Enabling Efficient Multithreaded MPI Communication Through a Library-Based Implementation of MPI Endpoints

James Dinan

Hybrid WG Plenary Session

December 9, 2014

Based on SC '14 paper by

Srinivas Sridharan, James Dinan, Dhiraj Kalamkar

Intel Corporation

intel

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2014 Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

Optimization Notice

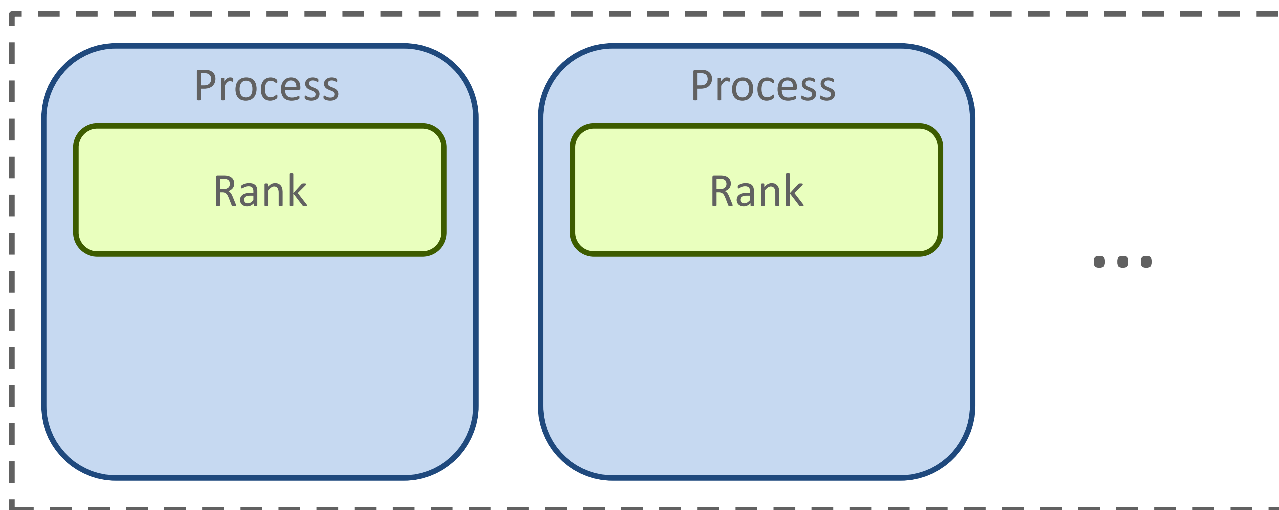
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Outline

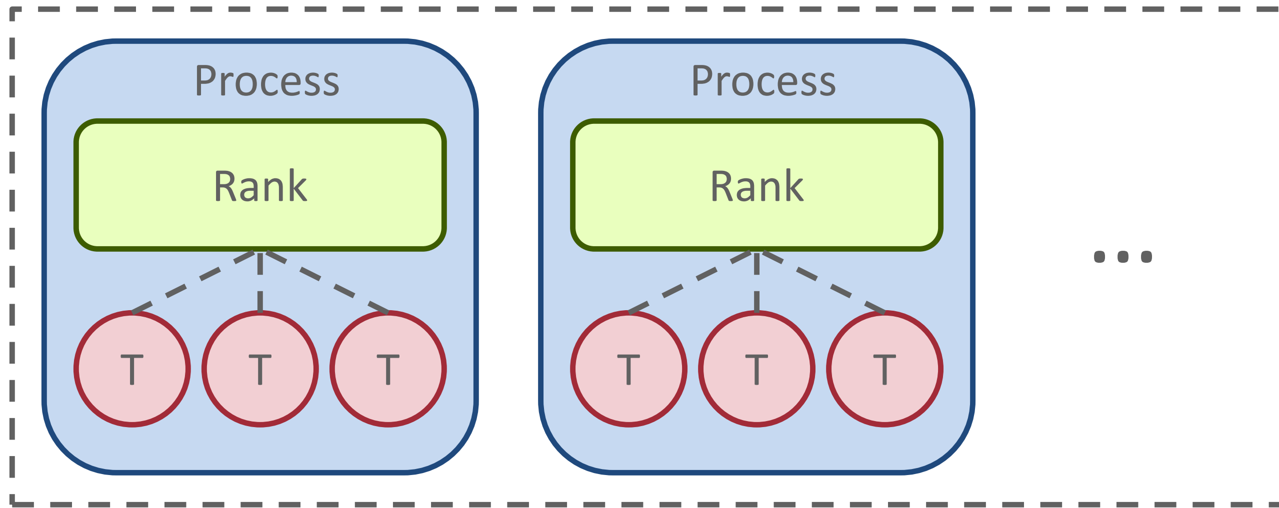
- Motivation for MPI Endpoints
- MPI Endpoints Library: Design and Implementation
- Experimental Results
- Wrap-up

Motivation



- MPI ranks have a 1-to-1 mapping with an OS process

Motivation

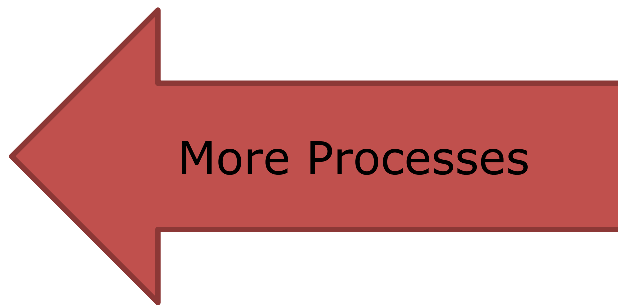


- MPI ranks have a 1-to-1 mapping with an OS process
- This was good in the past, but usage models have evolved
  - E.g. Hybrid parallel programming combining MPI and OpenMP
    - Need threads to act as first-class participants in MPI operations
    - Cannot isolate threads in MPI semantics (matching, ordering) and runtime

MPI+OpenMP: Process vs. Threads Tradeoff

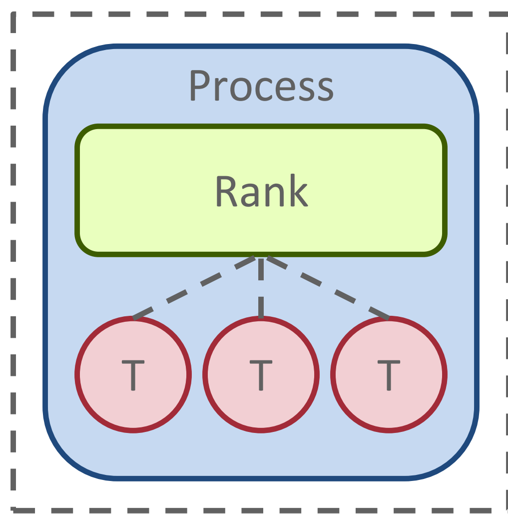
Communication throughput

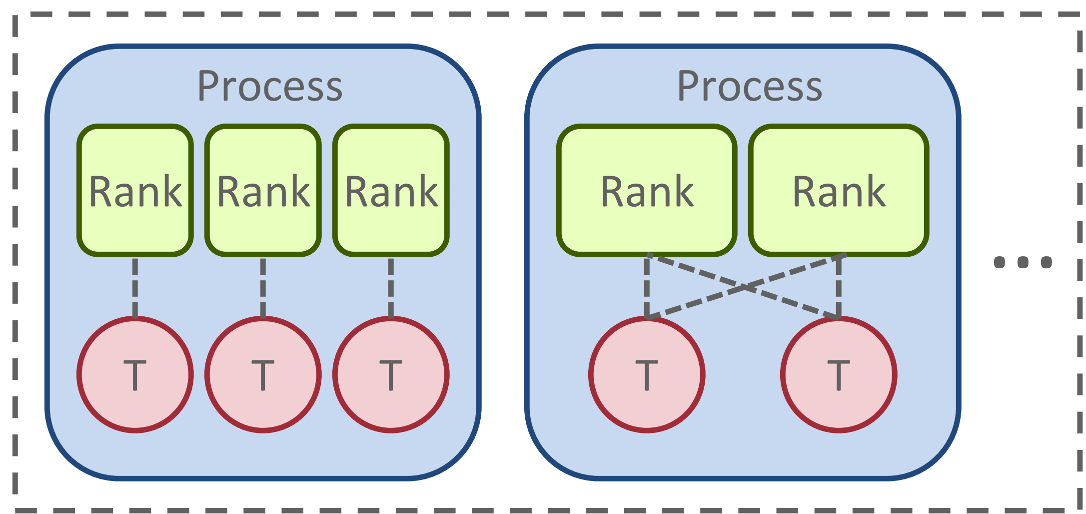
Reduce memory pressure
Better utilization of compute



- Users must make tradeoffs between number of processes per node and number of threads per process
  - Best choice depends on application behavior, system scale, MPI implementation, etc.

MPI Endpoints Proposal

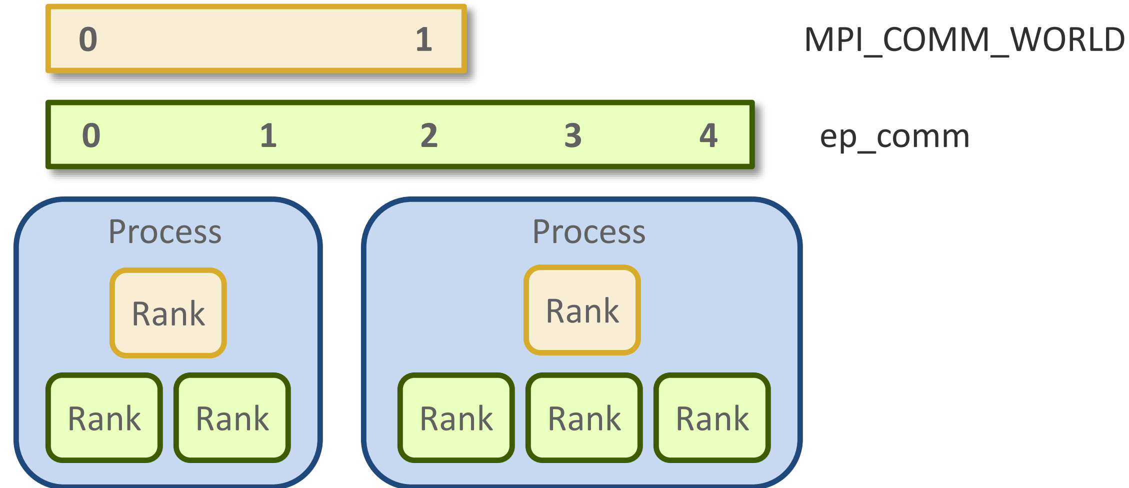




- A rank is an abstract entity representing a communication "endpoint"
  - Set of resources that supports the execution of MPI operations
- Proposal: Create new ranks from existing ranks in parent communicator to enable many-to-one mapping
  - Endpoint ranks behave like MPI processes (progress, matching, ordering rules)
  - Allocate per-thread messaging state and communication resources
    - Enable threads to achieve process-like communication performance
  - Improve interoperability and productivity of MPI+X

[1] Dinan et al., Enabling Communication Concurrency Through Flexible MPI Endpoints, IJHPCA Vol. 28, No. 4, Nov. 2014.

MPI Endpoints API




```
int MPI_Comm_create_endpoints(MPI_Comm parent_comm,
      int num_ep, MPI_Info info, MPI_Comm ep_comm[])
```

- Each rank in *parent_comm* gets *num_ep* ranks in *ep_comm*
  - *num_ep* can be different at each process
- Output is an array of communicator handles
  - Rank order: process 0's num_ep ranks, process 1's num_ep ranks, etc.
  - i^{th} handle corresponds to i^{th} endpoint rank
  - To use that endpoint, use the corresponding handle

Enabling OpenMP threads in MPI collectives

- Hybrid MPI+OpenMP code
- Endpoints are used to enable OpenMP threads to fully utilize MPI

```
int main(int argc, char **argv) {
    int world_rank, tl;
    int max_threads = omp_get_max_threads();
    MPI_Comm ep_comm[max_threads];

    MPI_Init_thread(&argc, &argv, MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

#pragma omp parallel
    {
        int nt = omp_get_num_threads();
        int tn = omp_get_thread_num();
        int ep_rank;
#pragma omp master
        {
            MPI_Comm_create_endpoints(MPI_COMM_WORLD,
                nt, MPI_INFO_NULL, ep_comm);
        }
#pragma omp barrier

        MPI_Comm_rank(ep_comm[tn], &ep_rank);
        ... // divide up work based on 'ep_rank'
        MPI_Allreduce(..., ep_comm[tn]);

        MPI_Comm_free(&ep_comm[tn]);
    }
    MPI_Finalize();
}
```

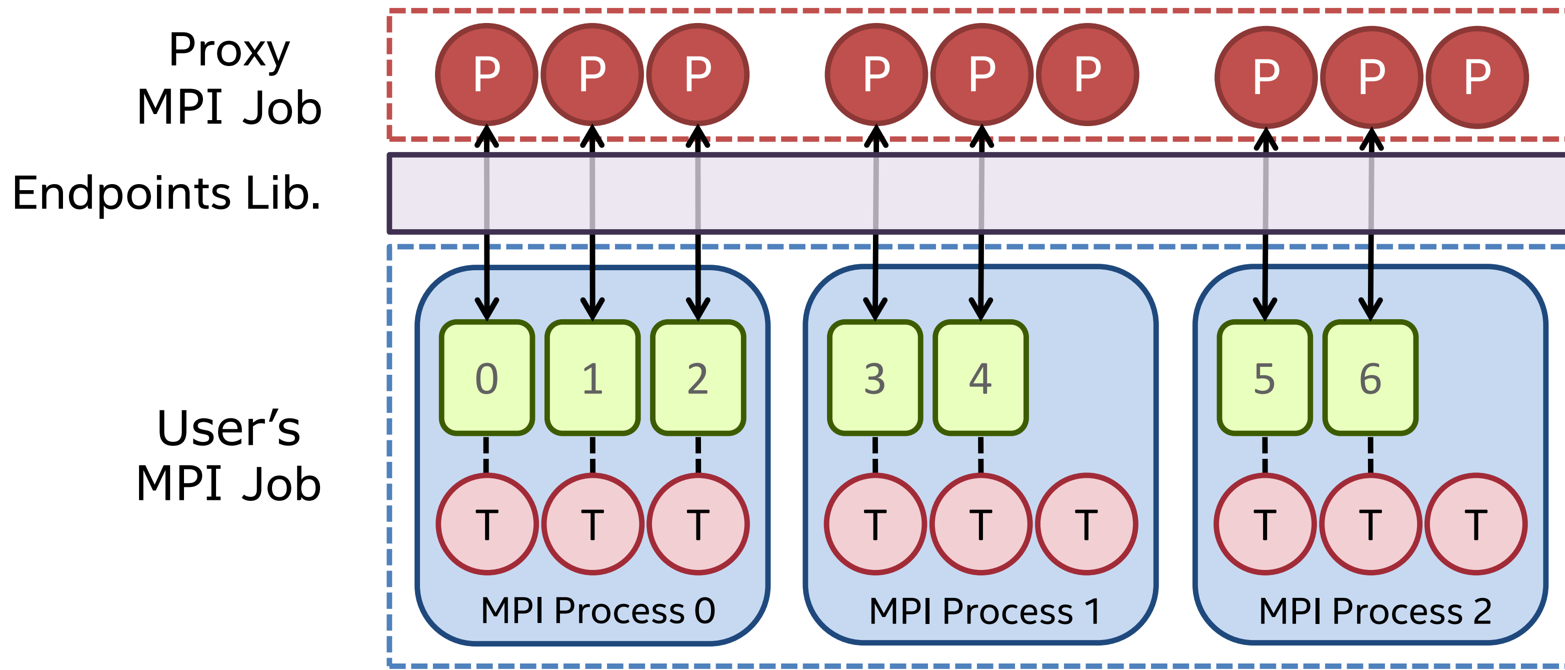
Outline

- Motivation for MPI Endpoints
- **MPI Endpoints Library: Design and Implementation**
- Experimental Results
- Wrap-up

Objective and Approach

- Objective:
  - Demonstrate the performance and programmability benefits of MPI endpoints
- Approach:
  - Default choice: natively within an MPI implementation
  - Our approach: as a library
    - Enables early exploration and performance study
    - Compatible with any existing MPI implementation

Design of Endpoints Library

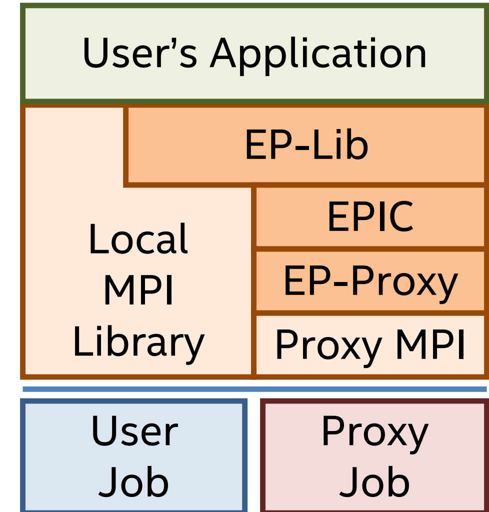


- Implement endpoint ranks using MPI processes
- Spawn a background MPI job
- Endpoints library forwards commands from user job to proxy job
- Proxy process performs MPI operation on behalf of user endpoint rank
- POSIX Shared memory coordination between user and proxy job

intel

Implementation Details

- Intercept MPI operations at PMPI interface
  - Operations on endpoints comm. passed to EP-Lib
  - Non-endpoints operations pass to local MPI
- Endpoints operations performed in proxy job
  - One proxy process per user endpoint
  - Proxy uses MPI library in single-threaded mode
    - Eliminates threading overheads



- POSIX Shared memory coordination between user and proxy job
  - Command queue: shared circular buffer
  - Send commands, receive completion (e.g. for nonblocking operations)
    - Proxy waits on command queue (default), or can drive async. progress
  - Message buffers are allocated in shared memory to avoid copies

Endpoints Communicator Creation

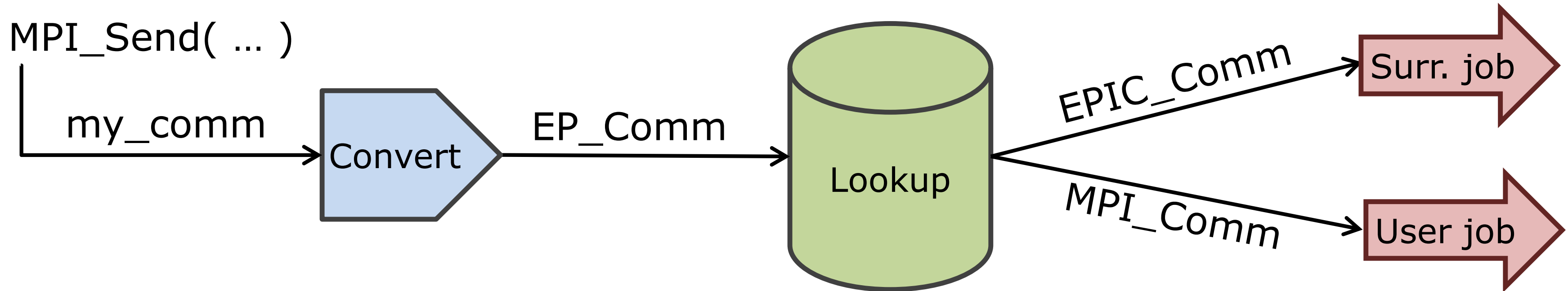
```
int MPI_Comm_create_endpoints(MPI_Comm parent_comm,
    int num_ep, MPI_Info info, MPI_Comm ep_comm[])
```

1. User context:
   - Determine total number of endpoints requested (MPI_allgather num_ep)
   - Establish thread-proxy connections, command queues
   - Issue communicator creation command to each proxy MPI process

2. Proxy context:
   - Translate list of proxy ranks into an MPI group
   - Call MPI_Comm_create_group on proxy global comm.
     - Called only by proxy ranks in new endpoints communicator
   - Each proxy returns a new communicator handle to EPIC

3. EPIC registers new communicator handles
   - Convert, aggregate, and return ep_comm handles to user

Management and Translation of MPI Objects



- We need to distinguish endpoint objects from non-endpoint objects
  - Route operations to user/proxy job using correct handle
  - Look up additional metadata needed to talk to proxy
- Two classes of MPI objects, managed by Endpoints library
  - Singleton objects: Communicator handles, non-blocking request handles
    - Exist either in user job or proxy, but not both
  - Replicated objects: Groups
    - Exist in both jobs, must be created/updated/freed in both
- Create dictionaries to translate handles, e.g. for communicators:
  - All handles are registered and a new EP_Comm handle is returned to application
  - Endpoints library translates handles for usage in user/proxy context

Advantages of Library approach

- Satisfies progress, matching, ordering rules
- Provide immediate access to benefits of endpoints
  - Enable threads to achieve process-like comm. performance
  - Enable early exploration and performance studies
  - Compatible with existing, highly tuned production MPI libraries
- Overcome thread-safety limitations and overheads in networking stack
  - Proxy communication technique achieves multiple private network instances within a shared memory process

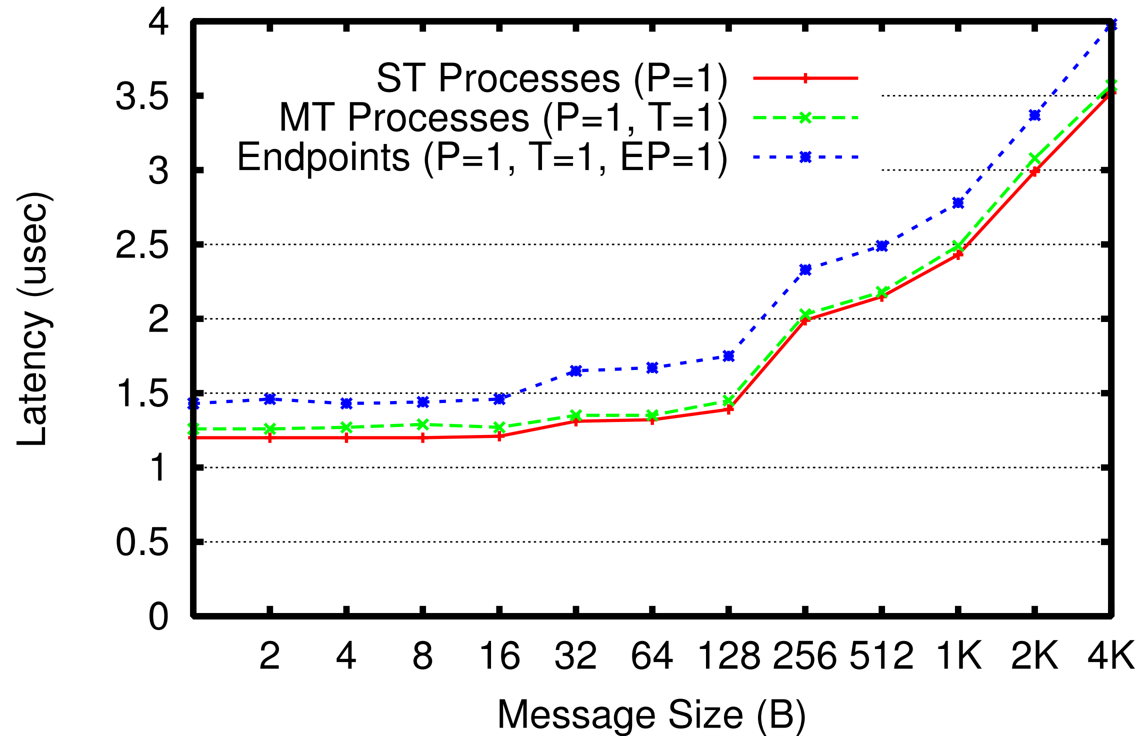
Outline

- Motivation for MPI Endpoints
- MPI Endpoints Library: Design and Implementation
- **Experimental Results**
- Wrap-up

Experimental Evaluation

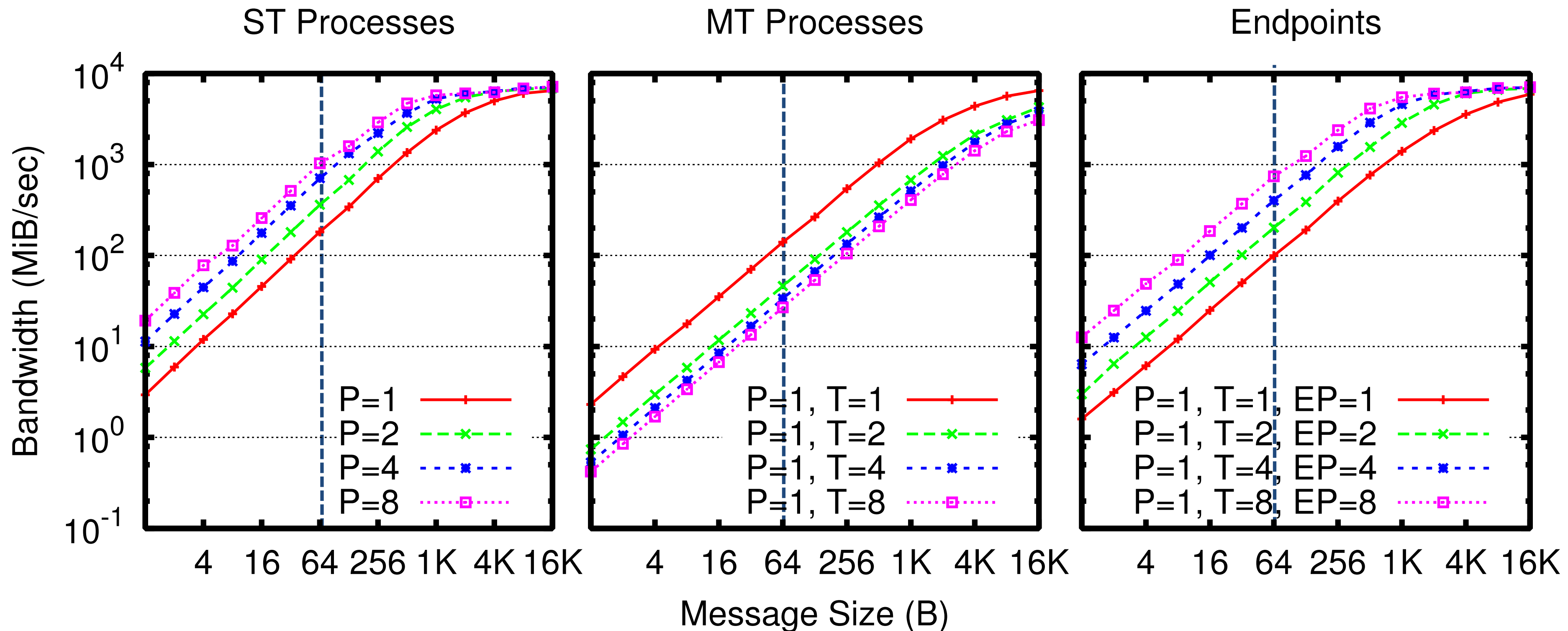
- Intel Endeavor cluster:
  - Node: 2x 12-core 2.7 GHz Intel® Xeon® E5-2697
    - Two threads per core, hyperthreading enabled
  - Fabric: Mellanox* InfiniBand* FDR, 2-level fat-tree
  - Intel® MPI Library v4.1.3, no modifications
- Highlights:
  - Latency – Fixed ~320ns overhead per operation
  - Throughput – At 64B, EP achieves 72% of ideal
  - FFT – More than 2x improvement up to 4kB messages
  - Lattice QCD – 1.87x improvement on 128 processes

Measurement of Overhead



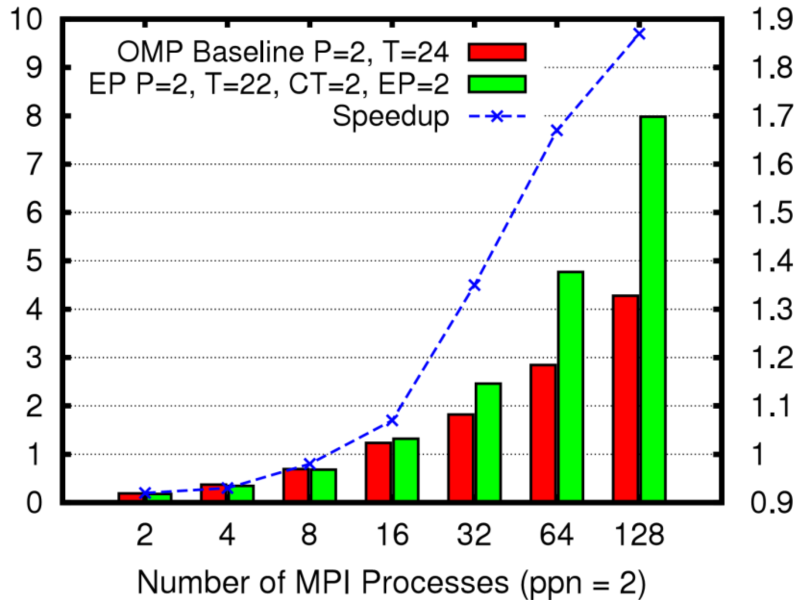
- Ping-pong benchmark, half round-trip latency
- Fixed ~320ns overhead incurred by EP-Lib
  - Cost of object translation and sending command to proxy
  - Less than synchronization overhead to MT case

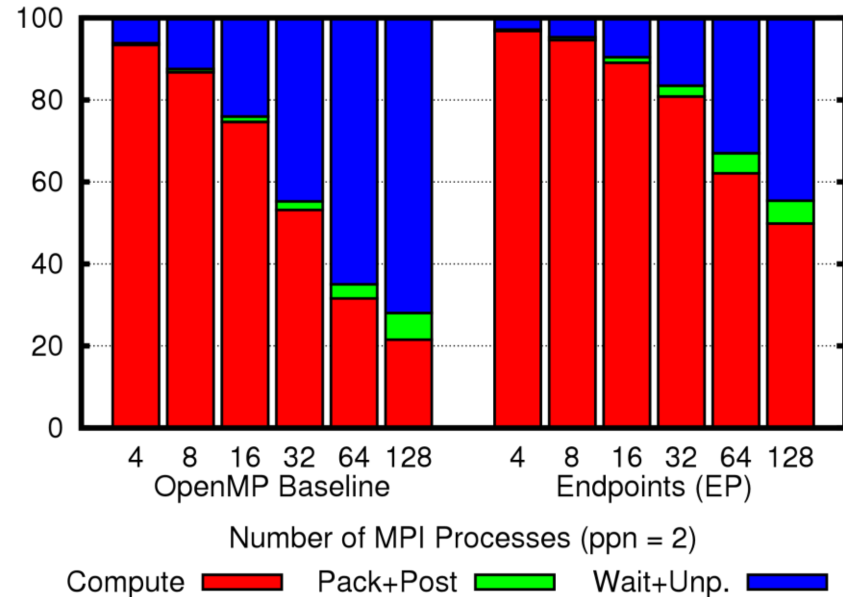
Impact on Throughput



- Single threaded (ST), multithreaded (MT), and endpoints cases
  - Two nodes, increase number of process, therads, or endpoints per node from 1 to 8
  - Same amount of resources in each case, vary how they are used
- Uni-directional BW comparison at 64B messages, using 8 cores
  - ST = 1029 (100%); MT = 27 (2.6%); EP = 742 (72%) MiB/sec

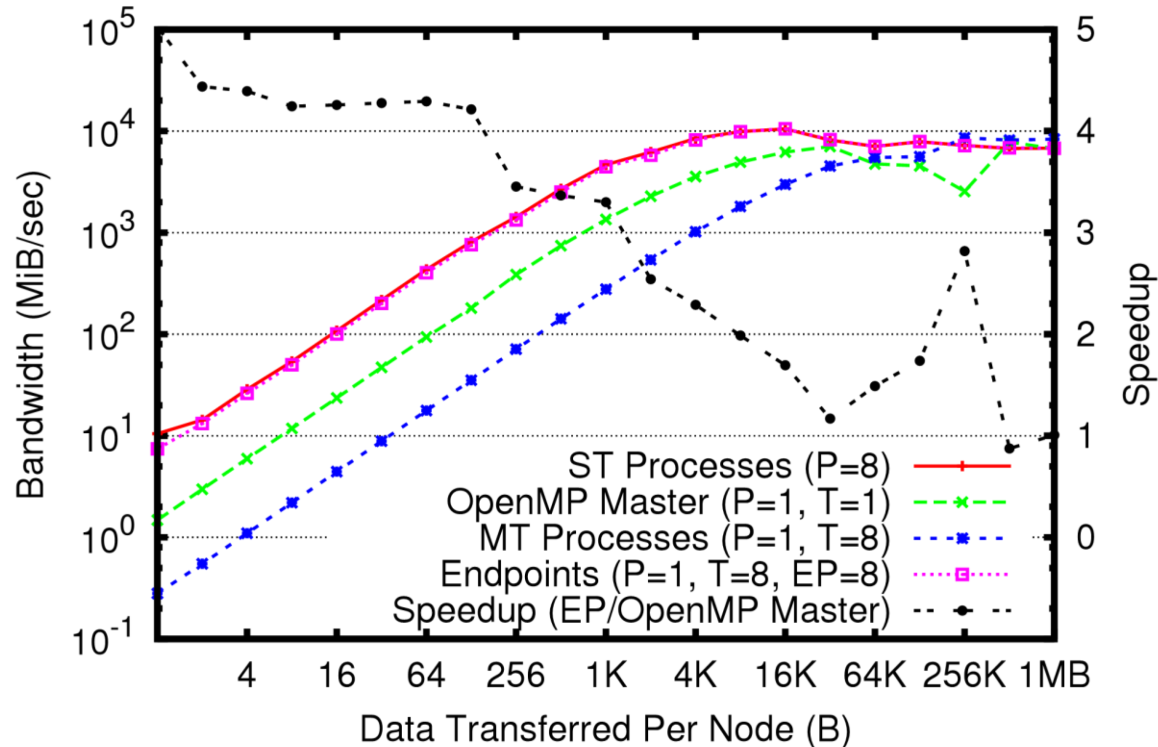
Lattice QCD Dslash Kernel





- Wilson Dslash operator from high energy physics
  - 4D decomposition with 2 neighbors in each direction
  - Halo exchange with at most 8 neighbors
- Strong scaling study, performance (left) and breakdown (right)
  - Baseline: 24 compute threads per process
  - EP: 22 threads, 2 used as proxies (2.9x better comm., 1.87x total)

FFT Performance



- FFT Exchange (all-to-all) communication benchmark on 32 nodes, ppn=1
  - Fixed volume of data, performance is dependent on throughput
- Compare single threaded (OMP Master), MT, and MT plus endpoints
  - Speedup ~3x for small messages, ~2x for medium, converges for large
  - Significant advantage over conventional OpenMP Master communicates approach

Wrap-up

- MPI+X models growing in importance with many-core
  - Multithreaded processes must be treated as first-class model
  - Threads must communicate to achieve high throughput
  - Enable by disentangling threads in semantics and mechanics

- Implemented MPI endpoints extension as a library
  - Enables early exploration and performance study
  - Compatible with any existing MPI implementation

- Endpoints improve comm. throughput for MT processes
  - Significant gains, in spite of EP-lib overheads
    - Overheads will be reduced in a native implementation
  - Tune comm. performance without changing number of processes

Thank You and Acknowledgments!

- We thank the *many* members of the MPI community and MPI forum who contributed to the MPI Endpoints Extension!

- Review the formal proposal:
  - https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/380
  - Proposal status: Entering the voting process for MPI 4.0

- Contact MPI Forum's hybrid working group

- Contact authors:

  - Srinivas Sridharan, srinivas.sridharan@intel.com
  - James Dinan, james.dinan@intel.com