Hybrid Programming in HPC – MPI+X

Claudia Blaas-Schenner[1)] claudia.blaas-schenner@tuwien.ac.at
Georg Hager[2)] georg.hager@fau.de
Rolf Rabenseifner[3)] rabenseifner@hlrs.de

[1)] VSC Research Center, TU Wien, Vienna, Austria (hands-on labs)
[2)] Erlangen National High Performance Computing Center (NHR@FAU), FAU, Germany
[3)] High Performance Computing Center (HLRS), University of Stuttgart, Germany

PTC ONLINE COURSE @ LRZ, June 22-24, 2022

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

General Outline (with slide numbers)

Motivation

Hardware and Programming Models

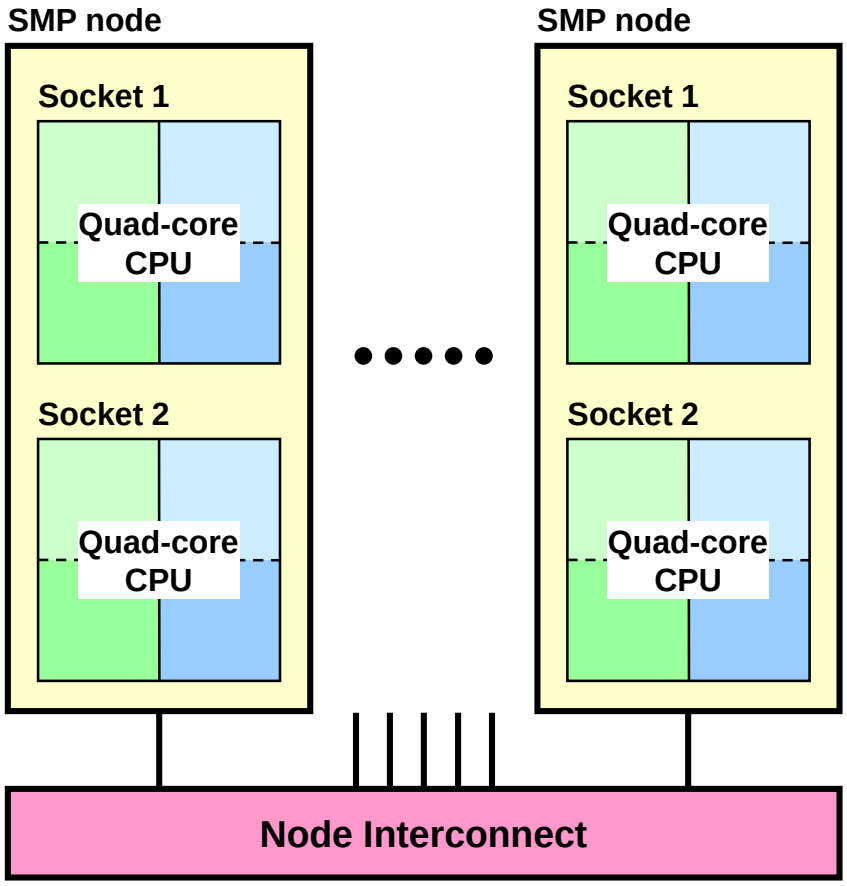
Hardware

- Cluster of
  - ccNUMA nodes with several multi-core CPUs
  - nodes with multi-core CPUs + GPU(s)
  - nodes with multi-core CPUs + FPGA(s)
  - …

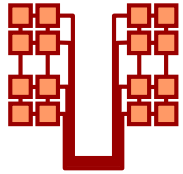
Programming models

- MPI + Threading
  - OpenMP
  - Cilk(+)
  - TBB (Threading Building Blocks)
- MPI + MPI shared memory
- MPI + Accelerator
  - OpenACC
  - OpenMP 4.0 / 4.5 accelerator support
  - CUDA
  - OpenCL
  - …
- Pure MPI communication

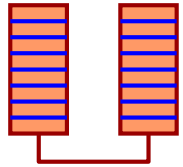
Options for running code



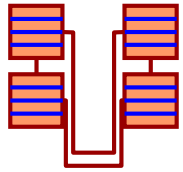
- Which programming model is fastest?
- MPI everywhere?



- Fully hybrid MPI & OpenMP?



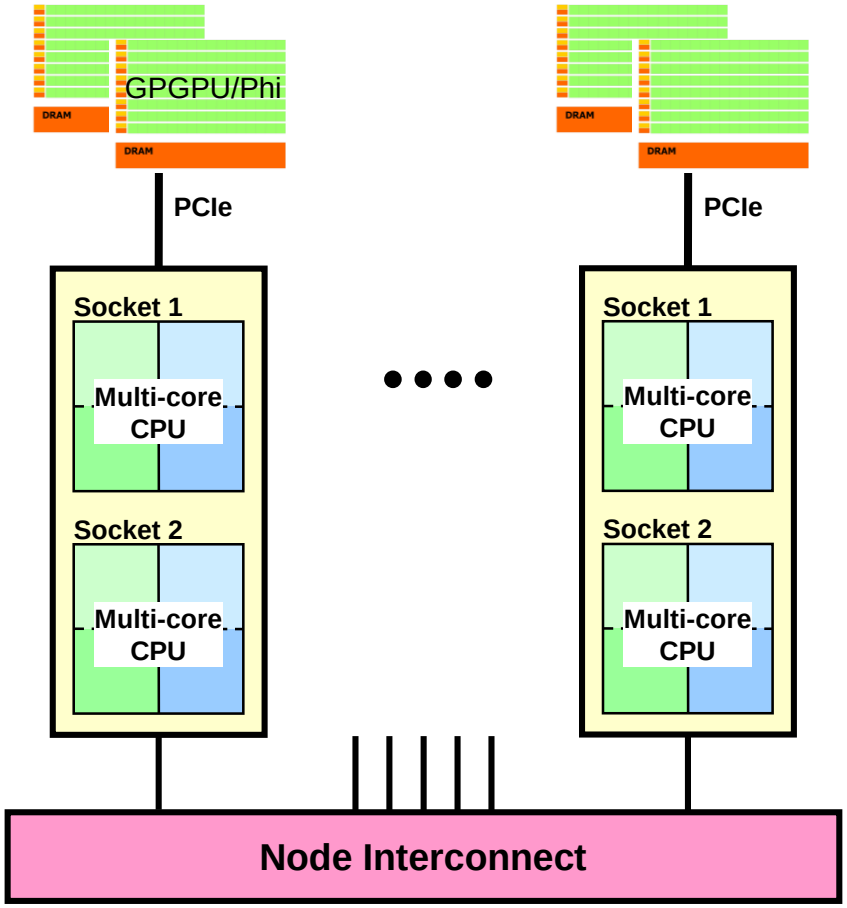
- Something between? (Mixed model)



- Often hybrid programming **slower** than pure MPI
  - Examples, Reasons, …

?

More Options



Number of options multiply if accelerators are added

- One MPI process per accelerator?
- One thread per accelerator?
- Which programing model on the accelerator?
  - OpenMP shared memory
  - MPI
  - OpenACC
  - OpenMP-4.0 / 4.5 accelerator support
  - CUDA
  - …

Splitting the Hardware Hierarchy

Hierarchical hardware

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

Hierarchical parallel programming

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

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

- **1 MPI process per shared memory node**
- **1 MPI process per CPU**
- **…**
- **…**
- **OpenMP for hyperthreading**
- **OpenMP only for vectorization**

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

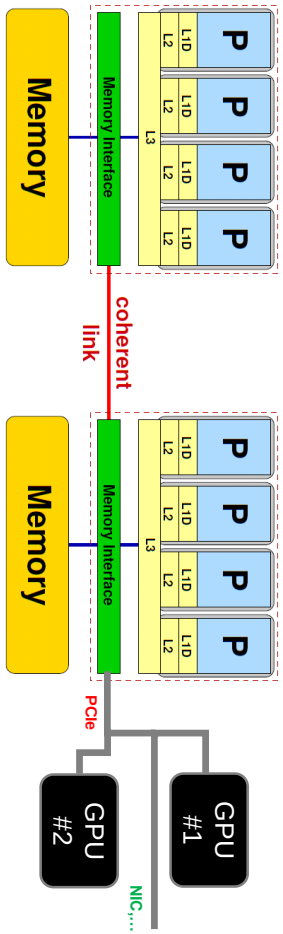
Outline

Introduction

Typical hardware bottlenecks and challenges

Hardware Bottlenecks

- Multicore cluster
  - Computation
  - Memory bandwidth
  - Inter-node communication
  - Intra-node communication (i.e., CPU-to-CPU)
  - Intra-CPU communication (i.e., core-to-core)
- Cluster with CPU+Accelerators
  - Within the accelerator
    - **Computation**
    - **Memory bandwidth**
    - **Core-to-Core communication**
  - Within the CPU and between the CPUs
    - **See above**
  - Link between CPU and accelerator



Hardware Bottlenecks

Example:

- Sparse matrix-vector-multiply with **stored matrix entries**

  → Bottleneck: memory bandwidth of each CPU

- Sparse matrix-vector-multiply with **calculated matrix entries** (many complex operations per entry)

  → Bottleneck: computational speed of each core

- Sparse matrix-vector multiply with **highly scattered matrix entries**

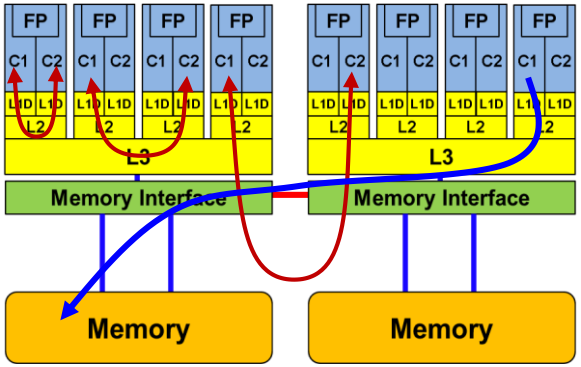
  → Bottleneck: Inter-node communication

skipped

Topology complicates matters

- Symmetric, UMA-type single-core compute nodes have vanished completely (NEC SX, Hitachi SR8k, IBM SP2)

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



Major problems with cluster of ccNUMA nodes?

MAJOR PROBLEMS

- The usual programming models do **not** really fit: **Pure MPI / Hybrid MPI+OpenMP**
- Where are major problems? – And how to address?
  - Significant differences between **(small) inter-node bandwidth** and (high) intra-node bandwidth
    - → **How to minimize inter-node communication**
      - Hybrid MPI + OpenMP
      - Pure MPI + optimized virtual topologies
  - **Replicated user data**
    - → **Waste of memory with pure MPI**
      - Hybrid MPI + OpenMP
      - Pure MPI + MPI-3 shared memory model
  - Processes and threads may **move** between cores and CPUs within ccNUMA nodes & fixed memory locations after ***"first touch"***
    - → **How to keep control?**
      - Pinning of threads and processes to the hardware
      - Explicit first touch programming with OpenMP, …
  - Are your application AND libraries **prepared for** MPI+**OpenMP?**

Remarks on Cost-Benefit Calculation

Will the effort for optimization pay off?

Remarks on Cost-Benefit Calculation

Costs

- for optimization effort
  - e.g., additional OpenMP parallelization
  - e.g., 3 person month 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

- MPI + OpenMP on multi/many-core
  + Exercise
- MPI + Accelerators
- MPI + MPI-3.0 shared memory
  + Exercise
- Pure MPI communication
  + Exercise

Programming models - MPI + OpenMP

Hybrid MPI+OpenMP Masteronly Style

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
```

Advantages

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

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 * thread_level_provided);
int MPI_Query_thread(  int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

- REQUIRED values (increasing order):
  - **MPI_THREAD_SINGLE:** **Only one thread will execute**
  - **THREAD_MASTERONLY:** **MPI processes may be multi-threaded,**
    (virtual value, **but only main[1] thread will make MPI-calls**
    not part of the standard) **AND only while other threads are sleeping**
  - **MPI_THREAD_FUNNELED:** **Only main[1] 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 or more than `required` by the application

[1] Main thread = thread that called MPI_Init_thread.
Recommendation:
Start MPI_Init_thread from OpenMP master thread → OpenMP master thread = MPI main thread

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 */
```

Usually, this barrier is programmed as an implicit barrier at the end of the preceding loop

No barrier inside

Barrier needed to prevent data races

No barrier at entry

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)

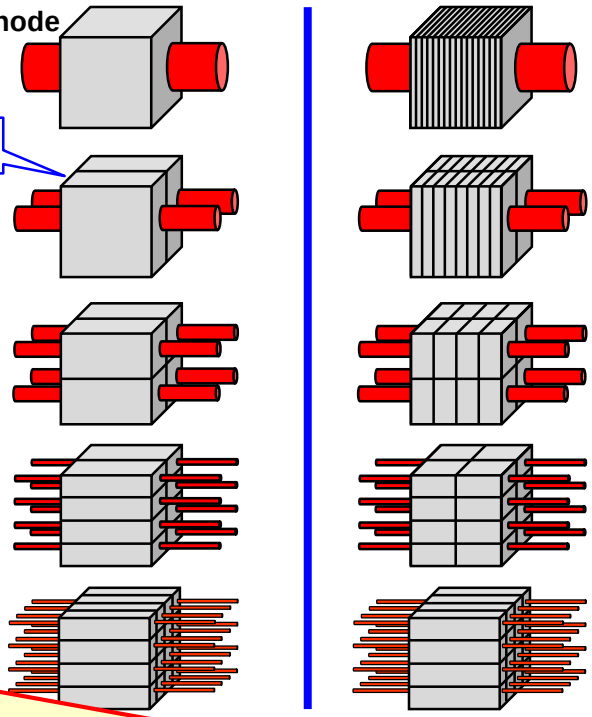
Pure MPI

Additional intra-node communi-cation with:

Internode: Irecv + Send

MPI processes within an SMP node

Latency	Accumulated inter-node bandwidth per node	Latency	Accumulated inter-node bandwidth per node	Additional intra-node communi-cation with:
4.1 µs,	**6.8 GB/s**	2.9 µs,	**4.4 GB/s**	Irecv+send
		3.4 µs,	**4.4 GB/s**	MPI-3.0 store
4.1 µs,	**7.1 GB/s**	3.0 µs,	**4.5 GB/s**	Irecv+send
		3.0 µs,	**4.6 GB/s**	MPI-3.0 store
4.1 µs,	**5.2 GB/s**	3.3 µs,	**4.4 GB/s**	Irecv+send
		3.5 µs,	**4.4 GB/s**	MPI-3.0 store
4.4 µs,	**4.7 GB/s**	5.2 µs,	**4.3 GB/s**	Irecv+send
		5.2 µs,	**4.4 GB/s**	MPI-3.0 store
10.2 µs,	**4.2 GB/s**	10.3 µs,	**4.5 GB/s**	Irecv+send
		10.1 µs,	**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

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

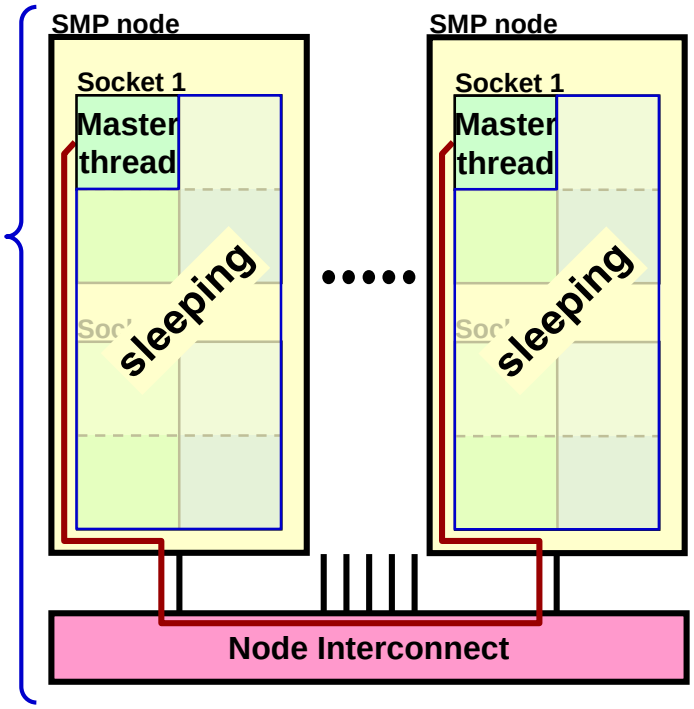
- On MPI level
  - **Dynamic load balancing** requires moving of parts of the data structure through the network
  - Significant runtime overhead
  - Complicated software / therefore not implemented

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

Programming models - MPI + OpenMP

How to compile, link, and run

How to compile, link and run

- Use appropriate OpenMP compiler switch (-openmp, -fopenmp, -mp, -qsmp=openmp, …) and MPI compiler script (if available)
- Link with MPI library
  - Usually wrapped in MPI compiler script
  - If required, specify to link against thread-safe MPI library
    - **Often automatic when OpenMP or auto-parallelization is switched on**
- Running the code
  - Highly non-portable! Consult system docs! (if available…)
  - If you are on your own, consider the following points
  - Make sure OMP_NUM_THREADS etc. is available on all MPI processes
    - **Start "env VAR=VALUE … <YOUR BINARY>" instead of your binary alone**
    - **Use an appropriate MPI launching mechanism (often multiple options available)**
  - Figure out how to start fewer MPI processes than cores on your nodes

Examples for compilation and execution

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

- **Intel Sandy Bridge (8-core 2-socket) cluster, Intel MPI/OpenMP**
  - `mpiifort -qopenmp ...`
  - `OMP_NUM_THREADS=8 mpirun -ppn 2 -np 4 \`
    `-env I_MPI_PIN_DOMAIN socket \`
    `-env KMP_AFFINITY scatter ./a.out`

Compiling from a single source

- Make use of pre-defined symbols

```
#ifdef _OPENMP  # _OpenMP defined with -qopenmp
       // all that is special for OpenMP
#endif

#ifdef USE_MPI  # USE_MPI defined with -DUSE_MPI
       // all that is special for  MPI
#endif

#ifdef USE_MPI
       MPI_Init(...);
       MPI_Comm_rank(..., &rank);
       MPI_Comm_size(..., &size);
#else          # recommended for non-MPI
       rank = 0;
       size = 1;
#endif
```

Compiling from a single source

- Handling the compilers (example: Intel)
- C

```
mpiicc -DUSE_MPI -qopenmp      ...
icc              -qopenmp      ...
```

- Fortran

```
mpiifort -fpp -DUSE_MPI -qopenmp      ...
ifort    -fpp           -qopenmp      ...
```

Learn about node topology

- A collection of tools is available
  - **`numactl --hardware`** (numatools)
  - **`lstopo --no-io`** (part of hwloc)
  - **`cpuinfo -A`** (part of Intel MPI)
  - **`likwid-topology`** (part of LIKWID tool suite http://tiny.cc/LIKWID)

```
$ likwid-topology -c -g
--------------------------------------------------------------------------------
CPU name: Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
CPU type: Intel Xeon IvyBridge EN/EP/EX processor
CPU stepping:       4
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:            2
Cores per socket:   8
Threads per core:   2

[... Some output omitted ...]
```

Learning about node topology

```
(...cont...)
********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+-----------------------------------------------------------------------------+
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 0 16  | | 1 17  | | 2 18  | | 3 19  | | 4 20  | | 5 21  | | 6 22  | | 7 23  | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------------------------------------------------------------------------+ |
| |                                  20MB                                   | |
| +-------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------+
```

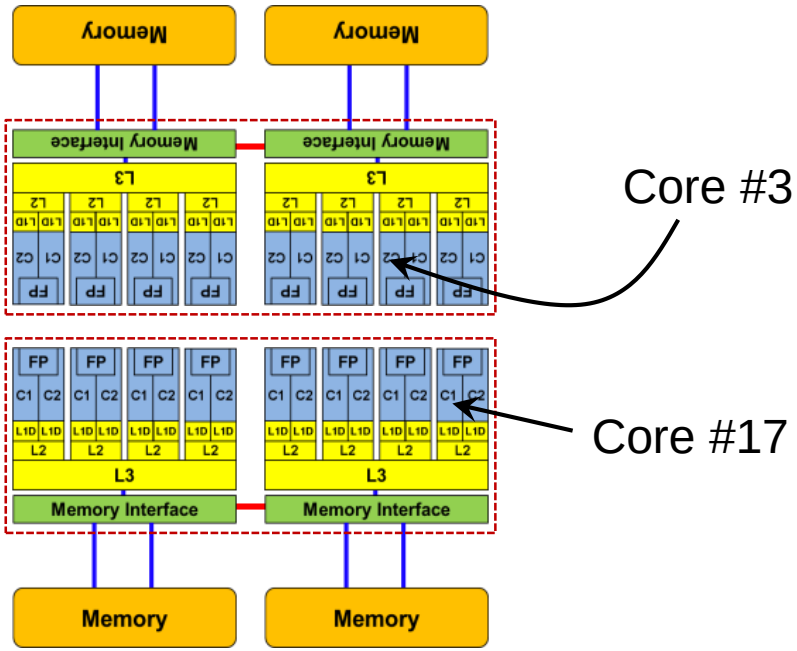
Learning about node topology

```
(...cont...)
********************************************************************************
Graphical Topology
********************************************************************************
Socket 1:
+-----------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  8 24  | |  9 25  | | 10 26  | | 11 27  | | 12 28  | | 13 29  | | 14 30  | | 15 31  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------+ |
| |                                  20MB                                   | |
| +-------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------+
```

How system topology, ccNUMA, and bandwidth bottlenecks impact performance

What is “topology”?

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



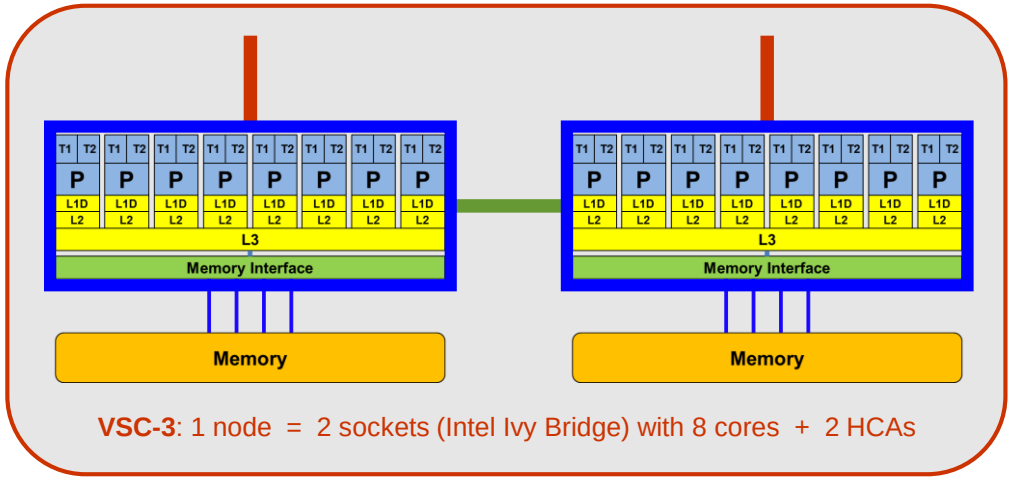
Why is this important?

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

Compute nodes – caches

VSC-3

LATENCY	← typical →	BW
1–2 ns	L1 cache	100 GB/s
3–10 ns	L2/L3 cache	50 GB/s
100 ns	memory	15 GB/s



VSC-3: 1 node = 2 sockets (Intel Ivy Bridge) with 8 cores + 2 HCAs

Info about nodes:

- **`numactl`** - control NUMA policy for processes or shared memory
  `numactl --show` (no info about caches)
  `numactl --hardware`
- **`module load intel/18 intel-mpi/2018 ; cpuinfo [-A]`**
- **`module load likwid/4.3.2 ; likwid-topology -c -g`**

skipped

Fat-tree Design

VSC-3:

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

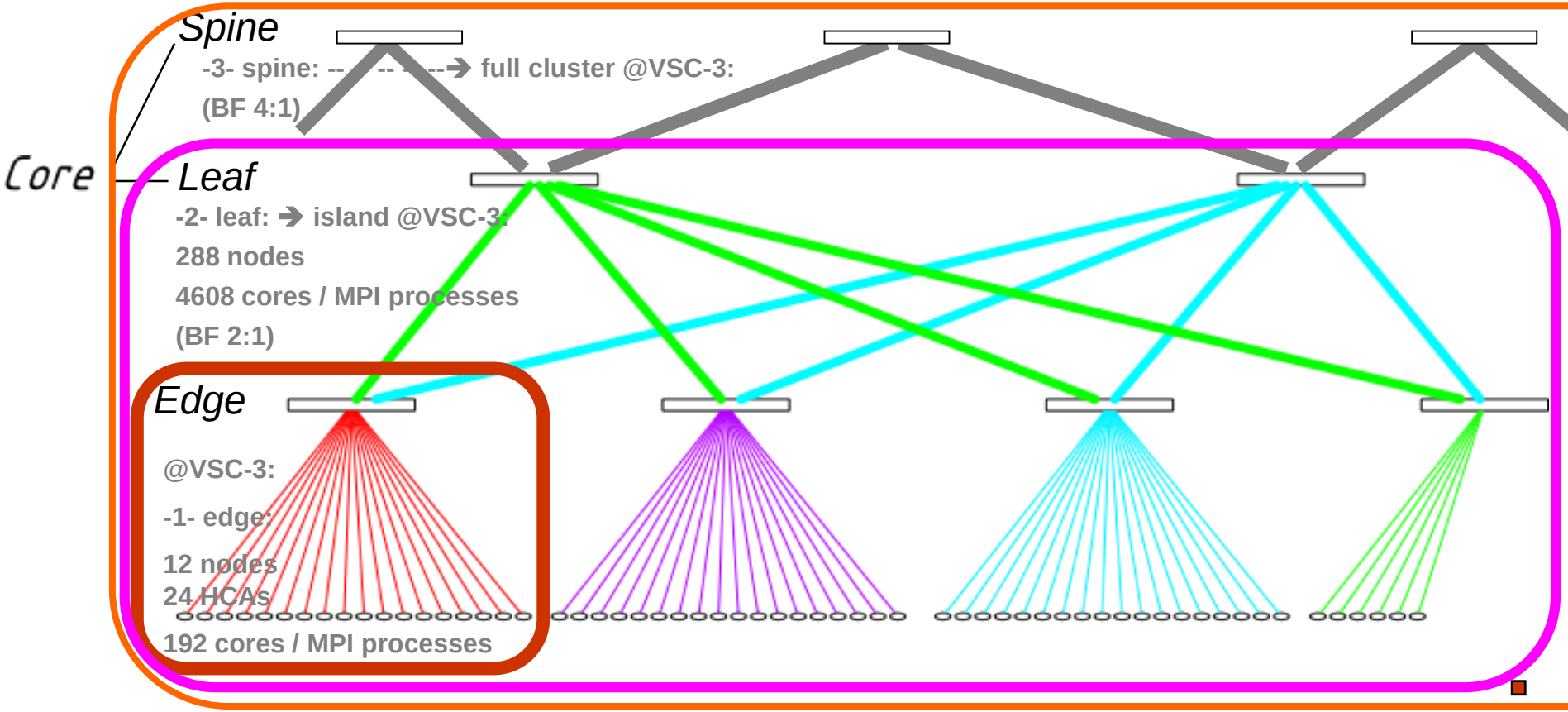
VSC-3: below numbers only, schematic figure

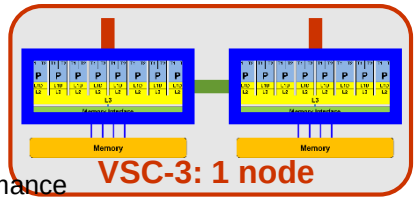
non-blocking: BF 1:1

blocking: BF down- : up-links

introduces a latency:

packets that would otherwise follow separate paths would eventually have to wait





VSC-3: 1 node

Ping-Pong Benchmark – Latency

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

Affinity matters!

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

Latency [µs]	MPI_Send(…)		typical latencies	
	OpenMPI	Intel-MPI	L1 cache	1–2 ns
intra-socket	0.3 µs	0.3 µs	L2/L3 c.	3–10 ns
inter-socket	0.6 µs	0.7 µs	memory	100 ns
IB -1- edge	1.2 µs	1.4 µs	HPC networks	1–10 µs
IB -2- leaf	1.6 µs	1.8 µs		
IB -3- spine	2.1 µs	2.3 µs		

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

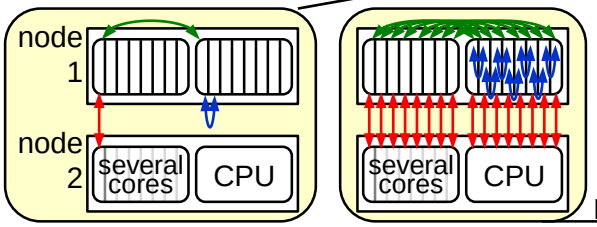
Ping-Pong Benchmark – Bandwidth

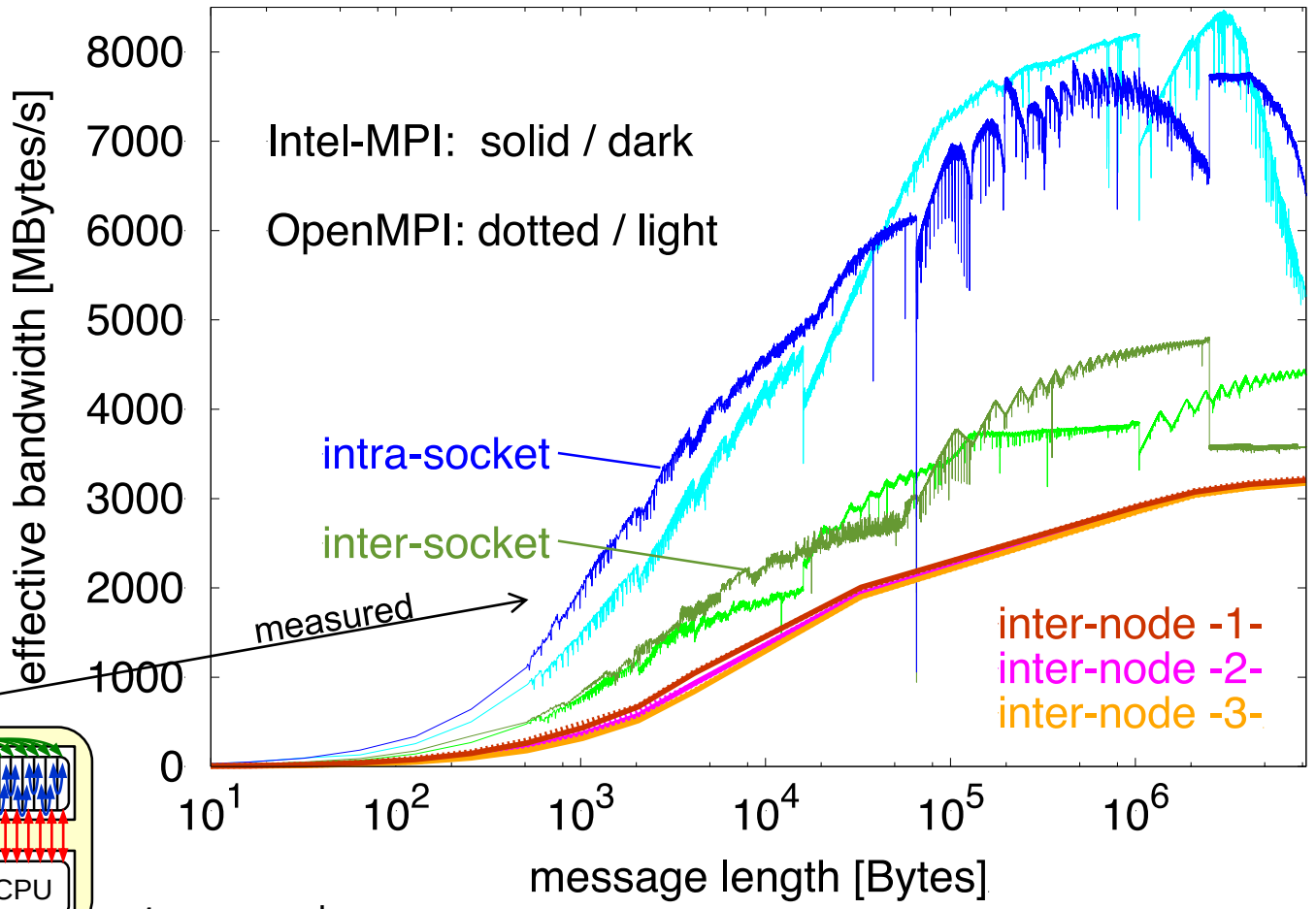
intra-node vs. inter-node on VSC-3

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

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

→ 1 HCA = ½ (2 HCAs)





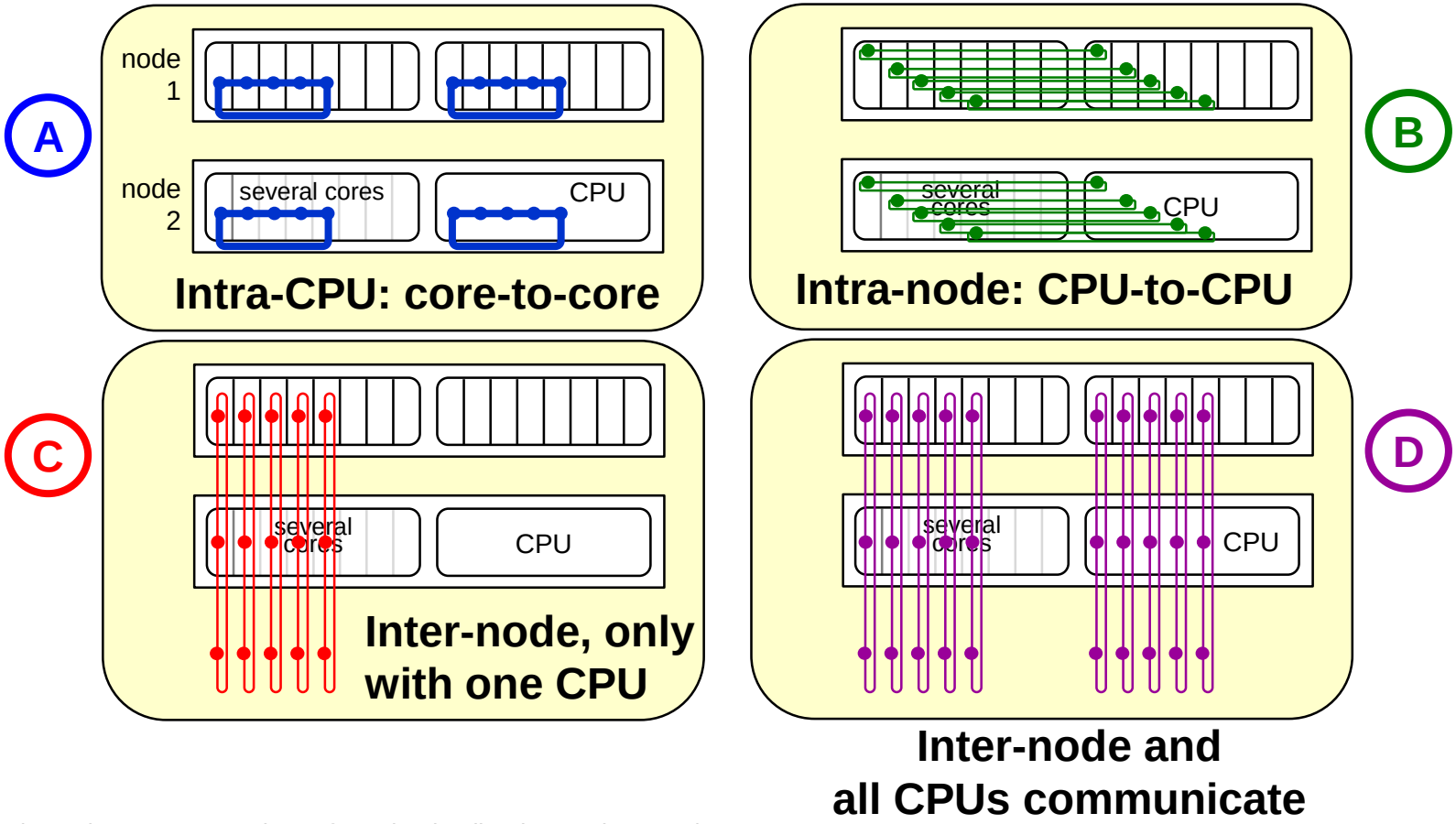
not measured

Multiple communicating rings

See HLRS online courses
http://www.hlrs.de/training/par-prog-ws/
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

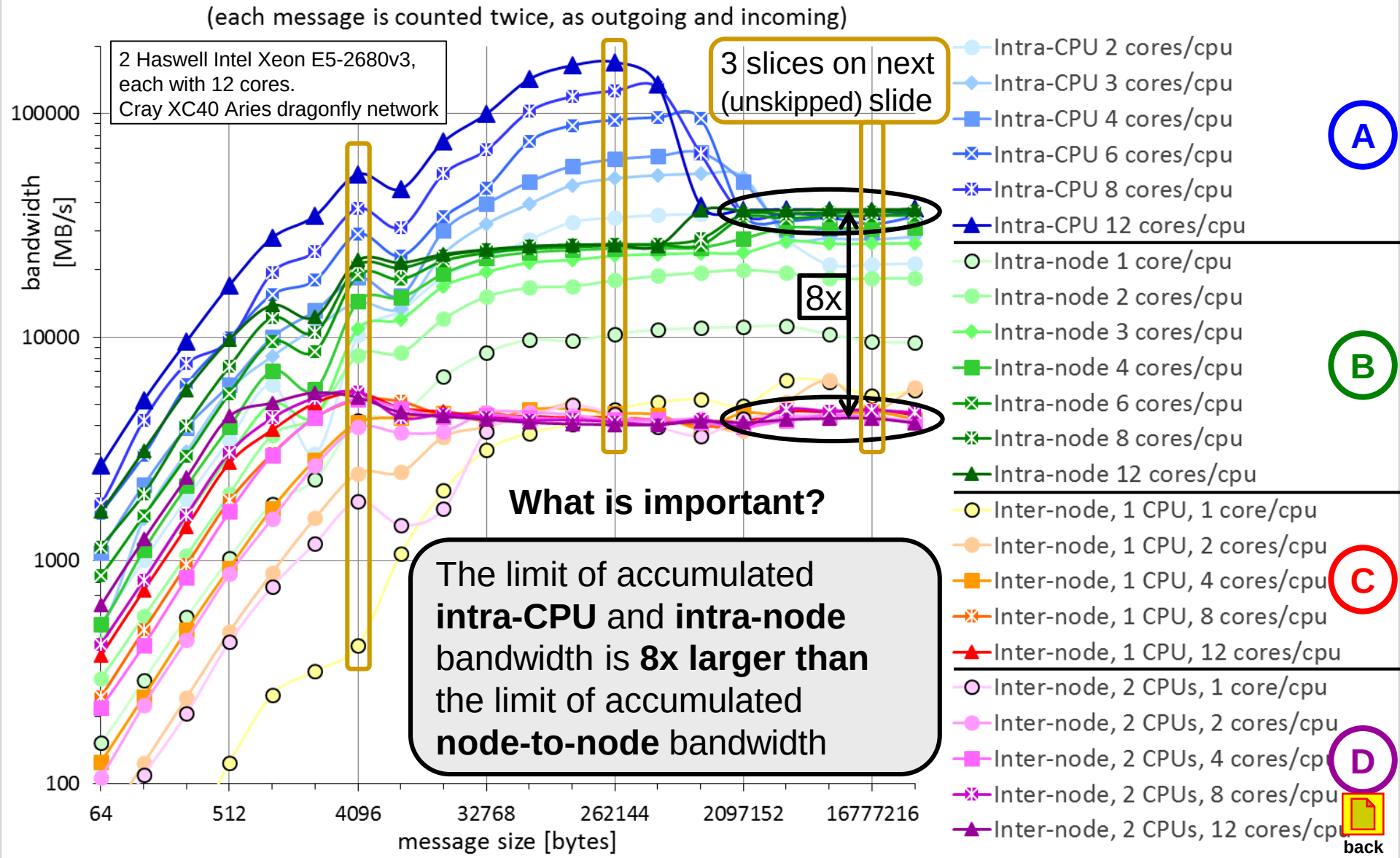
Benchmark halo_irecv_send_multiplelinks_toggle.c

- Varying message size,
- number of ***communication cores per CPU***, and
- four communication schemes (example with 5 ***communicating cores per CPU***)



See HLRS online courses **http://www.hlrs.de/training/par-prog-ws/**
→ Practical → MPI.tar.gz → subdirectory MPI/course/C/1sided/

Duplex accumulated ring bandwidth per node

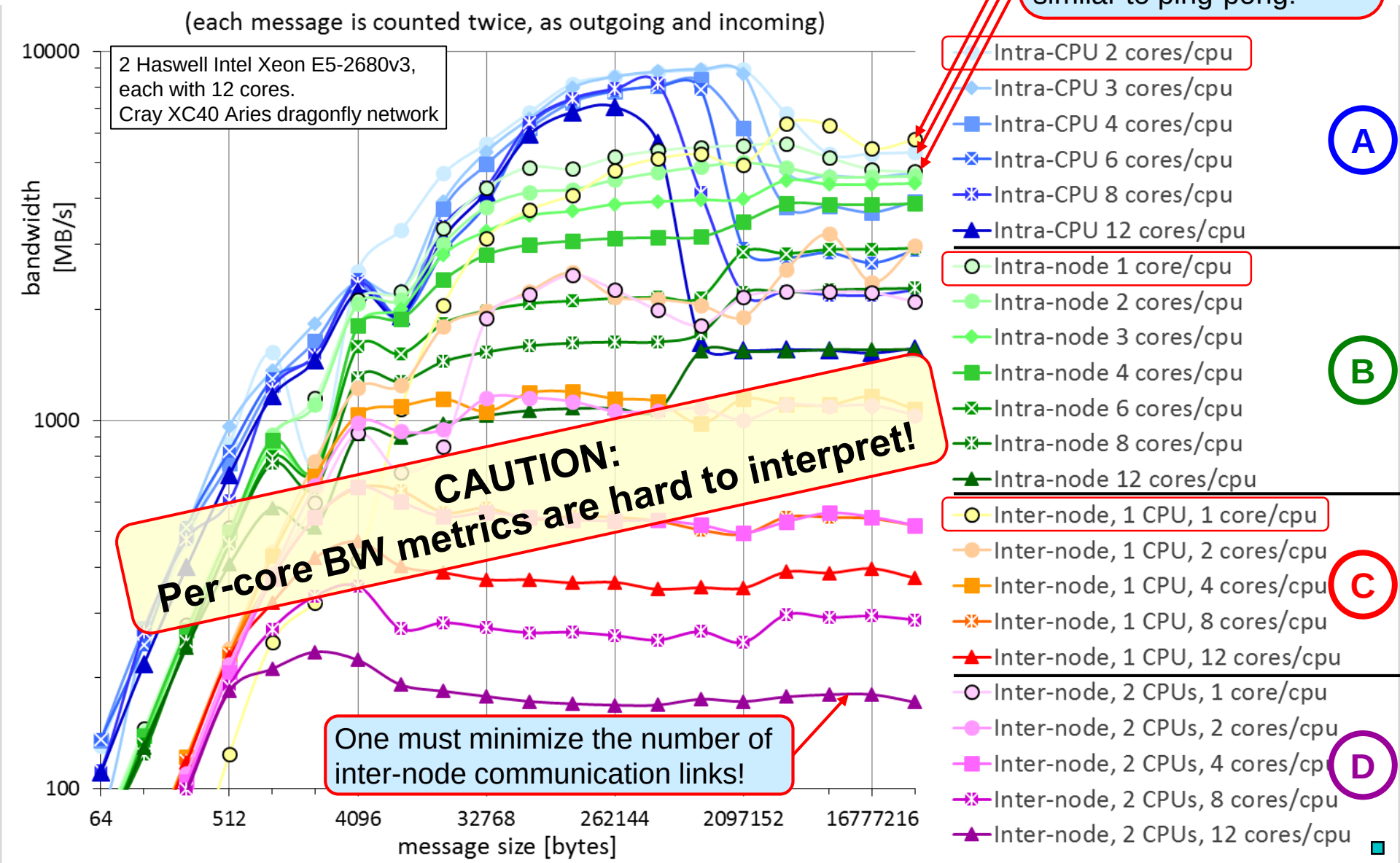


What is important?

The limit of accumulated **intra-CPU** and **intra-node** bandwidth is **8x larger than** the limit of accumulated **node-to-node** bandwidth

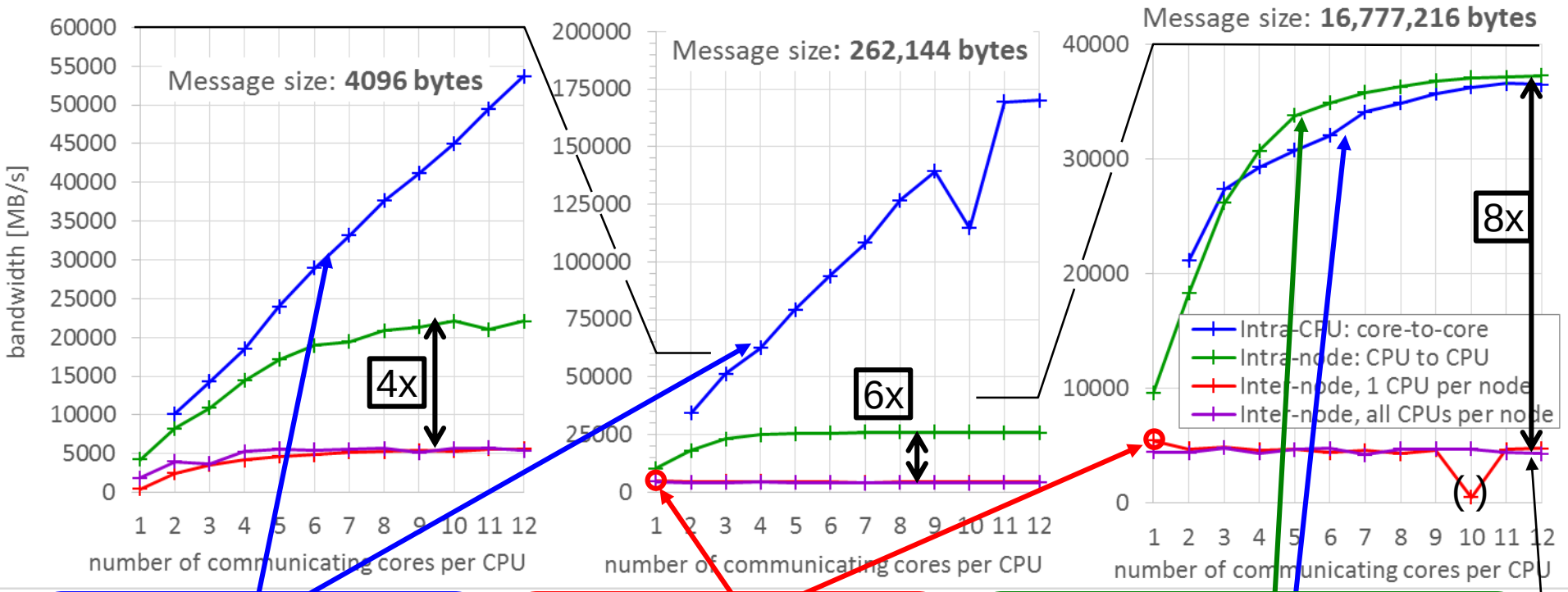
skipped

Duplex ring bandwidth per core



Measurement with halo_irecv_send_multiplelinks_toggle.c on 4 nodes of Cray XC40 hazelhen.hww.de, June 15, 2018, HLRS, by Rolf Rabenseifner (protocol 10)

Duplex accumulated ring bandwidth per node – scaling vs. asymptotic behavior



Core-to-core:
Linear scaling for small to medium size messages due to caches

Node-to-node:
One duplex link by **one core** already fully saturates the network

Core-to-core & CPU-to-CPU:
Long messages:
Same asymptotic limit through **memory bandwidth**

Result: The limit of accumulated **intra-CPU** and **intra-node** bandwidth is **8x larger than** the limit of accumulated **node-to-node** bandwidth

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

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

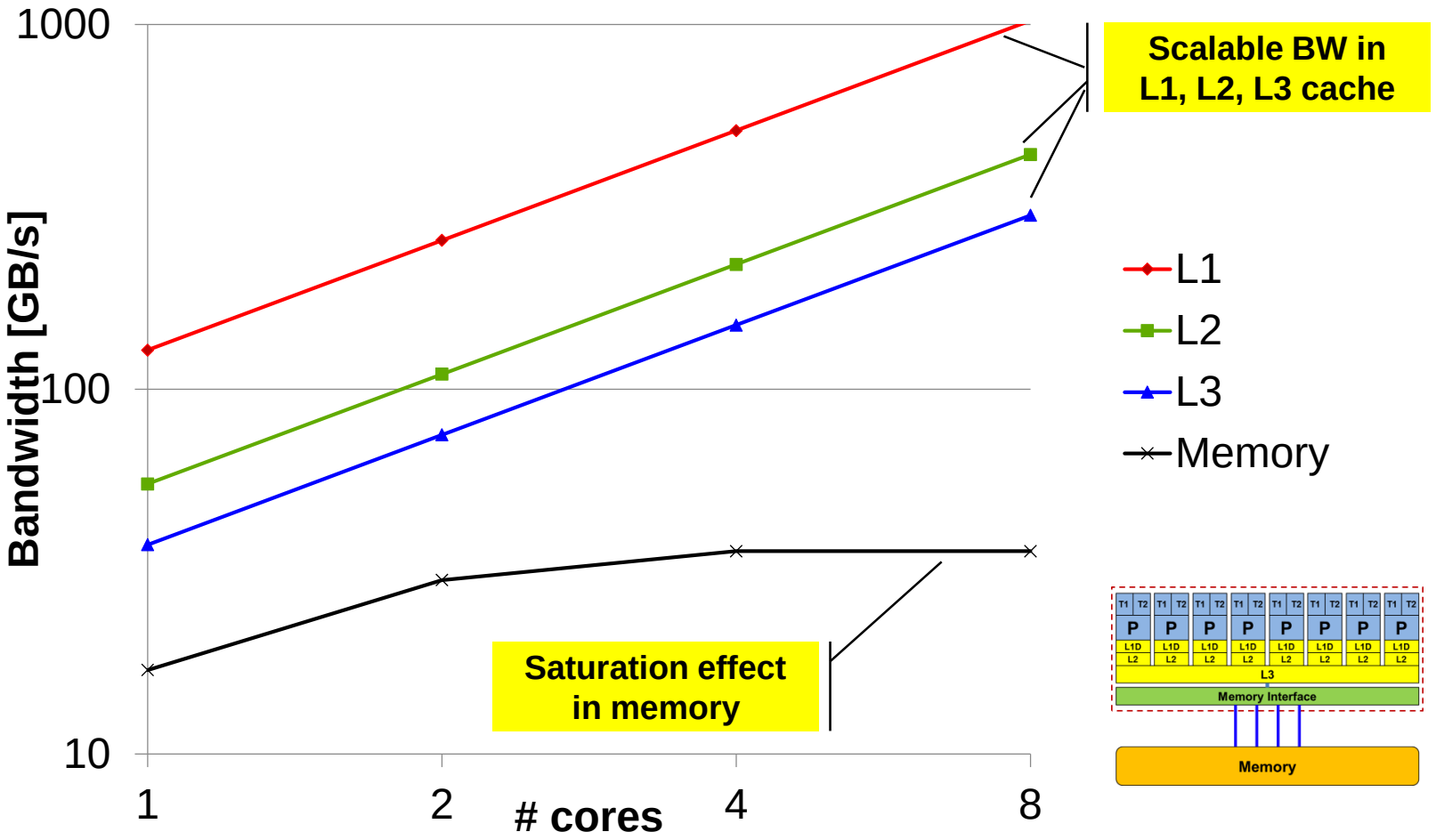
Repeat many times

Actual benchmark loop

Prevent smart-ass compilers from optimizing away the outer loop

- → pure hardware probing, no impact from OpenMP overhead

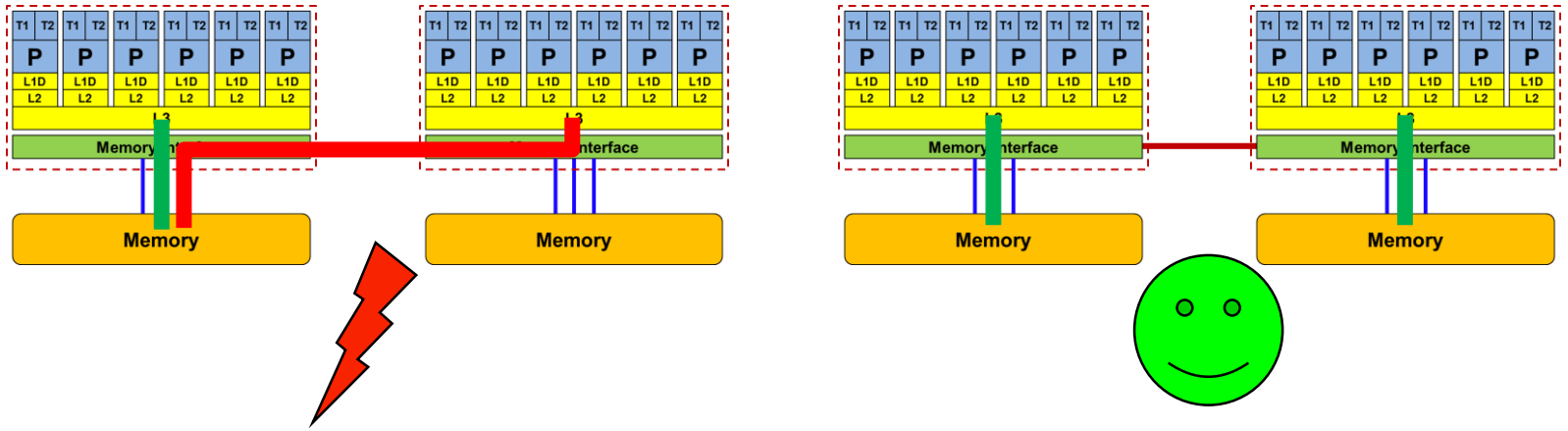
Bandwidth saturation vs. # cores
on Sandy Bridge socket (3 GHz)



ccNUMA: cache-coherent Non-Uniform Memory Architecture

A short introduction to ccNUMA

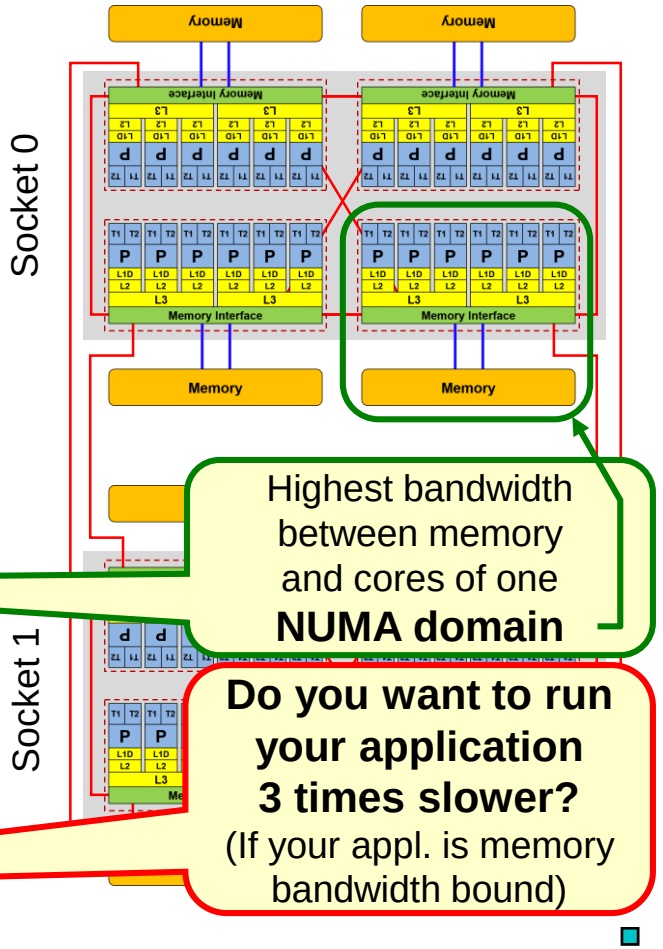
- ccNUMA:
  - whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
  - Memory placement occurs with OS page granularity (often 4 KiB)



How much bandwidth does non-local access cost?

- Example: AMD “Naples” 2-socket system (8 chips, 2 sockets, 48 cores): *STREAM Triad bandwidth measurements* [Gbyte/s]

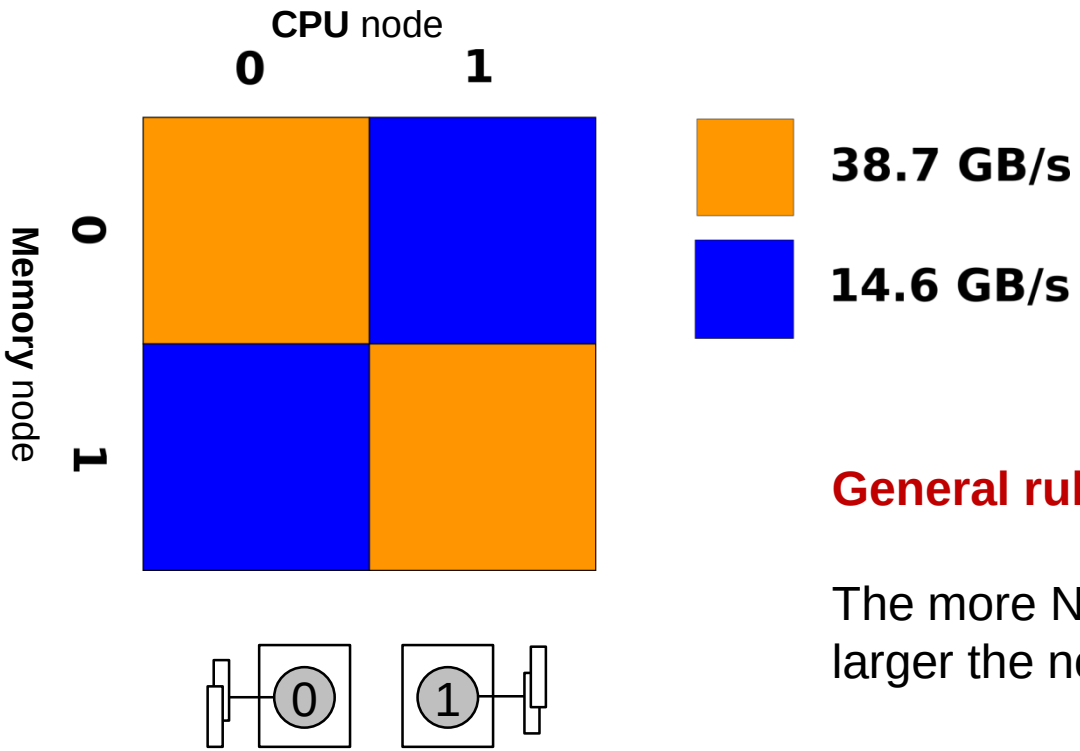
Memory node \ **CPU** node	0	1	2	3	4	5	6	7
0	32.4	21.4	21.8	21.9	10.6	10.6	10.7	10.8
1	21.5	32.4	21.9	21.9	10.6	10.5	10.7	10.6
2	21.8	21.9	32.4	21.5	10.6	10.6	10.8	10.7
3	21.9	21.9	21.5	32.4	10.6	10.6	10.6	10.7
4	10.6	10.7	10.6	10.6	32.4	21.4	21.9	21.9
5	10.6	10.6	10.6	10.6	21.4	32.4	21.9	21.9
6	10.6	10.7	10.6	10.6	21.9	21.9	32.3	21.4
7	10.7	10.6	10.6	10.6	21.9	21.9	21.4	32.5



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 NUMA domains, the larger the non-local access penalty

skipped

ccNUMA Memory Locality Problems

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

Avoiding locality problems

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

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

- **Taking control** is the key strategy!

Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:

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

- Consequences
  - Process/thread-core affinity is decisive!
  - **With OpenMP**, 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 NUMA domains, placement is not a problem
    - **e.g., with multi-threaded MPI process not spanning several ccNUMA domains, (for example with MPI+OpenMP and one MPI process on each ccNUMA domain)**
- Automatic page migration may help if program runs long enough
- See later for more details and examples

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 NUMA domain): fully automatic!

Most simple case: explicit initialization

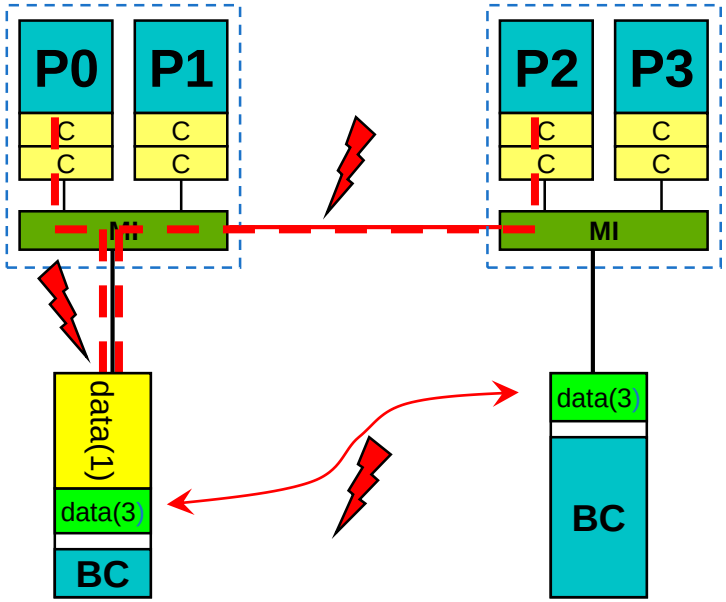
```
integer,parameter :: N=10000000
double precision A(N), B(N)



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
```

skipped

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!)

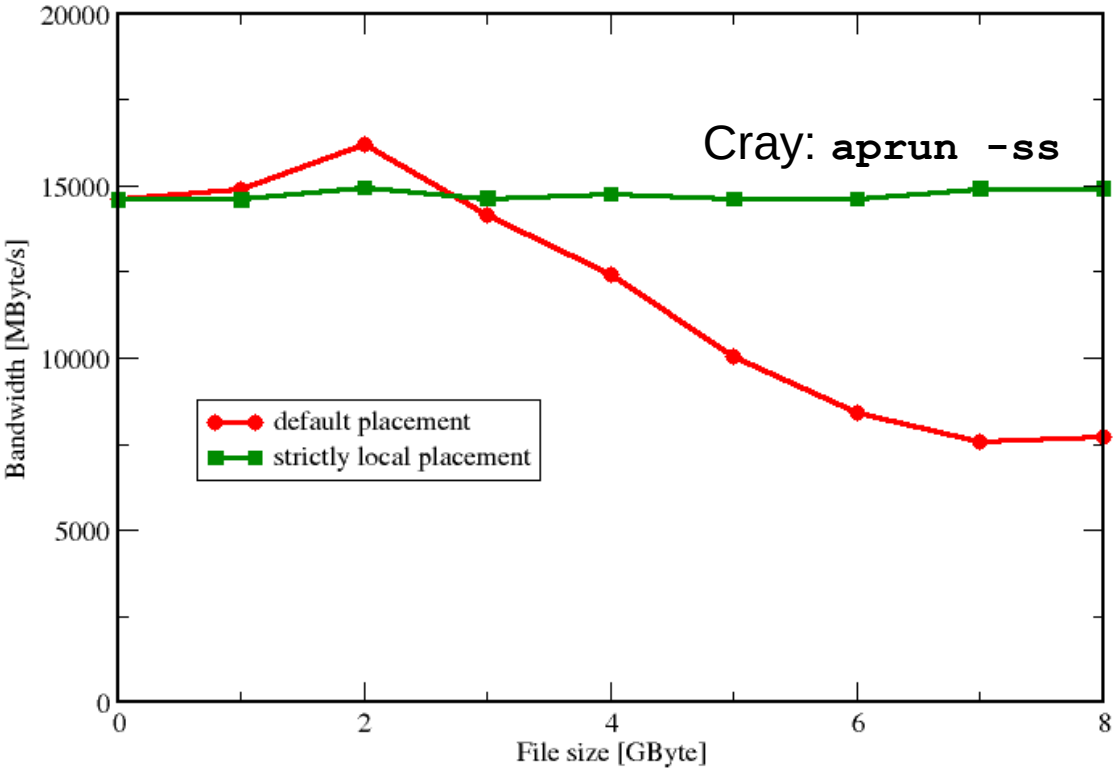
skipped

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!

Major prob… cluster of ccNUMA nodes?

Where are we?

MAJOR PROBLEMS

…sual programming models do **not** really fit: **Pure MPI / Hybrid MPI+OpenMP**

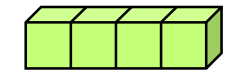
- Where are major problems? – And how to address?
  - Significant differences between **(small) inter-node bandwidth** and (high) intra-node bandwidth
    - → **How to minimize inter-node communication**
      - Hybrid MPI + OpenMP
      - Pure MPI + optimized virtual topologies
  - **Replicated user data**
    - → **Waste of memory with pure MPI**
      - Hybrid MPI + OpenMP
      - Pure MPI + MPI-3 shared memory model
  - Processes and threads may **move** between cores and CPUs within ccNUMA nodes & fixed memory locations after ***"first touch"***
    - → **How to keep control?**
      - Pinning of threads and processes to the hardware
      - Explicit first touch programming with OpenMP, …

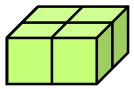
You remember:
- **With pure MPI, *first touch* comes for free**
- **It is an OpenMP problem**

  - Are your application AND libraries **prepared for** MPI+**OpenMP?**

How to handle ccNUMA in practice

- Problems appear when one process spans multiple NUMA domains:
  - **First touch** needed to "bind" the data to each socket → **otherwise loss of performance**
  - **Thread binding is mandatory!** The OS kernel does not know what you need!
  - Dynamic/guided schedule or tasking → **loss of performance**
- Practical solution:
  - One MPI process per NUMA domain
    → small number (>1) of MPI processes on each node
    → more complex affinity enforcement (binding)
    → more choices for rank mapping (4 sockets example):


surface=18N^2


surface=16N^2

Conclusions from the observed topology effects

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

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

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

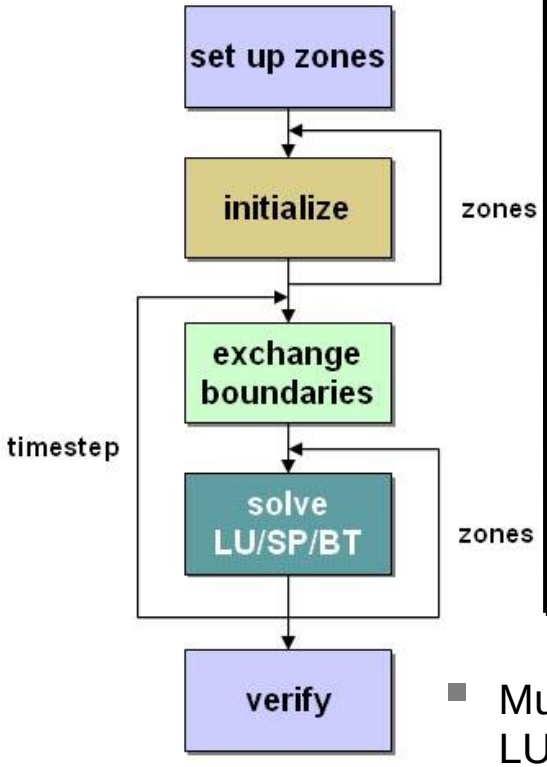
- **Leveraging topology effects is a part of code optimization!**

Programming models
- MPI + OpenMP

Case study:
The Multi-Zone NAS Parallel Benchmarks

The low-hanging fruits: load balancing and memory consumption

The Multi-Zone NAS Parallel Benchmarks

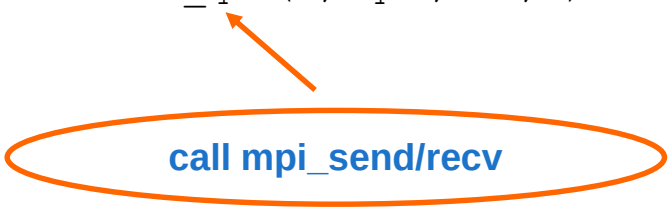


	MPI/ OpenMP	Seq	Nested OpenMP
Time step	sequential	sequential	sequential
inter-zones	MPI Processes	direct access	OpenMP
exchange boundaries	Call MPI	direct	OpenMP
intra-zones	OpenMP	sequential	OpenMP

- Multi-zone versions of the NAS Parallel Benchmarks LU,SP, and BT
- Two hybrid sample implementations
- Load balance heuristics part of sample codes
- https://www.nas.nasa.gov/publications/npb.html

MPI/OpenMP BT-MZ

```
call omp_set_numthreads (weight)
do step = 1, itmax
  call exch_qbc(u, qbc, nx,…)
```




```
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

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

skipped

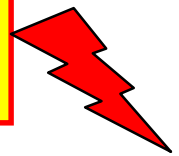
Pitfall: Remote memory access

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

Pinning A:

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

600 Gflops



Half of the threads access remote memory (other socket)

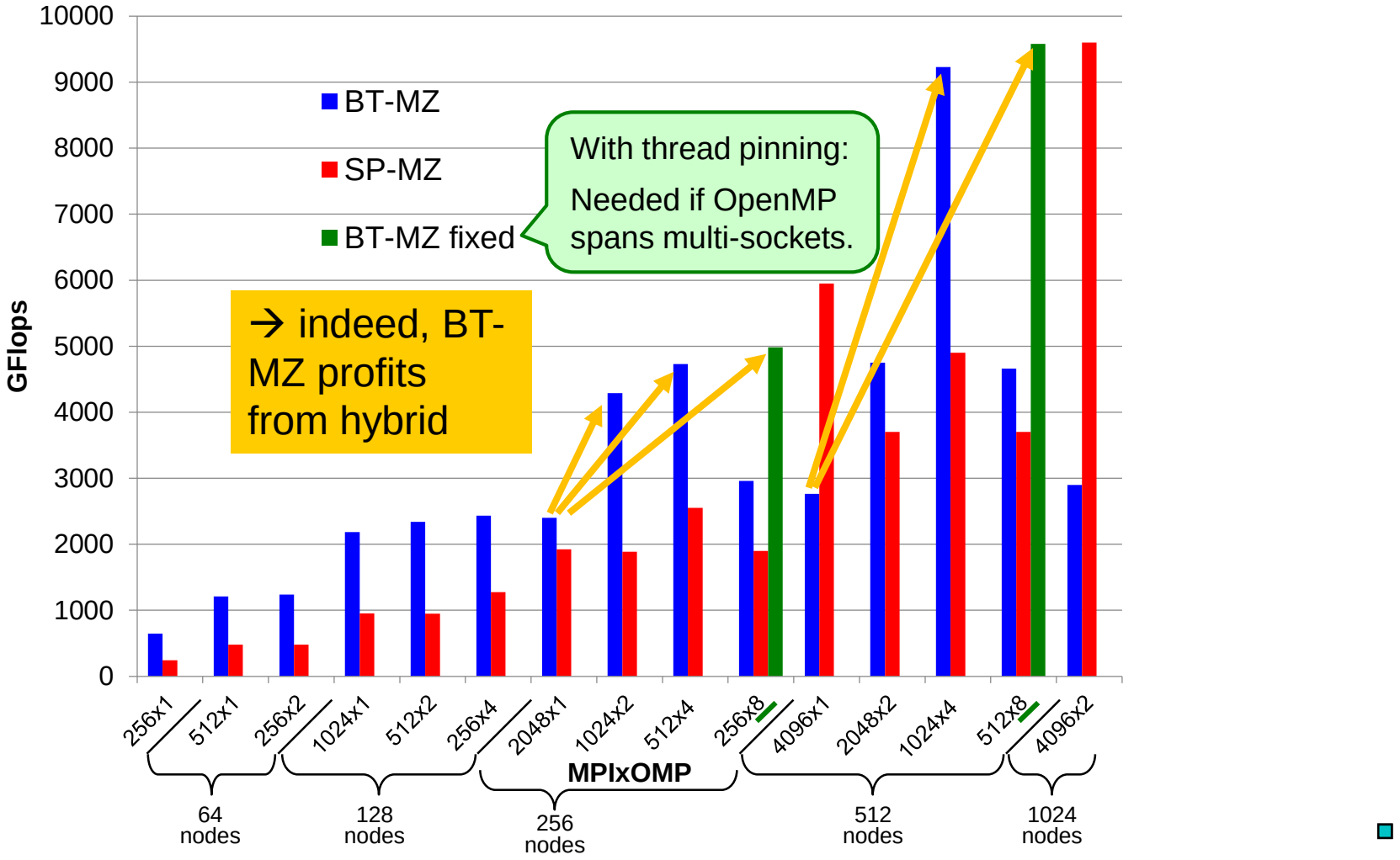
Pinning B:

```
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
```

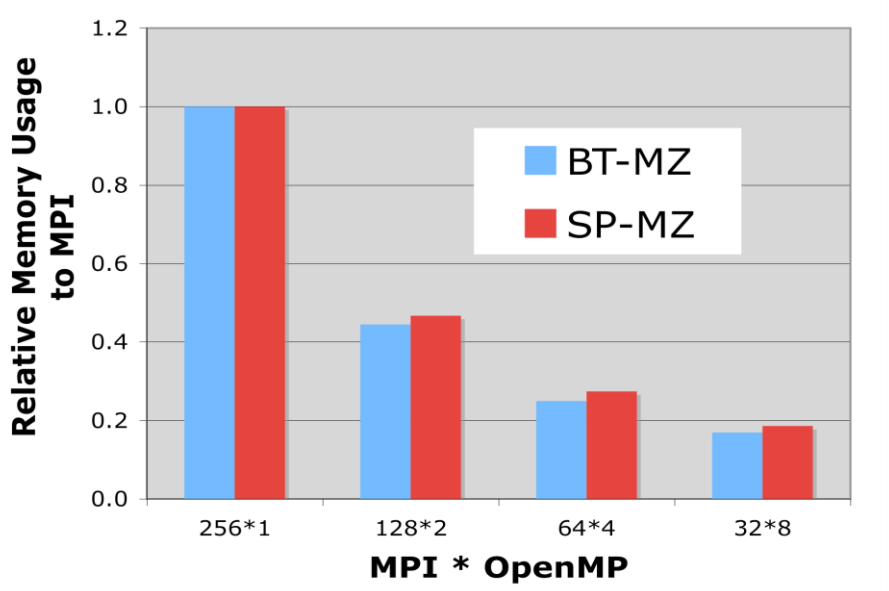
900 Gflops

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.

Slide, courtesy of Alice Koniges NERSC, LBLN

Programming models
- MPI + OpenMP

Topology and affinity on multicore

The OpenMP-parallel vector triad benchmark

Visualizing OpenMP overhead

- OpenMP work sharing in the benchmark loop

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

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

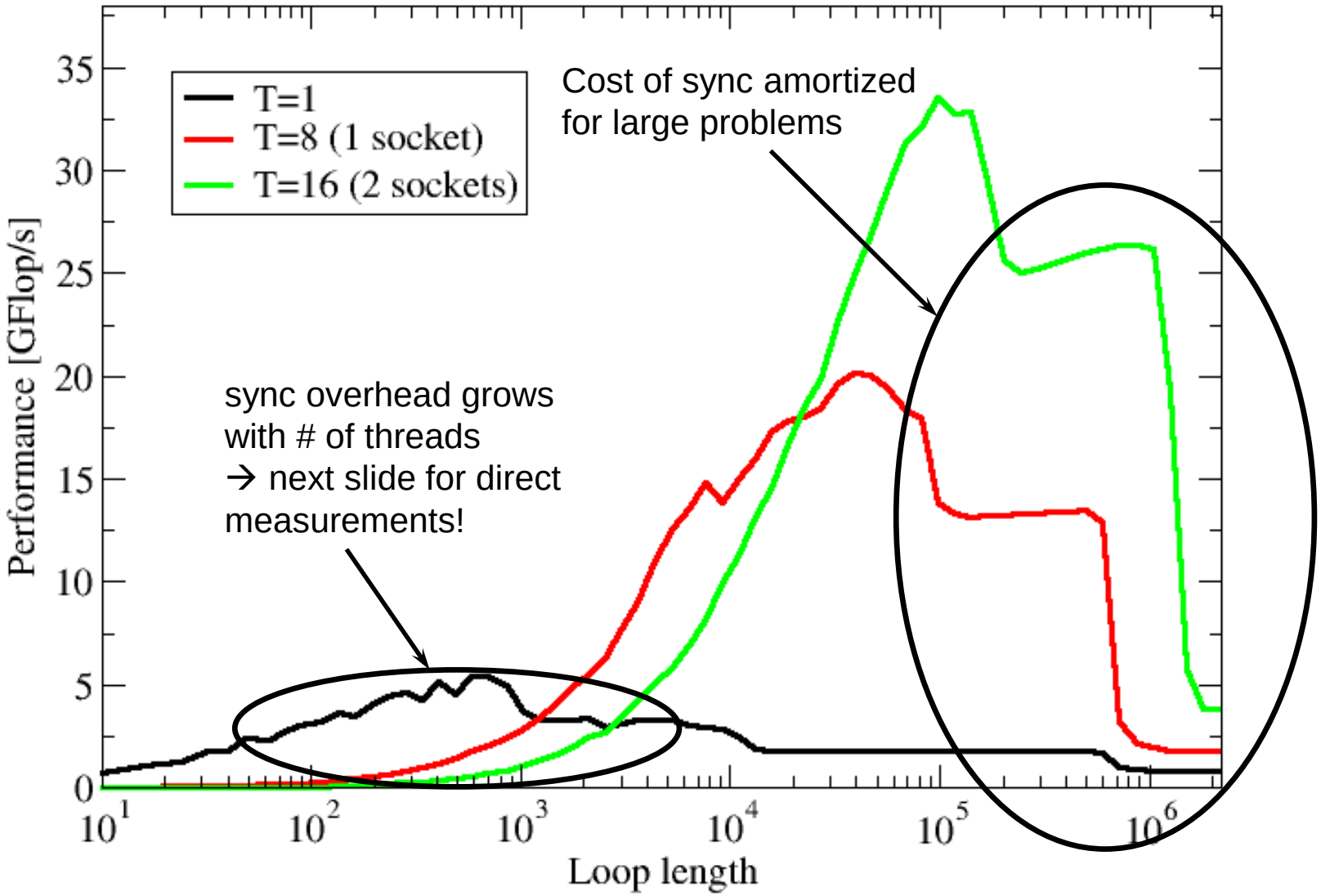
OpenMP work sharing

Numerical loop

Implicit barrier

… and then report performance vs. loop size for different #cores!

OpenMP vector triad on Intel Sandy Bridge node (2 sockets, 3 GHz)

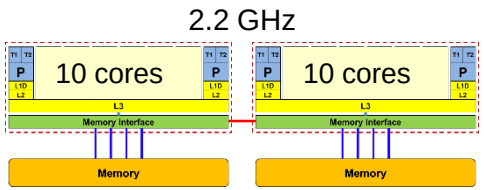


Thread synchronization overhead on IvyBridge-EP

Direct measurement of barrier overhead in CPU cycles

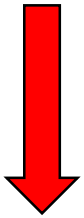
2 Threads	Intel 16.0	GCC 5.3.0
Shared L3	**599**	**425**
SMT threads	**612**	**423**
Other socket	**1486**	**1067**

Strong topology dependence!



Strong dependence on compiler, CPU and system env.!

Full domain	Intel 16.0	GCC 5.3.0
Socket (10 cores)	1934	1301
Node (20 cores)	**4999**	**7783**
Node +SMT	5981	9897



Overhead grows with thread count

Thread/Process Affinity (“Pinning”)

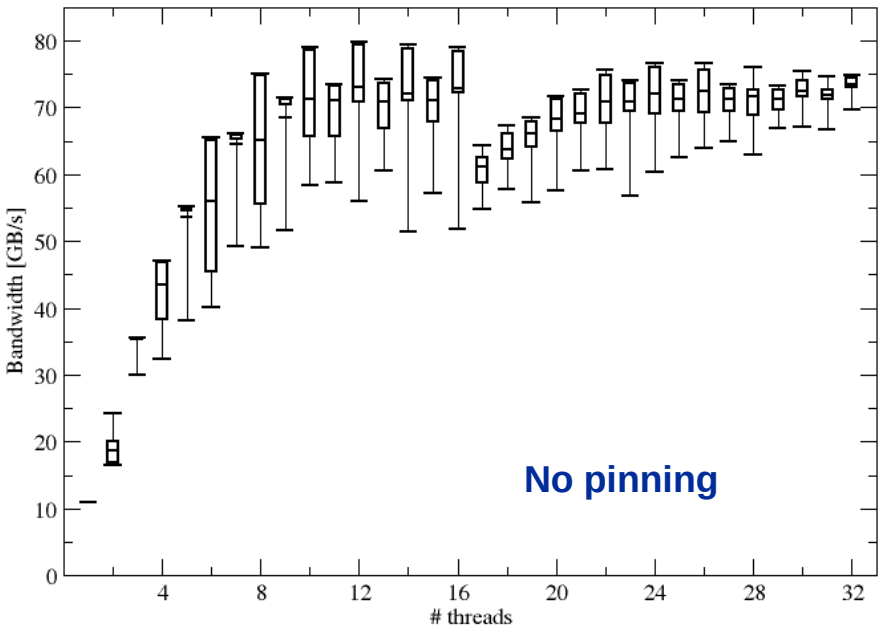
- Highly OS-dependent system calls
  - But available on all systems

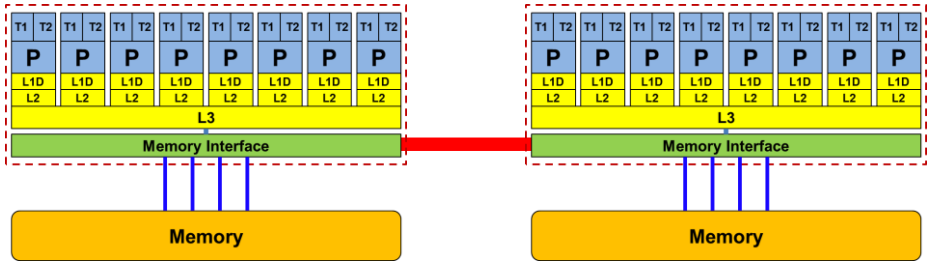
    Linux: **`sched_setaffinity()`**, PLPA → hwloc
    Windows: **`SetThreadAffinityMask()`**
    …

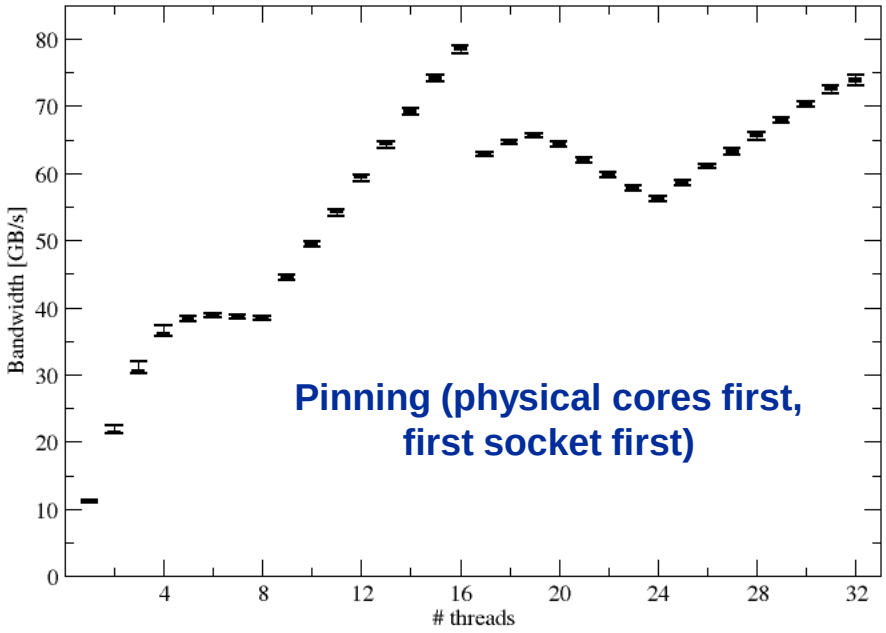
- Support for “semi-automatic” pinning in all modern compilers
  - Intel, GCC, PGI,…
  - OpenMP >=4.0 (places)
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`**

- 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

skipped

likwid-pin

- Binds 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`**

skipped

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
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
[... some STREAM output omitted ...]
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
[pthread wrapper] PIN_MASK: 0->1  1->4  2->5
[pthread wrapper] SKIP MASK: 0x1
[pthread wrapper 0] Notice: Using libpthread.so.0
        threadid 1073809728 -> SKIP
[pthread wrapper 1] Notice: Using libpthread.so.0
        threadid 1078008128 -> core 1 - OK
[pthread wrapper 2] Notice: Using libpthread.so.0
        threadid 1082206528 -> core 4 - OK
[pthread wrapper 3] Notice: Using libpthread.so.0
        threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]
```

Main PID always pinned

Knows about (and ignores) admin threads

Pin all spawned threads in turn

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

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

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 …

OMP_PROC_BIND variable / proc_bind() clause

- Determines how places are used for pinning:

OMP_PROC_BIND	Meaning
FALSE	Affinity disabled
TRUE	Affinity enabled, implementation defined strategy
CLOSE	Threads bind to consecutive places
SPREAD	Threads are evenly scattered among places
MASTER	Threads bind to the same place as the master thread that was running before the parallel region was entered

Used for **nested** OpenMP

- If there are more threads than places, consecutive threads are put into individual places (“balanced”)

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x10 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=10
OMP_PLACES=cores
OMP_PROC_BIND=close
```

- Intel Xeon Phi with 72 cores,
  32 cores to be used, 2 threads per physical core

```
OMP_NUM_THREADS=64
OMP_PLACES=cores(32)
OMP_PROC_BIND=close     # spread will also do
```

- Intel Xeon, 2 sockets, 4 threads per socket (no binding within socket!)

```
OMP_NUM_THREADS=8
OMP_PLACES=sockets
OMP_PROC_BIND=close     # spread will also do
```

- Intel Xeon, 2 sockets, 4 threads per socket, binding to cores

```
OMP_NUM_THREADS=8
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

Always prefer abstract places instead of HW thread IDs!

skipped

OpenMP places and proc_bind (see OpenMP-4.0 pages 49f, 239, 241-243)

`setenv OMP_PLACES "{0},{1},{2}, … {29},{30},{31}"` or

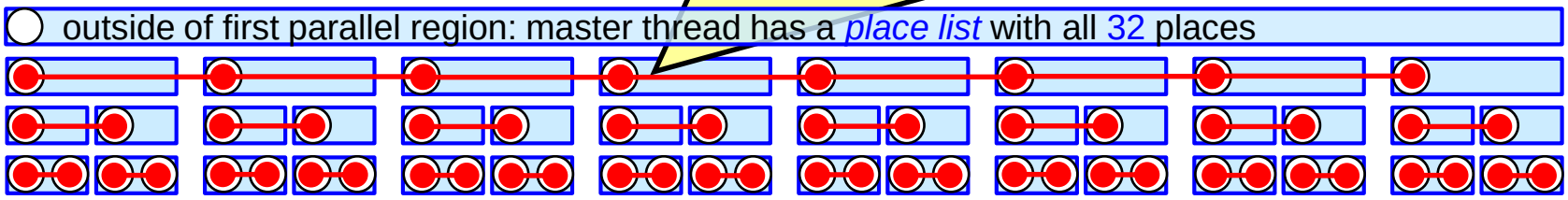
`setenv OMP_PLACES threads` (example with P=32 places)

- `setenv OMP_NUM_THREADS "8,2,2"`
  `setenv OMP_PROC_BIND "spread,spread,close"`
- Master thread encounters nested parallel regions:

```
#pragma omp parallel          → uses: num_threads(8)  proc_bind(spread)
   #pragma omp parallel       →   uses: num_threads(2)  proc_bind(spread)
      #pragma omp parallel    →     uses: num_threads(2)  proc_bind(close)
```

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.

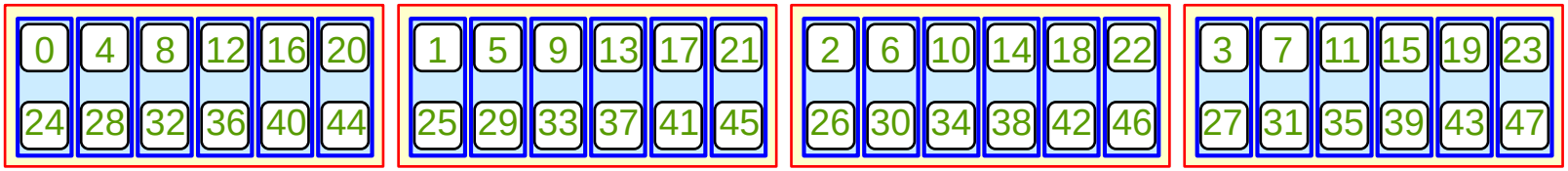
master: All new threads at the same place as the parent.

skipped

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

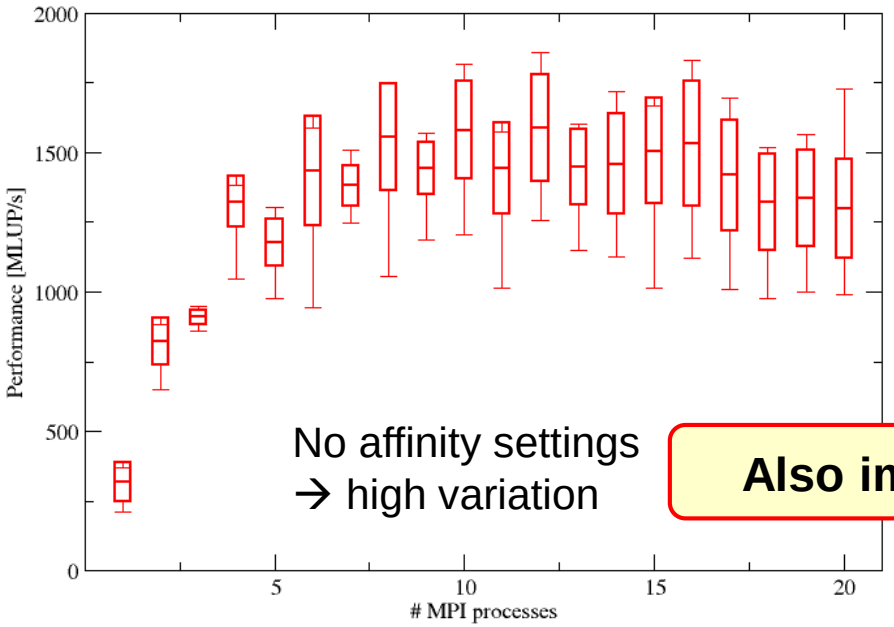
Examples should be **independent** of vendor's numbering!

- 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

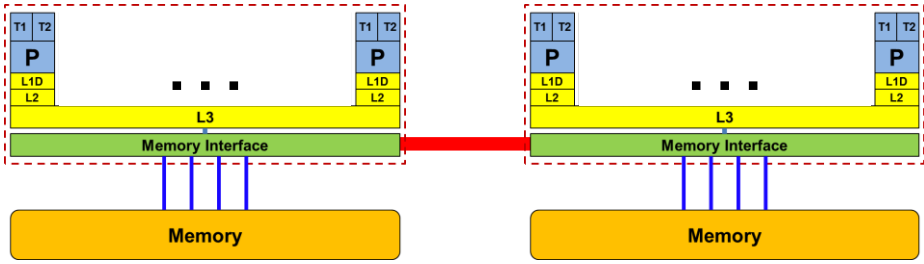
Pinning of MPI processes

- Pinning is helpful for all programming models
- Highly system-dependent!
- Intel MPI: env variable **`I_MPI_PIN_DOMAIN`**
- OpenMPI: choose between several mpirun options, e.g.,
  -bind-to-core, -bind-to-socket, -bycore, -byslot …
- Cray's aprun: pinning by default
- Platform-independent tools: likwid-mpirun
  (likwid-pin, numactl)

Anarchy vs. affinity with a heat equation solver

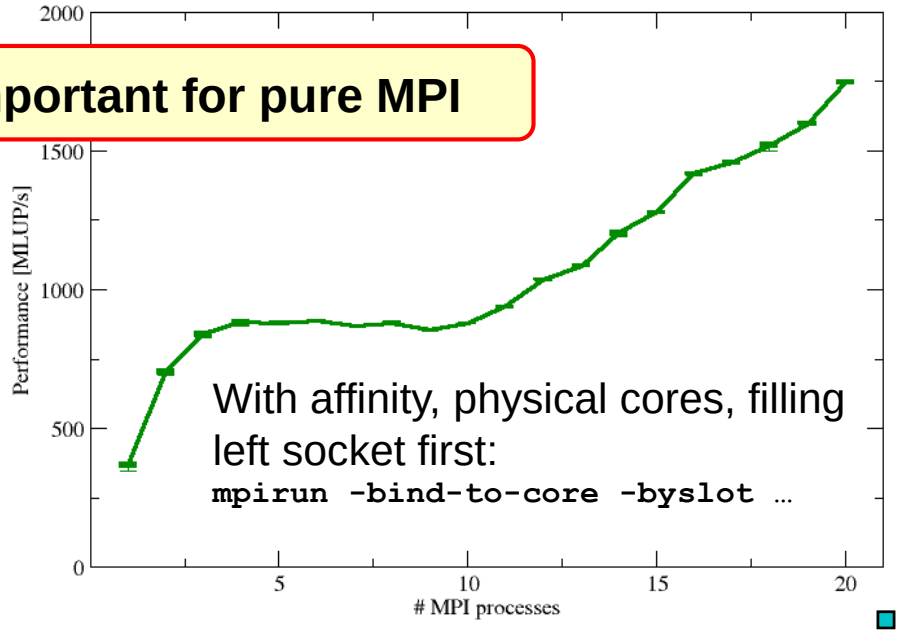


No affinity settings
→ high variation



2x 10-core Intel Ivy Bridge, OpenMPI

Also important for pure MPI



With affinity, physical cores, filling left socket first:

```
mpirun -bind-to-core -byslot …
```

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

skipped

Topology ("mapping") with MPI+OpenMP:

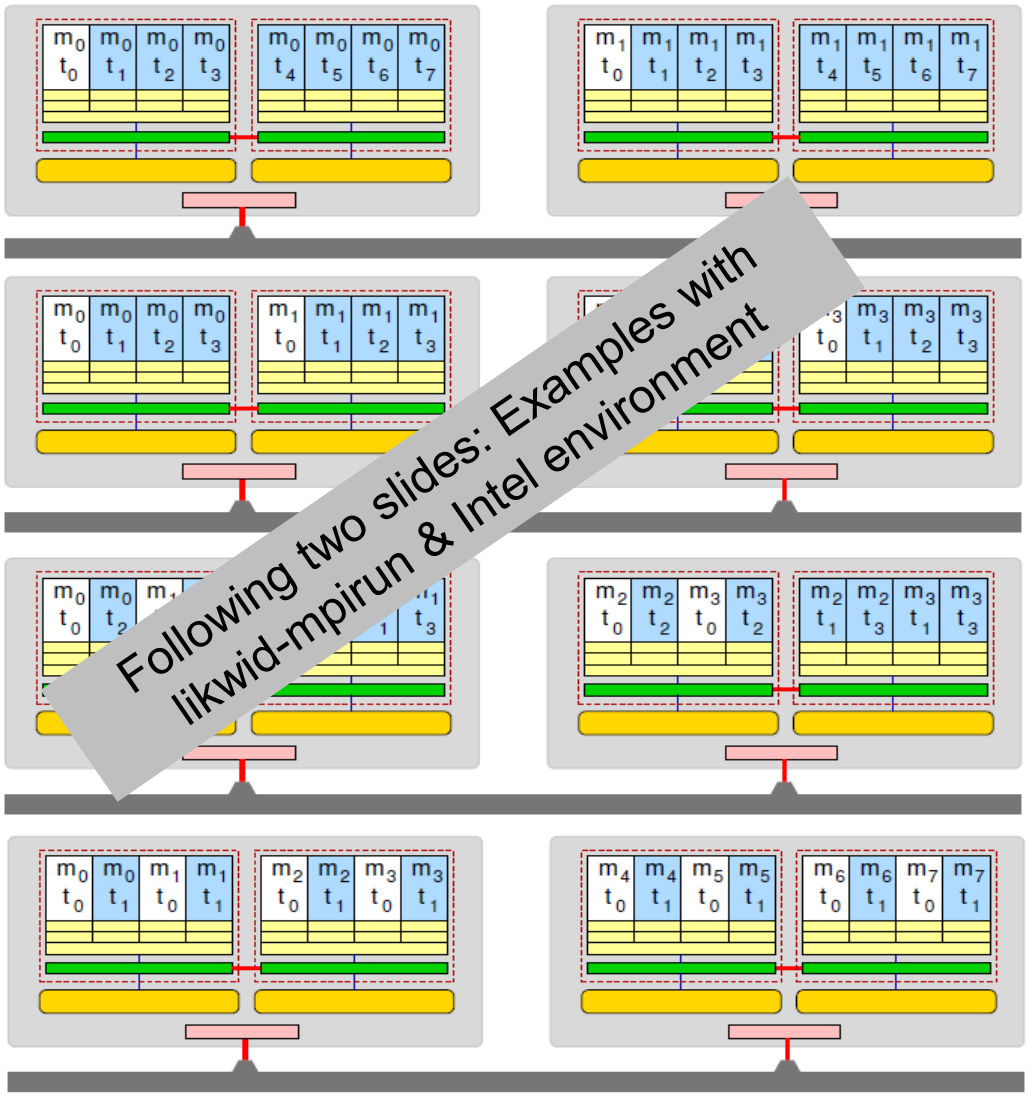
Lots of choices – solutions are highly system specific!

One MPI process per node

One MPI process per socket

OpenMP threads pinned "round robin" across cores in node

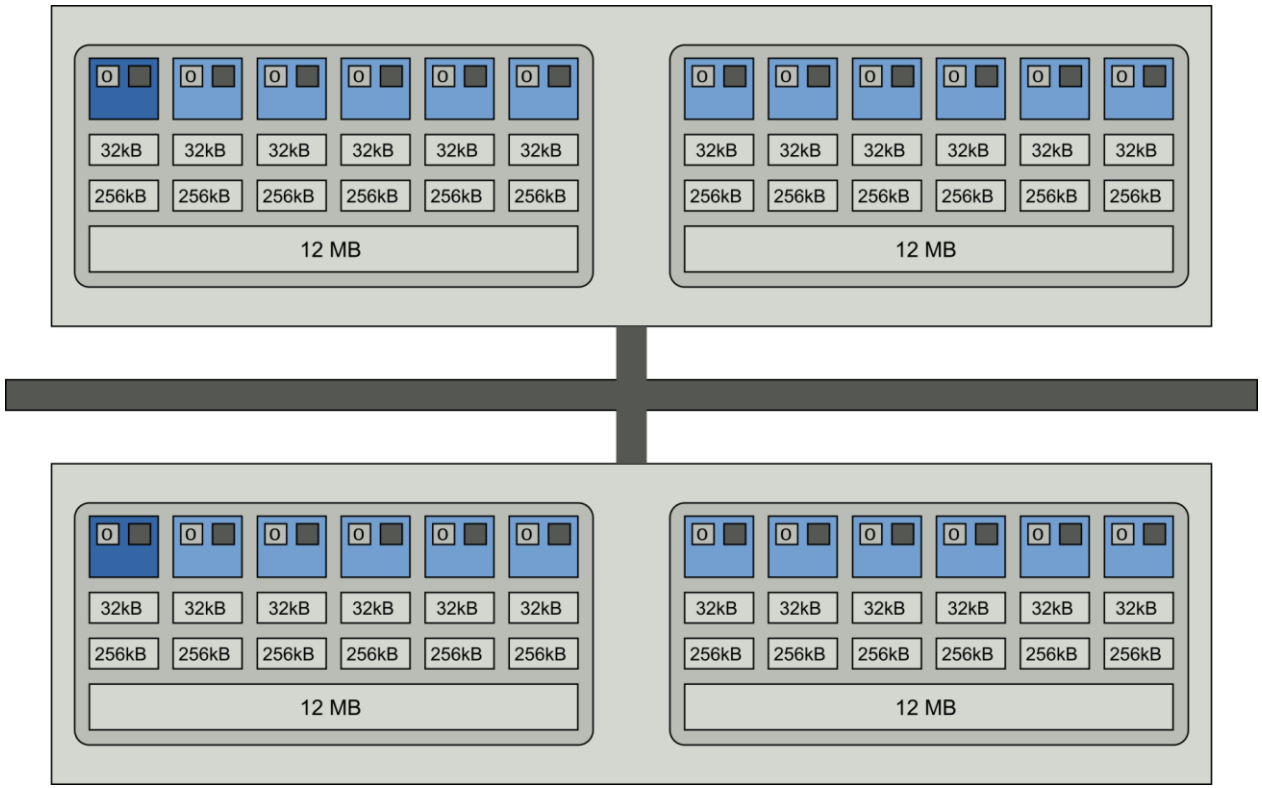
Two MPI processes per socket



likwid-mpirun

1 MPI process per node

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



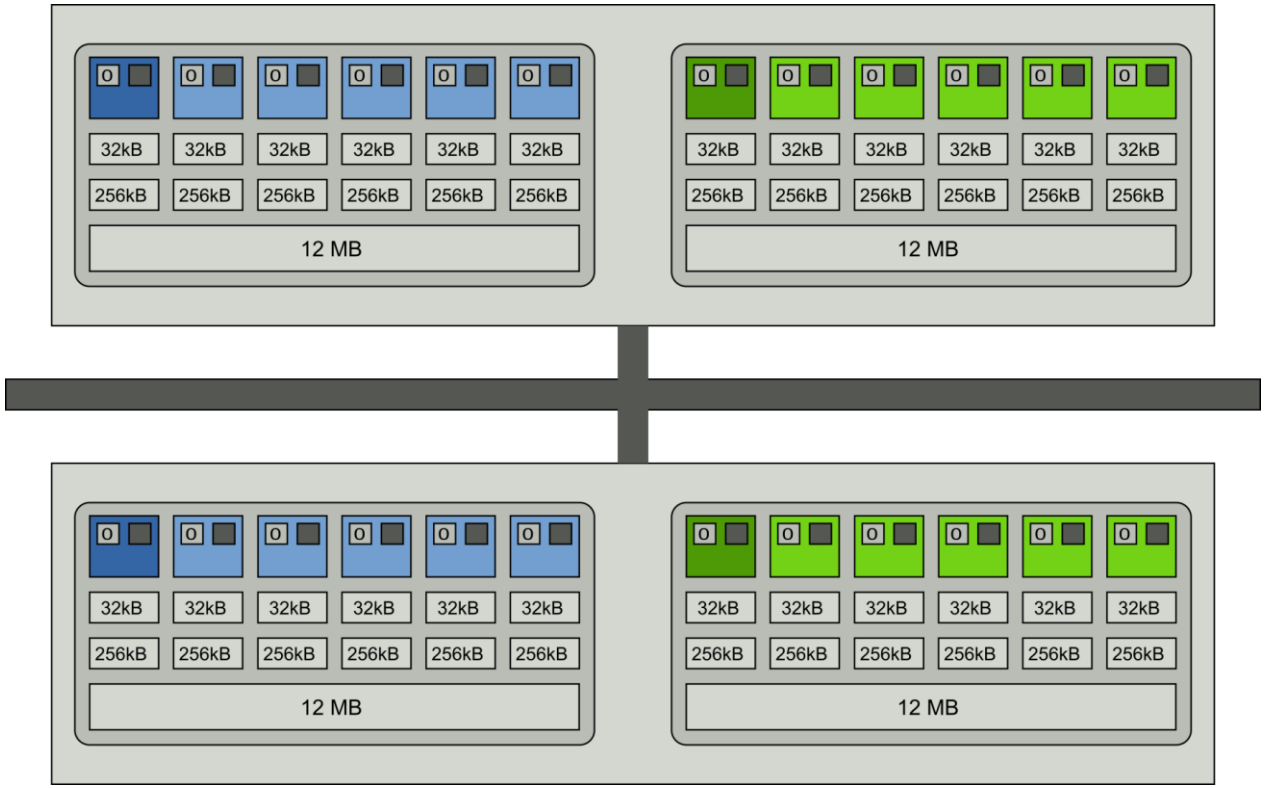
Intel MPI+compiler:

```
OMP_NUM_THREADS=12 mpirun -ppn 1 -np 2 -env KMP_AFFINITY scatter ./a.out
```

likwid-mpirun

1 MPI process per socket

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



Intel MPI+compiler:

```
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \
     -env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out
```

MPI/OpenMP ccNUMA and topology: Take-home messages

- **Learn** how **to take control** of hybrid execution!
  - Almost all performance features depend on topology and thread placement! (especially if SMT/Hyperthreading is on)
- Always observe the topology dependence of
  - Intranode MPI
  - OpenMP overheads
  - Saturation effects / scalability behavior with bandwidth-bound code
- Enforce proper thread/process to core binding, using appropriate tools (whatever you use, but use SOMETHING)

- Multi-domain OpenMP processes on ccNUMA nodes require correct page placement: Observe first touch policy!

Programming models
- MPI + OpenMP

Overlapping Communication and Computation

Overlapping communication and computation explicitly

```
if (my_thread_rank < 1) {
```

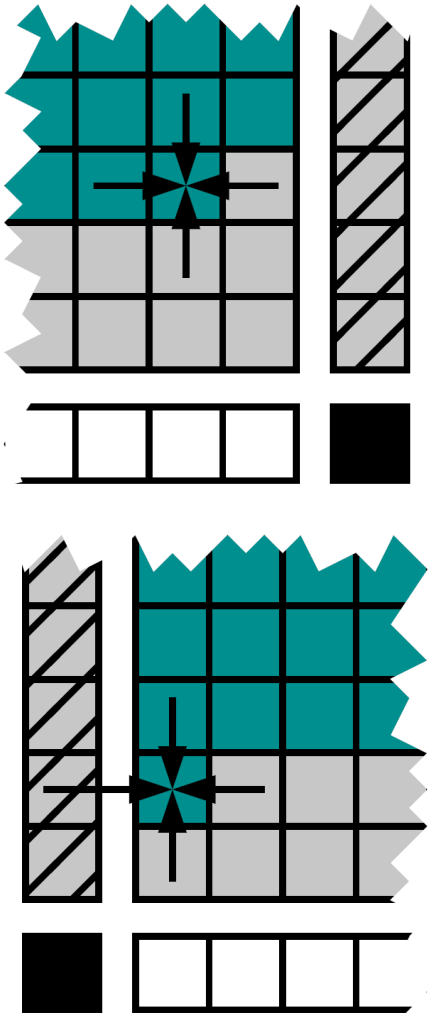
MPI_Send/Recv….
i.e., communicate all halo data

```
} 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

Three problems:

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

  ➔very hard to do !!!

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

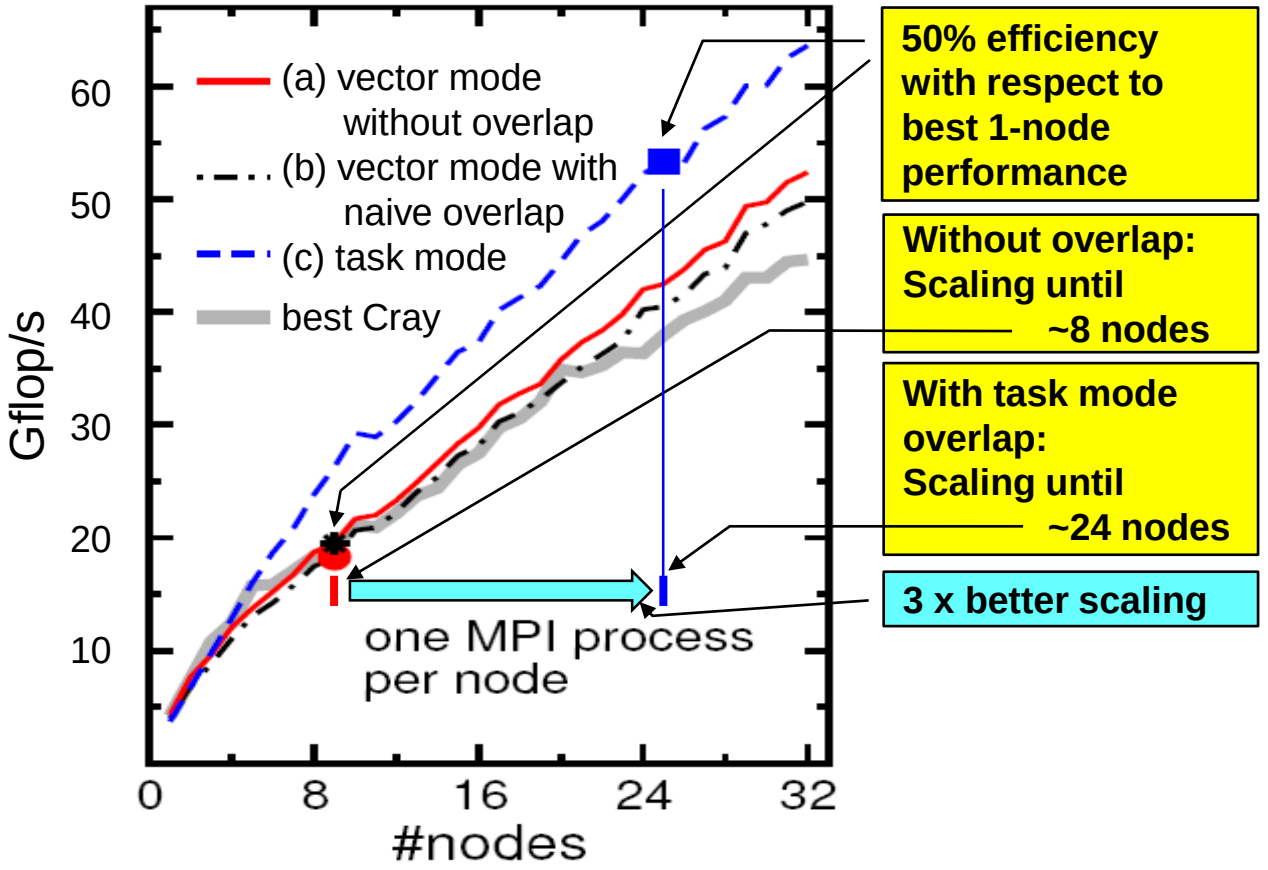
  ➔loss of major OpenMP support

- the load balancing problem

error-prone & clumsy

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

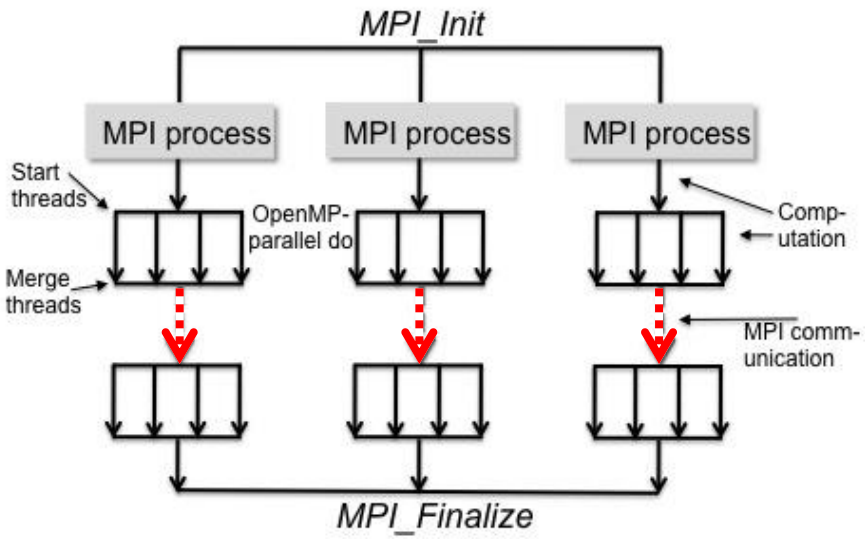
Example: sparse matrix-vector multiply (spMVM)

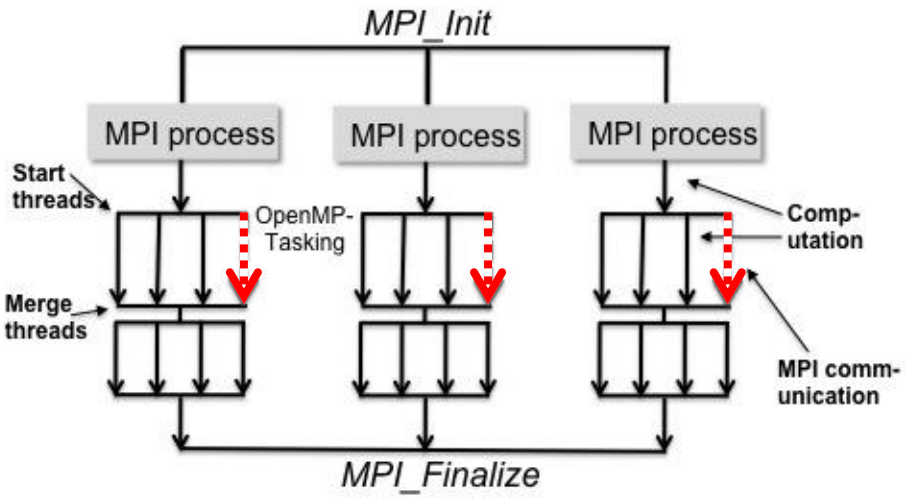


- spMVM on Intel Westmere cluster (6 cores/socket)
- "task mode" == explicit communication overlap using ded. thread
- "vector mode" == MASTERONLY
- "naïve overlap" == non-blocking MPI
- Memory bandwidth is already saturated by 5 cores

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

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

With OpenMP-4.5: New work-sharing directive: **taskloop**

OpenMP `taskloop` Directive – Syntax

A task can be run by any thread, across NUMA nodes → ☹ **perfect first touch impossible!**

- Immediately following loop executed in **several tasks.**
- **Not a work-sharing directive!**
- **→ Should be executed only by one thread!**

Fortran

- Fortran:

```
!$OMP taskloop [ clause [ [ , ] clause ] ... ]
    do_loop
[ !$OMP end taskloop [ nowait ] ]
```

Loop iterations must be independent, i.e., they can be executed in parallel

- If used, the `end do` directive must appear immediately after the end of the loop

C/C++

- C/C++:

```
#pragma omp taskloop [ clause [ [ , ] clause ] ... ] new-line
    for-loop
```

- The corresponding *for-loop* must have *canonical shape*
  → *next slide*

OpenMP `taskloop` Directive – Details

- *clause* can be one of the following:
  - `if([taskloop:] `*scalar-expr*`)` [a task clause]
  - `shared(`*list*`)` [a task clause]
  - `private(`*list*`), firstprivate(`*list*`)` [a do/for clause] [a task clause]
  - `lastprivate(`*list*`)` [a do/for clause]
  - `default(shared | none | …)` [a task clause]
  - `collapse(` *n* `)` [a do/for clause]
  - `grainsize(`*grain-size*`)`
  - `num_tasks(`*num-tasks*`)`

    Mutual exclusive
  - `untied, mergeable` [a task clause]
  - `final(` *scalar-expr* `), priority(` *priority-value* `)` [a task clause]

    final(true) → all children tasks from generated tasks will be sequentially included
  - `nogroup`
  - **`reduction (`*operator:list*`)`** [a do/for clause]

    Since OpenMP 5.0!
- *do/ for clauses that* are **not** valid on a taskloop:
  - `schedule(` *type* [, *chunk*] `), nowait`
  - `linear(`*list* [: *linear-step*] `), ordered [(` *n* `)]`

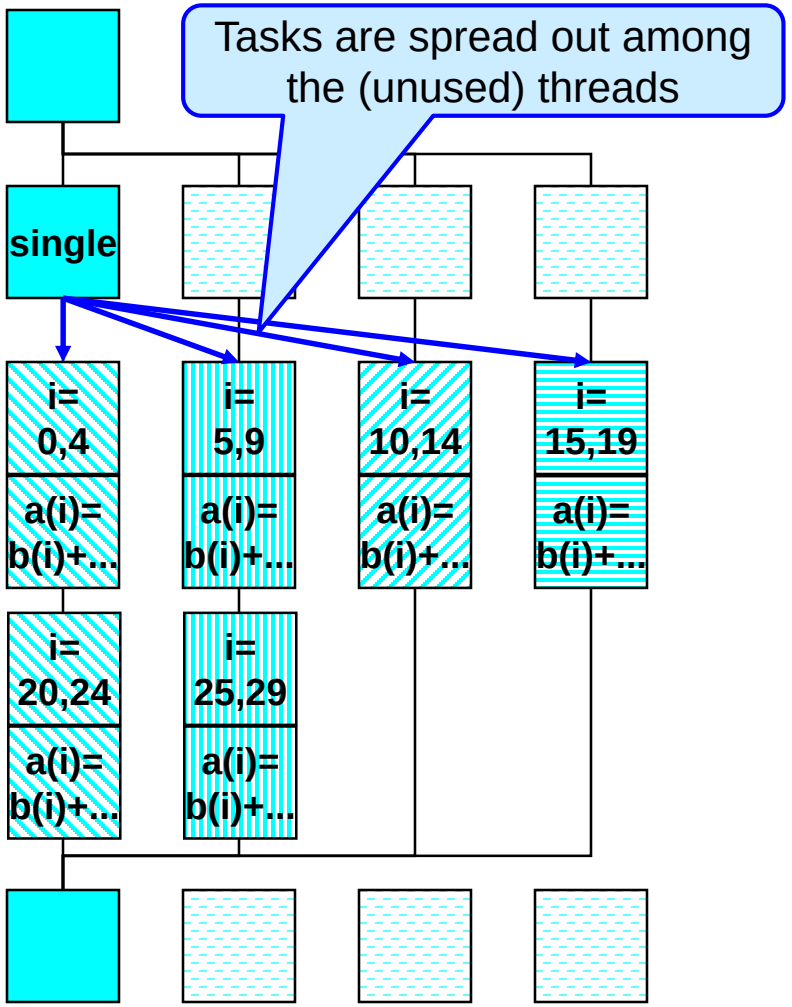
OpenMP `single` & `taskloop` Directives

C/C++

C / C++:

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskloop
      for (i=0; i<30; i++)
        a[i] = b[i] + f * (i+1);
  }
  } /*omp end single*/
} /*omp end parallel*/
```

A lot more tasks than threads may be produced to achieve a good load balancing



skipped

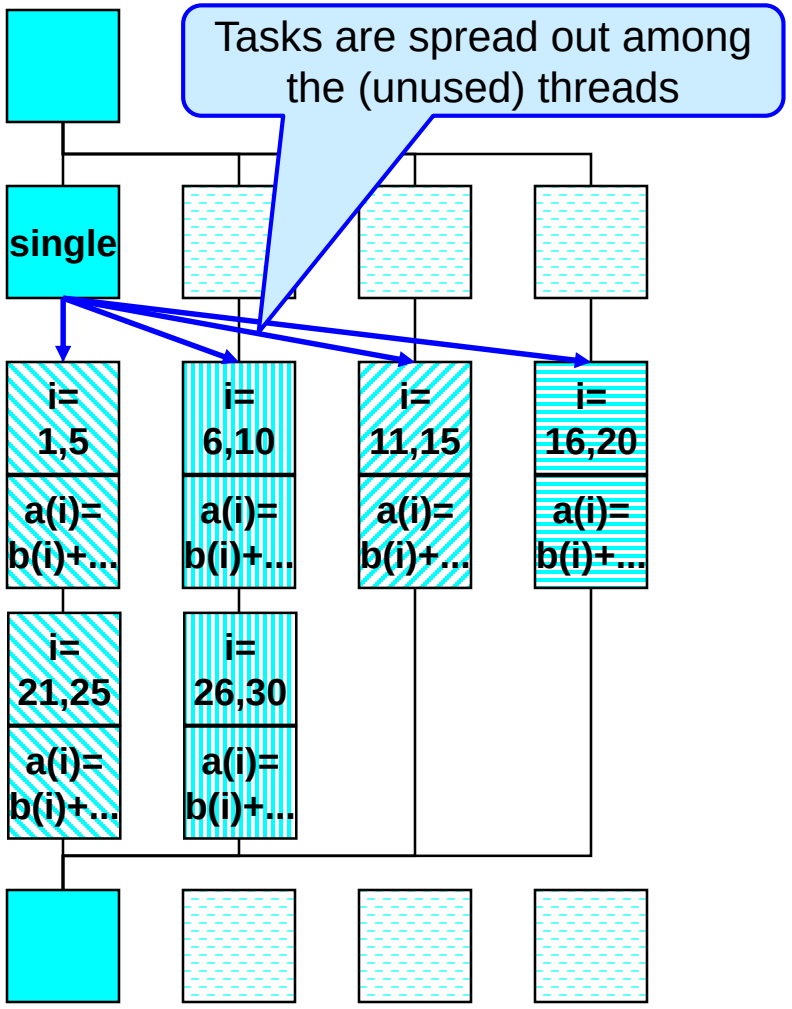
OpenMP `single` & `taskloop` Directives

Fortran

Fortran:

```
!$OMP PARALLEL
  !$OMP SINGLE
  !$OMP TASKLOOP
    do i=1,30
      a(i) = b(i) + f * i
    end do
  !$OMP END TASKLOOP
  !$OMP END SINGLE
!$OMP END PARALLEL
```

A lot more tasks than threads may be produced to achieve a good load balancing



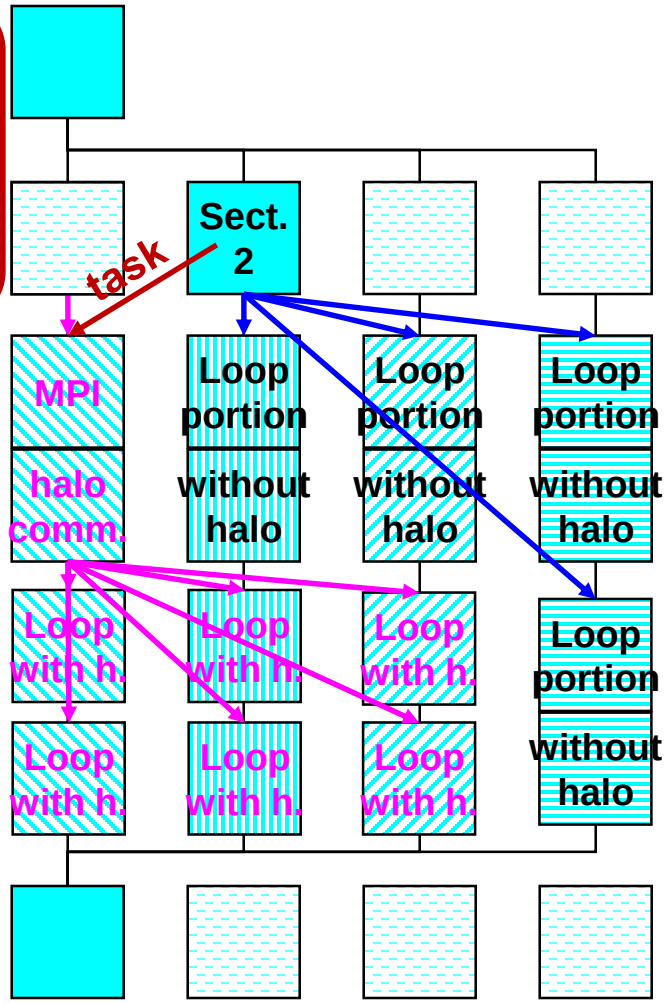
OpenMP `sections` & `taskloop` Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp sections        (single)
  {
    #pragma omp section       (task)
    {  // MPI halo communication:
       ….
       // numerical loop using halo data:
       #pragma omp taskloop
       for (i=0; i<100; i++)
          a[i] = b[i] +b[i-1]+b[i+1]+b[i-2]…;
    } /*omp end of "first" section*/
    #pragma omp section
    {  // numerical loop without using halo data:
       #pragma omp taskloop
       for (i=100; i<10000; i++)
          a[i] = b[i] +b[i-1]+b[i+1 ]+b[i-2]…;
       …
    } /*omp end of "second" section*/
  } /*omp end sections*/
} /*omp end parallel*/
```

Instead of sections, one can also use **single** and a **task** for the first section

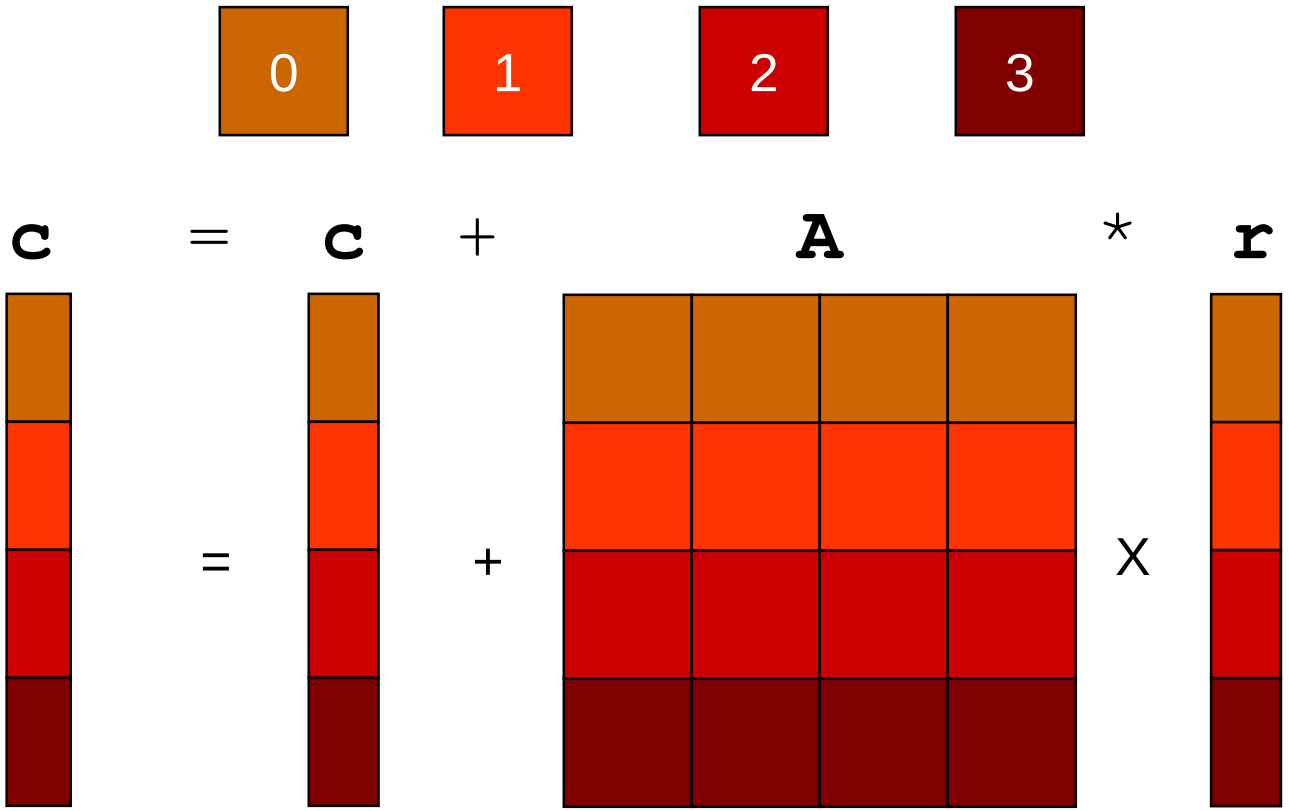
Number of tasks may be influenced with grainsize or num_tasks clauses



skipped

Tasking example: dense matrix-vector multiply with communication overlap

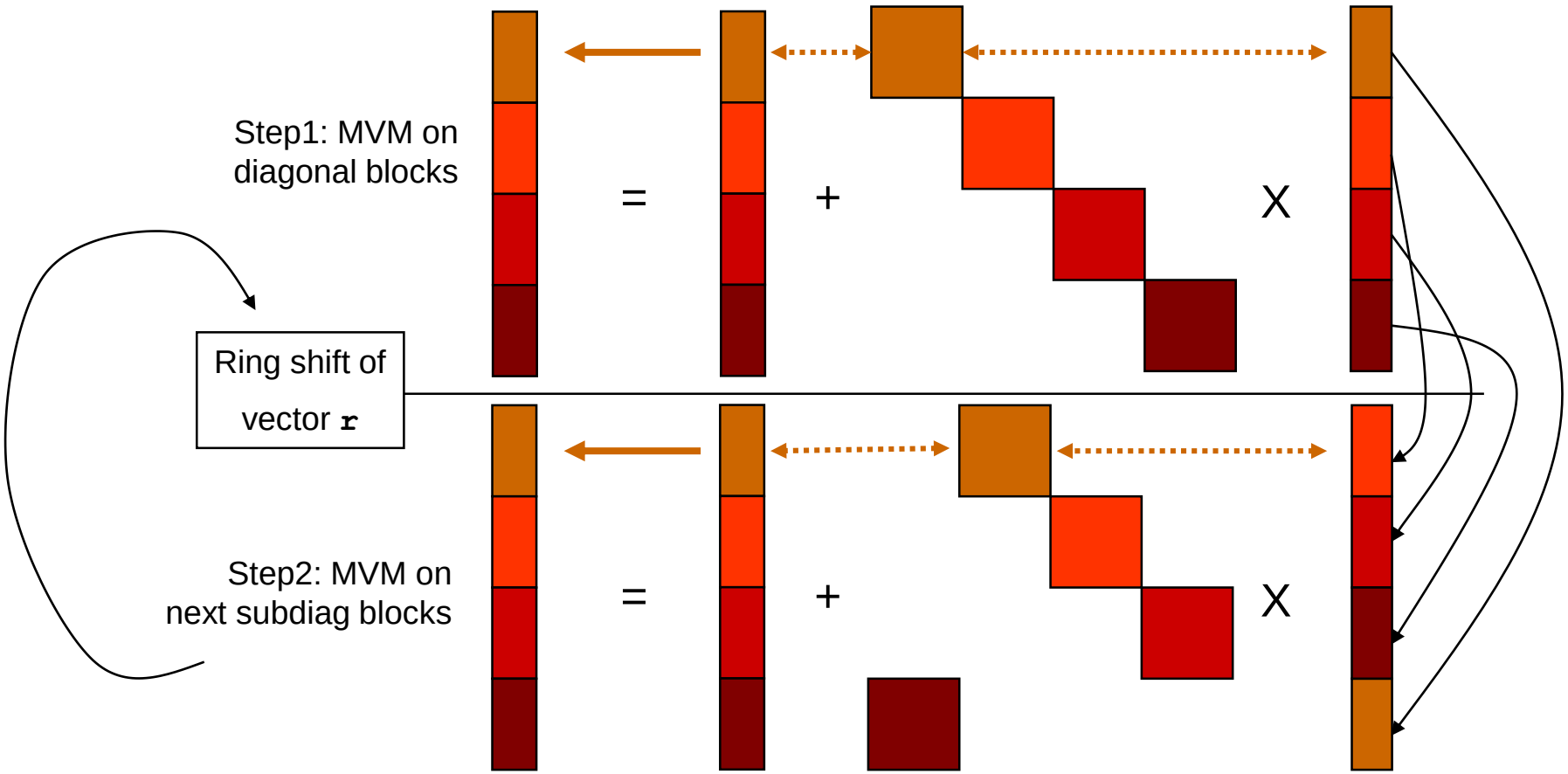
- Data distribution across processes:



skipped

Dense matrix-vector multiply with communication overlap via tasking

- Computation/communication scheme:



skipped

Dense matrix-vector multiply with communication overlap via tasking

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

Asynchronous communication (ring shift)

Current block of MVM (chunked by 4 rows)

New in MPI-4.0

Partitioned Point-to-Point Communication

- MPI-4.0:
  Partitioned communication is "partitioned" because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.
- A point-to-point operation (i.e., send or receive)
  - can be split into partitions,
  - and each partition is filled and then "send" with MPI_Pready by a thread;
  - And same for receiving.
- Technically provided as a new form of persistent communication.

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 NUMA domain per MPI process*
  - Alternative solution:
    One MPI process on each NUMA domain (i.e., chip)
- **Multicore / multisocket anisotropy effects**
  - Bandwidth bottlenecks, shared caches
  - Intra-node MPI performance
    - Core ↔ core vs. socket ↔ socket
    - OpenMP loop overhead
- **Amdahl's law** on both, MPI and OpenMP level
- Thread and process **pinning**
- **Other disadvantages through OpenMP**

Additional disadvantages when overlapping communication and computation:

- High programming overhead
- OpenMP is only partially prepared for this programming style → taskloop directive

Programming models
- MPI + OpenMP

Example / Exercise

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Example: MPI+OpenMP-Hybrid Jacobi solver

- Source code: See http://tiny.cc/MPIX-VSC LRZ

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

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

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

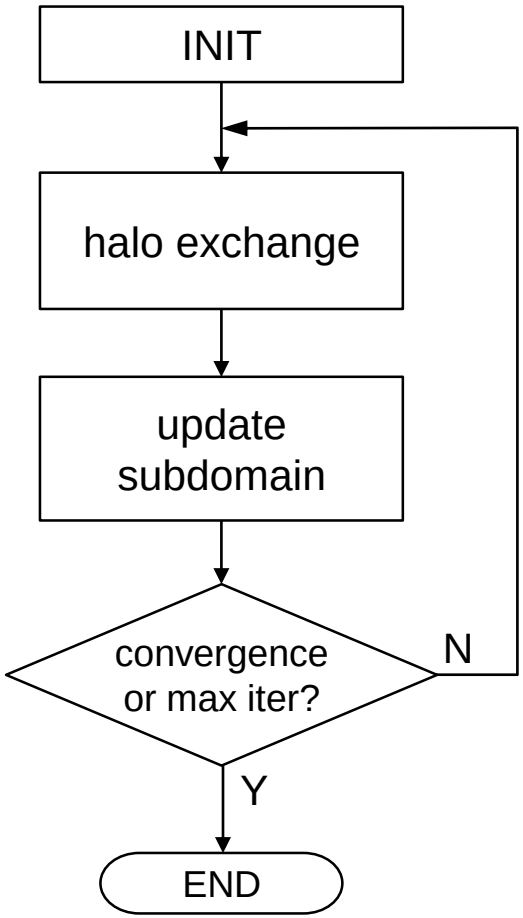
- Notes:
  - The code is strongly memory bound at the problem size set in the input file
  - Learn how to take control of affinity with MPI and especially with MPI+OpenMP
  - Always run multiple times and observe performance variations
  - If you know how, try to calculate the maximum possible performance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

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Example cont'd

- Tasks (we assume N_c cores per CPU socket):
  - Run the MPI-only code on one node with $1,\ldots,N_c,\ldots,2*N_c$ processes (1 full node) and observe the achieved performance behavior
  - Parallelize appropriate loops with OpenMP
  - Run with OpenMP and 1 MPI process ("OpenMP-only") on $1,\ldots,N_c,\ldots,2*N_c$ cores, compare with MPI-only run
  - Run hybrid variants with different MPI vs. OpenMP ratios
- Things to observe
  - Run-to-run performance variations
  - Does the OpenMP/hybrid code perform as well as the MPI code? If it doesn't, fix it!



http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

see also login-slides

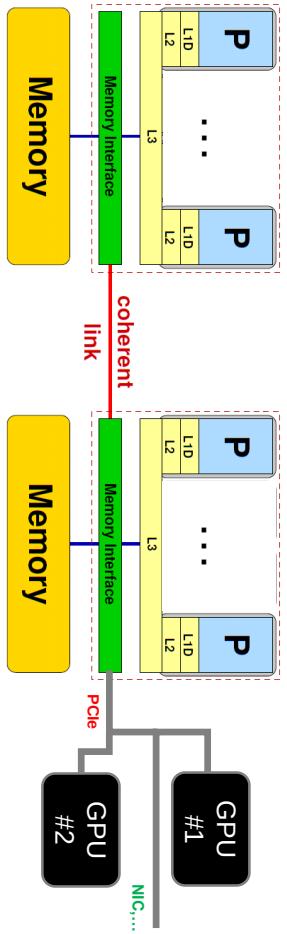
Programming models
- MPI + Accelerator

Parts
Courtesy of Gabriele Jost

Accelerator programming: Bottlenecks reloaded

Example: 2-socket Intel “Ice Lake” (2x36 cores) node with two NVIDIA A100 GPGPUs (PCIe 4)

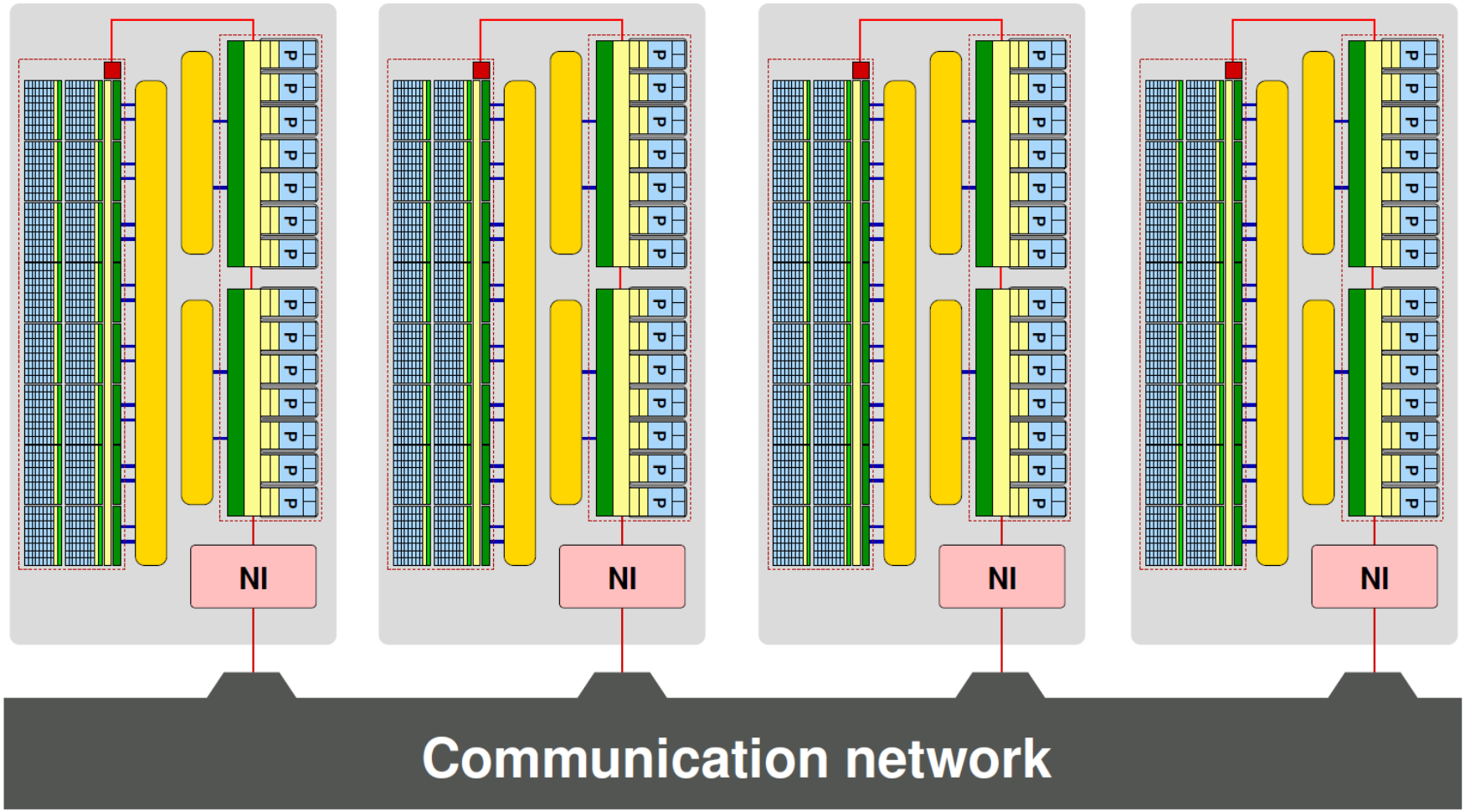
	per GPGPU	per CPU
DP peak performance	9.7 Tflop/s ⟵ 4x	2.3 Tflop/s
Machine balance	0.11 B/F	0.10 B/F
eff. memory (HBM) bandwidth	1300 Gbyte/s ⟵ 8x	170 Gbyte/s
inter-device bandwidth (PCIe)	≈ 30 Gbyte/s	
inter-device bandwidth (NVlink)	> 500 Gbyte/s	



→ Speedups can only be attained if communication overheads are under control

→ Basic estimates help

Accelerator + MPI: How does the data get from A to B?



Questions to ask

- Is the MPI implementation CUDA aware?
  - Yes: Can use device pointers in MPI calls
  - No: Explicit DtoH/HtoD buffer transfers required
  - Copying to consecutive halo buffers may still be necessary
- Is NVLink available?
  - Yes: Direct GPU-GPU MPI communication with MPI
    - **Supported by: P100, V100, A100, H100**
  - No: copies via host (even with NVIDIA GPUDirect)
- Unified Memory or explicit DtoH/HtoD transfers?
  - UM: Transparent sharing of host and device memory
- Actual bandwidths and latencies?
  - Highly system and implementation dependent!

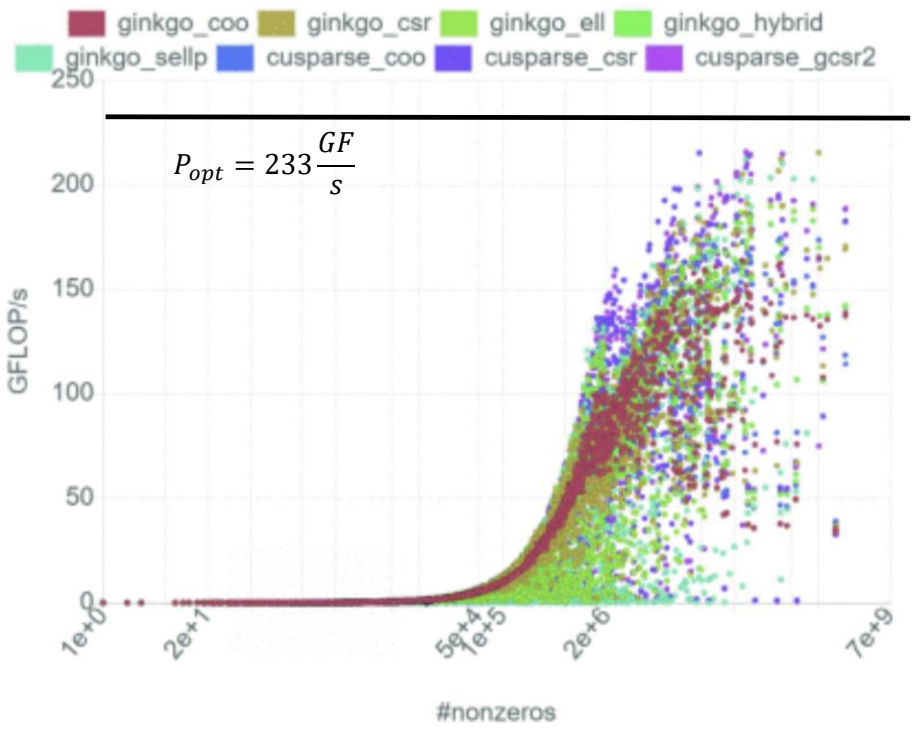
- https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/msa-seminar/2020-01-21-CUDA-aware-MPI.pdf

Never forget: hardware is not enough

SpMV on NVIDIA A100:

- Different data formats and libraries
- 2800 matrices (SuiteSparse Matrix collection)

Optimal matrix storage format is highly matrix and system dependent!



(a) SpMV performance profile on A100.SpMV

H. Anzt, et al; *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, GA, USA, 2020, pp. 26-38, DOI: 10.1109/PMBS51919.2020.00009.

Options for hybrid accelerator programming

multicore host
MPI
MPI+MPI3 shmem ext.
MPI+threading (**OpenMP**, pthreads, TBB,…)
threading only
PGAS (CAF, UPC,…)
…

accelerator
CUDA
OpenCL
OpenACC
OpenMP 4.0++
special purpose
…

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

Programming models - MPI + Accelerators

OpenMP support for co-processors

skipped

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

skipped

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, ….]

skipped

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

Programming models - MPI + Accelerators

OpenACC

skipped

What is OpenACC?

- API that supports offloading 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

A very simple OpenACC example (PGI 14.10): Schönauer Vector Triad

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

data mgmt

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

execution

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

skipped

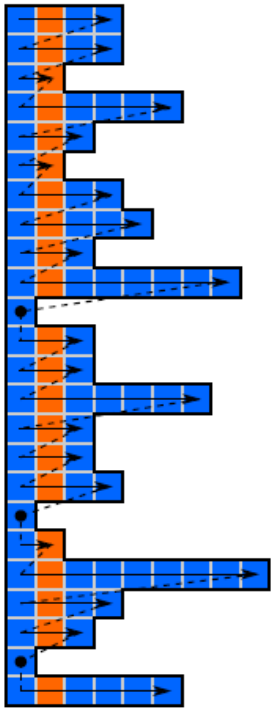
Example: 2D Jacobi smoother

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

skipped

Example: Sparse MVM (std. CSR format)

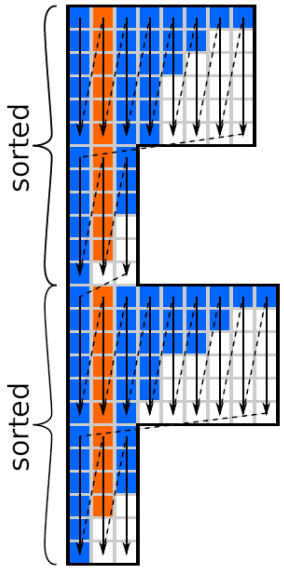
```
#pragma acc parallel present(val[0:numNonZeros], \
  colInd[0:numNonZeros],      \
  rowPtr[0:numRows+1],        \
  x[0:numRows],               \
  y[0:numRows])               \
  loop
for (int rowID=0; rowID<numRows; ++rowID)    {
  double tmp = y[rowID];
  // loop over all elements in row
  for (int rowEntry=rowPtr[rowID];
       rowEntry<rowPtr[rowID+1];
       ++rowEntry) {
    tmp += val[rowEntry] * x[ colInd[rowEntry] ];
  }
  y[rowID] = tmp;
}
```



skipped

Example: Sparse MVM (SELL-C-σ format)

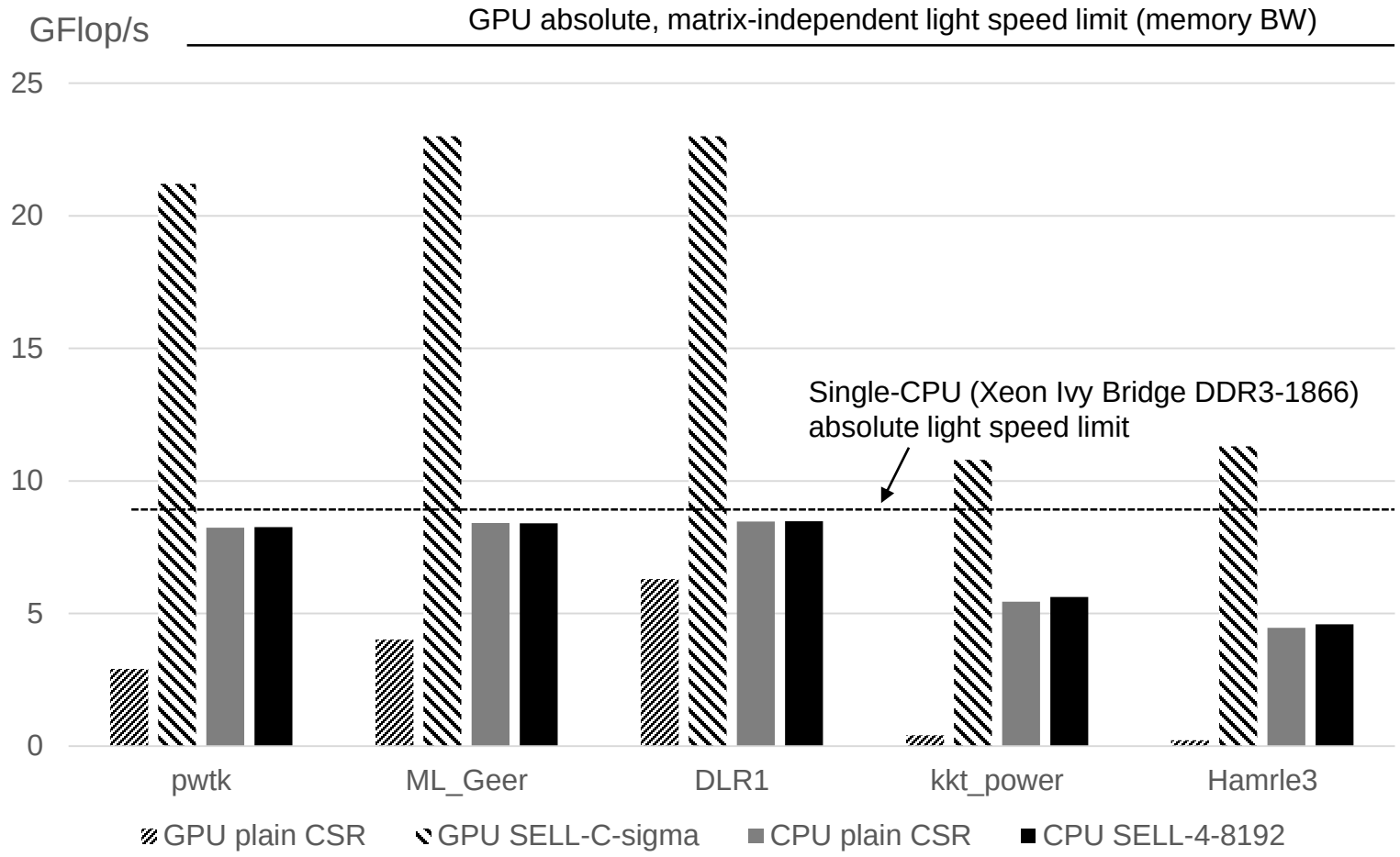
```
#pragma acc parallel present(val[0 : capacity],colInd[0 : capacity],\
    chunkPtr[0 : numberOfChunks], chunkLength[0 : numberOfChunks],  \
    x[0 : paddedRows],y[0 : paddedRows]) vector_length(chunkSize) loop
// loop over all chunks
for (int chunk=0; chunk < numberOfChunks; ++chunk) {
  int chunkOffset = chunkPtr[chunk];
  int rowOffset   = chunk*chunkSize;
  #pragma acc loop vector
  for (int chunkRow=0; chunkRow<chunkSize; ++chunkRow) {
    int globalRow = rowOffset + chunkRow;
    // fill tempory vector with values from y
    double tmp = y[globalRow];
    // loop over all row elements in chunk
    for (int rowEntry=0;
         rowEntry<chunkLength[chunk];
         ++rowEntry) {
      tmp += val [chunkOffset + rowEntry*chunkSize + chunkRow]
           * x[colInd[chunkOffset + rowEntry*chunkSize + chunkRow] ];
    }
    // write back result of y = alpha Ax + beta y
    y[globalRow] = tmp;
  }
}
```



M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units.* SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: 10.1137/130930352

skipped

Example: Sparse MVM CRS vs. SELL-128-8192 on Kepler K20 for different matrices



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 4.0/4.5 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
- Performance is not only about code; **data structures** can be decisive as well
- Not many compilers support accelerator pragmas (yet)

Programming models - MPI + MPI-3.0 shared memory

Major prob… cluster of ccNUMA nodes?

Where are we?

MAJOR PROBLEMS

- …usual programming models do **not** really fit: **Pure MPI / Hybrid MPI+OpenMP**
- Where are major problems? – And how to address?
  - Significant differences between **(small) inter-node bandwidth** and (high) intra-node bandwidth
    - → **How to minimize inter-node communication**
      - Hybrid MPI + OpenMP
      - Pure MPI + optimized virtual topologies
  - **Replicated user data**
    - → **Waste of memory with pure MPI**
      - Hybrid MPI + OpenMP
      - Pure MPI + MPI-3 shared memory model
  - Processes and threads may **move** between cores and CPUs within ccNUMA nodes & fixed memory locations after ***"first touch"***
    - → **How to keep control?**
      - Pinning of threads and processes to the hardware
      - Explicit first touch programming with OpenMP, …
  - Are your application AND libraries **prepared for** MPI+**OpenMP?**

Minor use-case:
Fast inner-node *"communication"*

An OpenMP-free alternative for replicated data

Hybrid MPI + MPI-3 shared memory

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

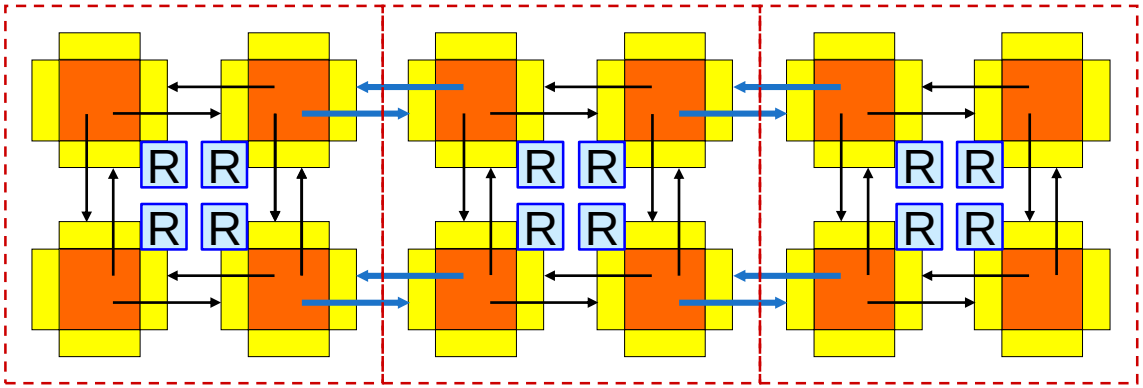
Advantages

- Simple method for reduced memory needs for replicated data
- No message passing inside of the SMP nodes
- Using only one parallel programming standard
- No OpenMP problems (e.g., thread-safety isn't an issue)

Major Problems

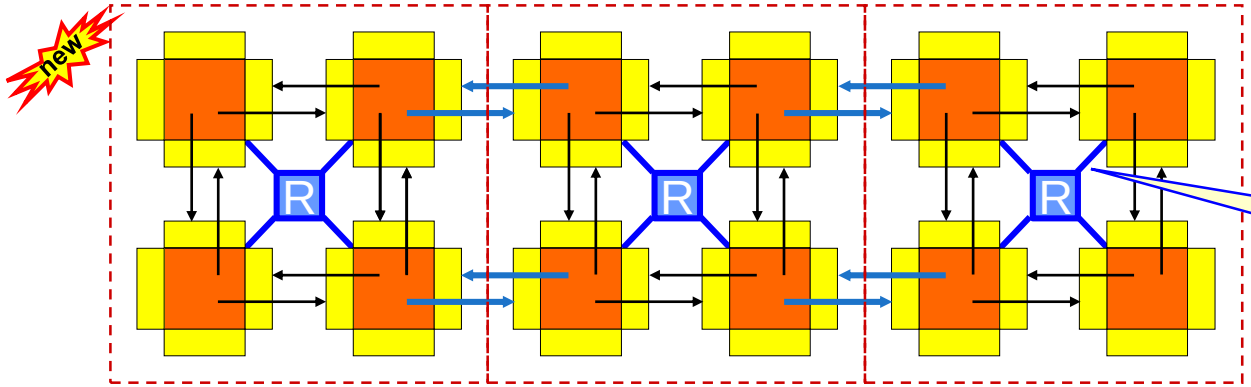
- Communicator must be split into shared memory islands
- To minimize shared memory communication overhead:
  Halos (or the data accessed by the neighbors) must be stored in
  MPI shared memory windows
- Same work-sharing as with pure MPI communication
- MPI-3.0/3.1 shared memory synchronization waits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

Programming opportunities with MPI shared memory: 1) Reducing memory space for replicated data



R = Replicated data in each MPI process

Example:
Cluster of SMP nodes **without** using MPI shared memory methods



R = Shared memory → replicated data only once within each SMP node

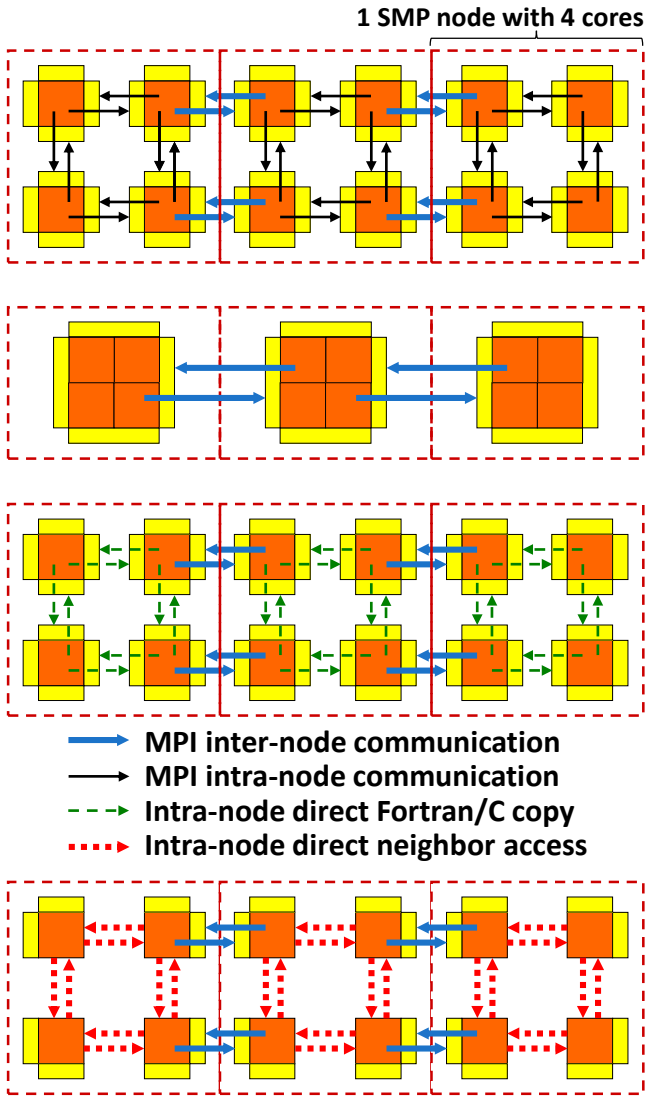
Direct loads & stores, no library calls

Using MPI shared memory methods

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

2) 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



Programming models
- MPI + MPI-3.0 shared memory

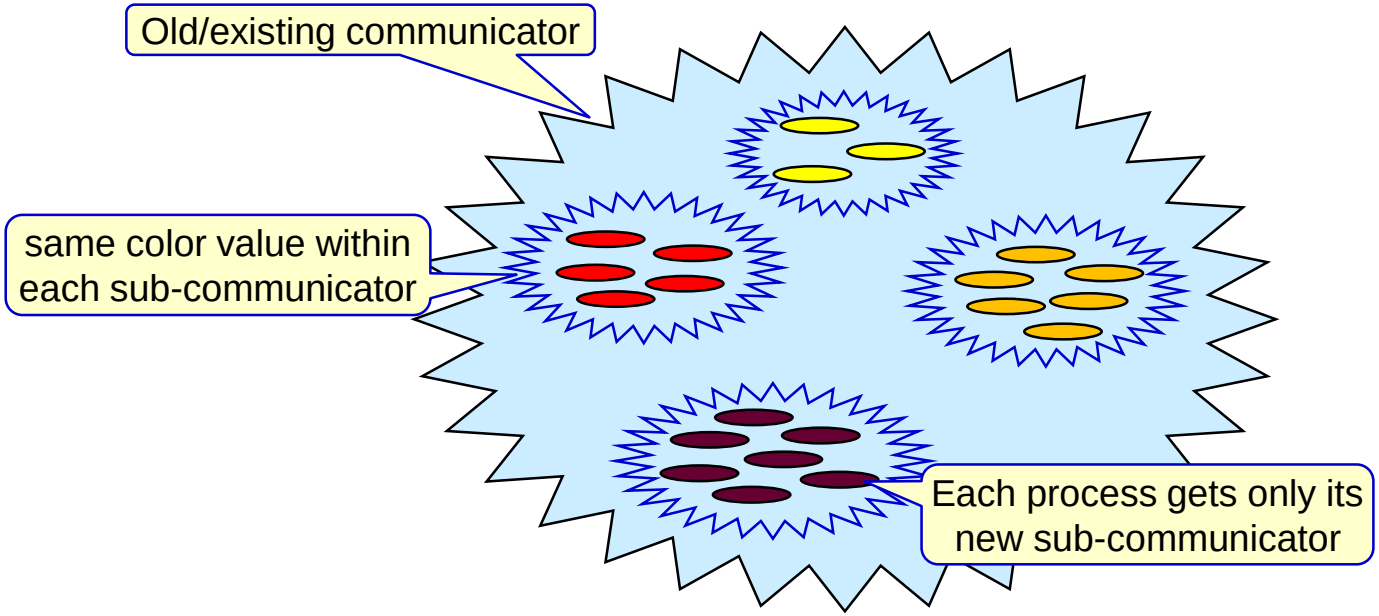
Re-cap

- MPI_Comm_split
- One-sided communication

Re-cap: New sub-communicators with MPI_Comm_split

- New sub-communicators via MPI_Comm_split
  - Each process must specify a color
  - Processes with same color are put together in new sub-communicators

& MPI_Comm_split_type (New in MPI-3.0) → shared memory



Example: MPI_Comm_split()

All processes with same color are grouped into separate sub-communicators

Creation is **collective** in the **old** communicator.

C

- C/C++: int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

Each process gets only its own sub-communicator

Fortran

- Fortran: MPI_COMM_SPLIT (comm, color, key, newcomm, ierror)

  mpi_f08:
  ```
  TYPE(MPI_Comm) :: comm, newcomm
  INTEGER :: color, key;
  INTEGER, OPTIONAL :: ierror
  ```

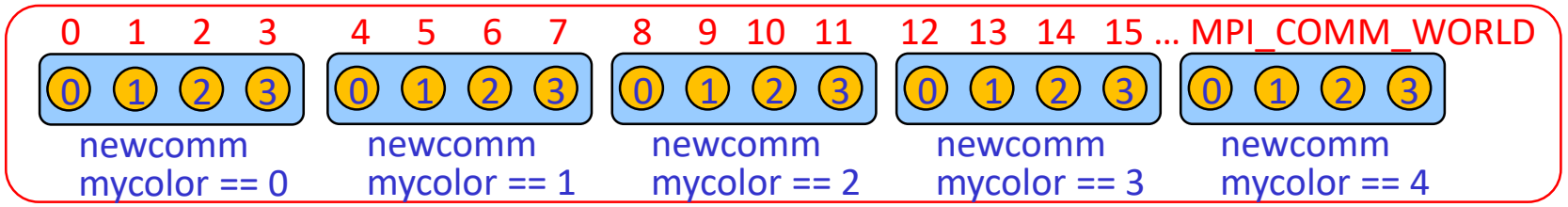
  mpi & mpif.h: INTEGER comm, color, key, newcomm, ierror

Example:

```
int my_rank, mycolor, key, my_newrank;
MPI_Comm newcomm;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
mycolor = my_rank/4;
key = 0;
MPI_Comm_split(MPI_COMM_WORLD, mycolor, key, &newcomm);
MPI_Comm_rank (newcomm, &my_newrank);
```

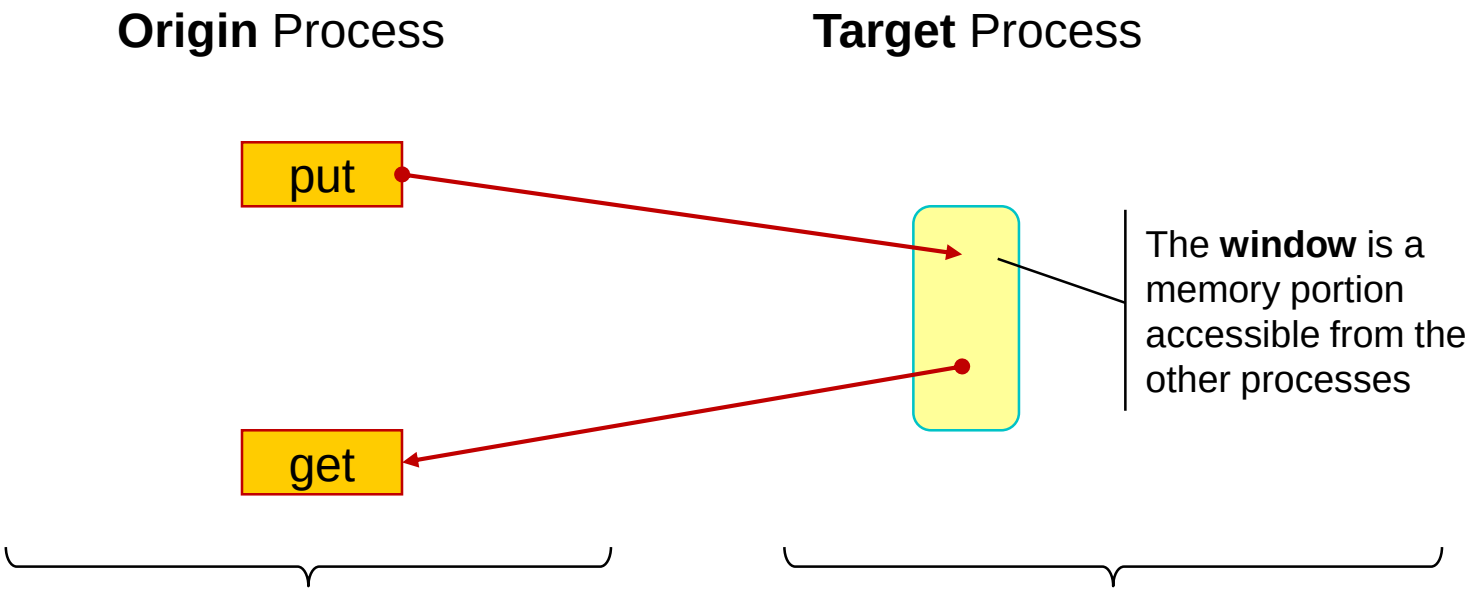
Always 4 process get same color → grouped in an own newcomm

key==0 → ranking in newcomm is sorted as in old comm
key ≠ 0 → ranking in newcomm is sorted according key values



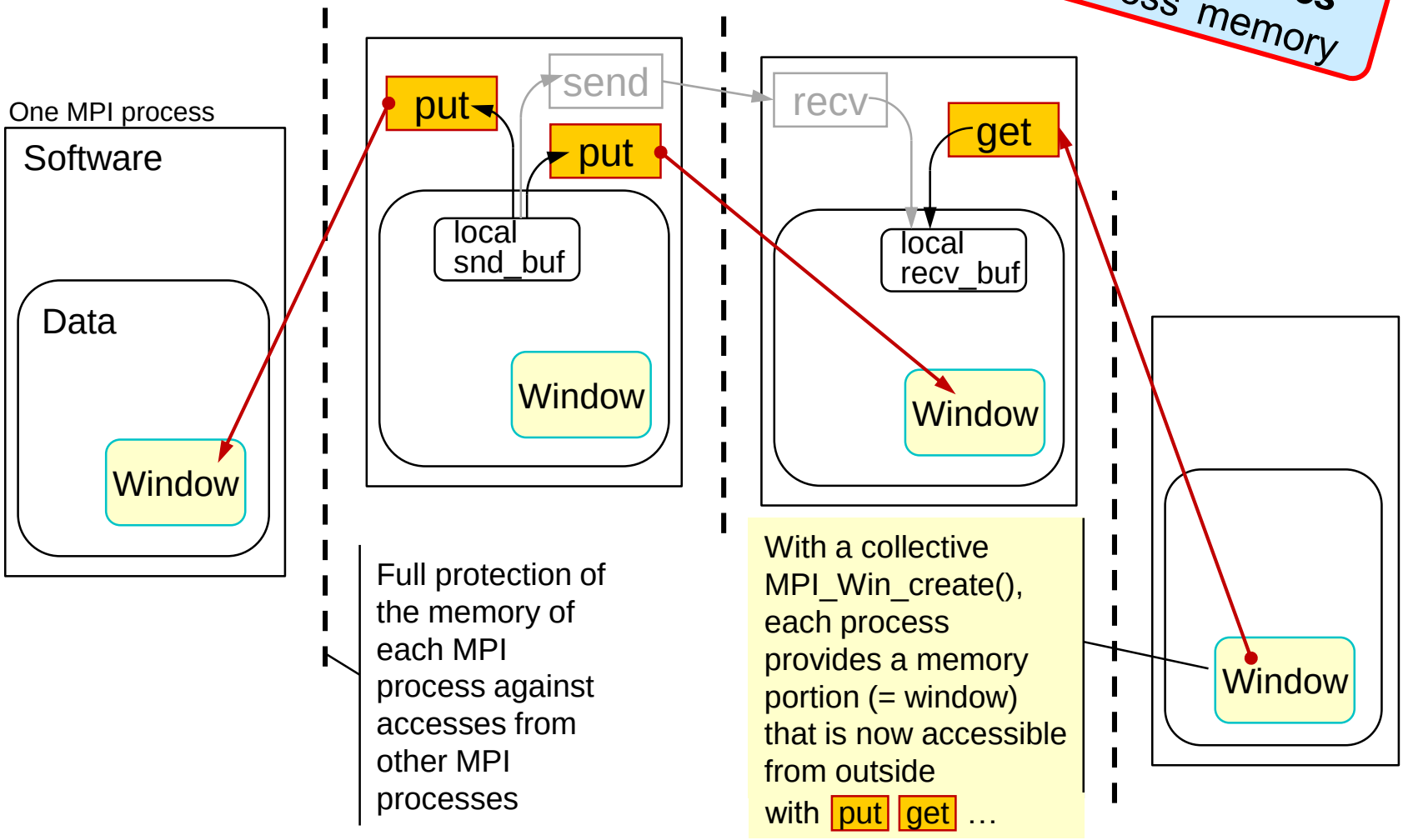
Re-cap: One-sided Communication

- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses



Typically, all processes are both, origin and target processes

Windows are **peepholes** into their process' memory



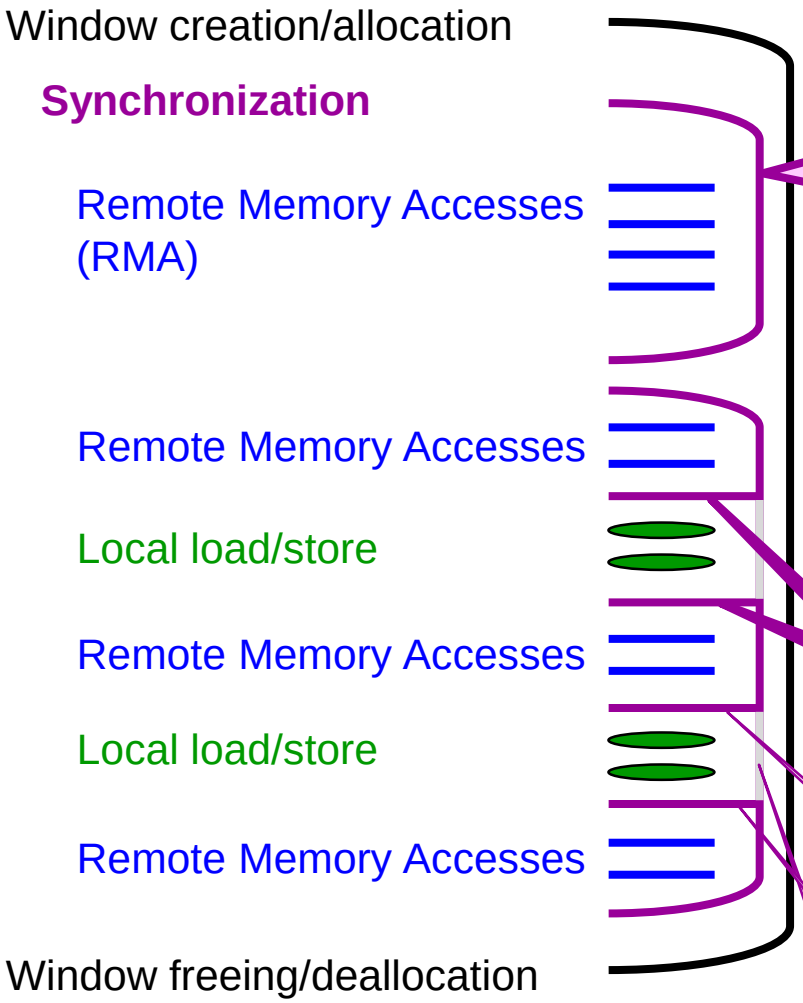
One-sided Operations

Three major sets of routines:

- Window creation or allocation
  - Each process in a group of processes **(defined by a communicator)**
  - defines a chunk of own memory – named ***window***,
  - which can be afterwards accessed by all other processes of the group.
- **R**emote **M**emory **A**ccess (RMA, nonblocking) routines
  - Access to remote windows:
    - **put, get, accumulate, …**
- Synchronization
  - The RMA routines are nonblocking and
  - must be surrounded by synchronization routines,
  - which guarantee
    - **that the RMA is locally and remotely finished**
    - **and that all necessary cache operation are implicitly done.**

Shared memory:
direct **loads** and **stores**
instead of MPI_Put/Get

Sequence of One-sided Operations



RMA operations must be surrounded by **synchronization** calls

To start and finish **exposure** and **access** epochs

RMA epoch

Local load/store epoch

Local load/store epochs must be separated from RMA epochs by **synchronization** calls

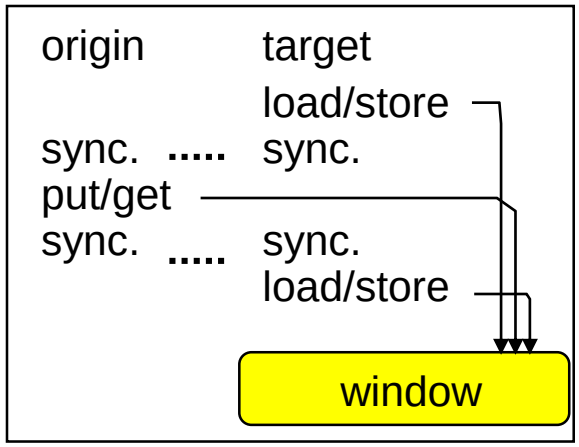
It looks like that
- local and RMA epochs are separated by synchronization calls, and
- local load/store epochs are also surrounded by synchronizations.

But correct is: only RMA epochs must be surrounded by synchronizations.

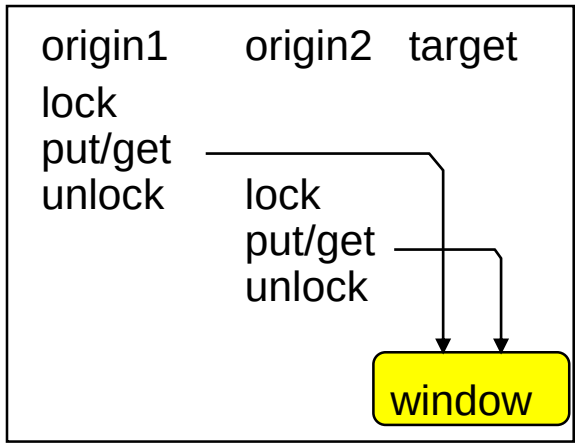
Synchronization Calls (1)

- Active target communication
  - communication paradigm similar to message passing model
  - target process participates only in the synchronization
  - fence or post-start-complete-wait

MPI_Win_fence is like a barrier



- Passive target communication
  - communication paradigm closer to shared memory model
  - only the origin process is involved in the communication
  - lock/unlock



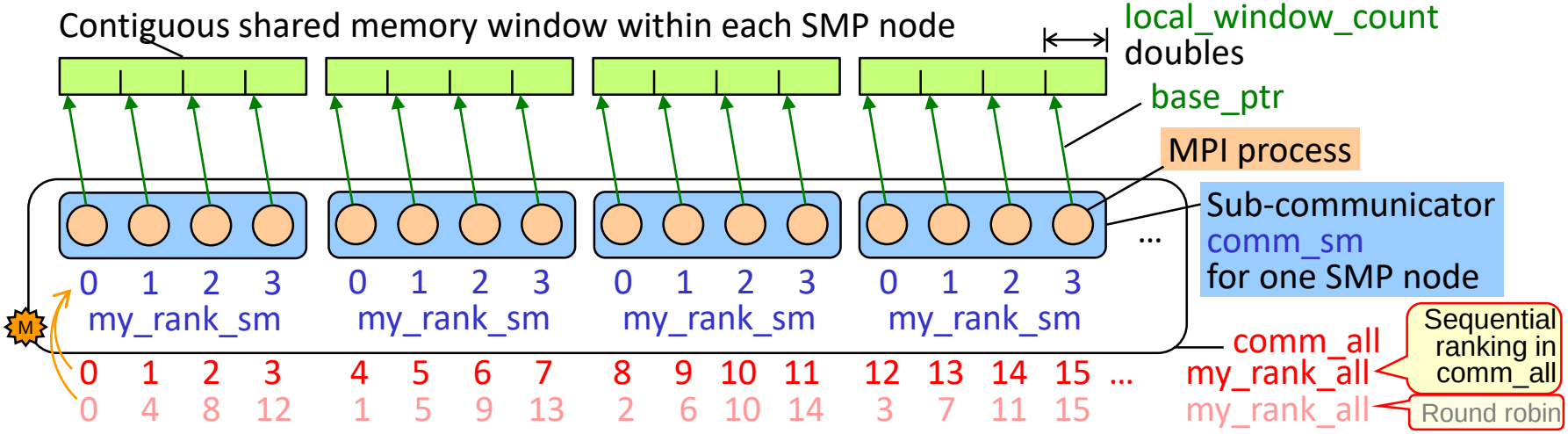
Programming models
- MPI + MPI-3.0 shared memory

How-To

MPI 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 sychronization
  - Normal assignments and expressions
  - No **MPI_PUT/GET** !
  - Normal MPI one-sided synchronization, e.g., **MPI_WIN_FENCE**
- Caution:
  - Memory may be already completely pinned to the physical memory of the process with rank 0, i.e., the first touch rule (as in OpenMP) does **not** apply!
    (First touch rule: a memory page is pinned to the physical memory of the processor that first writes a byte into the page)

Splitting the communicator & contiguous shared memory allocation




```
MPI_Aint /*IN*/ local_window_count=10;  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 ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                         MPI_INFO_NULL,  comm_sm,  &base_ptr,  &win_sm);
```

collective call

collective call

Sequence in comm_sm as in comm_all

Caution: If local_window_count is 0, some MPI libraries return a null pointer instead of pointing to next process' base.

M This mapping is based on the ranking in comm_all.

New in MPI-3.0

In all three Fortran support methods

All Memory Allocation with modern C-Pointer

C

```
float *buf;  MPI_Win win; int max_length; max_length = …;
MPI_Win_allocate_shared( (MPI_Aint)(max_length*sizeof(float)),  sizeof(float),
                                                  MPI_INFO_NULL,  comm_shm,  &buf,  &win);
// the window elements are buf[0] .. buf[max_length-1]
```

Fortran

```
USE mpi_f08
USE, INTRINSIC :: ISO_C_BINDING

INTEGER :: max_length,  disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: lb, size_of_real
REAL, POINTER, ASYNCHRONOUS :: buf(:)
TYPE(MPI_Win) :: win
INTEGER(KIND=MPI_ADDRESS_KIND) :: buf_size, target_disp
TYPE(C_PTR) :: cptr_buf

max_length = …

CALL MPI_Type_get_extent(MPI_REAL, lb, size_of_real)
buf_size = max_length * size_of_real
disp_unit = size_of_real
CALL MPI_Win_allocate_shared(buf_size, disp_unit, MPI_INFO_NULL, comm_shm,
                                                  cptr_buf, win)
CALL C_F_POINTER(cptr_buf, buf, (/max_length/) )
buf(0:) => buf    ! With this code, one may change the lower bound to 0 (instead of default 1)
! The window elements are buf(0) .. buf(max_length-1)
```

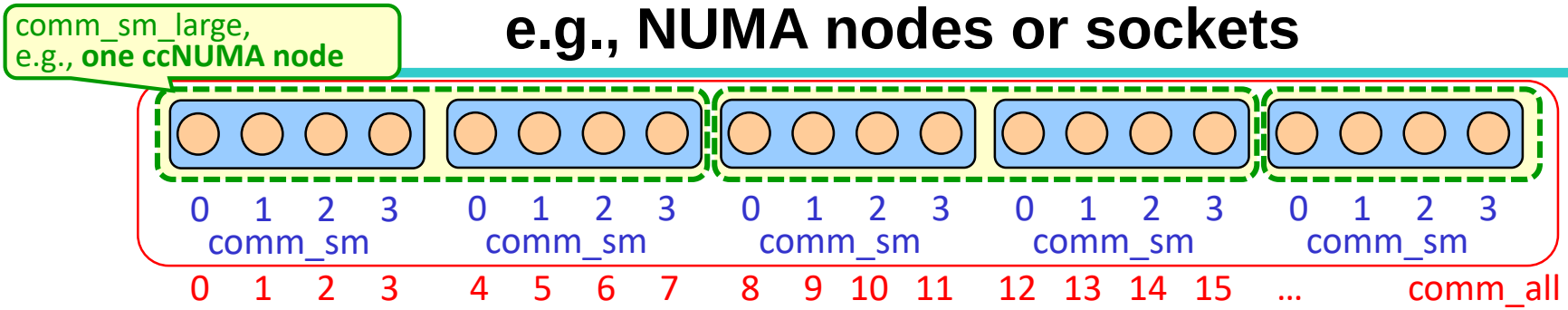
Fortran for Scientific Computing – a course in FutureLearn, a good Intro to Fortran / but without C_F_POINTER.
Trailer: https://www.youtube.com/watch?v=l6pEaUttWo8
By Geert Jan Bex et al – have fun with it ☺

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.

Splitting into smaller shared memory islands, e.g., NUMA nodes or sockets



- Subsets of shared memory nodes, e.g., one comm_sm on each socket with size_sm cores **(requires also sequential ranks in comm_all for each socket!)**

```
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm_large);
MPI_Comm_rank (comm_sm_large, &my_rank_sm_large); MPI_Comm_size (comm_sm_large, &size_sm_large);
MPI_Comm_split (comm_sm_large, /*color*/ my_rank_sm_large / size_sm,  0, &comm_sm);
MPI_Win_allocate_shared (…, comm_sm, …);
```

or (size_sm_large /number_of_sockets)

here 2

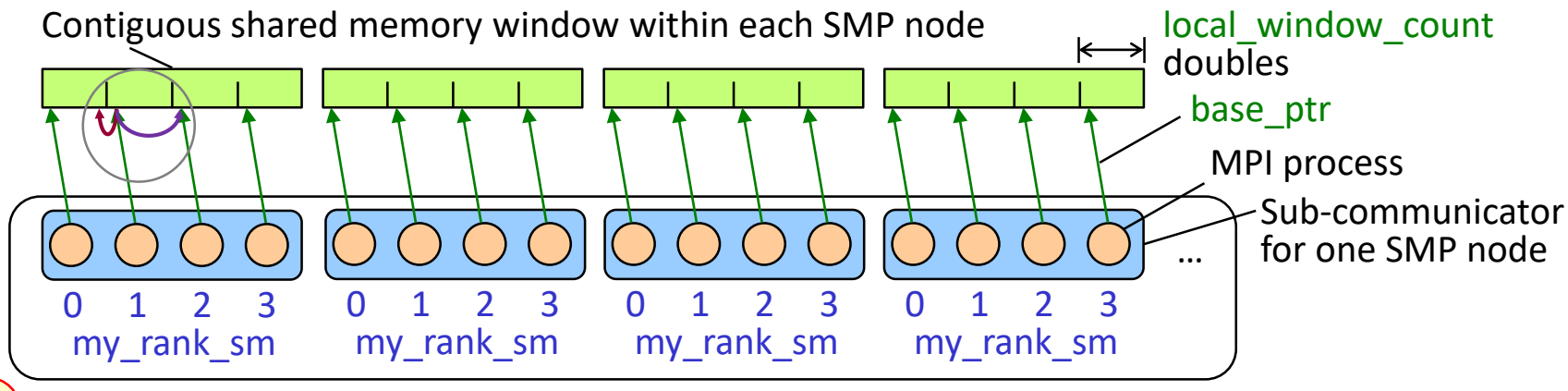
- Most MPI libraries have an non-standardized method to split a communicator into NUMA nodes (e.g., sockets): (see also Current support for split types in MPI implementations or MPI based libraries)
  - **OpenMPI:** choose split_type as OMPI_COMM_TYPE_NUMA
  - **HPE**: MPI_Info_create (&info); MPI_Info_set(info, "shmem_topo", "numa"); // or "socket"
    MPI_Comm_split_type(comm_all, MPI_COMM_TYPE_SHARED, 0, info, &comm_sm);
  - **mpich:** split_type=MPIX_COMM_TYPE_NEIGHBORHOOD, info_key= "SHMEM_INFO_KEY" and value= "machine", "socket", "package", **"numa"**, "core", "hwthread", "pu", "l1cache", … or "l5cache"

New in MPI-4.0

- Two additional standardized split types:
  - MPI_COMM_TYPE_HW_GUIDED and
  - MPI_COMM_TYPE_HW_UNGUIDED
- See also Exercise 3.

May not work with Intel-MPI

Shared memory access example



Recommendation: **No assertions**, because not clearly defined for shared memory windows

```
MPI_Aint /*IN*/ local_window_count;      double /*OUT*/  *base_ptr;
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                         MPI_INFO_NULL, comm_sm,  &base_ptr,  &win_sm);
```

Synchroni-zation

```
MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
```

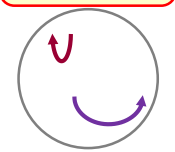
Local stores

```
for (i=0; i<local_window_count; i++)  base_ptr[i] = ... /* fill values into local portion */
```

Synchroni-zation

```
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] );
```

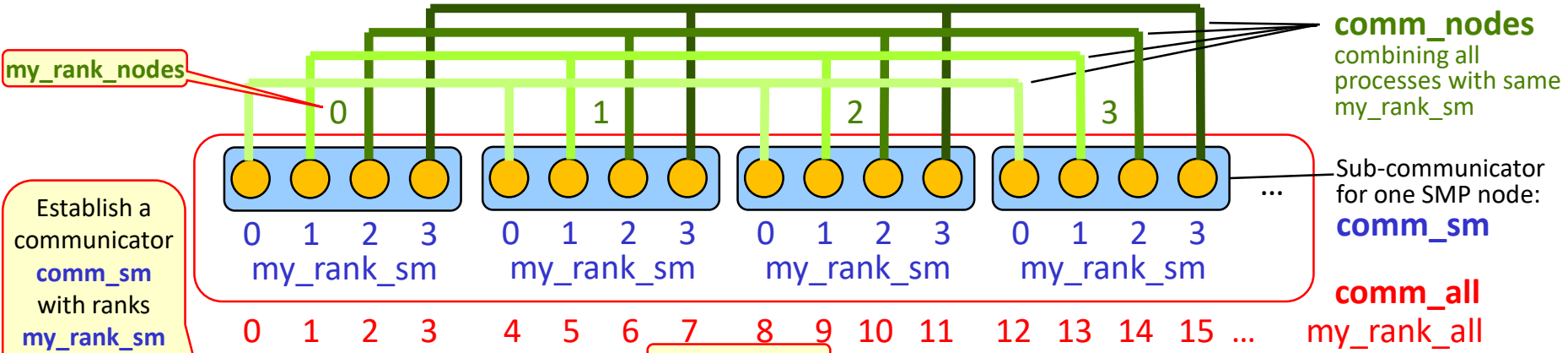
Direct load access to remote window portion



In Fortran, before and after the synchronization, on must add: CALL MPI_F_SYNC_REG (buffer)
to guarantee that register copies of buffer are written back to memory, respectively read again from memory.
The buffer should be declared as ASYNCHRONOUS, see course Chapter 10, slide "Fortran Problems with 1-Sided".

skipped

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



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

Input

```
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.

Exscan does not return value on the first rank, therefore

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

my_rank_nodes is not identical to the rank in comm_nodes if node sizes are not identical

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

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 NUMA 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

NUMA effects?
Significant impact of alloc_shared_noncontig

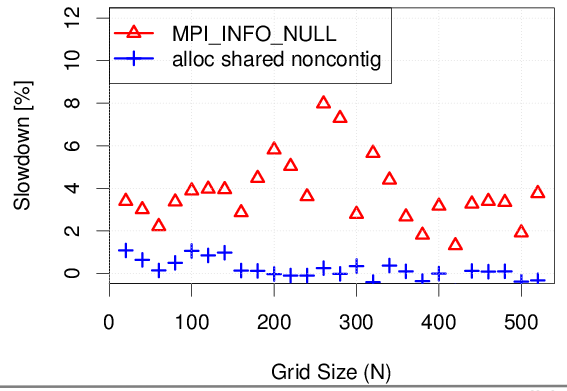
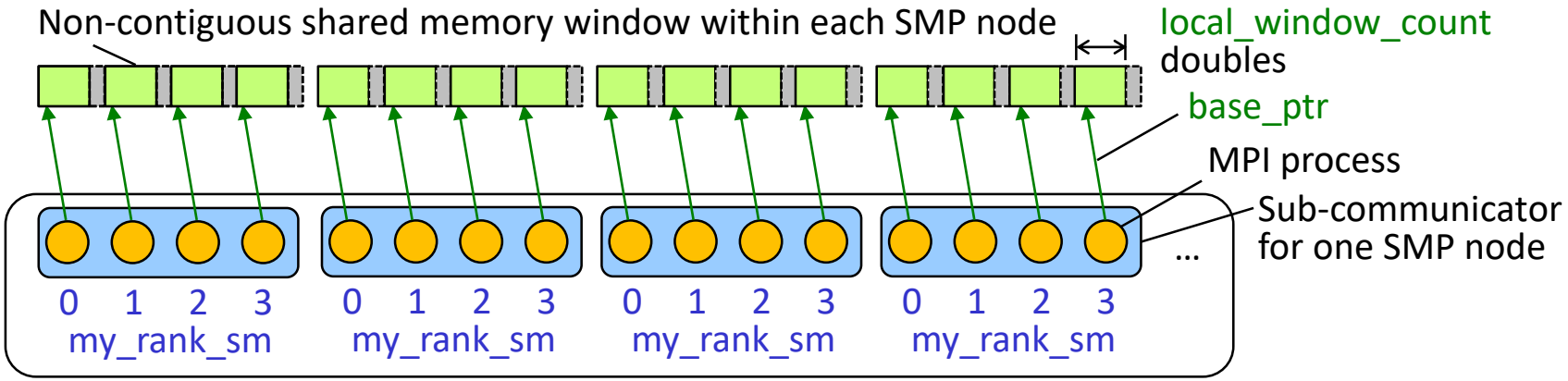


Image: Courtesy of Torsten Hoefler

skipped

Non-contiguous shared memory allocation

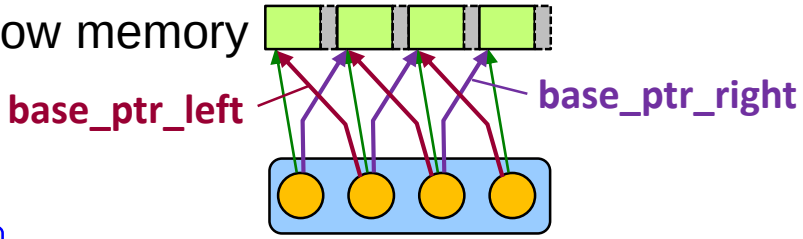



```
MPI_Aint /*IN*/ local_window_count;        double /*OUT*/  *base_ptr;
disp_unit = sizeof(double);  /* shared memory should contain doubles */
MPI_Info  info_noncontig;
MPI_Info_create (&info_noncontig);
MPI_Info_set (info_noncontig, "alloc_shared_noncontig", "true");
MPI_Win_allocate_shared ((MPI_Aint) 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 only one process allocates the whole window
→ to get the base_ptr, all processes call MPI_WIN_SHARED_QUERY



local call

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


Thanks to Steffen Weise (TU Freiberg) for testing and correcting the example codes.

Other technical aspects with MPI_Win_allocate_shared

Caution: On some systems
- the number of shared memory windows, and
- the total size of shared memory windows

may be limited.

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

to enlarge restricting defaults.

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 or /run/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.

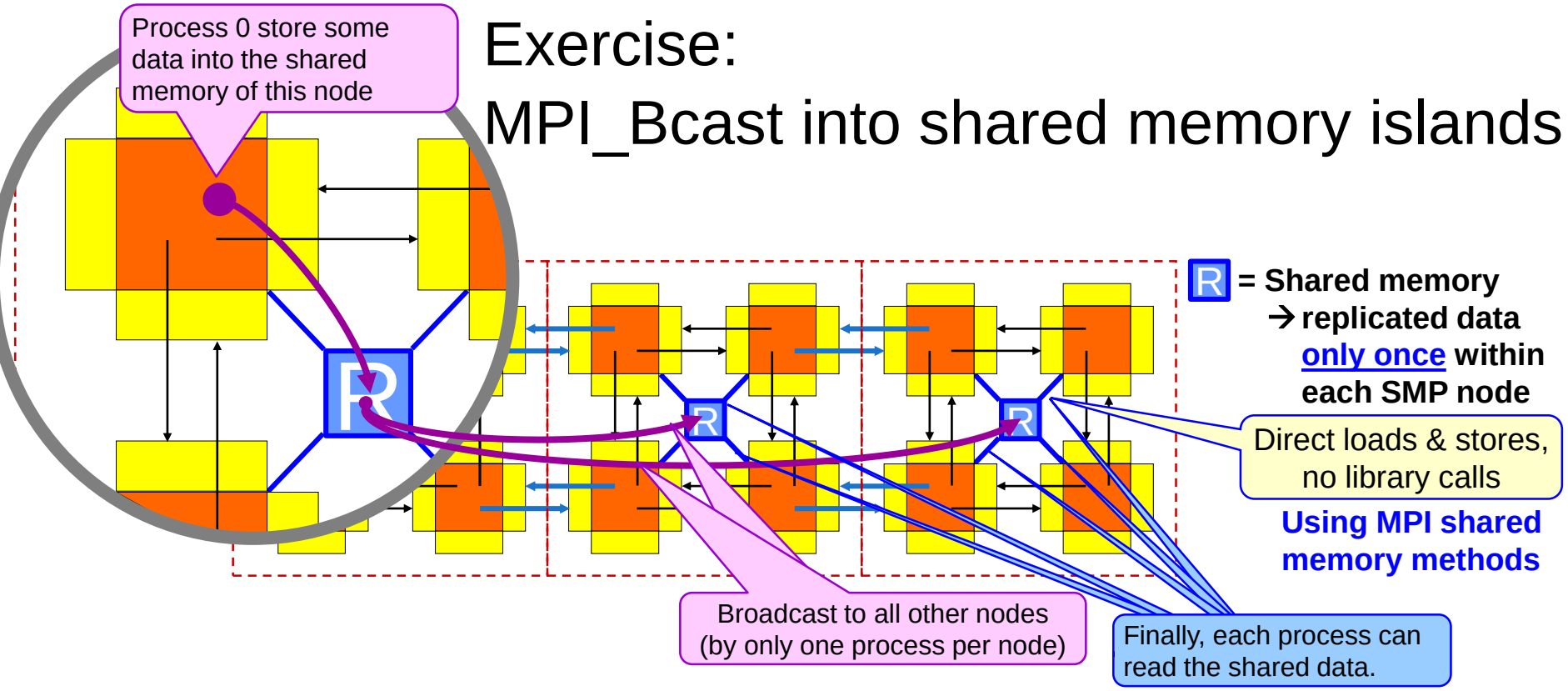
On a system without virtual memory (like CNK on BG/Q), you have to reserve a chunk of address space when the node is booted (default is 64 MB).

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

Programming models
- MPI + MPI-3.0 shared memory

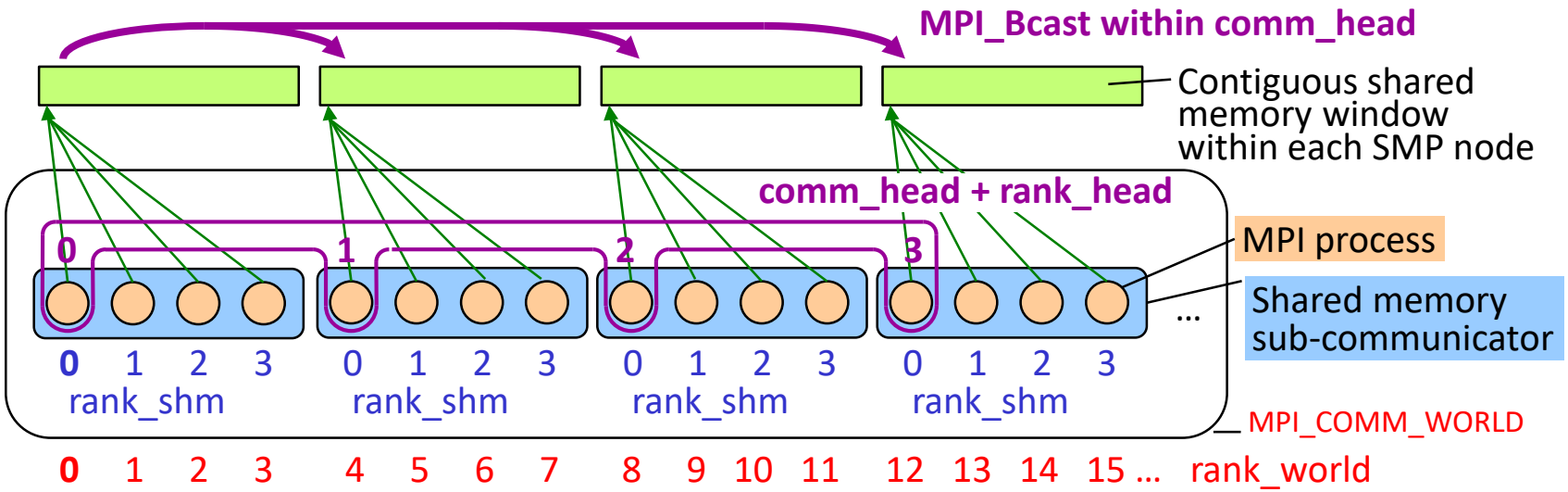
Exercise:

MPI_Bcast into shared memory islands



Exercise: MPI_Bcast into shared memory

- Now illustrated as in the previous slides
- Each ▭ represents such a replicated memory R of an SMP node

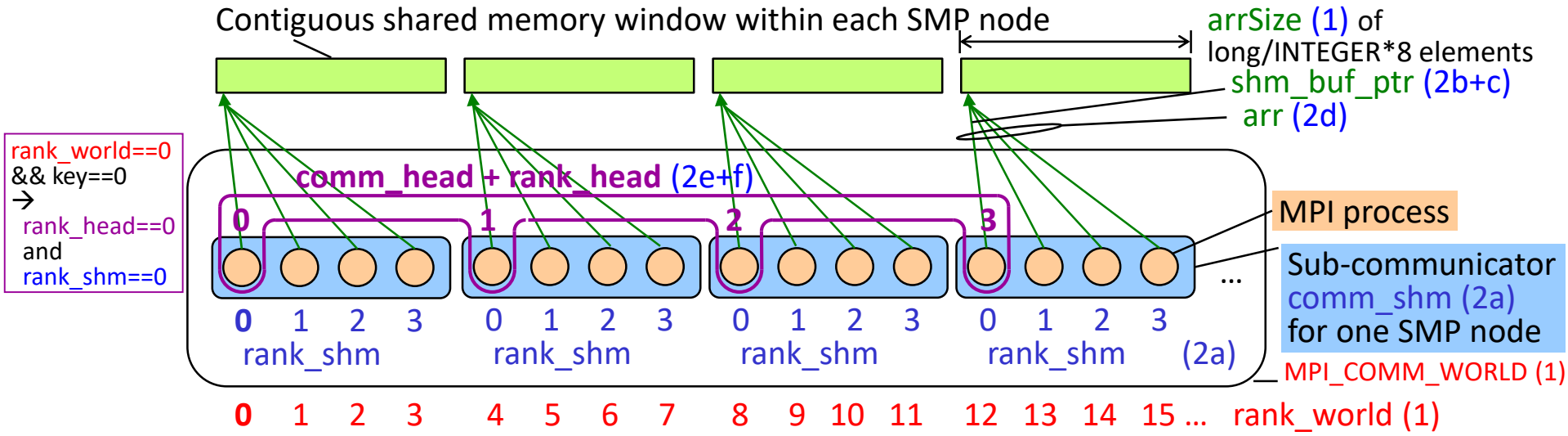


- Application: We'll store numbers 1, 2, … into the green array by process 0
- And then bcast it to all other shared memories
- At the end, each process calculates the sum of all number within *its* shared memory.

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Exercise steps:
(1-2) The allocation of the shared memory within each node



1st exercise step (~5 lines of code +2 lines printing)

(1) Given: arrSize, MPI_COMM_WORLD → rank_world

(2a) MPI_Comm_spit_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm

2nd exercise step (~11 lines of code +2 lines printing)

(2b) *if* (rank_shm == 0) *then* individualShmSize = arrSize *else* individualShmSize = 0

(2c) MPI_Win_allocate_shared (comm_shm → win & shm_base_ptr (but only if rank_shm== 0))

(2d) MPI_Win_shared_query (win & rank 0 → arr, i.e., the base pointer on all processes);

3rd exercise step (~12 lines of code +2 lines printing)

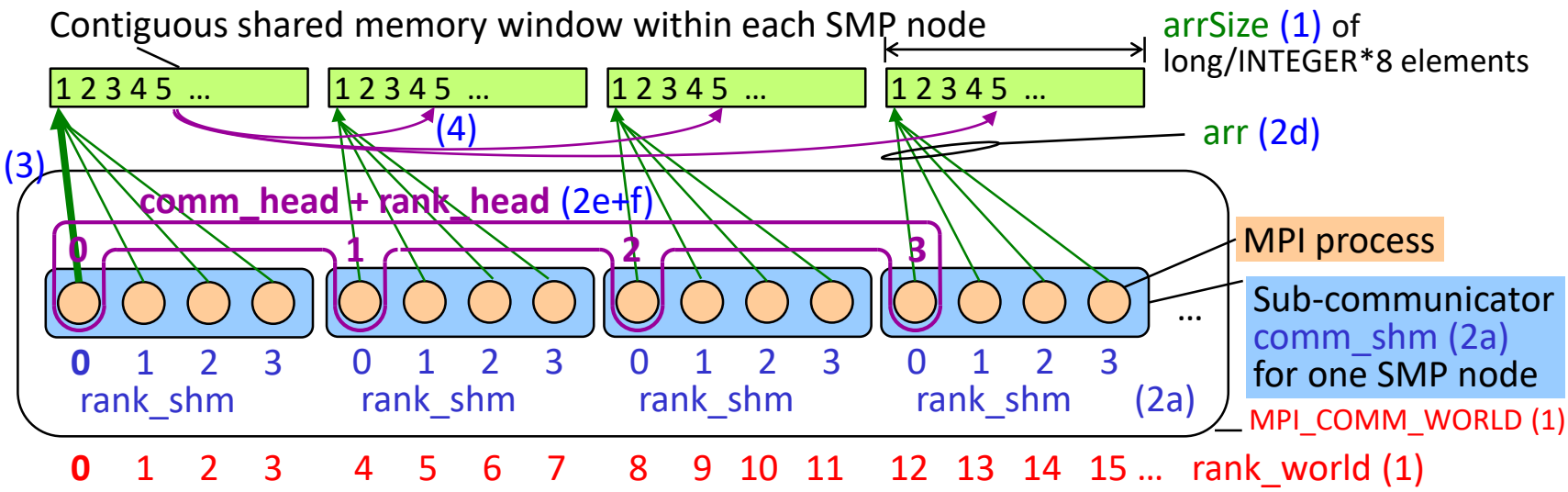
(2e) *if* (rank_shm == 0) *then* color=0 *else* color=MPI_UNDEFINED

(2f) MPI_Comm_split(MPI_COMM_WORLD, key=0, color → comm_head) → rank_head
and in all processes with color==MPI_UNDEFINED → MPI_COMM_NULL

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Exercise steps:
(3-6) The usage of the shared memory



Time step loop with index ***it*** and 3 iterations

(3-4) Store epoch: we store the replicated data in all shared memories
(don't forget **MPI_Win_fence()** within all comm_shm/win before starting the store epoch for **arr**)

4th exercise step (~5 lines of code +1 lines printing)

(3) Process with rank_world==0 **stores numbers** into ist green **arr**

(4) All processes in comm_head **MPI_Bcast()** the data from rank_head==0 to all others

(5) Local load epoch: each process reads the data and locally **calculates the sum**
(don't forget **MPI_Win_fence()** within all comm_shm / win before starting the local load epoch)

(6) Print the results

End of time step loop

5th exercise step (~1 lines of code)

(7) Finish the local load epoch → **MPI_Win_fence()** // free the window → **MPI_Win_free()**

1-slide Sol.

See login-slides

Exercise: MPI_Bcast into shared memory (Preparation 1)

- Directories in your personal account:

C

  - HY-VSC/LRZ/data-rep/C-data-rep:
    - **data-rep_base.c**
    - **data-rep_exercise.c**
    - data-rep_base_VSC/LRZ_2x16.sh / _4x48/4x28.sh (using 2 and 4 nodes)
    - data-rep_exercise_VSC/LRZ_2x16.sh (using only 2 nodes during the exercise)
    - data-rep_solution_VSC/LRZ_2x16.sh / _4x48/4x28.sh (again with 2 and 4 nodes)
    - **data-rep_exercise_orig.c** (only for: **diff data-rep_exercise_orig.c data-rep_exercise.c**)
    - **(already together with all solution files)**

Fortran

  - HY-VSC/LRZ/data-rep/F-data-rep:
    - **data-rep_base_30.f90**
    - **data-rep_exercise_30.f90**
    - data-rep_....._....._.....sh (ditto., see above)
    - **data-rep_exercise_orig_30.f90** (only for: **diff data-rep_exercise_orig_30.f90 data-rep_exercise_30.f90**)
    - **(already together with all solution files)**

- data-rep_base.c / _30.f90 is the original MPI program
- data-rep_exercise.c / _30.f90 is the basis for this shared memory exercise

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Exercise: MPI_Bcast into shared memory (Preparation, 10 Minutes)

- **data-rep_base.c / _30.f90** is the original MPI program: 

Do NOT edit
  - It copies data from the process rank 0 in MPI_COMM_WORLD to all processes.
  - On all processes it uses the data: in this example, just the sum is calculated.
  - Compile it and run it:
    - **module load intel intel-mpi**
    - **mpiicc -o data-rep_base data-rep_base.c**
    - **mpiifort -o data-rep_base data-rep_base_30.f90**
    - **sbatch data-rep_base_VSC_2x16.sh** (will use 2 nodes with only 16 processes [on 2 CPUs x 8 cores] per node and 4 nodes with all 2x24 = 48 cores per node)
    - **sq** (show queue)
    - **sinfo | grep idle** (if you do not have a reservation)

    http://tiny.cc/MPIX-LRZ

    http://tiny.cc/MPIX-VSC alternative for the exercises
  - Output will be written to: **slurm-*.out**
  - Output from only 2 nodes (each with 16 MPI processes) and 3 time steps:

```
it: 0, rank ( world: 31/32 ):   sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 1/32 ):    sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 0/32 ):    sum(i=0...i=99999999) = 4999999950000000
it: 1, rank ( world: 1/32 ):    sum(i=1...i=100000000) = 5000000050000000
it: 1, rank ( world: 0/32 ):    sum(i=1...i=100000000) = 5000000050000000
it: 1, rank ( world: 31/32 ):   sum(i=1...i=100000000) = 5000000050000000
it: 2, rank ( world: 1/32 ):    sum(i=2...i=100000001) = 5000000150000000
it: 2, rank ( world: 0/32 ):    sum(i=2...i=100000001) = 5000000150000000
it: 2, rank ( world: 31/32 ):   sum(i=2...i=100000001) = 5000000150000000
```

- 1st time step
- output from 3 processes per communicator:
- ranks 0, 1 & last rank

Parts of the software, courtesy of Irene Reichl (VSC, TU Wien)

Exercise: MPI_Bcast into shared memory (Step 2a)

- **data-rep_exercise.c / _30.f90** is the **skeleton** for all steps of this exercise
- Step 2a:
  - Declare variables comm_shm, size_shm, rank_shm (2 lines of code)
  - Split MPI_COMM_WORLD into shared memory island communicators comm_shm (use key == 0) (1 line of code)
  - Query size_shm, rank_shm (2 lines of code)
  - After this splitting: print and stop (3 lines of code, copy print statement from end of your source file)

```
/*TEST*/ // To minimize the output, we print only from 3 process per SMP node
/*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
             printf("\t\t rank ( world: %i, shm: %i)\n", rank_world, rank_shm);
/*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;
```

  - Expected output from 2 SMP nodes, each with 16 processes:

```
rank ( world: 0/32, shm: 0/16)
ALL finalize and return !!!.
rank ( world: 16/32, shm: 0/16)
rank ( world: 1/32, shm: 1/16)
rank ( world: 17/32, shm: 1/16)
rank ( world: 15/32, shm: 15/16)
rank ( world: 31/32, shm: 15/16)
```

Output from
1st SMP node
2nd SMP node

  - **After ~10 Minutes:**
    - **compare with solution: data-rep_sol_2a.c / _30.f90**
    - **In case of problems you may also look at the solution slide:** Sol.

```
diff  data-rep_exercise.c      data-rep_sol_2a.c
diff  data-rep_exercise_30.f90 data-rep_sol_2a_30.f90
```

Please edit and change it from step to step!

Exercise: MPI_Bcast into shared memory (Steps 2b-d)

- Steps 2b-d:
  - Declare needed variables (5 LOC)
  - (2b) *if* (rank_shm == 0) *then* individualShmSize = arrSize *else* individualShmSize = 0 (4 LOC)
  - (2c) MPI_Win_allocate_shared (comm_shm → win & shm_base_ptr (but only if rank_shm== 0)) (1 LOC)
  - (2d) MPI_Win_shared_query (win & rank 0 → arr, i.e., the base pointer on all processes); (1 LOC)
  - After this splitting: print and stop (3 lines of code)
  - Expected output from 2 SMP nodes, each with 16 processes:

Output from
- 1st SMP node
- 2nd SMP node

```
rank ( world: 0/32, shm: 0/16) arrSize 100000000 arrSize_ 800000000  shm_buf_ptr = 0x2b1738903000, arr_ptr =0x2b1738903000
ALL finalize and return !!!.
rank ( world: 16/32, shm: 0/16) arrSize 100000000 arrSize_ 800000000  shm_buf_ptr = 0x2b2489dfb000, arr_ptr =0x2b2489dfb000
rank ( world: 1/32, shm: 1/16) arrSize 100000000 arrSize_ 800000000  shm_buf_ptr = (nil), arr_ptr = 0x2aef69d3a000
rank ( world: 31/32, shm: 15/16) arrSize 100000000 arrSize_ 800000000  shm_buf_ptr = (nil), arr_ptr = 0x2b4dcb01e000
rank ( world: 15/32, shm: 15/16) arrSize 100000000 arrSize_ 800000000  shm_buf_ptr = (nil), arr_ptr = 0x2b56e7916000
rank ( world: 17/32, shm: 1/16) arrSize 100000000 arrSize_ 800000000  shm_buf_ptr = (nil), arr_ptr = 0x2b42516bb000
```

- **After ~20 Minutes:**
  - **compare with solution: data-rep_sol_2d.c / _30.f90**
  - **In case of problems you may also look at the solution slide:** 

Processes with **individualShmSize = 0**, do not get a buffer pointer from MPI_Win_allocate_shared

Each process within an SMP node has **different virtual addresses** for the **same** shared memory array

Exercise: MPI_Bcast into shared memory (Steps 2e-f)

- Steps 2e-f:
  - Declare needed variables (3 LOC)
  - (2e) *if* (rank_shm == 0) *then* color=0 *else* color=MPI_UNDEFINED (2 LOC)
  - (2f) MPI_Comm_split(MPI_COMM_WORLD, key=0, color → comm_head) → rank_head (8 LOC)
    and in all processes with color==MPI_UNDEFINED → MPI_COMM_NULL
  - After this splitting: print and stop (3 LOC)
  - Expected output from 2 SMP nodes, each with 16 processes:

rank (world: 1/32, shm: 1/16, head: -1/-1) arrSize 100000000 arrSize_ 800000000 shm_buf_ptr = (nil), arr_ptr = 0x2abc98db8000
rank (world: 0/32, shm: 0/16, **head: 0/2**) arrSize 100000000 arrSize_ 800000000 shm_buf_ptr = 0x2ab…, arr_ptr = 0x2ab4acc56000
ALL finalize and return !!!.
rank (world: 16/32, shm: 0/16, **head: 1/2**) arrSize 100000000 arrSize_ 800000000 shm_buf_ptr = 0x2ad…, arr_ptr = 0x2adbc5fe6000
rank (world: 15/32, shm: 15/16, head: -1/-1) arrSize 100000000 arrSize_ 800000000 shm_buf_ptr = (nil), arr_ptr = 0x2af4c52e5000
rank (world: 17/32, shm: 1/16, head: -1/-1) arrSize 100000000 arrSize_ 800000000 shm_buf_ptr = (nil), arr_ptr = 0x2b702ad9b000
rank (world: 31/32, shm: 15/16, head: -1/-1) arrSize 100000000 arrSize_ 800000000 shm_buf_ptr = (nil), arr_ptr = 0x2b6e54bdf000

- **After ~10 Minutes:**
  - **compare with solution: data-rep_sol_2f.c / _30.f90**
  - **In case of problems you may also look at the solution slide:** 

- **Whole exercise steps 2a-f: 40 Minutes**

Finished earlier?
→ Go to **advanced exercise** on **next slide**

  - Online course: please come back to the main room
  - Advanced exercise on a **copy** of your data-rep_exercise.c / _30.f90:
    Split your shared memory islands into NUMA domains

Advanced Exercise: MPI_Bcast into shared memory Breaking the world into NUMA islands (Step 2a-f-NUMA)

- Steps 2a-f: We split MPI_COMM_WORLD into ccNNUMA islands, each with 2 CPUs
- Step 2a-f-NUMA:
  - Copy your result or data-rep_sol_2f.c / _30.f90 into data-rep_exercise_**NUMA**.c / _30.f90
  - For this advanced exercise, switch from Intel-MPI to OpenMPI — Prepared for VSC only
    - **module purge**
    - **module load openmpi**
    - **mpicc -o data-rep_exercise_openmpi data-rep_exercise_NUMA.c**
    - **mpifort -o data-rep_exercise_openmpi data-rep_exercise_NUMA_30.f90**
    - **sbatch data-rep_exercise_VSC_2x16_OpenMPI.sh** (or only 1x16 → splitting into the 2 CPUs)
  - Split MPI_COMM_WORLD into NUMA islands → you expect the double amount of comm_shm
    - **Use the non-standardized method for OpenMPI**
  - Expected result: 4 shared memory islands, each consisting of the MPI processes running on a CPU

```
it: 0, rank ( world: 0/32, shm: 0/8, head: 0/4 ):   sum(i=0...i=9
it: 0, rank ( world: 1/32, shm: 1/8, head: -1/-1 ):  sum(i=0...i=9
it: 0, rank ( world: 7/32, shm: 7/8, head: -1/-1 ):  sum(i=0
it: 0, rank ( world: 8/32, shm: 0/8, head: 1/4 ):    sum(i=0...i=9
it: 0, rank ( world: 9/32, shm: 1/8, head: -1/-1 ):  sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 15/32, shm: 7/8, head: -1/-1 ): sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 24/32, shm: 0/8, head: 3/4 ):   sum
it: 0, rank ( world: 16/32, shm: 0/8, head: 2/4 ):   sum
it: 0, rank ( world: 25/32, shm: 1/8, head: -1/-1 ): sum
it: 0, rank ( world: 31/32, shm: 7/8, head: -1/-1 ): sum
it: 0, rank ( world: 17/32, shm: 1/8, head: -1/-1 ):
it: 0, rank ( world: 23/32, shm: 7/8, head: -1/-1 ):
…
```

4 different comm_shm communicators, each with 8 processes, first, second and last one generating such 3 lines

You may also play with different options in the batch script! E.g., without **--rank-by core**, the first CPU will have the world ranks **0,2**,4,6,8,10,12,**14** (bold=printed). Add **MPI_Bcast(&rank_head, 0, MPI_INT, 0, comm_shm)** to show which processes belong to same comm_shm.

  - Compare with solution: data-rep_sol_2f_NUMA_OpenMPI.c / _30.f90

Exercise: MPI_Bcast into shared memory (Steps 3-6)

- Steps 3-6 (6 lines of code)
  - **(3-4) Store epoch:** we store the replicated data in all shared memories
    (don't forget **MPI_Win_fence()** within all comm_shm/win before starting the store epoch for **arr**)
  - (3) Process with rank_world==0 **stores numbers** into its green **arr**
  - (4) All processes in comm_head **MPI_Bcast()** the data from rank_head==0 to all others
  - **(5) Local load epoch:** each process reads the data and locally **calculates the sum**
    (don't forget **MPI_Win_fence()** within all comm_shm / win before starting the local load epoch)
  - **(6) Print** the results
  - Expected output from 2 SMP nodes:

```
it: 0, rank ( world: 0/32, shm: 0/16, head: 0/2 ):      sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 16/32, shm: 0/16, head: 1/2 ):     sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 1/32, shm: 1/16, head: -1/-1 ):    sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 17/32, shm: 1/16, head: -1/-1 ):   sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 31/32, shm: 15/16, head: -1/-1 ): sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 15/32, shm: 15/16, head: -1/-1 ): sum(i=0...i=99999999) = 4999999950000000
it: 1, rank ( world: 16/32, shm: 0/16, head: 1/2 ):     sum(i=1...i=100000000) = 5000000050000000
it: 1, rank ( world: 0/32, shm: 0/16, head: 0/2 ):      sum(i=1...i=100000000) = 5000000050000000
...
it: 2, rank ( world: 17/32, shm: 1/16, head: -1/-1 ):   sum(i=2...i=100000001) = 5000000150000000
ALL finalize and return !!!.
it: 2, rank ( world: 1/32, shm: 1/16, head: -1/-1 ):    sum(i=2...i=100000001) = 5000000150000000
...
```

Same data in the shared memory arrays of both SMP nodes

Same data also in 2nd time step

Same on all processes also in 3rd time step

  - **After ~10 Minutes,**
    - **compare with solution: data-rep_sol_3-6.c / _30.f90**
    - **In case of problems you may also look at the solution slide:** 

Exercise: MPI_Bcast into shared memory (Step 7)

- Step 7 (6 lines of code)
  - (7) Finish the local load epoch → **MPI_Win_fence()** // free the window → **MPI_Win_free()**
  - Expected output from 2 SMP nodes **(same as after Step 6, but now without premature stop):**

```
it: 0, rank ( world: 0/32, shm: 0/16, head: 0/2 ):       sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 16/32, shm: 0/16, head: 1/2 ):      sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 1/32, shm: 1/16, head: -1/-1 ):     sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 17/32, shm: 1/16, head: -1/-1 ):    sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 31/32, shm: 15/16, head: -1/-1 ): sum(i=0...i=99999999) = 4999999950000000
it: 0, rank ( world: 15/32, shm: 15/16, head: -1/-1 ): sum(i=0...i=99999999) = 4999999950000000
it: 1, rank ( world: 16/32, shm: 0/16, head: 1/2 ):      sum(i=1...i=100000000) = 5000000050000000
it: 1, rank ( world: 0/32, shm: 0/16, head: 0/2 ):       sum(i=1...i=100000000) = 5000000050000000
...
it: 2, rank ( world: 17/32, shm: 1/16, head: -1/-1 ):    sum(i=2...i=100000001) = 5000000150000000
All finalize and return !!!
it: 2, rank ( world: 1/32, shm: 1/16, head: -1/-1 ):     sum(i=2...i=100000001) = 5000000150000000
...
```

  - After ~5 Minutes, in the solution directory:
    - **compare with solution: data-rep_sol_7.c / _30.f90**
    - **In case of problems you may also look at the solution slide:**  sol
  - And add-on: data-rep_solution.c / _30.f90 with additional analysis and output:

```
The number of shared memory islands is: 2 islands

The size of each shared memory islands is: 48 processes
```

  - **Whole exercise steps 3-6 & 7: approx. 20 Minutes**
  - Q & A & Discussion

Quiz on Shared Memory

A. Before you call **MPI_Win_allocate_shared**, what should you do?

_____

B. If your communicator within your shared memory island consists of 12 MPI processes, and each process wants to get an own window with 10 doubles (each 8 bytes),
   a. which **window size** must you specify in **MPI_Win_allocate_shared**?

      _____

   b. And how long is the totally allocated shared memory?

      _____

   c. The returned base_ptr, will it be identical on all 12 processes?

      _____

   d. If all 12 processes want to have a pointer that points to the beginning of the totally allocated shared memory, which MPI procedure should you use and with which major argument?

      _____

   e. If you do this, do these 12 pointers have identical values, i.e., are identical addresses?

      _____

C. Which is the major method to store data from one process into the shared memory window portion of another process?

_____

Programming models
- MPI + MPI-3.0 shared memory

MPI Memory Models & Synchronization

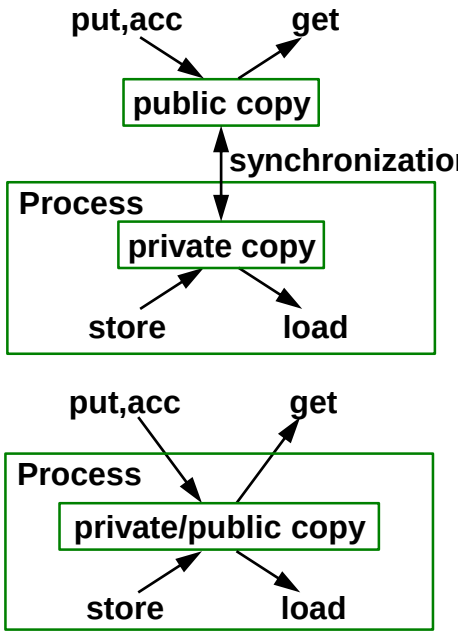
Lowest latencies

A key feature for strong scaling?

- Usage of MPI shared memory without one-sided synchronization methods
- MPI provides the shared memory, but used
  - only with compiler generated loads & stores
  - together with C++11 memory fences

Two memory models

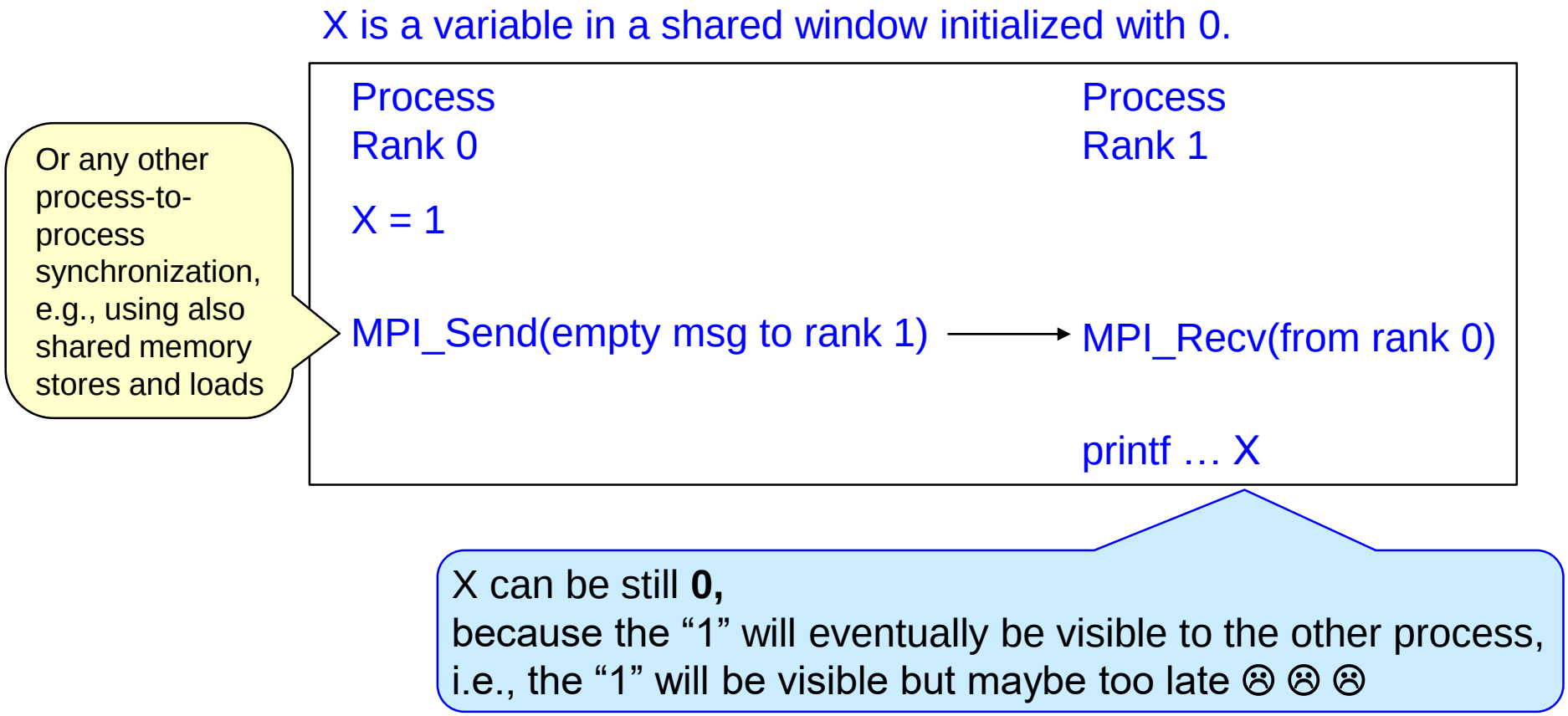
- Query for new attribute to allow applications to tune for cache-coherent architectures
  - Attribute MPI_WIN_MODEL with values
    - **MPI_WIN_SEPARATE model**
    - **MPI_WIN_UNIFIED model on cache-coherent systems**
- Shared memory windows always use the MPI_WIN_UNIFIED model
  - Public and private copies are **eventually** synchronized without additional RMA synchronization calls
    (MPI-3.1/MPI-4.0, Section 11/12.4, page 435/592 lines 43-46/42-45)
  - For synchronization **without delay**: MPI_WIN_SYNC()
    (MPI-3.1/-4.0 Section 11/12.7: "*Advice to users*. In the unified memory model…" in U5 on page 456/613f, and Section 11/12.8, Example 11/12.21 on pages 468f/626f)
  - or any other RMA synchronization:
    "A consistent view can be created in the unified memory model (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH)."
    (MPI-3.1/-4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26)



Figures: Courtesy of Torsten Hoefler

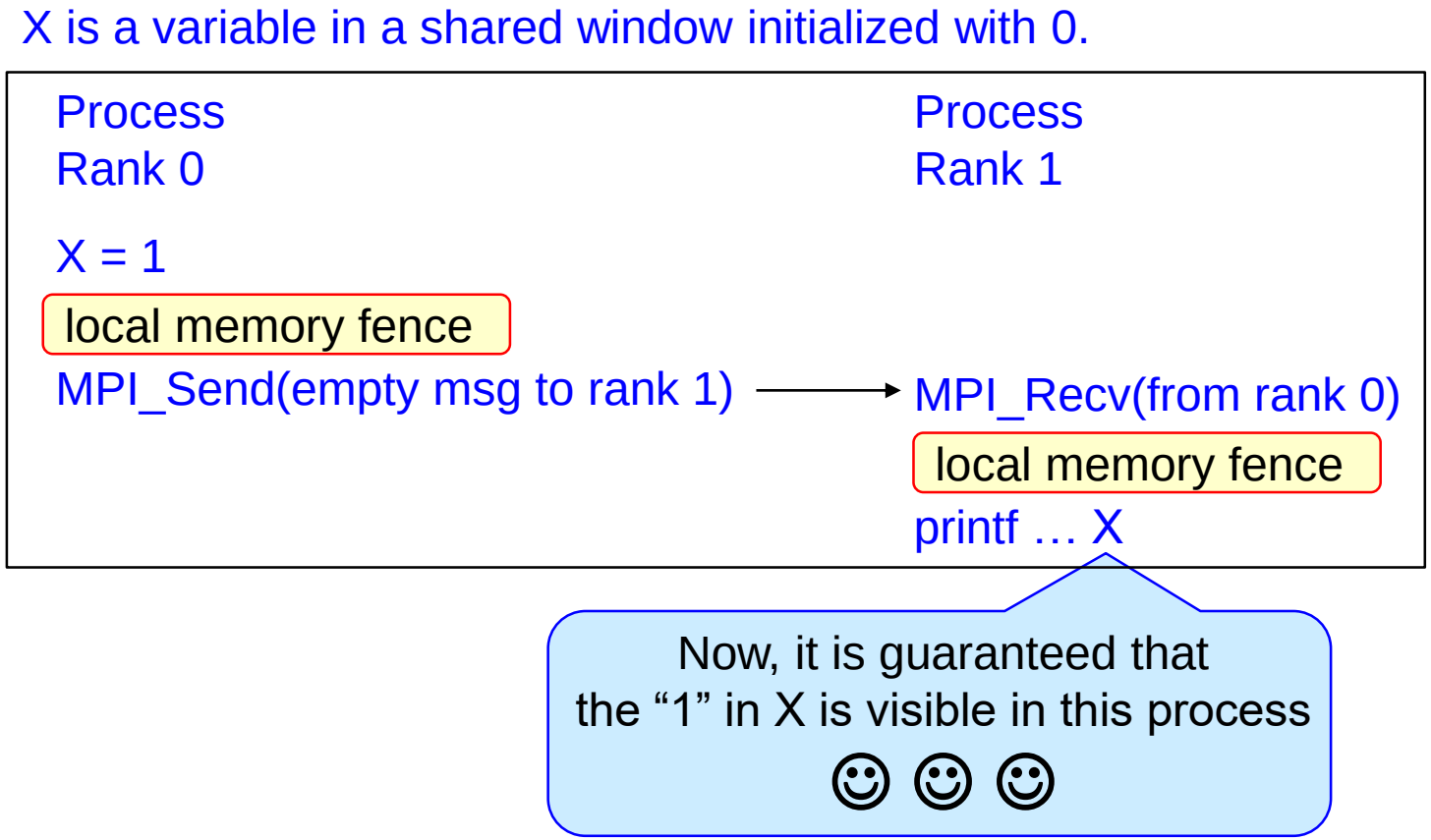
“eventually synchronized“ – the Problem

- The problem with shared memory programming using libraries is:



“eventually synchronized“ – the Solution

- A pair of local memory fences is needed:



“eventually synchronized“ – Last Question

Several options & heavy discussions in the MPI Forum

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

X = 1

local memory fence

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

local memory fence

printf … X

How to make the local memory fence ?

- C11 atomic_thread_fence(order)
  - **Advantage: one can choose appropriate order = memory_order_acquire, or …_release to achieve minimal latencies**
- MPI_Win_sync
  - **Advantage: works also for Fortran**
  - **Disadvantage: may be slower than C11 atomic_thread_fence with appro. order**
- Using RMA synchronization with integrated local memory fence instead of MPI_Send → MPI_Recv
  - **Advantage: May prevent double fences**
  - **Disadvantage: The synchronization itself may be slower**

5 sync methods, see next slide

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

X = 1

MPI_Win_fence Include needed memory fence → MPI_Win_fence Includes needed memory fence

printf … X

General MPI-3 shared memory synchronization rules

(based on MPI-3.0/3.1, MPI_Win_allocate_shared, page 410/408, lines 16-20/43-47: ***"A consistent view …"***)

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

being
- MPI_Win_post[1] → MPI_Win_start[1]

or
- MPI_Win_complete[1] → MPI_Win_wait[1]

or
- MPI_Win_fence[1] ⟷ MPI_Win_fence[1]

or
- MPI_Win_sync
- Any-process-sync[2] → Any-process-sync[2]
- MPI_Win_sync

or[3]
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 is granted first

and A, B, C are shared variables

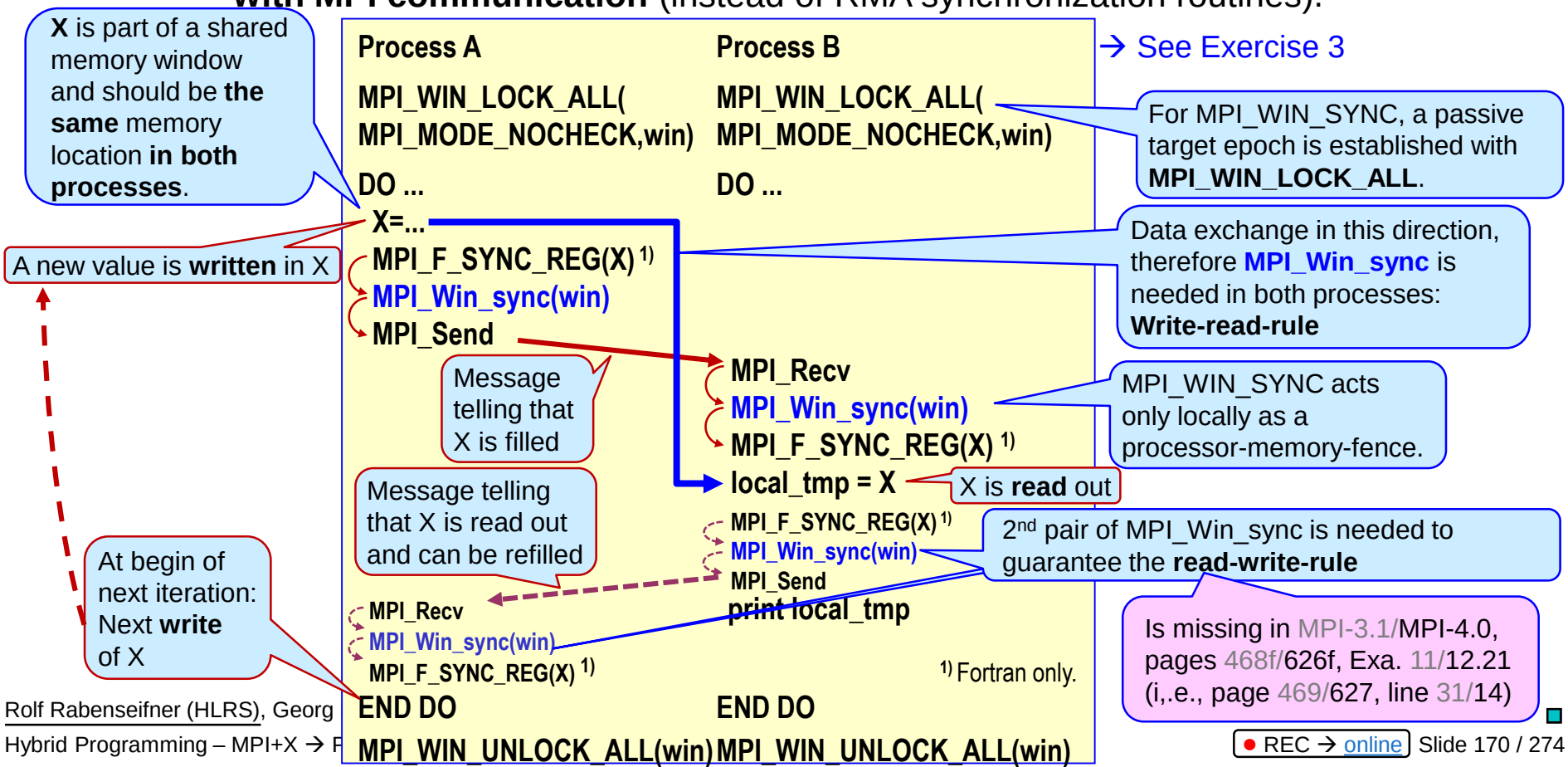
and having …	**then it is guaranteed that …**
A=val_1 Sync-from → Sync-to load(A)	⇒ **… the load(A) in P1 loads val_1** (this is the write-read-rule)
load(B) Sync-from → Sync-to B=val_2	⇒ **… the load(B) in P0 is not affected by the store of val_2 in P1** (read-write-rule)
C=val_3 Sync-from → Sync-to C=val_4 load(C)	⇒ **… that the load(C) in P1 loads val_4** (write-write-rule)

See next slide

1) Must be paired according to the general one-sided synchronization rules.
2) "Any-process-sync" may be done with methods from MPI (e.g. with send→recv as in MPI-3.1/MPI-4.0 Example 11/12.21, but also with some synchronization through MPI shared memory loads and stores, e.g. with C++11 atomic loads and stores).
3) No rule for MPI_Win_flush (according current forum discussion)

“Any-process-sync” & MPI_Win_sync on 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_Get or MPI_Put)
- and the **synchronization** between the processes is done **with MPI communication** (instead of RMA synchronization routines).



See HLRS online courses **http://www.hlrs.de/training/par-prog-ws/**
→ Practical → MPI.tar.gz → subdirectory MPI/course/C/1sided/

Halo communication benchmarking

- Goal:
  - Learn about the communication latency and bandwidth on your system
- Method:
  - **cp MPI/course/C/1sided/halo* .**
  - On a shared or distributed memory, run and compare:
    - halo_irecv_send.c
    - halo_isend_recv.c
    - halo_neighbor_alltoall.c
    - halo_1sided_put.c
    - halo_1sided_put_alloc_mem.c
    - halo_1sided_put_win_alloc.c
  - And run and compare on a shared memory only:
    - halo_1sided_store_win_alloc_shared.c
    - halo_1sided_store_win_alloc_shared_query.c (with alloc_shared_noncontig)
    - halo_1sided_store_win_alloc_shared_pscw.c
    - halo_1sided_store_win_alloc_shared_othersync.c
    - halo_1sided_store_win_alloc_shared_signal.c

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

Example 1

Example 2

Different communication methods

Different memory allocation methods

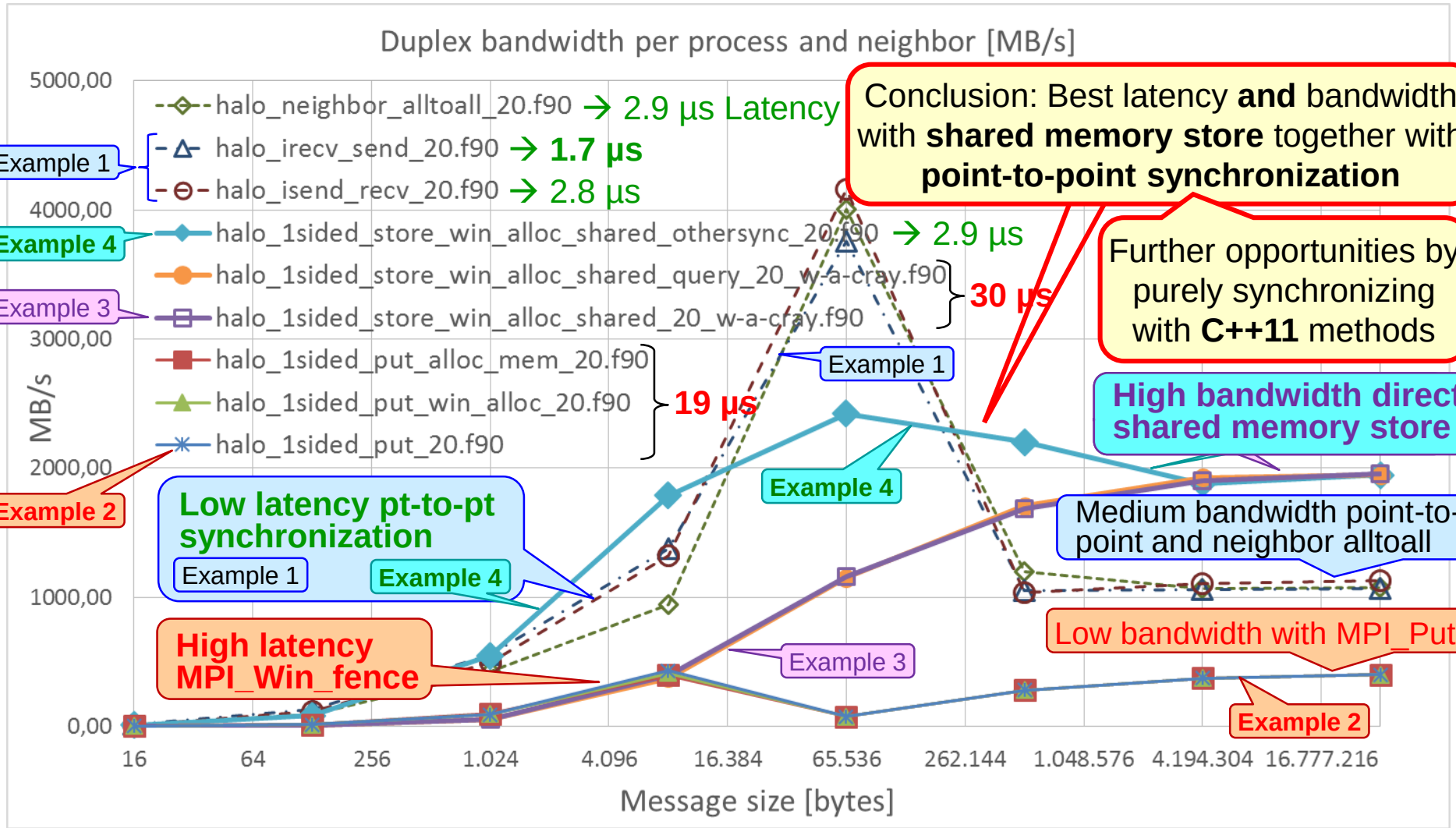
Example 3

Example 4

Example 5

Different communication methods

MPI Communication inside of the SMP nodes: Benchmark results on a Cray XE6 – 1-dim ring communication on 1 node with 32 cores



On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

Programming models
- MPI + MPI-3.0 shared memory

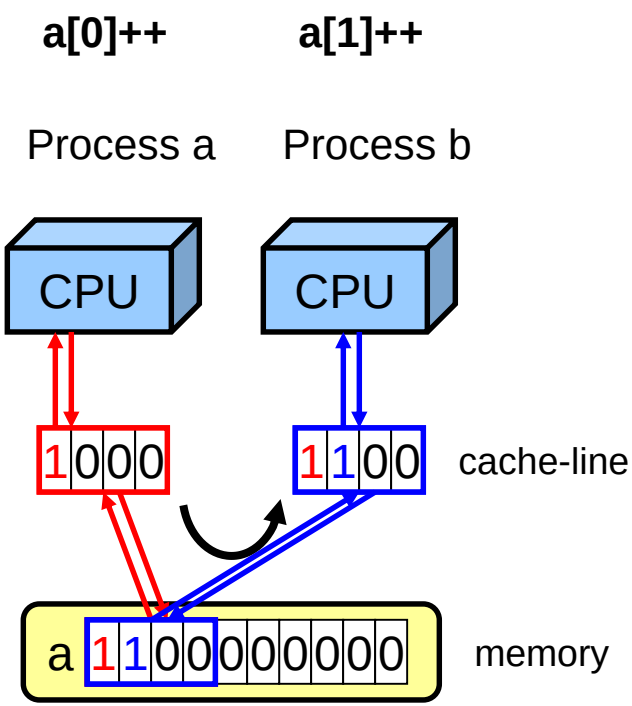
Shared memory problems

Shared memory problems (1/2)

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

Shared memory problems (2/2)

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



- Several processes are accessing shared data through the same cache-line.
- This cache-line has to be moved between these processes (cache coherence protocol).
- This is very time-consuming.

MPI+MPI-3.0 shared mem: Main advantages

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

MPI+MPI-3.0 shared mem: Main disadvantages

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

MPI+MPI-3.0 shared mem: Conclusions

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

Skip examples → Quiz

5 Examples / Exercises

Files can be downloaded from
http://www.hlrs.de/training/par-prog-ws/ → Practical → MPI.tar.gz

MPI processes using 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)
- Examples:
  1. Halo communication in a ring with pt-to-pt **(MPI_Isend / MPI_Recv)**
  2. Same with one-sided communication **(MPI_Win_fence / MPI_Put)**
  3. Same with shared memory accesses **(MPI_Put → normal assignment)**
  4. Substituting MPI_Win_fence by fast pt-to-pt synchronization
  5. Substituting the pt-to-pt synchronization by memory signals

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Example 1 — Rotating information around a ring

Numbers used on next slide

1

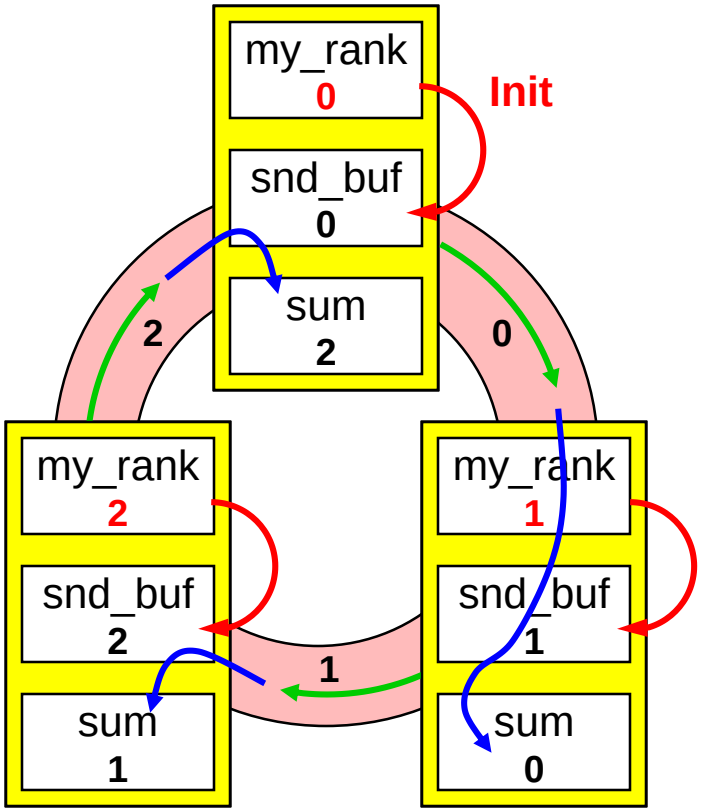
- A set of processes are arranged in a ring.
- Each process stores its rank in MPI_COMM_WORLD into an integer variable *snd_buf*.

2 3

- Each process passes this on to its neighbor on the right.

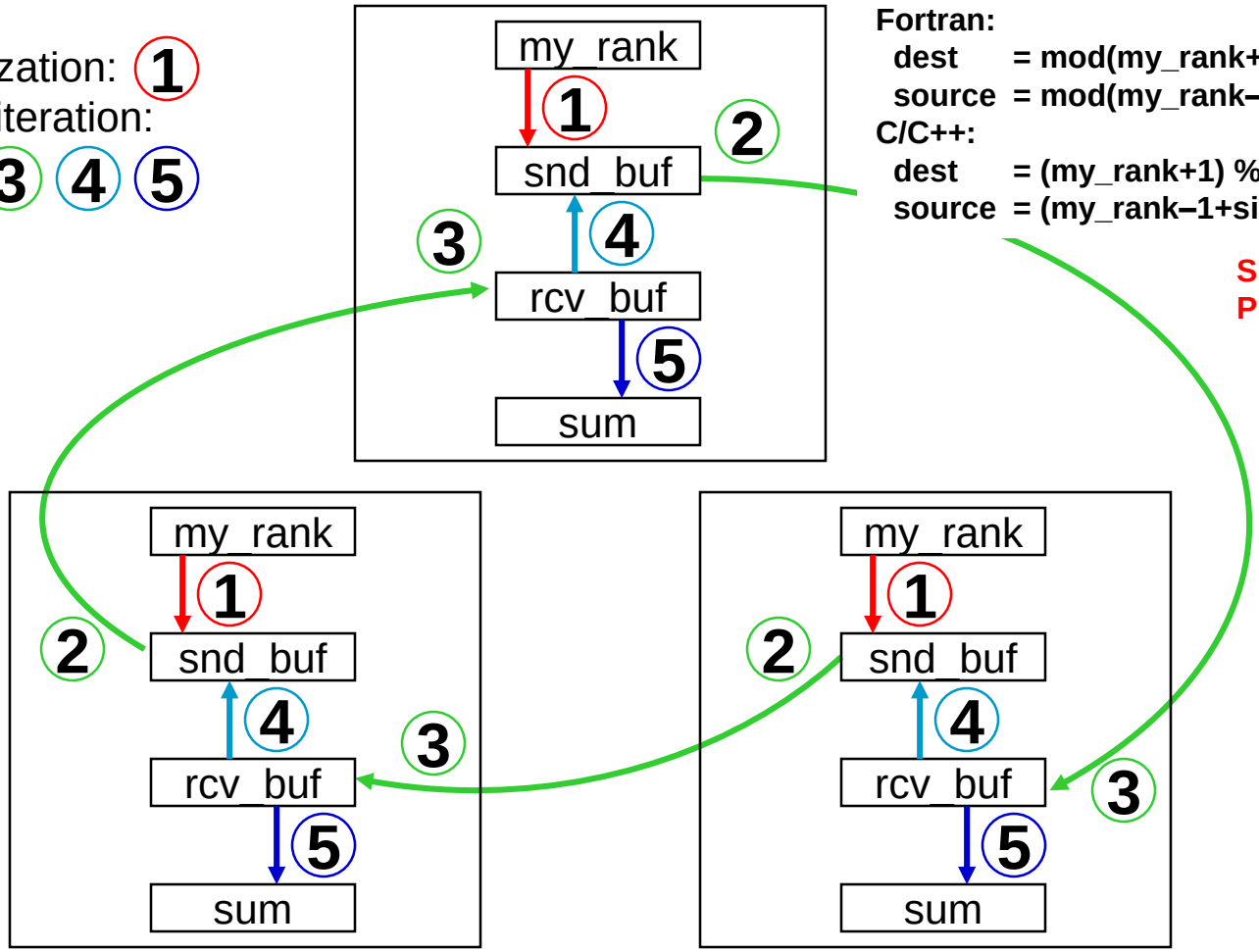
4 5

- Each processor calculates the sum of all values.
- Repeat "2-5" with "size" iterations (size = number of processes), i.e.
- each process calculates sum of all ranks.
- Use nonblocking MPI_Issend
  - to avoid deadlocks
  - to verify the correctness, because blocking synchronous send will cause a deadlock



Example 1 — Rotating information around a ring

Initialization: 1
Each iteration:
2 3 4 5



Fortran:
dest = mod(my_rank+1,size)
source = mod(my_rank–1+size,size)
C/C++:
dest = (my_rank+1) % size;
source = (my_rank–1+size) % size;

Single Program !!!

see also login-slides

Fortran: Do not forget MPI-3.0 → **<TYPE>, ASYNCHRONOUS :: …_buf** and
IF(.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING...) CALL MPI_F_SYNC_REG(…_buf)

Example 1: Nonblocking halo-copy in a ring

C

MPI/course/C/Ch4/ring.c

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

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

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

See HLRS online courses
http://www.hlrs.de/training/par-prog-ws/
→ Practical → MPI.tar.gz

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

1

2

3

4

5

Example 1: Nonblocking halo-copy in a ring

Fortran

MPI/course/F_30/Ch4/ring_30.f90

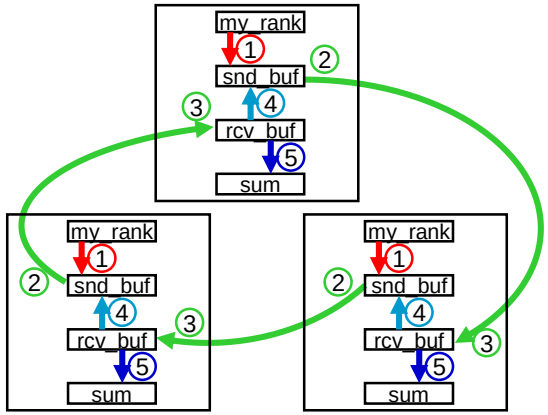
```
INTEGER, ASYNCHRONOUS :: snd_buf                                   2
INTEGER :: rcv_buf, sum, i, my_rank, size
TYPE(MPI_Status)  :: status
TYPE(MPI_Request) :: request
INTEGER(KIND=MPI_ADDRESS_KIND):: iadummy

CALL MPI_Init()
CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank)
CALL MPI_Comm_size(MPI_COMM_WORLD, size)
right = mod(my_rank+1,      size)
left  = mod(my_rank-1+size, size)
sum = 0
snd_buf = my_rank                                                  1
DO i = 1, size
   CALL MPI_Issend(snd_buf,1,MPI_INTEGER,right,17,MPI_COMM_WORLD, request)   2
   CALL MPI_Recv ( rcv_buf,1,MPI_INTEGER,left, 17,MPI_COMM_WORLD, status)    3
   CALL MPI_Wait(request, status)                                  2
!    CALL MPI_GET_ADDRESS(snd_buf, iadummy)
!    ... should be substituted as soon as possible by:
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(snd_buf)
   snd_buf = rcv_buf                                               4
   sum = sum + rcv_buf                                             5
END DO
WRITE(*,*) 'PE', my_rank, ': Sum =', sum
CALL MPI_Finalize()
```

See HLRS online courses
http://www.hlrs.de/training/par-prog-ws/
→ Practical → MPI.tar.gz

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

Example 2: Ring with fence and one-sided comm.

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

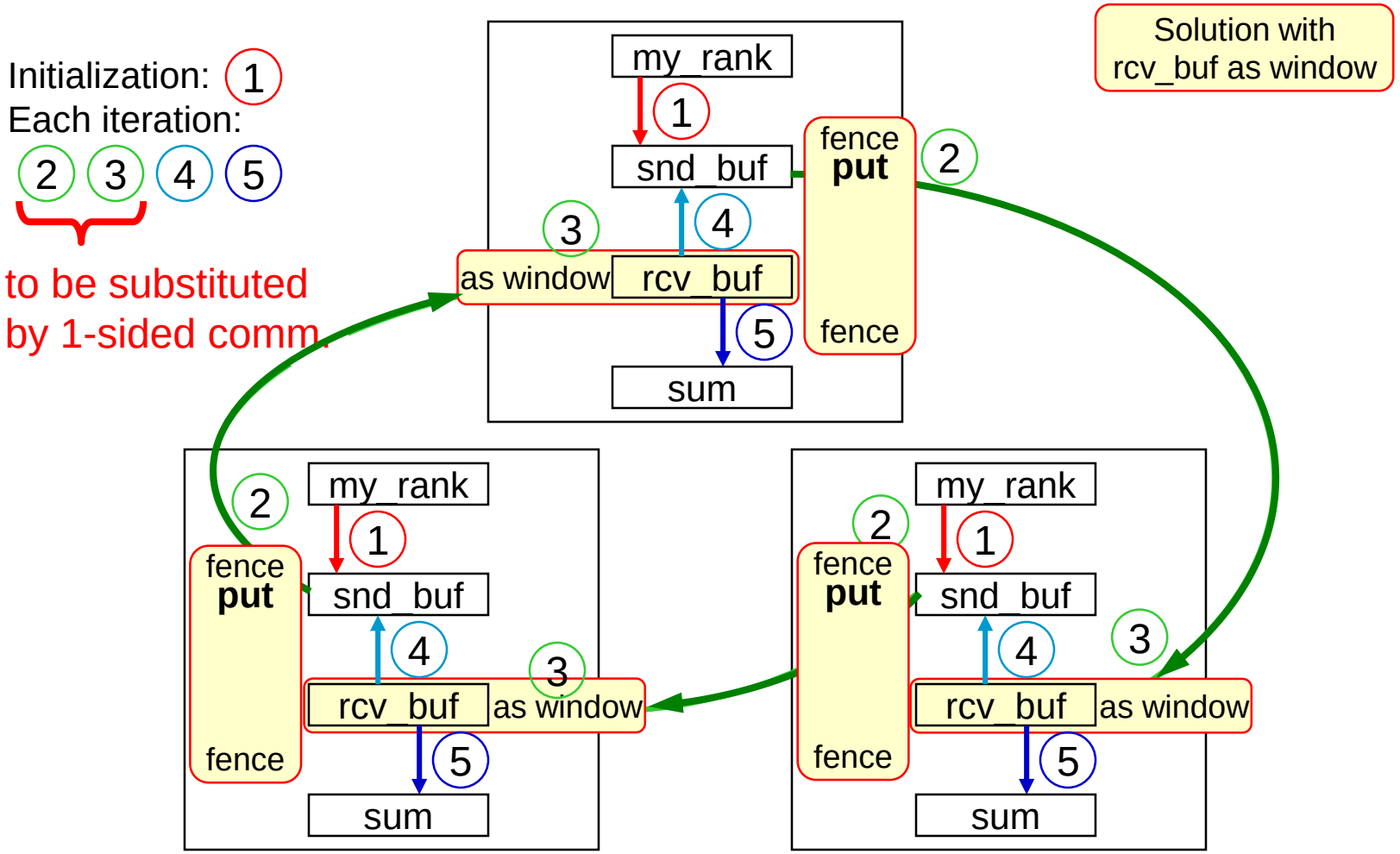


- **either rcv_buf = window**
  - MPI_Win_fence - the rcv_buf can be used to receive data
  - MPI_Put - to write the content of the local variable snd_buf into the remote window (rcv_buf)
  - MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled

Next slide: The code for this solution

- **or snd_buf = window**
  - MPI_Win_fence - the snd_buf is filled
  - MPI_Get - to read the content of the remote window (snd_buf) into the local variable rcv_buf
  - MPI_Win_fence - the one-sided communication is finished, rcv_buf is filled

Example 2: Ring with fence and one-sided comm.



Example 2: Ring with fence and one-sided comm.

C

MPI/course/C/1sided/ring_1sided_put.c

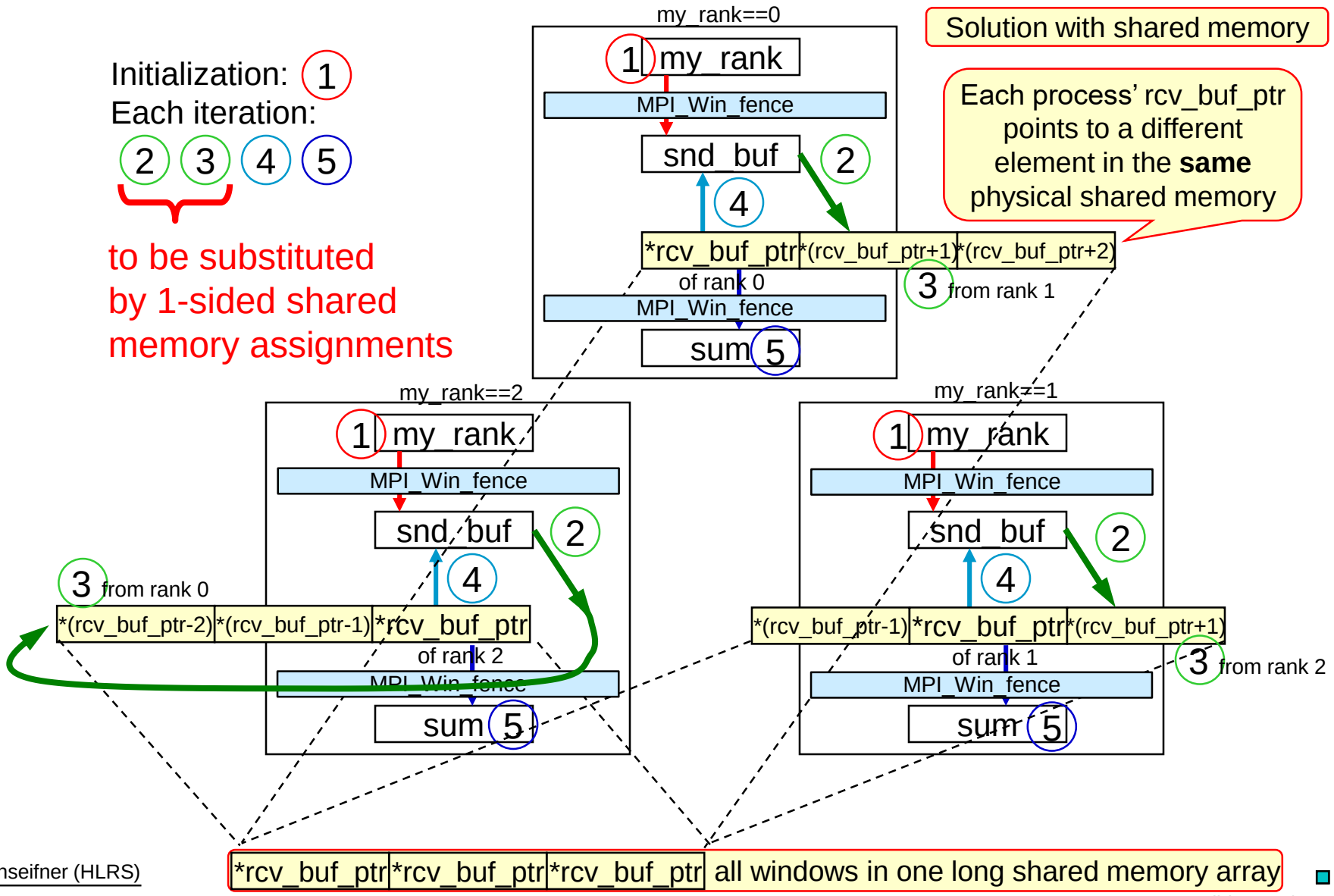
```
MPI_Win  win;
-------------------------------------------------------------------------------
/* Create the window once before the loop:  */
MPI_Win_create(&rcv_buf, (MPI_Aint) sizeof(int), sizeof(int), MPI_INFO_NULL,
               MPI_COMM_WORLD, &win);
-------------------------------------------------------------------------------
  /* Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait:  */
  MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPRECEDE, win);
  MPI_Put(&snd_buf, 1, MPI_INT, right, (MPI_Aint) 0, 1, MPI_INT, win);
  MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT | MPI_MODE_NOSUCCEED,win);
```

Fortran

MPI/course/F_30/1sided/ring_1sided_put_30.f90

```
INTEGER, ASYNCHRONOUS :: rcv_buf
TYPE(MPI_Win) :: win ; INTEGER :: disp_unit
INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, buf_size, target_disp
-------------------------------------------------------------------------------
target_disp = 0 ! This "long" integer zero is needed in the call to MPI_PUT
! Create the window once before the loop:
CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, integer_size)
buf_size = 1 * integer_size; disp_unit = integer_size
CALL MPI_WIN_CREATE(rcv_buf, buf_size, disp_unit, MPI_INFO_NULL, &
 &                  MPI_COMM_WORLD, win)
-------------------------------------------------------------------------------
! Inside of the loop; instead of MPI_Issend / MPI_Recv / MPI_Wait:
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
  CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE,MPI_MODE_NOPRECEDE), win)
  CALL MPI_PUT(snd_buf,1,MPI_INTEGER,right,target_disp,1,MPI_INTEGER, win)
  CALL MPI_WIN_FENCE(IOR(MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOSUCCEED),win)
  IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
```

Exercise/
Example 3: Shared memory ring communication



Exercise/
Example 3: Shared memory ring communication

- Use the given program as your baseline for the following exercise:

  cp ~/MPI/course/C/1sided/ring_1sided_put_win_alloc.c my_shared_exa2.c *or*
  cp ~/MPI/course/F_30/1sided/ring_1sided_put_win_alloc_30.f90 my_shared_exa2.f90

  (or F_20 …….. _20 …. with mpi module)

- Tasks: Substitute the distributed window by a shared window
  - Add **MPI_Comm_split_type** directly after MPI_Init. From there, use **comm_sm**
  - Substitute **MPI_Alloc_mem+MPI_Win_create** by **MPI_Win_allocate_shared**
    - Do not forget to also remove the **MPI_Free_mem**
  - Substitute the **MPI_Put** by a direct assignment:
    - ***rcv_buf_ptr** is the local rcv_buf
    - The rcv_buf of the right neighbor can be accessed through the word-offset **"+1"** in the direct assignment *(rcv_buf_ptr+(offset)) = snd_buf
    - In the ring, a word-offset with the value **+1** should be expressed with **(right – my_rank)**, which is normally **+1**, except for the last process, where it is **–size+1**
    - Fortran: a. Declare recv_buf as array: INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf(:)
      … CALL C_F_POINTER(ptr_rcv_buf, rcv_buf, (/1/)) … snd_buf = rcv_buf(1) …
      b. Be sure that that you add additional calls to MPI_F_SYNC_REG between both MPI_Win_fence and your direct assignment, i.e., directly before and after rcv_buf(1+(offset)) = snd_buf

- Compile and run this program on 4 cores & **all cores** of a shared memory node.

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

Solution / Example 3: Ring with shared memory one-sided comm.

C

MPI/course/C/1sided/ring_1sided_store_win_alloc_shared.c

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

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

Solution / Example 3: Ring with shared memory one-sided comm.

Fortran

MPI/course/F_30/1sided/ring_1sided_store_win_alloc_shared_30.f90

```
   USE mpi_f08
   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
   IMPLICIT NONE
-----------------------------------------------------------------------
   INTEGER :: snd_buf
   INTEGER, POINTER, ASYNCHRONOUS :: rcv_buf(:)
   TYPE(C_PTR) :: ptr_rcv_buf

   TYPE(MPI_Win) :: win
   INTEGER :: disp_unit
   INTEGER(KIND=MPI_ADDRESS_KIND) :: integer_size, lb, iadummy
   INTEGER(KIND=MPI_ADDRESS_KIND) :: rcv_buf_size, target_disp
-----------------------------------------------------------------------
   CALL MPI_Type_get_extent(MPI_INTEGER, lb, integer_size)
   rcv_buf_size = 1 * integer_size
   disp_unit = integer_size
   CALL MPI_Win_allocate_shared(rcv_buf_size, disp_unit,
                        MPI_INFO_NULL, MPI_COMM_WORLD, ptr_rcv_buf, win)
   CALL C_F_POINTER(ptr_rcv_buf, rcv_buf, (/1/) )
 ! target_disp = 0
-----------------------------------------------------------------------
```

See HLRS online courses **http://www.hlrs.de/training/par-prog-ws/** → Practical → MPI.tar.gz

Substitution of `MPI_Put` → see next slide

Solution / Example 3: Ring with shared memory one-sided comm.

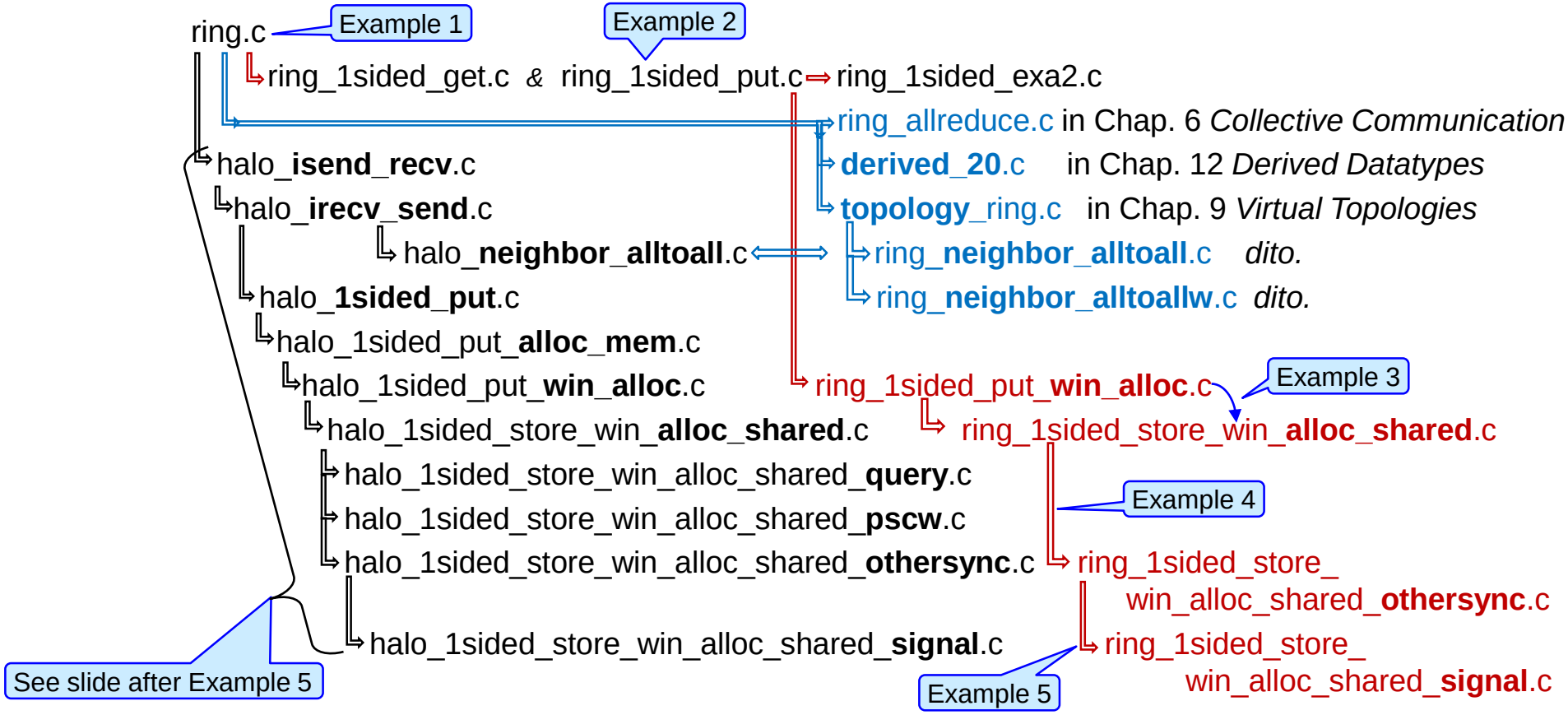
Fortran

```
   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
   CALL MPI_Win_fence( 0, win)   ! Workaround: no assertions

   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)

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

   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)

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

   IF (.NOT.MPI_ASYNC_PROTECTS_NONBLOCKING) CALL MPI_F_sync_reg(rcv_buf)
```

MPI_F_SYNC_REG(rcv_buf_right/left) guarantees that the assignments rcv_buf = … must not be moved across both MPI_Win_fence

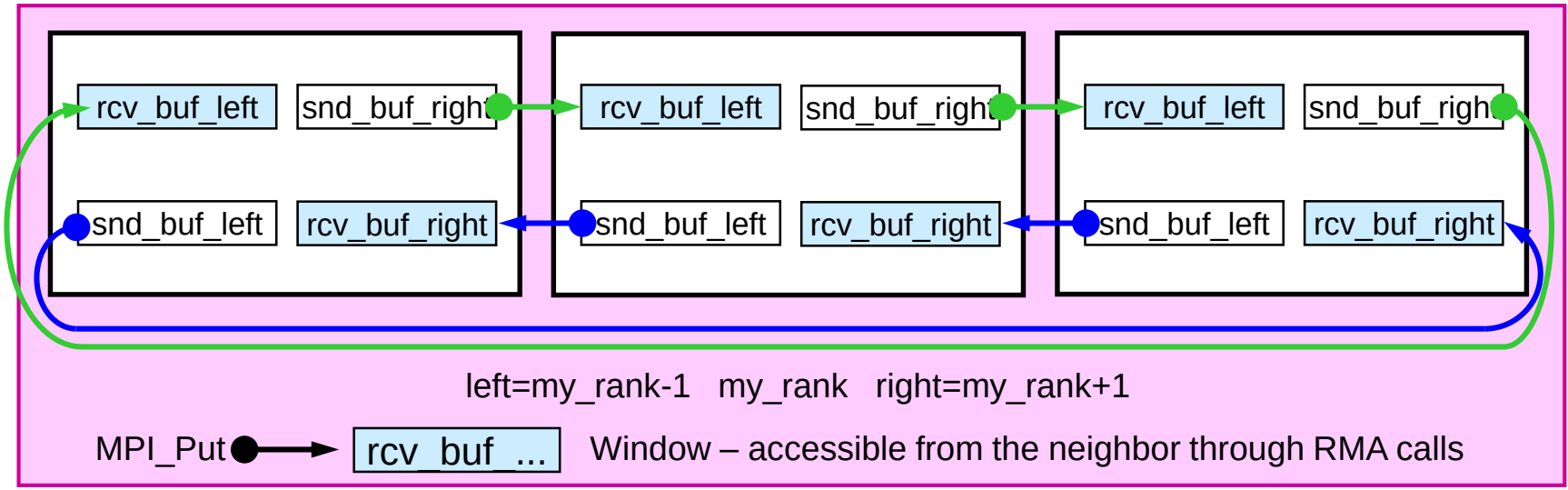
See HLRS online courses **http://www.hlrs.de/training/par-prog-ws/**
→ Practical → MPI.tar.gz

Summary of halo files (and some ring files)



Benchmark 1: Halo communication with MPI_Put

- Copy to your local directory and analyze the source code:
  cp ~/MPI/course/**C**/1sided/halo_1sided_put_win_alloc.c ***./***
  cp ~/MPI/course/**F_30**/1sided/halo_1sided_put_win_alloc_30.f90 ***./*** (or _20 with mpi module)
- halo… communicates along the 1-dim ring of processes in both directions
  - ➔ Into right direction: Put snd_buf_right into the rec_buf_left of the right neighbor
  - ← Into left direction: Put snd_buf_left into the rec_buf_right of the left neighbor

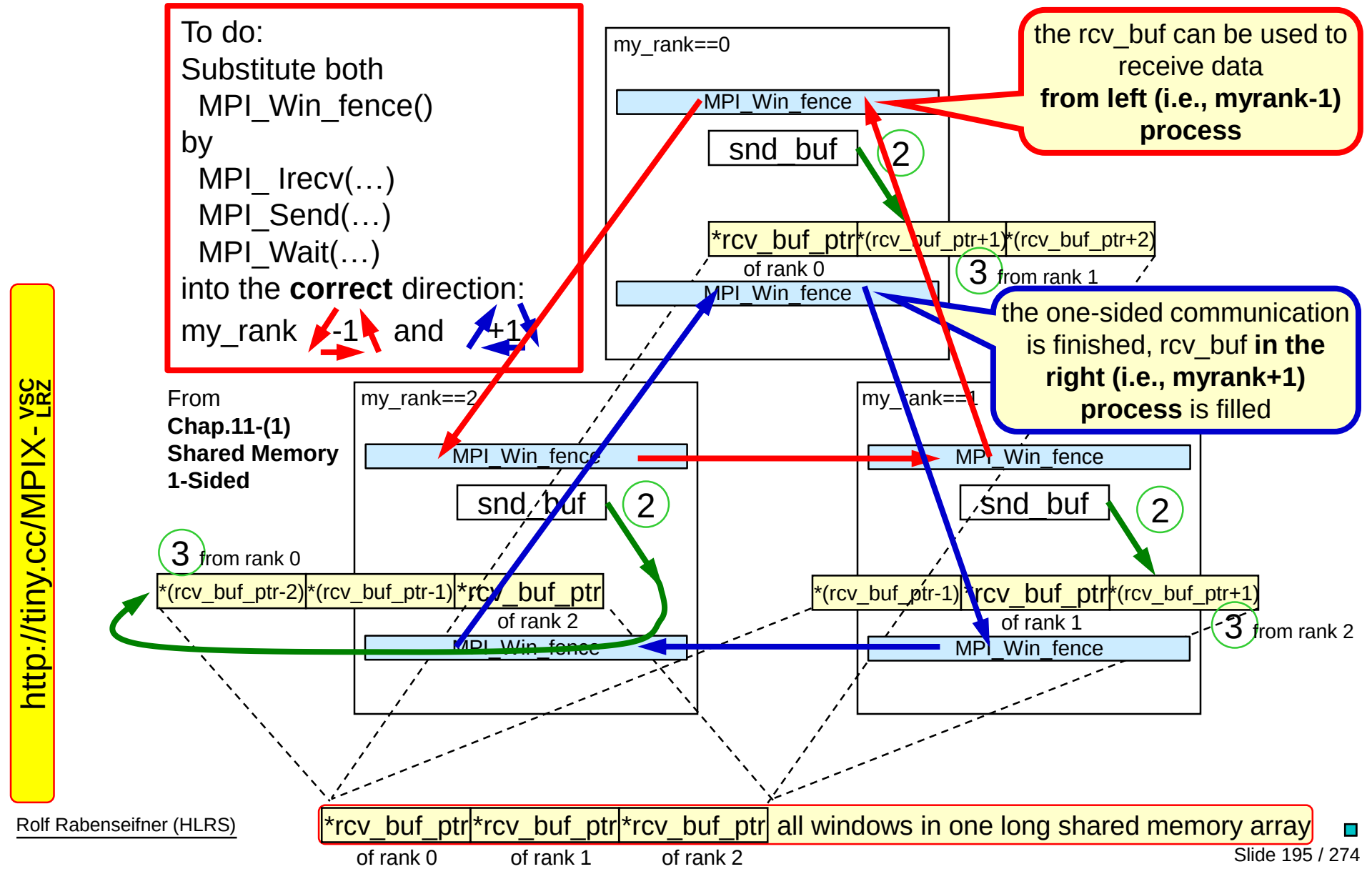


- Compile and run the original halo_1sided_put_win_alloc*.c/f90 program
  - With MPI processes on **4** cores & **all cores** of a shared memory node

Benchmark 2: Shared memory **halo** communication

- Copy to your local directory:
  cp ~/MPI/course/**C**/1sided/halo_1sided_**store**_win_alloc_**shared**.c **./**
  cp ~/MPI/course/**F_30**/1sided/halo_1sided_**store**_win_alloc_**shared_30.f90** **./**
  (or _20 with mpi module) (or _20 with mpi module)
- Compile and run shared memory program
  - With MPI processes on **4** cores & **all cores** of a shared memory node
- Compare latency and bandwidth
- Compare the source codes (with “diff”)

Exercise / Example 4: Ring – Using *other* synchronization



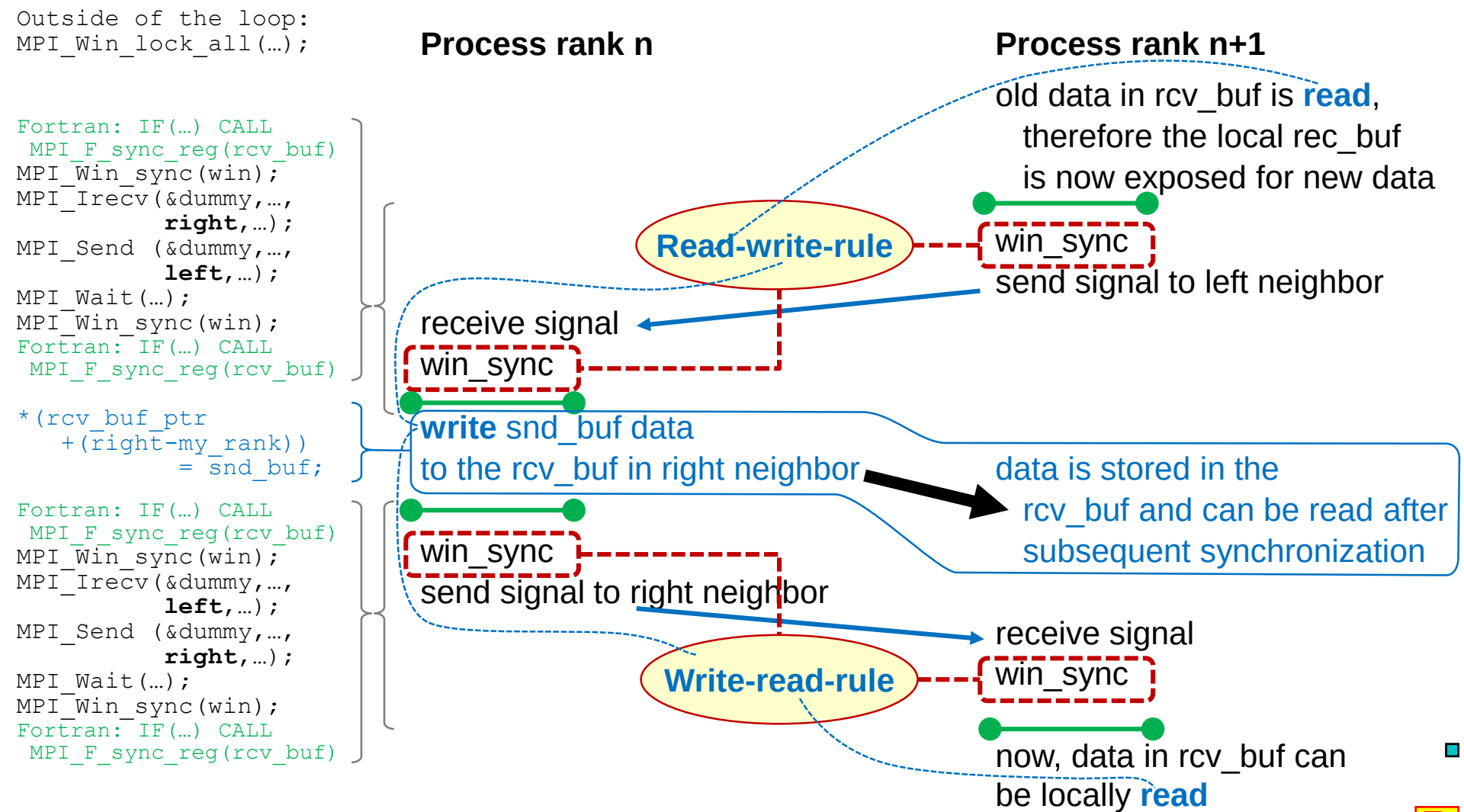
Exercise / Example 4: Ring – Using *other* synchronization

- Use your exa1 result or
  - ~/MPI/course/**C**/1sided/ring_1sided_store_win_alloc_shared.c
    ~/MPI/course/**F_30**/1sided/ring_1sided_store_win_alloc_shared_30.f90 (or _20)

  as your baseline **my_shared_exa3.c** *or* **…_20.f90** *or* **…_30.f90** for the following exercise:
- Tasks: Substitute the MPI_Win_fence synchronization by pt-to-pt communication
  - Use empty messages for synchronizing
  - Substitute the first MPI_Win_fence by ring-communication to the **left**, because it signals to the **left** neighbor that the local rcv_buf target is exposed for new data
    - MPI_Irecv(…right,…,&rq) ; MPI_Send(…**left**, …); MPI_Wait(&rq …);
  - Substitute the second Win_fence by ring-communication to the **right**, because it signals to the **right** neighbor that data is stored in the rcv_buf of the right neighb.
  - Local MPI_Win_sync is needed for write-read and read-write-rule
  - Requires (once, before the loop) MPI_Win_lock_all(MPI_MODE_NOCHECK, win);
    (and once after the loop) MPI_Win_unlock_all(win);
- Compile and run this program on 4 cores & **all cores** of a shared memory node.

Exercise / Example 4: Ring – Using *other* synchronization

- Communication pattern between each pair of neighbor processes



see also login-slides

Fortran: Are these the locations where MPI_F_SYNC_REG is needed?

Solution / Example 4: Ring with shared memory and MPI_Win_sync

C

MPI/course/C/Ch11/ring_1sided_store_win_alloc_shared_othersync.c

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

  *(rcv_buf_ptr+(right-my_rank)) = snd_buf;

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

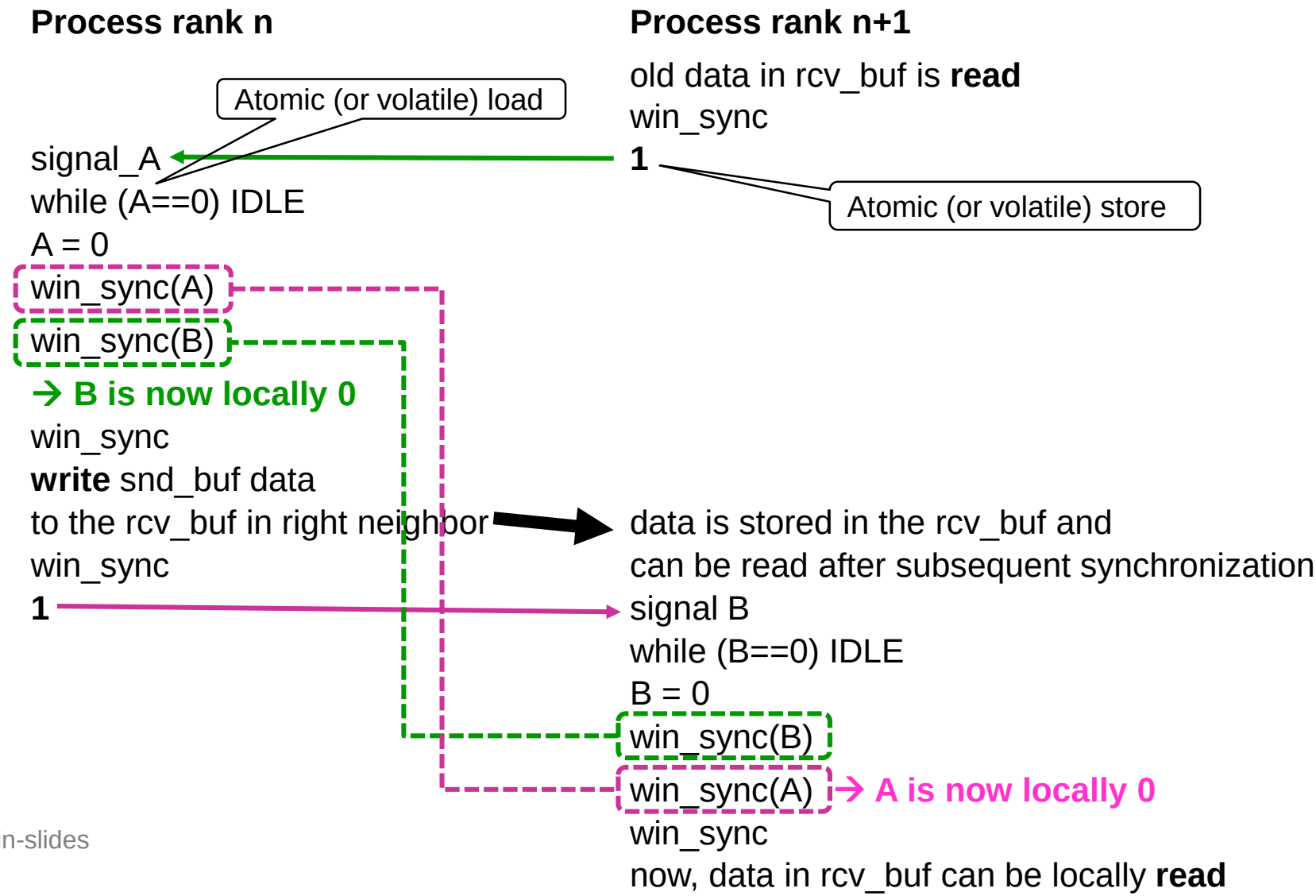
Instead of MPI_Win_fence(…)

Instead of MPI_Win_fence(…)

Example 5: Ring – with memory signals

- Goal:
  - Substitute the Irecv-Send-Wait communication by two shared memory flags
- Hints:
  - After initializing these shared memory variables with 0, an additional MPI_Win_sync + MPI_Barrier + MPI_Win_sync is needed
  - Normally, from three consecutive MPI_Win_sync, only one call may be needed, because one memory fence is enough
- Recommendation:
  - One may study and run both the solution files and compare the latency
    - **halo_1sided_store_win_alloc_shared_signal.c** (only solution in C)
    - **ring_1sided_store_win_alloc_shared_signal.c** (only solution in C)

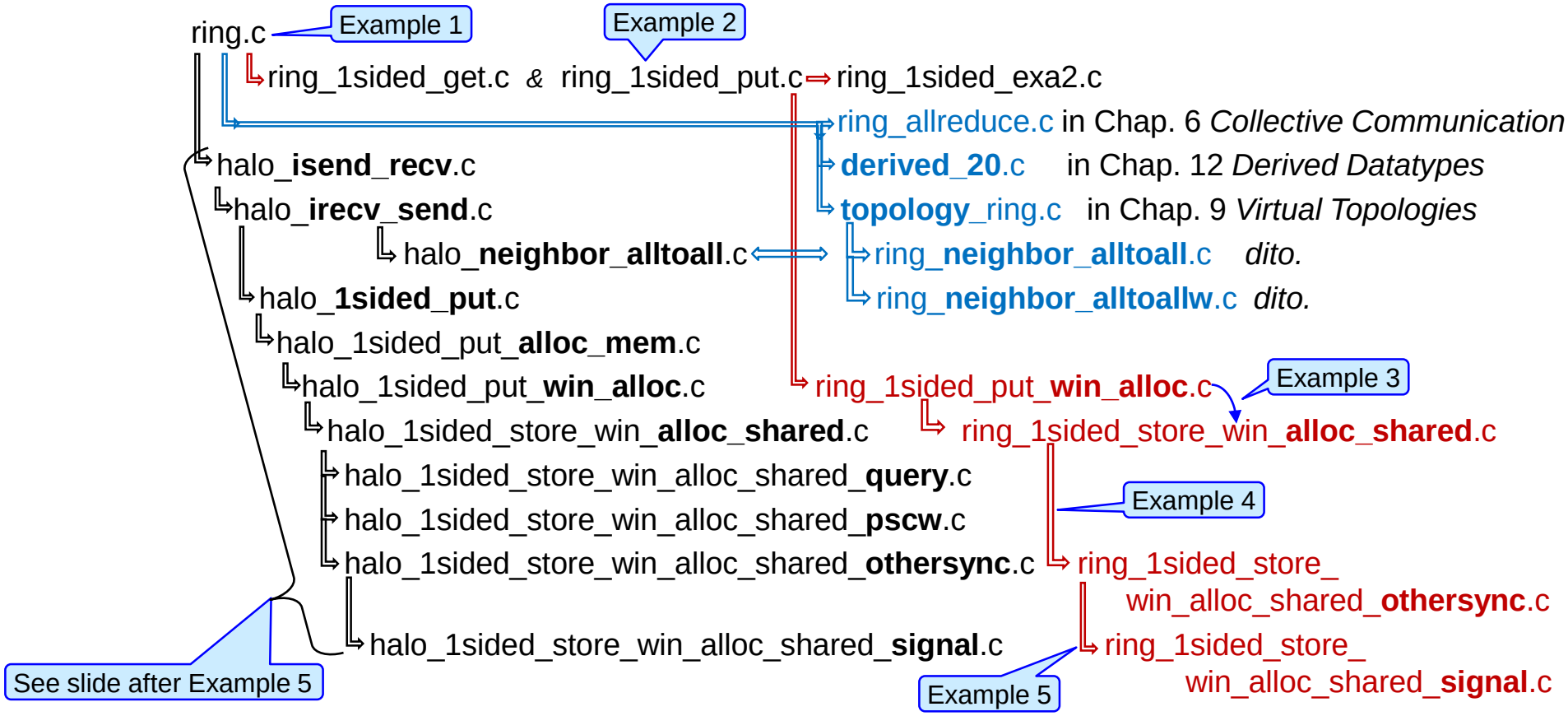
Example 5: Ring – with memory signals



see also login-slides

See HLRS online courses **http://www.hlrs.de/training/par-prog-ws/** → Practical → MPI.tar.gz

Summary of halo files (and some ring files)



Quiz on Shared Memory Model & Synchronization

A. Which MPI memory model applies to MPI shared memory?
   MPI_WIN_SEPARATE or MPI_WIN_UNIFIED ?

B. "Public and private copies are **. . . . ? . . . .** synchronized without additional RMA calls."

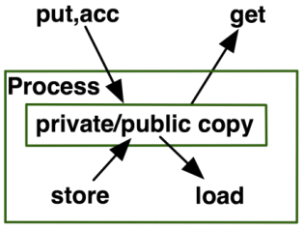


Figure: Courtesy of Torsten Hoefler

C. Which process-to-process synchronization methods can be used that, e.g., a store to a shared memory variable gets visible to another process (within the processes of the shared memory window)?

   1. _____
   2. _____
   3. _____

D. That such a store gets visible in another process after the synchronization is named here as "*write-read-rule*".
   Which other rules are implied by such synchronizations and what do they mean?

   1. _____
   2. _____

E. How can you define a **race-condition** and
   which problems arise from **cache-line false-sharing?**

   1. _____
   2. _____

Programming models
- pure MPI

Pure MPI communication

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

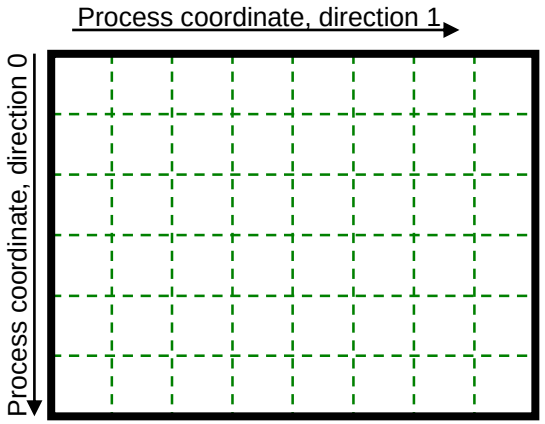
- Does application topology fit on hardware topology?
  - **Minimal communication between MPI processes AND between hardware SMP nodes**
- Does MPI library use different protocols internally?
  - **Shared memory inside of the SMP nodes**
  - **Network communication between the nodes**
- Is the network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

Programming models - pure MPI

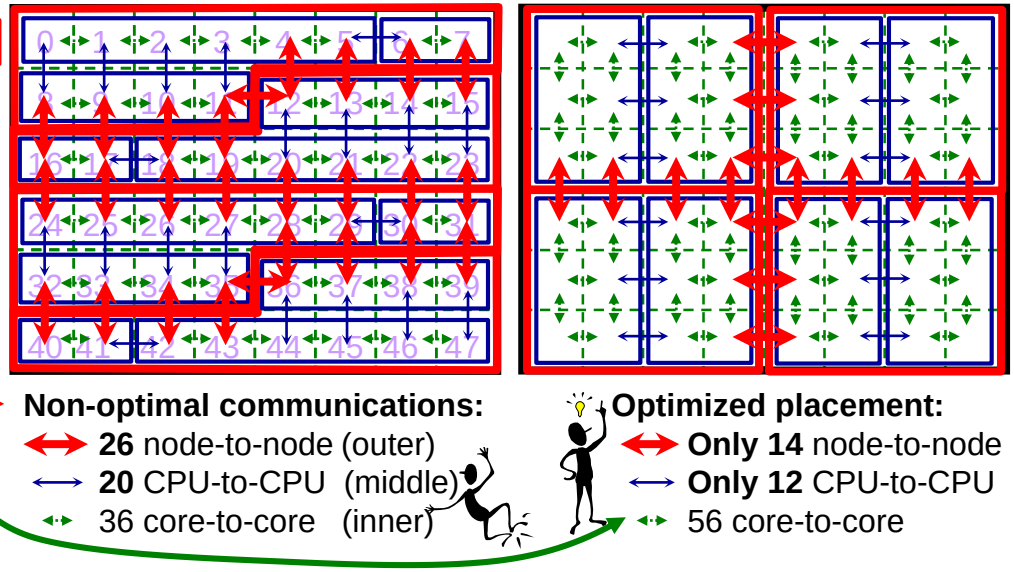
The Topology Problem

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

- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - → Subdomains as quadratic as possible
    - → **minimal circumference**
    - → **minimal halo communication**
  - → virtual 2-dim process grid: 6 x 8 with 1000 x 1010 mesh points/core



- Hardware example: 48 cores:
  - 4 ccNUMA nodes
  - each node with 2 CPUs
  - each CPU with 6 cores
- How to locate the MPI processes on the hardware?
  - Using sequential ranks in MPI_COMM_WORLD
  - Optimized placement
  - → See next slides and example code



Hierarchical Cartesian Domain Decomposition

Example:
24 SMP nodes
X
32 cores/node

Per node:
maximal
8+8+8+8+16+16*)=
48 or 64*)
connections
to neighbor
nodes

*) with cyclic communication

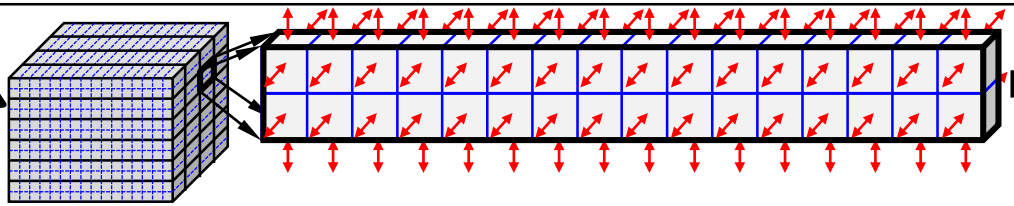
Primary and main optimization goal:
Whole communication from each node to all of its neighbors must be minimized!

Virtual location of an MPI process within an SMP node

All MPI processes of an SMP node

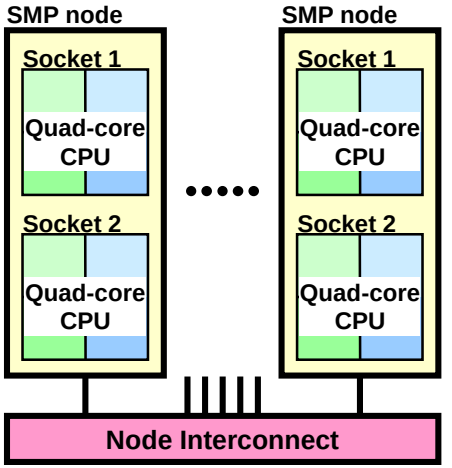
Second and minor optimization goal:
Whole intra-node communication must be minimized!

Without topology-optimization:
96 connections to other nodes

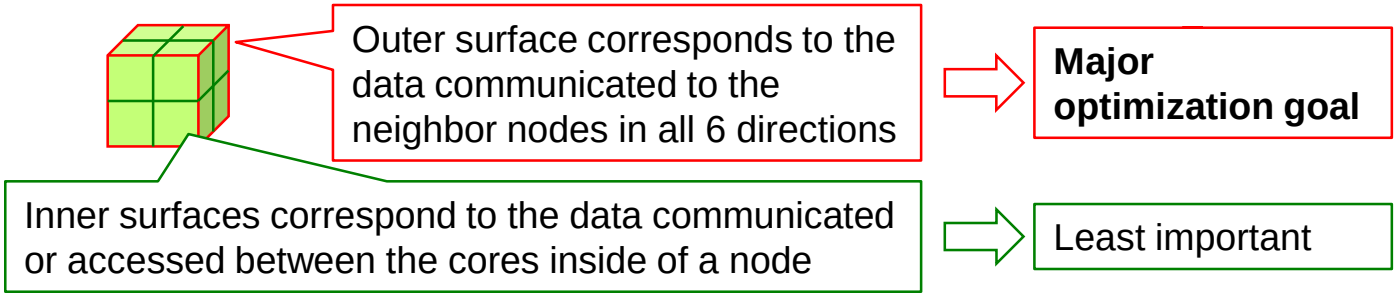


2 or 1.6*) times slower communication

Levels of communication & data access



- Three levels:
  - Between the SMP nodes
  - Between the sockets inside of a ccNUMA SMP node
  - Between the cores of a socket
- On all levels, the communication should be minimized:
  - With 3-dimensional sub-domains:
    - **They should be as cubic as possible = minimal surface = minimal communication**



    - **"as cubic as possible" may be qualified**
      due to different communication bandwidth in each direction
      caused by sending (fast) non-strided or (slow) strided data

Programming models
- pure MPI

The Topology Problem:
How to → *Virtual MPI Topologies*

Acknowledgement:
Virtual topology course slides are
based on the MPI-1 course of EPCC.

The topology problem: How to

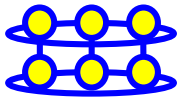
When do we need such rank re-numbering?

→ With more than 4 MPI processes per ccNUMA (SMP) node

- Communication win with 6 or 8 processes:
  - **Sequential (default) 6x1x1 or 8x1x1 topology → 26 or 34 inter-node neighbors**
  - **Renumbered 3x2x1 or 2x2x2 topology → 22 or 24 neighbors → 15% or 29% win**

How can we implement such rank re-numbering?

→ With MPI virtual topologies → A multi-dimensional process naming scheme



Major problems?

1. All MPI libraries provide the necessary interfaces ☺ ☺ ☺, but **without** re-numbering in nearly all MPI-libraries ☹ ☹ ☹
2. **The existing MPI-3.1 interfaces are not optimal**
   - **for cluster of ccNUMA node hardware,**
   - **nor for application specific data mesh sizes or direction-dependent bandwidth**
3. The application must be prepared for rank re-numbering

Example

Application data mesh

Virtual process grid

- Global array A(1:3000, 1:4000, 1:500) = $6 \cdot 10^9$ words
- on **3 x 4 x 5 = 60 processors**
- process coordinates **0..2, 0..3, 0..4**

- example:
  on process **ic_0=2, ic_1=0, ic_2=3 (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400) = $0.1 \cdot 10^9$ words

- **process coordinates:** handled with **virtual Cartesian topologies**
- Array decomposition: handled by the application program directly

Virtual Topologies

- Convenient process naming → *the process coordinates (if Cartesian)*
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications.

We try to provide a complete source code after EuroMPI 2018, see

- Christoph Niethammer and Rolf Rabenseifner. 2018. Topology aware Cartesian grid mapping with MPI. EuroMPI 2018. https://eurompi2018.bsc.es/ → Program → Poster Session → Abstract+Poster
- https://fs.hlrs.de/projects/par/mpi/EuroMPI2018-Cartesian/ → All info + slides + software
- http://www.hlrs.de/training/par-prog-ws/ → Practical → MPI.tar.gz → MPI/course/C/eurompi18/
- More details, see **„Parallel programming models on hybrid systems / MPI + OpenMP”** (this talk+slides) [former version: 

including MPI implementation infos]

Here, you get the new optimized interface + implementation + docu.

Another approach using the existing MPI_Cart_create() interface:

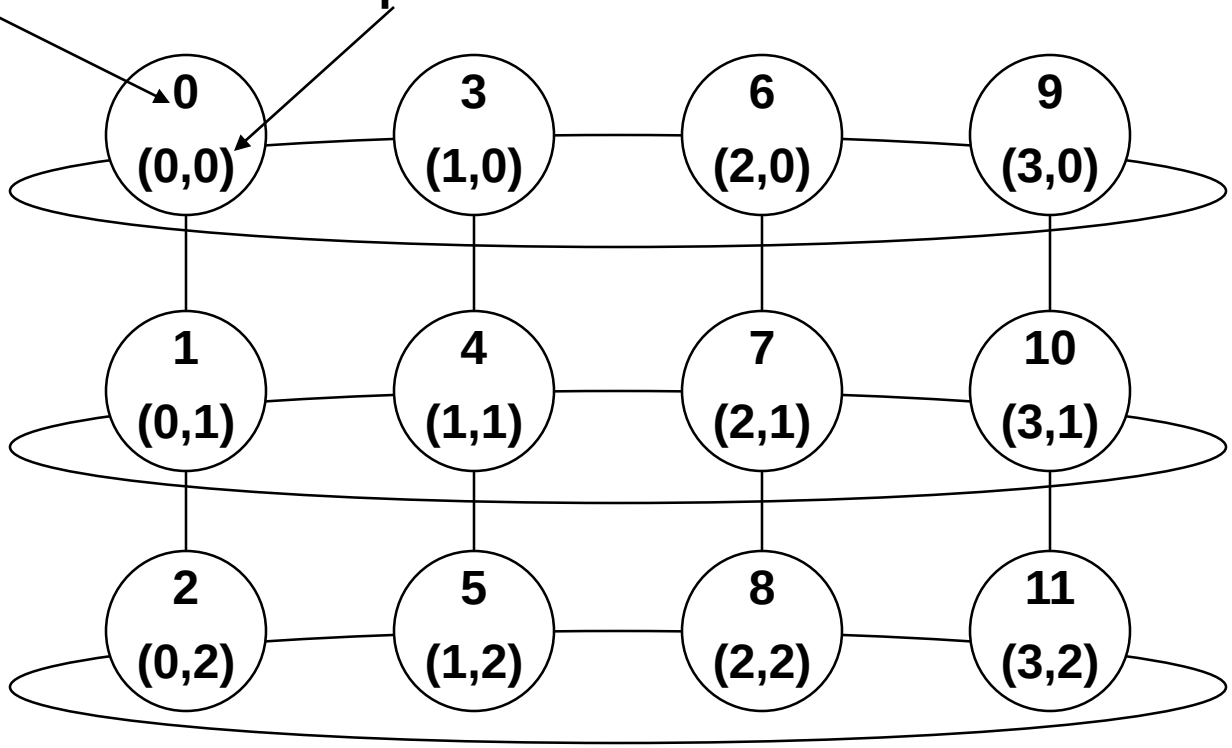
- W. D. Gropp, Using Node [and Socket] Information to Implement MPI Cartesian Topologies, Parallel Computing, 2019, and in: Proceedings of the 25th European MPI User' Group Meeting, EuroMPI'18, ACM, New York, NY, USA, 2018, pp. 18:1-18:9. doi:10.1145/3236367.3236377. Slides: http://wgropp.cs.illinois.edu/bib/talks/tdata/2018/nodecart-final.pdf

How to use a Virtual Topology

- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.

Example – A 2-dimensional Cylinder

- **Ranks** and **Cartesian process coordinates**



Topology Types

- Cartesian Topologies
  - each process is *connected* to its neighbor in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course,
    communication between any two processes is still allowed.

- Graph Topologies
  - general graphs,
  - two interfaces:
    - **MPI_GRAPH_CREATE** (since MPI-1)
    - **MPI_DIST_GRAPH_CREATE_ADJACENT &**
      MPI_DIST_GRAPH_CREATE (new scalable interface since MPI-2.2)
  - not covered here.

Creating a Cartesian Virtual Topology

C

Fortran

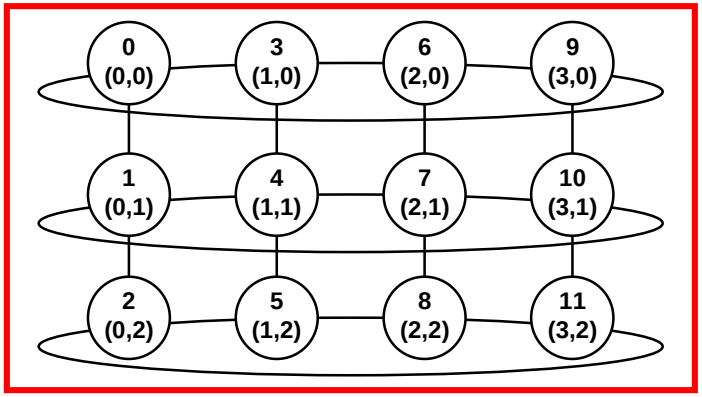
- C/C++: int MPI_Cart_create(MPI_Comm comm_old, int ndims,
  int *dims, int *periods, int reorder,
  MPI_Comm **comm_cart*)
- Fortran: MPI_CART_CREATE(comm_old, ndims, dims, periods,
  reorder, *comm_cart*, *ierror*)

mpi_f08:
```
TYPE(MPI_Comm)       :: comm_old, comm_cart
INTEGER              :: ndims, dims(*)
LOGICAL              :: periods(*), reorder
INTEGER, OPTIONAL    :: ierror
```

mpi & mpif.h:
```
INTEGER comm_old, ndims, dims(*), comm_cart, ierror
LOGICAL  periods(*), reorder
```

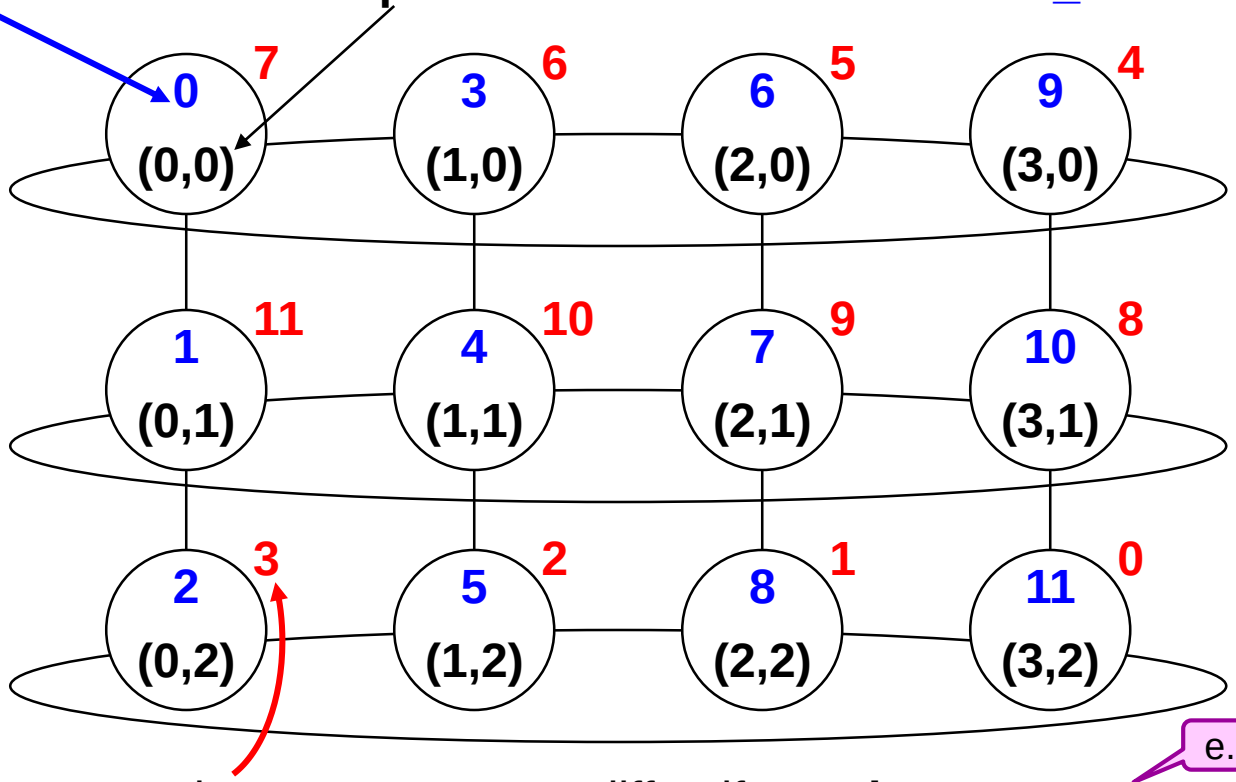
comm_old = MPI_COMM_WORLD
ndims = 2
dims = (4, 3)
periods = (1, 0) (in C)
periods = (.true., .false.) (in Fortran)
reorder = *see next slide*

e.g., size==12 factorized with MPI_Dims_create(), see advanced exercise



Example – A 2-dimensional Cylinder

- **Ranks** and **Cartesian process coordinates** in **comm_cart**



- Ranks in **comm** and **comm_cart** may differ, if **reorder == non-zero** or **.TRUE.** (e.g., 1)
- This reordering can allow MPI to optimize communications

Typical usage of MPI_Cart_create & MPI_Dims_create

```
#define ndims 3
int i, nnodes, world_myrank, cart_myrank, dims[ndims], periods[ndims], my_coords[ndims];  MPI_Comm  comm_cart;
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &world_myrank);
for (i=0; i<ndims; i++) { dims[i]=0; periods[i]=…; }
MPI_Dims_create(numprocs, ndims, dims); // computes factorization of numprocs
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods,1, &comm_cart);
MPI_Comm_rank(comm_cart, &cart_myrank);
MPI_Cart_coords(comm_cart, cart_myrank, ndims, my_coords, ierror)
```

From now, all communication should be based on
comm_cart & cart_myrank & my_coords

C

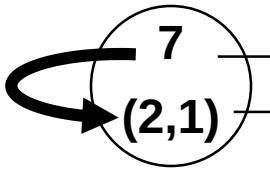
Fortran

- C/C++: int MPI_Dims_create(int nnodes, int ndims, int *dims)
- Fortran: MPI_DIMS_CREATE(nnodes, ndims, dims, IERROR)

```
mpi_f08:        INTEGER                 :: nnodes, ndims, dims(*)
                INTEGER, OPTIONAL       :: ierror
mpi & mpif.h:   INTEGER nnodes, ndims, dims(*), ierror
```

Array *dims* must be **initialized** with **zeros**
(other possibilities, see MPI standard)

Cartesian Mapping Functions



- Mapping ranks to virtual process grid coordinates

C

- C/C++: int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims, int **coords*)

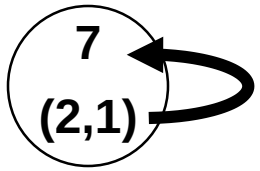
Fortran

- Fortran: MPI_CART_COORDS(comm_cart, rank, maxdims, *coords*, *ierror)*

```
mpi_f08:          TYPE(MPI_Comm)      :: comm_cart
                  INTEGER             :: rank, maxdims, coords(*)
                  INTEGER, OPTIONAL   :: ierror
mpi & mpif.h:     INTEGER  comm_cart, rank, maxdims, coords(*), ierror
```

Cartesian Mapping Functions

- Mapping process grid coordinates to ranks



C

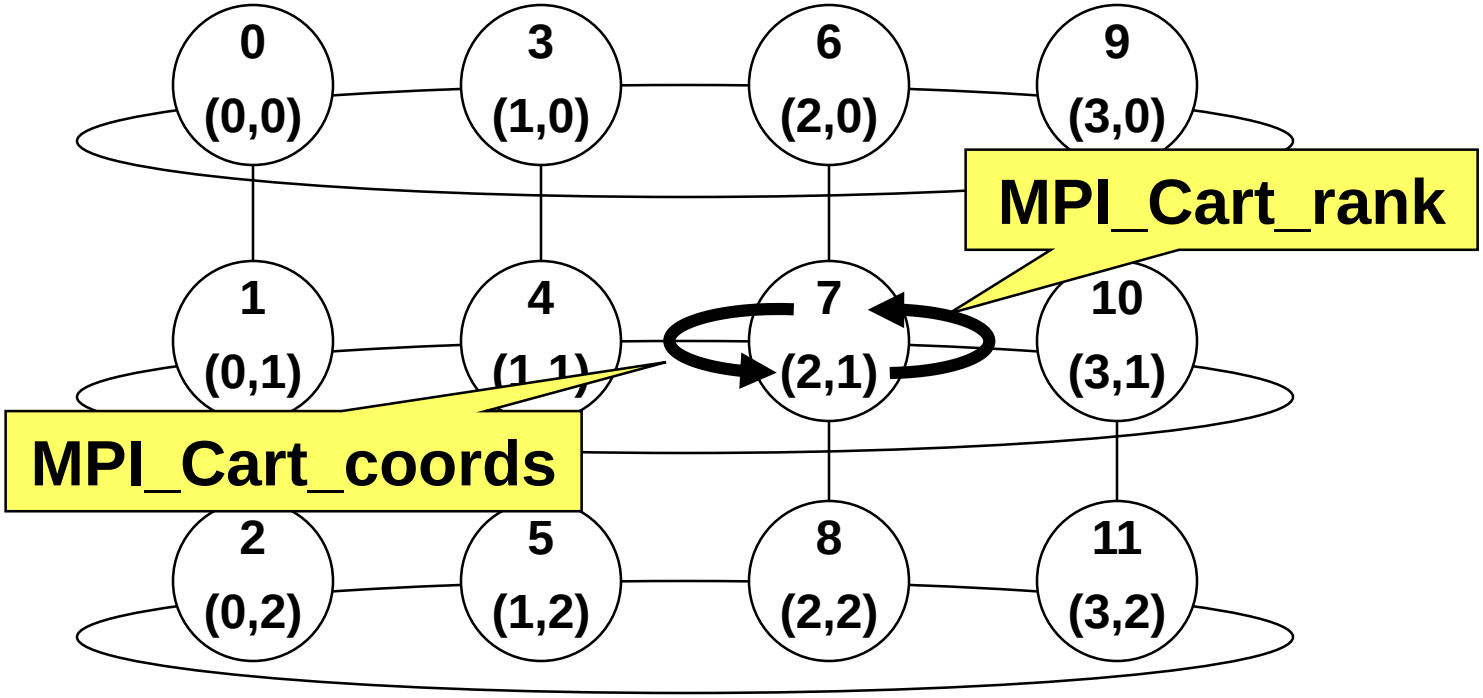
- C/C++: int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int **rank*)

Fortran

- Fortran: MPI_CART_RANK(comm_cart, coords, *rank*, *ierror*)

```
mpi_f08:          TYPE(MPI_Comm)       :: comm_cart
                  INTEGER              :: coords(*), rank
                  INTEGER, OPTIONAL    :: ierror
mpi & mpif.h:     INTEGER  comm_cart, coords(*), rank, ierror
```

Own coordinates



- Each process gets its own coordinates with (example in Fortran)

  CALL MPI_Comm_rank(comm_cart, ***my_rank***, *ierror*)

  CALL MPI_Cart_coords(comm_cart, **my_rank**, maxdims, ***my_coords***, *ierror*)

Cartesian Mapping Functions

- Computing ranks of neighboring processes

C

- C/C++: int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp, int **rank_source*, int **rank_dest*)

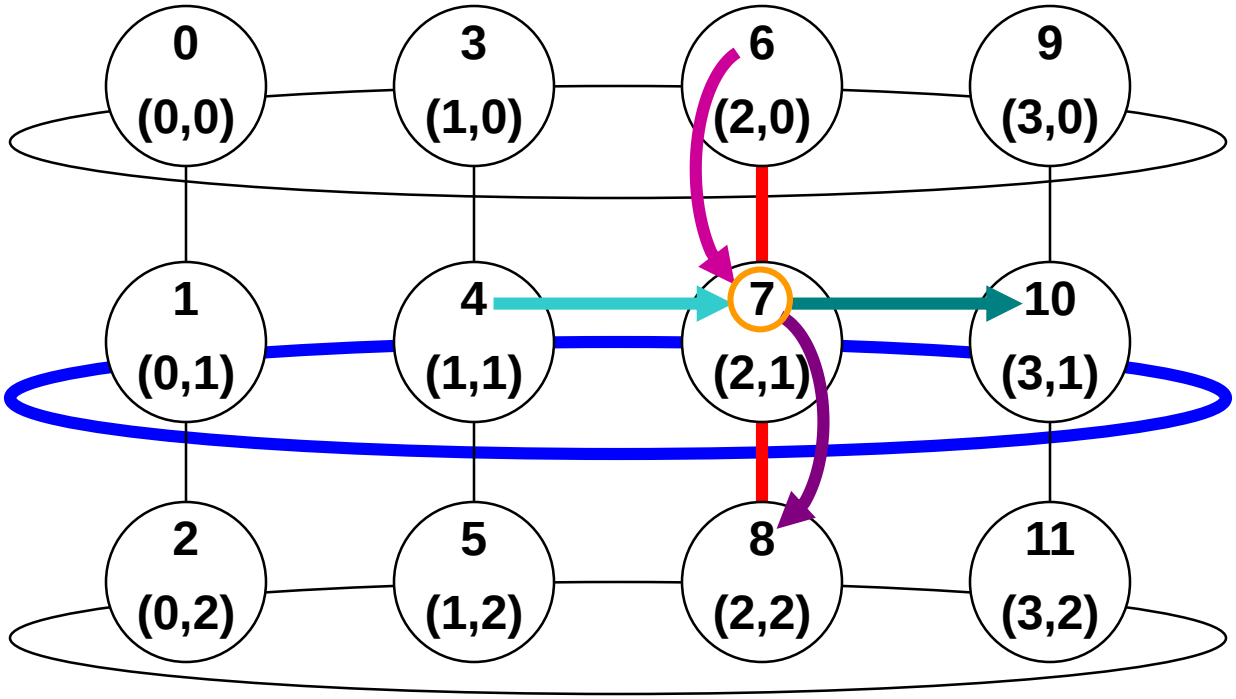
Fortran

- Fortran: MPI_CART_SHIFT(comm_cart, direction, disp, *rank_source*, *rank_dest*, *ierror*)

```
mpi_f08:          TYPE(MPI_Comm)       :: comm_cart
                  INTEGER              :: direction, disp, rank_source, rank_dest
                  INTEGER, OPTIONAL    :: ierror
mpi & mpif.h:     INTEGER  comm_cart, direction, disp, rank_source, rank_dest, ierror
```

- Returns MPI_PROC_NULL if there is no neighbor.
- MPI_PROC_NULL can be used as source or destination rank in each communication ➔ Then, this communication will be a no-operation!

MPI_Cart_shift – Example



(...) invisible input argument: **my_rank** in comm_cart

CALL MPI_Cart_shift (comm_cart, direction, disp, *rank_source*, *rank_dest*, *ierror*)

example on	**0** or	**+1**	**4**	**10**
process rank= **7**	**1**	**+1**	**6**	**8**

Cartesian Partitioning

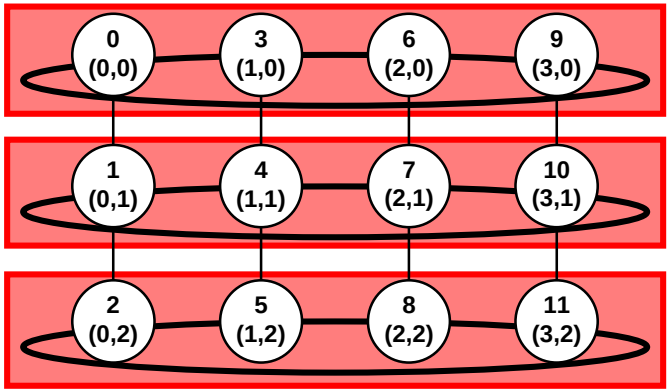
- Cut a grid up into *slices*.
- A new communicator is produced for each slice.
- Each slice can then perform its own collective communications.

C

- C/C++: int MPI_Cart_sub(MPI_Comm comm_cart, int *remain_dims, MPI_Comm **comm_slice*)

Fortran

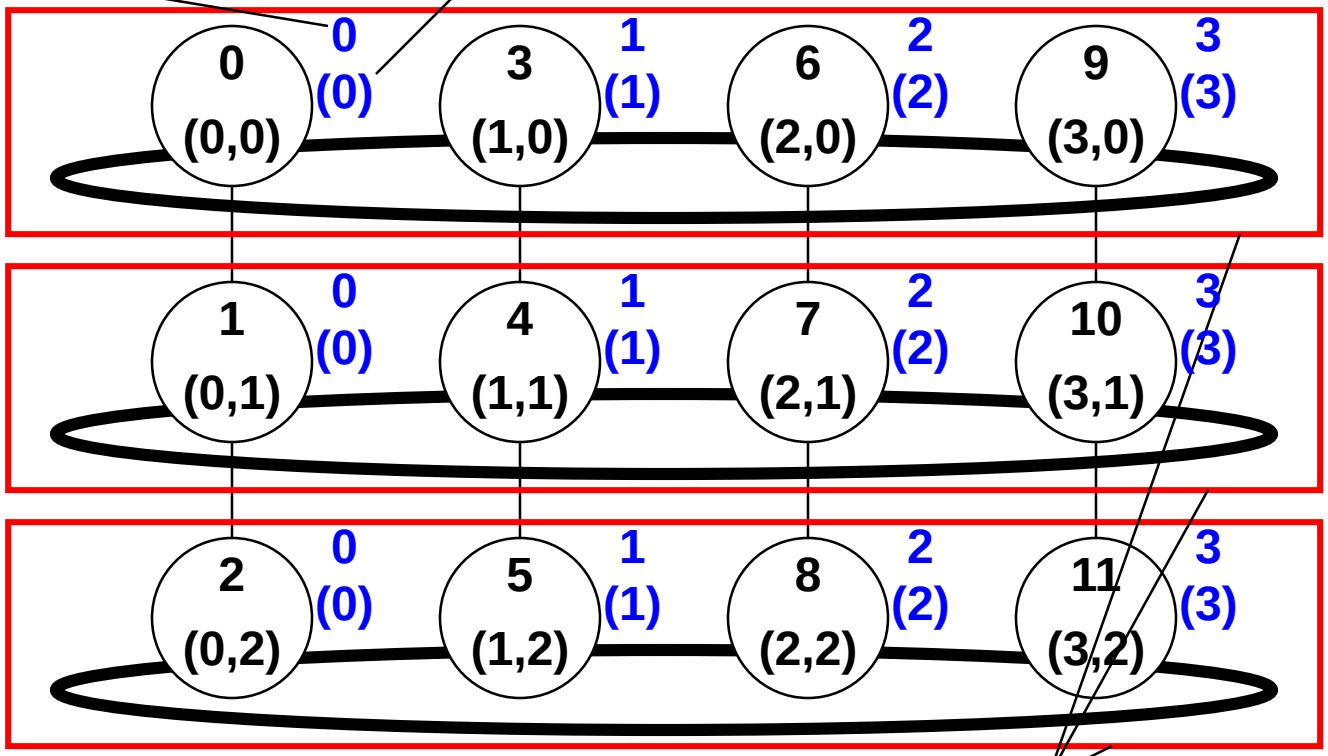
- Fortran: MPI_CART_SUB(comm_cart, remain_dims, *comm_slice*, *ierror*)




```
mpi_f08:        TYPE(MPI_Comm)      :: comm_cart
                LOGICAL             :: remain_dims(*)
                TYPE(MPI_Comm)      :: comm_slice
                INTEGER, OPTIONAL   :: ierror
mpi & mpif.h:   INTEGER comm_cart, comm_slice, ierror
                LOGICAL remain_dims(*)
```

MPI_Cart_sub – Example

- **Ranks** and **Cartesian process coordinates** in **comm_slice**



- CALL MPI_Cart_sub(comm_cart, remain_dims, ***comm_slice***, *ierror*)

(true, false)

Each process gets only its own sub-communicator

Sparse Collective Operations on Process Topologies

New in MPI-3.0

- Communication along MPI process topologies (Cartesian and (distributed) graph)
  - MPI_(I)NEIGHBOR_ALLGATHER(V)
  - MPI_(I)NEIGHBOR_ALLTOALL(V,W)
- If the topology is the full graph, then neighbor routine is identical to full collective communication routine
  - Exception: s/rdispls in MPI_NEIGHBOR_ALLTOALLW are MPI_Aint
- Allows for optimized communication scheduling and scalable resource binding
- Cartesian topology:
  - Sequence of buffer segments is communicated with:
    - **direction=0 source, direction=0 dest, direction=1 source, direction=1 dest, …**
  - Defined only for disp=1 (direction, source, dest and disp are defined as in MPI_CART_SHIFT)
  - If a source or dest rank is MPI_PROC_NULL then the buffer location is still there but the content is not touched.
  - See 2nd and 3rd advanced exercise
- See Examples 8.9 and 8.10 in MPI-4.0, pages 420 and 424.
- **Caution:** Most implementations have a **bug** for MPI_NEIGHBOR_ALLTOALL(V,W) in case of **ndims[i]==1 or 2** and **periods[i]=true**: the two receive (or send) buffers are interchanged → Solutions expected for 2020 in all libs! → https://github.com/mpi-forum/mpi-issues/issues/153

How to use several independent buffers within one MPI call

Other MPI features: MPI_BOTTOM and absolute addresses

- MPI_BOTTOM in point-to-point and collective communication:
  - Buffer argument is MPI_BOTTOM
  - Then absolute addresses can be used in
    - **Communication routines with byte displacement arguments, e.g., MPI_(I)NEIGHBOR_ALLTOALLW**
    - **Derived datatypes with byte displacements**
  - Displacements must be retrieved with MPI_GET_ADDRESS()
  - MPI_BOTTOM is an address, i.e., **cannot be assigned to a Fortran variable**!
  - MPI-3.1/MPI-4.0, Section 2.5.4, page 15 line 45 – page 16 line 6 / page 21 lines 14-23 shows all such address constants that cannot be used in expressions or assignments **in Fortran,** e.g.,
    - **MPI_STATUS_IGNORE (→ point-to-point comm.)**
    - **MPI_IN_PLACE (→ collective comm.)**
  - Fortran: Using MPI_BOTTOM & absolute displacement of variable X → **<type>, ASYNCHRONOUS :: X** and **MPI_F_SYNC_REG(X)** is needed:
    - **MPI_BOTTOM in a blocking MPI routine → MPI_F_SYNC_REG before and after this routine**
    - **in a nonblocking routine → MPI_F_SYNC_REG before this routine & after final WAIT/TEST**

Fortran

Back to the problems

1. All MPI libraries provide the necessary interfaces ☺ ☺ ☺,
   but **without** re-numbering in nearly all MPI-libraries ☹ ☹ ☹
   - **You may substitute MPI_Cart_create()**
     by the software solution of Bill Gropp (see Bill Gropp, EuroMPI 2018)
   - **Will hopefully resolved in many MPI libraries in 2020: e.g., HPE MPI (MPT) 3/2020**
2. **The existing MPI-3.1 interfaces are not optimal**
   - **for cluster of ccNUMA node hardware,**
     - **We substitute MPI_Dims_create() + MPI_Cart_create()**
       by MPIX_Cart_weighted_create(… MPIX_WEIGHTS_EQUAL …)
   - **nor for application specific data mesh sizes or direction-dependent bandwidth**
     - **by MPIX_Cart_weighted_create(… weights ….)**

   Reflecting the application data mesh

   → https://github.com/mpi-forum/mpi-issues/issues/120
3. **Caution: The application must be prepared for rank re-numbering**
   - **All communication through the newly created Cartesian communicator with re-numbered ranks!**
   - **One must not load data based on MPI_COMM_WOLRD ranks!**

MPIX routines, courtesy of Christoph Niethammer, HLRS

Goals of Cartesian MPI_Cart/Dims_create

- Given: comm_old (e.g., MPI_COMM_WORLD), ndims (e.g., 3 dimensions)
- Provide
  - a **factorization** of #processes (of comm_old) into the dimensions **dims[i]**$_{i=1..ndims}$
  - a Cartesian communicator **comm_cart**
  - a **optimized reordering** of the ranks in comm_old into the ranks of comm_cart to minimize the Cartesian communication time, e.g., of
    - MPI_Neighbor_alltoall
    - Equivalent communication pattern implemented with
      - MPI_Sendrecv
      - Nonblocking MPI point-to-point communication

The limits of MPI_Dims_create + MPI_Cart_create

- Not application topology aware
  - MPI_Dims_create can **only** map **evenly balanced** Cartesian topologies
    - Factorization of 48,000 processes into 20 x 40 x 60 processes
      (e.g. for a mesh with 200 x 400 x 600 mesh points)
      → no chance with current interface
- Only partially hardware topology aware
  - MPI_Dims_create has no communicator argument → not hardware aware
    - An application mesh with 3000x3000 mesh points
      on 25 nodes x 24 cores (=600 MPI processes)
      - Answer from MPI_Dims_create:
        - » 25 x 24 MPI processes
        - » Mapped by most libraries to 25 x 1 nodes
          with 120x3000 mesh points per node
          ➔ too much node-to-node communication

Major problems:
- No weights, no info
- Two separated interfaces for two common tasks:
  - Factorization of #processes
  - Mapping of the processes to the hardware

Goals of Cartesian MPI_Cart/Dims_create

- Remark: On a hierarchical hardware,
  - **optimized factorization and reordering** typically means **minimal node-to-node** communication,
  - which typically means that the communicating surfaces of the data on each node is as quadratic as possible (or the subdomain as cubic as possible)

- The current API, i.e.,
  - due to the missing weights
  - and the non-hardware aware MPI_Dims_create,

  does **not** allow such an optimized factorization and reordering in many cases.

The new interface – proposed for MPI-4.1

- **MPI_Dims_create_weighted** (

```
/*IN*/      int        nnodes,
/*IN*/      int        ndims,
/*IN*/      int        dim_weights[ndims],
/*IN*/      int        periods[ndims], /* for future use in
                                          combination with info */
/*IN*/      MPI_Info   info, /* for future use, currently MPI_INFO_NULL */
/*INOUT*/   int        dims[ndims]);
```

input for application-topology-awareness

  - Arguments have same meaning as in MPI_Dims_create
  - Goal (in absence of an info argument):
    - dims[i]•dim_weights[i] should be as close as possible,
    - i.e., the $\sum_{i=0..(ndims-1)}$ dims[i]•dim_weights[i] as small as possible (advice to implementors)

A new courtesy function: **Weighted factorization**

The new interface – proposed for MPI-4.1, continued

input for hardware-awareness

input for application-topology-awareness

- **MPI_Cart_create_weighted** (

```
/*IN*/      MPI_Comm    comm_old,
/*IN*/      int         ndims,
/*IN*/      int         dim_weights[ndims], /*or MPI_UNWEIGHTED*/
/*IN*/      int         periods[ndims],
/*IN*/      MPI_Info    info,       /* for future use, currently MPI_INFO_NULL */
/*INOUT*/   int         dims[ndims],
/*OUT*/     MPI_Comm    *comm_cart );
```

The new application & hardware topology aware interface

  - Arguments have same meaning as in MPI_Dims_create & MPI_Cart_create
  - See next slide for meaning of dim_weights[ndims]
  - Goal: chooses
    - an ndims-dimensional factorization of #processes of comm_old (→ dims)
    - and an appropriate reordering of the ranks (→ comm_cart),

    such that the execution time of a communication step along the virtual process grid (e.g., with MPI_NEIGHBOR_ALLTOALL or equivalent calls to MPI_SENDRECV as in the example in previous course Chapter 9.2) is as small as possible.

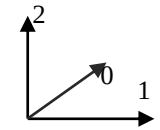
How to specify the dim_weights?

- Given: comm_old (e.g., MPI_COMM_WORLD), ndims (e.g., 3 dimensions)
- This means, **the domain decomposition has not yet taken place**!

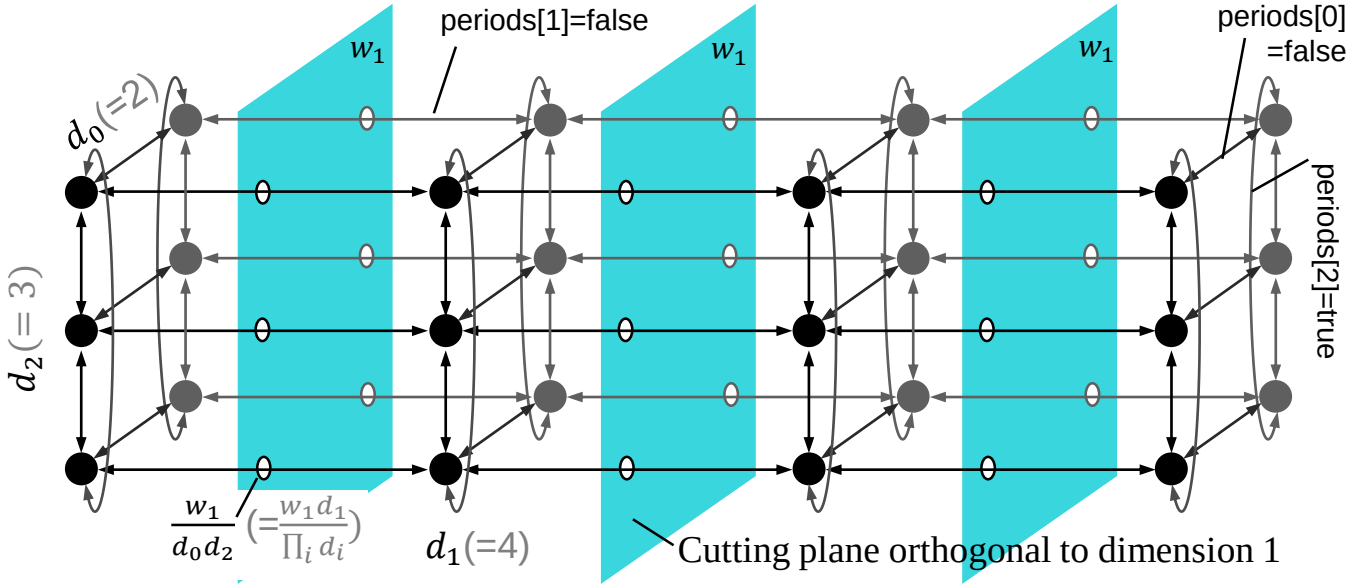
- Goals for dim_weights and the API at all:
  - Easy to understand
  - Easy to calculate
  - Relevant for typical Cartesian communication patterns (MPI_Neighbor_alltoall or alternatives)
  - Rules fit to usual design criteria of MPI
    - E.g., reusing MPI_UNWEIGHTED → integer array
    - Can be enhanced by vendors for their platforms → additional info argument for further specification
    - To provide also the less optimal two stage interface (in addition to the combined routine)

The dim_weights[i], example with 3 dimensions

Abbreviations:
d_i = dims[i]
w_i = dim_weights[i]
with
i = 0..(ndims-1)

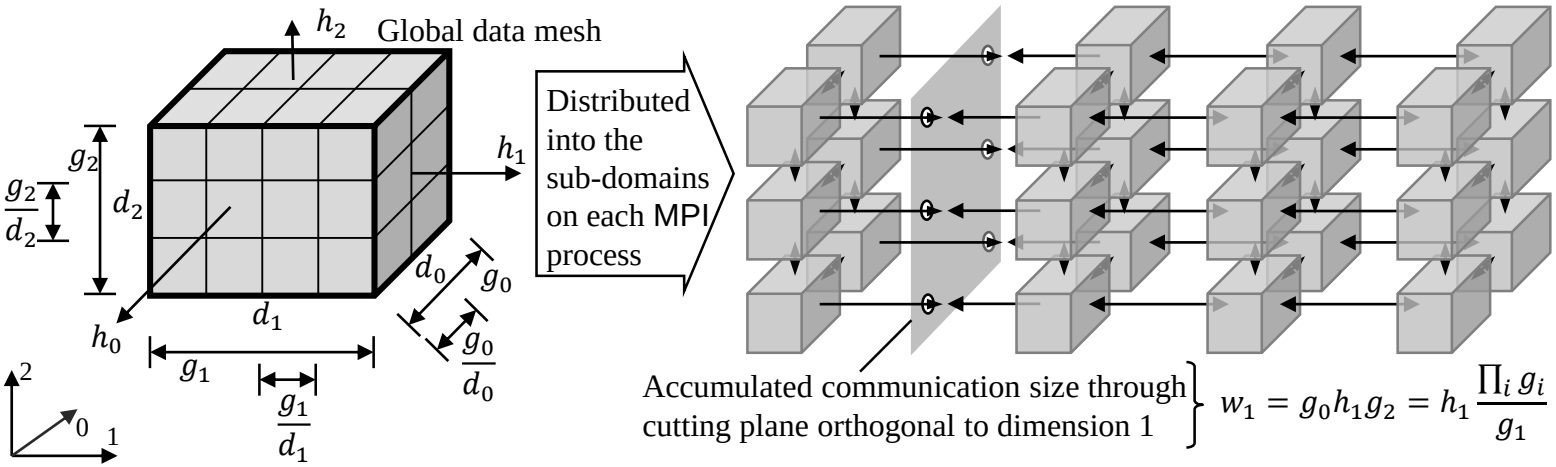


Three dimensions,
i.e., ndims=3



The arguments **dim_weights[i]** i =0::(ndims-1), abbreviated with w_i, should be specified as the accumulated message size (in bytes) communicated in one communication step through each **cutting plane** orthogonal to dimension d_i and in each of the two directions.

The dim_weights[i], example with 3 dimensions, continued



Abbreviations: g_i = data mesh size in dimension i, i=0..(ndims-1), w_i = dim_weights[i],
h_i = halo width in dimension i, d_i = dims[i]

Example for the calculation of the accumulated communication size $w_{i,i=0..2}$ in each dimension.

- g_i – The data mesh sizes $g_{i,i=0..2}$ express the three dimensions of the total application data mesh.
- h_i – The value h_i represents the halo width in a given direction when the 2-dimensional side of a subdomain is communicated to the neighbor process in that direction.

Output from MPI_Cart/Dims_create_weighted: The dimensions $d_{i,i=0..2}$

Important:

- The definition of the dim_weights (= w_i in this figure) is independent of the total number of processes and its factorization into the dimensions (= d_i in this figure)
- Result was $w_i = h_i \frac{\prod_j g_j}{g_i}$

Simple answers to our problems / examples

- Existing API is not application topology aware
  - Factorization of 48,000 processes into 20 x 40 x 60 processes
    → no chance with current API
    (e.g. for a mesh with 200 x 400 x 600 mesh points)
  - Use MPI_Cart_create_weighted with the dim_weights=(N/200, N/400, N/600)
    with N=200•400•600

- Existing API is only partially hardware topology aware
  - An application mesh with 3000x3000 mesh points (i.e., example with MPI_UNWEIGHTED)
    on 25 nodes x 24 cores (=600 MPI processes)
    - Current API must factorize into 25 x 24 MPI processes
      - » 25 x 1 nodes ➔ 120x3000 mesh points ➔ too much node to node communication
    - Optimized answer from MPI_Cart_create_weighted may be:
      - » 30 x 20 MPI processes
      - » Mapped to 5 x 5 nodes with 600x600 mesh points per node
        ➔ minimal node-to-node communication

The new interfaces – a real implementation

Substitute for / enhancement to existing MPI-1

- MPI_Dims_create (size_of_comm_old, ndims, *dims[ndims]*);
- MPI_Cart_create (comm_old, ndims, dims[ndims], periods, reorder, **comm_cart*);

Substitute for / enhancement to existing MPI-1
MPI_Dims_create (size_of_comm_old, ndims, *dims*);
MPI_Cart_create (comm_old, ndims, dims, periods, reorder, **comm_cart*);

New: (in MPI/tasks/C/Ch9/MPIX/)

- **MPIX_Cart_weighted_create** (

```
/*IN*/      MPI_Comm   comm_old,
/*IN*/      int        ndims,
/*IN*/      double     dim_weights[ndims], /*or MPIX_WEIGHTS_EQUAL*/
/*IN*/      int        periods[ndims],
/*IN*/      MPI_Info   info,        /* for future use, currently MPI_INFO_NULL */
/*INOUT*/   int        dims[ndims],
/*OUT*/     MPI_Comm   *comm_cart );
```

  - Arguments have same meaning as in MPI_Dims_create & MPI_Cart_create
  - See next slide for meaning of dim_weights[ndims]

- **MPIX_Dims_weighted_create** (int nnodes, int ndims, **double** dim_weights[ndims],
  */*OUT*/ int dims[ndims]*);

Further Interfaces

We proposed the algorithm in

- Christoph Niethammer and Rolf Rabenseifner. 2018.
  Topology aware Cartesian grid mapping with MPI.
  EuroMPI 2018. https://eurompi2018.bsc.es/
  → Program → Poster Session → Abstract+Poster
- https://fs.hlrs.de/projects/par/mpi/EuroMPI2018-Cartesian/
  → All info + slides + software
- http://www.hlrs.de/training/par-prog-ws/
  → Practical → MPI31.tar.gz → MPI/tasks/C/eurompi18/

Here, you get the new optimized interface + implementation + documentation

MPIX_Dims_weighted_create() is based on the ideas in:

- Jesper Larsson Träff and Felix Donatus Lübbe. 2015.
  Specification Guideline Violations by MPI Dims Create.
  In *Proceedings of the 22nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 19, 2 pages.

Full paper:

- Christoph Niethammer, Rolf Rabenseifner:
  An MPI interface for application and hardware aware cartesian topology optimization.
  EuroMPI 2019. Proceedings of the 26th European MPI Users' Group Meeting, September 2019, article No. 6, pages 1-8, https://doi.org/10.1145/3343211.3343217

MPIX routines, courtesy of Christoph Niethammer, HLRS

Remarks

- The portable MPIX routines internally use MPI_Comm_split_type(…, **MPI_COMM_TYPE_SHARED**, …) to split comm_old into ccNUMA nodes,
- plus (may be) additional splitting into NUMA domains.
- With using hyperthreads, it ***may be helpful*** to apply **sequential** ranking to the hyperthreads,
  - i.e., in MPI_COMM_WORLD, ranks 0+1 should be
    - **the first two hyperthreads**
    - of the first core
    - of the first CPU
    - of the first ccNUMA node
- Especially with weights w_i based on $\frac{G}{g_i}$, it is important
  - that the data of the mesh points is **not** read in based on (**old**) ranks in MPI_COMM_WORLD,
  - because the domain decomposition must be done based on **comm_cart** and its dimensions and (**new**) ranks

Major prob… cluster of ccNUMA nodes?

Where are we?

MAJOR PROBLEMS

- …usual programming models do **not** really fit: **Pure MPI / Hybrid MPI+OpenMP**
- Where are major problems? – And how to address?
  - Significant differences between **(small) inter-node bandwidth** and (high) intra-node bandwidth
    - → **How to minimize inter-node communication**
      - Hybrid MPI + OpenMP
      - Pure MPI + optimized virtual topologies

Here, only if many MPI processes per ccNUMA node

Here, optimization is always needed

  - **Replicated user data**
    - → **Waste of memory with pure MPI**
      - Hybrid MPI + OpenMP
      - Pure MPI + MPI-3 shared memory model
  - Processes and threads may **move** between cores and CPUs within ccNUMA nodes & fixed memory locations after ***"first touch"***
    - → **How to keep control?**
      - Pinning of threads and processes to the hardware
      - Explicit first touch programming with OpenMP, …
  - Are your application AND libraries **prepared for** MPI+**OpenMP?**

Exercise 1a: Adding a Cartesian Topology

- Given: a 3-D halo communication benchmark using irecv + send
  - halo_irecv_send_toggle_3dim_grid_skel.c
    - **cp ~/MPI/course/C/Ch9/MPIX_*/* .**
    - **The application halo_irecv_send_toggle_3dim_grid_skel.c uses a 3-D Cartesian communicator,**
    - **which is split into 1-D line communicators for communicating in the 3 dimensions**

My apologies for missing Fortran

- Overview:
  - *"complementing"* the not reordered Cartesian topology (cart_method==1) through optimizing algorithms (cart_method==2,3,4)
    - **cart_method==2: Add MPIX_Cart… (…MPIX_WEIGHTS_EQUAL…)**
    - **cart_method==3: Calculate the weights based on meshsize_avg_per_proc_startval Add MPIX_Cart… (…weights…)**
    - **cart_method==4: same as with cart_method==3, but without weights-calculation**

See **/* TODO** lines

  - Measure the communication bandwidth win
    - **For default meshsize 2 / 2 / 2**
    - **For other meshsizes, e.g., 1 / 2 / 4**

http://tiny.cc/MPIX-LRZ

http://tiny.cc/MPIX-VSC alternative for the exercises

Exercise 1a: Explanations

Start a 8 or 12-node batch-job with your own input file:
Report your acceleration factors to the course group

- Input per measurement, e.g.on 8 nodes x 2 CPUs x 12 cores: Example

Column 1
- cart_method: 2
  - **1=Dims_create+Cart_create,**
  - **2=Cart_weighted_create (MPIX_WEIGHTS_EQUAL),**
  - **3=dito(weights),**
  - **4=dito manually,**
  - **5=Cart_ml_create(dims_ml),**
  - **0=end of input**

These base values (per process) are multiplied with $\sqrt[3]{\#processes}$ and then with 1, 2, 4, 8, … 512,
e.g., with 192 processes: **2** · $\sqrt[3]{192}$ · 512 = **5910**
(rounded to a multiple of the dim. of the process grid).
See also later the slide explaining the output.
Recommendation for several experiments: **Use the same initial mesh volume** (here 8), e.g., 1x2x4, 2x2x2, 4x2x1.
Note that this application data mesh volume is **completely independent** of the number of hardware nodes, CPUs, cores.

Columns 2-4
- Data mesh sizes, integer start values (= ratio) 0 0 = contiguous 1 2 4

Columns 5-10
- Using MPI_Type_vector, **for each dimension a pair of blocklength&stride** 0 0 0 0 0 0

Columns 11-13
- weights (double values) **(only with cart_method==4)** 1.00 0.50 0.25

Column 11
- number of hardware levels **(only with cart_method==5)** 3

  dims_ml: for each of the 3 Cartesian dimensions a list of 3 dimensions from outer to inner hardware level, e.g., 8 nodes x 2 CPUs x 12 cores are split into 1x2x4 nodes x 2x1x1 CPUs x 2x3x2 cores

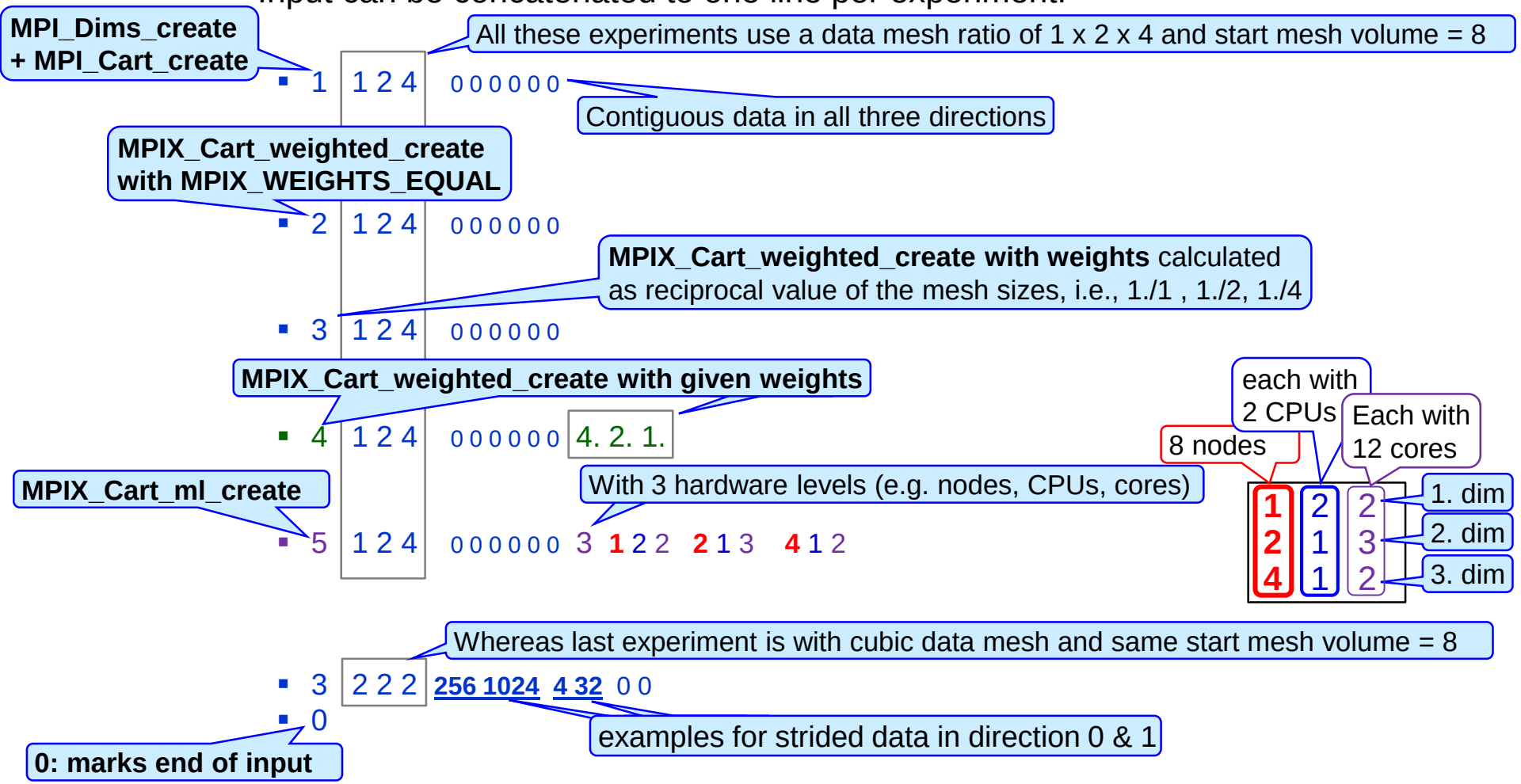
Columns 12-14
- **dims_ml[d=0] =** 1 2 2

Columns 15-17
- **dims_ml[d=1] =** 2 1 3

Columns 18-20
- **dims_ml[d=2] =** 4 1 2

Exercise 1a: Explanations, continued

- Input can be concatenated to one line per experiment:



Exercise 1a: Additional Remarks

- Caution with stdout and stdin when switching I/O from process world_rank==0 to cart_rank==0:
  - **Before** establishing the new comm_cart,
    all I/O on stdout/stdin is done by world_rank==0 (in **MPI_COMM_WORLD)**
  - **After** establishing the new comm_cart,
    all I/O on stdout/stdin is done by cart_rank==0 (in comm_cart)
  - In between, we recommended (although it is not guaranteed that an *output on comm_cart* may overtake an *output on MPI_COMM_WORLD*):
    - **MPI_Barrier(MPI_COMM_WORLD);**
    - **sleep(1);** // costs nearly nothing, e.g., 30 Mio € TCO/year / (365 days/year * 24 hours/day * 3600 sec/hour) * 1 sec = 1€
    - **MPI_Barrier(comm_cart);**
- The following slide shows the win through the re-ranking by the new routines:
  - Less % is better – the communication time reduction factors are:
    - **1.1-1.2**
    - **1.75**
    - **2.75**
    - **4.5-5.0**

Halosize/process ~= 26 MB (Depend on chosen dims)		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	8x2x12	84.545 = baseline	8 = 8 x 1 x 1 6 = 1 x 2 x 3 4 = 1 x 1 x 4	52.666 = 62%	8 = 2 x 2 x 2 6 = 2 x 1 x 3 4 = 2 x 1 x 2	48.556 = 57%	8 = 2 x 2 x 2 6 = 2 x 1 x 3 4 = 2 x 1 x 2
	64x2x12	194.856 = baseline	16=16 x 1 x 1 12= 4 x 1 x 3 8= 1 x 2 x 4	73.756 = 38%	16= 4 x 2 x 2 12= 4 x 1 x 3 8= 4 x 1 x 2	72.051 = 37%	16= 4 x 2 x 2 12= 4 x 1 x 3 8= 4 x 1 x 2
	512x2x12	247.631 = baseline	32=32 x 1 x 1 24=16 x 1x1.5 16= 1 x 2 x 8	85.530 = 35%	32= 8 x 2 x 2 24= 8 x 1 x 3 16= 8 x 1 x 2	85.491 = 35%	32= 8 x 2 x 2 24= 8 x 1 x 3 16= 8 x 1 x 2
1 x 2 x 4	8x2x12	172.850 = baseline	8 = 8 x 1 x 1 6 = 1 x 2 x 3 4 = 1 x 1 x 4	63.796 = 37%	8 = 2 x 2 x 2 6 = 2 x 1 x 3 4 = 2 x 1 x 2	37.953 = 22%	4 = 1 x 2 x 2 6 = 2 x 1 x 3 8 = 4 x 1 x 2
	64x2x12	360.364 = baseline	16=16 x 1 x 1 12= 4 x 1 x 3 8= 1 x 2 x 4	91.524 = 25%	16= 4 x 2 x 2 12= 4 x 1 x 3 8 = 4 x 1 x 2	74.199 = 21%	8 = 2 x 2 x 2 12= 4 x 1 x 3 16= 8 x 1 x 2
	512x2x12	457.858 = baseline	32=32 x 1 x 1 24=16 x 1x1.5 16= 1 x 2 x 8	125.468 = 27%	32= 8 x 2 x 2 24= 8 x 1 x 3 16= 8 x 1 x 2	93.615 = 20%	16= 4 x 2 x 2 24= 8 x 1 x 3 32=16 x 1 x 2
4 x 2 x 1	8x2x12	40.050 = baseline	8 = 8 x 1 x 1 6 = 1 x 2 x 3 4 = 1 x 1 x 4	59.421 =148%	8 = 2 x 2 x 2 6 = 2 x 1 x 3 4 = 2 x 1 x 2	36.778 =92%	16 = 4 x 2 x 2 6 = 2 x 1 x 3 2 = 1 x 1 x 2
	64x2x12	78.503 = baseline	16=16 x 1 x 1 12= 4 x 1 x 3 8= 1 x 2 x 4	100.203 =128%	16= 4 x 2 x 2 12= 4 x 1 x 3 8 = 4 x 1 x 2	69.802 =89%	32= 8 x 2 x 2 12= 4 x 1 x 3 4 = 2 x 1 x 2
	512x2x12	103.002 = baseline	32=32 x 1 x 1 24=16 x 1x1.5 16= 1 x 2 x 8	93.189 = 90%	32= 8 x 2 x 2 24= 8 x 1 x 3 16= 8 x 1 x 2	85.044 =83%	64=16 x 2 x 2 24= 8 x 1 x 3 8= 4 x 1 x 2

Used process dimensions

Internally applied dimen-sions on each hardware level

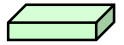
Base factors, not absolute values

By default, communica-tion time strongly depends on cuboid's direction

Optimized communi-cation times are nearly indepen-dent of cubid's form

On Cray XC40 Hazel hen at HLRS Stuttgart, Jan 2019

Exercise 1a: To do (1)

- **cp ~/MPI/course/C/Ch9/MPIX_*/* .**
  - You get the benchmark skeleton halo_irecv_send_toggle_3dim_grid_skel.c
  - And all MPIX_*.c files and the header MPIX_interface_proposal.h
- **mpicc –o halo.exe halo_irecv_send_toggle_3dim_grid_skel.c MPIX_*.c –lm**
- First test with non-optimized cart_method==1**, i.e., MPI_Dims_create + MPICart_create**
  - Choose your batch job: **halo_skel_[LRZ|VSC|HLRS].sh** which contains
    - **Number of nodes and cores/node**
    - **and, e.g.,**
      mpirun –np 192 ./halo.exe < input-skel.txt
  - Start your batchjob
  - Try to understand the output:
    - **It contains two experiments: a mesh with cubic**  **and one with non-cubic**  **ratio**
    - **The number of MPI processes, e.g. 192, is factorized**
      → domain decomposition into, e.g., 8 x 6 x 4 processes
    - **The measurements are done for 10 global meshsizes**
    - **The domain decomposition implies the local meshsizes**
    - **The local meshsizes imply the size of the halos in each direction**
    - **→ the sum of the time for the communication into the 3 dimensions x 2 directions (left+right)**

```
1 222 000000
1 124 000000
0
```

mpiicc in Vienna

lib math

See **next slide**

Exercise 1a: To do (2)

192 processes: Input for **MPI_Dims_create()**

These base values (per process) are multiplied with $\sqrt[3]{\#processes}$ and then with 1, 2, 4, 8, … 512, e.g., **2** · $\sqrt[3]{192}$ · 512 = **5910** (rounded to a multiple of the dimension of the process grid)

divide by dims[i]

First value for our table

* 2 directions * 4 byte

multiply for 2-d halo array size

Second value for our table

Same values, because MPI_Dims_create() factorizes the #processes independent from the user's meshsizes.

```
cart_method = 1
start mesh sizes integer start values for 3 dimensions = 2 2 2
blocklength & stride pairs for each of the 3 dimensions = 0 0   0 0   0 0
Creating the Cartesian communicator and further input arguments:
cart_method == 1: MPI_Dims_create + MPI_Cart_create
[MPI_Barrier and switching to output via stdout through rank==0 in comm_cart]
ndims=3 dims= 8 6 4
  message size    transfertime  duplex bandwidth per process and neighbor (mesh&halo in #floats)

                                          meshsizes total=   per process=   halosizes=
     128 bytes      34.537 usec   3.706 MB/s    16   12   12    2   2   3      16=    6 +    6 +    4
     432 bytes      39.840 usec  10.843 MB/s    24   24   24    3   4   6      54=   24 +   18 +   12
    1728 bytes      41.122 usec  42.021 MB/s    48   48   48    6   8  12     216=   96 +   72 +   48
    6688 bytes      23.961 use                  96   96   92   12  16  23     836=  368 +  276 +  192
   25576 bytes      93.703 use                 184  186  184   23  31  46    3197= 1426 + 1058 +  713
  104408 bytes     271.721 use                 376  372  372   47  62  93   13051= 5766 + 4371 + 2914
  411192 bytes    1033.001 usec    .056 MB/s   744  738  740   93 123 185   51399= 22755+17205 + 11439
 1636392 bytes    4398.680 use    372.019 MB/s 1480 1476 1476  185 246 369  204549= 90774+68265 + 45510
 6561336 bytes   18173.518 usec  361.038 MB/s 2960 2958 2956  370 493 739  820167=364327+273430+182410
26194104 bytes   76132.216 usec  344.061 MB/s 5912 5910 5908  739 985 1477 3274263=1454845+1091503+727915

cart_method = 1
start mesh sizes integer start values for 3 dimensions = 1 2 4
blocklength & stride pairs for each of the 3 dimensions = 0 0   0 0   0 0
cart_method == 1: MPI_Dims_create + MPI_Cart_create
ndims=3 dims= 8 6 4
  message size    transfertime  duplex bandwidth per process and neighbor (mesh&halo in #floats)
     160 byte       14.720 usec  10.870 MB/s    8   12   24    1   2   6      20=    12 +    6 +    2
       …
34936960 bytes  156869.278 usec  222.714 MB/s 2960 5910 11816  370 985 2954 4367120=2909690+1092980+364450
```

Exercise 1a: To do (3)

- **Fill in the table**

Given from MPI_Dims_create()

Nodes CPUs cores

d=0: **8** = **8** x 1 x 1
d=1: **6** = **1** x 2 x 3
d=2: **4** = **1** x 1 x 4

Total 192 = **8 x 2 x12**

Have to be calculated by hand:
Fill in maximal factors.
Factorize first the cores and start with d=2.
Then the CPUs & then the nodes.
(All based on sequential ranking of MPI_COMM_WORLD)

Defined in batch job + hardware knowledge

Execution time of **largest** mesh and halo size of both measurements

Given by batchjob and hardware

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPI_Cart_weighted_create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	__x__x__	= baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _				
1 x 2 x 4	Same as above	= baseline	Same as above				

Exercise 1a: To do (4)

- **cp halo_irecv_send_toggle_3dim_grid_skel.c halo_optim.c**
- **Edit halo_optim.c**
  - On lines 160, 165, and 171, substitute the **/* TODO: … */** by correct code

```
153  if (cart_method == 1) {
154    if (my_world_rank==0) printf("cart_method == 1: MPI_Dims_create + MPI_Cart_create\n");
155    MPI_Dims_create(size, ndims, dims);
156    MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, 0, &comm_cart);
157  } else if (cart_method == 2) {
158    if (my_world_rank==0) printf("cart_method == 2: MPIX_Cart_weighted_create( MPIX_WEIGHTS_EQUAL )\n");

160    /* TODO: Appropriate call to MPIX_Cart_weighted_create(...) with MPIX_WEIGHTS_EQUAL
161          instead of calling MPI_Dims_create() and MPI_Cart_create() as in method 1 */

163  } else if (cart_method == 3) {

165    /* TODO: Appropriate calculation of weights[ ] based on meshsize_avg_per_proc_startval[ ] */

167    if (my_world_rank==0) { printf("cart_method == 3: MPIX_Cart_weighted_create( weights := _____TODO_____)\n");
168      printf("weights= "); for (d=0; d<ndims; d++) printf(" %lf",weights[d]); printf("\n");
169    }

171    /* TODO: Appropriate call to MPIX_Cart_weighted_create(...) with weights
172          instead of MPIX_WEIGHTS_EQUAL as in method 2 */

174  } else …
```

Exercise 1a: To do (5)

mpiicc in Vienna

- **mpicc –o halo_optim.exe halo_optim.c MPIX_*.c**
- Check: **diff halo_optim.c halo_irecv_send_toggle_3dim_grid_solution.c**
- Now, use all three cart_method==1, 2, 3
  - Choose your batch job:
    - **halo_optim_[LRZ|VSC|HLRS].sh** which contains, e.g.:
    - **mpirun –np 192 ./halo_optim.exe < input-optim.txt**
  - Start your batchjob → output file **output_optim.txt**
  - Fill in the table

Note, that the optimization changes the dims-array → modified halo sizes!

Although halos may be larger, the optimized communication time should be shorter!

Base meshsizes

Block length + stride for each dimension

Cart_method

```
1  2 2 2  0 0 0 0 0 0
2  2 2 2  0 0 0 0 0 0
1  1 2 4  0 0 0 0 0 0
2  1 2 4  0 0 0 0 0 0
3  1 2 4  0 0 0 0 0 0
0
```

0 = end

cart_method==1 cart_method==2 cart_method==3

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat.	dims_ml[d=0] [d=1] [d=2]
2 x 2 x 2	__x__x__	_____ = baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _	_____ = ____% of baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _	Reported by MPI_Cart_weighted_create()	Same as with MPIX_WEIGHTS_EQUAL
1 x 2 x 4	__x__x__	_____ = baseline	Same as above	_____ = ____% of baseline	Same as above	_____ = ____% of baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _

Exercise 1a: Results – HLRS, Stuttgart, hazelhen

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_ create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	**8** x 2 x 12	**78.748** = baseline	8 = **8** x 1 x 1 6 = **1** x 2 x 3 4 = **1** x 1 x 4	**50.971** = **65%** of baseline	8 = **2** x 2 x 2 6 = **2** x 1 x 3 4 = **2** x 1 x 2	Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	**8** x 2 x 12	**168.891** = baseline	Same as above	**64.691** = **38%** of baseline	Same as above	**38.406** = **23%** of baseline	4 = **1** x 2 x 2 6 = **2** x 1 x 3 8 = **4** x 1 x 2

Exercise 1a: Results – LRZ, Garching, ivyMUC

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_ create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	**12** x 2 x 8	**34.814** = baseline	8 =**12**x 2/3 x 1 6 = **1** x 3 x 2 4 = **1** x 1 x 4	**26.675** = **77%** of baseline	6 = **3** x 1 x 2 8 = **2** x 2 x 2 4 = **2** x 1 x 2	Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	**12** x 2 x 8	**54.344** = baseline	Same as above	**35.665** = **66%** of baseline	Same as above	**22.933** = **42%** of baseline	4 = **1** x 2 x 2 6 = **3** x 1 x 2 8 = **4** x 1 x 2

Exercise 1a: Results – VSC, Vienna, vsc3

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_ create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	**12** x 2 x 8	**61.803** = baseline	8 = **12** x 2/3 x 1 6 = **1** x 3 x 2 4 = **1** x 1 x 4	**49.722** = **80%** of baseline	6 = **3** x 1 x 2 8 = **2** x 2 x 2 4 = **2** x 1 x 2	Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	**12** x 2 x 8	**97.658** = baseline	Same as above	**67.208** = **69%** of baseline	Same as above	**40.283** = **41%** of baseline	4 = **1** x 2 x 2 6 = **3** x 1 x 2 8 = **4** x 1 x 2

Exercise 1a: Your result: _____

Halosize/process ~= 26 MB		MPI_Dims_create + MPI_Cart_create		MPIX_Cart_weighted_ create(MPIX_WEIGHTS_EQUAL)		MPIX_Cart_weighted_ create(...weights...)	
Base mesh sizes	Nodes x CPUs x cores	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communication time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]	Communicat. time [ms]	dims_ml[d=0] dims_ml[d=1] dims_ml[d=2]
2 x 2 x 2	__x__x__	_____ = baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _	_____ = _____% of baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _	Same as with MPIX_WEIGHTS_EQUAL	
1 x 2 x 4	__x__x__	_____ = baseline	Same as above	_____ = _____% of baseline	Same as above	_____ = _____% of baseline	__ = _ x _ x _ __ = _ x _ x _ __ = _ x _ x _

skipped

The exercises from the Virtual Topology chapter are skipped within this MPI+X course

Exercise 1b — One-dimensional ring topology

- A: Rewrite the pass-around-the-ring program using a one-dimensional ring topology – or just
  B: compare the neighbor halo example with others

see end of Chap. 11-(1) "One-sided", after Exerci.1

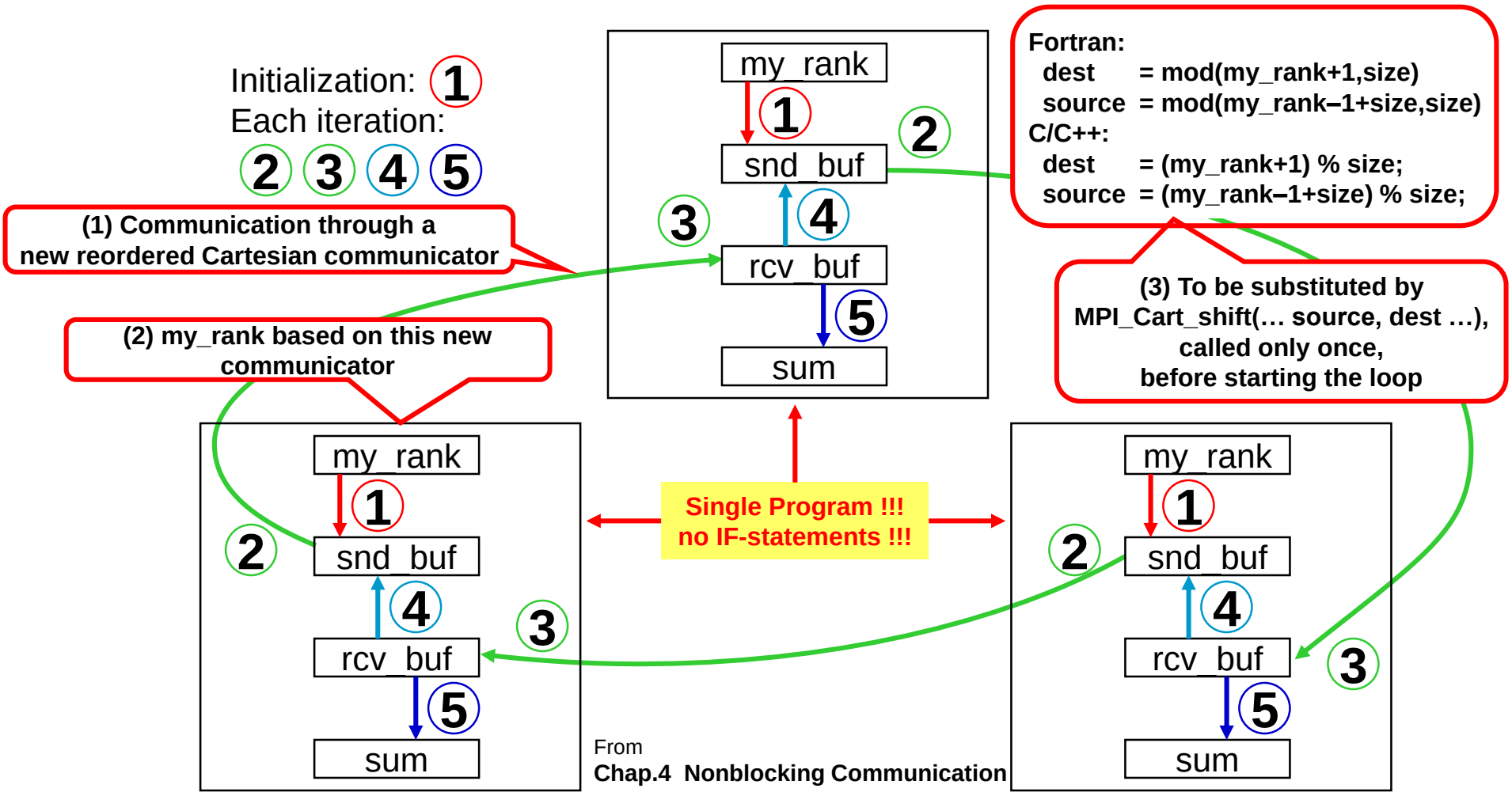
Fortran

C

- Use the results from course Chapter 4 *Nonblocking Communication*:
  ~/MPI/course/**F_30**/Ch4/ring_30.f90 (with mpi_f08 module)
  ~/MPI/course/**F_20**/Ch4/ring_20.f90 (with mpi module)
  ~/MPI/course/**C**/Ch4/ring.c
- Hints:
  - After calling MPI_Cart_create,
    - there should be no further usage of MPI_COMM_WORLD, and
    - the my_rank must be recomputed on the base of comm_cart.
  - the cryptic way to compute the neighbor ranks should be substituted by one call to MPI_Cart_shift, that should be before starting the loop.
  - Only **one**-dimensional:
    - → only `direction`=0
    - → In C: `dims` and `period` as normal variables, i.e., no arrays, but call by reference with &dims, …
    - → In Fortran: `dims` and `period` must be arrays (i.e., with only 1 element)
    - → coordinates are not necessary, because coord==rank

skipped

Slide from Chap. 4 — Rotating information around a ring



skipped

1st Advanced Exercise — Two-dimensional topology

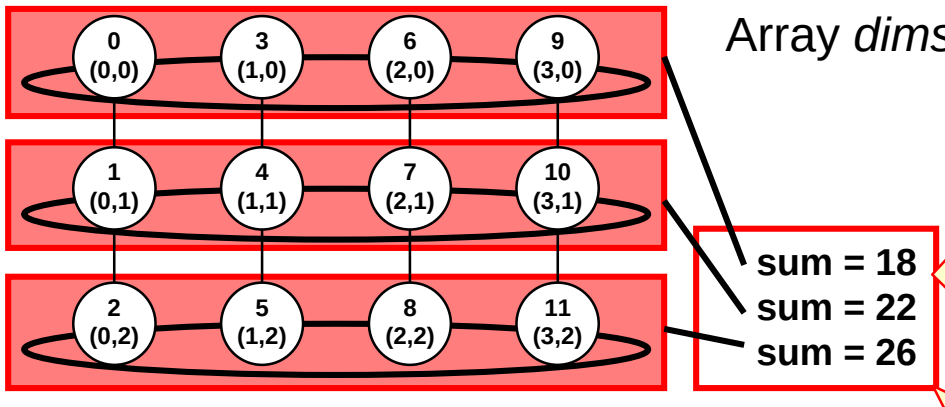
- Rewrite the exercise in two dimensions, as a cylinder.
- Each row of the cylinder, i.e. each ring, should compute its own separate sum of the original ranks in the two dimensional comm_cart.
- Compute the two dimensional factorization with MPI_Dims_create().

C

Fortran

- C/C++: int MPI_Dims_create(int nnodes, int ndims, int **dims*)
- Fortran: MPI_DIMS_CREATE(nnodes, ndims, *dims*, *IERROR*)

```
mpi_f08:        INTEGER                 :: nnodes, ndims, dims(*)
                INTEGER, OPTIONAL       :: ierror
mpi & mpif.h:   INTEGER nnodes, ndims, dims(*), ierror
```



Array *dims* must be **initialized** with **(0,0)**

Summing up the myrank of the 2-dimensional Cartesian topology:
Advanced Exercise 1a:
Ring-communication in the comm_slice, and using the ring with myrank, left, right and size of the comm_slice.

Additional Advanced Exercise 1b:
Using MPI_Allreduce within the comm_slice instead of the ring communication algorithm.
Solution, see 2nd Adv. Exe. in Chapter 6

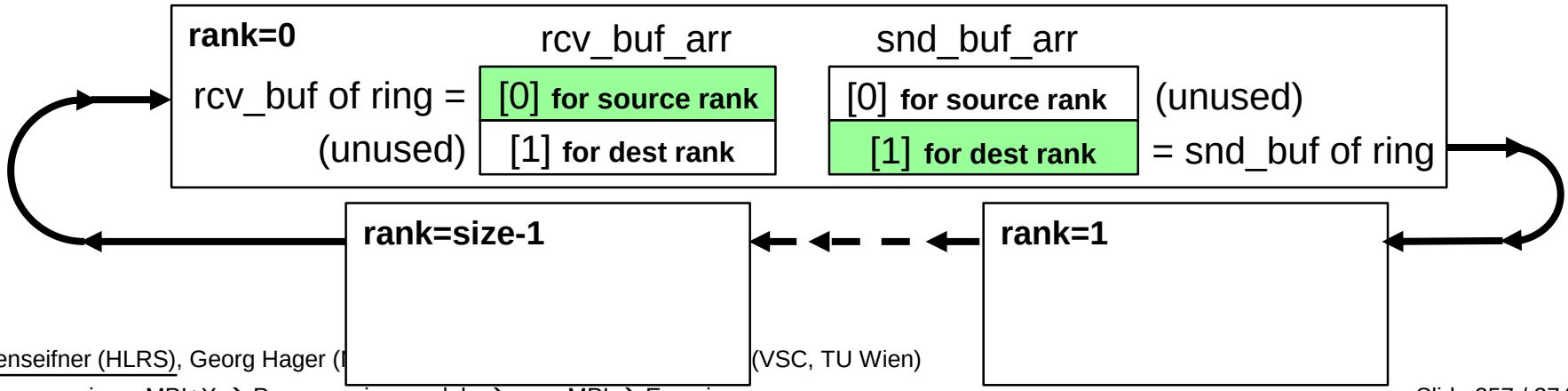
New in MPI-3.0

skipped

2nd Advanced Exercise — Neighbor Collective Communication

Keep the ring communication in the virtual topology example, but substitute the point-to-point communication by neighborhood collective:

- Use the results from course Chapter 9 *Virtual Topologies*:
  - ~/MPI/course/**F_30**/Ch9/topology_ring_30.f90 (with mpi_f08 module)
  - ~/MPI/course/**F_20**/Ch9/topology_ring_20.f90 (with mpi module)
  - ~/MPI/course/**C**/Ch9/topology_ring.c
- I.e., Isend-Recv-Wait → one call to MPI_Neighbor_alltoall
- rcv_buf and snd_buf must be combined into a buf_arr with rcv_buf_arr[0] as rcv_buf and snd_buf_arr[1] as snd_buf, i.e., according to the sequence rule for the buffer segments.



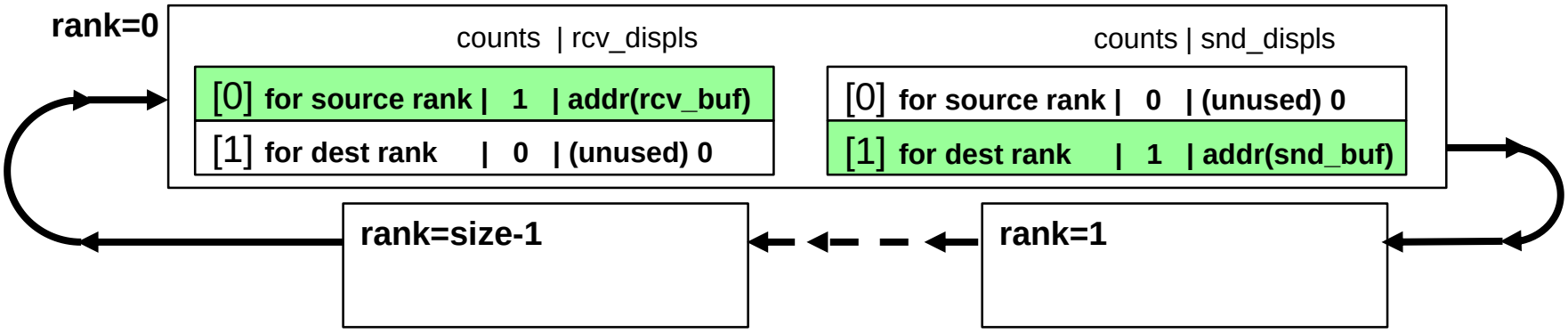
New in MPI-3.0

skipped

3rd Advanced Exercise — Neighbor Collective Communication & MPI_BOTTOM

Start again from the virtual topology example, but substitute the point-to-point communication by MPI_NEIGHBOR_ALLTOALLW with MPI_BOTTOM and absolute addresses of rcv_buf and snd_buf:

- Use the results from course Chapter 9 *Virtual Topologies*:
  - ~/MPI/course/**F_30**/Ch9/topology_ring_30.f90 (with mpi_f08 module)
  - ~/MPI/course/**F_20**/Ch9/topology_ring_20.f90 (with mpi module)
  - ~/MPI/course/**C**/Ch9/topology_ring.c
- I.e., Isend-Recv-Wait → one call to MPI_Neighbor_alltoallw
- Fortran: Do not forget to call MPI_F_SYNC_REG for the real variables behind MPI_BOTTOM (i.e., snd_buf, rcv_buf) **before & after** the communication call!



CAUTION: Officially, this example is not portable, because address differences are allowed only inside of structures or arrays, i.e., snd_buf and rcv_buf need to be part of a common space → MPI-3.1, 4.1.12

Programming models
- pure MPI

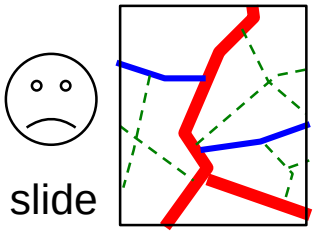
The Topology Problem: Wrap Up

Virtual MPI Topologies

- With large ccNUMA nodes and many ccNUMA nodes
- The new optimizing routines are easily to use for Cartesian problems
- Be aware that the MPI_cart_..._create routines renumber the communicator with other ranks!
- **And what to do with unstructured grids?**
- See paper from Torsten Höfler and references in Bill Gropp's paper:
  - T. Hoefler and M. Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*. ACM, 75–85.
  - Bill Gropp. 2018. Using Node Information to Implement MPI Cartesian Topologies. In *Proceedings of the 25nd European MPI Users' Group Meeting (EuroMPI '18)*, September 23–26, 2018, Barcelona, Spain. ACM, New York, NY, USA, 9 pages.
- The MPI libraries are still not optimizing the graph topologies, but this should come.
- Additional application problem:
  Your application may read data in before creating the virtual graph topology.
  → The re-numbering of the processes may require that you
    - send such data to the new process (with the old rank), or
    - need to re-read such data from disk.

How to achieve a hierarchical DD for unstructured grids?

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

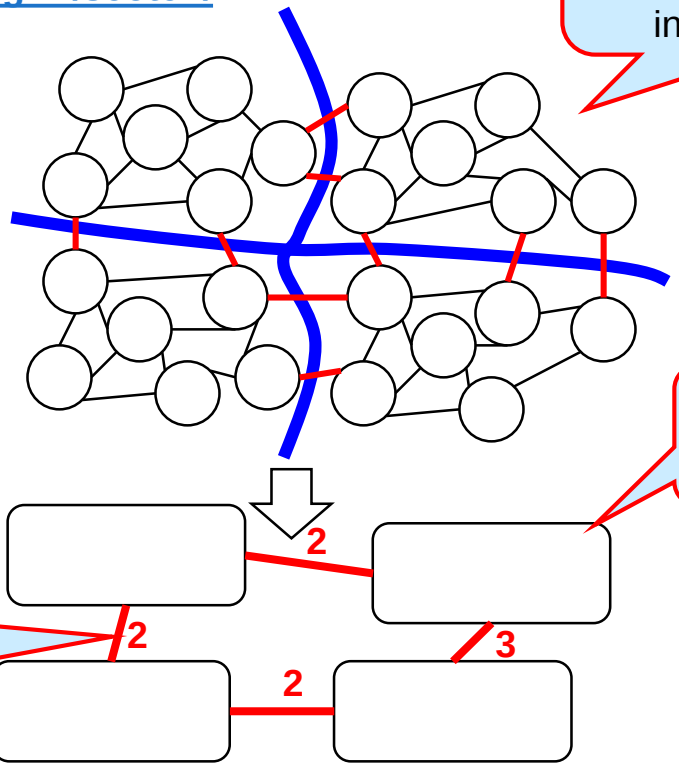


    - Bottom-up: Low level DD
      + higher level recombination
      → based on DD of the grid of subdomains

Unstructured Grid / Data Mesh

- Mesh partitioning with special load balancing libraries
  - Metis (George Karypis, University of Minnesota)
  - ParMetis (internally parallel version of Metis)
    - http://glaros.dtc.umn.edu/gkhome/views/metis/metis.html
  - Scotch & PT-Scotch (Francois Pellegrini, LaBRI, France)
    - https://www.labri.fr/perso/pelegrin/scotch/
  - Goals:
    - **Same work load in each sub-domain**
    - **Minimizing the maximal number of neighbor-connections between sub-domains**
    - **Minimizing the total number of neighbor sub-domains of each sub-domain**

Result of (Par)Metis or (PT-)Scotch: Sort out all mesh points into sub-domains



Each sub-domain is stored on one MPI process

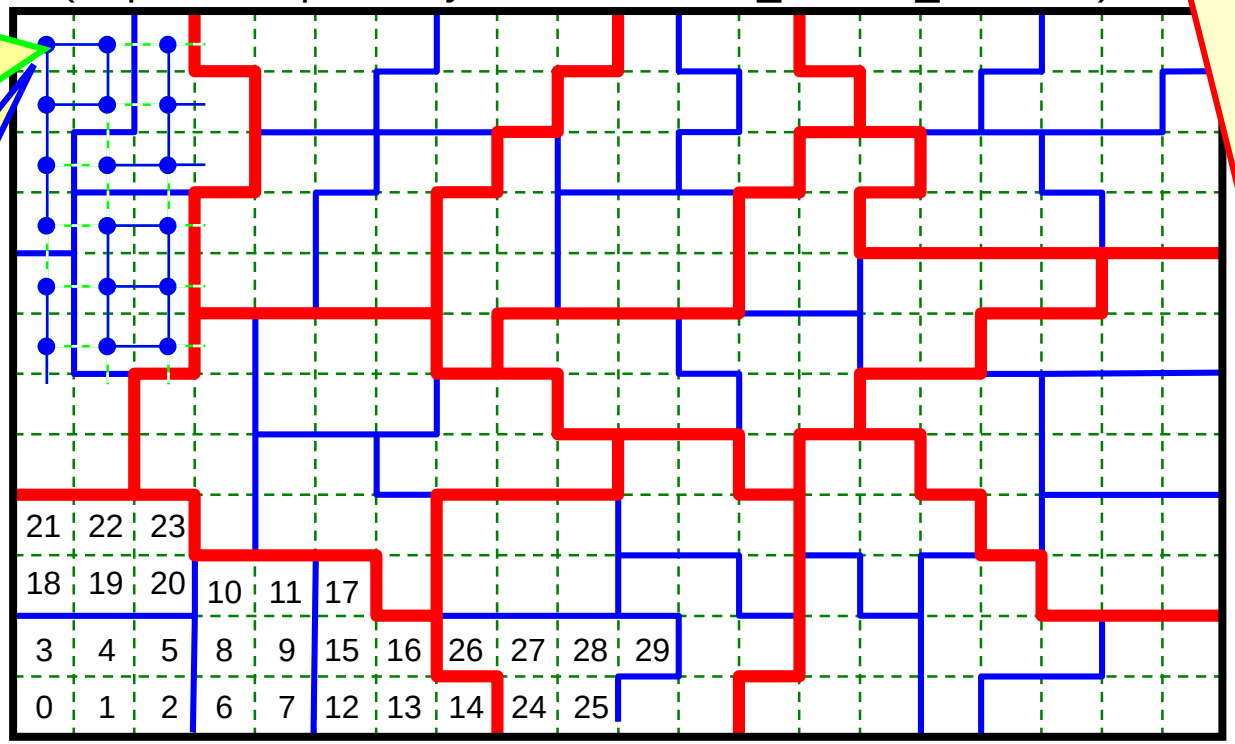
The weighted communication graph of the virtual process grid can be used as input for MPI_Dist_graph_create(_adjacent)

Bottom-up – Multi-level DD through recombination

1. **Core-level DD:** partitioning of (large) application's data grid, e.g., } with Metis / Scotch
2. **NUMA-domain-level DD:** recombining of core-domains }
3. **SMP node level DD:** recombining of socket-domains
4. **Numbering** from core to socket to node (requires sequentially numbered MPI_COMM_WORLD)

Graph of all sub-domains (core-sized)

Divided into sub-graphs for each socket



- **Problem:** Recombination must **not** calculate patches that are smaller or larger than the average
- In this example the load-balancer **must** combine always
  - 6 cores, and
  - 4 NUMA domains (i.e., sockets or dies)
- **Advantage:** Communication is balanced!

skipped

Profiling solution

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

- Example: CrayPat and CrayApprentice

As far as we know, such methods exist only on Cray systems

Quiz on Virtual topologies

A. Which types of MPI topologies for virtual process grids exist?

B. And for which use cases?

1. _____

For _____

2. _____

For _____

C. Where are limits for using virtual topologies, i.e., which use cases do not really fit?

_____

Programming models - pure MPI

Scalability

To overcome MPI scaling problems

- MPI has a few scaling problems when, 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)
    - **Use MPI-2.2 MPI_Dist_graph_create_adjacent**
  - Creation of disjunct sub-communicators with MPI_Comm_create
    - **MPI-2.2 introduces a new scaling meaning of MPI_Comm_create**
  - MPI internal memory consumption for, e.g.,
    - **Internal data structure for large communicators**
    - **Internal communication buffers**
  - … see also P. Balaji, et al.: **MPI on a Million Processors.**
    P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traff: MPI on Millions of Cores. Parallel Processing Letters, 21(01):45-60, 2011.
    Originally, Proceedings EuroPVM/MPI 2009.
- Hybrid programming reduces all these problems (due to a smaller number of processes)

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

Programming models
- pure MPI

Advantages & disadvantages, conclusions

Pure MPI communication: Main advantages

- Simplest programming model
- Library calls need not to be thread-safe
- The hardware is typically prepared for many MPI processes per SMP node
- Only minor problems if pinning is not applied
- No first-touch problems as with OpenMP
  (in hybrid MPI+OpenMP)

Pure MPI communication: Main disadvantages

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

Pure MPI communication: Conclusions

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

Tools

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

Tools for Thread/Process Affinity (“Pinning”)

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

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

- OpenMP 4.0 thread/core/socket binding

- Vendor-specific solutions
  - Intel, IBM, Cray, GCC, OpenMPI,…

Tools

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

Thread Correctness – Intel Inspector 1/3

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

With new Intel Inspector XE 2015: Command line interface must be used within mpirun / mpiexec

```
Application finished
_____________________________________________________________________________
|ID|Short De|Sever|C|Contex|Description                          |1st Acc|2nd Acc|
|  |scriptio|ity  |o|t[Best|                                     |ess[Bes|ess[Bes|
|  |n       |Name |u|]     |                                     |t]     |t]     |
|  |        |     |n|      |                                     |       |       |
|  |        |     |t|      |                                     |       |       |
_____________________________________________________________________________
|1 |Write ->|Error|1|"pthre|Memory write of global_variable at|"pthrea|"pthrea|
|  |Write da|     | |ad_rac|"pthread_race.c":31 conflicts with|d_race.|d_race.|
|  |ta-race |     | |e.c":2|a prior memory write of           |c":31  |c":31  |
|  |        |     | |5     |global_variable at                |       |       |
|  |        |     | |      |"pthread_race.c":31 (output       |       |       |
|  |        |     | |      |dependence)                       |       |       |
_____________________________________________________________________________
```

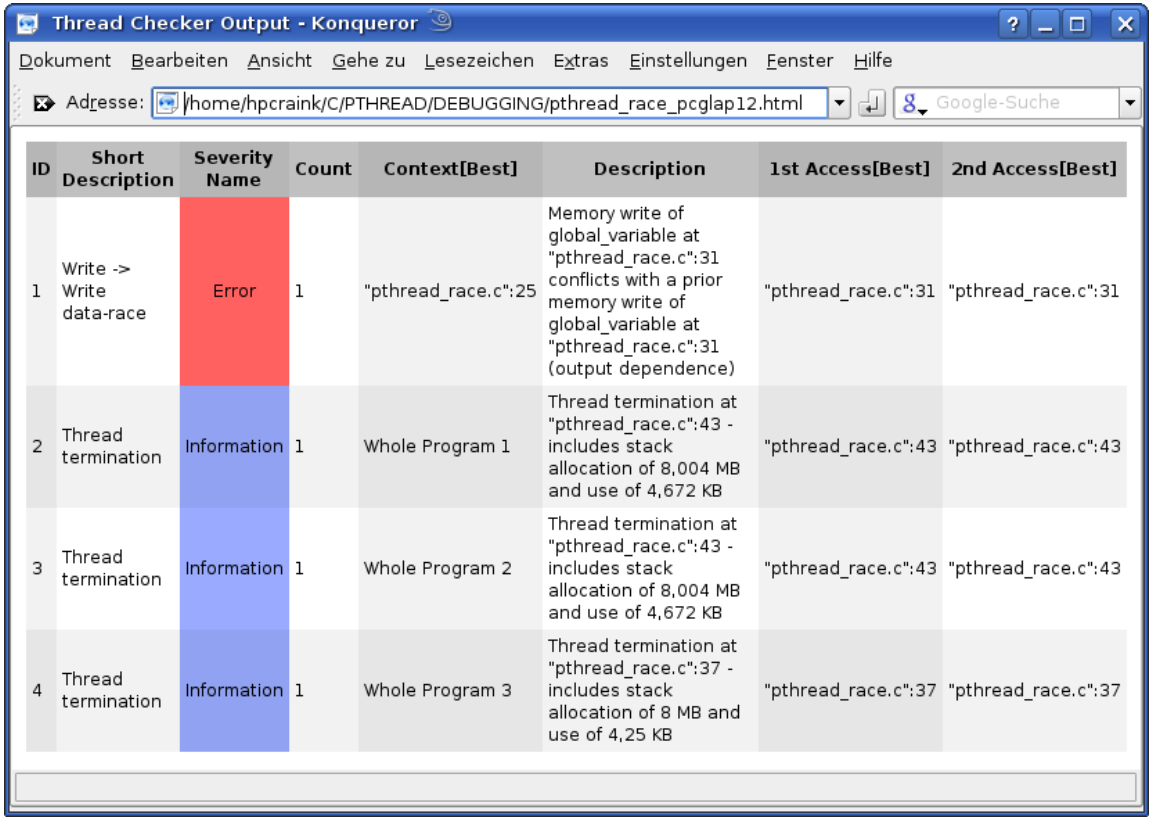
Courtesy of Rainer Keller
HLRS, ORNL and FhT

skipped

Thread Correctness – Intel Inspector 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

Courtesy of Rainer Keller
HLRS, ORNL and FhT

skipped

Thread Correctness – Intel Inspector 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'
```

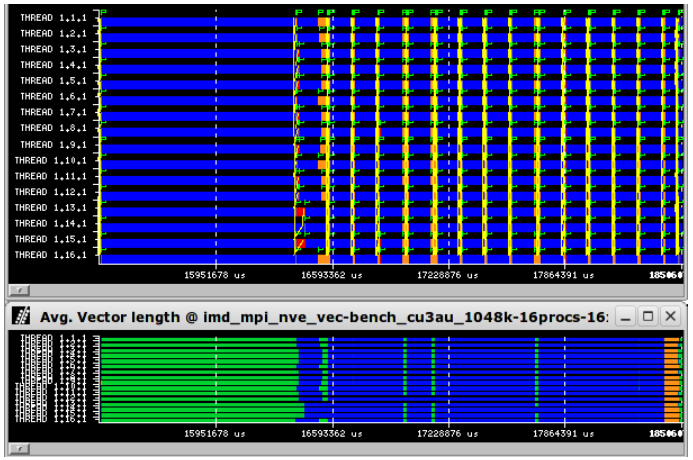
- Then run with:

```
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 …
```

Courtesy of Rainer Keller
HLRS, ORNL and FhT

Performance Tools Support for Hybrid Code

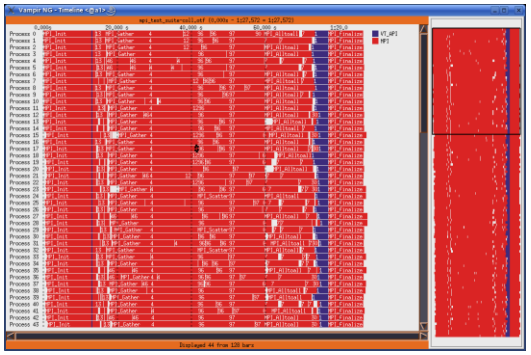
- Paraver examples have already been shown, tracing is done with linking against (closed-source) `omptrace` or `ompitrace`



- For Vampir/Vampirtrace performance analysis:

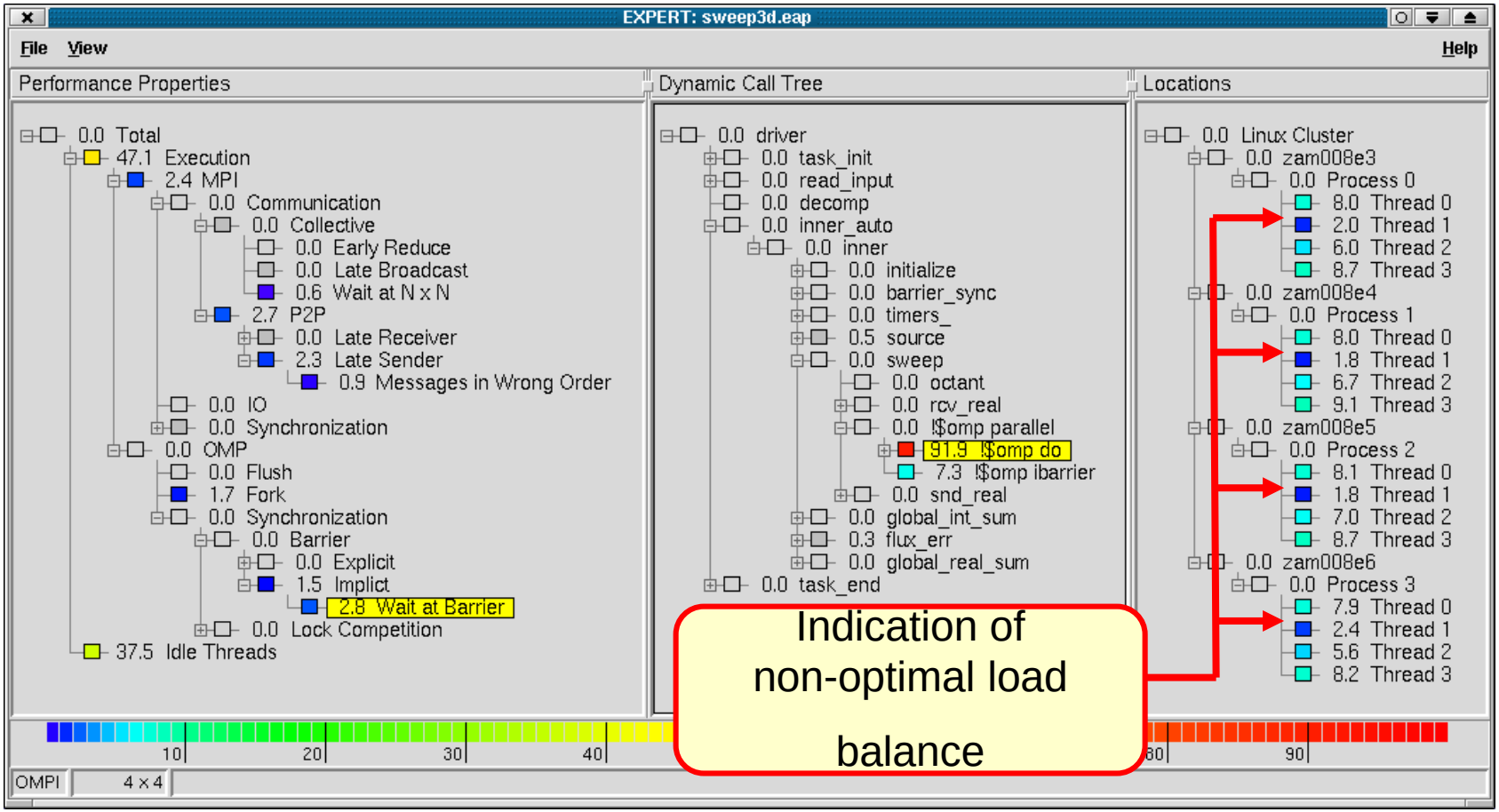
```
./configure -enable-omp
  -enable-hyb
  -with-mpi-dir=/opt/OpenMPI/1.3-icc
CC=icc F77=ifort FC=ifort
```

(Attention: does not wrap `MPI_Init_thread`!)



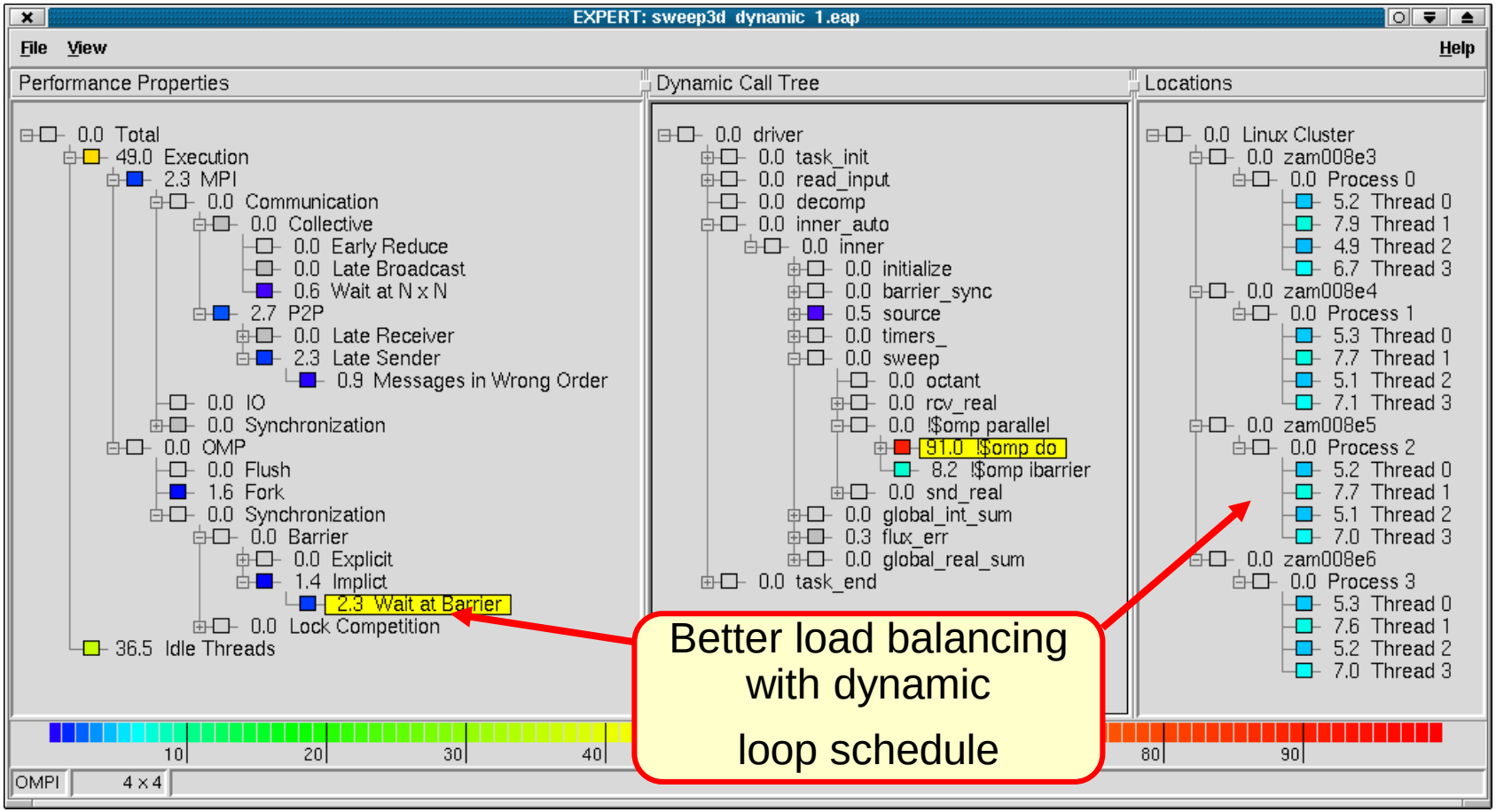
Courtesy of Rainer Keller
HLRS, ORNL and FhT

Scalasca – Example “Wait at Barrier”



Screenshots, courtesy of KOJAK JSC, FZ Jülich

Scalasca – Example “Wait at Barrier”, Solution



Screenshots, courtesy of KOJAK JSC, FZ Jülich

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

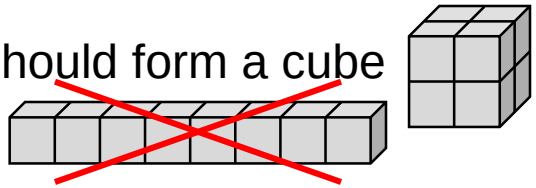
MPI+OpenMP versus MPI+MPI-3.0 shared mem.

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**

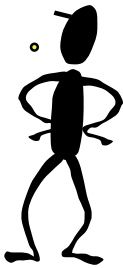
Acknowledgements

- We want to thank
  - Gabriele Jost, Supersmith, Maximum Performance Software, USA
    - **Co-author of several slides and previous tutorials**
  - Irene Reichl, VSC, TU Wien, Vienna, Austria
    - **Co-author of the date-rep exercise**
  - 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
  - Christoph Niethammer, HLRS
  - Jesper Träff, TU Wien, Faculty of Informatics
  - Bill Gropp, National Center for Supercomputing Applications, University of Illinois Urbana-Champaign

Conclusions

- Future hardware will be more complicated
  - Heterogeneous → GPU, FPGA, …
  - Node-level ccNUMA is here to stay, but will only be one of your problems
  - ….
- High-end programming → more complex → many pitfalls
- Medium number of cores → more simple
  (**#cores / SMP-node** will not shrink but rather grow)
- **MPI + OpenMP → workhorse on large systems**
  - Major pros: **reduced memory needs** and **second level of parallelism**
- **MPI + MPI-3 → only for special cases and medium rank number**
- Pure MPI communication → still viable if it does the job
- OpenMP only → on large ccNUMA nodes (almost gone in HPC)

Thank you for your interest



Q & A

Please fill out the feedback sheet – Thank you

Appendix

- Abstract
- Authors
- Solutions

Abstract

MPI+X – Introduction to Hybrid Programming in HPC

Tutorial (Content levels: 0:00h [=0%] Beginners, 1:30h [=10%] Intermediate, 13:30h [=90%] Advanced)

Authors: Claudia Blaas-Schenner, VSC Research Center, TU Wien, Vienna, Austria
Georg Hager, Erlangen Regional Computing Center (RRZE), University of Erlangen, Germany
Rolf Rabenseifner, High Performance Computing Center (HLRS), University of Stuttgart, Germany

Abstract: Most HPC systems are clusters of shared memory nodes. To use such systems efficiently both memory consumption and communication time has to be optimized. Therefore, hybrid 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 course 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 has 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. Numerous case studies and micro-benchmarks demonstrate the performance-related aspects of hybrid programming.

Hands-on sessions are included on all days. Tools for hybrid programming such as thread/process placement support and performance analysis are presented in a "how-to" section. This course provides scientific training in Computational Science and, in addition, the scientific exchange of the participants among themselves.

URL:
2022-HY-VSC http://vsc.ac.at/training/2022/HY-VSC 2022-HY-LRZ http://www.hlrs.de/training/2022/HY-LRZ
2021-HY-VSC http://vsc.ac.at/training/2021/HY-VSC
2020-HY-VSC http://vsc.ac.at/training/2020/HY-VSC 2020-HY-S http://www.hlrs.de/training/2020/HY-S
2019-HY-G https://www.lrz.de/services/compute/courses/archive/2019/2019-01-28_hhyp1w18/
ISC 2017 https://www.isc-hpc.com/agenda2017/sessiondetails23ac.html?t=session&o=510

Claudia Blaas-Schenner

Claudia Blaas-Schenner holds a diploma (1992) and PhD (1996) in Technical Physics from TU Wien (Vienna, Austria). As a postdoc and later as a research fellow she continued to work in computational materials science, both at TU Wien and at the University of Vienna, with research stays at TU Dresden (Germany) and at the Academy of Sciences of the Czech Republic in Prague (Czech Republic). In 2014 she joined the VSC Research Center at TU Wien (Vienna, Austria), where she is responsible for developing a training and education program in HPC. She develops the curriculum of the training courses and teaches mainly parallel programming with MPI and OpenMP as well as hybrid programming techniques MPI+X. In addition, she is involved in performance optimization of user codes. Claudia is an active member of the MPI Forum, which is the standardization body for the Message Passing Interface (MPI; https://www.mpi-forum.org), and is acting as a chapter committee chair for "MPI Terms and Conventions“, which is essential for the MPI standard as a whole (https://www.mpi-forum.org/mpi-41). For PRACE-6IP she is the project manager at TU Wien, leads the recently established PRACE Training Center (PTC) at the VSC Research Center of TU Wien, and acts as the Management Board representative of Austria in PRACE-6IP.

Georg Hager

Georg Hager holds a PhD and a Habilitation degree in Computational Physics from the University of Greifswald. He leads the Training & Support Division at Erlangen National High Performance Computing Center (NHR@FAU) and is an associate lecturer at the Institute of Physics at the University of Greifswald. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes on chip and system levels, and the analytic modeling of structure formation in large-scale parallel codes. Georg Hager has authored and co-authored more than 100 peer-reviewed publications and was instrumental in developing and refining the Execution-Cache-Memory (ECM) performance model and energy consumption models for multicore processors. In 2018, he won the “ISC Gauss Award” (together with Johannes Hofmann and Dietmar Fey) for a paper on accurate analytic performance and power modeling. He received the “2011 Informatics Europe Curriculum Best Practices Award” (together with Jan Treibig and Gerhard Wellein) for outstanding contributions to teaching in computer science. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended or required reading in many HPC-related lectures and courses worldwide. Together with colleagues from FAU, HLRS Stuttgart, and TU Wien he develops and conducts successful international tutorials on node-level performance engineering and hybrid programming. A full list of publications, talks, and other things he is interested in can be found in his blog: https://blogs.fau.de/hager.

Rolf Rabenseifner

Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he is working 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 vendors' 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 is a member of the MPI-2 Forum and since Dec. 2007, he is in the steering committee of the MPI-3 Forum. Rolf was responsible for the MPI-2.1 version of the standard. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at the TU Dresden. 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 at many universities and labs in Germany, and also in Austria and Switzerland. In January 2012, the Gauss Centre of Supercomputing (GCS), with HLRS, LRZ in Garching and the Jülich Supercomputing Center as members, was selected as one of six PRACE Advanced Training Centers (PATCs). Rolf Rabenseifner was appointed as the GCS' PATC director.

Solutions of MPI shared memory exercise: datarep

- Solution files:
  - data-rep_sol_2a.c
  - data-rep_sol_2d.c
  - data-rep_sol_2f.c
  - data-rep_sol_3-6.c
  - data-rep_sol_7.c
  - data-rep_solution.c
- Quiz solution

Solutions of MPI shared memory exercise: datarep
(a 1-slide-solution-summary)

MPI/tasks/C/Ch11/data-rep/data-rep_solution.c

C

```
arr = (arrType *) malloc(arrSize * sizeof(arrType));
```
grey = original code

```
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, /*key=*/ 0,
                    MPI_INFO_NULL, &comm_shm);
MPI_Comm_size(comm_shm, &size_shm);
MPI_Comm_rank(comm_shm, &rank_shm);
if ( rank_shm == 0 ) { individualShmSize = arrSize ; }
else                 { individualShmSize = 0 ; }
MPI_Win_allocate_shared(
  (MPI_Aint)(individualShmSize) * (MPI_Aint)(sizeof(arrType)),
  sizeof(arrType), MPI_INFO_NULL, comm_shm, &shm_buf_ptr, &win );
MPI_Win_shared_query( win, 0, &arrSize_, &disp_unit, &arr );
```

process is head of one of the shared memory islands

```
color=MPI_UNDEFINED ;  if (rank_shm==0) { color = 0; }
MPI_Comm_split(MPI_COMM_WORLD, color, /*key=*/ 0, &comm_head);
if( comm_head != MPI_COMM_NULL )
{MPI_Comm_size(comm_head, &size_head);MPI_Comm_rank(comm_head, &rank_head);}
MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
```
Starting write epoch

```
if(rank_world==0) for( i=0; i<arrSize; i++) arr[i]=i+it;
```
Filling arr

```
if( comm_head != MPI_COMM_NULL ) {
   MPI_Bcast(arr, arrSize, arrDataType, 0, comm_head);
}
```
Only the heads of the shared memory islands fill arr by …

… broadcasting to all heads

instead of MPI_COMM_WORLD

```
MPI_Win_fence(/*workaround:no assertions:*/0,win);
```
Starting read epoch by all proc's

```
sum=0; for( i=0; i<arrSize; i++) sum+= arr[i];
```
Reading arr

The following slides show a step-by-step solving of this exercise

Solutions of MPI shared memory exercise: datarep

data-rep_base.c

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

typedef long arrType ;
#define arrDataType MPI_LONG /* !!!!!   C A U T I O N :   MPI_Type must fit to arrType       !!!!! */
static const int arrSize=16*1.6E7 ;

int main (int argc, char *argv[])
{
  int it ;
  int rank_world, size_world;
  arrType *arr ;
  int i;
  long long sum ;

/* ===> 1 <=== */

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank_world);
  MPI_Comm_size(MPI_COMM_WORLD, &size_world);

/* ===> 2 <=== */
    arr = (arrType *) malloc(arrSize * sizeof(arrType));
    if(arr == NULL)
    {   printf("arr NOT allocated, not enough memory\n");
        MPI_Abort(MPI_COMM_WORLD, 0);
    }
...
```

During the exercise steps, you may add additional declarations

In each process, allocating an array for the replicated
TODO: Allocating only once per shared memory node!
This will be done in 3 steps: 2a, 2b-d, 2e-f

Solutions of MPI shared memory exercise: datarep

data-rep_base.c (continued)

```
...
/* ===> 3 <=== */
 for( it = 0; it < 3; it++)
 {
 /* only rank_world=0 initializes the array arr */
   if( rank_world == 0 )
   {
     for( i = 0; i < arrSize; i++)
     { arr[i] = i + it ; }
   }
/* ===> 4 <=== */
   MPI_Bcast( arr, arrSize, arrDataType, 0, MPI_COMM_WORLD );

/* Now, all arrays are filled with the same content. */

/* ===> 5 <=== */
   sum = 0;
   for( i = 0; i < arrSize; i++)
   {
     sum+= arr [ i ] ;
   }

/* ===> 6 <=== */
  /*TEST*/ // To minimize the output, we print only from 3 process per SMP node
  /*TEST*/ if ( rank_world == 0 || rank_world == 1 || rank_world == size_world - 1 )
      printf ("it: %i, rank ( world: %i/%i ):\tsum(i=%i...i=%i) = %lld \n",
                  it, rank_world, size_world, it, arrSize-1+it, sum );
 }

/* ===> 7 <=== */
  free(arr);
  MPI_Finalize();
}
```

Time step loop

Steps (3)-(6) are done together

Filling the array by one process.
Will be unchanged.

Broadcasting it to all other processes.
TODO: Only one process per SMP node should broadcast!

Calculating some numerical result in each process. Same result on each process that it is easy to verify.
Will be unchanged.

And printing it out
Will be unchanged.

Freeing the allocated array.
TODO: We must free the window instead.

Last step!

Solutions of MPI shared memory exercise: datarep

data-rep_sol_2a.c

```
…
  MPI_Comm comm_shm;
  int size_shm, rank_shm;
…

/* ===> 2 <=== */
  /* Create --> shared memory islands and --> shared memory window inside */
  /*           -->    comm_shm         and      -->    win                  */

  MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, /*key=*/ 0, MPI_INFO_NULL, &comm_shm);
  MPI_Comm_size(comm_shm, &size_shm);
  MPI_Comm_rank(comm_shm, &rank_shm);
  /*TEST*/ // To minimize the output, we print only from 3 process per SMP node
  /*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
      printf("\t\trank ( world: %i/%i, shm: %i/%i)\n", rank_world, size_world, rank_shm, size_shm);
  /*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;

/*  TO DO:
 *  substitute the following malloc
 */
…
```

Solutions of MPI shared memory exercise: datarep

C

data-rep_sol_2d.c

```
…
  MPI_Win  win;
  int individualShmSize ;
  arrType *shm_buf_ptr;

  /* output MPI_Win_shared_query */
  MPI_Aint arrSize_ ;
  int disp_unit ;
…
/* ===> 2 <=== */
…
 /* instead of:  arr = (arrType *) malloc(arrSize * sizeof(arrType)); */
  if ( rank_shm == 0 )
  { individualShmSize = arrSize ; }
  else
  { individualShmSize = 0 ; }
  MPI_Win_allocate_shared( (MPI_Aint)(individualShmSize) * (MPI_Aint)(sizeof(arrType)),
                           sizeof(arrType), MPI_INFO_NULL, comm_shm, &shm_buf_ptr, &win );
  /* shm_buf_ptr is not used because it is only available in process rank_shm==0 */
  MPI_Win_shared_query( win, 0, &arrSize_, &disp_unit, &arr );

  /*TEST*/ // To minimize the output, we print only from 3 process per SMP node
  /*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
    printf("\t\trank ( world: %i/%i, shm: %i/%i) arrSize %i arrSize_ %i shm_buf_ptr=%p arr_ptr=%p \n",
            rank_world, size_world, rank_shm, size_shm, arrSize, (int) (arrSize_), shm_buf_ptr, arr );
  /*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;

/* TO DO: Create communicator comm_head with MPI_Comm_split  -->  including all the rank_shm == 0
processes.
…
```

data-rep_sol_2d_30.f90

Fortran

```
…
!  INTEGER*8, DIMENSION(:), ALLOCATABLE :: arr
   INTEGER*8, DIMENSION(:), POINTER :: arr
…
/* ===> 1 <=== */
…
/* ===> 2 <=== */
…
```

similar to C

→ See next slide

back

Solutions of MPI shared memory exercise: datarep

Fortran

data-rep_sol_2d_f90.c

```
!  INTEGER*8, DIMENSION(:), ALLOCATABLE :: arr
  INTEGER*8, DIMENSION(:), POINTER :: arr
/* ===> 1 <=== */
 TYPE(MPI_Win) :: win
  INTEGER :: individualShmSize
  TYPE(C_PTR) :: arr_ptr, shm_buf_ptr
  INTEGER(KIND=MPI_ADDRESS_KIND) :: arrDataTypeSize, lb, ShmByteSize
! /* output MPI_Win_shared_query */
  INTEGER(kind=MPI_ADDRESS_KIND) :: arrSize_
  INTEGER :: disp_unit
/* ===> 2 <=== */
 ! instead of:  ALLOCATE(arr(1:arrSize))
  IF ( rank_shm == 0 ) THEN
    individualShmSize = arrSize
  ELSE
    individualShmSize = 0
  ENDIF
  CALL MPI_Type_get_extent(arrDataType, lb, arrDataTypeSize)
  ShmByteSize = individualShmSize * arrDataTypeSize
  disp_unit = arrDataTypeSize
  CALL MPI_Win_allocate_shared( ShmByteSize, disp_unit, MPI_INFO_NULL, comm_shm, shm_buf_ptr, win )
  !  /* shm_buf_ptr is not used because it is only available in process rank_shm==0 */
  CALL MPI_Win_shared_query( win, 0, arrSize_, disp_unit, arr_ptr )
  CALL C_F_POINTER(arr_ptr, arr, (/arrSize/) )
  ! TEST: To minimize the output, we print only from 3 process per SMP node
  IF ( (rank_shm == 0) .OR. (rank_shm == 1) .OR. (rank_shm == size_shm - 1) ) THEN
    WRITE(*,*) 'rank( world=',rank_world,' shm=',rank_shm,')',' arrSize=',arrSize,' arrSize_=',arrSize_
  ENDIF
  IF (rank_world == 0) WRITE(*,*) 'ALL finalize and return!!!'; CALL MPI_Finalize(); STOP
...
```

Solutions of MPI shared memory exercise: datarep

data-rep_sol_2f.c

```
…
  int color ;
  MPI_Comm comm_head;
  int size_head, rank_head;
…
/* ===> 2 <=== */
…
  /* Create communicator including all the rank_shm = 0               */
  /* with the MPI_Comm_split: in color 0 all the rank_shm = 0 ,
   * all other ranks are color = 1                                    */

  color=MPI_UNDEFINED ;
  if (rank_shm==0) color = 0 ;

  MPI_Comm_split(MPI_COMM_WORLD, color, /*key=*/ 0, &comm_head);
  rank_head = -1; // only used in the print statements to differentiate unused rank==-1 from used rank==0
  if( comm_head != MPI_COMM_NULL ) // if( color == 0 ) // rank is element of comm_head, i.e., it is head
of one of the islands in comm_shm
  {
    MPI_Comm_size(comm_head, &size_head);
    MPI_Comm_rank(comm_head, &rank_head);
  }

  /*TEST*/ // To minimize the output, we print only from 3 process per SMP node
  /*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
      printf("\t\trank ( world: %i/%i, shm: %i/%i, head: %i/%i) arrSize %i arrSize_ %i  shm_buf_ptr = %p,
arr_ptr = %p \n",
 rank_world,size_world,rank_shm,size_shm,rank_head,size_head, arrSize, (int)(arrSize_), shm_buf_ptr, arr);
  /*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;…
```

back

Solutions of MPI shared memory exercise: datarep

data-rep_sol_3-6.c (on this slide steps 3-4)

```
…
  /* ===> 3 <=== */
 for( it = 0; it < 3; it++)
 {
 /* only rank_world=0 initializes the array arr                    */
 /* all rank_shm=0 start the write epoch: writing arr to their shm */
   MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
   if( rank_world == 0 ) /* from those rank_shm=0 processes, only rank_world==0 fills arr */
   {
     for( i = 0; i < arrSize; i++)
     { arr[i] = i + it ; }
   }

/* ===> 4 <=== */
 /* Instead of all processes in MPI_COMM_WORLD, now only the heads of the
  * shared memory islands communicate (using comm_head).
  * Since we used key=0 in both MPI_Comm_split(...), process rank_world = 0
  * - is also rank 0 in comm_head
  * - and rank 0 in comm_shm in the color it belongs to.                    */

   if( comm_head != MPI_COMM_NULL ) // if( color == 0 )
   {
     MPI_Bcast(arr, arrSize, arrDataType, 0, comm_head);
     /* with this Bcast, all other rank_shm=0 processes write the data into their arr */
   }
…
```

Solutions of MPI shared memory exercise: datarep

data-rep_sol_3-6.c (on this slide steps 5-6)

```
…
/* ===> 5 <=== */
   MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
                              // after the fence all processes start a read epoch

/* Now, all other ranks in the comm_sm shared memory islands are allowed to access their shared memory
array. */
/* And all ranks rank_sm access the shared mem in order to compute sum  */
   sum = 0;
   for( i = 0; i < arrSize; i++)
   {
     //sum+= *( shm_buf_ptr - rank_shm * shmSize + i ) ;
     sum+= arr [ i ] ;
   }

/* ===> 6 <=== */
  /*TEST*/ // To minimize the output, we print only from 3 process per SMP node
  /*TEST*/ if ( rank_shm == 0 || rank_shm == 1 || rank_shm == size_shm - 1 )
     printf ("it: %i, rank ( world: %i/%i, shm: %i/%i, head: %i/%i ):\tsum(i=%d...i=%d) = %lld \n",
                  it,rank_world,size_world,rank_shm,size_shm,rank_head,size_head,it,arrSize-1+it,sum);
 }
  /*TEST*/ if(rank_world==0) printf("ALL finalize and return !!!.\n"); MPI_Finalize(); return 0;
…
```

Solutions of MPI shared memory exercise: datarep

data-rep_sol_7.c

```
…
/* ===> 7 <=== */
  MPI_Win_fence(/*workaround: no assertions:*/ 0, win);
                  // free destroys the shm. fence to guarantee that read epoch has been finished
  MPI_Win_free(&win);
…
```

data-rep_solution.c

Trick:
Calculate the minimum through
calculating the maximum for the negative values

```
…
/* ===> 2 <=== */
…
// ADD ON: calculates the minimum and maximum size of size_shm
  int mm[2], minmax[2]; mm[0] = -size_shm ; mm[1] = size_shm ;

  if( comm_head != MPI_COMM_NULL )
  {
    MPI_Reduce( mm, minmax, 2, MPI_INT, MPI_MAX, 0, comm_head) ;
  }
  if( rank_world == 0 )
  {
    printf("\n\tThe number of shared memory islands is: %i islands \n", size_head  ) ;
    if ( minmax[0] + minmax[1] == 0 )
    printf("\tThe size of all shared memory islands is: %i processes\n", -minmax[0] ) ;
    else
    printf("\tThe size of the shared memory islands is between min = %i and max = %i processes \n",
                                                                  -minmax[0], minmax[1]);
  }
// End of ADD ON. Note that the following algorithm does not require same sizes of the shared memory
islands

/* ===> 3 <=== */
…
```

Quiz on Shared Memory

A. Before you call **MPI_Win_allocate_shared**, what should you do?

MPI_Comm_split_type(comm_old, **MPI_COMM_TYPE_SHARED,** ..., &comm_sm)
will guarantee that comm_sm contains only processes **of the same shared memory island**.

B. If your communicator within your shared memory island consists of 12 MPI processes, and each process wants to get an own window with 10 doubles (each 8 bytes),

   a. which **window size** must you specify in **MPI_Win_allocate_shared**?

   10 * 8 = **80 bytes**

   b. And how long is the totally allocated shared memory?

   80 * 12 = **960 bytes**

   c. The returned base_ptr, will it be identical on all 12 processes?

   No, within each process, the base_ptr points to its own portion of the totally allocated shared mem.

   d. If all 12 processes want to have a pointer that points to the beginning of the totally allocated shared memory, which MPI procedure should you use and with which major argument?

   MPI_Win_shared_query with **rank = 0**

   e. If you do this, do these 12 pointers have identical values, i.e., are identical addresses?

   No, they point to the **same physical address**, but each MPI process may use **different virtual addresses** for this.

C. Which is the major method to store data from one process into the shared memory window portion of another process?

Normal assignments (with C/C++ or Fortran) to the correct location, i.e., **no** calls to MPI_Put/Get.

Quiz on Shared Memory Model & Synchronization

A. Which MPI memory model applies to MPI shared memory?
MPI_WIN_SEPARATE or MPI_WIN_UNIFIED ?

B. "Public and private copies are **eventually** synchronized without additional RMA calls."

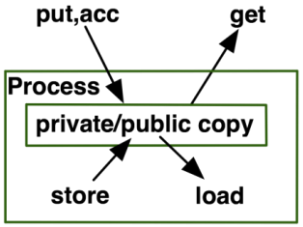


Figure: Courtesy of Torsten Hoefler

C. Which process-to-process synchronization methods can be used that, e.g., a store to a shared memory variable gets visible to another process (within the processes of the shared memory window)?
   1. **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   2. **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**
   3. **Any** (MPI) **synchronization** together with a **pair of C11 atomic_thread_fence(order)**

D. That such a store gets visible in another process after the synchronization is named here as "***write-read-rule***".
Which other rules are implied by such synchronizations and what do they mean?
   1. ***Read-write-rule***: a **load** (=*read*) in one process before the synchronization cannot be affected by a store (=*write*) in another process after the synchronization.
   2. ***Write-write-rule***: a **store** (=*write*) in one process before the synchronization cannot overwrite a **store** (=*write*) in another process after the synchronization.

E. How can you define a **race-condition** and which problems arise from **cache-line false-sharing?**
   1. Two processes access the **same shared variable** and at **least one process modifies** the variable **and the accesses are concurrent**.
   2. Significant performance problems if two or more processes **often access different portions of the same cache-line**.

Quiz on Virtual topologies

A. Which types of MPI topologies for virtual process grids exist?

B. And for which **use cases**?

1. **Cartesian topologies**
   - **For** Cartesian data meshes with identical compute time per mesh element
   - **For** any Cartesian process grid with identical compute time per process and numerical epoch, and its communication mainly on the virtual Cartesian grid between the processes
2. **Distributed graph topologies and graph topologies**
   - **For** applications with unstructured grids

C. Where are **limits** for using virtual topologies, i.e., which use cases do not really fit?

– Applications with mesh refinements, dynamic load balancing and diffusion of mesh elements to other processes
→ all cases with **changing virtual process grids over time;**

– Communication pattern not known in advance.