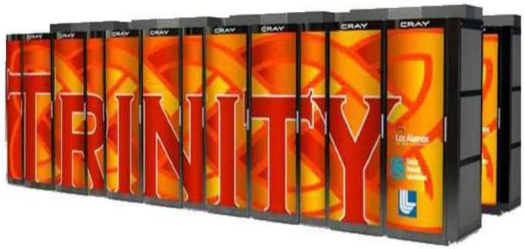
SAND2020-0692PE

MPI Partitioned Communication



PRESENTED BY

Ryan E. Grant,

Scalable System Software (1423)

Images courtesy of: Sandia National Lal

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.

Why Threading?

- MPI use cases continue to evolve
  - MPI+X implies the use of threads, e.g. OpenMP
- Potentially thousands of MPI processes on a single node (without threading)
  - Can complicate network resource management
- Sources of concurrency
  - Many core architectures
    - KNL was many core with 61 cores, now we have 56 core CPUS…
  - Increasing core counts
    - Traditional CPU design continues to add cores in new generations
  - OpenMP
    - Can we use MPI inside of OpenMP parallel regions?
  - New runtimes
    - Tasking runtimes potentially introduce many more concurrent uses of the MPI library

Living in a World with Threads

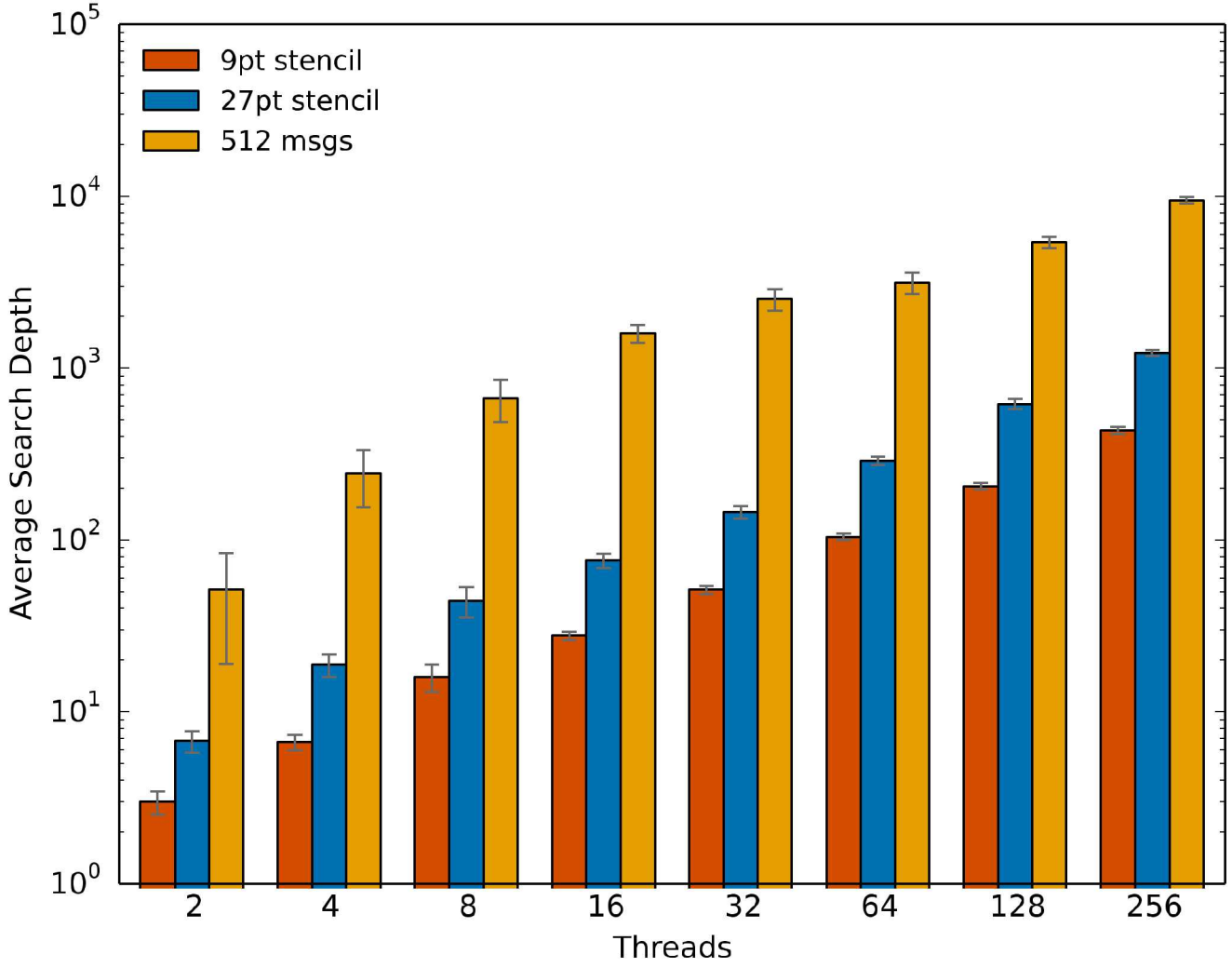
Desirable/required features:

- Low overhead
  - Having many messages and ranks causes message matching/steering overhead (Endpoints)
- Similar semantics to existing threading (minimal changes)
  - Ease of programmability
- Each thread has access to the communication library (no funneling)
  - Also ease programmability, e.g. use MPI calls in an OpenMP region
- Communication endpoint granularity matched to the work
  - Not too fine, not too coarse, just right…
  - Fine granularity at endpoints requires networking resources
    - Keeping track of many ranks, caching state related to these ranks, etc.

The problem with threads

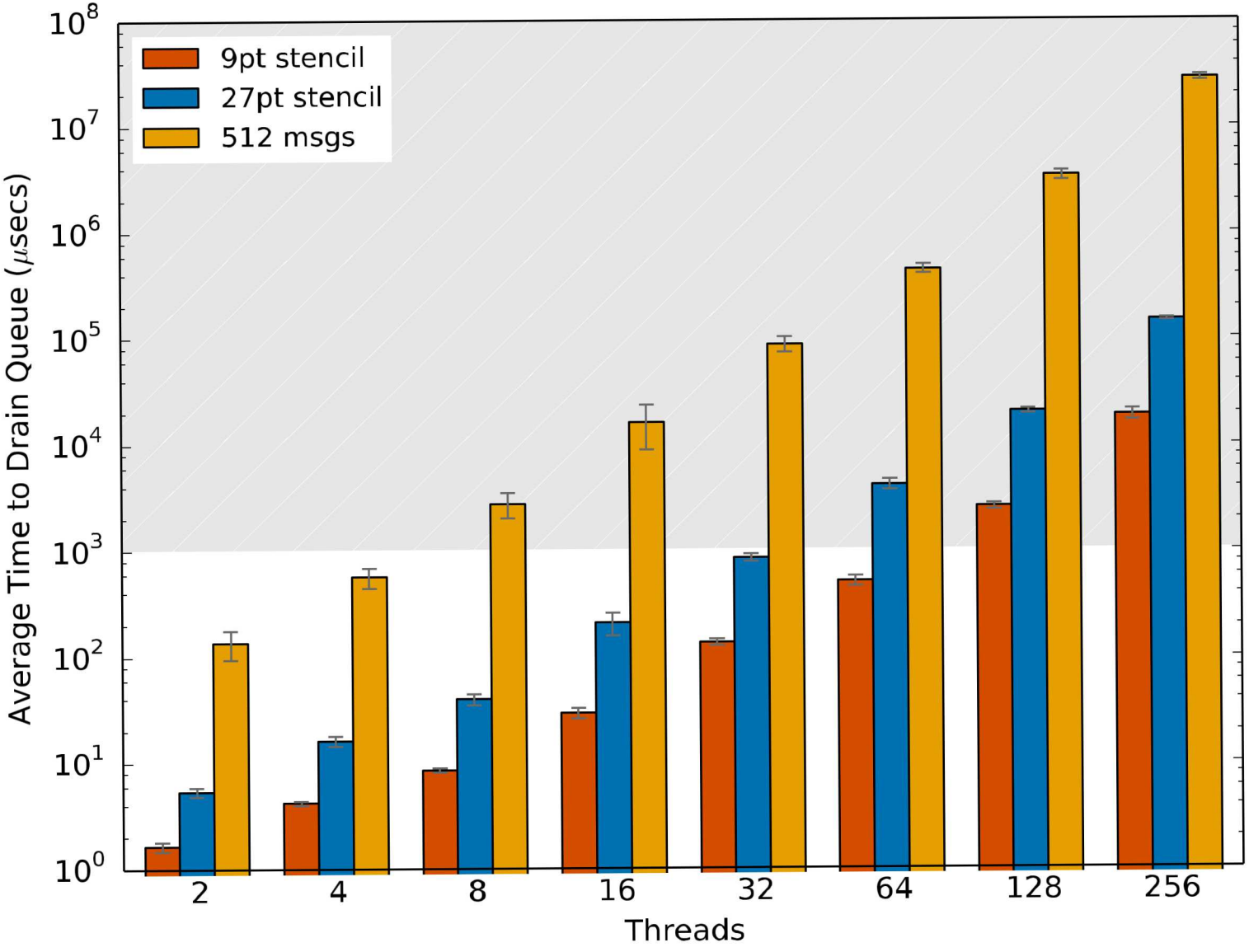
- Threads introduce significant issues with concurrency in existing MPI implementations
  - MPI_THREAD_MULTIPLE is hard and implementations don't do it well
  - Difficult to support concurrency without encountering conflicts
- Fixing MPI concurrency issues is hard, so multiple approaches proposed:
  - 1) Expose the parallelism (Endpoints)
    - Allow individual threads to have ranks in MPI (rank space explosion)
    - Exposes lots of complexity, thread addressability on the network
    - Uses MPI's existing threading over entire library approch
  - 2) Allow for parallelism in a way that doesn't require MPI to handle it
    - New concept: threads can contribute independently to a larger communication operation inside of MPI, doesn't require the same synchronization methods

The Coming Thread Storm

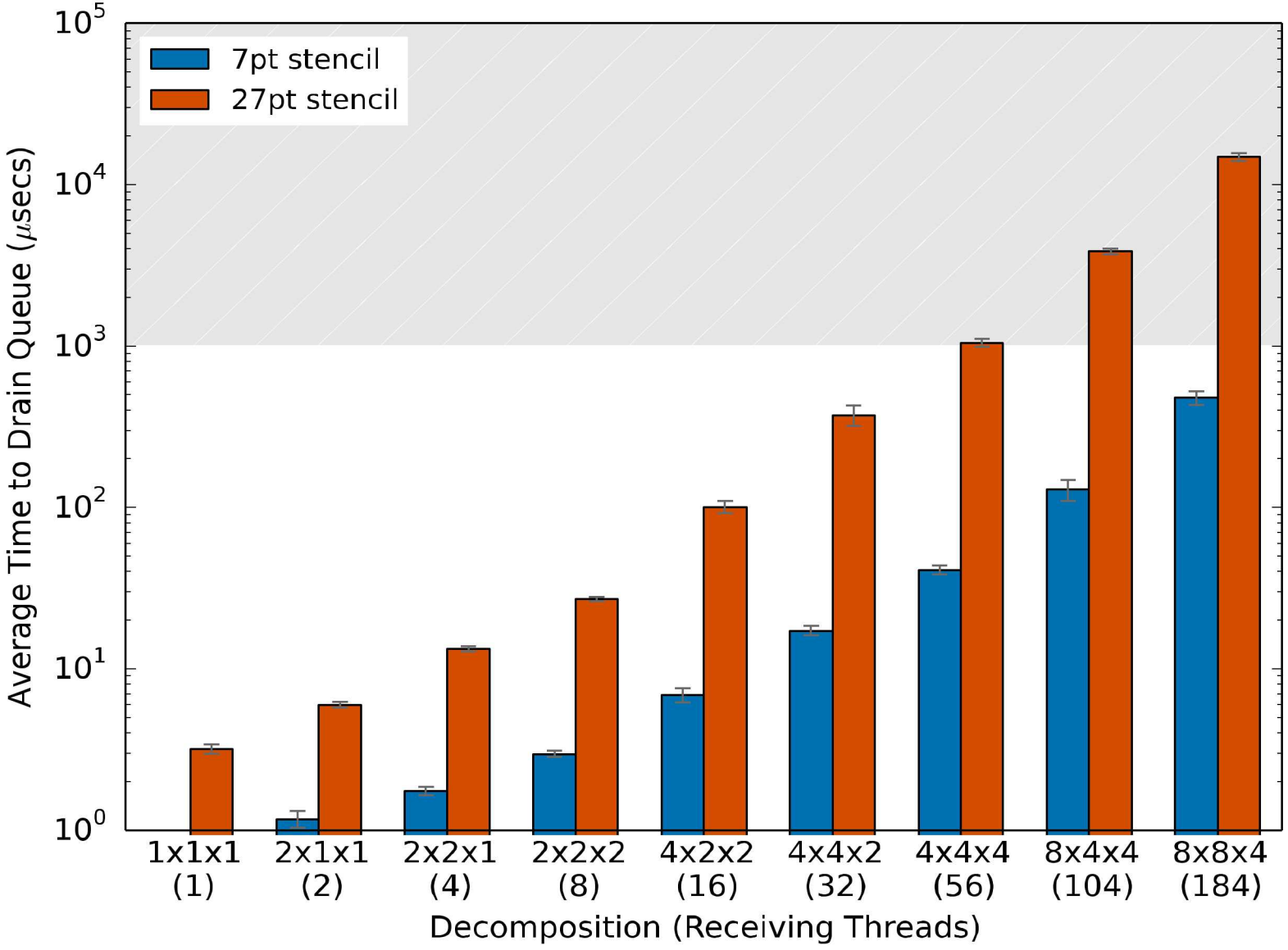


Data from Measuring Multithreaded Message Matching Misery, EuroPar 2018, Whit Schonbein et al.

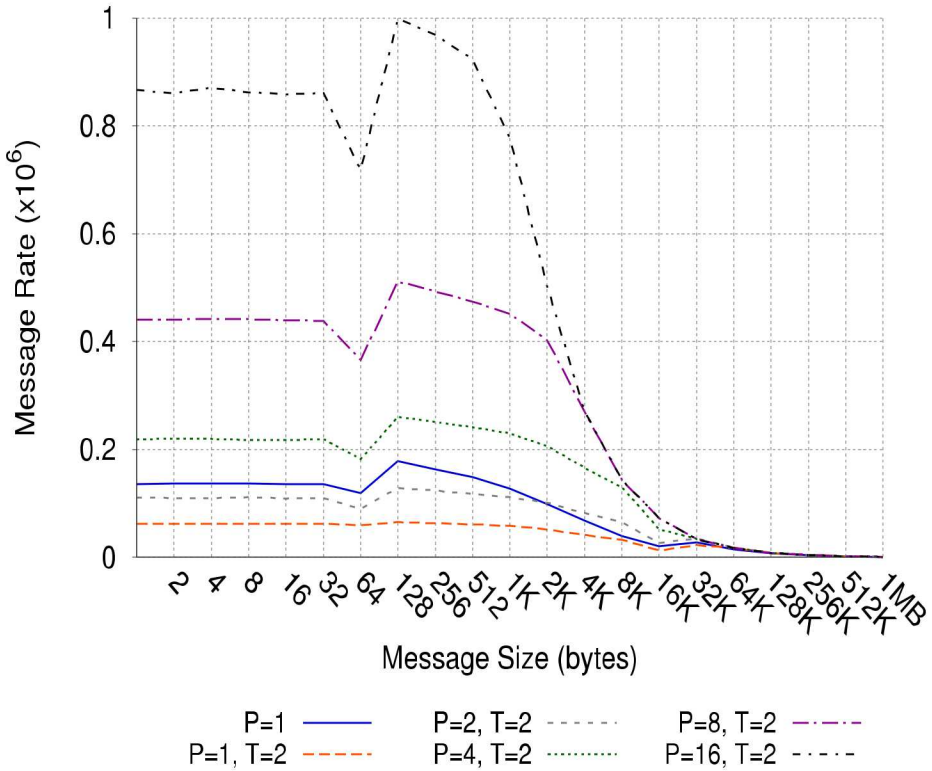
The Coming Thread Storm

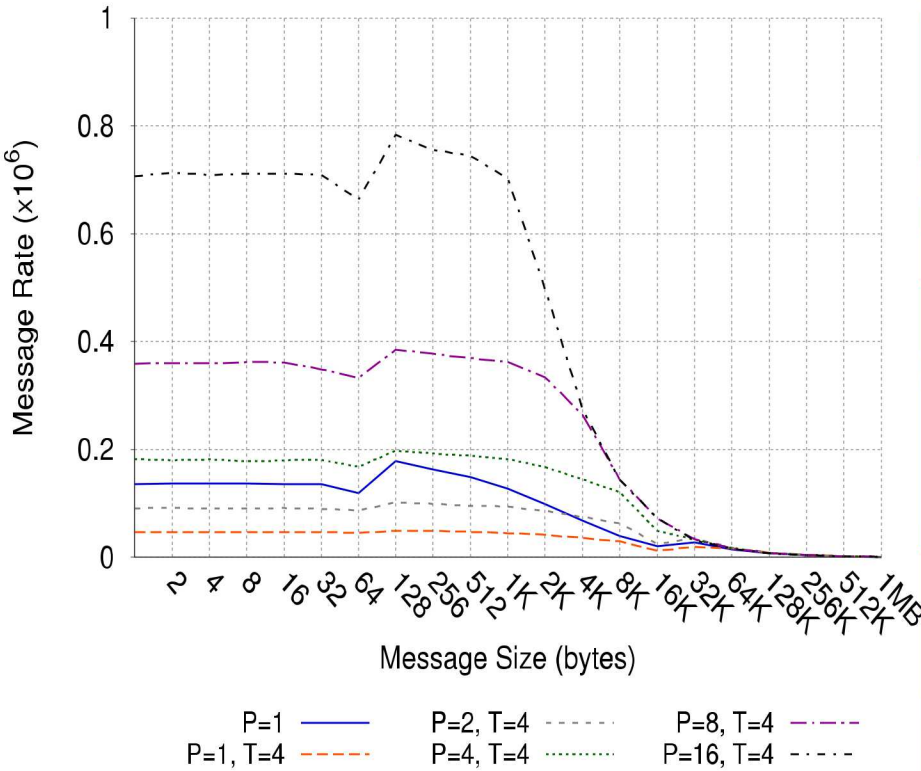


The Coming Thread Storm



Exposing Parallelism in MPI





Impact of communication concurrency on message rate per second for P processes and T threads per process on an Intel Phi

Takeaway: Fewer threads = better performance, due to poor MPI multi-threading

James Dinan, Ryan E. Grant, Pavan Balaji, Dave Goodell, Douglas Miller, Marc Snir, Rajeev Thakur, "Enabling communication concurrency through flexible MPI Endpoints", International Journal of High Performance Computing Applications, Volume 28, Issue 4, pp. 390-405, November 2014. Impact factor: 1.63

The Coming Thread Storm

- Trouble is around the corner
- Need a solution that gives us performance more like today
- Don’t want to expand rank space
- Don’t want to blow up the message queue

GPUs and Accelerators

- GPUs cannot run MPI libraries natively
- Need coordination for network transfers
  - But don't want to setup communications from step one on the GPU/Accelerator
- Minimize overhead of communication initialization
  - Many potential notifications – must be lightweight
- Existing NIC hardware can use triggering
  - Need a mechanism in MPI to do lightweight triggering
- Communication can be optimized by host CPU
  - CPU can optimize to network before the transfer takes place
  - Optimize number of transfers, when things trigger

Allow for Better Parallelism in MPI

- Concept of many actors (threads) contributing to a larger operation in MPI
  - Same number of messages as today!
  - No new ranks
- E.G. many threads work together to assemble a message
  - MPI only has to manage knowing when completion happens
  - These are actor/action counts, not thread level collectives, to better enable tasking models
- 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
- Bonus: Match well with Offloaded NIC capabilities
  - Use counters for sending/receiving
  - Utilize triggered operations to offload sends to the NIC

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 MPIX_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 MPIX_Pready( void* buf, int count, MPI_Datatype in_datatype,
   int offset_index, MPI_Request *request);
```

- Wait on completion

```
MPI_Wait(request)
```

- Optional: Use the same partitioned send over again

```
MPI_ Start(request)
```

Example from MPI-4.0


```
#define NUM_THREADS 8
#define NUM_TASKS 64
#define PARTITIONS NUM_TASKS
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main( int argc, char *argv[]) /* send-side partitioning */
{
  double message[MESSAGE_LENGTH];
  int send_partitions = PARTITIONS,
     send_partlength = PARTLENGTH,
     recv_partitions = 1,
     recv_partlength = PARTITIONS*PARTLENGTH;
  int count = 1, source = 0, dest = 1, tag = 1,
  flag = 0;
  int myrank;
  int provided;
  MPI_Request request;
  MPI_Info info = MPI_INFO_NULL;
  MPI_Datatype send_type;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  if (provided < MPI_THREAD_SERIALZED)
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
  MPI_Type_commit(&send_type);

  if (myrank == 0)    /* code for process zero */
  {
    MPI_Psend_init(message, send_partitions, count, send_type, dest,
tag,
            info, MPI_COMM_WORLD, &request);
    MPI_Start(&request);

    #pragma omp parallel shared(request)
num_threads(NUM_THREADS)
    {
     #pragma omp single
      {
       /* single thread creates 64 tasks to be executed by 8 threads */
       for (int
partition_num=0;partition_num<NUM_TASKS;partition_num++)
       {
        #pragma omp task firstprivate(partition_num)
        {
  /* compute and fill partition #partition_num, then mark ready: */
          /* buffer is filled in arbitrary order from each task */
          MPI_Pready(request, partition_num);
        } /*end task*/
      } /* end for */
    } /* end single */
  } /* end parallel */
  while(!flag)
  {
    /* Do useful work */
    MPI_Test(&request, &flag,  MPI_STATUS_IGNORE);
    /* Do useful work */
  }
  MPI_Request_free(&request);
 }
 else if (myrank == 1)  /* code for process one */
 {
   MPI_Precv_init(message, recv_partitions, recv_partlength,
MPI_DOUBLE,
          source, tag, info, MPI_COMM_WORLD, &request);

   MPI_Start(&request);
 while(!flag)
   {
     /* Do useful work */
     MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
     /* Do useful work */
   }
   MPI_Request_free(&request);
 }
 MPI_Finalize();
 return 0;
}
```

Works for non-Persistent Comms

- We can have similar functionality with a MPI_Psend

int MPIX_Ipsend(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);

- Works just like a regular send with contribution counts
- First thread to reach Psend gets a request handle back that can be shared with other threads – some MPI locking
- Setup happens on first call
- Track by comm and buff addr

Opportunities for Optimization

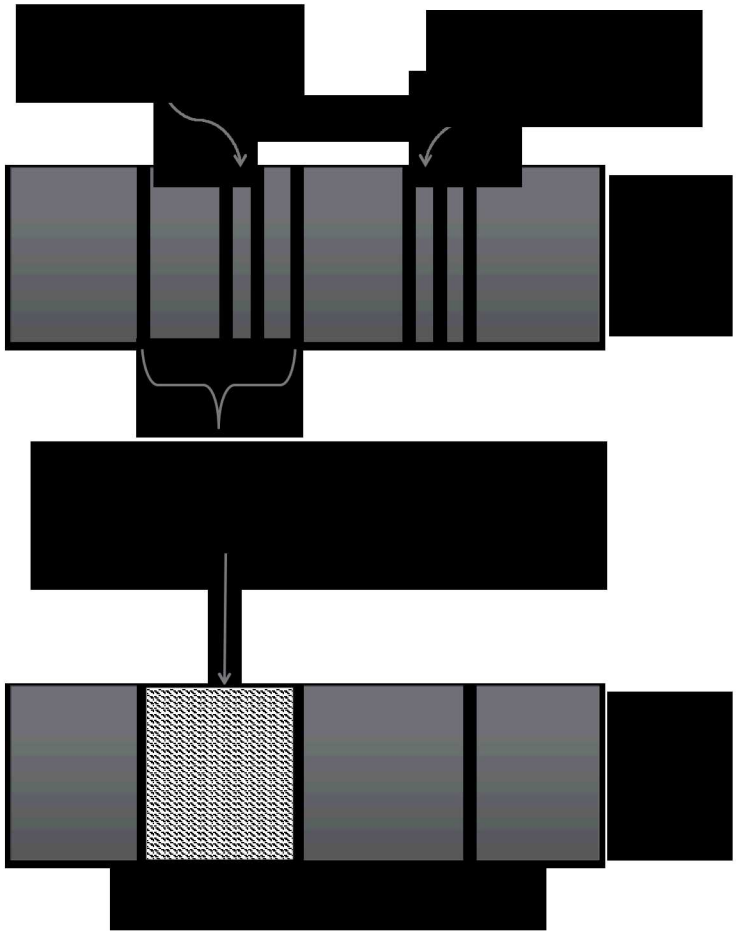
MPI implementations can optimize data transfer under the covers:

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

  $1 < \#\text{messages} \leq \#\text{threads/tasks}$

  For a partition with 1 part per thread

- Reduces the total number of messages sent, decreasing matching overheads

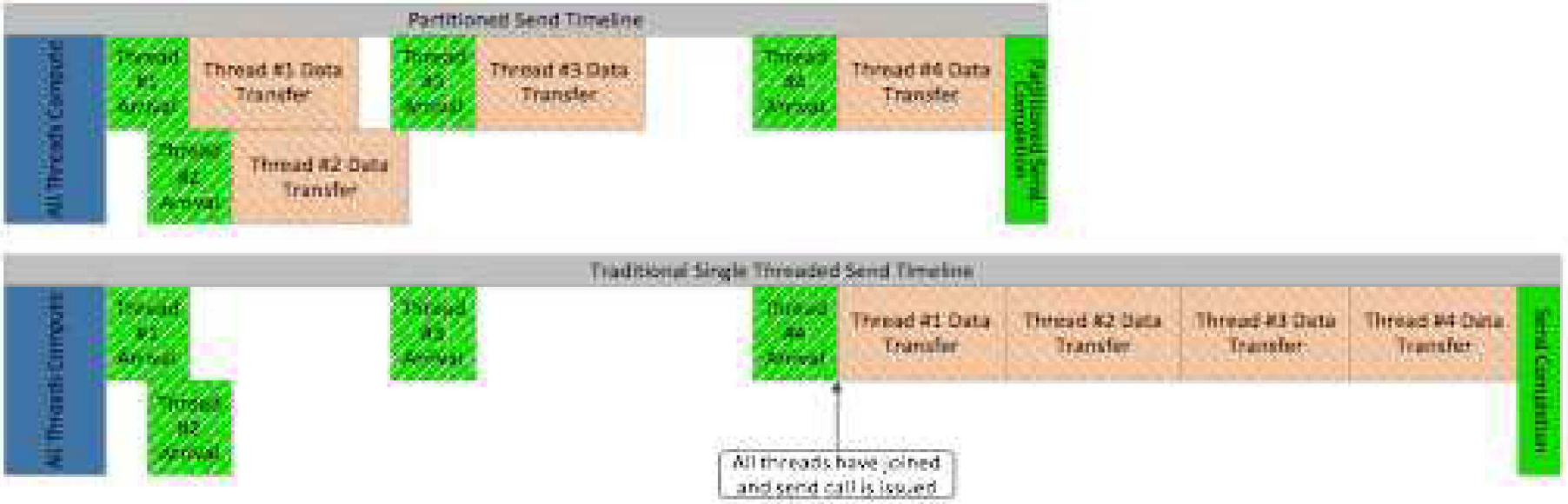


Different Approach to Threading

- Partitioned buffer operations can always be consider as multi-threaded
- Using partitioned sends doesn't necessarily require locking in other parts of the MPI library – confine threading in MPI
- Technically, using partitioned buffers would work with MPI_THREAD_SERIALIZED
  - If only using partitioned sends, no need for locking in the library
- No more thread_multiple?
  - Not quite, but we can have alternative threading modes for MPI, where user management of data is guaranteed and only inherently thread safe operations are called, MPI_THREAD_PARTITIONED

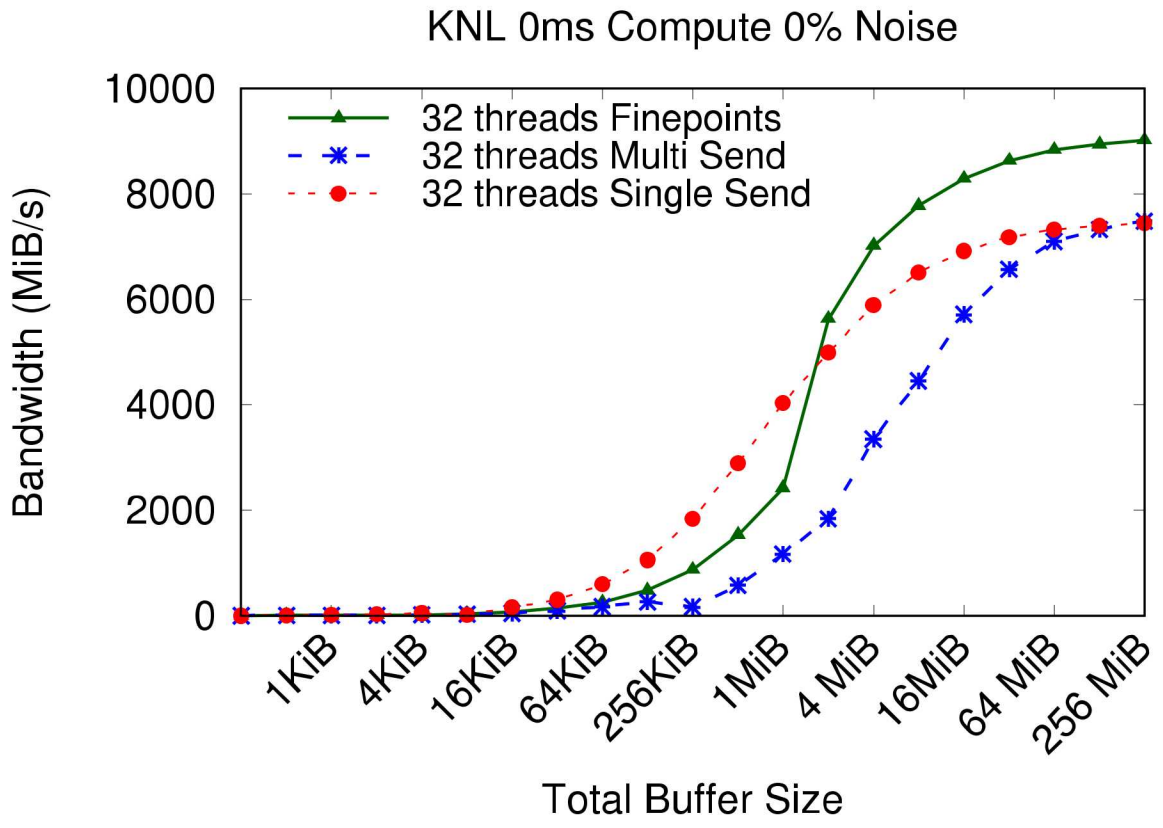
New Type of Overlap

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



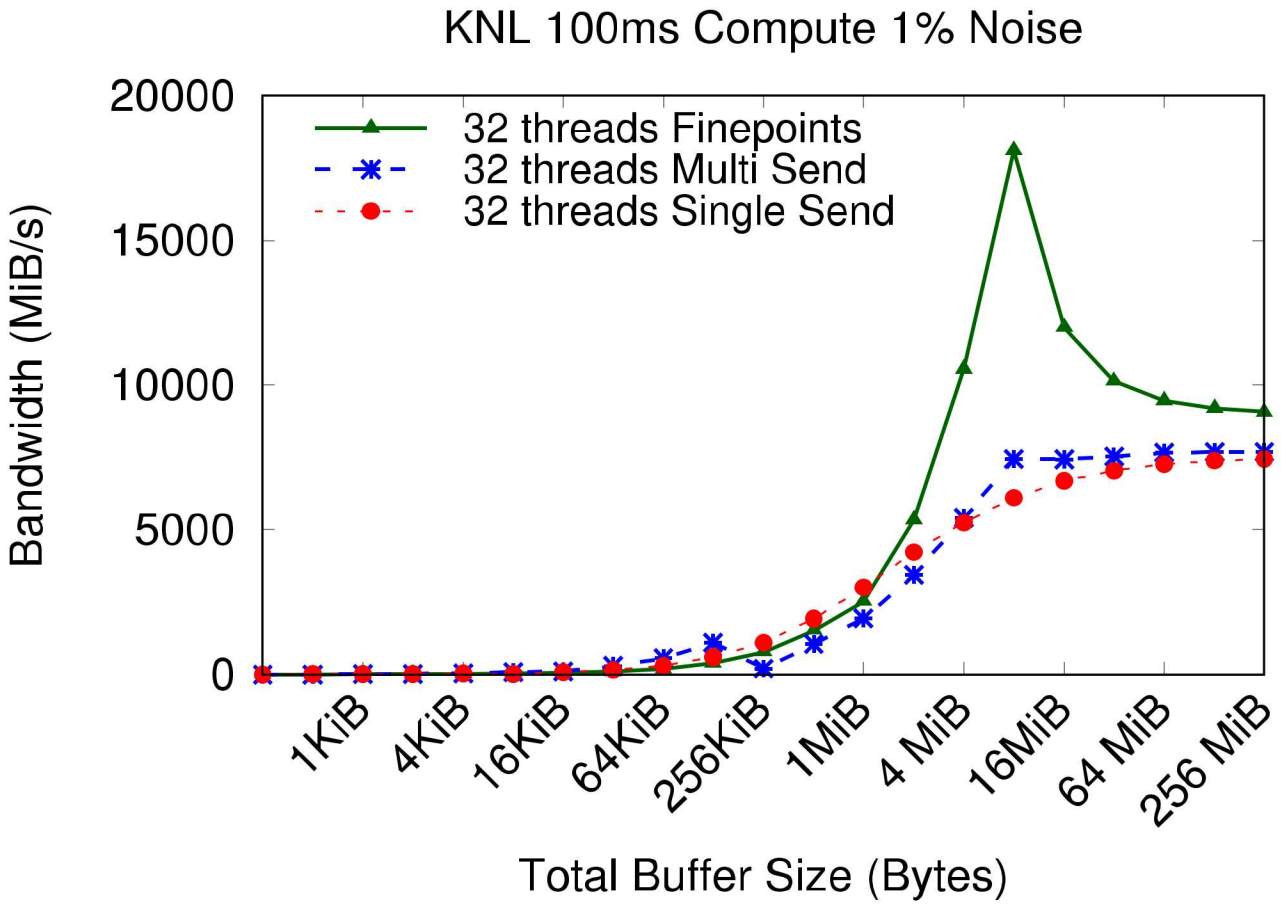
Performance Results

- Still not super tuned, basic library sitting on top of MPI
- Results with 32 threads



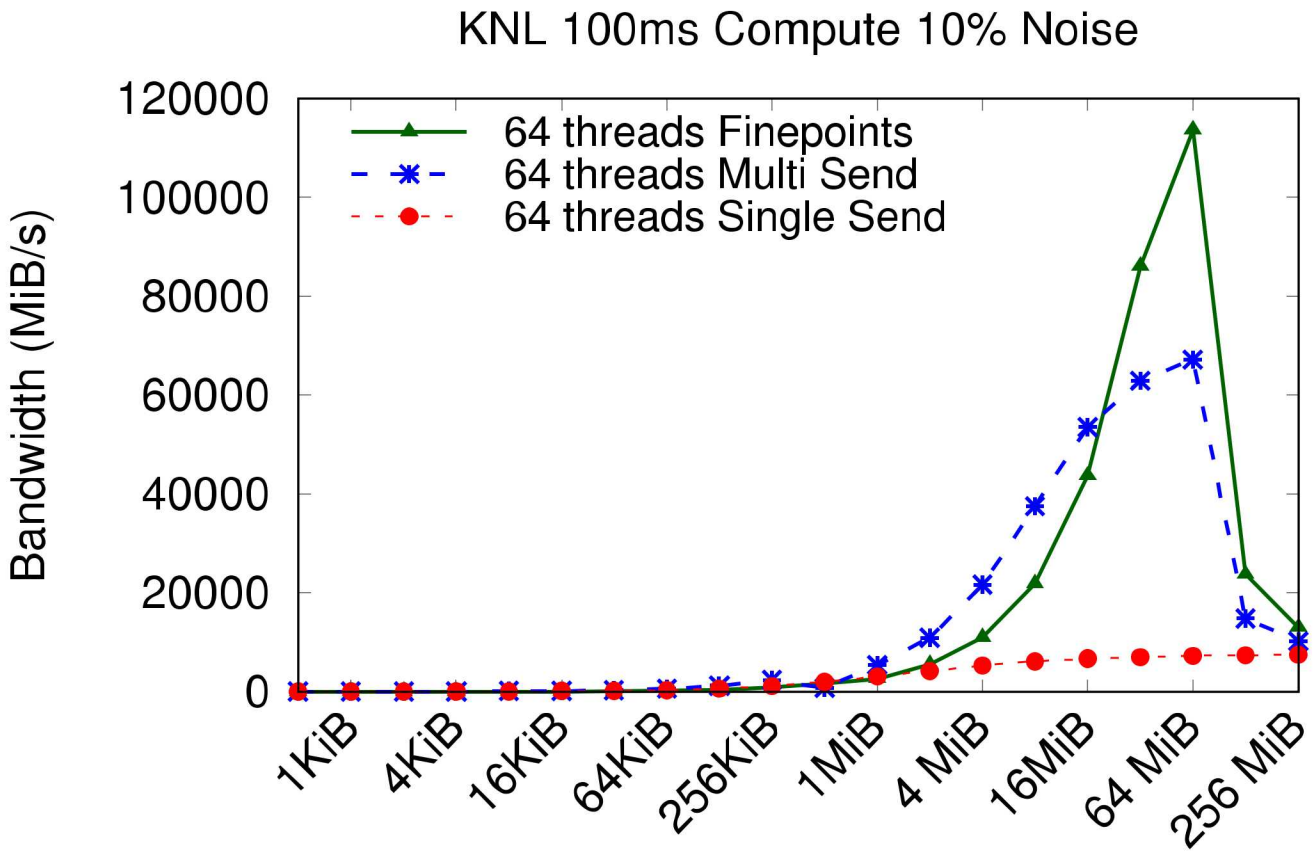
Performance Results

- Still not super tuned, basic library sitting on top of MPI
- Results with 32 threads



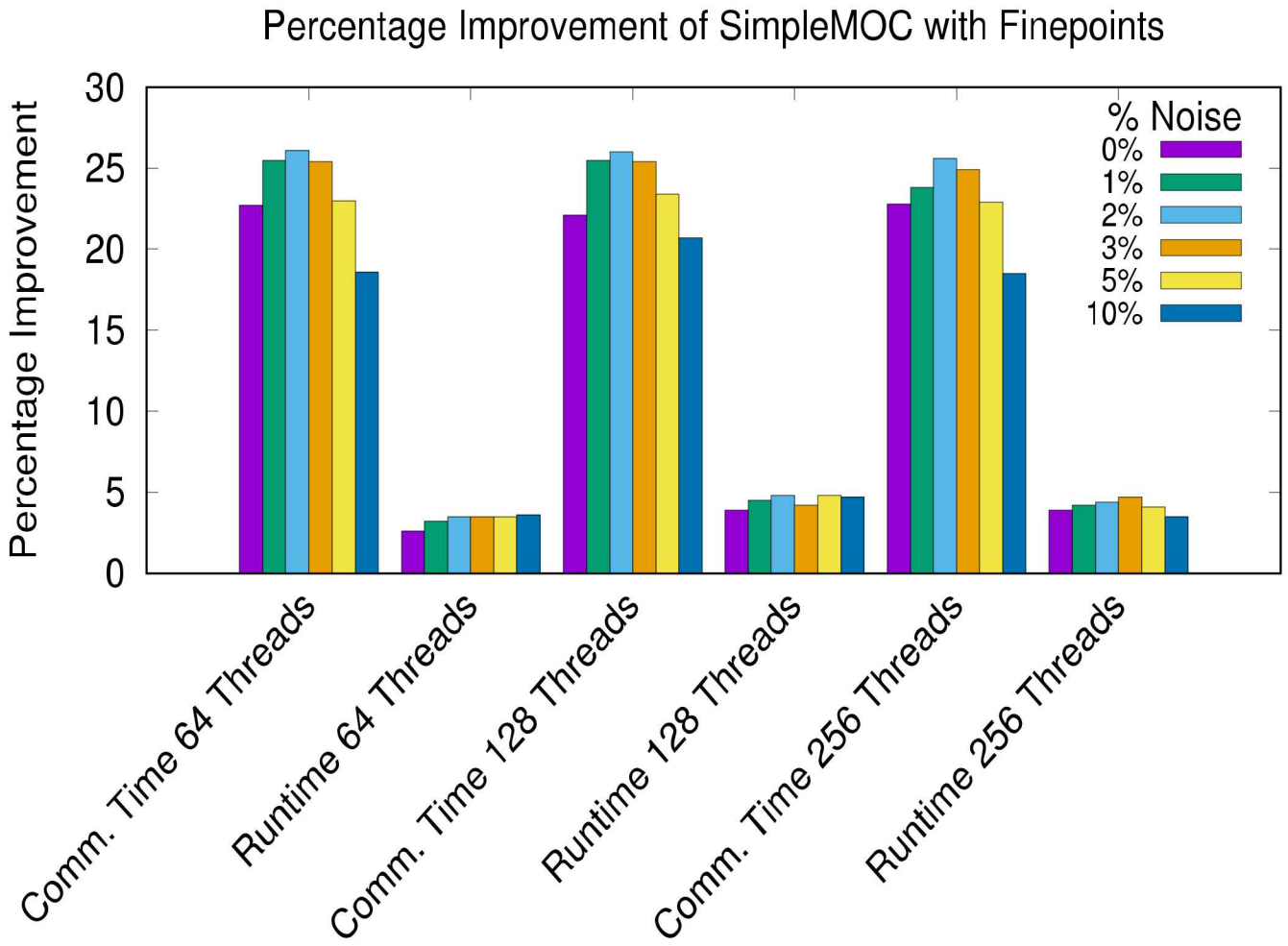
Performance Results

- Still not super tuned, basic library sitting on top of MPI
- Results with 64 threads, full overlap, 12X more “bandwidth”



Application Benefit

- Real reactor physics proxy app: SimpleMOC



Partitioned Comm. Benefits

- Performance benefit of early-bird overlap
  - Better than current fork-join-communicate methods
- Lightweight thread synchronization
  - Single atomic
- Same message/matching load as today
  - Avoid the coming storm
- A great way to adapt code to use RDMA underneath
  - Keeping existing send/recv completion semantics
- Easy to translate code
  - May be able to automatically translate send/recv with simple parallel loops over to psend

Takeaways

- Implementation work already done for library
  - Library works for OMPI and MPICH so very low effort for base functionality
  - Integration underway for OMPI
    - There are things we cannot optimize at the library level because we do not know enough about the low level network details
- Simplifies Multi-threading work
  - Confines multi-threading to a small portion of the library
  - Concentrate on performance only in multi-threaded specific calls
- Proposal builds a base to build other future multi-threaded solutions
  - This is essentially the shared-buffer partitioned send here
  - Could have some receive-side partitioning when careful about overheads

Thank you

Questions?

Latest Proposal Text: https://github.com/regrant/mpi-standard

Acknowledgments:

This work was funded through the Computational Systems and Software Environment sub-program of the Advanced Simulation and Computing Program funded by the National Nuclear Security Administration

References – more background reading

[1] B. W. Barrett, R. Brightwell, R. Grant, S. D. Hammond, and K. S. Hem- mert, “An evaluation of mpi message rate on hybrid-core processors,” 2014.

[2] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, “Early experiences co-scheduling work and communication tasks for hybrid mpi+ x applications,” in 2014 Workshop on Exascale MPI at Supercomputing Conference. IEEE, 2014, pp. 9–19.

[3] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, “The portals 4.1 networking programming interface,” 2017.

[4] M. G. Dosanjh, R. E. Grant, P. G. Bridges, and R. Brightwell, “Re- evaluating network onload vs. offload for the many-core era,” in 2015 IEEE International Conference on Cluster Computing. IEEE, 2015, pp. 342–350.

[5] R. E. Grant, A. Skjellum, and V. Purushotham, “Lightweight threading with mpi using persistent communications semantics,” 2015.

[6] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, “A survey of mpi usage in the us exascale computing project,” Concurrency and Computation: Practice and Experience, 2018.

[7] W. Schonbein, M. G. Dosanjh, R. E. Grant, and P. Bridges, “Measuring multithreaded message matching misery,” in International European Conference on Parallel and Distributed Computing (EuroPar), 2018.

[8] K. B. Ferreira, S. Levy, K. Pedretti, and R. E. Grant, “Characterizing mpi matching via trace-based simulation,” in 24th European MPI Users’ GroupMeeting(EuroMPI). DOI:10.1145/3127024.3127040,2017,pp. 1–8. 10-Jan 2019

[9] S. Levy, K. B. Ferreira, W. Schonbein, R. E. Grant, and M. G. Dosanjh, “Using simulation to examine the effect of mpi message matching costs on application performance,” Parallel Computing, vol. 84, pp. 63–74, 2019.

[10] K. Ferreira, R. E. Grant, M. J. Levenhagen, S. Levy, and T. Groves, “Hardware mpi message matching: Insights into mpi matching behavior to inform design,” Concurrency and Computation: Practice and Expe- rience, p. e5150, 2019.

[11] R. E. Grant, M. G. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, “Finepoints: Partitioned multithreaded mpi communication,” in International Conference on High Performance Computing. Springer, Cham, 2019, pp. 330–350.

[12] M. G. Dosanjh, W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Ghaz- imirsaeed, and A. Afsahi, “Fuzzy matching: Hardware accelerated mpi communication middleware,” in 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019), 2019.

[13] W. P. Marts, M. G. Dosanjh, W. Schonbein, R. E. Grant, and P. G. Bridges, “Mpi tag matching performance on connectx and arm,” in Proceedings of the 26th European MPI Users’ Group Meeting, 2019, pp. 1–10.

[14] R. E. Grant and A. Afsahi, “Power-performance efficiency of asym- metric multiprocessors for multi-threaded scientific applications,” in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006.