MT-MPI:
Multi-threaded MPI for Many-core Environments

Min Si

msi@mcs.anl.gov

Research Aide at Argonne National Laboratory

advisor: Dr. Antonio Pena, Dr. Pavan Balaji

Ph.D. student at The University of Tokyo

advisor: Prof. Yutaka Ishikawa

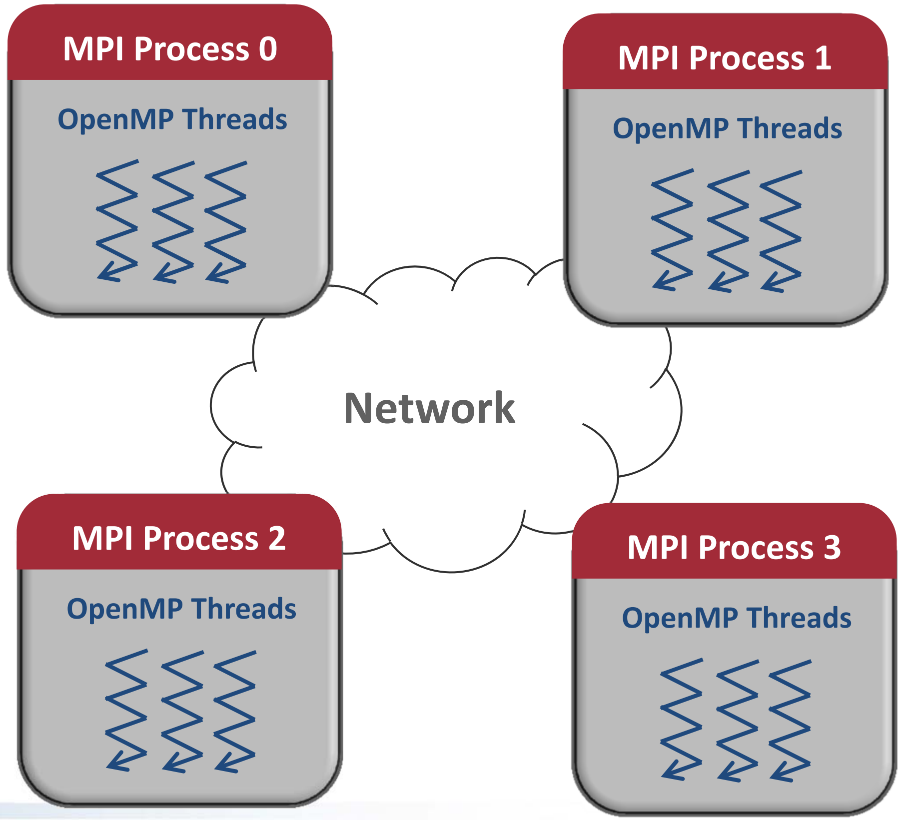
Many-core Architecture

- Massively parallel environment
- Intel® Xeon Phi co-processor
  - 60 cores inside a single chip, 240 hardware threads
  - SELF-HOSTING in next generation, NATIVE mode in current version

- Blue Gene/Q
  - 16 cores per node, 64 hardware threads

Hybrid OpenMP + MPI Programming

- Multi-threads of a process shares local resources
- Parallelize local computation more efficiently
- MPI between nodes



Four levels of MPI Thread Safety

- MPI_THREAD_SINGLE
  - MPI only, no threads
- **MPI_THREAD_FUNNELED**
  - Outside OpenMP parallel region, or OpenMP master region

```
#pragma omp parallel for
 for (i = 0; i < N; i++) {
        uu[i] = (u[i] + u[i - 1] +  u[i + 1])/5.0;
 }

 MPI_Function ( );
```

Four levels of MPI Thread Safety

- **MPI_THREAD_SERIALIZED**
  - Outside OpenMP parallel region, or OpenMP single region, or critical region

```
#pragma omp parallel
{
        /* user computation */
        #pragma omp single
        MPI_Function ();
}
```

```
#pragma omp parallel
{
        /* user computation */
        #pragma omp critical
        MPI_Function ();
}
```

- MPI_THREAD_MULTIPLE
  - Multiple threads, any thread is allowed to make MPI calls at any time.

Problem: Idle Resources during MPI Calls

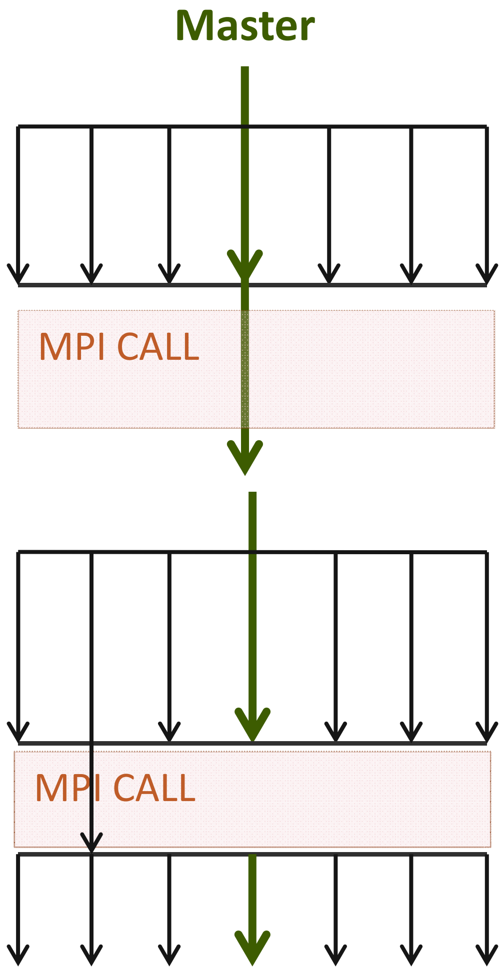
- Threads are only active in the computation phase
- Threads are IDLE during MPI calls

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
        uu[i] = (u[i] + u[i - 1] +  u[i + 1])/5.0;
}

MPI_Function ( );
```

(a) Funneled mode

```
#pragma omp parallel
{
        /* user computation */

        #pragma omp single
        MPI_Function ();
}
```

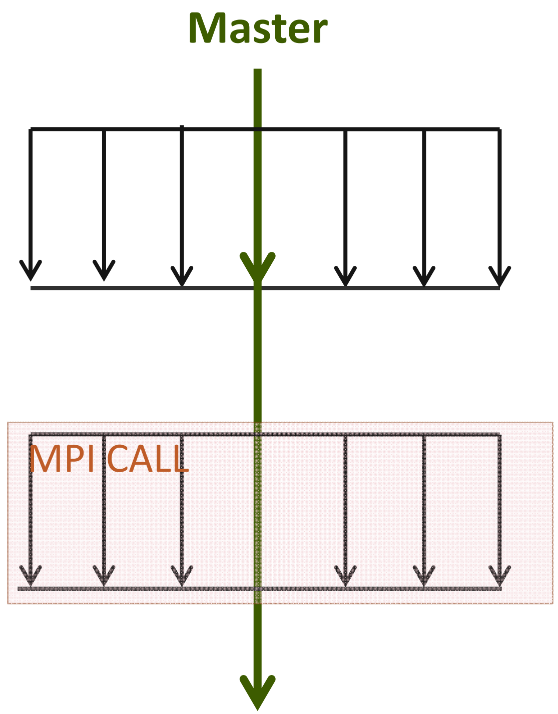
(b) Serialized mode



Solution:
Sharing Idle Threads with Application inside MPI

```
#pragma omp parallel
{
        /* user computation */

        #pragma omp single
        MPI_Function () {
                #pragma omp parallel
                {
                        /* MPI internal task */
                }
        }
}
```

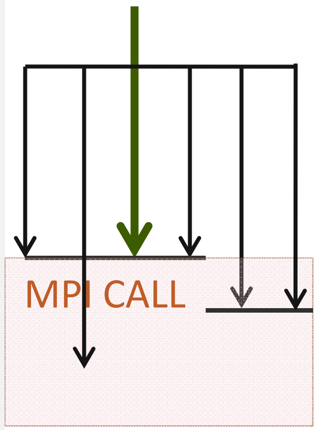
(b) Serialized mode



Challenges

- Some parallel algorithms are not efficient with insufficient threads , need tradeoff, but the number of available threads is UNKNOWN !
- Nested parallelism
  - Simply creates new Pthreads
  - Offloads thread scheduling to OS, caused threads OVERRUNNING issue

```
#pragma omp parallel
{
    /* user computation */

    #pragma omp single
    MPI_Function( ){
        ...
    }
}
```



(a) Unknown number of IDLE threads

```
#pragma omp parallel                    Creates N Pthreads
{
    #pragma omp single
    {
        #pragma omp parallel
        { ... }                         Creates N Pthreads
    }
}
```

(b) Threads overrunning

Outline

- Motivation
- Problem Statement and Solution
- Design and Implementation
  - OpenMP runtime
  - MPI Internal Parallelism
- Evaluation
- Conclusion

OpenMP Runtime Extension 1

- Expose the number of idle threads

```
#pragma omp parallel
#pragma omp single
{
        #pragma omp parallel num_threads(omp_get_num_idle_threads())
        { … }
}
```

  - $N_{IDLE\ threads} \leq OMP\_NUM\_THREADS$
  - $N_{IDLE\ threads} = N_{threads\ in\ pool} + N_{waiting\ threads} + 1\ Master\ thread$

OpenMP Runtime Extension 2

- Waiting progress in barrier
  - **SPIN LOOP** until timeout !
  - May cause **OVERSUBSCRIBING**

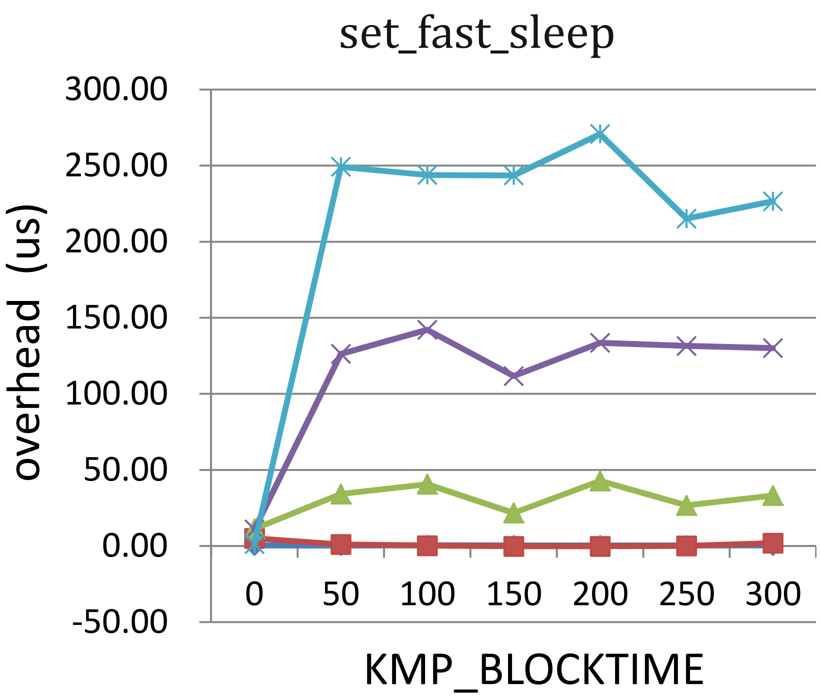
```
while (time < KMP_BLOCKTIME){
        if (done) break;
        /* spin loop */
}
pthread_cond_wait (…);
```

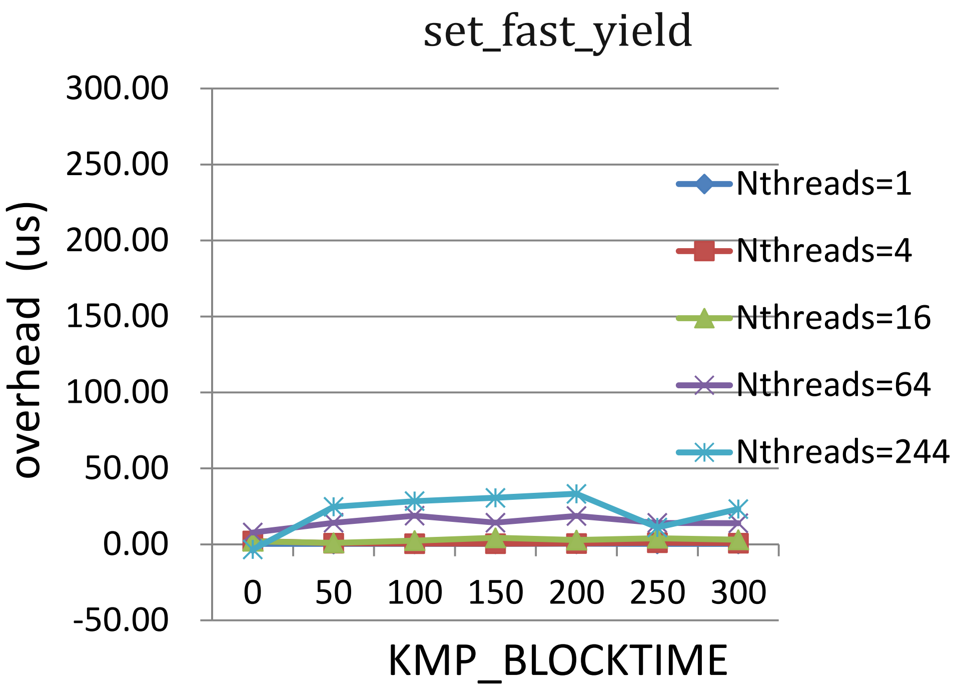
- Solution: Force waiting threads to enter in a passive wait mode inside MPI
  - set_fast_yield (sched_yield)
  - set_fast_sleep (pthread_cond_wait)

```
# pragma omp parallel
# pragma omp single
{
        set_fast_yield (1);

        #pragma omp parallel
        { … }

}
```

OpenMP Runtime Extension 2

- set_fast_sleep VS set_fast_yield
  - Test bed: Intel Xeon Phi cards (stepping B0, 61 cores)



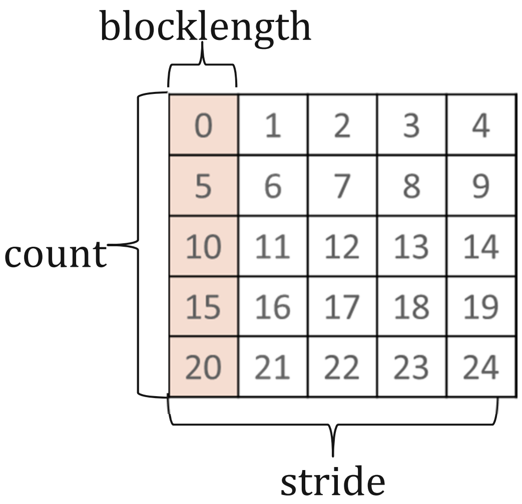


Outline

- Motivation
- Problem Statement and Solution
- Design and Implementation
  - OpenMP runtime
  - MPI Internal Parallelism
    - Datatype Related Functions
    - Intra-node Large Message Communication
    - Netmod Optimizations
- Evaluation
- Conclusion

Derived Data Type Packing Processing

- MPI_Pack / MPI_Unpack
- Communication using Derived Data Type
  - Transfer non-contiguous data
  - Pack / unpack data internally




```
#pragma omp parallel for
for (i=0;  i<count;  i++){
        dest[i] = src[i * stride];
}
```

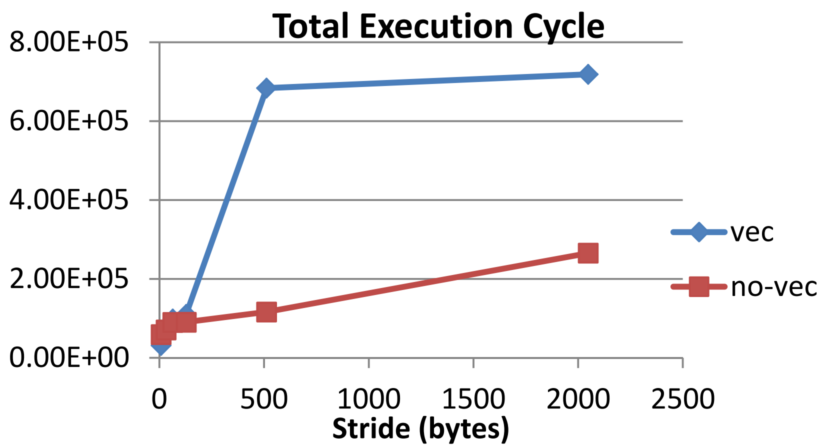
Prefetching issue when compiler vectorized non-contiguous data

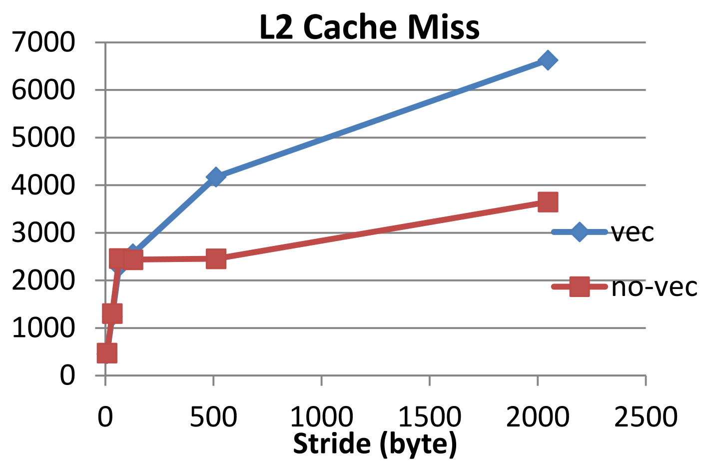
```
for (i=0; i<count; i++){
        *dest++ = *src;
        src += stride;
}
```

(a) Sequential implementation (not vectorized)

```
#pragma omp parallel for
for (i=0; i<count; i++){
        dest[i] = src[i * stride];
}
```

(b) Parallel implementation (vectorized)





Outline

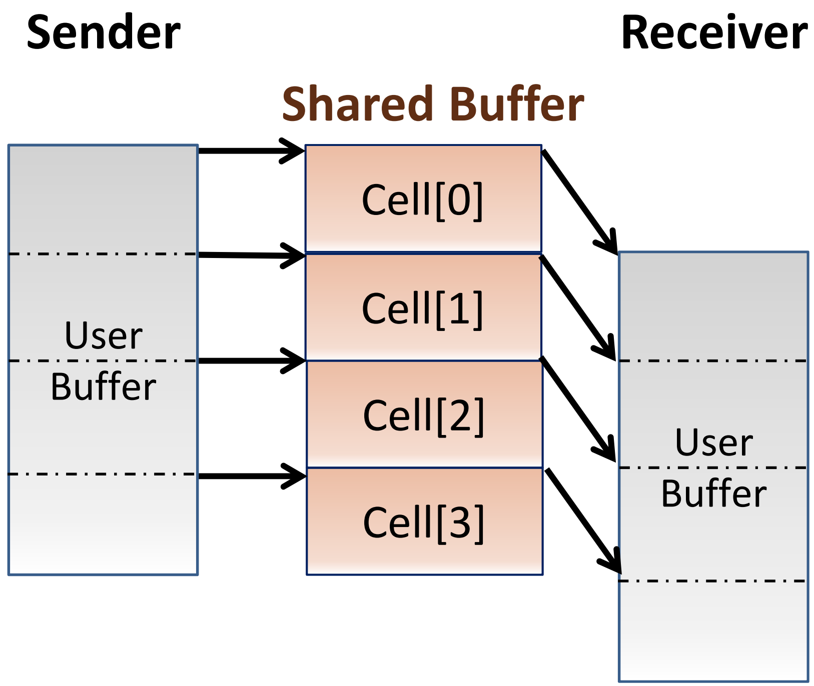
- Motivation
- Problem Statement and Solution
- Design and Implementation
  - OpenMP runtime
  - MPI Internal Parallelism
    - Datatype Related Functions
    - Intra-node Large Message Communication
    - Netmod Optimizations
- Evaluation
- Conclusion

Sequential pipelining LMT

- Shared user space buffer
- Pipelining copy on both sender side and receiver side

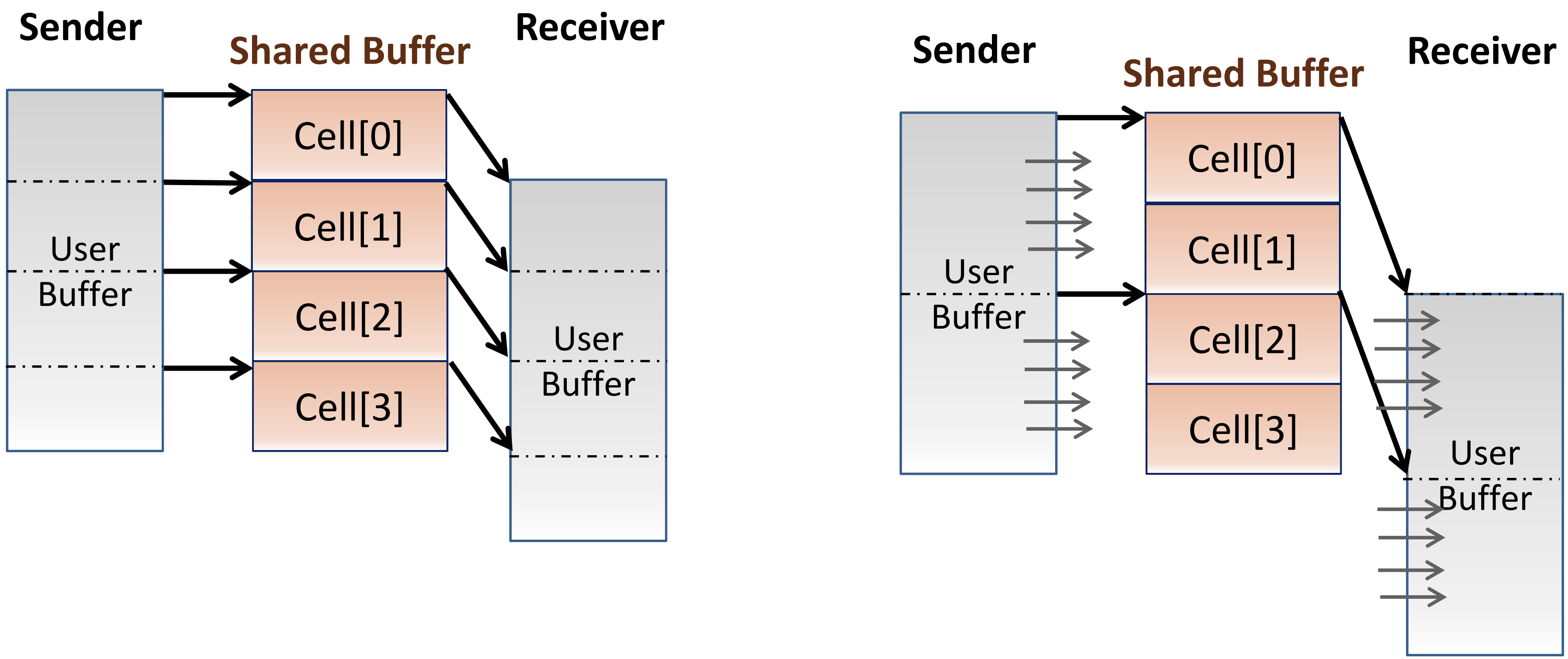
Sender
Get a EMTPY cell from shared buffer, and copies data into this cell, and marks the cell FULL; Then, fill next cell.

Receiver
Get a FULL cell from shared buffer, then copies the data out, and marks the cell EMTPY ; Then, clear next cell.



Parallel pipelining LMT

- Get as many available cells as we can
- Parallelizing large data movement

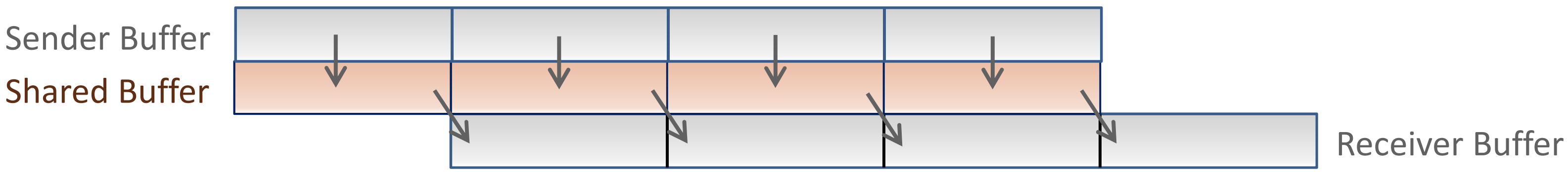


(a) Sequential Pipelining

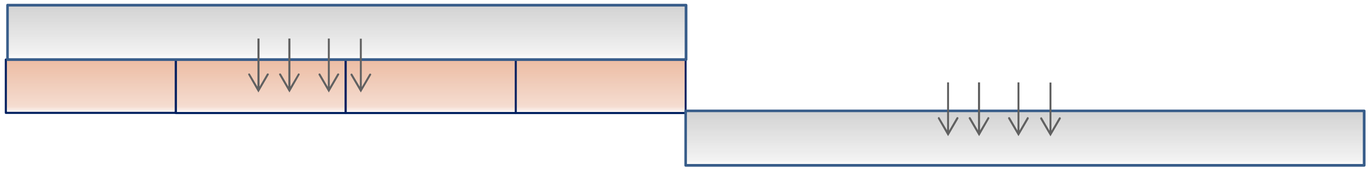
(b) Parallel pipelining

Sequential Pipelining VS Parallelism

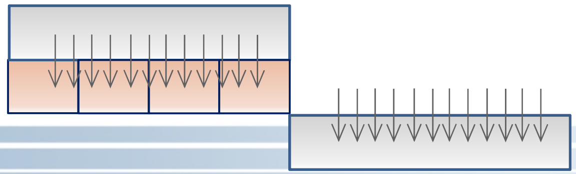
- Small Data transferring (< 128K)
  - Threads synchronization overhead > parallel improvement
- Large Data transferring
  - Data transferred using Sequential Fine-Grained Pipelining



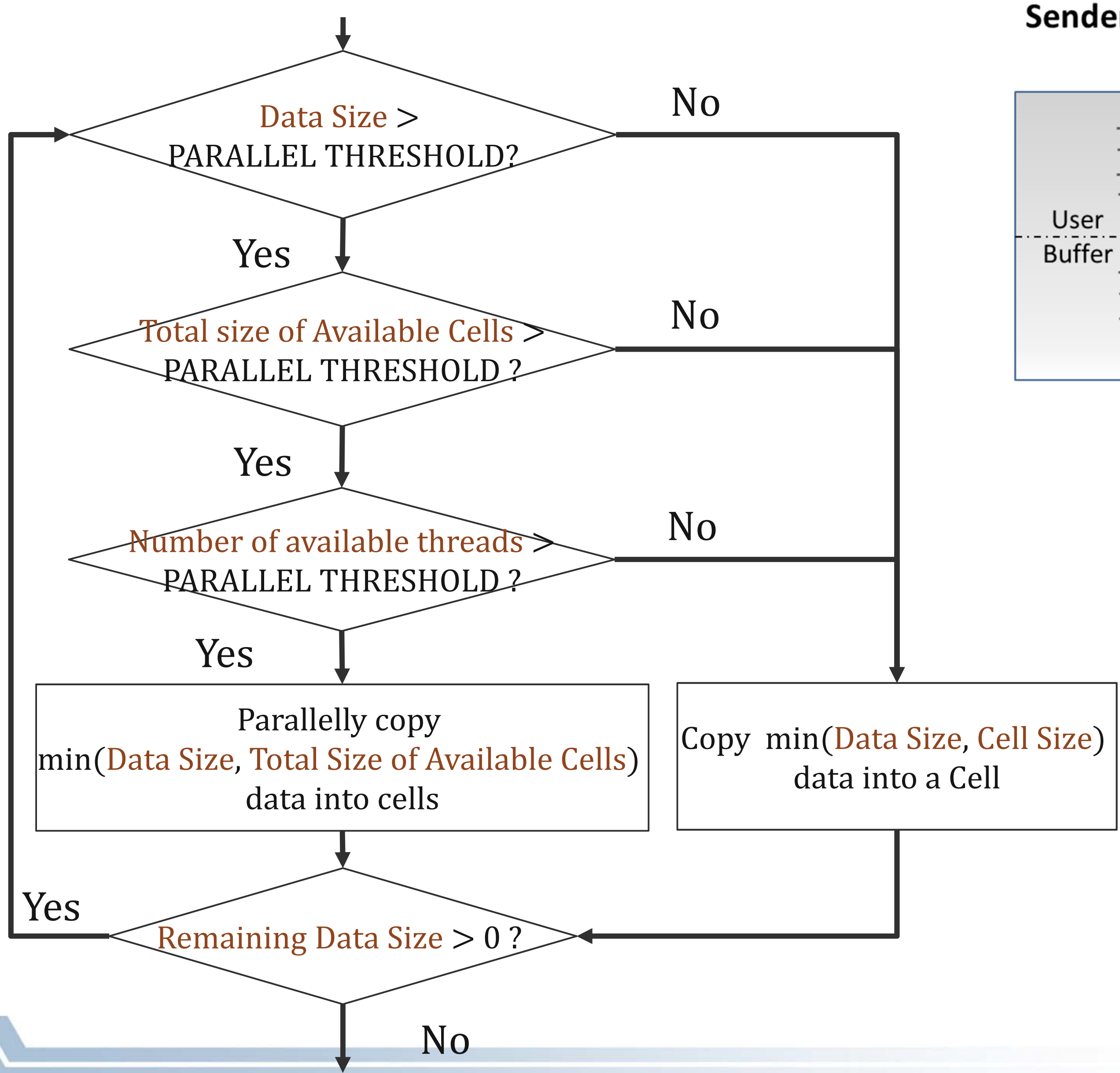
  - Data transferred using Parallelism with only a few of threads (worse)

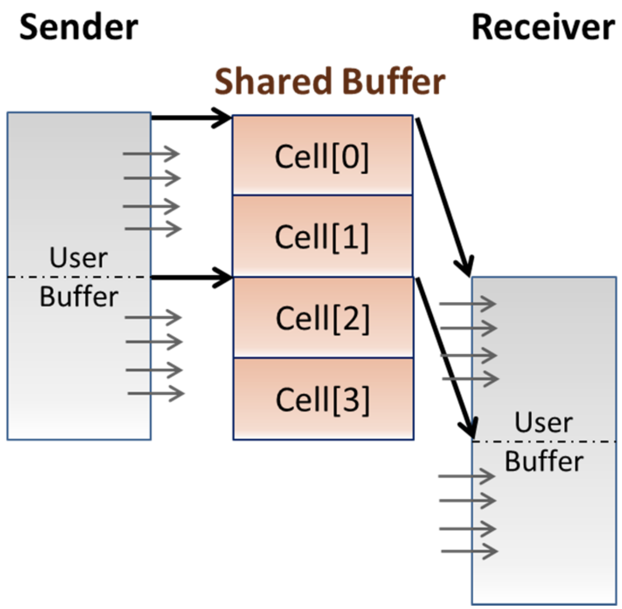


  - Data transferred using Parallelism with many threads (better)



Parallel pipelining LMT algorism



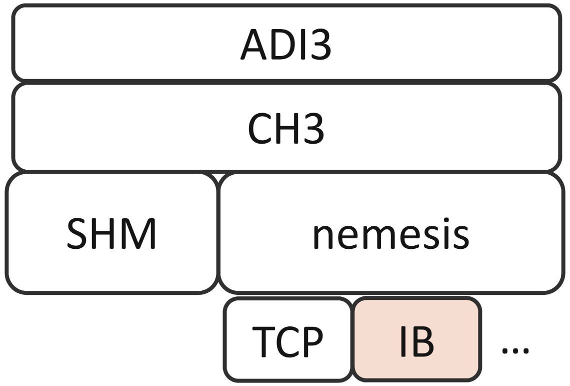


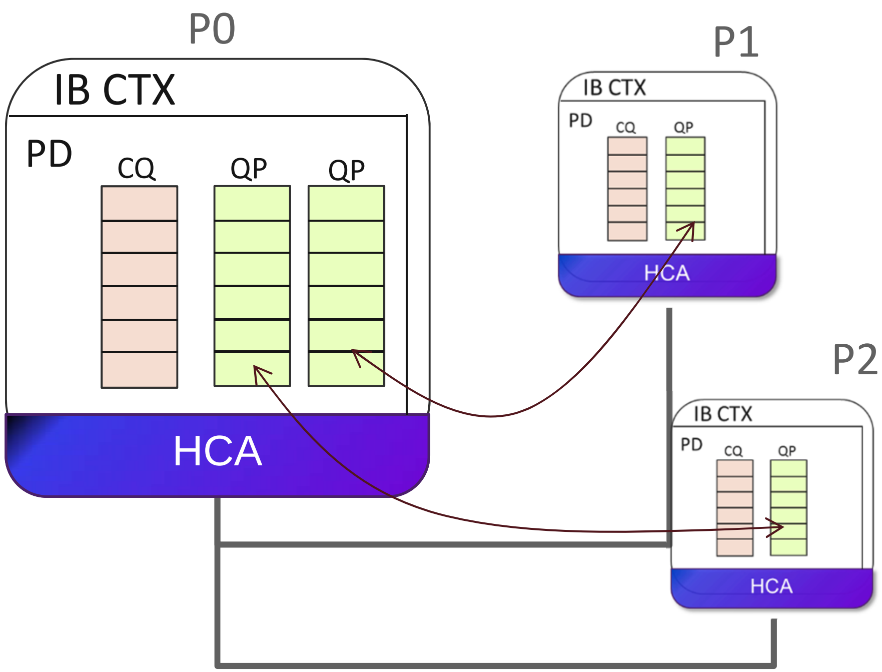
Outline

- Motivation
- Problem Statement and Solution
- Design and Implementation
  - OpenMP runtime
  - MPI Internal Parallelism
    - Datatype Related Functions
    - Intra-node Large Message Communication
    - Netmod Optimizations
- Evaluation
- Conclusion

InfiniBand Communication

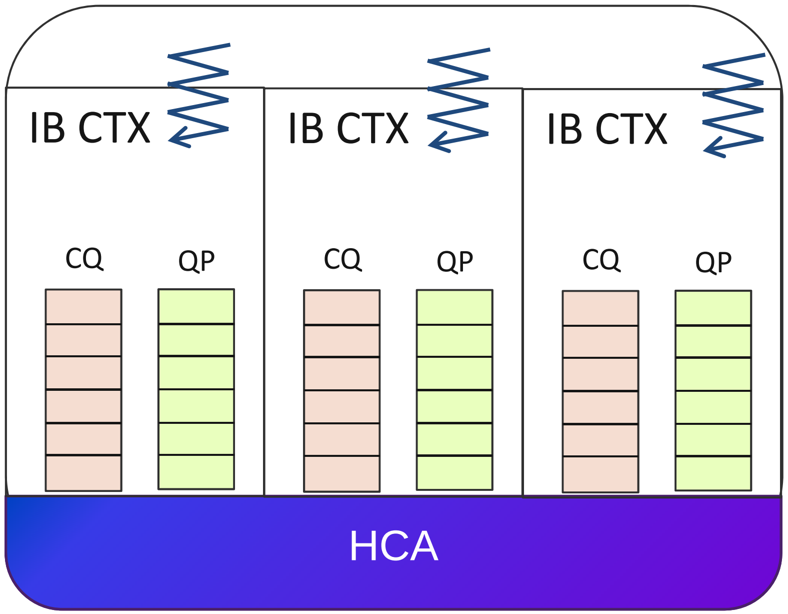
- Structures
  - IB context
  - Protection Domain
  - Queue Pair (critical)
    - 1 QP per connection
  - Completion Queue (critical)
    - Shared by 1 or more QPs
- RDMA communication
  - Post RDMA operation to QP
  - Poll completion from CQ
- Internally supports Multi-threading
- OpenMP contention issue



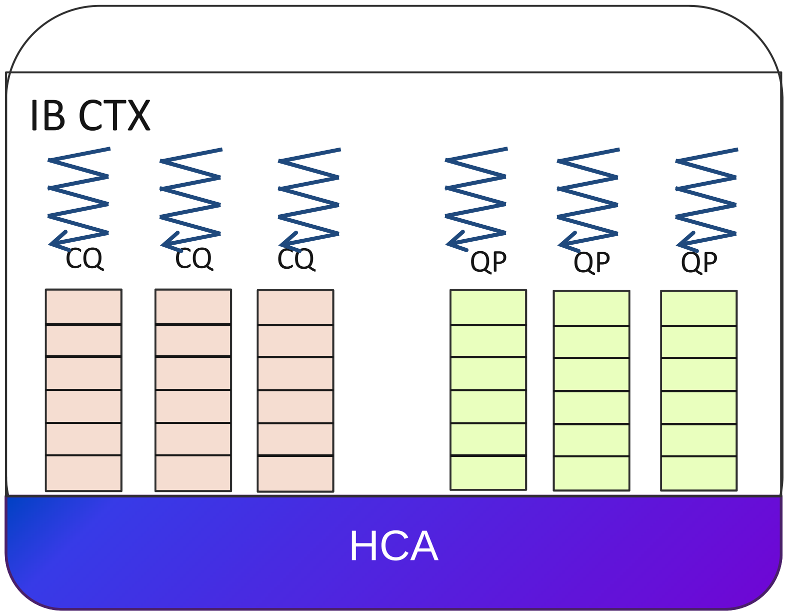


Parallel InfiniBand communication

- Two level parallel policies
  - Parallelize the operations to different IB CTXs
  - Parallelize the operations to different CQs / QPs



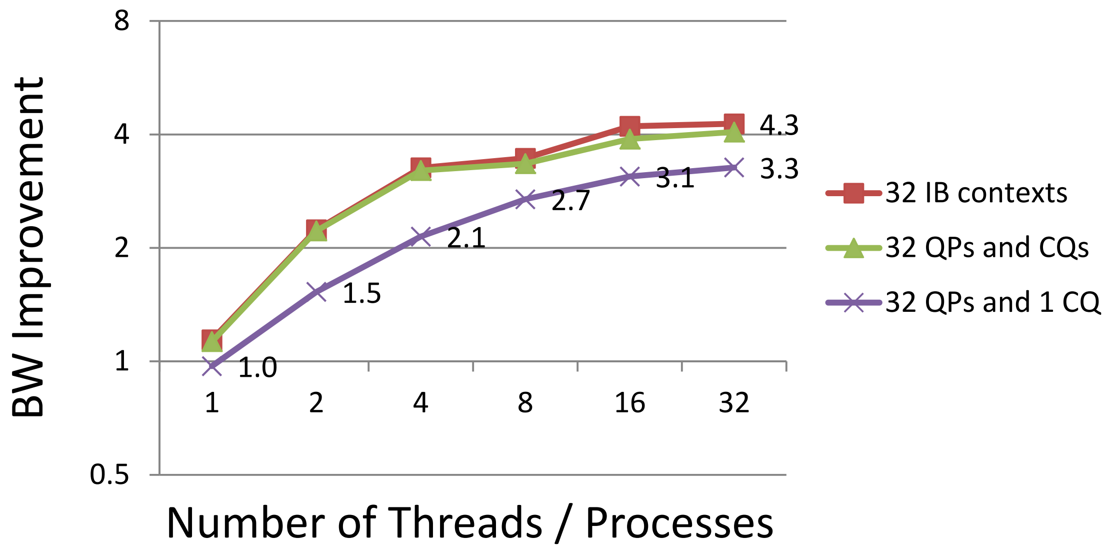
(a) Parallelism on different IB CTXs



(b) Parallelism on different CQs / QPs

Parallelize InfiniBand Small Data Transfer

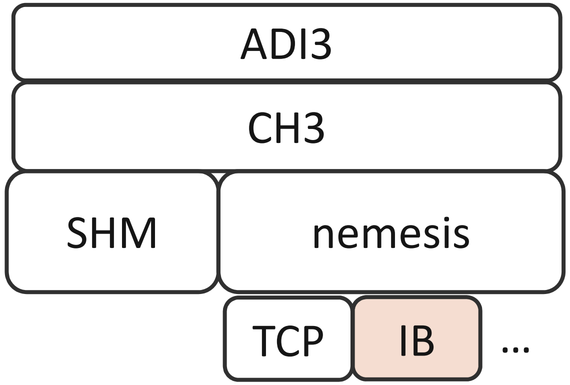
- 3 parallelism experiments based on ib_write_bw:
  1. 1 process per node, 32 IB CTX per process, 1 QP + 1 CQ per IB CTX
  2. 1 process per node, 1 IB CTX per process, 32 QPs + 32 CQs per IB CTX
  3. 1 process per node, 1 IB CTX per process, 32 QPs + 1 shared CQ per IB CTX

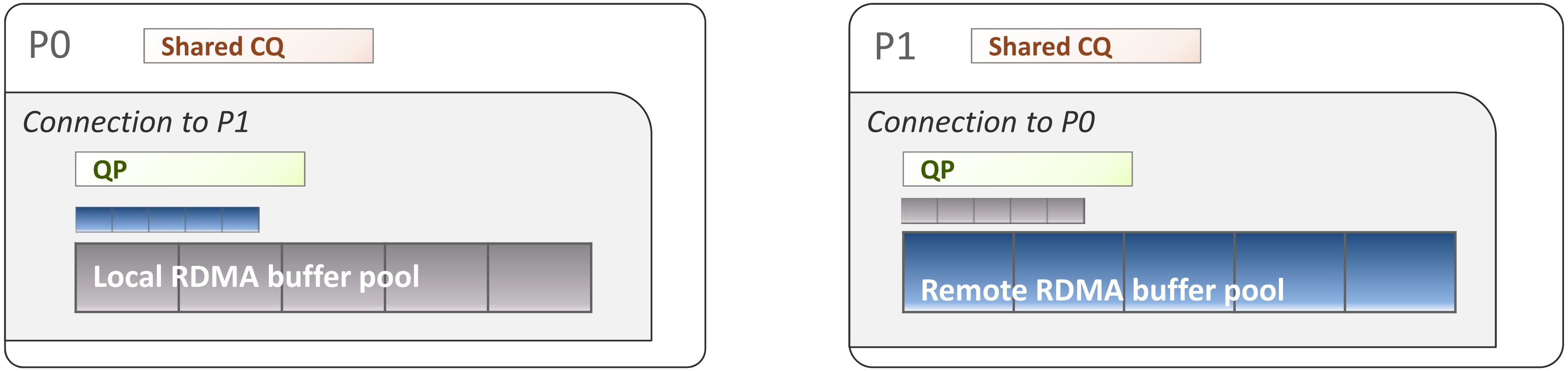


Test bed: Intel Xeon Phi cards (stepping B1, 60 cores), InfiniBand QDR
Data size: 2 Bytes

MPI IB netmod

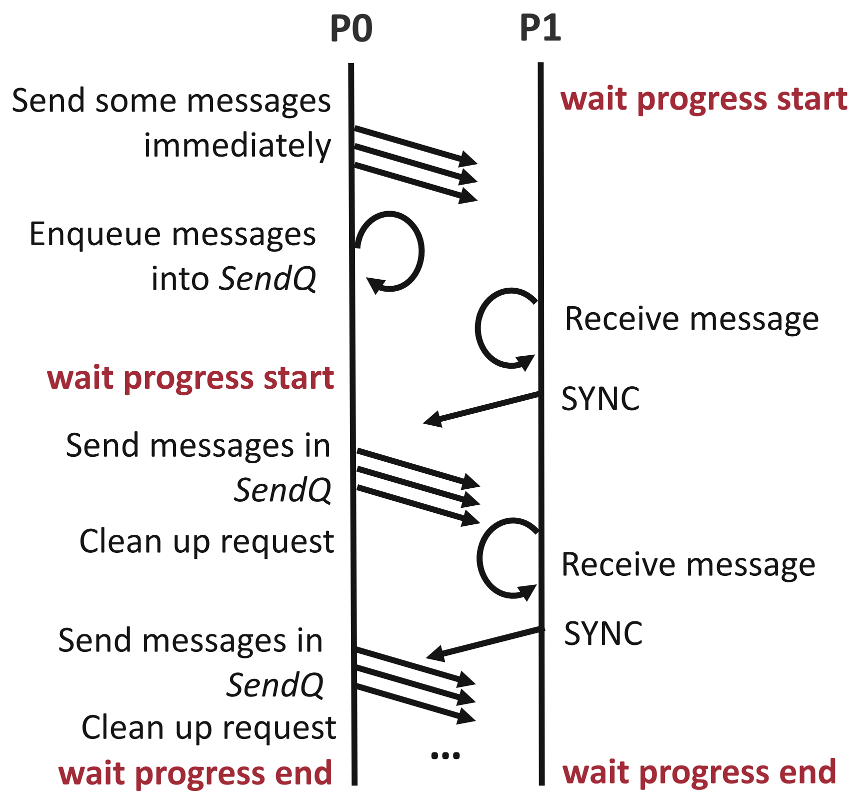
- Basic components
  - Local RDMA buffer pool
  - Remote RDMA buffer pool





Eager Message Transferring in IB netmod

- When send many small messages
  - Limited IB resources
    - QP, CQ, remote RDMA buffer
  - Most of the messages are enqueued into *SendQ*
  - All *sendQ* messages are sent out in wait progress
- Major steps in wait progress
  - Clean up issued requests
  - Receiving *Parallelizable*
    - Poll RDMA-buffer
    - Copy received messages out
  - Sending *Parallelizable*
    - Copy sending messages from user buffer
    - Issue RDMA op

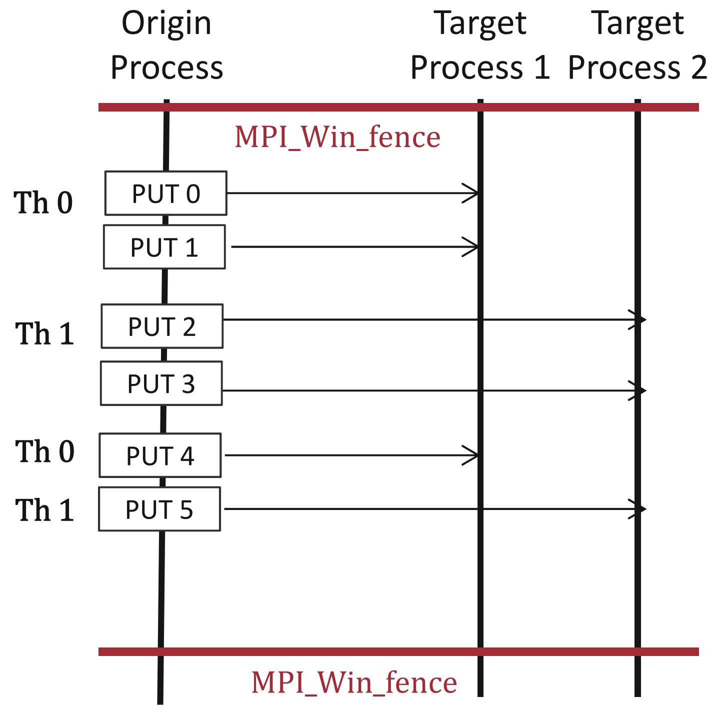


Parallel Eager protocol in IB netmod

- Parallel policy
  - Parallelize large set of messages sending to different connections
    - Different QPs : Sending processing
      - Copy sending messages from user buffer
      - Issue RDMA op
    - Different RDMA buffers : Receiving processing
      - Poll RDMA-buffer
      - Copy received messages out

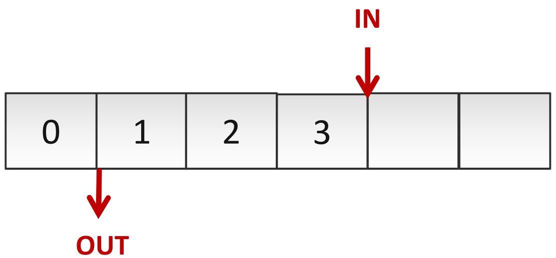
Target Applications: One-sided Communication

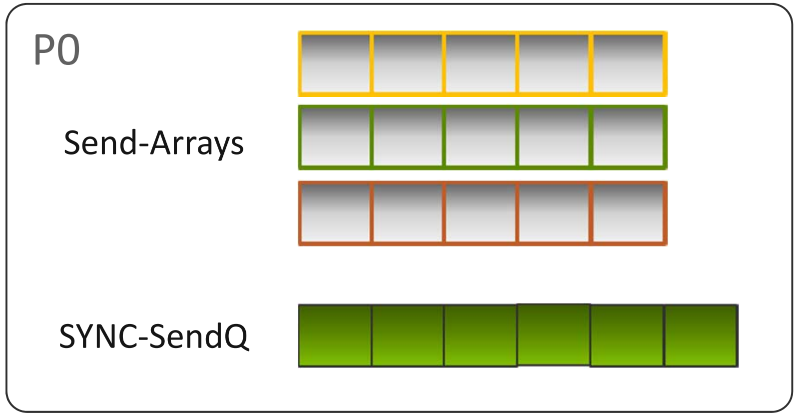
- Feature
  - Large amount of small non-blocking RMA operations sending to many targets
  - Wait ALL the completion at the second synchronization call (MPI_Win_fence)
- MPICH implementation
  - Issue all the operations in the second synchronization call



Parallel One-sided communication

- Challenges in parallelism
  - Global SendQ
    - Group messages by targets
  - Queue structures
    - Stored in [Ring Array + Head / Tail ptr]





  - Netmod internal messages (SYNC etc.)
    - Enqueue to another SendQ (SYNC-SendQ)

Parallel One-sided communication

- Optimization
  - Every OP is issued through long critical section
    - Issue all Ops together
  - Create large number of Requests
    - Only create one request

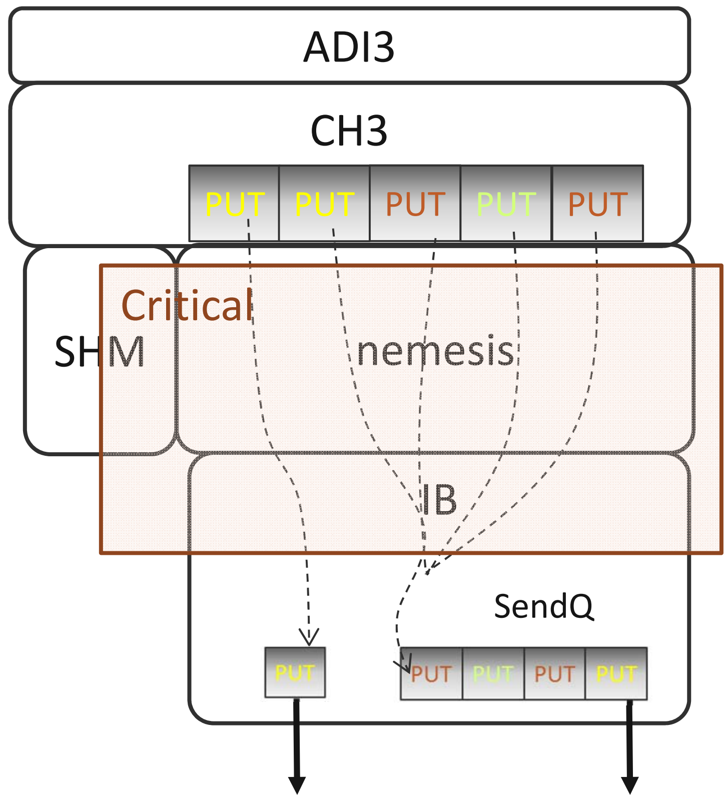
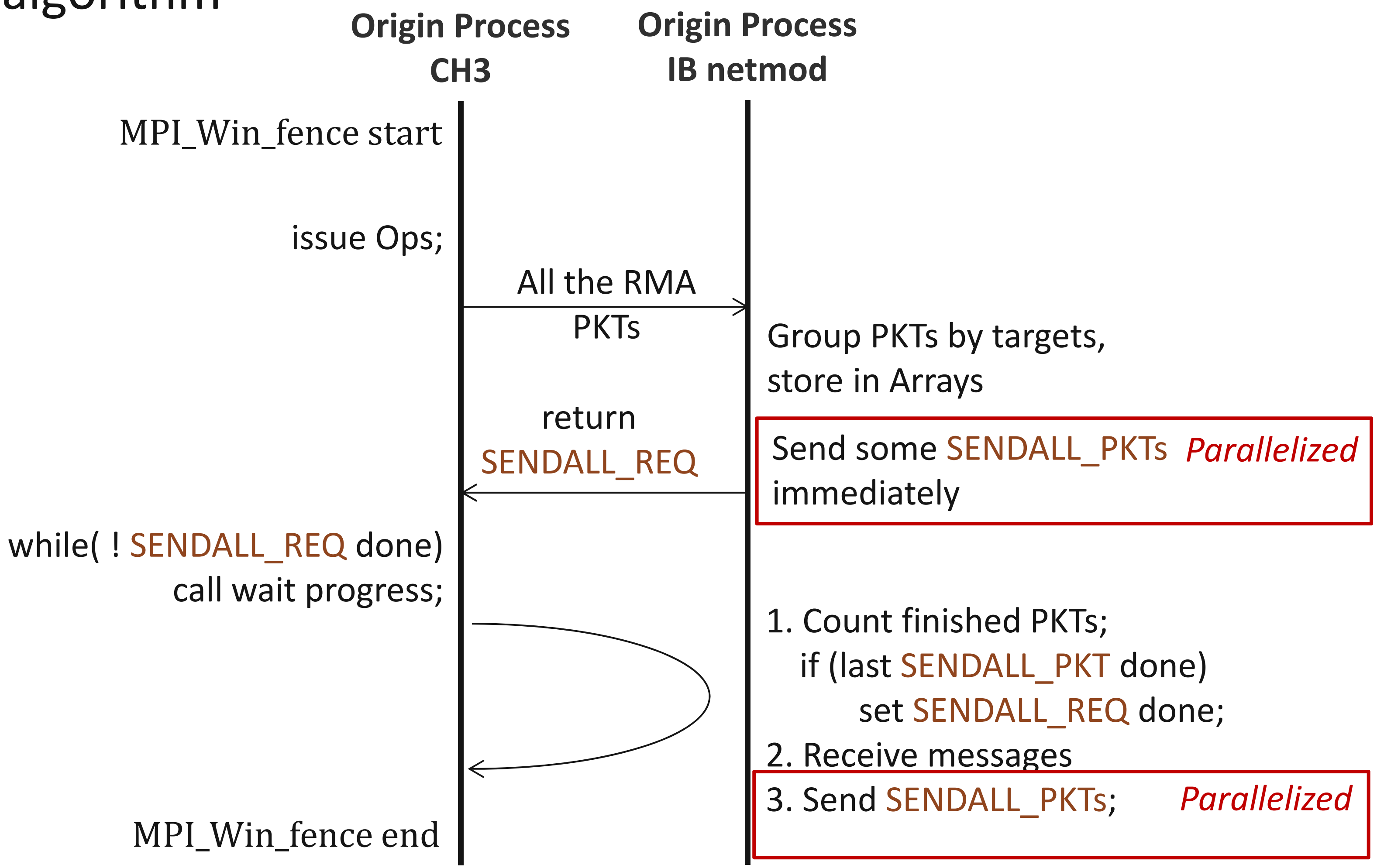


Fig. Issue RMA operations from CH3 to IB netmod

Parallel One-sided communication

- Final algorithm



Outline

- Motivation
- Problem Statement and Solution
- Design and Implementation
- Evaluation
- Conclusion

Experimental settings

- Knight (only for inter-card experiments)
  - Single node
  - Intel Xeon X5680 CPU, 20 GB of main memory
  - Two Intel Xeon Phi Knights Corner cards (stepping B0, 61 cores) featuring 8 GB of GDDR5 RAM (5.5 GT/s)
- KNCC
  - A cluster composed of 8 compute nodes
  - Dual Intel Xeon E5-2670 CPUs, 64 GB of RAM
  - Single Knights Corner card (stepping B1, 60 cores) with 8 GB of on-board RAM (5.0 GT/s)

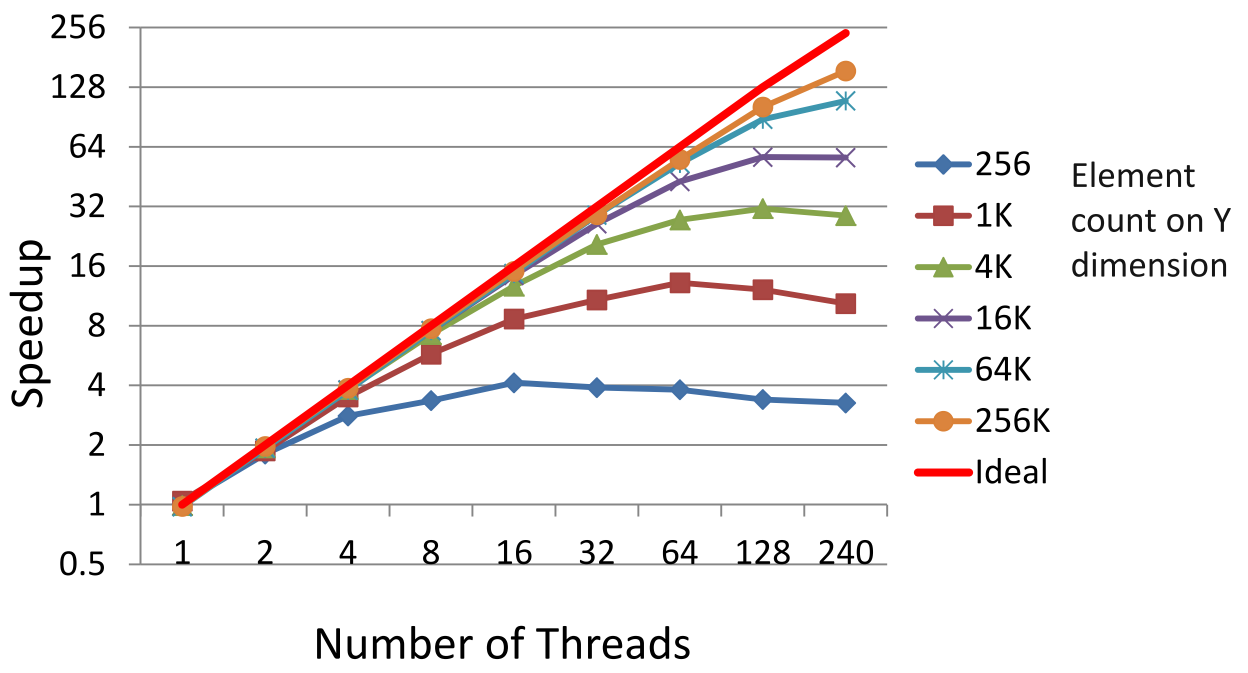
Evaluation: Datatype-related Functions

- Parallel 2D matrix packing
  - Fixed area size and varying X and Y dimensions
  - Element type : double

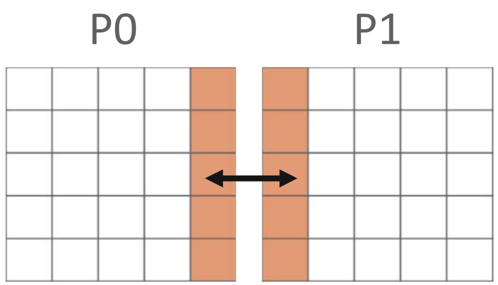
X

Y

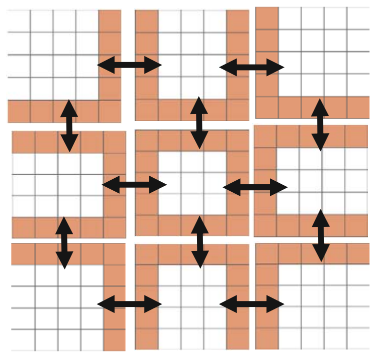
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24



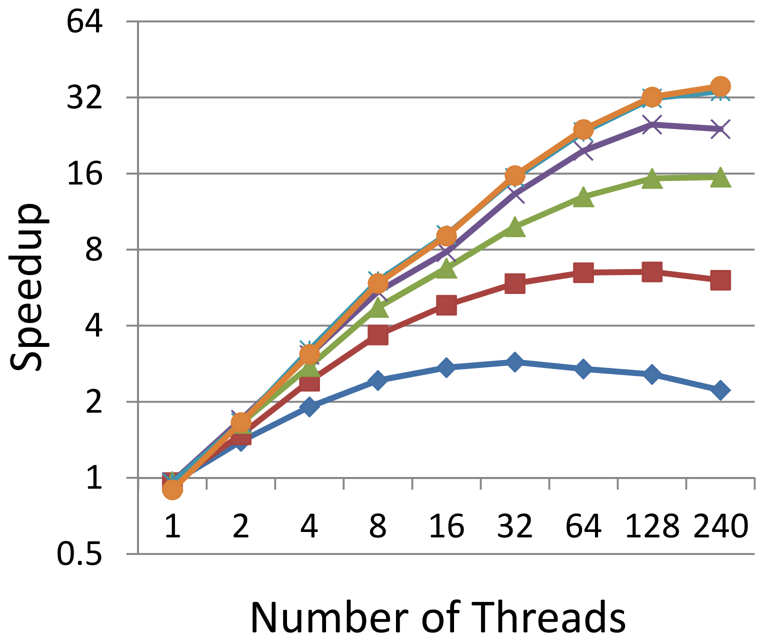
- Inter-node 2D Halo exchange

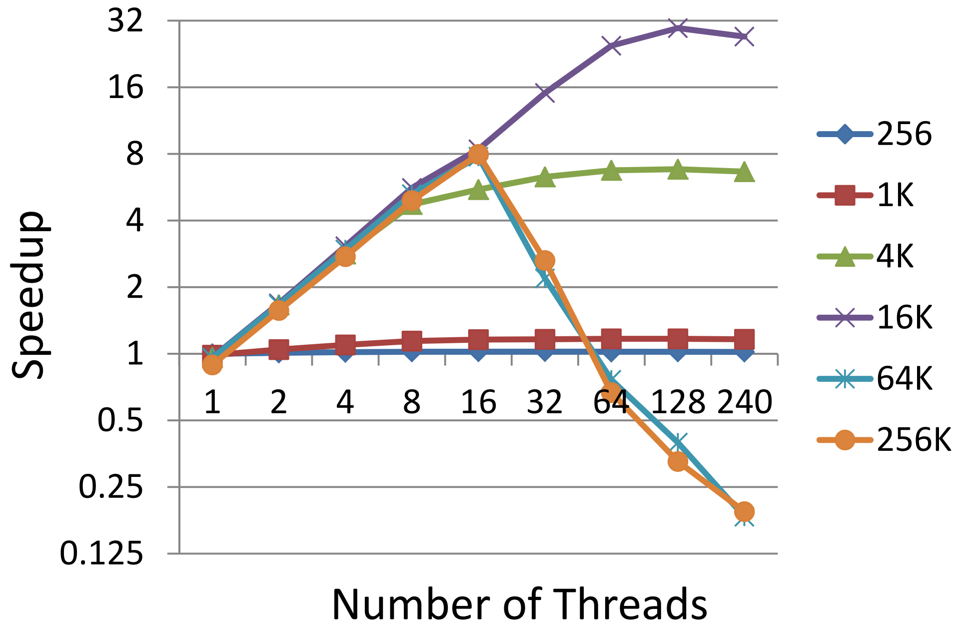


2 MPI processes on 2 nodes



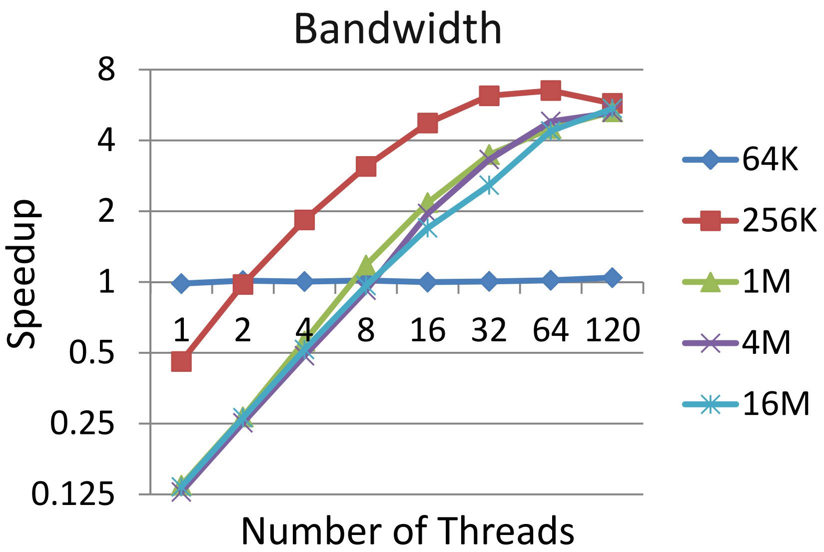
9 MPI processes on 9 nodes

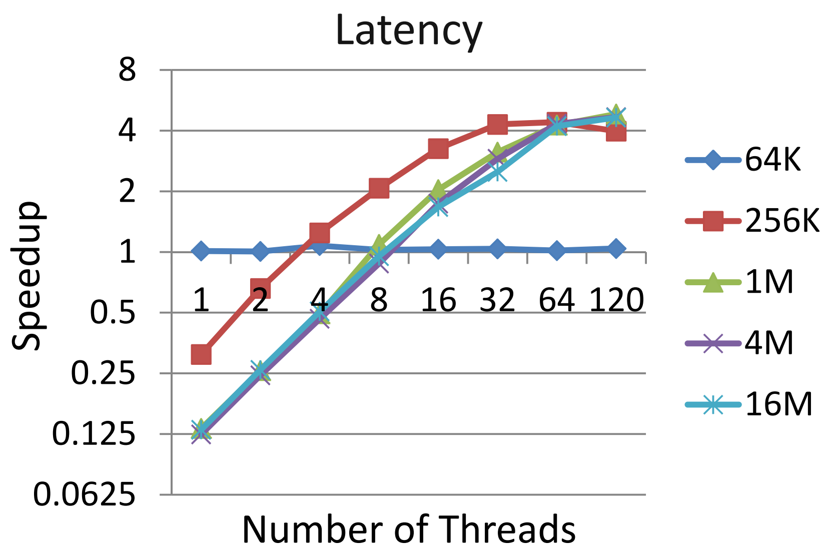


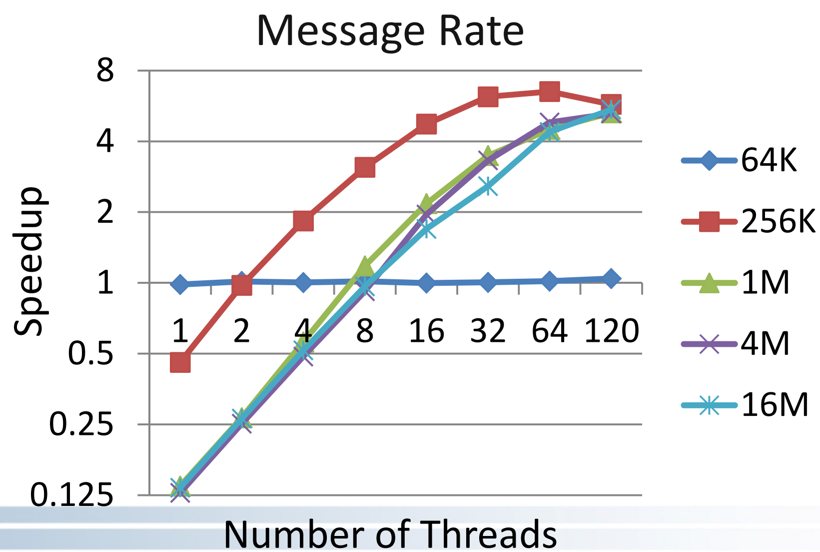


Intra-node Large Message Communication

- OSU P2P benchmark

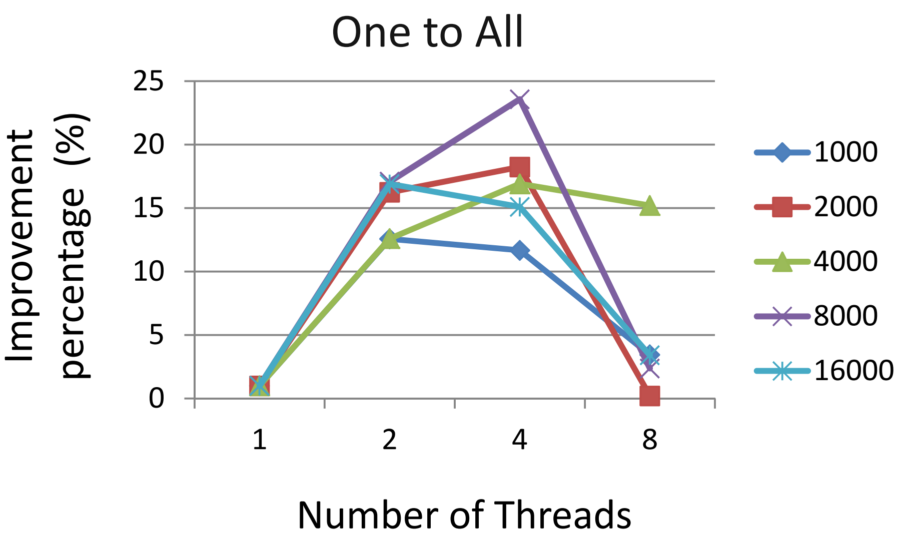


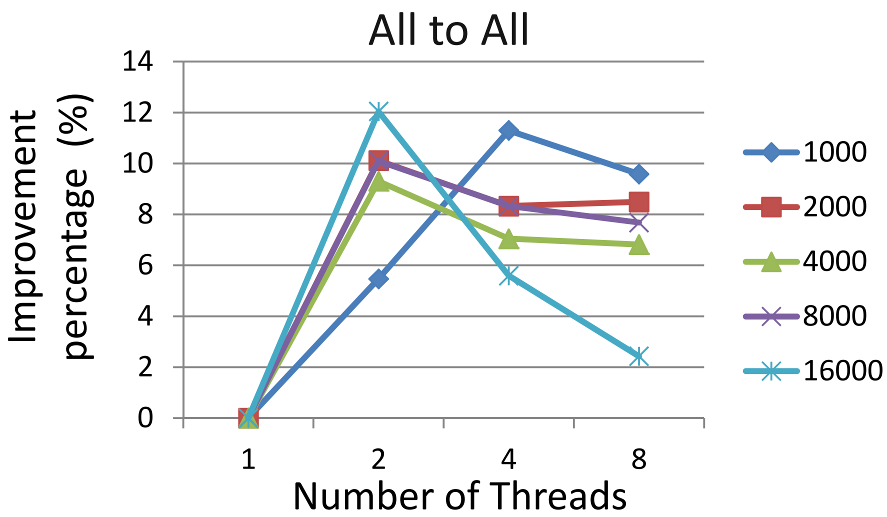




One-sided Operations and Low-Level Optimizations

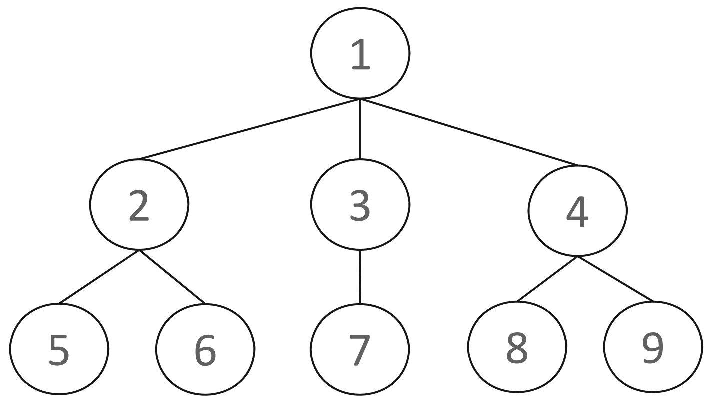
- Micro benchmark
  - One to All experiment using 9 processes
    - Process 0 sends many MPI_PUT operations to all the other processes
  - All to All experiment using 9 processes
    - Every process sends many MPI_PUT operations to the other processes

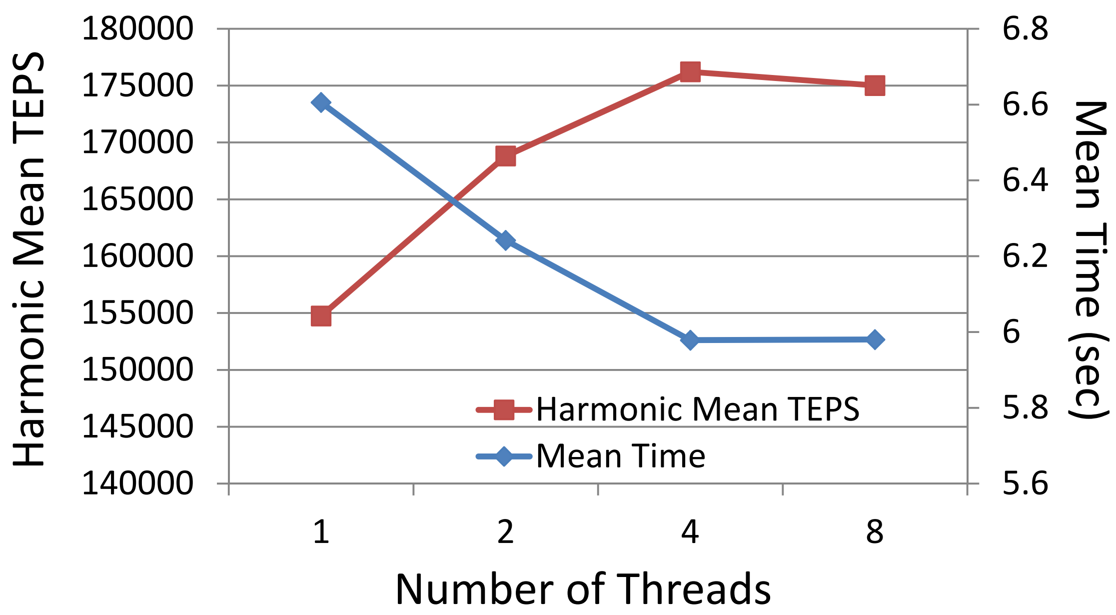




Graph500 benchmark

- Kernels
  - Graph Construction
  - Breadth-First Search (BFS)
    - MPI_Accumulate operations
  - Validation





Conclusions

- Hybrid OpenMP + MPI programming model is important for many-core environments
- User threads are idle during MPI calls in hybrid application, WASTE of computational resources
- MT-MPI internally shares IDLE threads with the application
- Various aspects of the MPI processing could be parallelized by multi-threads
  - ✓ Packing for derived datatype communication
  - ✓ Data movement for large shared memory communication
  - ✓ Network I/O for small message communication