This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2022-6026C

How To Leverage New MPI Features for Exascale Applications

EXASCALE COMPUTING PROJECT

Matthew G. F. Dosanjh[1]
W. Pepper Marts[1]
Jan Ciesko[1]
Howard Pritchard[2]

[1]Sandia National Laboratory
[2]Los Alamos National Lab

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

U.S. DEPARTMENT OF ENERGY | Office of Science

Tutorial Outline

- Partitioned Communication – Matthew Dosanjh
- Insights on Adapting Codes to Fine Grained Communication – Pepper Marts
- User Level Threading in Open MPI – Jan Ciesko
- MPI Sessions Overview – Howard Prichard

Partitioned Communication

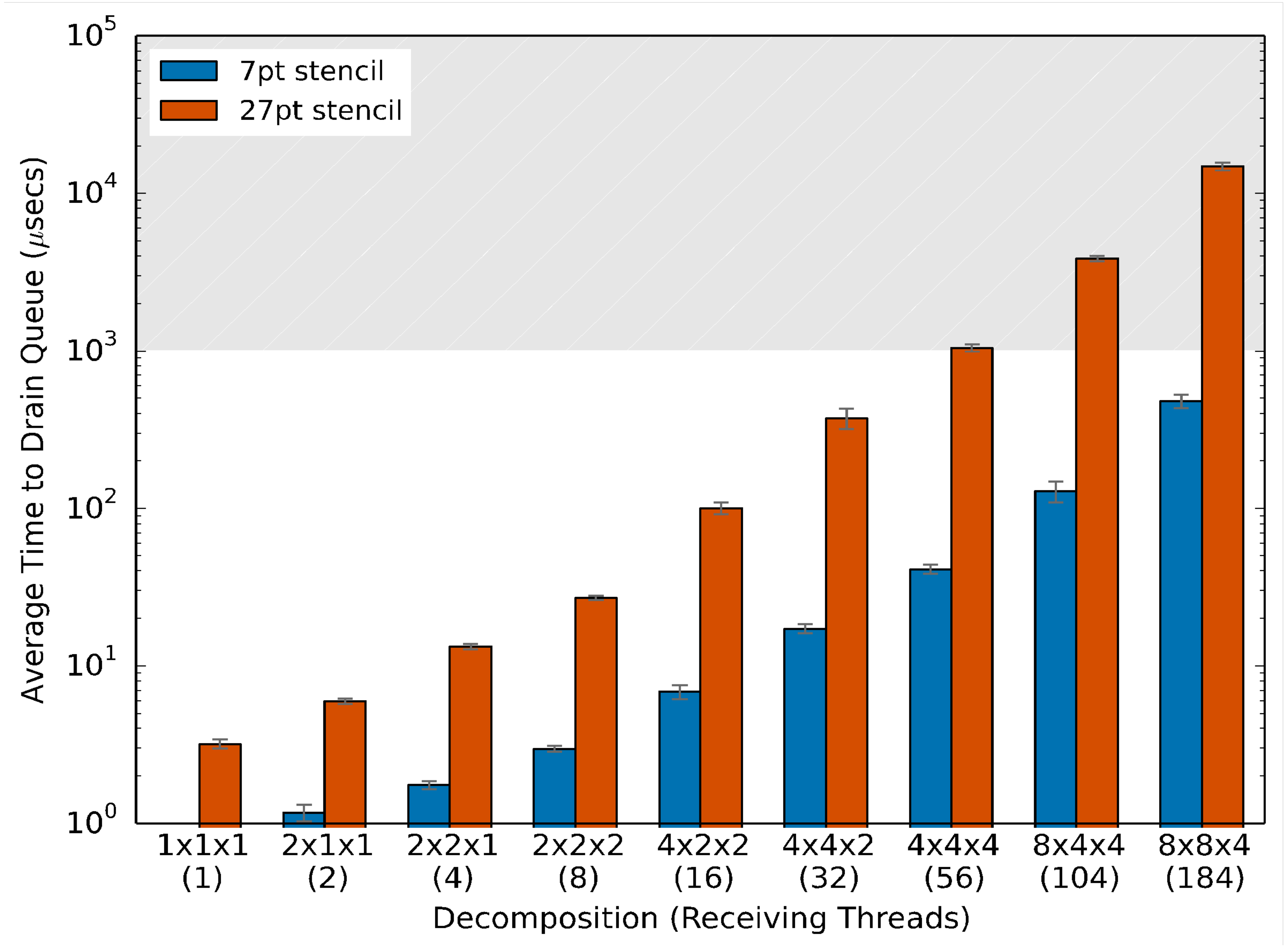
Matthew Dosanjh
Sandia National Laboratory

Special Thanks:
Ryan Grant, Queens University

Why Partitioned Communication?

- MPI use cases continue to evolve
  - CPU design continues to increase the number of cores and hardware threads
  - Threading and Tasking models are increasing the number of threads per process
  - Accelerator induced communication is the next frontier
- Potentially thousands of MPI processes on a single node (without threading)
- Can we use MPI for concurrent communication?

The Coming Thread Storm

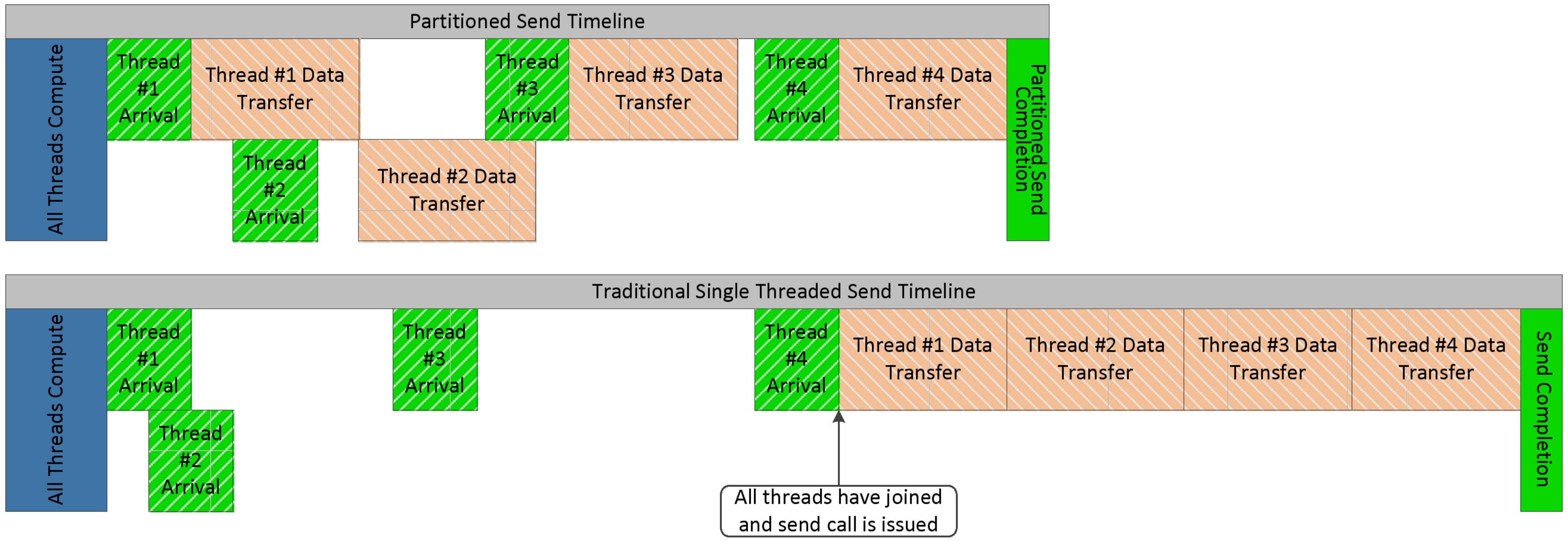


MPI Partitioned Communication Concepts

- Many actors (threads) contributing to a larger operation in MPI
  - Same number of messages
  - No new ranks
- Many threads work together to assemble a message
  - MPI only manages completion notification
  - These are actor/action counts, not thread level collectives
- Persistent-type communication
  - Init…(Start…test/wait)…free
- No heavy MPI thread concurrency handling required
  - Leave the placement/management of the data to the user
  - Knowledge required: number of workers, which is easily available
- No more complicated packing of data, send structures when they become available

New Type of Overlap

- “Early bird communication”
- Early threads can start moving data right away
- Could implement using RDMA to avoid message matching



Persistent Partitioned Buffers

- Expose the “ownership” of a buffer as a shared to MPI
- Need to describe the operation to be performed before contributing segments
- MPI implementation doesn’t have to care about sharing
  - Only needs to understand how many times it will be called
- Threads are required to manage their own buffer ownership such that the buffer is valid
  - The same as would be done today for code that has many threads working on a dataset (that’s not a reduction)
- Result: MPI is thread agnostic with a minimal synchronization overhead (atomic_inc)
  - Can alternatively use task model instead of threads, IOVEC instead of contiguous buffer

Example for Persistence

- Like persistent communications, setup the operation

```
int MPI_Partitioned_send_init( void *buf, int count, MPI_Datatype data_type,
                               int to_rank, int to_tag, int num_partitions,  MPI_Info info, MPI_Comm comm,
                               MPI_Request *request);
```

- Start the request

```
MPI_ Start(request)
```

- Add items to the buffer

```
#omp parallel for …
int MPI_Pready( void* buf, int count, MPI_Datatype in_datatype,
                int offset_index, MPI_Request *request);
```

- Wait on completion

```
MPI_Wait(request)
```

- Optional: Use MPI_Parrived to test individual partition completion

Using Part Comm in an MPI Program

- Things to keep in mind:
  - Parallelism data structures can lead to good partitioning
  - Balance communication granularity with overlap
    - Not too small of message but not too large either
    - Current networks suggest 64KiB to 1MiB are good targets for 32-64 partitions.
    - Partition sizes 1KiB to 64KiB
  - Application development – a high quality implementation

What is MPI doing to help?

- When designing your application keep in mind what MPI may be doing to help

- Think about overlap: send early and often
  - Take advantage of “noise” in computation for extra overlap

- Consider overheads: Don’t send very small partitions (single digit bytes) back to back, function call overhead will slow things down

EXASCALE COMPUTING PROJECT

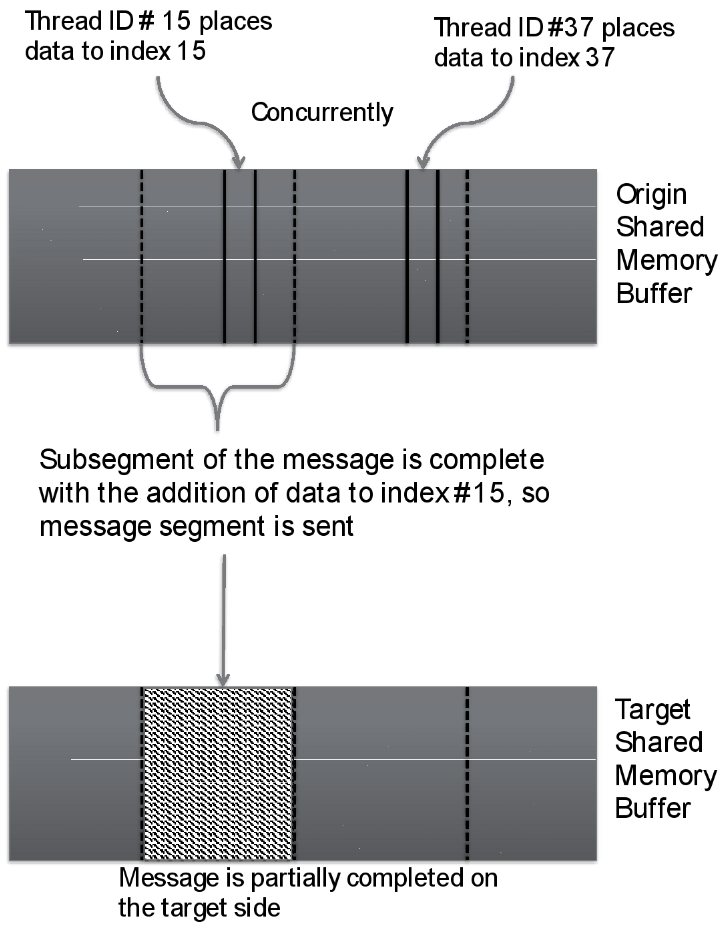
Opportunities for Optimization

MPI implementations can optimize data transfer under the covers:

- Subdivide larger buffers and send data when ready
- Could be optimized to specific networks (MTU size)
- Number of messages will be:

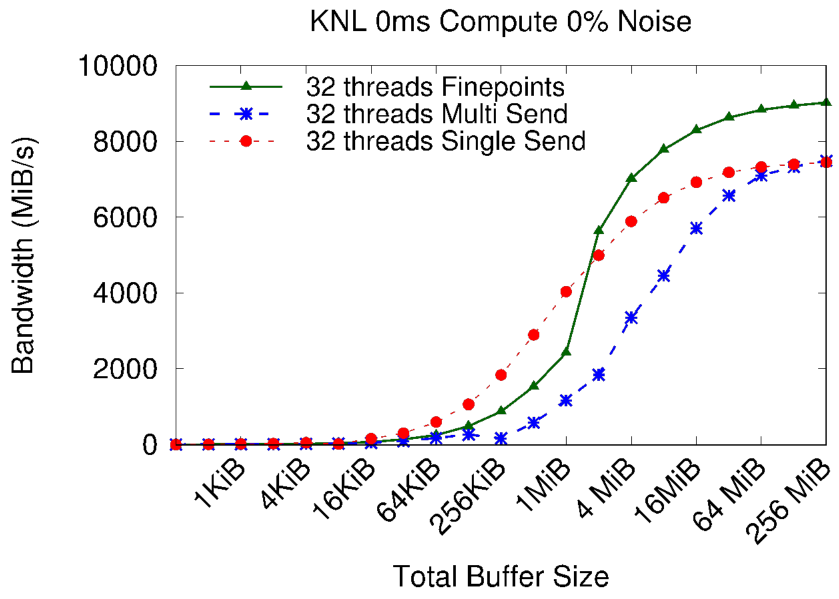
  1 < #messages ≤ #threads/tasks

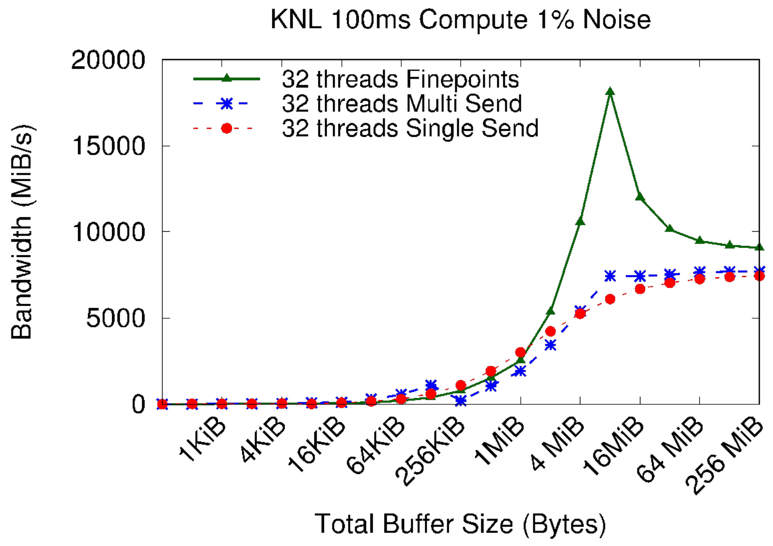
  For a partition with 1 part per thread
- Reduces the total number of messages sent, decreasing matching overheads

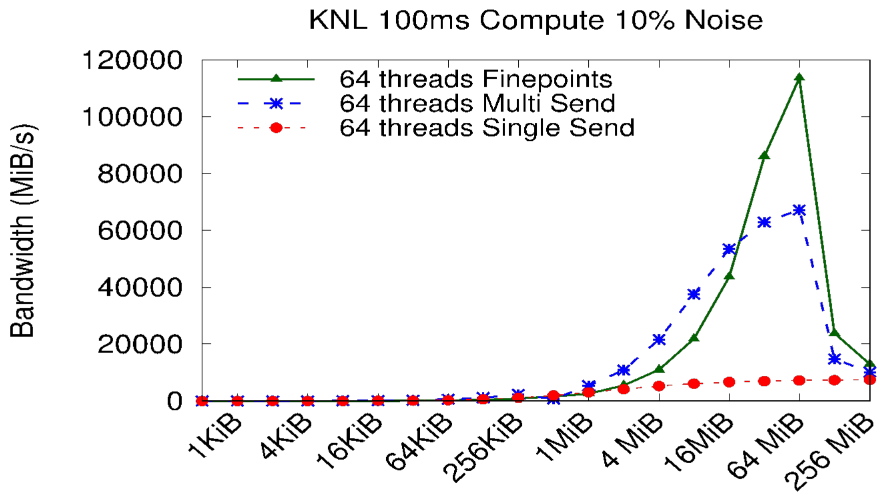

Thread ID # 15 places data to index 15
Concurrently
Thread ID #37 places data to index 37
Origin Shared Memory Buffer
Subsegment of the message is complete with the addition of data to index #15, so message segment is sent
Target Shared Memory Buffer
Message is partially completed on the target side

Performance Results

- External Implementation, leveraging the RMA interface.







Example Program

```
double runBenchmark(int rank, int numIterations, char* sendBuf, char* recvBuf,
                                        int numThreads, size_t threadPart, double compTime, double noise ) {
MPI_Request myReq;
double start, extra_time = 0;
int other = (rank + 1) % 2, TAG = 0x1234, rc = 0;
if ( rank == 0 ) {
    rc = MPI_Psend_init(sendBuf, numThreads, threadPart, MPI_CHAR, other, TAG,
                                         MPI_COMM_WORLD, &myReq);
} else {
    rc = MPI_Precv_init(recvBuf, numThreads,  threadPart,  MPI_CHAR, other, TAG,
                                       MPI_COMM_WORLD, &myReq);
}
start = MPI_Wtime();
    srand(time(NULL));
    long sleep = compTime * 1000000000;
    long sleepPlus = (compTime + ( compTime * noise)) * 1000000000 ;
```

```
#pragma omp parallel
shared(rank,numIterations,sendBuf,recvBuf,threadPart,myReq,sleep,sleepPlus)
num_threads(numThreads)

{

    int tid = omp_get_thread_num(), iteration = 0;

    struct timespec req,rem; req.tv_sec = 0;

    if ( numThreads > 1 && tid == numThreads - 1 ) req.tv_nsec = sleepPlus;

    else req.tv_nsec = sleep;

    for ( iteration = 0; iteration < numIterations; iteration++ ) {

#pragma omp master

        {

            rc = MPI_Start(&myReq);

            MPI_Barrier( MPI_COMM_WORLD );

        }

#pragma omp barrier
```

```
  double duration = MPI_Wtime() - start;;
  MPI_Barrier(MPI_COMM_WORLD);
  if ( numThreads > 1 ) duration -=  sleepPlus / 1000000000.0 * numIterations;
  else duration -=  sleep / 1000000000.0 * numIterations;
  if( 1 == rank )
          printf("RECV_STATS %d %ld %f %f\n", numThreads, threadPart * numThreads,
                   duration, extra_time);
  MPI_Request_free(&myReq);
  return duration;
}
```

Takeaways

- Partitioned communication useful for concurrent programming models
- Send data close to computation, ***not a Bulk Synchronous model***
- Accelerators/GPUs support targeted for a future MPI release

Usage model - Kernel communication triggering

Host (CPU) side

```
MPI_Psend_init(..., &request);
for (...) {
  MPI_Start(&request);
  MPI_Pbuf_prepare
  kernel<<<...>>>(..., request);
  MPI_Wait(&request);
}
MPI_Request_free(&request);
```

Kernel:

```
__device__ kernel(..., MPI_Request request)
{
  int i = my_partition[my_id];
  /* Compute and fill partition i then mark
ready: */
  MPI_Pready(i, request);
}
```

Note: CPU does communication setup and completion steps for MPI. Setup commands on NIC and poll for completion of entire operation. Kernel just indicates when NIC/MPI can send data. Ideally want to trigger communication from GPU to fire off when data is ready without communication setup/completion in kernel

Pbuf_prepare Example

MPI_PSEND_INIT

MPI_START

MPI_PBUF_PREPARE (blocking/non-local)

MPI_…(nonblocking)

MPI_WAIT (completing)

MPI_START, MPI_PSYNC

MPI_PREADY...MPI_PREADY

MPI_WAIT

MPI_PRECV_INIT

MPI_START

MPI_PBUF_PREPARE (blocking/non-local)

Optional – MPI_PARRIVED (nonblocking)

MPI_WAIT (completing)

MPI_START, MPI_PSYNC

MPI_PARRIVED...MPI_PARRIVED

MPI_WAIT

Fine Grained Communication: Insights for Applications

W. Pepper Marts
Sandia National Laboratory

Introduction

How do we implement fine grained communication in new and existing applications?

Let’s explore my experience working with miniFE

What Is Needed

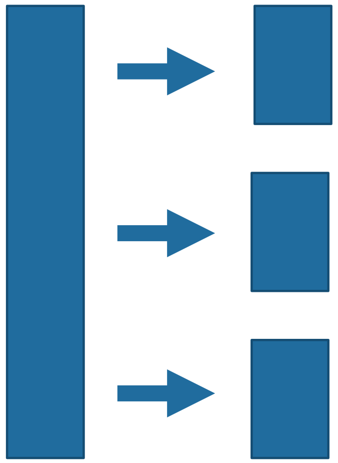
There are three major tasks to complete:

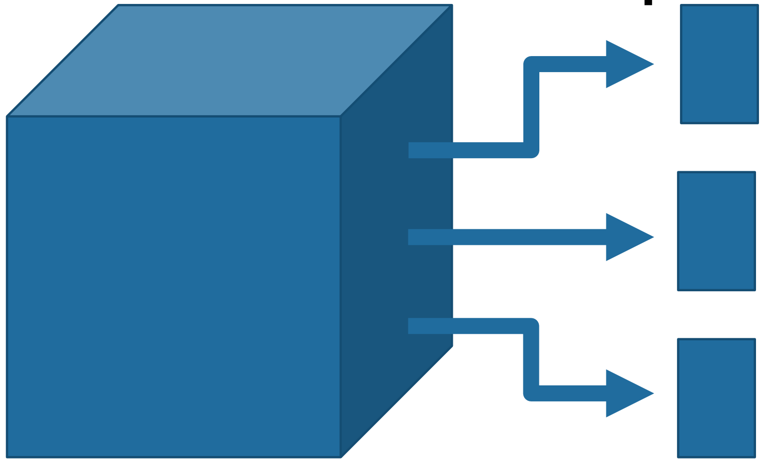
- Mapping data elements to communicated partitions
- Packing and marking ready from a threaded context
- Tuning the partitioning for best performance

Mapping Work to Partition

Partitioning ease depends on workload:

- Easy if there is strong coupling of thread to partition
- Not possible for the halo communication in miniFE
- We had to separate send buffers into partitions
- Maintain a map from computed element to partition





Mapping Work to Partition

Example Data-Structures:

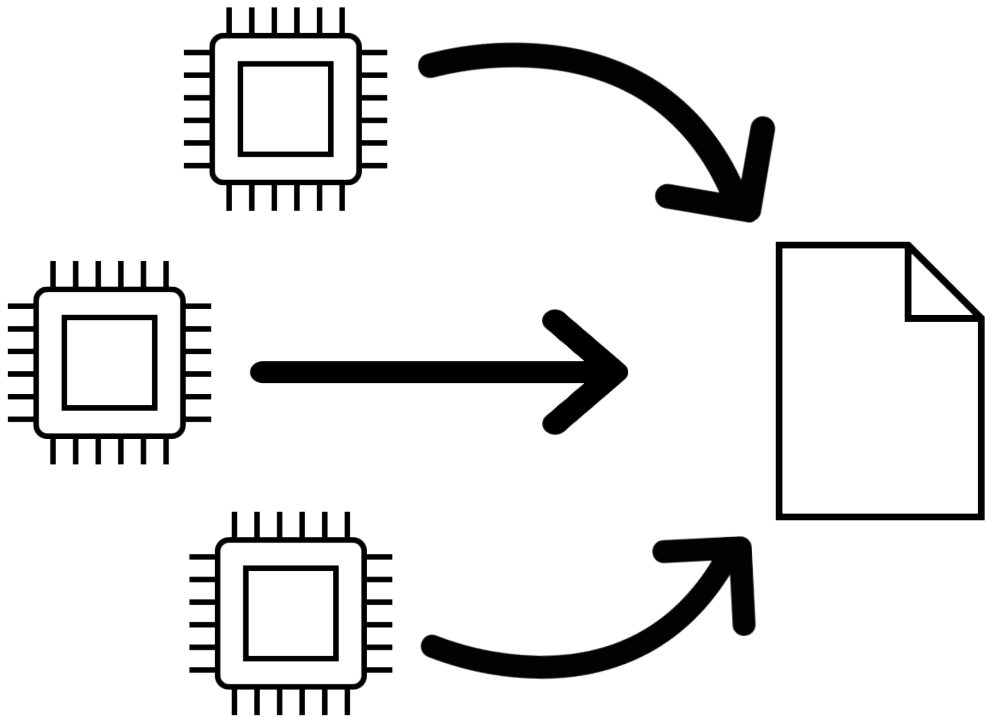
- Is an element sent?
- To whom is it sent?
- Where is it packed?
- In what partition?
- How full is it?

```
A[x][y][z] = B[x][y] * C[z];//work complete
//allow most iterations to skip entirely
If(data.isSent(x,y,z){
  sr = data.sendRank(x,y,z);
  loc = data.sendLoc(x,y,z);
  buffer[sr][loc] = A[x][y][z];//pack early
  part = loc / PART_SIZE;//find partition
  data.count[sr][part]++;//count elements

  //only send if completed
  if(data.count[sr][part] == PART_SIZE){
    MPI_Pready(request[sr],part);
  }
}
```

Packing and Readying

There is a performance impact to thread safe concurrent access to any data structure or hardware resource:



Packing and Readying

Packing and Readying:

- Pack at completion, potentially element by element
- Threaded apps require thread-safe structures
- Quick access to data structures, low cache impact, and few atomic operations are key.

Reference miniFE Example

```
template<typename VectorType>
void waxpby([...])
{
  typedef typename VectorType::ScalarType ScalarType;

       int n = x.coefs.size();
  const ScalarType* xcoefs = &x.coefs[0];
  const ScalarType* ycoefs = &y.coefs[0];
        ScalarType* wcoefs = &w.coefs[0];

  for(int i=0; i<n; ++i) {
    wcoefs[i] = alpha*xcoefs[i] + beta*ycoefs[i];
  }
}
```

Modified miniFE Example

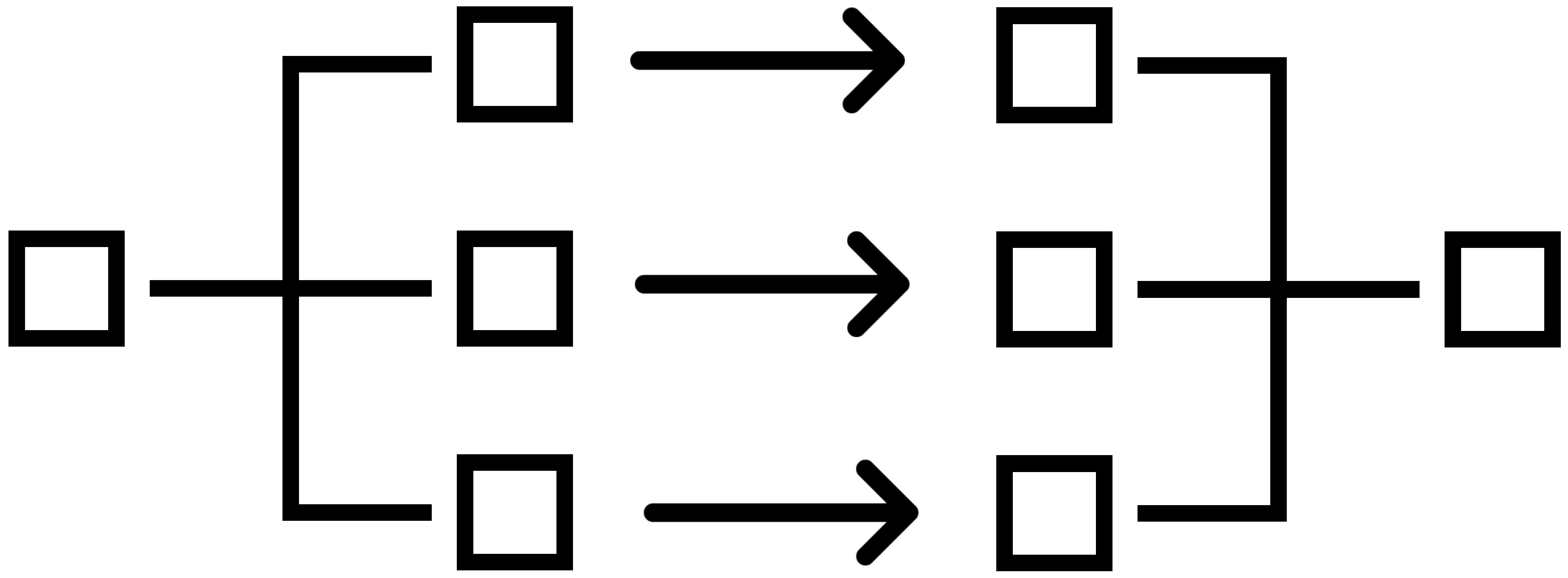
```
template<typename VectorType, typename MatrixType>
void waxpby_send([...])
{
  typedef typename VectorType::ScalarType ScalarType;
  typedef typename MatrixType::LocalOrdinalType LocalOrdinal;
  typedef typename MatrixType::GlobalOrdinalType GlobalOrdinal;

  int n = x.coefs.size();
  const ScalarType* xcoefs = &x.coefs[0];
  const ScalarType* ycoefs = &y.coefs[0];
        ScalarType* wcoefs = &w.coefs[0];

  for(int i=0; i<n; ++i) {
    wcoefs[i] = alpha * xcoefs[i] + beta * ycoefs[i];
    if (!A.is_sent[i]) continue;                        //shortcut if possible
    for(int s = 0; s < A.is_sent[i]; ++s){
      int s_num = A.is_sent_to_id[i][s];
      int s_dir = A.is_sent_to_dir[i][s];
      int local = A.is_sent_to_index[i][s];
      A.gran_send_bufs[s_dir][local] = wcoefs[i];          //pack at completion
      int part = local/A.part_elems[s_dir];
      A.part_ready[s_dir][part]++;                         //atomic increment
      if(A.part_ready[s_dir][part] == A.part_elems[s_dir]){ //atomic fetch
        gran_stencil_ready(A.gran_req, s_dir, part);       //send from threaded context
      }
    }
  }
```

Granularity

Fine grained application performance is sensitive to variation in the aggregation of partitions/bins sent over the network:



Key Takeaway

Messages too big→ No early bird perceived bandwidth bump

- Reduced time between first and last partition completion

Messages too small → Lower bandwidth/hardware utilization

- Not saturating the network
- Higher call overheads

The tuning of message granularity is necessary to leverage fine grained communication in applications.

Questions?

Email: wmarts@sandia.gov

User-Level Threading Support in Open MPI

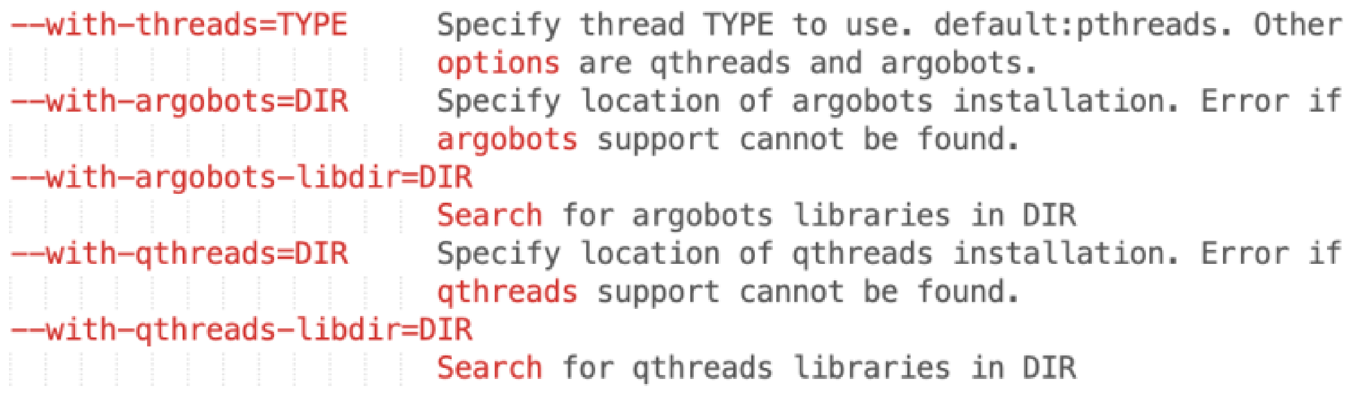
Jan Ciesko
Sandia National Laboratory

User-level Threading in Open MPI

- **Open MPI 4.x and MPICH 3.4.x have ULT* support**
- Easy to use with configure options
- The use of ULTs can be beneficial for performance
- Hybrid applications require ULT support in the MPI implementation for correctness (progress guarantees)

***Note: User-level Threading (ULT)** refers to any threading implementation where the operating system is not aware of such threads. Such threads or "tasks" are light-weight but require cooperative behavior for progress guarantees (cooperative multithreading)*

Configure options: Open MPI



```
--with-threads=TYPE     Specify thread TYPE to use. default:pthreads. Other
                        options are qthreads and argobots.
--with-argobots=DIR     Specify location of argobots installation. Error if
                        argobots support cannot be found.
--with-argobots-libdir=DIR
                        Search for argobots libraries in DIR
--with-qthreads=DIR     Specify location of qthreads installation. Error if
                        qthreads support cannot be found.
--with-qthreads-libdir=DIR
                        Search for qthreads libraries in DIR
```

MPICH

```
--with-thread-package=package
--with-argobots=[PATH]
--with-argobots-include=PATH
--with-argobots-lib=PATH
```

Note: Use *ompi_info --config* to see your MPI configuration

User-level Threading in Open MPI

Experiment with “MPI Partix”

- Application test suite for user-level threading and partitioned communication.
- Contains API examples, benchmarks and correctness tests
- Works with different threading backends

Example code:

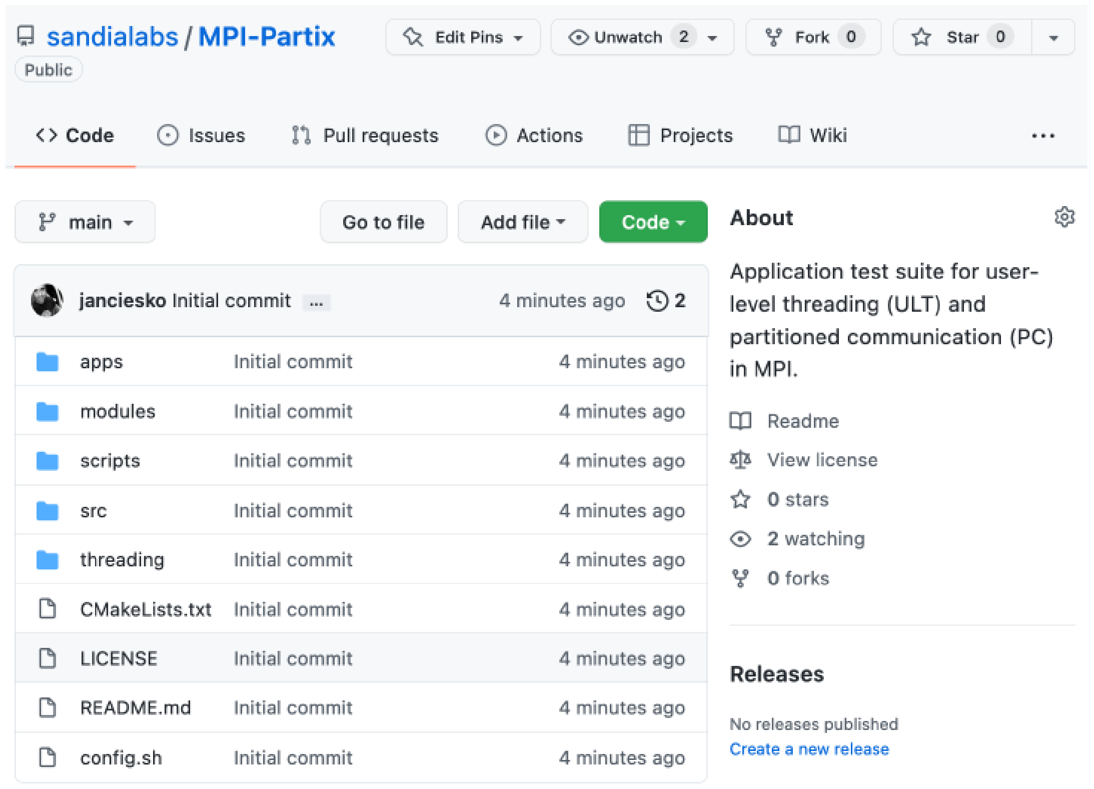
```
#include <cstdio>
#include <partix.h>

void task(partix_task_args_t *args) { printf("Hello World\n"); }

int main(int argc, char *argv[]) {
  partix_config_t conf;
  partix_init(argc, argv, &conf);
  partix_library_init();

  partix_context_t ctx;

  for (int i = 0; i < conf.num_tasks; ++i) {
    partix_task(&task /*functor*/, NULL, &ctx);
  }
  partix_taskwait(&ctx);
  partix_library_finalize();
  return 0;
}
```



jancieско Initial commit ... 4 minutes ago 2

apps	Initial commit	4 minutes ago
modules	Initial commit	4 minutes ago
scripts	Initial commit	4 minutes ago
src	Initial commit	4 minutes ago
threading	Initial commit	4 minutes ago
CMakeLists.txt	Initial commit	4 minutes ago
LICENSE	Initial commit	4 minutes ago
README.md	Initial commit	4 minutes ago
config.sh	Initial commit	4 minutes ago

About

Application test suite for user-level threading (ULT) and partitioned communication (PC) in MPI.

Readme
View license
0 stars
2 watching
0 forks

Releases

No releases published
Create a new release

https://github.com/sandialabs/MPI-Partix

User-level Threading in Open MPI

Cooperative multithreading requires support in MPI implementations

- MPI Partix: **“ULTCorrectness1”**

```c
//set context
partix_context_t ctx;

#if defined(OMP)
#pragma omp parallel num_threads(conf.num_threads)
#pragma omp single
#endif
  // Create in this sequence that starts and ends with recv tasks
  // This tests correct ULT functionality with ULT libs with
  // FIFO or LIFO schedulers
  for (int i = 0; i < conf.num_tasks; i += 2) {
    if (i < 2) {
      partix_task(&task_recv, &task_args, &ctx);
      partix_task(&task_send, &task_args, &ctx);
    } else {
      partix_task(&task_send, &task_args, &ctx);
      partix_task(&task_recv, &task_args, &ctx);
    }
  }

  partix_taskwait(&ctx);

  assert(reduction_var == DEFAULT_VALUE * conf.num_tasks);
```

```c
void task_send(partix_task_args_t *args) {
  int ret;
  MPI_Request request;
  task_args_t *task_args=(task_args_t *)args->user_task_args;

  MPI_Isend(&task_args->some_data, 1, MPI_INT, task_args->target,
            0, comm, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);

  partix_mutex_enter(&mutex);
  reduction_var += task_args->some_data;
  partix_mutex_exit(&mutex);
}
```

```c
void task_recv(partix_task_args_t *args) {
  int ret, tmp;
  MPI_Request request;
  task_args_t *task_args=(task_args_t *)args->user_task_args;

  MPI_Irecv(&tmp, 1, MPI_INT, task_args->target, 0,
            comm, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);

  partix_mutex_enter(&mutex);
  reduction_var += tmp;
  partix_mutex_exit(&mutex);
}
```

Supported for Argobots and Qthreads.

OpenMP tasking requires MPI Continuations here.

https://github.com/sandialabs/MPI-Partix

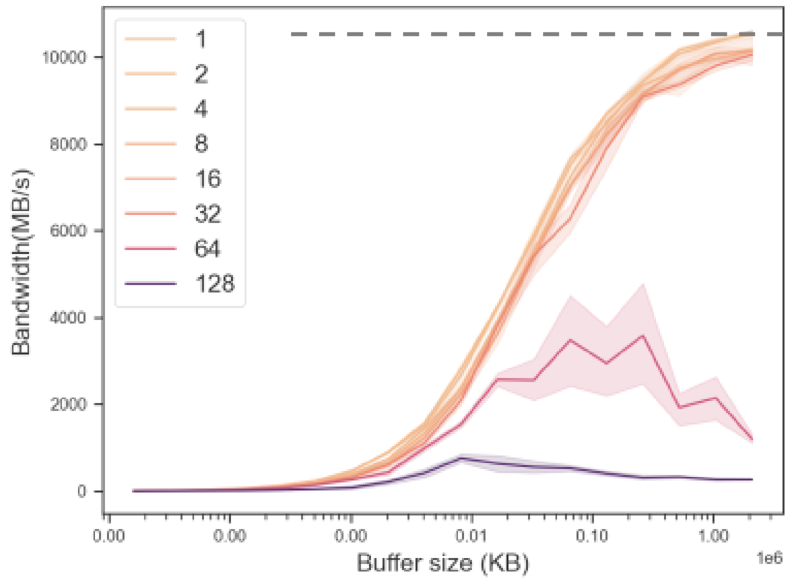
EXASCALE COMPUTING PROJECT

User-level Threading in Open MPI

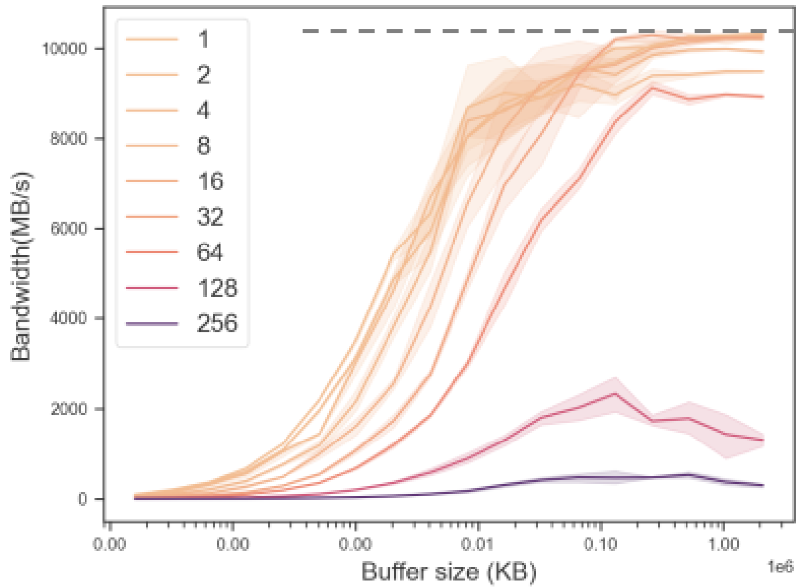
User-level threading (ULT) + Partitioned Communication (Basic)

- MPI Partix: **“Bench1”,** 16KB-2GB buffer size, 1:1 partitions to task mapping

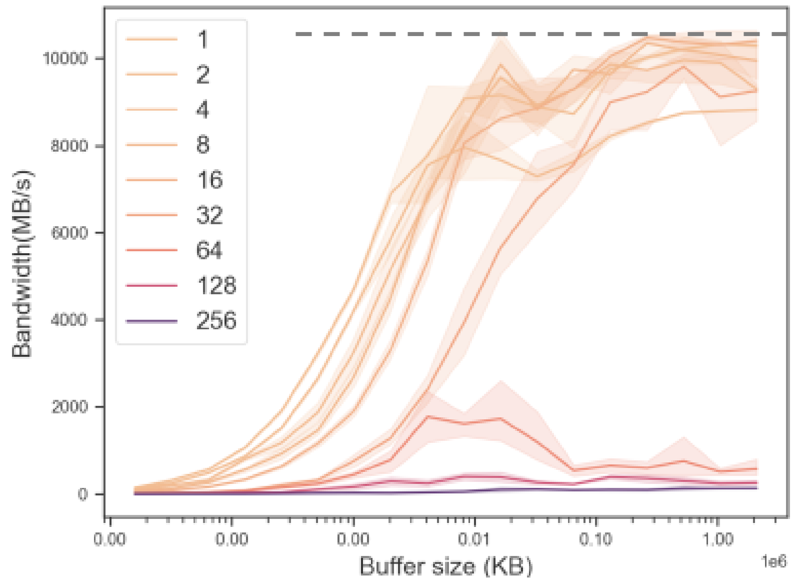
You milage may vary ☺



Blake, x86, UCX, OMPI 5.0.X, **Pthreads**, 1-128 partitions, 1:1 P2T



Blake, x86, UCX, OMPI 5.0.X, **Qthreads**, 1-256 partitions, 1:1 P2T



Blake, x86, UCX, OMPI 5.0.X, **OMP Task***, 1-256 partitions, 1:1 P2T

More benchmarks included. Feel free to experiment and report back!

https://github.com/sandialabs/MPI-Partix

EXASCALE COMPUTING PROJECT

*gcc/10.2.0