Optimize and perform with Intel® MPI

Roger Philp

Intel HPC Software Workshop Series 2016
HPC Code Modernization for Intel® Xeon and Xeon Phi™
February 17th 2016, Barcelona
Agenda

- Levels of parallelism
- What is MPI – Message Passing Interface?
- MPI a little bit of history
- What does an MPI program look like?
- Why do we need MPI-3?
- Review important MPI 3.0 additions
- MPI-3 Non-blocking Collectives (NBC)
- MPI-3 Neighborhood Collectives (NC)
- One-sided Communications (a.k.a Remote Memory Access (RMA))
- mpitune
What Levels of Parallelism exist?

- **ILP - Instruction Level Parallelism**
  - Pipelined Execution
  - Super-scalar execution

- **DLP - Data Level Parallelism**
  - SIMD (Single Instruction Multiple Data) vector processing
  - Implemented via vector registers and instructions

- **TLP - Thread-Level Parallelism**
  - Hardware support for hyper-threading
  - Multi-core architecture
  - Cache-coherent multiple sockets

- **CLP - Cluster Level Parallelism**
  - Multiple platforms connected via interconnection network
  - No hardware-supported cache coherence
What is MPI – Message Passing Interface?

MPI environment

Box 1

Box 2

Interconnect /Fabric

Message is packed and sent to target

Message is received and unpacked and put into place

OS: Unix, Linux, Windows
Shared disk space
Disparate disk space
Message Passing Interface (MPI) – towards millions of cores

- MPI is an open standard library interface for message passing distributed memory systems
- Version 1.0 (1994), 2.2 (2009), 3.0 (2012)
- MPI-3 contains significant enhancements from MPI-2.2 in many areas
- Documentation is available at www.mpi-forum.org/docs, Bound version is available at Amazon
- The Intel® MPI Library supports MPI-3 since version 5
- Answer to today’s largest clusters: 100k-3mln. cores, up to 30 Petaflops. Challenge: coming exascale systems with even more cores
Intel® MPI library 5.1

High performance Message Passing Interface (MPI) library
  • Efficiency and scalability
  • Interconnect independence
  • Flexible runtime fabric selection

MPI standard support
  • Full MPI 3.0 support
  • Binary compatibility with MPI 1.x, MPI 2.x and MPICH ABI

Extended support
  • Mixed operating systems
  • Latest Intel hardware

Ethernet, Infiniband*, Myrinet*, etc..
  DAPL*, OFA*, TMI*, etc..
Example mpi program

include "mpif.h"                   !-rnp mpi defs
integer  :: inode, nnodes, idest, isrc, itag, mlength      !-rnp node info
integer  :: iflag, comm, istatus(mpi_status_size)    !-rnp status flag
real     :: x = 0

call mpi_init(iflag)
call mpi_comm_rank(mpi_comm_world, inode, iflag)
call mpi_comm_size(mpi_comm_world, nnodes, iflag)
  comm = mpi_comm_world
if (inode.eq.0)  x = 3.87656
  write(6,*) "!-rnp ", inode, " : before communication  x = ", x
if (inode.eq.0) then
  idest = 1 ; itag = 13 ; mlength = 1
  call mpi_send(x, mlength, mpi_real, idest, itag, comm,iflag)
elseif (inode.eq.1) then
  isrc = 0 ; itag = 13 ; mlength = 1
  call mpi_recv(x, mlength, mpi_real, isrc, itag, comm, istatus, iflag)
endif
write(6,*) "!-rnp ", inode, " : after communication  x = ", x
call mpi_finalize(iflag)
When run we have:

\[-\text{rnp} \ 1 \ : \ \text{before communication} \quad x = \ 0.0E+0\]
\[-\text{rnp} \ 0 \ : \ \text{before communication} \quad x = \ 3.87656\]
\[-\text{rnp} \ 1 \ : \ \text{after communication} \quad x = \ 3.87656\]
\[-\text{rnp} \ 0 \ : \ \text{after communication} \quad x = \ 3.87656\]
Why do we need MPI-3?

- Why do we need MPI-3?
- scaling to millions of MPI ranks
- Performance irregularities dynamic communication patterns
- complex architecture of coming HPC clusters requires a new generation of MPI library
- Better handling of Fault tolerance
- Heterogeneous systems, Intel® Xeon® + Xeon Phi™ nodes
- Increased network performance: high-performance network gear such as Intel® True Scale
- Better enabling of MPI for Hybrid apps - MPI+X where X is threading/OpenMP
- Enabling PGAS
MPI 3.0 Openmp Hybrids
Example mpi + openmp

Use mpi.h
use omp_lib
integer :: inode, nnode, idest, isrc, itag, mlength  !-rnp node info
integer :: iflag, comm, istatus(mpi_status_size)  !-rnp status flag

    call mpi_init(iflag)
    call mpi_comm_rank(mpi_comm_world, inode, iflag)
    call mpi_comm_size(mpi_comm_world, nnode, iflag)

!$omp parallel

    // if inode.eq. ...

!$omp end parallel

    call mpi_finalize(iflag)
MPI usage models

Example: Intel® Xeon Phi™ coprocessors

Homogeneous vs heterogeneous nodes
Native vs offload execution
Pure vs hybrid tasking

Better results when combining hybrid MPI+OpenMP execution model
Review important MPI 3.0 additions

- Non-blocking collectives
- Neighborhood (aka Sparse) collectives, e.g., for stencil ops
- One-sided communication enhancements with focus on Shared Memory capability
- Matched MPI_MProbe – important for shared memory programming
- More – not covered here
  - Additional RMA features
  - Fault Tolerance additions
  - MPI Tool interface MPI_T
  - Non-collective, Scalable communicator creation
  - Topology Aware Communicator Creation
  - New Fortran interface
  - Large data counts
  - Enhanced Datatype support
MPI-3 Non-blocking Collectives (NBC)

• Refresher: Non-blocking Point-2-Point in MPI-2
• MPI-3 Non-blocking Collectives Syntax
• New Intel® MPI Benchmarks (IMB) 4.0 for NBC
MPI-2 Non-Blocking Point-to-Point communications (P2P)
(example is from llnl.gov)

- Express Communication/Computation Overlap
  (MPI_Isend/MPI_Irecv paired with MPI_Wait/MPI_Test)

Example: Nearest neighbor exchange in a ring topology

```c
prev = rank-1;
ext = rank+1;
if (rank == 0) prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
  { do some work } 
MPI_Waitall(4, reqs, stats);
```
MPI-3: Non-blocking Collectives (NBC)

- New Non-blocking versions of collective operations in MPI-3
  - MPI_IBcast, MPI_IBarrier, MPI_I Gather, MPI_I Alltoall, etc.
- More than one non-blocking collective operation can be active on a given communicator
- Pseudo-code for a non-blocking reduction (from Optimizing a Conjugate Gradient Solver with NBC operations, Torsten Hoefler et al.)

```c
MPI_Request req;
int sbuf1[SIZE], rbuf1[SIZE];
.....
/* start non-blocking allreduce of sbuf1 */

MPI_Iallreduce(sbuf1, rbuf1, SIZE, MPI_INT, MPI_SUM, MPI_COMM_WORLD, &req);
.....
MPI_Wait(&req, &stat);
```
New Intel MPI Benchmarks for NBC: IMB 4.0 benchmarks for NBC

• To observe performance benefits with IMB NBC benchmarks asynchronous progress support in Intel® MPI 5.0 library needs to be enabled.
  • Set `MPICH_ASYNC_PROGRESS=1`

• 2 nodes test

  
  ```
  mpirun -n 48 -ppn 24 -genv MPICH_ASYNC_PROGRESS 1 ./IMB-NBC-MT ialtoall
  ```

  • shows almost ideal (98%) overlap between computation and communication for large (> 64 Kbytes) message sizes with 2 IVT nodes connected by Mellanox ConnectX adapters

  • Current implementation (in Intel® MPI Library) still introduces significant overhead for small messages (work in progress to improve)
Example: IMB 4.0 NBC test results using Intel® Endeavor Cluster.

t_CPU – time needed for computation

t_pure – time for communication

t_ovrl total time for communication and computation (co Overlap[%] = t_pure/t_ovrl

Endeavor cluster:

Intel® Xeon® E5 v2 processors (Ivy Town) with 12 cores.
Frequency: 2.7 GHz

2 processors per node (24 cores per node)

Mellanox QDR Infiniband

Operating system: RedHat EL 6.4
IMB 4.0 NBC test results using Intel® Endeavor Cluster.

**Async=0**

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#repetitions</th>
<th>t_ovrl[usec]</th>
<th>t_pure[usec]</th>
<th>t_CPU[usec]</th>
<th>Overlap[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>38.88</td>
<td>19.73</td>
<td>19.56</td>
<td>2.05</td>
</tr>
<tr>
<td>.....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1048576</td>
<td>40</td>
<td>252119.93</td>
<td>126327.85</td>
<td>126320.76</td>
<td>0.42</td>
</tr>
</tbody>
</table>

**Async=1**

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#repetitions</th>
<th>t_ovrl[usec]</th>
<th>t_pure[usec]</th>
<th>t_CPU[usec]</th>
<th>Overlap[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>67.23</td>
<td>53.21</td>
<td>53.68</td>
<td>73.85</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>86.54</td>
<td>69.06</td>
<td>69.74</td>
<td>74.93</td>
</tr>
<tr>
<td>.....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1048576</td>
<td>40</td>
<td>128536.33</td>
<td>125450.00</td>
<td>126107.06</td>
<td>97.55</td>
</tr>
</tbody>
</table>
Intel® Trace Analyzer and Collector (ITAC 9.1)

Powerful MPI application profiling and analysis tool

- Evaluate profiling statistics and load balancing
- Learn about communication patterns, parameters, and performance data
- Identify communication hotspots and bottlenecks
Intel® MPI Performance Snapshot (MPS)

Provided as part of Intel® ITAC
Low overhead, scalable MPI application profiler

- **Lightweight** - Low overhead profiling for 32K+ Ranks
- **Scalability** - Performance variation at scale can be detected sooner
- **Identifying Key Metrics** - Shows PAPI counters and MPI/OpenMP imbalances

“Snapshot” view of metrics that hone application’s scalability and load imbalance bottlenecks
Summary: Non-blocking Collectives

• Use model/Semantics similar to familiar MPI-2 point-to-point communications

• Non-blocking Collectives allow overlap computations and communications to improve performance

• Getting the best performance may require change of underlying algorithm
MPI-3 Neighborhood Collectives (NC)

• Why do we need Neighborhood Collectives? Example: Sparse neighborhoods such as stencils
• Starting point: Process topologies
• MPI-3: Only 2 new main Neighborhood Collectives operations
Why do we need Neighborhood Collectives?

- some communication patterns may fit into one of MPI-2 collectives
- e.g. Fast Fourier Transform -> alltoall
- convergence check -> allreduce
- However, sparse communications are not supported in MPI-2
  - 2d and 3d Cartesian neighborhoods that occur in many stencil computations
  - irregular process patterns (e.g., load balancing, adaptive mesh refinement, etc.)
- MPI-2 collectives include global synchronization. Until MPI-3, smarter sparse communications algorithms had to rely on p2p MPI-2 communications to reduce such global dependencies
Example: domain decomposed finite difference code

Peel and remainder performed first

East – West data transfer
Neighborhood Collectives

• Process topologies:
  • n-dimensional Cartesian topologies (with optional wrap-around)
  or
  • arbitrary (distributed) graph

• Neighborhood collectives not necessarily the full graph
Neighborhood Collectives: downloadable source example

- 2D Cartesian Process topology, 5-point stencil (Ref. [5])
Two Neighborhood Collectives operations

MPI_Neighbor_Allgather is sending the same buffer to all destinations and receives into distinct buffers from all sources

MPI_Comm comm_cart;
MPI_Cart_create (MPI_COMM_WORLD, 2 ,dims , periods , 0 ,&comm_cart ) ;
MPI_Neighbor_allgather (sbuf , 1 , MPI_INT , rbuf , 1 , MPI_INT , comm_cart ) ;
MPI_Neighbor_Alltoall is sending different (personalized) buffers to all destinations while receiving into different buffers from all sources.

the communication pattern for each function is specified by an MPI process topology.
Summary: Neighborhood Collectives – simplicity and performance

• Neighborhood Collectives can simplify stencil codes that often written using point-2-point operations.
• But what about performance?
• Example: Weather Research and Forecast (WRF) Model which already implemented an MPI Cartesian communicator on 2D grid was modified to use Neighborhood Collectives (Ref. [6]) - NC version was shown to produce 7-10% performance improvement on average
• Call to Action: need more experiments.
One-sided Communications - Remote Memory Access RMA

• Review of One-sided Communications Concepts
• IMB-4 RMA benchmark test
• Using the RMA interface for a new MPI-based Shared Memory Model in MPI-3
History of One-sided comm. in MPI-2

- Made popular initially by
  - PGAS libraries like SHMEM (Cray)
  - Many proprietary one-sided non-MPI APIs – need a standard one based on MPI – MPI-3 to the rescue

- Unlike traditional two-sided communications and collectives,
  - one-sided comm. decouples data movement from synchronization, eliminating overhead from unneeded synchronization and allowing for greater concurrency.
  - Greater Overlapping potential for communication and computation
MPI-2 one-sided communications performance issues

- **Theory**: message matching and buffering overheads that are required for two-sided communications are eliminated, leading to a significant reduction in communication costs.

- **Practice**: Synchronization functions often added significant overhead resulting in one-sided communication performing much worse than p2p for short- and medium-sized messages.

- Classical critique paper of MPI-2 one-sided standard: “Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations” by Dan Bonachea and Jason Duell (Berkeley)
One-sided MPI-2 Concepts

- one process to specify all communication parameters, both for the sending side and for the receiving side.
- Separate communication and synchronization
- directly read from or write to the memory of a remote process (called window – next slide)
  - Create Window with `MPI_Win_create` (MPI-3 has new Allocate/Deallocate memory APIs: `MPI_Win_Allocate`, `MPI_Win_Free`)
  - `put/get/accumulate` (like a put with reduction); these functions are non-blocking
What is one-sided comm. Window?

• memory that a process allows other processes to access via one-sided communication is called a window
• Group of Processes specify their local windows to other processes by calling the collective function MPI_Win_create
Example with MPI_WIN_CREATE

```c
int main(int argc, char **argv)
{
    int *a; MPI_Win win;
    MPI_Init(&argc, &argv);
    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* use private memory like you normally would */
    a[0] = 1; a[1] = 2;
    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array ‘a’ is now accessibly by all processes in MPI_COMM_WORLD */
    MPI_Win_free(&win);
    MPI_Free_mem(a);
    MPI_Finalize(); return 0;
}
```
One-sided communications fall in two categories

**active target communication**
- both processes are explicitly involved in the communication - similar to 2-sided message passing, except that
- all the data transfer arguments are provided by one process,
- the second process only participates in the synchronization.

**passive target communication**
- only the origin process is explicitly involved in the transfer
- two origin processes may communicate by accessing the same location in a target window.
- This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.
Synchronization mechanisms defined in MPI 2.2

Three synchronization mechanisms defined in MPI 2.2:

• Fence Active Target and collective
• Post-Start-Complete-Wait (PSCW), Active target
• Passive Target: Lock-Unlock sync
for (i=0; i<ntimes; i++) {
    MPI_Win_fence(..., win);
    for (j=0; j < neighbours; j++) {
        MPI_Put(sbuf + j*n, n, MPI_INT, nbr[j], j, n, MPI_INT, win);
    }
    MPI_Win_fence(..., win);
}

Note:
- Synchronisation
- Collective
Active Target: Post-start-complete-wait (PSCW)

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_start(1)</td>
<td>MPI_Win_post(0,2)</td>
<td>MPI_Win_start(1)</td>
</tr>
<tr>
<td>MPI_Put(1)</td>
<td></td>
<td>MPI_Put(1)</td>
</tr>
<tr>
<td>MPI_Get(1)</td>
<td></td>
<td>MPI_Get(1)</td>
</tr>
<tr>
<td>MPI_Win_complete(1)</td>
<td></td>
<td>MPI_Win_complete(win)</td>
</tr>
<tr>
<td></td>
<td>MPI_Win_wait(0,2)</td>
<td></td>
</tr>
</tbody>
</table>

Note the usage of MPI_Group in

- MPI_Win_start
- MPI_Win_post
- Neighbour collectives
Passive Target: Lock-Unlock sync.

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Win_create(&amp;win)</code></td>
<td><code>MPI_Win_create(&amp;win)</code></td>
<td><code>MPI_Win_create(&amp;win)</code></td>
</tr>
<tr>
<td><code>MPI_Win_lock(shared,1)</code></td>
<td><code>MPI_Win_lock(shared,1)</code></td>
<td><code>MPI_Win_lock(shared,1)</code></td>
</tr>
<tr>
<td><code>MPI_Put(1)</code></td>
<td><code>MPI_Put(1)</code></td>
<td><code>MPI_Put(1)</code></td>
</tr>
<tr>
<td><code>MPI_Get(1)</code></td>
<td><code>MPI_Get(1)</code></td>
<td><code>MPI_Get(1)</code></td>
</tr>
<tr>
<td><code>MPI_Win_unlock(1)</code></td>
<td><code>MPI_Win_unlock(1)</code></td>
<td><code>MPI_Win_unlock(1)</code></td>
</tr>
<tr>
<td><code>MPI_Win_free(&amp;win)</code></td>
<td><code>MPI_Win_free(&amp;win)</code></td>
<td><code>MPI_Win_free(&amp;win)</code></td>
</tr>
</tbody>
</table>

- **SHARED**: Other processes using shared can access concurrently
- **EXCLUSIVE**: No other processes can access concurrently
- **MPI-2** Passive synchronization is actively expanded in MPI-3
- **Lots of scope for interpretation**
IMB 4.0 one-sided (RMA) tests

• Target and origin process should be synchronized prior to each MPI_Put() operation.
• For correct measurement of the time spent by the target process in computation function should be comparable to the time needed for successful completion of MPI_Put() operation by the origin process.
• To achieve this, the target process may send the timing value needed for MPI_Put() completion to the target process.
• Source of IMB 4.0 truly passive mode test is in IMB_rma_put.c, function IMB_rma_passive_put
• Uses MPI_Win_lock, that Begins an RMA access epoch at the target process, The name of this routine is misleading. In particular, this routine need not block, except when the target process is the calling process.
Shared-memory (one-sided) windows in MPI-3 to change Hybrid Programming

• Typical Hybrid MPI + OpenMP codes today often need to scale to 60+ cores on the node
  • Leading to two-stage parallelization – MPI and OpenMP layers for domain decomposition – additional complexity
  • This is in addition to loop level parallelization

• MPI-3 Shared-memory windows allow
  • shared memory to be shared across different MPI processes with a single level of domain decomposition (using MPI).
  • OpenMP can be used at the loop level only

• why stop on shared memory: Locales in planned MPI-3.1: racks of servers, or servers belonging to the same switch, etc.
Call to action

1. Run IMB 4.0 NBC and RMA tests on your system with the latest Intel® MPI Library 5.0 (that implements MPI-3) to study computation/communication overlap

2. Check if your code can benefit from Non-Blocking collectives.

3. If your code uses stencils and implements them using P2P MPI operations – try to use Neighborhood Collectives to simplify code base and improve performance

4. Experiment with Shared-Memory Model provided by MPI-3 One-Sided Communication interface.
### mpitune

<table>
<thead>
<tr>
<th>Cluster specific mode</th>
<th>Application Specific mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpopular hardware and/or software configuration including network topology</td>
<td>The application is run on regular base</td>
</tr>
<tr>
<td>MPI benchmarks do not reproduce network bandwidth/latency benchmark with “acceptable overhead”</td>
<td>All other optimizations were done or almost impossible</td>
</tr>
<tr>
<td>MPI operations have “dips” on message range in benchmarking</td>
<td>Heterogeneity of the system: network topology, accelerators, threading runtime, etc.</td>
</tr>
<tr>
<td>Using “untypical” node/ppn count for regular tasks on a cluster</td>
<td>Using “untypical” sizes of messages, communicators</td>
</tr>
<tr>
<td></td>
<td>Using “untypical” process count including per node</td>
</tr>
</tbody>
</table>
Why do we need MPI-3?

- Scaling to millions of MPI ranks
- Performance irregularities dynamic communication patterns
- Complex architecture of coming HPC clusters requires a new generation of MPI library
- Better handling of Fault tolerance
- Enabling PGAS
- Heterogeneous systems, Intel® Xeon® + Xeon Phi™ nodes
- Better enabling of MPI for Hybrid apps - MPI+X where X is threading/OpenMP
- Increased network performance: high-performance network gear such as Intel® True Scale
Thank you
Message Passing Interface (MPI) – towards millions of cores

- MPI is an open standard library interface for message passing distributed memory systems
- Version 1.0 (1994), 2.2 (2009), 3.0 (2012)
- MPI-3 contains significant enhancements from MPI-2.2 in many areas
- Documentation is available at www.mpi-forum.org/docs, Bound version is available at Amazon
- The Intel® MPI Library supports MPI-3 since version 5
- Answer to today’ largest clusters: 100k-3mln. cores, up to 30 Petaflops. Challenge: coming exascale systems with even more cores
Additional Resources

1. Advanced MPI 2.2 and 3.0 Tutorial, Torsten Hoefler
2. A Case for Standard Non-Blocking Collective Operations, Torsten Hoefler et.al
3. A One-Sided View of HPC: Global-View Models and Portable Runtime Systems, James Dinan
5. Leveraging non-blocking Collective Communication in high-performance Applications, Torsten Hoefler, et. al
6. Optimization Principles for Collective Neighborhood Communications, Torsten Hoefler and Timo Schneider
7. An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface, James Dinan et al.
8. Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming, Torsten Hoefler et.al
MPI 3.0 Openmp Hybrids

Interconnect

- Head node (SB)
  - KNC
  - openmp

- Head node (SB)
  - KNC
  - openmp

- Head node (SB)
  - KNC
  - openmp
2D Poisson solver, 5p stencil, Ref [7]

- Iteration: halo exchanges in each direction, updates all local points

```c
MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm);

MPI_Win_allocate_shared(size*sizeof(double), info, shmcomm, &mem, &win);
MPI_Win_shared_query(win, north, &sz, &northptr);
// ... south, east, west directions

for(iter=0; iter<niters; ++iter) {
    MPI_Win_fence(0, win); // start new access and exposure epoch
    if(north != MPI_PROC_NULL) // the "communication" – direct local access to
        // remote window portion
        for(int i=0; i<bx; ++i) a2[ind(i+1,0)] = northptr[ind(i+1,by)];
    // ... south, east, west directions
    update_grid(&a1, &a2); // apply operator and swap arrays
}
```
Shared-memory extension using MPI-3 one-sided comm. (Ref. [7])

- MPI_Win_allocate_shared collectively allocates and maps shared memory across all processes in the given communicator
- permits direct (load/store) use of shared memory within MPI-only programs
BACK UP
Intel Endeavor Cluster Environment

- Intel® Xeon® E5 v2 processors (Ivy Town) with 12 cores. Frequency: 2.7 GHz
- 2 processors per node (24 cores per node)
- Mellanox QDR Infiniband
- Operating system: RedHat EL 6.4
MPI-2 Separate memory model

- MPI-2 implementations maintain two copies of the window in order to facilitate both remote and local updates on non-coherent systems.
- When a window synchronization is performed, the MPI implementation must synchronize the public and private window copies.
- A lot of restrictions – portable but too conservative.
  - next slide - case 1: MPI-2 forbids concurrent overlapping operations when any of the operations writes to the window data.
  - next slide - case 2 and 3: local updates cannot be performed concurrently with any other operations.
- MPI library is unaware of which locations are updated when the window buffer is directly accessed by the hosting process.
- Any violation of these semantics is defined to be an MPI error.
MPI-2 Separate Memory Model (Ref [2])

1. Get
2. Put/acc
3. Put/acc
4. Get

Public Copy

Private Copy

Synchronization

Load

Store
New MPI-3 “Unified” Memory Model

- “Unified” - public and private copies are logically the same
- for cache-coherent systems provides lightweight local and remote synchronization
- updates to either “copy” automatically propagate
- Explicit synchronization operations can be used to ensure completion of individual or groups of operations (request, flush, atomic)
New One-Sided Request-Based Operations in MPI-3

- MPI-2 one-sided operations do not return a request handle
  - Instead, completion is managed through synchronization operations such as fence, PSCW, and lock/unlock
- MPI-3 “R” versions provide MPI request handles: MPI_Rput, MPI_Rget, MPI_Raccumulate, MPI_Rget_accumulate;
  - requests can be passed to the usual MPI request completion operation routines (e.g., MPI_Waitany) to ensure local completion of the operation
  - provides an alternative mechanism for controlling operation completion with fine granularity
- Only valid within **passive-target epoch**
  - E.g., between MPI_Win_lock/MPI_Win_unlock
  - Provides one way to complete MPI RMA operations within a passive target epoch
MPI-3 One-Sided Synchronization Operations - Flush

- Permitted only within passive target epoch
- MPI_Win_flush, MPI_Win_flush_all completes all pending RMA operations at origin and target
- MPI_Win_flush_local, MPI_Win_flush_local_all completes all pending RMA operations at origin
MPI-3 One-Sided **Atomic** Operations

- New Read-Modify-Write Operations
- MPI_Get_accumulate – extends MPI_Accumulate to also return value
- MPI_Fetch_and_op, MPI_Compare_and_swap – Atomic, single word updates; intended to provide higher performance than general MPI_Get_accumulate
- Now possible perform mutex-free updates
Other MPI-3 RMA Extensions

• Some behavior, such as conflicting accesses, now have undefined behavior rather than erroneous
• Behavior of correct MPI 2.2 programs unchanged
• Simplifies use of MPI as a target for other RMA programming models (PGAS) that allow conflicting accesses
• Accumulate operations ordered by default
  • No “right” choice – some algorithms much easier if RMA operations ordered; some hardware much faster if ordering not required.
  • Info key “accumulate_ordering” (on window create) can request relaxation of ordering
• New MPI_Win_lock_all/MPI_Win_unlock_all for passive target epoch for all processes in Win.
MPI-3: Scalable Communicator Creation

- MPI-2: Communicator creation is all-collective
- MPI-3: New group-collective creation to Lower overhead when creating small communicators
  - Collective only on members of new communicator
  - Create communicators without involving all processes in the parent communicator.
  - Avoid unnecessary synchronization - Reduce overhead
  - Enable asynchronous multi-level parallelism
  - Useful for dynamic load balancing, fault tolerance
MPI-3: Topology-aware communicator creation

Allows to create a communicator whose processes can create a shared memory region

- MPI_Comm_split_type( comm, comm_type, key, info, split_comm)
- More generally: it splits a communicator into sub-communicators split_comm of a certain type:
  - MPI_COMM_TYPE_SHARED: shared memory capability
  - Other implementation specific types are possible: rack, switch, etc.
MPI-3: Fault Tolerance and MPI_T

• Fault Tolerance
  • Run-through stabilization (MPI-3.0) - non-failed processes can continue to use MPI and can determine which ranks have failed
  • Process recovery (planned for MPI-3.1) - replace the failed process in all existing communicators, windows.

• MPI Tool Interface MPI_T
  • Attempt to create a standard for querying of the MPI library state (vars and counters) at runtime; API similar to PAPI
  • Primarily for MPI performance tools (ITAC, Scalasca, Vampir, Tau and others) - hooks on MPI internal information
MPI-3: Three methods of Fortran support

• **USE mpi_f08**
  • This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TR 29113 and later).
  • This method is highly recommended for all MPI applications.
  • Mandatory compile-time argument checking & unique MPI handle types.
  • Convenient migration path.

• **USE mpi**
  • This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended.
  • It exists only for backwards compatibility.
  • Mandatory compile-time argument checking (but all handles match with INTEGER)
MPI-3: Three methods of Fortran support, cont’d

• INCLUDE ‘mpif.h’
  • The use of the include file mpif.h is strongly discouraged starting with MPI-3.0.
  • Does not guarantee compile-time argument checking.
  • Does not solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard.
  • It exists only for backwards compatibility with legacy MPI applications.
Large Counts

- **MPI-2.2**
  - All counts are int / INTEGER
  - Producing longer messages through derived datatypes may cause problems

- **MPI-3.0**
  - New type to store long counts:
    - MPI_Count / INTEGER(KIND=MPI_COUNT_KIND)
  - Additional routines to handle “long” derived datatypes:
    - MPI_Type_size_x, MPI_Type_get_extent_x, MPI_Type_get_true_extent_x
    - “long” count information within a status:
      - MPI_Get_elements_x, MPI_Status_set_elements_x
  - Communication routines are not changed !!!