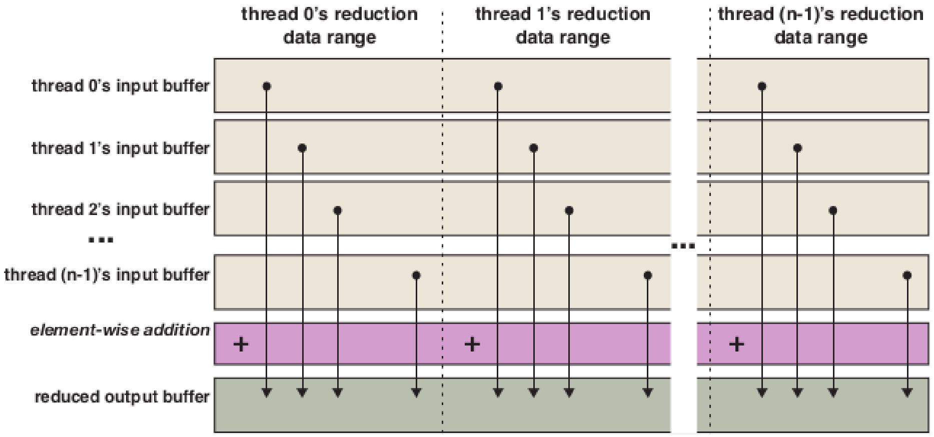


Communication Implementation

collective communication

For large messages:

- Threads share buffer pointers with each other.
- Buffers are equally segmented into chunks, and threads reduce on chunks according to their ranks.
- Leader threads perform MPI_Allreduce.



thread 0's reduction data range
thread 1's reduction data range
thread (n-1)'s reduction data range
thread 0's input buffer
thread 1's input buffer
thread 2's input buffer
...
thread (n-1)'s input buffer
element-wise addition
+
reduced output buffer

Evaluation

Performed on NERSC Cori supercomputer, each node with two E5 2698v3, 32 hardware threads totally. Up to 1024 nodes.

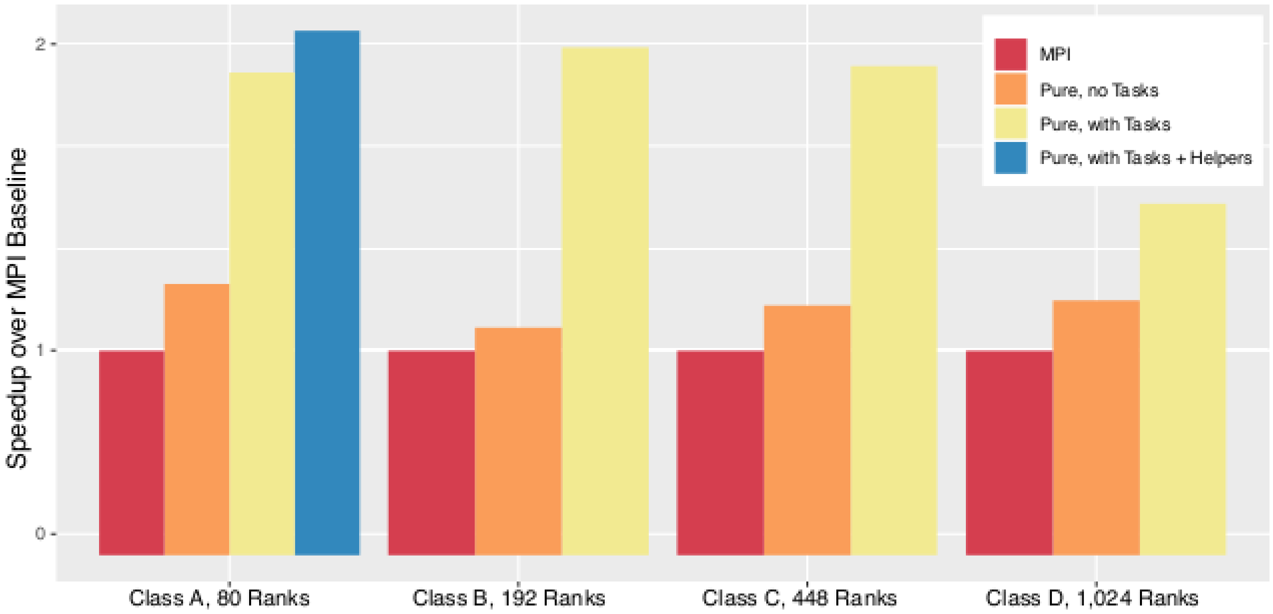
Baseline: Cray MPICH MPI

Evaluation: NAS DT

NAS DT: a data traffic MPI benchmark.

For size D, 16 ranks per node; for size A, 40 ranks per node; for size B and C, 64 ranks per node.

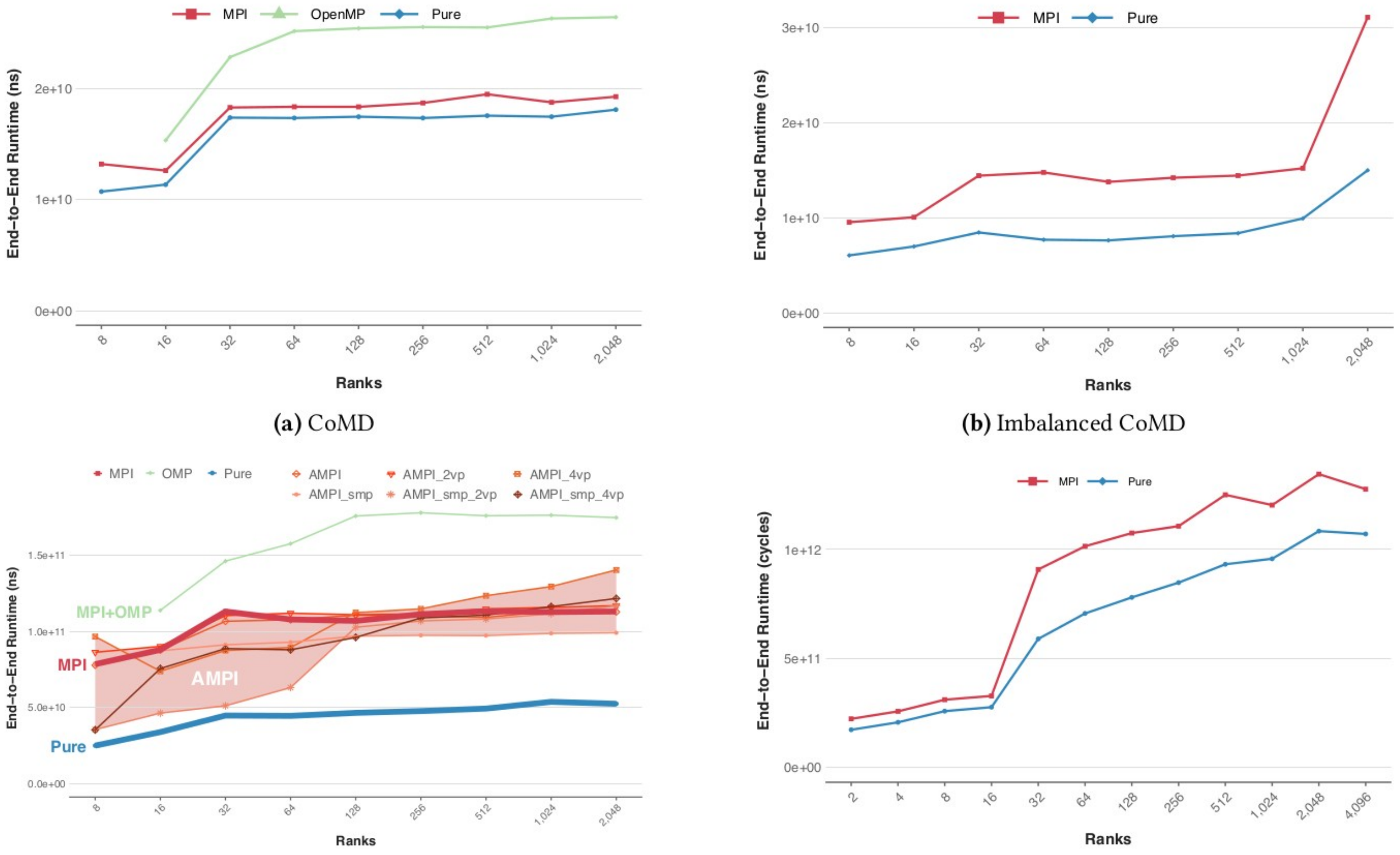
Blue bar: unused cores as helpers



Evaluation: Applications

OpenMP: 4 threads per rank.

AMPI: a MPI compliant runtime with load balance.



(a) CoMD

(b) Imbalanced CoMD

(c) Dynamic imbalanced CoMD

(d) miniAMR

Evaluation: Microbench

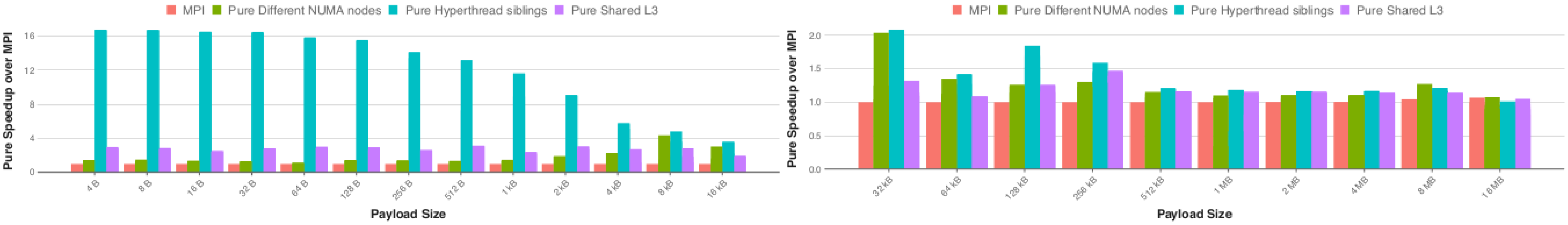
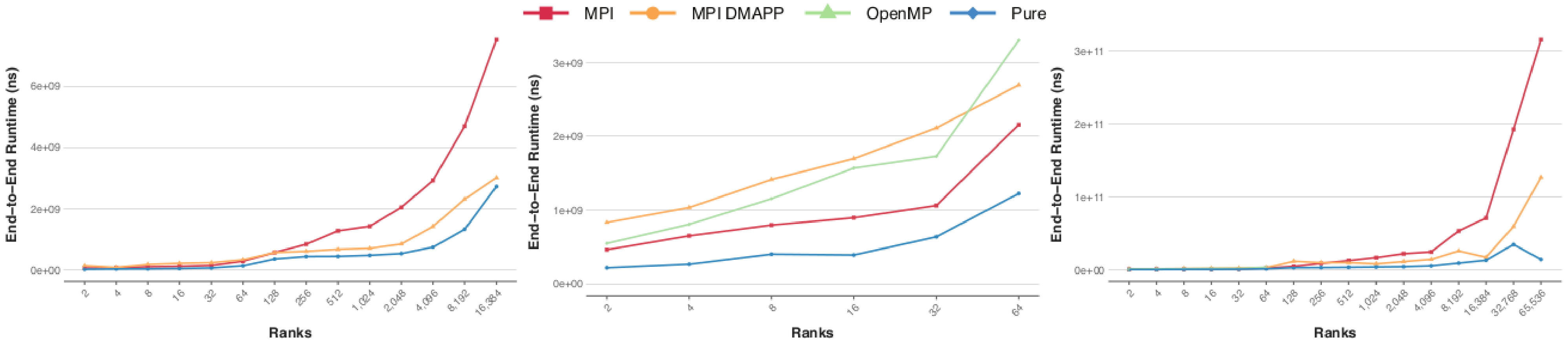


Figure 6. Pure speedup for intra-node point-to-point messaging, with payloads from 4 B–16 MB



(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

Summary

MPI+thread

- Explicit and hierarchical
- Implicit and flat

More details on rank management?

Extensible to other model?