

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

Rolf Rabenseifner¹⁾

Rabenseifner@hlrs.de

Georg Hager²⁾

Georg.Hager@rrze.uni-erlangen.de

¹⁾ High Performance Computing Center (HLRS), University of Stuttgart, Germany

²⁾ Erlangen Regional Computing Center (RRZE), University of Erlangen, Germany

Tutorial tut146 at SC14,
Nov. 16, 2014, New Orleans, LA, USA



Höchstleistungsrechenzentrum Stuttgart



General Outline (with slide numbers)

- Motivation (3)
 - Cluster hardware today: Multicore, multi-socket, accelerators (4)
 - Options for running code (7)
- Introduction (8)
 - Prevalent hardware bottlenecks (9)
 - Interlude: ccNUMA (13)
 - The role of machine topology (20)
 - Cost-Benefit Calculation (30)
- Programming models (32)
 - Pure MPI (33)
 - MPI + MPI-3.0 shared memory (60)
 - MPI + OpenMP on multi/many-core (82)
 - MPI + Accelerators (138)
- Tools (154)
 - Topology & Affinity (155)
 - Performance (156)
- Conclusions (163)

Major opportunities and challenges of “MPI+X”

 - MPI+OpenMP (164)
 - MPI+MPI-3.0 (167)
 - Pure MPI (168)
 - Acknowledgements (169)
 - Conclusions (170)
- Appendix (171)
 - Abstract (172)
 - Authors (173)
 - References (175)



Motivation



Hardware and Programming Models

Hardware

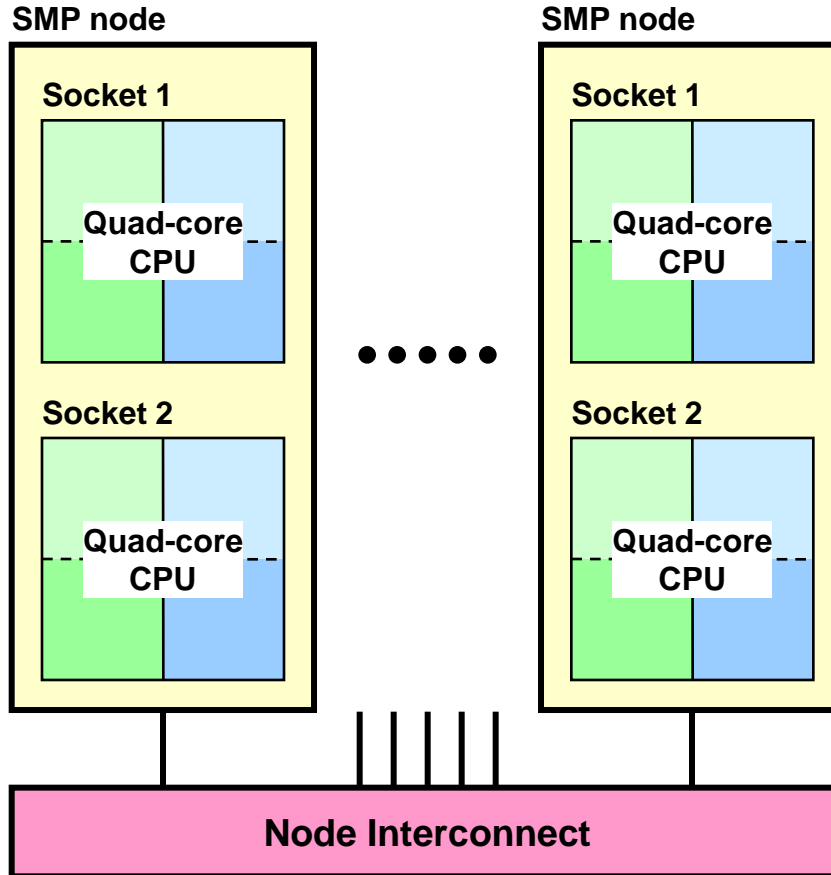
- Cluster of
 - ccNUMA node with several multi-core CPUs
 - nodes with multi-core CPU + GPU
 - nodes with multi-core CPU + Intel PHI
 - ...

Programming models

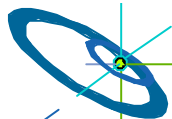
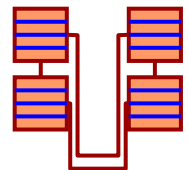
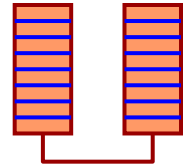
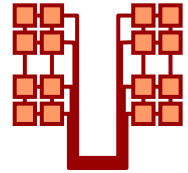
- MPI + Threading
 - OpenMP
 - Cilk
 - TBB (Threading Building Blocks)
- MPI + MPI shared memory
- MPI + Accelerator
 - OpenACC
 - OpenMP 4.0 accelerator support
 - CUDA
 - OpenCL
 - ...
- Pure MPI



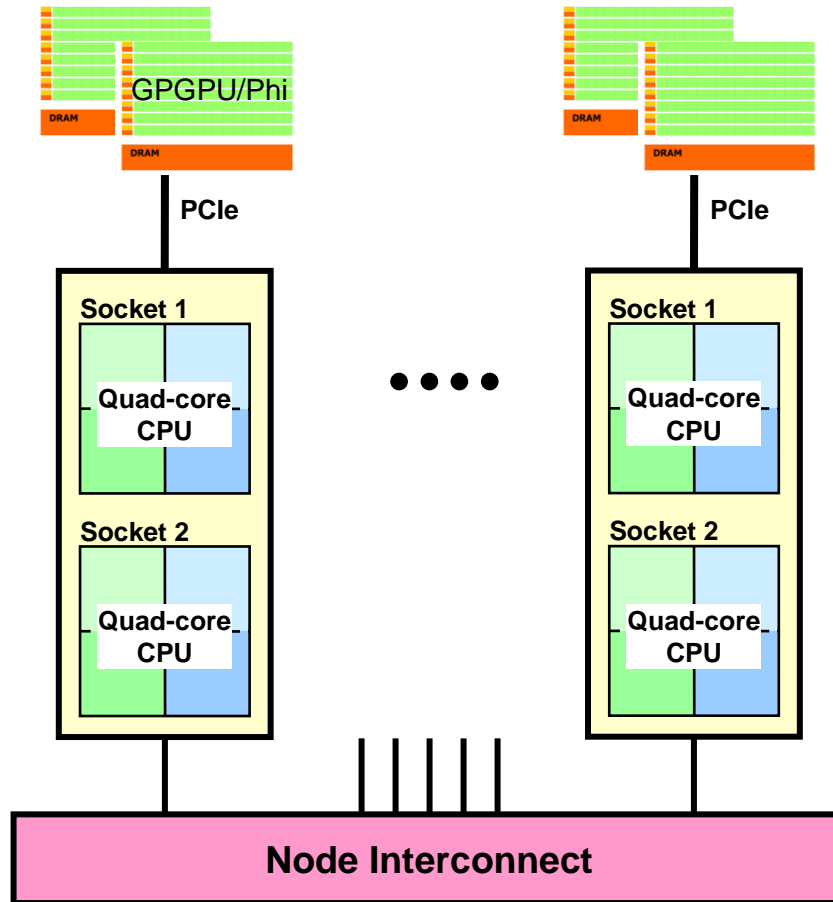
Motivation



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



More Options



Number of options multiply if accelerators are added

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



Splitting the Hardware Hierarchy

Hierarchical hardware

- Cluster of

- ccNUMA nodes with

- CPU's with

- $N \times$

- M cores with

- Hyperthreads

Hierarchical parallel programming

- MPI (outer level) +

- X (e.g. OpenMP)

Many possibilities for splitting the hardware hierarchy into MPI + X:

- 1 MPI process per ccNUMA node

-
-
-

- OpenMP only for hyperthreading

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

Outline

Motivation

Introduction

Pure MPI

MPI + MPI-3.0 shared memory

MPI + OpenMP on multi/many-core

MPI + Accelerators



Introduction

Typical hardware bottlenecks and challenges

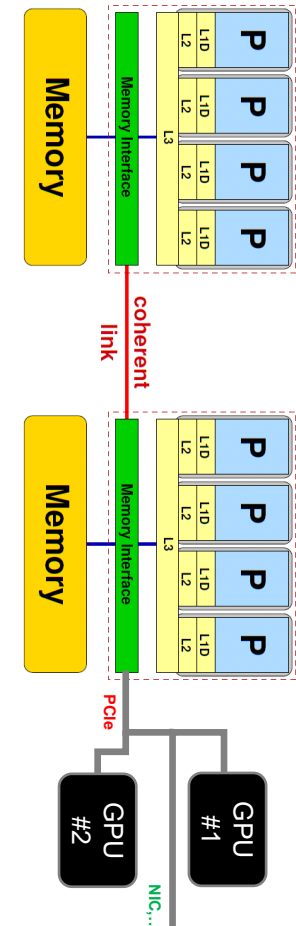


H L R I S



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

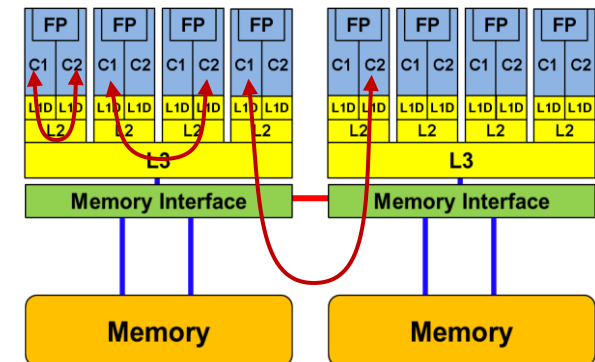
Example:

- 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



Running the code *efficiently*?

- Symmetric, UMA-type compute nodes have become rare animals
 - NEC SX
 - Intel 1-socket (Xeon 12XX) – rare in cluster environments
 - Hitachi SR8000, IBM SP2, single-core multi-socket Intel Xeon... (all dead)
- Instead, systems have become “non-isotropic” on the node level, with rich *topology*:
 - **ccNUMA** (AMD Opteron, SGI UV, IBM Power, Intel Nehalem/SandyBridge/...)
 - Inter-domain access, **contention**
 - Consequences of **file I/O** for page placement
 - Placement of MPI buffers
 - **Multi-core, multi-socket**
 - Intra-node vs. inter-node MPI performance
 - Shared caches, bandwidth bottlenecks
 - Topology-dependent OpenMP overhead

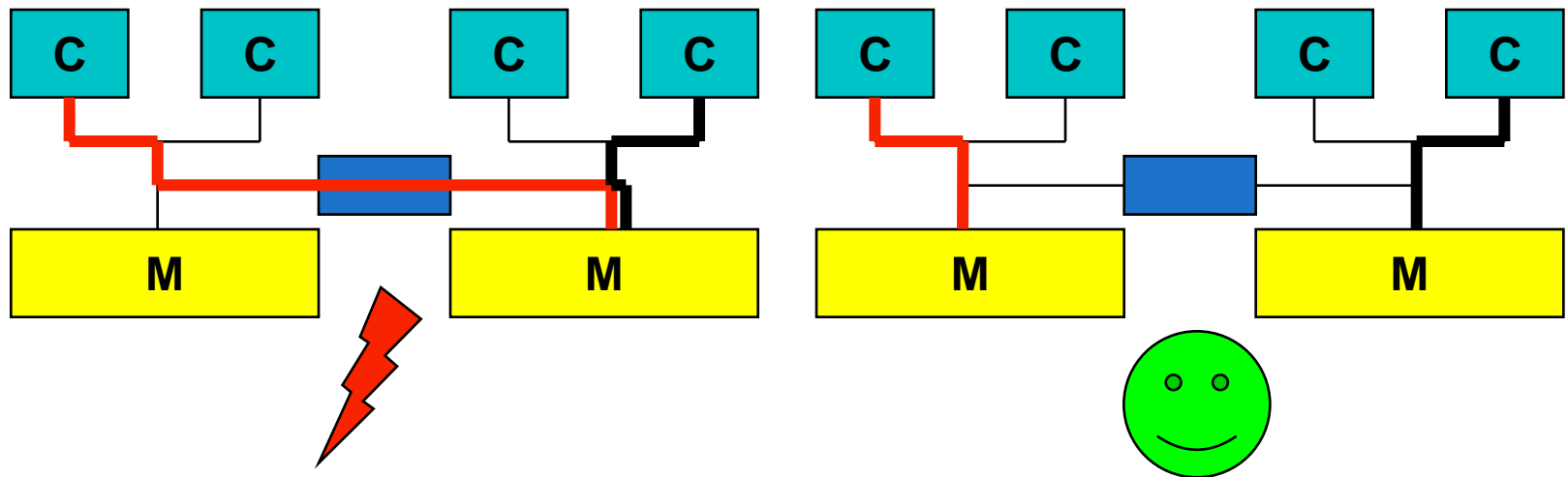


Interlude: ccNUMA



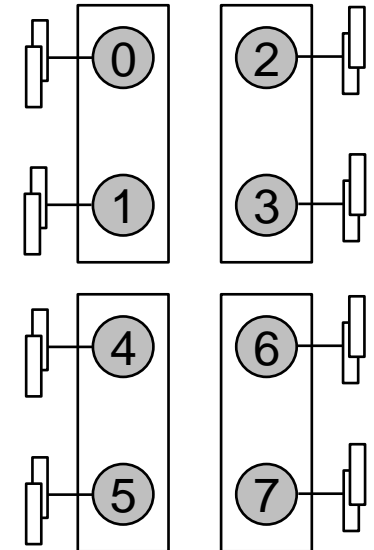
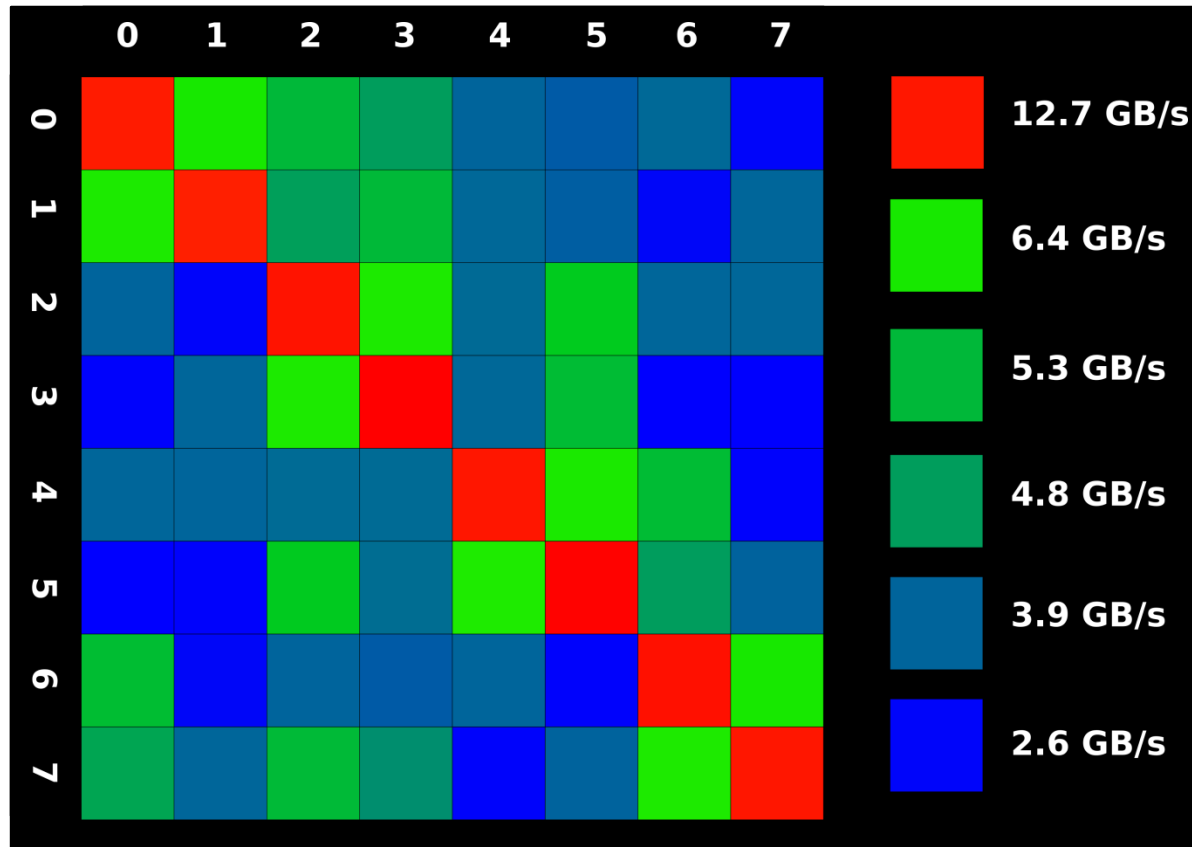
A short introduction to ccNUMA

- 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?

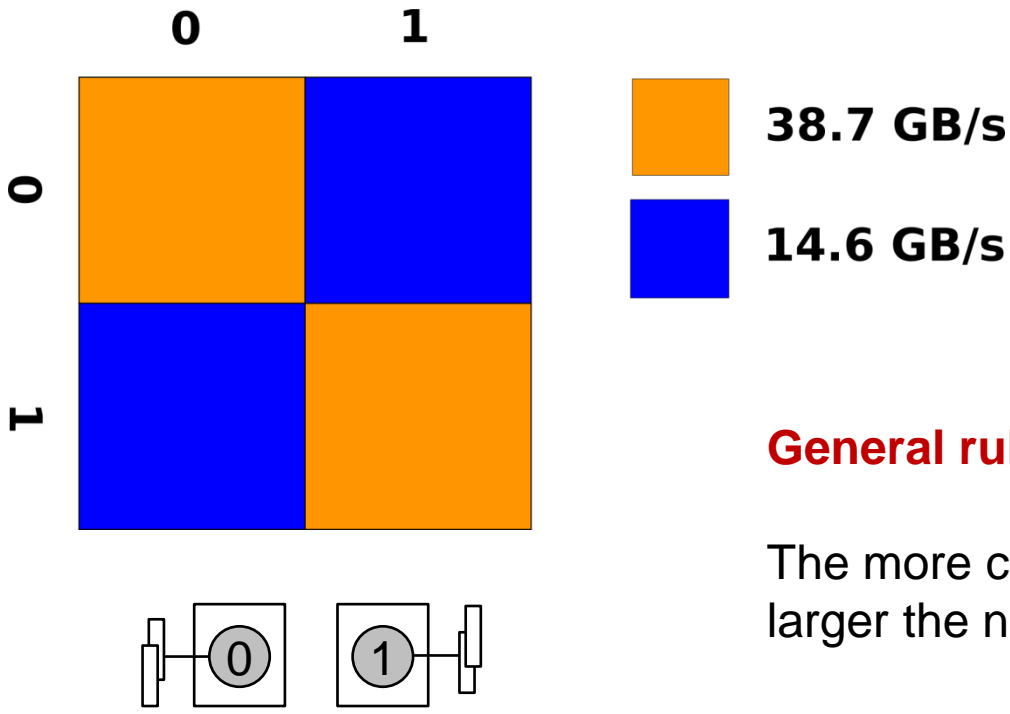
- Example: AMD Magny Cours 4-socket system (8 chips, 4 sockets)
STREAM Triad bandwidth measurements



skipped

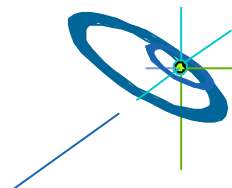
How much bandwidth does non-local access cost?

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



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

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

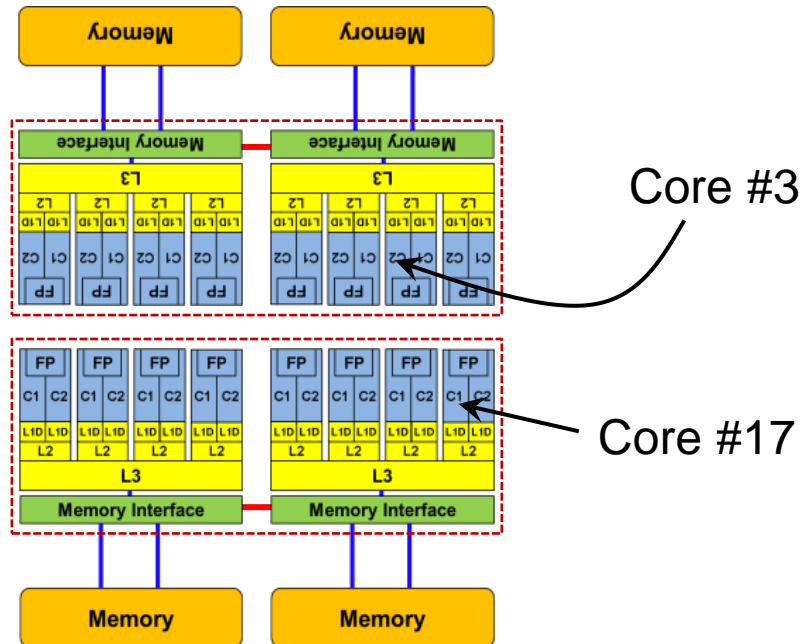
- See later for more details and examples

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!)



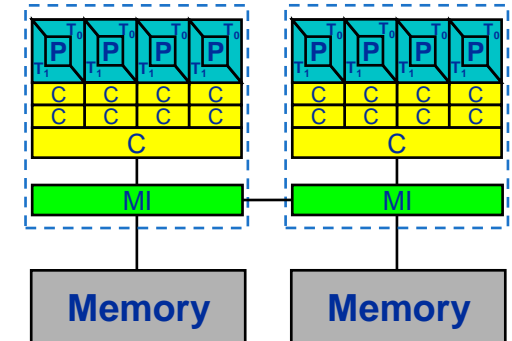
Output of likwid-topology

```
CPU name:      Intel Core i7 processor
CPU clock:     2666683826 Hz
*****
```

```
Hardware Thread Topology
*****
```

```
Sockets:      2
Cores per socket: 4
Threads per core: 2
```

HWThread	Thread	Core	Socket
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	2	0
5	1	2	0
6	0	3	0
7	1	3	0
8	0	0	1
9	1	0	1
10	0	1	1
11	1	1	1
12	0	2	1
13	1	2	1
14	0	3	1
15	1	3	1



likwid-topology continued

Socket 0: (0 1 2 3 4 5 6 7)

Socket 1: (8 9 10 11 12 13 14 15)

Cache Topology

Level: 1

Size: 32 kB

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

Level: 2

Size: 256 kB

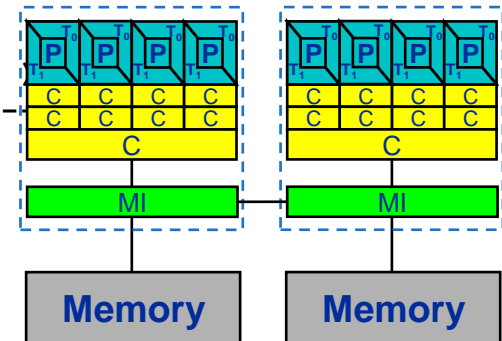
Cache groups: (0 1) (2 3) (4 5) (6 7) (8 9) (10 11) (12 13) (14 15)

Level: 3

Size: 8 MB

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

- ... and also try the ultra-cool **-g** option!



Intra-node MPI characteristics: IMB Ping-Pong benchmark

- Code (to be run on 2 cores):

```

wc = MPI_WTIME()

do i=1,NREPEAT

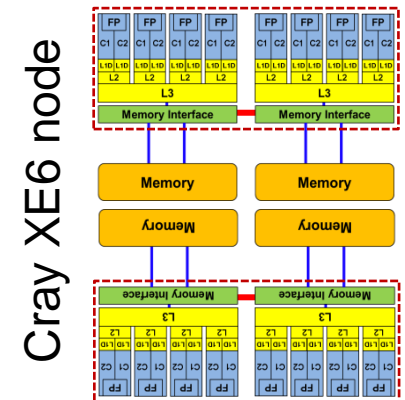
  if(rank.eq.0) then
    MPI_SEND(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD,ierr)
    MPI_RECV(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD, &
              status,ierr)

  else
    MPI_RECV(...)
    MPI_SEND(...)
  endif

enddo

wc = MPI_WTIME() - wc
  
```

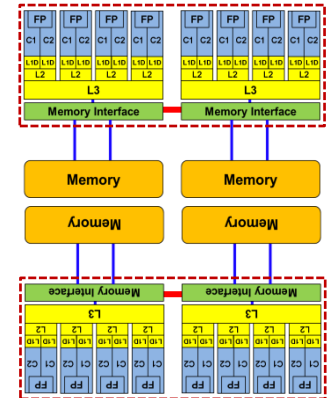
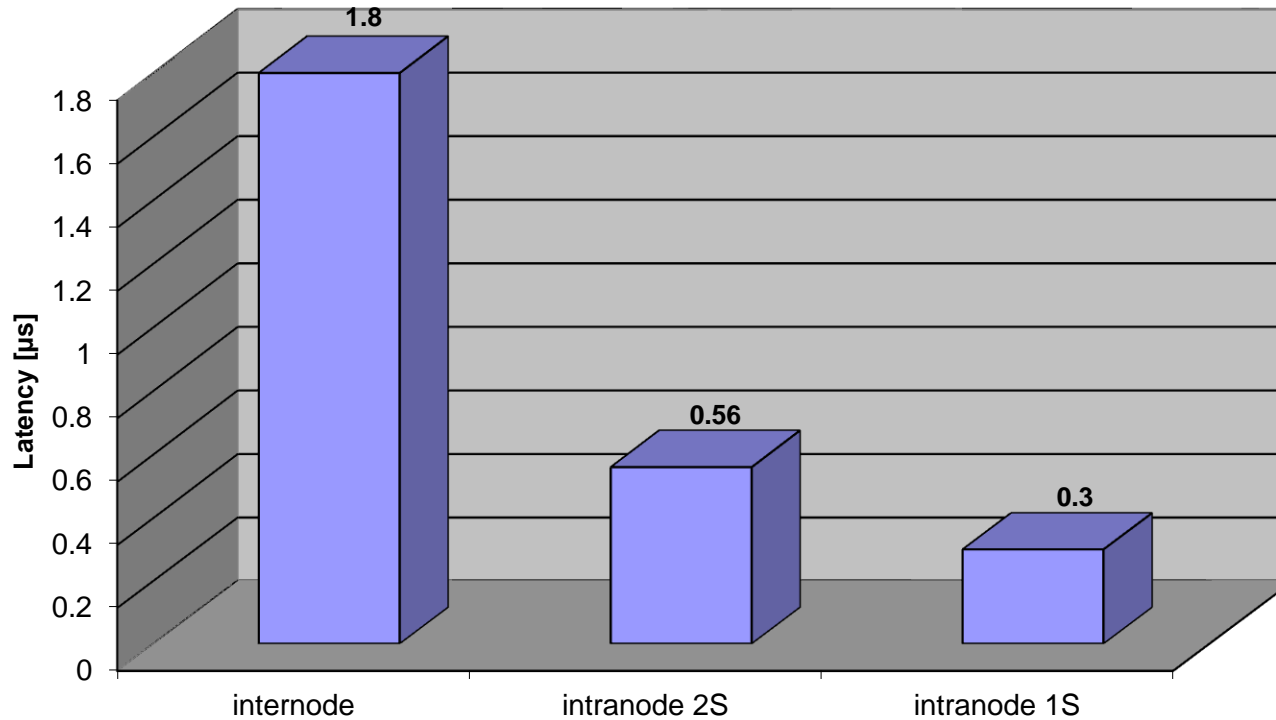
- Intranode (1S): `aprun -n 2 -cc 0,1 ./a.out`
- Intranode (2S): `aprun -n 2 -cc 0,16 ./a.out`
- Internode: `aprun -n 2 -N 1 ./a.out`



skipped

IMB Ping-Pong: Latency

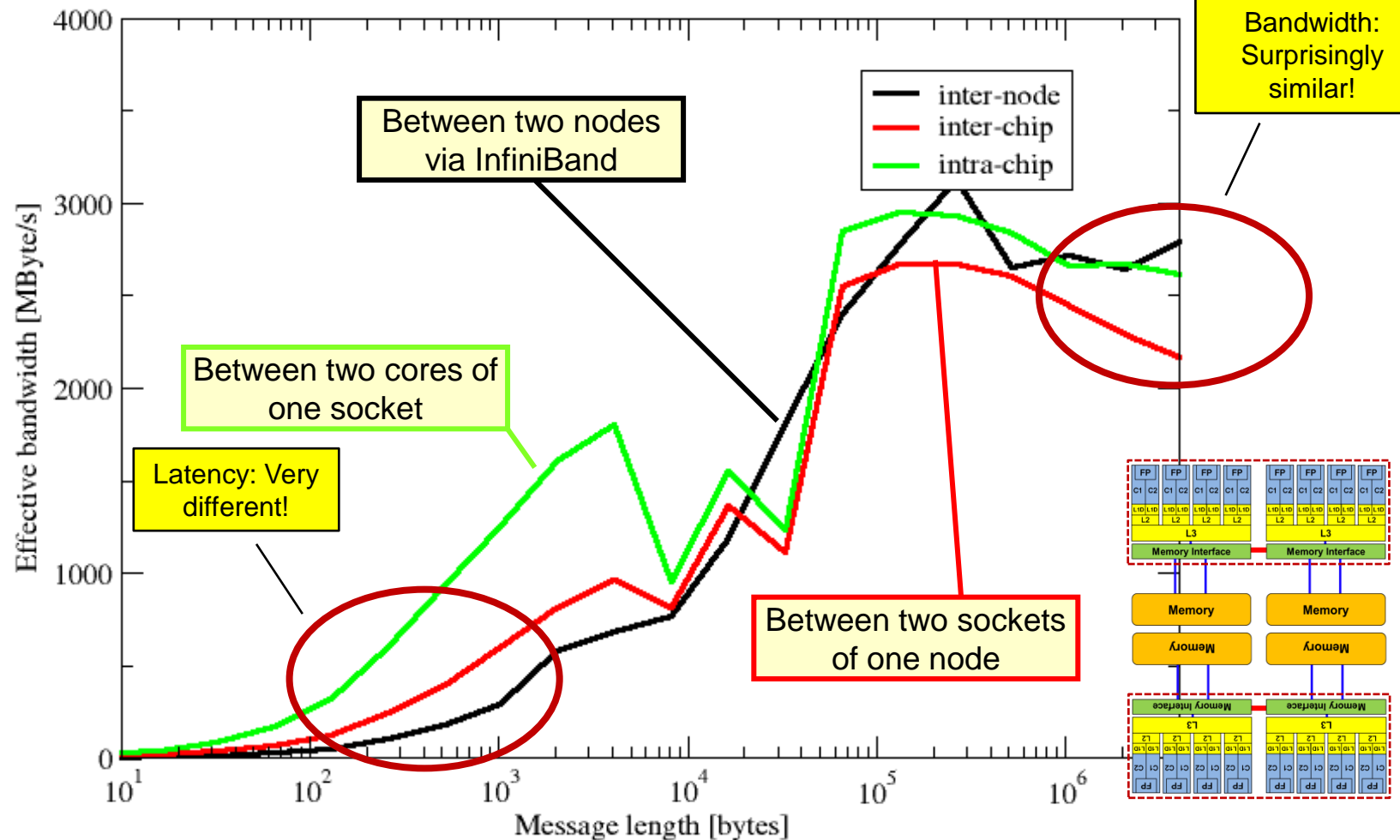
Intra-node vs. Inter-node on Cray XE6



Affinity matters!

IMB Ping-Pong: Bandwidth Characteristics

Intra-node vs. Inter-node on Cray XE6



The throughput-parallel vector triad benchmark

Microbenchmarking for architectural exploration

- Every core runs its own, independent bandwidth benchmark

```
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
```

Repeat many times

```
do i=1,N
```

```
A(i) = B(i) + C(i) * D(i)
```

Actual benchmark loop

```
enddo
```

```
if(.something.that.is.never.true.) then
```

```
call dummy(A,B,C,D)
```

```
endif
```

Prevent smart-ass compilers from optimizing away the outer loop

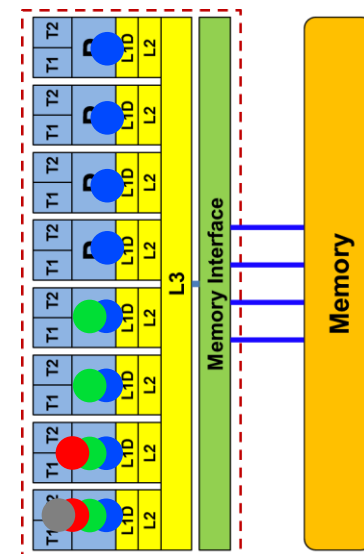
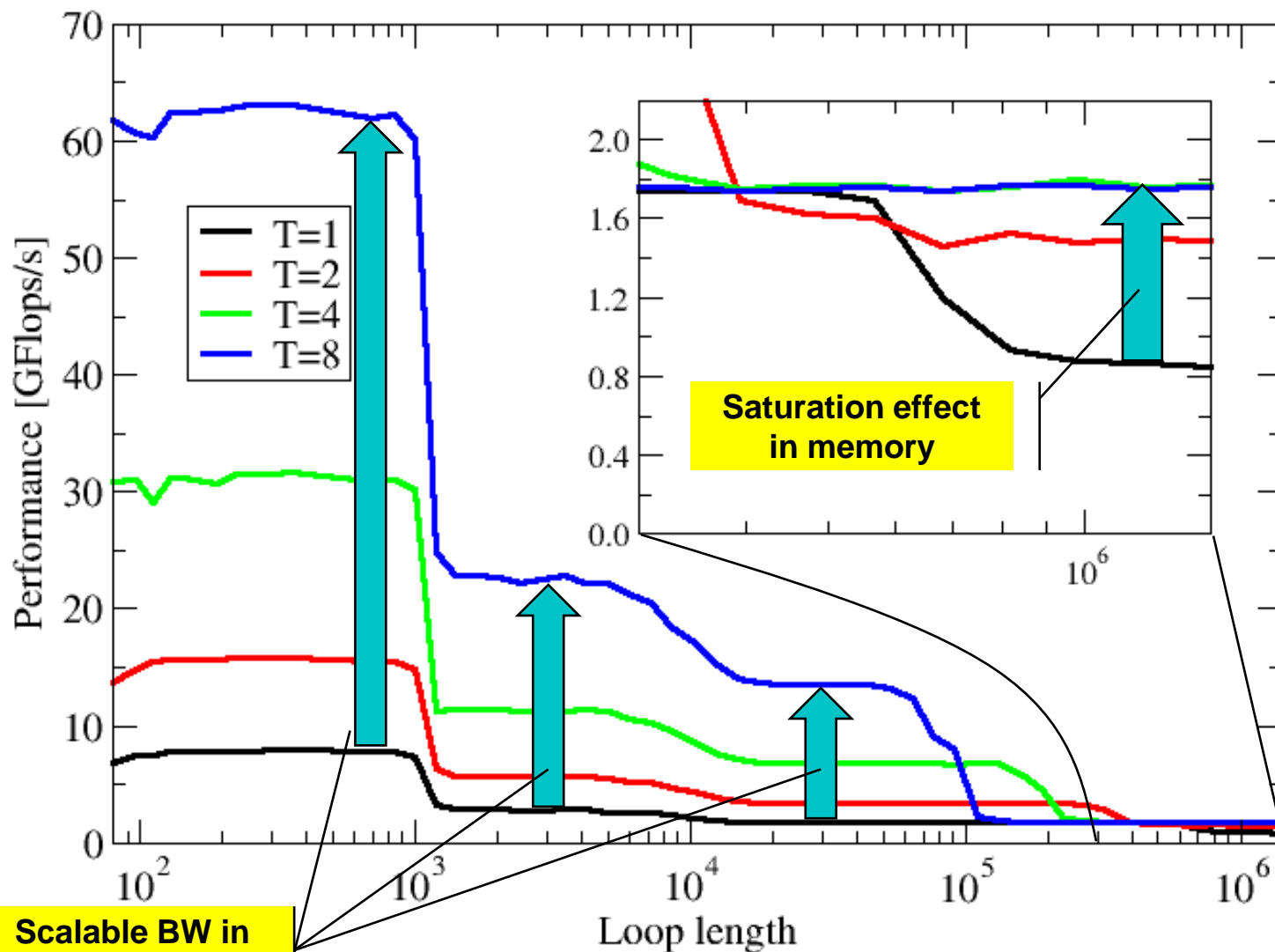
```
enddo
```

```
!$OMP END PARALLEL
```

- pure hardware probing, no impact from OpenMP overhead



Throughput vector triad on Sandy Bridge socket (3 GHz)



Scalable BW in L1, L2, L3 cache

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



Remarks on Cost-Benefit Calculation



Remarks on Cost-Benefit Calculation

Costs

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

Benefit

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

Question: Do you want to spend work hours without a final benefit?

Programming models



Programming models - pure MPI



Pure MPI

pure MPI
one MPI process
on each core

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?

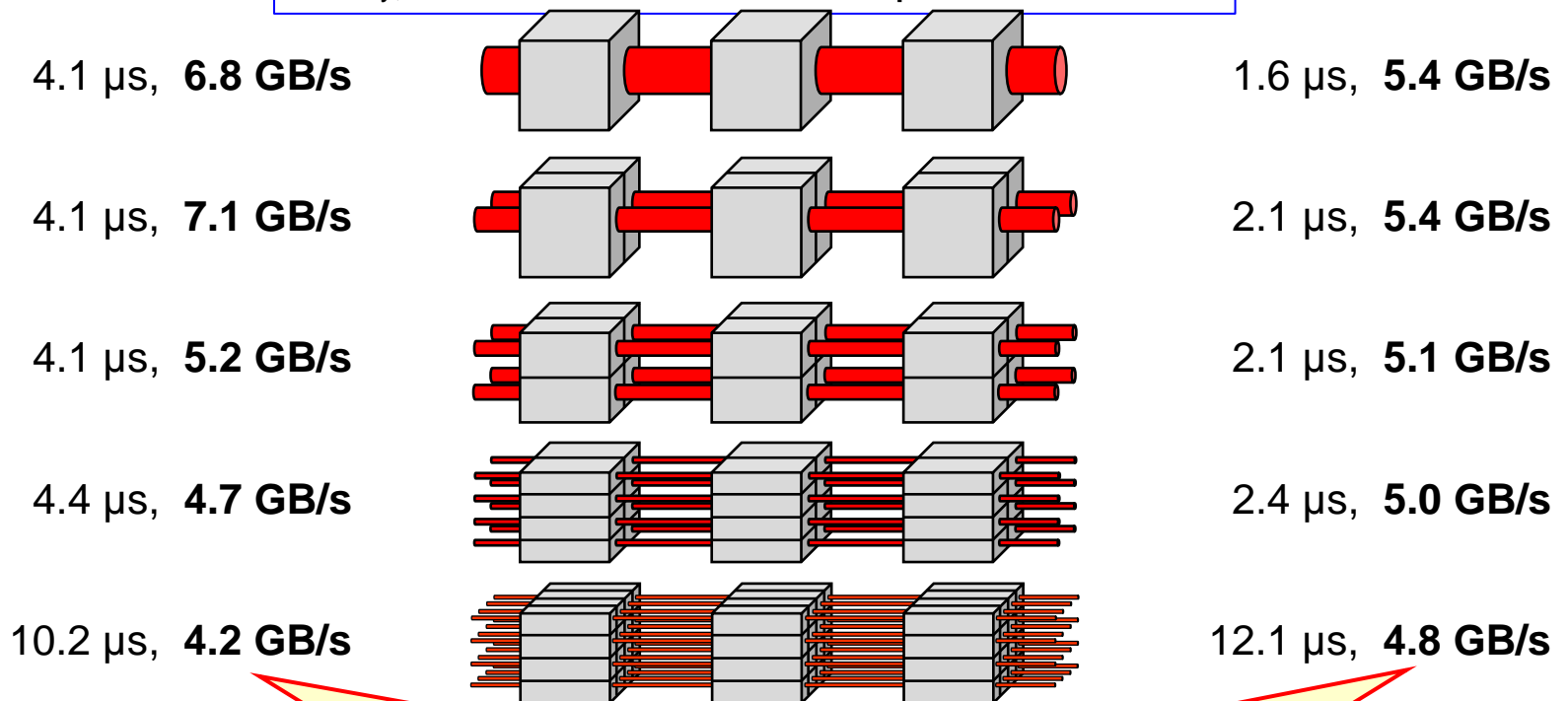
- Bandwidth of parallel communication links between SMP nodes

Cray XC30
(Sandybridge @ HLRS)

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**

Xeon+Infiniband
(beacon @ NICS)

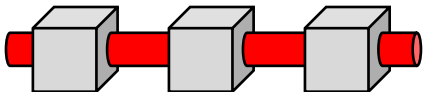
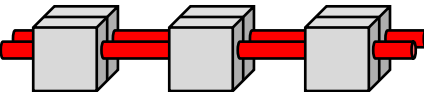
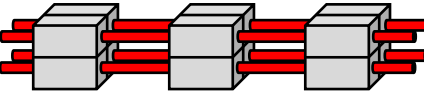
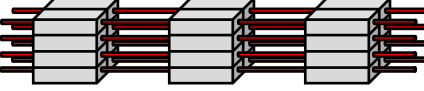


Conclusion:

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**

	Links per Phi	<u>One Phi per node</u> (beacon @ NICS)	<u>4 Phis on one node</u> (beacon @ NICS)
	1x	15 μ s, 0.83 GB/s	15 μ s, 0.83 GB/s
	2x	26 μ s, 0.87 GB/s	
	4x	25 μ s, 0.91 GB/s	
	8x	23 μ s, 0.91 GB/s	
⋮	16x	24 μ s, 0.92 GB/s	
	30x	21 μ s, 0.91 GB/s	
	60x	51 μ s, 0.90 GB/s	

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)

– Number of MPI processes	Latency (16 byte msg)	Bandwidth (bi-directional, 512 kB messages, per process)
4	9 μ s	0.80 GB/s
16	11 μ s	0.75 GB/s
30	15 μ s	0.66 GB/s
60	29 μ s	0.50 GB/s
120	149 μ s	0.19 GB/s
240	745 μ s	0.05 GB/s

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



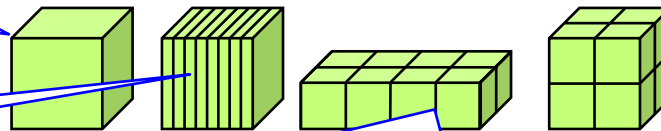
Levels of communication or data access

- Three levels:
 - Between the SMP nodes
 - Between the sockets inside of a ccNUMA SMP node
 - Between the cores of a socket
- On all levels, the communication should be minimized:
 - With 3-dimensional sub-domains:

- **They should be as cubic as possible**

Green = Shape of data.
Optimal sub-domain within an SMP node

Sub-sub-domain within a core



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

Optimal surfaces on SMP and core level

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

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



Originally perfectly optimized shape for each MPI process; but terrible when clustered only in one dimension
→ next slide

Loss of communication bandwidth if not cubic



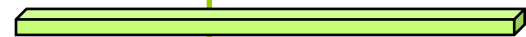
$$N^3 = N \times N \times N$$



$$N^3 = 2 \frac{N}{\sqrt[3]{2}} \times 1 \frac{N}{\sqrt[3]{2}} \times 1 \frac{N}{\sqrt[3]{2}}$$



$$N^3 = 4 \frac{N}{\sqrt[3]{4}} \times 1 \frac{N}{\sqrt[3]{4}} \times 1 \frac{N}{\sqrt[3]{4}}$$



$$N^3 = 8 \frac{N}{\sqrt[3]{8}} \times 1 \frac{N}{\sqrt[3]{8}} \times 1 \frac{N}{\sqrt[3]{8}}$$



$$N^3 = 16 \frac{N}{\sqrt[3]{16}} \times 1 \frac{N}{\sqrt[3]{16}} \times 1 \frac{N}{\sqrt[3]{16}}$$

- $N^3 = 32 \frac{N}{\sqrt[3]{32}} \times 1 \frac{N}{\sqrt[3]{32}} \times 1 \frac{N}{\sqrt[3]{32}}$

- $N^3 = 64 \frac{N}{\sqrt[3]{64}} \times 1 \frac{N}{\sqrt[3]{64}} \times 1 \frac{N}{\sqrt[3]{64}}$

$$bw = 100\% \cdot bw_{optimal}$$

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

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

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

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

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

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

Slow down factors of your application (communication footprint calculated with optimal bandwidth)

- With 20% communication footprint: **Slow down** by 1.01, 1.04, 1.08, 1.14, 1.23, or 1.34
- With **50%** communication footprint: **Slow down** by 1.03, 1.10, 1.20, 1.36, 1.56, or 1.85!

The topology problem: How to fit application sub-domains to hierarchical hardware

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

- Not needed
 - with pure MPI+OpenMP, i.e., one MPI process per SMP node
 - ccNUMA-aware hybrid MPI+OpenMP, i.e., with one MPI process per physical ccNUMA domain (e.g., socket) **and** the number of ccNUMA domain is small, e.g., only 2.

In these cases, one-level domain-decomposition is enough

- Needed for
 - ccNUMA-aware hybrid MPI+OpenMP with several MPI processes per SMP node, e.g., one process per socket, and 4 or more sockets
 - MPI + MPI-3.0 shared memory
 - Pure MPI



Pure MPI – multi-core aware

pure MPI
ccNUMA aware hybrid
Hybrid MPI+MPI

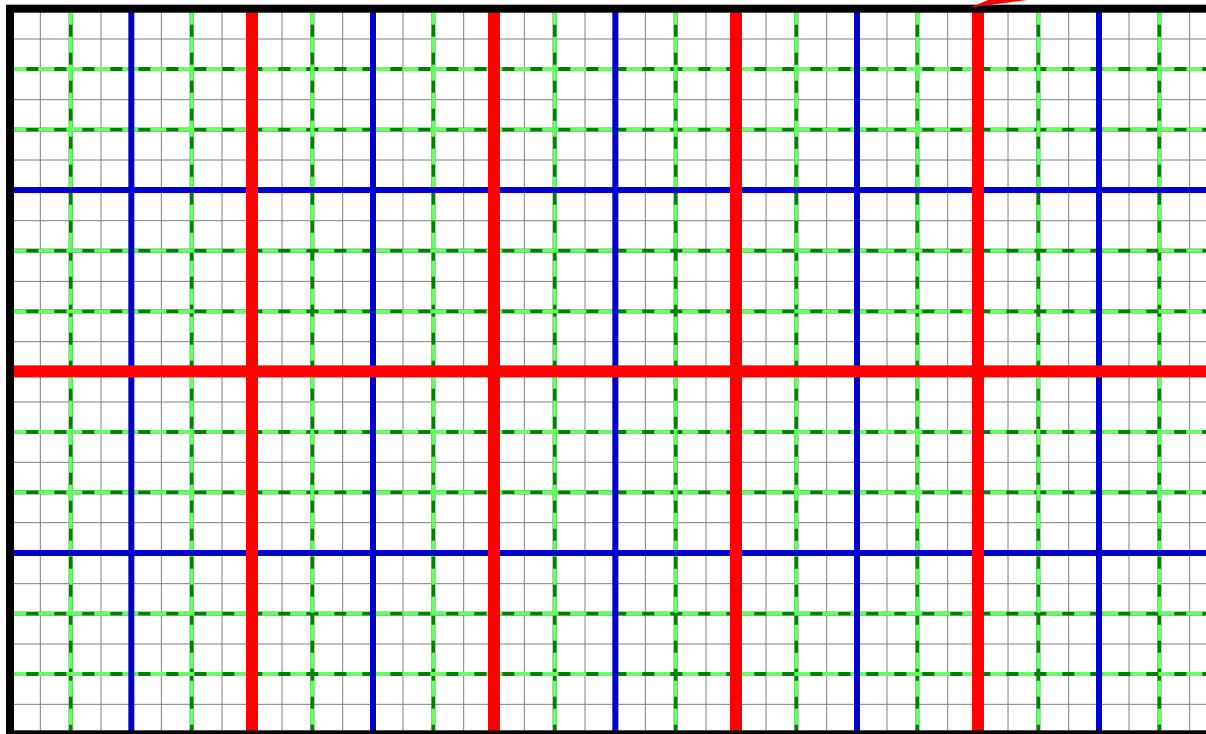
- **Hierarchical domain decomposition**
(or distribution of Cartesian arrays)

Domain decomposition:
1 sub-domain / **SMP node**

Further
partitioning:
1 sub-domain /
socket

1 / **core**

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



Example on 10 nodes, each with 4 sockets, each with 6 cores.

How to achieve such hardware-aware domain decomposition (DD)?

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI

- 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
 - ...
 - Cartesian/structured domain decomposition on finest MPI level
 - E.g., sockets (with ccNUMA-aware hybrid MPI+OpenMP)
 - E.g., cores (with pure MPI or MPI+MPI-3.0 shared memory)
 - Hierarchical re-numbering the MPI processes together with MPI Cartesian virtual coordinates
→ next slides
- Unstructured grids → coming later



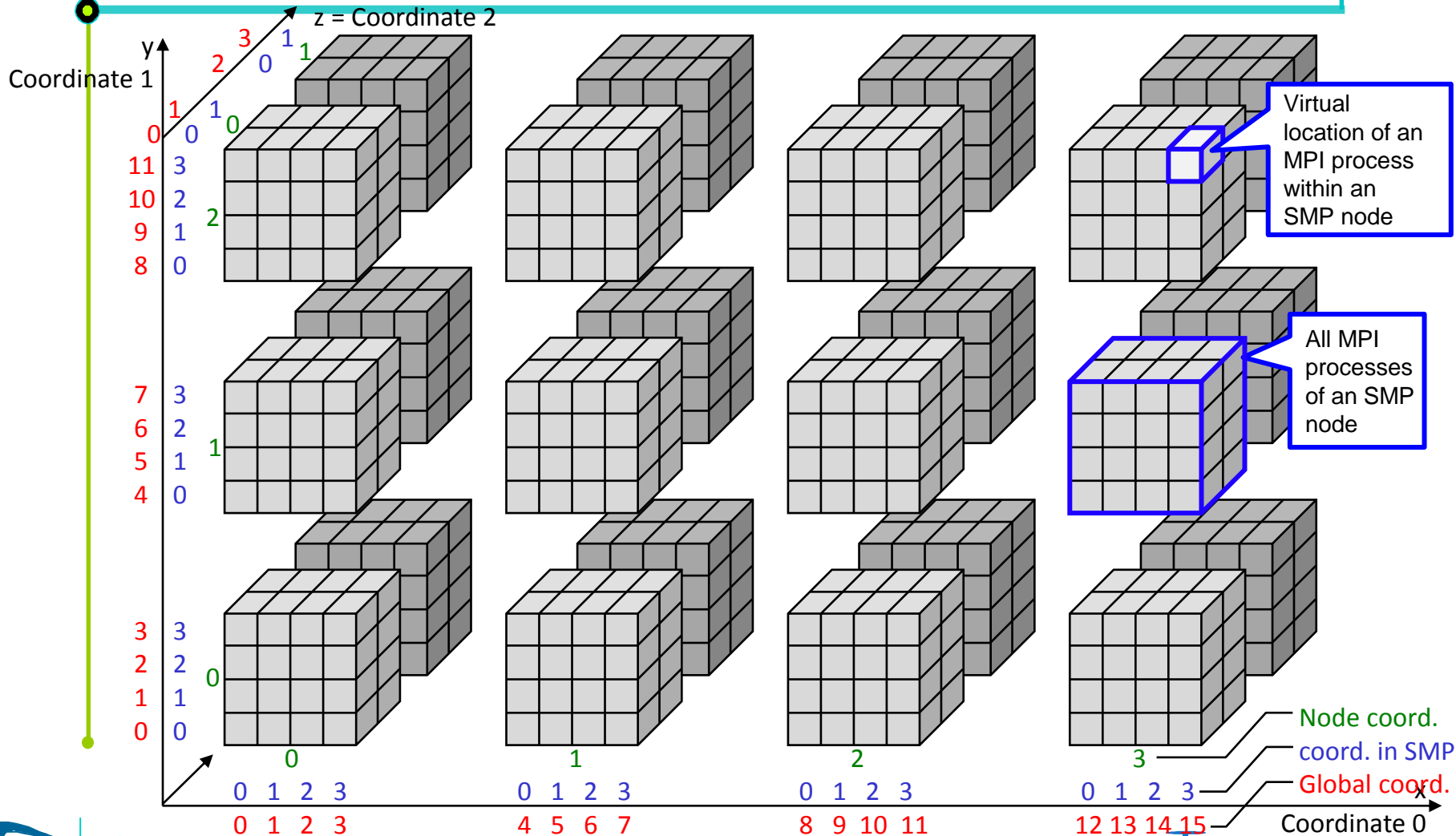
Implementation hints on following (skipped) slide

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI

Hierarchical Cartesian DD



skipped

Hierarchical Cartesian DD

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI

```
// 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); }
```



skipped

Hierarchical Cartesian DD

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI

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

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

**Copying it for the
other processes in
each SMP node**



H

L

R

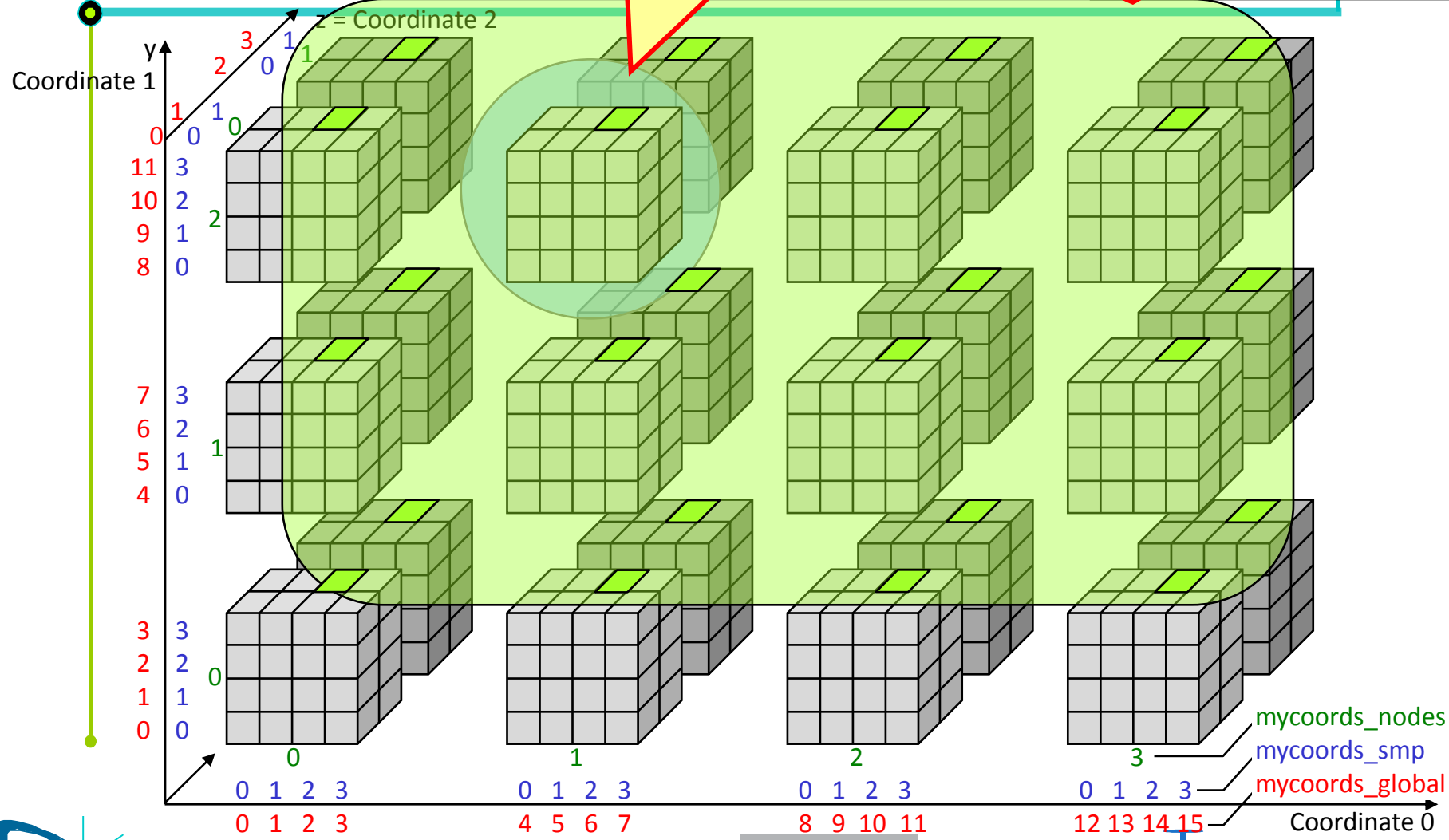
skipped

Hierarchical

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

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

the MPI
aware hybrid
hybrid MPI+MPI



H L R S



skipped

Hierarchical Cartesian DD

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI

```
// 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
```



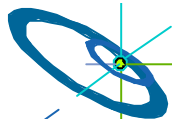
How to achieve a hierarchical DD for unstructured grids?

pure MPI

ccNUMA aware hybrid

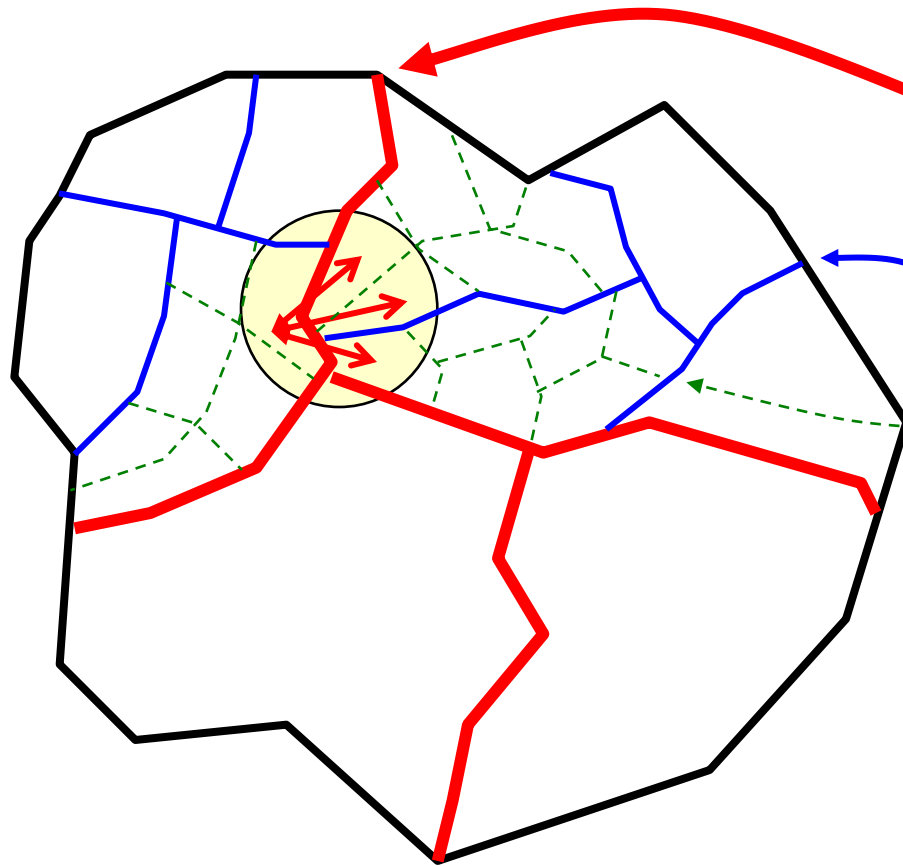
Hybrid MPI+MPI

- **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



pure MPI
ccNUMA aware hybrid
Hybrid MPI+MPI

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**



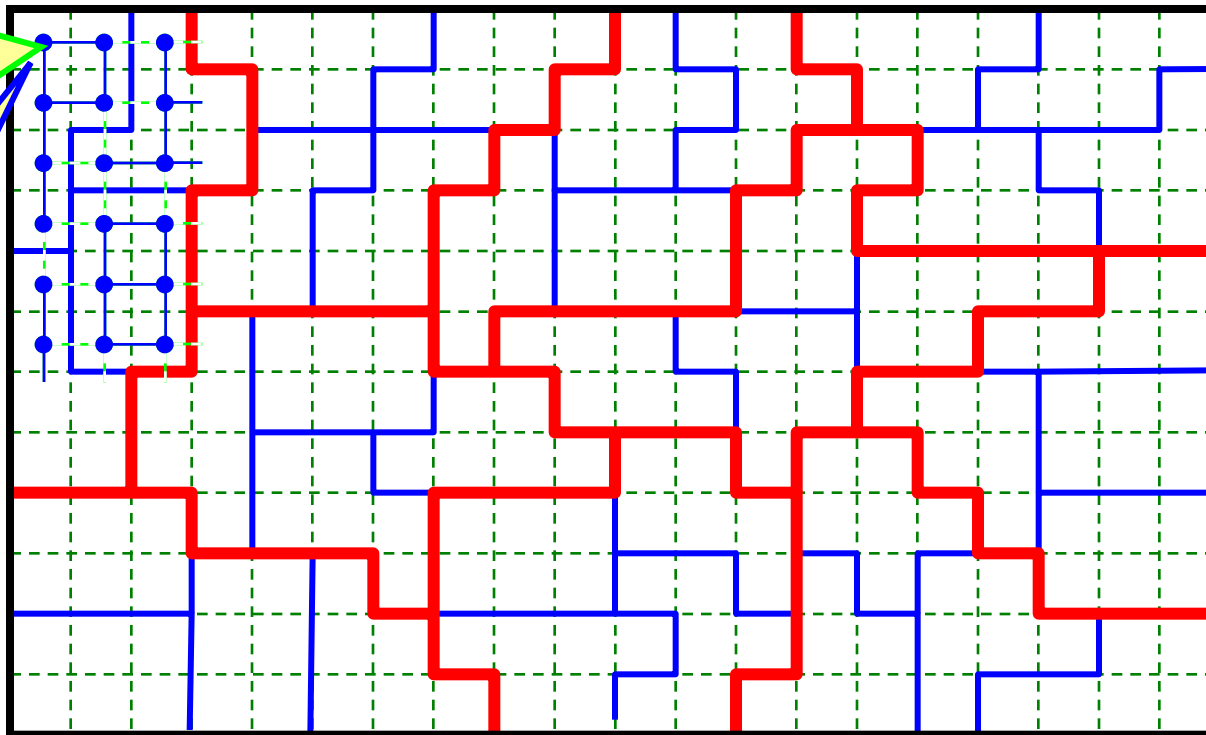
Bottom-up – Multi-level DD through recombination

pure MPI
ccNUMA aware hybrid
Hybrid MPI+MPI

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

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 **exactly**
 - 6 cores, and
 - 4 numa-domains (i.e., sockets or dies)
- **Advantage:** Communication is balanced!



— skipped —

Profiling solution

pure MPI

ccNUMA aware hybrid

Hybrid MPI+MPI

- 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 SC'14
Tutorial:
**Node-Level
Performance
Engineering**



Scalability of MPI to hundreds of thousands ...

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 arraysone can expect:
 - Significant, but still scalable memory overhead for halo cells.
 - MPI library is internally scalable:
 - **E.g., mapping ranks → hardware grid**
 - Centralized storing in shared memory (OS level)
 - In each MPI process, only used neighbor ranks are stored (cached) in process-local memory.
 - **Tree based algorithm with $O(\log N)$**
 - From 1000 to 1000,000 process $O(\log N)$ only doubles!

The vendors should deliver scalable MPI libraries for their largest systems!

To overcome MPI scaling problems

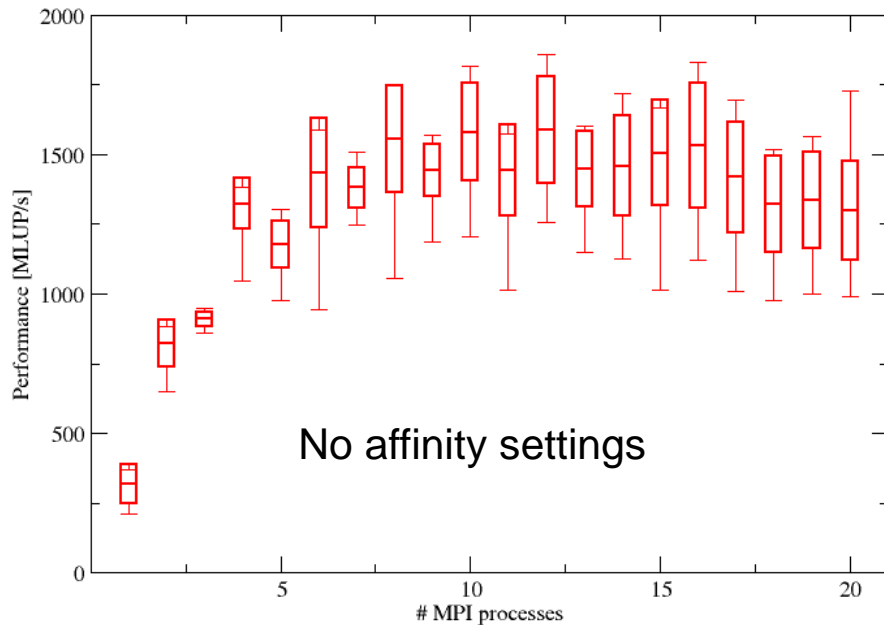
- Reduced number of MPI messages, reduced aggregated message size } compared to pure MPI
- MPI has a few scaling problems
 - Handling of more than 10,000 MPI processes
 - Irregular Collectives: MPI_....v(), e.g. MPI_Gatherv()
 - **Scaling applications should not use MPI_....v() routines**
 - MPI-2.1 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**. Proceedings EuroPVM/MPI 2009.
- Hybrid programming reduces all these problems (due to a smaller number of processes)



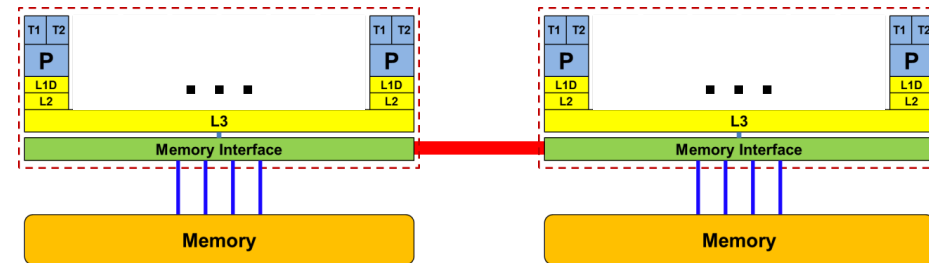
Pinning of MPI processes

- Pinning is helpful for all programming models
- Highly system-dependent!
- Intel MPI: env variable I_MPI_PIN
- OpenMPI:
mpirun options `-bind-to-core`, `-bind-to-socket`, `-bycore`, `-byslot` ...

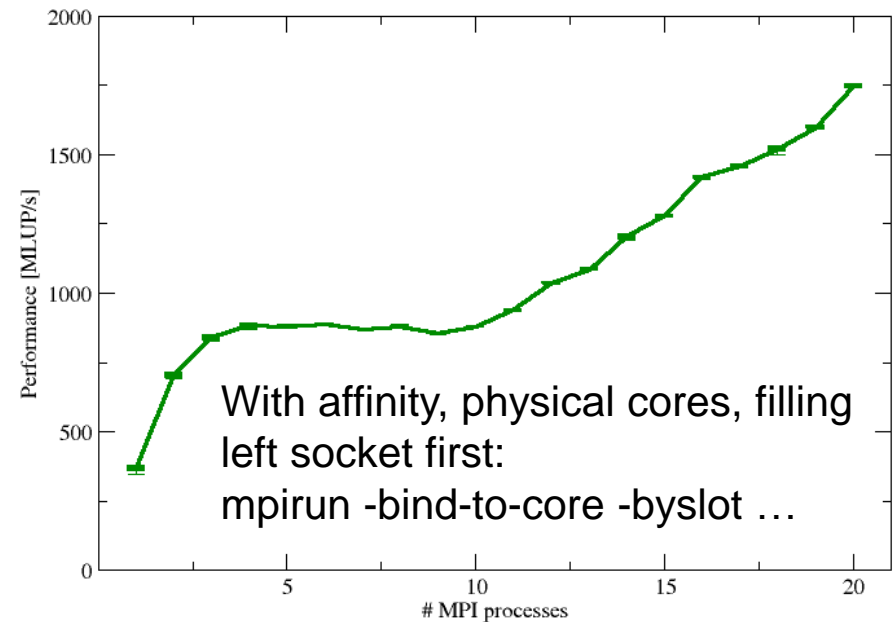
Anarchy vs. affinity with a heat equation solver



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



2x 10-core Intel Ivy Bridge, OpenMPI



Pure MPI: Main advantages

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

Pure MPI: Main disadvantages

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



Pure MPI: Conclusions

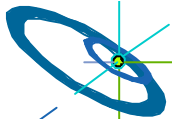
- Still a good programming model for small and medium size applications.
- Major problem may be memory consumption



Programming models

- MPI + MPI-3.0

shared memory



Hybrid MPI + MPI-3 shared memory

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

Advantages

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

Major Problems

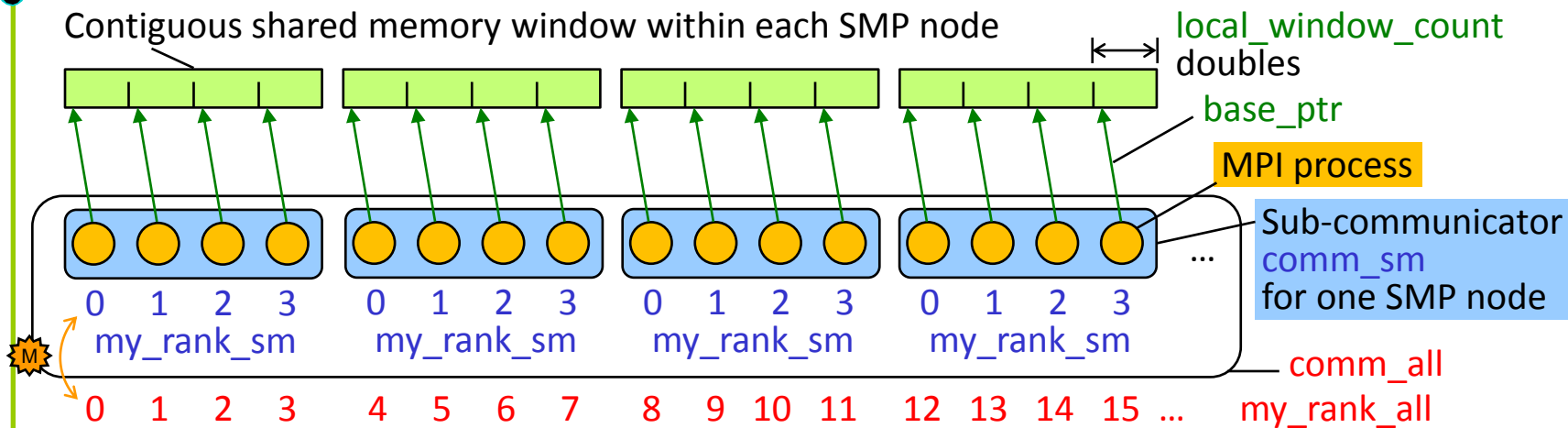
- Communicator must be split into shared memory islands
- To minimize shared memory communication overhead:
 Halos (or the data accessed by the neighbors) must be stored in
 MPI shared memory windows
- Same work-sharing as with pure MPI
- MPI-3.0 shared memory synchronization waits for clarification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

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



```

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);

```

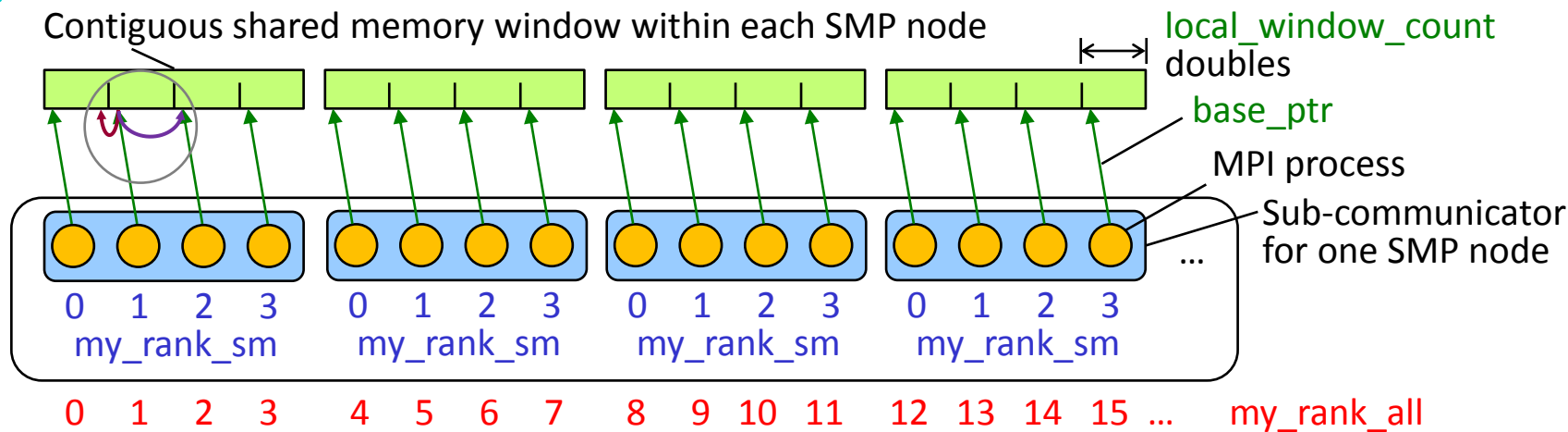
Sequence in comm_sm as in comm_all

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



```
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);
```

**Synchroni-
zation**

**Synchroni-
zation**

```
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] );
```

Local stores

Direct load access to remote window portion

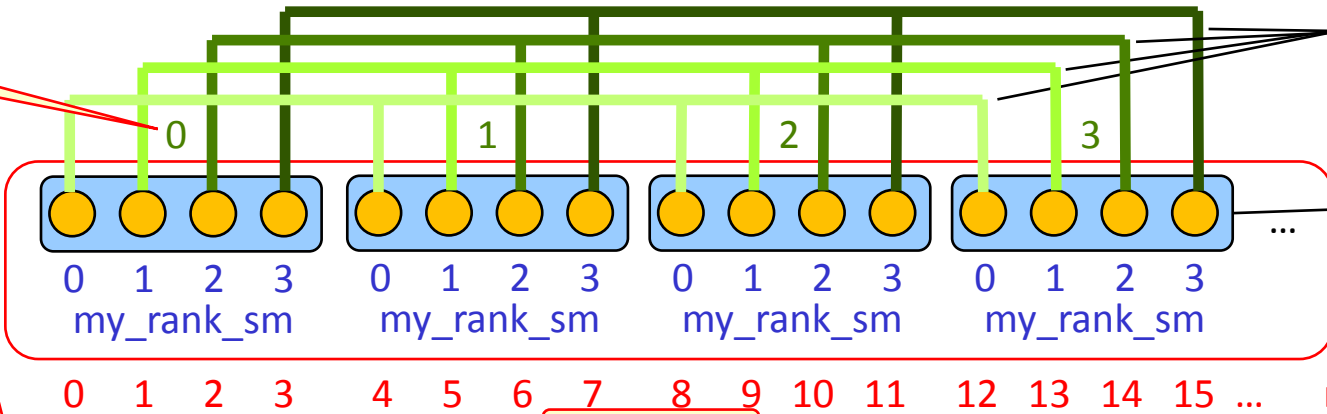
Establish comm_sm, comm_nodes, comm_all, if SMPs are not contiguous within comm_orig

my_rank_nodes

comm_nodes
combining all processes with same my_rank_sm

Sub-communicator for one SMP node:
comm_sm

comm_all
my_rank_all



```

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);

```

Result: comm_nodes combines all processes with a given my_rank_sm into a separate communicator.

On processes with my_rank_sm > 0, this comm_nodes is unused because node-numbering within these comm_nodes may be different.

Expanding the numbering from **comm_nodes** with my_rank_sm == 0 to all new node-to-node communicators **comm_nodes**.

Calculating **my_rank_all** and establishing global communicator **comm_all** with sequential SMP subsets.

Exscan does not return value on the first rank, therefore

Establish a communicator **comm_sm** with ranks **my_rank_sm** on each SMP node

skipped

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.**

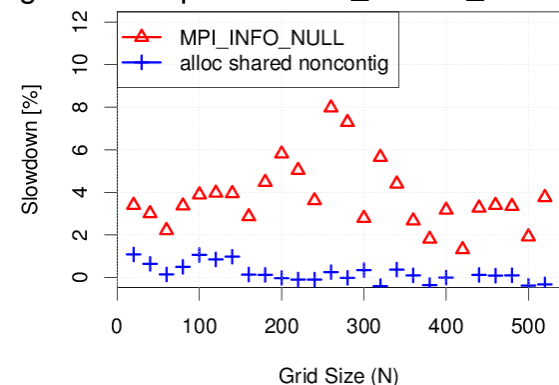
<http://link.springer.com/content/pdf/10.1007%2Fs00607-013-0324-2.pdf>

Hybrid Parallel Programming

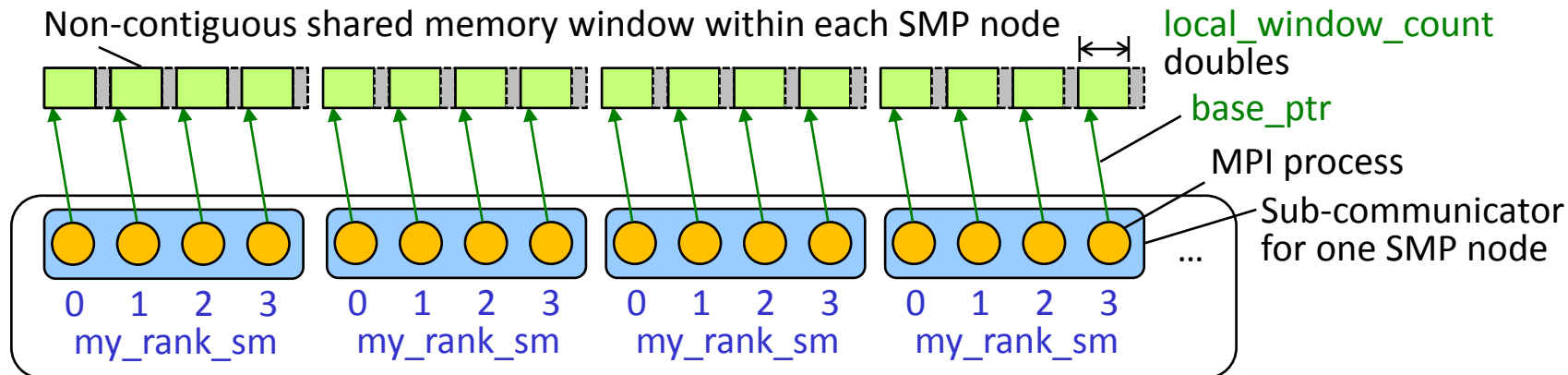
Slide 67 / 170

Rabenseifner, Hager, Jost

NUMA effects?
Significant impact of alloc_shared_noncontig



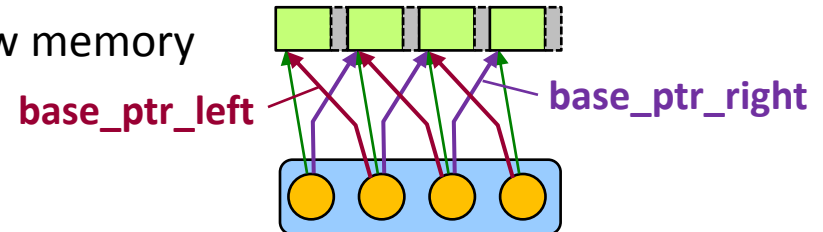
Non-contiguous shared memory allocation



```
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



```

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 ] );
    
```

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.

Another restriction in a
low-quality MPI:
MPI_COMM_SPLIT_TYPE
may return always
MPI_COMM_SELF

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`.

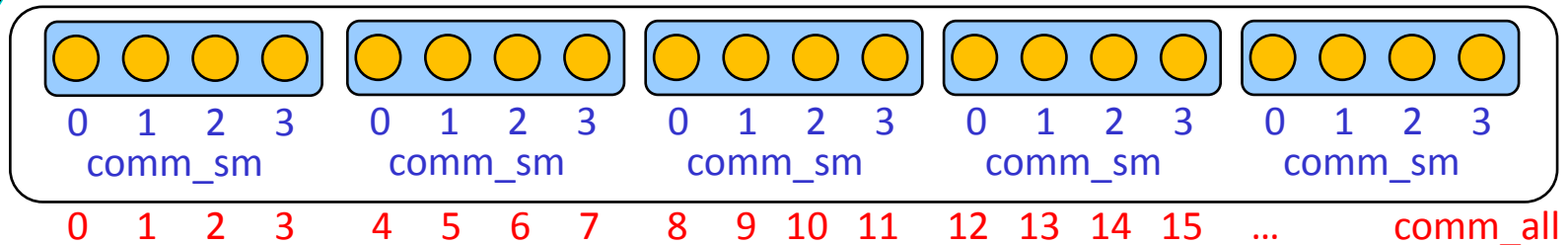
Due to default limit
of context IDs
in mpich

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

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

Thanks to Jeff Hammond and Jed Brown (ANL), Brian W Barrett (SANDIA), and Steffen Weise (TU Freiberg),
for input and discussion.

Splitting the communicator without MPI_COMM_SPLIT_TYPE



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

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

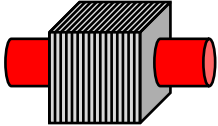
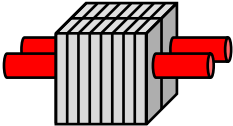
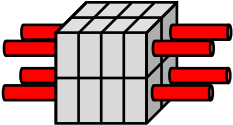
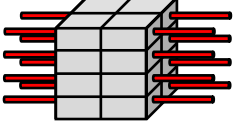
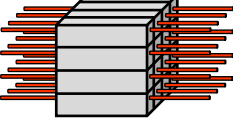
```
MPI_Comm_rank(comm_all, &my_rank);
MPI_Comm_split(comm_all, /*color*/ my_rank / size_sm, 0, &comm_sm);
MPI_Win_allocate_shared(...);
```

To guarantee shared memory, one may add an additional **`MPI_Comm_split_type(comm_sm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm_really);`**

Pure MPI versus MPI+MPI-3.0 shared memory

Internode: Irecv + Send

Additional intra-node communication with:

	Latency	Accumulated inter-node bandwidth per node		
	2.9 μ s,	4.4 GB/s	Irecv+send	← Pure MPI
	3.4 μ s,	4.4 GB/s	MPI-3.0 store	← MPI+MPI-3.0 shared memory
	3.0 μ s,	4.5 GB/s	Irecv+send	
	3.0 μ s,	4.6 GB/s	MPI-3.0 store	
	3.3 μ s,	4.4 GB/s	Irecv+send	
	3.5 μ s,	4.4 GB/s	MPI-3.0 store	
	5.2 μ s,	4.3 GB/s	Irecv+send	
	5.2 μ s,	4.4 GB/s	MPI-3.0 store	
	10.3 μ s,	4.5 GB/s	Irecv+send	
	10.1 μ s,	4.5 GB/s	MPI-3.0 store	

Conclusion:
No win through
MPI-3.0 shared
memory
programming

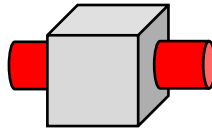
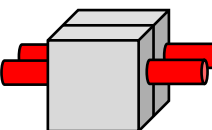
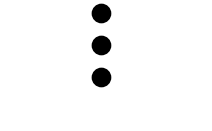
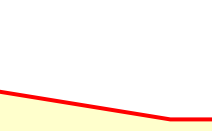
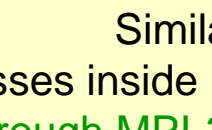
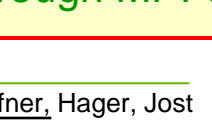

Measurements: bi-directional halo exchange in a ring with 4 SMP nodes
(with 16 and 512kB per message; bandwidth: each message is counted only once, i.e., not twice at sender and receiver) on Cray XC30 with Sandybridge @ HLRS

1 MPI process *versus* several MPI processes

(1 Intel Xeon Phi per node)

1 MPI process per Intel Xeon Phi

Intel Xeon Phi + Infiniband
beacon @ NICS

Latency	Accumulated inter-node bandwidth per	Links per Phi	Internode: Irecv + Send
15 μ s,	0.83 GB/s	1x	
26 μ s,	0.87 GB/s	2x	
25 μ s,	0.91 GB/s	4x	
23 μ s,	0.91 GB/s	8x	
24 μ s,	0.92 GB/s	16x	
21 μ s,	0.91 GB/s	30x	
51 μ s,	0.90 GB/s	60x	

4 MPI processes per Intel Phi

Latency	Accumulated inter-node bandwidth per
19 μ s,	0.54 GB/s
25 μ s,	0.52 GB/s

Additional intra-node communication with:

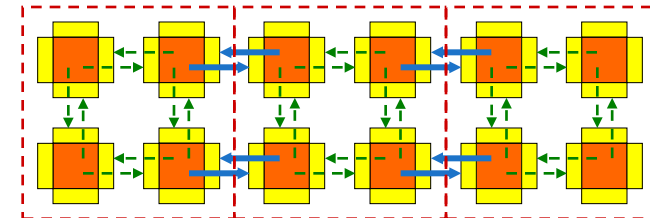
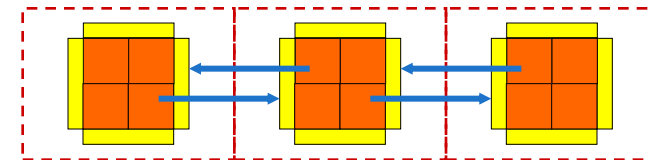
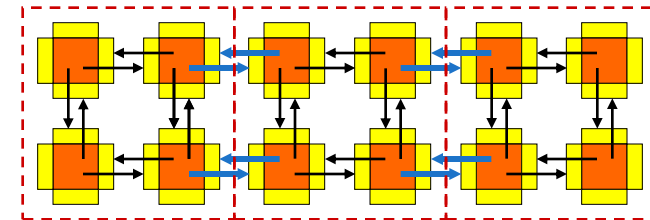
Irecv+send
MPI-3.0 store

Similar Conclusion:

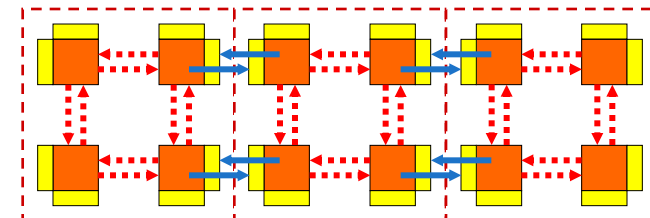
- Several MPI processes inside Phi (in a line) cause slower communication
 - No win through MPI-3.0 shared memory programming

Hybrid shared/cluster programming models

- MPI on each core (not hybrid)
 - Halos between all cores
 - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
 - Multi-threaded MPI processes
 - Halos communica. only between MPI processes
- new** • 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
- new** • 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



- MPI inter-node communication
- MPI intra-node communication
- - - Intra-node direct Fortran/C copy
- ... Intra-node direct neighbor access



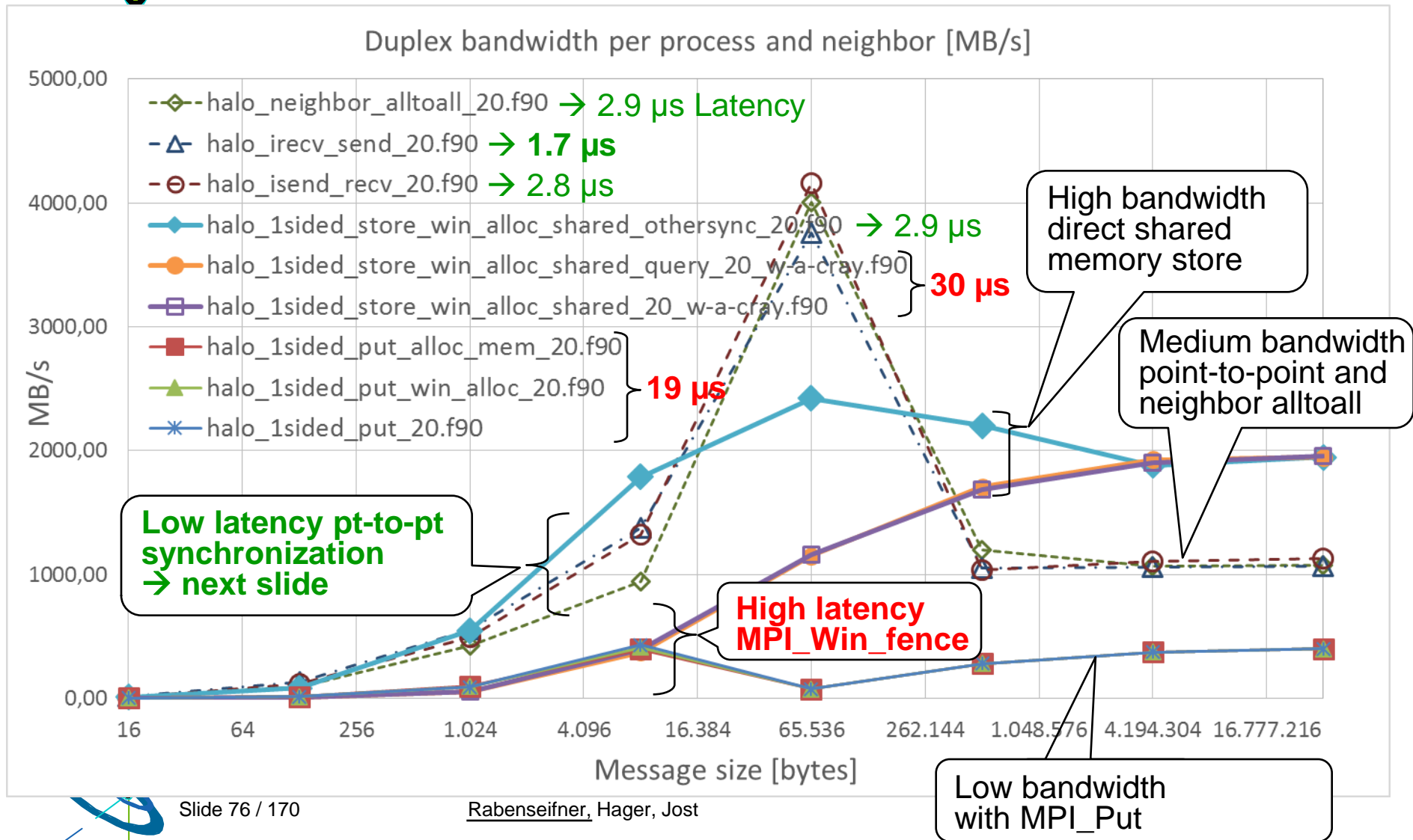
Halo Copying within SMP nodes

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)
 - **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
 - **Point-to-point synchronization for shared memory requires MPI_Win_sync**
 - **MPI-3.0 forgot to define the synchronization methods**
 - See errata coming Dec. 2014 or March 2015
 - Current proposal see
<https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/456>

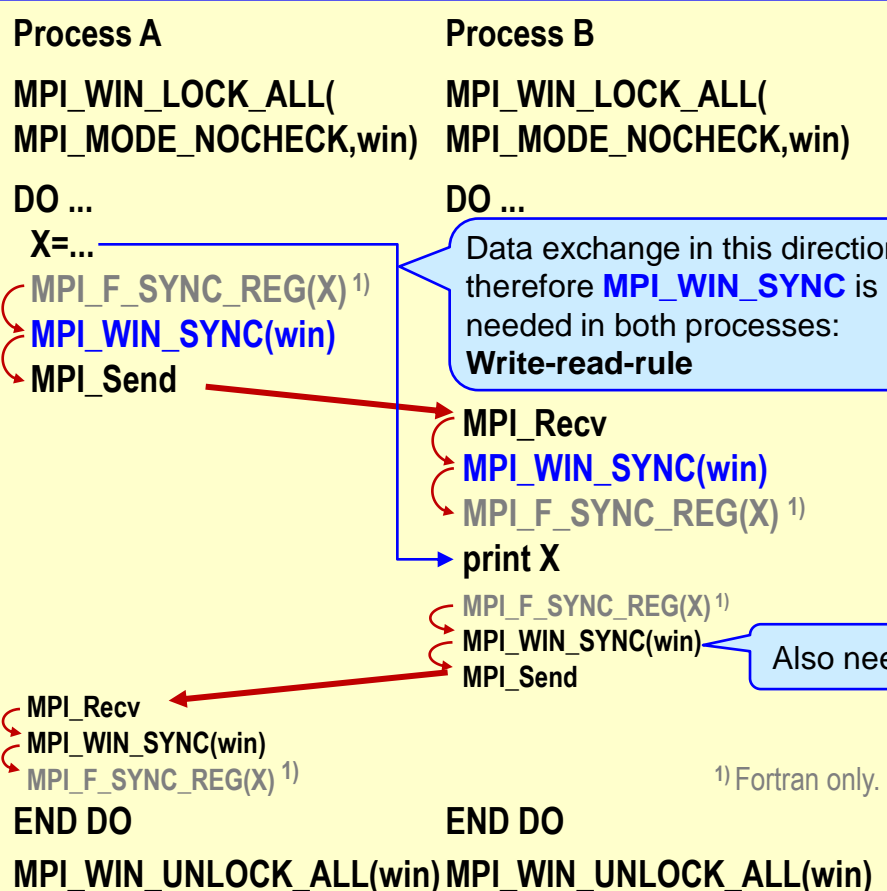


MPI Communication inside of the SMP nodes



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).



- 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**.

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)

Number of MPI processes	Latency (16 byte msg)	Bandwidth (bi-directional, 512 kB messages, per process)	Shared mem. bandwidth
4	9 μ s	0.80 GB/s	0.25 GB/s
16	11 μ s	0.75 GB/s	0.24 GB/s
30	15 μ s	0.66 GB/s	0.24 GB/s
60	29 μ s	0.50 GB/s	0.22 GB/s
120	149 μ s	0.19 GB/s	0.20 GB/s
240	745 μ s	0.05 GB/s	

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

MPI pt-to-pt substituted by MPI-3.0 shared memory store

Conclusion: Slow

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 not yet fully defined (MPI-3.0 errata is needed)
- Same problems as with all library based shared memory (e.g., pthreads)
 - Should be solved through the rules in future errata
 - (See <https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/456>)
- Does not reduce the number of MPI processes



MPI+MPI-3.0 shared mem: Conclusions

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

Programming models

- MPI + OpenMP

- General considerations slide 83
- How to compile, link, and run 90
- Case-study: The Multi-Zone NAS Parallel Benchmarks 95
- Memory placement on ccNUMA systems 104
- Topology and affinity on multicore 110
- Overlapping communication and computation 124
- Main advantages, disadvantages, conclusions 135

Hybrid MPI+OpenMP Masteronly Style

Masteronly

MPI only outside
of parallel regions

Advantages

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

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

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

Major Problems

- All other threads are sleeping while master thread communicates!
- Which inter-node bandwidth?
- MPI-lib must support at least MPI_THREAD_FUNNELED

MPI rules with OpenMP /

Automatic SMP-parallelization

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

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

- REQUIRED values (increasing order):

- **MPI_THREAD_SINGLE:** Only one thread will execute
- **THREAD_MASTERONLY:** MPI processes may be multi-threaded, but only master thread will make MPI-calls AND only while other threads are sleeping
- **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!

```

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

```

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

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!

skipped

... the barrier is necessary – example with MPI_Recv

```
!$OMP PARALLEL
!$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
```

```
#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)

Pure MPI

Additional intra-node communication with:

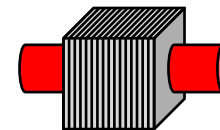
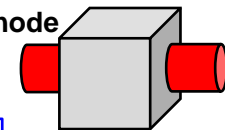
Latency Accumulated inter-node bandwidth per node

Internode: Irecv + Send

Latency Accumulated inter-node bandwidth per node

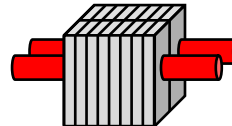
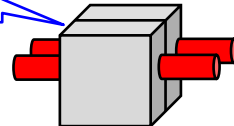
MPI processes within an SMP node

4.1 μ s, **6.8 GB/s**



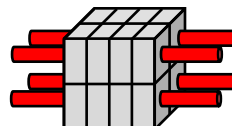
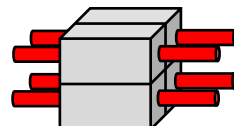
2.9 μ s, **4.4 GB/s** Irecv+send
3.4 μ s, **4.4 GB/s** MPI-3.0 store

4.1 μ s, **7.1 GB/s**



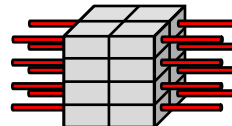
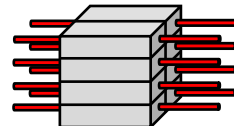
3.0 μ s, **4.5 GB/s** Irecv+send
3.0 μ s, **4.6 GB/s** MPI-3.0 store

4.1 μ s, **5.2 GB/s**



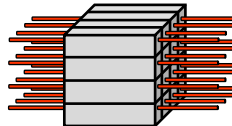
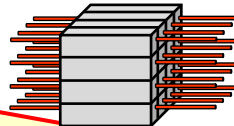
3.3 μ s, **4.4 GB/s** Irecv+send
3.5 μ s, **4.4 GB/s** MPI-3.0 store

4.4 μ s, **4.7 GB/s**



5.2 μ s, **4.3 GB/s** Irecv+send
5.2 μ s, **4.4 GB/s** MPI-3.0 store

10.2 μ s, **4.2 GB/s**



10.3 μ s, **4.5 GB/s** Irecv+send
10.1 μ s, **4.5 GB/s** MPI-3.0 store

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

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

- **MPI & OpenMP**

- Simple static load-balancing on MPI level, dynamic or guided on OpenMP level
- } **medium quality cheap implementation**

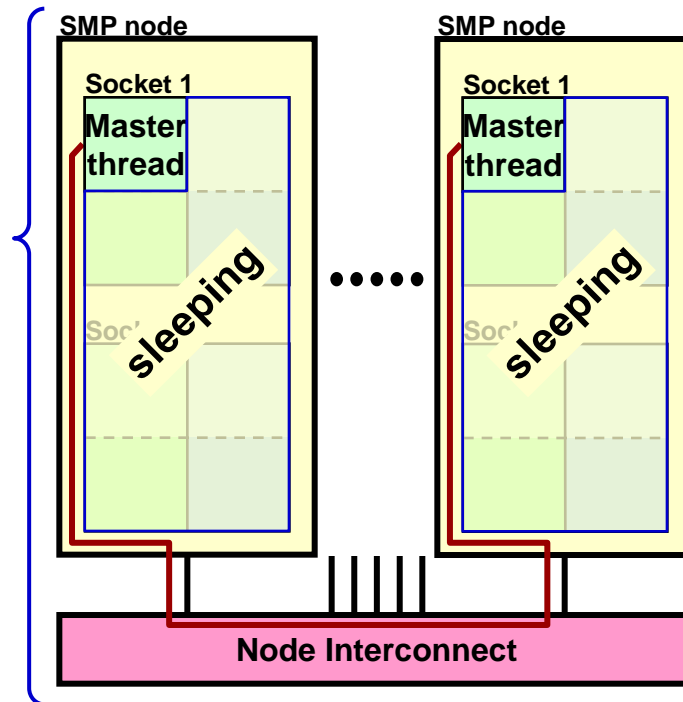


Sleeping threads with Masteronly

MPI only outside of
parallel regions

```
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
```



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 → **next slides**

Programming models

- MPI + OpenMP

How to compile, link, and run



How to compile, link and run

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



Examples for compilation and execution

- **Cray XE6** (4 NUMA domains w/ 8 cores each):
 - `ftn -h omp ...`
 - `export OMP_NUM_THREADS=8`
 - `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 “**-npnode #**” options – does not require messing around with nodefiles



Thread support within OpenMPI

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

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

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

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

- This (additionally) turns on:
 - Progress threads to asynchronously transfer/receive data per network BTL.
- Additional Feature:
 - Compiling **with** debugging support, but **without** threads will check for recursive locking

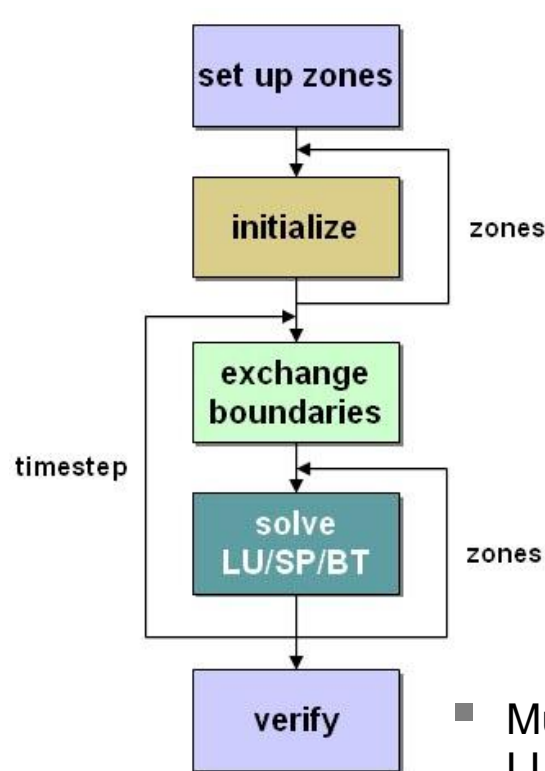
Programming models

- MPI + OpenMP

Case-study:
The Multi-Zone NAS Parallel Benchmarks



The Multi-Zone NAS Parallel Benchmarks



	MPI/Open MP	Seq	Nested OpenMP
Time step	sequential	sequential	sequential
inter-zones	MPI Processes	direct access	OpenMP
exchange boundaries	Call MPI	direct	OpenMP
intra-zones	OpenMP	sequential	OpenMP

- 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



MPI/OpenMP BT-MZ

```

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
!$OMP DO
do j = 2, ny-1
do i = 2, nx-1
do m = 1, 5
u(m,i,j,k)=
dt*rsd(m,i,j,k-1)
end do
end do
end do
!$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 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

Socket 0: (1 3 5 7 9 11)

Socket 1: (0 2 4 6 8 10)

Careful!
 Numbering scheme of
 cores is system dependent



Pitfall (2): Cause remote memory access

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

Pinning A:

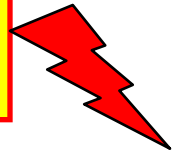
```
if [ $localrank == 0 ]; then
exec numactl --physcpubind=0,1,2,3,4,5 -m 0 $*
elif [ $localrank == 1 ]; then
exec numactl --physcpubind=6,7,8,9,10,11 -m 1 $*
fi
```

Running 128 MPI Procs, 6 threads each

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
```

600
Gflops



Half of the threads
access remote memory

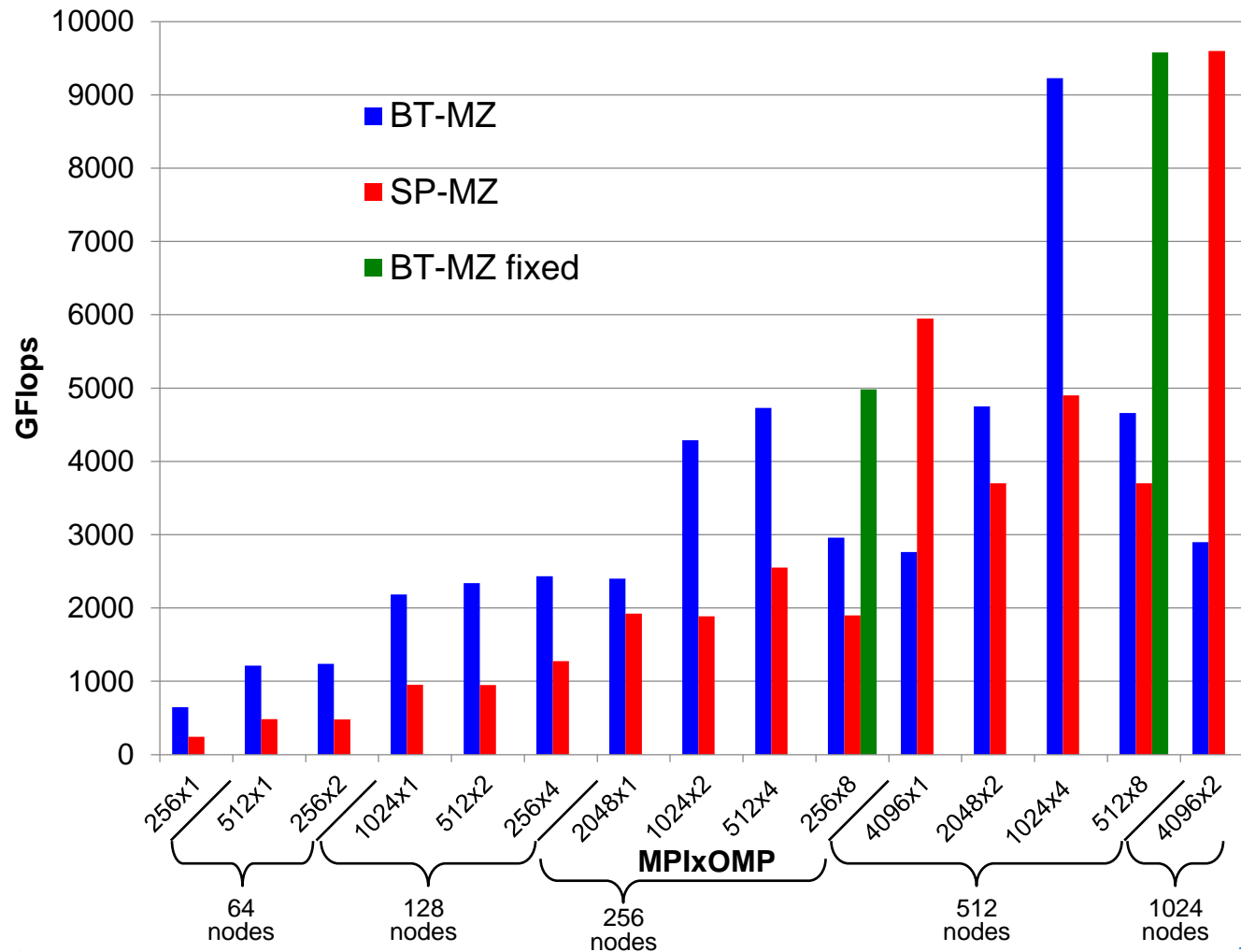
900
Gflops



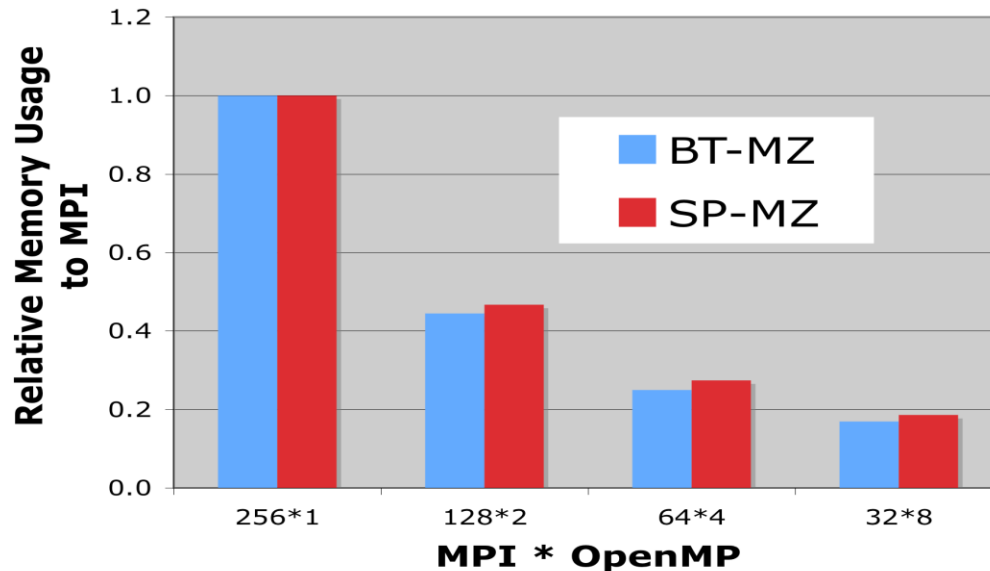
Only local memory
access



NPB-MZ Class E Scalability on Lonestar



MPI+OpenMP memory usage of NPB-MZ



Always same number of cores

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

Hongzhang Shan, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, Nicholas J. Wright:

Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms.

Proceedings, CUG 2010, Edinburgh, GB, May 24-27, 2010.

Programming models

- MPI + OpenMP

Memory placement on ccNUMA systems

Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:

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

- Except if there is not enough local memory available
- Some OSs allow to influence placement in more direct ways
 - → libnuma (Linux)

- **Caveat:** "touch" means "write", not "allocate"

- Example:

```
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!
```

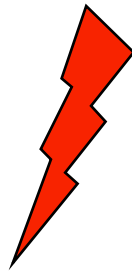
- It is sufficient to touch a single item to map the entire page
- With pure MPI (or process per ccNUMA domain): **fully automatic!**

Important

Most simple case: explicit initialization

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

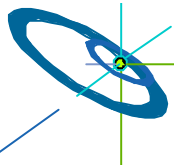
`A=0.d0`



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

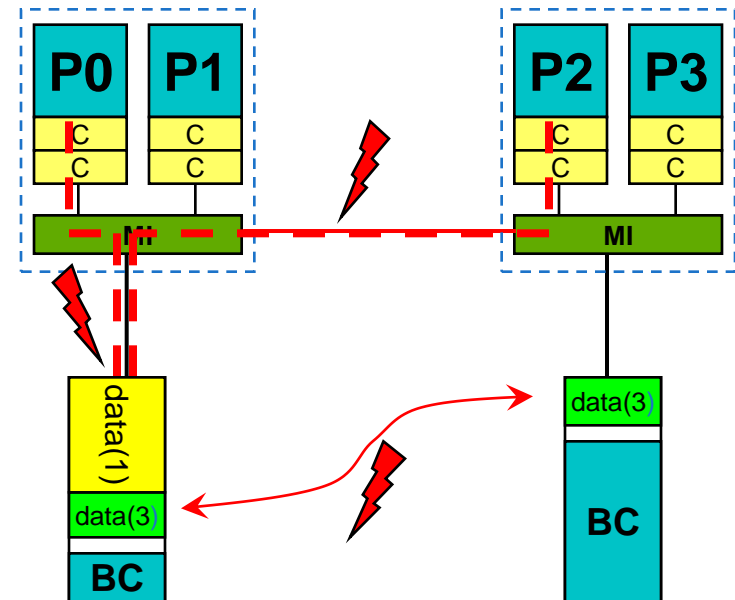
```
integer,parameter :: N=10000000
double precision A(N),B(N)
```

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



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
 - **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

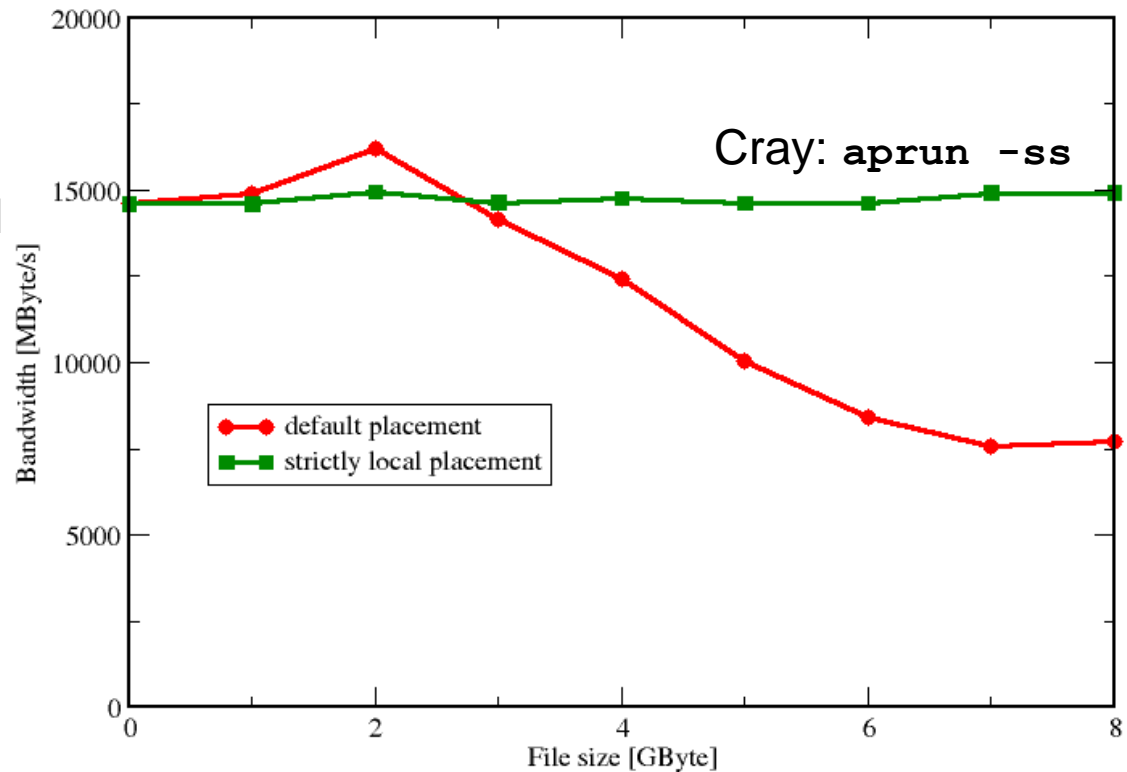
Real-world example: ccNUMA and the Linux buffer cache

Benchmark:

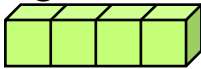
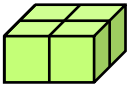
1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory installed in LD0

Result: By default, Buffer cache is given priority over local page placement

→ restrict to local domain if possible!



How to overcome ccNUMA problems

- Problems when one process spans multiple ccNUMA domains:
 - The memory is physically distributed across the ccNUMA domains.
 - First touch is needed to “bind” the data to the OpenMP threads of each socket → **otherwise loss of performance**
 - Dynamic and guided load-balancing automatically access the memory of all sockets → **loss of performance**
- Possible way out:
 - One MPI process on each socket
 - small number (>1) of MPI processes on each SMP node
 - e.g., 1-dimensional: 4 sockets in one line:
 - simple programming with structured grids
 - non-optimal communication shape  surface= $18N^2$
 - or, 3-dimensional: 2x2x1 socket:
 - less node-to-node communication due to minimal better shape  surface= $16N^2$
 - but rank re-numbering is needed



1) Provided that the application has a 20% communication footprint.

Programming models

- MPI + OpenMP

Topology and affinity on multicore



The OpenMP-parallel vector triad benchmark

Visualizing OpenMP overhead

- OpenMP work sharing in the benchmark loop

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

```
allocate (A(1:N),B(1:N),C(1:N),D(1:N))
```

```
A=1.d0; B=A; C=A; D=A
```

```
!$OMP PARALLEL private(i,j)
```

```
do j=1,NITER
```

```
!$OMP DO
```

Real work sharing

```
do i=1,N
```

```
A(i) = B(i) + C(i) * D(i)
```

```
enddo
```

```
!$OMP END DO
```

Implicit barrier

```
if(.something.that.is.never.true.) then
```

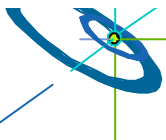
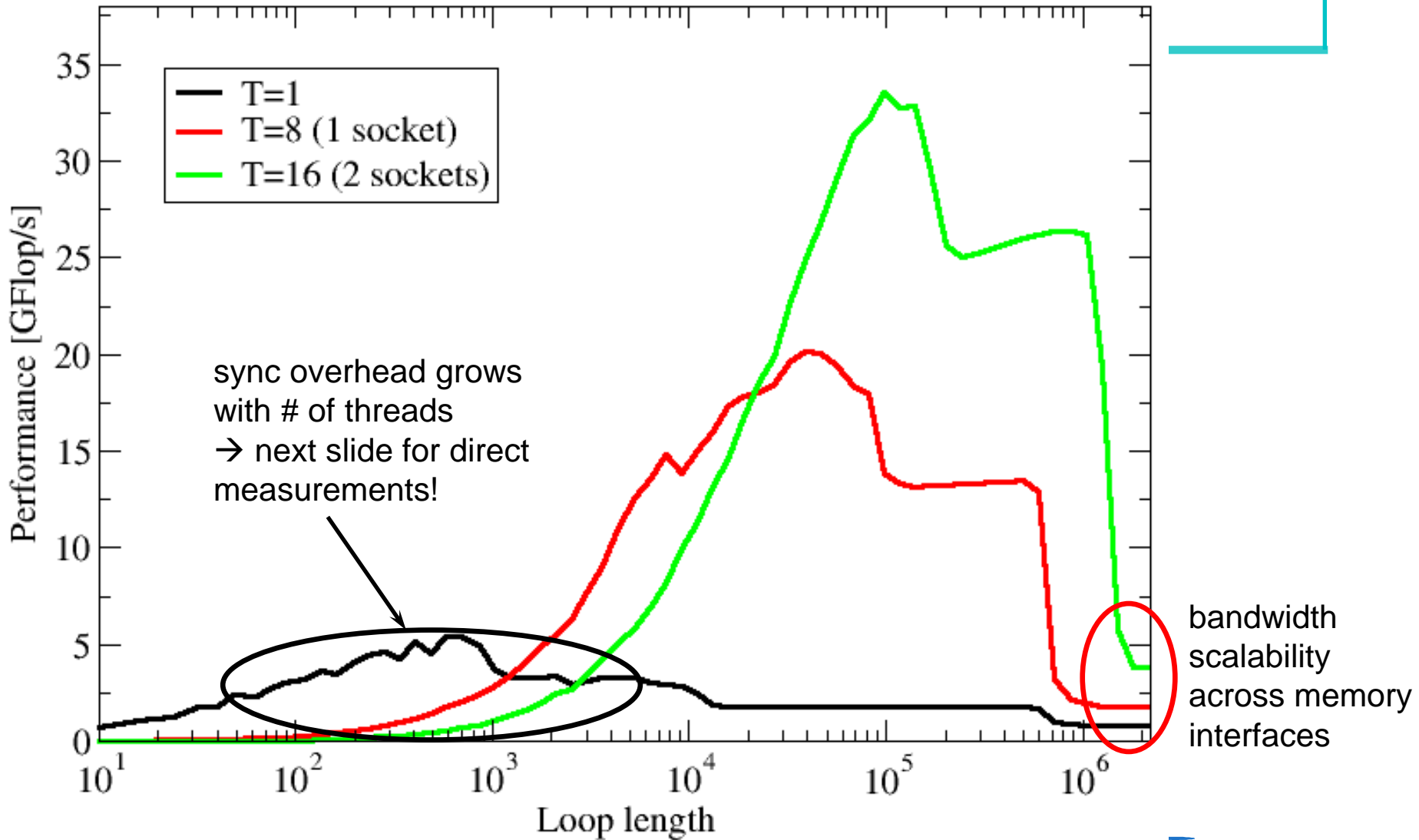
```
call dummy(A,B,C,D)
```

```
endif
```

```
enddo
```

```
!$OMP END PARALLEL
```

OpenMP vector triad on Sandy Bridge socket (3 GHz)



Thread synchronization overhead on SandyBridge-EP

Direct measurement of barrier overhead in CPU cycles

2 Threads	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Shared L3	384	5242	4616
SMT threads	2509	3726	3399
Other socket	1375	5959	4909

Strong topology
dependence!

Full domain	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Socket	1497	14546	14418
Node	3401	34667	29788
Node +SMT	6881	59038	58898

See also <http://blogs.fau.de/hager/archives/6883>



Thread synchronization overhead on Intel Xeon Phi

Barrier overhead in CPU cycles

	SMT1	SMT2	SMT3	SMT4
One core	n/a	1597	2825	3557
Full chip	10604	12800	15573	18490

That does not look too bad for 240 threads!

2 threads on
distinct cores: 1936

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

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

→ 7.5 x more work done on Xeon Phi per cycle

- 2.7 x higher barrier penalty (cycles) on Phi but 3x slower clock speed

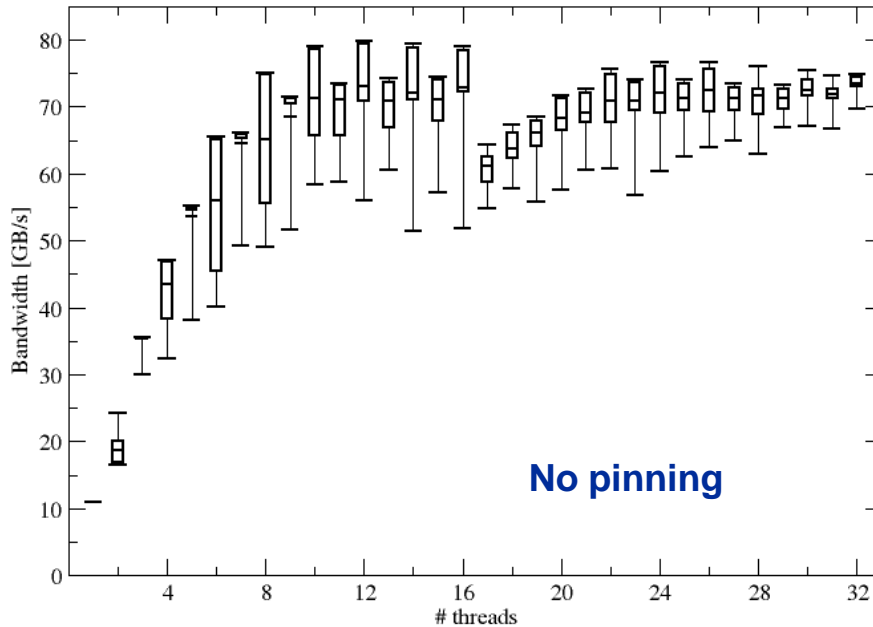
→ One barrier causes $2.7 \times 7.5 / 3 \approx 7x$ more pain ☺.

Thread/Process Affinity (“Pinning”)

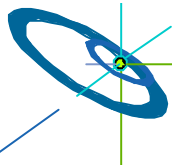
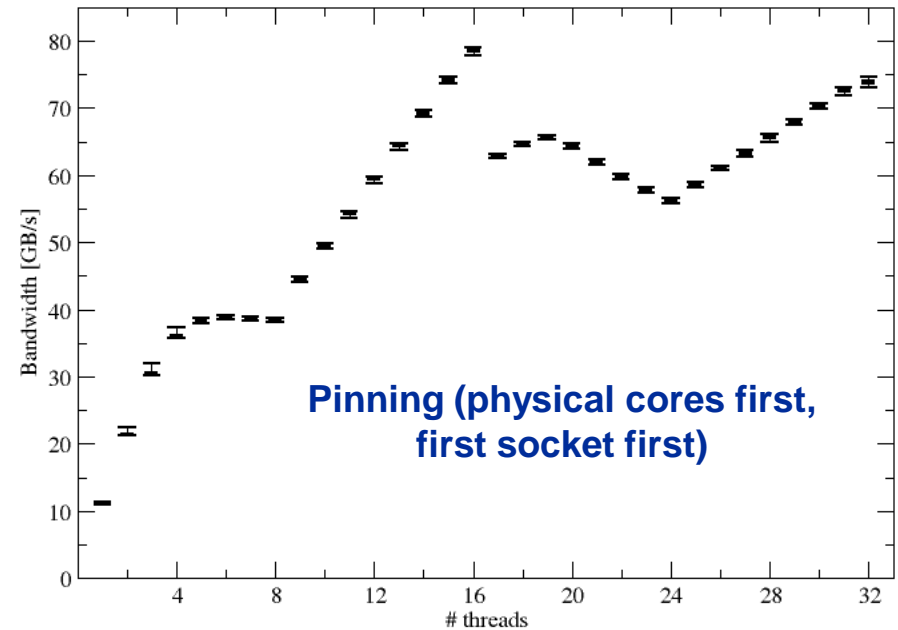
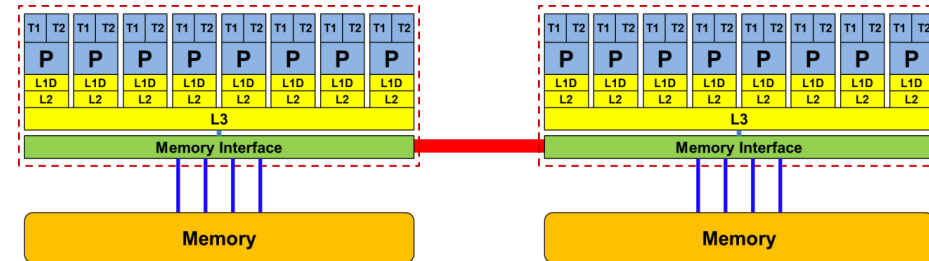
- Highly OS-dependent system calls
 - But available on all systems
 - Linux: `sched_setaffinity()`, PLPA → `hwloc`
 - Solaris: `processor_bind()`
 - Windows: `SetThreadAffinityMask()`
 - ...
- Support for “semi-automatic” pinning in all modern compilers
 - Intel, GCC, PGI,...
 - OpenMP 4.0
 - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
- Affinity awareness in MPI libraries
 - Cray MPI
 - OpenMPI
 - Intel MPI
 - ...



Anarchy vs. affinity with OpenMP STREAM



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



likwid-pin

- Binds process and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify “skip mask” (i.e., supports many different compiler/MPI combinations)
- **Replacement for taskset**
- Uses logical (contiguous) core numbering when running inside a restricted set of cores
- Supports logical core numbering inside node, socket, core
- Usage examples:
 - `env OMP_NUM_THREADS=6 likwid-pin -c 0,2,4-6 ./myApp parameters`
 - `env OMP_NUM_THREADS=6 likwid-pin -c S0:0-2@S1:0-2 ./myApp`



Likwid-pin

Example: Intel OpenMP

- Running the STREAM benchmark with likwid-pin:

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

Main PID always pinned

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

Skip shepherd thread

[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

Pin all spawned threads in turn

[... rest of STREAM output omitted ...]

OMP_PLACES and Thread Affinity (see OpenMP-4.0 page 7 lines 29-32, p. 241-243)

A *place* consists of one or more *processors*.

Pinning on the level of *places*.

Free migration of the threads on a place between the *processors* of that place.

processor is the smallest unit to run a thread or task

- **setenv OMP_PLACES threads**

abstract_name

→ Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)

- **setenv OMP_PLACES cores**

→ Each place corresponds to the processors (one or more hardware threads) of a single core

- **setenv OMP_PLACES sockets**

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

- **setenv OMP_PLACES *abstract_name*(num_places)**

→ In general, the number of places may be explicitly defined

<lower-bound>:<number of entries>[:<stride>]

- 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}, ... {29},{30},{31}" or
```

```
setenv OMP_PLACES threads (example with P=32 places)
```

- `setenv OMP_NUM_THREADS "8,2,2"`

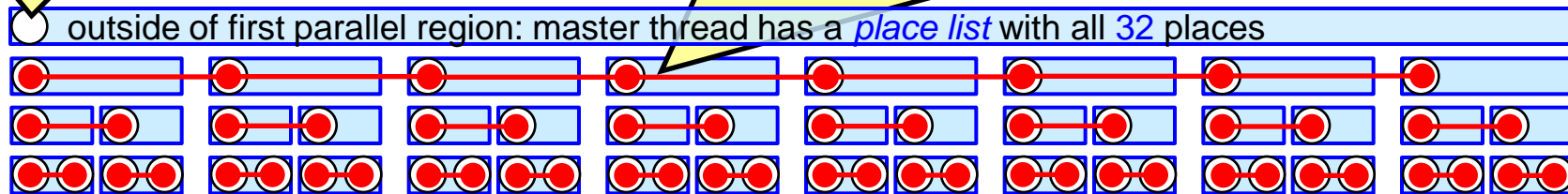
```
setenv OMP_PROC_BIND "spread,spread,close"
```

- Master thread encounters nested parallel regions:

<code>#pragma omp parallel</code>	→ uses: num_threads(8) proc_bind(spread)
<code>#pragma omp parallel</code>	→ uses: num_threads(2) proc_bind(spread)
<code>#pragma omp parallel</code>	→ uses: num_threads(2) proc_bind(close)

Only one place is used

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



spread: Sparse distribution of the 8 threads among the 32 places; partitioned place lists.

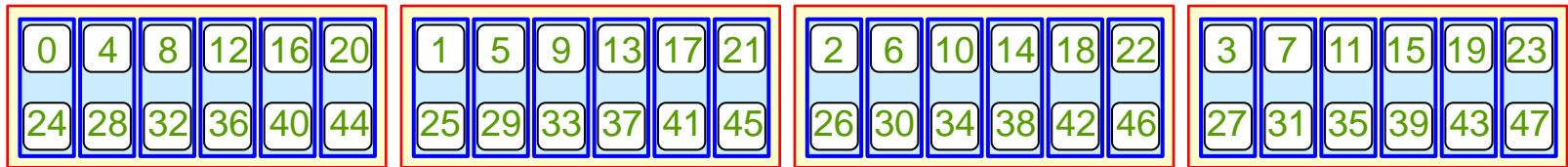
close: New threads as close as possible to the parent's place; same place lists.

master: All new threads at the same place as the parent.

Goals behind OMP_PLACES and proc_bind

Example: 4 sockets x 6 cores x 2 hyper-threads = 48 processors

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



- setenv OMP_PLACES threads** (= {0},{24},{4},{28},{8},{32},{12},{36},{16},{40},{20},{44},{1},{25}, ... , {23},{47})
 → OpenMP threads/tasks are **pinned** to hardware hyper-threads
- setenv OMP_PLACES cores** (= {0,24}, {4,28}, {8,32}, {12,36}, {16,40}, {20,44}, {1,25}, ... , {23,47})
 → OpenMP threads/tasks are **pinned** to hardware cores
 and can migrate between hyper-threads of the core
- setenv OMP_PLACES sockets** (= {0, 24, 4, 28, 8, 32, 12, 36, 16, 40, 20, 44}, {1,25,...}, {...} , {...,23,47})
 → OpenMP threads/tasks are **pinned** to hardware sockets
 and can migrate between cores & hyper-threads of the socket

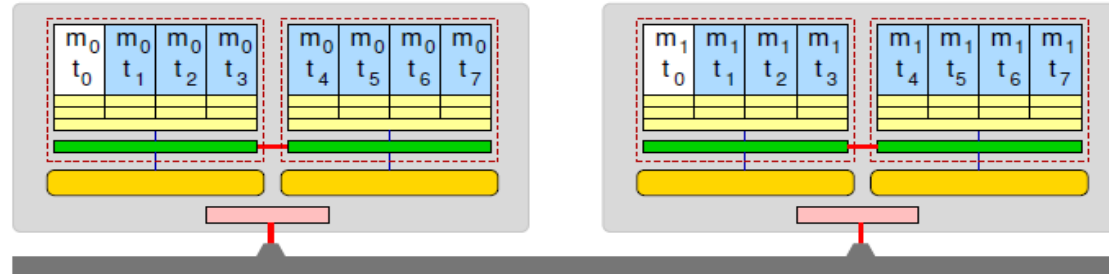
Examples should be **independent** of **vendor's numbering** & **chosen pinning**!

- Without nested parallel regions:
#pragma omp parallel num_threads(4*6) proc_bind(spread) → one thread per core
- With nested regions:
#pragma omp parallel num_threads(4) proc_bind(spread) → one thread per socket
#pragma omp parallel num_threads(6) proc_bind(spread) → one thread per core
#pragma omp parallel num_threads(2) proc_bind(close) → one thread per hyper-thread

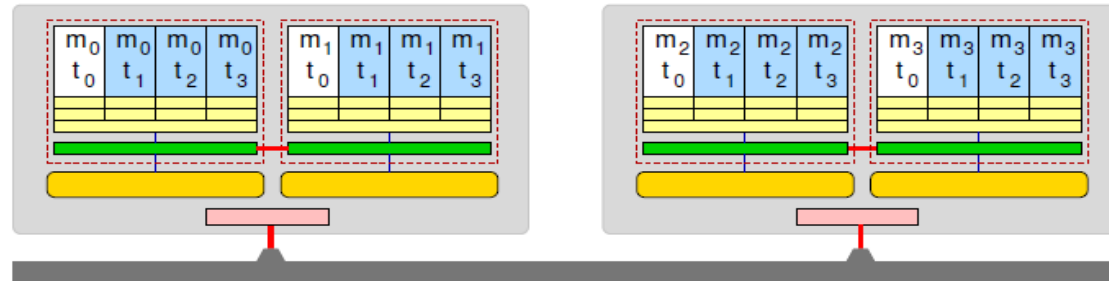
Topology (“mapping”) with MPI+OpenMP:

Lots of choices – solutions are highly system specific!

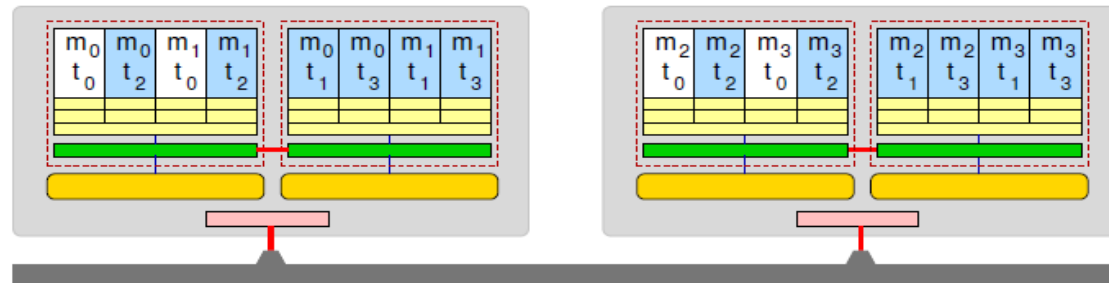
One MPI process per node



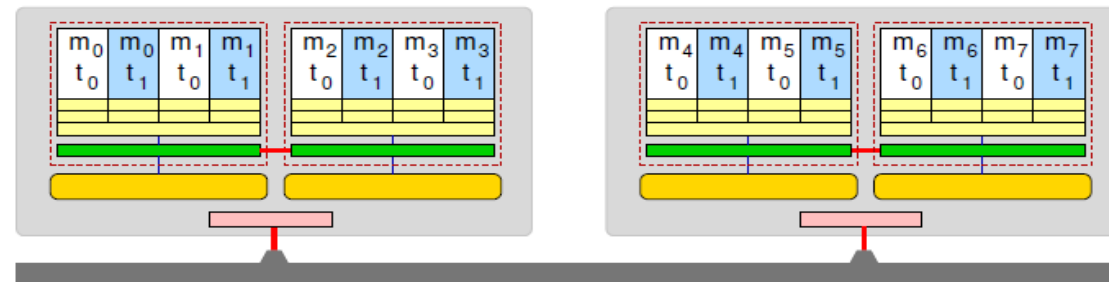
One MPI process per socket



OpenMP threads pinned “round robin” across cores in node



Two MPI processes per socket



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!
- Be aware of intranode MPI behavior
- 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-LD OpenMP processes on **ccNUMA** nodes require correct **page placement**: Observe **first touch policy**!



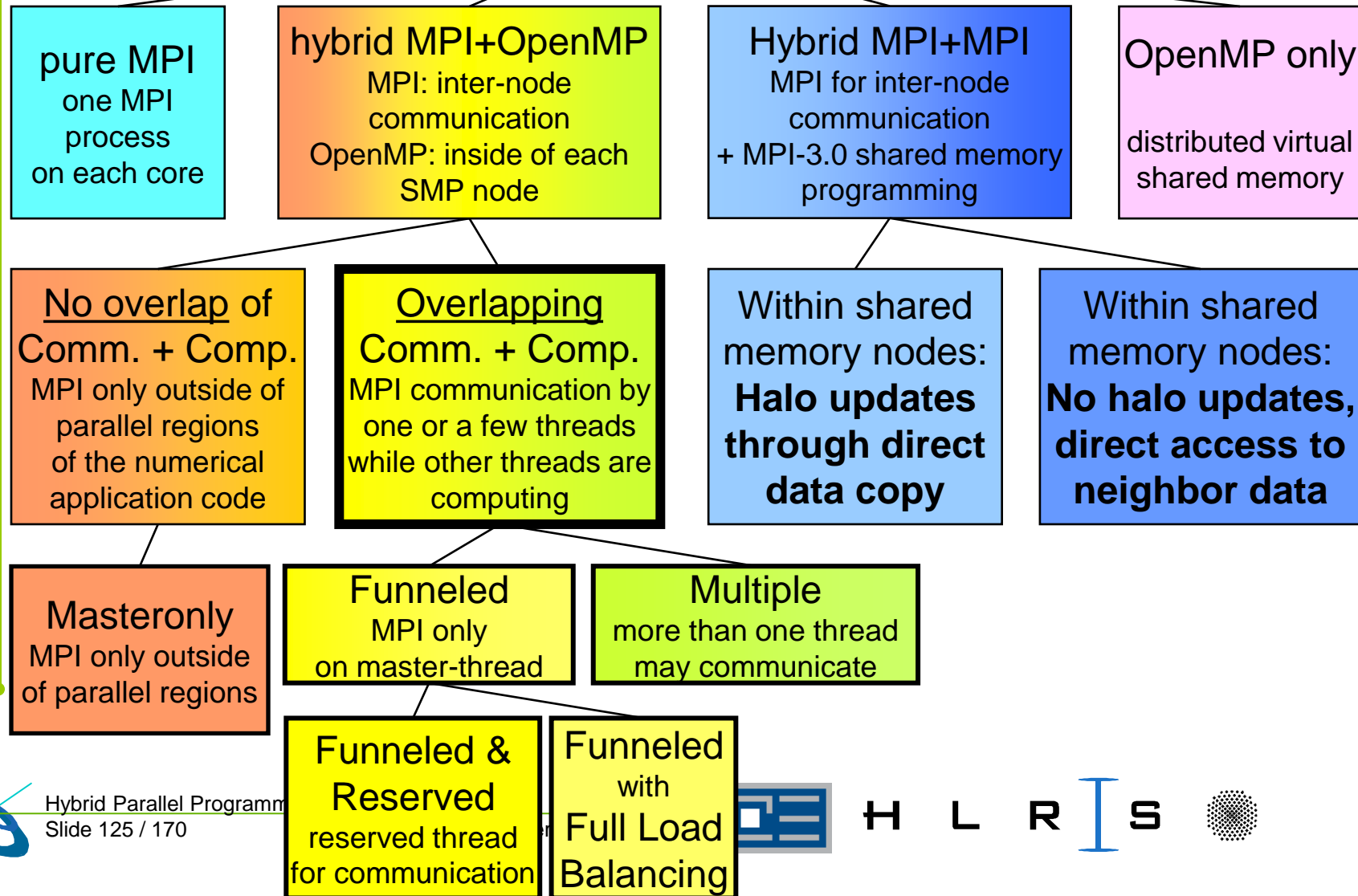
Programming models

- MPI + OpenMP

Overlapping Communication and Computation



Parallel Programming Models on Hybrid Platforms



Overlapping Communication and Computation

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

```

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)
  
```



Overlapping Communication and Computation

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

Three problems:

- the application problem:
 - one must separate application into:
 - **code that can run before the halo data is received**
 - **code that needs halo data**

→ very hard to do !!!

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

→ loss of major OpenMP support
 (see next slide)

- the load balancing problem

```

if (my_thread_rank < 1) {
    MPI_Send/Recv....
} else {
    my_range = (high-low-1) / (num_threads-1) + 1;
    my_low = low + (my_thread_rank+1)*my_range;
    my_high=high+ (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

Not yet part of the OpenMP standard

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

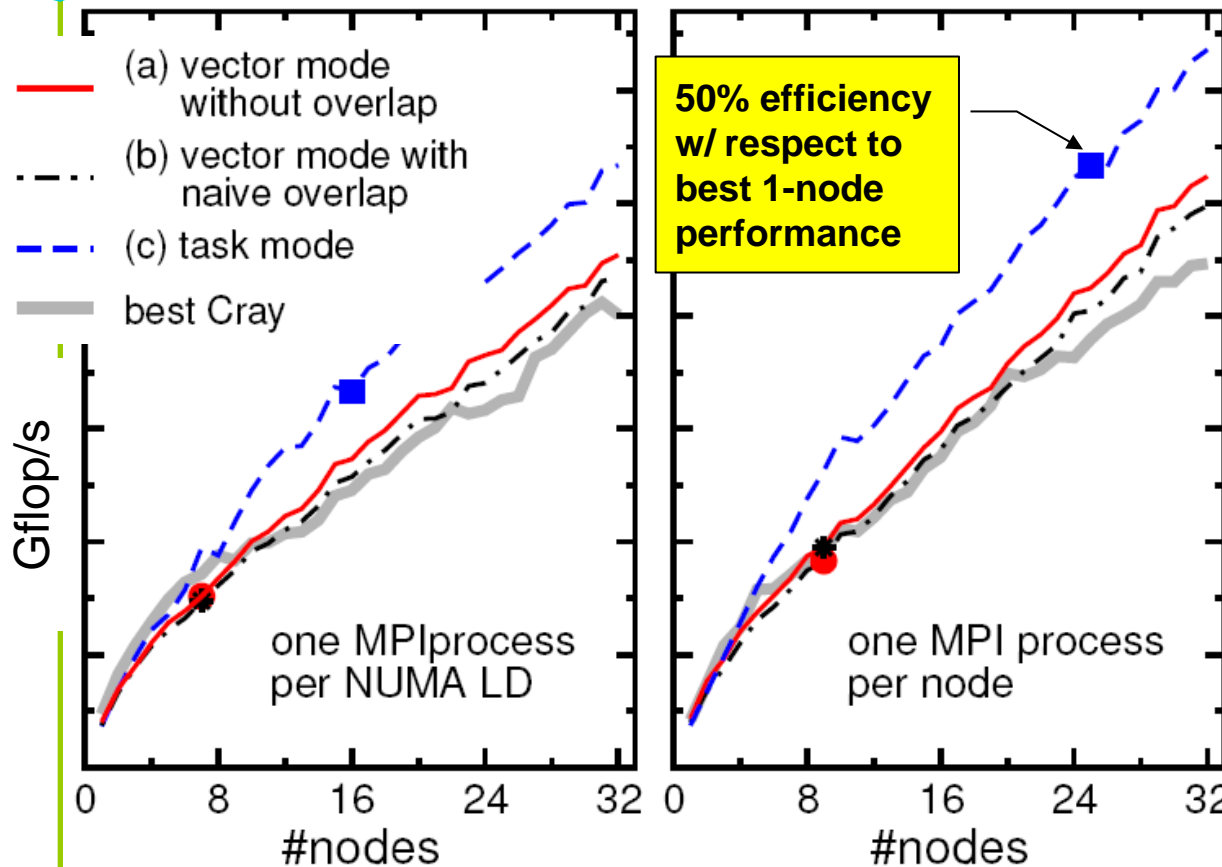
Barbara Chapman et al.:
Toward Enhancing OpenMP's
Work-Sharing Directives.
In proceedings, W.E. Nagel et
al. (Eds.): Euro-Par 2006,
LNCS 4128, pp. 645-654,
2006.

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

Workarounds today:

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

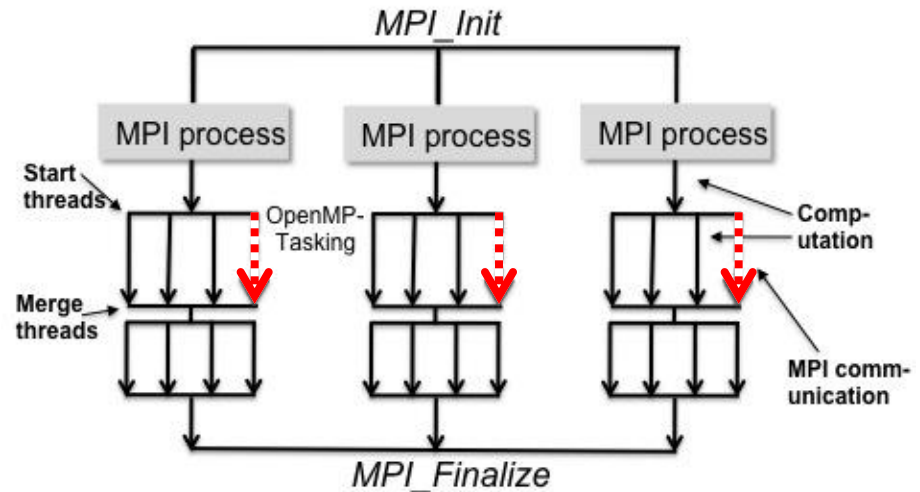
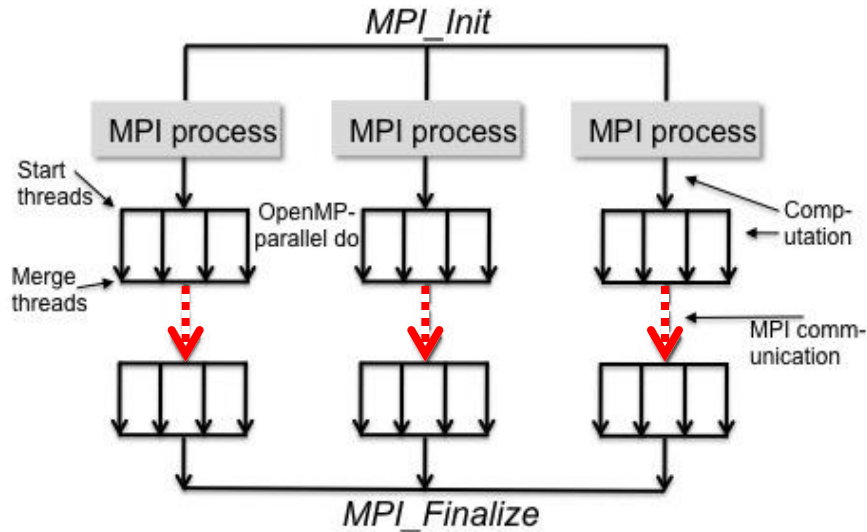
Example: sparse matrix-vector multiply (spMVM)



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

G. Schubert, H. Fehske, G. Hager, and G. Wellein: *Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems*. Parallel Processing Letters **21**(3), 339-358 (2011). DOI: [10.1142/S0129626411000254](https://doi.org/10.1142/S0129626411000254)

Overlapping: Using OpenMP tasks



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

Alice Koniges et al.:
Application Acceleration on Current and Future Cray Platforms.
Proceedings, CUG 2010, Edinburgh, GB, May 24-27, 2010.

Slides, courtesy of Alice Koniges, NERSC, LBNL



skipped

Case study: Communication and Computation in Gyrokinetic Tokamak Simulation (GTS) shift routine

INDEPENDENT

```
do iterations=1,N
!compute particles to be shifted
!$omp parallel do
  shift_p=particles_to_shift(p_array);

!communicate amount of shifted
! particles and return if equal to 0
  shift p=x+y
  MPI_ALLREDUCE(shift_p, sum_shift_p)
  if(sum_shift_p==0) { return; }

!pack particle to move right and left
!$omp parallel do
  do m=1,x
    sendright(m)=p_array(f(m));
  enddo
!$omp parallel do
  do n=1,y
    sendleft(n)=p_array(f(n));
  enddo
```

```
1  !reorder remaining particles: fill holes
   fill_hole(p_array);
3  !send number of particles to move right
   MPI_SENDRECV(x, length=2,...);
5  !send to right and receive from left
   MPI_SENDRECV(sendright, length=g(x),...);
7  !send number of particles to move left
   MPI_SENDRECV(y, length=2,...);
9  !send to left and receive from right
   MPI_SENDRECV(sendleft, length=g(y),...);
11 !adding shifted particles from right
   !$omp parallel do
   do m=1,x
     p_array(h(m))=sendright(m);
   enddo
13
15 !adding shifted particles from left
   !$omp parallel do
   do n=1,y
     p_array(h(n))=sendleft(n);
   enddo
17
19
21 }
```

INDEPENDENT

SEMI-INDEPENDENT

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**.

Slides, courtesy of Alice Koniges, NERSC, LBNL

skipped

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

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

```
2    !pack particle to move left
3    do n=1,y-stride, stride
4        !$omp task
5        do nn=0, stride-1, 1
6            sendleft(n+nn)=p_array(f(n+nn));
7        enddo
8        !$omp end task
9    enddo
10   !$omp task
11   do n=n,y
12       sendleft(n)=p_array(f(n));
13   enddo
14   !$omp end task
15   MPI_ALLREDUCE(shift_p, sum_shift_p);
16   !$omp end master
17   !$omp end parallel
18   if(sum_shift_p==0) { return; }
```

Overlapping MPI_Allreduce with particle work

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

Slides, courtesy of Alice Koniges, NERSC, LBNL



skipped

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
}
```

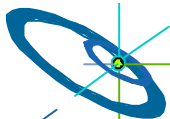
Overlapping particle reordering

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

```
1 !$omp parallel
2 !$omp master
3 !adding shifted particles from right
4 do m=1,x-stride , stride
5     !$omp task
6     do mm=0,stride-1,1
7         p_array ( h ( m ) ) = sendright ( m );
8     enddo
9     !$omp end task
10 enddo
11 !$omp task
12 do m=m,x
13     p_array ( h ( m ) ) = sendright ( m );
14 enddo
15 !$omp end task
16 MPI_SENDRECV ( sendleft , length = g ( y ) , .. );
17 !$omp end master
18 !$omp end parallel
20
21 !adding shifted particles from left
22 !$omp parallel do
23 do n=1,y
24     p_array ( h ( n ) ) = sendleft ( n );
25 enddo
```

Overlapping remaining MPI_Sendrecv

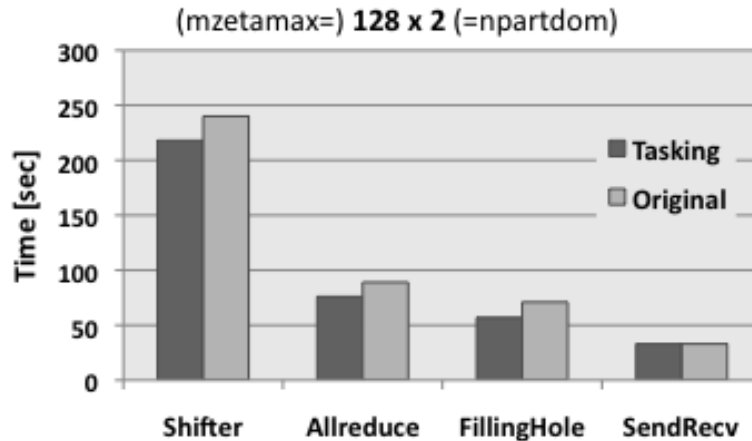
Slides, courtesy of Alice Koniges, NERSC, LBNL



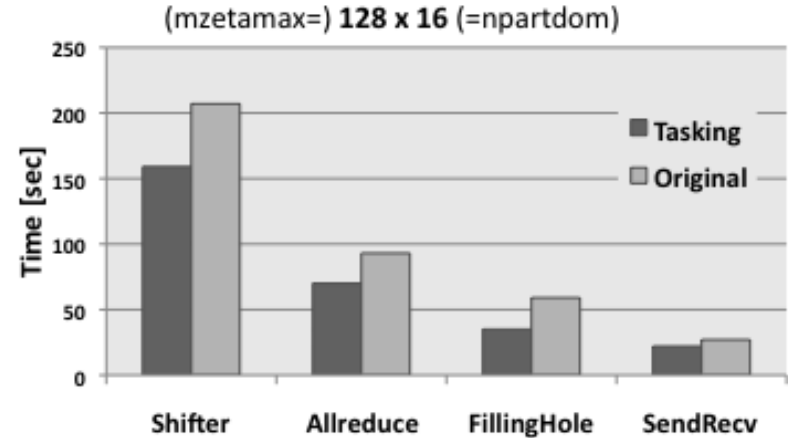
skipped

OpenMP tasking version outperforms original shifter, especially in larger poloidal domains

256 size run



2048 size run



- 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 / multsocket 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



Programming models - MPI + Accelerator

Courtesy of Gabriele Jost



OpenMP 4.0 Support for Co-Processors

- **New concepts:**
 - **Device:** An implementation defined logical execution engine; local storage which could be shared with other devices; device could have one or more processors
- **Extension to the previous Memory Model:**
 - **Previous:** Relaxed-Consistency Shared-Memory
 - **Added in 4.0 :**
 - **Device** with local storage
 - Data movement can be explicitly indicated by compiler directives
 - **League:** Set of thread teams created by a “teams” construct
 - **Contention group:** threads within a team; OpenMP synchronization restricted to contention groups.
- **Extension to the previous Execution Model**
 - **Previous:** Fork-join of OpenMP threads
 - **Added in 4.0:**
 - Host device offloads a region for execution on a **target device**
 - Host device waits for completion of execution on the target device



OpenMP Accelerator Additions

Target data

Place objects on the device

Target

Move execution to a device

Target update

Update objects on the device or host

Declare target

Place objects on the device, eg 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

```

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 team 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]);

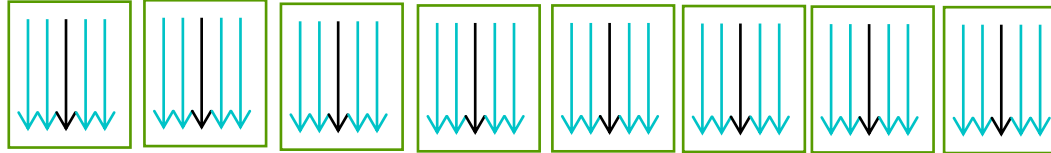
        tmp = a;  a = b;  b = tmp;
    } }

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

```

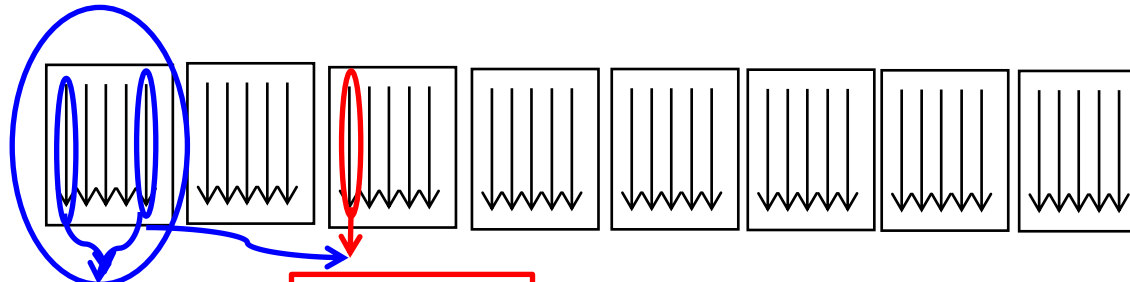
OpenMP 4.0 *Team* and *Distribute* Construct

```
#pragma omp target device(acc)
#pragma omp team num_teams(8) num_maxthreads(5)
{
```



`Stmt1;` only executed by master thread of each team

```
#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 can
synchronize

Threads cannot
synchronize

skipped

NAS Parallel Benchmark SP

```

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) + ...
          ...
        enddo
      enddo
    enddo
  !$omp end target
  !$omp target update from (rhs)

```



What is OpenACC?

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

skipped

OpenACC Simple Example

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

        tmp = a;  a = b;  b = tmp;
    } }
In main:
#pragma acc data copy (b[0:n*m],a[0:n*m])
{
    smooth( a, b, w0, w1, w2, n, m, iters );
}
```

CAPS HMPPWorkbench compiler:

acc_test.c:11: Loop 'j' was vectorized(32)
acc_test.c:9: Loop 'i' was shared among
gangs(16) and workers(8)

Mantevo miniGhost on Cray XK7

- Mantevo 1.0.1 miniGhost 1.0
 - Finite-Difference Proxy Application
 - 27 PT Stencil + Boundary Exchange of Ghost Cells
 - Implemented in Fortran;
 - MPI+OpenMP and MPI+OpenACC
 - <http://www.mantevo.org>
- Test System:
 - Located at HLRS Stuttgart,
- Test Case: Problem size 384x796x384, 10 variables, 20 time steps
- Compilation:
 - pgf90 13.4-0 -O3 -fast -fastsse
 - m -acc

```

!$acc data present ( GRID)

! Back boundary

IF ( NEIGHBORS(BACK) /= -1 ) THEN
    TIME_START_DIR = MG_TIMER ()
!$acc data present ( SEND_BUFFER_BACK )
!$acc parallel loop

DO J = 0, NY+1
DO I = 0, NX+1
    SEND_BUFFER_BACK(COUNT_SEND_BACK + J*(NX+2) + I + 1) = &
        GRID ( I, J, 1 )
    END DO
END DO

!$acc end data
#endif

...

```

Packing of boundary data

```

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

```

Unpacking of boundary data



Mantevo miniGhost: 27-PT Stencil

```

#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)

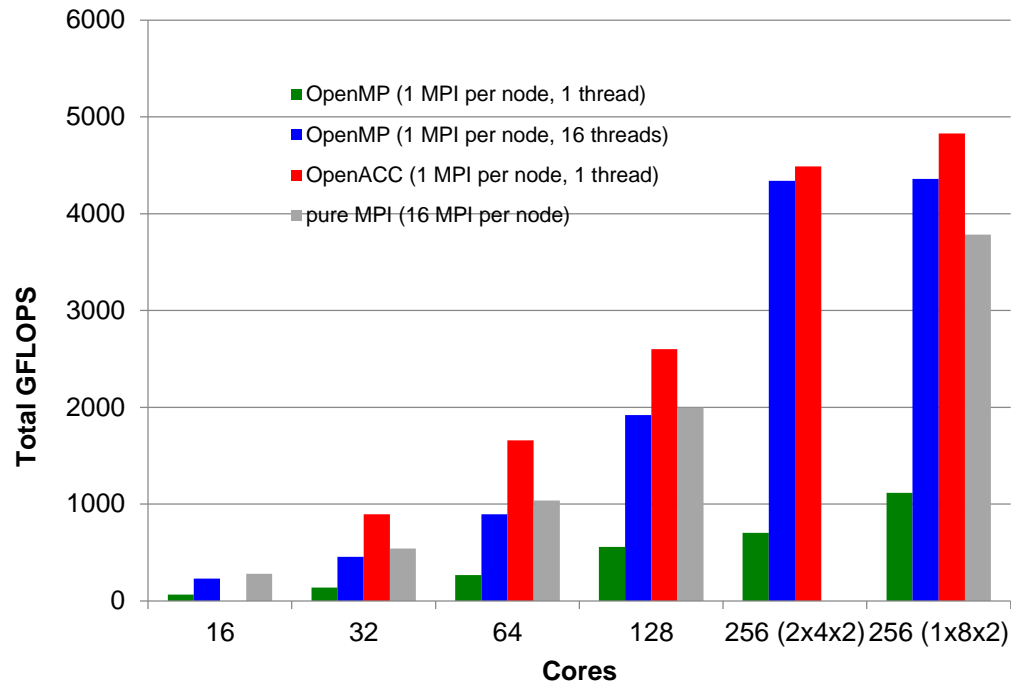
          SLICE_MINE = GRID(I-1,J-1,K) + GRID(I-1,J,K) + GRID(I-1,J+1,K) + &
            GRID(I,J-1,K) + GRID(I,J,K) + GRID(I,J+1,K) + &
            GRID(I+1,J-1,K) + GRID(I+1,J,K) + GRID(I+1,J+1,K)

          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
  
```


Scalability of miniGhost on Cray XK7



	Total Time(sec)	Comm. Time (sec)
OpenMP (16x1t)	12.1	0.4
OpenMP (16x16t)	1.9	0.16
OpenACC (16x16t)	1.17	0.34
Pure MPI (256 Ranks)	1.5	0.28

Elapsed time as reported by the application
Communication includes packing/unpacking

skipped

Profiling Information: export PGI_ACC_TIME=1

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

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

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

```
device time(us): total=12,618 max=633 min=630 avg=630
/univ_1/ws1/ws/hpcjost-ISC13_GJOST-0/miniGhost_OpenACC_1.0/MG_PACK.F
mg_pack  NVIDIA  devicenum=0
time(us): 9,615
91: kernel launched 200 times
grid: [98] block: [256]
device time(us): total=2,957 max=68 min=13 avg=14
elapsed time(us): total=11,634 max=107 min=51 avg=58
```

skipped

Profiling Information: export PGI_ACC_TIME=1

Accelerator Kernel Timing data

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

mg_stencil_3d27pt NVIDIA devicenum=0

time(us): 1,064,197

330: kernel launched 200 times

grid: [160] block: [256]

device time(us): total=1,064,197 max=5,351 min=5,299 avg=5,320

elapsed time(us): total=1,074,081 max=5,442 min=5,348 avg=5,370

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

mg_pack NVIDIA devicenum=0

time(us): 9,568

91: kernel launched 200 times

grid: [98] block: [256]

device time(us): total=2,924 max=70 min=12 avg=14

elapsed time(us): total=11,624 max=110 min=51 avg=58

195: kernel launched 200 times

grid: [162] block: [256]

device time(us): total=3,432 max=120 min=15 avg=17

elapsed time(us): total=11,385 max=160 min=53 avg=56

221: kernel launched 200 times

grid: [162] block: [256]

device time(us): total=3,212 max=19 min=15 avg=16

elapsed time(us): total

MPI+Accelerators: Main advantages

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



MPI+Accelerators: Main challenges

- Considerable implementation effort for **basic usage**, depending on complexity of the application
- **Efficient usage** of pragmas may require high implementation effort and good understanding of performance issues
- Not many compilers support accelerator pragmas (yet)



Tools

- **Topology & Affinity**
- Tools for debugging and profiling MPI+OpenMP

Tools for Thread/Process Affinity (“Pinning”)

- Likwid tools → slides in section MPI+OpenMP
 - likwid-topology prints SMP topology
 - likwid-pin binds threads to cores / HW threads
- numactl
 - Standard in Linux numatools, enables restricting movement of thread team but no individual thread pinning
- OpenMP 4.0 thread/core/socket binding



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`:

**With new Intel Inspector XE 2011:
Command line interface must be
used within mpirun / mpiexec**

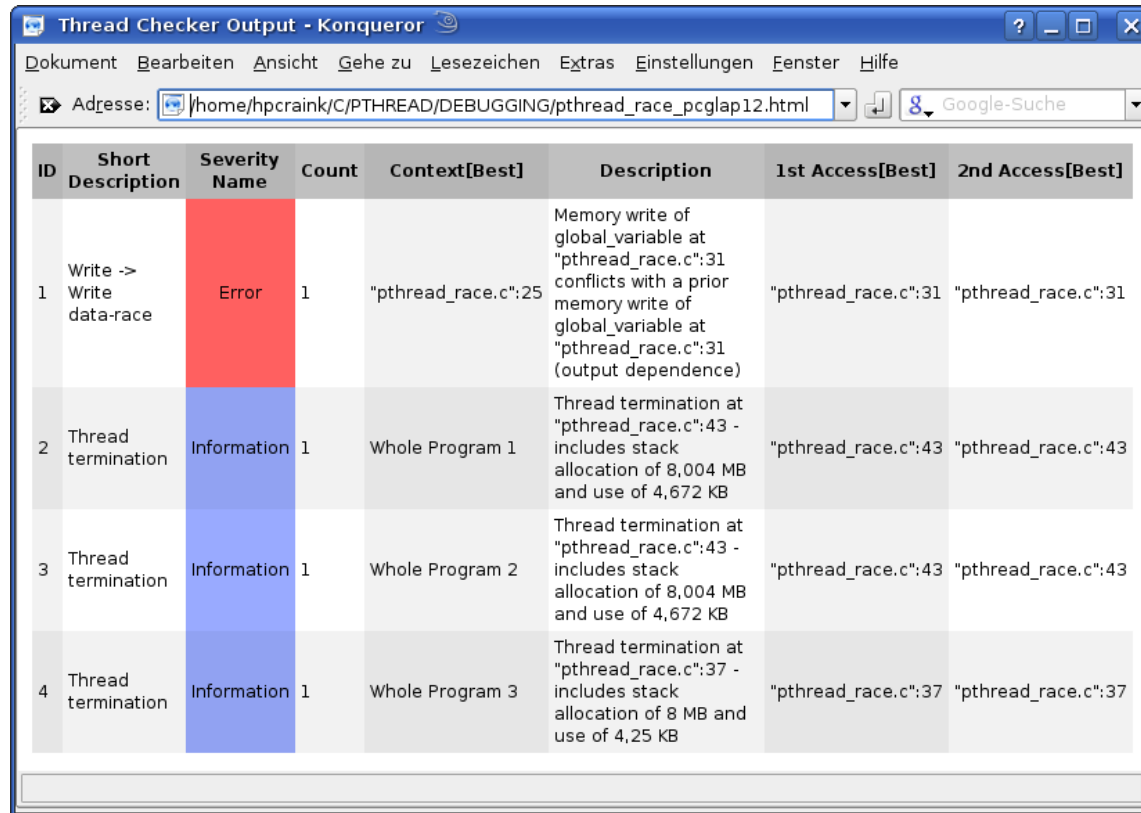
Application finished

ID	Short Description	Severity	Context	Description	1st Access	2nd Access
	Script	Info	[Best]		Access	Access
	Name	Unit			Time	Time
		Unit				
		Time				
1	Write ->	Error	1	"pthread_race.c":31 conflicts with a prior memory write of global_variable at "pthread_race.c":2	"pthread_race.c":31	"pthread_race.c":31
	Write data race		5	"pthread_race.c":31 (output dependence)		

Thread Correctness – Intel ThreadChecker 2/3

- One may output to HTML:

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



ID	Short Description	Severity Name	Count	Context[Best]	Description	1st Access[Best]	2nd Access[Best]
1	Write -> Write data-race	Error	1	"pthread_race.c":25	Memory write of global_variable at "pthread_race.c":31 conflicts with a prior memory write of global_variable at "pthread_race.c":31 (output dependence)	"pthread_race.c":31	"pthread_race.c":31
2	Thread termination	Information	1	Whole Program 1	Thread termination at "pthread_race.c":43 - includes stack allocation of 8,004 MB and use of 4,672 KB	"pthread_race.c":43	"pthread_race.c":43
3	Thread termination	Information	1	Whole Program 2	Thread termination at "pthread_race.c":43 - includes stack allocation of 8,004 MB and use of 4,672 KB	"pthread_race.c":43	"pthread_race.c":43
4	Thread termination	Information	1	Whole Program 3	Thread termination at "pthread_race.c":37 - includes stack allocation of 8 MB and use of 4,25 KB	"pthread_race.c":37	"pthread_race.c":37

skipped

Thread Correctness – Intel ThreadChecker 3/3

- If one wants to compile with threaded Open MPI (option for **IB**):

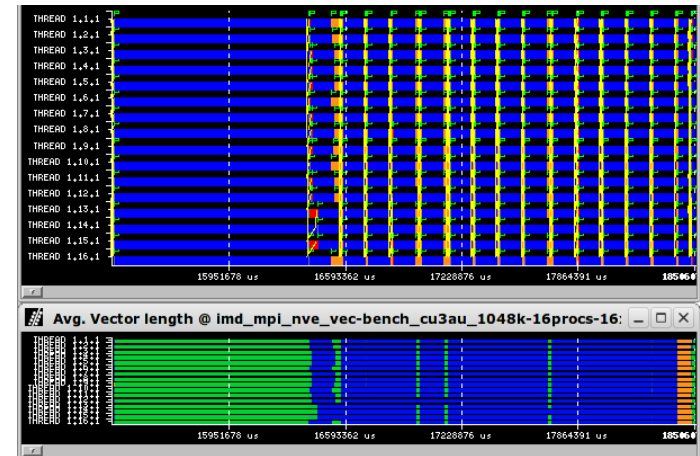
```
configure --enable-mpi-threads
          --enable-debug
          --enable-mca-no-build=memory-ptmalloc2
CC=icc F77=ifort FC=ifort
CFLAGS='-debug all -inline-debug-info tcheck'
CXXFLAGS='-debug all -inline-debug-info tcheck'
FFLAGS='-debug all -tcheck'      LDFLAGS='tcheck'
```

- Then run with:

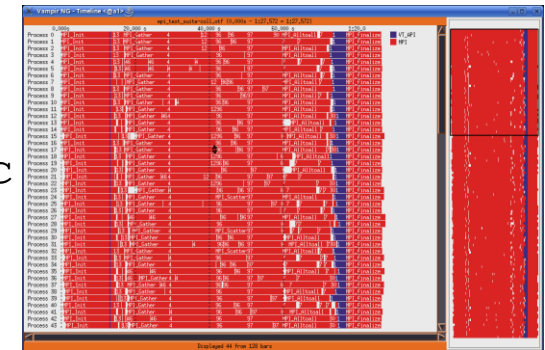
```
mpirun --mca tcp,sm,self -np 2 tcheck_cl \
      --reinstrument -u full --format html \
      --cache_dir '/tmp/my_username_$$__tc_cl_cache' \
      --report 'tc_mpi_test_suite_$$' \
      --options 'file=tc_my_executable_%H_%I, \
                pad=128, delay=2, stall=2' -- \
./my_executable my_arg1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

Performance Tools Support for Hybrid Code

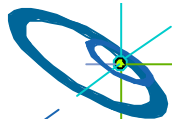
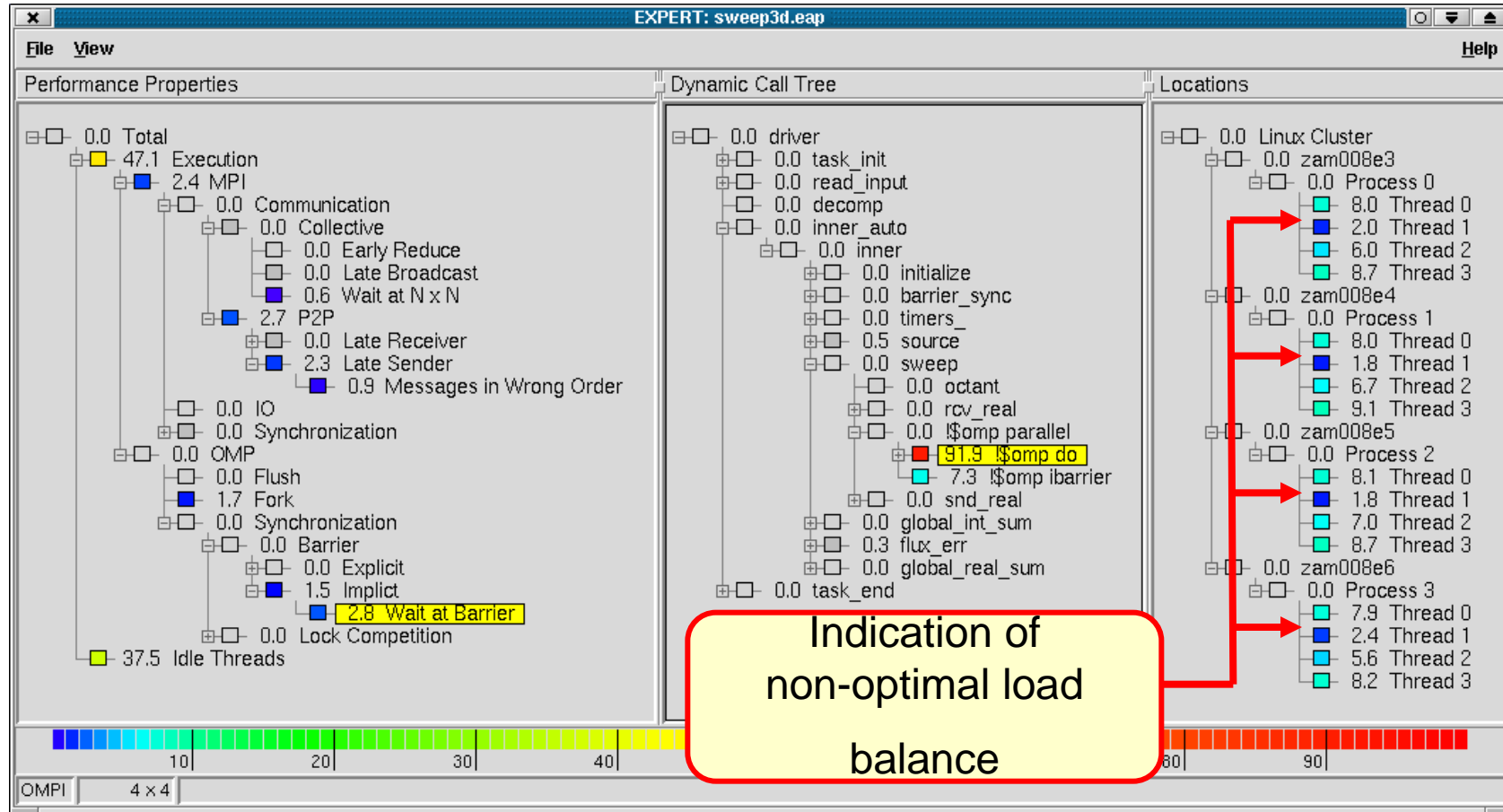
- Paraver examples have already been shown, tracing is done with linking against (closed-source) `omptrace` or `ompttrace`



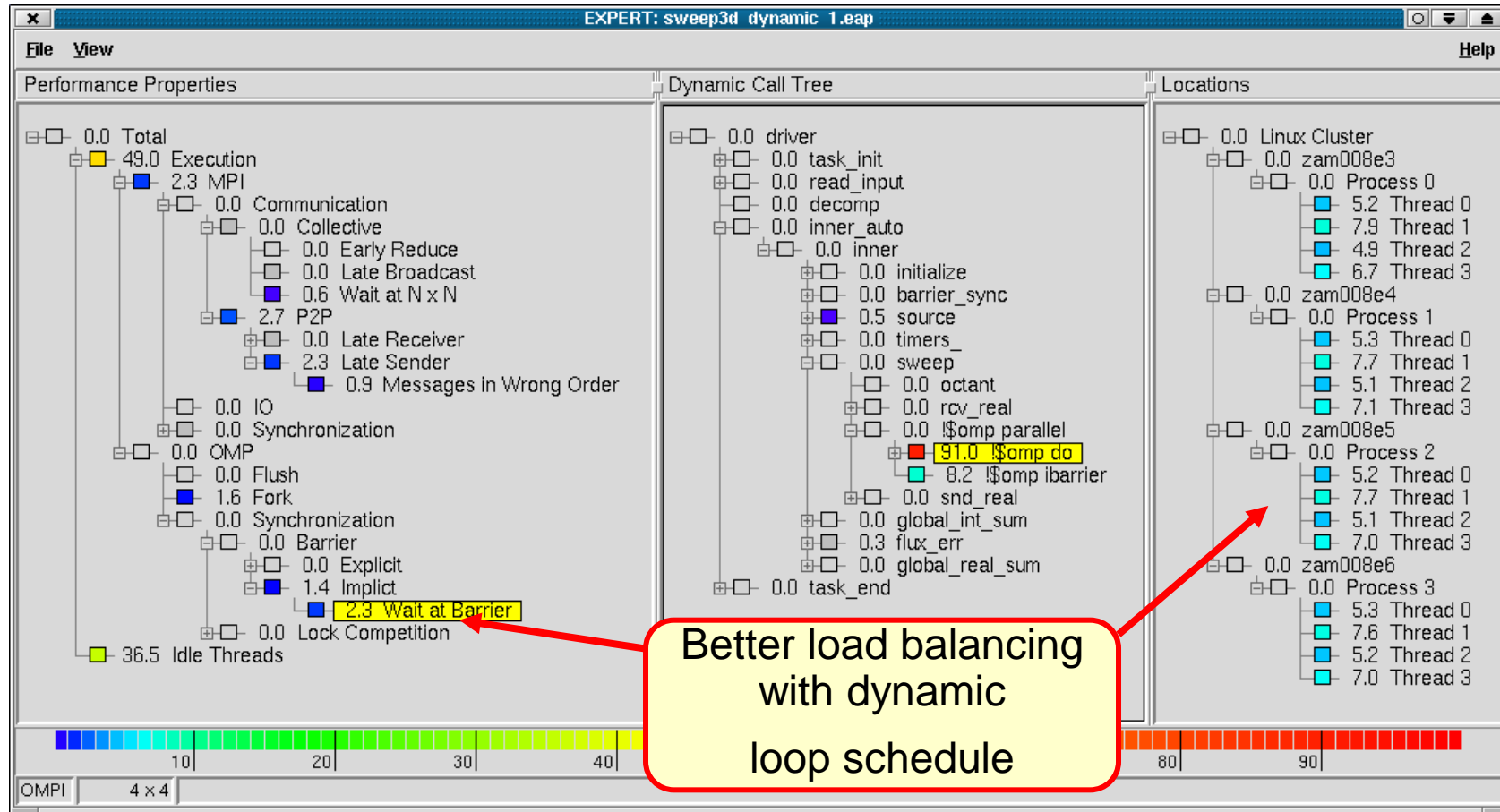
- For Vampir/Vampirtrace performance analysis:
`./configure --enable-omp`
`--enable-hyb`
`--with-mpi-dir=/opt/OpenMPI/1.3-icc`
`CC=icc F77=ifort FC=ifort`
 (Attention: does not wrap `MPI_Init_thread`!)



Scalasca – Example “Wait at Barrier”



Scalasca – Example “Wait at Barrier”, Solution



Conclusions



Major advantages of hybrid MPI+OpenMP

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

Major advantages of MPI+OpenMP:

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

Most important arguments on many-core systems, e.g., Intel Phi



Major advantages of hybrid MPI+OpenMP, continued

- **Reduced communication overhead**
 - No intra-node communication
 - Longer messages between nodes and fewer parallel links may imply better bandwidth
- **“Cheap” load-balancing methods** on OpenMP level
 - Application developer can split the load-balancing issues between course-grained MPI and fine-grained OpenMP

Disadvantages of MPI+OpenMP

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

See, e.g., the necessary **-O4** flag with `mpxlf_r` on IBM Power6 systems

MPI+OpenMP versus MPI+MPI-3.0 shared mem.

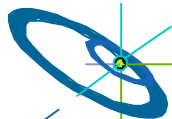
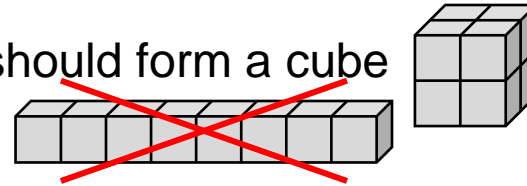
MPI+3.0 shared memory

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



Lessons for pure MPI and ccNUMA-aware hybrid MPI+OpenMP

- MPI processes on an SMP node should form a cube and not a long chain
 - Reduces inter-node communication volume
- For structured or Cartesian grids:
 - Adequate renumbering of MPI ranks and process coordinates
- For unstructured grids:
 - Two levels of domain decomposition
 - **First fine-grained on the core-level**
 - **Recombining cores to SMP-nodes**



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



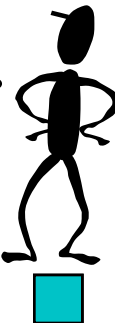
Conclusions

- 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 → 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

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

Abstract

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

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

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

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

Details. <https://fs.hlrs.de/projects/rabenseifner/publ/SC2014-hybrid.html>

Rolf Rabenseifner



Dr. 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 losing 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. 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, and in Jan. 2012, he was appointed as GCS' PATC director.



Georg Hager



Georg Hager holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). His daily work encompasses all aspects of HPC user support and training, assessment of novel system and processor architectures, and supervision of student projects and theses. Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended reading for many HPC-related courses and lectures worldwide. 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,
Hybrid Parallel Programming on HPC Platforms.
 In proceedings of the Fifth European Workshop on OpenMP, EWOMP '03, Aachen, Germany, Sept. 22-26, 2003, pp 185-194, www.compunity.org.
- 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.
- Rolf Rabenseifner and Gerhard Wellein,
Comparison of Parallel Programming Models on Clusters of SMP Nodes.
 In Modelling, Simulation and Optimization of Complex Processes (Proceedings of the International Conference on High Performance Scientific Computing, March 10-14, 2003, Hanoi, Vietnam) Bock, H.G.; Kostina, E.; Phu, H.X.; Rannacher, R. (Eds.), pp 409-426, Springer, 2004.
- Rolf Rabenseifner and Gerhard Wellein,
Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.
 In the **International Journal of High Performance Computing Applications**, Vol. 17, No. 1, 2003, pp 49-62. Sage Science Press.



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, 2474, pp 410-420, Springer, 2002.
- 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 WOMPEI 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

- Georg Hager and Gerhard Wellein:
Introduction to High Performance Computing for Scientists and Engineers.
CRC Press, ISBN 978-1439811924.
- Barbara Chapman et al.:
Toward Enhancing OpenMP's Work-Sharing Directives.
In proceedings, W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 645-654, 2006.
- Barbara Chapman, Gabriele Jost, and Ruud van der Pas:
Using OpenMP.
The MIT Press, 2008.
- Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur and Jesper Larsson Traeff:
MPI on a Million Processors.
EuroPVM/MPI 2009, Springer.
- Alice Koniges et al.: **Application Acceleration on Current and Future Cray Platforms.**
Proceedings, CUG 2010, Edinburgh, GB, May 24-27, 2010.
- H. Shan, H. Jin, K. Fuerlinger, A. Koniges, N. J. Wright: **Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms.** Proceedings, CUG 2010, Edinburgh, GB, May 24-27, 2010.



References

- J. Treibig, G. Hager and G. Wellein:
LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.
 Proc. of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010.
 Preprint: <http://arxiv.org/abs/1004.4431>
- H. Stengel:
Parallel programming on hybrid hardware: Models and applications.
 Master's thesis, Ohm University of Applied Sciences/RRZE, Nuremberg, 2010.
<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.
<http://link.springer.com/content/pdf/10.1007%2Fs00607-013-0324-2.pdf>



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
- Luis F. Romero, Eva M. Ortigosa, Sergio Romero, Emilio L. Zapata,
Nesting OpenMP and MPI in the Conjugate Gradient Method for Band Systems,
11th European PVM/MPI Users' Group Meeting in conjunction with DAPSYS'04,
Budapest, Hungary, September 19-22, 2004
- Nikolaos Drosinos and Nectarios Koziris,
Advanced Hybrid MPI/OpenMP Parallelization Paradigms for Nested Loop Algorithms onto Clusters of SMPs,
10th European PVM/MPI Users' Group Conference (EuroPVM/MPI'03), Venice, Italy,
29 Sep - 2 Oct, 2003



Further references

- Holger Brunst and Bernd Mohr,
Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with VampirNG
Proceedings for IWOMP 2005, Eugene, OR, June 2005.
- Felix Wolf and Bernd Mohr,
Automatic performance analysis of hybrid MPI/OpenMP applications
Journal of Systems Architecture, Special Issue "Evolutions in parallel distributed and network-based processing", Volume 49, Issues 10-11, Pages 421-439, November 2003.
- Felix Wolf and Bernd Mohr,
Automatic Performance Analysis of Hybrid MPI/OpenMP Applications
short version: Proceedings of the 11-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP 2003), Genoa, Italy, February 2003.
long version: Technical Report FZJ-ZAM-IB-2001-05.



Further references

- Frank Cappello and Daniel Etiemble,
MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks,
in Proc. Supercomputing'00, Dallas, TX, 2000.
<http://www.sc2000.org/techpaper/papers/pap.pap214.pdf>
- Jonathan Harris,
Extending OpenMP for NUMA Architectures,
in proceedings of the Second European Workshop on OpenMP, EWOMP 2000.
- D. S. Henty,
Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling,
in Proc. Supercomputing'00, Dallas, TX, 2000.
<http://www.sc2000.org/techpaper/papers/pap.pap154.pdf>



Further references

- Matthias Hess, Gabriele Jost, Matthias Müller, and Roland Rühle,
Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster,
in WOMPAT2002: Workshop on OpenMP Applications and Tools, Arctic Region Supercomputing Center, University of Alaska, Fairbanks, Aug. 5-7, 2002.
- John Merlin,
Distributed OpenMP: Extensions to OpenMP for SMP Clusters,
in proceedings of the Second European Workshop on OpenMP, EWOMP 2000.
- Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka,
Design of OpenMP Compiler for an SMP Cluster,
in proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden, Sep. 1999, pp 32-39.
- Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel,
Transparent Adaptive Parallelism on NOWs using OpenMP,
in proceedings of the Seventh Conference on Principles and Practice of Parallel Programming (PPoPP '99), May 1999, pp 96-106.



Further references

- Weisong Shi, Weiwu Hu, and Zhimin Tang,
Shared Virtual Memory: A Survey,
 Technical report No. 980005, Center for High Performance Computing,
 Institute of Computing Technology, Chinese Academy of Sciences, 1998,
www.ict.ac.cn/chpc/dsm/tr980005.ps.
- Lorna Smith and Mark Bull,
Development of Mixed Mode MPI / OpenMP Applications,
 in proceedings of Workshop on OpenMP Applications and Tools (WOMPAT 2000),
 San Diego, July 2000.
- Gerhard Wellein, Georg Hager, Achim Basermann, and Holger Fehske,
Fast sparse matrix-vector multiplication for TeraFlop/s computers,
 in proceedings of VECPAR'2002, 5th Int'l Conference on High Performance Computing
 and Computational Science, Porto, Portugal, June 26-28, 2002, part I, pp 57-70.
<http://vecpar.fe.up.pt/>



Further references

- Agnieszka Debudaj-Grabysz and Rolf Rabenseifner,
Load Balanced Parallel Simulated Annealing on a Cluster of SMP Nodes.
 In proceedings, W. E. Nagel, W. V. Walter, and W. Lehner (Eds.): Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Aug. 29 - Sep. 1, Dresden, Germany, LNCS 4128, Springer, 2006.
- Agnieszka Debudaj-Grabysz and Rolf Rabenseifner,
Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes.
 In Recent Advances in Parallel Virtual Machine and Message Passing Interface, Beniamino Di Martino, Dieter Kranzlmueller, and Jack Dongarra (Eds.), Proceedings of the 12th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2005, Sep. 18-21, Sorrento, Italy, LNCS 3666, pp 18-27, Springer, 2005

