MPI+X - Hybrid Programming on Modern Compute Clusters with Multicore Processors and Accelerators

Rolf Rabenseifner\textsuperscript{1)} \hspace{1cm} Georg Hager\textsuperscript{2)}

\textsuperscript{1)} High Performance Computing Center (HLRS), University of Stuttgart, Germany
\textsuperscript{2)} Erlangen Regional Computing Center (RRZE), University of Erlangen, Germany

LRZ, Garching near Munich, Jan. 14, 2016
Based on Tutorial 137 at SC15, Nov. 15, 2015, Austin, Texas, USA
General Outline (with slide numbers)

- **Motivation** (3)
  - Cluster hardware today: Multicore, multi-socket, accelerators (4)
  - Options for running code (5)

- **Introduction** (8)
  - Prevalent hardware bottlenecks (9)
  - Interlude: ccNUMA (13)
  - The role of machine topology (20)
  - Cost-Benefit Calculation (31)

- **Programming models** (33)
  - Pure MPI (34)
  - MPI + MPI-3.0 shared memory (61)
    - **Two exercises** (77 and 95)
  - MPI + OpenMP on multi/many-core (101)
  - MPI + Accelerators (164)

- **Tools** (182)
  - Topology & Affinity (182)
  - Debug, profiling (184)

- **Conclusions** (191)
  Major opportunities and challenges of “MPI+X”
  - MPI+OpenMP (192)
  - MPI+MPI-3.0 (195)
  - Pure MPI (196)
  - Acknowledgements (197)
  - Conclusions (198)

- **Appendix** (199)
  - Exercise-solutions (200)
  - Abstract (207)
  - Authors (208)
  - References (210)
Motivation
Hardware and Programming Models

Hardware
- Cluster of
  - ccNUMA nodes with several multi-core CPUs
  - nodes with multi-core CPUs + GPU
  - nodes with multi-core CPUs + Intel Phi
  - ...
Options for running code

- Which programming model is fastest?
- MPI everywhere?
- Fully hybrid MPI & OpenMP?
- Something between? (Mixed model)
- Often hybrid programming slower than pure MPI
  - Examples, Reasons, …

Cluster hardware today

Motivation
- Introduction
- Programming models
- Tools
- Conclusions
More Options

Number of options multiply if accelerators are added

- One MPI process per accelerator?
- One thread per accelerator?
- Which programming model on the accelerator?
  - OpenMP shared memory
  - MPI
  - OpenACC
  - OpenMP-4.0 accelerator
  - CUDA
  - …
Splitting the Hardware Hierarchy

Hierarchical hardware

- Cluster of
  - ccNUMA nodes with
    - CPUs with
      - N x M cores with
        - Hyperthreads (simultaneous multithreading)

Hierarchical parallel programming

- MPI (outer level) +
  - X (e.g. OpenMP)

Many possibilities for splitting the hardware hierarchy into MPI + X:
- 1 MPI process per ccNUMA node
- ...
- OpenMP only for hyperthreading

Where is the main bottleneck?
Ideal choice may be extremely problem-dependent.
No ideal choice for all problems.
Outline

Motivation
Introduction
Pure MPI
MPI + MPI-3.0 shared memory
MPI + OpenMP on multi/many-core
MPI + Accelerators
Introduction
Typical hardware bottlenecks and challenges
Hardware Bottlenecks

- Multicore cluster
  - Computation
  - Memory bandwidth
  - Inter-node communication
  - Intra-node communication (i.e., CPU-to-CPU)
  - Intra-CPU communication (i.e., core-to-core)

- Cluster with CPU+Accelerators
  - Within the accelerator
    - Computation
    - Memory bandwidth
    - Core-to-Core communication
  - Within the CPU and between the CPUs
    - See above
  - Link between CPU and accelerator
Hybrid Parallel Programming

Example:

- Sparse matrix-vector-multiply with **stored matrix entries**
  → Bottleneck: memory bandwidth of each CPU

- Sparse matrix-vector-multiply with **calculated matrix entries**
  (many complex operations per entry)
  → Bottleneck: computational speed of each core

- Sparse matrix-vector multiply with **highly scattered matrix entries**
  → Bottleneck: Inter-node communication
Topology complicates matters

- Symmetric, UMA-type single-core compute nodes have become rare animals (NEC SX, Hitachi SR8k, IBM SP2)

- Instead, systems have become “non-isotropic” on the node level, with rich topology:
  - ccNUMA (all modern multi-core architectures)
    - Where does the code run vs. where is the memory?
  - Multi-core, multi-socket (dito)
    - Bandwidth bottlenecks on multiple levels
    - Communication performance heterogeneity
  - Accelerators (GPGPU, Intel Phi)
    - Threads, warps, blocks, SMX
    - SMT threads, cores, caches, mem. controllers
    - PCIe structure
Interlude: ccNUMA
A short introduction to ccNUMA

- ccNUMA:
  - whole memory is **transparently accessible** by all processors
  - but **physically distributed**
  - with varying bandwidth and latency
  - and **potential contention** (shared memory paths)
  - Memory placement occurs with **OS page granularity** (often 4 KiB)
How much bandwidth does non-local access cost?

- Example: AMD Magny Cours 4-socket system (8 chips, 4 sockets)
  
  *STREAM Triad bandwidth measurements*

![Bandwidth Measurement Diagram]
How much bandwidth does non-local access cost?

- Example: Intel Sandy Bridge 2-socket system (2 chips, 2 sockets)
  *STREAM Triad bandwidth measurements*

![Diagram showing bandwidth measurements](image)

- 38.7 GB/s
- 14.6 GB/s

**General rule:**

The more ccNUMA domains, the larger the non-local access penalty
ccNUMA Memory Locality Problems

- Locality of reference is key to scalable performance on ccNUMA
  - Less of a problem with pure MPI, but see below
- What factors can destroy locality?
  - MPI programming:
    - processes lose their association with the CPU the mapping took place on originally
    - OS kernel tries to maintain strong affinity, but sometimes fails
  - Shared Memory Programming (OpenMP, hybrid):
    - threads losing association with the CPU the mapping took place on originally
    - improper initialization of distributed data
    - Lots of extra threads are running on a node, especially for hybrid
  - All cases:
    - Other agents (e.g., OS kernel) may fill memory with data that prevents optimal placement of user data (“ccNUMA buffer cache problem”)
Avoiding locality problems

- How can we make sure that memory ends up where it is close to the CPU that uses it?
  - See next slide

- How can we make sure that it stays that way throughout program execution?
  - See later in the tutorial

- **Taking control** is the key strategy!
Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:

  A memory page gets mapped into the local memory of the processor that first touches it!

- Consequences
  - Process/thread-core affinity is decisive!
  - Data initialization code becomes important even if it takes little time to execute ("parallel first touch")
  - Parallel first touch is automatic for pure MPI
  - If thread team does not span across ccNUMA domains, placement is not a problem

- See later for more details and examples
Interlude: Influence of topology on low-level operations
What is “topology”?

Where in the machine does core (or hardware thread) #n reside?

Why is this important?

- Resource sharing (cache, data paths)
- Communication efficiency (shared vs. separate caches, buffer locality)
- Memory access locality (ccNUMA!)

Core #3

Core #17
Output of likwid-topology

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Hybrid Parallel Programming

Slide 23 / 198

Rabenseifner, Hager, Jost

likwid-topology continued

Socket 0: ( 0 1 2 3 4 5 6 7 )
Socket 1: ( 8 9 10 11 12 13 14 15 )

********************************************************************************
Cache Topology
********************************************************************************
Level: 1
Size: 32 kB
Cache groups: ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 )

Level: 2
Size: 256 kB
Cache groups: ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 )

Level: 3
Size: 8 MB
Cache groups: ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 )

• ... and also try the ultra-cool \(-g\) option!
Intra-node MPI characteristics: IMB Ping-Pong benchmark

- Code (to be run on 2 cores):
  
  ```
  wc = MPI_WTIME()
  do i=1,NREPEAT
      if(rank.eq.0) then
          MPI_SEND(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD,ierr)
          MPI_RECV(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD, & status,ierr)
      else
          MPI_RECV(…)
          MPI_SEND(…)
      endif
  enddo
  wc = MPI_WTIME() - wc
  ```

- Intranode (1S): `aprun -n 2 -cc 0,1 ./a.out`
- Intranode (2S): `aprun -n 2 -cc 0,16 ./a.out`
- Internode: `aprun -n 2 -N 1 ./a.out`
IMB Ping-Pong: Latency

Intra-node vs. Inter-node on Cray XE6

Affinity matters!
IMB Ping-Pong: Bandwidth Characteristics

Intra-node vs. Inter-node on Cray XE6

Bandwidth: Surprisingly similar!

Between two cores of one socket

Latency: Very different!

Between two nodes via InfiniBand

Between two sockets of one node

Effective bandwidth [MByte/s]

Message length [bytes]
The throughput-parallel vector triad benchmark

Microbenchmarking for architectural exploration

- Every core runs its own, independent bandwidth benchmark

\[
\text{double precision, dimension}(:,), \text{allocatable} :: \text{A,B,C,D}
\]

\[
\begin{align*}
!&\text{OMP PARALLEL private}(i,j,A,B,C,D) \\
al\text{locate}(A(1:N),B(1:N),C(1:N),D(1:N)) \\
A=1.d0; B=A; C=A; D=A \\
do \text{j}=1,NITER \\
do \text{i}=1,N \\
A(i) = B(i) + C(i) \times D(i) \\
enddo \\
\text{if (.something.that.is.never.true.) then} \\
call \text{dummy}(A,B,C,D) \\
endif \\
enddo \\
!&\text{OMP END PARALLEL}
\]

\[ \rightarrow \text{pure hardware probing, no impact from OpenMP overhead} \]
Bandwidth saturation vs. # cores on Sandy Bridge socket (3 GHz)

- L1
- L2
- L3
- Memory

Scalable BW in L1, L2, L3 cache

Motivation
Introduction
Programming models
Tools
Conclusions

Prevalent hardware bottleneck
Interlude: ccNUMA
The role of machine topology
Cost-Benefit Calculation
Throughput vector triad on Sandy Bridge socket (3 GHz)

Saturation effect in memory

Scalable BW in L1, L2, L3 cache

Performance [GFlops/s] vs Loop length
Conclusions from the observed topology effects

- Know your hardware characteristics:
  - Hardware topology (use tools such as likwid-topology)
  - Typical hardware bottlenecks
    - These are independent of the programming model!
  - Hardware bandwidths, latencies, peak performance numbers

- Learn how to take control
  - Affinity control is key! (What is running where?)
  - Affinity is usually controlled at program startup
    → know your system environment

- See later in the “How-To” section for more on affinity control

- **Leveraging topology effects is a part of code optimization!**
Remarks on Cost-Benefit Calculation
Remarks on Cost-Benefit Calculation

Costs
• for optimization effort
  – e.g., additional OpenMP parallelization
  – e.g., 3 person month x 5,000 € = 15,000 € (full costs)

Benefit
• from reduced CPU utilization
  – e.g., Example 1:
    100,000 € hardware costs of the cluster
    x 20% used by this application over whole lifetime of the cluster
    x 7% performance win through the optimization
    = 1,400 € $\rightarrow$ total loss = 13,600 €
  – e.g., Example 2:
    10 Mio € system x 5% used x 8% performance win
    = 40,000 € $\rightarrow$ total win = 25,000 €

Question: Do you want to spend work hours without a final benefit?
Programming models
Programming models - pure MPI
Pure MPI

Advantages
- No modifications on existing MPI codes
- MPI library need not to support multiple threads

Major problems
- Does MPI library use different protocols internally?
  - Shared memory inside of the SMP nodes
  - Network communication between the nodes
- Is the network prepared for many communication links?
- Does application topology fit on hardware topology?
  - Minimal communication between MPI processes AND between hardware SMP nodes
- Unnecessary MPI-communication inside of SMP nodes!
- Generally “a lot of” communicating processes per node
- Memory consumption: Halos & replicated data
Does the network support many concurrent communication links?

- Bandwidth of parallel communication links between SMP nodes

Measurements: bi-directional halo exchange in a ring with 4 SMP nodes (with 16B and 512kB per message; bandwidth: each message is counted only once, i.e., not twice at sender and receiver); reported:

- Latency, accumulated bandwidth of all links per node

<table>
<thead>
<tr>
<th>Cray XC30</th>
<th>Xeon+Infiniband</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Sandybridge @ HLRS)</td>
<td>(beacon @ NICS)</td>
</tr>
<tr>
<td>4.1 µs, 6.8 GB/s</td>
<td>1.6 µs, 5.4 GB/s</td>
</tr>
<tr>
<td>4.1 µs, 7.1 GB/s</td>
<td>2.1 µs, 5.4 GB/s</td>
</tr>
<tr>
<td>4.1 µs, 5.2 GB/s</td>
<td>2.1 µs, 5.1 GB/s</td>
</tr>
<tr>
<td>4.4 µs, 4.7 GB/s</td>
<td>2.4 µs, 5.0 GB/s</td>
</tr>
<tr>
<td>10.2 µs, 4.2 GB/s</td>
<td>12.1 µs, 4.8 GB/s</td>
</tr>
</tbody>
</table>

Conclusion:
One communicating core per node (i.e., hybrid programming) may be better than many communicating cores (e.g., with pure MPI)
To minimize communication?

- Bandwidth of parallel communication links between **Intel Xeon Phi**

<table>
<thead>
<tr>
<th>Links per Phi</th>
<th>One Phi per node (beacon @ NICS)</th>
<th>4 Phis on one node (beacon @ NICS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>15 µs, 0.83 GB/s</td>
<td>15 µs, 0.83 GB/s</td>
</tr>
<tr>
<td>2x</td>
<td>26 µs, 0.87 GB/s</td>
<td></td>
</tr>
<tr>
<td>4x</td>
<td>25 µs, 0.91 GB/s</td>
<td></td>
</tr>
<tr>
<td>8x</td>
<td>23 µs, 0.91 GB/s</td>
<td></td>
</tr>
<tr>
<td>16x</td>
<td>24 µs, 0.92 GB/s</td>
<td></td>
</tr>
<tr>
<td>30x</td>
<td>21 µs, 0.91 GB/s</td>
<td></td>
</tr>
<tr>
<td>60x</td>
<td>51 µs, 0.90 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

**Conclusions:**

Intel Xeon Phi is well prepared for one MPI process per Phi. Communication is no reason for many MPI processes on each Phi.
MPI communication on Intel Phi

- Communication of MPI processes inside of an Intel Phi:
  (bi-directional halo exchange benchmark with all processes in a ring;
  bandwidth: each message is counted only once, i.e., not twice at sender and receiver)

<table>
<thead>
<tr>
<th>Number of MPI processes</th>
<th>Latency (µs)</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>0.80</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>0.75</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>0.66</td>
</tr>
<tr>
<td>60</td>
<td>29</td>
<td>0.50</td>
</tr>
<tr>
<td>120</td>
<td>149</td>
<td>0.19</td>
</tr>
<tr>
<td>240</td>
<td>745</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Conclusion:
MPI on Intel Phi works fine on up to 60 processes, but the 4 hardware threads per core require OpenMP parallelization.
Levels of communication or data access

- Three levels:
  - Between the SMP nodes
  - Between the sockets inside of a ccNUMA SMP node
  - Between the cores of a socket

- On all levels, the communication should be minimized:
  - With 3-dimensional sub-domains:
    - They should be as cubic as possible

```
Outer surface corresponds to the data communicated to the neighbor nodes in all 6 directions

Inner surfaces correspond to the data communicated or accessed between the cores inside of a node
```

- Pure MPI on clusters of SMP nodes may result in inefficient SMP-sub-domains:

```
Originally perfectly optimized shape for each MPI process; but terrible when clustered only in one dimension. Slow-down with 20–50% communication footprint:
  - 8–20% slowdown with 8 cores
  - 23–46% slowdown with 32 cores
```

Details → next slide (skipped)
Loss of communication bandwidth if not cubic

\[ N^3 = N \times N \times N \]

\[ N^3 = 2 \frac{N}{\sqrt{2}} \times \frac{N}{\sqrt{2}} \times \frac{N}{\sqrt{2}} \]
\[ N^3 = 4 \frac{N}{\sqrt{4}} \times \frac{N}{\sqrt{4}} \times \frac{N}{\sqrt{4}} \]
\[ N^3 = 8 \frac{N}{\sqrt{8}} \times \frac{N}{\sqrt{8}} \times \frac{N}{\sqrt{8}} \]
\[ N^3 = 16 \frac{N}{\sqrt{16}} \times \frac{N}{\sqrt{16}} \times \frac{N}{\sqrt{16}} \]

- \[ N^3 = 32 \frac{N}{\sqrt{32}} \times \frac{N}{\sqrt{32}} \times \frac{N}{\sqrt{32}} \]
- \[ N^3 = 64 \frac{N}{\sqrt{64}} \times \frac{N}{\sqrt{64}} \times \frac{N}{\sqrt{64}} \]

\[ bw = \frac{3(\sqrt{2})^2}{2 \cdot 1 + 2 \cdot 1 + 1 \cdot 1} \cdot bw_{opt.} = 95\% \cdot bw_{opt.} \]
\[ bw = \frac{3(\sqrt{4})^2}{4 \cdot 1 + 4 \cdot 1 + 1 \cdot 1} \cdot bw_{opt.} = 84\% \cdot bw_{opt.} \]
\[ bw = \frac{3(\sqrt{8})^2}{8 \cdot 1 + 8 \cdot 1 + 1 \cdot 1} \cdot bw_{opt.} = 71\% \cdot bw_{opt.} \]
\[ bw = \frac{3(\sqrt{16})^2}{16 \cdot 1 + 16 \cdot 1 + 1 \cdot 1} \cdot bw_{opt.} = 58\% \cdot bw_{opt.} \]
\[ bw = \frac{3(\sqrt{32})^2}{32 \cdot 1 + 32 \cdot 1 + 1 \cdot 1} \cdot bw_{opt.} = 47\% \cdot bw_{opt.} \]
\[ bw = \frac{3(\sqrt{64})^2}{64 \cdot 1 + 64 \cdot 1 + 1 \cdot 1} \cdot bw_{opt.} = 37\% \cdot bw_{opt.} \]

**Slow down factors** of your application (communication footprint calculated with optimal bandwidth)
- With 20% communication footprint: **Slow down** by 1.01, 1.04, 1.08, 1.14, 1.23, or 1.34
- With 50% communication footprint: **Slow down** by 1.03, 1.10, 1.20, 1.36, 1.56, or 1.85!
The topology problem: How to fit application sub-domains to hierarchical hardware

When do we need a **multi-level** domain decomposition?

- Not needed with
  - pure MPI+OpenMP, i.e., one MPI process per SMP node
  - mixed level hybrid MPI+OpenMP with only 2 or 4 MPI-processes/SMP node.
  
  In these cases, one-level domain-decomposition is enough

- Needed for
  - mixed level hybrid MPI+OpenMP with > 4 MPI-processes/SMP node
  - MPI + MPI-3.0 shared memory
  - Pure MPI
Pure MPI – multi-core aware

- Hierarchical domain decomposition (or distribution of Cartesian arrays)

Domain decomposition:
1 sub-domain / SMP node

Further partitioning:
1 sub-domain / socket

Cache optimization:
Blocking inside of each core, block size relates to cache size. 1-3 cache levels.

Example on 10 nodes, each with 4 sockets, each with 6 cores.
How to achieve such hardware-aware domain decomposition (DD)?

• Maybe simplest method for structured/Cartesian grids:
  – Sequentially numbered MPI_COMM_WORLD
    • Ranks 0-7: cores of 1st socket on 1st SMP node
    • Ranks 8-15: cores of 2nd socket on 1st SMP node
    • ...
  – Hierarchical re-numbering the MPI processes together with MPI Cartesian virtual coordinates → next slides

• Unstructured grids → coming later
Hierarchical Cartesian DD

Coordinate 0

Coordinate 1

Coordinate 2

Node coord.

coord. in SMP

Global coord.

Virtual location of an MPI process within an SMP node

All MPI processes of an SMP node

Implementation hints on following (skipped) slide

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI
Hierarchical Cartesian DD

// Input: Original communicator: MPI_Comm comm_orig; (e.g. MPI_COMM_WORLD)
// Number of dimensions: int ndims = 3;
// Global periods:         int periods_global[] = /*e.g.*/ {1,0,1};
MPI_Comm_size (comm_orig, &size_global);
MPI_Comm_rank (comm_orig, &myrank_orig);

// Establish a communicator on each SMP node:
MPI_Comm_split_type (comm_orig, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_smp_flat);
MPI_Comm_size (comm_smp_flat, &size_smp);
int dims_smp[] = {0,0,0}; int periods_smp[] = {0,0,0} /*always non-period*/;
MPI_Dims_create (size_smp, ndims, dims_smp);
MPI_Cart_create (comm_smp_flat, ndims, dims_smp, periods_smp, /*reorder=*/ 1, &comm_smp_cart);
MPI_Comm_free (&comm_smp_flat);
MPI_Comm_rank (comm_smp_cart, &myrank_smp);
MPI_Cart_coords (comm_smp_cart, myrank_smp, ndims, mycoords_smp);

// This source code requires that all SMP nodes have the same size. It is tested:
MPI_Allreduce (&size_smp, &size_smp_min, 1, MPI_INT, MPI_MIN, comm_orig);
MPI_Allreduce (&size_smp, &size_smp_max, 1, MPI_INT, MPI_MAX, comm_orig);
if (size_smp_min < size_smp_max) { printf("non-equal SMP sizes\n"); MPI_Abort (comm_orig, 1); }
Hierarchical Cartesian DD

// Establish the node rank. It is calculated based on the sequence of ranks in comm_orig
// in the processes with myrank_smp == 0:
MPI_Comm_split (comm_orig, myrank_smp, 0, &comm_nodes_flat);
// Result: comm_nodes_flat combines all processes with a given myrank_smp into a separate communicator.
// Caution: The node numbering within these comm_nodes-flat may be different.
// The following source code expands the numbering from comm_nodes_flat with myrank_smp == 0
// to all node-to-node communicators:
MPI_Comm_size (comm_nodes_flat, &size_nodes);
int dims_nodes[] = {0,0,0}; for (i=0; i<ndims; i++) periods_nodes[i] = periods_global[i];
MPI_Dims_create (size_nodes, ndims, dims_nodes);
if (myrank_smp==0) {
    MPI_Cart_create (comm_nodes_flat, ndims, dims_nodes, periods_nodes, 1, &comm_nodes_cart);
    MPI_Comm_rank (comm_nodes_cart, &myrank_nodes);
    MPI_Comm_free (&comm_nodes_cart); /*was needed only to calculate myrank_nodes*/
}
MPI_Comm_free (&comm_nodes_flat);
MPI_Bcast (&myrank_nodes, 1, MPI_INT, 0, comm_smp_cart);
MPI_Comm_split (comm_orig, myrank_smp, myrank_nodes, &comm_nodes_flat);
MPI_Cart_create (comm_nodes_flat, ndims, dims_nodes, periods_nodes, 0, &comm_nodes_cart);
MPI_Cart_coords (comm_nodes_cart, myrank_nodes, ndims, mycoords_nodes);
MPI_Comm_free (&comm_nodes_flat);

Optimization according to inter-node network of the first processes in each SMP node

Copying it for the other processes in each SMP node
Hybrid Parallel Programming

Motivation
Introduction
Programming models
Tools
Conclusions

Hybrid MPI
Pure MPI
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators

Hierarchical Cartesian DD

comm_smp_cart for all processes with coord_nodes== {1,2,0}

comm_nodes_cart for all processes with mycoord_smp== {2,3,1}
Hierarchical Cartesian DD

// Establish the global Cartesian communicator:
for (i=0; i<ndims; i++) {
    dims_global[i] = dims_smp[i] * dims_nodes[i];
    mycoords_global[i] = mycoords_nodes[i] * dims_smp[i] + mycoords_smp[i];
}

myrank_global = mycoords_global[0];
for (i=1; i<ndims; i++) {
    myrank_global = myrank_global * dims_global[i] + mycoords_global[i];
}

MPI_Comm_split (comm_orig, /*color*/ 0, myrank_global, &comm_global_flat);
MPI_Cart_create (comm_global_flat, ndims, dims_global, periods_global, 0, &comm_global_cart);
MPI_Comm_free (&comm_global_flat);

// Result:
// Input was:
// comm_orig, ndims, periods_global
// Result is:
// comm_smp_cart, size_smp, myrank_smp, dims_smp, periods_smp, my_coords_smp,
// comm_nodes_cart, size_nodes, myrank_nodes, dims_nodes, periods_nodes, my_coords_nodes,
// comm_global_cart, size_global, myrank_global, dims_global, my_coords_global
How to achieve a hierarchical DD for unstructured grids?

- **Unstructured grids:**
  - Single-level DD (finest level)
    - Analysis of the communication pattern in a first run (with only a few iterations)
    - Optimized rank mapping to the hardware before production run
    - E.g., with CrayPAT + CrayApprentice
  - Multi-level DD:
    - **Top-down:** Several levels of (Par)Metis
      → unbalanced communication
      → demonstrated on next (skipped) slide
    - **Bottom-up:** Low level DD
      + higher level recombination
      → based on DD of the grid of subdomains
Top-down – several levels of (Par)Metis

Steps:
- Load-balancing (e.g., with ParMetis) on outer level, i.e., between all SMP nodes
- Independent (Par)Metis inside of each node
- Metis inside of each socket
  - Subdivide does not care on balancing of the outer boundary
  - processes can get a lot of neighbors with inter-node communication
  - unbalanced communication
Bottom-up –

Multi-level DD through recombination

1. Core-level DD: partitioning of application’s data grid
2. Numa-domain-level DD: recombining of core-domains
3. SMP node level DD: recombining of socket-domains

Problem:
Recombination must not calculate patches that are smaller or larger than the average

In this example the load-balancer must combine always exactly
- 6 cores, and
- 4 numa-domains (i.e., sockets or dies)

Advantage:
Communication is balanced!
Profiling solution

- First run with profiling
  - Analysis of the communication pattern
- Optimization step
  - Calculation of an optimal mapping of ranks in MPI_COMM_WORLD to the hardware grid (physical cores / sockets / SMP nodes)
- Restart of the application with this optimized locating of the ranks on the hardware grid

- Example: CrayPat and CrayApprentice
Remarks on Cache Optimization

- After all parallelization domain decompositions (DD, up to 3 levels) are done:
  - Cache-blocking is an additional DD into data blocks
    - Blocks fulfill size conditions for optimal spatial/temporal locality
    - It is done inside of each MPI process (on each core).
    - Outer loops run from block to block
    - Inner loops inside of each block
    - Cartesian example: 3-dim loop is split into
      
      \[
      \begin{align*}
      &\text{do } i_{\text{block}}=1,ni,\text{stride}_i \\
      &\quad \text{do } j_{\text{block}}=1,nj,\text{stride}_j \\
      &\quad \quad \text{do } k_{\text{block}}=1,nk,\text{stride}_k \\
      &\quad \quad \quad \quad \text{do } i=i_{\text{block}},\min(i_{\text{block}}+\text{stride}_i-1, ni) \\
      &\quad \quad \quad \quad \quad \text{do } j=j_{\text{block}},\min(j_{\text{block}}+\text{stride}_j-1, nj) \\
      &\quad \quad \quad \quad \quad \quad \text{do } k=k_{\text{block}},\min(k_{\text{block}}+\text{stride}_k-1, nk) \\
      &\quad \quad \quad \quad \quad \quad \quad a(i,j,k) = f( b(i\pm0,1,2, j\pm0,1,2, k\pm0,1,2) ) \\
      \end{align*}
      \]
      
      … … … end do

      Access to 13-point stencil

See SC’15 tutorial:
Node-Level Performance Engineering
Monday, Nov. 16, room 18C
Scalability of pure MPI

- As long as the application does **not** use
  - `MPI_ALLTOALL`
  - `MPI_<collectives>V` (i.e., with length arrays)
  and application
  - distributes all data arrays
  one can expect:
    - Significant, but still scalable memory overhead for halo cells.
    - MPI library is internally scalable:
      - E.g., **mapping ranks → hardware grid**
        - Centralized storing in shared memory (OS level)
        - In each MPI process, only used neighbor ranks are stored (cached) in process-local memory.
      - **Tree based algorithm with O(log N)**
        - From 1000 to 1000,000 process O(Log N) only doubles!
To overcome MPI scaling problems

- MPI has a few scaling problems
  - Handling of more than 10,000 MPI processes
  - Irregular Collectives: MPI_...v(), e.g. MPI_Gatherv()
    - Scaling applications should not use MPI_...v() routines
  - MPI-2.1 Graph topology (MPI_Graph_create)
    - MPI-2.2 MPI_Dist_graph_create_adjacent
  - Creation of sub-communicators with MPI_Comm_create
    - MPI-2.2 introduces a new scaling meaning of MPI_Comm_create

- Hybrid programming reduces all these problems (due to a smaller number of processes)
Pinning of MPI processes

- Pinning is helpful for all programming models
- Highly system-dependent!
- Intel MPI: env variable I_MPI_PIN
- OpenMPI: mpirun options –bind-to-core, -bind-to-socket, -bycore, -byslot …
Anarchy vs. affinity with a heat equation solver

- Reasons for caring about affinity:
  - Eliminating performance variation
  - Making use of architectural features
  - Avoiding resource contention

No affinity settings \(\rightarrow\) high variation

With affinity, physical cores, filling left socket first:
mpirun -bind-to-core -byslot ...
Pure MPI: Main advantages

- Simplest programming model
- Library calls need not to be thread-safe
- The hardware is typically prepared for many MPI processes per SMP node
- Only minor problems if pinning is not applied
Pure MPI: Main disadvantages

- Unnecessary communication
- Too much memory consumption for
  - Halo data for communication between MPI processes on same SMP node
  - Other replicated data on same SMP node
  - MPI buffers due to the higher number of MPI processes
- Additional programming costs for minimizing node-to-node communication,
  - i.e. for optimizing the communication topology
- No efficient use of hardware-threads (hyper-threads)
Pure MPI: Conclusions

- Still a good programming model for small and medium size applications.
- Major problem may be memory consumption
Hybrid Parallel Programming

Programming models
- MPI + MPI-3.0
- shared memory
Hybrid MPI + MPI-3 shared memory

Advantages

- No message passing inside of the SMP nodes
- Using only one parallel programming standard
- No OpenMP problems (e.g., thread-safety isn’t an issue)

Major Problems

- Communicator must be split into shared memory islands
- To minimize shared memory communication overhead:
  Halos (or the data accessed by the neighbors) must be stored in
  MPI shared memory windows
- Same work-sharing as with pure MPI
- MPI-3.0/3.1 shared memory synchronization waits for some clarification \(\rightarrow\) MPI-4.0
MPI-3 shared memory

- Split main communicator into shared memory islands
  - \texttt{MPI\_Comm\_split\_type}
- Define a shared memory window on each island
  - \texttt{MPI\_Win\_allocate\_shared}
  - Result (by default):
    - contiguous array, directly accessible by all processes of the island
- Accesses and synchronization
  - Normal assignments and expressions
  - No \texttt{MPI\_PUT/GET} !
  - Normal MPI one-sided synchronization, e.g., \texttt{MPI\_WIN\_FENCE}
Splitting the communicator & contiguous shared memory allocation

Contiguous shared memory window within each SMP node

MPI_Aint /*IN*/ local_window_count; double /*OUT*/ *base_ptr;

MPI_Comm comm_all, comm_sm; int my_rank_all, my_rank_sm, size_sm, disp_unit;

MPI_Comm_rank (comm_all, &my_rank_all);

MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
MPI_INFO_NULL, &comm_sm);

MPI_Comm_rank (comm_sm, &my_rank_sm); MPI_Comm_size (comm_sm, &size_sm);

disp_unit = sizeof(double); /* shared memory should contain doubles */

MPI_Win_allocate_shared (local_window_count*disp_unit, disp_unit, MPI_INFO_NULL,
comm_sm, &base_ptr, &win_sm);

In Fortran, MPI-3.0, page 341, Examples 8.1 (and 8.2) show how to convert buf_ptr into a usable array a.

This mapping is based on a sequential ranking of the SMP nodes in comm_all.
Within each SMP node – Essentials

- The allocated shared memory is contiguous across process ranks,
  i.e., the first byte of rank i starts right after the last byte of rank i-1.
- Processes can calculate remote addresses’ offsets with local information only.
- Remote accesses through load/store operations,
  i.e., without MPI RMA operations (MPI_GET/PUT, …)
- Although each process in comm_sm accesses the same physical memory, the virtual start address of the whole array may be different in all processes!
  → **linked lists** only with offsets in a shared array, but **not with binary pointer addresses**!

Following slides show only the shared memory accesses, i.e., communication between the SMP nodes is not presented.
**Shared memory access example**

Contiguous shared memory window within each SMP node

```c
MPI_Aint /*IN*/ local_window_count; double /*OUT*/ *base_ptr;
MPI_Win_allocate_shared (local_window_count*disp_unit, disp_unit, MPI_INFO_NULL, comm_sm, &base_ptr, &win_sm);
MPI_Win_fence (0, win_sm); /*local store epoch can start*/
for (i=0; i<local_window_count; i++) base_ptr[i] = ... /* fill values into local portion */
MPI_Win_fence (0, win_sm); /* local stores are finished, remote load epoch can start */
if (my_rank_sm > 0) printf("left neighbor’s rightmost value = %lf \n", base_ptr[-1]);
if (my_rank_sm < size_sm-1) printf("right neighbor’s leftmost value = %lf \n", base_ptr[local_window_count]);
```

In Fortran, before and after the synchronization, one must add: CALL MPI_F_SYNC_REG (buffer) to guarantee that register copies of buffer are written back to memory, respectively read again from memory.
Establish comm_sm, comm_nodes, comm_all, if SMPs are not contiguous within comm_orig

MPI_Comm_split_type (comm_orig, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm);
MPI_Comm_size (comm_sm, &size_sm); MPI_Comm_rank (comm_sm, &my_rank_sm);
MPI_Comm_split (comm_orig, my_rank_sm, 0, &comm_nodes);
MPI_Comm_size (comm_nodes, &size_nodes);
if (my_rank_sm==0) {
    MPI_Comm_rank (comm_nodes, &my_rank_nodes);
    MPI_Exscan (&size_sm, &my_rank_all, 1, MPI_INT, MPI_SUM, comm_nodes);
    if (my_rank_nodes == 0) my_rank_all = 0;
}
MPI_Comm_free (&comm_nodes);
MPI_Bcast (&my_rank_nodes, 1, MPI_INT, 0, comm_sm);
MPI_Comm_split (comm_orig, my_rank_sm, my_rank_nodes, &comm_nodes);
MPI_Bcast (&my_rank_all, 1, MPI_INT, 0, comm_sm); my_rank_all = my_rank_all + my_rank_sm;
MPI_Comm_split (comm_orig, /*color*/ 0, my_rank_all, &comm_all);
Alternative: Non-contiguous shared memory

- Using info key "alloc_shared_noncontig"
- MPI library can put processes’ window portions
  - on page boundaries,
    - (internally, e.g., only one OS shared memory segment with some unused padding zones)
  - into the local ccNUMA memory domain + page boundaries
    - (internally, e.g., each window portion is one OS shared memory segment)

Pros:
- Faster local data accesses especially on ccNUMA nodes

Cons:
- Higher programming effort for neighbor accesses: MPI_WIN_SHARED_QUERY

Further reading:
Non-contiguous shared memory allocation

- Non-contiguous shared memory window within each SMP node
- local_window_count doubles
- base_ptr
- MPI process
- Sub-communicator for one SMP node

```
MPI_Aint /*IN*/ local_window_count;  double /*OUT*/ *base_ptr;
disp_unit = sizeof(double);  /* shared memory should contain doubles */
MPI_Info info_noncontig;
MPI_Info_create (&info_noncontig);
MPI_Info_set (info_noncontig, "alloc_shared_noncontig", "true");
MPI_Win_allocate_shared (local_window_count*disp_unit, disp_unit, info_noncontig, comm_sm, &base_ptr, &win_sm);
```
Non-contiguous shared memory: Neighbor access through MPI_WIN_SHARED_QUERY

- Each process can retrieve each neighbor’s base_ptr with calls to MPI_WIN_SHARED_QUERY
- Example: only pointers to the window memory of the left & right neighbor

```c
if (my_rank_sm > 0) MPI_Win_shared_query (win_sm, my_rank_sm - 1, &win_size_left, &disp_unit_left, &base_ptr_left);
if (my_rank_sm < size_sm - 1) MPI_Win_shared_query (win_sm, my_rank_sm + 1, &win_size_right, &disp_unit_right, &base_ptr_right);
...
MPI_Win_fence (0, win_sm); /* local stores are finished, remote load epoch can start */
if (my_rank_sm > 0) printf("left neighbor’s rightmost value = %lf \n", base_ptr_left[ win_size_left/disp_unit_left - 1 ]);
if (my_rank_sm < size_sm - 1) printf("right neighbor’s leftmost value = %lf \n", base_ptr_right[ 0 ]);
```

Thanks to Steffen Weise (TU Freiberg) for testing and correcting the example codes.
Other technical aspects with MPI_WIN_ALLOCATE_SHARED

**Caution:** On some systems
- the number of shared memory windows, and
- the total size of shared memory windows may be limited.

Some OS systems may provide options, e.g.,
- at job launch, or
- MPI process start,
to enlarge restricting defaults.

If MPI shared memory support is based on POSIX shared memory:
- Shared memory windows are located in memory-mapped /dev/shm
- Default: 25% or 50% of the physical memory, but a maximum of ~2043 windows!
- Root may change size with: `mount -o remount,size=6G /dev/shm`.

Cray XT/XE/XC (XPMEM): No limits.

On a system without virtual memory (like CNK on BG/Q), you have to reserve a chunk of address space when the node is booted (default is 64 MB).

Thanks to Jeff Hammond and Jed Brown (ANL), Brian W Barrett (SANDIA), and Steffen Weise (TU Freiberg), for input and discussion.
Splitting the communicator without MPI_COMM_SPLIT_TYPE

Alternatively, if you want to group based on a fixed amount size_sm of shared memory cores in comm_all:

- Based on sequential ranks in comm_all
- Pro: comm_sm can be restricted to ccNUMA locality domains
- Con: MPI does not guarantee MPI_WIN_ALLOCATE_SHARED() on whole SMP node (MPI_COMM_SPLIT_TYPE() may return MPI_COMM_SELF or partial SMP node)

MPI_Comm_rank (comm_all, &my_rank);
MPI_Comm_split (comm_all, /*color*/ my_rank / size_sm, 0, &comm_sm);
MPI_Win_allocate_shared (...);

To guarantee shared memory, one may add an additional

MPI_Comm_split_type (comm_sm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm_really);

Input from outside
Hybrid Parallel Programming

Slide 73 / 198

Rabenseifner, Hager, Jost

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators

Internode: Irecv + Send

<table>
<thead>
<tr>
<th>Latency</th>
<th>Accumulated inter-node bandwidth per node</th>
<th>Additional intra-node communication with:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9 µs, 4.4 GB/s</td>
<td>Irecv+send</td>
<td>Pure MPI</td>
</tr>
<tr>
<td>3.4 µs, 4.4 GB/s</td>
<td>MPI-3.0 store</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>3.0 µs, 4.5 GB/s</td>
<td>Irecv+send</td>
<td></td>
</tr>
<tr>
<td>3.0 µs, 4.6 GB/s</td>
<td>MPI-3.0 store</td>
<td></td>
</tr>
<tr>
<td>3.3 µs, 4.4 GB/s</td>
<td>Irecv+send</td>
<td></td>
</tr>
<tr>
<td>3.5 µs, 4.4 GB/s</td>
<td>MPI-3.0 store</td>
<td></td>
</tr>
<tr>
<td>5.2 µs, 4.3 GB/s</td>
<td>Irecv+send</td>
<td></td>
</tr>
<tr>
<td>5.2 µs, 4.4 GB/s</td>
<td>MPI-3.0 store</td>
<td></td>
</tr>
<tr>
<td>10.3 µs, 4.5 GB/s</td>
<td>Irecv+send</td>
<td></td>
</tr>
<tr>
<td>10.1 µs, 4.5 GB/s</td>
<td>MPI-3.0 store</td>
<td></td>
</tr>
</tbody>
</table>

Measurements: bi-directional halo exchange in a ring with 4 SMP nodes
(with 16 and 512kB per message; bandwidth: each message is counted only once, i.e., not twice at sender and receiver) on Cray XC30 with Sandybridge @ HLRS

Conclusion: No win through MPI-3.0 shared memory programming
1 MPI process *versus* several MPI processes
(1 Intel Xeon Phi per node)

### 1 MPI process per Intel Xeon Phi
Intel Xeon Phi + Infiniband beacon @ NICS

- Latency: 15 µs, Accumulated inter-node bandwidth per Phi: 0.83 GB/s
- Links per Phi: 1x

- Latency: 26 µs, Accumulated inter-node bandwidth per Phi: 0.87 GB/s
- Links per Phi: 2x

- Latency: 25 µs, Accumulated inter-node bandwidth per Phi: 0.91 GB/s
- Links per Phi: 4x

- Latency: 23 µs, Accumulated inter-node bandwidth per Phi: 0.91 GB/s
- Links per Phi: 8x

- Latency: 24 µs, Accumulated inter-node bandwidth per Phi: 0.92 GB/s
- Links per Phi: 16x

- Latency: 21 µs, Accumulated inter-node bandwidth per Phi: 0.91 GB/s
- Links per Phi: 30x

- Latency: 51 µs, Accumulated inter-node bandwidth per Phi: 0.90 GB/s
- Links per Phi: 60x

### 4 MPI processes per Intel Phi

- Latency: 19 µs, Accumulated inter-node bandwidth per Phi: 0.54 GB/s
- Links per Phi: 1x

- Latency: 25 µs, Accumulated inter-node bandwidth per Phi: 0.52 GB/s
- Links per Phi: 2x

- Latency: 15 µs, Accumulated inter-node bandwidth per Phi: 0.83 GB/s
- Links per Phi: 3x

- Latency: 25 µs, Accumulated inter-node bandwidth per Phi: 0.91 GB/s
- Links per Phi: 4x

- Latency: 23 µs, Accumulated inter-node bandwidth per Phi: 0.91 GB/s
- Links per Phi: 8x

- Latency: 24 µs, Accumulated inter-node bandwidth per Phi: 0.92 GB/s
- Links per Phi: 16x

- Latency: 21 µs, Accumulated inter-node bandwidth per Phi: 0.91 GB/s
- Links per Phi: 30x

- Latency: 51 µs, Accumulated inter-node bandwidth per Phi: 0.90 GB/s
- Links per Phi: 60x

**Similar Conclusion:**
- Several MPI processes inside Phi (in a line) cause slower communication
- No win through MPI-3.0 shared memory programming
Hybrid shared/cluster programming models

- MPI on each core (not hybrid)
  - Halos between all cores
  - MPI uses internally shared memory and cluster communication protocols

- MPI+OpenMP
  - Multi-threaded MPI processes
  - Halos communication only between MPI processes

- MPI cluster communication + MPI shared memory communication
  - Same as “MPI on each core”, but
  - within the shared memory nodes, halo communication through direct copying with C or Fortran statements

- MPI cluster comm. + MPI shared memory access
  - Similar to “MPI+OpenMP”, but
  - shared memory programming through work-sharing between the MPI processes within each SMP node
Halo Copying within SMP nodes

MPI process use halos:

- Communication overhead depends on communication method
  - (Nonblocking) message passing (since MPI-1)
  - One-sided communication (typically not faster, since MPI-2.0)
  - MPI_Neighbor_alltoall (since MPI-3.0)

- Shared memory remote loads or stores (since MPI-3.0)
  - Point-to-point synchronization for shared memory requires MPI_Win_sync
    -> next slides
  - Benchmarks on halo-copying inside of an SMP node
    - On Cray XE6: Fastest is shared memory copy
      + point-to-point synchronization with zero-length msg

-> end of this section
Exercise 1: Shared memory ring communication

- Use the given program as your baseline for the following exercise:
  
  ```
  cp ~/MPI/course/C/1sided/ring_1sided_put_win_alloc.c my_shared_exa2.c 
  or 
  cp ~/MPI/course/F_30/1sided/ring_1sided_put_win_alloc_30.f90 my_shared_exa2.f90
  (or F_20 ................................................................. _20 .... with mpi module)
  ```

- Tasks: Substitute the distributed window by a shared window
  
  - Substitute `MPI_Alloc_mem+MPI_Win_create` by `MPI_Win_allocate_shared`
  - Do not forget to also remove the `MPI_Free_mem`
  
  - Substitute the `MPI_Put` by a direct assignment:
    - `*rcv_buf_ptr` is the local `rcv_buf`
    - The `rcv_buf` of the right neighbor can be accessed through the word-offset `+1` in the direct assignment `*(rcv_buf_ptr+(offset)) = snd_buf`
    - In the ring, the word-offset `+1` should be expressed with `(right – my_rank)`
    - Fortran: Be sure that that you add additional calls to `MPI_F_SYNC_REG` between both `MPI_Win_fence` and your direct assignment, i.e., directly before and after `rcv_buf(1+(offset)) = snd_buf`

- Compile and run shared memory program
  
  - With MPI processes on 4 cores & all cores of a shared memory node

Problem with MPI-3.0 and MPI-3.1: The role of assertions in RMA synchronization used for direct shared memory accesses (i.e., without RMA calls) is not clearly defined! Implication: `MPI_Win_fence should be used, but only with assert = 0`. (State March 01, 2015)
Summary of halo files (and some ring files)

- ring.c
- halo_isend_recv.c
- halo_irecv_send.c
- halo_1sided_put.c
- halo_1sided_put_alloc_mem.c
- halo_1sided_put_win_alloc.c
- halo_1sided_store_win_alloc_shared.c
- halo_1sided_store_win_alloc_shared_w-a-cray.c
- halo_1sided_store_win_alloc_shared_query.c
- halo_1sided_store_win_alloc_shared_query_w-a-cray.c
- halo_1sided_store_win_alloc_shared_pscw.c
- halo_1sided_store_win_alloc_shared_onesync.c
- halo_1sided_store_win_alloc_shared_signal.c
- ring_1sided_get.c & ring_1sided_put.c● ring_1sided_exa2.c
- ring_allreduce.c
- derived_20.c
- topology_ring.c
- ring_neighbor_alltoall.c
dito.
- ring_neighbor_alltoallw.c
dito.
- ring_1sided_put_win allocating.c
- ring_1sided_store_win_alloc_shared.c
- ring_1sided_store_win_alloc_shared_w-a-cray.c
- ring_1sided_store_win_alloc_shared_query_w-a-cray.c
- ring_1sided_store_win_alloc_shared_pscw.c
- ring_1sided_store_win_alloc_shared_onesync.c
- ring_1sided_store_win_alloc_shared_signal.c

Chap. 11 Shared Mem., Exercise 1
Chap. 11 Shared Mem., Part (2), Exercise 3
Chap. 11 Shared Mem., Part (2), Exercise 4
Chap. 11 Shared Mem., Part (2), Exercise 5: Benchmarking all solutions

See also login-slides
Exercise 2 (advanced): Halo communication with MPI_Put

- Copy to your local directory and analyze the source code:
  ```
  cp ~/MPI/course/C/1sided/halo_1sided_put_win_alloc.c ./
  cp ~/MPI/course/F_30/1sided/halo_1sided_put_win_alloc_30.f90 ./
  ```
  (or _20 with mpi module)

- Halo... communicates along the 1-dim ring of processes in both directions
  - **Into right direction**: Put `snd_buf_right` into the `rec_buf_left` of the right neighbor
  - **Into left direction**: Put `snd_buf_left` into the `rec_buf_right` of the left neighbor

- Compile and run the original halo_1sided_put_win_alloc*.c/f90 program
  - With MPI processes on 4 cores & all cores of a shared memory node

```
left=my_rank-1  my_rank  right=my_rank+1

MPI_Put → rcv_buf... Window – accessible from the neighbor through RMA calls
```

Hybrid Parallel Programming
Slide 79 / 198
Rabenseifner, Hager, Jost
Chap.11 Shared Memory 1-Sided
(Exercise 2a: Shared memory **halo** communication)

- Copy to your local directory:
  ```
  cp ~/MPI/course/C/1sided/halo_1sided_store_win_alloc_shared.c ./
  cp ~/MPI/course/F_30/1sided/halo_1sided_store_win_alloc_shared_30.f90 ./
  ```
  (or _20 with mpi module)

- Compile and run shared memory program
  - With MPI processes on **4** cores & **all cores** of a shared memory node

- Compare latency and bandwidth

- Compare the source codes (with “diff”)
(Exercise 2b: Shared memory halo communication)

- Use the given program as your baseline for the following exercise:
  - `cp halo_1sided_put_win_alloc.c my_shared_exa2.c` or `...20.f90` or `...30.f90`

- Tasks: Substitute the distributed window by a shared window
  - Substitute `MPI_Mem_alloc+MPI_Win_allocate` by `MPI_Win_allocate_shared`
  - Substitute both `MPI_Put` by direct assignments:
    - `rcv_buf_right[i]` of the left neighbor can be now accessed directly through own `rcv_buf_right` as `rcv_buf_right[i+offset_left]` with `offset_xxx = (xxx – my_rank) * max_length`.
    - `xxx` = `left` or `right`. The formula is correct for any rank.
    - `max_length` is the number of elements in the window of each process.
    - Fortran: Be sure that that you add additional calls to `MPI_F_SYNC_REG` between both `MPI_Win_fence` and your direct assignment, i.e., directly before and after `rcv_buf...(...+offset_xxx : ...+offset_xxx) = snd_buf...(... : ...)`

- Compile and run shared memory program
  - With MPI processes on 4 cores & all cores of a shared memory node

Problem with MPI-3.0 and MPI-3.1: The role of assertions in RMA synchronization used for direct shared memory accesses (i.e., without RMA calls) is not clearly defined!
Implication: `MPI_Win_fence` should be used, but only with `assert = 0`. (State March 01, 2015)
Details on MPI-3.0 and 3.1: Two memory models

- Query for new attribute to allow applications to tune for cache-coherent architectures
  - Attribute MPI_WIN_MODEL with values
    - MPI_WIN_SEPARATE model
    - MPI_WIN_UNIFIED model on cache-coherent systems

- Shared memory windows always use the MPI_WIN_UNIFIED model
  - Public and private copies are **eventually** synchronized without additional RMA calls
    (MPI-3.0/MPI-3.1, Section 11.4, page 436/435 lines 37-40/43-46)
  - For synchronization **without delay**: MPI_WIN_SYNC()
    (MPI-3.1 Section 11.7: “Advice to users. In the unified memory model…” on page 456, and Section 11.8, Example 11.21 on pages 468-469)
  - or any other RMA synchronization:
    “A consistent view can be created in the unified memory model (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH).”
    (MPI-3.0/MPI-3.1, MPI_Win_allocate_shared, page 410/408, lines 16-20/43-47)
“eventually synchronized“ – the Problem

- The problem with shared memory programming using libraries is:

  X is a variable in a shared window initialized with 0.

  Process
  Rank 0
  X = 1

  MPI_Send(empty msg to rank 1) → MPI_Recv(from rank 0)

  printf … X

  X can be still 0, because the “1” will be eventually visible to the other process, i.e., the “1” will be visible but maybe too late ☹️ ☹️ ☹️

Or any other process-to-process synchronization, e.g., using also shared memory stores and loads.
“eventually synchronized” – the Solution

• A pair of local memory fences is needed:

X is a variable in a shared window initialized with 0.

<table>
<thead>
<tr>
<th>Process</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank 0</td>
<td>Rank 1</td>
</tr>
<tr>
<td>X = 1</td>
<td></td>
</tr>
</tbody>
</table>

X is a variable in a shared window initialized with 0.

```c
// Process Rank 0
X = 1
MPI_Send(empty msg to rank 1)

// Process Rank 1
MPI_Recv(from rank 0)
printf ... X
```

Now, it is guaranteed that the “1” in X is visible in this process.
"eventually synchronized“ – Last Question

How to program the local memory fence?

- C11 atomic_thread_fence(order)
  - Advantage: one can choose appropriate order = memory_order_acquire, or ..._release to achieve minimal latencies

- MPI_Win_sync
  - Advantage: works also for Fortran
  - Disadvantage: may be slower than C11 atomic_thread_fence with appro. order

- Using RMA synchronization with integrated local memory fence instead of MPI_Send → MPI_Recv
  - Advantage: May prevent double fences
  - Disadvantage: The synchronization itself may be slower

X is a variable in a shared window initialized with 0.

\[
\begin{align*}
X &= 1 \\
\text{local memory fence} \\
\text{MPI_Send(} \text{empty msg to rank } 1) &\rightarrow \text{MPI_Recv(from rank } 0) \\
\text{local memory fence} \\
\text{printf } \ldots X
\end{align*}
\]
General MPI-3 shared memory synchronization rules
(based on MPI-3.1, MPI_Win_allocate_shared, page 408, lines 43-47: “A consistent view …”)

Defining Proc 0 Proc 1
Sync-from → Sync-to

being
MPI_Win_post
or
MPI_Win_complete
or
MPI_Win_fence
or
MPI_Win_sync
Any-process-sync
or
MPI_Win_unlock

and having ...

A=val_1
Sync-from → Sync-to
load(A)

⇒ … the load(A) in P1 loads val_1
(this is the write-read-rule)

load(B)
Sync-from → Sync-to
B=val_2

⇒ … the load(B) in P0 is not affected by the store of val_2 in P1
(read-write-rule)

C=val_3
Sync-from → Sync-to
C=val_4
load(C)

⇒ … that the load(C) in P1 loads val_4
(write-write-rule)

1) Must be paired according to the general on-sided synchronization rules.
2) "Any-process-sync" may be done with methods from MPI (e.g. with send --> recv as in MPI-3.1 Example 11.21, but also with some synchronization through MPI shared memory loads and stores, e.g. with C++11 atomic loads and stores).
3) No rule for MPI_Win_flush (according current forum discus.)
Conclusion: Best latency and bandwidth with shared memory store together with point-to-point synchronization (→ next slide)

Further opportunities by purely synchronizing with C++11 methods

High bandwidth direct shared memory store

Medium bandwidth point-to-point and neighbor alltoall

Low bandwidth with MPI_Put

Low latency pt-to-pt synchronization (→ next slide)

High latency MPI_Win_fence
Other synchronization on MPI-3.0 shared memory

- If the shared memory data transfer is done without RMA operation, then the synchronization can be done by other methods.
- This example demonstrates the rules for the unified memory model if the data transfer is implemented only with load and store (instead of MPI_PUT or MPI_GET) and the synchronization between the processes is done with MPI communication (instead of RMA synchronization routines).

Process A

MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK, win)

DO ... 
X=...

MPI_F_SYNC_REG(X)\(^1\)

MPI_WIN_SYNC(win)

MPI_Send

MPI_Recv

MPI_WIN_SYNC(win)

MPI_F_SYNC_REG(X)\(^1\)

print X

MPI_SEND

MPI_WIN_UNLOCK_ALL(win)

END DO

\(^1\) Fortran only.

Process B

MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK, win)

DO ...

MPI американская

Data exchange in this direction, therefore MPI_WIN_SYNC is needed in both processes:

Write-read-rule

MPI_Recv

MPI_WIN_SYNC(win)

MPI_F_SYNC_REG(X)\(^1\)

print X

MPI_SEND

MPI_WIN_UNLOCK_ALL(win)

Also needed due to read-write-rule

• The used synchronization must be supplemented with MPI_WIN_SYNC, which acts only locally as a processor-memory-barrier. For MPI_WIN_SYNC, a passive target epoch is established with MPI_WIN_LOCK_ALL.

• X is part of a shared memory window and should be the same memory location in both processes.
Shared memory problems (1/2)

- **Race conditions**
  - as with OpenMP or any other shared memory programming models
  - Data-Race: *Two processes access the same shared variable and at least one process modifies the variable and the accesses are concurrent, i.e. unsynchronized, i.e., in is not define which access is first*
    - The outcome of a program depends on the detailed timing of the accesses
    - This is often caused by unintended access to the same variable, or missing memory fences
Shared memory problems (2/2)

- **Cache-line false-sharing**
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory

- Several processes are accessing shared data through the same cache-line.
- This cache-line has to be moved between these processes (cache coherence protocol).
- This is very time-consuming.
**MPI communication & MPI-3.0 Shared Memory on Intel Phi**

- **MPI-3.0 shared memory accesses inside of an Intel Phi:**
  - They work, but
  - MPI communication may be faster than user-written loads and stores.

- **Communication of MPI processes inside of an Intel Phi:**
  (bi-directional halo exchange benchmark with all processes in a ring; bandwidth: each message is counted only once, i.e., not twice at sender and receiver)

<table>
<thead>
<tr>
<th>Number of MPI processes</th>
<th>Latency (µs)</th>
<th>Bandwidth (GB/s)</th>
<th>Shared mem. bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>0.80</td>
<td>0.25</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>0.75</td>
<td>0.24</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>0.66</td>
<td>0.24</td>
</tr>
<tr>
<td>60</td>
<td>29</td>
<td>0.50</td>
<td>0.22</td>
</tr>
<tr>
<td>120</td>
<td>149</td>
<td>0.19</td>
<td>0.20</td>
</tr>
<tr>
<td>240</td>
<td>745</td>
<td>0.05</td>
<td>0.05</td>
</tr>
</tbody>
</table>

**Conclusion:**

MPI on Intel Phi works fine on up to 60 processes, but the 4 hardware threads per core require OpenMP parallelization.
MPI+MPI-3.0 shared mem: Main advantages

- A new method for replicated data
  - To allow only one replication per SMP node
- Interesting method for direct access to neighbor data (without halos!)
- A new method for communicating between MPI processes within each SMP node
- On some platforms significantly better bandwidth than with send/recv
- Library calls need not be thread-safe
MPI+MPI-3.0 shared mem: Main disadvantages

- Synchronization is defined, but still under discussion
- Similar problems as with all library based shared memory (e.g., pthreads)
- Does not reduce the number of MPI processes
- The meaning of the assertions for shared memory is still undefined
MPI+MPI-3.0 shared mem: Conclusions

- Add-on feature for pure MPI
- Opportunity for reducing communication within SMP nodes
- Opportunity for reducing memory consumption (halos & replicated data)
Exercise 3: Ring – Using *other* synchronization

- Use your exa1 result or
  - ~/MPI/course/C/1sided/ring_1sided_store_win_alloc_shared.c
  - ~/MPI/course/F_30/1sided/ring_1sided_store_win_alloc_shared_30.f90 (or _20)
  as your baseline *my_shared_exa3.c* or ..._20.f90 or ..._30.f90 for the following exercise:

- Tasks: Substitute the MPI_Fence synchronization by pt-to-pt communication
  - Use empty messages for synchronizing
  - Substitute the first MPI_Fence by ring-communication to the *left*, because it signals to the *left* neighbor that the local rcv_buf target is exposed for new data
    - MPI_Irecv(...right,...,&rq) ; MPI_Send(...left, ...); MPI_Wait(&rq ...);
  - Substitute the second MPI_Fence by ring-communication to the *right*, because it signals to the *right* neighbor that data is stored in the rcv_buf of the right neighb.
  - Local MPI_Win_sync is needed for write-read and read-write-rule

- Compile and run shared memory program
  - With MPI processes on 4 cores & *all cores* of a shared memory node
Exercise 3: Ring – Using *other* synchronization

- Communication pattern between each pair of neighbor processes

**Process rank n**

- *Receive signal* `win_sync`
- *Write* `snd_buf` data to the `rcv_buf` in right neighbor
- *Send signal to right neighbor* `win_sync`

**Process rank n+1**

- Old data in `rcv_buf` is *read*, therefore the local `rcv_buf` is now exposed for new data
- *Send signal to left neighbor* `win_sync`
- Data is stored in the `rcv_buf` and can be read after subsequent synchronization
- *Receive signal* `win_sync`
- Now, data in `rcv_buf` can be locally *read*

---

Fortran: Are these the locations where MPI_F_SYNC_REG is needed?

See also login-slides.
Exercise 3b (advanced): **Halo** – Using *other* synchronization

- Use your exa1 result or
  - */MPI/course/C/1sided/halo_1sided_store_win_alloc_shared.c*
  - */MPI/course/F_30/1sided/halo_1sided_store_win_alloc_shared_30.f90* (or _20) as your baseline for the following exercise:
    - `cp my_shared_exa1.c my_shared_exa2.c` or ..._20.f90 or ..._30.f90

- Tasks: Substitute the MPI_Fence synchronization by pt-to-pt communication
  - Use empty messages for synchronizing
  - Substitute each pair of MPI_Fence by
    - `MPI_Irecv(...right,...,rq[1] ...) ; MPI_Irecv(...left, ...,rq[2] ...) ;`
    - `MPI_Send(...left, ...) ; MPI_Send(...right, ...) ; MPI_Waitall(2,rq ...);`
  - Local MPI_Win_sync is needed for write-read and read-write-rule

- Compile and run shared memory program
  - With MPI processes on 4 cores & *all cores* of a shared memory node
Exercise 4 (advanced): Ring – with memory signals

- **Goal:**
  - Substitute the Irecv-Send-Wait communication by two shared memory flags

- **Hints:**
  - After initializing these shared memory variables with 0, an additional MPI_Win_sync + MPI_Barrier + MPI_Win_sync is needed
  - Normally, from three consecutive MPI_Win_sync, only one call may be needed, because one memory fence is enough

- **Recommendation:**
  - One may study and run both the solution files and compare the latency
    - `halo_1sided_store_win_alloc_shared_signal.c` (only solution in C)
    - `ring_1sided_store_win_alloc_shared_signal.c` (only solution in C)
Exercise 4 (advanced): Ring – with memory signals

Process rank n

old data in rcv_buf is **read**

win_sync

1

Atomic (or volatile) store

Process rank n+1

while (A==0) IDLE

A = 0

**win_sync**(A)

**win_sync**(B)

→ B is now locally 0

**win_sync**

**write** snd_buf data
to the rcv_buf in right neighbor

**win_sync**

1

data is stored in the rcv_buf and can be read after subsequent synchronization

signal B

while (B==0) IDLE

B = 0

**win_sync**(B)

**win_sync**(A)

→ A is now locally 0

**win_sync**

now, data in rcv_buf can be locally **read**
Exercise 5 (advanced): Halo communication benchmarking

- **Goal:**
  - Learn about the communication latency and bandwidth on your system

- **Method:**
  - `cp MPI/course/C/1sided/halo* .`
  - On a shared or distributed memory, run and compare:
    - `halo_irecv_send.c`
    - `halo_isend_recv.c`
    - `halo_neighbor_alltoall.c`
    - `halo_1sided_put.c`
    - `halo_1sided_put_alloc_mem.c`
    - `halo_1sided_put_win_alloc.c`
    - Different communication methods

  - And run and compare on a shared memory only:
    - `halo_1sided_store_win_alloc_shared.c`
    - `halo_1sided_store_win_alloc_shared_query.c` (with `alloc_shared_noncontig`)
    - `halo_1sided_store_win_alloc_shared_pscw.c`
    - `halo_1sided_store_win_alloc_shared_thersync.c`
    - `halo_1sided_store_win_alloc_shared_signal.c`
    - Different communication methods

- Make a diff from one version to the next version of the source code
- Compare latency and bandwidth
# Programming models

- MPI + OpenMP

- General considerations  
- How to compile, link, and run  
- Case-study: The Multi-Zone NAS Parallel Benchmarks  
- Memory placement on ccNUMA systems  
- Topology and affinity on multicore  
- Overlapping communication and computation  
- Main advantages, disadvantages, conclusions

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Pure MPI</th>
<th>MPI+MPI-3.0 shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>MPI+OpenMP</td>
<td>MPI+Accelerators</td>
</tr>
<tr>
<td>Programming models</td>
<td>Tools</td>
<td>Conclusions</td>
</tr>
</tbody>
</table>
Hybrid MPI+OpenMP Masteronly Style

Advantages
- No message passing inside of the SMP nodes
- No topology problem

for (iteration ....)
{
    #pragma omp parallel
    numerical code
    /*end omp parallel */

    /* on master thread only */
    MPI_Send (original data to halo areas in other SMP nodes)
    MPI_Recv (halo data from the neighbors)
} /*end for loop

Major Problems
- All other threads are sleeping while master thread communicates!
- Which inter-node bandwidth?
- MPI-lib must support at least MPI_THREAD_FUNNELED
MPI rules with OpenMP / Automatic SMP-parallelization

- Special MPI-2 Init for multi-threaded MPI processes:

```c
int MPI_Init_thread( int * argc, char ** argv[],
                     int thread_level_required,
                     int * thead_level_provided);
int MPI_Query_thread( int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

- REQUIRED values (increasing order):
  - `MPI_THREAD_SINGLE`: Only one thread will execute
  - `THREAD_MASTERONLY`: MPI processes may be multi-threaded, but only master thread will make MPI-calls AND only while other threads are sleeping (virtual value, not part of the standard)
  - `MPI_THREAD_FUNNELED`: Only master thread will make MPI-calls
  - `MPI_THREAD_SERIALIZED`: Multiple threads may make MPI-calls, but only one at a time
  - `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI, with no restrictions

- returned `provided` may be less than REQUIRED by the application
Calling MPI inside of OMP MASTER

- Inside of a parallel region, with ”OMP MASTER”
- Requires MPI_THREAD_FUNNELED, i.e., only master thread will make MPI-calls

**Caution:** There isn’t any synchronization with “OMP MASTER”! Therefore, “OMP BARRIER” normally necessary to guarantee, that data or buffer space from/for other threads is available before/after the MPI call!

```plaintext
!$OMP BARRIER
!$OMP MASTER
   call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER

#pragma omp barrier
#pragma omp barrier
#pragma omp master
MPI_Xxx(...);
```

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!
... the barrier is necessary – example with MPI_Recv

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
  int i;
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int size, rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);

  MPI_Barrier(comm);

  if (rank == 0) {
    MPI_Recv(buf, size, MPI_DOUBLE, 1, 0, comm, &status);
    if (rank == 1) {
      MPI_Send(buf, size, MPI_DOUBLE, 0, 0, comm);
    }
  }

  MPI_Barrier(comm);

  #pragma omp parallel
  {
    #pragma omp for nowait
    for (i=0; i<1000; i++)
      a[i] = buf[i];
    #pragma omp barrier
    #pragma omp master
    MPI_Recv(buf, size, MPI_DOUBLE, 0, 0, comm, &status);
    #pragma omp barrier
    #pragma omp for nowait
    for (i=0; i<1000; i++)
      c[i] = buf[i];
  }

  MPI_Finalize();
  return 0;
}
```

#pragma omp parallel
{
  #pragma omp for nowait
  for (i=0; i<1000; i++)
    a[i] = buf[i];
  #pragma omp barrier
  #pragma omp master
  MPI_Recv(buf,...);
  #pragma omp barrier

  #pragma omp for nowait
  for (i=0; i<1000; i++)
    c[i] = buf[i];
}

/* omp end parallel */
### MPI + OpenMP versus pure MPI (Cray XC30)

#### Measurements:
- Bi-directional halo exchange in a ring with 4 SMP nodes
- With 16 and 512kB per message;
- Bandwidth: each message is counted only once, i.e., not twice at sender and receiver.

#### Internode: Irecv + Send

<table>
<thead>
<tr>
<th>Latency</th>
<th>Accumulated inter-node bandwidth per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 µs</td>
<td>6.8 GB/s</td>
</tr>
<tr>
<td>4.1 µs</td>
<td>7.1 GB/s</td>
</tr>
<tr>
<td>4.1 µs</td>
<td>5.2 GB/s</td>
</tr>
<tr>
<td>4.4 µs</td>
<td>4.7 GB/s</td>
</tr>
<tr>
<td>10.2 µs</td>
<td>4.2 GB/s</td>
</tr>
</tbody>
</table>

#### Pure MPI

<table>
<thead>
<tr>
<th>Latency</th>
<th>Accumulated inter-node bandwidth per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9 µs</td>
<td>4.4 GB/s</td>
</tr>
<tr>
<td>3.0 µs</td>
<td>4.5 GB/s</td>
</tr>
<tr>
<td>3.0 µs</td>
<td>4.6 GB/s</td>
</tr>
<tr>
<td>3.3 µs</td>
<td>4.4 GB/s</td>
</tr>
<tr>
<td>3.5 µs</td>
<td>4.4 GB/s</td>
</tr>
<tr>
<td>5.2 µs</td>
<td>4.3 GB/s</td>
</tr>
<tr>
<td>5.2 µs</td>
<td>4.4 GB/s</td>
</tr>
<tr>
<td>10.3 µs</td>
<td>4.5 GB/s</td>
</tr>
<tr>
<td>10.1 µs</td>
<td>4.5 GB/s</td>
</tr>
</tbody>
</table>

#### Conclusion:
- MPI+OpenMP is faster (but not much)
- Best bandwidth with only 1 or 2 communication links per node
- No win through MPI-3.0 shared memory programming
Load-Balancing
(on same or different level of parallelism)

- OpenMP enables
  - Cheap **dynamic** and **guided** load-balancing
  - Just a parallelization option (clause on omp for / do directive)
  - Without additional software effort
  - Without explicit data movement

- On MPI level
  - **Dynamic load balancing** requires
    moving of parts of the data structure through the network
  - Significant runtime overhead
  - Complicated software / therefore not implemented

- **MPI & OpenMP**
  - Simple static load-balancing on MPI level,
    dynamic or guided on OpenMP level

```c
#pragma omp parallel for schedule(dynamic)
for (i=0; i<n; i++) {
    /* poorly balanced iterations */ …
}
```
Sleeping threads with

**Problem:**
- Sleeping threads are wasting CPU time

**Solution:**
- Overlapping of computation and communication

**Limited benefit:**
- In the best case, communication overhead can be reduced from 50% to 0% → speedup of 2.0
- Usual case of 20% to 0% → speedup is 1.25
- Achievable with significant work → later

```c
for (iteration ....) {
    #pragma omp parallel
    numerical code
    /*end omp parallel */

    /* on master thread only */
    MPI_Send (original data to halo areas in other SMP nodes)
    MPI_Recv (halo data from the neighbors)
} /*end for loop
```
Programming models - MPI + OpenMP

How to compile, link, and run
How to compile, link and run

- Use appropriate **OpenMP compiler switch** (-openmp, -fopenmp, -mp, -qsmp=openmp, …) and MPI compiler script (if available)
- Link with **MPI library**
  - Usually wrapped in MPI compiler script
  - If required, specify to link against thread-safe MPI library
    - Often automatic when OpenMP or auto-parallelization is switched on
- Running the code
  - Highly non-portable! Consult system docs! (if available…)
  - If you are on your own, consider the following points
  - Make sure **OMP_NUM_THREADS** etc. is available on all MPI processes
    - Start “env VAR=VALUE … <YOUR BINARY>” instead of your binary alone
    - Use Pete Wyckoff’s **mpiexec** MPI launcher (see below): [http://www.osc.edu/~djohnson/mpiexec/](http://www.osc.edu/~djohnson/mpiexec/)
  - Figure out how to start fewer MPI processes than cores on your nodes
Examples for compilation and execution

- **Cray XC40** (2 NUMA domains w/ 12 cores each):
  - `ftn -h omp ...
  - `export OMP_NUM_THREADS=12`
  - `aprun -n nprocs -N nprocs_per_node \ -d $OMP_NUM_THREADS a.out`

- **Intel Sandy Bridge** (8-core 2-socket) cluster, **Intel MPI/OpenMP**
  - `mpiifort -openmp ...
  - `OMP_NUM_THREADS=8 mpirun -ppn 2 -np 4 \ -env I_MPI_PIN_DOMAIN socket \ -env KMP_AFFINITY scatter ./a.out`
Interlude: Advantages of mpiexec or similar mechanisms

• Startup mechanism should use a resource manager interface to spawn MPI processes on nodes
  – As opposed to starting remote processes with ssh/rsh:
    • Correct CPU time accounting in batch system
    • Faster startup
    • Safe process termination
    • Allowing password-less user login not required between nodes
  – Interfaces directly with batch system to determine number of procs

• Provisions for starting fewer processes per node than available cores
  – Required for hybrid programming
  – E.g., “-pernode” and “-npерnode #” options – does not require messing around with nodefiles
Thread support within OpenMPI

- In order to enable thread support in Open MPI, configure with:
  
  ```
  configure --enable-mpi-threads
  ```

- This turns on:
  - Support for full `MPI_THREAD_MULTIPLE`
  - Internal checks when run with threads (`--enable-debug`) 

  ```
  configure --enable-mpi-threads --enable-progress-threads
  ```

- This (additionally) turns on:
  - Progress threads to asynchronously transfer/receive data per network BTL.

- Additional Feature:
  - Compiling **with** debugging support, but **without** threads will check for recursive locking
Programming models
- MPI + OpenMP

Case-study:
The Multi-Zone NAS Parallel Benchmarks
The Multi-Zone NAS Parallel Benchmarks

- Multi-zone versions of the NAS Parallel Benchmarks LU, SP, and BT
- Two hybrid sample implementations
- Load balance heuristics part of sample codes

www.nas.nasa.gov/Resources/Software/software.html
call omp_set_numthreads (weight)
do step = 1, itmax
    call exch_qbc(u, qbc, nx,...)
call mpi_send/recv

do zone = 1, num_zones
    if (iam .eq. pzone_id(zone)) then
        call zsolve(u,rsd,...)
    end if
end do

end do
...
Benchmark Characteristics

- Aggregate sizes:
  - Class D: 1632 x 1216 x 34 grid points
  - Class E: 4224 x 3456 x 92 grid points

- **BT-MZ:** *(Block tridiagonal simulated CFD application)*
  - Alternative Directions Implicit (ADI) method
  - #Zones: 1024 (D), 4096 (E)
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance

- **SP-MZ:** *(Scalar Pentadiagonal simulated CFD application)*
  - #Zones: 1024 (D), 4096 (E)
  - Size of zones identical
    - no load-balancing required

Expectations:
- Pure MPI: Load-balancing problems!
- Good candidate for MPI+OpenMP
- Load-balanced on MPI level: Pure MPI should perform best
Hybrid code on modern architectures

- **OpenMP:**
  - Support only per MPI process
  - Version 3.1 has support for binding of threads via OMP_PROC_BIND environment variable.
  - Version 4.0:
    - The proc_bind clause (see Section 2.4.2 in Spec OpenMP 4.0)
    - OMP_PLACES environment variable (see Section 4.5) were added to support thread affinity policies

- **MPI:**
  - Initially not designed for multicore/ccNUMA architectures or mixing of threads and processes, MPI-2 supports threads in MPI
  - API does not provide support for memory/thread placement

- **Vendor specific APIs to control thread and memory placement:**
  - Environment variables
  - System commands like numactl, taskset, dplace, omplace, likwid-pin etc
  → See later for more!
CPU type: Intel Core Westmere processor

Hardware Thread Topology

Sockets: 2
Cores per socket: 6
Threads per core: 1

Socket 0: ( 1 3 5 7 9 11 )
Socket 1: ( 0 2 4 6 8 10 )

Careful!
Numbering scheme of cores is system dependent.
Pitfall: Remote memory access

Running NPB BT-MZ Class D 128 MPI Procs, 6 threads each 2 MPI per node

Pinning A:
if [ $localrank == 0 ]; then
exec numactl --physcpubind=0,1,2,3,4,5 -m 0 $*
elif [ $localrank == 1 ]; then
exec numactl --physcpubind=6,7,8,9,10,11 -m 1 $*
fi

Pinning B:
if [ $localrank == 0 ]; then
exec numactl --physcpubind=0,2,4,6,8,10 -m 0 $*
elif [ $localrank == 1 ]; then
exec numactl --physcpubind=1,3,5,7,9,11 -m 1 $*
fi

Half of the threads access remote memory (other socket)

600 Gflops

Only local memory access

900 Gflops
NPB-MZ Class E Scalability on Lonestar

- BT-MZ
- SP-MZ
- BT-MZ fixed

With thread pinning:
Needed if OpenMP spans multi-sockets.

→ indeed, BT-MZ profits from hybrid

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
Using more OpenMP threads reduces the memory usage substantially, up to five times on Hopper Cray XT5 (eight-core nodes).

Hongzhang Shan, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, Nicholas J. Wright: *Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms.*
Programming models
- MPI + OpenMP

Memory placement on ccNUMA systems
Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:
  A memory page gets mapped into the local memory of the processor that first touches it!
  - Except if there is not enough local memory available
  - Some OSs allow to influence placement in more direct ways
    - → libnuma (Linux)
- Caveat: "touch" means "write", not "allocate"
- Example:

  ```c
  double *huge = (double*)malloc(N*sizeof(double));
  // memory not mapped yet
  for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0;  // mapping takes place here!
  ```

- It is sufficient to touch a single item to map the entire page
- With pure MPI (or process per ccNUMA domain): fully automatic!
Most simple case: explicit initialization

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

A = 0.d0

!$OMP parallel do
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end parallel do

integer, parameter :: N = 10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   A(i) = 0.d0
end do
!$OMP end do
!$OMP end parallel
```

Motivation

Introduction

Programming models

Tools

Conclusions

Pure MPI

MPI+MPI-3.0 shared memory

MPI+OpenMP

MPI+Accelerators
ccNUMA problems beyond first touch

- OS uses part of main memory for disk buffer (FS) cache
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - $\rightarrow$ non-local access
  - Locality problem even on hybrid and pure MPI

- Remedies
  - Drop FS cache pages after user job has run (admin’s job)
    - Only prevents cross-job buffer cache “heritage”
  - “Sweeper” code (run by user)
  - Flush buffer cache after I/O if necessary (“sync” is not sufficient!)
ccNUMA problems beyond first touch: Buffer cache

Real-world example: ccNUMA and the Linux buffer cache

Benchmark:
1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory installed in LD0

Result: By default, Buffer cache is given priority over local page placement → restrict to local domain if possible!
How to handle ccNUMA in practice

- Problems appear when one process spans multiple ccNUMA domains:
  - **First touch** needed to “bind” the data to each socket → **otherwise loss of performance**
  - **Thread binding is mandatory!** The OS kernel does not know what you need!
  - Dynamic/guided schedule or tasking → **loss of performance**

- Practical solution:
  - One MPI process per ccNUMA domain
    → small number (>1) of MPI processes on each node
    → more complex affinity enforcement (binding)
    → more choices for rank mapping (4 sockets example):

- Provided that the application has a 20% surface
Programming models
- MPI + OpenMP

Topology and affinity on multicore
The OpenMP-parallel vector triad benchmark

Visualizing OpenMP overhead

- OpenMP work sharing in the benchmark loop
  
  ```
  double precision, dimension(:,), allocatable :: A,B,C,D
  allocate(A(1:N),B(1:N),C(1:N),D(1:N))
  !$OMP PARALLEL private(i,j)
  !$OMP DO
  do i=1,N
    A(i)=1.d0; B(i)=1.d0; C(i)=1.d0; D(i)=1.d0
  enddo
  !$OMP END DO
  do j=1,NITER
    !$OMP DO
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
    !$OMP END DO
    if(.something.that.is.never.true.) then
      call dummy(A,B,C,D)
    endif
  enddo
  !$OMP END PARALLEL
  ```

  Initialization with same work sharing scheme as used within the numerical loop

  Numerical loop

  Real work sharing

  Implicit barrier

  ... and then report performance vs. loop size for different #cores!
OpenMP vector triad on Sandy Bridge socket (3 GHz)

- Cost of sync amortized for large problems
- Sync overhead grows with # of threads
- Next slide for direct measurements!
Thread synchronization overhead on SandyBridge-EP

Direct measurement of barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>384</td>
<td>5242</td>
<td>4616</td>
</tr>
<tr>
<td>SMT threads</td>
<td>2509</td>
<td>3726</td>
<td>3399</td>
</tr>
<tr>
<td>Other socket</td>
<td>1375</td>
<td>5959</td>
<td>4909</td>
</tr>
</tbody>
</table>

See also [http://blogs.fau.de/hager/archives/6883](http://blogs.fau.de/hager/archives/6883)
Thread synchronization overhead on Intel Xeon Phi

Barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>One core</td>
<td>n/a</td>
<td>1597</td>
<td>2825</td>
<td>3557</td>
</tr>
<tr>
<td>Full chip</td>
<td>10604</td>
<td>12800</td>
<td>15573</td>
<td>18490</td>
</tr>
</tbody>
</table>

That does not look too bad for 240 threads!

Still the “pain” may be much larger, because more work can be done in one cycle on Phi compared to a full (16-core) Sandy Bridge node:

- 3.75 x cores (16 vs 60) on Phi
- 2 x more operations per cycle on Phi

→ 7.5 x more work done on Xeon Phi per cycle

- 2.7 x higher barrier penalty (cycles) on Phi

→ One barrier causes 2.7 x 7.5 ≈ 20x more pain 😊.
Thread/Process Affinity ("Pinning")

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`, PLPA → hwloc
    - Solaris: `processor_bind()`
    - Windows: `SetThreadAffinityMask()`
    - ...

- Support for "semi-automatic" pinning in all modern compilers
  - Intel, GCC, PGI,…
  - OpenMP 4.0
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)

- Affinity awareness in MPI libraries
  - Cray MPI
  - OpenMPI
  - Intel MPI
  - …
Anarchy vs. affinity with OpenMP STREAM

- Reasons for caring about affinity:
  - Eliminating performance variation
  - Making use of architectural features
  - Avoiding resource contention

No pinning

Pinning (physical cores first, first socket first)
likwid-pin

- Binds process and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify “skip mask” (i.e., supports many different compiler/MPI combinations)
- Replacement for taskset
- Uses logical (contiguous) core numbering when running inside a restricted set of cores
- Supports logical core numbering inside node, socket, core

- Usage examples:
  - env OMP_NUM_THREADS=6 likwid-pin -c 0,2,4-6 ./myApp parameters
  - env OMP_NUM_THREADS=6 likwid-pin -c S0:0-2@S1:0-2 ./myApp
Likwid-pin

Example: Intel OpenMP

• Running the STREAM benchmark with likwid-pin:

```bash
$ export OMP_NUM_THREADS=4
$ likwid-pin -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK

Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word

[... some STREAM output omitted ...]
The *best* time for each test is used
*EXCLUDING* the first and last iterations
[pthread wrapper] PIN_MASK: 0->1 1->4 2->5
[pthread wrapper] SKIP MASK: 0x1
[pthread wrapper 0] Notice: Using libpthread.so.0
threadid 1073809728 -> SKIP
[pthread wrapper 1] Notice: Using libpthread.so.0
threadid 1078008128 -> core 1 - OK
[pthread wrapper 2] Notice: Using libpthread.so.0
threadid 1082206528 -> core 4 - OK
[pthread wrapper 3] Notice: Using libpthread.so.0
threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]
```

Main PID always pinned
Skip shepherd thread
Pin all spawned threads in turn
OMP_PLACES and Thread Affinity  
(see OpenMP-4.0 page 7 lines 29-32, p. 241-243)

A place consists of one or more processors.
Pinning on the level of places.
Free migration of the threads on a place between the processors of that place.

- `setenv OMP_PLACES threads abstract_name`
  
  → Each place corresponds to the single processor of a single hardware thread (hyper-thread)

- `setenv OMP_PLACES cores`
  
  → Each place corresponds to the processors (one or more hardware threads) of a single core

- `setenv OMP_PLACES sockets`
  
  → Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)

- `setenv OMP_PLACES abstract_name(num_places)`
  
  → In general, the number of places may be explicitly defined

- Or with explicit numbering, e.g. 8 places, each consisting of 4 processors:
  - `setenv OMP_PLACES "\{0,1,2,3\},\{4,5,6,7\},\{8,9,10,11\}, \ldots \{28,29,30,31\}"`
  - `setenv OMP_PLACES "\{0:4\},\{4:4\},\{8:4\}, \ldots \{28:4\}"`
  - `setenv OMP_PLACES "\{0:4\}:8:4"`

CAUTION: The numbers highly depend on hardware and operating system, e.g.,
\(\{0,1\}\) = hyper-threads of 1st core of 1st socket, or
\(\{0,1\}\) = 1st hyper-thread of 1st core of 1st and 2nd socket, or …
OpenMP places and proc_bind  (see OpenMP-4.0 pages 49f, 239, 241-243)

```bash
csetenv OMP_PLACES "{0},{1},{2}, ... {29},{30},{31}" or
csetenv OMP_PLACES threads (example with P=32 places)
```

- `sentenv OMP_NUM_THREADS "8,2,2"
  sentenv OMP_PROC_BIND "spread,spread,close"

- Master thread encounters nested parallel regions:
  ```
  #pragma omp parallel → uses: num_threads(8) proc_bind(spread)
  #pragma omp parallel → uses: num_threads(2) proc_bind(spread)
  #pragma omp parallel → uses: num_threads(2) proc_bind(close)
  ```

- `spread:` Sparse distribution of the 8 threads among the 32 places; partitioned place lists.
- `close:` New threads as close as possible to the parent’s place; same place lists.
- `master:` All new threads at the same place as the parent.

Outside of first parallel region: master thread has a **place list** with all 32 places.

After first `#pragma omp parallel`:
8 threads in a team, each on a **partitioned place list** with 32/8=4 places.

Only one place is used.
Goals behind OMP_PLACES and proc_bind

Example: 4 sockets \( \times \) 6 cores \( \times \) 2 hyper-threads = 48 processors

Vendor’s numbering: round robin over the sockets, over cores, and hyperthreads

<table>
<thead>
<tr>
<th>Sockets</th>
<th>Cores</th>
<th>Hyper-threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 24 25 26 27</td>
<td>0 1 2 3 4 5 6 7</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>8 9 10 11 12 13 14 15</td>
<td>8 9 10 11 12 13 14 15</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>16 17 18 19 20 21 22 23</td>
<td>16 17 18 19 20 21 22 23</td>
<td>16 17 18 19 20 21 22 23</td>
</tr>
</tbody>
</table>

- `setenv OMP_PLACES threads` (= \( \{0,24,4,28,8,32,12,36,16,40,20,44,1,25,\ldots\} \)) → OpenMP threads/tasks are pinned to hardware hyper-threads

- `setenv OMP_PLACES cores` (= \( \{0,24,4,28,8,32,12,36,16,40,20,44,1,25,\ldots\} \)) → OpenMP threads/tasks are pinned to hardware cores and can migrate between hyper-threads of the core

- `setenv OMP_PLACES sockets` (= \( \{0,24,4,28,8,32,12,36,16,40,20,44,1,25,\ldots\} \)) → OpenMP threads/tasks are pinned to hardware sockets and can migrate between cores & hyper-threads of the socket

Examples should be independent of vendor’s numbering!

- Without nested parallel regions:
  
  `#pragma omp parallel num_threads(4*6) proc_bind(spread)` → one thread per core

- With nested regions:
  
  `#pragma omp parallel num_threads(4) proc_bind(spread)` → one thread per socket
  
  `#pragma omp parallel num_threads(6) proc_bind(spread)` → one thread per core
  
  `#pragma omp parallel num_threads(2) proc_bind(close)` → one thread per hyper-thread
Topology ("mapping") with MPI+OpenMP:
Lots of choices – solutions are highly system specific!

One MPI process per node

One MPI process per socket

OpenMP threads pinned "round robin" across cores in node

Two MPI processes per socket

Following two slides: Examples with likwid-mpirun & Intel environment
likwid-mpirun
1 MPI process per node

likwid-mpirun -np 2 -pin N:0-11 ./a.out

Intel MPI+compiler:
OMP_NUM_THREADS=12 mpirun -ppn 1 -np 2 -env KMP_AFFINITY scatter ./a.out

Motivation
Introduction
Programming models
Tools
Conclusions

LIKWID: Topology/Affinity

Hybrid Parallel Programming
(c) RRZE 2014
Rabenseifner, Hager, Jost
likwid-mpirun
1 MPI process per socket

likwid-mpirun -np 4 -pin S0:0-5_S1:0-5 ./a.out

Intel MPI+compiler:
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \n-env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out

LIKWID: Topology/Affinity
MPI/OpenMP ccNUMA and topology: Take-home messages

- **Learn how to take control** of hybrid execution!
  - Almost all performance features depend on topology and thread placement! (especially if SMT/Hyperthreading is on)
- Always observe the **topology dependence** of
  - Intranode MPI
  - OpenMP overheads
  - Saturation effects / scalability behavior with bandwidth-bound code
- Enforce proper thread/process to core **binding**, using appropriate tools (whatever you use, but use SOMETHING)
- Multi-domain OpenMP processes on ccNUMA nodes require correct page placement: Observe first touch policy!
Programming models
- MPI + OpenMP

Overlapping Communication and Computation
Parallel Programming Models on Hybrid Platforms

- **pure MPI**
  - one MPI process on each core
  - **No overlap of Comm. + Comp.**
  - MPI only outside of parallel regions of the numerical application code
  - **Masteronly**
  - MPI only outside of parallel regions

- **hybrid MPI+OpenMP**
  - MPI: inter-node communication
  - OpenMP: inside of each SMP node
  - **Overlapping Comm. + Comp.**
  - MPI communication by one or a few threads while other threads are computing
  - **Funneled**
  - MPI only on master-thread
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **Hybrid MPI+MPI**
  - MPI for inter-node communication + MPI-3.0 shared memory programming
  - **Within shared memory nodes: Halo updates through direct data copy**
  - **Multiple**
  - more than one thread may communicate
  - **Funneled & Reserved**
  - reserved thread for communication

- **OpenMP only**
  - distributed virtual shared memory
  - **Within shared memory nodes: No halo updates, direct access to neighbor data**

- **Pure MPI**
  - MPI-3.0 shared memory programming
  - **Halo updates through direct data copy**
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **MPI+OpenMP**
  - inside of each SMP node
  - **Funneled**
  - MPI only on master-thread
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **MPI+Accelerators**
  - MPI for inter-node communication + MPI-3.0 shared memory programming
  - **Within shared memory nodes: Halo updates through direct data copy**
  - **Multiple**
  - more than one thread may communicate
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **Pure MPI**
  - MPI-3.0 shared memory programming
  - **Halo updates through direct data copy**
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **MPI+OpenMP**
  - inside of each SMP node
  - **Funneled**
  - MPI only on master-thread
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **MPI+Accelerators**
  - MPI for inter-node communication + MPI-3.0 shared memory programming
  - **Within shared memory nodes: Halo updates through direct data copy**
  - **Multiple**
  - more than one thread may communicate
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **Pure MPI**
  - MPI-3.0 shared memory programming
  - **Halo updates through direct data copy**
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **MPI+OpenMP**
  - inside of each SMP node
  - **Funneled**
  - MPI only on master-thread
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**

- **MPI+Accelerators**
  - MPI for inter-node communication + MPI-3.0 shared memory programming
  - **Within shared memory nodes: Halo updates through direct data copy**
  - **Multiple**
  - more than one thread may communicate
  - **Funneled & Reserved**
  - reserved thread for communication
  - **Funneled with Full Load Balancing**
Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

```java
if (my_thread_rank < ...) {
    MPI_Send/Recv....
    i.e., communicate all halo data
} else {
    Execute those parts of the application
    that do not need halo data
    (on non-communicating threads)
}

Execute those parts of the application
that need halo data
(on all threads)
```
Hybrid Parallel Programming

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

Three problems:
• the application problem:
  – one must separate application into:
    • code that can run before the halo data is received
    • code that needs halo data

  ➞ very hard to do !!!

• the thread-rank problem:
  – comm. / comp. via thread-rank
  – cannot use work-sharing directives

  ➞ loss of major OpenMP support
  (see next slide)

• the load balancing problem

if (my_thread_rank < 1) {
  MPI_Send/Recv....
} else {
  my_range = (high-low-1) / (num_threads-1) + 1;
  my_low = low + (my_thread_rank+1)*my_range;
  my_high=low + (my_thread_rank+1+1)*my_range;
  my_high = max(high, my_high)
  for (i=my_low; i<my_high; i++) {
    ....
  }
}
**Overlapping Communication and Computation**

MPI communication by one or a few threads while other threads are computing

**Subteams**

- **Proposal**
  - for OpenMP 3.x
  - or OpenMP 4.x
  - or OpenMP 5.x

Barbara Chapman et al.:
Toward Enhancing OpenMP’s Work-Sharing Directives.

```c
#pragma omp parallel
{
    #pragma omp single onthreads( 0 )
    {
        MPI_Send/Recv ....
    }
    #pragma omp for onthreads( 1 : omp_get_numthreads()-1 )
    for (………)
    { /* work without halo information */
        /* barrier at the end is only inside of the subteam */
        ...
    }
    #pragma omp barrier
    #pragma omp for
    for (………)
    { /* work based on halo information */
    }
} /*end omp parallel */
```

**Workarounds today:**

- nested parallelism: one thread MPI + one for computation → nested (n-1) threads
- Loop with guided/dynamic schedule and first iteration invokes communication
Example: sparse matrix-vector multiply (spMVM)

- spMVM on Intel Westmere cluster (6 cores/socket)
- “task mode” == explicit communication overlap using ded. thread
- “vector mode” == MASTERONLY
- “naïve overlap” == non-blocking MPI
- Memory bandwidth is already saturated by 5 cores

Overlapping: Using OpenMP tasks

NEW OpenMP Tasking Model gives a new way to achieve more parallelism form hybrid computation.

Alice Koniges et al.:
Application Acceleration on Current and Future Cray Platforms.

Slides, courtesy of Alice Koniges, NERSC, LBNL
Tasking example: dense matrix-vector multiply with communication overlap

- Data distribution across processes:

\[
\mathbf{c} = \mathbf{c} + \mathbf{A} \times \mathbf{r}
\]
Dense matrix-vector multiply with communication overlap via tasking

- Computation/communication scheme:

  Step 1: MVM on diagonal blocks
  = + X

  Ring shift of vector $r$

  Step 2: MVM on next subdiag blocks
  = + X
Dense matrix-vector multiply with communication overlap via tasking

```c
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int n_start=rank*my_size+min(rest,rank), cur_size=my_size;
    // loop over RHS ring shifts
    for(int rot=0; rot<ranks; rot++) {
        #pragma omp single
        {
            if(rot!=ranks-1) {
                #pragma omp task
                {
                    MPI_Isend(buf[0], ..., r_neighbor, ..., &request[0]);
                    MPI_Irecv(buf[1], ..., l_neighbor, ..., &request[1]);
                    MPI_Waitall(2, request, status);
                }
            }
        }
    }
    #pragma omp single
    for(int row=0; row<my_size; row+=4) {
        #pragma omp task
        do_local_mvm_block(a, y, buf, row, n_start, cur_size, n);
    }
    #pragma omp single
    tmpbuf = buf[1]; buf[1] = buf[0]; buf[0] = tmpbuf;
    n_start += cur_size;
    if(n_start>=size) n_start=0; // wrap around
    cur_size = size_of_rank(l_neighbor,ranks,size);
}
```

Asynchronous communication (ring shift)

Current block of MVM (chunked by 4 rows)
Case study: Communication and Computation in Gyrokinetic Tokamak Simulation (GTS) shift routine

Work on particle array (packing for sending, reordering, adding after sending) can be overlapped with data independent MPI communication using OpenMP tasks.

GTS shift routine

Slides, courtesy of Alice Koniges, NERSC, LBNL
Overlapping can be achieved with OpenMP tasks (1st part)

```fortran
integer stride=1000
!$omp parallel
!$omp master
!pack particle to move right
  do m=1,x-stride,stride
    !$omp task
    do mm=0,stride-1,1
      sendright (m+mm)= p_array (f(m+mm));
    enddo
    !$omp end task
  enddo
!$omp end task

!$omp task
  do m=x,stride
    sendright (m)= p_array (f(m));
  enddo
!$omp end task
```

Overlapping MPI_Allreduce with particle work

- **Overlap**: Master thread encounters (!$omp master) tasking statements and creates work for the thread team for deferred execution. MPI Allreduce call is immediately executed.
- MPI implementation has to support at least MPI_THREAD_FUNNELED
- Subdividing tasks into smaller chunks to allow better load balancing and scalability among threads.

Slides, courtesy of Alice Koniges, NERSC, LBNL
Overlapping can be achieved with OpenMP tasks (2\textsuperscript{nd} part)

```c
!$omp parallel
!$omp master
  !$omp task
    fill_hole(p_array);
  !$omp end task
MPI_SENDRECV(x, length=2, ...);
MPI_SENDRECV(sendright, length=g(x), ...);
MPI_SENDRECV(y, length=2, ...);
!$omp end master
!$omp end parallel
```

Overlapping particle reordering

Particle reordering of remaining particles (above) and adding sent particles into array (right) & sending or receiving of shifted particles can be independently executed.

```c
!$omp parallel
!$omp master
  !$omp task
    do m=1, x - stride, stride
      !$omp task
        do mm=0, stride - 1, 1
          p_array(h(m)) = sendright(m);
        !$omp end task
      !$omp end task
    enddo
!$omp end task
!$omp task
  do m=m, x
    p_array(h(m)) = sendright(m);
  !$omp end task
!$omp end parallel
```

```c
MPI_SENDRECV(sendleft, length=g(y), ...);
```

Overlapping remaining MPI_Sendrecv

```
!$omp parallel
!$omp task
  do n=1, y
    p_array(h(n)) = sendleft(n);
  !$omp end task
enddo
```

Slides, courtesy of Alice Koniges, NERSC, LBNL
OpenMP tasking version outperforms original shifter, especially in larger poloidal domains

- Performance breakdown of GTS shifter routine using 4 OpenMP threads per MPI process with varying domain decomposition and particles per cell on Franklin Cray XT4.
- MPI communication in the shift phase uses a **toroidal MPI communicator** (constantly 128).
- Large performance differences in the 256 MPI run compared to 2048 MPI run!
- Speed-Up is expected to be higher on larger GTS runs with hundreds of thousands CPUs since MPI communication is more expensive.
MPI+OpenMP: Main advantages

Masteronly style (i.e., MPI outside of parallel regions)

• **Increase parallelism**
  – Scaling to higher number of cores
  – Adding OpenMP with incremental additional parallelization

• **Lower memory requirements** due to smaller number of MPI processes
  – Reduced amount of application halos & replicated data
  – Reduced size of MPI internal buffer space
  – Very important on systems with many cores per node

• **Lower communication overhead** (possibly)
  – Few multithreaded MPI processes vs many single-threaded processes
  – Fewer number of calls and smaller amount of data communicated
  – Topology problems from pure MPI are solved
    (was application topology versus multilevel hardware topology)

• Provide for **flexible load-balancing** on coarse and fine levels
  – Smaller #of MPI processes leave room for assigning workload more evenly
  – MPI processes with higher workload could employ more threads

Additional advantages when overlapping communication and computation:
  – No sleeping threads
MPI+OpenMP: Main disadvantages & challenges

Masteronly style (i.e., MPI outside of parallel regions)

- **Non-Uniform Memory Access:**
  - Not all memory access is equal: ccNUMA locality effects
  - Penalties for access across NUMA domain boundaries
  - First touch is needed for *more than one ccNUMA node per MPI process*
  - Alternative solution: *One MPI process on each ccNUMA domain (i.e., chip)*

- **Multicore / multisocket anisotropy effects**
  - Bandwidth bottlenecks, shared caches
  - Intra-node MPI performance
    - Core ↔ core vs. socket ↔ socket
    - OpenMP loop overhead

- **Amdahl’s law** on both, MPI and OpenMP level
- Thread and process *pinning*
- **Other disadvantages through OpenMP**

Additional disadvantages when overlapping communication and computation:

- High programming overhead
- OpenMP is not prepared for this programming style
MPI+OpenMP: Conclusions

Work-horse on large systems:

- **Increase parallelism** with MPI+OpenMP
- **Lower memory requirements** due to smaller number of MPI processes
- **Lower communication overhead**
- **More flexible load balancing**
- Challenges due to ccNUMA
  - May be solved by using multi-threading only within ccNUMA domains
  - Pinning
- Overlapping communication & computation
  - Benefit calculation: compute time versus programming time
Exercise: MPI+OpenMP-Hybrid Jacobi solver

- Use the given program as your baseline for the following exercise:
  
  ```
  cp -a ~/MPIOMP/course/C-hyb-Jacobi/* .  or  
  cp -a ~/MPIOMP/course/F-hyb-Jacobi/* .  
  ```

- This is a Jacobi solver (2D stencil code) with domain decomposition and halo exchange
- The given code is MPI-only. You can build it with make (take a look at the `Makefile`) and run it with something like this (adapt to local requirements):

  ```
  $ <mpirun-or-whatever> -np <numprocs> ./jacobi.exe < input
  ```

  Your task is to parallelize it with OpenMP to get a hybrid MPI+OpenMP code, and run it effectively on the given hardware.

- Notes:
  - The code is strongly memory bound at the problem size set in the input file
  - Learn how to take control of affinity with MPI and especially with MPI+OpenMP
  - Always run multiple times and observe performance variations
  - If you know how, try to calculate the maximum possible performance and use it as a “light speed” baseline
Exercise cont’d

- Tasks (we assume $N_c$ cores per CPU socket):
  - Run the MPI-only code on one node with $1,\ldots,N_c,\ldots,2*N_c$ processes (1 full node) and observe the achieved performance behavior
  - Parallelize appropriate loops with OpenMP
  - Run with OpenMP and 1 MPI process (“OpenMP-only”) on $1,\ldots,N_c,\ldots,2*N_c$ cores, compare with MPI-only run
  - Run hybrid variants with different MPI vs. OpenMP ratios

- Things to observe
  - Run-to-run performance variations
  - Does the OpenMP/hybrid code perform as well as the MPI code? If it doesn’t, fix it!
Programming models - MPI + Accelerator

Courtesy of Gabriele Jost
OpenMP 4.0 Support for Co-Processors

- **New concepts:**
  - **Device:** An implementation defined logical execution engine; local storage which could be shared with other devices; device could have one or more processors

- **Extension to the previous Memory Model:**
  - **Previous:** Relaxed-Consistency Shared-Memory
  - **Added in 4.0:**
    - **Device** with local storage
    - Data movement can be explicitly indicated by compiler directives
    - **League:** Set of thread teams created by a “teams” construct
    - **Contention group:** threads within a team; OpenMP synchronization restricted to contention groups.

- **Extension to the previous Execution Model**
  - **Previous:** Fork-join of OpenMP threads
  - **Added in 4.0:**
    - Host device offloads a region for execution on a target device
    - Host device waits for completion of execution on the target device
### OpenMP Accelerator Additions

- **Target data**
  - Place objects on the device

- **Target**
  - Move execution to a device

- **Target update**
  - Update objects on the device or host

- **Declare target**
  - Place objects on the device, e.g., common blocks
  - Place subroutines/functions on the device

- **Teams**
  - Start multiple *contention groups*

- **Distribute**
  - Similar to the OpenACC loop construct, binds to teams construct

### OpenMP 4.0 Specification:

#### Motivation

- **Introduction**
- **Programming models**
- **Tools**
- **Conclusions**

#### Pure MPI
- MPI+MPI-3.0 shared memory
- MPI+OpenMP
- MPI+Accelerators
OpenMP 4.0 Simple Example

```c
void smooth( float* restrict a, float* restrict b,
            float w0, float w1, float w2, int n, int m, int niters )
{
    int i, j, iter;
    float* tmp;

    #pragma omp target mapto(b[0:n*m]) map(a[0:n*m])
    #pragma omp teams num_teams(8) num_maxthreads(5)
    for( iter = 1; iter < niters; ++iter ){
        #pragma omp distribute dist_schedule(static) // chunk across teams
            for( i = 1; i < n-1; ++i )
                #pragma omp parallel for // chunk across threads
                    for( j = 1; j < m-1; ++j )
                        a[i*m+j] = w0 * b[i*m+j] +
                                   w1*(b[(i-1)*m+j] + b[(i+1)*m+j] + b[i*m+j-1] +
                                        b[i*m+j+1]) +
                                   w2*(b[(i-1)*m+j-1] + b[(i-1)*m+j+1] +b[(i+1)*m+j-1] +
                                        b[(i+1)*m+j+1]);
        tmp = a;  a = b;  b = tmp;
    }

In main:
    #pragma omp target data map(b[0:n*m],a[0:n*m])
    {
        smooth( a, b, w0, w1, w2, n, m, iters );
    }
```
OpenMP 4.0 **Team** and **Distribute** Construct

```c
#pragma omp target device(acc)
#pragma omp team num_teams(8) num_maxthreads(5)
{
    Stmt1;
    #pragma omp distribute // chunk across thread blocks
    for (i=0; i<N; i++)
        #pragma omp parallel for // chunk across threads
        for (j=0; j<M; j++)
    {
        Threads cannot synchronize
        Threads can synchronize
    }
}
```

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
subroutine z_solve
....
include 'header.h' !omp declare target (/fields/)

!omp declare target (lhsinit)
...
!omp target update to (rhs)
....
!omp target
!omp parallel do default(shared) private(i,j,k,k1,k2,m,...)
do  j = 1, ny2
   call lhsinit(lhs, ....)
do  i = 1, nx
...
do  k = 0, nz2 + 1
   rtmp(1,k) = rhs(1,i,j,k)
....
do  k = 0, nz2 + 1rhs(1,i,j,k) = rtmp(1,k)+ ....
....
!omp end target
!omp target update from (rhs)
What is OpenACC?

- API that supports off-loading of loops and regions of code (e.g. loops) from a host CPU to an attached accelerator in C, C++, and Fortran
- Managed by a nonprofit corporation formed by a group of companies:
  - CAPS Enterprise, Cray Inc., PGI and NVIDIA
- Set of compiler directives, runtime routines and environment variables
- Simple programming model for using accelerators (focus on GPGPUs)
- Memory model:
  - Host CPU + Device may have completely separate memory; Data movement between host and device performed by host via runtime calls; Memory on device may not support memory coherence between execution units or need to be supported by explicit barrier
- Execution model:
  - Compute intensive code regions offloaded to the device, executed as kernels; Host orchestrates data movement, initiates computation, waits for completion; Support for multiple levels of parallelism, including SIMD (gangs, workers, vector)
  - Example constructs: `acc parallel loop, acc data`
**A very simple OpenACC example (PGI 14.10): Schönauer Vector Triad**

```c
int main ()
{
    double a[N], b[N], c[N], d[N];
    ...
    #pragma acc data \
    copyin(b[0:N],c[0:N],d[0:N])
    #pragma acc data copyout (a[0:N])
    compute(a ,b , c ,d ,N);
    ...
}

void compute (double *restrict a , double *b,...) {
    #pragma acc kernels
    #pragma acc loop vector (1024)
    for(int i=0; i<N ; ++i) {
        a[i] = b[i] + c [i] * d[i];
    }
}
```

pgcc -ta=nvidia , cc35 -Minfo -fast -c triad.c
compute:
9 , Generating present or copyout (a [ :N])
Generating present or copyin (b [ :N])
Generating present or copyin (c [ :N])
Generating present or copyin (d [ :N])
Generating Tesla code
10 , Loop is parallelizable
Accelerator kernel generated
10 , #pragma acc loop gang , vector (1024)...

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
Example: 2D Jacobi smoother

```c
#pragma acc data copy(phi1[0:sizex*sizey],phi2[0:sizex*sizey])
{
  for(n=0; n<iter; n++) {
    #pragma acc kernels
    #pragma acc loop independent
    for(kk=1; kk<sizey-1; kk+=block) {
      #pragma acc loop independent private(ofs)
      for(i=1; i<sizex-1; ++i) {
        ofs = i*sizey;
        #pragma acc loop independent
        for(k=0; k<block; ++k) {
          if(kk+k<sizey-1)
            phi1[ofs+kk+k] = oos * (phi2[ofs+kk+k-1] + phi2[ofs+kk+k+1] + phi2[ofs+kk+k-sizey] + phi2[ofs+kk+k+sizey]);
        }
      }
    }
    swap(phi1,phi2);
  }
}
```
OpenACC Simple Example

```c
void smooth( float* restrict a, float* restrict b,
            float w0, float w1, float w2, int n, int m, int niters )
{
    int i, j, iter;
    float* tmp;
    for( iter = 1; iter < niters; ++iter ){
        #pragma acc parallel loop gang(16) worker(8) // chunk across gangs and workers
        for( i = 1; i < n-1; ++i )
            #pragma acc vector (32) // execute in SIMD mode
                for( j = 1; j < m-1; ++j )
                    a[i*m+j] = w0 * b[i*m+j] +
                                w1*(b[(i-1)*m+j] + b[(i+1)*m+j] + b[i*m+j-1] +
                                        b[i*m+j+1]) +
                                w2*(b[(i-1)*m+j-1] + b[(i-1)*m+j+1] +b[(i+1)*m+j-1] +
                                        b[(i+1)*m+j+1]);
        tmp = a;  a = b;  b = tmp;
    }
}
In main:
#pragma acc data copy (b[0:n*m],a[0:n*m])
{
    smooth( a, b, w0, w1, w2, n, m, iters );
}
```

CAPS HMPPWorkbench compiler:

```
acc_test.c:11: Loop 'j' was vectorized(32)
acc_test.c:9 : Loop 'i' was shared among gangs(16) and workers(8)
```
Mantevo miniGhost on Cray XK7

- Mantevo 1.0.1 miniGhost 1.0
  - Finite-Difference Proxy Application
  - 27 PT Stencil + Boundary Exchange of Ghost Cells
  - Implemented in Fortran;
  - MPI+OpenMP and MPI+OpenACC
    - http://www.mantevo.org

- Test System:
  - Located at HLRS Stuttgart,

- Test Case:
  - Problem size 384x796x384, 10 variables, 20 time steps

- Compilation:
  - *pgf90 13.4-0 -O3 -fast -fastsse -m -acc

```fortran
!$acc data present (GRID)

! Back boundary

IF ( NEIGHBORS(BACK) /= -1 ) THEN
  TIME_START_DIR = MG_TIMER ()
!$acc data present (SEND_BUFFER_BACK)
!$acc parallel loop
  DO J = 0, NY+1
    DO I = 0, NX+1
      SEND_BUFFER_BACK(COUNT_SEND_BACK + J*(NX+2) + I + 1) = &
GRID (I, J, 1)
    END DO
  END DO
!$acc end data
#endif
```

Packing of boundary data

```fortran
CALL MPI_WAITANY (MAX_NUM_SENDS + MAX_NUM_RECVS, MSG_REQS, ... )
....
!$acc data present (RECV_BUFFER_BACK)
!$acc update device (RECV_BUFFER_BACK)
!$acc end data
```

Unpacking of boundary data
Mantevo miniGhost: 27-PT Stencil

```c
#if defined _MOG_OMP
!$OMP PARALLEL DO PRIVATE(SLICE_BACK, SLICE_MINE, SLICE_FRONT)
#else
!$acc data present ( WORK )
!$acc parallel
!$acc loop
#endif
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
      SLICE_BACK = GRID(I-1,J-1,K-1) + GRID(I-1,J,K-1) + GRID(I-1,J+1,K-1) + &
                   GRID(I,J-1,K-1) + GRID(I,J,K-1) + GRID(I,J+1,K-1) + &
                   GRID(I+1,J-1,K-1) + GRID(I+1,J,K-1) + GRID(I+1,J+1,K-1)
      SLICE_MINE =  GRID(I-1,J-1,K)   + GRID(I-1,J,K)   + GRID(I-1,J+1,K) + &
                    GRID(I,J-1,K)   + GRID(I,J,K)   + GRID(I,J+1,K) + &
                    GRID(I+1,J-1,K) + GRID(I+1,J,K) + GRID(I+1,J+1,K)
      SLICE_FRONT = GRID(I-1,J-1,K+1) + GRID(I-1,J,K+1) + GRID(I-1,J+1,K+1) + &
                    GRID(I,J-1,K+1) + GRID(I,J,K+1) + GRID(I,J+1,K+1) + &
                    GRID(I+1,J-1,K+1) + GRID(I+1,J,K+1) + GRID(I+1,J+1,K+1)
      WORK(I,J,K) = ( SLICE_BACK + SLICE_MINE + SLICE_FRONT ) / 27.0
    END DO
  END DO
END DO
```
Cray XK7 Hermit

- Located at HLRS Stuttgart, Germany ([https://wickie.hlrs.de/platforms/index.php/Cray_XE6](https://wickie.hlrs.de/platforms/index.php/Cray_XE6))
- 3552 compute nodes 113.664 cores
- Two AMD 6276 Interlagos processors with 16 cores each, running at 2.3 GHz (TurboCore 3.3GHz) per node
- Around 1 Pflop theoretical peak performance
- 32 GB of main memory available per node
- 32-way shared memory system
- High-bandwidth interconnect using Cray Gemini communication chips

---

CPU type: AMD Interlagos processor

Hardware Thread Topology

Sockets: 1
Cores per socket: 16
Threads per core: 1

Socket 0:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
<td>16kB</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Scalability of miniGhost on Cray XK7

- **OpenMP (1 MPI per node, 1 thread)**
- **OpenMP (1 MPI per node, 16 threads)**
- **OpenACC (1 MPI per node, 1 thread)**
- **Pure MPI (16 MPI per node)**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Total Time (sec)</th>
<th>Comm. Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP (16x11)</td>
<td>12.1</td>
<td>0.4</td>
</tr>
<tr>
<td>OpenMP (16x16t)</td>
<td>1.9</td>
<td>0.16</td>
</tr>
<tr>
<td>OpenACC (16x16t)</td>
<td>1.17</td>
<td>0.34</td>
</tr>
<tr>
<td>Pure MPI (256 Ranks)</td>
<td>1.5</td>
<td>0.28</td>
</tr>
</tbody>
</table>

Elapsed time as reported by the application.
Communication includes packing/unpacking.
Profiling Information: export PGI_ACC_TIME=1

/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_UNPACK_BSPMA.F
mg_unpack_bspma NVIDIA devicenum=0
time(us): 36,951
124: data copyin reached 20 times
device time(us): total=8,603 max=431 min=429 avg=430
...

/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_STENCIL_COMPS.F
mg_stencil_3d27pt NVIDIA devicenum=0
time(us): 1,063,875
330: kernel launched 200 times
grid: [160] block: [256]
device time(us): total=1,063,875 max=5,337 min=5,302 avg=5,319
elapsed time(us): total=1,073,817 max=5,444 min=5,349 avg=5,369
...

/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_SEND_BSPMA.F
mg_send_bspma NVIDIA devicenum=0
time(us): 33,150
94: data copyout reached 20 times
device time(us): total=7,800 max=392 min=389 avg=390
...

device time(us): total=12,618 max=633 min=630 avg=630

/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_PACK.F
mg_pack NVIDIA devicenum=0
time(us): 9,615
91: kernel launched 200 times
grid: [98] block: [256]
device time(us): total=2,957 max=68 min=13 avg=14
elapsed time(us): total=11,634 max=107 min=51 avg=58
Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators

Profiling Information: export PGI_ACC_TIME=1

Accelerator Kernel Timing data
/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_STENCIL_COMPS.F

mg_stencil_3d27pt  NVIDIA  devicenum=0
time(us): 1,064,197
330: kernel launched 200 times
  grid: [160]  block: [256]
  device time(us): total=1,064,197  max=5,351  min=5,299  avg=5,320
  elapsed time(us): total=1,074,081  max=5,442  min=5,348  avg=5,370

/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_PACK.F

mg_pack  NVIDIA  devicenum=0
time(us): 9,568
91: kernel launched 200 times
  grid: [98]  block: [256]
  device time(us): total=2,924  max=70  min=12  avg=14
  elapsed time(us): total=11,624  max=110  min=51  avg=58
195: kernel launched 200 times
  grid: [162]  block: [256]
  device time(us): total=3,432  max=120  min=15  avg=17
  elapsed time(us): total=11,385  max=160  min=53  avg=56
221: kernel launched 200 times
  grid: [162]  block: [256]
  device time(us): total=3,212  max=19  min=15  avg=16
  elapsed time(us): total
MPI+Accelerators: Main advantages

- Hybrid MPI/OpenMP and MPI/OpenACC can leverage accelerators and yield performance increase over pure MPI on multicore
- Compiler pragma based API provides relatively easy way to use coprocessors
- OpenACC targeted toward GPU type coprocessors
- OpenMP 4.0 extensions provide flexibility to use a wide range of heterogeneous coprocessors (GPU, APU, heterogeneous many-core types)
MPI+Accelerators: Main challenges

- Considerable implementation effort for **basic usage**, depending on complexity of the application

- **Efficient usage** of pragmas may require high implementation effort and good understanding of performance issues

- Not many compilers support accelerator pragmas (yet)
Tools

- **Topology & Affinity**
- Tools for debugging and profiling
  MPI+OpenMP
Tools for Thread/Process Affinity ("Pinning")

- Likwid tools → slides in section MPI+OpenMP
  - `likwid-topology` prints SMP topology
  - `likwid-pin` binds threads to cores / HW threads
  - `likwid-mpirun` manages affinity for hybrid MPI+OpenMP

- `numactl`
  - Standard in Linux numatools, enables restricting movement of thread team but no individual thread pinning
  - `taskset` provides a subset of `numactl` functionality

- OpenMP 4.0 thread/core/socket binding

- Vendor-specific solutions
  - Intel, IBM, Cray, GCC, OpenMPI,...
Tools

- Topology & Affinity
- Tools for debugging and profiling
  MPI+OpenMP
Thread Correctness – Intel ThreadChecker  1/3

- Intel ThreadChecker operates in a similar fashion to helgrind,
- Compile with `-tcheck`, then run program using `tcheck_cl`:

```
Application finished
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity</th>
<th>Context</th>
<th>Description</th>
<th>1st Acc</th>
<th>2nd Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write -&gt; Error</td>
<td>1</td>
<td>pthread</td>
<td>Memory write of global_variable at pthread pthread</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d_race.c&quot;:31 conflicts with d_race.d_race</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;pthread_race.c&quot;:31 (output dependence)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With new Intel Inspector XE 2015: Command line interface must be used within mpirun / mpiexec
Thread Correctness – Intel ThreadChecker 2/3

- One may output to HTML:

  tcheck_cl --format HTML --report pthread_race.html pthread_race

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write -&gt; Write data race</td>
<td>Error</td>
<td>1</td>
<td>pthread_race.c:25</td>
<td>Memory write of global variable at pthread_race.c:31 conflicts with a prior memory write of global_variable at pthread_race.c:31 (output dependence)</td>
<td>pthread_race.c:31</td>
<td>pthread_race.c:31</td>
</tr>
<tr>
<td>2</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 1</td>
<td>Thread termination at pthread_race.c:43 - includes stack allocation of 8,004 MB and use of 4,672 KB</td>
<td>pthread_race.c:43</td>
<td>pthread_race.c:43</td>
</tr>
<tr>
<td>3</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 2</td>
<td>Thread termination at pthread_race.c:43 - includes stack allocation of 8,004 MB and use of 4,672 KB</td>
<td>pthread_race.c:43</td>
<td>pthread_race.c:43</td>
</tr>
<tr>
<td>4</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 3</td>
<td>Thread termination at pthread_race.c:37 - includes stack allocation of 8 MB and use of 4,25 KB</td>
<td>pthread_race.c:37</td>
<td>pthread_race.c:37</td>
</tr>
</tbody>
</table>
• If one wants to compile with threaded Open MPI (option for IB):

```bash
configure --enable-mpi-threads --enable-debug --enable-mca-no-build=memory-ptmalloc2
CC=icc F77=ifort FC=ifort CFLAGS='--debug all -inline-debug-info tcheck'
CXXFLAGS='--debug all -inline-debug-info tcheck'
FFLAGS='--debug all -tcheck' LDFLAGS='tcheck'
```

• Then run with:

```bash
mpirun --mca tcp,sm,self -np 2 tcheck_cl
    --reinstrument -u full --format html
    --cache_dir '/tmp/my_username_$$__tc_cl_cache'
    --report 'tc_mpi_test_suite_$$'
    --options 'file=tc_my_executable_%H_%I,
        pad=128, delay=2, stall=2' ./my_executable my_arg1 my_arg2 ...
```
Performance Tools Support for Hybrid Code

- Paraver examples have already been shown, tracing is done with linking against (closed-source) `omptrace` or `ompitrace`.

- For Vampir/Vampirtrace performance analysis:
  ```
  ./configure --enable-omp
  --enable-hyb
  --with-mpi-dir=/opt/OpenMPI/1.3-icc
  CC=icc F77=ifort FC=ifort
  (Attention: does not wrap MPI_Init_thread!)
  ```
Indication of non-optimal load balance.
Scalasca – Example “Wait at Barrier”, Solution

Better load balancing with dynamic loop schedule
Conclusions
In principle, none of the programming models perfectly fits to clusters of SMP nodes

Major advantages of hybrid MPI+OpenMP:

- Only one level of sub-domain “surface-optimization”:
  - SMP nodes, or
  - Sockets
- Second level of parallelization
  - Application may scale to more cores
- Smaller number of MPI processes implies:
  - Reduced size of MPI internal buffer space
  - Reduced space for replicated user-data

Most important arguments on many-core systems, e.g., Intel Phi
Major advantages of hybrid MPI+OpenMP, continued

• Reduced communication overhead
  – No intra-node communication
  – Longer messages between nodes and fewer parallel links may imply better bandwidth

• “Cheap” load-balancing methods on OpenMP level
  – Application developer can split the load-balancing issues between course-grained MPI and fine-grained OpenMP
Disadvantages of MPI+OpenMP

- Using OpenMP
  → may prohibit compiler optimization
  → may cause significant loss of computational performance
- Thread fork / join overhead
- On ccNUMA SMP nodes:
  - Loss of performance due to missing memory page locality or missing first touch strategy
    - E.g., with the MASTERONLY scheme:
      - One thread produces data
      - Master thread sends the data with MPI
        → data may be internally communicated from one memory to the other one
- Amdahl’s law for each level of parallelism
- Using MPI-parallel application libraries? → Are they prepared for hybrid?
- Using thread-local application libraries? → Are they thread-safe?

See, e.g., the necessary \(-O4\) flag with mpxlf_r on IBM Power6 systems.
MPI+OpenMP versus MPI+MPI-3.0 shared mem.

MPI+3.0 shared memory
- **Pro:** Thread-safety is not needed for libraries.
- **Con:** No work-sharing support as with OpenMP directives.
- **Pro:** Replicated data can be reduced to one copy per node:
  May be helpful to save memory,
  **if pure MPI scales in time, but not in memory.**
- Substituting intra-node communication by shared memory loads or stores
  has only limited benefit (and only on some systems),
  especially if the communication time is dominated by inter-node communication
- **Con:** No reduction of MPI ranks
  → no reduction of MPI internal buffer space
- **Con:** Virtual addresses of a shared memory window
  may be different in each MPI process
  → no binary pointers
  → i.e., linked lists must be stored with offsets rather than pointers
Lessons for pure MPI and ccNUMA-aware hybrid MPI+OpenMP

- MPI processes on an SMP node should form a cube and not a long chain
  - Reduces inter-node communication volume

- For structured or Cartesian grids:
  - Adequate renumbering of MPI ranks and process coordinates

- For unstructured grids:
  - Two levels of domain decomposition
    - First fine-grained on the core-level
    - Recombining cores to SMP-nodes
We want to thank

- Gabriele Jost, Supersmith, Maximum Performance Software, USA
  - Co-author of several slides and previous tutorials
- Gerhard Wellein, RRZE
- Alice Koniges, NERSC, LBNL
- Rainer Keller, HLRS and ORNL
- Jim Cownie, Intel
- SCALASCA/KOJAK project at JSC, Research Center Jülich
- HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS
- Steffen Weise, TU Freiberg
- Vincent C. Betro et al., NICS – access to beacon with Intel Xeon Phi
Conclusions

• Future hardware will be more complicated
  – Heterogeneous → GPU, FPGA, …
  – ccNUMA quality may be lost on cluster nodes
  – ….  
• High-end programming → more complex → many pitfalls
• Medium number of cores → more simple
  (if \#cores / SMP-node will not shrink)
• MPI + OpenMP → work horse on large systems
  – Major pros: reduced memory needs and second level of parallelism
• MPI + MPI-3 → only for special cases and medium rank number
• Pure MPI → still on smaller cluster
• OpenMP only → on large ccNUMA nodes

Thank you for your interest

Q & A

Please fill out the feedback sheet – Thank you
Appendix

- Solutions to the exercises
- Abstract
- Authors
- References (with direct relation to the content of this tutorial)
- Further references
Chapter 11: Ring with shared memory one-sided comm.

```c
int snd_buf;
int *rcv_buf_ptr;

MPI_Alloc_mem((MPI_Aint)(1*sizeof(int)), MPI_INFO_NULL, &rcv_buf_ptr);
MPI_Win_create(rcv_buf_ptr, (MPI_Aint)(1*sizeof(int)), sizeof(int),
               MPI_INFO_NULL, MPI_COMM_WORLD, &win);

MPI_Win_allocate_shared((MPI_Aint)(1*sizeof(int)), sizeof(int),
                        MPI_INFO_NULL, MPI_COMM_WORLD,
                        &rcv_buf_ptr, &win);

And all fences without assertions (as long as not otherwise standardized):
for( i = 0; i < size; i++)
{
    MPI_Win_fence(/*workaround: no assertions*/ 0, win);
    /* MPI_Put(&snd_buf,1,MPI_INT,right,(MPI_Aint) 0, 1, MPI_INT, win); */
    /* ... is substituted by (with offset "right-my_rank" to store
       into right neighbor's rcv_buf): */
    *(rcv_buf_ptr+(right-my_rank)) = snd_buf;
    MPI_Win_fence(/*workaround: no assertions*/ 0, win);
    snd_buf = *rcv_buf_ptr;
    sum += *rcv_buf_ptr;
}
printf ("PE%i:\tSum = %i\n", my_rank, sum);
MPI_Win_free(&win);
MPI_Free_mem(rcv_buf_ptr);
```
Chapter 11: Ring with shared memory one-sided comm.

```fortran
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
IMPLICIT NONE

INTEGER :: snd_buf
INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf(:)
TYPE(C_PTR) :: ptr_rcv_buf

TYPE(MPI_Win) :: win
INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, iadummy
INTEGER(KIND=MPI_ADDRESS_KIND) :: rcv_buf_size, target_disp

CALL MPI_Type_get_extent(MPI_INTEGER, lb, integer_size)
rcv_buf_size = 1 * integer_size
disp_unit = integer_size
CALL MPI_Win_allocate_shared(rcv_buf_size, disp_unit, MPI_INFO_NULL, MPI_COMM_WORLD, ptr_rcv_buf, win)
CALL C_F_POINTER(ptr_rcv_buf, rcv_buf, (/1/))
! target_disp = 0

Substitution of MPI_Put → see next slide
```
Chapter 11: Ring with shared memory one-sided comm.

Fortran

CALL MPI_GET_ADDRESS(rcv_buf, iadummy)
! ... should be substituted as soon as possible by:
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)

CALL MPI_Win_fence( 0, win) ! Workaround: no assertions

CALL MPI_GET_ADDRESS(rcv_buf, iadummy)
! ... should be substituted as soon as possible by:
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)

CALL MPI_PUT(snd_buf,1,MPI_INTEGER,right,target_disp,1,MPI_INTEGER,win)
rcv_buf(1+(right-my_rank)) = snd_buf

CALL MPI_GET_ADDRESS(rcv_buf, iadummy)
! ... should be substituted as soon as possible by:
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)

CALL MPI_WIN_FENCE( 0, win) ! Workaround: no assertions

CALL MPI_GET_ADDRESS(rcv_buf, iadummy)
! ... should be substituted as soon as possible by:
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)

MPI_F_SYNC_REG(rcv_buf_right/left) guarantees that the assignments rcv_buf = … must not be moved across both MPI_Win_fence
int k;
int offset_left, offset_right;

MPI_Win_allocate_shared((MPI_Aint)(max_length*sizeof(float)),
    sizeof(float), MPI_INFO_NULL, MPI_COMM_WORLD, &rcv_buf_left,
    &win_rcv_buf_left);
MPI_Win_allocate_shared((MPI_Aint)(max_length*sizeof(float)),
    sizeof(float), MPI_INFO_NULL, MPI_COMM_WORLD, &rcv_buf_right,
    &win_rcv_buf_right);

/*offset_left is defined so that rcv_buf_left(xxx+offset_left) in
 process 'my_rank' is the same location as rcv_buf_left(xxx) in
 process 'left': */
offset_left = (left-my_rank)*max_length;

/*offset_right is defined so that rcv_buf_right(xxx+offset_right) in
 process 'my_rank' is the same location as rcv_buf_right(xxx) in
 process 'right': */
offset_right = (right-my_rank)*max_length;

MPI_Put(snd_buf_left, length, MPI_FLOAT, left, (MPI_Aint)0, length,
    MPI_FLOAT, win_rcv_buf_right); /*
MPI_Put(snd_buf_right, length, MPI_FLOAT, right, (MPI_Aint)0, length,
    MPI_FLOAT, win_rcv_buf_left ); ... is substited by: */
for(k=0; k<length; k++) rcv_buf_right[k+offset_left] = snd_buf_left [k];
for(k=0; k<length; k++) rcv_buf_left [k+offset_right]= snd_buf_right[k];

And all fences without assertions (as long as not otherwise standardized):
MPI_Win_fence( /*workaround: no assertions:*/ 0, ...);
INTEGER :: offset_left, offset_right

CALL MPI_Win_allocate_shared(buf_size, disp_unit, MPI_INFO_NULL,
    MPI_COMM_WORLD, ptr_rcv_buf_left, win_rcv_buf_left)
CALL C_F_POINTER(ptr_rcv_buf_left, rcv_buf_left, (/max_length/))
! offset_left is defined so that rcv_buf_left(xxx+offset_left) in process
! 'my rank' is the same location as rcv_buf_left(xxx) in process 'left':
    offset_left = (left-my_rank)*max_length

CALL MPI_Win_allocate_shared(buf_size, disp_unit, MPI_INFO_NULL,
    MPI_COMM_WORLD, ptr_rcv_buf_right, win_rcv_buf_right)
CALL C_F_POINTER(ptr_rcv_buf_right, rcv_buf_right, (/max_length/))
! offset_right is defined so that rcv_buf_right(xxx+offset_right) in proc.
! 'my rank' is the same location as rcv_buf_right(xxx) in process 'right':
    offset_right = (right-my_rank)*max_length

Substitution of MPI_Put \rightarrow see next slide
Chapter 11: Halo-copy shared memory one-sided comm.

CALL MPI_Win_fence( 0, win_rcv_buf_left) ! Workaround: no assertions
CALL MPI_Win_fence( 0, win_rcv_buf_right) ! Workaround: no assertions

! CALL MPI_Get_address(rcv_buf_right, iadummy)
! CALL MPI_Get_address(rcv_buf_left, iadummy)
! ... or with MPI-3.0 and later:
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_right)
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_left)

CALL MPI_Put(snd_buf_left, length, MPI_REAL, left, target_disp, &
length, MPI_REAL, win_rcv_buf_right)
CALL MPI_Put(snd_buf_right, length, MPI_REAL, right, target_disp, &
length, MPI_REAL, win_rcv_buf_left)
...
... is substited by:
rcv_buf_right(1+offset_left:length+offset_left) = snd_buf_left(1:length)
rcv_buf_left(1+offset_right:length+offset_right) = snd_buf_right(1:length)

CALL MPI_Get_address(rcv_buf_right, iadummy)
CALL MPI_Get_address(rcv_buf_left, iadummy)
... or with MPI-3.0 and later:
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_right)
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_SYNC_REG(rcv_buf_left)

CALL MPI_Win_fence( 0, win_rcv_buf_left) ! Workaround: no assertions
CALL MPI_Win_fence( 0, win_rcv_buf_right) ! Workaround: no assertions

MPI_F_SYNC_REG(rcv_buf_right/left) guarantees that the assignments rcv_uf_right/left = …
must not be moved across both MPI_Win_fence
Chapter 11: Ring with shared memory and MPI_Win_sync

```c
MPI_Request rq;
MPI_Status status;
int snd_dummy, rcv_dummy;

MPI_Win_allocate_shared(...);
MPI_Win_lock_all(MPI_MODE_NOCHECK, win);

/* In Fortran, a register-sync would be here needed:
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf) */
MPI_Win_sync(win);
MPI_Irecv(&rcv_dummy, 0, MPI_INTEGER, right, 17, MPI_COMM_WORLD, &rq);
MPI_Send (&snd_dummy, 0, MPI_INTEGER, left, 17, MPI_COMM_WORLD);
MPI_Wait(&rq, &status);
MPI_Win_sync(win);

/* In Fortran ... IF (...) CALL MPI_F_sync_reg(rcv_buf) */
*(rcv_buf_ptr+(right-my_rank)) = snd_buf;

/* In Fortran ... IF (...) CALL MPI_F_sync_reg(rcv_buf) */
MPI_Win_sync(win);
MPI_Irecv(&rcv_dummy, 0, MPI_INTEGER, left, 17, MPI_COMM_WORLD, &rq);
MPI_Send (&snd_dummy, 0, MPI_INTEGER, right, 17, MPI_COMM_WORLD);
MPI_Wait(&rq, &status);
MPI_Win_sync(win);

/* In Fortran ... IF (...) CALL MPI_F_sync_reg(rcv_buf) */

MPI_Win_unlock_all(win);
MPI_Win_free(&win);
```
Abstract

Half-Day Tutorial  (Level: 25% Introductory, 50% Intermediate, 25% Advanced)

Authors.  Rolf Rabenseifner, HLRS, University of Stuttgart, Germany
Georg Hager, University of Erlangen-Nuremberg, Germany

Abstract.  Most HPC systems are clusters of shared memory nodes. Such SMP nodes can be small multi-core CPUs up to large many-core CPUs. Parallel programming may combine the distributed memory parallelization on the node interconnect (e.g., with MPI) with the shared memory parallelization inside of each node (e.g., with OpenMP or MPI-3.0 shared memory).

This tutorial analyzes the strengths and weaknesses of several parallel programming models on clusters of SMP nodes. Multi-socket-multi-core systems in highly parallel environments are given special consideration. MPI-3.0 introduced a new shared memory programming interface, which can be combined with inter-node MPI communication. It can be used for direct neighbor accesses similar to OpenMP or for direct halo copies, and enables new hybrid programming models. These models are compared with various hybrid MPI+OpenMP approaches and pure MPI. This tutorial also includes a discussion on OpenMP support for accelerators. Benchmark results are presented for modern platforms such as Intel Xeon Phi and Cray XC30. Numerous case studies and micro-benchmarks demonstrate the performance-related aspects of hybrid programming. The various programming schemes and their technical and performance implications are compared. Tools for hybrid programming such as thread/process placement support and performance analysis are presented in a "how-to" section.

URL.  http://sc15.supercomputing.org/schedule/event_detail?evid=tut137
http://www.lrz.de/services/compute/courses/2016-01-14_hhyp1w15/
http://www.hlrs.de/training/2016/HY-G
Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he has worked at the High-Performance Computing-Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without loosing the full MPI interface. In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum and since Dec. 2007 he is in the steering committee of the MPI-3 Forum and was responsible for new MPI-2.1 standard. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology.
Currently, he is head of Parallel Computing - Training and Application Services at HLRS. He is involved in MPI profiling and benchmarking, e.g., in the HPC Challenge Benchmark Suite. In recent projects, he studied parallel I/O, parallel programming models for clusters of SMP nodes, and optimization of MPI collective routines. In workshops and summer schools, he teaches parallel programming models in many universities and labs in Germany. In January 2012, the Gauss Center of Supercomputing (GCS), with HLRS, LRZ in Garching and the Jülich Supercomputing Center as members, was selected as one of six PRACE Advanced Training Centers (PATCs) and he was appointed as GCS’PATC director.
Georg Hager

Georg Hager studied theoretical physics at the University of Bayreuth, specializing in nonlinear dynamics, and holds a PhD in Computational Physics from the University of Greifswald. He is a senior researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE), which is part of the University of Erlangen-Nuremberg. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes on chip and system levels, and special topics in shared memory and hybrid programming. His daily work encompasses all aspects of user support in High Performance Computing like tutorials and training, code parallelization, profiling and optimization, and the assessment of novel computer architectures and tools. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended or required reading in many HPC-related lectures and courses worldwide.

In his teaching activities he puts a strong focus on performance modeling techniques that lead to a better understanding of the interaction of program code with the hardware. A full list of publications, talks, and other things he is interested in can be found in his blog:
http://blogs.fau.de/hager.
References (with direct relation to the content of this tutorial)

- **NAS Parallel Benchmarks:**
  http://www.nas.nasa.gov/Resources/Software/npb.html


References


- Rolf Rabenseifner, Comparison of Parallel Programming Models on Clusters of SMP Nodes. In proceedings of the 45nd Cray User Group Conference, CUG SUMMIT 2003, May 12-16, Columbus, Ohio, USA.


References


References


• Barbara Chapman, Gabriele Jost, and Ruud van der Pas: *Using OpenMP.* The MIT Press, 2008.


References

- J. Treibig, G. Hager and G. Wellein:  
  **LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.**  

- H. Stengel:  
  **Parallel programming on hybrid hardware: Models and applications.**  
  http://www.hpc.rrze.uni-erlangen.de/Projekte/hybrid.shtml

- Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, Rajeev Thakur:  
  **MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory.**  
Further references

• Sergio Briguglio, Beniamino Di Martino, Giuliana Fogaccia and Gregorio Vlad, *Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of Symmetric MultiProcessors*, 10th European PVM/MPI Users' Group Conference (EuroPVM/MPI'03), Venice, Italy, 29 Sep - 2 Oct, 2003

• Barbara Chapman, *Parallel Application Development with the Hybrid MPI+OpenMP Programming Model*, Tutorial, 9th EuroPVM/MPI & 4th DAPSYS Conference, Johannes Kepler University Linz, Austria September 29-October 02, 2002


• Nikolaos Drosinos and Nectarios Koziris, *Advanced Hybrid MPI/OpenMP Parallelization Paradigms for Nested Loop Algorithms onto Clusters of SMPs*, 10th European PVM/MPI Users' Group Conference (EuroPVM/MPI'03), Venice, Italy, 29 Sep - 2 Oct, 2003
Further references

- Holger Brunst and Bernd Mohr, *Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with VampirNG*

- Felix Wolf and Bernd Mohr, *Automatic performance analysis of hybrid MPI/OpenMP applications*

- Felix Wolf and Bernd Mohr, *Automatic Performance Analysis of Hybrid MPI/OpenMP Applications*
Further references


• Jonathan Harris, **Extending OpenMP for NUMA Architectures**, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000.

Further references


Further references


Further references
