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

Rolf Rabenseifner¹)
Rabenseifner@hlrs.de

Georg Hager²)
Georg.Hager@fau.de

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

LRZ Garching, January 12, 2017

VSC slides by courtesy of C. Blaas-Schenner, TU Vienna
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 (42)

• **Programming models** (44)
  – Pure MPI communication (45)
  – MPI + MPI-3.0 shared memory (76)
    • **Five examples** (93 and 110)
  – MPI + OpenMP on multi/many-core (123)
  – MPI + Accelerators (187)

• **Tools** (208)
  – Topology & Affinity (208)
  – Debug, profiling (210)

• **Conclusions** (217)
  Major opportunities and challenges of “MPI+X”
  – MPI+OpenMP (217)
  – MPI+MPI-3.0 (221)
  – Pure MPI comm. (222)
  – Acknowledgements (223)
  – Conclusions (224)

• **Appendix** (225)
  – Examples (225)
  – Abstract (226)
  – Authors (227)
  – References (229)
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
  – …

Programming models
• MPI + Threading
  – OpenMP
  – Cilk(+)
  – TBB (Threading Building Blocks)
• MPI + MPI shared memory
• MPI + Accelerator
  – OpenACC
  – OpenMP 4.0 accelerator support
  – CUDA
  – OpenCL
  – …
• Pure MPI communication

Motivation
Introduction
Programming models
Tools
Conclusions
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, …
Number of options multiply if accelerators are added

- One MPI process per accelerator?
- One thread per accelerator?
- Which programing model on the accelerator?
  - OpenMP shared memory
  - MPI
  - OpenACC
  - OpenMP-4.0 accelerator
  - CUDA
  - ...

Options for running code
Hierarchical hardware

- Cluster of
  - ccNUMA nodes with
    - CPUs/GPUs/accel. with
      - N x
        - M cores with
          - Hyperthreads/
            SIMD/
            CUDA “cores” …

Hierarchical parallel programming

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

Many possibilities for splitting the hardware hierarchy into MPI + X:
- 1 MPI process per shared memory 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 communication

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
Hardware Bottlenecks

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
How much bandwidth does non-local access cost?

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38.7 GB/s</td>
</tr>
<tr>
<td>1</td>
<td>14.6 GB/s</td>
</tr>
</tbody>
</table>

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!)
Info about nodes:

- **numactl** - control NUMA policy for processes or shared memory
- **numactl --show** (no info about caches)
- **numactl --hardware**
- **module load intel/16.0.3 intel-mpi/5.1.3 ; cpuinfo [-A]**
- **module load likwid/4.0 ; likwid-topology -c -g**
$ likwid-topology -c -g

-----------------------------------------------------------------------------------
CPU name: Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
CPU type: Intel Xeon IvyBridge EN/EP/EX processor
CPU stepping: 4
*********************************************************************************
Hardware Thread Topology
*********************************************************************************
Sockets: 2
Cores per socket: 8
Threads per core: 2
-----------------------------------------------------------------------------------
<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>Socket 0:</td>
<td>( 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket 1:</td>
<td>( 8 24 9 25 10 26 11 27 12 28 13 29 14 30 15 31 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
-----------------------------------------------------------------------------------
VSC-3 – output of: likwid-topology -c -g (cont.)

(...cont.-compressed...)

***********************************************************************
Cache Topology
***********************************************************************

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>32 kB</td>
<td>256 kB</td>
<td>20 MB</td>
</tr>
<tr>
<td>Type</td>
<td>Data cache</td>
<td>Unified cache</td>
<td>Unified cache</td>
</tr>
<tr>
<td>Associativity</td>
<td>8</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>Number of sets</td>
<td>64</td>
<td>512</td>
<td>16384</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Cache type</td>
<td>Non Inclusive</td>
<td>Non Inclusive</td>
<td>Inclusive</td>
</tr>
<tr>
<td>Shared by threads</td>
<td>2</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

Cache groups: ( 0 16 ) ( 1 17 ) ( 2 18 ) ( 3 19 ) ( 4 20 ) ( 5 21 ) ( 6 22 ) ( 7 23 )

L1 / L2: ( 8 24 ) ( 9 25 ) ( 10 26 ) ( 11 27 ) ( 12 28 ) ( 13 29 ) ( 14 30 ) ( 15 31 )

Cache groups: ( 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 )

L3: ( 8 24 9 25 10 26 11 27 12 28 13 29 14 30 15 31 )
NUMA Topology

NUMA domains: 2

Domain: 0
Processors: (0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23)
Distances: 10 21
Free memory: 634.172 MB
Total memory: 32734.9 MB

Domain: 1
Processors: (8 24 9 25 10 26 11 27 12 28 13 29 14 30 15 31)
Distances: 21 10
Free memory: 1578.21 MB
Total memory: 32768 MB
VSC-3 – output of: likwid-topology -c -g (cont.)

(...cont...)

Graphical Topology

Socket 0:
VSC-3 – output of: likwid-topology -c -g (cont.)

(...cont...)

Graphical Topology

Socket 1:
Output of likwid-topology

CPU name: Intel Core i7 processor
CPU clock: 266683826 Hz

Hardware Thread Topology

<table>
<thead>
<tr>
<th>HWTThread</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>

Motivation
Introduction
Programming models
Tools
Conclusions

Prevalent hardware bottleneck
Interlude: ccNUMA
The role of machine topology
Cost-Benefit Calculation
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 \texttt{-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

Latency [µs]

- Internode: 1.8
- Intranode 2S: 0.56
- Intranode 1S: 0.3

Affinity matters!
HPC networks – Topologies

**Torus:** 2d, 3d, 5d, 6d torus,
was: BlueGene, Cray, ...
is: Fujitsu K-Computer

**Fat-tree:**
is: Cray XC → **Dragonfly** (scalable)
→ clusters → next slide

Topology:
- bandwidth
- latency
- cost
- scalability
**Fat-tree Design**

**VSC-3:**
dual rail Intel QDR-80 = 3-level fat-tree (BF: 2:1 / 4:1)

VSC-3: below numbers only, schematic figure

non-blocking: BF 1:1

blocking: BF down- : up-links

introduces a latency:
packets that would otherwise follow separate paths would eventually have to wait

*Leaf*

-2- leaf: island @VSC-3:
288 nodes
4608 cores / MPI processes
(BF 2:1)

*Spine*

-3- spine: -- -- full cluster @VSC-3:
(BF 4:1)

*Core*

-2- leaf: island @VSC-3:
288 nodes
4608 cores / MPI processes
(BF 2:1)

*Edge*

@VSC-3:
-1- edge
12 nodes
24 HCAs
192 cores / MPI processes

VSC-3: likwid-topology, ping-pong benchmark

Claudia Blaas-Schenner

Vienna Scientific Cluster

Slide 33 / 224
### Ping-Pong Benchmark – Latency

**intra-node vs. inter-node on VSC-3**

- **nodes** = 2 sockets (Intel Ivy Bridge) with 8 cores + 2 HCAs
- **inter-node** = IB fabric = dual rail Intel QDR-80 = 3-level fat-tree (BF: 2:1 / 4:1)

Avoiding slow data paths is the key to most performance optimizations!

<table>
<thead>
<tr>
<th>Latency [µs]</th>
<th>MPI_Send(…)</th>
<th>Typical latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenMPI</td>
<td>Intel-MPI</td>
</tr>
<tr>
<td>intra-socket</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>inter-socket</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td>IB -1- edge</td>
<td>1.2</td>
<td>1.4</td>
</tr>
<tr>
<td>IB -2- leaf</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td>IB -3- spine</td>
<td>2.1</td>
<td>2.3</td>
</tr>
</tbody>
</table>
Ping-Pong Benchmark – Bandwidth

**intra-node vs. inter-node on VSC-3**

inter-node:
IB fabric
dual rail (2 HCAs)
Intel QDR-80
3-level fat-tree
BF: 2:1 / 4:1

QDR-80 (2 HCAs)
link: 80 Gbit/s
max 8 Gbytes/s
eff. 6.8 Gbytes/s

→ 1 HCA = ½ (2 HCAs)
Ping-Pong Benchmark – Bandwidth – log

intra-node vs. inter-node on VSC-3

<table>
<thead>
<tr>
<th></th>
<th>typical bandwidths</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>100 GB/s</td>
</tr>
<tr>
<td>L2/L3 c.</td>
<td>50 GB/s</td>
</tr>
<tr>
<td>memory</td>
<td>10 GB/s</td>
</tr>
<tr>
<td>HPC networks</td>
<td>1–8 GB/s</td>
</tr>
</tbody>
</table>

Bandwidth very similar

Effective bandwidth [MBytes/s]

message length [Bytes]

Intel-MPI: solid / dark
OpenMPI: dotted / light

Latency is different

intra-socket
inter-socket
inter-node -1- edge
inter-node -2- leaf
inter-node -3- spine
IMB Ping-Pong: Bandwidth Characteristics

Intra-node vs. Inter-node on Cray XE6

- Bandwidth: Surprisingly similar!
- Latency: Very different!

Between two nodes via InfiniBand
Between two cores of one socket
Between two sockets of one node
The throughput-parallel vector triad benchmark

Microbenchmarking for architectural exploration

- Every core runs its own, independent bandwidth benchmark

```fortran
double precision, dimension(:,), allocatable :: A,B,C,D

!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
endo
!$OMP END PARALLEL
```

- → pure hardware probing, no impact from OpenMP overhead
Bandwidth saturation vs. # cores on Sandy Bridge socket (3 GHz)

- Scalable BW in L1, L2, L3 cache
- Saturation effect in memory

Motivation

Programming models

Tools

Conclusions

Prevalent hardware bottleneck

Interlude: ccNUMA

The role of machine topology

Cost-Benefit Calculation
Scalable BW in L1, L2, L3 cache

Performance [GFlops/s]

Throughput vector triad on Sandy Bridge socket (3 GHz)

Motivation
Introduction
Programming models
Tools
Conclusions

Prevalent hardware bottleneck
Interlude: ccNUMA
The role of machine topology
Cost-Benefit Calculation
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
Will the effort for optimization pay off?
Remarks on Cost-Benefit Calculation

Costs

- for optimization effort
  - e.g., additional OpenMP parallelization
  - e.g., 3 person month \( \times 5,000 \text{ €} = 15,000 \text{ €} \) (full costs)

Benefit

- from reduced CPU utilization
  - e.g., Example 1:
    - 100,000 € **hardware costs** of the cluster
    - \( \times 20\% \) used by this application over whole lifetime of the cluster
    - \( \times 7\% \) performance win through the optimization
    - \( = 1,400 \text{ €} \) \( \rightarrow \) **total loss = 13,600 €**
  - e.g., Example 2:
    - 10 Mio € **system** \( \times 5\% \) used \( \times 8\% \) performance win
    - \( = 40,000 \text{ €} \) \( \rightarrow \) **total win = 25,000 €**

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

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>Network Configuration</th>
<th>Latency (µs)</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray XC30 (Sandybridge @ HLRS)</td>
<td>4.1</td>
<td>6.8</td>
</tr>
<tr>
<td>Xeon+Infiniband (beacon @ NICS)</td>
<td>1.6</td>
<td>5.4</td>
</tr>
<tr>
<td>Xeon+Infiniband (beacon @ NICS)</td>
<td>4.1</td>
<td>7.1</td>
</tr>
<tr>
<td>Cray XC30 (Sandybridge @ HLRS)</td>
<td>4.1</td>
<td>5.2</td>
</tr>
<tr>
<td>Xeon+Infiniband (beacon @ NICS)</td>
<td>2.1</td>
<td>5.4</td>
</tr>
<tr>
<td>Cray XC30 (Sandybridge @ HLRS)</td>
<td>4.1</td>
<td>5.2</td>
</tr>
<tr>
<td>Xeon+Infiniband (beacon @ NICS)</td>
<td>2.1</td>
<td>5.1</td>
</tr>
<tr>
<td>Cray XC30 (Sandybridge @ HLRS)</td>
<td>4.4</td>
<td>4.7</td>
</tr>
<tr>
<td>Xeon+Infiniband (beacon @ NICS)</td>
<td>2.4</td>
<td>5.0</td>
</tr>
<tr>
<td>Cray XC30 (Sandybridge @ HLRS)</td>
<td>10.2</td>
<td>4.2</td>
</tr>
<tr>
<td>Xeon+Infiniband (beacon @ NICS)</td>
<td>12.1</td>
<td>4.8</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</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(16 byte msg)</td>
<td>(bi-directional, 512 kB messages, per process)</td>
</tr>
<tr>
<td>4</td>
<td>9 µs</td>
<td>0.80 GB/s</td>
</tr>
<tr>
<td>16</td>
<td>11 µs</td>
<td>0.75 GB/s</td>
</tr>
<tr>
<td>30</td>
<td>15 µs</td>
<td>0.66 GB/s</td>
</tr>
<tr>
<td>60</td>
<td>29 µs</td>
<td>0.50 GB/s</td>
</tr>
<tr>
<td>120</td>
<td>149 µs</td>
<td>0.19 GB/s</td>
</tr>
<tr>
<td>240</td>
<td>745 µs</td>
<td>0.05 GB/s</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 & 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

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

  - 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

  - 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

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

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

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

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

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

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

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

\[ bw = 100\% \cdot bw_{\text{optimal}} \]

\[ bw = \frac{3 \left(\frac{3}{2}\right)^2}{2 \cdot 1 + 2 \cdot 1 + 1 \cdot 1} \cdot bw_{\text{opt.}} = 95\% \cdot bw_{\text{opt.}} \]

\[ bw = \frac{3 \left(\frac{3}{4}\right)^2}{4 \cdot 1 + 4 \cdot 1 + 1 \cdot 1} \cdot bw_{\text{opt.}} = 84\% \cdot bw_{\text{opt.}} \]

\[ bw = \frac{3 \left(\sqrt{3}\right)^2}{8 \cdot 1 + 8 \cdot 1 + 1 \cdot 1} \cdot bw_{\text{opt.}} = 71\% \cdot bw_{\text{opt.}} \]

\[ bw = \frac{3 \left(\frac{3}{16}\right)^2}{16 \cdot 1 + 16 \cdot 1 + 1 \cdot 1} \cdot bw_{\text{opt.}} = 58\% \cdot bw_{\text{opt.}} \]

\[ bw = \frac{3 \left(\frac{3}{32}\right)^2}{32 \cdot 1 + 32 \cdot 1 + 1 \cdot 1} \cdot bw_{\text{opt.}} = 47\% \cdot bw_{\text{opt.}} \]

\[ bw = \frac{3 \left(\frac{3}{64}\right)^2}{64 \cdot 1 + 64 \cdot 1 + 1 \cdot 1} \cdot bw_{\text{opt.}} = 37\% \cdot bw_{\text{opt.}} \]

**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
  - *node-level* hybrid MPI+OpenMP, i.e., one MPI process per SMP node
  - *mixed level* hybrid MPI+OpenMP with only 2 or 4 MPI-processes/SMP node.

- Needed for
  - *mixed level* hybrid MPI+OpenMP with > 4 MPI-processes/SMP node
  - MPI + MPI-3.0 shared memory
  - Pure MPI communication

**one-level**

- domain-decomposition is enough

**multi-level domain-decomposition**

- Optimized communication with
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
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- Example with 48 cores on:
  - 4 ccNUMA nodes
  - each node with 2 CPUs
  - each CPU with 6 cores

- 2-dim application with 6000 x 8080 gridpoints
  - Minimal communication with 2-dim domain composition with 1000 x 1010 gridpoints/core (shape as quadratic as possible → minimal circumference → minimal halo communication)
  - virtual 2-dim process grid: 6 x 8

- How to locate the MPI processes on the hardware?
  - Using sequential ranks in MPI_COMM_WORLD
  - Optimized placement
  → Exercise 4 at the end of this chapter

Order of the new ranks: Last coordinate is running contiguously → Perfect basis for MPI_Cart_create() without reorder, i.e. with reorder==0 / .FALSE.
Cartesian Grids – 2 dim –

Renumbering on a cluster of SMPs (cores / CPUs / nodes)

- dim0 = outer_d0 * mid_d0 * outer_d0 = 3 * 2 * 3 = 18
- dim1 = inner_d1 * mid_d1 * outer_d1 = 3 * 2 * 3 = 18
- 0 1 2 oc1 = outer coordinate in 0..(outer_d1-1)
- 0 1 0 1 0 1 mc1 = middle coordinate in 0..(mid_d1-1)
- 0 1 2 0 1 2 0 1 2 0 1 2 ic1 = inner coordinate in 0..(inner_d1-1)
- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 c1 = process coordinate

- Outer = several ccNUMA nodes
- Middle = CPUs of a ccNUMA node
- Inner = cores of a CPU

Number of communication links:

- Without re-numbering:
  - 150 outer
  - 72 mid
  - 180 inner

- With new ranks:
  - 60 outer
  - 90 mid
  - 252 inner

Order of the new ranks: last coordinate is running contiguously

→ Perfect basis for MPI_Cart_create() with reorder==0 / .FALSE.
Small 2-dim example —
Renumbering on a cluster of SMPs (cores / CPUs / nodes)

- Six nested loops over
  oc0, mc0, ic0, oc1, mc1, ic1

- idim = inner_d0 * inner_d1

- mdim = mid_d0 * mid_d1

- old_rank =
  ic1 + inner_d1 * ic0
  + (mc1 + mid_d1 * mc0) * idim
  + (oc1 + outer_d1 * oc0) * mdim * idim

- c0 = ic0 + inner_d0 * (mc0 + mid_d0 * oc0)
- c1 = ic1 + inner_d1 * (mc1 + mid_d1 * oc1)

new_rank = c1 + dim1 * c0

- ranks(new_rank) = old_rank
  → re-numbering: MPI_Group_incl(ranks)
  → MPI_Comm_create()

- Details in 2D & 3D → next slides

Number of communication links with sequential ranks and \( \rightarrow \) new ranks:
14 \( \rightarrow \) 8 outer (btw. nodes), 8 \( \rightarrow \) 6 mid (btw. CPUs), 16 \( \rightarrow \) 24 inner (btw. cores)
Cartesian Grids – 2 dim –
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

Product = number of cores/CPU × number of CPUs/node × number of nodes

/* Input: */
inner_d0 = …; mid_d0 = …; outer_d0 = …;
inner_d1 = …; mid_d1 = …; outer_d1 = …;

dim0 = inner_d0 × mid_d0 × outer_d0;
dim1 = inner_d1 × mid_d1 × outer_d1;
idim = inner_d0 × inner_d1;
mdim = mid_d0 × mid_d1;
odim = outer_d0 × outer_d1;
whole_size = dim0 × dim1;

ranks = malloc(whole_size*sizeof(int));
for (oc0 = 0; oc0 < outer_d0; oc0++)
for (mc0 = 0; mc0 < mid_d0; mc0++)
for (ic0 = 0; ic0 < inner_d0; ic0++)
for (oc1 = 0; oc1 < outer_d1; oc1++)
for (mc1 = 0; mc1 < mid_d1; mc1++)
for (ic1 = 0; ic1 < inner_d1; ic1++)
{
    old_rank = ic1 × inner_d1 × ic0 + (mc1 × mid_d1 × mc0) × idim
            + (oc1 × outer_d1 × oc0) × mdim;
    c0 = ic0 × inner_d0 × mc0 + inner_d0 × mid_d0 × oc0;
    c1 = ic1 × inner_d1 × mc1 + inner_d1 × mid_d1 × oc1;
    new_rank = c1 × dim1 × c0;
    ranks[new_rank] = old_rank;
}

/* Establishing new_comm with the new ranking in a array "ranks": */
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
MPI_Group_incl(world_group, world_size, ranks, &new_group);
free(ranks);
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
/* MPI_Cart_create → see next slide */

Fortran → in principle, no difference to C

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
Cartesian Grids – 3 dim –
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

Product = number of cores/CPU \times \text{number of CPUs/node} \times \text{number of nodes}

/*Input:*/
inner_d0=…; mid_d0=…; outer_d0=…;
inner_d1=…; mid_d1=…; outer_d1=…;
inner_d2=…; mid_d2=…; outer_d2=…;

dim0=inner_d0*mid_d0*outer_d0;
dim1=inner_d1*mid_d1*outer_d1; dim2=inner_d2*mid_d2*outer_d2;

idim=inner_d0*inner_d1*inner_d2; mdim=mid_d0*mid_d1*mid_d2; odim=outer_d0*outer_d1*outer_d2;
whole_size=dim0*dim1*dim2 /* or =idim*mdim*odim */;
ranks= malloc(whole_size*sizeof(int));

for (oc0=0; oc0<outer_d0; oc0++)
  for (mc0=0; mc0<mid_d0; mc0++)
    for (ic0=0; ic0<inner_d0; ic0++)
      for (oc1=0; oc1<outer_d1; oc1++)
        for (mc1=0; mc1<mid_d1; mc1++)
          for (ic1=0; ic1<inner_d1; ic1++)
            for (oc2=0; oc2<outer_d2; oc2++)
              for (mc2=0; mc2<mid_d2; mc2++)
                for (ic2=0; ic2<inner_d2; ic2++)
                  old_rank = (ic2 + inner_d2*(ic1 + inner_d1*ic0))
                    + (mc2 + mid_d2*(mc1 + mid_d1*mc0)) + idim
                    + (oc2 + outer_d2*(oc1 + outer_d1*oc0)) + mdim*mdim;
                  c0 = ic0 + inner_d0*mc0 + inner_d0*mid_d0*oc0;
                  c1 = ic1 + inner_d1*mc1 + inner_d1*mid_d1*oc1;
                  c2 = ic2 + inner_d2*mc2 + inner_d2*mid_d2*oc2;
                  new_rank = c2 + dim2*(c1 + dim1*c0);
                ranks[new_rank] = old_rank;

MPI_Comm_group(MPI_COMM_WORLD, &world_group);
MPI_Group_incl(world_group, world_size, ranks, &new_group); free(ranks);
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
dims[0] = dim0; dims[1] = dim1; dims[2] = dim2;
MPI_Cart_create(new_comm, 3, dims, periods, 0 /*=false*/, &comm_cart);

/* final output */

Useful and correct factorization that the sub-grid in each core is as cubic as possible.

in principle, no difference to C
Hierarchical Cartesian DD

Implementation with automatic SMP detection on following (skipped) slide

Virtual location of an MPI process within an SMP node

All MPI processes of an SMP node

Node coord.

coord. in SMP

Global coord.

Coordinate 0

Coord 1

Coord 2

Coord 3

y

z = Coordinate 2

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3
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_Cart_rank (comm_nodes_cart, &myrank_nodes);
    MPI_Cart_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_Cart_free (&comm_nodes_cart);

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

Copying it for the other processes in each SMP node
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\}

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
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 (large) application’s data grid, e.g., with Metis / Scotch
2. **Numa-domain-level DD**: recombining of core-domains
3. **SMP node level DD**: recombining of socket-domains
4. **Numbering** from core to socket to node (requires sequentially numbered MPI_COMM_WORLD)

**Graph of all sub-domains (core-sized)**

**Divided into sub-graphs for each socket**

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

- In this example the load-balancer **must** combine always
  - 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
      
      ```
      do i_block=1,ni,stride_i
        do j_block=1,nj,stride_j
          do k_block=1,nk,stride_k
            do i=i_block,min(i_block+stride_i-1, ni)
              do j=j_block,min(j_block+stride_j-1, nj)
                do k=k_block,min(k_block+stride_k-1, nk)
                  a(i,j,k) = f( b(i±0,1,2, j±0,1,2, k±0,1,2) )
                end do
              end do
            end do
          end do
        end do
      end do
      ...
      ```

      Access to 13-point stencil

See tutorial: Node-Level Performance Engineering and courses at LRZ and HLRS
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 \( \rightarrow \) 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
  - ... see P. Balaji, et al.: MPI on a Million Processors.

- 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: choose between several mpirun options, e.g.,
  -bind-to-core, -bind-to-socket, -bycore, -byslot …
- Cray’s aprun: pinning by default
- Platform-independent tools: likwid, numaclt

Details later
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 → high variation

With affinity, physical cores, filling left socket first:
mpirun -bind-to-core -byslot …

2x 10-core Intel Ivy Bridge, OpenMPI
Pure MPI communication: 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
- No first-touch problems as with OpenMP (in hybrid MPI+OpenMP)
Pure MPI communication: 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,
  - e.g., implementing the multi-level domain-decomposition
- No efficient use of hardware-threads (hyper-threads)
Pure MPI communication: Conclusions

- Still a good programming model for small and medium size applications.
- Major problem may be memory consumption
Programming models
- MPI + MPI-3.0
shared memory
Advantages

– Simple method for reduced memory needs for replicated data
– 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 communication
– MPI-3.0/3.1 shared memory synchronization waits for some clarification → MPI-4.0
MPI-3 shared memory

- Split main communicator into shared memory islands
  - `MPI_Comm_split_type`
- Define a shared memory window on each island
  - `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 `MPI_PUT/GET`!
  - Normal MPI one-sided synchronization, e.g., `MPI_WIN_FENCE`
Splitting the communicator & contiguous shared memory allocation

Contiguous shared memory window within each SMP node

local_window_count doubles

base_ptr

MPI process

Sub-communicator comm_sm for one SMP node

my_rank_all

my_rank_sm

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

…

my_rank_sm

my_rank_sm

my_rank_sm

my_rank_sm

…

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

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

Establish a communicator comm_sm with ranks my_rank_sm on each SMP node

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:
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.
Non-contiguous shared memory allocation

Non-contiguous shared memory window within each 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)`

```c
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

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

---

**Hybrid MPI+MPI**

- MPI for inter-node communication
- MPI-3.0 shared memory programming
Pure MPI versus MPI+MPI-3.0 shared memory

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

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 communication</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>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>3.0 µs, 4.6 GB/s</td>
<td>MPI-3.0 store</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>3.3 µs, 4.4 GB/s</td>
<td>Irecv+send</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>3.5 µs, 4.4 GB/s</td>
<td>MPI-3.0 store</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>5.2 µs, 4.3 GB/s</td>
<td>Irecv+send</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>5.2 µs, 4.4 GB/s</td>
<td>MPI-3.0 store</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>10.3 µs, 4.5 GB/s</td>
<td>Irecv+send</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
<tr>
<td>10.1 µs, 4.5 GB/s</td>
<td>MPI-3.0 store</td>
<td>MPI+MPI-3.0 shared memory</td>
</tr>
</tbody>
</table>

Conclusion: No performance-win through MPI-3.0 shared memory programming
### 1 MPI process versus several MPI processes

(1 Intel Xeon Phi per node)

<table>
<thead>
<tr>
<th>1 MPI process per Intel Xeon Phi</th>
<th>4 MPI processes per Intel Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi + Infiniband</td>
<td>Latency</td>
</tr>
<tr>
<td>beacon @ NICS</td>
<td>Latency</td>
</tr>
<tr>
<td>15 µs, 0.83 GB/s</td>
<td>Internode: Irecv + Send</td>
</tr>
<tr>
<td>26 µs, 0.87 GB/s</td>
<td>1x</td>
</tr>
<tr>
<td>25 µs, 0.91 GB/s</td>
<td>2x</td>
</tr>
<tr>
<td>23 µs, 0.91 GB/s</td>
<td>4x</td>
</tr>
<tr>
<td>24 µs, 0.92 GB/s</td>
<td>8x</td>
</tr>
<tr>
<td>21 µs, 0.91 GB/s</td>
<td>16x</td>
</tr>
<tr>
<td>51 µs, 0.90 GB/s</td>
<td>30x</td>
</tr>
<tr>
<td></td>
<td>60x</td>
</tr>
<tr>
<td></td>
<td>19 µs, 0.54 GB/s</td>
</tr>
<tr>
<td></td>
<td>25 µs, 0.52 GB/s</td>
</tr>
<tr>
<td></td>
<td>26 µs, 0.52 GB/s</td>
</tr>
<tr>
<td></td>
<td>23 µs, 0.91 GB/s</td>
</tr>
<tr>
<td></td>
<td>24 µs, 0.92 GB/s</td>
</tr>
<tr>
<td></td>
<td>21 µs, 0.91 GB/s</td>
</tr>
<tr>
<td></td>
<td>51 µs, 0.90 GB/s</td>
</tr>
</tbody>
</table>

**Similar Conclusion:**
- Several MPI processes inside Phi (in a line) cause slower communication
- No performance-win through MPI-3.0 shared memory programming
Programming opportunities with MPI shared memory:
1) Reducing memory space for replicated data

- Replicated data in each MPI process

Cluster of SMP nodes without using MPI shared memory methods

- Replicated data only once within each SMP node

Direct loads & stores, no library calls

Using MPI shared memory methods

MPI-3.0 shared memory can be used to significantly reduce the memory needs for replicated data.

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
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 communica. 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
Example 1 — Rotating information around a ring

- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable `snd_buf`.
- Each process passes this on to its neighbor on the right.
- Each processor calculates the sum of all values.
- Repeat "2-5" with "size" iterations (size = number of processes), i.e.
  - each process calculates sum of all ranks.
- Use nonblocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock.
Example 1 — Rotating information around a ring

**Initialization:**
- Each iteration:
  1. my_rank
  2. snd_buf
  3. rcv_buf
  4. sum

**For each iteration:**
- my_rank
- snd_buf
- rcv_buf
- sum

**Fortran:**
- dest = mod(my_rank+1, size)
- source = mod(my_rank-1+size, size)

**C/C++:**
- dest = (my_rank+1) % size
- source = (my_rank-1+size) % size

**Single Program!!!**

---

**Fortran:** Do not forget MPI-3.0 → `<TYPE>, ASYNCHRONOUS :: ..._buf` and `IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING...) CALL MPI_F_SYNC_REG(..._buf)`
Example 1: Nonblocking halo-copy in a ring

```c
int snd_buf, rcv_buf, sum;
int right, left;
int sum, i, my_rank, size;
MPI_Status status;
MPI_Request request;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
right = (my_rank+1) % size;
left  = (my_rank-1+size) % size;
sum = 0;
snd_buf = my_rank;
for( i = 0; i < size; i++) {
    MPI_Issend(&snd_buf, 1, MPI_INT, right, 17, MPI_COMM_WORLD, &request);
    MPI_Recv ( &rcv_buf, 1, MPI_INT, left, 17, MPI_COMM_WORLD, &status);
    MPI_Wait(&request, &status);
    snd_buf = rcv_buf;
    sum += rcv_buf;
}
printf("PE%i: \tSum = %i\n", my_rank, sum);
MPI_Finalize();
```

Synchronous send (Issend) instead of standard send (Isend) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem. A real application would use standard Isend().

See HLRS online courses

http://www.hlrs.de/training/par-prog-ws/

→ Practical → MPI.tar.gz
Example 1: Nonblocking halo-copy in a ring

```
INTEGER, ASYNCHRONOUS :: snd_buf
INTEGER :: rcv_buf, sum, i, my_rank, size
TYPE(MPI_Status) :: status
TYPE(MPI_Request) :: request
INTEGER(KIND=MPI_ADDRESS_KIND):: iadummy

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
right = mod(my_rank+1, size)
left = mod(my_rank-1+size, size)
sum = 0
snd_buf = my_rank
DO i = 1, size
   CALL MPI_Issend(snd_buf,1,MPI_INTEGER,right,17,MPI_COMM_WORLD, request)
   CALL MPI_Recv (rcv_buf,1,MPI_INTEGER,left, 17,MPI_COMM_WORLD, status)
   CALL MPI_Wait(request, status)
   ! CALL MPI_GET_ADDRESS(snd_buf, iadummy)
   ! ... should be substituted as soon as possible by:
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
   snd_buf = rcv_buf
   sum = sum + rcv_buf
END DO
WRITE(*,*) 'PE', my_rank, ': Sum =', sum
CALL MPI_Finalize()
```

Synchronous send (Issend) instead of standard send (Isend) is used only to demonstrate the use of the nonblocking routine resolves the deadlock (or serialization) problem. A real appl. would use Isend.

See HLRS online courses
http://www.hlrs.de/training/par-prog-ws/ → Practical → MPI.tar.gz
Example 2: Ring with fence and one-sided comm.

- **Tasks:**
  - Substitute the nonblocking communication by one-sided communication. Two choices:

  - **either** `rcv_buf` = window
    - `MPI_Win_fence` - the `rcv_buf` can be used to receive data
    - `MPI_Put` - to write the content of the local variable `snd_buf` into the remote window (`rcv_buf`)
    - `MPI_Win_fence` - the one-sided communication is finished, `rcv_buf` is filled

  - **or** `snd_buf` = window
    - `MPI_Win_fence` - the `snd_buf` is filled
    - `MPI_Get` - to read the content of the remote window (`snd_buf`) into the local variable `rcv_buf`
    - `MPI_Win_fence` - the one-sided communication is finished, `rcv_buf` is filled
Example 2: Ring with fence and one-sided comm.

C

```c
MPI_Win win;

/* Create the window once before the loop: */
MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL,
                MPI_COMM_WORLD, &win);

/* Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait: */
MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPRECEDE, win);
MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT | MPI_MODE_NOSUCCEED, win);

INTEGER, ASYNCHRONOUS :: rcv_buf
TYPE(MPI_Win) :: win ; INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, buf_size, target_disp

target_disp = 0 ! This "long" integer zero is needed in the call to MPI_PUT
! Create the window once before the loop:
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, integer_size)
buf_size = 1 * integer_size; disp_unit = integer_size
CALL MPI_WIN_CREATE(rcv_buf, buf_size, disp_unit, MPI_INFO_NULL, &
                    MPI_COMM_WORLD, win)

/* Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait: */
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE,MPI_MODE_NOPRECEDE), win)
CALL MPI_PUT(snd_buf,1,MPI_INTEGER,right,target_disp,1,MPI_INTEGER, win)
CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOSUCCEED),win)
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
```

Fortran

```fortran
INTEGER, ASYNCHRONOUS :: rcv_buf
TYPE(MPI_Win) :: win ; INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, buf_size, target_disp

target_disp = 0 ! This "long" integer zero is needed in the call to MPI_PUT
! Create the window once before the loop:
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, integer_size)
buf_size = 1 * integer_size; disp_unit = integer_size
CALL MPI_WIN_CREATE(rcv_buf, buf_size, disp_unit, MPI_INFO_NULL, &
                    MPI_COMM_WORLD, win)

/* Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait: */
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE,MPI_MODE_NOPRECEDE), win)
CALL MPI_PUT(snd_buf,1,MPI_INTEGER,right,target_disp,1,MPI_INTEGER, win)
CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOSUCCEED),win)
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
```
Example 3: Shared memory ring communication

- 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 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)
Example 3: 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);
```

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
Example 3: 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
Example 3: Ring with shared memory one-sided comm.

```fortran
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
CALL MPI_Win_fence(0, win) ! Workaround: no assertions
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
IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
CALL MPI_WIN_FENCE(0, win) ! Workaround: no assertions
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
```
Summary of halo files (and some ring files)

- ring.c
- ring_1sided_get.c & ring_1sided_put.c → ring_1sided_exa2.c
- ring_allreduce.c in Chap. 6 Collective Communication
- derived_20.c in Chap. 12 Derived Datatypes
- topology_ring.c in Chap. 9 Virtual Topologies
- halo_isend_recv.c
- halo_irecv_send.c
- halo_neighbor_alltoall.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_query.c
- halo_1sided_store_win_alloc_shared_pscw.c
- halo_1sided_store_win_alloc_shared_othersync.c
- halo_1sided_store_win_alloc_shared_signal.c
- halo_1sided_put_win_alloc.c
- ring_1sided_store_win_alloc_shared.c
- ring_neighbor_alltoall.c
dito.
- ring_neighbor_alltoallw.c
dito.
- Example 1
- Example 2
- Example 3
- Example 4
- Example 5
- See slide after Example 5

see also login-slides
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)

Process
Rank 1

MPI_Recv(from rank 0)

printf … X

Or any other process-to-process synchronization, e.g., using also shared memory stores and loads

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 😱 😱 😱
“eventually synchronized“ – the Solution

• A pair of local memory fences is needed:

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

---

Process
Rank 0

X = 1

MPI_Send(empty msg to rank 1)

local memory fence

Process
Rank 1

MPI_Recv(from rank 0)

local memory fence

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 ...
  - Disadvantage: may be slower than C11 atomic_thread_fence with appro. order
- 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

5 sync methods, see next slide

X = 1
local memory fence
MPI_Send( empty msg to rank 1 ) → MPI_Recv( from rank 0 )
local memory fence
printf ... X

X is a variable in a shared window initialized with 0.
**General MPI-3 shared memory synchronization rules**

(based on MPI-3.1, MPI_Win_allocate_shared, page 408, lines 43-47: "A consistent view ...")

<table>
<thead>
<tr>
<th>Defining</th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sync-from</td>
<td>Sync-to</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>being</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_post(^1)</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>MPI_Win_complete(^1)</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>MPI_Win_fence(^1)</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>MPI_Win_sync</td>
</tr>
<tr>
<td>or(^3)</td>
</tr>
<tr>
<td>MPI_Win_unlock(^1)</td>
</tr>
</tbody>
</table>

and the lock on process 0 is granted first

and having ...

then it is **guaranteed** that ...

\[ A = \text{val}_1 \]

Sync-from → Sync-to → load(A) → \[ A = \text{val}_1 \] (this is the write-read-rule)

\[ \text{load}(B) \]

Sync-from → Sync-to → B = \text{val}_2 → \[ \text{load}(B) \] in P0 is not affected by the store of \text{val}_2 in P1 (read-write-rule)

\[ C = \text{val}_3 \]

Sync-from → Sync-to → C = \text{val}_4 → \[ \text{load}(C) \] in P1 loads \text{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\(\rightarrow\)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.)
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

1. MPI_WIN_LOCK_ALL(
   MPI_MODE_NOCHECK, win)
2. DO ...
3. X = ...
4. MPI_F_SYNC_REG(X)
5. MPI_WIN_SYNC(win)
6. MPI_Send

Process B

1. MPI_WIN_LOCK_ALL(
   MPI_MODE_NOCHECK, win)
2. DO ...
3. print X
4. MPI_F_SYNC_REG(X)
5. MPI_SEND
6. MPI_RECV
7. MPI_WIN_SYNC(win)

- 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.

Also needed due to read-write-rule:

- Data exchange in this direction, therefore MPI_WIN_SYNC is needed in both processes: Write-read-rule.
Example 4: 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
Example 4: Ring – Using *other* synchronization

- Communication pattern between each pair of neighbor processes

**Process rank n**

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

**Process rank n+1**

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

**Write-read-rule**

- Now, data in `rcv_buf` can be locally **read**

For Fortran: Are these the locations where `MPI_F_SYNC_REG` is needed?

See also login-slides.
Example 4: Ring with shared memory and MPI_Win_sync

```
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);
```
Example 5: 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)
Example 5: Ring – with memory signals

Process rank n

- signal_A
- while (A==0) IDLE
- A = 0
- win_sync(A)
- win_sync(B)

→ B is now locally 0

Process rank n+1

- old data in rcv_buf is read
- win_sync
- 1

Atomic (or volatile) load

Atomic (or volatile) store

- signal_B
- while (B==0) IDLE
- B = 0
- win_sync(B)
- win_sync(A)

→ A is now locally 0

- data is stored in the rcv_buf and can be read after subsequent synchronization
- win_sync
- write snd_buf data to the rcv_buf in right neighbor

Atomic (or volatile) load

see also login-slides
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:
    - different communication methods
    - different memory allocation methods
  - And run and compare on a shared memory only:
    - different communication methods

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

Example 1
- `halo_recv_send.c`
- `halo_isend_recv.c`

Example 2
- `halo_neighbor_alltoall.c`
- `halo_1sided_put.c`
- `halo_1sided_put_alloc_mem.c`
- `halo_1sided_put_win_alloc.c`

Example 3
- `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`
MPI Communication inside of the SMP nodes

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

Low latency point-to-point synchronization → next slide

High latency MPI_Win_fence

Low bandwidth with MPI_Put

Medium bandwidth point-to-point and neighbor alltoall

On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1
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 (16 byte msg)</th>
<th>Bandwidth (bi-directional, 512 kB messages, per process)</th>
<th>Shared mem. bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9 µs</td>
<td>0.80 GB/s</td>
<td>0.25 GB/s</td>
</tr>
<tr>
<td>16</td>
<td>11 µs</td>
<td>0.75 GB/s</td>
<td>0.24 GB/s</td>
</tr>
<tr>
<td>30</td>
<td>15 µs</td>
<td>0.66 GB/s</td>
<td>0.24 GB/s</td>
</tr>
<tr>
<td>60</td>
<td>29 µs</td>
<td>0.50 GB/s</td>
<td>0.22 GB/s</td>
</tr>
<tr>
<td>120</td>
<td>149 µs</td>
<td>0.19 GB/s</td>
<td>0.20 GB/s</td>
</tr>
<tr>
<td>240</td>
<td>745 µs</td>
<td>0.05 GB/s</td>
<td></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.

Conclusion: Slow

MPI pt-to-pt substituted by MPI-3.0 shared memory store
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:
  - The meaning of the assertions for shared memory is still undefined
- Similar problems as with all library based shared memory (e.g., pthreads)
- Does not reduce the number of MPI processes
MPI+MPI-3.0 shared mem: Conclusions

- Add-on feature for pure MPI communication
- Opportunity for reducing communication within SMP nodes
- Opportunity for reducing memory consumption (halos & replicated data)
Programming models
- MPI + OpenMP

- General considerations  slide 113
- How to compile, link, and run  120
- Case-study: The Multi-Zone NAS Parallel Benchmarks  125
- Memory placement on ccNUMA systems  134
- Topology and affinity on multicore  140
- Overlapping communication and computation  157
- Main advantages, disadvantages, conclusions  171
- Example  174
Hybrid MPI+OpenMP Masteronly Style

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

```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
```

**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 * thread_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!

```
!$OMP BARRIER  #pragma omp barrier
!$OMP MASTER   #pragma omp master
  call MPI_Xxx(...)  MPI_Xxx(...);
!$OMP END MASTER
!$OMP BARRIER  #pragma omp barrier
```

- 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
!$OMP PARALLEL
!$OMP DO
do i=1,1000
   a(i) = buf(i)
end do
!$OMP END DO NOWAIT
!$OMP BARRIER
!$OMP MASTER
call MPI_RECV(buf,...)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO
do i=1,1000
   c(i) = buf(i)
end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```c
#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)**

**MPI+OpenMP**
- Cray XC30
- Sandybridge @ HLRS

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>3.4 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

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

Masteronly
MPI only outside of parallel regions

Hybrid Parallel Programming
Rabenseifner, Hager, Jost

Motivation
Introduction
Programming models
Tools
Conclusions

Hybrid
MPI+OpenMP

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
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 an appropriate MPI launching mechanism (often multiple options available)
  - 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 "-npernode #" 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

Courtesey of Rainer Keller, HLRS and ORNL
Programming models
- MPI + OpenMP

Case study:
The Multi-Zone NAS Parallel Benchmarks

The low-hanging fruits: load balancing and memory consumption
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](http://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
...
subroutine zsolve(u, rsd, ...)
...
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP& PRIVATE(m, i, j, k...)
do k = 2, nz-1
!$OMP DO
    do j = 2, ny-1
        do i = 2, nx-1
            do m = 1, 5
                u(m, i, j, k) =
                dt*rsd(m, i, j, k-1)
            end do
        end do
    end do
end do
!$OMP END DO NOWAIT
end do
!$OMP END PARALLEL
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!
Dell Linux Cluster Lonestar Topology

CPU type: Intel Core Westmere processor

************************************

Hardware Thread Topology

************************************

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

Careful!
Numbering scheme of cores is system dependent

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

900 Gflops

Only local memory access
With thread pinning:
Needed if OpenMP spans multi-sockets.

→ indeed, BT-MZ profits from hybrid
Using more OpenMP threads reduces the memory usage substantially, up to five times on Hopper Cray XT5 (eight-core nodes).

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
    • \( \rightarrow \) 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

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

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

Motivation

Introduction

Programming models

Tools

Conclusions

Pure MPI communication

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):

  ```
  surface=18N^2
  surface=16N^2
  ```

1) Provided that the application has a 20%
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

```fortran
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
endo
!$OMP END DO
do j=1,NITER
!$OMP DO
do i=1,N
   A(i) = B(i) + C(i) * D(i)
endo
!$OMP END DO
if(.something.that.is.never.true.) then
   call dummy(A,B,C,D)
endif
endo
!$OMP END PARALLEL
```

... and then report performance vs. loop size for different #cores!
OpenMP vector triad on Intel Sandy Bridge node (2 sockets, 3 GHz)

Cost of sync amortized for large problems

sync overhead grows with # of threads → next slide for direct measurements!
Thread synchronization overhead on IvyBridge-EP

Direct measurement of barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>599</td>
<td>425</td>
</tr>
<tr>
<td>SMT threads</td>
<td>612</td>
<td>423</td>
</tr>
<tr>
<td>Other socket</td>
<td>1486</td>
<td>1067</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full domain</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket (10 cores)</td>
<td>1934</td>
<td>1301</td>
</tr>
<tr>
<td>Node (20 cores)</td>
<td>4999</td>
<td>7783</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>5981</td>
<td>9897</td>
</tr>
</tbody>
</table>

Strong topology dependence!

Overhead grows with thread count

Strong dependence on compiler, CPU and system env.!
## Thread synchronization overhead on SandyBridge-EP

*Direct measurement of barrier overhead in CPU cycles*

<table>
<thead>
<tr>
<th></th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Threads</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>

Strong topology dependence!

<table>
<thead>
<tr>
<th></th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full domain</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket</td>
<td>1497</td>
<td>14546</td>
<td>14418</td>
</tr>
<tr>
<td>Node</td>
<td>3401</td>
<td>34667</td>
<td>29788</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>6881</td>
<td>59038</td>
<td>58898</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 (20-core) Ivy Bridge node:

- 3 x cores (20 vs 60) on Phi
- 2 x more operations per cycle on Phi

→ 6 x more work done on Xeon Phi per cycle

- 3 x higher barrier penalty (cycles) on Phi

→ One barrier causes 3 x 6 = **18x more pain 😊**.
Thread/Process Affinity ("Pinning")

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`, PLPA \( \rightarrow \) 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

Pinning (physical cores first, first socket first)

No pinning
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
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 ...]
```
**OMP_PLACES and Thread Affinity**

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**
  - 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\}, ... \{28,29,30,31\}"`
  - `setenv OMP_PLACES "\{0:4\},\{4:4\},\{8:4\}, ... \{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)

- `setenv OMP_PLACES "\{0\},\{1\},\{2\}, \ldots \{29\},\{30\},\{31\}"` or
- `setenv OMP_PLACES threads` (example with P=32 places)
  - `setenv OMP_NUM_THREADS "8,2,2"`
  - `setenv 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: \textbf{4 sockets} x \textbf{6 cores} x \textbf{2 hyper-threads} = \textbf{48 processors}

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

\begin{verbatim}
0  4  8 12 16 20
24 28 32 36 40 44
1  5  9 13 17 21
25 29 33 37 41 45
2  6 10 14 18 22
26 30 34 38 42 46
3  7 11 15 19 23
27 31 35 39 43 47
\end{verbatim}

\textbf{setenv OMP\_PLACES threads} (= \{0,\{24\},\{4\},\{28\},\{8\},\{32\},\{12\},\{36\},\{16\},\{40\},\{20\},\{44\},\{1\},\{25\}, ..., \{23\},\{47\} )

\rightarrow OpenMP threads/tasks are \textbf{pinned} to hardware hyper-threads

\textbf{setenv OMP\_PLACES cores} (= \{0,\{24\}, \{4,28\}, \{8,32\}, \{12,36\}, \{16,40\}, \{20,44\}, \{1,25\}, ..., \{23,47\} )

\rightarrow OpenMP threads/tasks are \textbf{pinned} to hardware cores

and can migrate between hyper-threads of the core

\textbf{setenv OMP\_PLACES sockets} (= \{0, 24, 4, 28, 8, 32, 12, 36, 16, 40, 20, 44\}, \{1,25,...\}, {...}, \{...,23,47\} )

\rightarrow OpenMP threads/tasks are \textbf{pinned} to hardware sockets

and can migrate between cores & hyper-threads of the socket

\textbf{Examples should be independent of vendor’s numbering!}

- Without nested parallel regions:
  \textbf{#pragma omp parallel num_threads(4*6) proc_bind(spread)} \rightarrow \text{one thread per core}

- With nested regions:
  \textbf{#pragma omp parallel num_threads(4) proc_bind(spread)} \rightarrow \text{one thread per socket}
  \textbf{#pragma omp parallel num_threads(6) proc_bind(spread)} \rightarrow \text{one thread per core}
  \textbf{#pragma omp parallel num_threads(2) proc_bind(close)} \rightarrow \text{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
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

LIKWID: Topology/Affinity
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 \
-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

- **hybrid MPI+OpenMP**
  - MPI: inter-node communication
  - OpenMP: inside of each SMP node

- **Hybrid MPI+MPI**
  - MPI for inter-node communication
  - MPI-3.0 shared memory programming

- **OpenMP only**
  - Distributed virtual shared memory

---

- **No overlap of Comm. + Comp.**
  - MPI only outside of parallel regions of the numerical application code

- **Overlapping Comm. + Comp.**
  - MPI communication by one or a few threads while other threads are computing

- **Within shared memory nodes:**
  - Halo updates through direct data copy
  - No halo updates, direct access to neighbor data

- **Master only**
  - MPI only outside of parallel regions

- **Funneled**
  - MPI only on master-thread

- **Funneled & Reserved**
  - Reserved thread for communication

- **Funneled with Full Load Balancing**
  - More than one thread may communicate

---

**Programming models**
- Pure MPI communication
- MPI+MPI-3.0 shared memory
- MPI+OpenMP
- MPI+Accelerators

**Conclusions**
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)
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

- **the load balancing problem**

```c
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.

```
#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 \(
    \rightarrow
\) 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


Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
Overlapping: Using OpenMP tasks

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


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

- Data distribution across processes:
Dense matrix-vector multiply with communication overlap via tasking

- Computation/communication scheme:

  Step 1: MVM on diagonal blocks

  Ring shift of vector \( r \)

  Step 2: MVM on next subdiagonal blocks
Hybrid Parallel Programming

Motivation
Introduction
Programming models
Tools
Conclusions

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

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);
        }
      }
      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.
Overlapping can be achieved with OpenMP tasks (1st part)

```plaintext
integer stride=1000
!$omp parallel
!$omp master
!$omp task
do m=1,x-stride , stride
   !$omp task
   do mm=0,stride-1,1
      sendright (m+mm)= p_array ( f(m+mm))
   enddo
!$omp end task
!$omp end task
endo
endo
!$omp task
do m=x,stride , stride
   sendright (m)= p_array ( f(m))
!$omp end task
endo

!pack particle to move left
do n=1,y-stride , stride
!$omp task
do nn=0,stride-1,1
   sendleft(n+nn)= p_array(f(n+nn))
enddo
!$omp end task
endo
endo

!$omp task
do n=n,y
   sendleft(n)= p_array(f(n))
enddo
!$omp end task
endo
endo
!$omp master
!$omp end parallel
if (sum_shift_p==0) { return; }
```

- **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.

---

**Motivation**

**Introduction**

**Programming models**

**Tools**

**Conclusions**

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

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

```
!$omp parallel
!$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.

```
!$omp parallel
!$omp master
! adding shifted particles from right
  do m=1,x_stride,stride
    !$omp task
    do mm=0, stride - 1, 1
      p_array(h(m))=sendright(m);
    enddo
    !$omp end task
  enddo
!$omp end task

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

!$omp task
  MPI_SENDRECV(sendleft, length=g(y), ...);
!$omp end master
!$omp end parallel
! adding shifted particles from left
!$omp parallel do
  do n=1,y
    p_array(h(n))=sendleft(n);
  enddo
!$omp end parallel
```

Overlapping remaining MPI_Sendrecv

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
Example: MPI+OpenMP-Hybrid Jacobi solver

- Source code: 2016-HY-G-GeorgHager-Jacobi-w-MPI+OpenMP.tgz
- 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

Task: 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
Example cont’d

• Tasks (we assume $N_c$ cores per CPU socket):
  – Run the MPI-only code on one node with
    $1,\ldots,N_c,\ldots,2N_c$ 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 $1,\ldots,N_c,\ldots,2N_c$ 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

Parts
Courtesy of Gabriele Jost
Accelerator programming: Bottlenecks reloaded

Example: 2-socket Intel “Broadwell” (2x18 cores) node with two Nvidia P100 GPGPUs (PCIe 2.0)

<table>
<thead>
<tr>
<th></th>
<th>per GPGPU</th>
<th>per CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP peak performance</td>
<td>4.6 Tflop/s</td>
<td>660 Gflop/s</td>
</tr>
<tr>
<td>Machine balance</td>
<td>0.11 B/F</td>
<td>0.10 B/F</td>
</tr>
<tr>
<td>eff. memory (HBM) bandwidth</td>
<td>510 Gbyte/s</td>
<td>63 Gbyte/s</td>
</tr>
<tr>
<td>inter-device bandwidth</td>
<td>≈ 20 Gbyte/s</td>
<td></td>
</tr>
</tbody>
</table>

→ Speedups can only be attained if communication overheads are under control
   → Basic estimates help
Options for hybrid accelerator programming

<table>
<thead>
<tr>
<th>multicore host</th>
<th>accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>CUDA</td>
</tr>
<tr>
<td>MPI+MPI3 shmem ext.</td>
<td>OpenCL</td>
</tr>
<tr>
<td>MPI+threading (OpenMP, pthreads, TBB,…)</td>
<td>OpenACC</td>
</tr>
<tr>
<td>threading only</td>
<td>OpenMP 4.0</td>
</tr>
<tr>
<td>PGAS (CAF, UPC,…)</td>
<td>special purpose</td>
</tr>
</tbody>
</table>

- Which model/combination is the best???
  → the one that allows you to address the relevant hardware bottleneck(s)
Directive-based accelerator programming

• Pros
  – Simple to use, similar to OpenMP
  – Incremental parallelism
  – No change in language/paradigm
  – Easier to write code that works also for host-only

• Cons
  – Abstractions hide performance-relevant details
  – Blind flying without good compiler diagnostics
  – Programming style fosters too much trust in compiler
  – Traditional languages geared towards standard architectures

• Solution?
  – There is no solution other than knowing what to expect
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:*
http://openmp.org/wp/openmp-specifications/

- The **“target data”** construct:
  - When a target data construct is encountered, a new device data environment is created, and the encountering task executes the target data region
  
  `pragma omp target data [device, map, if]`

- The **“target”** construct:
  - Creates device data environment and specifies that the region is executed by a device. The encountering task waits for the device to complete the target region at the end of the construct
  
  `pragma omp target [device, map, if]`

- The **“teams”** construct:
  - Creates a league of thread teams. The master thread of each team executes the teams region
  
  `pragma omp teams [num_teams, num_threads, ...]`

- The **“distribute”** construct:
  - Specifies that the iterations of one or more loops will be executed by the thread teams. The iterations of the loop are distributed across the master threads of all teams
  
  `pragma omp distribute [collapse, dist_schedule, ...]`
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
    }
}
```
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 + 1 rhs(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: \textit{acc parallel loop, acc data}
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];
    }
}
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);
  }
}
```
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 );
}
**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](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`

---

```
!$acc data present (GRID)

! Back boundary
IF (NEIGHBORS(BACK) /= -1) THEN
  TIME_START_DIR = MG_TIMER()
ENDIF

!$acc data present (SEND_BUFFER_BACK)
!$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

---

```
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

---

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

**Pure MPI communication**
- MPI+MPI-3.0 shared memory
- MPI+OpenMP
- MPI+Accelerators
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
```

Motivation
Introduction
Programming models
Tools
Conclusions

Pure MPI communication
MPI+MPI-3.0 shared memory
MPI+OpenMP
MPI+Accelerators
Cray XK7 Hermit

- Located at HLRS Stuttgart, Germany (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:

- Located at HLRS Stuttgart, Germany (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
Scalability of miniGhost on Cray XK7

<table>
<thead>
<tr>
<th></th>
<th>Total Time (sec)</th>
<th>Comm. Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP (16x1t)</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
...
```

```
/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
```
Profilng 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></td>
<td>Write -&gt; Error</td>
<td>Error</td>
<td></td>
<td>Memory write of global_variable at</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;pthread&quot;</td>
<td></td>
<td></td>
<td>&quot;pthread&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write data</td>
<td></td>
<td></td>
<td>&quot;pthread_race.c&quot;:31 conflicts with</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ta-race</td>
<td></td>
<td></td>
<td>a prior memory write of</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;e.c&quot;:2</td>
<td></td>
<td></td>
<td>&quot;c&quot;:31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;global_variable at</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;pthread_race.c&quot;:31 (output</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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:
  
  \texttt{tcheck\_cl --format HTML --report pthread\_race.html pthread\_race}

![Thread Checker Output](image)

- Motivation
- Introduction
- Programming models
- Tools
- Conclusions

- Topology & Affinity
- Debug, profiling

Courtesy of Rainer Keller, HLRS, ORNL and FhT
• If one wants to compile with threaded Open MPI (option for IB):

```
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:

```
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) `ompitrace` 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

Screenshots, courtesy of KOJAK JSC, FZ Jülich
Conclusions
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

In principle, none of the programming models perfectly fits to clusters of SMP nodes

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
Acknowledgements

• 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 communication → 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

- Examples
  - MPI+MPI-3.0 shared memory → see Examples 1-5 in the section
  - MPI+OpenMP-Hybrid Jacobi solver

See also [http://www.hlrs.de/training/par-prog-ws/ → Practical](http://www.hlrs.de/training/par-prog-ws/) → *MPI.tar.gz* and [2017-HY-G-GeorgHager-Jacobi-w-MPI+OpenMP.tgz](http://www.hlrs.de/training/par-prog-ws/)

- Abstract
- Authors
- References (with direct relation to the content of this tutorial)
- Further references
Abstract

Full-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.  https://www.lrz.de/services/compute/courses/2017-01-12_hhyp1w16/
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 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

• R.v.d. Wijngaart and H. Jin,
  NAS Parallel Benchmarks, Multi-Zone Versions,
  NAS Technical Report NAS-03-010, 2003

• H. Jin and R. v.d.Wijngaart,
  Performance Characteristics of the multi-zone NAS Parallel Benchmarks,
  Proceedings IPDPS 2004

• G. Jost, H. Jin, D. an Mey and F. Hatay,
  Comparing OpenMP, MPI, and Hybrid Programming,
  Proc. Of the 5th European Workshop on OpenMP, 2003

• E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost,
  Employing Nested OpenMP for the Parallelization of Multi-Zone CFD Applications,
  Proc. Of IPDPS 2004
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

• Rolf Rabenseifner,
  Communication and Optimization Aspects on Hybrid Architectures.
  In Recent Advances in Parallel Virtual Machine and Message Passing Interface, J.
  Dongarra and D. Kranzlmüller (Eds.), Proceedings of the 9th European PVM/MPI
  Users' Group Meeting, EuroPVM/MPI 2002, Sep. 29 - Oct. 2, Linz, Austria, LNCS,

• Rolf Rabenseifner and Gerhard Wellein,
  Communication and Optimization Aspects of Parallel Programming Models on
  Hybrid Architectures.
  In proceedings of the Fourth European Workshop on OpenMP (EWOMP 2002),
  Roma, Italy, Sep. 18-20th, 2002.

• Rolf Rabenseifner,
  Communication Bandwidth of Parallel Programming Models on Hybrid
  Architectures.
  Proceedings of WOMPMEI 2002, International Workshop on OpenMP: Experiences
  and Implementations, part of ISHPC-IV, International Symposium on High
  Performance Computing, May, 15-17., 2002, Kansai Science City, Japan, LNCS 2327,
  pp 401-412.
References

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.**

• M. Kreutzer, G. Hager, G. Wellein, A. Pieper, A. Alvermann, and H. Fehske: **Performance Engineering of the Kernel Polynomial Method on Large-Scale CPU-GPU Systems.**
  Proc. IPDPS15, May 25-29, 2015, Hyderabad, India. DOI: 10.1109/IPDPS.2015.76
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


Further references


Further references


Further references


Further references
