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 @ VSC Vienna, Dec 12-14, 2022

http://tiny.cc/MPIX-VSC

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

Warmup survey

- For quizzes and surveys,
  - Keep a browser tab open on https://menti.com

    Links is also in Moodle
  - To join the quizzes and surveys,
    enter the number given in the menti.com screen share on the top of the screen
  - Alternatively, click on the link in the Zoom chat
  - Have fun ;-)

General outline

Introduction

Programming Models (13)

- MPI + OpenMP on multi/many-core (14) + Exercises
- MPI + Accelerators (99)
- MPI + MPI-3 shared memory (115) + Exercise (143)
- Pure MPI communication (175)

Conclusions (Summary (232), Acknowledgements (238), Conclusions (239))

Appendix (240) (Abstract (240), Authors (240), Solutions (245))

Introduction

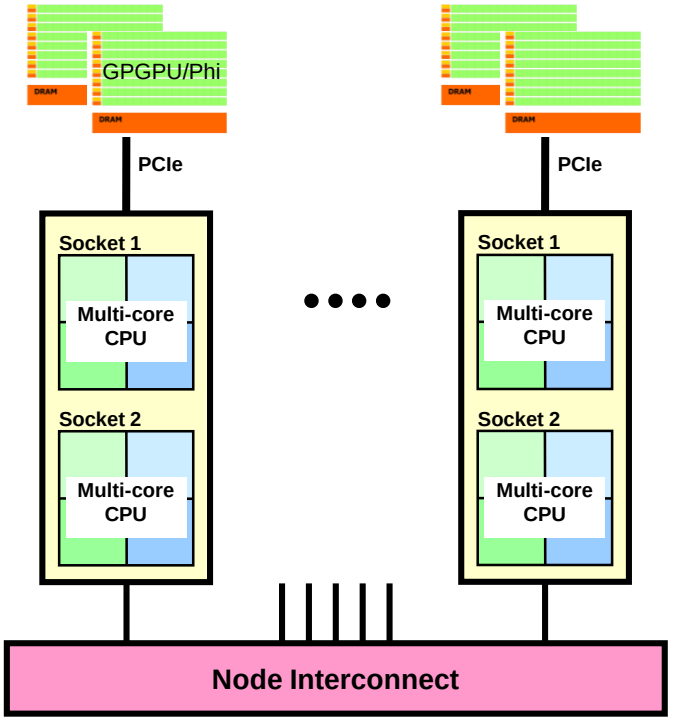
Hardware and programming models

Hardware Bottlenecks

Questions addressed in this tutorial

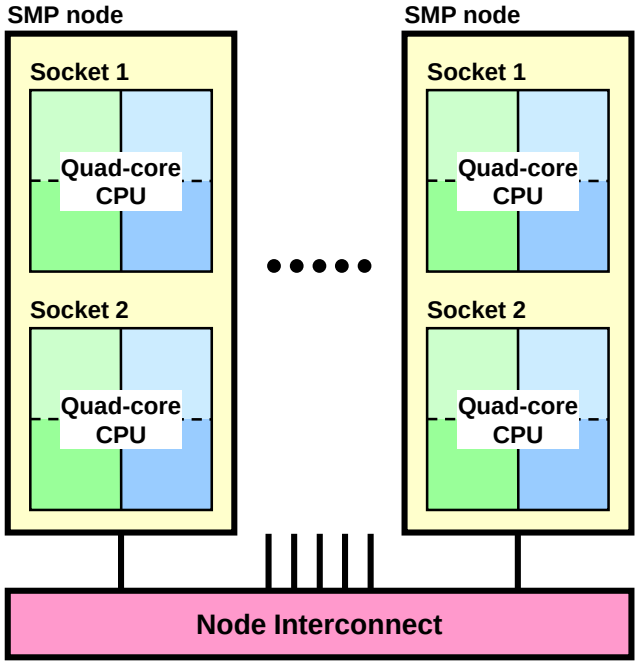
Remarks on Cost-Benefit Calculation

Hardware and programming models



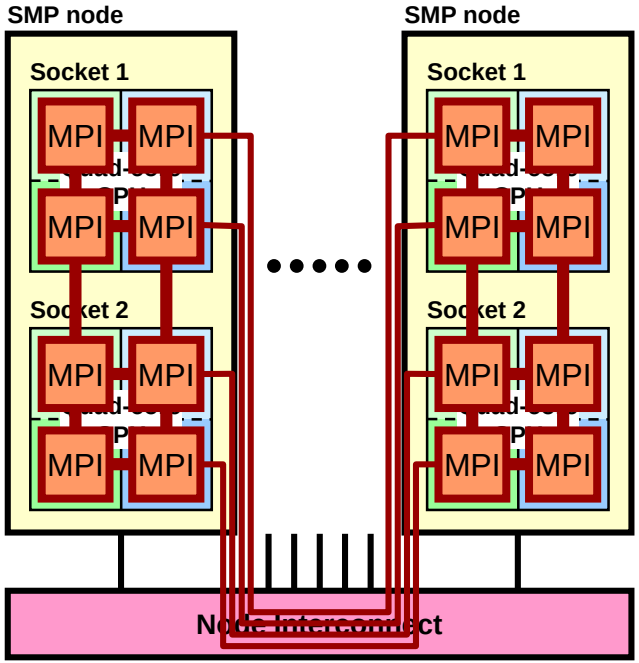
- MPI + threading
  - OpenMP
  - Cilk(+)
  - TBB (Threading Building Blocks)
- MPI + MPI shared memory
- MPI + accelerator
  - OpenACC
  - OpenMP accelerator support
  - CUDA
  - OpenCL, Kokkos, SYCL,…
- Pure MPI communication

Options for running code on multicore clusters

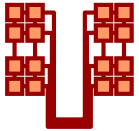


- Which programming model is fastest?

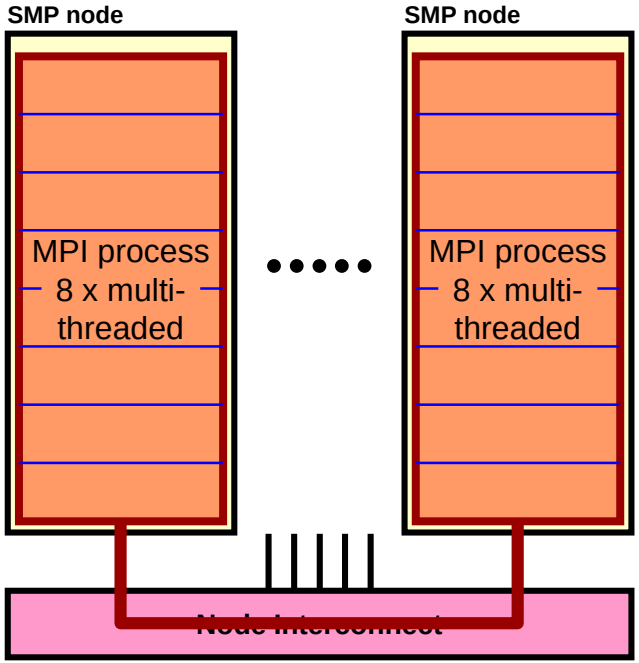
Options for running code on multicore clusters



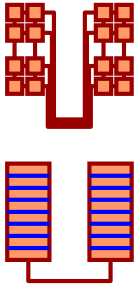
- Which programming model is fastest?
  - MPI everywhere?



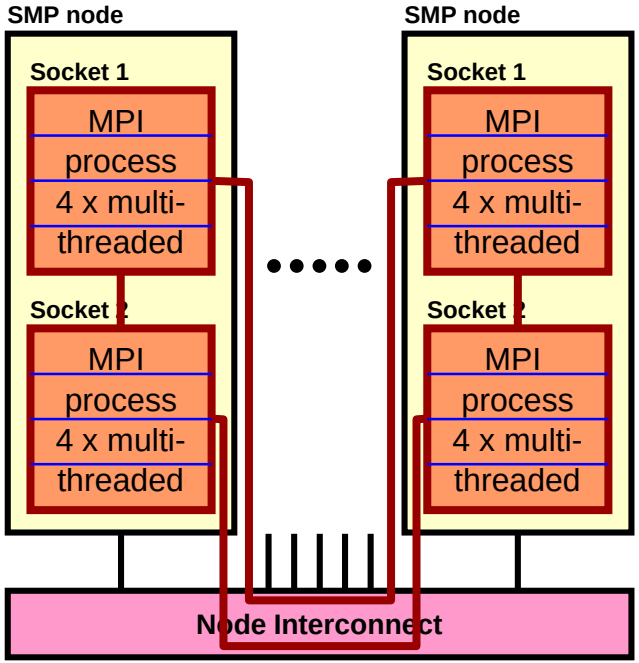
Options for running code on multicore clusters



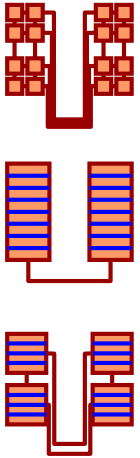
- Which programming model is fastest?
  - MPI everywhere?
  - Fully hybrid MPI & OpenMP?



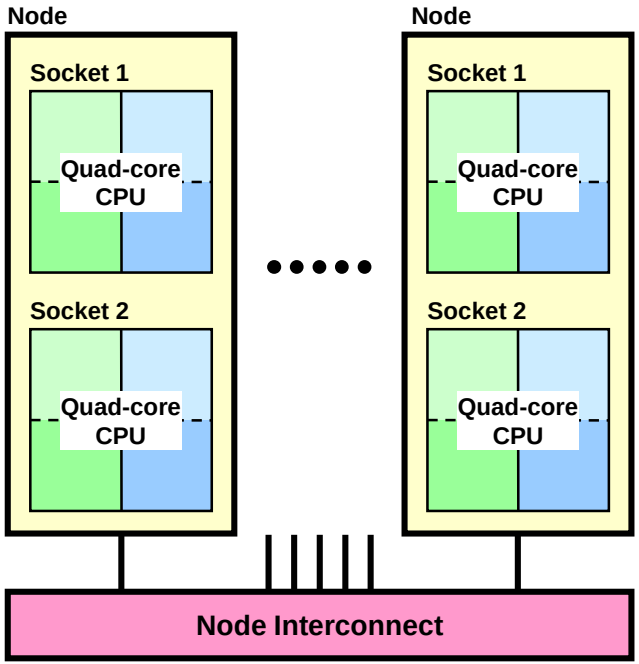
Options for running code on multicore clusters



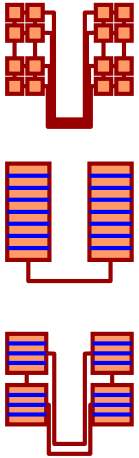
- Which programming model is fastest?
  - MPI everywhere?
  - Fully hybrid MPI & OpenMP?
  - Something between? (Mixed model)



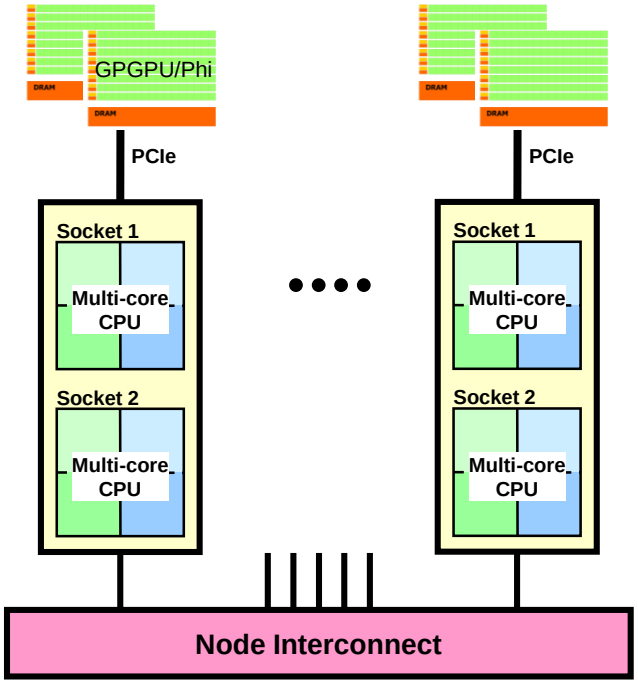
Options for running code on multicore clusters



- 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 with accelerators



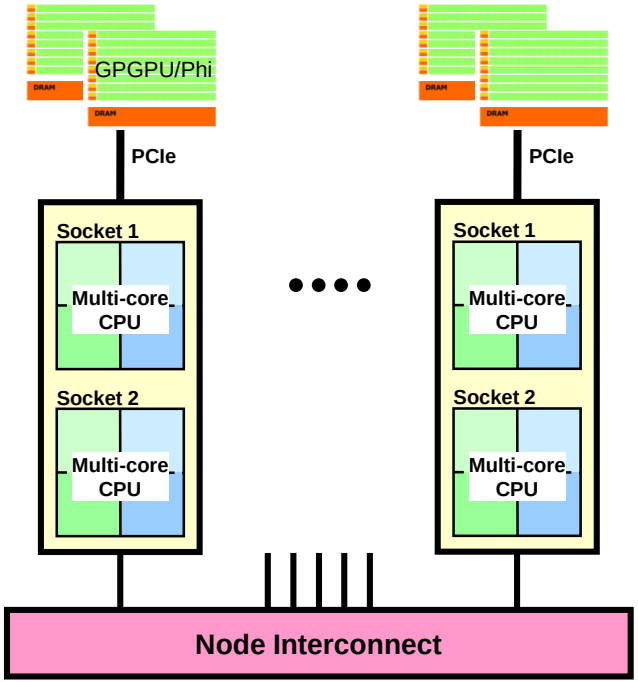
Hierarchical hardware

- Many levels

Hierarchical parallel programming

- Many options for MPI+X:
  one MPI process per
  - node
  - CPU
  - ccNUMA domain
  - […]
  - core
  - hyper-thread

More Options with accelerators



Hierarchical hardware

- Many levels

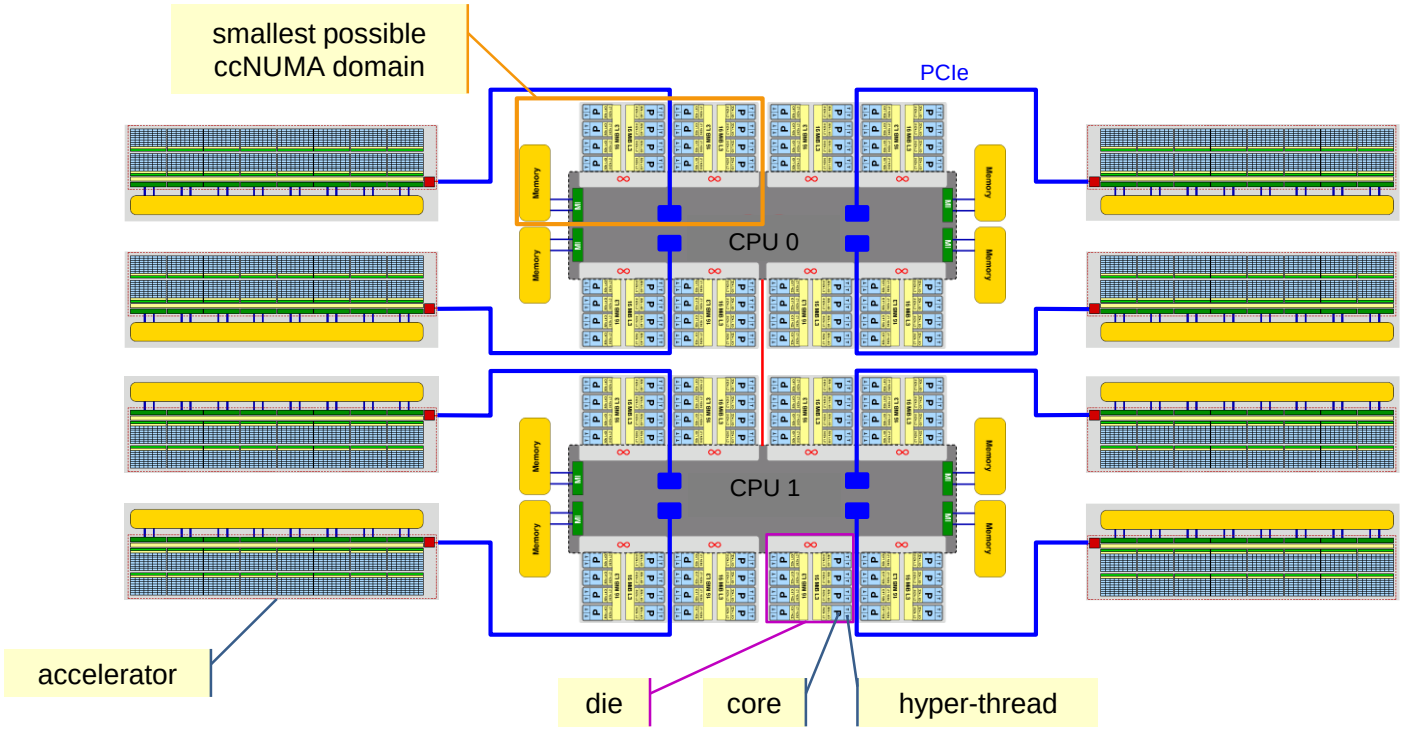
Hierarchical parallel programming

- Many options for MPI+X: one MPI process per
  - node
  - CPU
  - ccNUMA domain
  - […]
  - core
  - hyper-thread

Where is the bottleneck?

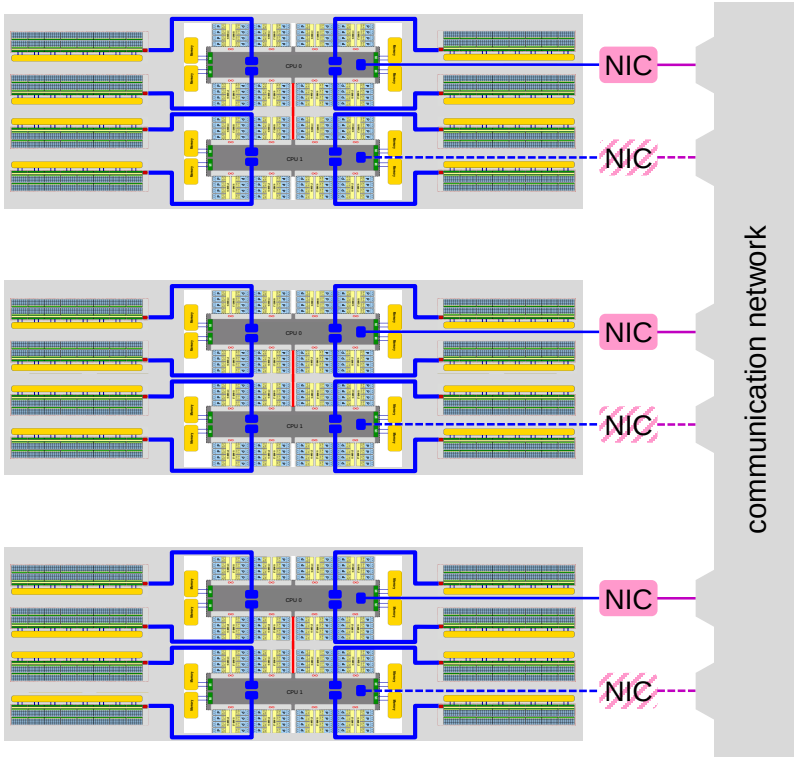
Dual-CPU ccNUMA + accelerator node architecture

Actual topology of a modern compute node



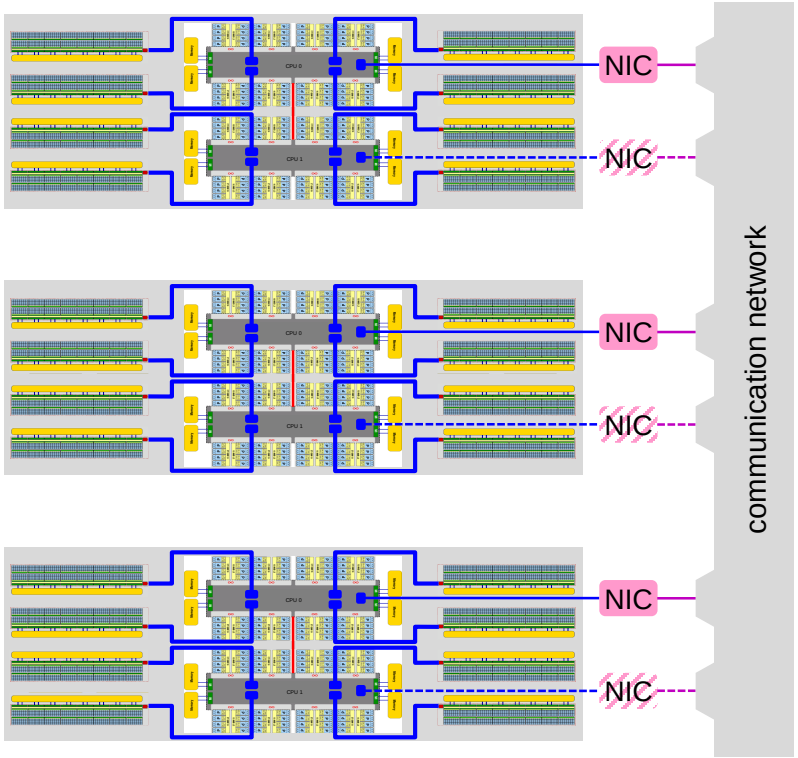
Hardware bottlenecks

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



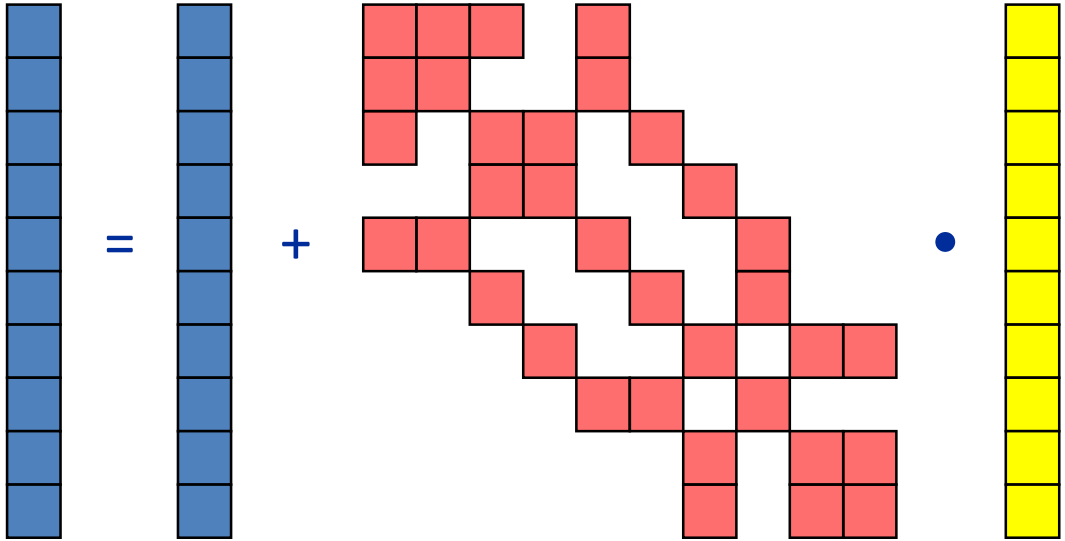
Hardware bottlenecks

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



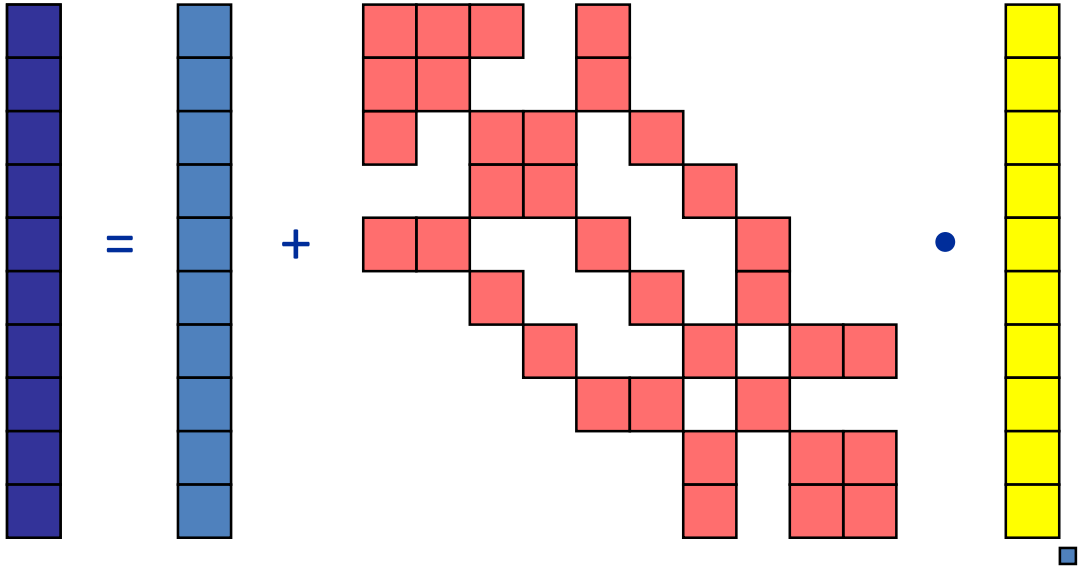
Example: Hardware bottlenecks in SpMV

- Sparse matrix-vector-multiply with stored matrix entries
  - Bottleneck: memory bandwidth of each CPU
- SpMV with calculated matrix entries (many complex operations per entry)
  - Bottleneck: computational speed of each core
- SpMV with highly scattered matrix entries
  - Bottleneck: Inter-node communication



Example: Hardware bottlenecks in SpMV

- Sparse matrix-vector-multiply with stored matrix entries
  - Bottleneck: memory bandwidth of each CPU
- SpMV with calculated matrix entries (many complex operations per entry)
  - Bottleneck: computational speed of each core
- SpMV with highly scattered matrix entries
  - Bottleneck: Inter-node communication



Questions addressed in this tutorial

- What is the performance impact of system topology?

Questions addressed in this tutorial

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?

Questions addressed in this tutorial

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?

Questions addressed in this tutorial

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?
- How can I leverage multiple accelerators?
  - What are typical challenges?

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 + MPI-3.0 shared memory + Exercise**
- **Pure MPI communication + Exercise**
- **MPI + Accelerators**

Programming models - MPI + OpenMP

General considerations	slide 15
How to compile, link, and run	20
Hands-on: Hello hybrid!	28
System topology, ccNUMA, and memory bandwidth	30
Memory placement on ccNUMA systems	42
Topology and affinity on multicore	51
Hands-on: Pinning	66
Case study: The Multi-Zone NAS Parallel Benchmarks	67
Hands-on: Masteronly hybrid Jacobi	74
Overlapping communication and computation	77
Communication overlap with OpenMP taskloops	84
Hands-on: Taskloop-based hybrid Jacobi	94
Main advantages, disadvantages, conclusions	95

Programming models - MPI + OpenMP

General considerations

> **General considerations**
- How to compile, link, and run
  - Hands-on: Hello hybrid!
- System topology, ccNUMA, and memory bandwidth
  - Memory placement on ccNUMA systems
- Topology and affinity on multicore
  - Hands-on: Pinning
- Case study: The Multi-Zone NAS Parallel Benchmarks
  - Hands-on: Masteronly hybrid Jacobi
- Overlapping communication and computation
  - Communication overlap with OpenMP taskloops
  - Hands-on: Taskloop-based hybrid Jacobi
- Main advantages, disadvantages, conclusions

Potential advantages of MPI+OpenMP

Simple level

- Leverage additional levels of parallelism
  - Scaling to higher number of cores
  - Adding OpenMP with incremental additional parallelization
- Enable flexible load balancing on OpenMP level
  - Fewer MPI processes leave room for assigning workload more evenly
  - MPI processes with higher workload could employ more threads
  - Cheap OpenMP load balancing (tasking, dynamic/guided loops)
- Lower communication overhead (possibly)
  - Few “fat” MPI processes vs many “skinny” processes
  - Fewer messages and smaller amount of data communicated
- Lower memory requirements due to fewer MPI processes
  - Reduced amount of application halos & replicated data
  - Reduced size of MPI internal buffer space

Advanced level

- Explicit communication/computation overlap

MPI + any threading model

Special MPI init for multi-threaded MPI processes is required:

```
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);
```

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

MPI + any threading model

Special MPI init for multi-threaded MPI processes is required:

```
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);
```

- Possible values for **`thread_level_required`** (increasing order):
  - **`MPI_THREAD_SINGLE`** Only one thread will execute
  - **`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

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

MPI + any threading model

Special MPI init for multi-threaded MPI processes is required:

```
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);
```

- Possible values for **thread_level_required** (increasing order):
  - **MPI_THREAD_SINGLE** Only one thread will execute
  - **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

may imply higher latencies due to some internal locks

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

MPI + any threading model

Special MPI init for multi-threaded MPI processes is required:

```
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);
```

- Possible values for **thread_level_required** (increasing order):
  - **MPI_THREAD_SINGLE** Only one thread will execute
  - **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

may imply higher latencies due to some internal locks

- returned **thread_level_provided** may be less or more than **thread_level_required**

  → **if (thread_level_provided < thread_level_required) MPI_Abort(…);**

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

MPI + any threading model

Special MPI init for multi-threaded MPI processes is required:

```
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);
```

- Possible values for **thread_level_required** (increasing order):
  - **MPI_THREAD_SINGLE** Only one thread will execute
  - **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

may imply higher latencies due to some internal locks

- returned **thread_level_provided** may be less or more than **thread_level_required**

  → **if (thread_level_provided < thread_level_required) MPI_Abort(…);**

recommended directly after MPI_Init_thread

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

Hybrid MPI+OpenMP masteronly style

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

  /* on master only */
     MPI_Isend();
     MPI_Irecv();
     MPI_Waitall();
} /* end for loop */
```

masteronly style:
MPI only outside of parallel regions

Hybrid MPI+OpenMP masteronly style

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

  /* on master only */
     MPI_Isend();
     MPI_Irecv();
     MPI_Waitall();
} /* end for loop */
```

masteronly style:
MPI only outside of parallel regions

Advantages

- Simplest possible hybrid model
- Thread-parallel execution and MPI communication strictly separate
- Minimally required MPI thread support level: **`MPI_THREAD_FUNNELED`**

Hybrid MPI+OpenMP masteronly style

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

  /* on master only */
     MPI_Isend();
     MPI_Irecv();
     MPI_Waitall();
} /* end for loop */
```

masteronly style:
MPI only outside of parallel regions

Advantages

- Simplest possible hybrid model
- Thread-parallel execution and MPI communication strictly separate
- Minimally required MPI thread support level: **`MPI_THREAD_FUNNELED`**

Major Problems

- All other threads are sleeping while master thread communicates!
- Only one thread per process communicating
  → possible underutilization of network bandwidth

Masteronly style within large parallel region

```
#pragma omp parallel
for(iterations) {
  #pragma omp for
  for(i=0; ...)  {
    // ... numerics
  } // barrier here
  #pragma omp single
  {
     MPI_Isend();
     MPI_Irecv();
     MPI_Waitall();
  } // Barrier here
} /* end iter loop */
```

- Barrier before MPI required
  - May be implicit
  - Prevent race conditions on communication buffer data
    - Between multi-threaded numerics
    - and MPI access by master thread
  - Enforce flush of variables
- Barrier after MPI required
  - May be implicit
  - Numerical loop(s) may need communicated data

Programming models - MPI + OpenMP

How to compile, link, and run

General considerations
> **How to compile, link, and run**
Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
Memory placement on ccNUMA systems
Topology and affinity on multicore
Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
Communication overlap with OpenMP taskloops
Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

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)

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

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…)

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…)
  - Figure out how to start fewer MPI processes than cores per node

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…)
  - Figure out how to start fewer MPI processes than cores per node
  - Pinning (who is running where?) is extremely important → see later

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 compilers

- Intel MPI + Intel C

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

- Intel MPI + Intel Fortran

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

Compiling from a single source

Handling compilers

- Intel MPI + Intel C

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

- Intel MPI + Intel Fortran

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

- OpenMPI + gcc

```
mpicc        -DUSE_MPI -fopenmp      ...
gcc                    -fopenmp      ...
```

- OpenMPI + gfortran

```
mpif90    -cpp -DUSE_MPI -fopenmp      ...
gfortran  -cpp           -fopenmp      ...
```

Examples for compilation and execution

- Cray XC40 (2 NUMA domains w/ 12 cores each):
  - `ftn -h omp ...`
  - `OMP_NUM_THREADS=12 aprun -n 4 -N 2 \`
    `-d $OMP_NUM_THREADS ./a.out`

Examples for compilation and execution

- Cray XC40 (2 NUMA domains w/ 12 cores each):
  - `ftn -h omp ...`
  - ```
    OMP_NUM_THREADS=12 aprun -n 4 -N 2 \
                             -d $OMP_NUM_THREADS ./a.out
    ```
- Intel Ice Lake (36-core 2-socket) cluster, Intel MPI/OpenMP
  - `mpiifort -qopenmp ...`
  - ```
    mpirun -ppn 2 -np 4 \
           -env OMP_NUM_THREADS 36
           -env I_MPI_PIN_DOMAIN socket \
           -env KMP_AFFINITY scatter ./a.out
    ```

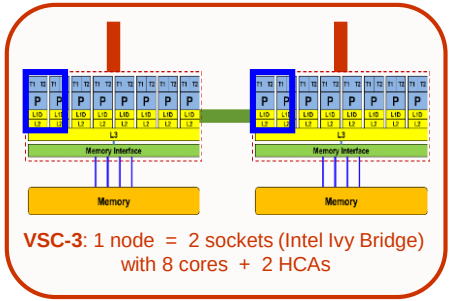
Examples for compilation and execution

- Cray XC40 (2 NUMA domains w/ 12 cores each):
  - `ftn -h omp ...`
  - ```
    OMP_NUM_THREADS=12 aprun -n 4 -N 2 \
                             -d $OMP_NUM_THREADS ./a.out
    ```
- Intel Ice Lake (36-core 2-socket) cluster, Intel MPI/OpenMP
  - `mpiifort -qopenmp ...`
  - ```
    mpirun -ppn 2 -np 4 \
           -env OMP_NUM_THREADS 36
           -env I_MPI_PIN_DOMAIN socket \
           -env KMP_AFFINITY scatter ./a.out
    ```
- Intel Ice Lake (36-core 2-socket) cluster, Intel MPI/OpenMP + likwid-mpirun
  - `likwid-mpirun -np 4 -pin S0:0-35_S1:0-35 ./a.out`

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 ...]
```



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

Learning about node topology

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

Caveat:
Numbering may differ for different setups of same CPU!

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                                         | |
| +-------------------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------+
```

Programming models - MPI + OpenMP

Hands-On #1

Hello hybrid!

General considerations
How to compile, link, and run
> **Hands-on: Hello hybrid!**
System topology, ccNUMA, and memory bandwidth
Memory placement on ccNUMA systems
Topology and affinity on multicore
Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
Communication overlap with OpenMP taskloops
Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

Hands-On #1

he-hy - Hello Hybrid! - compiling, starting

1. FIRST THINGS FIRST - PART 1: find out about a (new) cluster - login node
2. FIRST THINGS FIRST - PART 2: find out about a (new) cluster - batch jobs
3. MPI+OpenMP: :**TODO**: how to compile and start an application how to do conditional compilation
4. MPI+OpenMP: :**TODO**: get to know the hardware - needed for pinning

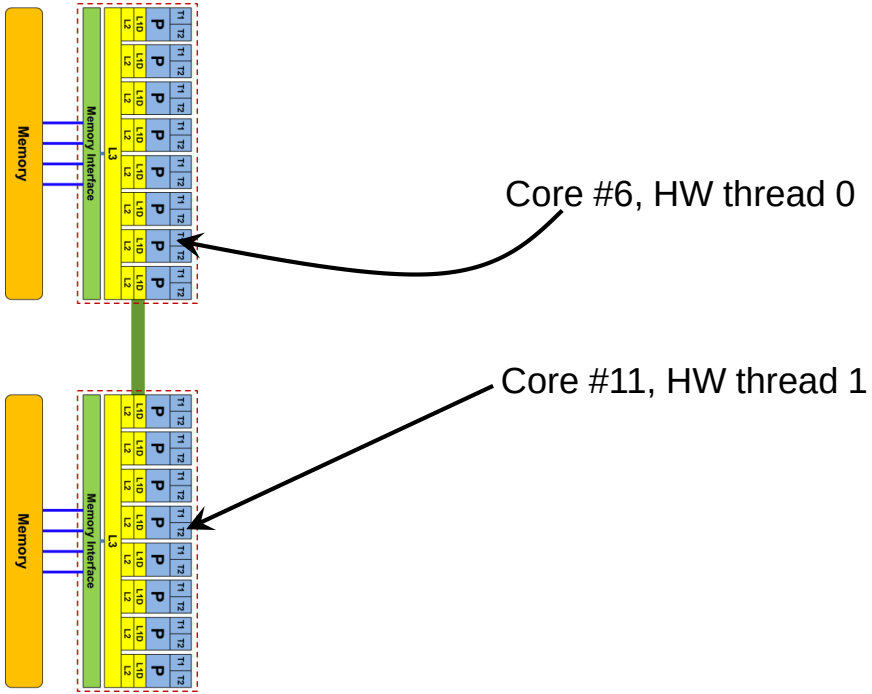
Programming models - MPI + OpenMP

System topology, ccNUMA, and memory bandwidth

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
> **System topology, ccNUMA, and memory bandwidth**
  Memory placement on ccNUMA systems
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
  Communication overlap with OpenMP taskloops
  Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

What is “topology”?

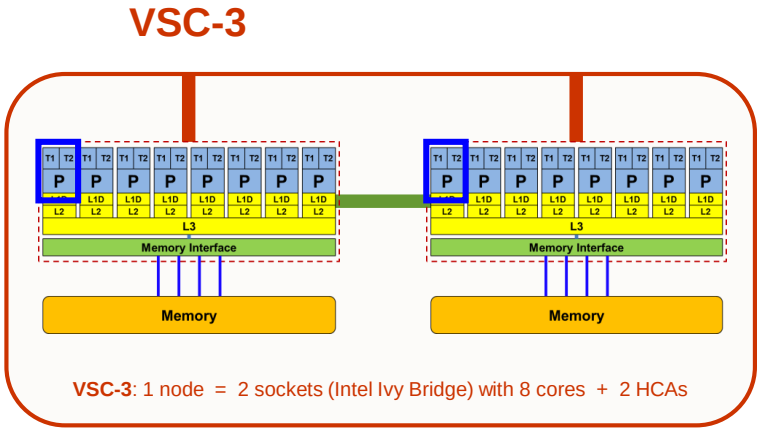
Where in the machine does core (or hardware thread) #n reside?



Why is this important?

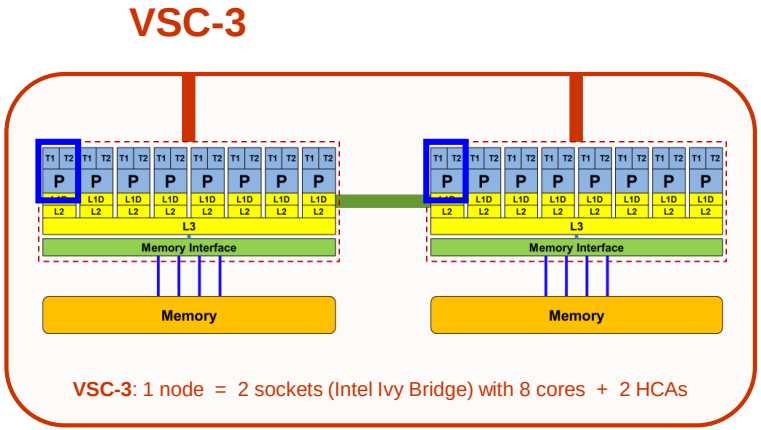
- Resource sharing (cache, data paths)
- Communication efficiency (shared vs. separate caches, buffer locality)
- Memory access locality (ccNUMA!)

Compute nodes – caches



Compute nodes – caches

Latency	← typical →	Bandwidth
1–2 ns	L1 cache	200 GB/s
3–10 ns	L2/L3 cache	50 GB/s
100 ns	memory	20 GB/s (1 core)



Ping-Pong Benchmark – Latency

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

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

```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
  E = get_walltime()
  GBYTES = 2*N/(E-S)/1.d9 ! Gbyte/s rate
  TIME = (E-S)/2*1.d6     ! transfer time
else
  targetID = 0
  call Receive_message(buffer,N,targetID)
  call Send_message(buffer,N,targetID)
endif
```

Ping-Pong Benchmark – Latency

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

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

```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
  E = get_walltime()
  GBYTES = 2*N/(E-S)/1.d9 ! Gbyte/s rate
  TIME = (E-S)/2*1.d6     ! transfer time
else
  targetID = 0
  call Receive_message(buffer,N,targetID)
  call Send_message(buffer,N,targetID)
endif
```

Latency [µs]	MPI_Send(…)	
	OpenMPI	Intel MPI
intra-socket	0.3 µs	0.3 µs
inter-socket	0.6 µs	0.7 µs
IB -1- edge	1.2 µs	1.4 µs
IB -2- leaf	1.6 µs	1.8 µs
IB -3- spine	2.1 µs	2.3 µs

For comparison: typical latencies	
L1 cache	1–2 ns
L2/L3 c.	3–10 ns
memory	100 ns
HPC networks	1–10 µs

Ping-Pong Benchmark – Latency

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

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

Affinity matters!

```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
  E = get_walltime()
  GBYTES = 2*N/(E-S)/1.d9 ! Gbyte/s rate
  TIME = (E-S)/2*1.d6     ! transfer time
else
  targetID = 0
  call Receive_message(buffer,N,targetID)
  call Send_message(buffer,N,targetID)
endif
```

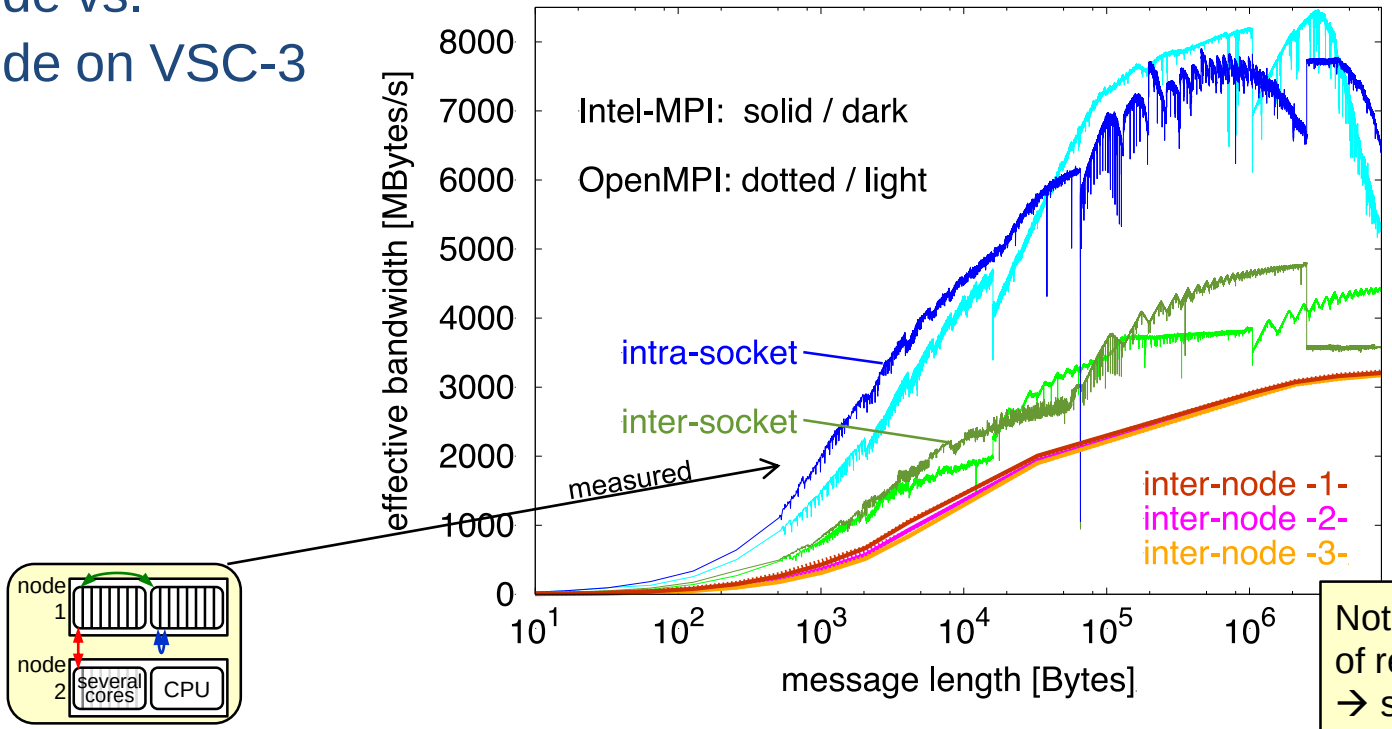
Latency [µs]	MPI_Send(…)	
	OpenMPI	Intel MPI
intra-socket	0.3 µs	0.3 µs
inter-socket	0.6 µs	0.7 µs
IB -1- edge	1.2 µs	1.4 µs
IB -2- leaf	1.6 µs	1.8 µs
IB -3- spine	2.1 µs	2.3 µs

For comparison: typical latencies	
L1 cache	1–2 ns
L2/L3 c.	3–10 ns
memory	100 ns
HPC networks	1–10 µs

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

Ping-Pong 1-on-1 Benchmark – Effective Bandwidth

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

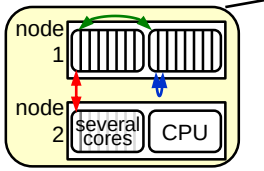


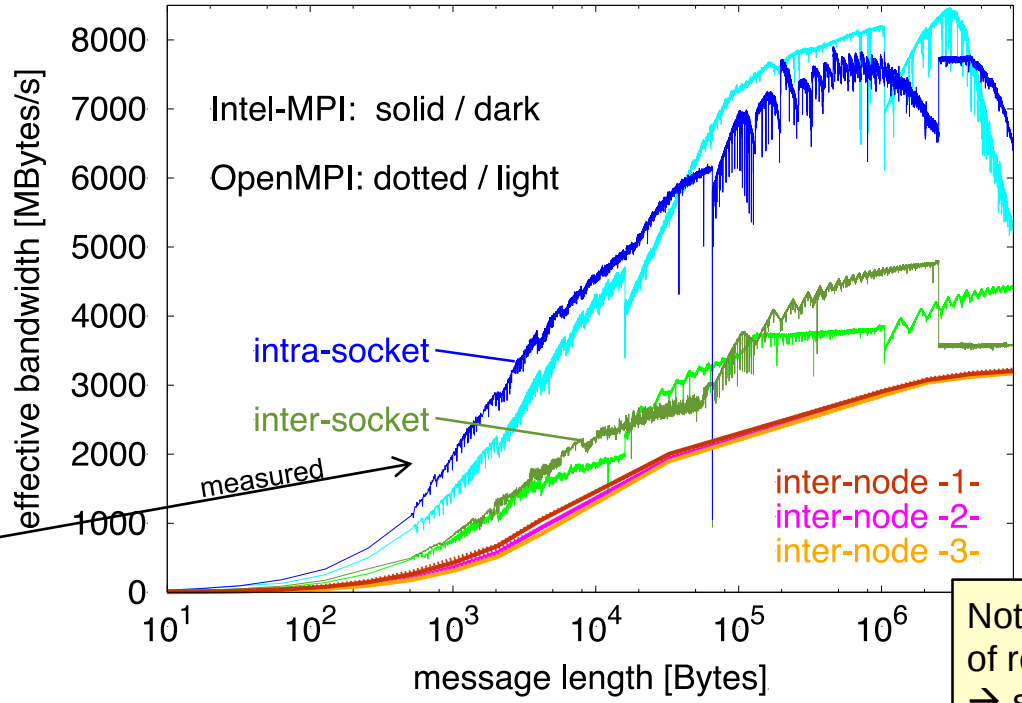
Not representative of real applications → see next slide(s)

Ping-Pong 1-on-1 Benchmark – Effective 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





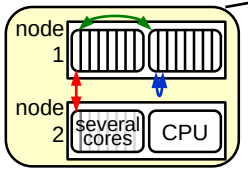
Not representative of real applications → see next slide(s)

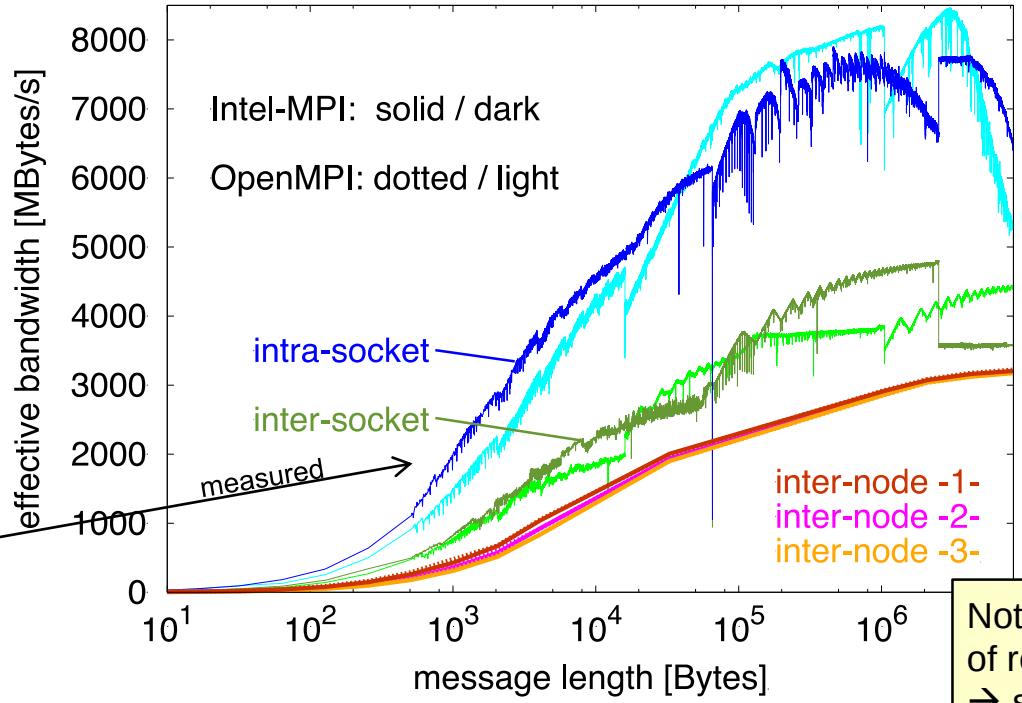
Ping-Pong 1-on-1 Benchmark – Effective Bandwidth

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





Not representative of real applications → see next slide(s)

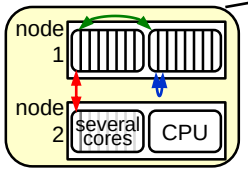
Ping-Pong 1-on-1 Benchmark – Effective Bandwidth

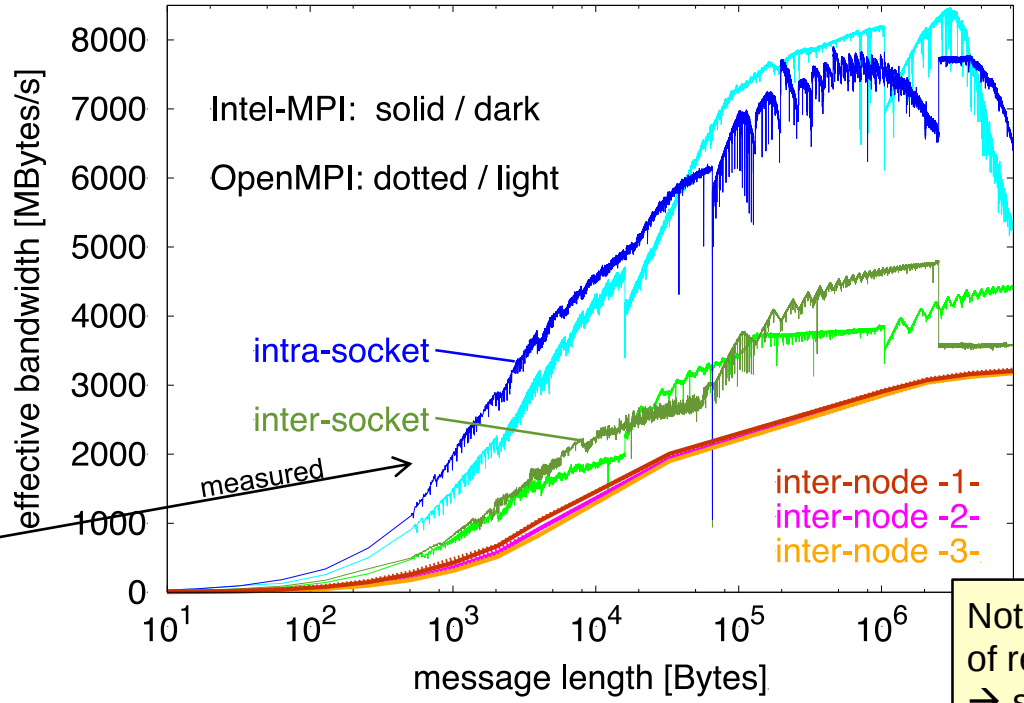
intra-node vs. inter-node on VSC-3

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)





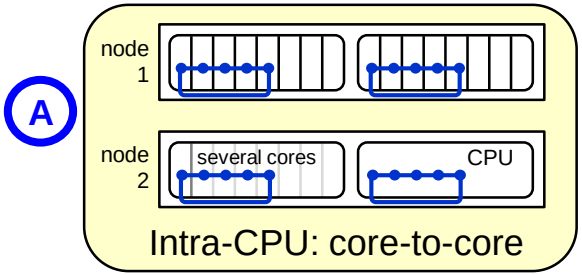
Not representative of real applications → see next slide(s)

Multiple communicating rings

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/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

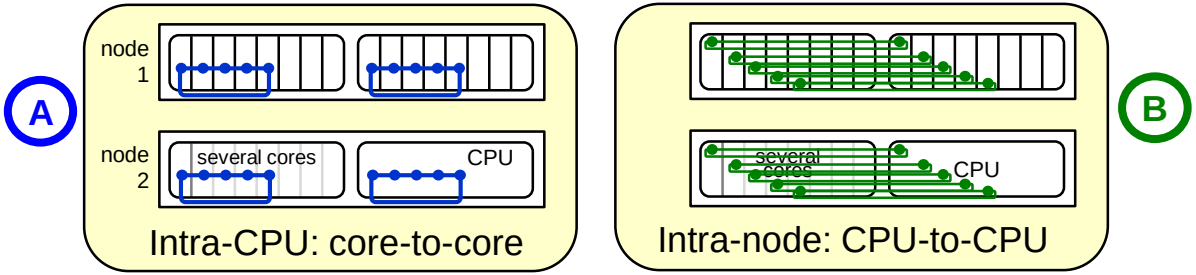


Multiple communicating rings

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/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

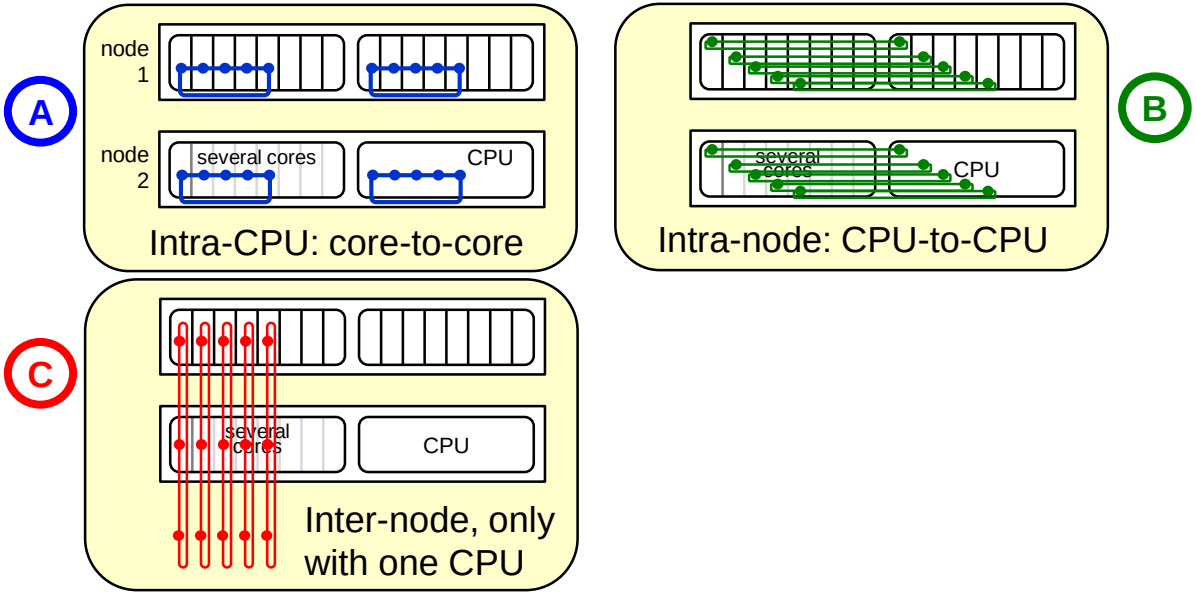


Multiple communicating rings

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/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

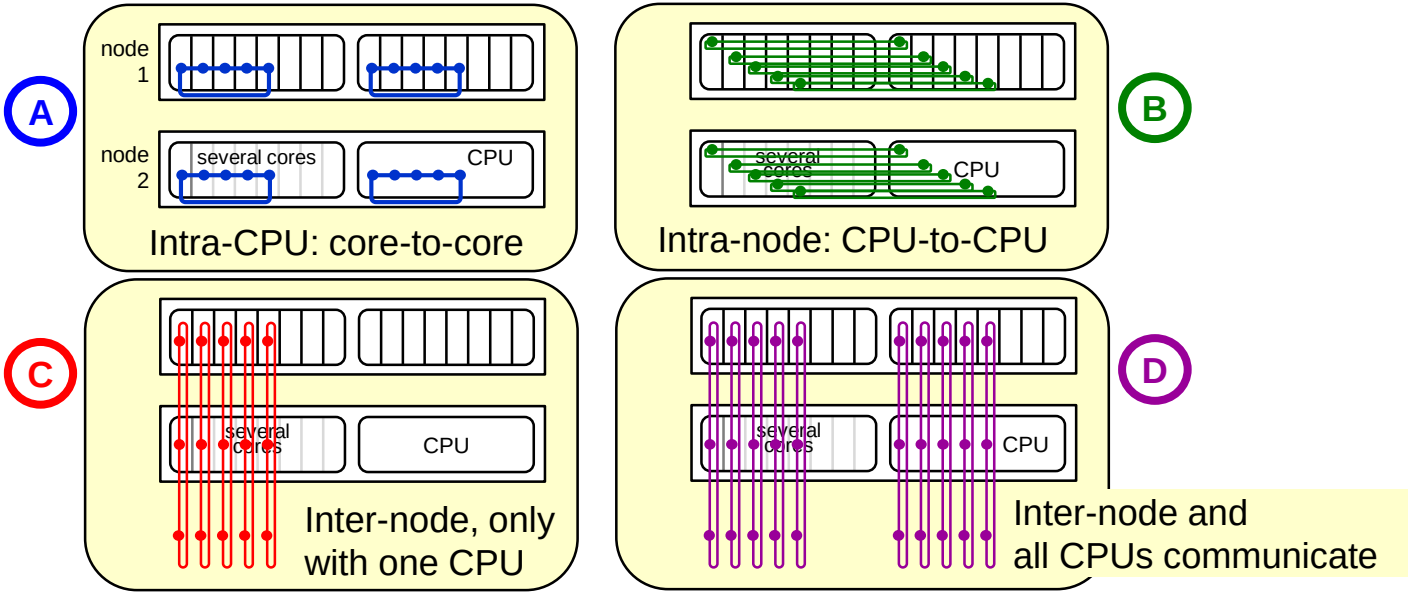


Multiple communicating rings

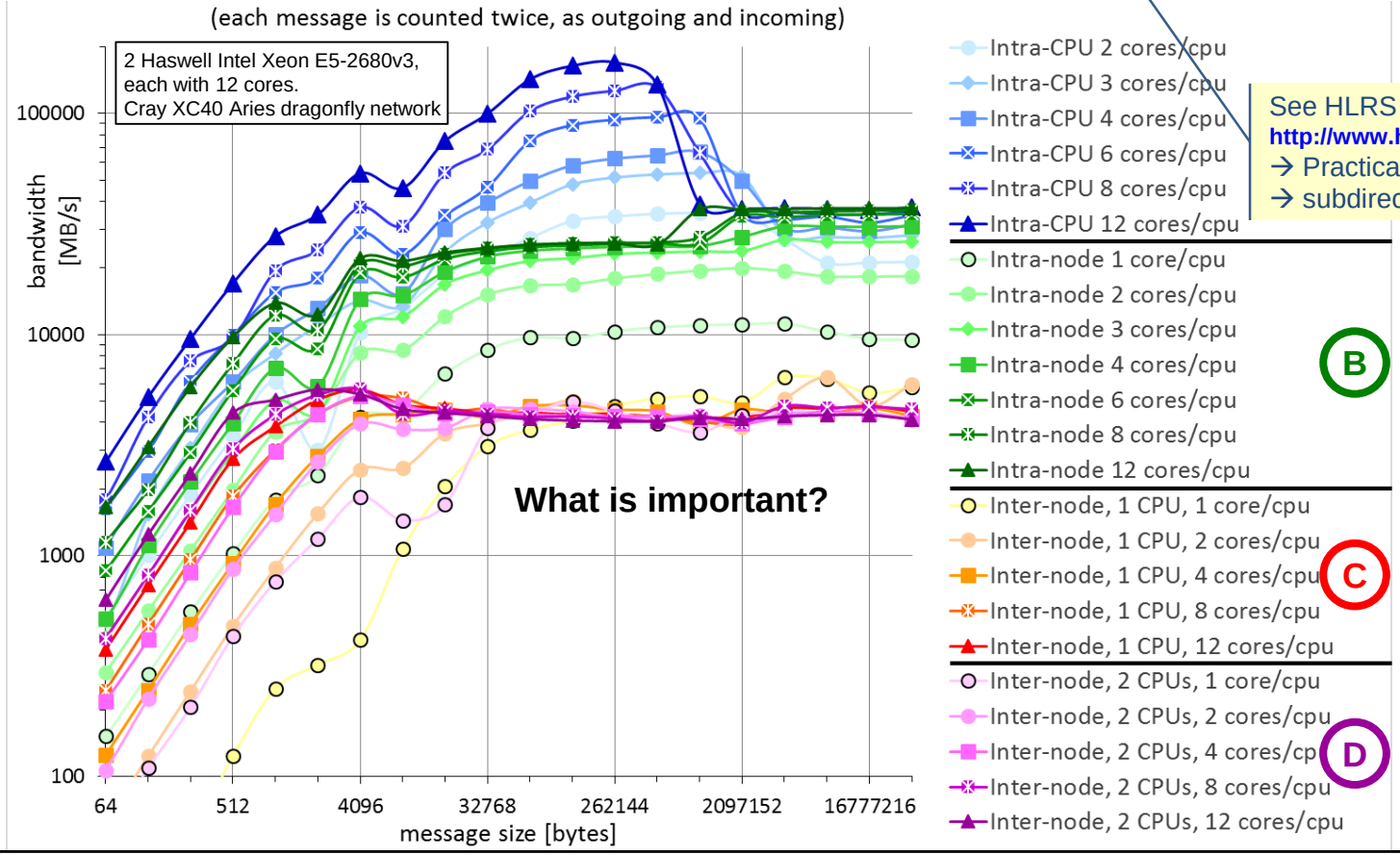
Benchmark halo_irecv_send_multiplelinks_toggle.c

- 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/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/



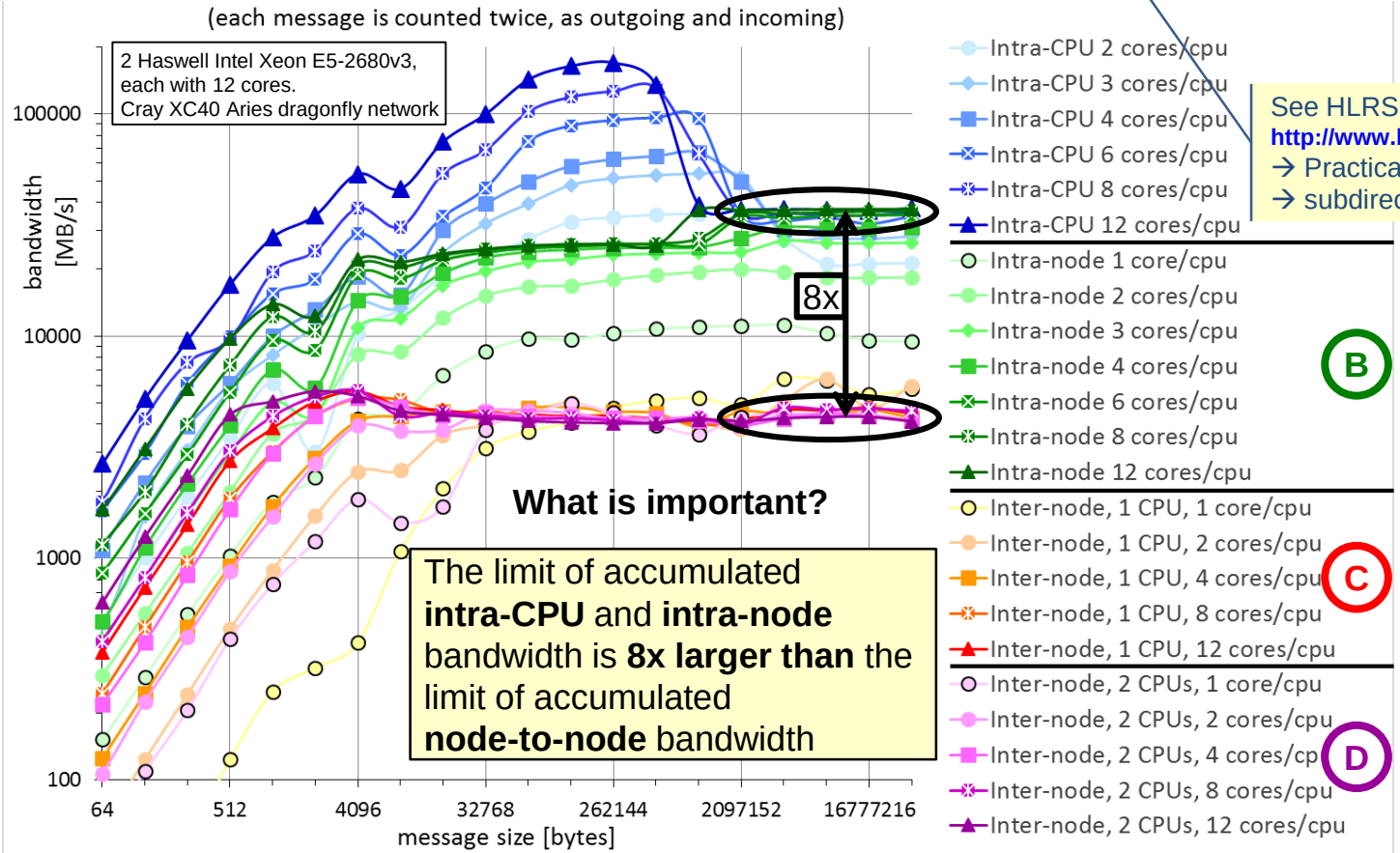
Duplex accumulated ring bandwidth per node



See HLRS online courses
http://www.hlrs.de/training/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

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)

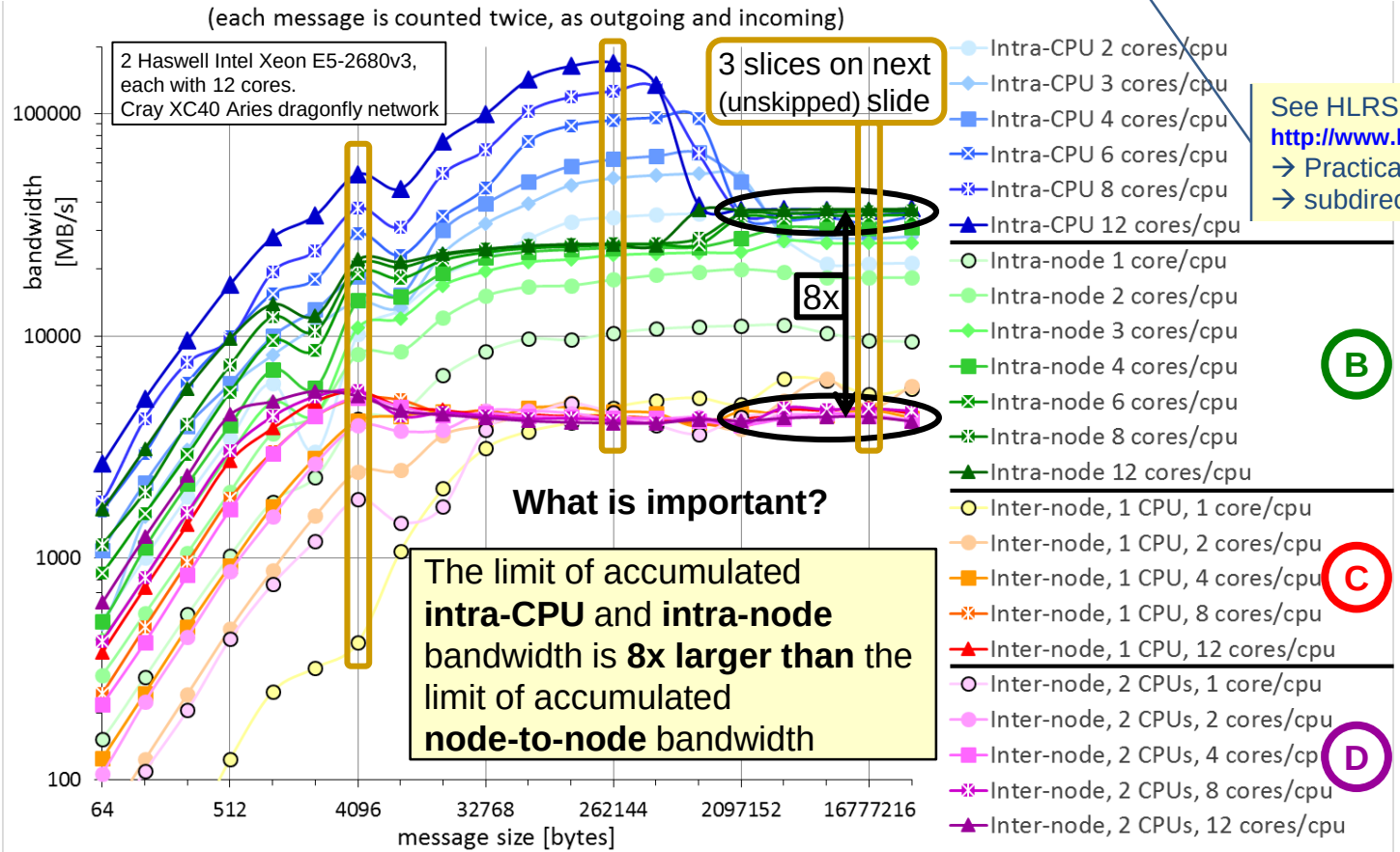
Duplex accumulated ring bandwidth per node



See HLRS online courses
http://www.hlrs.de/training/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

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

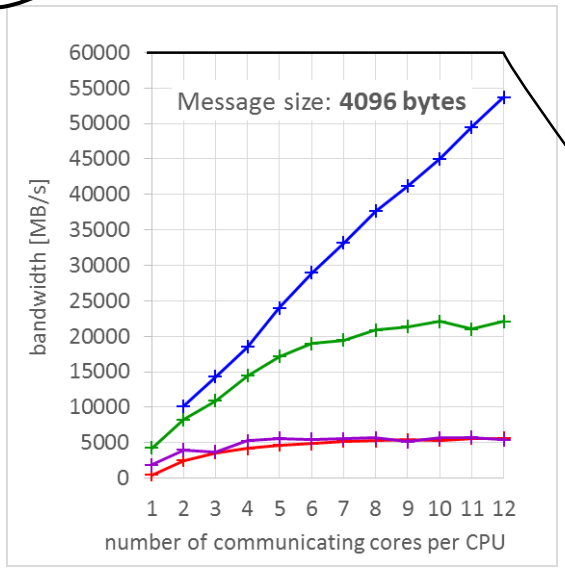


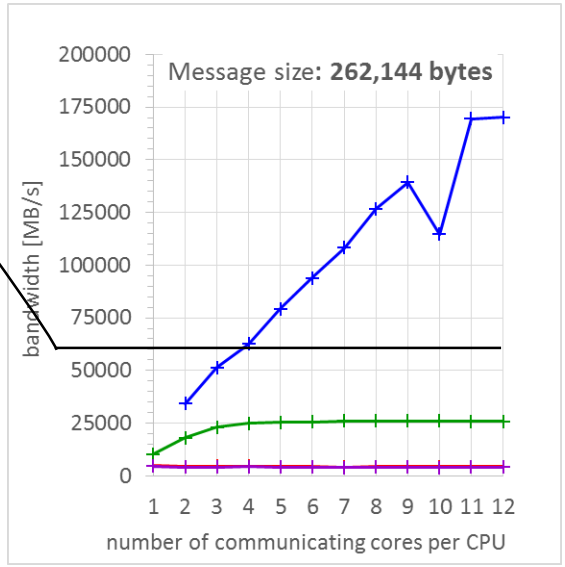
See HLRS online courses
http://www.hlrs.de/training/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

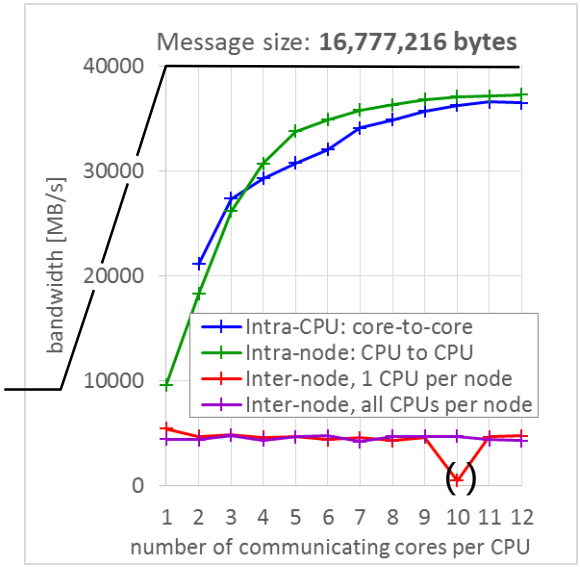
Measurement with halo_irecv_send_multiplelinks_toggle.c on 4 nodes of Cray XC40 hazelhen.hww.de, June 15, 2018, HLRS, by Rolf Rabenseifner (protocol 10)

Accumulated – scaling vs. asymptotic behavior

skipped

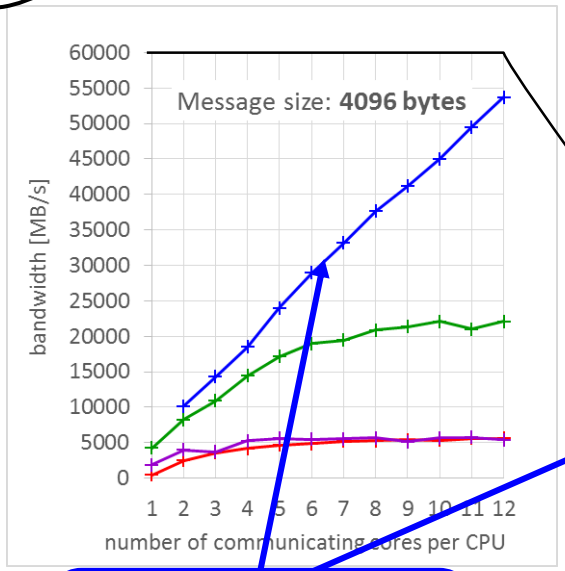


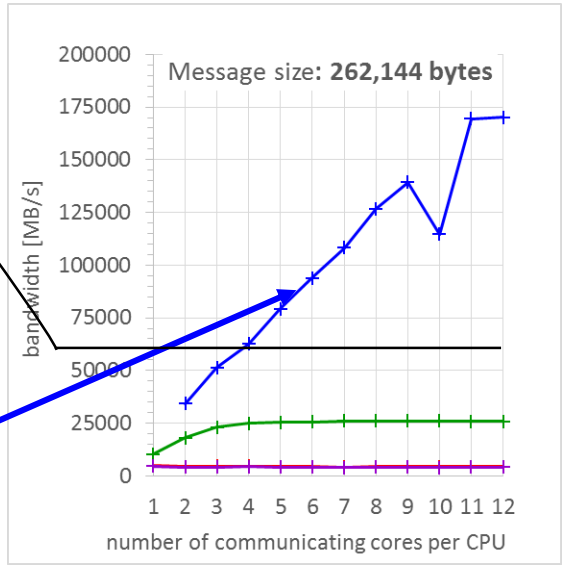


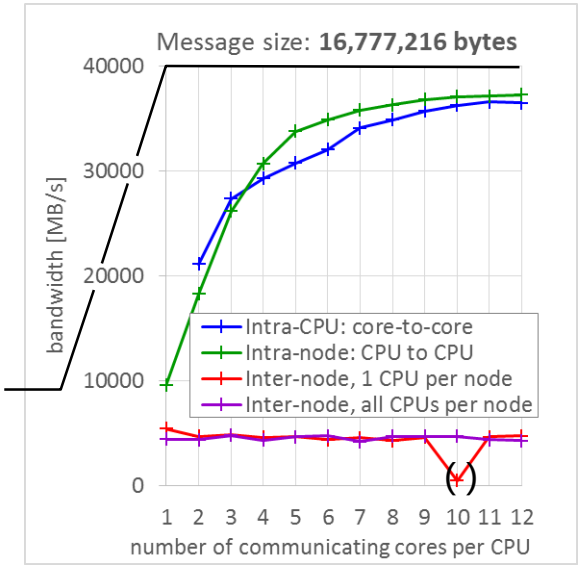


skipped

Accumulated – scaling vs. asymptotic behavior



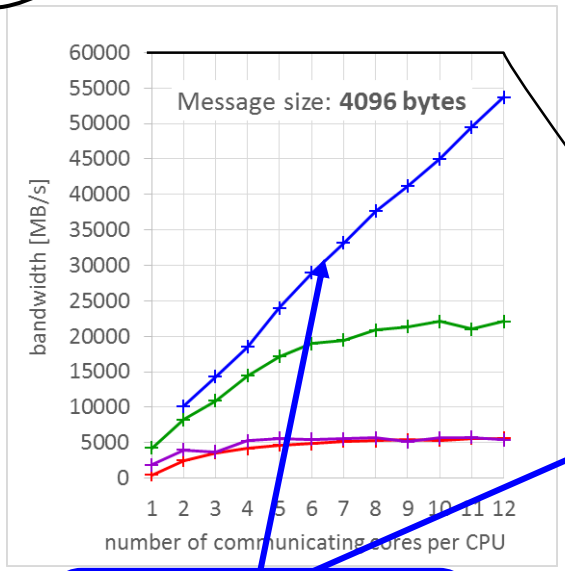


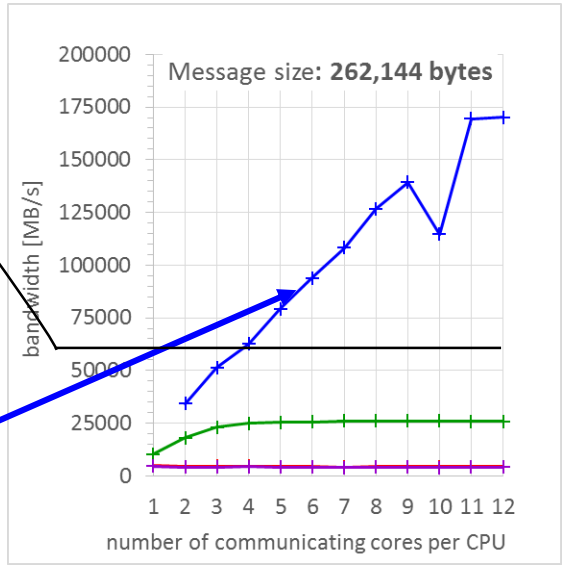


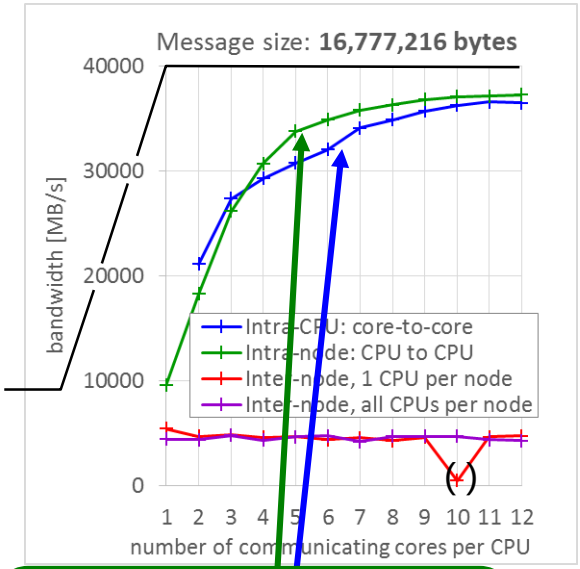
Core-to-core:
Linear scaling for small to medium size messages due to caches

Accumulated – scaling vs. asymptotic behavior

skipped





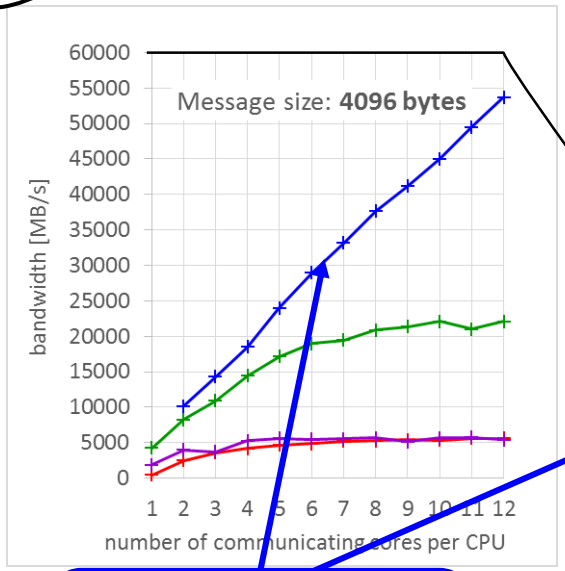


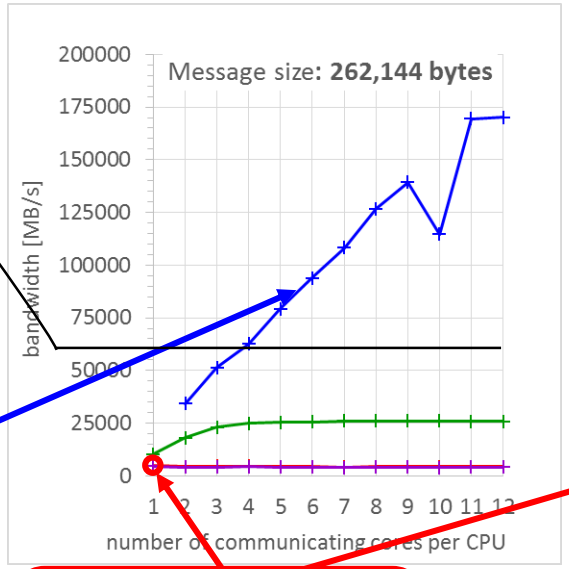
Core-to-core:
Linear scaling for small to medium size messages due to caches

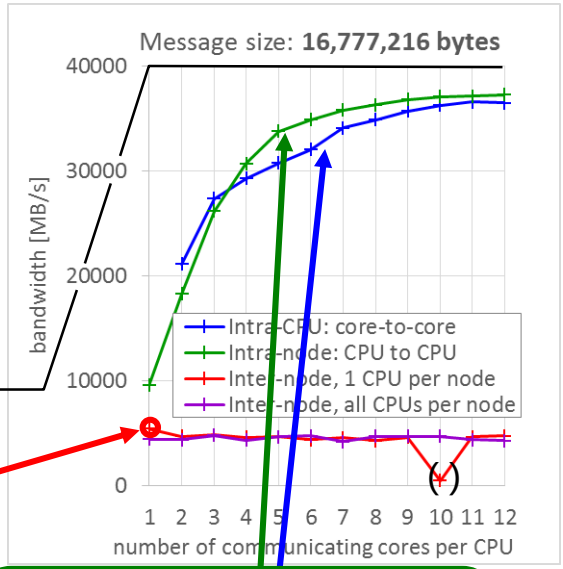
Core-to-core & CPU-to-CPU:
Long messages:
Same asymptotic limit through **memory bandwidth**

Accumulated – scaling vs. asymptotic behavior

skipped







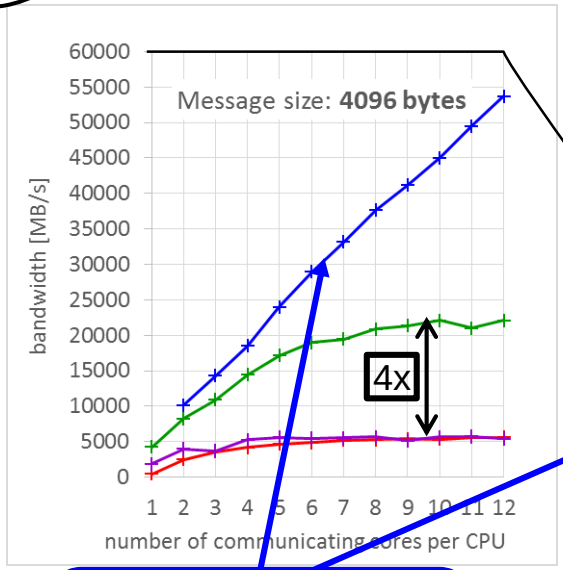
Core-to-core:
Linear scaling for small to medium size messages due to caches

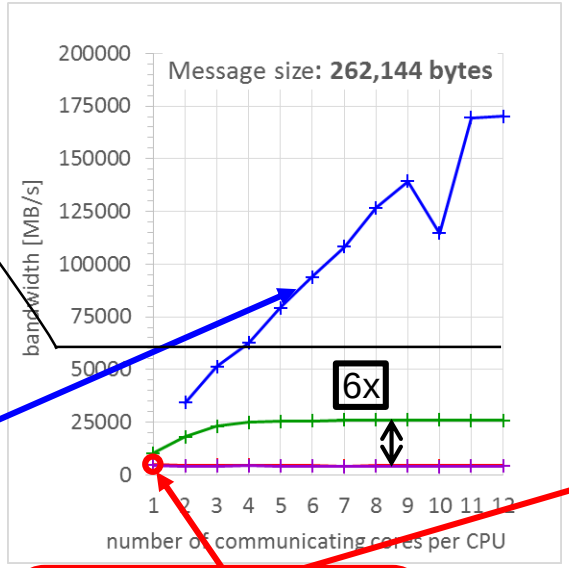
Node-to-node:
One duplex link by **one core** already fully saturates the network

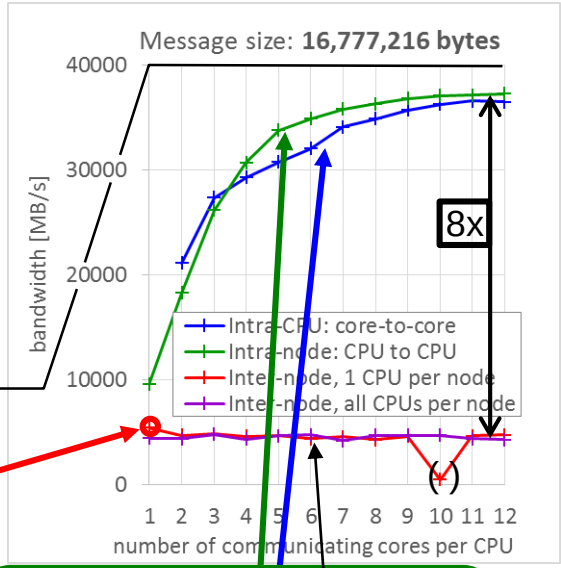
Core-to-core & CPU-to-CPU:
Long messages:
Same asymptotic limit through **memory bandwidth**

Accumulated – scaling vs. asymptotic behavior

skipped







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**

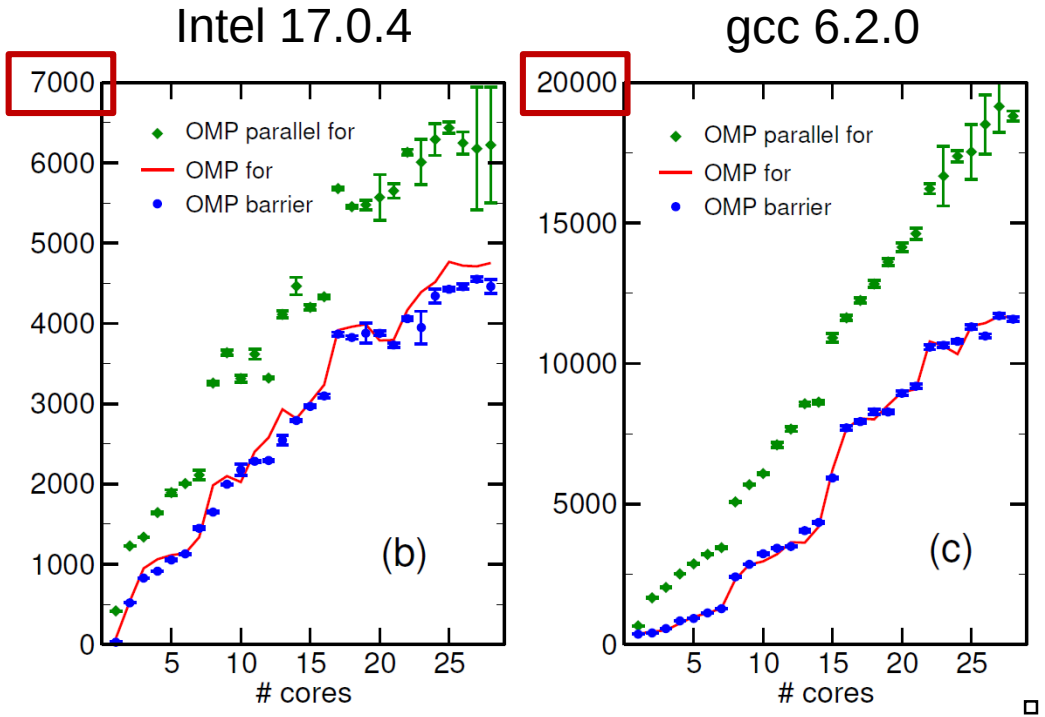
Result: The limit of accumulated **intra-CPU** and **intra-node** bandwidth is **8x larger than** the limit of accumulated **node-to-node** bandwidth

OpenMP barrier synchronization cost

Comparison of barrier synchronization cost with increasing number of threads

- 2x Haswell 14-core (CoD mode)
- Optimistic measurements (repeated 1000s of times)
- No impact from previous activity in cache

→ Barrier sync time highly dependent on system topology & OpenMP runtime implementation

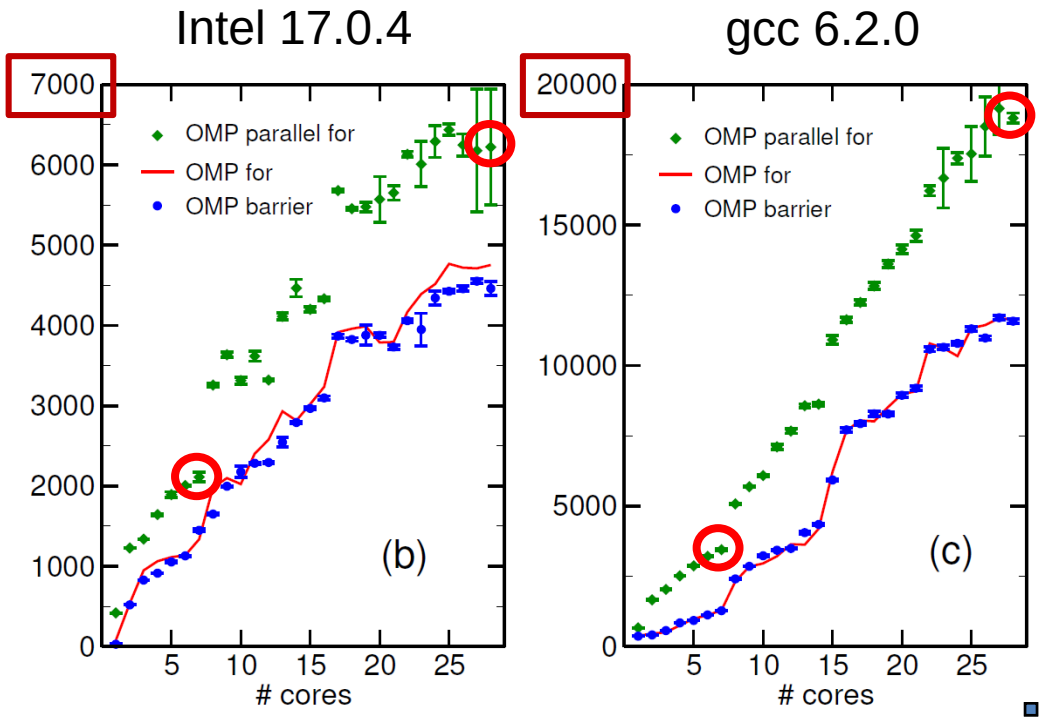


OpenMP barrier synchronization cost

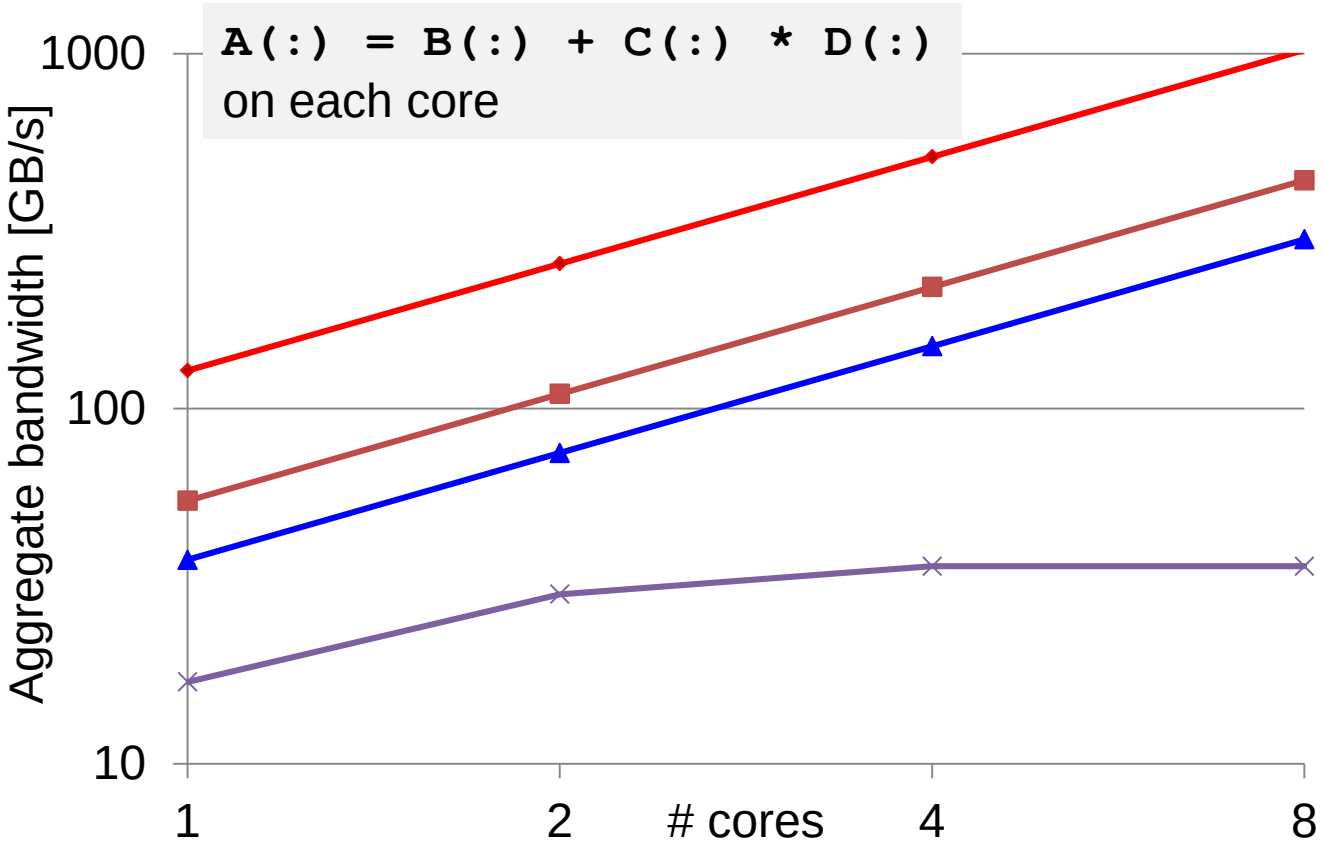
Comparison of barrier synchronization cost with increasing number of threads

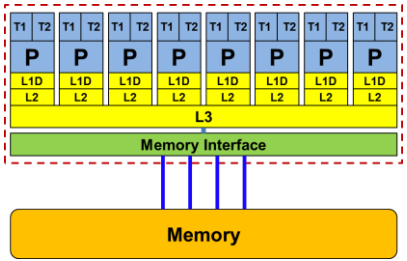
- 2x Haswell 14-core (CoD mode)
- Optimistic measurements (repeated 1000s of times)
- No impact from previous activity in cache

→ Barrier sync time highly dependent on system topology & OpenMP runtime implementation



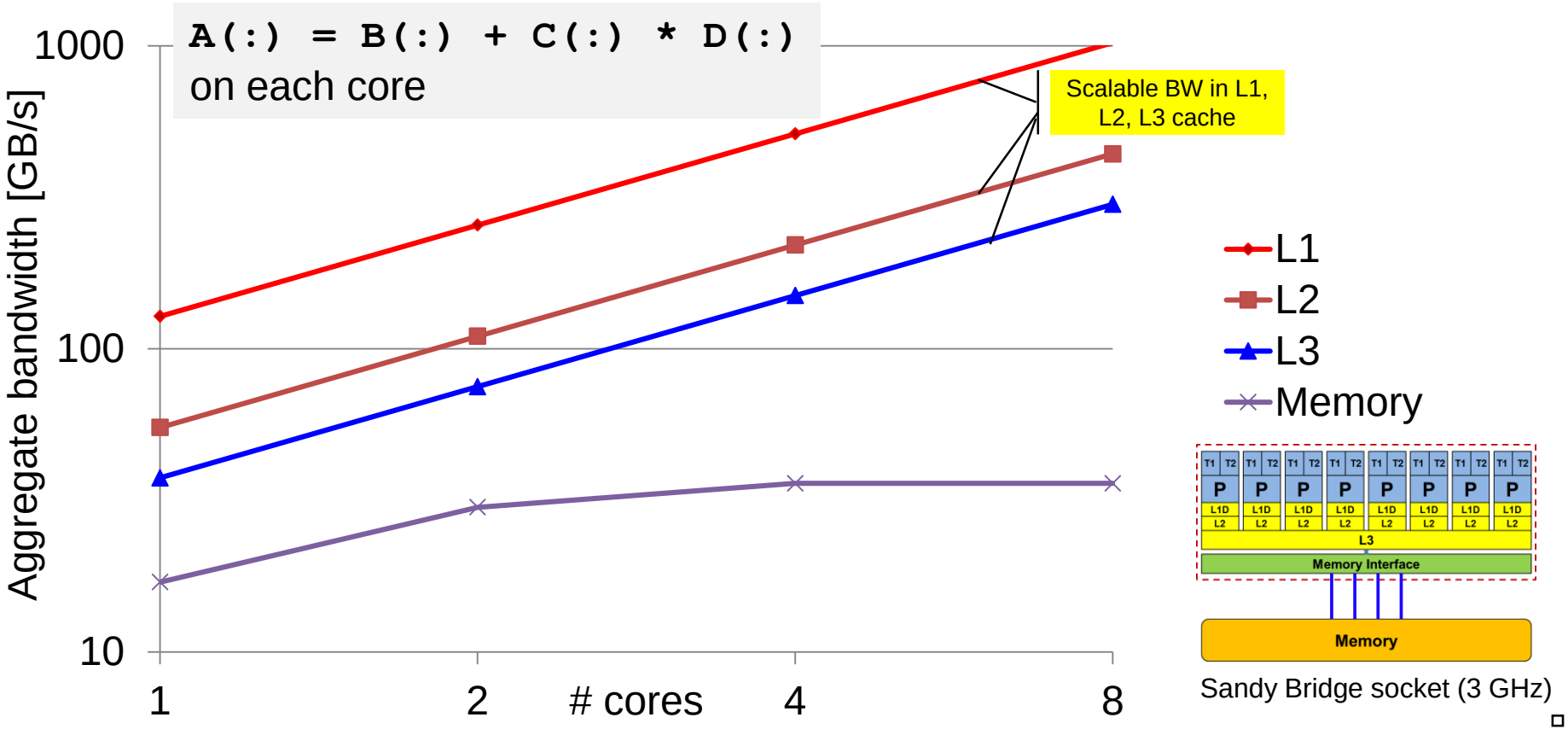
Accumulated bandwidth saturation vs. # cores



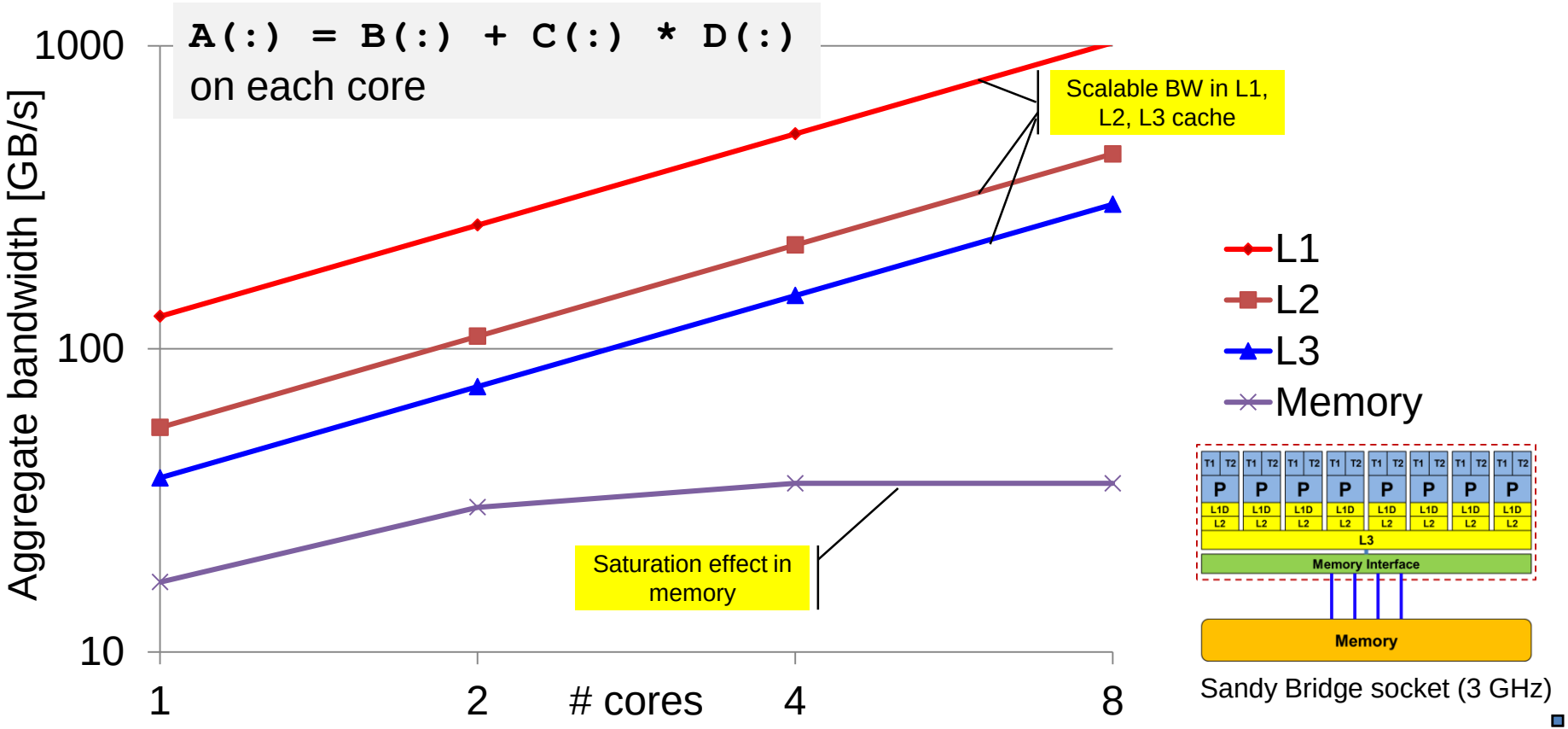


Sandy Bridge socket (3 GHz)

Accumulated bandwidth saturation vs. # cores



Accumulated bandwidth saturation vs. # cores



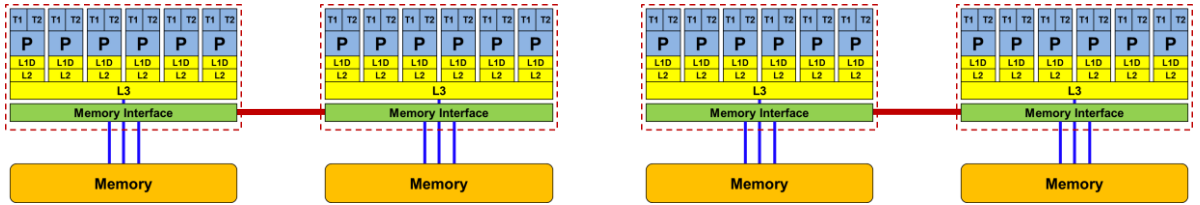
Programming models - MPI + OpenMP

Memory placement on ccNUMA systems

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
> **Memory placement on ccNUMA systems**
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
  Communication overlap with OpenMP taskloops
  Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

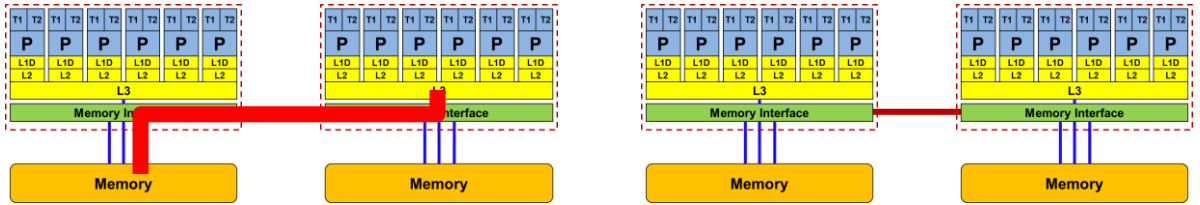
A short introduction to ccNUMA

- 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)



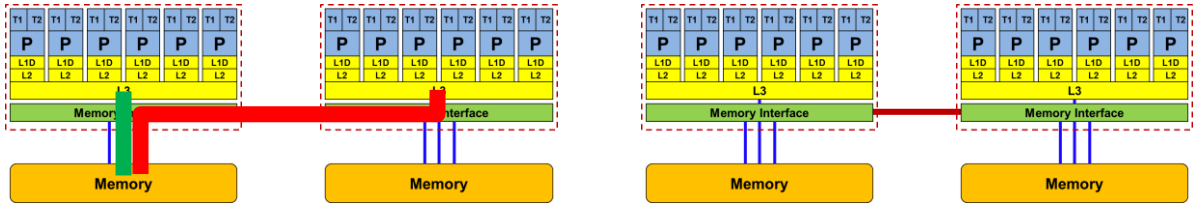
A short introduction to ccNUMA

- 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)



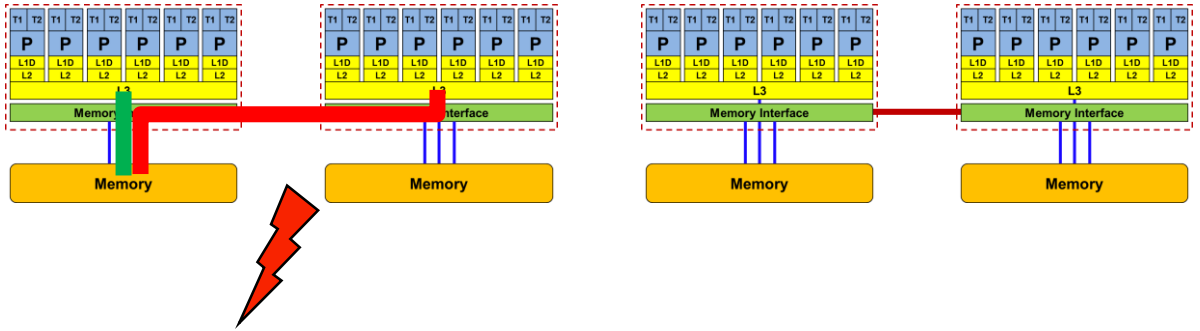
A short introduction to ccNUMA

- 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)



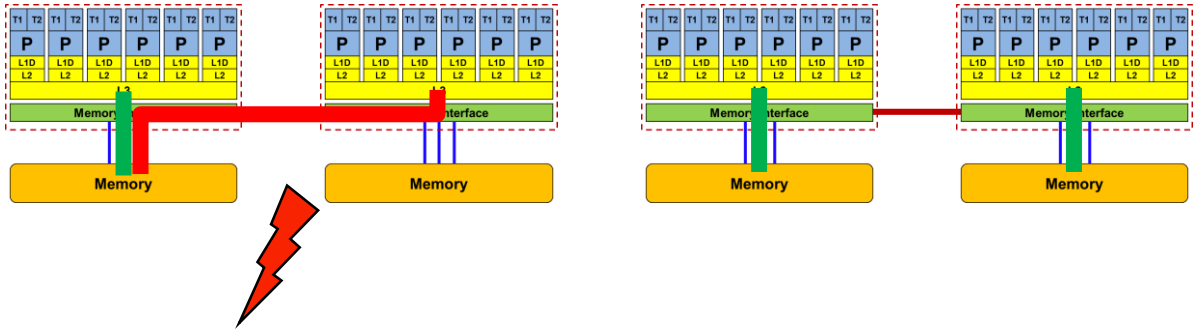
A short introduction to ccNUMA

- 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)



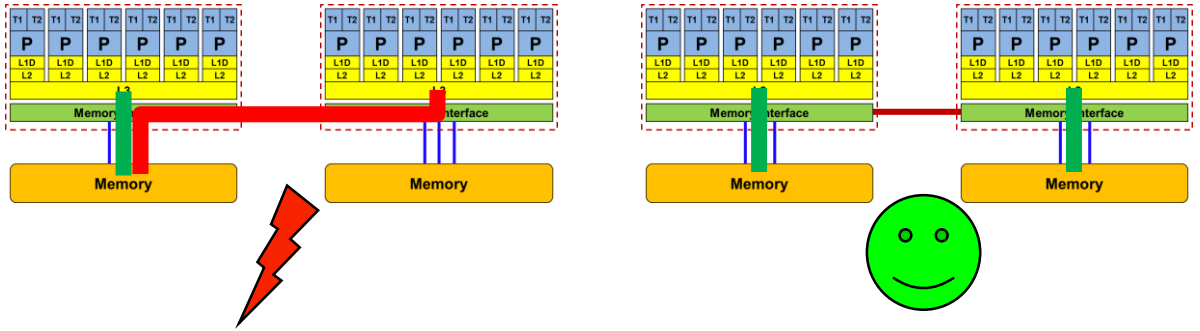
A short introduction to ccNUMA

- 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)



A short introduction to ccNUMA

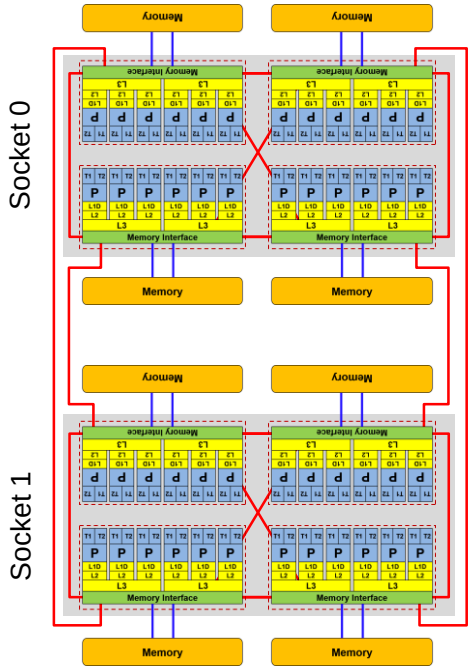
- ccNUMA:
  - whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
  - Memory placement occurs with OS page granularity (often 4 KiB)



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

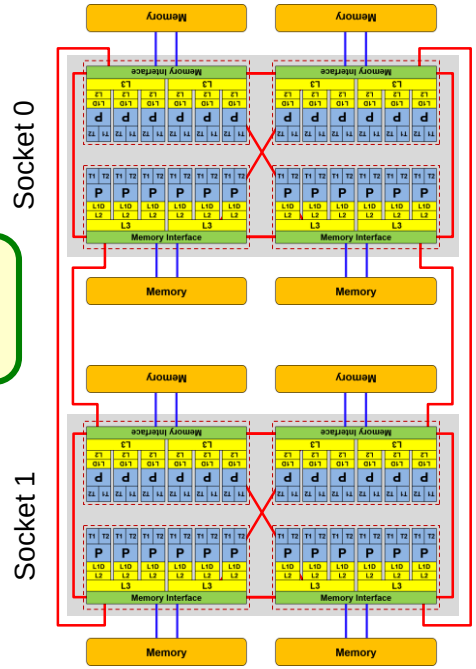


How much bandwidth does non-local access cost?

- 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	10.6	10.6
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

Highest bandwidth between memory and cores of one **NUMA domain**



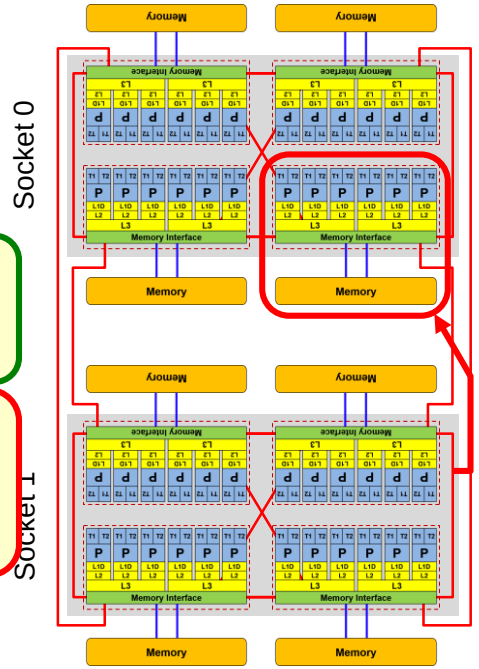
How much bandwidth does non-local access cost?

- 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	10.6	10.6
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

Highest bandwidth between memory and cores of one **NUMA domain**

Do you want to run your application 3 times slower?
(If your appl. is memory bandwidth bound)



Avoiding locality problems

- How can we make sure that memory ends up where it is close to the CPU that uses it?
  - See next slides (first-touch initialization)
- How can we make sure that it stays that way throughout program execution?
  - See later in the tutorial (pinning)
- 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, memory mapping is not a problem

- Automatic page migration may help if memory is used long enough

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” or “read”

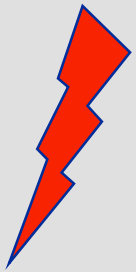
Solving Memory Locality Problems: First Touch

- "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” or “read”
- 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!
```

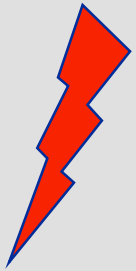
Most simple case: explicit initialization

```
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
```



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
```

Handling ccNUMA in practice

- Solution A
  - One (or more) MPI process(es) per ccNUMA domain
  - Pro: optimal page placement (perfectly local memory access) for free
  - Con: higher number (>1) of MPI processes on each node

Handling ccNUMA in practice

- Solution A
  - One (or more) MPI process(es) per ccNUMA domain
  - Pro: optimal page placement (perfectly local memory access) for free
  - Con: higher number (>1) of MPI processes on each node
- Solution B
  - One MPI process per node or one MPI process spans multiple ccNUMA domains
  - Pro: Smaller number of MPI processes compared to Solution A
  - Cons:
    - Explicitly parallel initialization needed to “bind” the data to each ccNUMA domain → otherwise loss of performance
    - Dynamic/guided schedule or tasking → loss of performance

Handling ccNUMA in practice

- Solution A
  - One (or more) MPI process(es) per ccNUMA domain
  - Pro: optimal page placement (perfectly local memory access) for free
  - Con: higher number (>1) of MPI processes on each node
- Solution B
  - One MPI process per node or one MPI process spans multiple ccNUMA domains
  - Pro: Smaller number of MPI processes compared to Solution A
  - Cons:
    - Explicitly parallel initialization needed to “bind” the data to each ccNUMA domain → otherwise loss of performance
    - Dynamic/guided schedule or tasking → loss of performance
- Thread binding is mandatory for A and B! – Never trust the defaults!

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

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
- Know your software characteristics
  - Typical numbers for communication latencies, bandwidths
  - Typical OpenMP overheads

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
- Know your software characteristics
  - Typical numbers for communication latencies, bandwidths
  - Typical OpenMP overheads
- Learn how to take control
  - See next chapter on affinity control
- Leveraging topology effects is a part of code optimization!

Programming models - MPI + OpenMP

Topology and affinity on multicore

General considerations
How to compile, link, and run
Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
Memory placement on ccNUMA systems
> **Topology and affinity on multicore**
Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
Communication overlap with OpenMP taskloops
Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

Thread/Process Affinity (“Pinning”)

- Highly OS-dependent system calls
  - But available on all OSs
  - Non-portable

Thread/Process Affinity (“Pinning”)

- Highly OS-dependent system calls
  - But available on all OSs
  - Non-portable
- Support for user-defined pinning for OpenMP threads in all compilers
  - Compiler specific
  - Standardized in OpenMP (places)
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`**

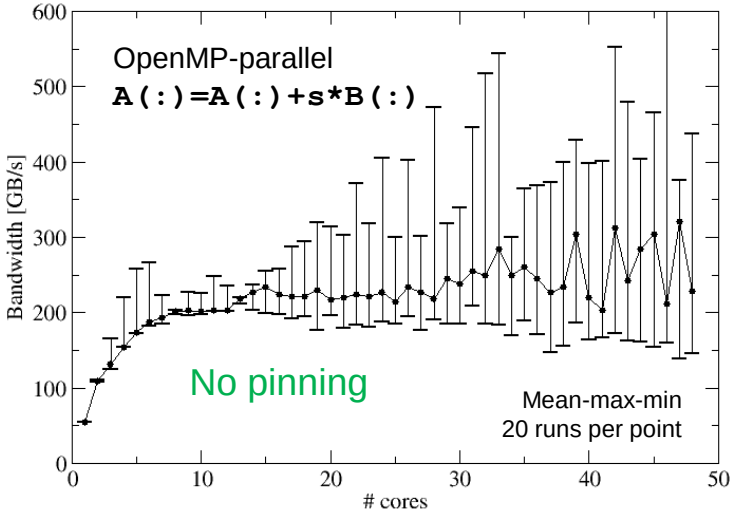
Thread/Process Affinity (“Pinning”)

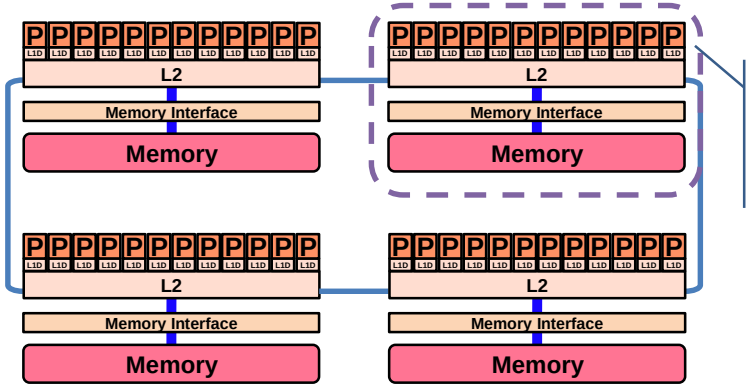
- Highly OS-dependent system calls
  - But available on all OSs
  - Non-portable
- Support for user-defined pinning for OpenMP threads in all compilers
  - Compiler specific
  - Standardized in OpenMP (places)
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`**
- Affinity awareness in all MPI libraries
  - Not defined by the MPI standard (as of 4.0)
  - Necessarily non-portable feature of the startup mechanism (mpirun, …)

Thread/Process Affinity (“Pinning”)

- Highly OS-dependent system calls
  - But available on all OSs
  - Non-portable
- Support for user-defined pinning for OpenMP threads in all compilers
  - Compiler specific
  - Standardized in OpenMP (places)
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`**
- Affinity awareness in all MPI libraries
  - Not defined by the MPI standard (as of 4.0)
  - Necessarily non-portable feature of the startup mechanism (mpirun, …)
- Affinity awareness in batch scheduler
  - Batch scheduler must work with MPI + OpenMP affinity
  - Difficult, non-portable, every combination is different

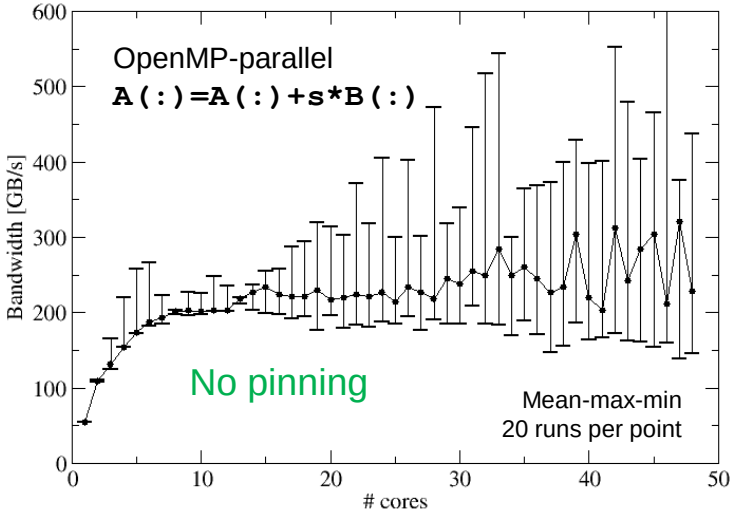
Anarchy vs. affinity with OpenMP STREAM

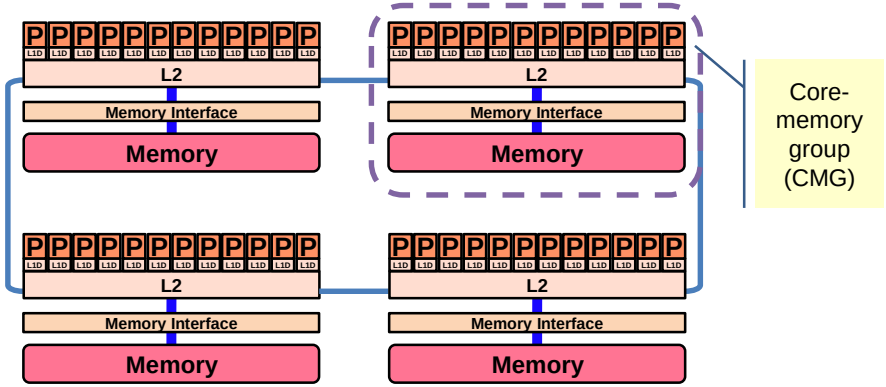




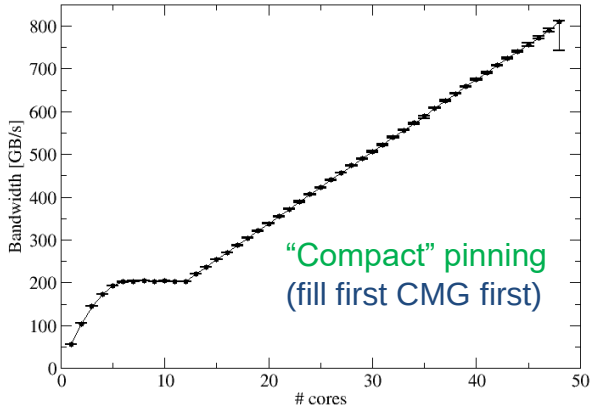
Fujitsu A64FX CPU

Anarchy vs. affinity with OpenMP STREAM

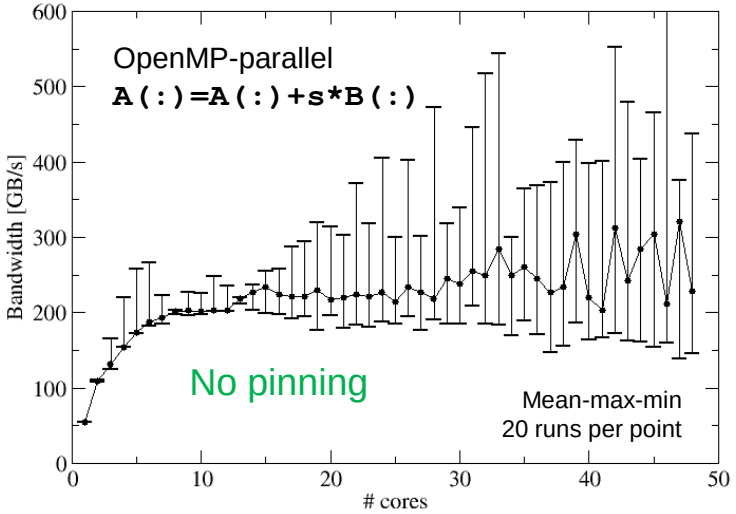


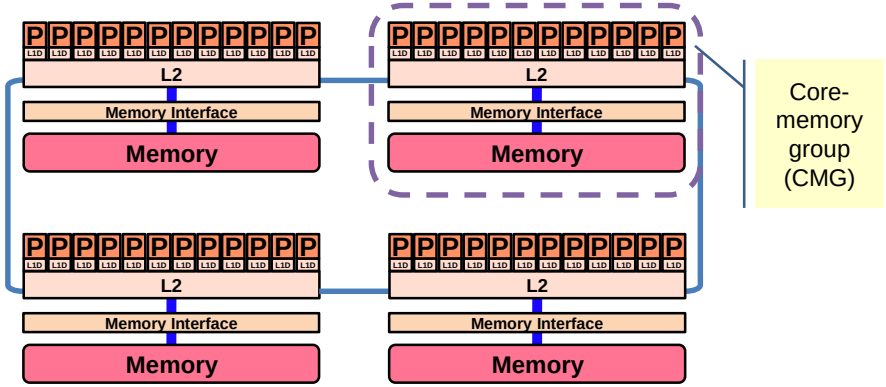


Fujitsu A64FX CPU

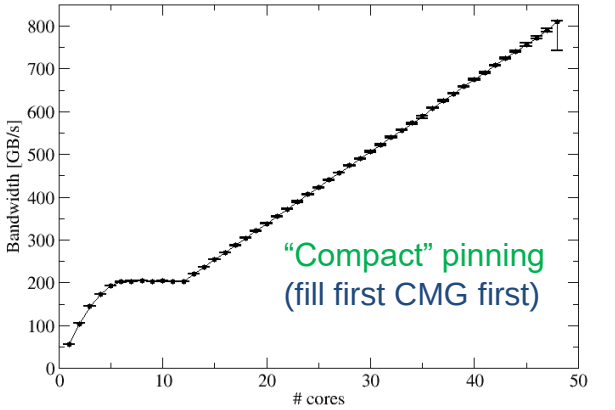


Anarchy vs. affinity with OpenMP STREAM





Fujitsu A64FX CPU



There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

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

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

- **OMP_PLACES=threads** *abstract_name*
  - → Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)
- **OMP_PLACES=cores**
  - → Each place corresponds to the processors (one or more hardware threads) of a single core
- **OMP_PLACES=sockets**
  - → Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)
- **OMP_PLACES=*abstract_name(num_places)***
  - → In general, the number of places may be explicitly defined

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

- **OMP_PLACES=threads** *abstract_name*
  → Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)
- **OMP_PLACES=cores**
  → Each place corresponds to the processors (one or more hardware threads) of a single core
- **OMP_PLACES=sockets**
  → Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)
- **OMP_PLACES=*abstract_name(num_places)***
  → In general, the number of places may be explicitly defined

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

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

- **OMP_PLACES=threads** (*abstract_name*)
  → Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)
- **OMP_PLACES=cores**
  → Each place corresponds to the processors (one or more hardware threads) of a single core
- **OMP_PLACES=sockets**
  → Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)
- **OMP_PLACES=*abstract_name(num_places)***
  → In general, the number of places may be explicitly defined
- Or with explicit numbering, e.g. 8 places, each consisting of 4 processors:
  - `setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11}, … {28,29,30,31}"`
  - `setenv OMP_PLACES "{0:4},{4:4},{8:4}, … {28:4}"`
  - `setenv OMP_PLACES "{0:4}:8:4"`

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

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

- **OMP_PLACES=threads** (*abstract_name*)
  → Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)
- **OMP_PLACES=cores**
  → Each place corresponds to the processors (one or more hardware threads) of a single core
- **OMP_PLACES=sockets**
  → Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)
- **OMP_PLACES=*abstract_name(num_places)***
  → In general, the number of places may be explicitly defined
- Or with explicit numbering, e.g. 8 places, each consisting of 4 processors:
  - `setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11}, … {28,29,`
  - `setenv OMP_PLACES "{0:4},{4:4},{8:4}, … {28:4}"`
  - `setenv OMP_PLACES "{0:4}:8:4"`

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

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

Some simple OMP_PLACES examples

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x36 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=36
OMP_PLACES=cores
OMP_PROC_BIND=close
```

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x36 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=36
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
```

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x36 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=36
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
```

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x36 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=36
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
```

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x36 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=36
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

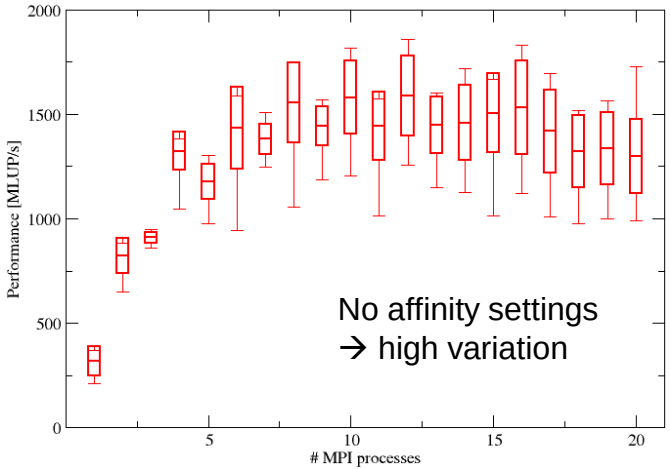
```
OMP_NUM_THREADS=8
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

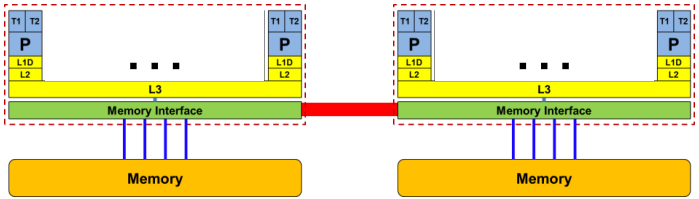
Always prefer abstract places instead of HW thread IDs!

Pinning of MPI processes

- 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)

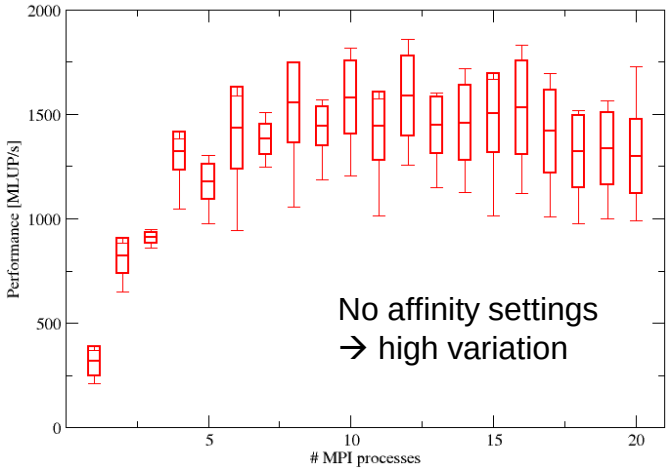
Anarchy vs. affinity with a heat equation solver

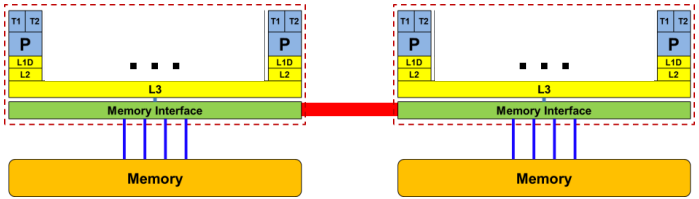




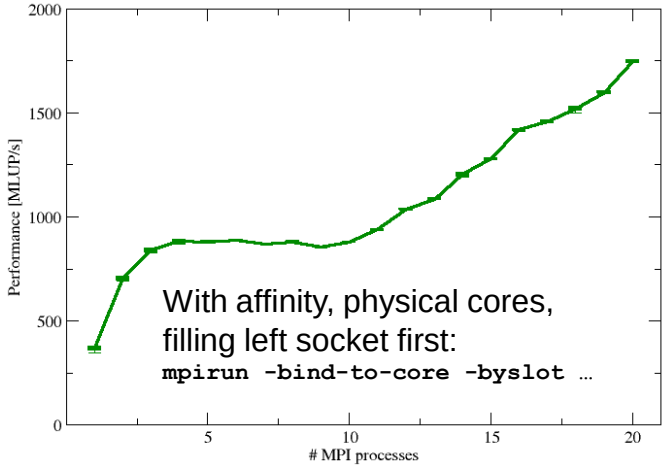
2x 10-core Intel Ivy Bridge, OpenMPI

Anarchy vs. affinity with a heat equation solver

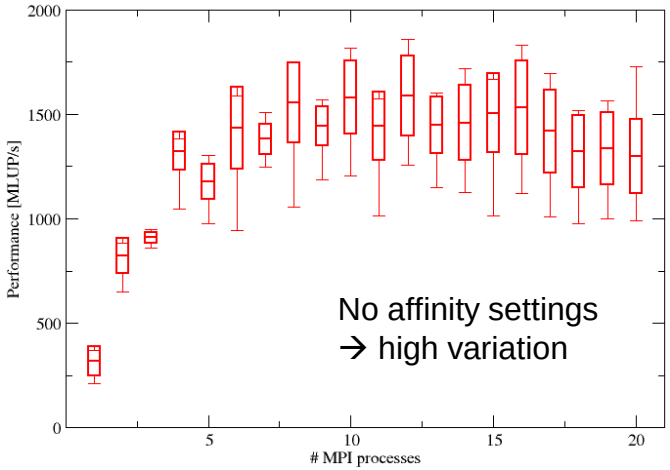


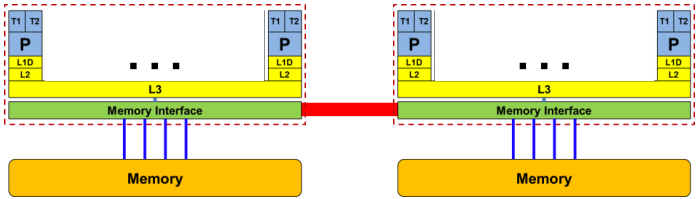


2x 10-core Intel Ivy Bridge, OpenMPI

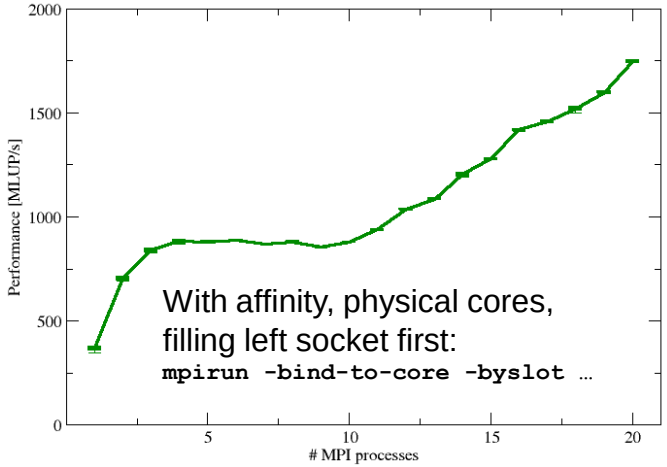


Anarchy vs. affinity with a heat equation solver





2x 10-core Intel Ivy Bridge, OpenMPI

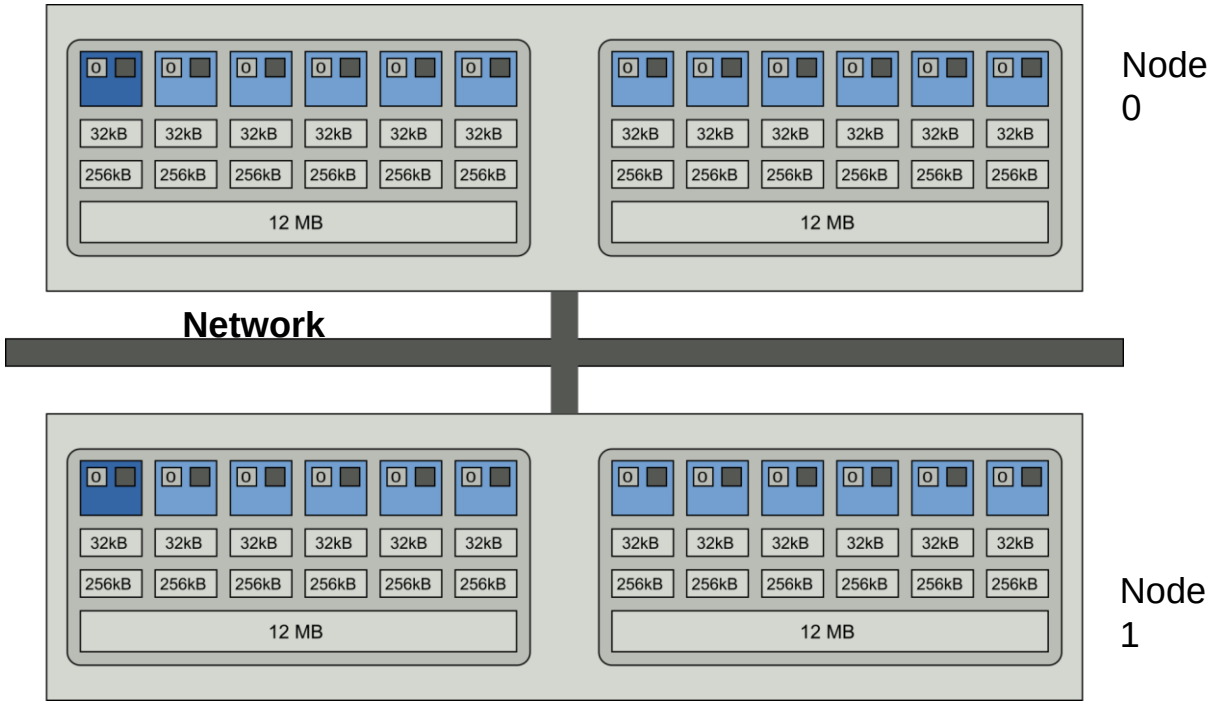


Reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

likwid-mpirun: *1 MPI process per node*

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

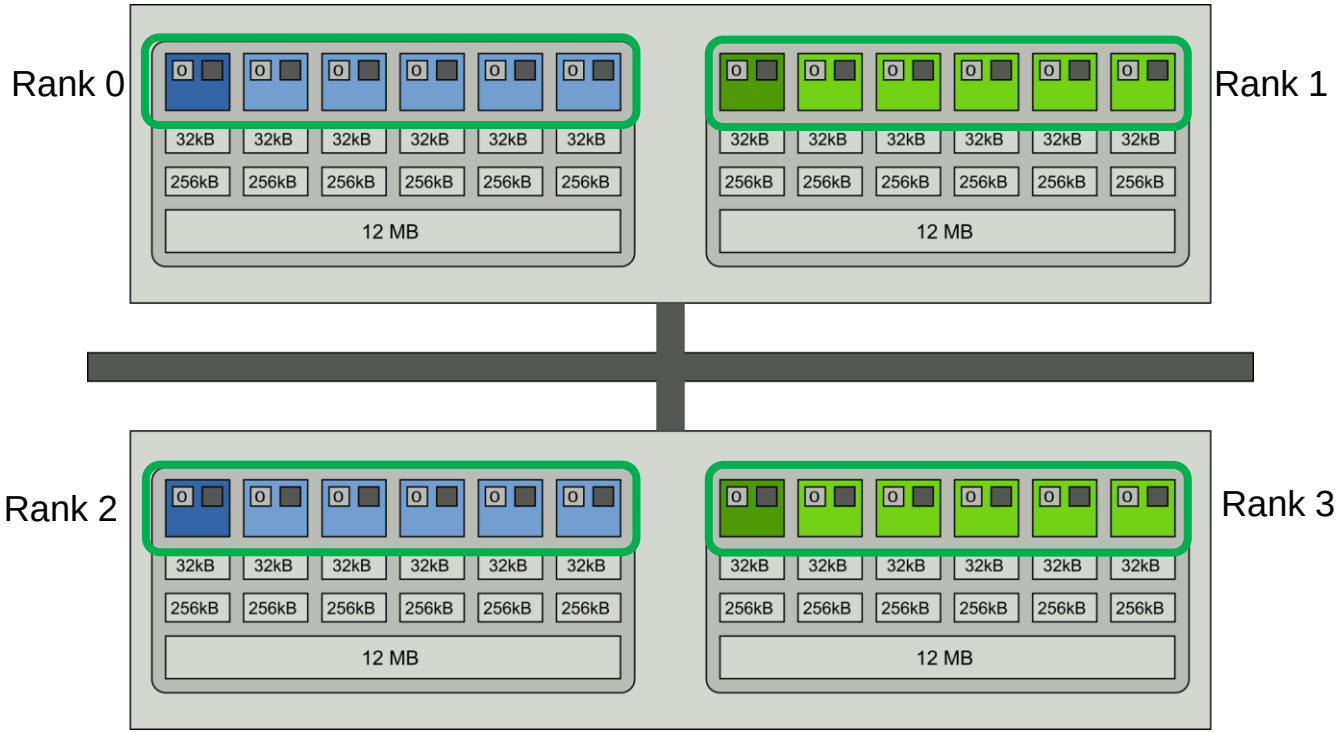


Intel MPI+compiler:

```
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 affinity: 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)

MPI/OpenMP affinity: 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 performance
  - OpenMP overheads
  - Saturation effects / scalability behavior with bandwidth-bound code

MPI/OpenMP affinity: 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 performance
  - 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)

MPI/OpenMP affinity: 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 performance
  - 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)
- Memory page placement on ccNUMA nodes
  - Automatic optimal page placement for one (or more) MPI processes per ccNUMA domain (solution A)
  - Explicitly parallel first-touch initialization only required for multi-domain MPI processes (solution B)

Programming models - MPI + OpenMP

Hands-On #2

Pinning

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
  Memory placement on ccNUMA systems
Topology and affinity on multicore
> **Hands-on: Pinning**
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
  Communication overlap with OpenMP taskloops
  Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

Programming models - MPI + OpenMP

Case study: The Multi-Zone NAS Parallel Benchmarks

General considerations
How to compile, link, and run
Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
Memory placement on ccNUMA systems
Topology and affinity on multicore
Hands-on: Pinning
> **Case study: The Multi-Zone NAS Parallel Benchmarks**
Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
Communication overlap with OpenMP taskloops
Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

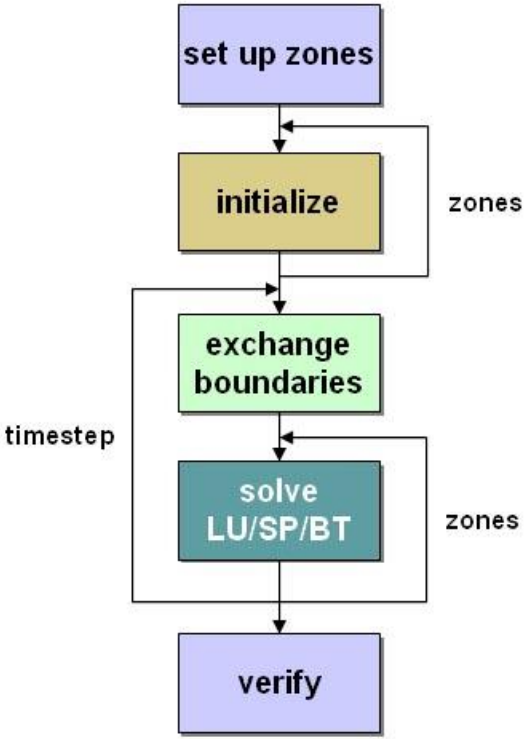
Load Balancing with hybrid programming

- On same or different level of parallelism
- OpenMP enables
  - cheap ***dynamic*** and ***guided*** load-balancing
  - via 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 → rarely implemented
- MPI & OpenMP
  - Simple static load balancing on MPI level, dynamic or guided on OpenMP level } medium-quality, cheap implementation

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 structure

```
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
  ...
```

call mpi_send/recv

```
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

Benchmark Characteristics

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

Expectations:

- 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

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

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

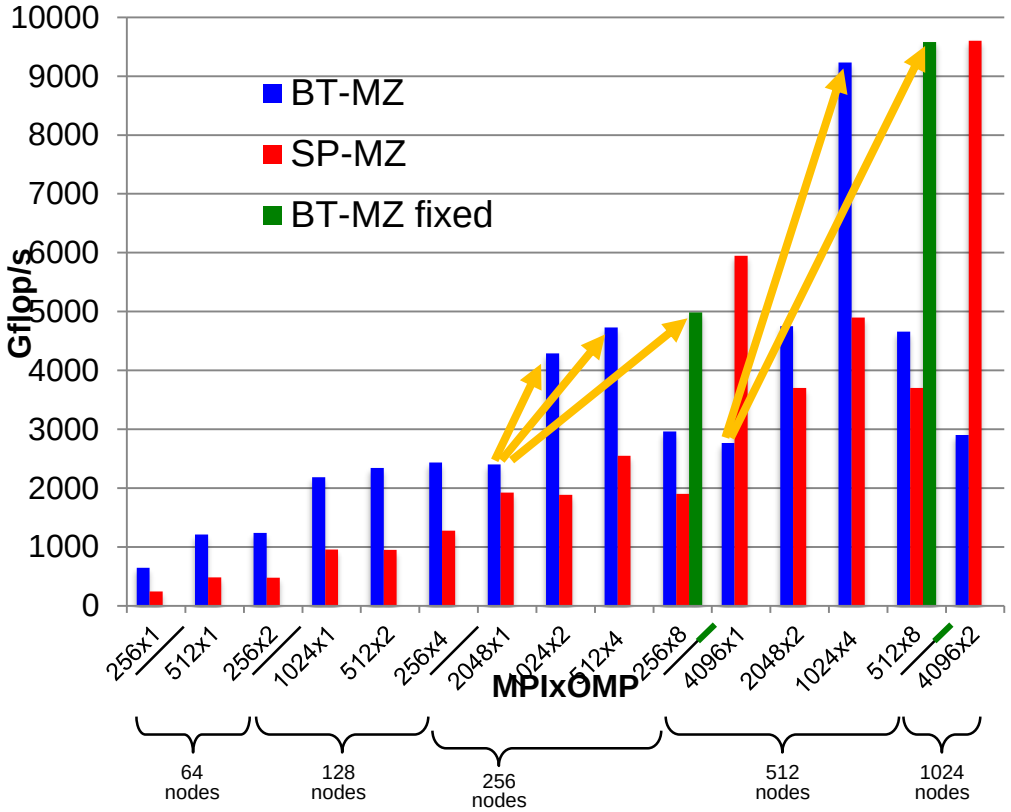
Expectations:

Pure MPI: Load-balancing problems!

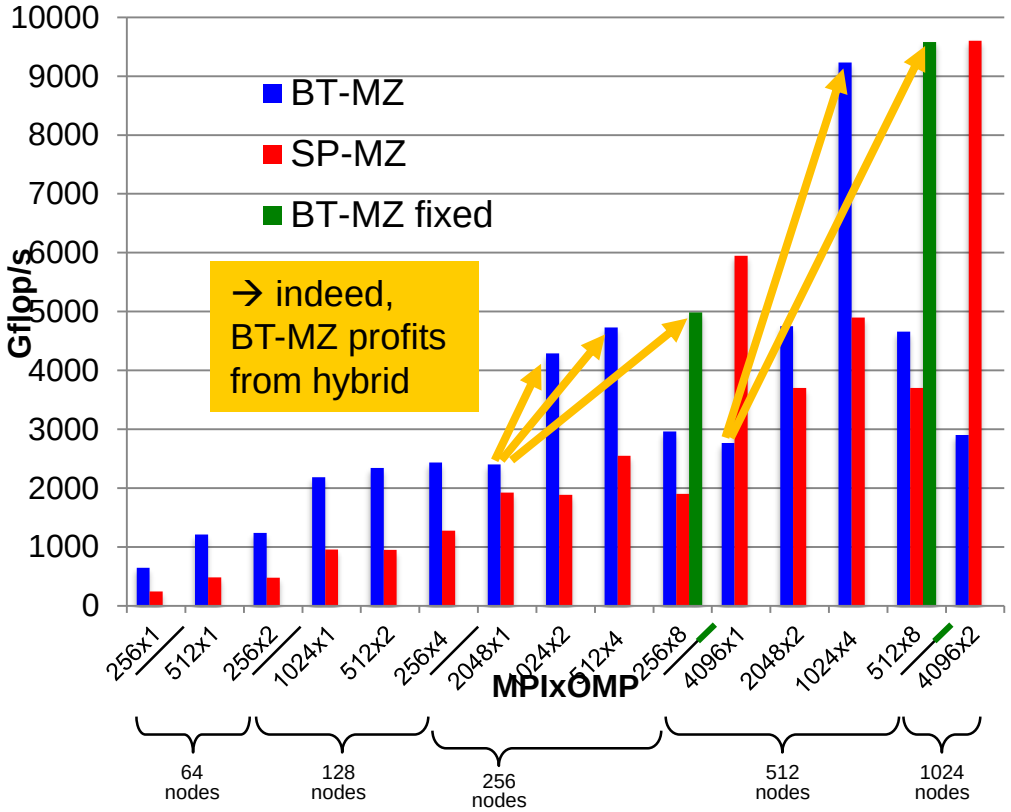
Good candidate for MPI+OpenMP

Load-balanced on MPI level: Pure MPI should perform best

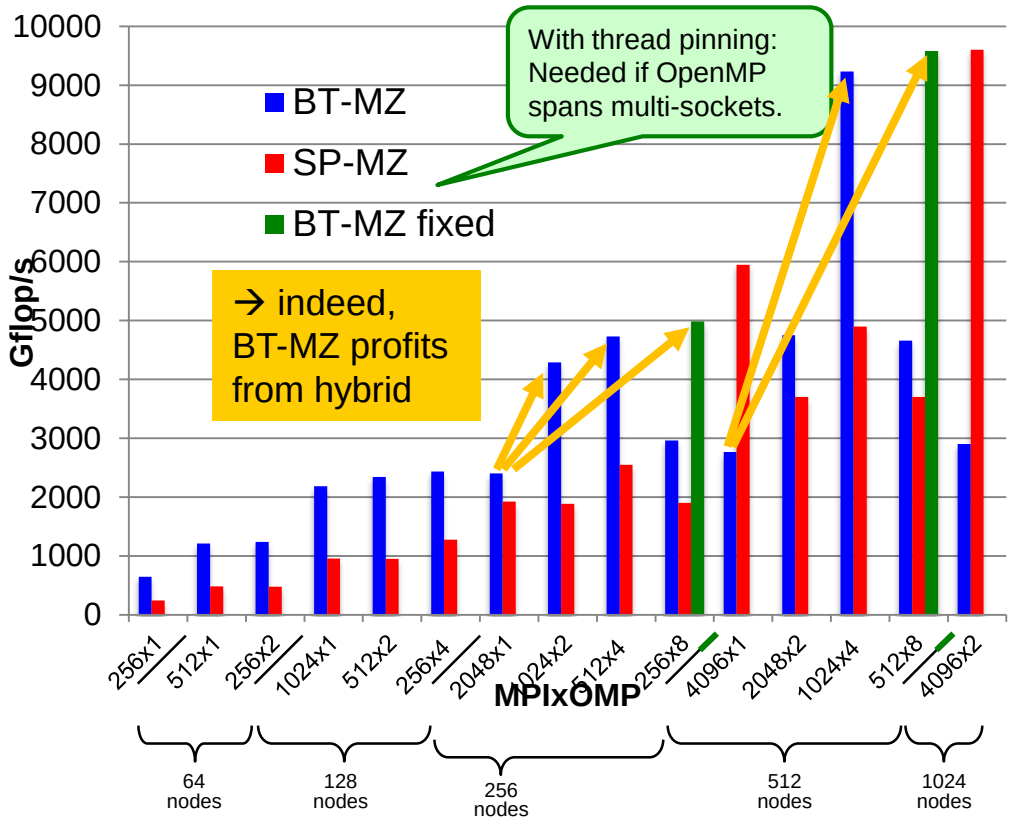
NPB-MZ Class E Scalability on Lonestar



NPB-MZ Class E Scalability on Lonestar



NPB-MZ Class E Scalability on Lonestar



Programming models - MPI + OpenMP

Hands-On #3

Masteronly hybrid Jacobi

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
  Memory placement on ccNUMA systems
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
> **Hands-on: Masteronly hybrid Jacobi**
Overlapping communication and computation
  Communication overlap with OpenMP taskloops
  Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

Example: MPI+OpenMP-Hybrid Jacobi solver

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

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.

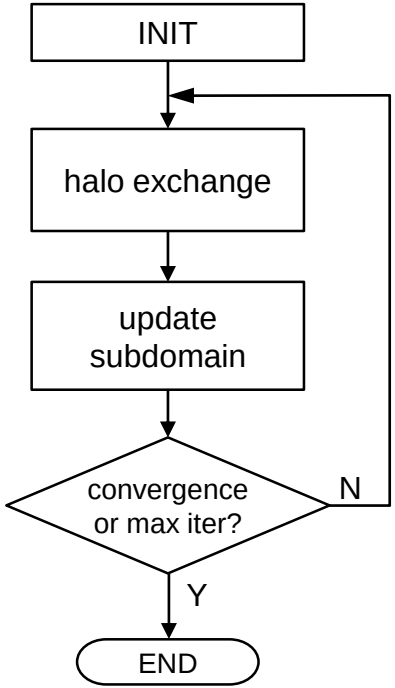
- Notes:
  - The code is strongly memory bound at the problem size set in the input file
  - Learn how to take control of affinity with MPI and especially with MPI+OpenMP
  - Always run multiple times and observe performance variations
  - If you know how, try to calculate the maximum possible performance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

http://tiny.cc/MPIX-VSC

http://tiny.cc/MPIX-LRZ 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,…,N_c,…,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,…,N_c,…,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-VSC

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

see also login-slides

Programming models - MPI + OpenMP

Overlapping Communication and Computation

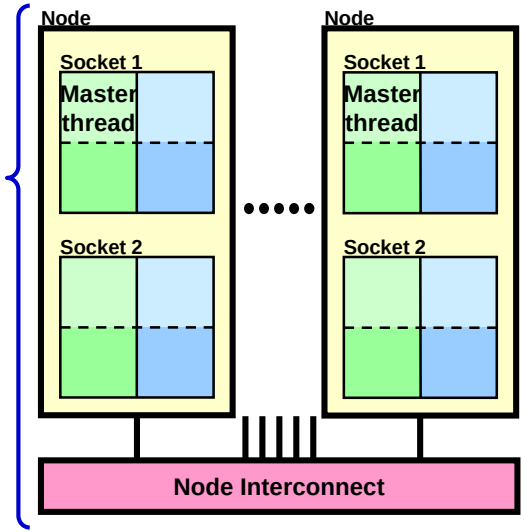
General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
  Memory placement on ccNUMA systems
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
> **Overlapping communication and computation**
  Communication overlap with OpenMP taskloops
  Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

Sleeping threads with masteronly style

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

  /* on master only */
    MPI_Send(halos);
    MPI_Recv(halos);
} /*end for loop*/
```

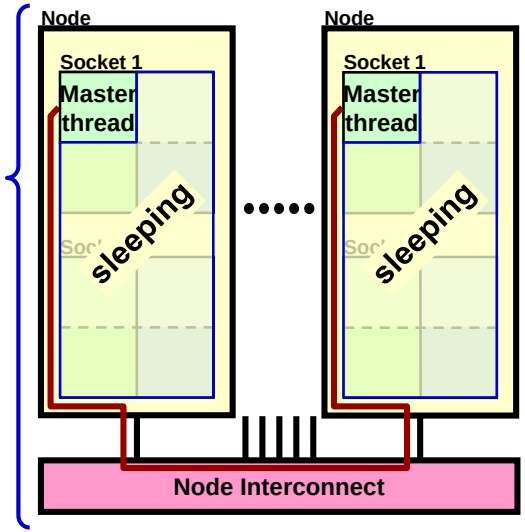
Sleeping threads with masteronly style

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

  /* on master only */
    MPI_Send(halos);
    MPI_Recv(halos);
} /*end for loop*/
```



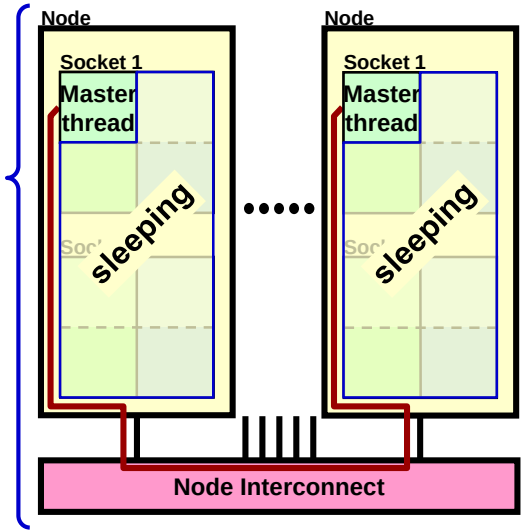
Sleeping threads with masteronly style

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

  /* on master only */
    MPI_Send(halos);
    MPI_Recv(halos);
} /*end for loop*/
```



Sleeping threads with masteronly style

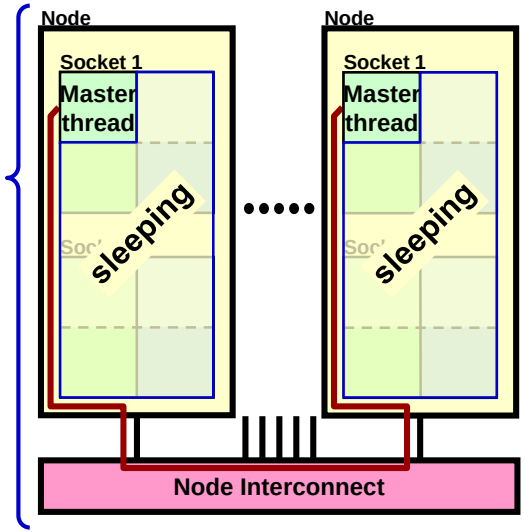
```
for (iteration ….)
{
  #pragma omp parallel
     numerical code
  /* end parallel */

  /* on master only */
    MPI_Send(halos);
    MPI_Recv(halos);
} /*end for loop*/
```



- Problem:
  - Sleeping threads are wasting CPU time

Sleeping threads with masteronly style

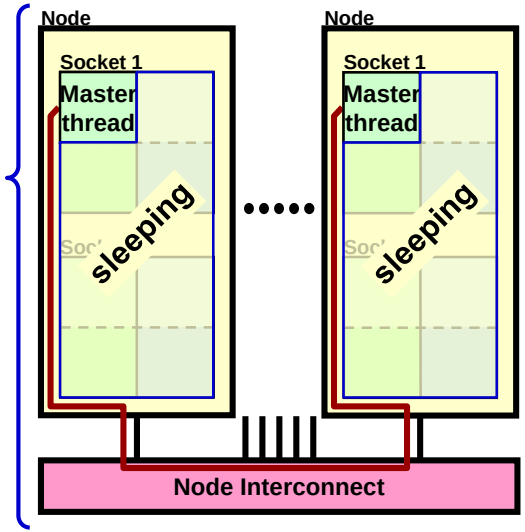
```
for (iteration ….)
{
  #pragma omp parallel
     numerical code
  /* end parallel */

  /* on master only */
    MPI_Send(halos);
    MPI_Recv(halos);
} /*end for loop*/
```



- Problem:
  - Sleeping threads are wasting CPU time
- Solution:
  - Overlapping of computation and communication

Sleeping threads with masteronly style

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

  /* on master only */
    MPI_Send(halos);
    MPI_Recv(halos);
} /*end for loop*/
```

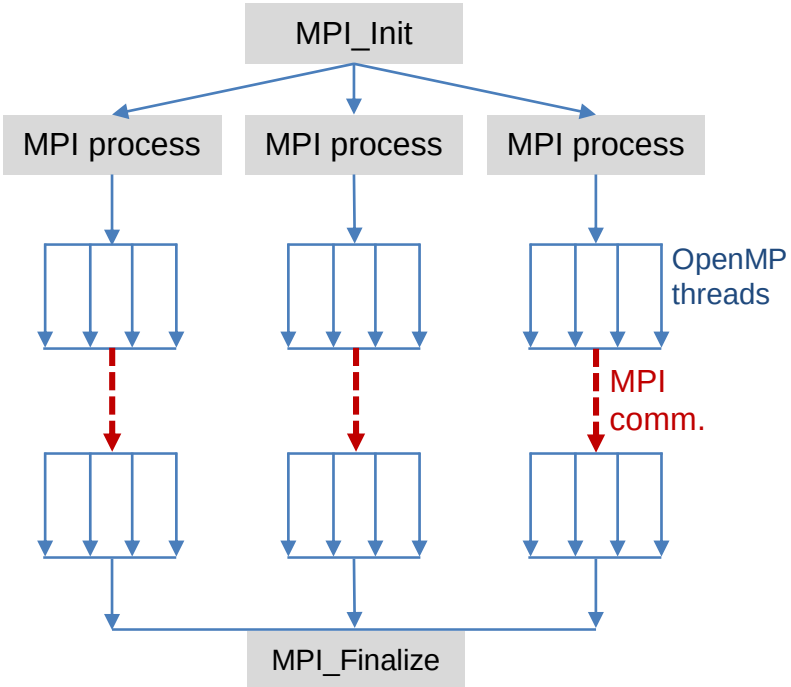


- Problem:
  - Sleeping threads are wasting CPU time
- Solution:
  - Overlapping of computation and communication
- Limited benefit:
  - Best case: reduces communication overhead from 50% to 0%
    → speedup of 2x
  - Usual case of 20% to 0%
    → speedup of 1.25x
  - Requires significant work → later

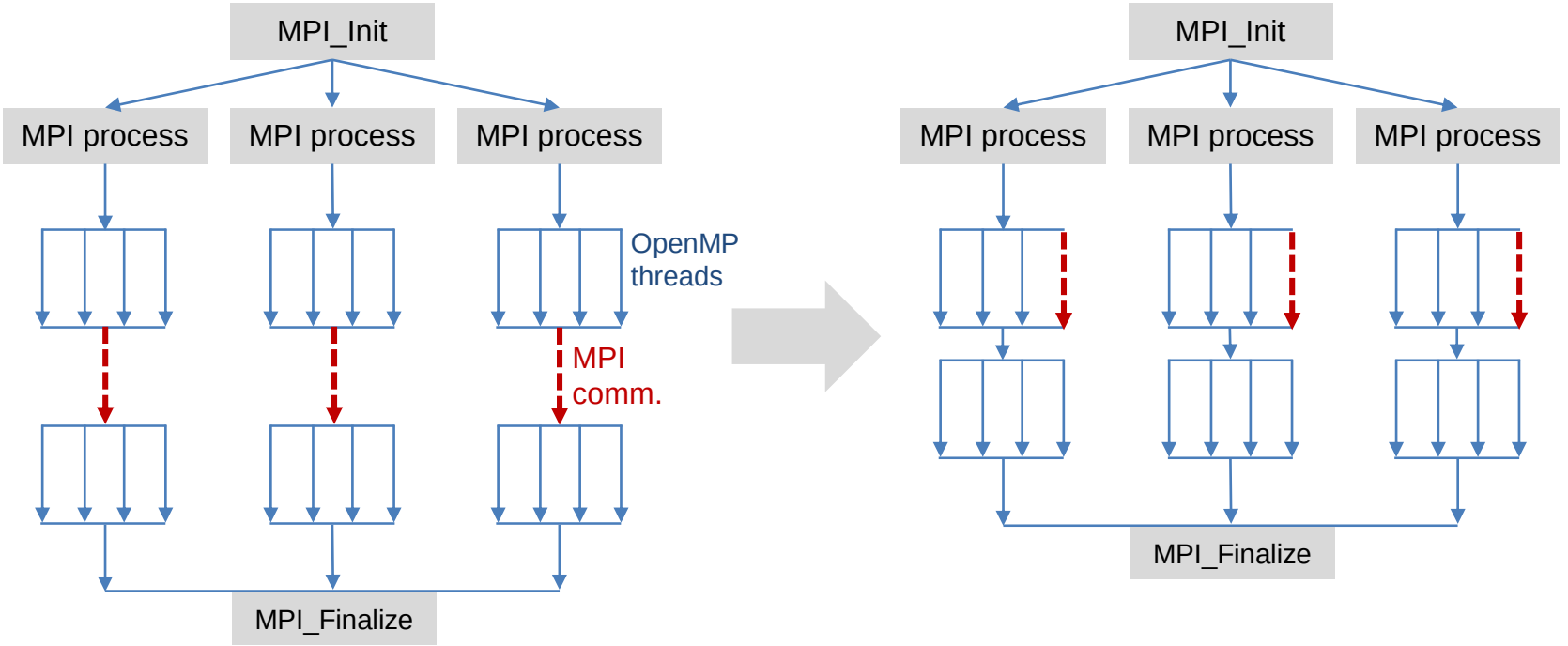
Nonblocking vs. threading for overlapped comm.

- Why not use nonblocking calls?
  - Asynchronous progress not guaranteed
  - Options (implementation dependent):
    - Communication offload to NIC
    - Additional internal progress thread (MPI_ASYNC… with MPICH)
  - Intranode and internode communication may be handled very differently
- Using threading for communication overlap
  - One or more threads/tasks handles communication, rest of team “do the work”
  - How to organize the work sharing among all threads?
    - Non-communicating threads
    - Communicating threads after communication is over
  - Not all of the work can usually be overlapped → see next slide

Using threading/tasking for comm. overlap



Using threading/tasking for comm. overlap



Explicit overlapping of communication and computation

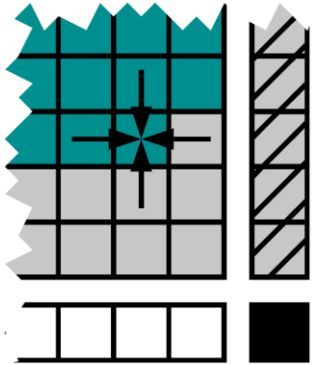
The basic principle appears simple:

```
#pragma omp parallel
{
  // ... do other parallel work
  if (thread_ID < 1) {
    MPI_Send/Recv ...  // comm. halo data
  } else {
    // Work on data that is independent
    // of halo data
  }
} // end omp parallel

// Now work on data that needs the
// halo data (all threads)
```

Explicit overlapping of communication and computation

The basic principle appears simple:

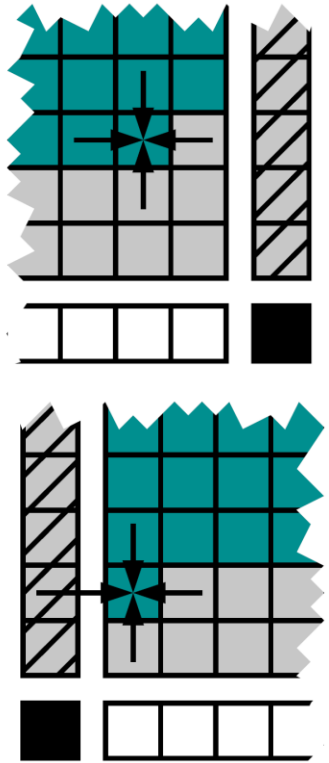
```
#pragma omp parallel
{
  // ... do other parallel work
  if (thread_ID < 1) {
    MPI_Send/Recv ...  // comm. halo data
  } else {
    // Work on data that is independent
    // of halo data
  }
} // end omp parallel

// Now work on data that needs the
// halo data (all threads)
```



Explicit overlapping of communication and computation

The basic principle appears simple:

```
#pragma omp parallel
{
  // ... do other parallel work
  if (thread_ID < 1) {
    MPI_Send/Recv ...  // comm. halo data
  } else {
    // Work on data that is independent
    // of halo data
  }
} // end omp parallel

// Now work on data that needs the
// halo data (all threads)
```



Overlapping communication with computation

Three problems:

- Application problem: separate application into
  - code that can run before the halo data is received
  - code that needs halo data
  - **May be hard to do**
- Thread-rank problem: distinguish comm. / comp. via thread ID
  - Work sharing and load balancing is harder
  - Options
    - Fully manual work distribution
    - Nested parallelism
    - Tasking & taskloops
    - Partitioned comm (MPI-4.0)
- Optimal memory placement on ccNUMA may be difficult

Overlapping communication with computation

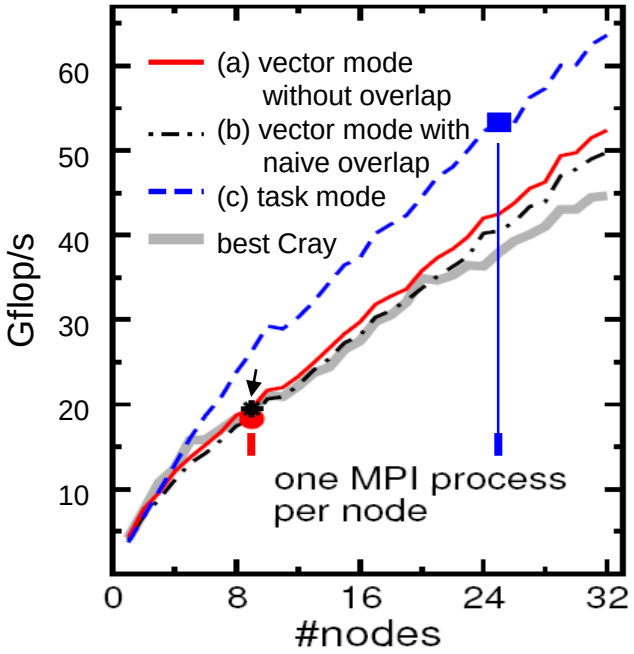
Three problems:

- Application problem: separate application into
  - code that can run before the halo data is received
  - code that needs halo data
  - **May be hard to do**
- Thread-rank problem: distinguish comm. / comp. via thread ID
  - Work sharing and load balancing is harder
  - Options
    - Fully manual work distribution
    - Nested parallelism
    - Tasking & taskloops
    - Partitioned comm (MPI-4.0)
- Optimal memory placement on ccNUMA may be difficult

error-prone & clumsy

```
if (my_thread_ID < 1) {
   MPI_Send/Recv
} else {
   my_thread_range=(high-low-1)/(num_threads-1)+1;
   my_thread_low=low+(my_thread_ID-1)*my_thread_range;
   my_thread_high=low+(my_thread_ID-1+1)
                *my_thread_range;
   my_thread_high=min(high, my_thread_high);
   for (i=my_thread_low; i<my_thread_high; i++) {
            ...
   }
}
```

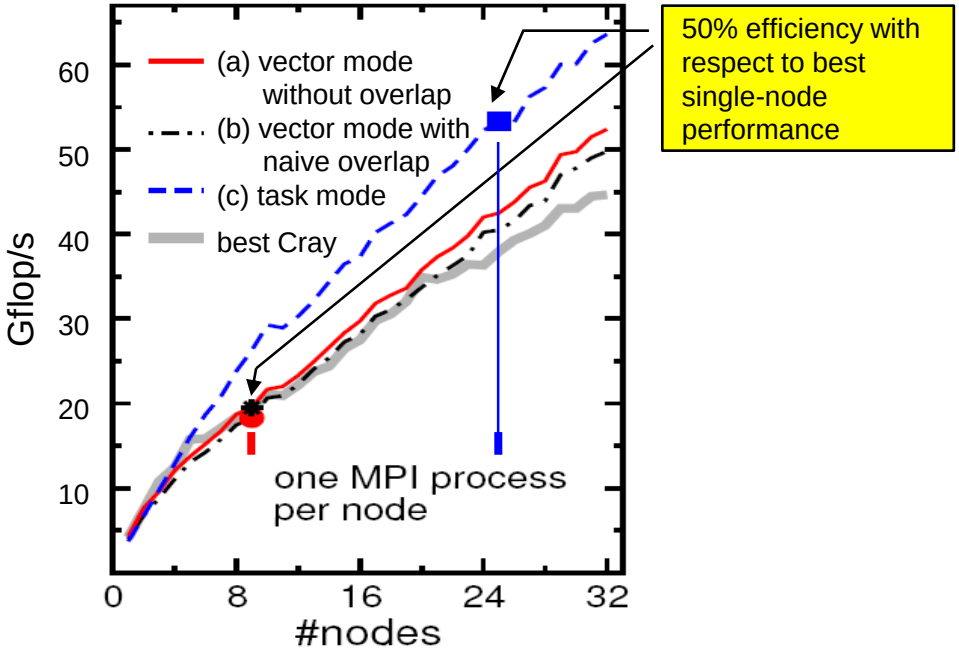
Example: sparse matrix-vector multiply (spMVM)



- spMVM on Intel Westmere cluster (6 cores/socket)
- "task mode" == explicit communication overlap using dedicated 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

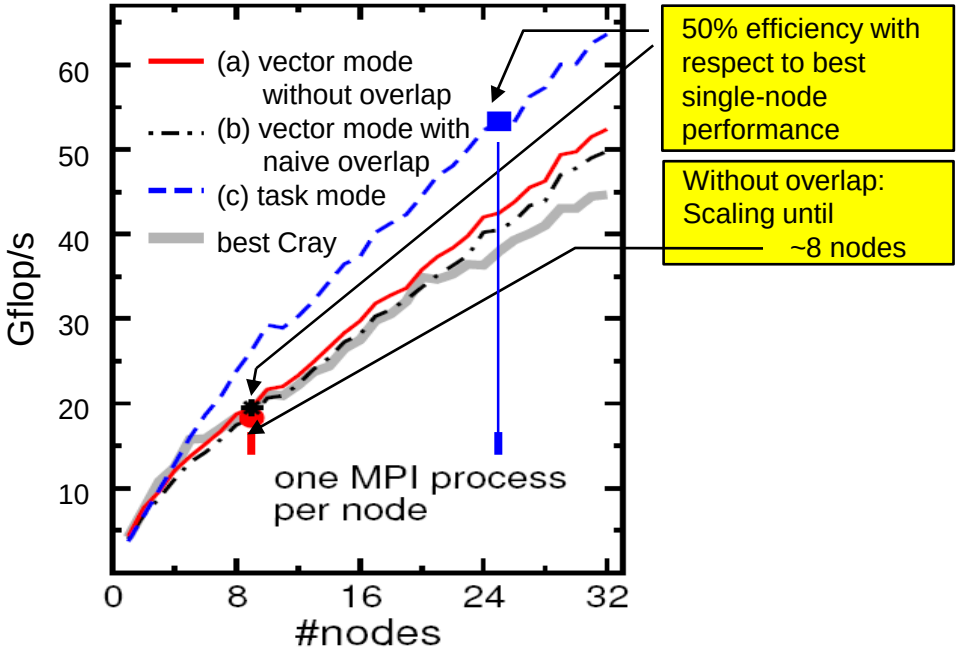
Example: sparse matrix-vector multiply (spMVM)



- spMVM on Intel Westmere cluster (6 cores/socket)
- "task mode" == explicit communication overlap using dedicated 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

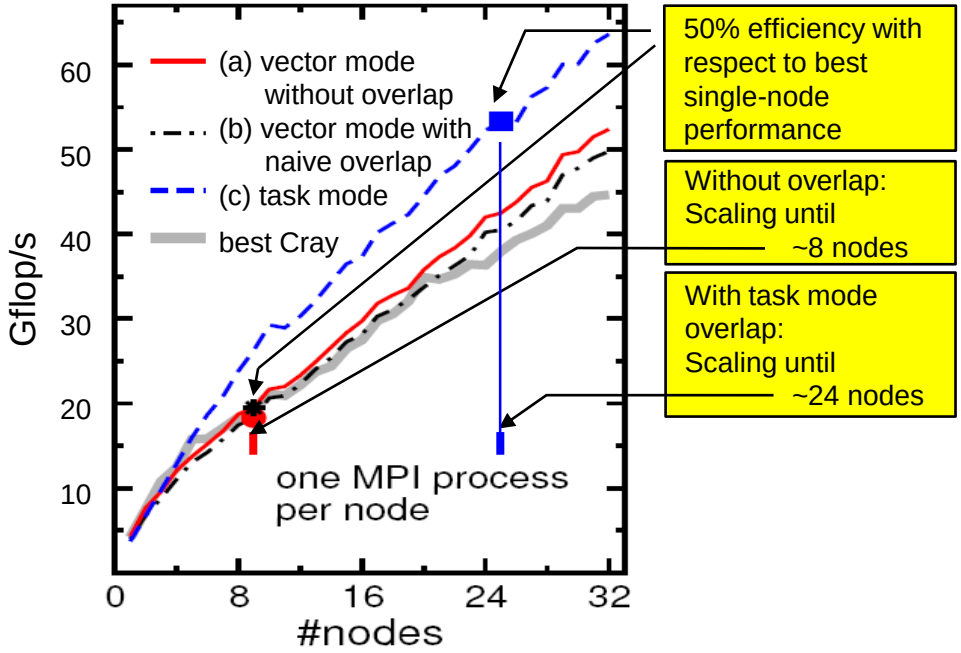
Example: sparse matrix-vector multiply (spMVM)



- spMVM on Intel Westmere cluster (6 cores/socket)
- "task mode" == explicit communication overlap using dedicated 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

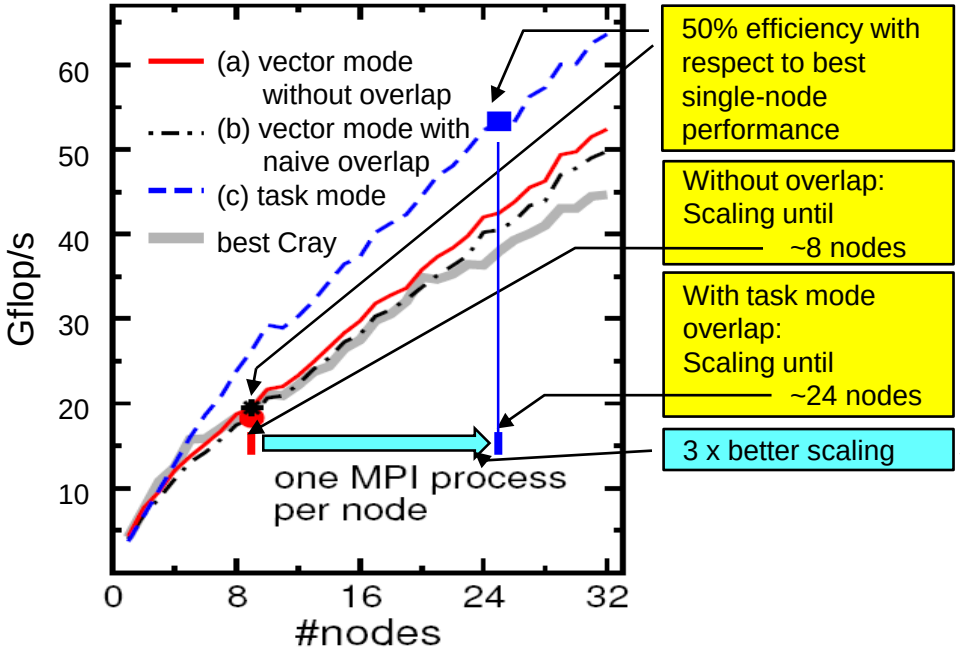
Example: sparse matrix-vector multiply (spMVM)



- spMVM on Intel Westmere cluster (6 cores/socket)
- "task mode" == explicit communication overlap using dedicated 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

Example: sparse matrix-vector multiply (spMVM)



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

It's not just the saved communication time; scaling may be much improved!

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

Programming models - MPI + OpenMP

Communication overlap with OpenMP taskloops

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
  Memory placement on ccNUMA systems
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
> **Communication overlap with OpenMP taskloops**
  Hands-on: Taskloop-based hybrid Jacobi
Main advantages, disadvantages, conclusions

OpenMP `taskloop` Directive – Syntax

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

OpenMP `taskloop` Directive – Syntax

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

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

OpenMP `taskloop` Directive – Syntax

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

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

- Fortran:

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

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

OpenMP `taskloop` Directive – Syntax

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

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

- 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

OpenMP `taskloop` Directive – Syntax

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

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

- 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++:

```
#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*`)`
  - **`untied, mergeable`** [a task clause]
  - **`final`**`(` *scalar-expr* `)`**`, priority`**`(` *priority-value* `)` [a task clause]
  - **`nogroup`**
  - **`reduction`** **(*operator:list*)** [a do/for clause]
- do/ for clauses that are not valid on a taskloop:
  - **`schedule`**`(` *type* [, *chunk*] `)`**`, nowait`**
  - **`linear`**`(`*list* [: *linear-step*] `)`**`,`** **`ordered`** [`(` *n* `)`]

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*`)`**

    Mutually exclusive
  - **`untied, mergeable`** [a task clause]
  - **`final`** **`(`** *scalar-expr* **`)`**`,` **`priority`** **`(`** *priority-value* **`)`** [a task clause]
  - **`nogroup`**
  - **`reduction`** **(*operator:list*)** [a do/for clause]
- do/ for clauses that are not valid on a taskloop:
  - **`schedule`** **`(`** *type* [, *chunk*] **`)`**`,` **`nowait`**
  - **`linear`** **`(`***list* [: *linear-step*] **`)`**`,` **`ordered`** [**`(`** *n* **`)`**]

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* **`)`**

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

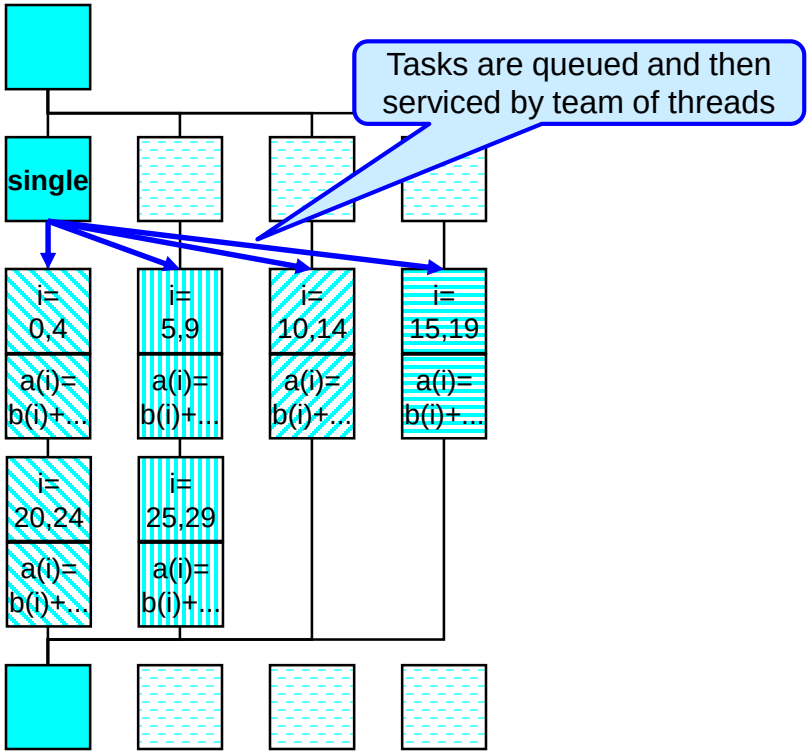
OpenMP `single` & `taskloop` Directives

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

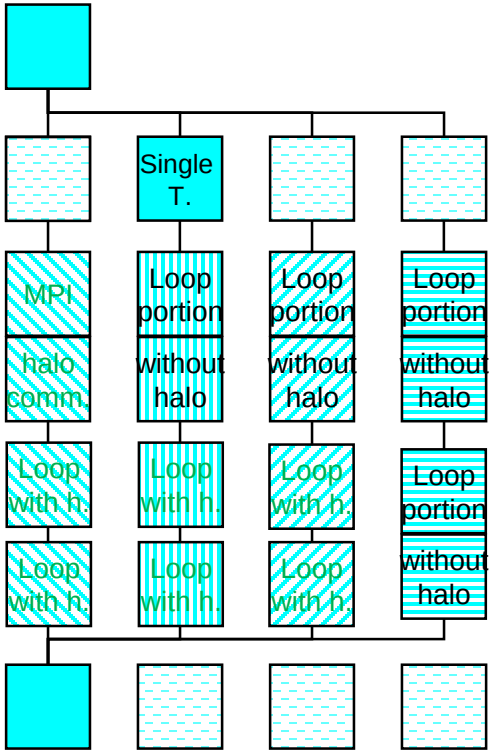


Comm. overlap with task & taskloop Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    { // MPI halo communication:
        MPI_Send/Recv...
      // 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 halo task */

    // numerical loop without 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 single */
} /*omp end parallel*/
```

Number of tasks may be influenced with grainsize or num_tasks clauses

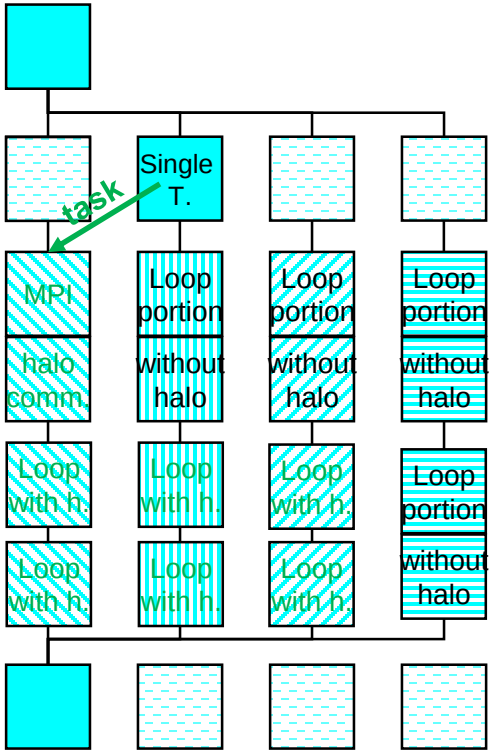


Comm. overlap with task & taskloop Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    { // MPI halo communication:
        MPI_Send/Recv...
      // 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 halo task */

    // numerical loop without 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 single */
} /*omp end parallel*/
```

Number of tasks may be influenced with grainsize or num_tasks clauses

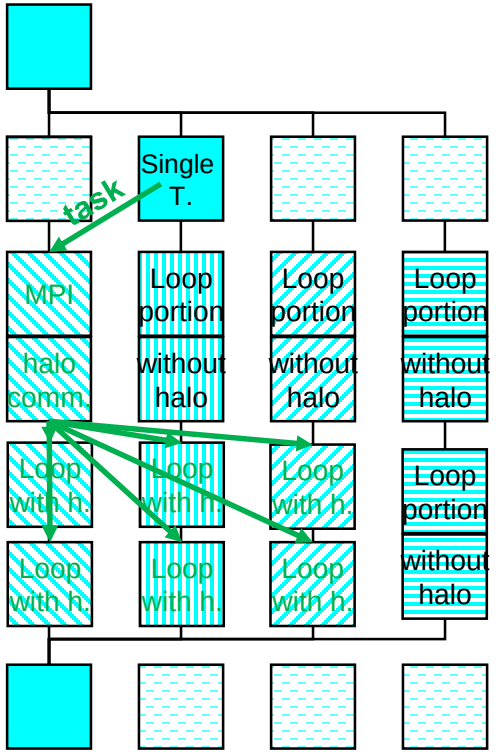


Comm. overlap with task & taskloop Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    { // MPI halo communication:
        MPI_Send/Recv...
      // 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 halo task */

    // numerical loop without 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 single */
} /*omp end parallel*/
```

Number of tasks may be influenced with grainsize or num_tasks clauses

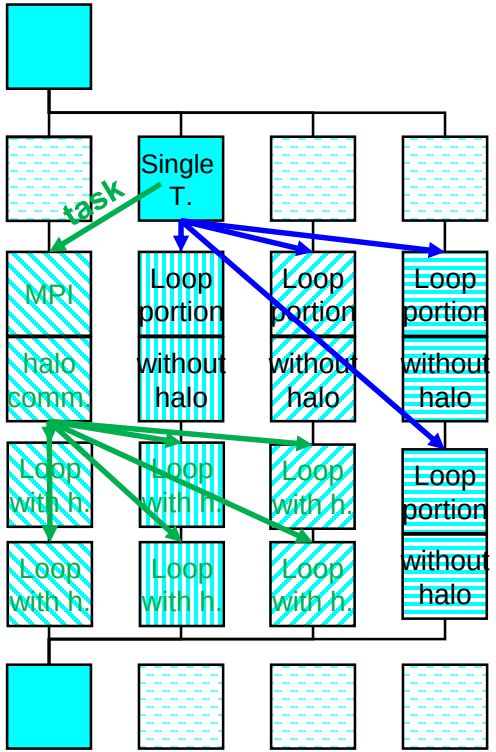


Comm. overlap with task & taskloop Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    { // MPI halo communication:
        MPI_Send/Recv...
      // 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 halo task */

    // numerical loop without 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 single */
} /*omp end parallel*/
```

Number of tasks may be influenced with grainsize or num_tasks clauses



Partitioned Point-to-Point Communication

- New in 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 "sent" with **`MPI_Pready`** by a thread;
  - same for receiving

- Technically provided as a new form of persistent communication.

Programming models - MPI + OpenMP

Hands-On #4

Taskloop-based hybrid Jacobi

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
  Memory placement on ccNUMA systems
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
  Communication overlap with OpenMP taskloops
> **Hands-on: Taskloop-based hybrid Jacobi**
Main advantages, disadvantages, conclusions

Programming models - MPI + OpenMP

Main advantages, disadvantages, conclusions

General considerations
How to compile, link, and run
  Hands-on: Hello hybrid!
System topology, ccNUMA, and memory bandwidth
  Memory placement on ccNUMA systems
Topology and affinity on multicore
  Hands-on: Pinning
Case study: The Multi-Zone NAS Parallel Benchmarks
  Hands-on: Masteronly hybrid Jacobi
Overlapping communication and computation
  Communication overlap with OpenMP taskloops
  Hands-on: Taskloop-based hybrid Jacobi
> Main advantages, disadvantages, conclusions

MPI+OpenMP: Main advantages

- 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

- 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
- Complex thread and process pinning

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

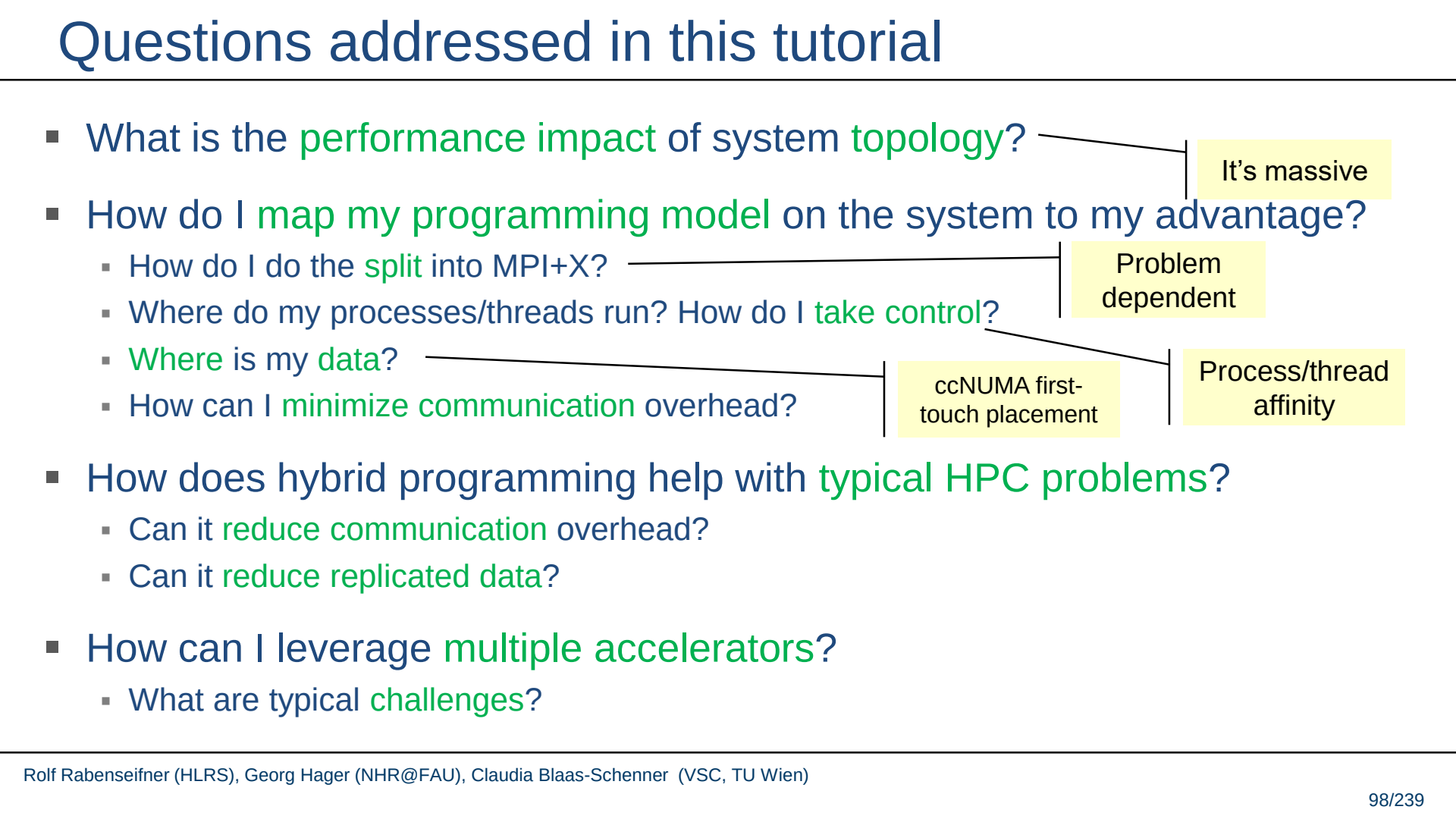
- Sleeping threads

Additional disadvantages when overlapping communication and computation:

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

Questions addressed in this tutorial

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?
- How can I leverage multiple accelerators?
  - What are typical challenges?



Programming models - MPI + Accelerator

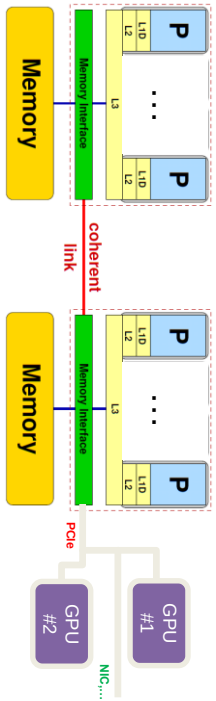
General considerations slide 100
OpenACC 105
Advantages & main challenges 112

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	2.3 Tflop/s
eff. memory (HBM) bandwidth	1300 Gbyte/s	170 Gbyte/s
inter-device bandwidth (PCIe)	≈ 30 Gbyte/s	
inter-device bandwidth (NVlink)	> 500 Gbyte/s	



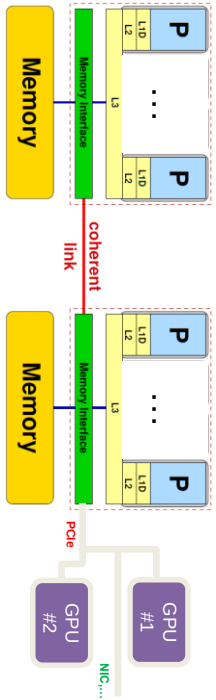
→ Speedups can only be attained if communication overheads are under control

→ Basic estimates help

Accelerator 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
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	



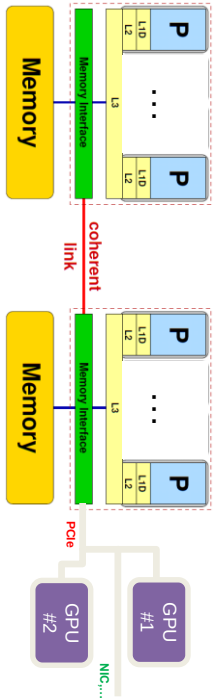
→ Speedups can only be attained if communication overheads are under control

→ Basic estimates help

Accelerator programming: Bottlenecks reloaded

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

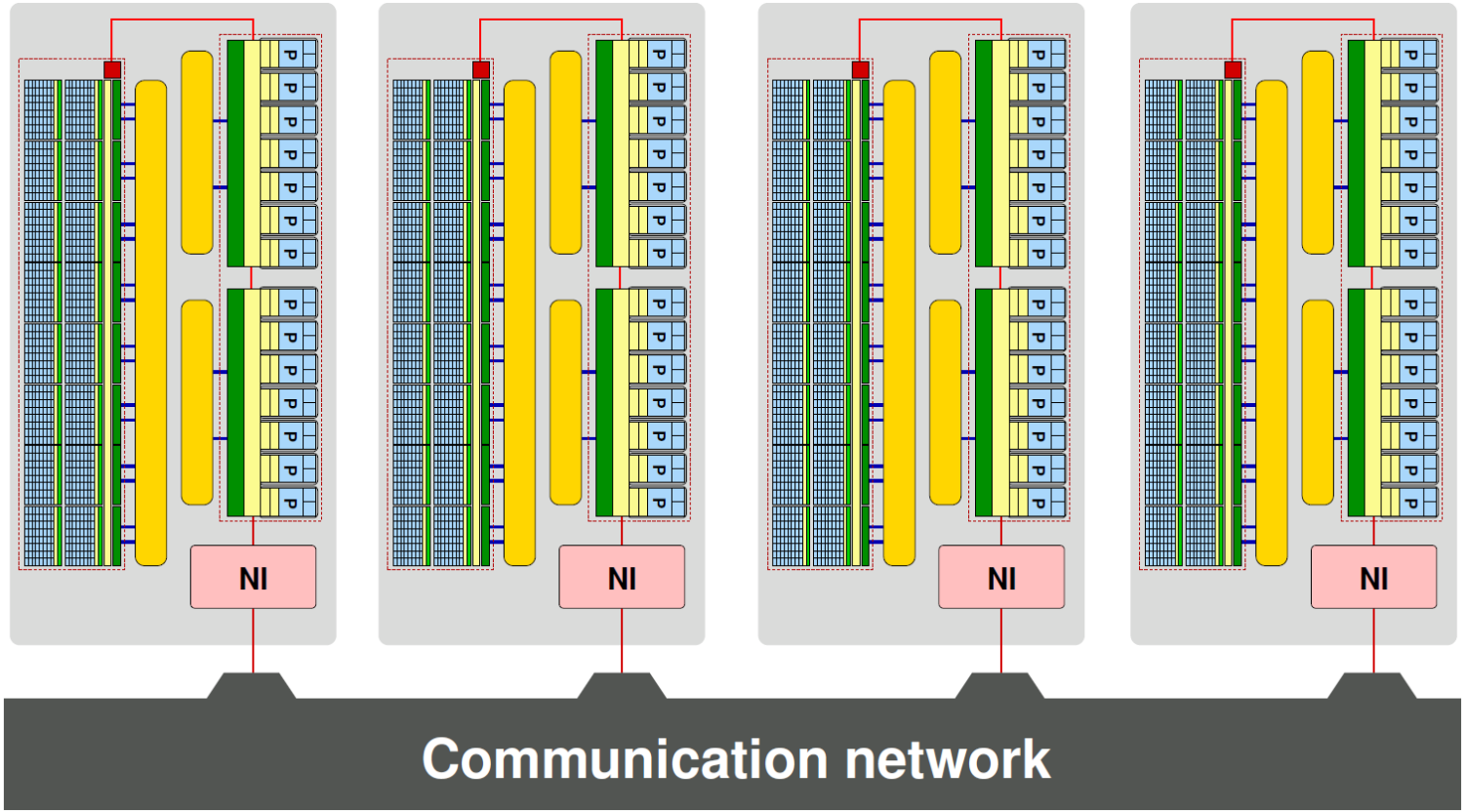
	per GPGPU	per CPU
DP peak performance	9.7 Tflop/s ⟸ 4x	2.3 Tflop/s
Machine balance	0.11 B/F	0.10 B/F
eff. memory (HBM) bandwidth	1300 Gbyte/s ⟸ 8x	170 Gbyte/s
inter-device bandwidth (PCIe)	≈ 30 Gbyte/s	
inter-device bandwidth (NVlink)	> 500 Gbyte/s	



→ 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

See also:
https://www.fz-juelich.de/en/ias/jsc/news/events/seminars/msa-seminar/2020-01-21-cuda-aware-mpi

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)

See also:
https://www.fz-juelich.de/en/ias/jsc/news/events/seminars/msa-seminar/2020-01-21-cuda-aware-mpi

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

See also:
https://www.fz-juelich.de/en/ias/jsc/news/events/seminars/msa-seminar/2020-01-21-cuda-aware-mpi

Questions to ask

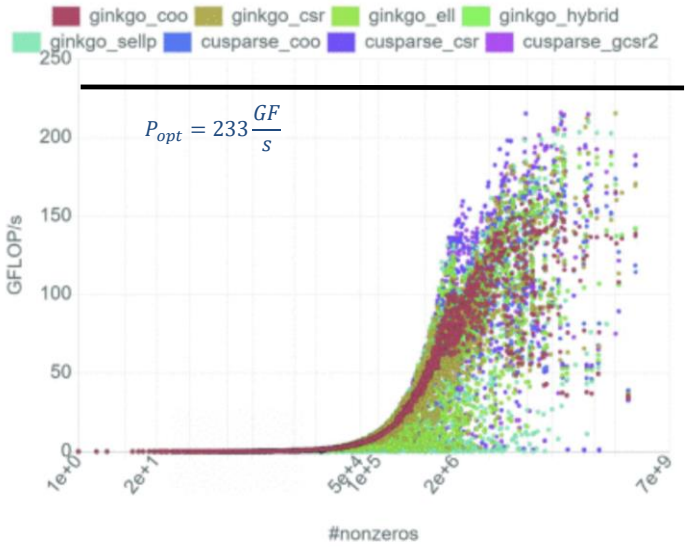
- Is the MPI implementation CUDA aware?
  - Yes: Can use device pointers in MPI calls
  - No: Explicit DtoH/HtoD buffer transfers required
  - Copying to consecutive halo buffers may still be necessary
- Is NVLink available?
  - Yes: Direct GPU-GPU MPI communication with MPI
    - Supported by: P100, 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!

See also:
https://www.fz-juelich.de/en/ias/jsc/news/events/seminars/msa-seminar/2020-01-21-cuda-aware-mpi

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)*, 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
…

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???

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 + Accelerator

OpenACC

General considerations
> OpenACC
Advantages & main challenges

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

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)

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

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)

A very simple OpenACC example (PGI 14.10): 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);
...
}
```

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

data mgmt

```
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);
...
}
```

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

data mgmt

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

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

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

data mgmt

```
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);
...
}
```

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];
  }
}
```

A very simple OpenACC example (PGI 14.10): 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

```
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];
  }
}
```

execution

```
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)...
```

Example: 2D Jacobi smoother

```
#pragma acc data copy(phi1[0:sizex*sizey],phi2[0:sizex*sizey])
{
  for(n=0; n<iter; n++) {


    for(int i=1; i<sizex-1; ++i) {
      ofs = i*sizey;

      for(int j=1; j<sizey-1; ++j) {
          phi1[ofs+j] = oos * (phi2[ofs+j-1] +
                               phi2[ofs+j+1] +
                               phi2[ofs+j-sizey] +
                               phi2[ofs+j+sizey]);
      }
    }
  }
  swap(phi1,phi2);
}
```

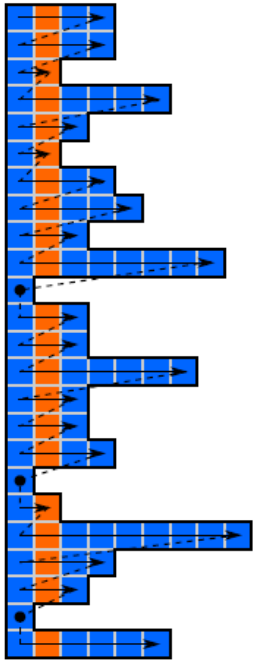
Example: 2D Jacobi smoother

```
#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 private(ofs)
    for(int i=1; i<sizex-1; ++i) {
      ofs = i*sizey;
#pragma acc loop independent
      for(int j=1; j<sizey-1; ++j) {
          phi1[ofs+j] = oos * (phi2[ofs+j-1] +
                               phi2[ofs+j+1] +
                               phi2[ofs+j-sizey] +
                               phi2[ofs+j+sizey]);
      }
    }
  }
  swap(phi1,phi2);
}
```

skipped

Example: Sparse MVM (std. CSR format)

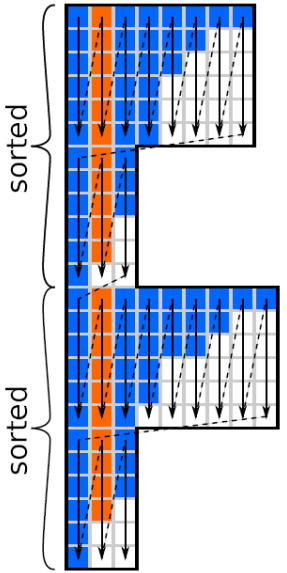
```
#pragma acc parallel present(val[0:numNonZeros], \
  colInd[0:numNonZeros],      \
  rowPtr[0:numRows+1],        \
  x[0:numRows],               \
  y[0:numRows])               \
  loop
for (int rowID=0; rowID<numRows; ++rowID)    {
  double tmp = y[rowID];
  // loop over all elements in row
  for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



skipped

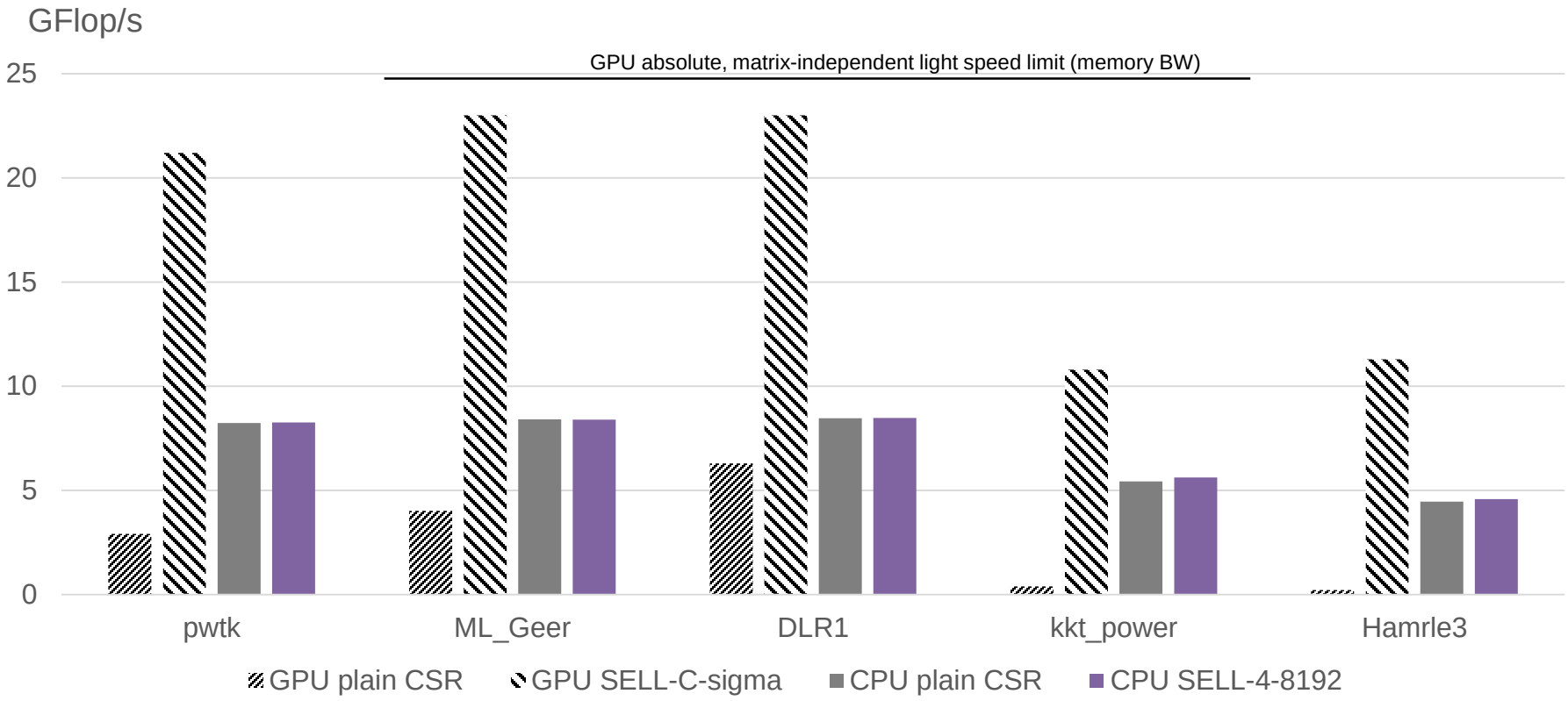
Example: Sparse MVM (SELL-C-σ format)

```
#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;
  }
}
```

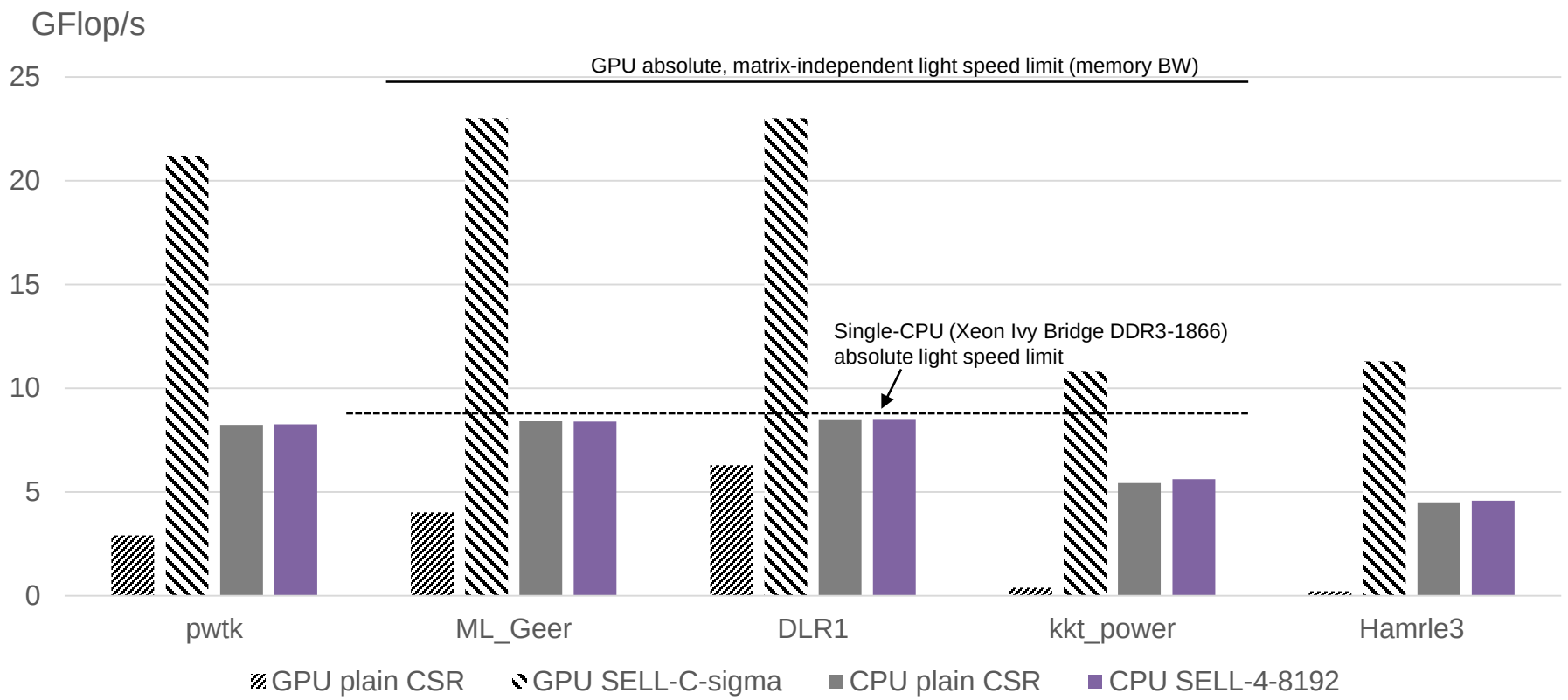


M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units.* SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: 10.1137/130930352

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



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



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 requires good understanding of performance issues
  - Performance is not only about code; data structures can be decisive as well
- Support for accelerator pragmas still restricted to certain environments
  - NVIDIA GPUs have best support

Questions addressed in this tutorial

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?
- How can I leverage multiple accelerators?
  - What are typical challenges?

Data structures are decisive, inter-device communication support varies

Programming models - MPI + MPI-3 shared memory

General considerations & uses cases	slide 116
Re-cap: MPI_Comm_split & one-sided communication	120
How-to	128
Exercise: MPI_Bcast	143
Quiz 1	155
MPI memory models & synchronization	156
Shared memory problems	166
Advantages & disadvantages, conclusions	169
Quiz 2	174

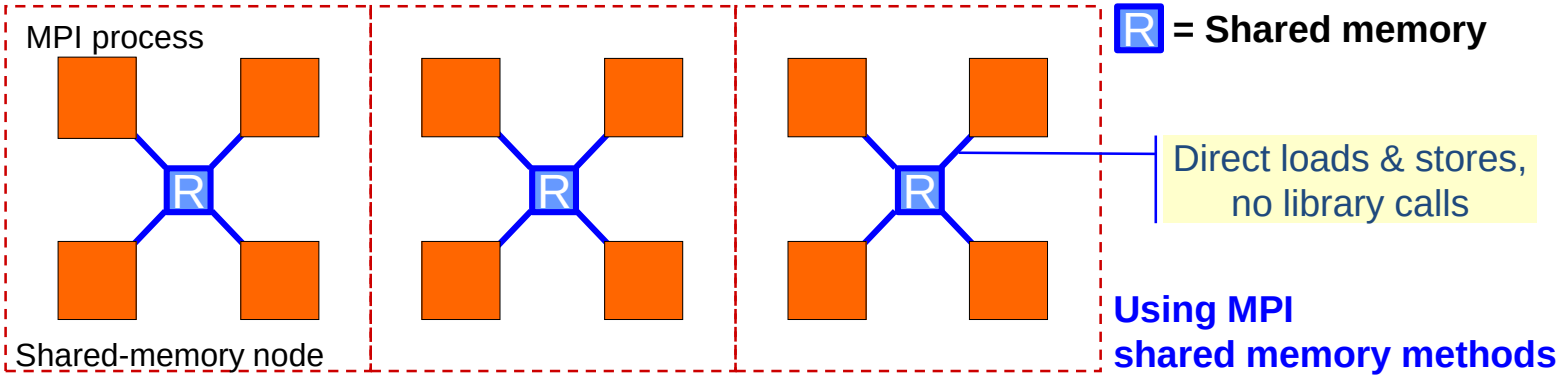
Hybrid MPI + MPI-3 shared memory

What is it?

- Addon to pure message passing
- MPI processes can share memory segments within a node

Use cases/advantages

- A: Reducing replicated data → Reduced memory requirements
- B: Reducing intra-node message passing → Reduced intra-node communication time

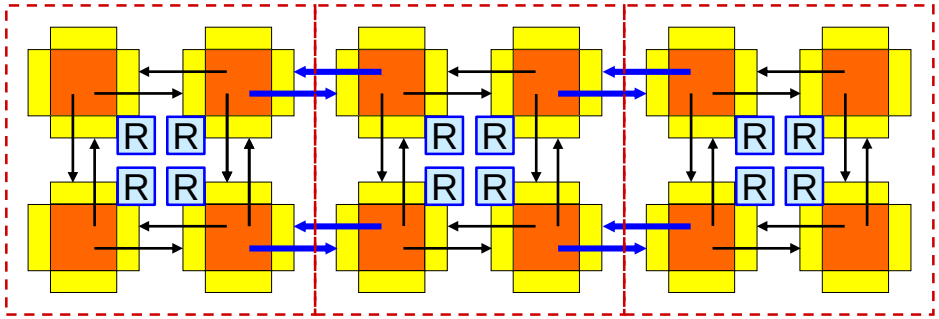


Hybrid MPI + MPI-3 shared memory

- Further advantages
  - Using only one parallel programming model
  - No OpenMP problems (e.g., thread-safety isn't an issue)
- Major Problems
  - Communicator must be split into shared memory islands
  - No increase in exploitable parallelism
  - None of the "automatic" advantages of MPI+OpenMP
    - Exploiting advantages requires programming effort

See MPI+OpenMP summary

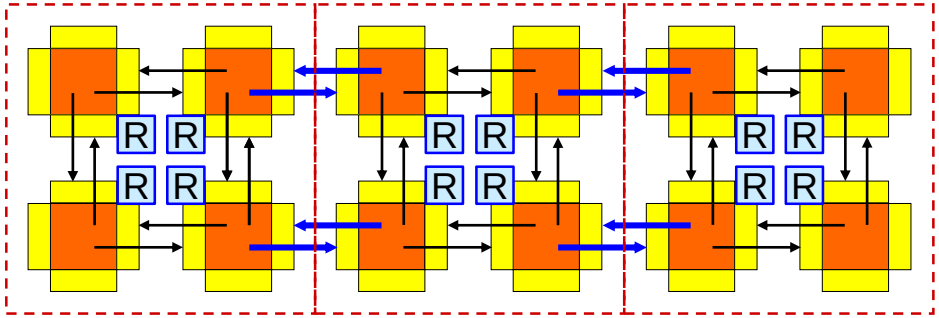
Use case A: Reducing memory requirements



R = **Replicated data in each MPI process**

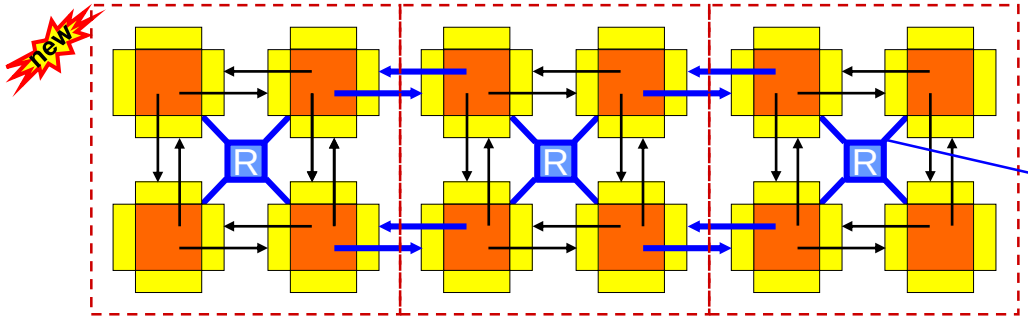
Example:
Cluster of SMP nodes
without using MPI shared memory methods

Use case A: Reducing memory requirements



R = Replicated data
in each MPI process

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

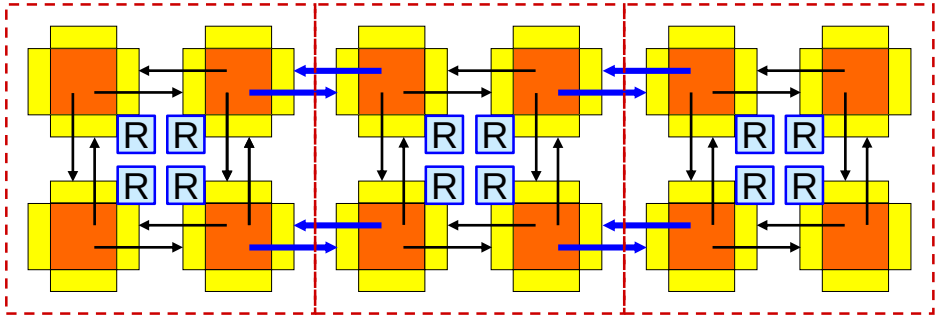


R = Shared memory
→ replicated data only once
within each SMP node

Direct loads & stores, no library calls

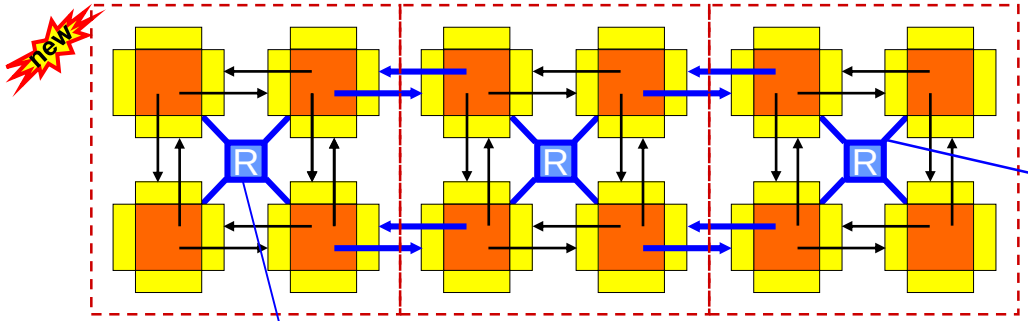
Using MPI
shared memory methods

Use case A: Reducing memory requirements



R = Replicated data
in each MPI process

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



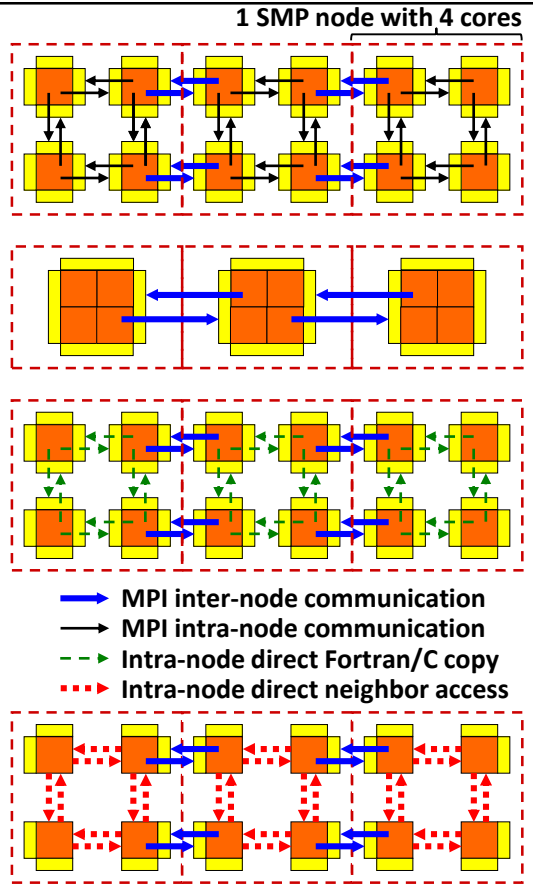
R = Shared memory
→ replicated data only once
within each SMP node

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**.

Use case B: Reducing intra-node message passing



- MPI on each core (not hybrid)
  - Halos between all cores
  - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
  - Multi-threaded MPI processes
  - Halos communication only between MPI processes
- MPI cluster communication + MPI shared memory communication
  - Same as “MPI on each core”, but
  - within the shared memory nodes, halo communication through direct copying with C or Fortran statements

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

new

Programming models
- MPI + MPI-3.0 shared memory

Re-cap

- MPI_Comm_split
- One-sided communication

General considerations & uses cases
> **Re-cap: MPI_Comm_split & one-sided communication**
How-to
Exercise: MPI_Bcast
Quiz 1
MPI memory models & synchronization
Shared memory problems
Advantages & disadvantages, conclusions
Quiz 2

New sub-communicators with MPI_Comm_split

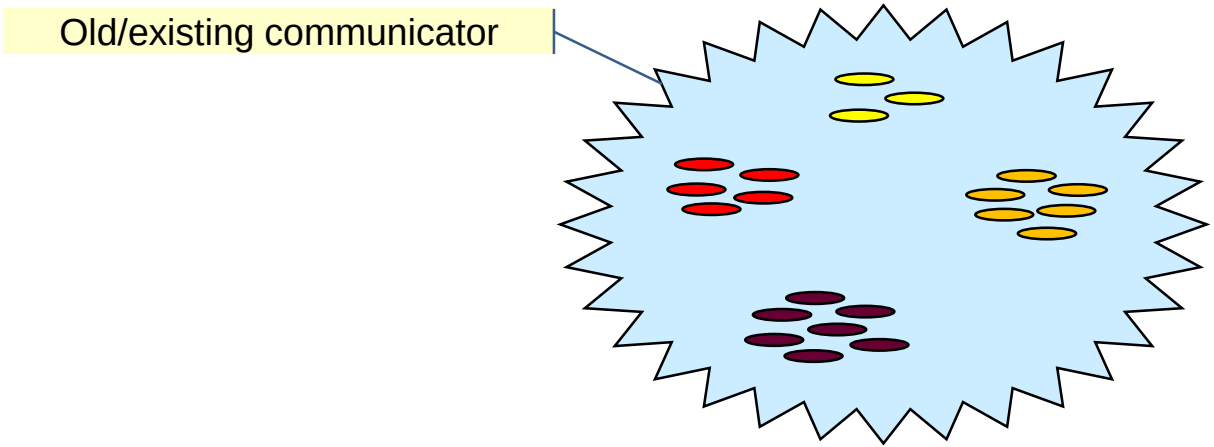
- New sub-communicators via MPI_Comm_split

New sub-communicators with MPI_Comm_split

- New sub-communicators via MPI_Comm_split
  - Each process must specify a color

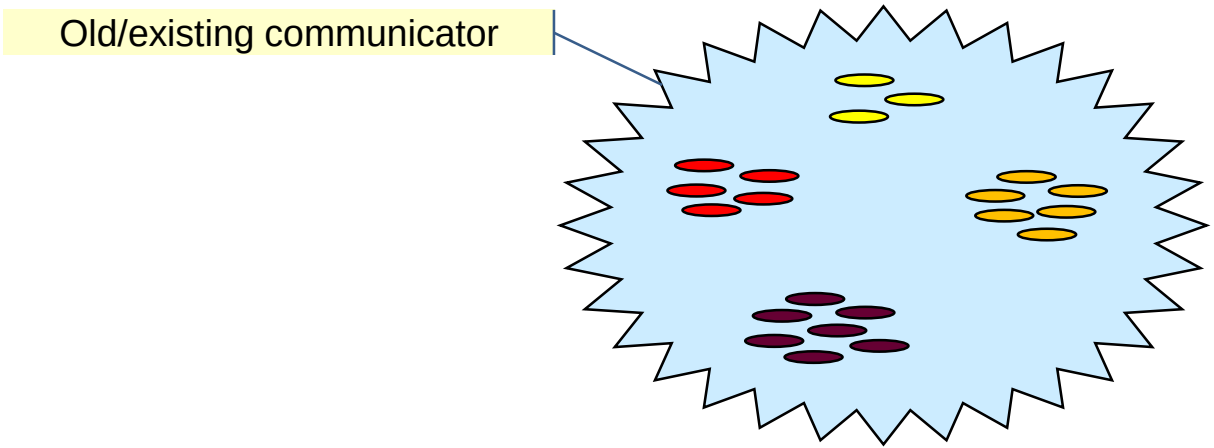
New sub-communicators with MPI_Comm_split

- New sub-communicators via MPI_Comm_split
  - Each process must specify a color



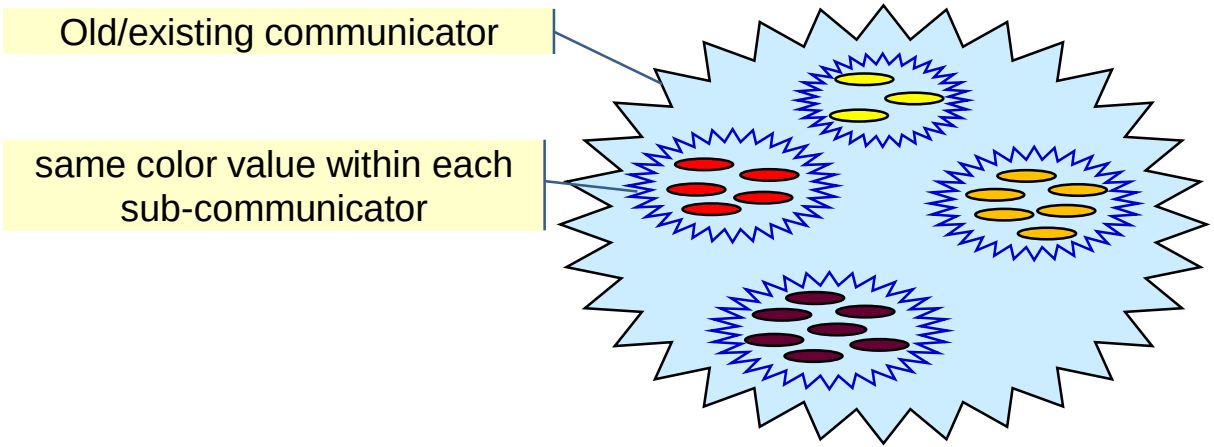
New sub-communicators with MPI_Comm_split

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



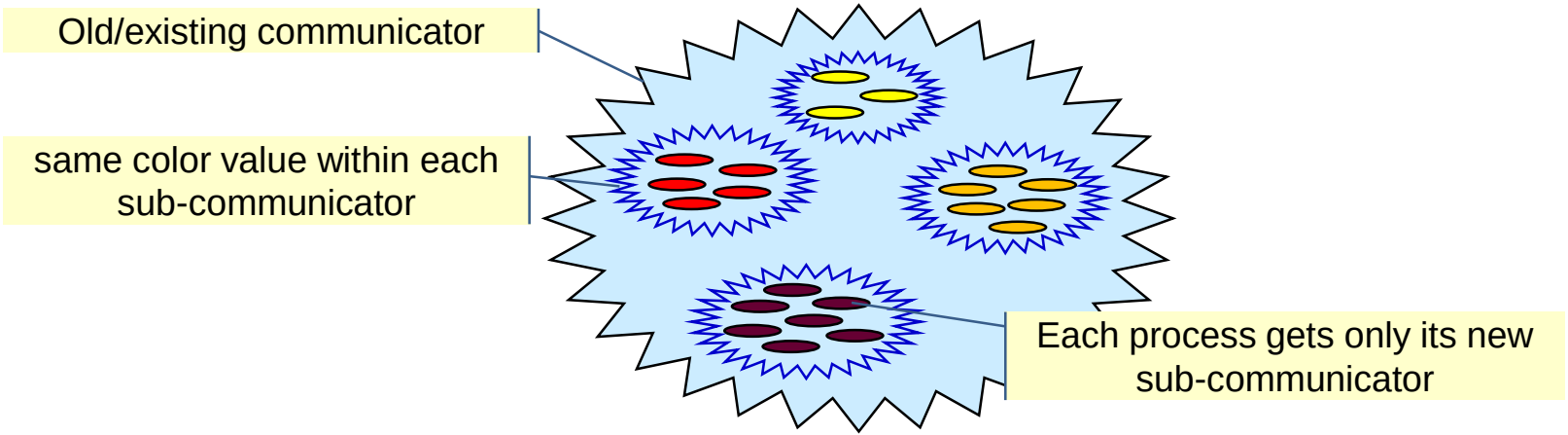
New sub-communicators with MPI_Comm_split

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



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

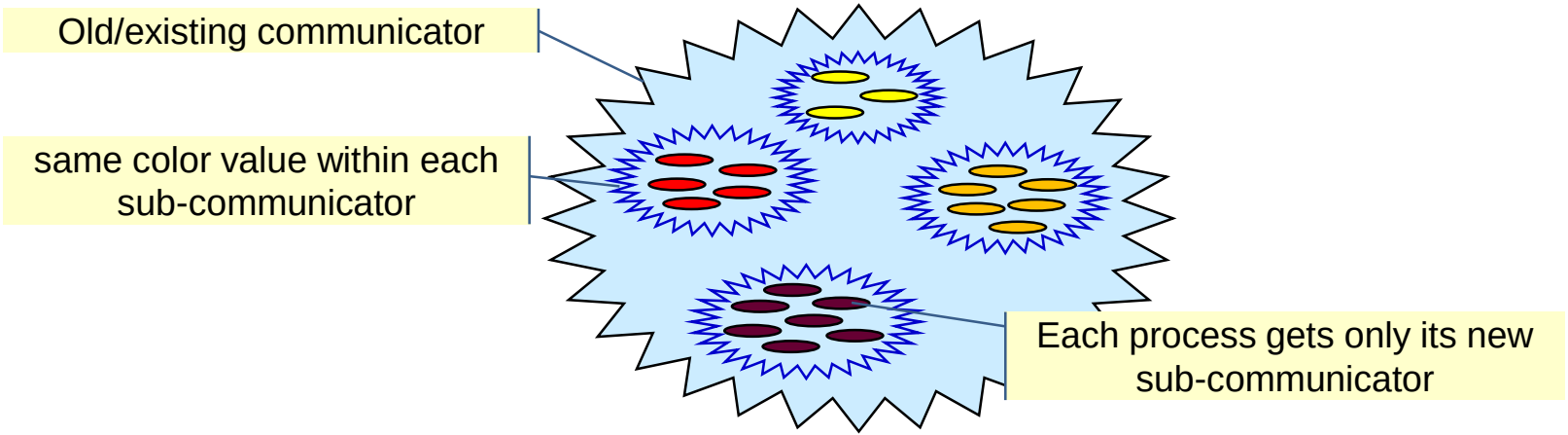


New sub-communicators with MPI_Comm_split

- 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
→ shared memory

New in MPI-3.0



Example: MPI_Comm_split()

- **C/C++**

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,
                    MPI_Comm *newcomm)
```

- **Fortran**

```
MPI_COMM_SPLIT (comm, color, key, newcomm, ierror)
```

mpi_f08:

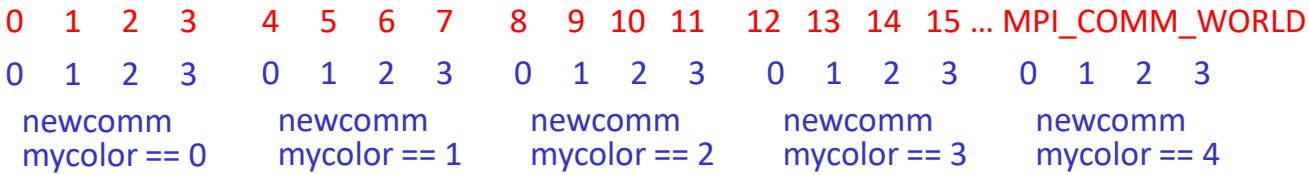
```
TYPE(MPI_Comm) :: comm, newcomm
INTEGER :: color, key;
INTEGER, OPTIONAL :: ierror
```

mpi & mpif.h:

```
INTEGER comm, color, key, newcomm, ierror
```

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

Each process gets only its own sub-communicator



Example: MPI_Comm_split()

- **C/C++**

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,
                    MPI_Comm *newcomm)
```

- **Fortran**

```
MPI_COMM_SPLIT (comm, color, key, newcomm, ierror)
```

mpi_f08:

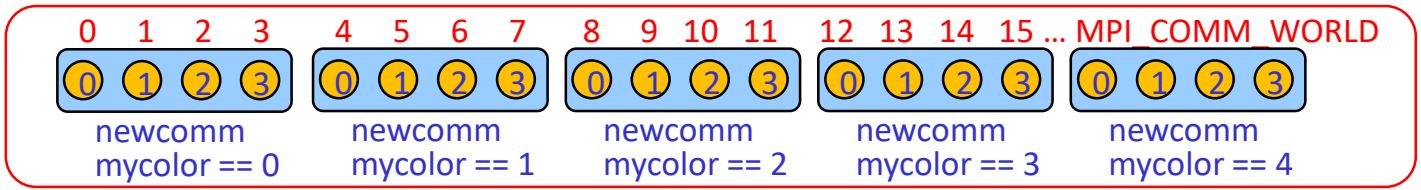
```
TYPE(MPI_Comm) :: comm, newcomm
INTEGER :: color, key;
INTEGER, OPTIONAL :: ierror
```

mpi & mpif.h:

```
INTEGER comm, color, key, newcomm, ierror
```

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

Each process gets only its own sub-communicator



Example: MPI_Comm_split()

- **C/C++**

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,
                    MPI_Comm *newcomm)
```

- **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:

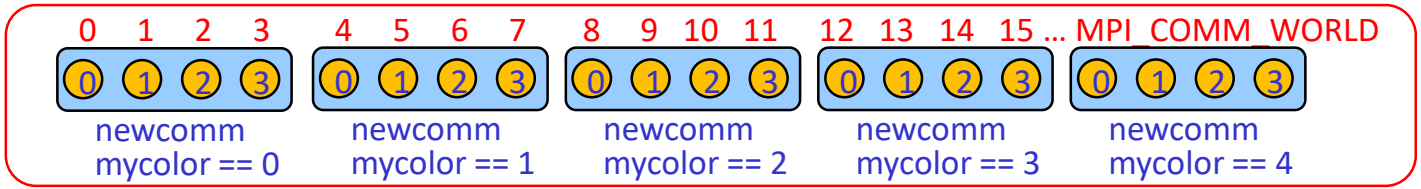
```
INTEGER comm, color, key, newcomm, ierror
```

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

Each process gets only its own sub-communicator

Example:

```
int my_rank, mycolor, key, my_newrank;
PI_Comm newcomm;
AMPI_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);
```



Example: MPI_Comm_split()

- **C/C++**

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,
                    MPI_Comm *newcomm)
```

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

- **Fortran**

mpi_f08:

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

mpi & mpif.h:

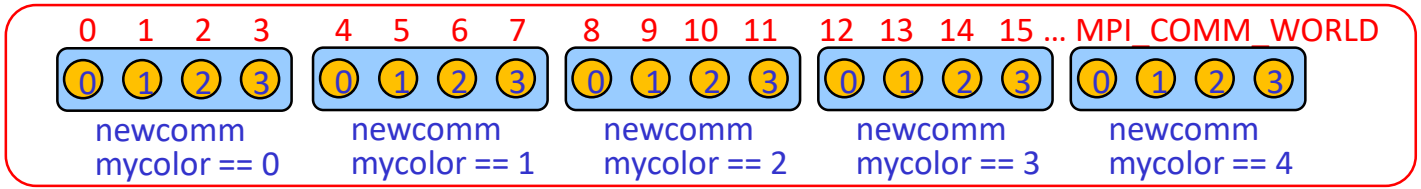
```
  INTEGER comm, color, key, newcomm, ierror
```

Each process gets only its own sub-communicator

Example:

```
int my_rank, mycolor, key, my_newrank;
PI_Comm newcomm;
AMPI_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



Example: MPI_Comm_split()

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

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

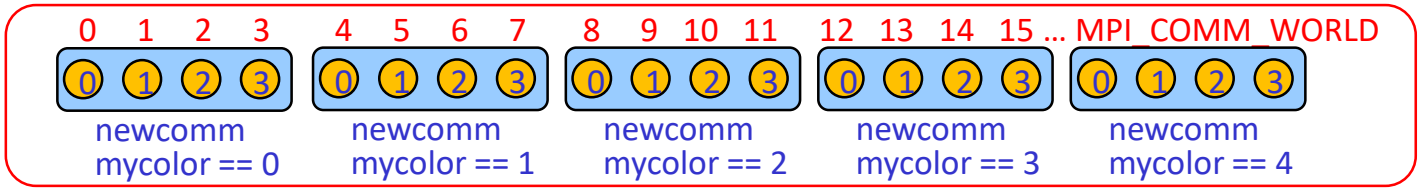
Each process gets only its own sub-communicator

Example:

```
int my_rank, mycolor, key, my_newrank;
PI_Comm newcomm;
AMPI_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



Example: MPI_Comm_split()

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

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

Each process gets only its own sub-communicator

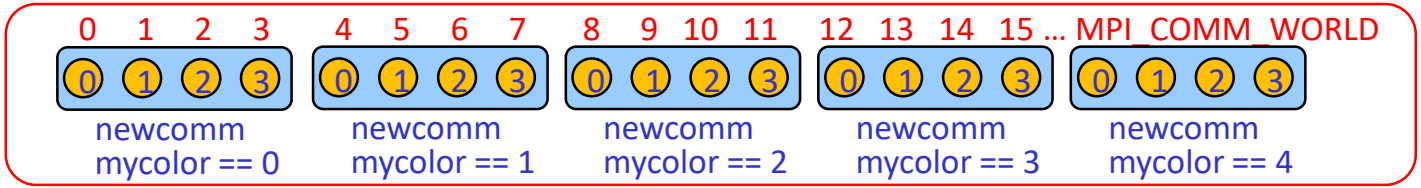
Example:

```
int my_rank, mycolor, key, my_newrank;
PI_Comm newcomm;
AMPI_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

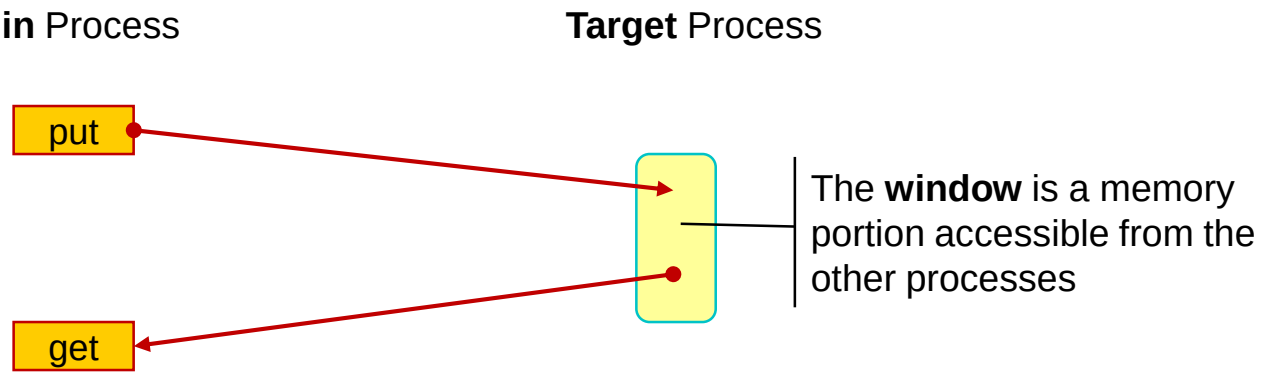
key==0 → ranking in newcomm is sorted as in old comm
key ≠ 0 → ranking in newcomm is sorted according key values

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



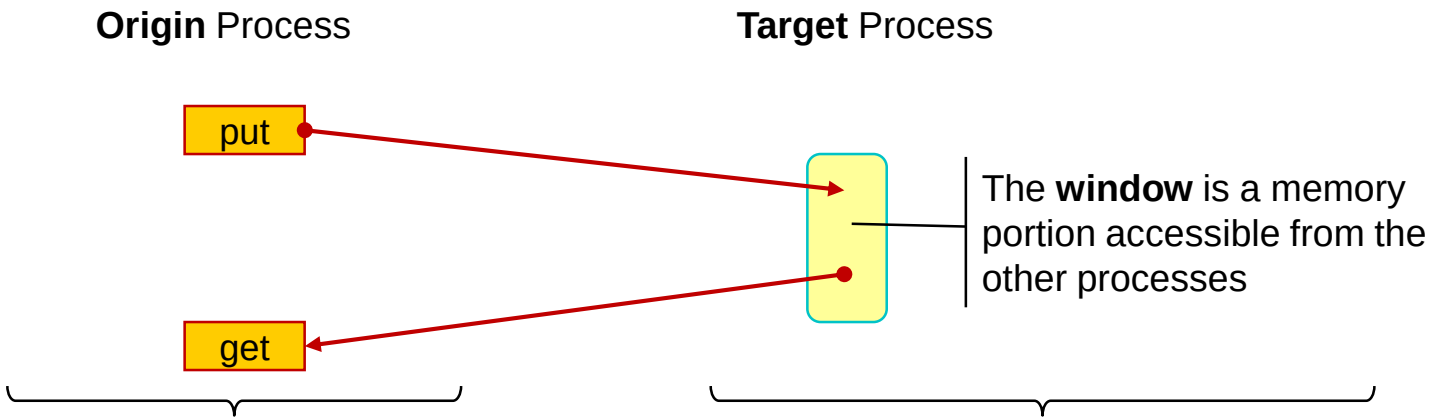
Re-cap: One-sided Communication

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

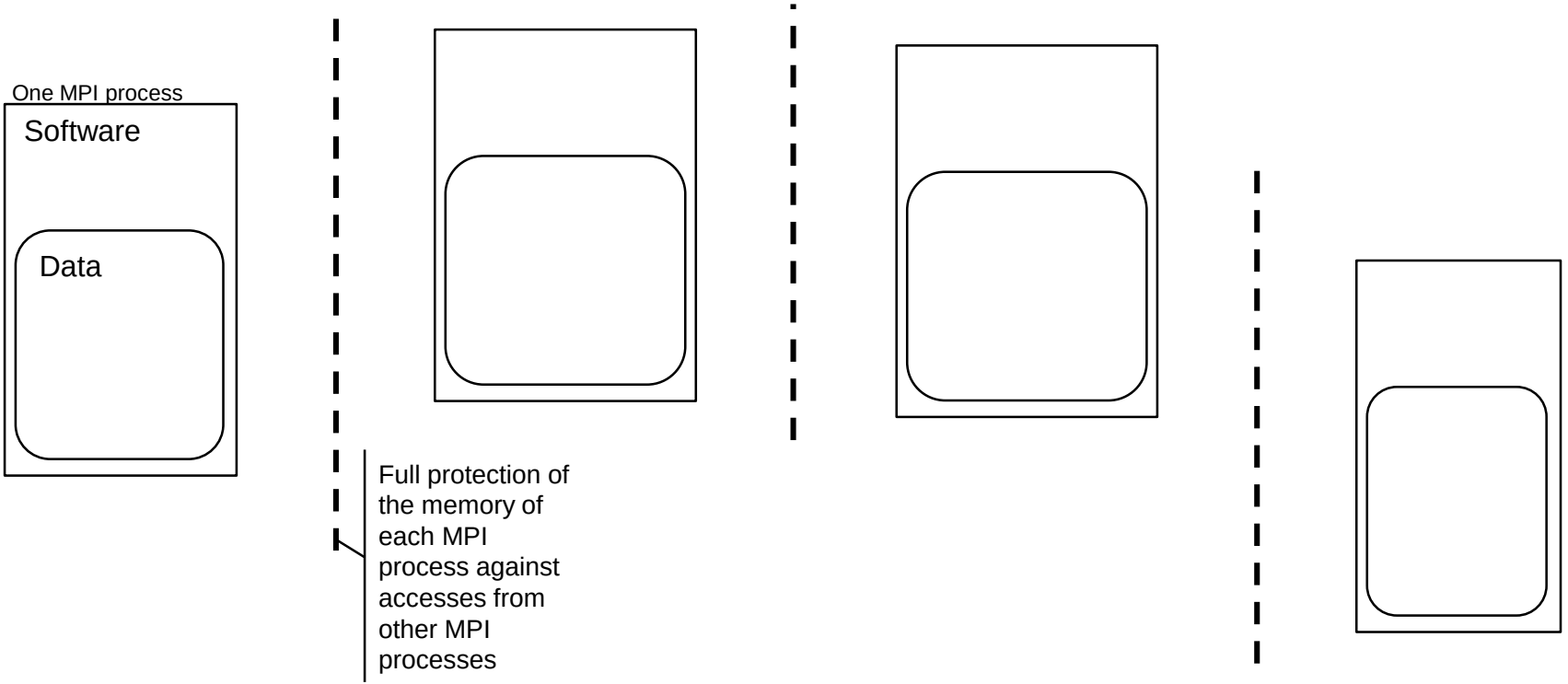


Re-cap: One-sided Communication

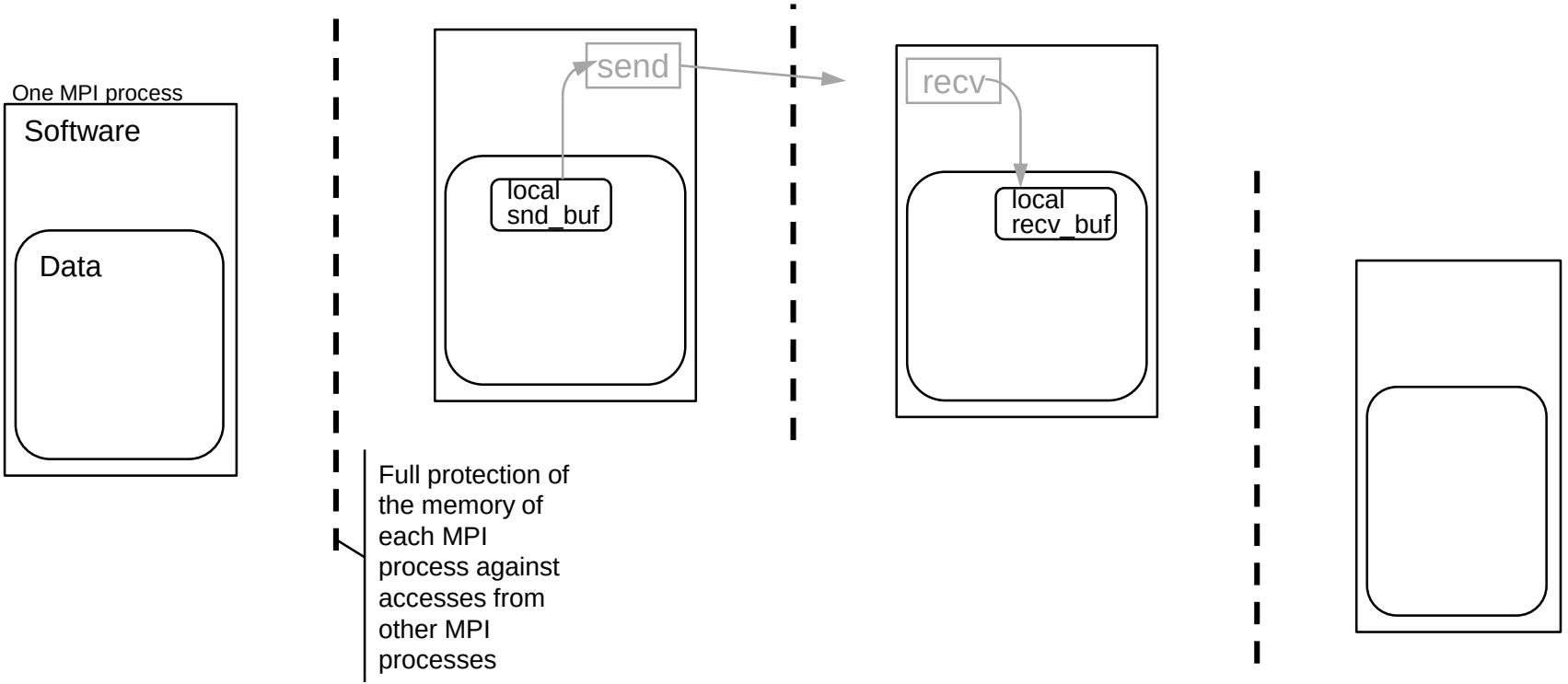
- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses



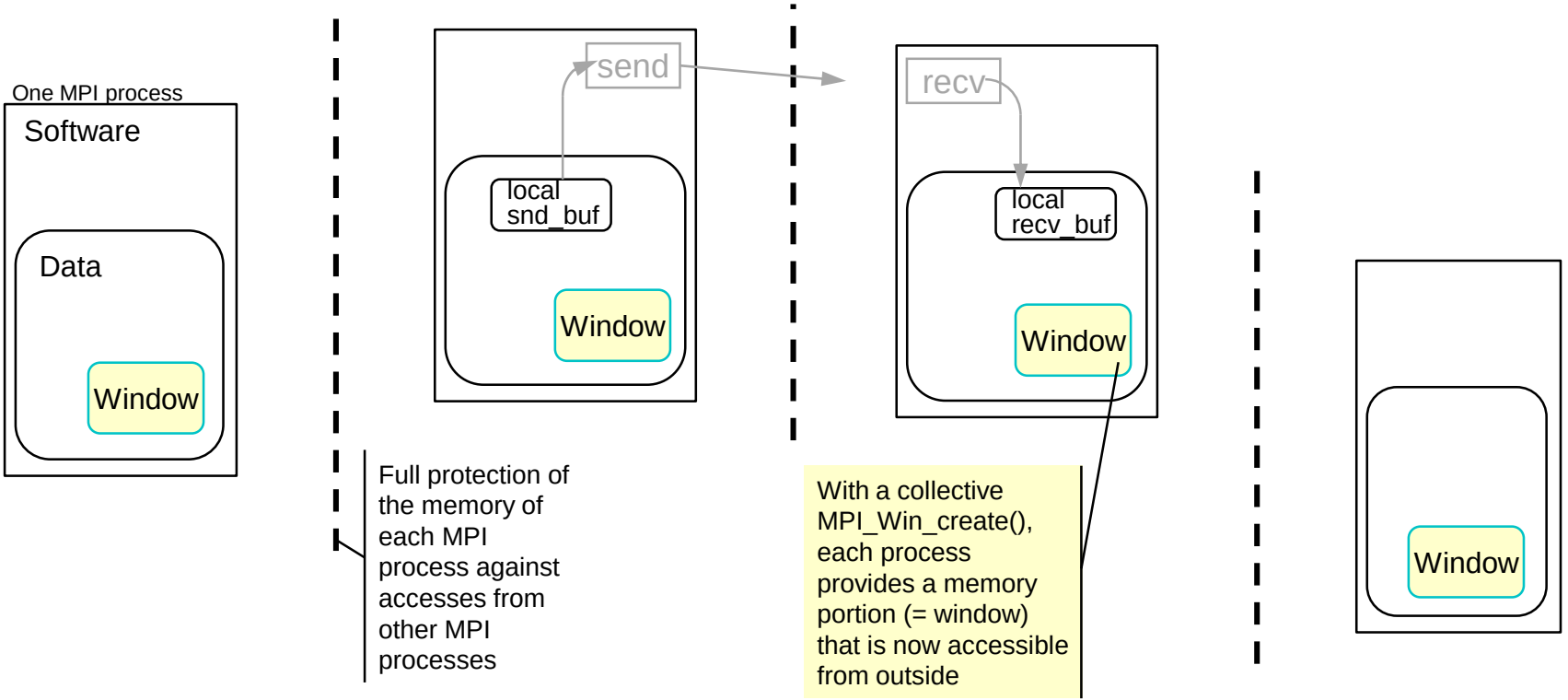
Typically, all processes are both, origin and target processes



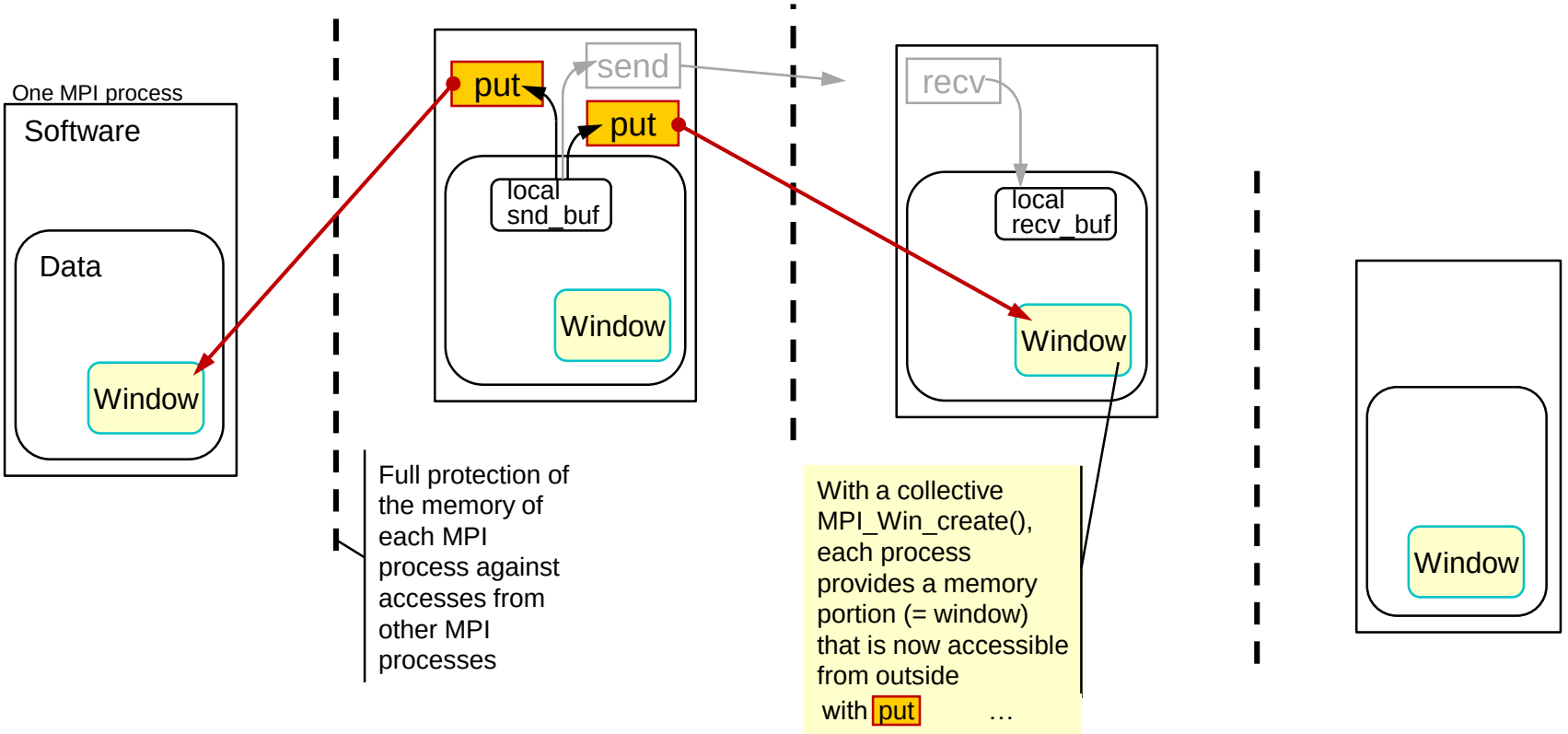
Typically, all processes are both, origin and target processes



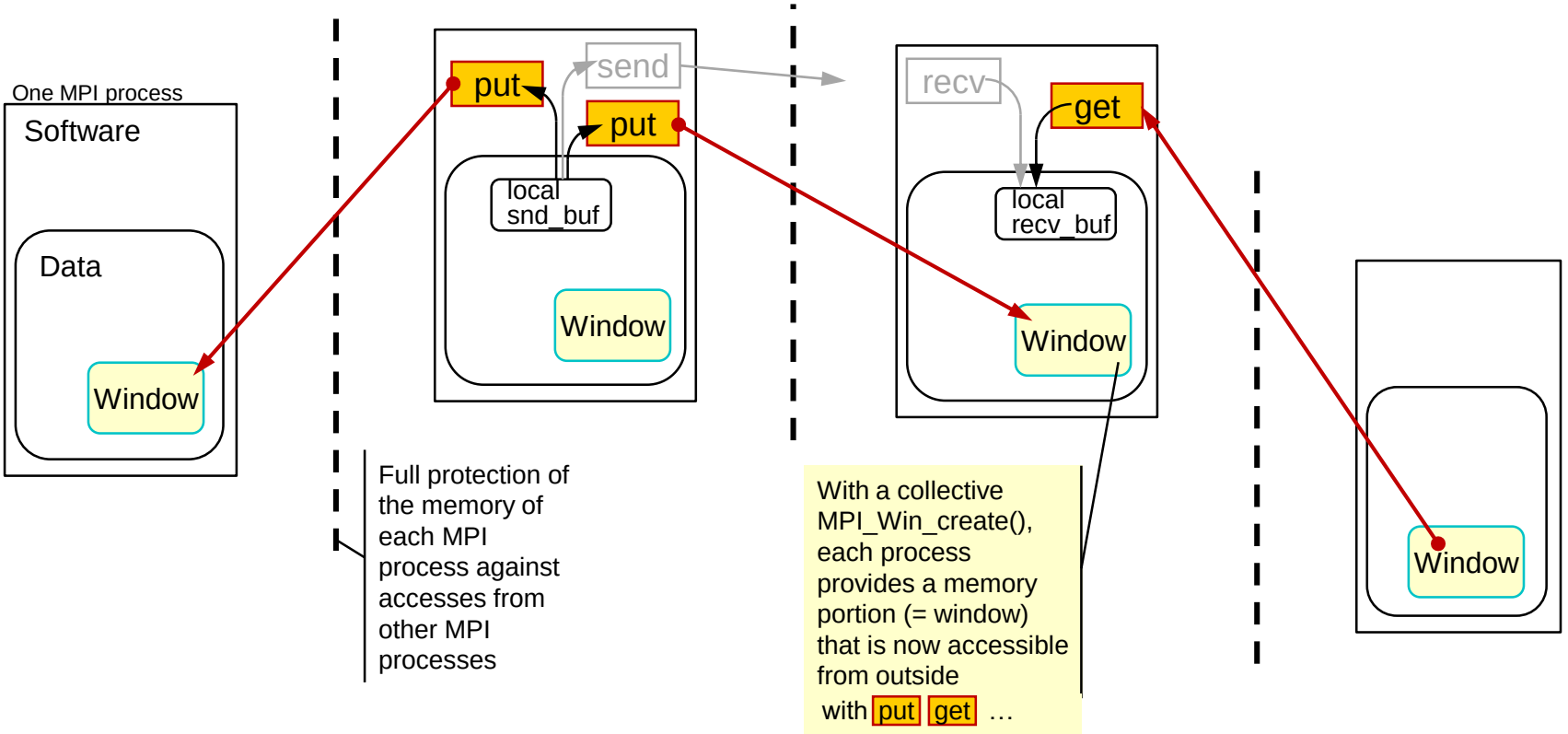
Typically, all processes are both, origin and target processes



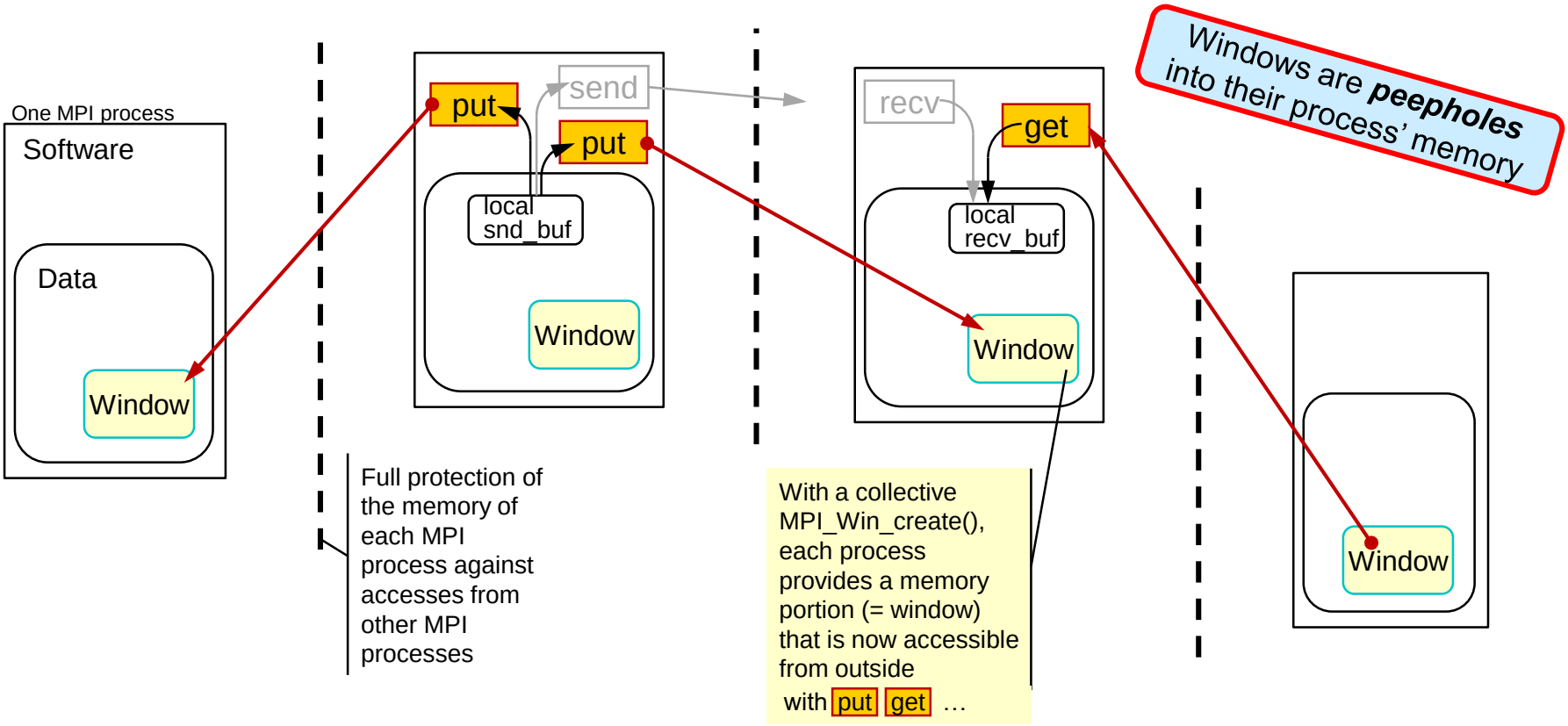
Typically, all processes are both, origin and target processes



Typically, all processes are both, origin and target processes



Typically, all processes are both, origin and target processes



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.

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.
- Remote Memory Access (RMA, nonblocking) routines
  - Access to remote windows: put, get, accumulate, …

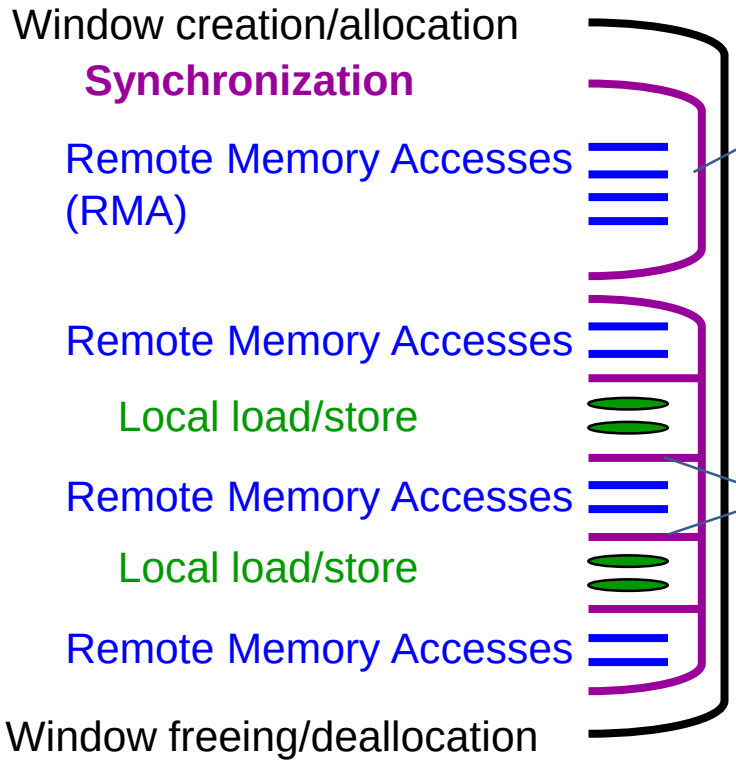
One-sided Operations

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.
- Remote Memory Access (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

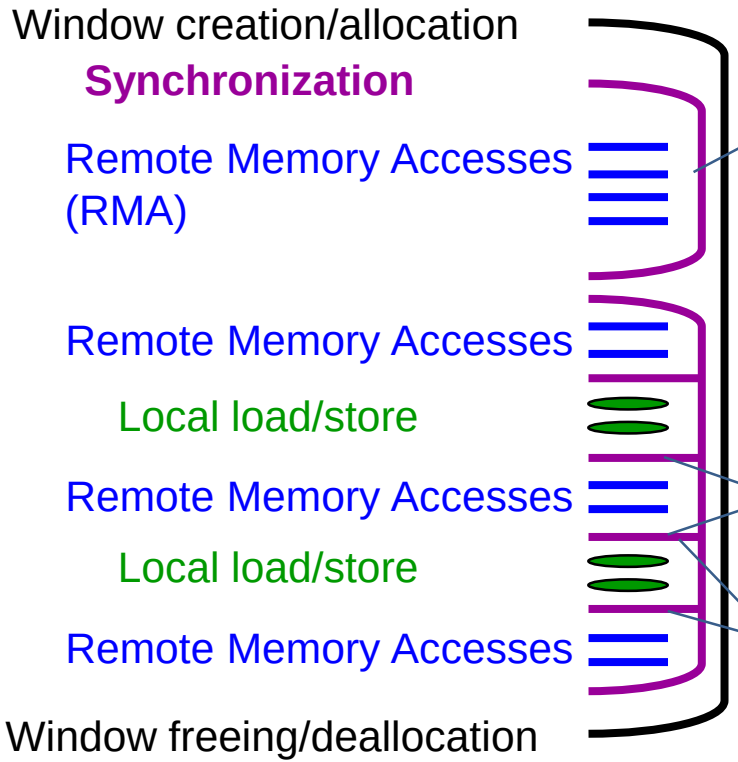
To start and finish **exposure** and **access** epochs

RMA epoch

Local load/store epoch

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

Sequence of One-sided Operations



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

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

RMA epoch

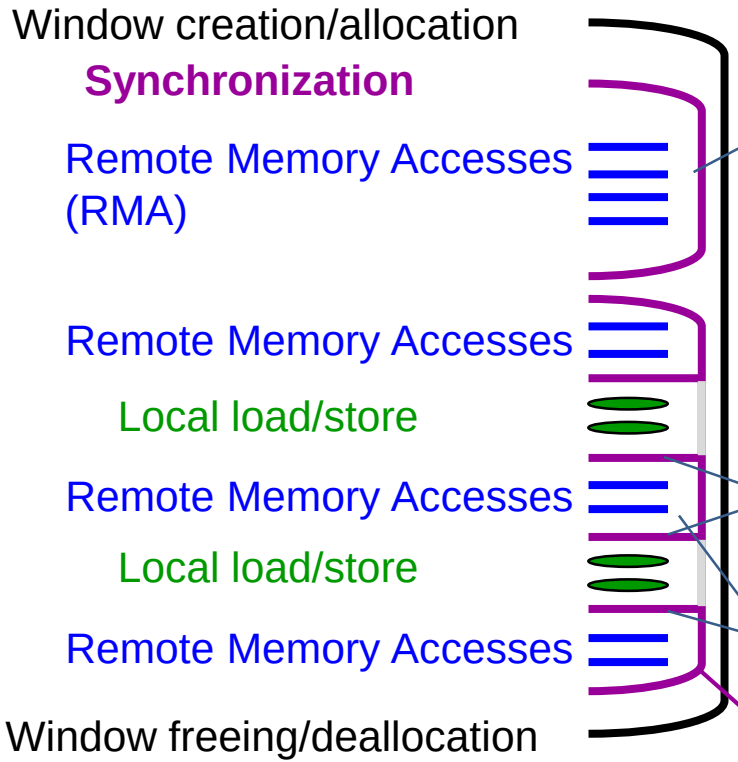
Local load/store epoch

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

It looks like that additionally local load/store epochs are also surrounded by synchronizations

But correct is: only **RMA epochs** must be **surrounded by synchronization** calls

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

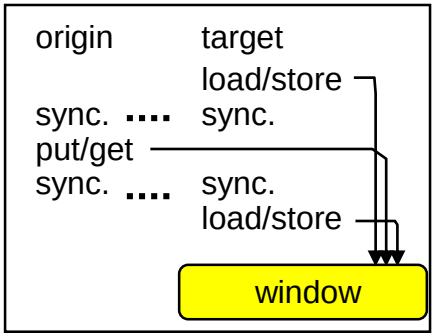
It looks like that additionally local load/store epochs are also surrounded by synchronizations

But correct is: only **RMA epochs** must be **surrounded by synchronization** calls

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

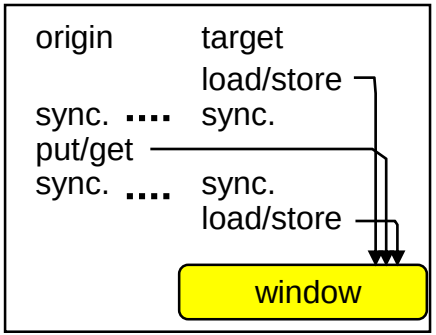
    MPI_Win_fence is like a barrier



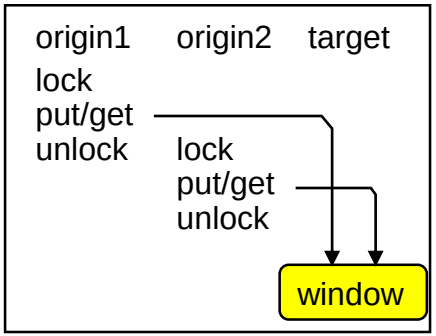
Synchronization Calls (1)

- 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

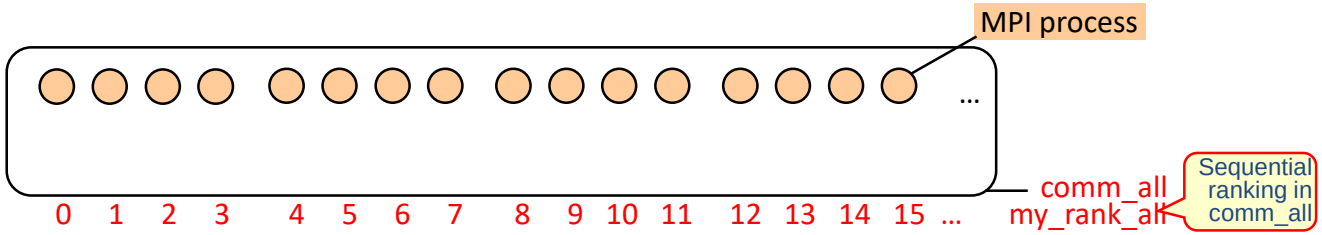
How-to

General considerations & uses cases
Re-cap: MPI_Comm_split & one-sided communication
> **How-to**
Exercise: MPI_Bcast
Quiz 1
MPI memory models & synchronization
Shared memory problems
Advantages & disadvantages, conclusions
Quiz 2

MPI shared memory

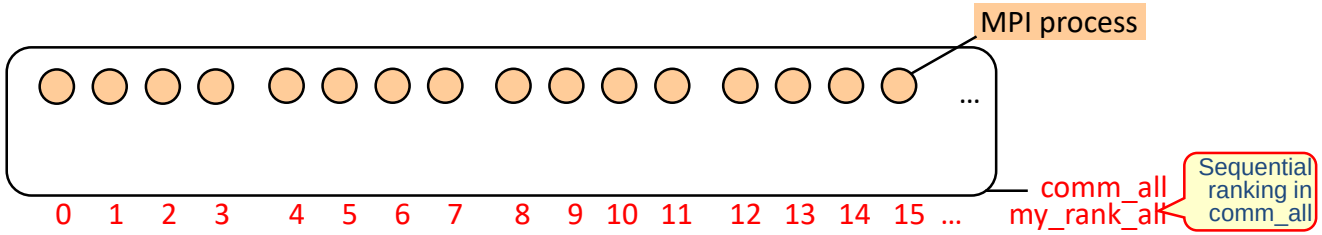
- Split main communicator into shared memory islands (automatically)
  - **MPI_Comm_split_type**
- Define a shared memory window on each island
  - **MPI_Win_allocate_shared**
  - Result (by default): contiguous array, directly accessible by all processes of the island
- Accesses and synchronization
  - This is normal memory: Language-based expressions and assignments
  - **MPI_PUT/GET** still allowed, but this is not the spirit!
  - 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 & shared memory allocation




```
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);
```

Splitting & shared memory allocation

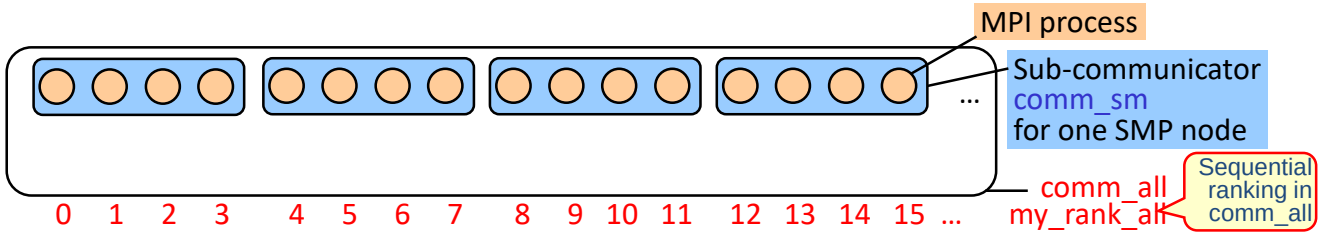



```
MPI_Comm comm_all, comm_sm;      int my_rank_all, my_rank_sm, size_sm, disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
        MPI_INFO_NULL, &comm_sm);
```

collective call

Sequence in comm_sm as in comm_all

Splitting & shared memory allocation

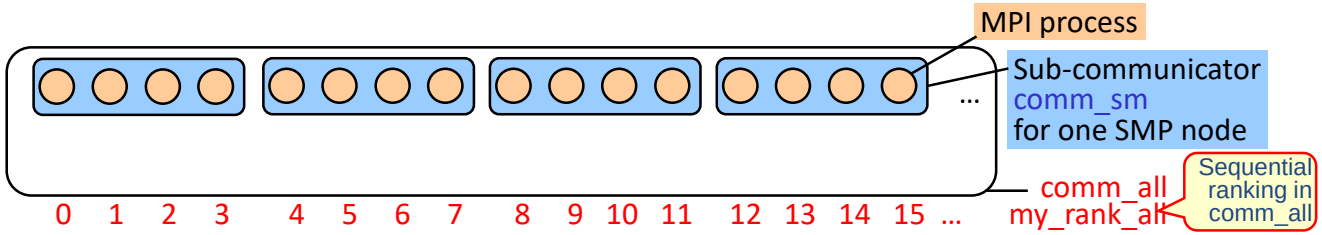



```
MPI_Comm comm_all, comm_sm;      int my_rank_all, my_rank_sm, size_sm, disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
                     MPI_INFO_NULL, &comm_sm);
```

collective call

Sequence in comm_sm as in comm_all

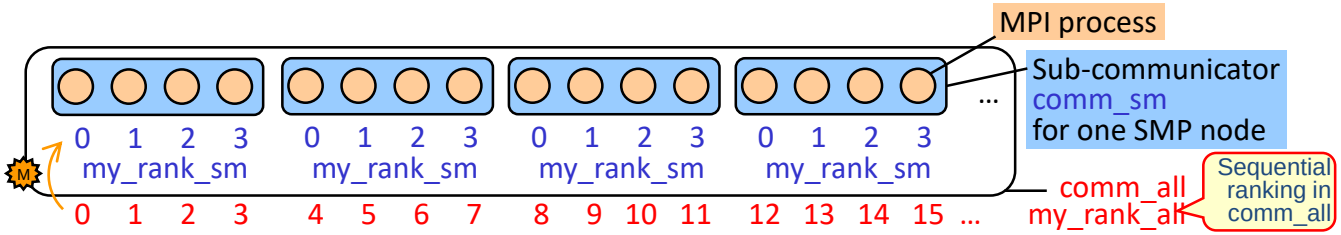
Splitting & shared memory allocation




```
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,
        collective call      MPI_INFO_NULL, &comm_sm);
MPI_Comm_rank (comm_sm, &my_rank_sm); MPI_Comm_size (comm_sm, &size_sm);
```

Sequence in comm_sm as in comm_all

Splitting & shared memory allocation



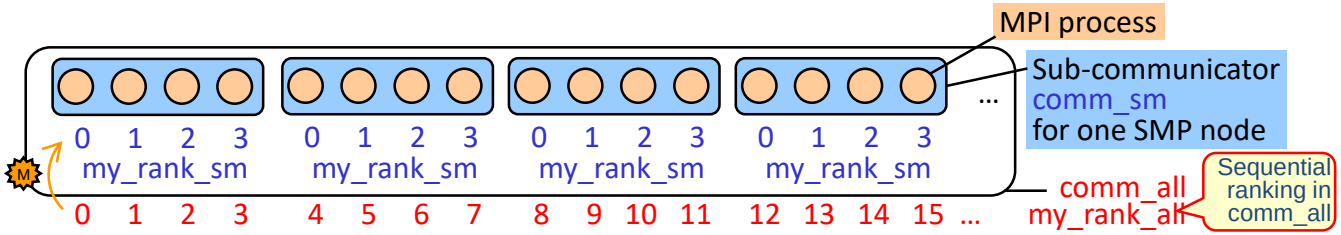

```
MPI_Comm comm_all, comm_sm;      int my_rank_all, my_rank_sm, size_sm, disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
          MPI_INFO_NULL, &comm_sm);
MPI_Comm_rank (comm_sm, &my_rank_sm); MPI_Comm_size (comm_sm, &size_sm);
```

collective call

Sequence in comm_sm as in comm_all

This mapping is based on the ranking in comm_all.

Splitting & shared memory allocation



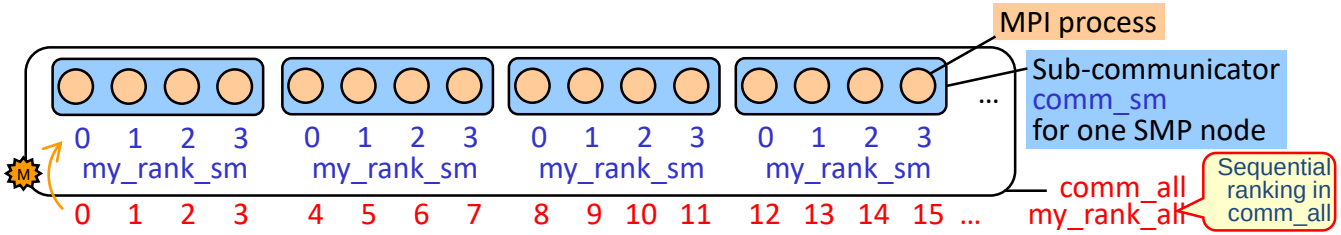

```
MPI_Aint /*IN*/ local_window_count=10;  double /*OUT*/  *base_ptr;
MPI_Comm  comm_all,  comm_sm;      int  my_rank_all,  my_rank_sm,  size_sm,  disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
          MPI_INFO_NULL,  &comm_sm);
MPI_Comm_rank (comm_sm, &my_rank_sm);  MPI_Comm_size (comm_sm, &size_sm);
```

collective call

Sequence in comm_sm as in comm_all

M This mapping is based on the ranking in comm_all.

Splitting & shared memory allocation

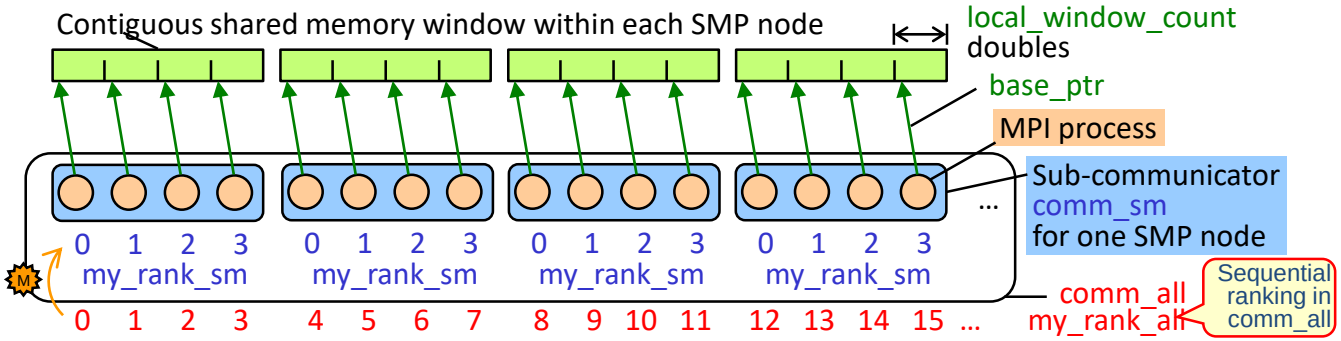



```
MPI_Aint /*IN*/ local_window_count=10;  double /*OUT*/  *base_ptr;
MPI_Comm  comm_all,  comm_sm;       int  my_rank_all,  my_rank_sm,  size_sm,  disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
          collective call       MPI_INFO_NULL,  &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,
          collective call             MPI_INFO_NULL,  comm_sm,  &base_ptr,  &win_sm);
```

Sequence in comm_sm as in comm_all

M This mapping is based on the ranking in comm_all.

Splitting & shared memory allocation



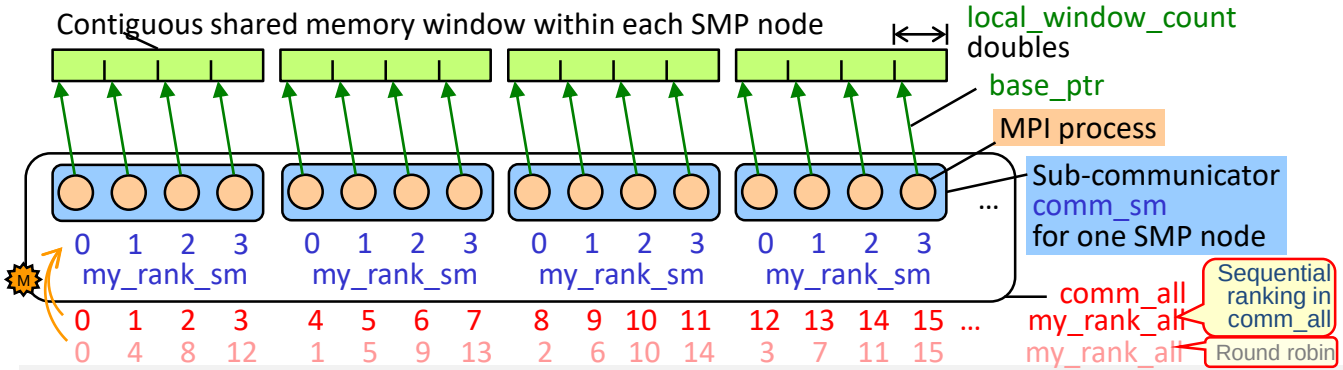

```
MPI_Aint /*IN*/ local_window_count=10;  double /*OUT*/  *base_ptr;
MPI_Comm  comm_all,  comm_sm;       int  my_rank_all,  my_rank_sm,  size_sm,  disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
        collective call      MPI_INFO_NULL,  &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,
        collective call      MPI_INFO_NULL,  comm_sm,  &base_ptr,  &win_sm);
```

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.

Splitting & 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,
        collective call      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,
        collective call      MPI_INFO_NULL,  comm_sm,  &base_ptr,  &win_sm);
```

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.

Shared-memory allocation in Fortran uses C pointer!

New in MPI-3.0

In all three Fortran support methods

C

```
float *buf;  MPI_Win win; int max_length; max_length = … /* = array size in elements */;
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]
```

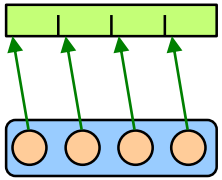
Fortran

```
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)
```

Translates C pointer to std Fortran pointer

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 shared-memory island: 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
- Remote accesses through load/store operations,
  - i.e., without MPI RMA operations (MPI_Get/Put, …)

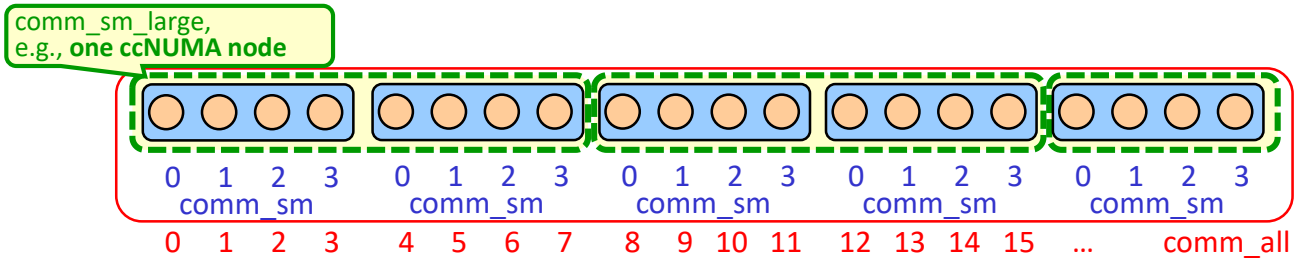
- Caution:
  Although each process in comm_sm accesses the same physical memory, the virtual start address of the whole array may be different in all processes!

  → **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., splitting into 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 1 or 2

Splitting into smaller shared memory islands

- 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"

May not work with Intel-MPI

- Two additional standardized split types:
  - MPI_COMM_TYPE_HW_GUIDED
  - MPI_COMM_TYPE_HW_UNGUIDED

New in MPI-4.0

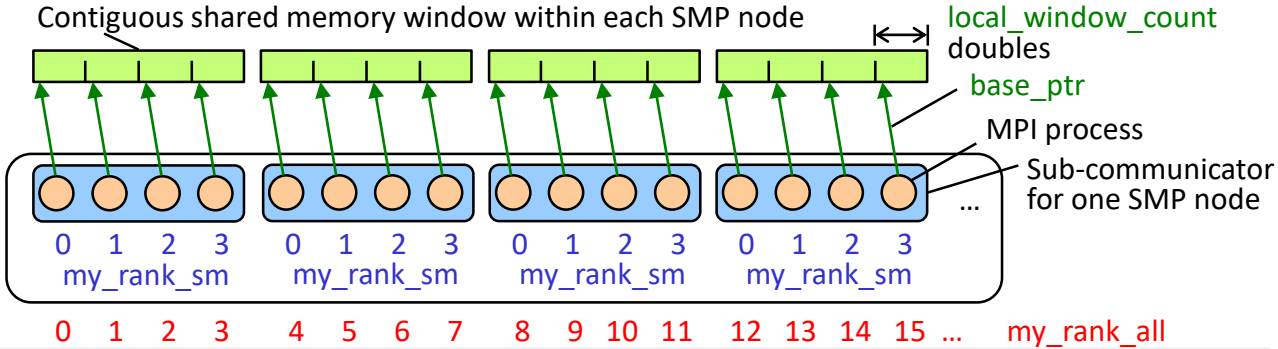
May be fixed in MPI-4.1

Drawback: no standardized key values

Drawback:
- two splits are needed
  - 1st with MPI_COMM_TYPE_SHARED
  - 2nd with MPI_COMM_TYPE_HW_UNGUIDED
- problematic if number of NUMA domains is not identical in all shared memory islands of 1st split

- See also Exercise 3.

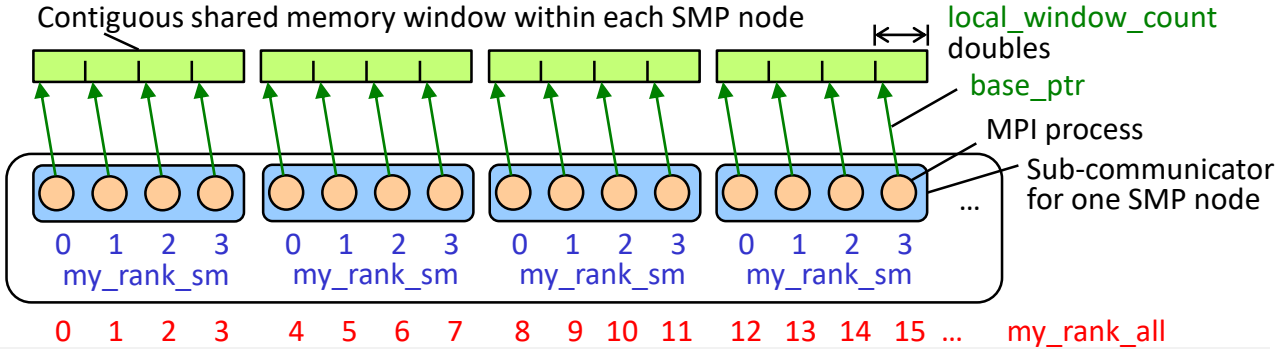
Shared memory access example




```
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);
```

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (**https://www.hlrs.de/training/self-study-materials**)

Shared memory access example

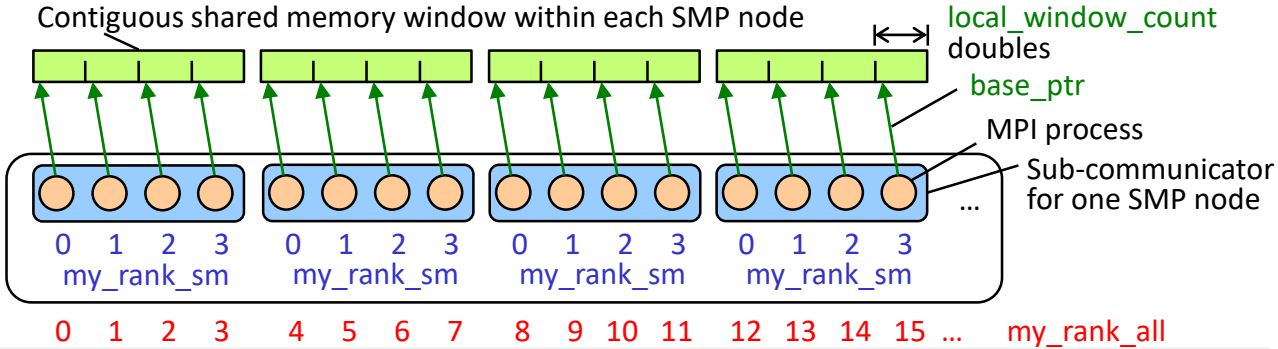



```
MPI_Aint /*IN*/ local_window_count;       double /*OUT*/  *base_ptr;
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                         MPI_INFO_NULL, comm_sm,  &base_ptr,  &win_sm);
MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
```

Synchroni-zation

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example



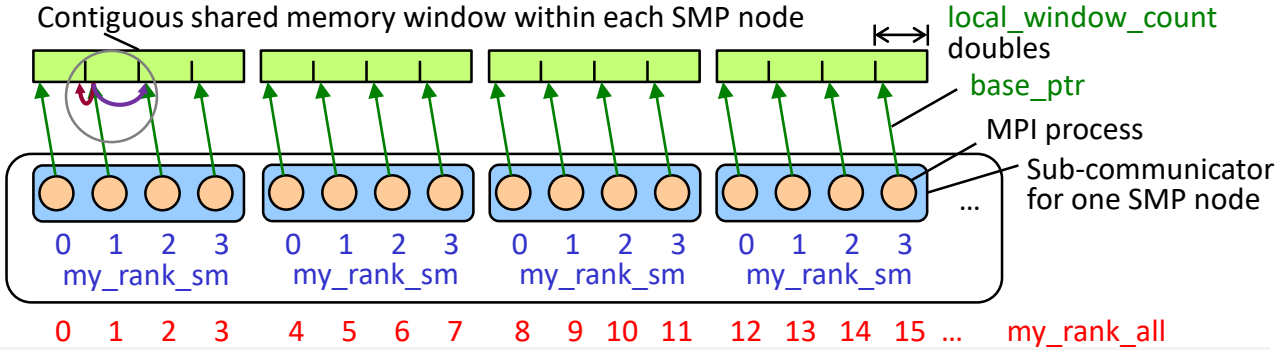

```
MPI_Aint /*IN*/ local_window_count;        double /*OUT*/  *base_ptr;
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                         MPI_INFO_NULL, comm_sm,  &base_ptr,  &win_sm);
MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
for (i=0; i<local_window_count; i++)  base_ptr[i] = ... /* fill values into local portion */
```

Synchroni-zation

Local stores

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example



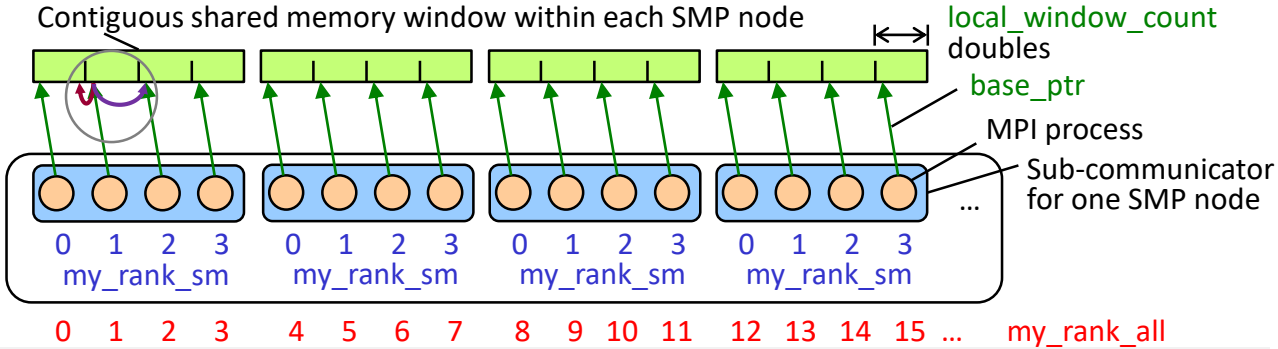

```
MPI_Aint /*IN*/ local_window_count;        double /*OUT*/  *base_ptr;
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
                         MPI_INFO_NULL, comm_sm,  &base_ptr,  &win_sm);
MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
for (i=0; i<local_window_count; i++)  base_ptr[i] = ... /* fill values into local portion */
```

Synchroni-zation

Local stores

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example




```
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);
MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
for (i=0; i<local_window_count; i++)  base_ptr[i] = ... /* fill values into local portion */
MPI_Win_fence (0, win_sm);  /* local stores are finished, remote load epoch can start */
```

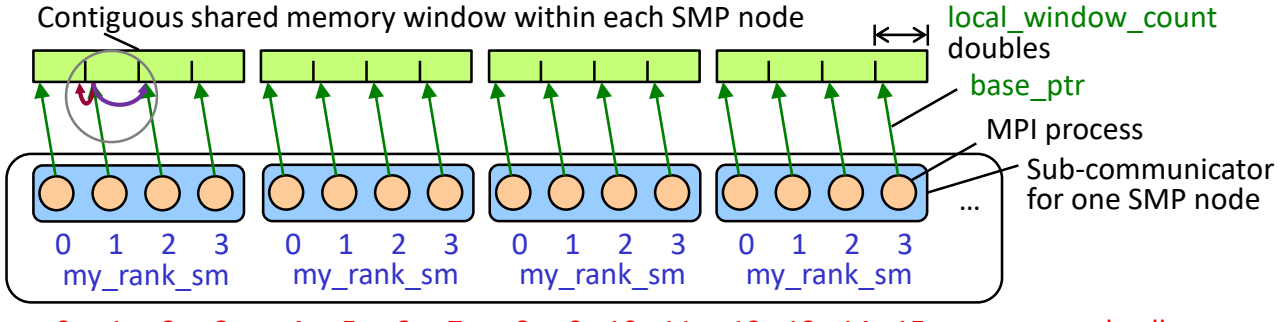
Synchroni-zation

Synchroni-zation

Local stores

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example




```
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);

MPI_Win_fence (0, win_sm);  /*local store epoch can start*/
for (i=0; i<local_window_count; i++)  base_ptr[i] = ... /* fill values into local portion */
MPI_Win_fence (0, win_sm);  /* local stores are finished, remote load epoch can start */
if (my_rank_sm > 0)              printf("left neighbor's rightmost value = %lf \n", base_ptr[-1] );
```

Synchroni-zation

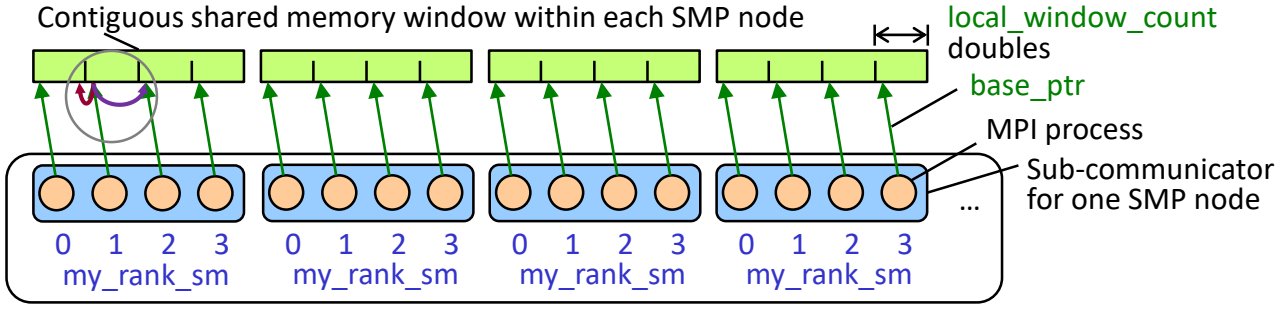
Synchroni-zation

Local stores

Direct load access to remote window portion

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example




```
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);

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

Synchroni-zation

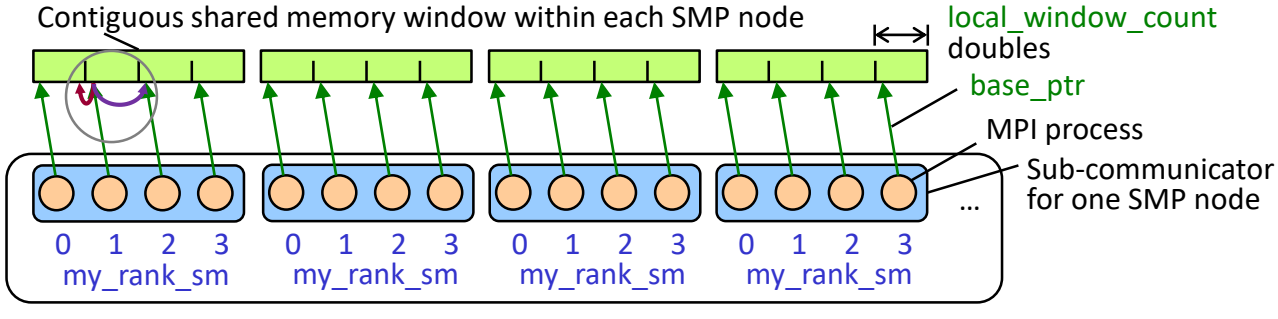
Synchroni-zation

Local stores

Direct load access to remote window portion

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example




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

Synchroni-zation

Synchroni-zation

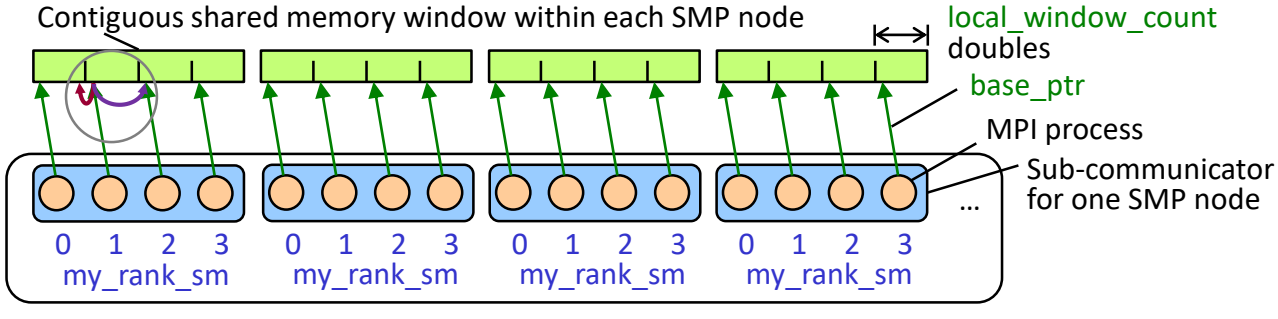
Local stores

Direct load access to remote window portion

Such **out of bound addressing** is only available in C and Fortran..

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (https://www.hlrs.de/training/self-study-materials)

Shared memory access example




```
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);

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

Synchroni-zation

Synchroni-zation

Local stores

Direct load access to remote window portion

Such **out of bound addressing** is only available in C and Fortran..

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".

see High Performance Computing Center Stuttgart (HLRS) → Self-Study Materials → MPI-Course material → end of Chapter 4 (**https://www.hlrs.de/training/self-study-materials**)

Alternative: Non-contiguous shared memory

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

Pros: Faster local data accesses especially on ccNUMA nodes

Cons: Higher programming effort for neighbor accesses: MPI_WIN_SHARED_QUERY

NUMA effects?
Significant impact of alloc_shared_noncontig

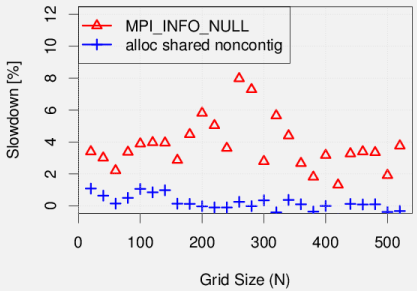


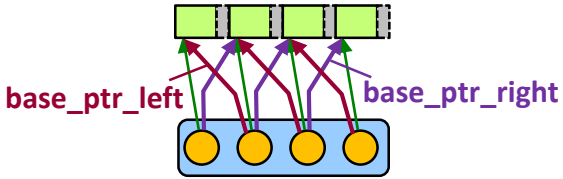
Image: Courtesy of Torsten Hoefler

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

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



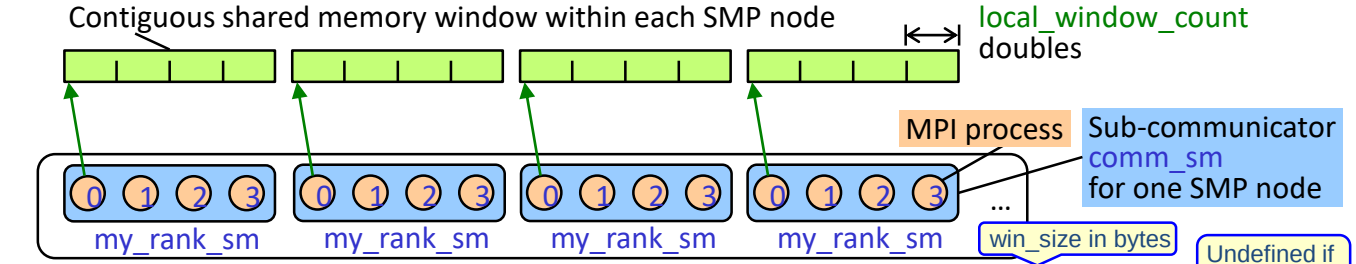
local call

```
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.

skipped

Whole shared memory allocation by rank 0 in comm_sm

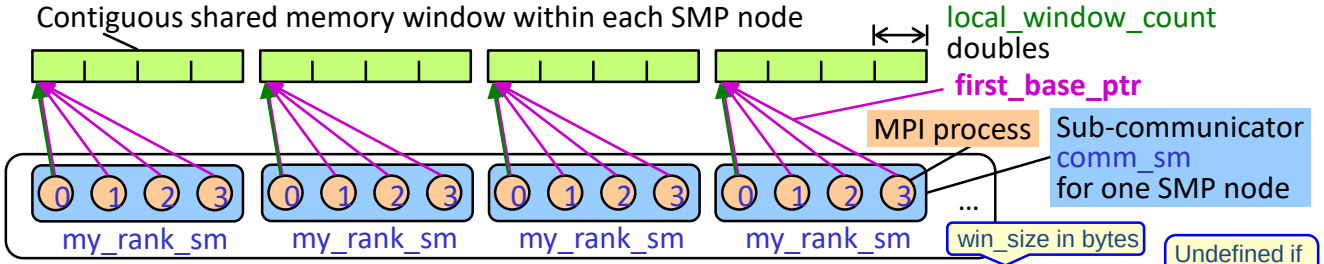


if (my_rank_sm==0) win_size = local_window_count*disp_unit***size_sm else win_size = 0;**

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

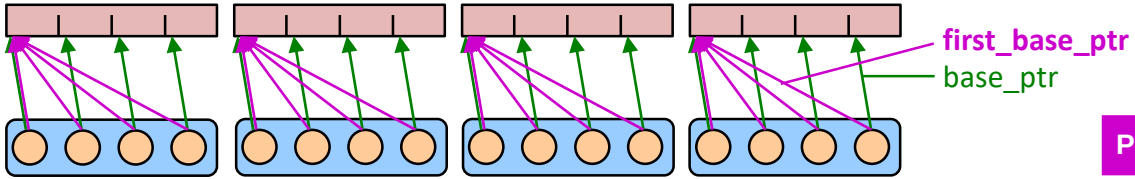
skipped

Whole shared memory allocation by rank 0 in comm_sm




```
if (my_rank_sm==0) win_size = local_window_count*disp_unit*size_sm else win_size = 0;
MPI_Win_allocate_shared (win_size,  disp_unit, MPI_INFO_NULL, comm_sm, &base_ptr, &win_sm);
MPI_Win_shared_query (win_sm, /*rank=*/ 0, &win_size,  &disp_unit,  &first_base_ptr);
```

Describes the whole array



Python

only for Python, we use this **first_base_ptr** to define the **recv_buf array** in Exercise 2

```
win_size = local_window_count*disp_unit*size_sm;
MPI_Win_allocate_shared (win_size,  disp_unit, MPI_INFO_NULL, comm_sm, &base_ptr, &win_sm);
MPI_Win_shared_query (win_sm, /*rank=*/ 0, &win_size,  &disp_unit,  &first_base_ptr);
```

Describes only first portion

CAUTION: Aliasing may be forbidden in your programming language, i.e., within one process, do not access the same window element through two different pointers. **Recommendation here**: use ↘ to access the *own* window portion, and use ↖ to access *remote* elements.

Other technical aspects with MPI_Win_allocate_shared

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

may be limited.

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

to enlarge restricting defaults.

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

due to default limit of context IDs in mpich

On some systems: No limits.

On a system without virtual memory you have to reserve a chunk of address space when the node is booted (at job script launch).

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

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

Questions addressed in this tutorial

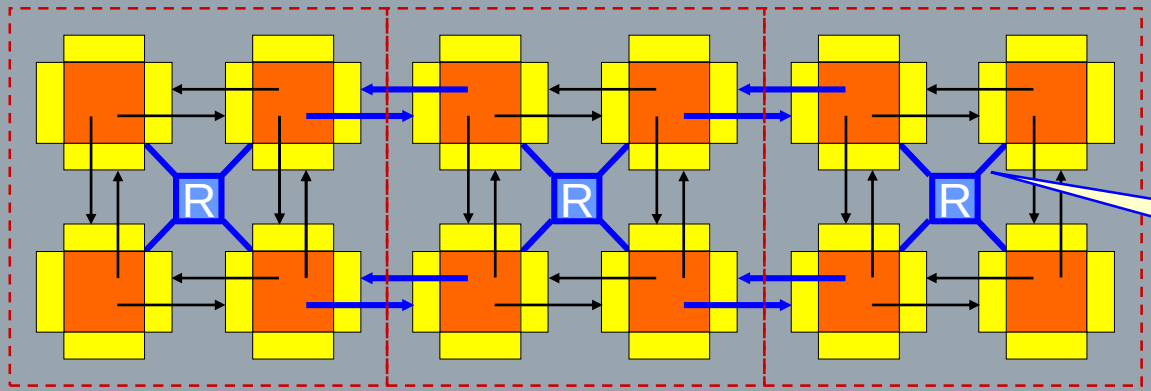
Where we are?

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

- How can I leverage multiple accelerators?
  - What are typical challenges?

Programming models
- MPI + MPI-3.0 shared memory
Exercise:
MPI_Bcast into shared memory islands



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

Direct loads & stores,
no library calls

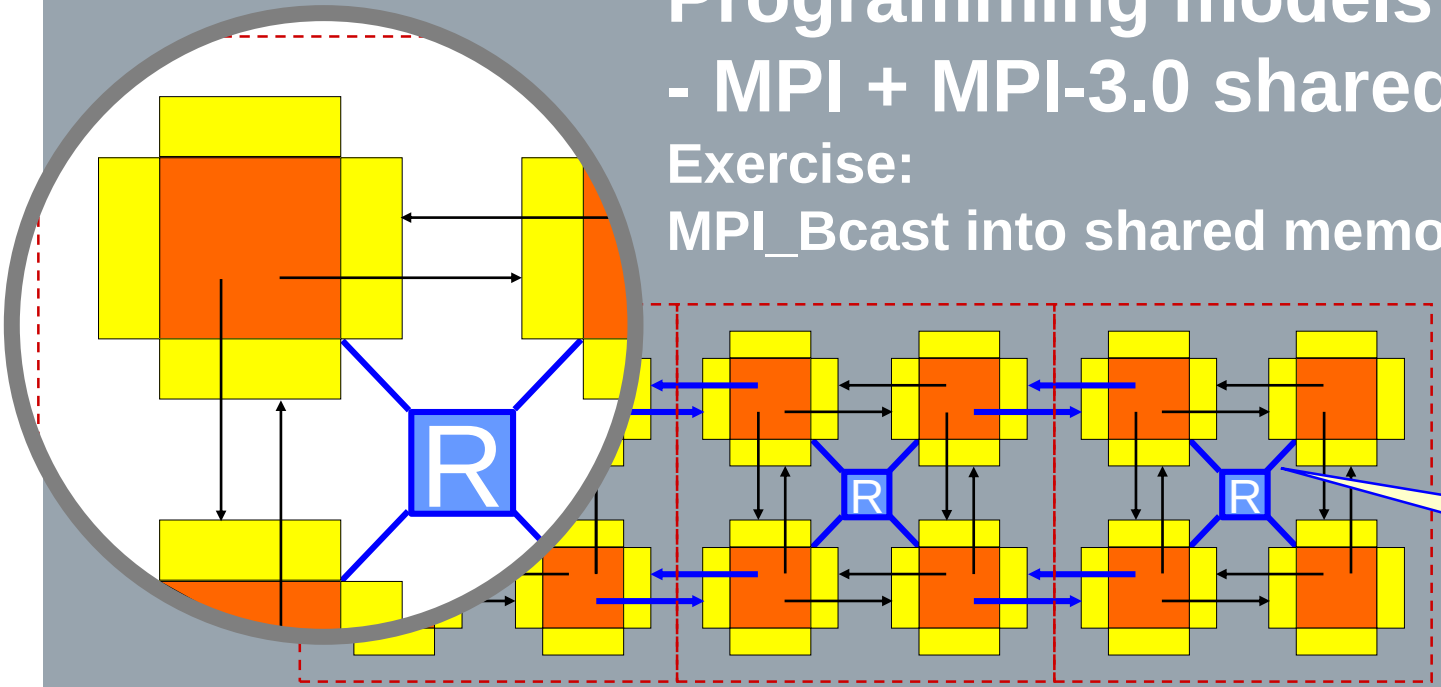
Using MPI shared memory methods

http://tiny.cc/MPIX-VSC

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

Programming models
- MPI + MPI-3.0 shared memory

Exercise:
MPI_Bcast into shared memory islands



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

Direct loads & stores,
no library calls

Using MPI shared memory methods

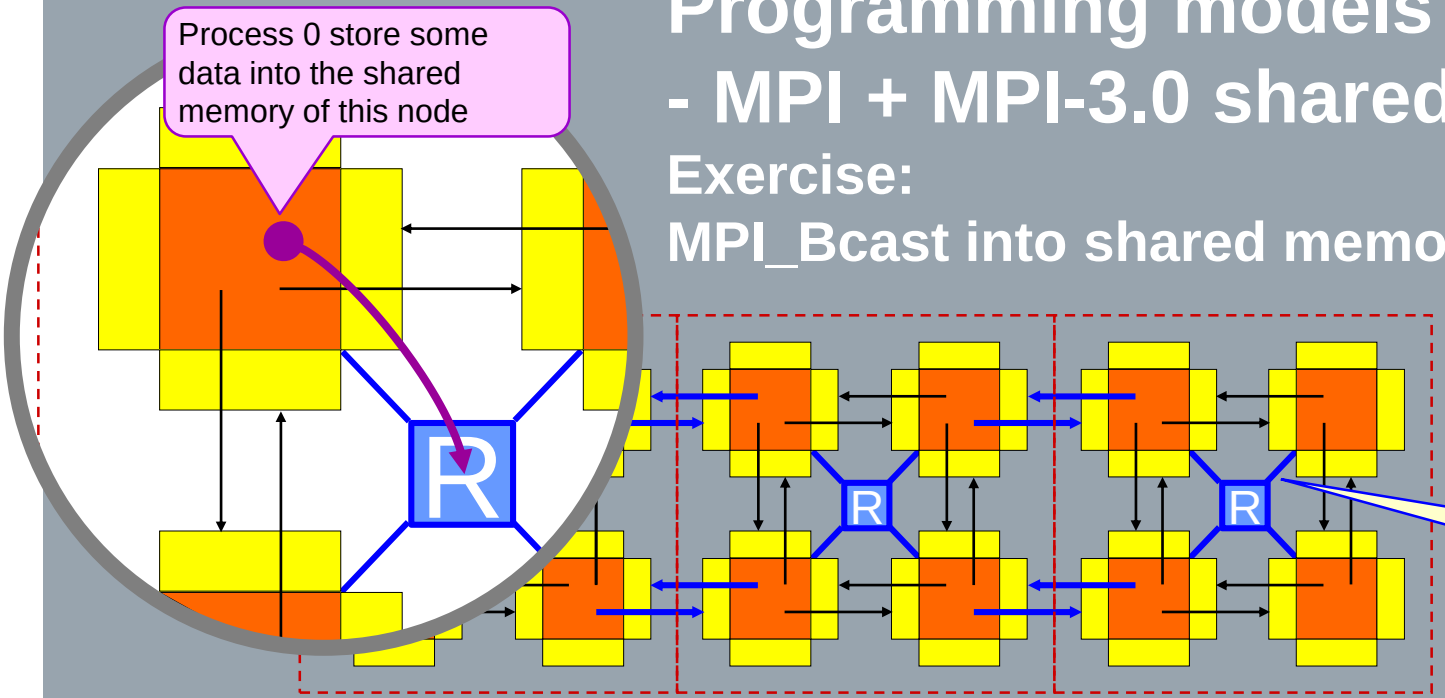
http://tiny.cc/MPIX-VSC

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

Programming models - MPI + MPI-3.0 shared memory

Exercise:

MPI_Bcast into shared memory islands



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

Direct loads & stores, no library calls

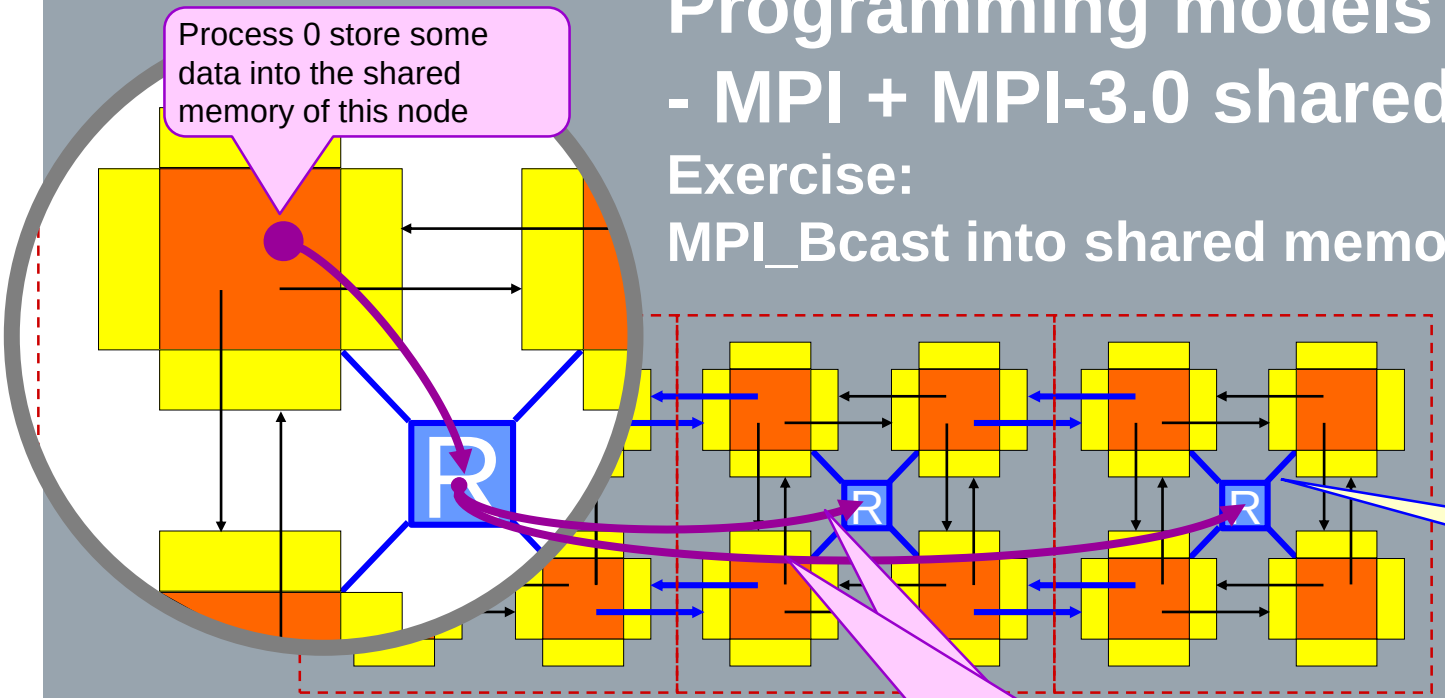
Using MPI shared memory methods

http://tiny.cc/MPIX-VSC

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

Programming models - MPI + MPI-3.0 shared memory

Exercise:
MPI_Bcast into shared memory islands



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

Direct loads & stores, no library calls

Using MPI shared memory methods

Broadcast to all other nodes (by only one process per node)

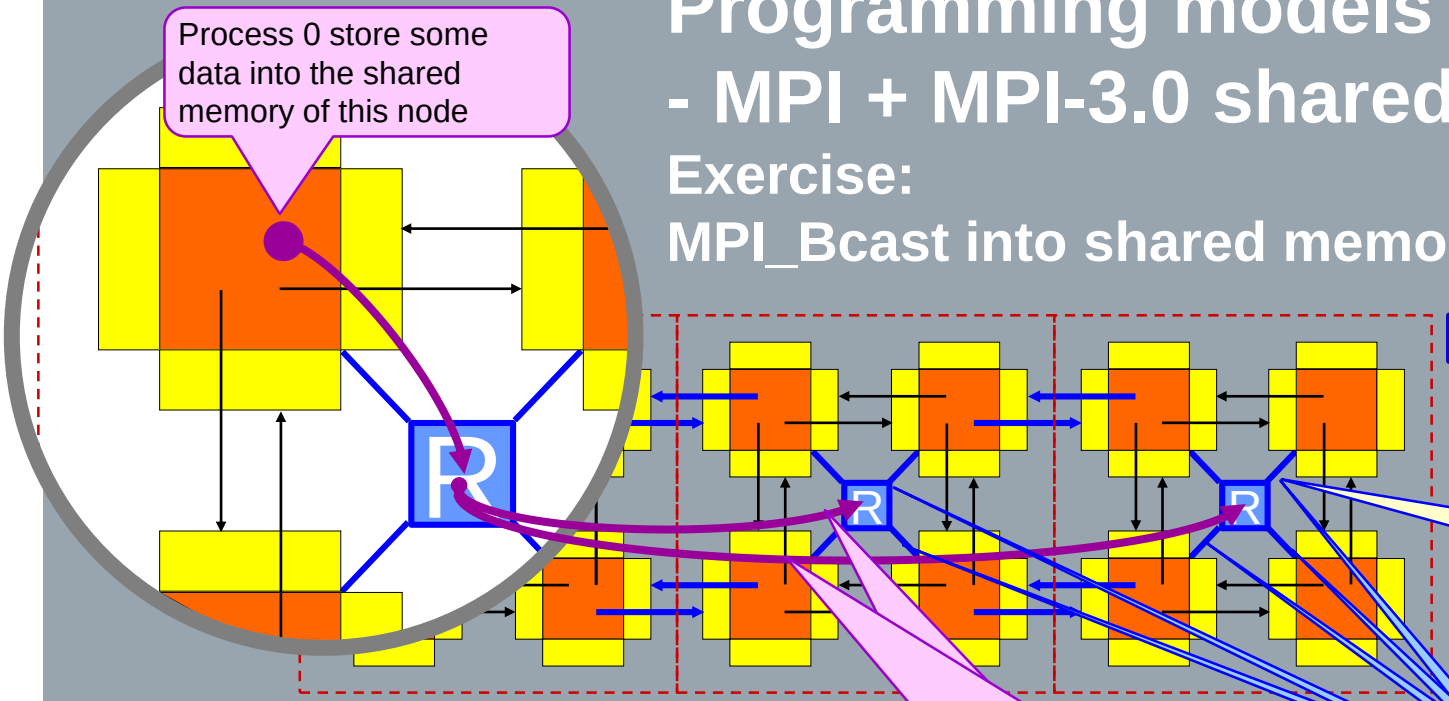
http://tiny.cc/MPIX-VSC

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

Programming models - MPI + MPI-3.0 shared memory

Exercise:

MPI_Bcast into shared memory islands



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

Direct loads & stores, no library calls

Using MPI shared memory methods

Broadcast to all other nodes (by only one process per node)

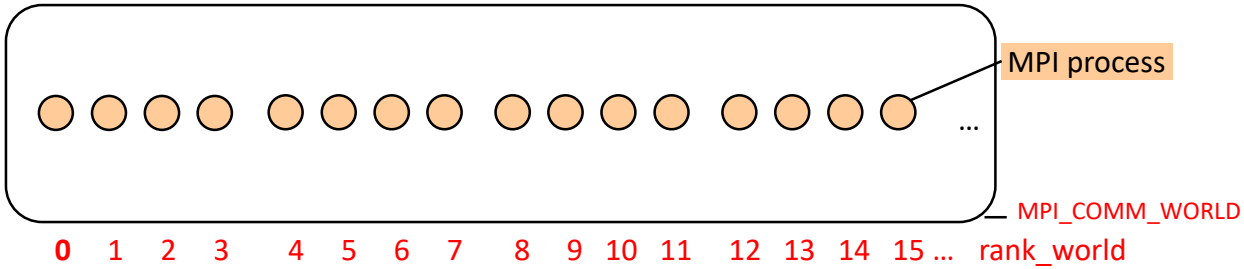
Finally, each process can read the shared data.

http://tiny.cc/MPIX-VSC

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

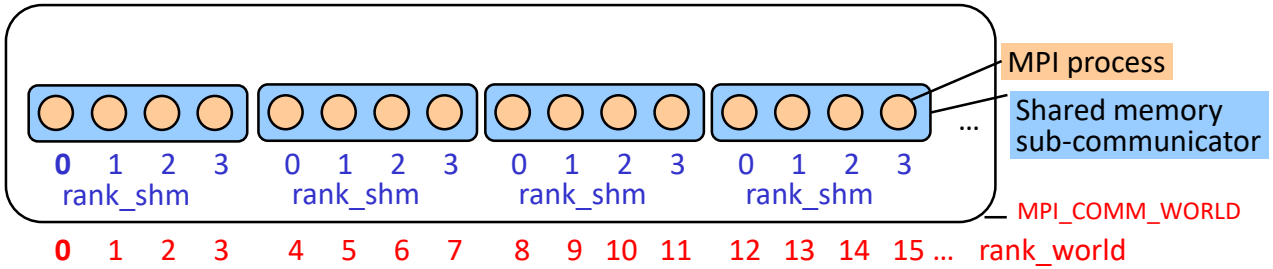
Exercise: MPI_Bcast into shared memory

- Now illustrated as in the previous slides



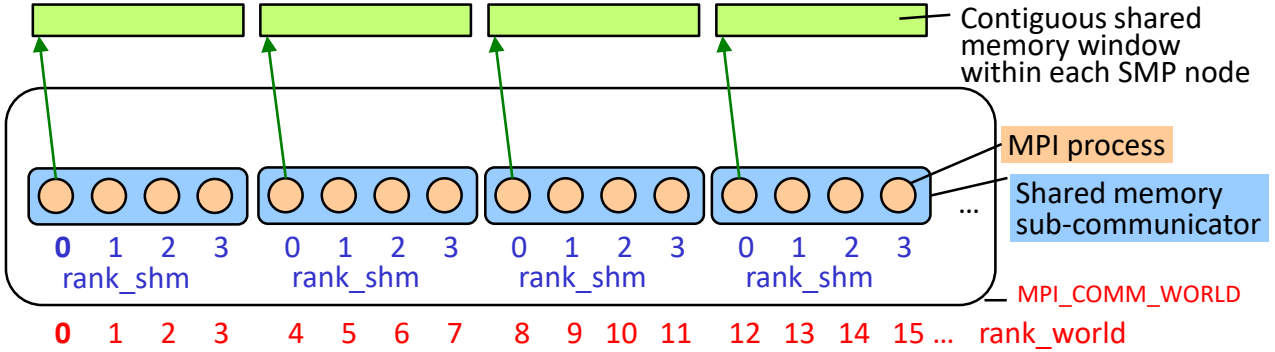
Exercise: MPI_Bcast into shared memory

- Now illustrated as in the previous slides



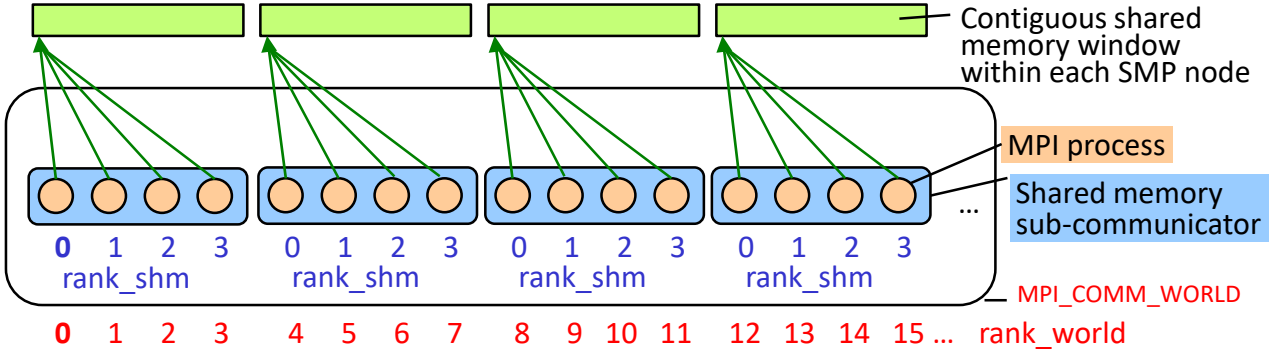
Exercise: MPI_Bcast into shared memory

- Now illustrated as in the previous slides
- Each represents such a replicated memory R within an island



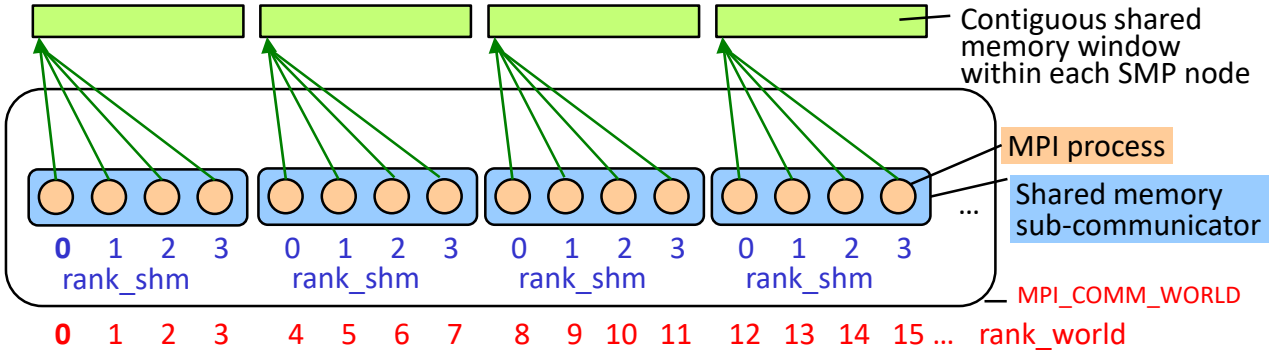
Exercise: MPI_Bcast into shared memory

- Now illustrated as in the previous slides
- Each represents such a replicated memory R within an island



Exercise: MPI_Bcast into shared memory

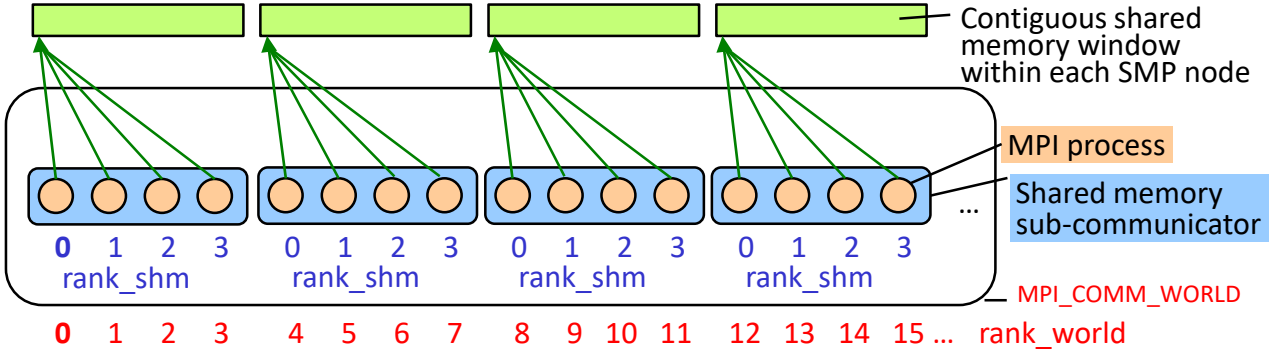
- Now illustrated as in the previous slides
- Each ▭ represents such a replicated memory R within an island



- Application: We'll store numbers 1, 2, … into the green array by process 0

Exercise: MPI_Bcast into shared memory

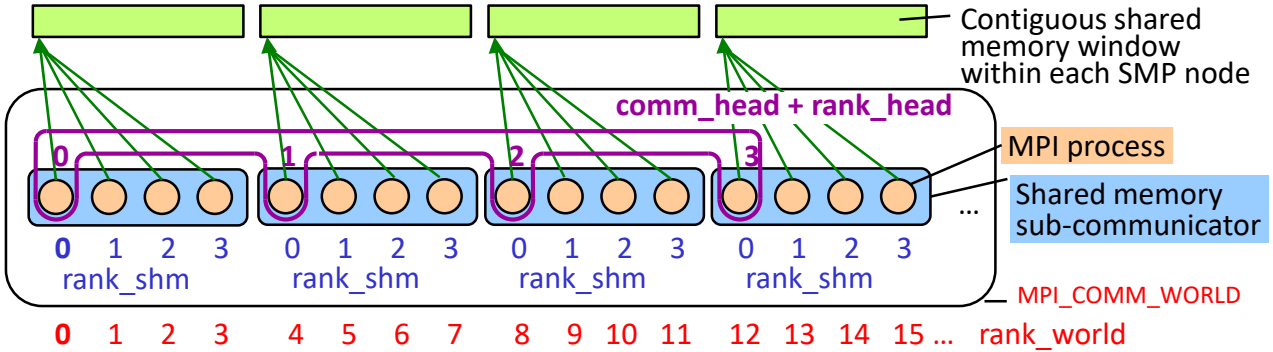
- Now illustrated as in the previous slides
- Each represents such a replicated memory R within an island



- Application: We'll store numbers 1, 2, … into the green array by process 0
- And then bcast it to all other shared memory islands

Exercise: MPI_Bcast into shared memory

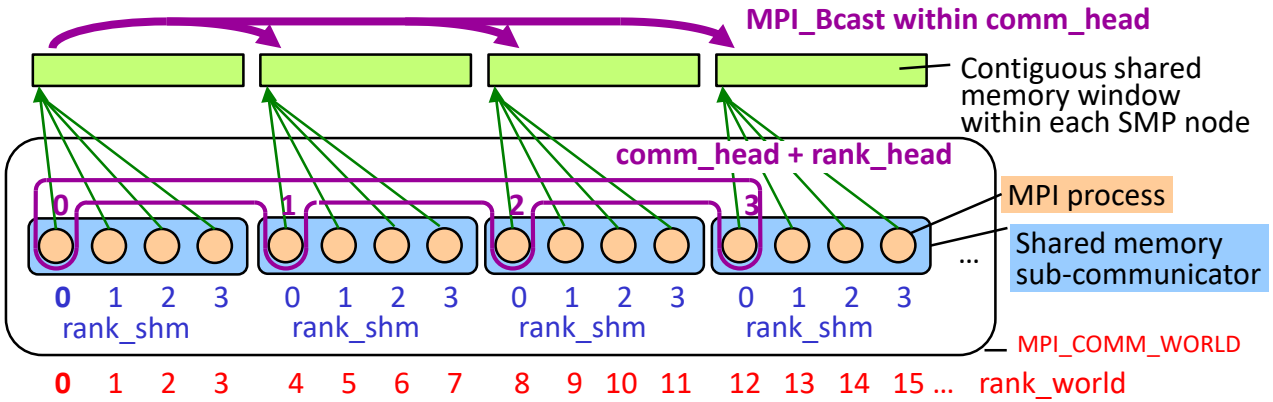
- Now illustrated as in the previous slides
- Each ▭ represents such a replicated memory R within an island



- Application: We'll store numbers 1, 2, … into the green array by process 0
- And then bcast it to all other shared memory islands

Exercise: MPI_Bcast into shared memory

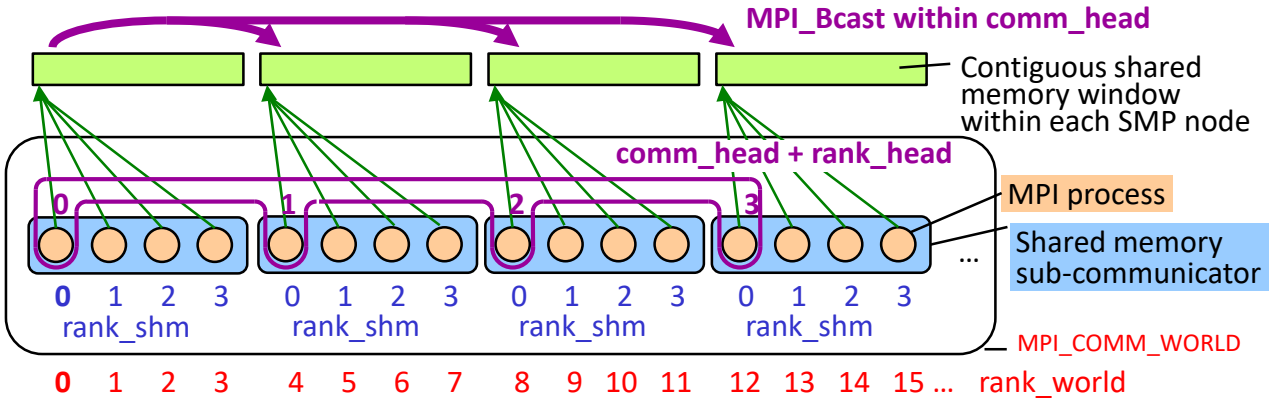
- Now illustrated as in the previous slides
- Each ▭ represents such a replicated memory R within an island



- Application: We'll store numbers 1, 2, … into the green array by process 0
- And then bcast it to all other shared memory islands

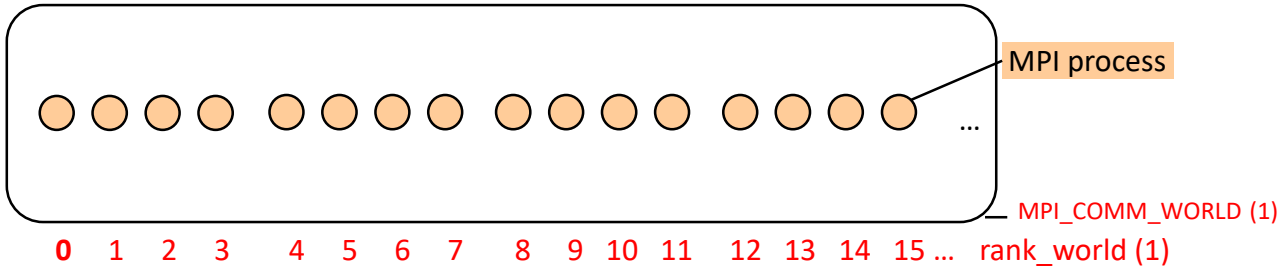
Exercise: MPI_Bcast into shared memory

- Now illustrated as in the previous slides
- Each represents such a replicated memory R within an island



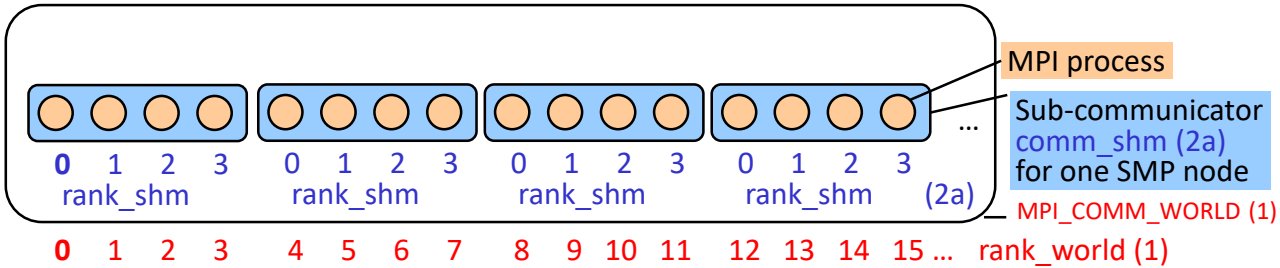
- Application: We'll store numbers 1, 2, … into the green array by process 0
- And then bcast it to all other shared memory islands
- At the end, each process calculates the sum of all numbers within *its* shared memory

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



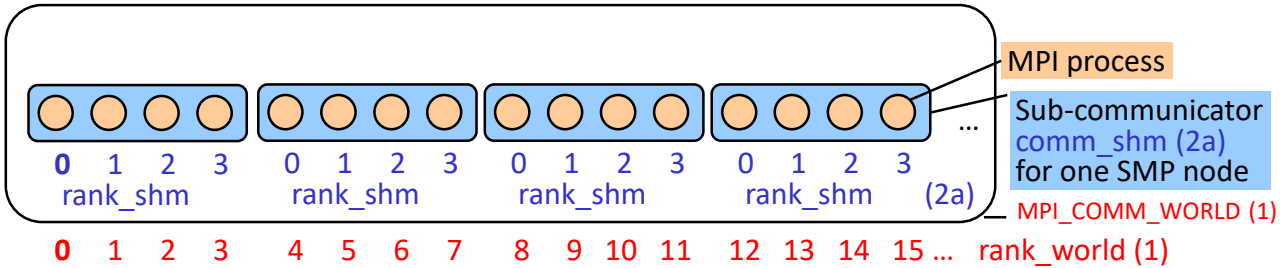
(1) Given: arrSize, MPI_COMM_WORLD → rank_world

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



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

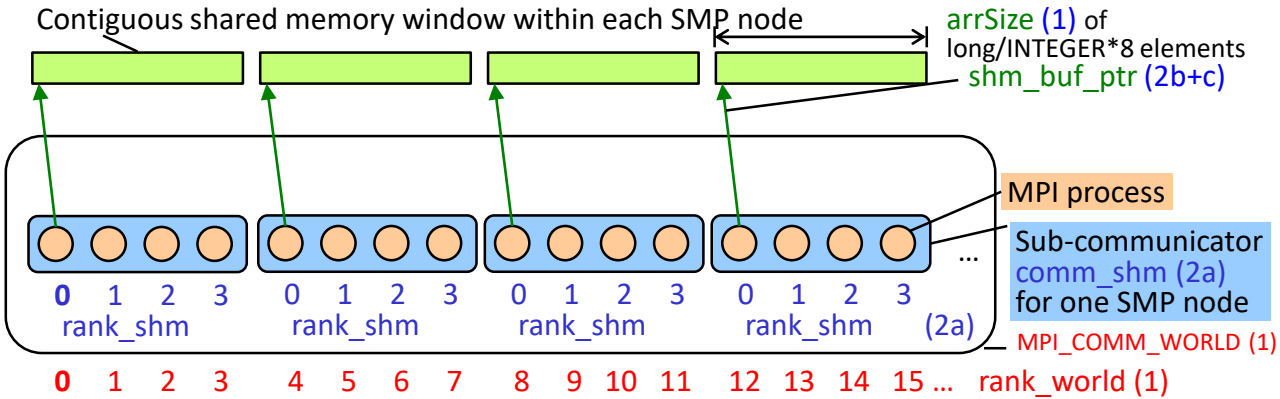
Exercise steps:
(1-2) The allocation of the shared memory within each node



(1) **Given: arrSize, MPI_COMM_WORLD → rank_world**

(2a) **MPI_Comm_split_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm**

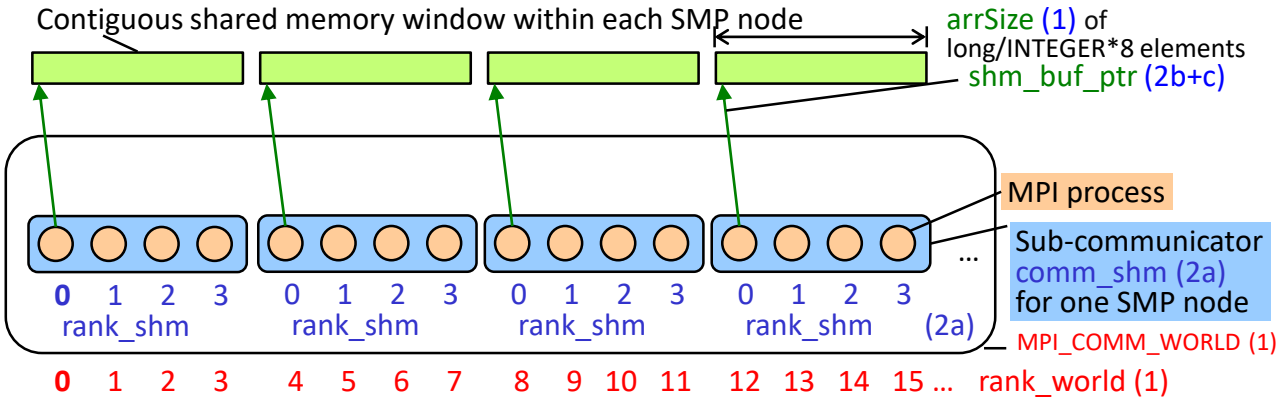
Exercise steps:
(1-2) The allocation of the shared memory within each node



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

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

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

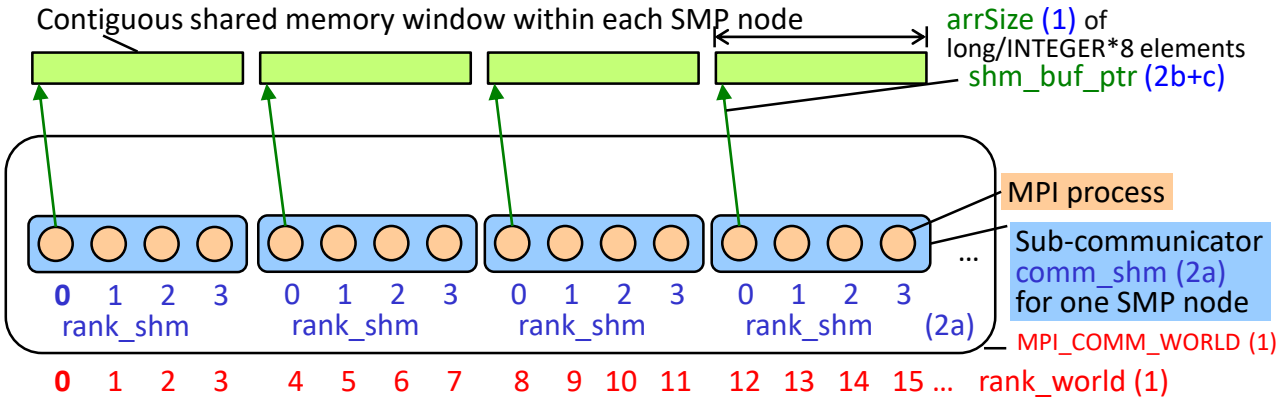


(1) **Given: arrSize, MPI_COMM_WORLD → rank_world**

(2a) **MPI_Comm_split_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm**

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

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



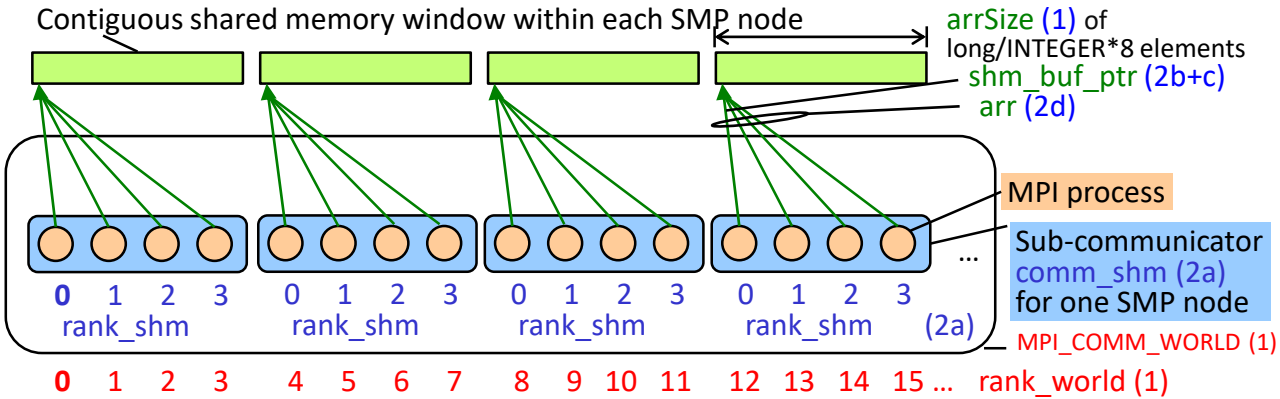
(1) **Given: arrSize, MPI_COMM_WORLD → rank_world**

(2a) **MPI_Comm_split_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm**

(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))**

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



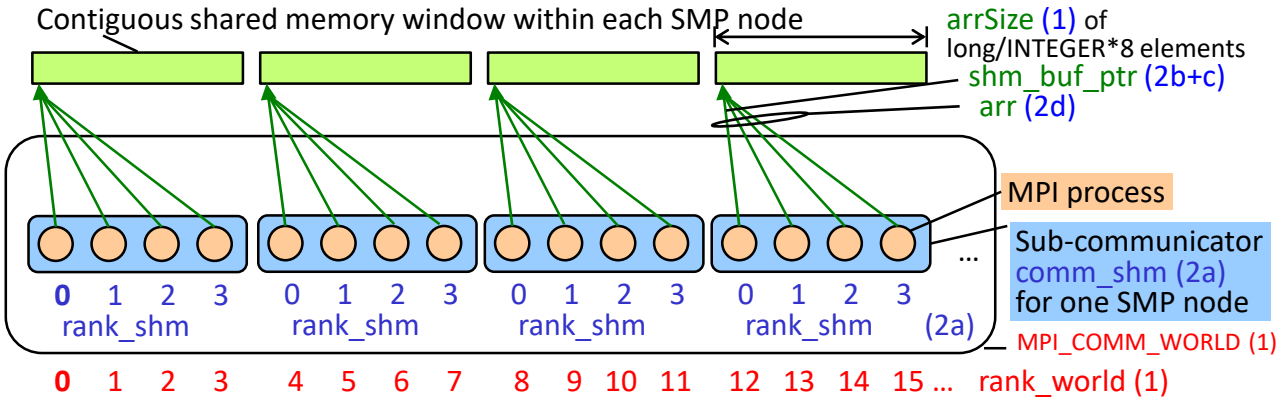
(1) Given: arrSize, MPI_COMM_WORLD → rank_world

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

(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))

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



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

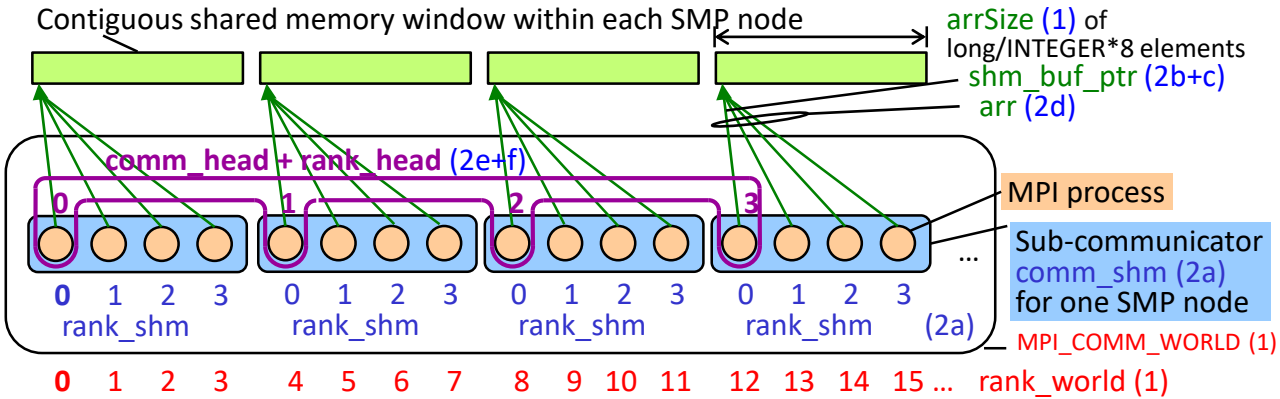
(2a) MPI_Comm_split_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm

(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);

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



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

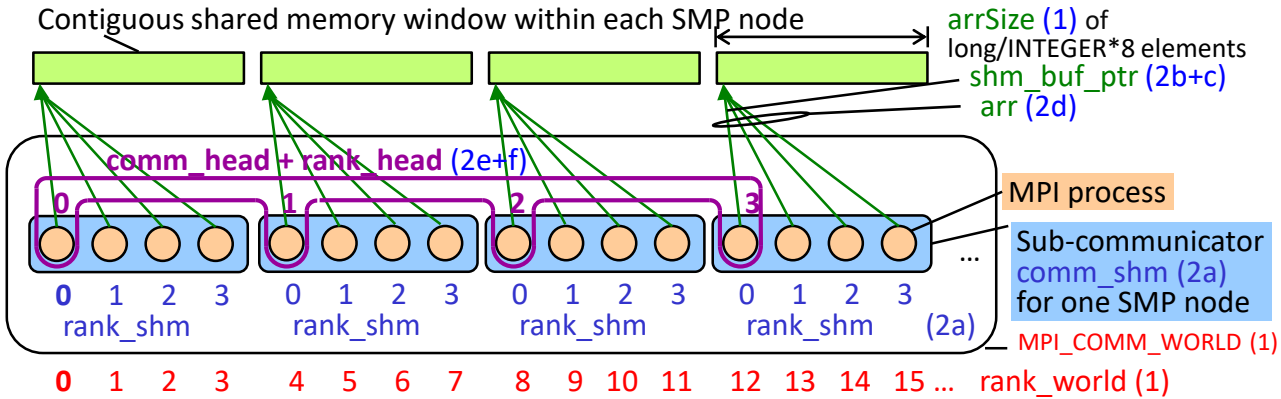
(2a) MPI_Comm_split_type(key=0) → comm_shm → MPI_Comm_rank() → rank_shm

(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);

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



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

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

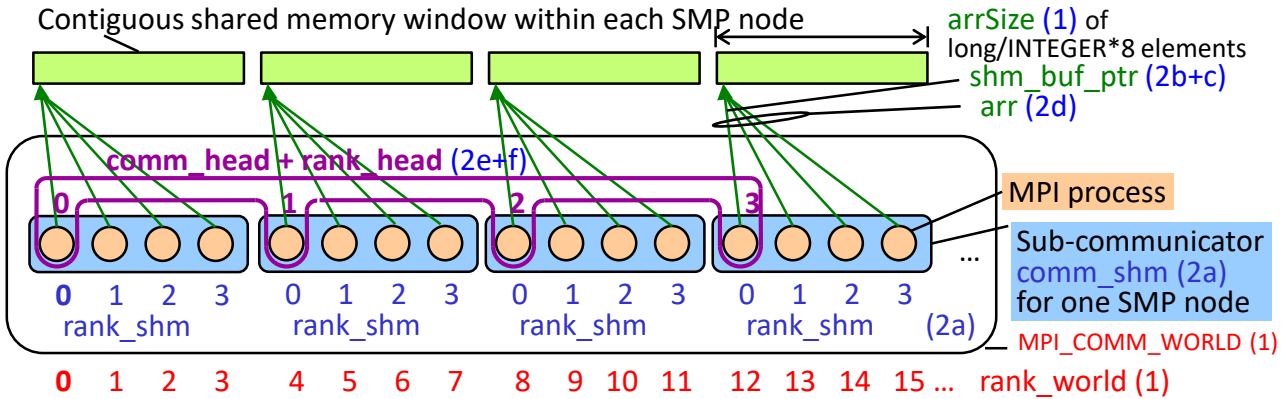
(2b) *if* (rank_shm == 0) *then* individualShmSize = arrSize *else* individualShmSize = 0

(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);

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

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



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

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

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

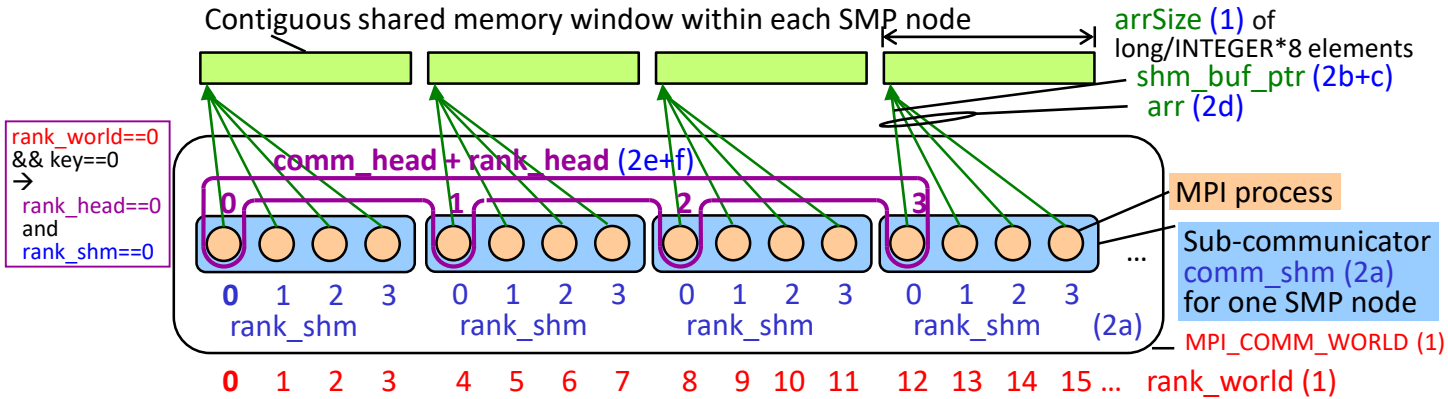
(2c) MPI_Win_allocate_shared (comm_shm → win & shm_base_ptr (but only if rank_shm== 0))

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

(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

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



(1) Given: arrSize, MPI_COMM_WORLD → rank_world

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

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

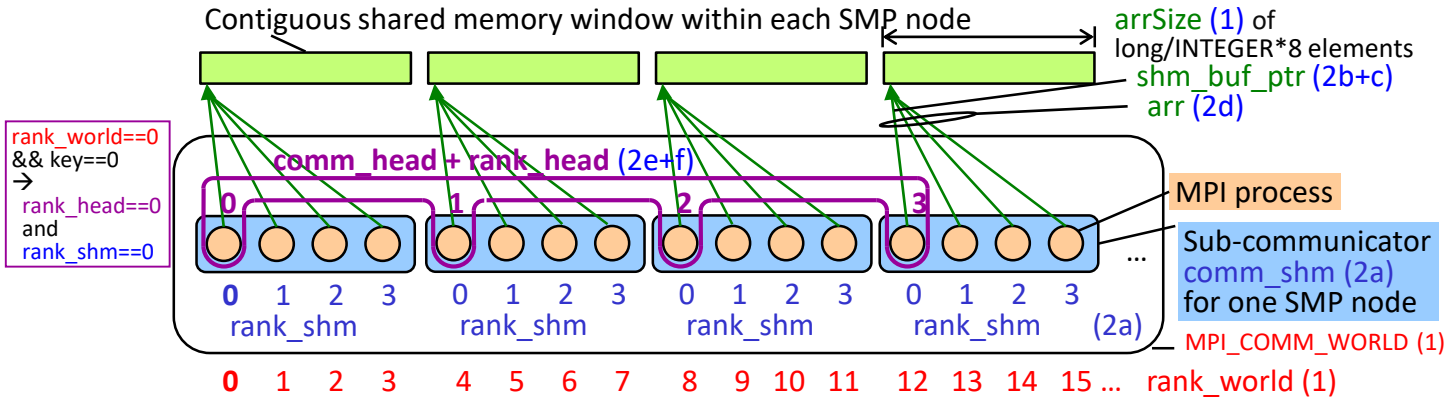
(2c) MPI_Win_allocate_shared (comm_shm → win & shm_base_ptr (but only if rank_shm== 0))

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

(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

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_split_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))

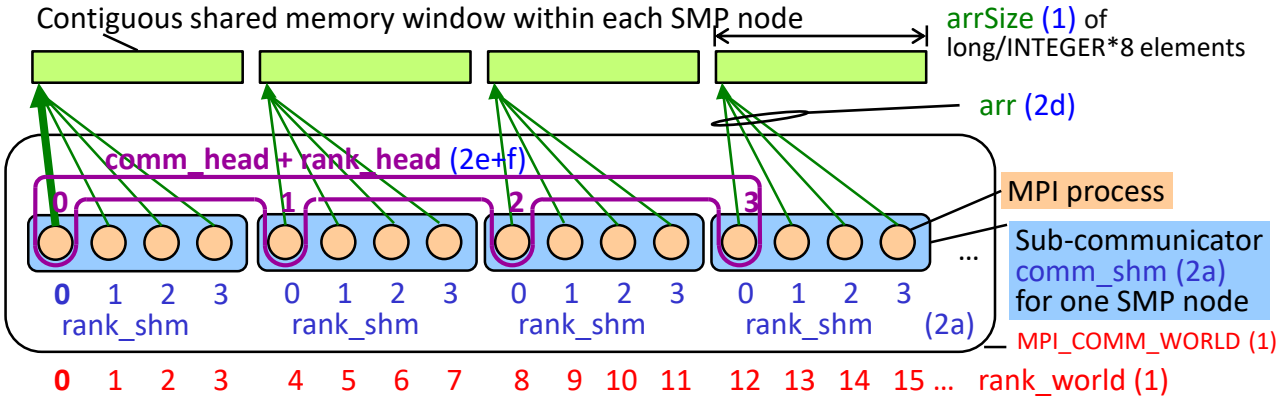
(2d) MPI_Win_shared_query (win & rank 0 → arr, i.e., the base pointer on all processes);

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

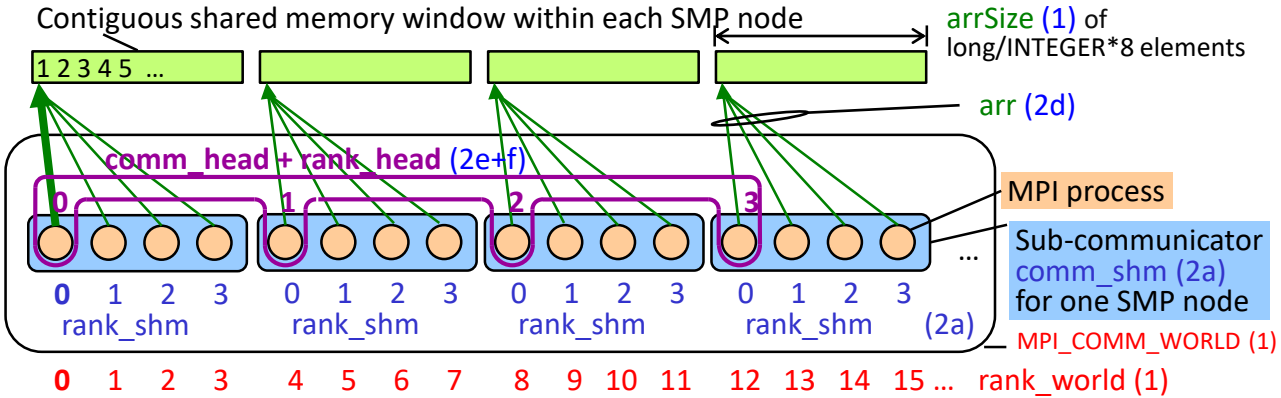
Exercise steps:
(3-6) The usage of the shared memory



Time step loop with index *it* and only 1 iteration

End of time step loop

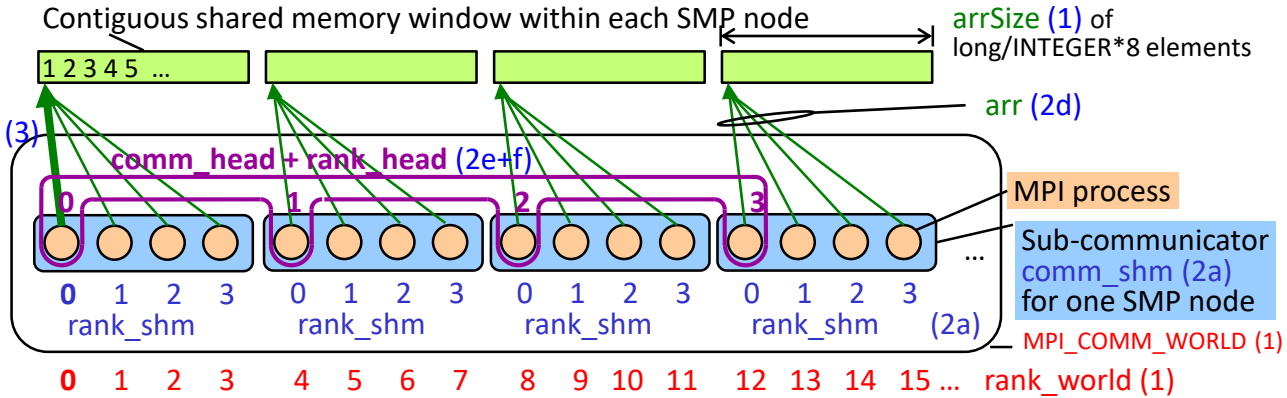
Exercise steps:
(3-6) The usage of the shared memory



Time step loop with index *it* and only 1 iteration

End of time step loop

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



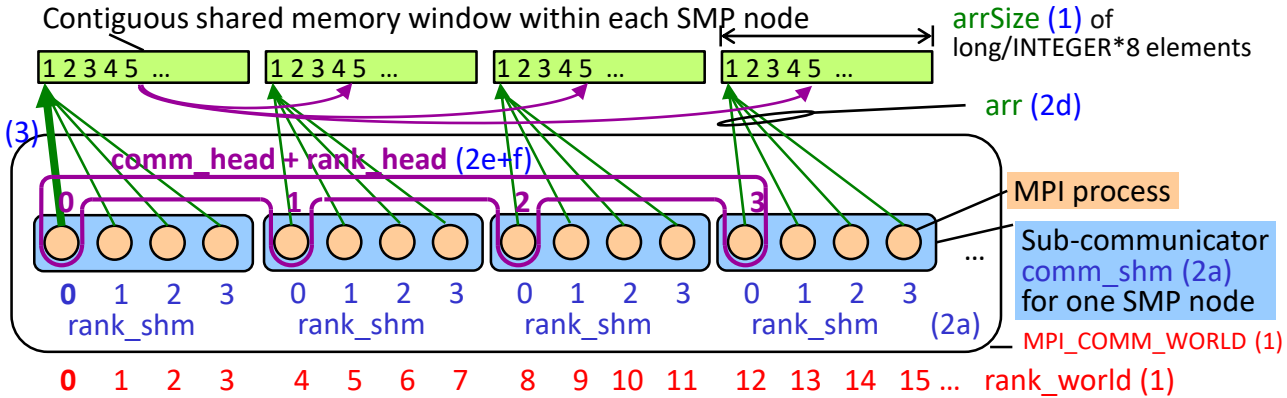
Time step loop with index *it* and only 1 iteration

(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 ist green arr

End of time step loop

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



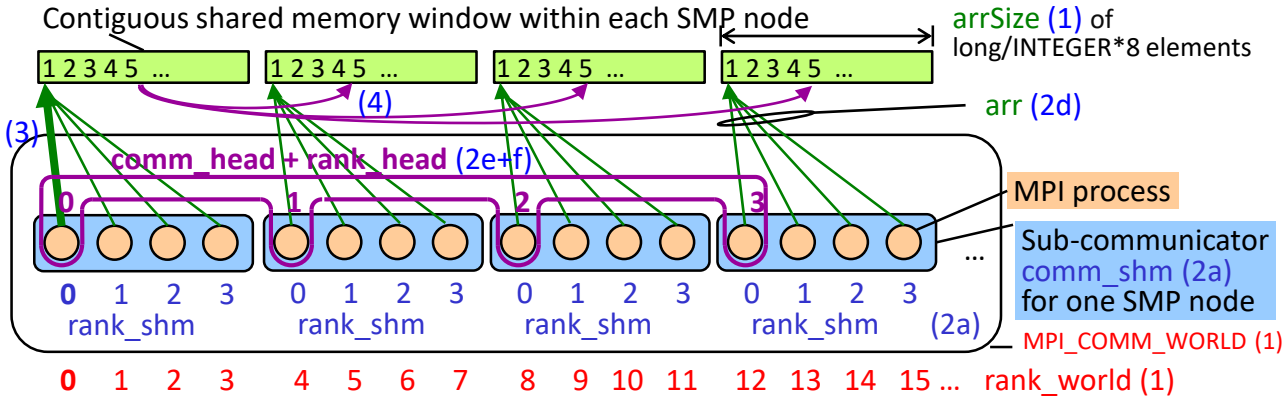
Time step loop with index *it* and only 1 iteration

(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 ist green arr

End of time step loop

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



Time step loop with index *it* and only 1 iteration

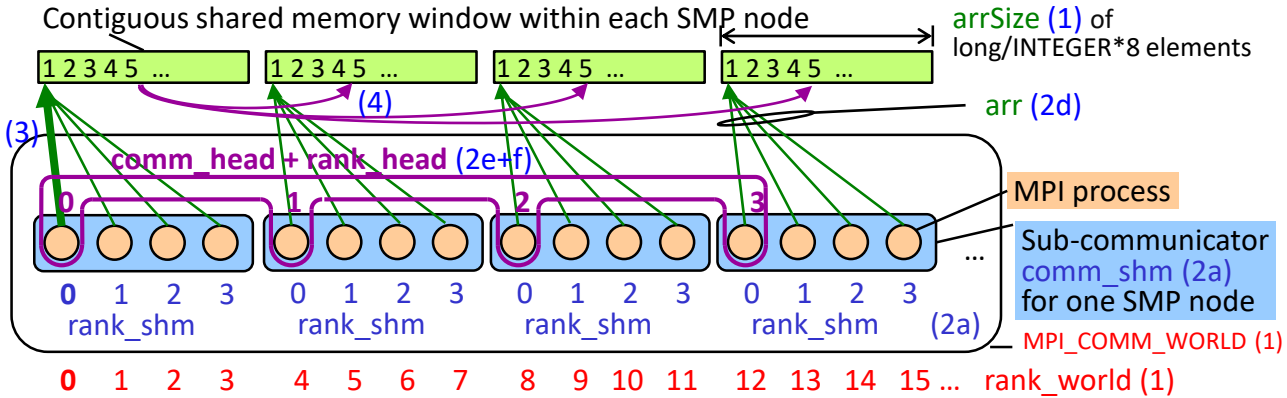
(3-4) Store epoch: we store the replicated data in all shared memories
(don't forget MPI_Win_fence() within all comm_shm/win before starting the store epoch for arr)

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

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

End of time step loop

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



Time step loop with index *it* and only 1 iteration

(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)

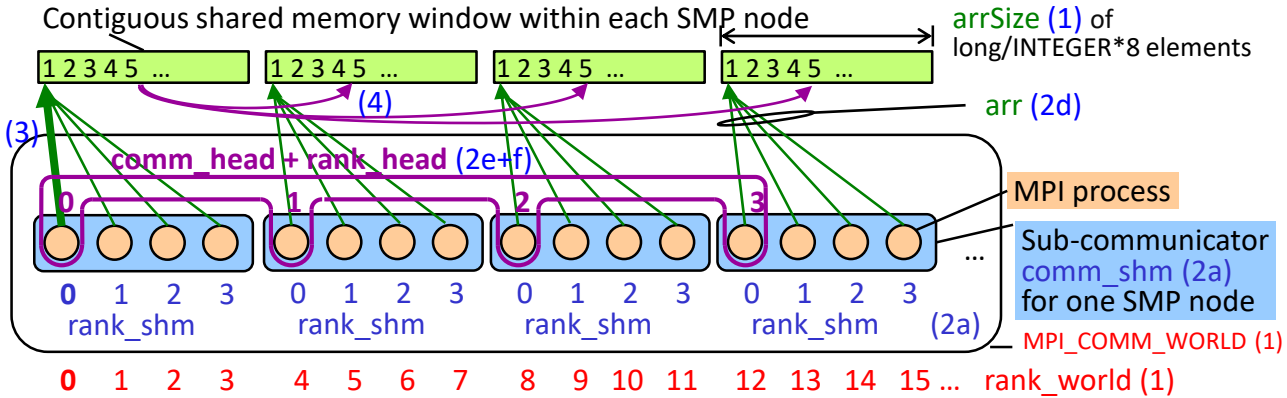
(3) Process with rank_world==0 stores numbers into ist green arr

(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)

End of time step loop

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



Time step loop with index *it* and only 1 iteration

(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 ist green arr

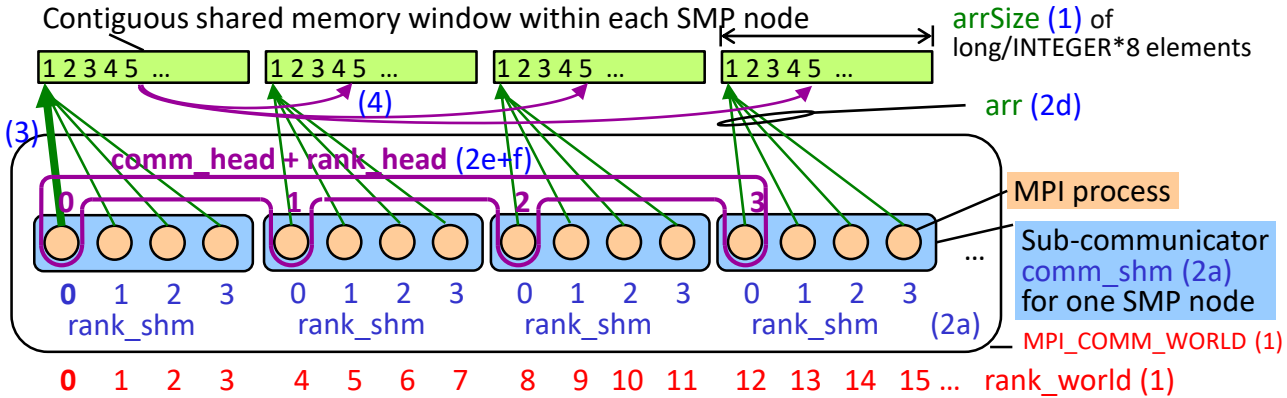
(4) All processes in comm_head MPI_Bcast() the data from rank_head==0 to all others

(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

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



Time step loop with index *it* and only 1 iteration

(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 ist green arr

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

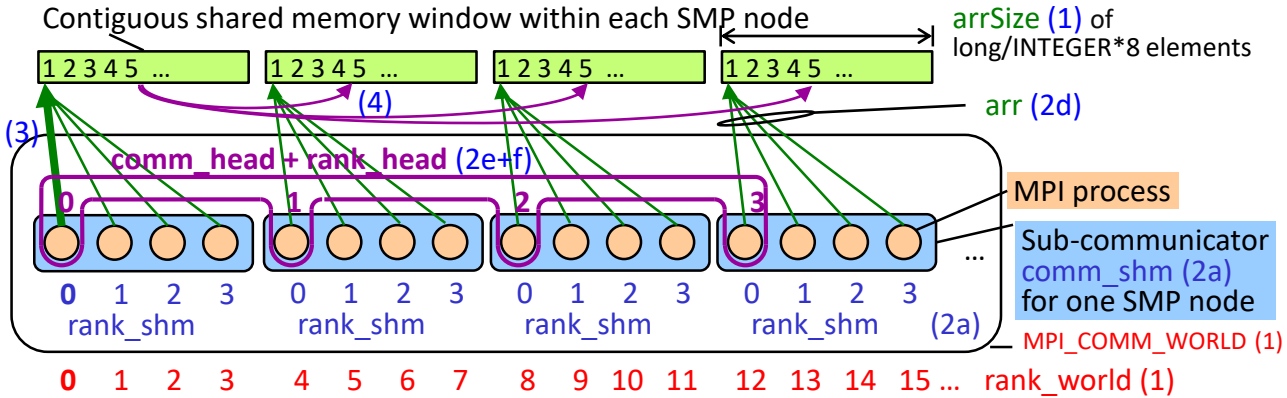
(5) Local load epoch: each process reads the data and locally calculates the sum
(don't forget MPI_Win_fence() within all comm_shm / win before starting the local load epoch)

(6) Print the results

End of time step loop

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

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



Time step loop with index *it* and only 1 iteration

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

- **(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 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()**

Exercise: MPI_Bcast into shared

Preparation

- 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)

mpi_f08 module is used → substitute, e.g.,

```
!_____________ :: comm_shm     by
TYPE(MPI_Comm) :: comm_shm
```

- 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

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 LRZ**_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)
  - Output will be written to: **slurm-*.out**
  - Output from only 2 nodes (each with 16 MPI processes):

```
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
```

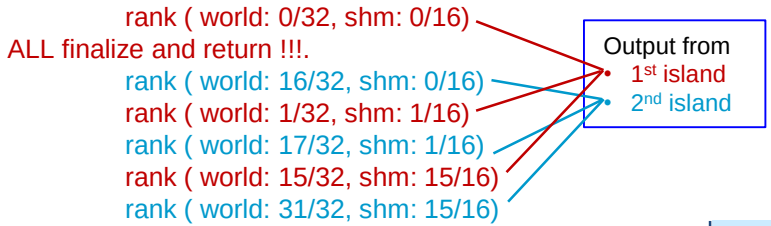
- 1st time step
- output from 3 processes per communicator:
- ranks 0, 1 & last rank

Exercise: MPI_Bcast into shared memory

- data-rep_exercise.c / _30.f90 is the skeleton for all steps of this exercise
  - Please edit and change it from step to step!
- 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 islands, each with 16 processes:



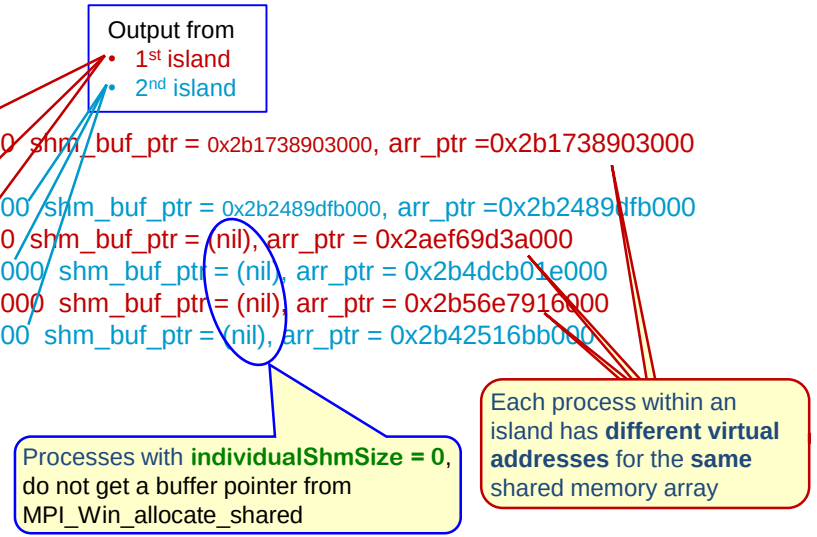
  - **After ~10 Minutes:**
    - compare with solution: data-rep_sol_2a.c / _30.f90
    - In case of problems you may also look at the solution slide:

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

Exercise: MPI_Bcast into shared memory

- **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 islands, each with 16 processes:

```
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: Sol.

Exercise: MPI_Bcast into shared memory

- 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 islands, 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
  - 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

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

Advanced Exe: Breaking the world into NUMA islands

- 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=99999999) = 499999995000
it: 0, rank ( world: 1/32, shm: 1/8, head: -1/-1 ):  sum(i=0...i=99999999) = 499999995000
it: 0, rank ( world: 7/32, shm: 7/8, head: -1/-1 ):  sum(i=0...i=99999999) = 499999995000
it: 0, rank ( world: 8/32, shm: 0/8, head: 1/4 ):    sum(i=0...i=99999999) = 4999999950000000
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(i=0...i=99999999) = 4999
it: 0, rank ( world: 16/32, shm: 0/8, head: 2/4 ):   sum(i=0...i=99999999) = 4999
it: 0, rank ( world: 25/32, shm: 1/8, head: -1/-1 ): sum(i=0...i=99999999) = 4999
it: 0, rank ( world: 31/32, shm: 7/8, head: -1/-1 ): sum(i=0...i=99999999) = 4999
it: 0, rank ( world: 17/32, shm: 1/8, head: -1/-1 ): sum(i=0...i=99999999) = 49
it: 0, rank ( world: 23/32, shm: 7/8, head: -1/-1 ): sum(i=0...i=99999999) = 499
...
```

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 (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 islands:

```
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
```

Same data in the shared memory arrays of both SMP nodes

- 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 (6 lines of code)
  - (7) Finish the local load epoch → **MPI_Win_fence()** // free the window → **MPI_Win_free()**
  - Expected output from 2 islands **(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
...
```

  - 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

General considerations & uses cases
Re-cap: MPI_Comm_split & one-sided communication
How-to
Exercise: MPI_Bcast
Quiz 1
> MPI memory models & synchronization
Shared memory problems
Advantages & disadvantages, conclusions
Quiz 2

How to achieve even lower latencies

A key feature for strong scaling?

Outlook

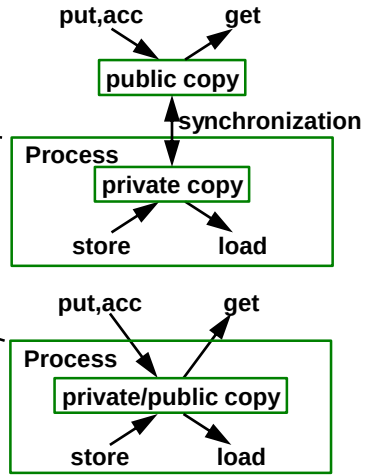
- Use of MPI shared memory without (slow) MPI one-sided synchronization methods (e.g., win_fence)
- To do this, use memory variables for synchronization together with memory fences (C++11 or MPI based)

Alternative:

- Fast MPI point-to-point sync together with memory fences

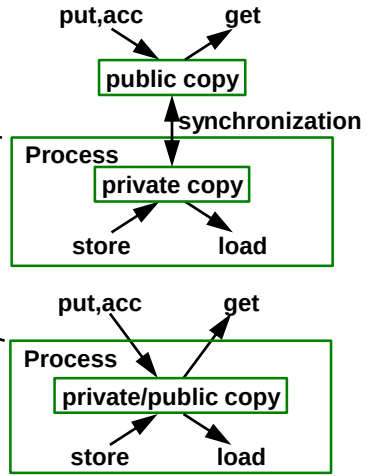
Two memory models

- 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



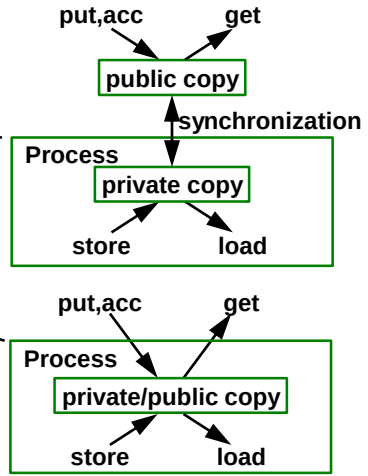
Two memory models

- 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)



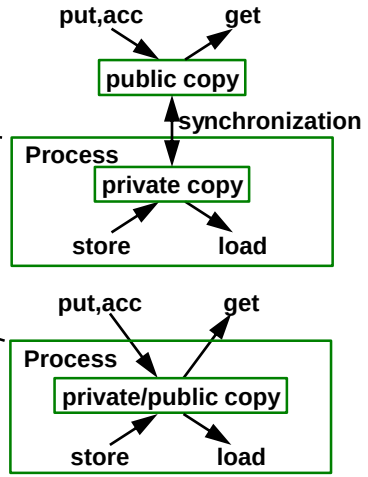
Two memory models

- 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)



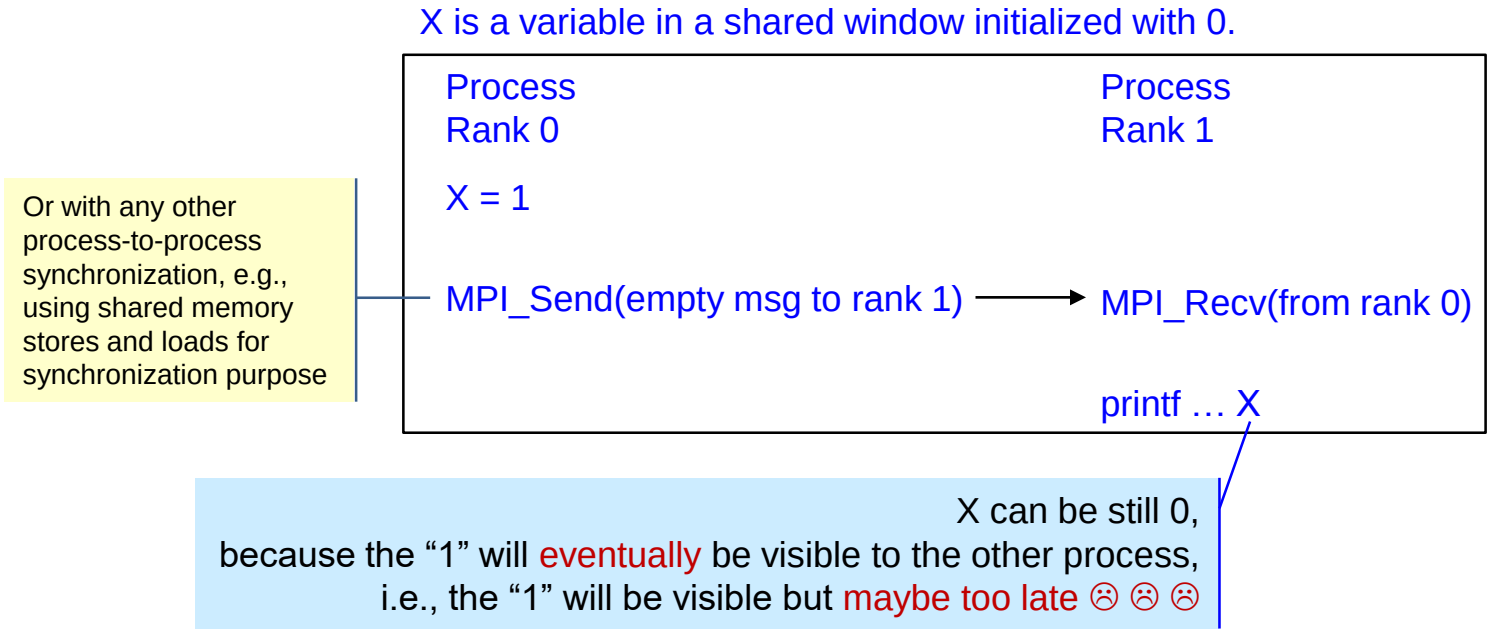
Two memory models

- 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)



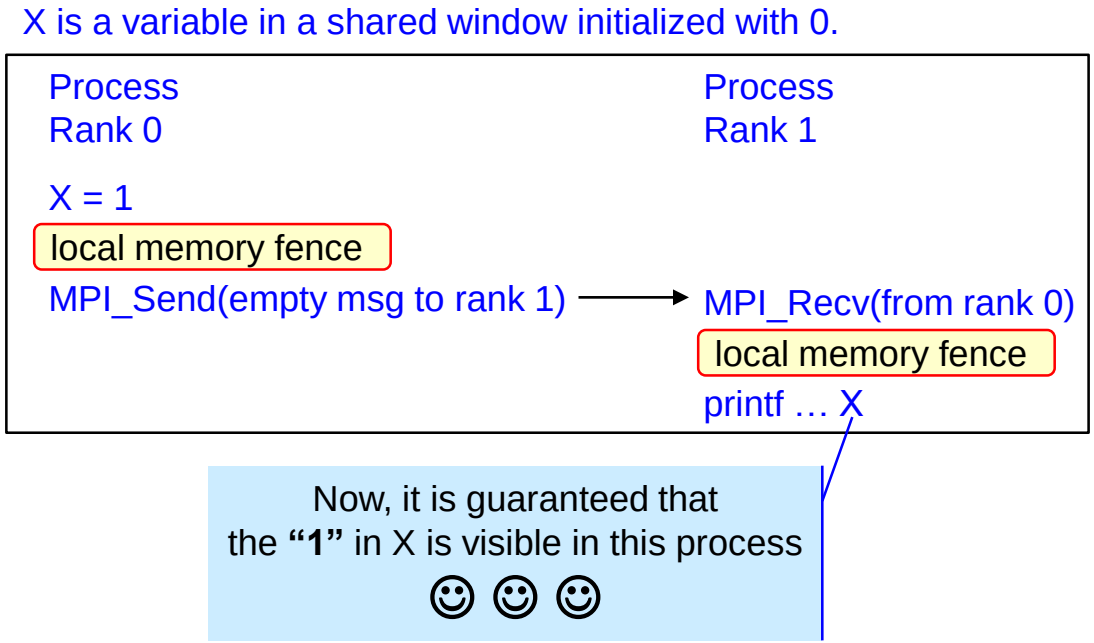
“eventually synchronized“ – the problem

- The problem with shared memory programming using libraries is:



“eventually synchronized“ – the Solution

- A pair of local memory fences is needed:

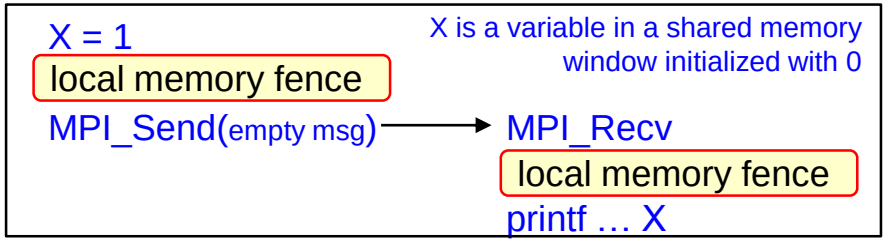


“eventually synchronized“ – Last Question

Several options & heavy discussions in the MPI Forum

How to make the local memory fence ?

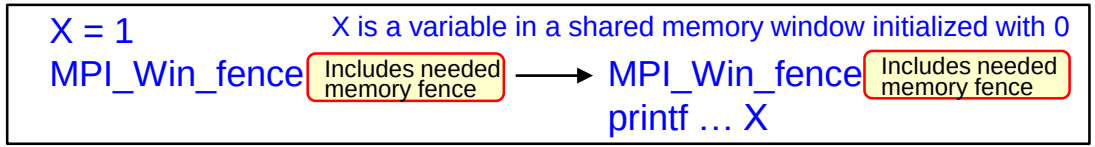
- C++11 atomic_thread_fence(order)
  - Advantage: one can choose appropriate order = memory_order_release, or …_acquire to achieve minimal latencies



- MPI_Win_sync
  - Advantage: works also for Fortran
  - Disadvantage: may be slower than C11 atomic_thread_fence with appropriate order
- Using RMA synchronization with integrated local memory fence instead of MPI_Send → MPI_Recv

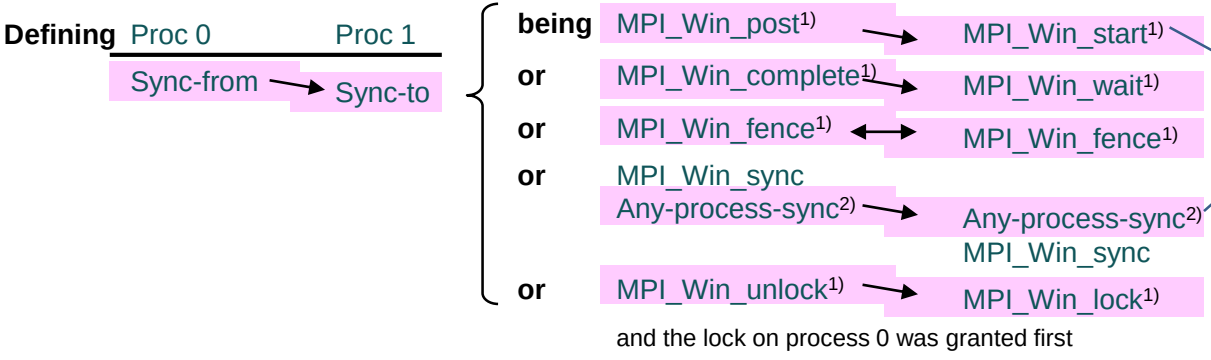
5 sync methods, see next slide

  - Advantage: May prevent double fences
  - Disadvantage: The synchronization itself may be slower



General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"A consistent view …"*)



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).

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"A consistent view …"*)

Defining Proc 0 → Proc 1

Sync-from → 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 MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

and A, B, C are shared variables
and having …

A=val_1

Sync-from → Sync-to

load(A)

[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).

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"A consistent view …"*)

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

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
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

[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).

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)

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"A consistent view …"*)

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

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
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

[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).

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	

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"A consistent view …"*)

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

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
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

[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).

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)

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"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
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

[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).

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)	

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"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
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

[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).

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)

General MPI shared memory synchronization rules

(based on MPI-3.1/4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: *"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
- MPI_Win_unlock[1] → MPI_Win_lock[1]

and the lock on process 0 was granted first

[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).

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

“Any-process-sync” & MPI_Win_sync on shared memory

“Any-process-sync” & MPI_Win_sync on shared memory

Process A	Process B
MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK,win)	MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK,win)
DO ...	DO ...
X=...	
MPI_F_SYNC_REG(X) 1)	
MPI_Win_sync(win)	
MPI_Send	
	MPI_Recv
	MPI_Win_sync(win)
	MPI_F_SYNC_REG(X) 1)
	local_tmp = X
	MPI_F_SYNC_REG(X) 1)
	MPI_Win_sync(win)
	MPI_Send
MPI_Recv	print local_tmp
MPI_Win_sync(win)	
MPI_F_SYNC_REG(X) 1)	1) Fortran only.
END DO	END DO
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)

“Any-process-sync” & MPI_Win_sync on shared memory

X is part of a shared memory window and should be **the same** memory location **in both processes**.

Process A	Process B
MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK,win)	MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK,win)
DO ...	DO ...
X=...	
MPI_F_SYNC_REG(X) [1]	
MPI_Win_sync(win)	
MPI_Send	
	MPI_Recv
	MPI_Win_sync(win)
	MPI_F_SYNC_REG(X) [1]
	local_tmp = X
	MPI_F_SYNC_REG(X) [1]
	MPI_Win_sync(win)
	MPI_Send
MPI_Recv	print local_tmp
MPI_Win_sync(win)	
MPI_F_SYNC_REG(X) [1]	
END DO	END DO
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)

[1] Fortran only.

“Any-process-sync” & MPI_Win_sync on shared memory

X is part of a shared memory window and should be **the same** memory location **in both processes**.

A new value is **written** in X

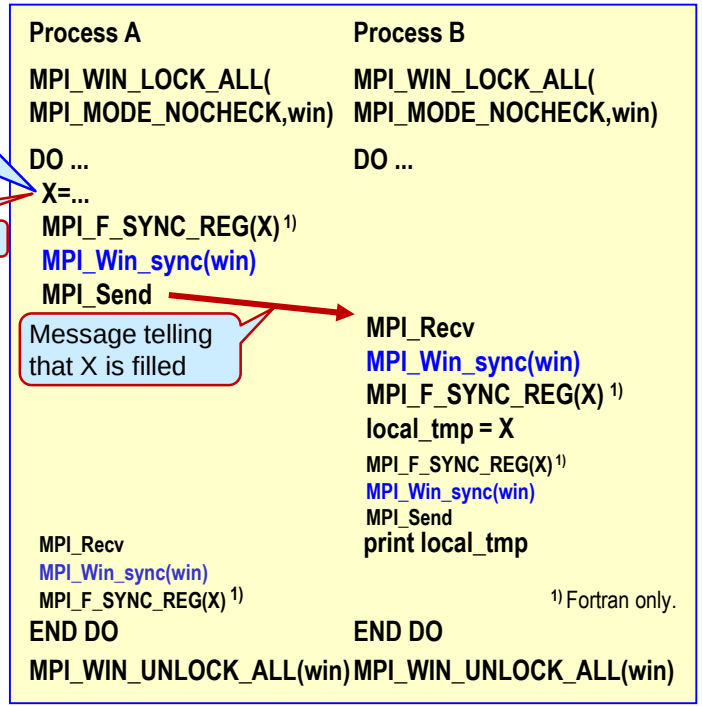
Process A	Process B
MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK,win)	MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK,win)
DO ...	DO ...
X=...	
MPI_F_SYNC_REG(X) [1]	
MPI_Win_sync(win)	
MPI_Send	
	MPI_Recv
	MPI_Win_sync(win)
	MPI_F_SYNC_REG(X) [1]
	local_tmp = X
	MPI_F_SYNC_REG(X) [1]
	MPI_Win_sync(win)
	MPI_Send
MPI_Recv	print local_tmp
MPI_Win_sync(win)	
MPI_F_SYNC_REG(X) [1]	
END DO	END DO
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)

[1] Fortran only.

“Any-process-sync” & MPI_Win_sync on shared memory

X is part of a shared memory window and should be **the same** memory location **in both processes**.

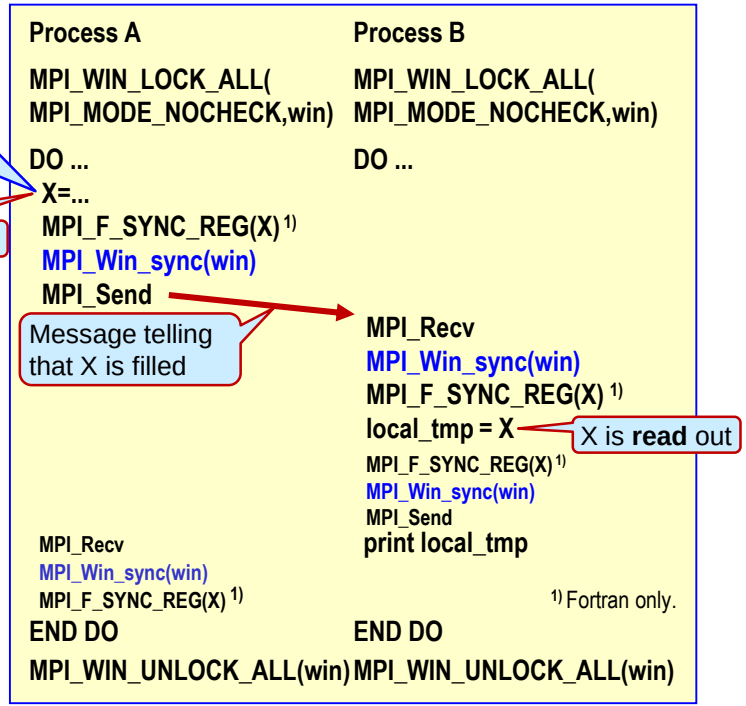
A new value is **written** in X



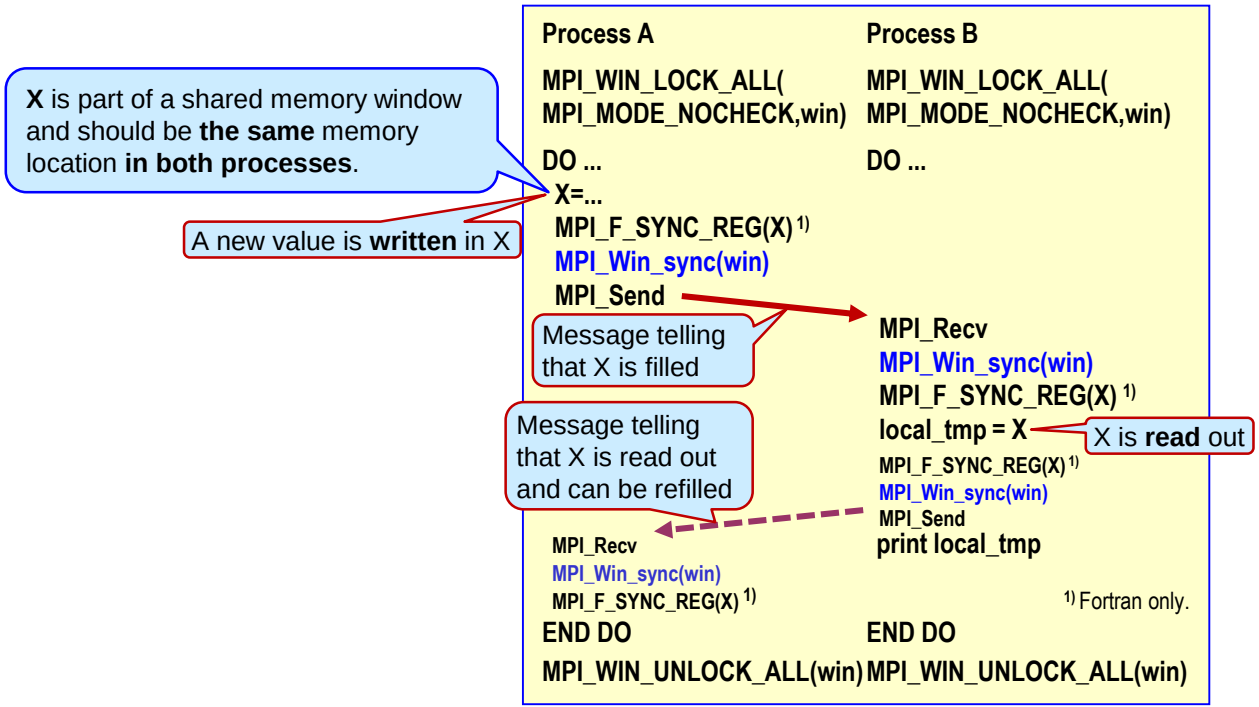
“Any-process-sync” & MPI_Win_sync on shared memory

X is part of a shared memory window and should be **the same** memory location **in both processes**.

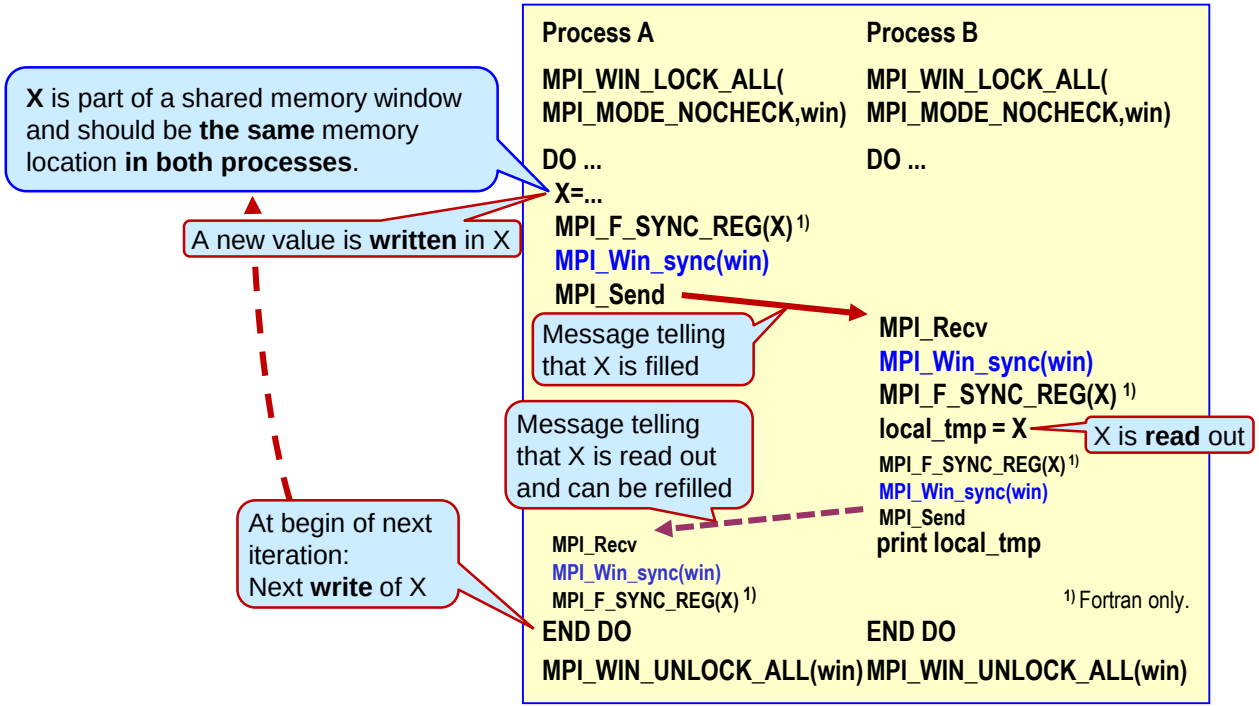
A new value is **written** in X



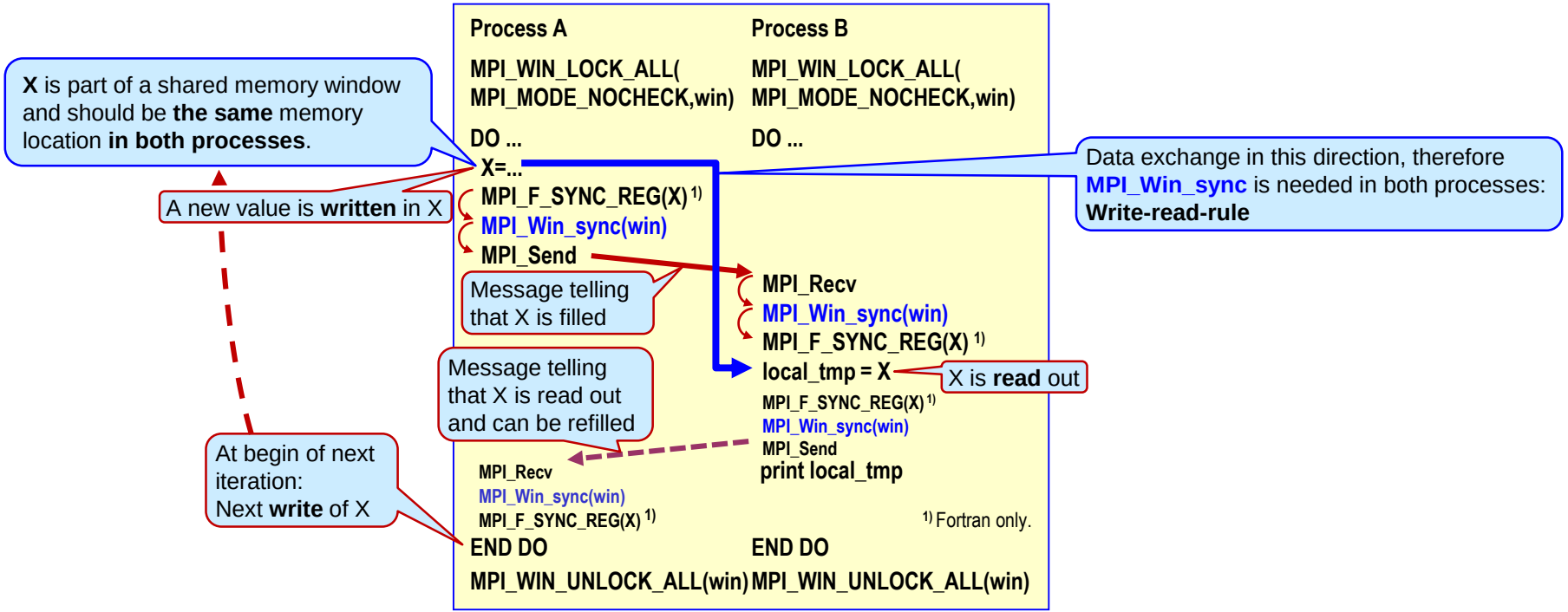
“Any-process-sync” & MPI_Win_sync on shared memory



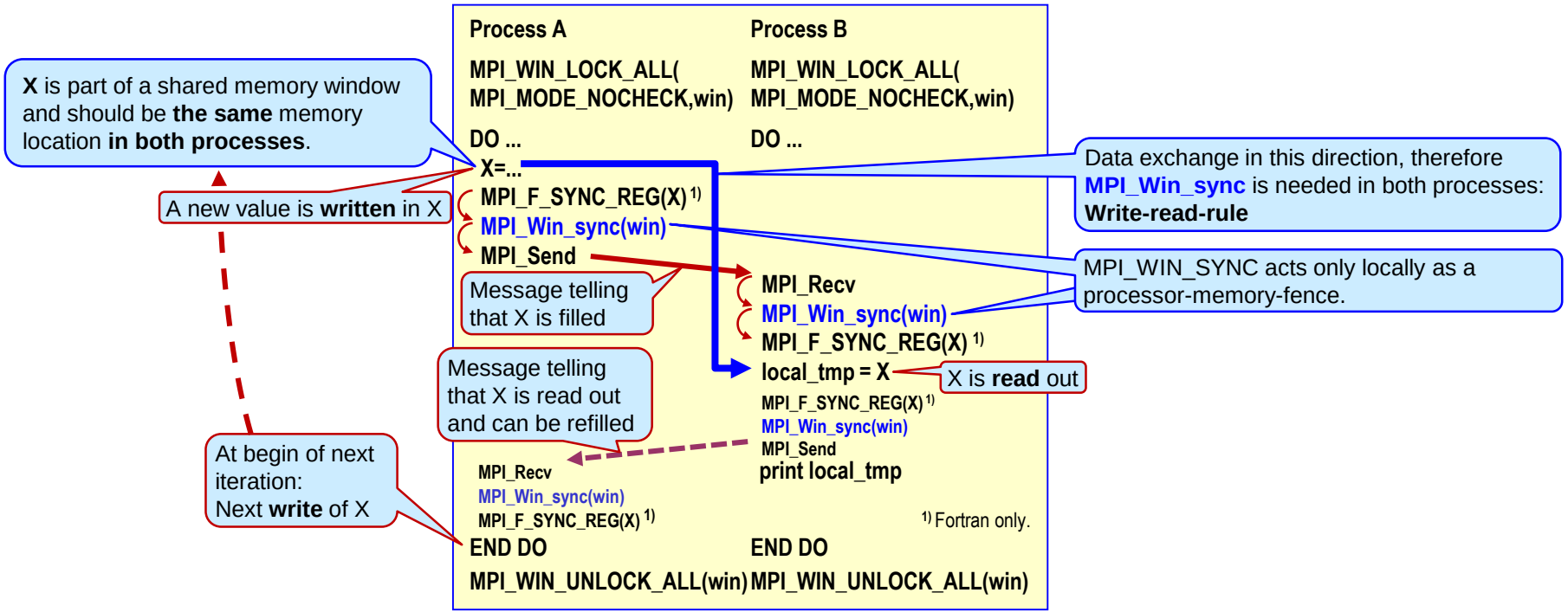
“Any-process-sync” & MPI_Win_sync on shared memory



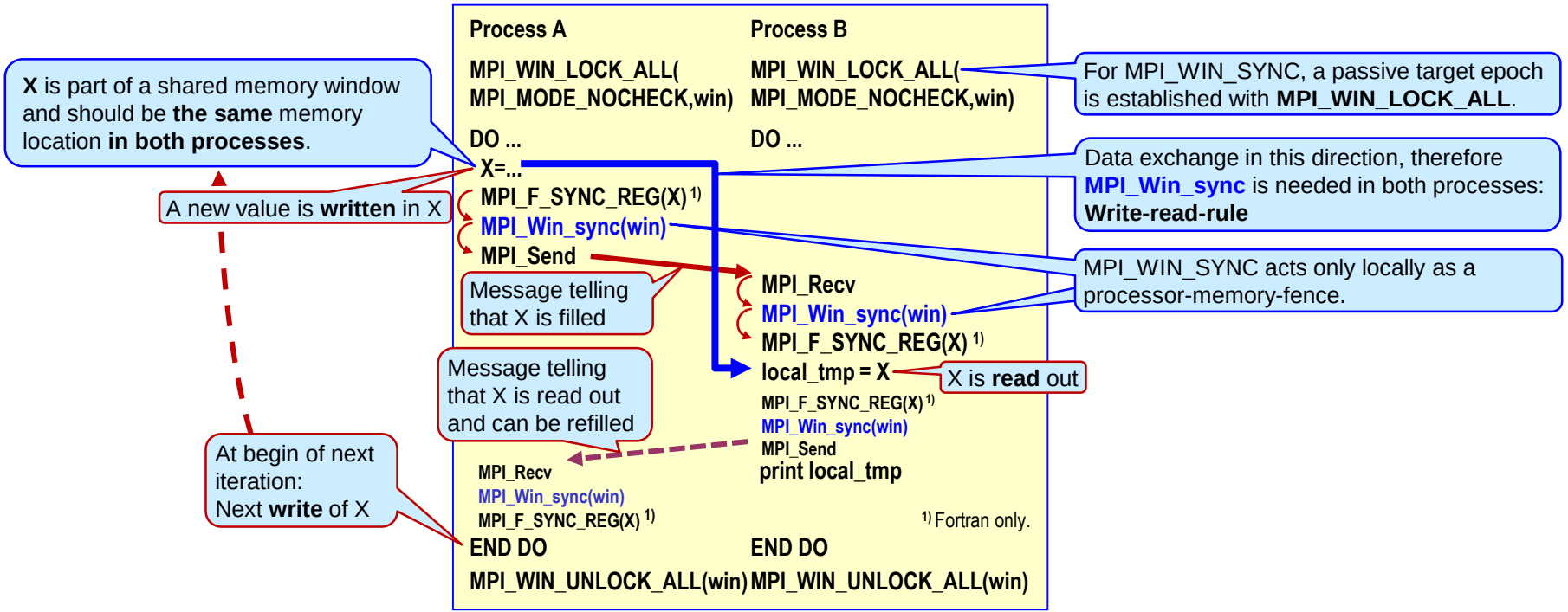
“Any-process-sync” & MPI_Win_sync on shared memory



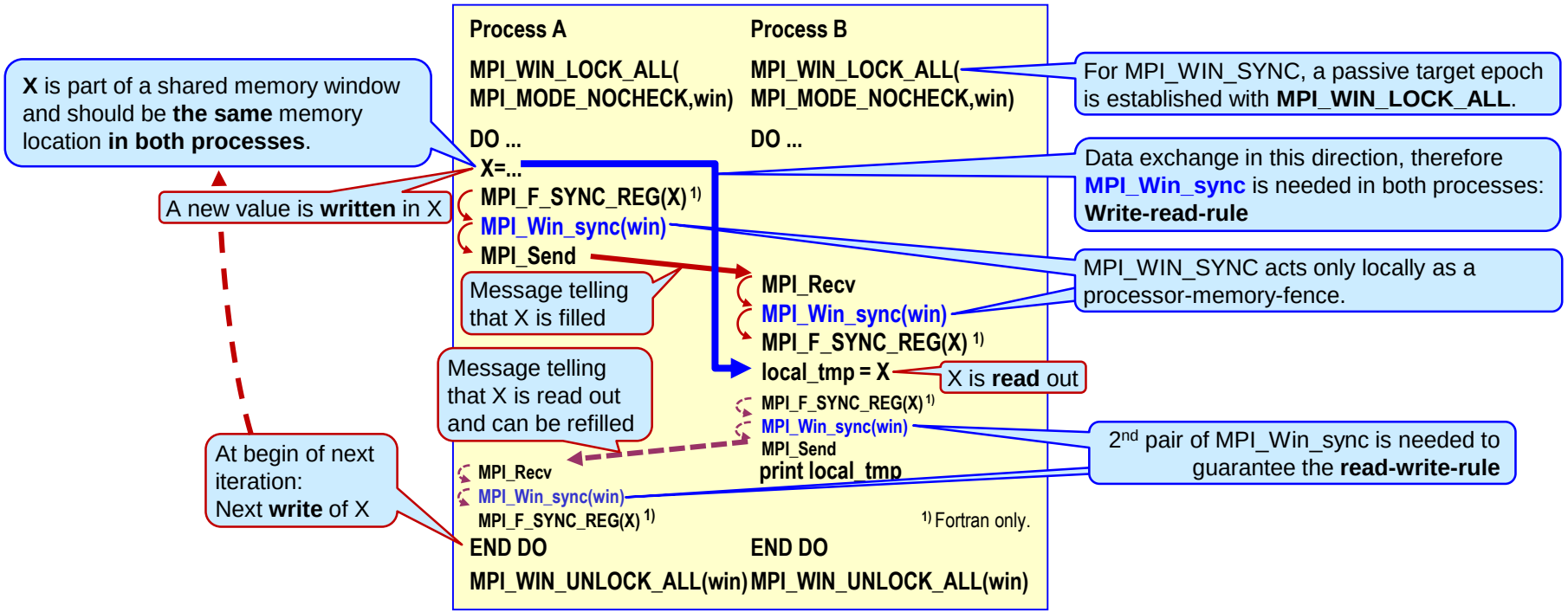
“Any-process-sync” & MPI_Win_sync on shared memory



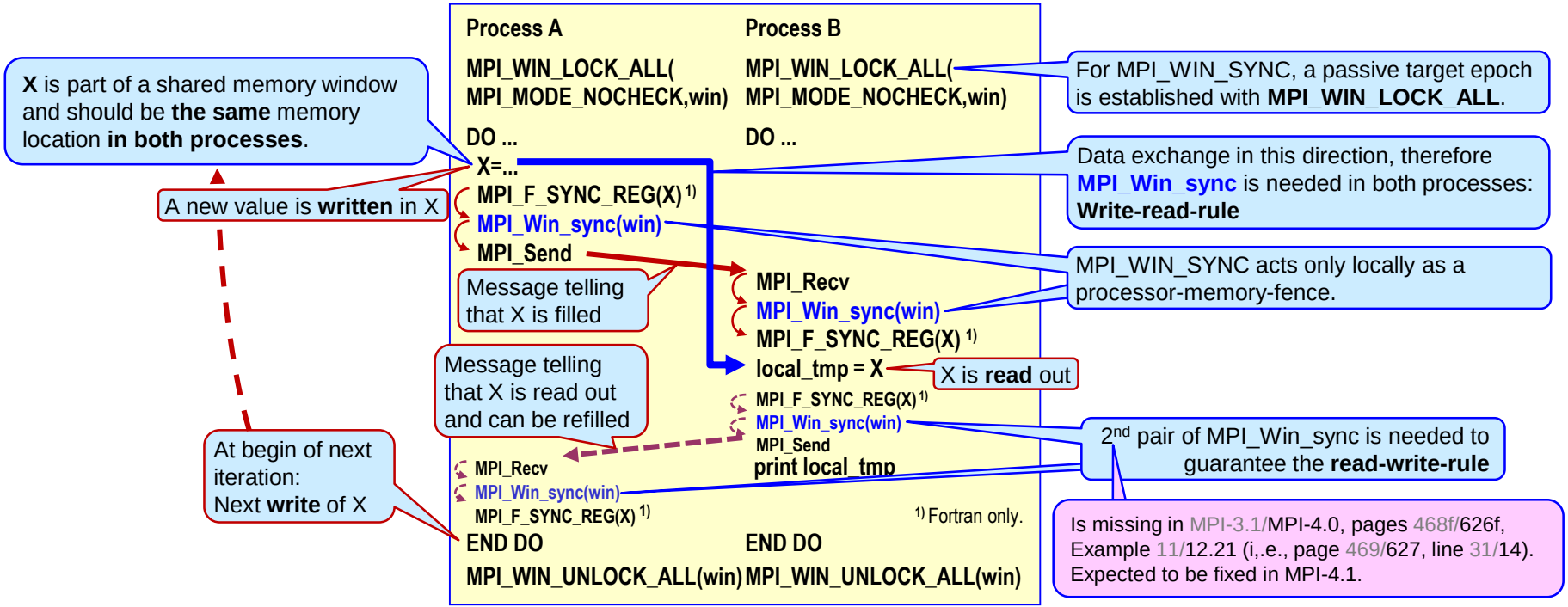
“Any-process-sync” & MPI_Win_sync on shared memory



“Any-process-sync” & MPI_Win_sync on shared memory



“Any-process-sync” & MPI_Win_sync on shared memory



Halo communication benchmarking

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

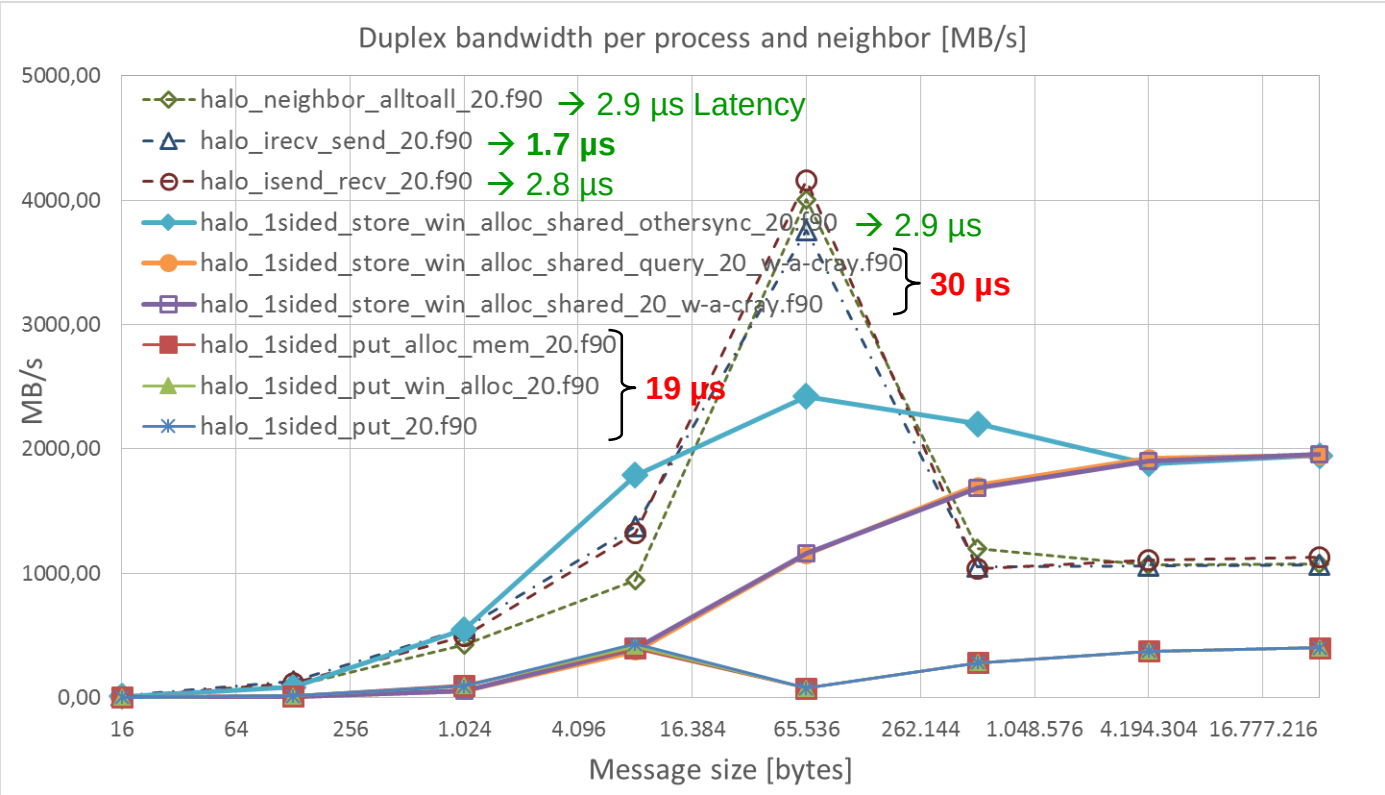
- Goal:
  - Learn about the communication latency and bandwidth on your system
- Method:
  - **cp MPI/course/C/1sided/halo* .**
  - On a shared or distributed memory, run and compare:
    - halo_irecv_send.c
    - halo_isend_recv.c
    - halo_neighbor_alltoall.c
    - halo_1sided_put.c
    - halo_1sided_put_alloc_mem.c
    - halo_1sided_put_win_alloc.c

    Example 1 (halo_irecv_send.c, halo_isend_recv.c)
    Example 2 (halo_1sided_put.c)
    Different communication methods
    Different memory allocation methods

    - Make a diff from one version to the next version of the source code
    - Compare latency and bandwidth
  - And run and compare on a shared memory only:
    - halo_1sided_store_win_alloc_shared.c (Example 3)
    - 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 (Example 4)
    - halo_1sided_store_win_alloc_shared_signal.c (Example 5)

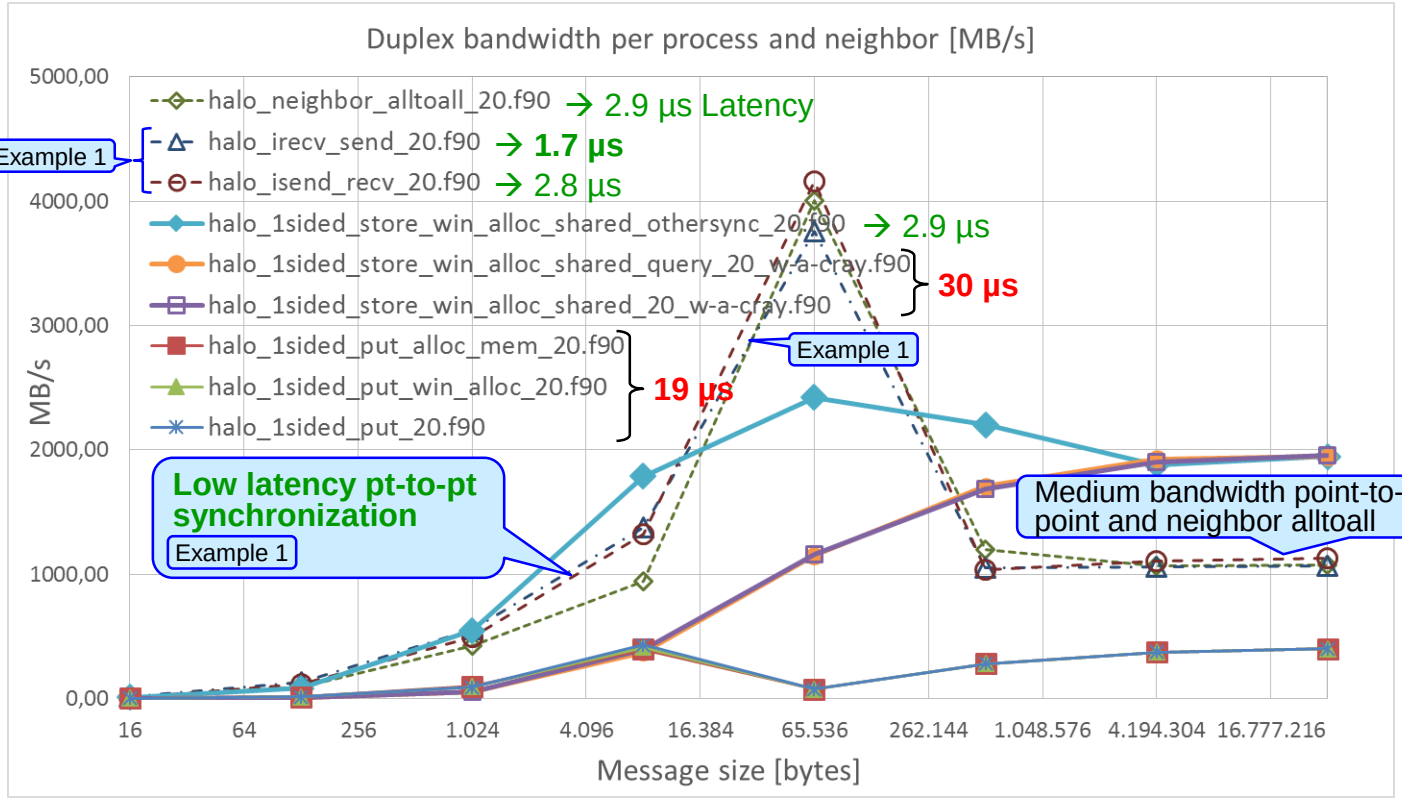
    Different communication methods

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



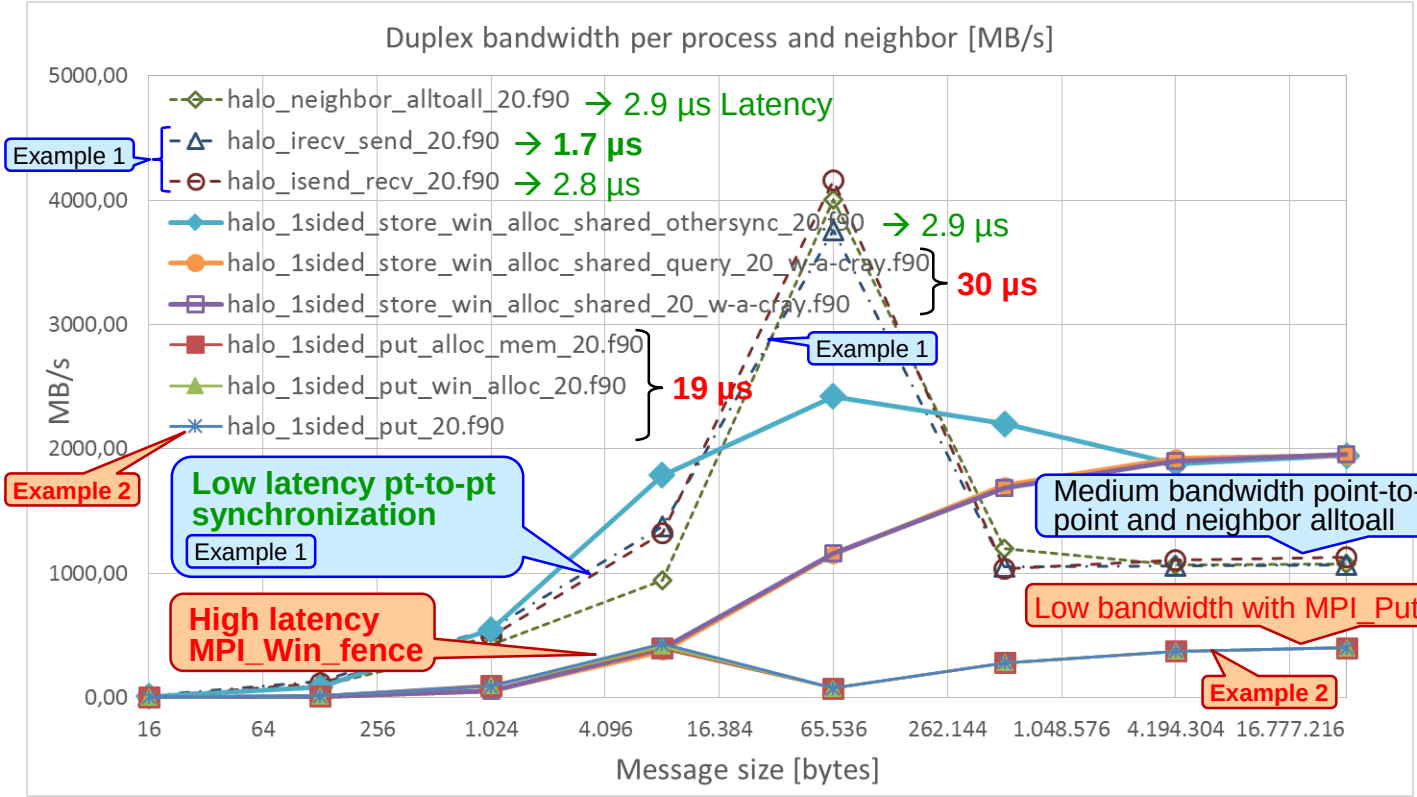
On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

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



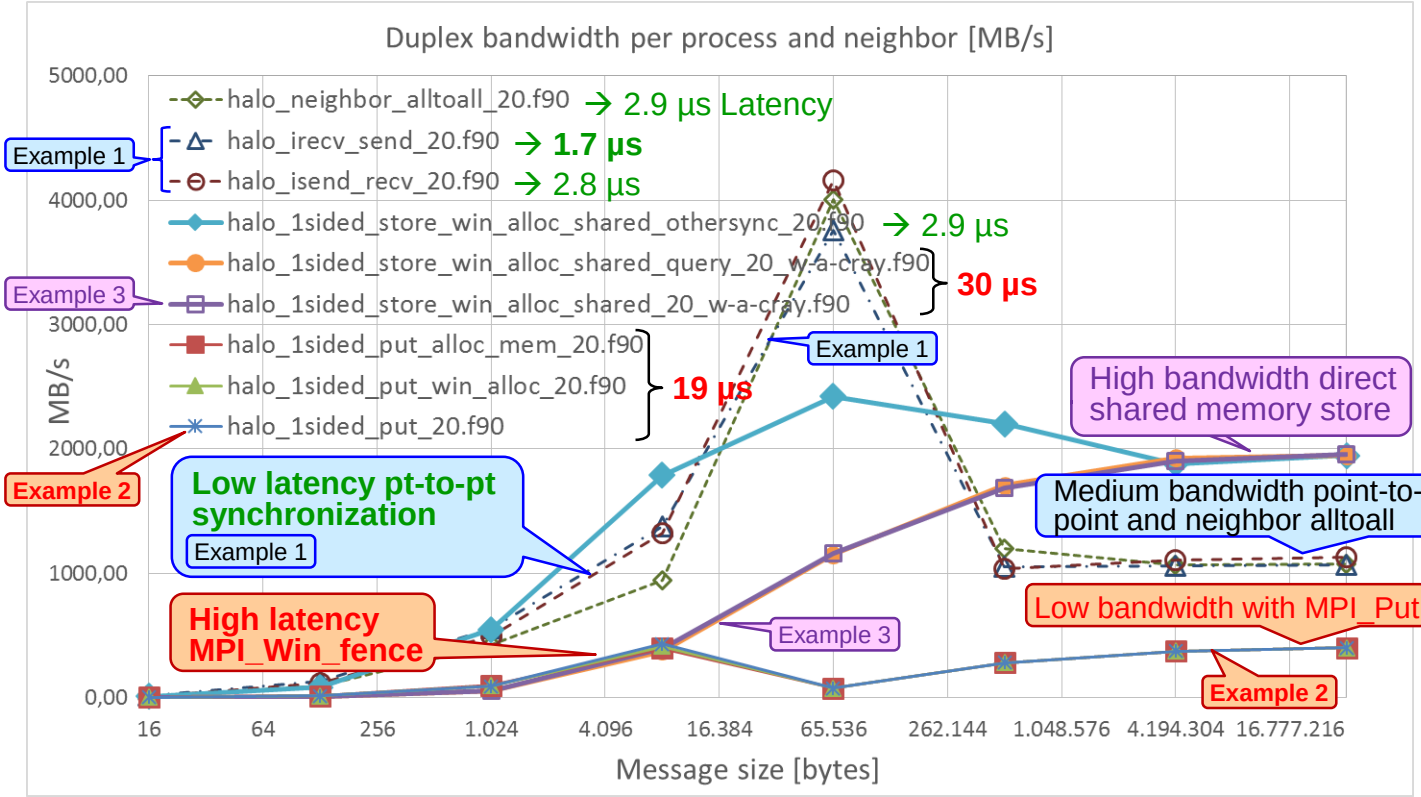
On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

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



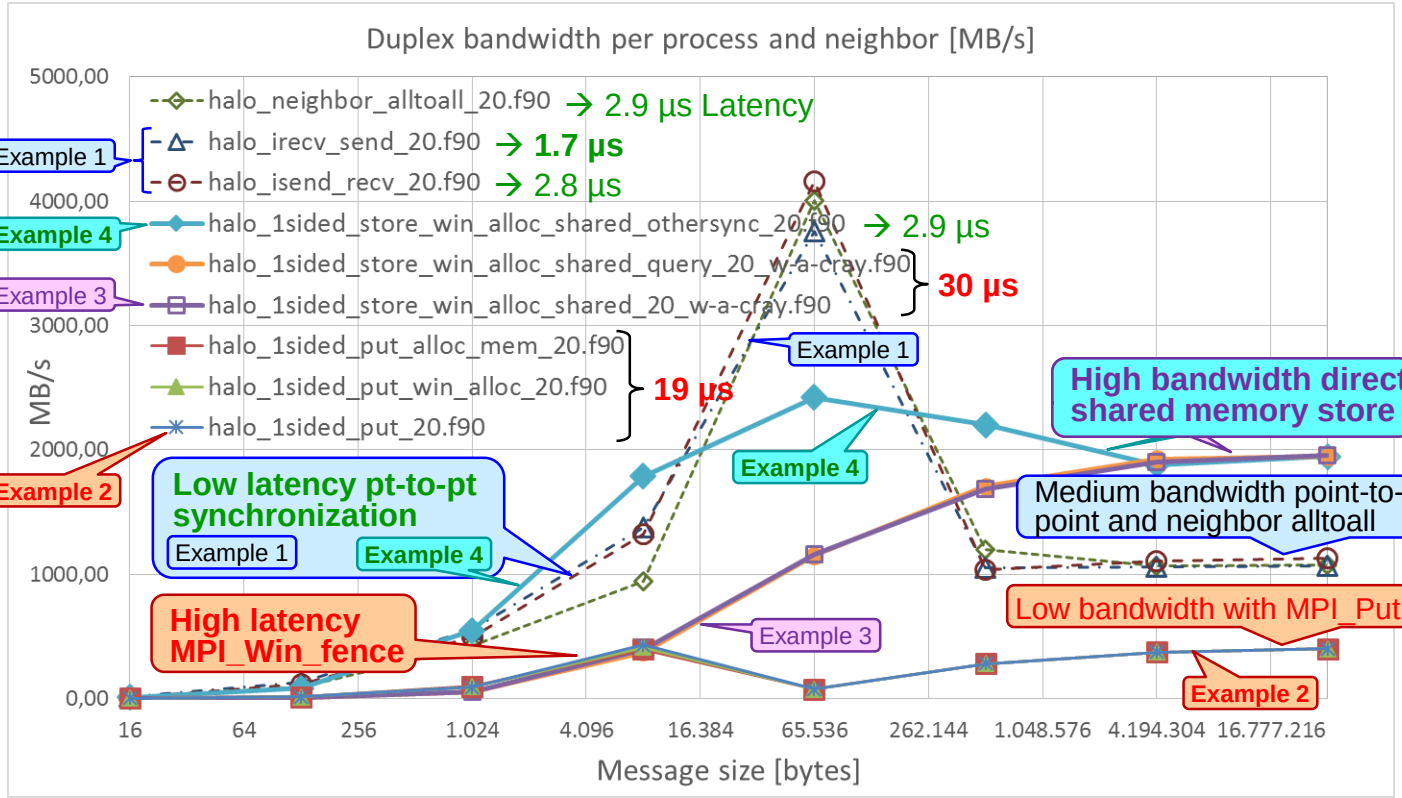
On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

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



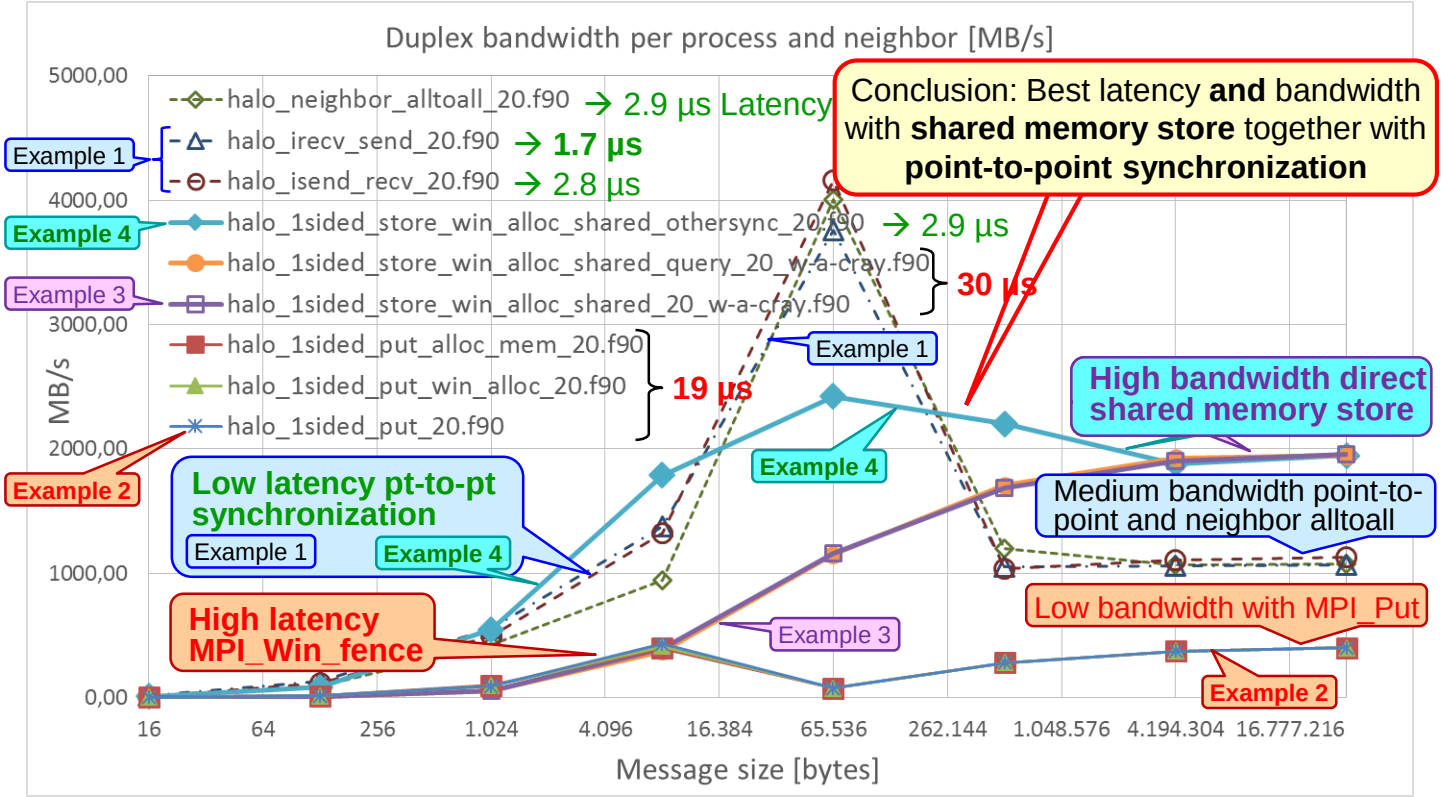
On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

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



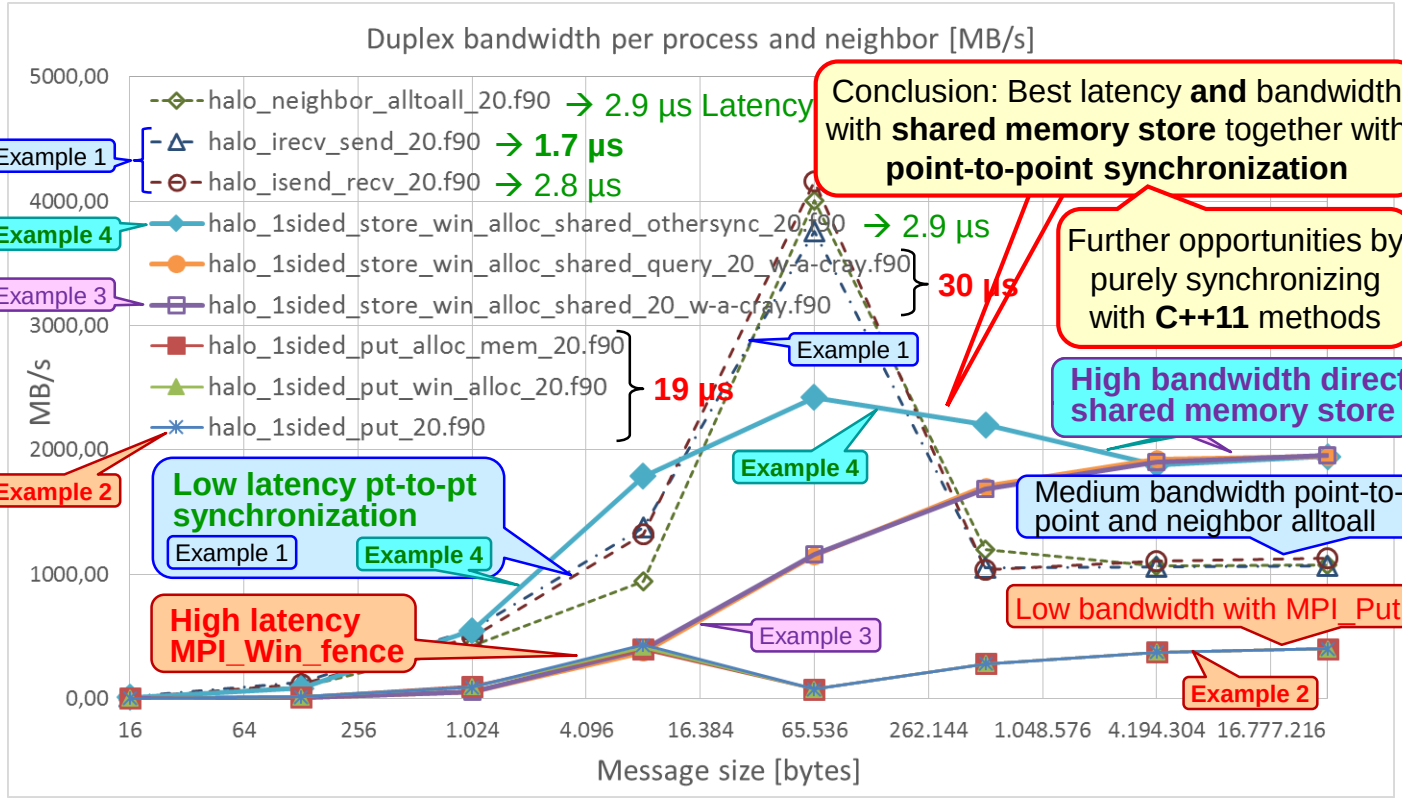
On Cray XE6 Hermit at HLRS with aprun –n 32 –d 1 –ss, best values out of 6 repetitions, modules PrgEnv-cray/4.1.40 and cray-mpich2/6.2.1

MPI communication inside of 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

MPI communication inside of 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

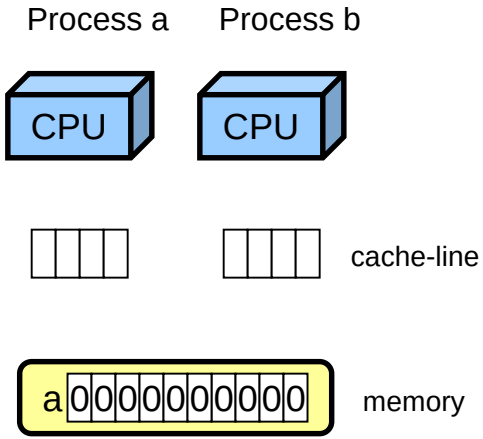
General considerations & uses cases
Re-cap: MPI_Comm_split & one-sided communication
How-to
Exercise: MPI_Bcast
Quiz 1
MPI memory models & synchronization
> **Shared memory problems**
Advantages & disadvantages, conclusions
Quiz 2

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)

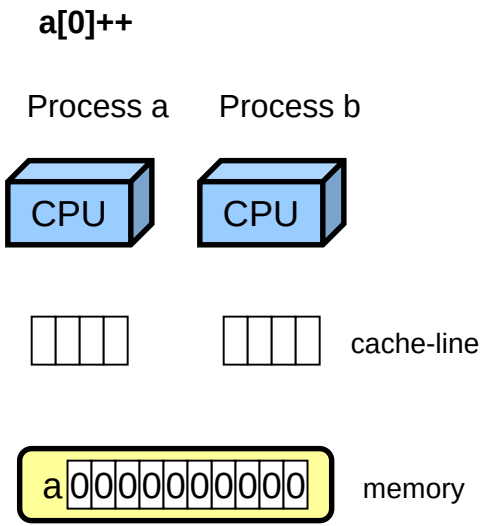
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

Shared memory problems (2/2)

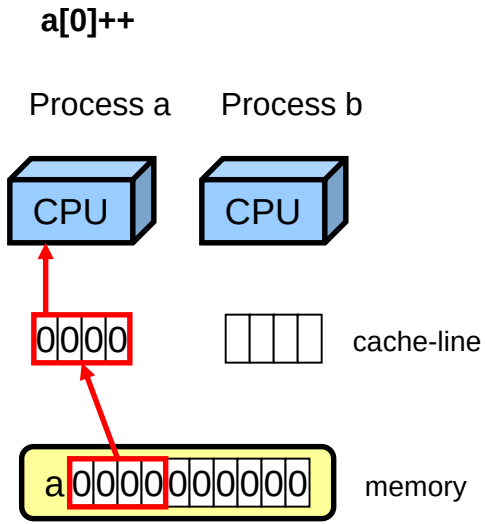
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

Shared memory problems (2/2)

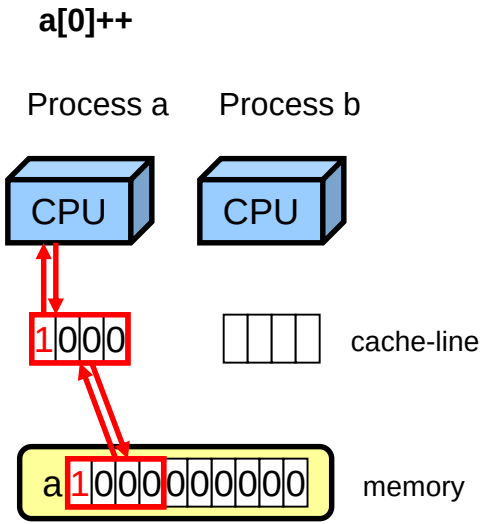
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

Shared memory problems (2/2)

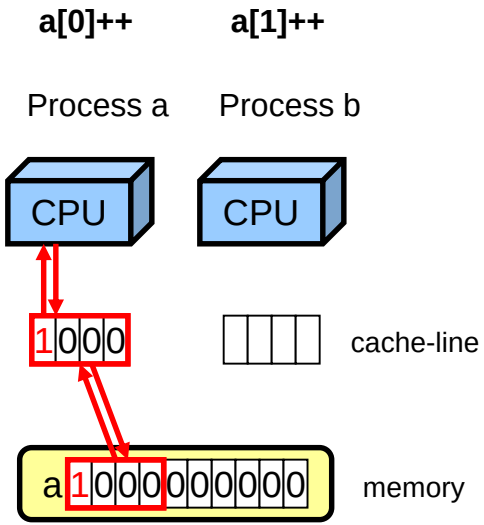
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

Shared memory problems (2/2)

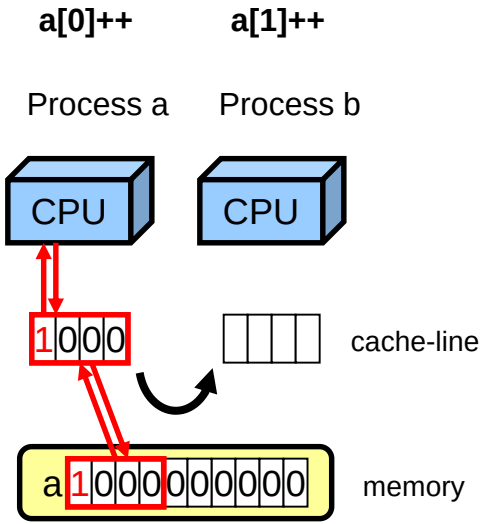
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

Shared memory problems (2/2)

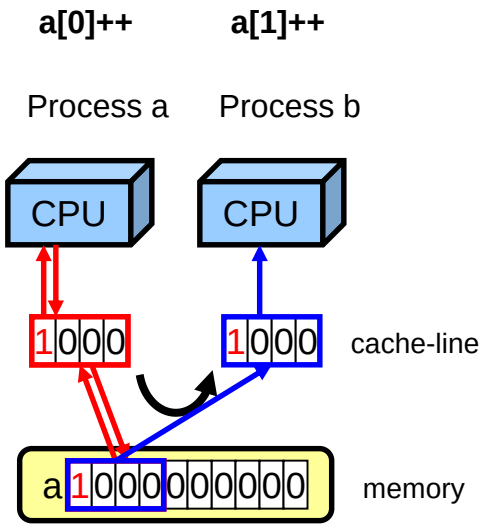
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

Shared memory problems (2/2)

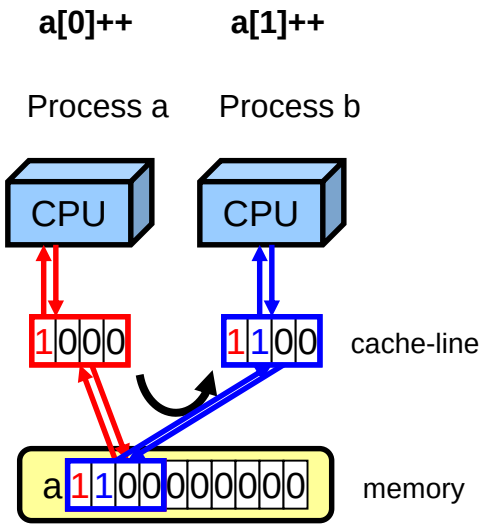
- Cache-line false-sharing
  - As with OpenMP or any other shared memory programming models
  - The cache-line is the smallest entity usually accessible in memory



- 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.

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.

Programming models
- MPI + MPI-3.0 shared memory

Advantages & disadvantages, conclusions

General considerations & uses cases
Re-cap: MPI_Comm_split & one-sided communication
How-to
Exercise: MPI_Bcast
Quiz 1
MPI memory models & synchronization
Shared memory problems
> **Advantages & disadvantages, conclusions**
Quiz 2

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?

Fastest accesses between MPI processes on a shared memory

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?

Fastest accesses between MPI processes on a shared memory

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?

Fastest accesses between MPI processes on a shared memory

- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?

Fastest accesses between MPI processes on a shared memory

- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
  - Can it reduce replicated data?

MPI-3 shared memory as a real alternative to OpenMP shared memory, especially when OpenMP hard to be used

- How can I leverage multiple accelerators?
  - What are typical challenges?

MPI+MPI-3.0 shared mem: Main advantages

- A new method for reducing memory consumption for replicated data
  - To allow only one replication per shared-memory island
- Interesting method for direct access to neighbor data (without halos!)
- A new method for communicating between MPI processes within each shared-memory node
- On some platforms significantly better bandwidth than with send/recv
- Library calls need not be “thread safe” because we do not have threads

MPI+MPI-3.0 shared mem: Main challenges

- Synchronization is defined, but still under discussion:
  - The meaning of the assertions for shared memory is still undefined as of MPI 4.0
- Similar problems as with all shared memory (e.g., pthreads, OpenMP,…)
  - Race conditions, false sharing, memory fences
- 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 shared-memory nodes
- Opportunity for reducing memory consumption (halos & replicated data)

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."

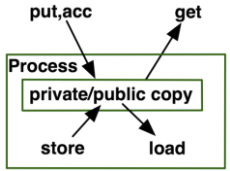


Figure: Courtesy of Torsten Hoefler

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)?

- _____
- _____
- _____

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?

- _____
- _____

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

- _____
- _____

Programming models - pure MPI

General considerations	slide 176
The topology problem	177
The topology problem: How-to / Virtual Toplogies	182
Rank renumbering for optimization	202
The Topology Problem: Unstructured Grids	220
Quiz	225
Scalability	226
Advantages & disadvantages, conclusions	228

Pure MPI communication

Advantages

- MPI library need not to support multiple threads (may have performance advantages)

Major problems

- Does application topology fit on hardware topology?
  - Want minimal communication between MPI processes AND between cluster nodes
- Does the MPI library employ shared memory protocols internally?
- Is the network prepared for massive numbers of messages?
- MPI communication inside of shared memory nodes also costs time
- Generally “a lot of” communicating processes per node
- Memory consumption may be a problem (halos, replicated data, internal MPI buffers)

Programming models - pure MPI

The Topology Problem

General considerations
> **The topology problem**
The topology problem: How-to / Virtual Toplogies
Rank renumbering for optimization
The Topology Problem: Unstructured Grids
Quiz
Scalability
Advantages & disadvantages, conclusions

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

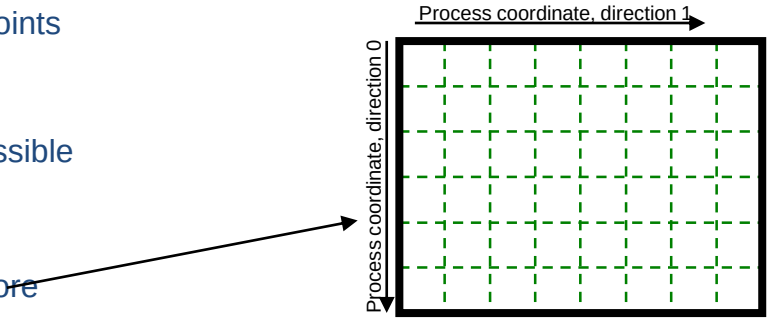
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores

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

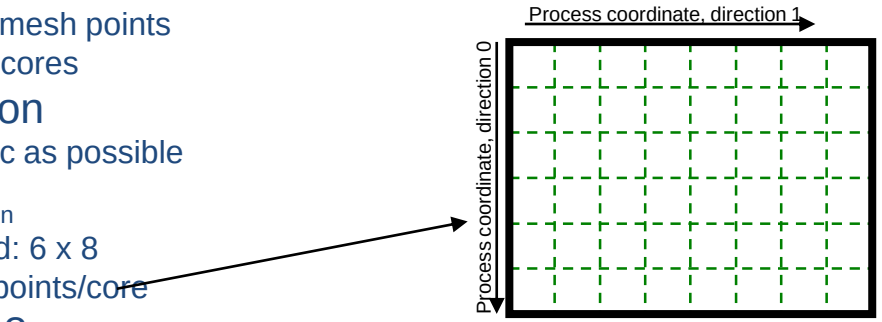
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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



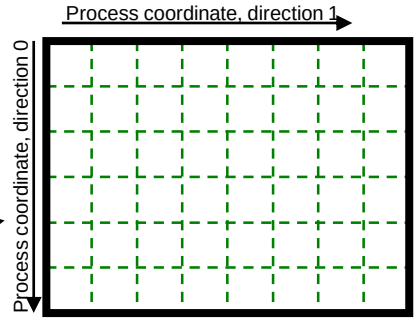
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

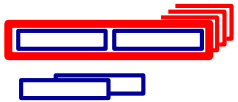
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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



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

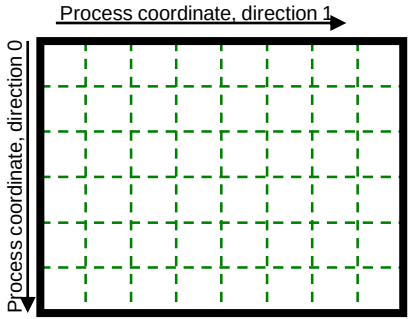
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

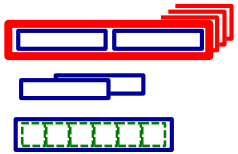




Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

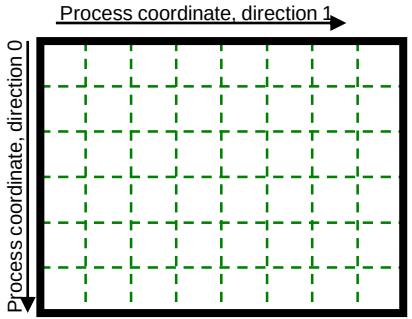
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware example: 48 cores:
  - 4 compute nodes
  - each node with 2 CPUs
  - each CPU with 6 cores



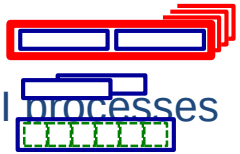


Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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



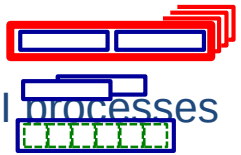
- Hardware example: 48 cores:
  - 4 compute nodes
  - each node with 2 CPUs
  - each CPU with 6 cores

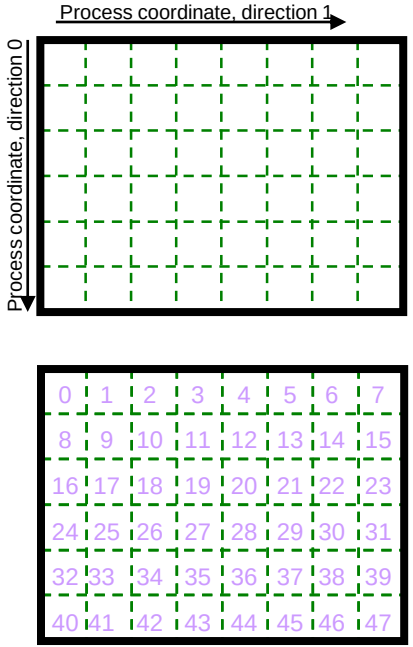


- How to locate the MPI processes on the hardware?
  - Using sequential ranks in MPI_COMM_WORLD

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

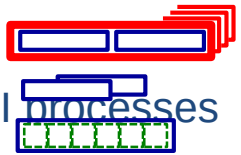
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

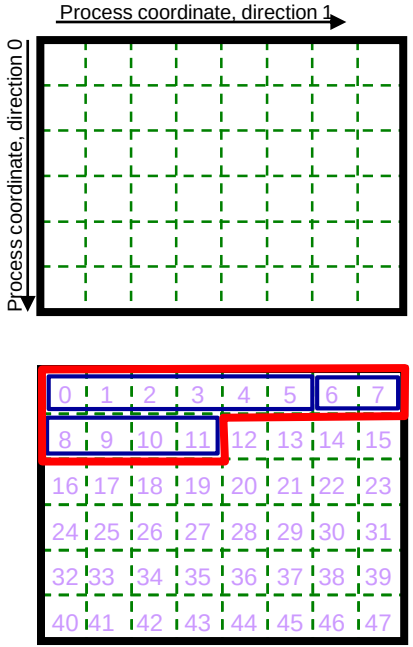




Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

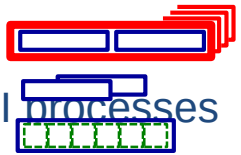
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

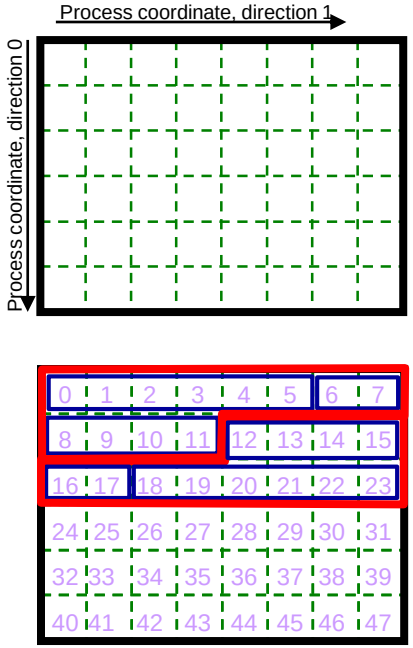




Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

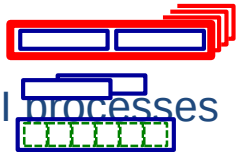
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

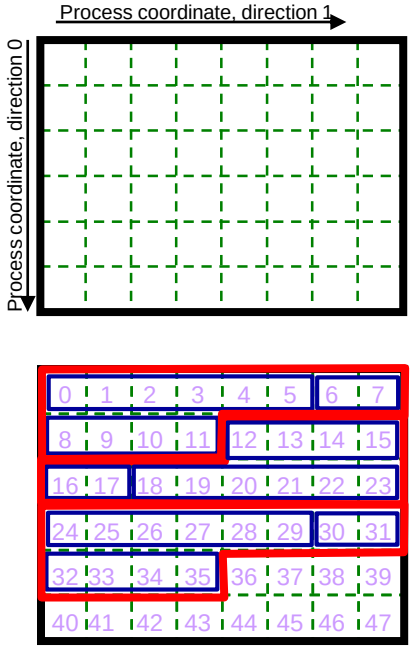




Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

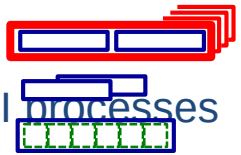
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

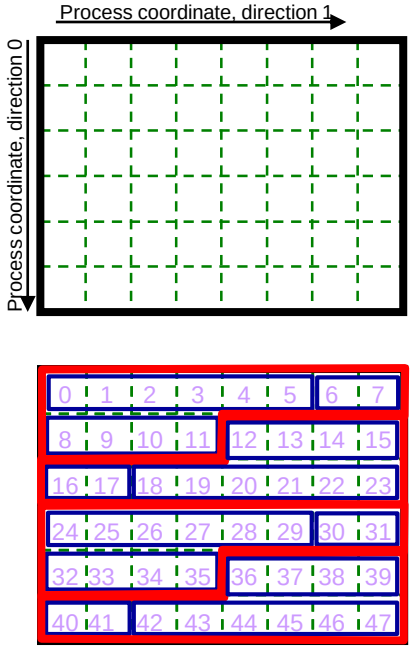




Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

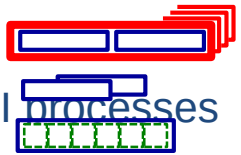
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

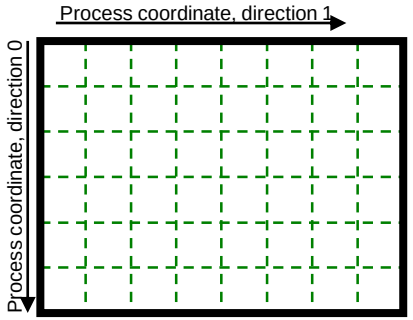


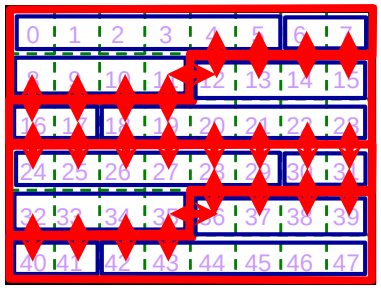


Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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 compute 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



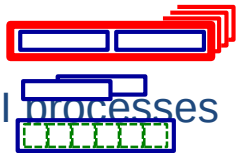


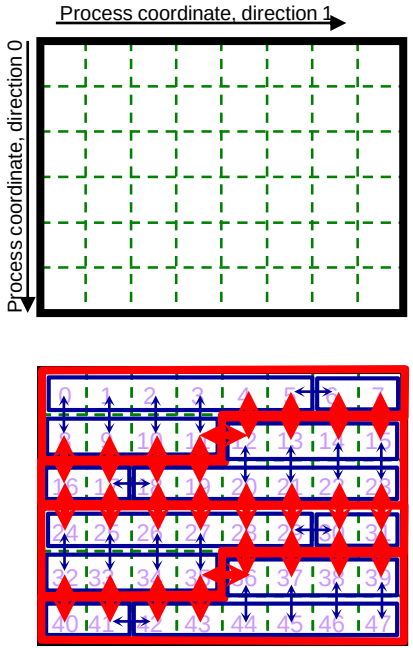


26 node-to-node (outer)

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

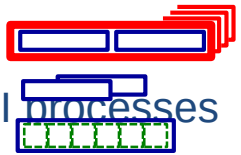
- Example:
  - 2-dim 6000 x 8080 data mesh points
  - To be parallelized on 48 cores
- Minimal communication
  - Subdomains as quadratic as possible
    - → minimal circumference
    - → minimal halo communication
  - virtual 2-dim process grid: 6 x 8
    with 1000 x 1010 mesh points/core
- Hardware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

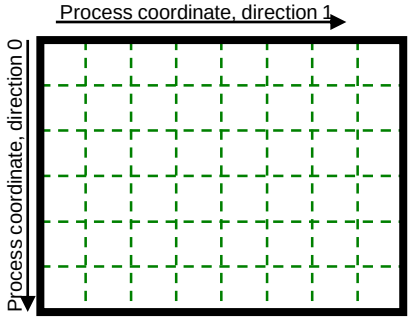


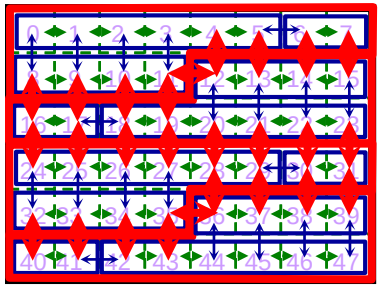


Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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 compute 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



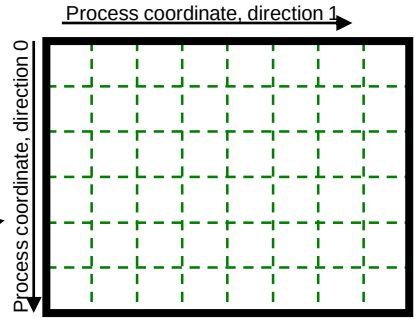


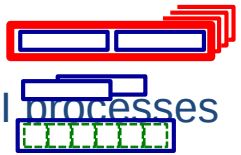


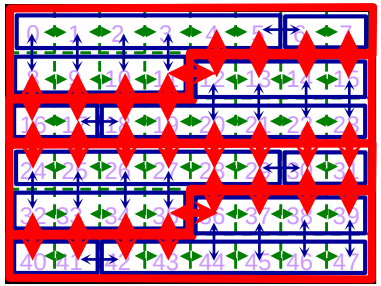
Non-optimal communications:
- **26** node-to-node (outer)
- **20** CPU-to-CPU (middle)
- 36 core-to-core (inner)

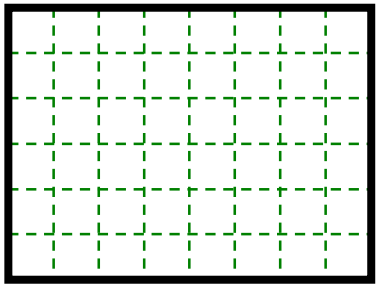
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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 compute 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







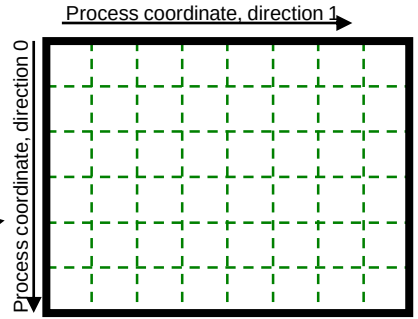


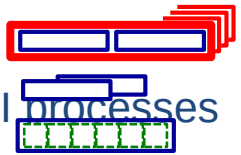
Non-optimal communications:
26 node-to-node (outer)
20 CPU-to-CPU (middle)
36 core-to-core (inner)

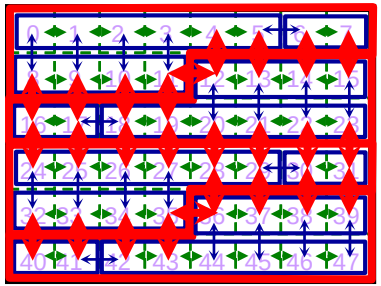
Optimized placement:

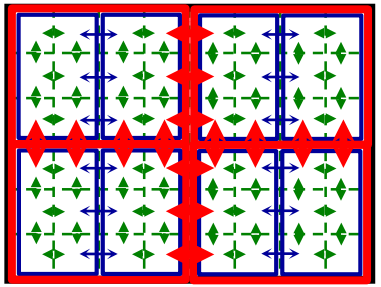
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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 compute 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







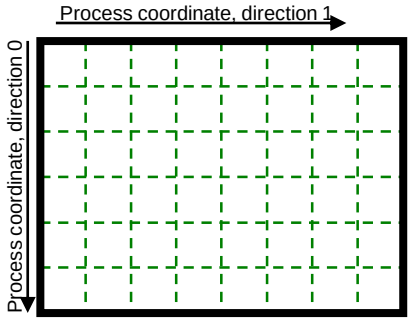


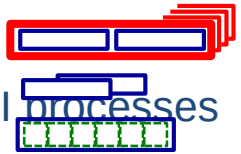
Non-optimal communications:
26 node-to-node (outer)
20 CPU-to-CPU (middle)
36 core-to-core (inner)

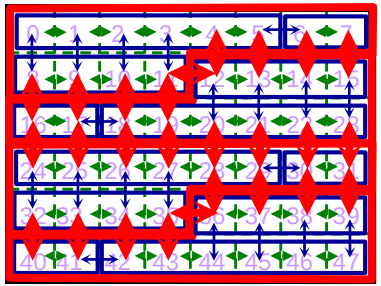
Optimized placement:
Only 14 node-to-node
Only 12 CPU-to-CPU
56 core-to-core

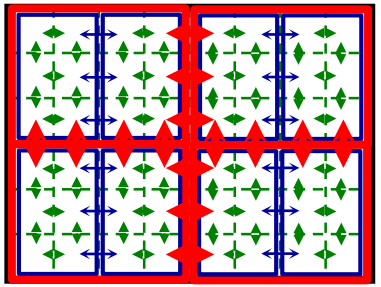
Re-numbering on a cluster of SMPs (cores / CPUs / nodes)

- 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 compute 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





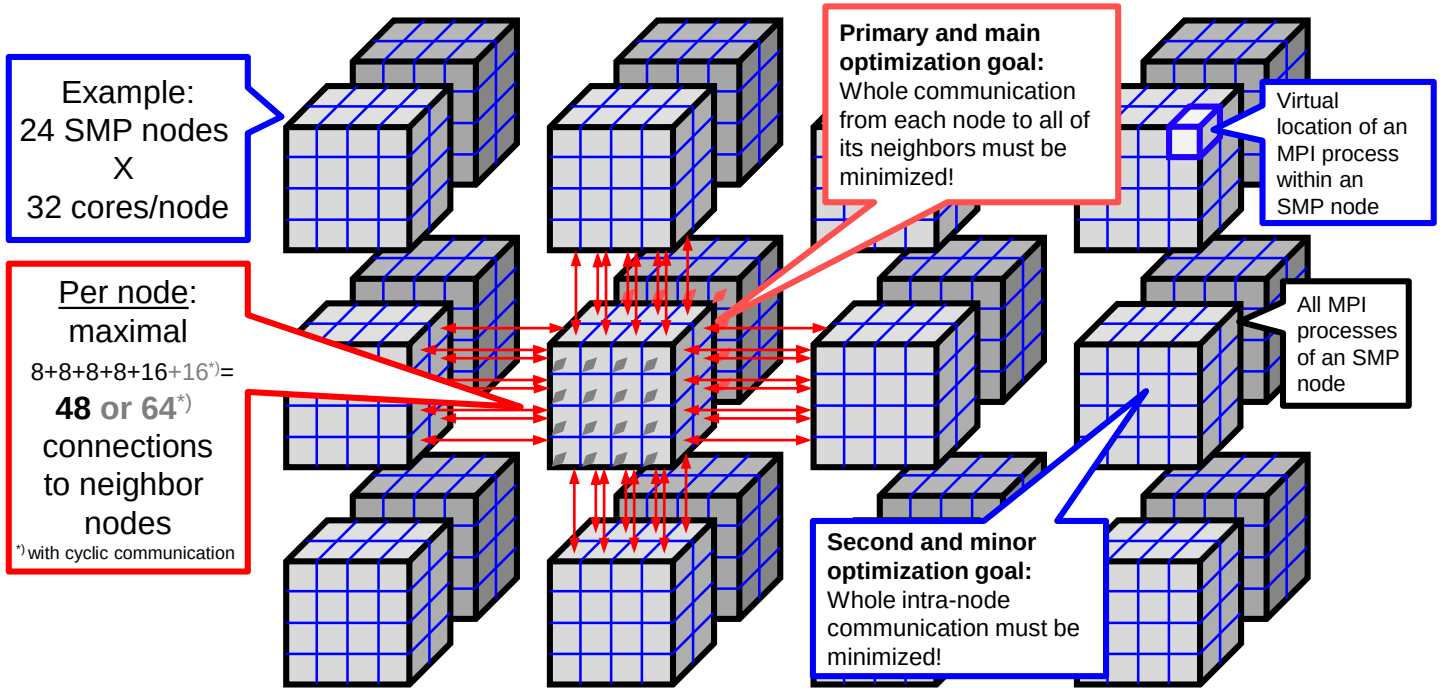




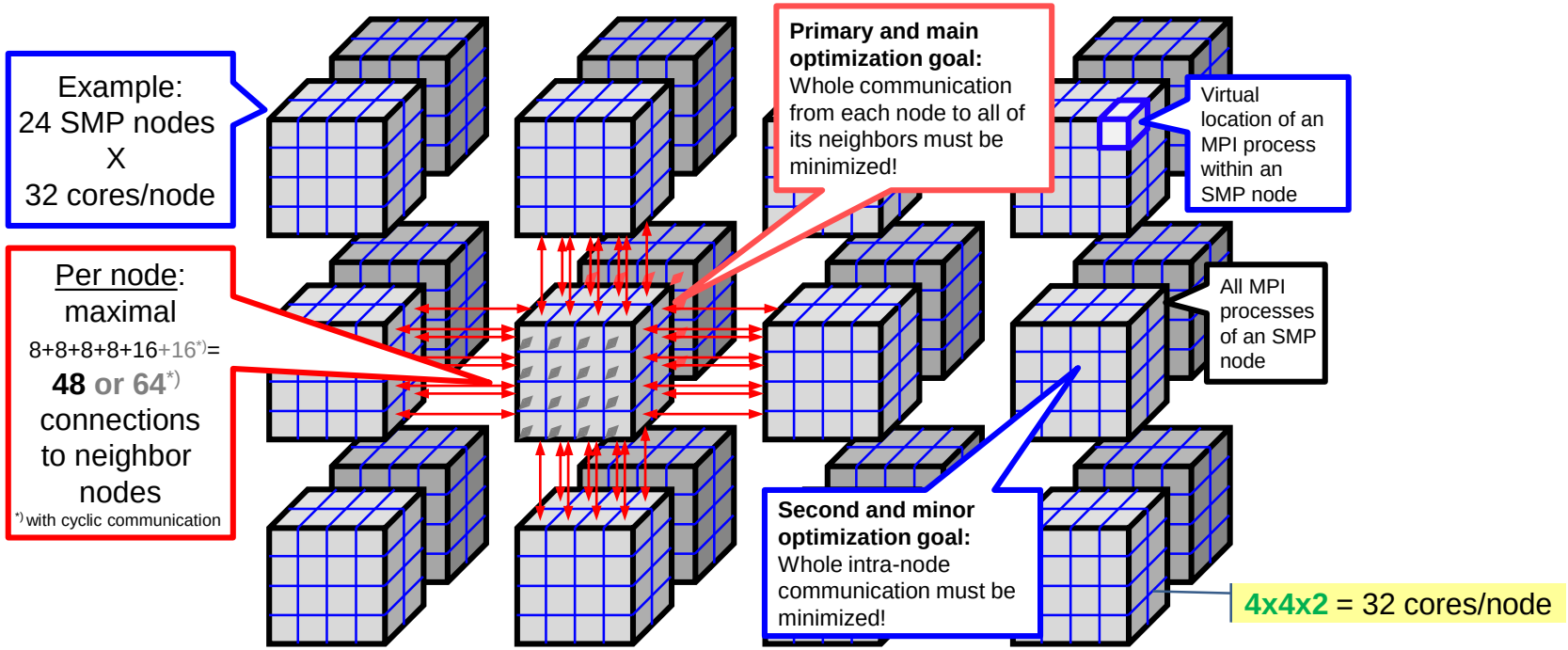
Non-optimal communications:
26 node-to-node (outer)
20 CPU-to-CPU (middle)
36 core-to-core (inner)

Optimized placement:
Only 14 node-to-node
Only 12 CPU-to-CPU
56 core-to-core

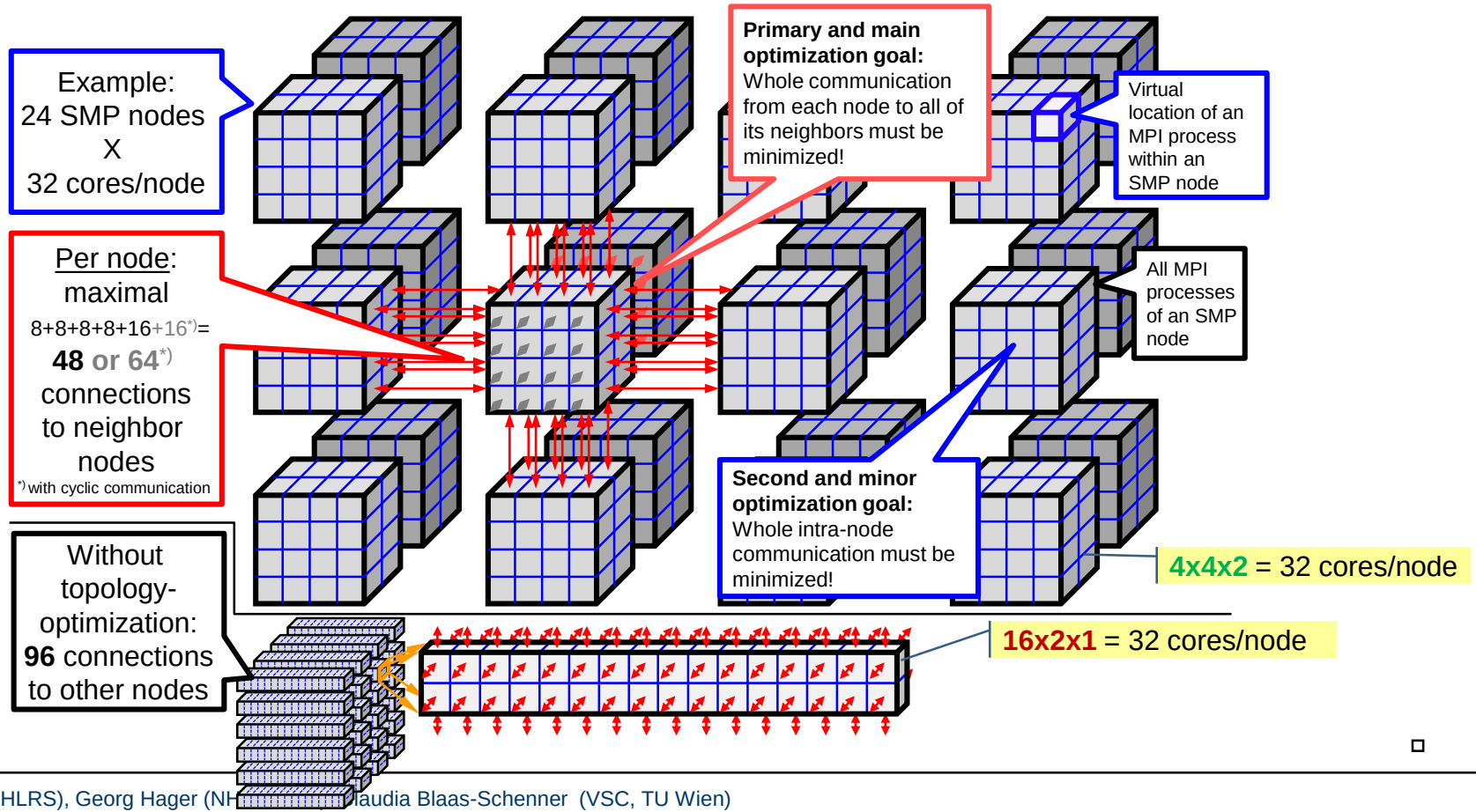
Hierarchical Cartesian Domain Decomposition



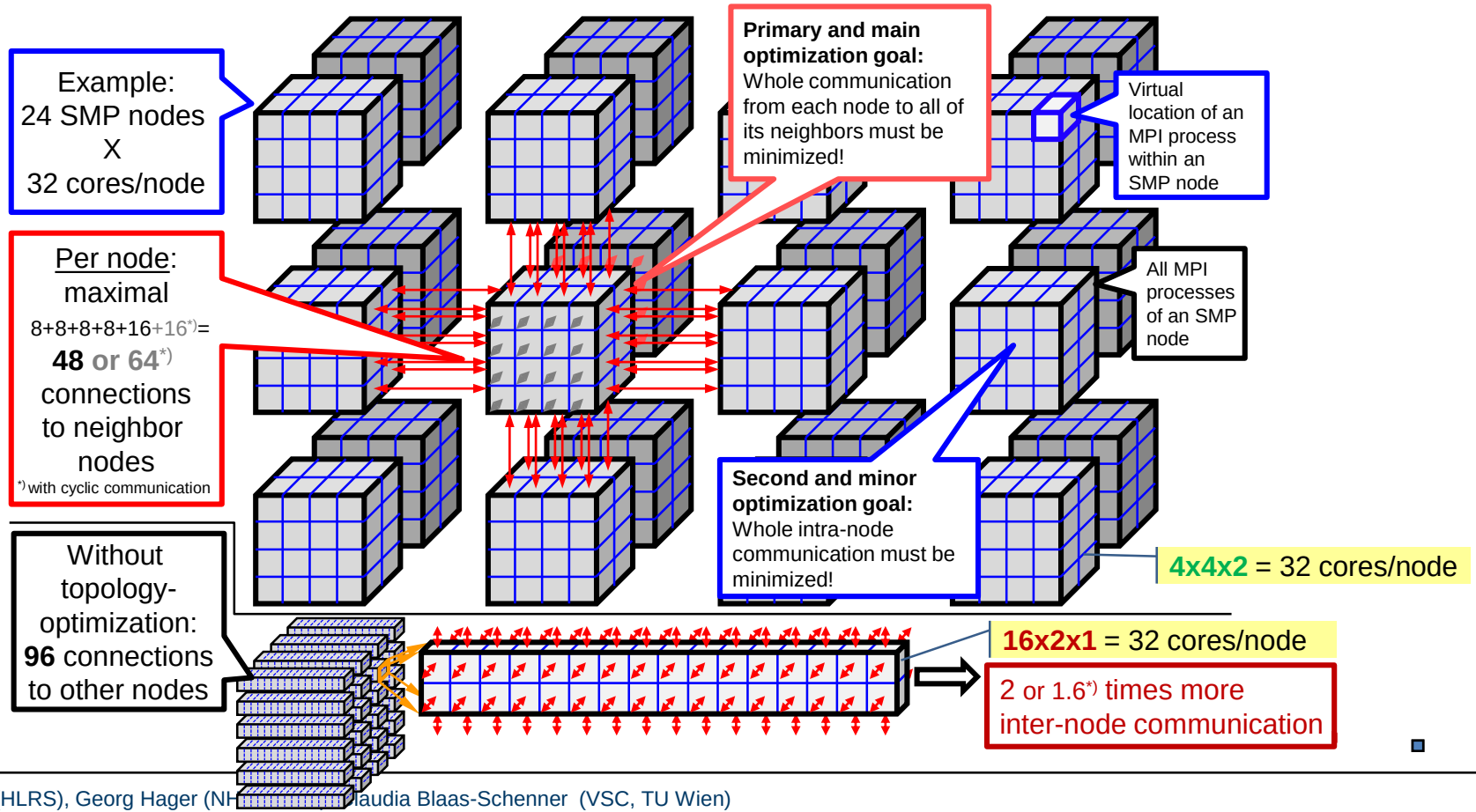
Hierarchical Cartesian Domain Decomposition



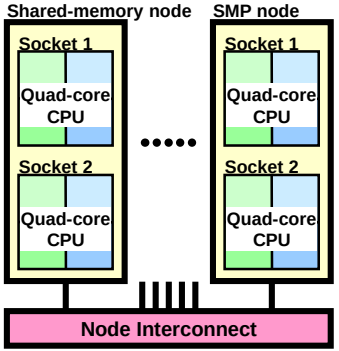
Hierarchical Cartesian Domain Decomposition



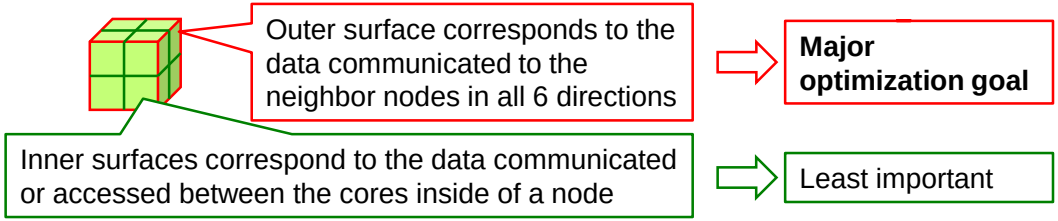
Hierarchical Cartesian Domain Decomposition



Levels of communication & data access



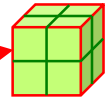
- Three levels:
  - Between the SMP nodes
  - Between the sockets inside of shared-memory node
  - Between the cores of a socket
- On all levels, the communication should be minimized:
  - With 3-dimensional sub-domains:
    - They should be as cubic as possible = minimal surface = minimal communication



    - "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

Levels of communication & data access

- Major goal: minimize inter-node communication
  - →Minimize sum of all outer subdomain surfaces
  - →Whole node subdomain shape as cubic as possible



- Secondary goal: minimize intra-node communication
  - →Minimize sum of all inner subdomain surfaces
  - →Inner subdomain shape as cubic as possible

Next slides:
MPI facilities to map topology to ranks in a communicator
→ Virtual Topologies

Programming models - pure MPI

How to → MPI Virtual Topologies

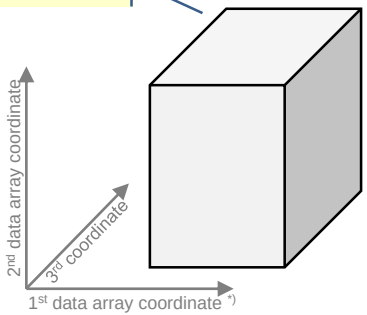
General considerations
The topology problem
> **The topology problem: How-to / Virtual Toplogies**
Rank renumbering for optimization
The Topology Problem: Unstructured Grids
Quiz
Scalability
Advantages & disadvantages, conclusions

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

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh

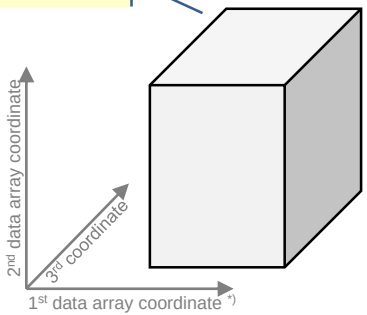


*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh



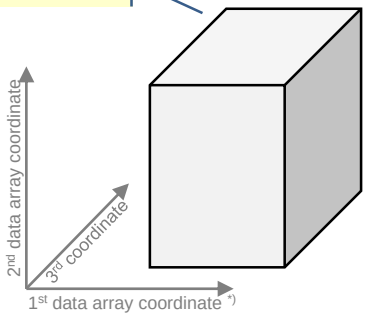
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

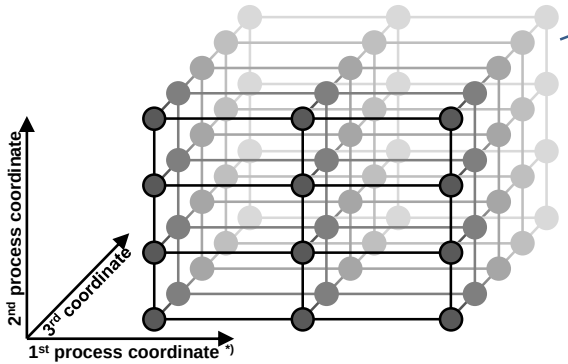
Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh



Virtual process grid



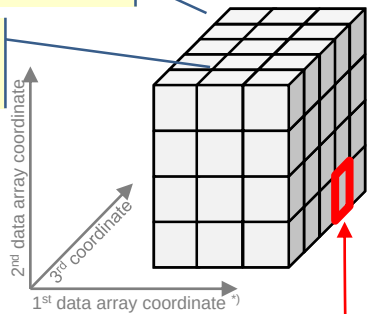
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

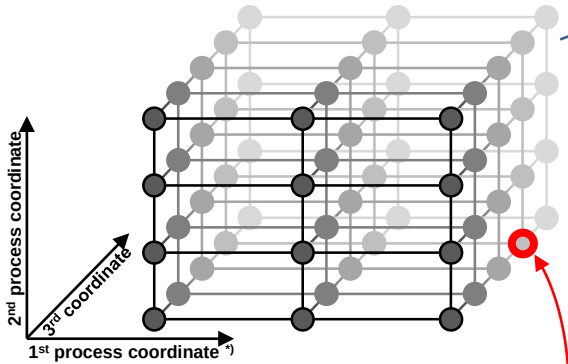
Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh with domain decomposition



Virtual process grid



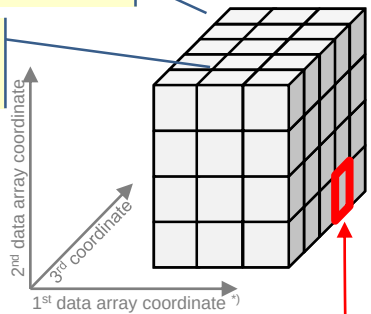
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**
- example:
  on process **ic_0=2, ic_1=0, ic_2=3 (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400)

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

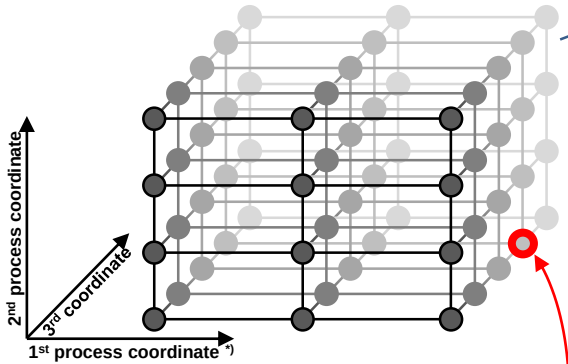
Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh with domain decomposition



Virtual process grid



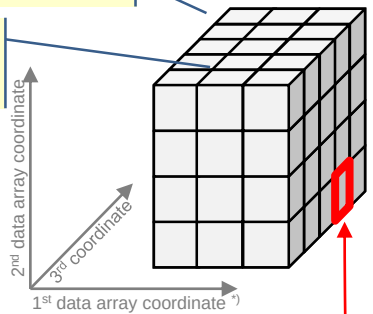
- on **3 x 4 x 5 = 60 processes**
- process coordinates **0..2, 0..3, 0..4**
- example:
  on process **ic_0=2, ic_1=0, ic_2=3 (rank=43)**
  decomposition, e.g., A(2001:3000, 1:1000, 301:400)
- **process coordinates:** handled with **virtual Cartesian topologies**

*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

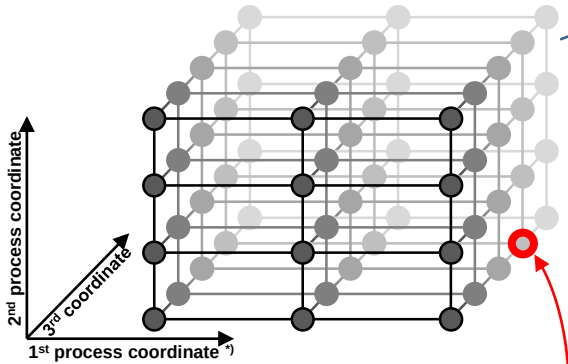
Domain decomposition example

- Global data array A(1:3000, 1:4000, 1:500)

Application data mesh with domain decomposition



Virtual process grid



- on **3 x 4 x 5 = 60 processes**
- 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)
- **process coordinates:** handled with **virtual Cartesian topologies**
- array decomposition: handled by the application program directly

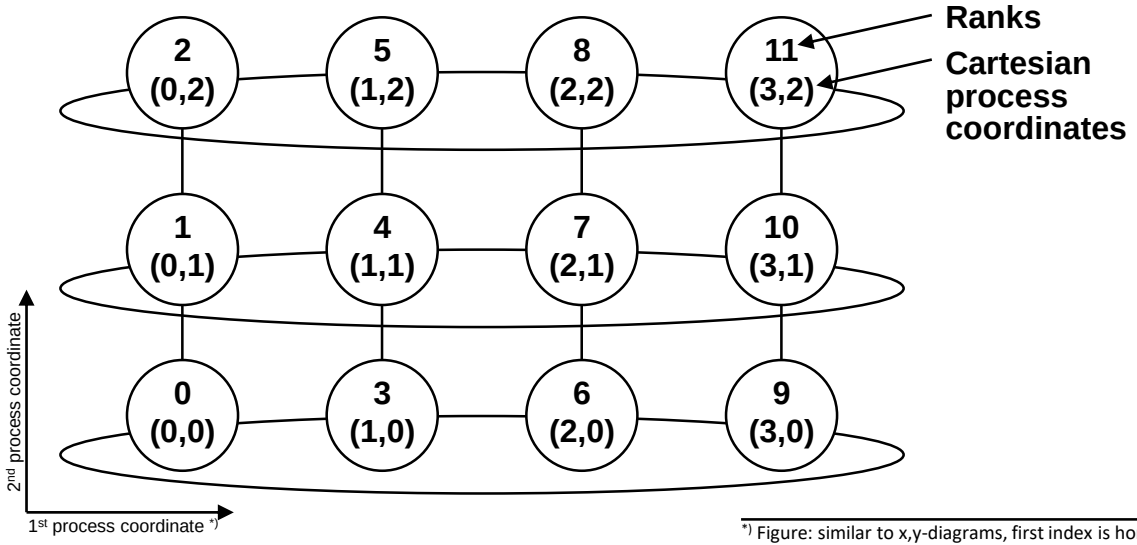
*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

Virtual Topologies

- Convenient process naming.
- Naming scheme to fit the communication pattern.
- Simplifies writing of code.
- Can allow MPI to optimize communications → see course Chapter 9-(3)

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:
  2-dimensional cylinder



*) Figure: similar to x,y-diagrams, first index is horizontal (i.e., not vertical as in a math matrix)

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.

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,

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)

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/C++

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm  *comm_cart)
```

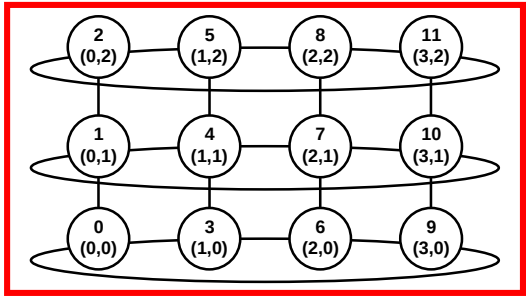
Fortran

```
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
```

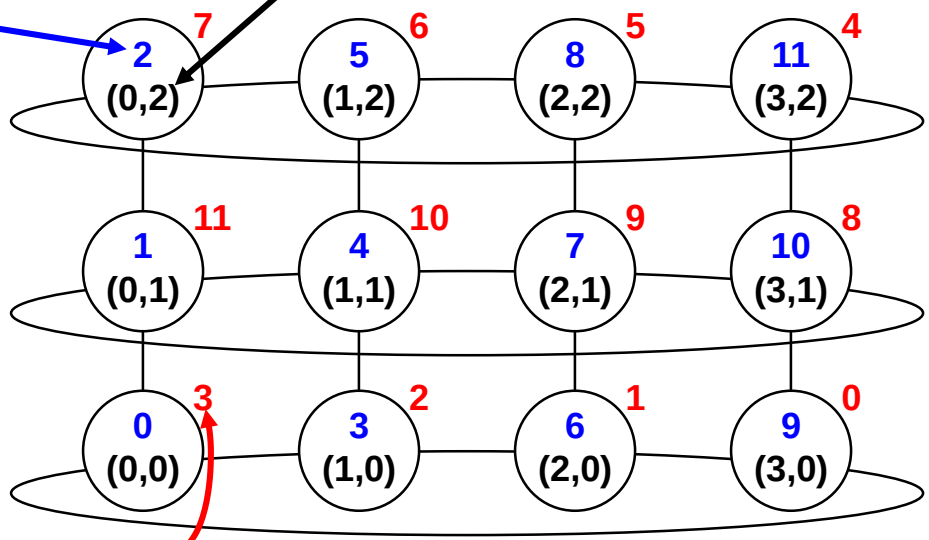
```
comm_old  =  MPI_COMM_WORLD
   ndims  =  2
    dims  = ( 4,      3      )
 periods  = ( 1,      0      )   (in C)
 periods  = ( .true., .false. )   (in Fortran)
 reorder  =  see next slide
```

e.g., size==12 factorized with MPI_Dims_create(), see later the slide „**Typical usage of MPI_Cart_create & MPI_Dims_create**"



Reordering

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



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

Typical use 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)
```

With reorder

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

C/C++

int MPI_Dims_create(int nnodes, int ndims, int *dims)

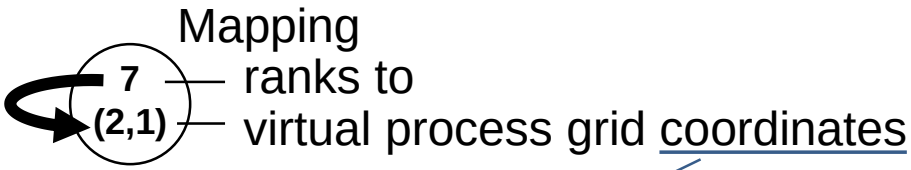
Fortran

MPI_DIMS_CREATE(nnodes, ndims, dims, ierror)

mpi_f08:
```
INTEGER                 :: nnodes, ndims, dims(*)
INTEGER, OPTIONAL       :: ierror
```

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

Cartesian Mapping Functions



C/C++

```
int MPI_Cart_coords(MPI_Comm comm_cart, int rank,
                    int maxdims, int *coords)
```

Fortran

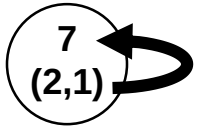
```
MPI_CART_COORDS(comm_cart, rank, maxdims, coords, ierror)
```

mpi_f08:

```
TYPE(MPI_Comm)      :: comm_cart
INTEGER             :: rank, maxdims, coords(*)
INTEGER, OPTIONAL   :: ierror
```

Cartesian Mapping Functions

Mapping process grid coordinates to ranks



C/C++

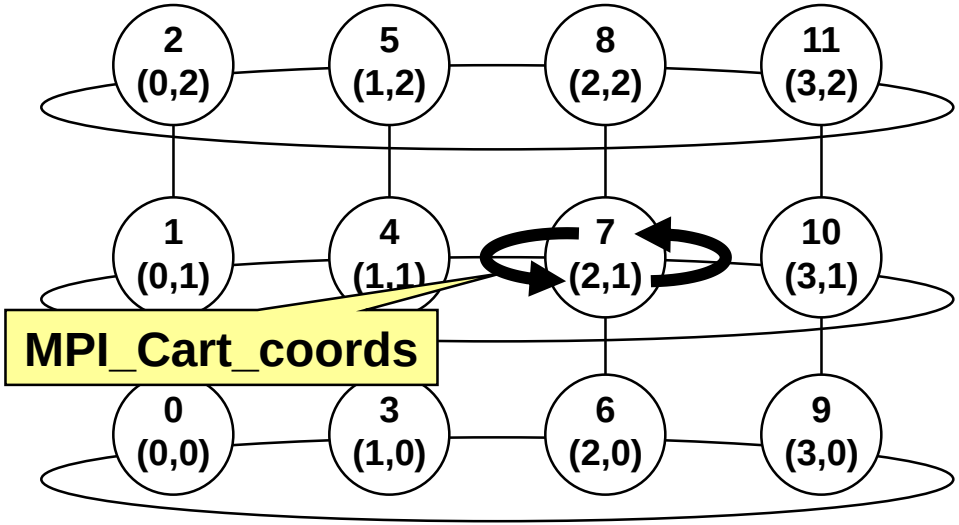
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int **rank*)

Fortran

MPI_CART_RANK(comm_cart, coords, *rank*, *ierror)*

```
mpi_f08:  TYPE(MPI_Comm)       :: comm_cart
          INTEGER              :: coords(*), rank
          INTEGER, OPTIONAL    :: ierror
```

A process' 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)
```

Ranks of neighboring processes

C/C++

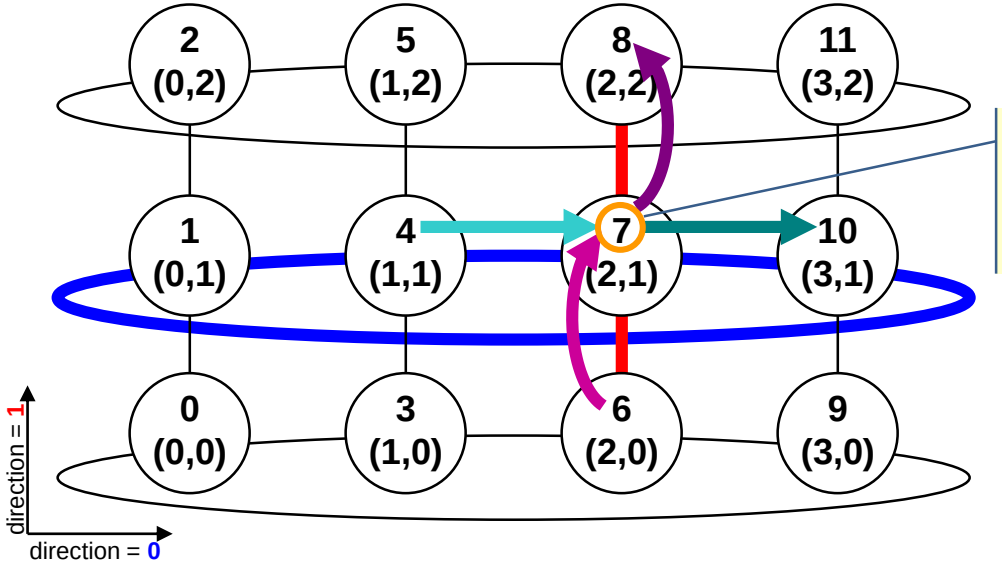
```
int MPI_Cart_shift(MPI_Comm comm_cart, int direction, int disp,
                   int *rank_source, int *rank_dest)
```

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
```

- 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



my_rank in comm_cart is invisible input argument to MPI_Cart_shift

```
call MPI_Cart_shift(comm_cart, direction, disp, rank_source, rank_dest, ierror)
```

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

Sparse Collective Operations on Process Topologies

- Sparse neighbor communication within MPI process topologies (Cartesian and (distributed) graph): — New in MPI-3.0
  - MPI_(I)NEIGHBOR_ALLTOALL (V,W)
  - MPI_(I)NEIGHBOR_ALLGATHER (V)

  = perfect scalable !?
- 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

Sparse Collective Operations on Process Topologies

- Sparse neighbor communication — New in MPI-3.0
  within MPI process topologies (Cartesian and (distributed) graph):
  - MPI_(I)NEIGHBOR_ALLTOALL (V,W)
  - MPI_(I)NEIGHBOR_ALLGATHER (V)

  = perfect scalable !?
- 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

Sparse Collective Operations on Process Topologies

- Sparse neighbor communication — New in MPI-3.0
  within MPI process topologies (Cartesian and (distributed) graph):
  - MPI_(I)NEIGHBOR_ALLTOALL (V,W)
  - MPI_(I)NEIGHBOR_ALLGATHER (V)

  = perfect scalable !?
- 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)

Sparse Collective Operations on Process Topologies

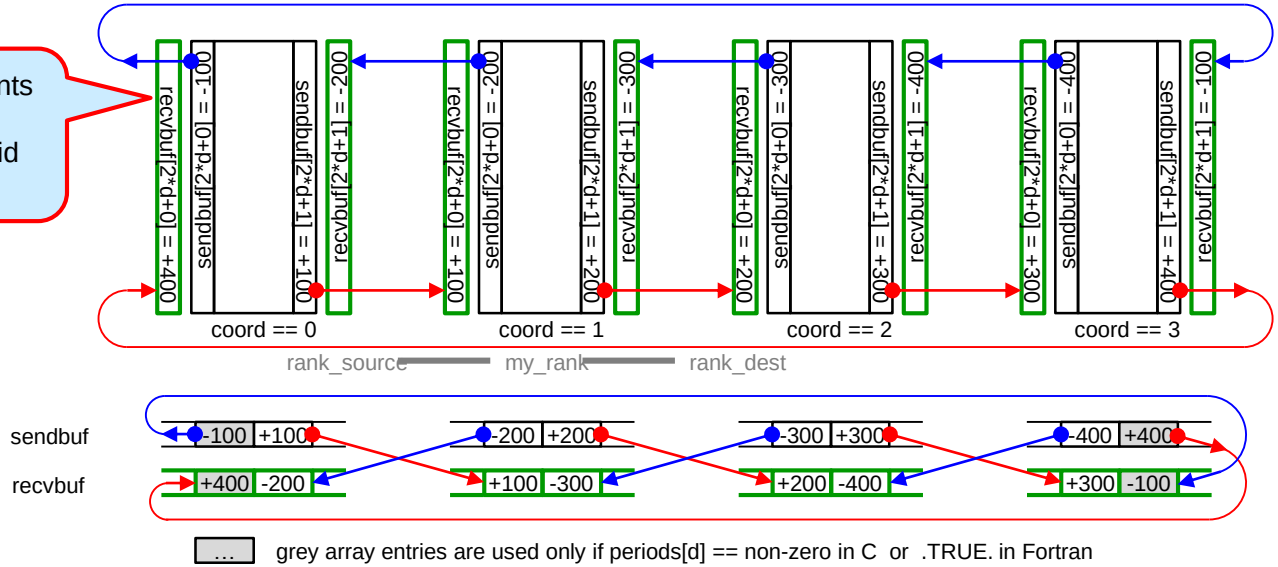
- Sparse neighbor communication within MPI process topologies (Cartesian and (distributed) graph): — New in MPI-3.0
  - MPI_(I)NEIGHBOR_ALLTOALL (V,W)
  - MPI_(I)NEIGHBOR_ALLGATHER (V)

  = perfect scalable !?
- 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.

Periodic MPI_NEIGHBOR_ALLTOALL in direction *d* with 4 processes

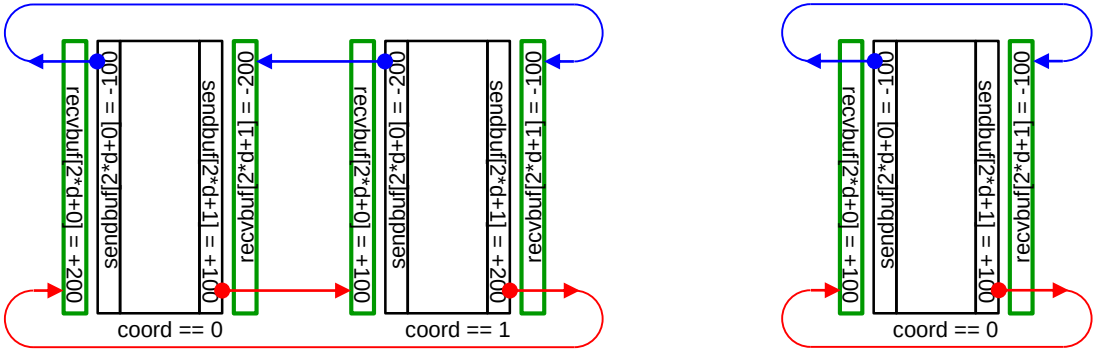
Clarified in MPI-4.0

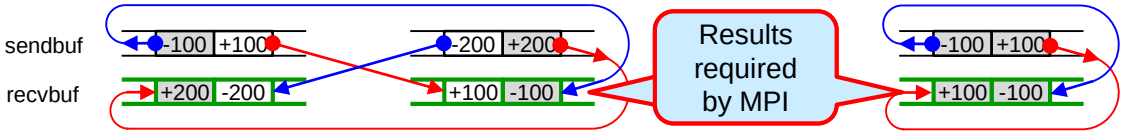
This figure represents one direction *d*. Of course, it is valid for any direction



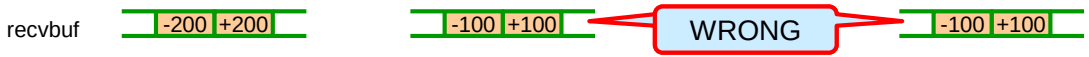
skipped

Wrong implementations of periodic MPI_NEIGHBOR_ALLTOALL with only 2 and 1 processes





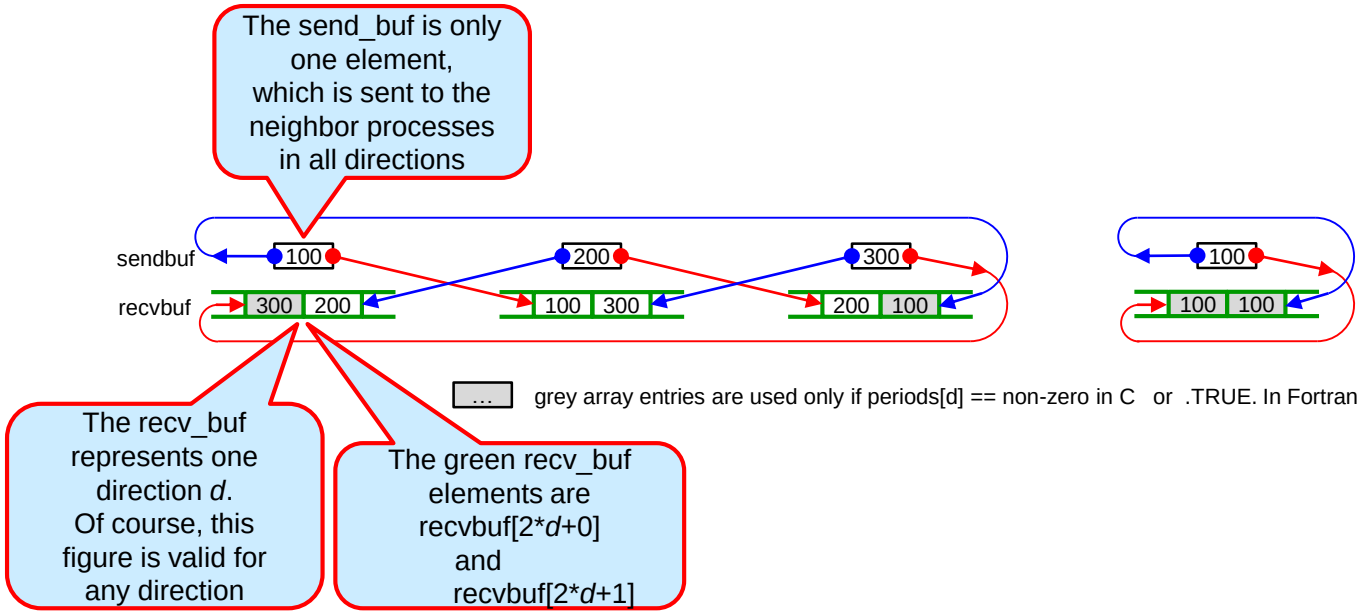
Wrong results with **openmpi/4.0.1-gnu-8.3.0** and **cray-mpich/7.7.6** with 2 and 1 processes:



skipped

Communication pattern of MPI_NEIGHBOR_ALLGATHER

Clarified in MPI-4.0

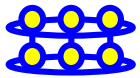


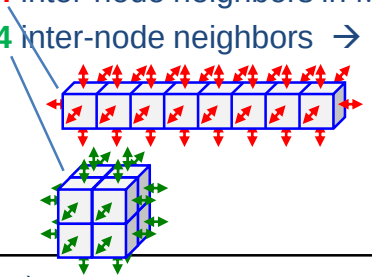
Programming models - pure MPI

Rank renumbering for optimization

General considerations
The topology problem
The topology problem: How-to / Virtual Toplogies
> **Rank renumbering for optimization**
The Topology Problem: Unstructured Grids
Quiz
Scalability
Advantages & disadvantages, conclusions

Rank renumbering for optimization

- **When is it not needed?**
  - → Hybrid MPI+OpenMP with 1, 2, or 3 MPI processes per shared-memory node
- **When is it not helpful?**
  - → Dynamic load balancing that changes the process-to-process communication pattern (typically only with graph topologies)
- **When do we need it?**
  - → Communication win with >= 4 MPI processes per shared-memory node
  - Example with 6 or 8 processes per shared-memory node:
    - Sequential ranking 6x1x1 or **8x1x1** topology → 26 or **34** inter-node neighbors in MPI_COMM_WORLD
    - Renumbered 3x2x1 or **2x2x2** topology → 22 or **24** inter-node neighbors → 15% or **29%** win via Cartesian topo.
- **How can we implement it?**
  - → MPI virtual topologies 



Rank renumbering for optimization – problems

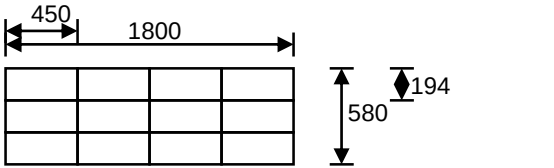
1. All MPI libraries provide the necessary interfaces ☺ ☺ ☺,
   but **without** renumbering in some MPI-libraries ☹ ☹ ☹

2. The existing MPI-4.0 interfaces are not optimal:
   - Application topology awareness:
     application-specific data mesh sizes or direction-dependent communication requirements are not accounted for → next slide
   - Hardware topology awareness:
     the factorization of the number of processes into several dimensions cannot leverage hardware topology information → next slide

3. The application must be prepared for rank renumbering
   - Ideally, data distribution happens after renumbering (see slide Typical use of MPI_Cart_create & MPI_Dims_create)

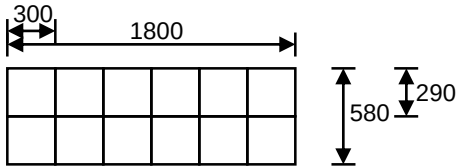
The existing MPI-4.0 interfaces are not optimal: examples

- Application topology awareness
  - 2-D example with 12 MPI processes and data mesh size 1800x580
    - MPI_Dims_create → 4x3



Boundary of a subdomain = 2(450+194) = **1288** ☹

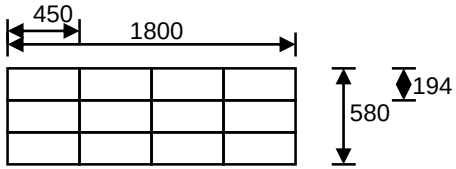
- data mesh aware → 6x2 processes



Boundary of a subdomain = 2(300+290) = **1180** ☺

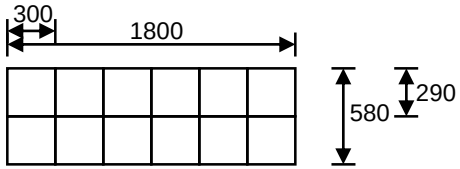
The existing MPI-4.0 interfaces are not optimal: examples

- Application topology awareness
  - 2-D example with 12 MPI processes and data mesh size 1800x580
    - MPI_Dims_create → 4x3



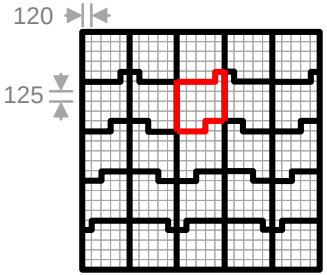
Boundary of a subdomain = 2(450+194) = **1288** ☹

- data mesh aware → 6x2 processes



Boundary of a subdomain = 2(300+290) = **1180** ☺

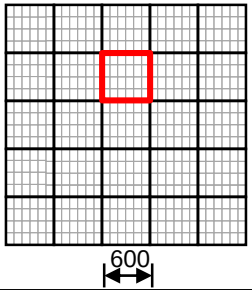
- Hardware topology awareness
  - 2-D example with 25 nodes x 24 cores and data mesh size 3000x3000
    - MPI_Dims_create → 25 x 24



Accumulated communication per node

O(10x120+12x125) = O(**2700**) ☹

- Hardware aware → 30 x 20 = (**5 nodes** x 6 cores) **X** (**5 nodes** x 4 cores)



Accumulated communication per node

O(4x600) = O(**2400**) ☺

Goals of MPI_Dims_create + MPI_Cart_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**
  - an **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

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 without comm arg. → 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 120 x 3000 mesh points per node
          ➔ too much node-to-node 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 without comm arg. → 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 120 x 3000 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_Dims+Cart_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 & 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)

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

- **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 );
```

input for **hardware**-awareness

and **application-topology**-awareness

  - Arguments: see existing MPI_Dims_create & MPI_Cart_create **/** dim_weights[ndims] → next slide
  - Goals:
    - **Choose** 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 is minimal (e.g., with MPI_NEIGHBOR_ALLTOALL, MPI_SENDRECV, or nonblockuing MPI_ISEND/IRECV)

The new interface – proposed for MPI-4.1, continued

- **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 );
```

input for **hardware**-awareness

and **application-topology**-awareness

The new hardware- & application-topology-aware interface

  - Arguments: see existing MPI_Dims_create & MPI_Cart_create **/** dim_weights[ndims] → next slide
  - Goals:
    - **Choose** an ndims-dimensional factorization of #processes of comm_old (→ dims)
    - **and** an appropriate reordering of the ranks (→ comm_cart),

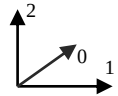
    such that the execution time of a communication step along the virtual process grid is minimal (e.g., with MPI_NEIGHBOR_ALLTOALL, MPI_SENDRECV, or nonblockuing MPI_ISEND/IRECV)

How to specify the dim_weights?

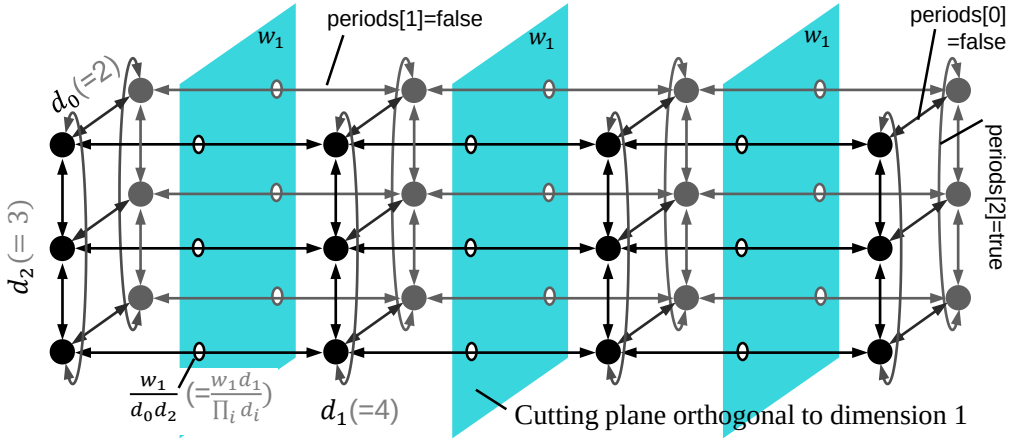
- Given: comm_old (e.g., MPI_COMM_WORLD), ndims (e.g., 3 dimensions)
- This means, **the domain decomposition has not yet taken place**!
- Goals for dim_weights and the API at all:
  - Easy to understand
  - Easy to calculate
  - Relevant for typical Cartesian communication patterns (MPI_Neighbor_alltoall or similar)
  - 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)

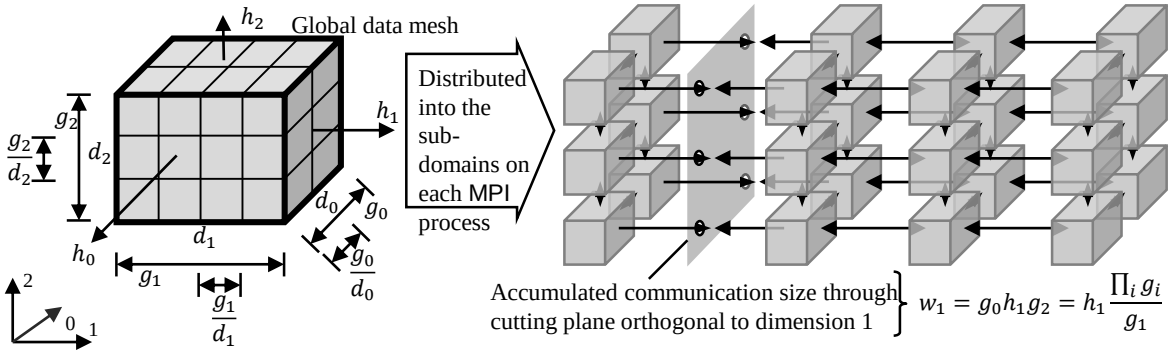


Three dimensions,
i.e., ndims=3



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



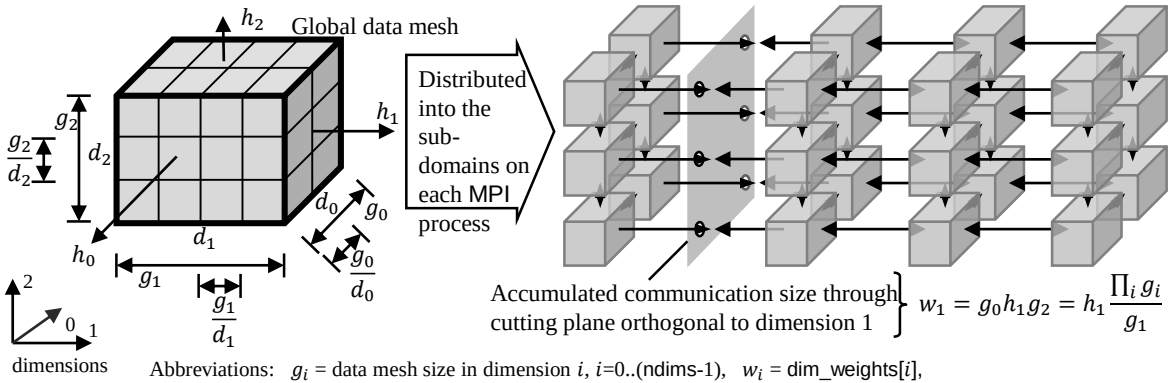
Example for the calculation of the accumulated communication size $\boldsymbol{w_{i,i=0..2}}$ in each dimension.

Given:

- 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 $\boldsymbol{d_{i,i=0..2}}$

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 $\boldsymbol{w_{i,i=0..2}}$ in each dimension.

Given:

- 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 $\boldsymbol{d_{i,i=0..2}}$

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

$$w_i = h_i \frac{\prod_j g_j}{g_i}$$

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*);

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 );
```

- 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/self-study-materials
    → Practical → MPI31.tar.gz → MPI/tasks/C/eurompi18/
  - More details, see this talk+slides „Hybrid Programming in HPC – MPI+X”

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

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

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?

Communication time
Memory access time

Through rank reordering

rank reordering may still help
if ≥ 4 MPI processes per SMP node

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
  - Communication time
  - Memory access time
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
    - Through rank reordering

rank reordering may still help
if ≥ 4 MPI processes per SMP node

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
  - Communication time
  - Memory access time
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
    - Through rank reordering
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
    - rank reordering may still help
    - if ≥ 4 MPI processes per SMP node
  - Can it reduce replicated data?

Questions addressed in this tutorial

Where we are?

- What is the performance impact of system topology?
  - Communication time
  - Memory access time
- How do I map my programming model on the system to my advantage?
  - How do I do the split into MPI+X?
  - Where do my processes/threads run? How do I take control?
  - Where is my data?
  - How can I minimize communication overhead?
    - Through rank reordering
- How does hybrid programming help with typical HPC problems?
  - Can it reduce communication overhead?
    - rank reordering may still help
    - if ≥ 4 MPI processes per SMP node
  - Can it reduce replicated data?
- How can I leverage multiple accelerators?
  - What are typical challenges?

Typical use of MPIX_Cart_weighted_create

```
#define ndims 3
int i, nnodes, world_myrank, cart_myrank, dims[ndims], periods[ndims], my_coords[ndims];
int global_array_dim[ndims], halo_width[ndims], local_array_dim[ndims], local_array_size=1;
double dim_weights[ndims], global_array_size=1.0;
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]=…;
  global_array_dim[i]=…; halo_width[i]=…;
  global_array_size = global_array_size * (double)(global_array_dim[i]);
}
for (i=0; i<ndims; i++) {
  dim_weights[i] = (double)(halo_width[i]) * global_array_size / (double)(global_array_dim[i]);
}
MPIX_Cart_weighted_create(MPI_COMM_WORLD, ndims, dim_weights, dims, periods, MPI_INFO_NULL, dims, &comm_cart);
MPI_Comm_rank(comm_cart, &cart_myrank);
MPI_Cart_coords(comm_cart, cart_myrank, ndims, my_coords, ierror)
for (i=0; i<ndims; i++) {
  local_array_dim[i] = global_array_dim[i] / dims[i];
  local_array_size = local_array_size * local_array_dim[i];
}
local_data_array = malloc(sizeof(…) * local_array_size);
```

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

Virtual Cartesian MPI topologies – summary

- Relevant for modern clusters comprising multicore nodes
- Optimizes only the communication

  If communication is irrelevant (in €)
  → don't care about reordering
  (observe cost/benefit)
- The new (and weighted) optimizing routines are easy to use for Cartesian problems
- Be aware that the MPI_Cart_..._create routines renumber the communicator

  of course with reorder=true

Programming models - pure MPI

The Topology Problem: Unstructured Grids

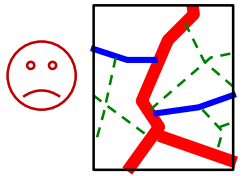
General considerations
The topology problem
The topology problem: How-to / Virtual Toplogies
Rank renumbering for optimization
> **The Topology Problem: Unstructured Grids**
Quiz
Scalability
Advantages & disadvantages, conclusions

Virtual MPI Topologies – 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.
- Many MPI libraries still do not optimize the graph topologies ...
  - an (not too complicated) alternative is shown on next slides
- 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 file system

Hierarchical DD for 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 (not verified by us authors)
- Multi-level DD:
  - Top-down: Several levels of (Par)Metis
    → unbalanced communication

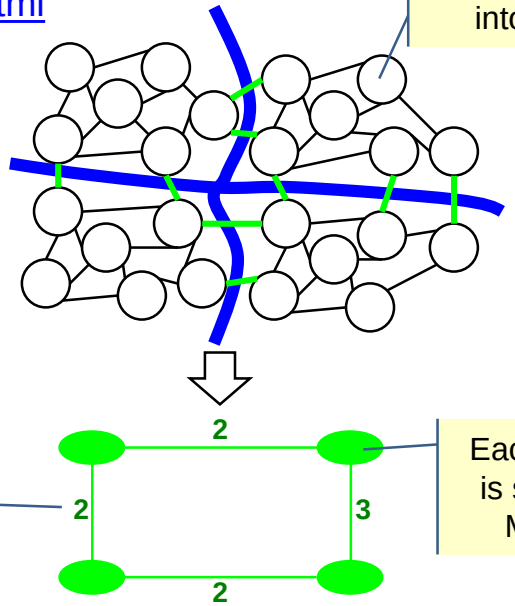


  - Bottom-up: Low level DD
    + higher level recombination
    → based on DD of the grid of subdomains

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)

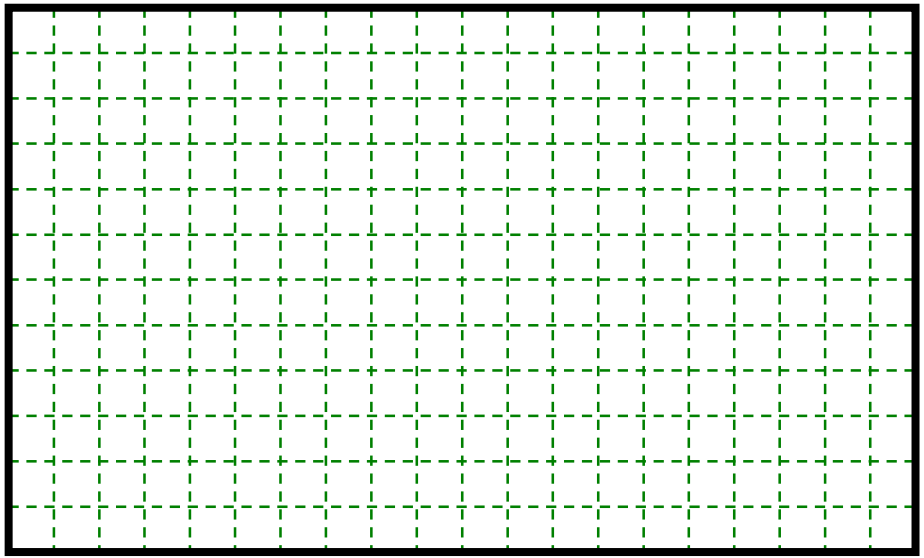
Unstructured Grid / Data Mesh

- Multi-level Domain Decomposition through Recombination

Unstructured Grid / Data Mesh

- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid — e.g., with Metis / Scotch

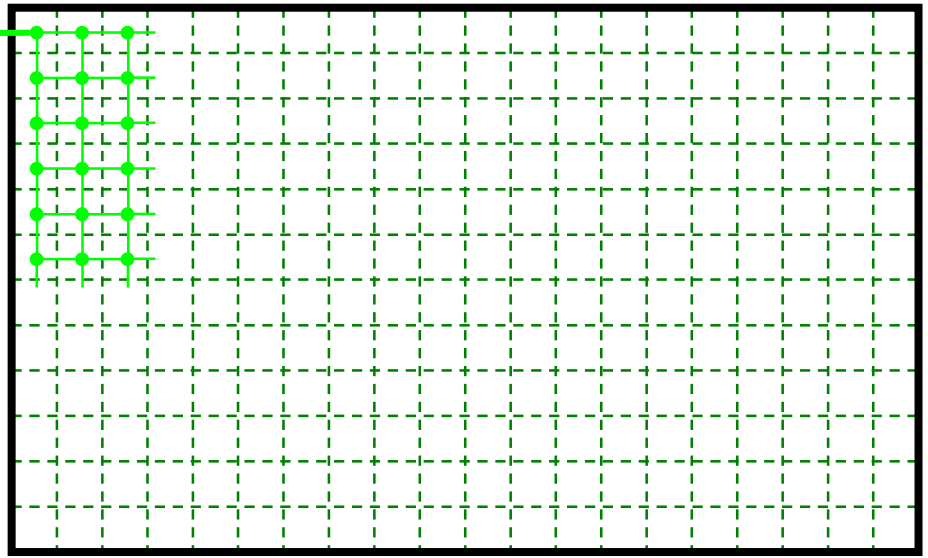


Unstructured Grid / Data Mesh

- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid — e.g., with Metis / Scotch

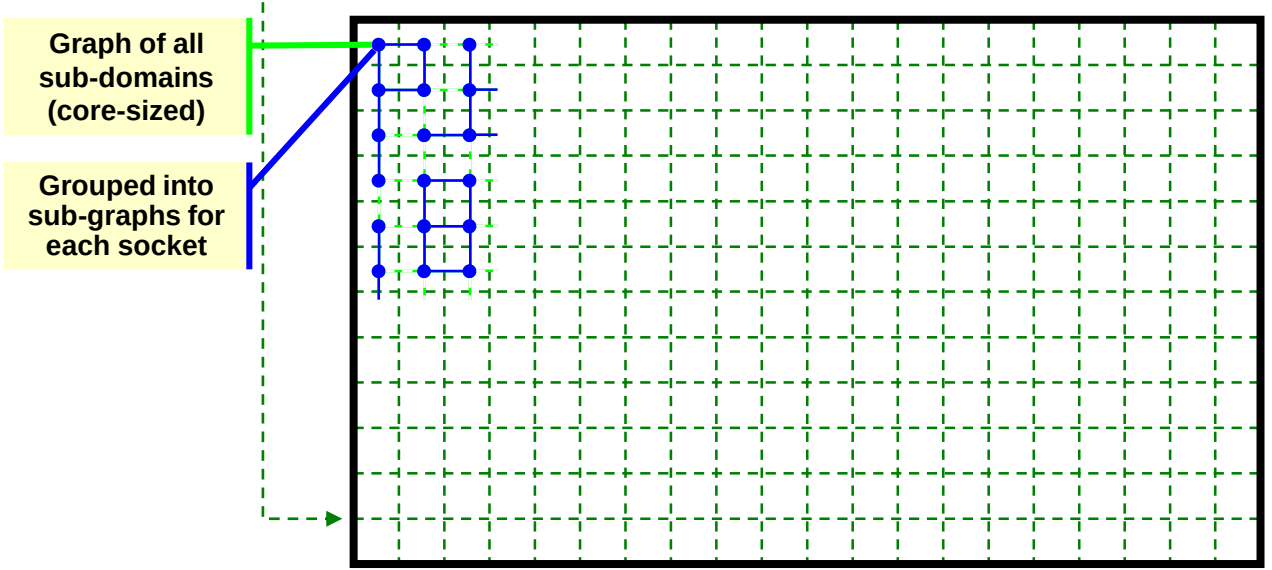
Graph of all sub-domains (core-sized)



Unstructured Grid / Data Mesh

- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid — e.g., with Metis / Scotch



Unstructured Grid / Data Mesh

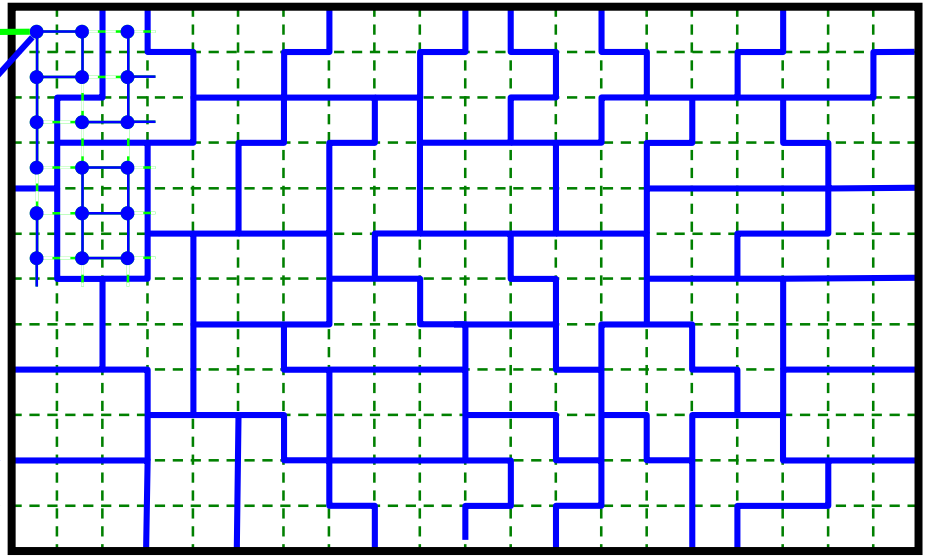
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
2. **Numa-domain-level DD:** recombining of core-domains

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped into sub-graphs for each socket



Unstructured Grid / Data Mesh

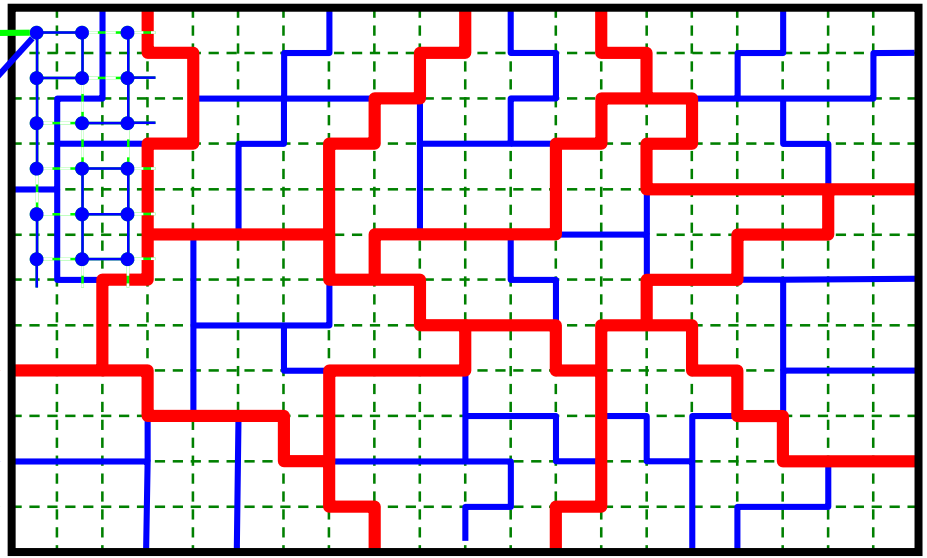
- Multi-level Domain Decomposition through Recombination

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

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped into sub-graphs for each socket



Unstructured Grid / Data Mesh

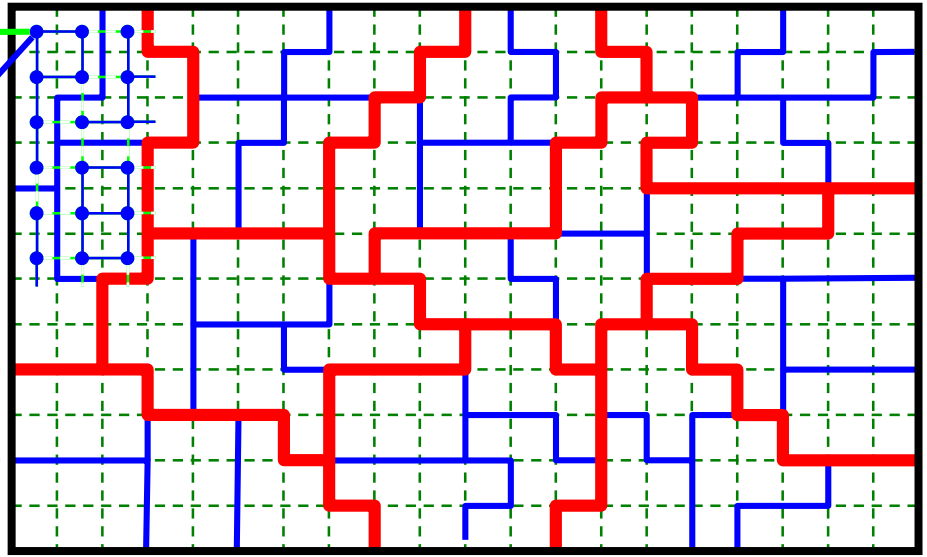
- Multi-level Domain Decomposition through Recombination

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

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

Unstructured Grid / Data Mesh

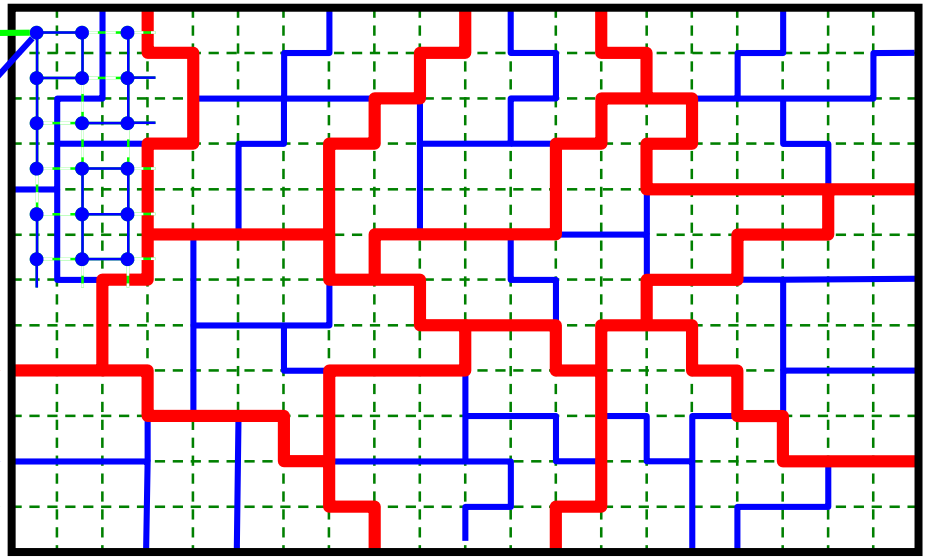
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
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
   as done in MPI_COMM_WORLD (e.g., sequentially)

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

Unstructured Grid / Data Mesh

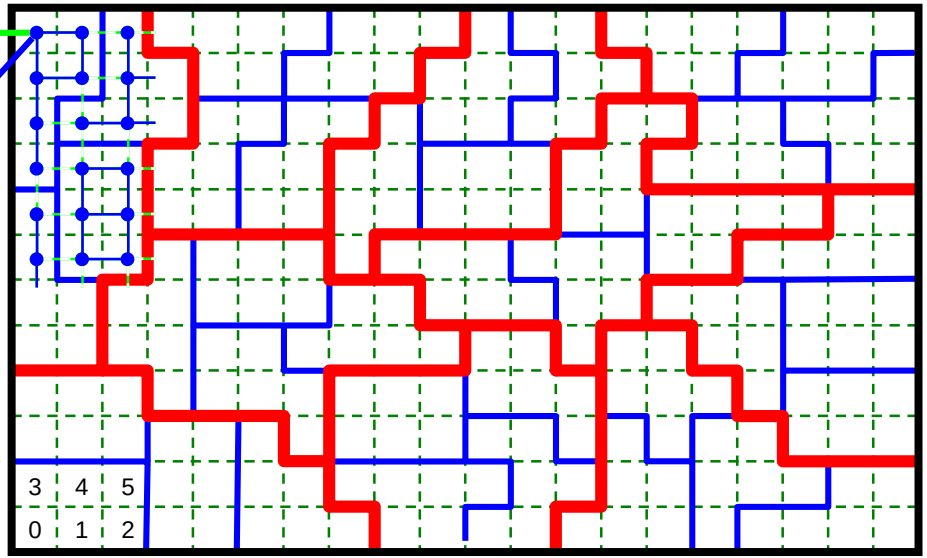
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
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
   as done in MPI_COMM_WORLD (e.g., sequentially)

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

Unstructured Grid / Data Mesh

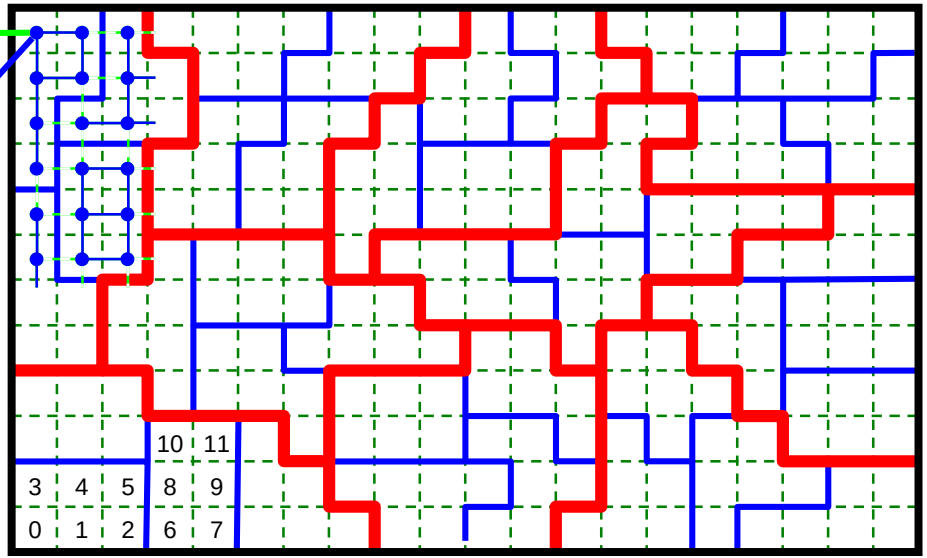
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
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
   as done in MPI_COMM_WORLD (e.g., sequentially)

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

Unstructured Grid / Data Mesh

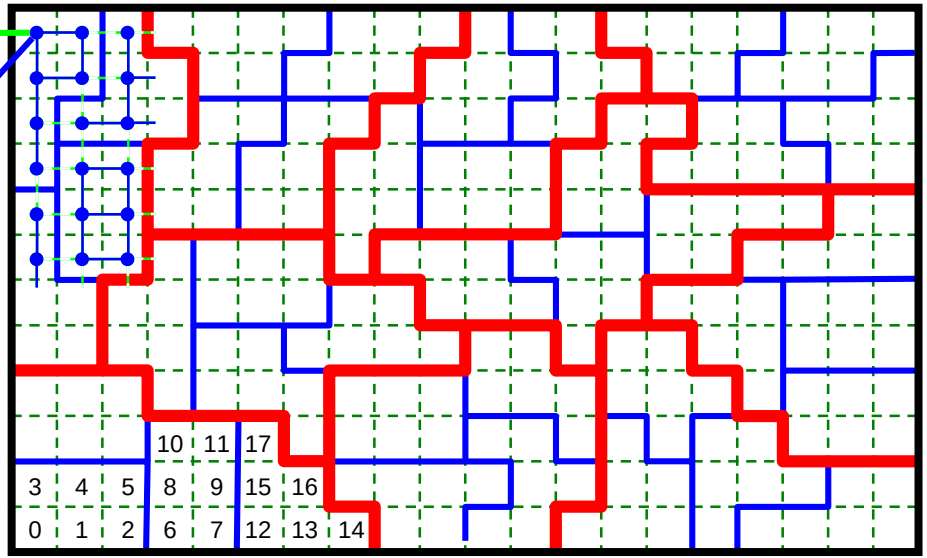
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
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
   as done in MPI_COMM_WORLD (e.g., sequentially)

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

Unstructured Grid / Data Mesh

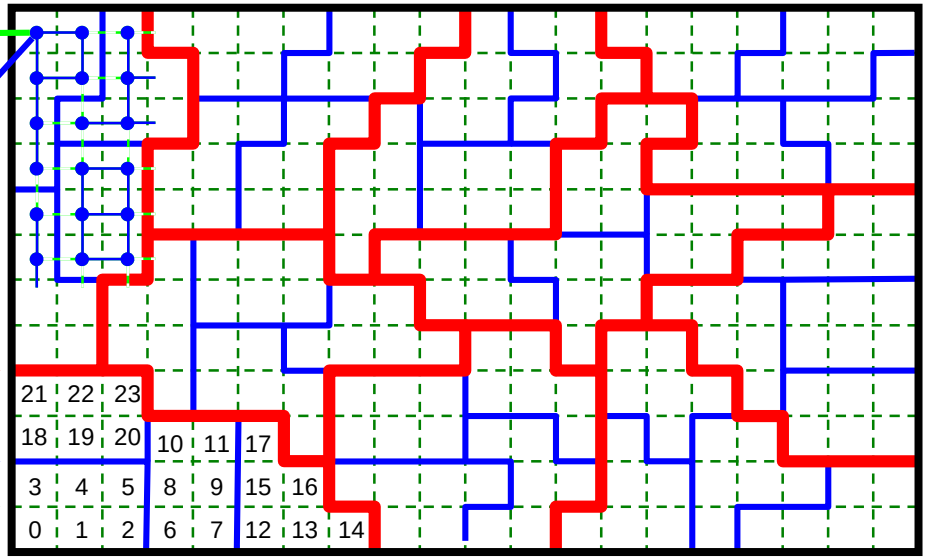
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
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
   as done in MPI_COMM_WORLD (e.g., sequentially)

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

Unstructured Grid / Data Mesh

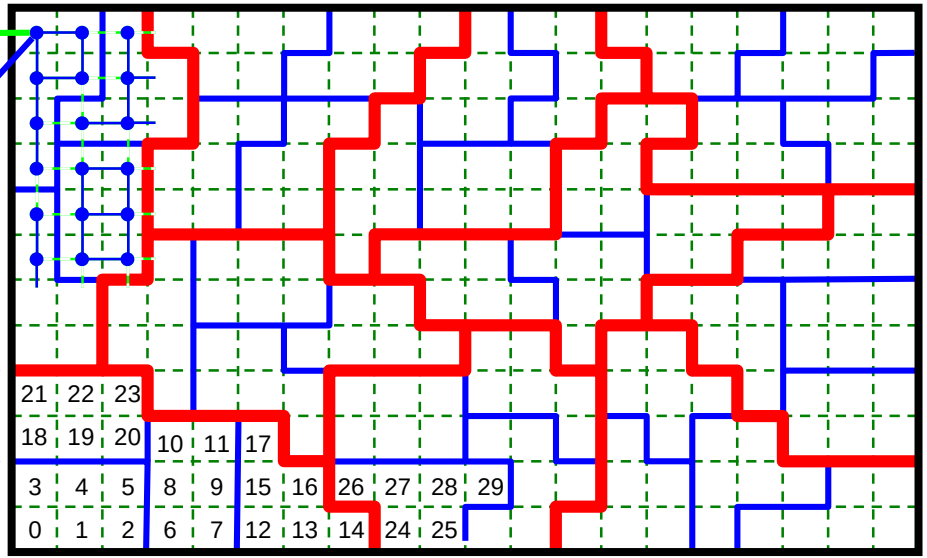
- Multi-level Domain Decomposition through Recombination

1. **Core-level DD:** partitioning of (large) application's data grid
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
   as done in MPI_COMM_WORLD (e.g., sequentially)

e.g., with Metis / Scotch

Graph of all sub-domains (core-sized)

Grouped 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!

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

General considerations
The topology problem
The topology problem: How-to / Virtual Toplogies
Rank renumbering for optimization
The Topology Problem: Unstructured Grids
Quiz
> **Scalability**
Advantages & disadvantages, conclusions

To overcome MPI scaling problems

- MPI has a few scaling problems with more than 10,000 MPI processes
  - MPI_Alltoall* is not scalable with longer messages
  - Irregular Collectives: MPI_....v, e.g. MPI_Gatherv
    - Scaling applications should not use MPI_....v routines
  - MPI Graph topology (MPI_Graph_create)
    - Use scalable interface MPI_Dist_graph_create_adjacent
  - Creation of many disjoint sub-communicators
    - Creation possible in a single call to MPI_Comm_split or MPI_Comm_create
  - MPI internal memory consumption for, e.g.,
    - Internal data structures 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)

Protocol switches are implementation dependent

Current implementations consider this

Programming models - pure MPI

Advantages & disadvantages, conclusions

General considerations
The topology problem
The topology problem: How-to / Virtual Toplogies
Rank renumbering for optimization
The Topology Problem: Unstructured Grids
Quiz
Scalability
> **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

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 or NUMA domains
- 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

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

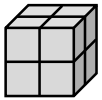
- Using OpenMP
  - → may prohibit compiler optimization
  - → **may cause significant loss of computational performance**
- Thread fork / join overhead
- On ccNUMA SMP nodes:
  - Loss of performance due to missing memory page locality or missing first touch strategy
  - E.g., with the MASTERONLY scheme:
    - One thread produces data
    - Master thread sends the data with MPI
    - → data may be internally communicated from one NUMA domain 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?

MPI+OpenMP versus MPI+MPI-3.0 shared memory

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

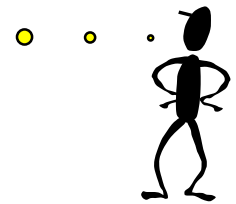
Conclusions

- 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** still grows)
- **MPI + OpenMP → workhorse on large systems**
  - Major pros: **reduced memory needs** and **second level of parallelism**
- **MPI + MPI shared memory → only for special cases and medium #processes**
- Pure MPI communication → still viable if it does the job
- OpenMP only → on large ccNUMA nodes (almost gone in HPC)

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** still grows)
- **MPI + OpenMP → workhorse on large systems**
  - Major pros: **reduced memory needs** and **second level of parallelism**
- **MPI + MPI shared memory → only for special cases and medium #processes**
- 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



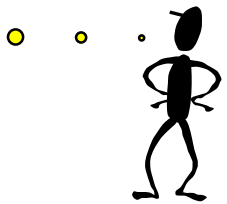
Conclusions

- 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** still grows)
- **MPI + OpenMP → workhorse on large systems**
  - Major pros: **reduced memory needs** and **second level of parallelism**
- **MPI + MPI shared memory → only for special cases and medium #processes**
- 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-Dec	https://vsc.ac.at/training/2022/HY-VSC-Dec	2022-HY-LRZ	http://www.hlrs.de/training/2022/HY-LRZ
2022-HY-VSC	http://vsc.ac.at/training/2022/HY-VSC		
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));
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 );

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);

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

if( comm_head != MPI_COMM_NULL ) {
   MPI_Bcast(arr, arrSize, arrDataType, 0, comm_head);
}

MPI_Win_fence(/*workaround:no assertions:*/0,win);

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

grey = original code

process is head of one of the shared memory islands

Starting write epoch

Filling arr **by process 0**

Only the heads of the shared memory islands fill arr by …

… broadcasting to all heads

instead of MPI_COMM_WORLD

Starting read epoch by all proc's

Reading arr **by all processes**

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)

During the exercise, you should reduce it to **1** time-step

Time step loop

```
...
/* ===> 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();
}
```

Filling the array by one process. Will be unchanged.

Steps (3)-(6) are done together

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.
…
```

Fortran

data-rep_sol_2d_30.f90

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

similar to C

→ See next slide

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;…
```

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?

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?

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**?

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?

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?

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?

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?

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?

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?

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?

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?

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

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?

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 . . synchronized without additional RMA calls."

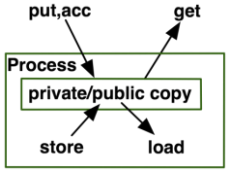


Figure: Courtesy of Torsten Hoefler

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)?

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?

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

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."

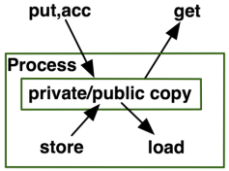


Figure: Courtesy of Torsten Hoefler

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)?

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?

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

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."

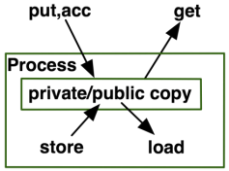


Figure: Courtesy of Torsten Hoefler

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)?

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?

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

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."

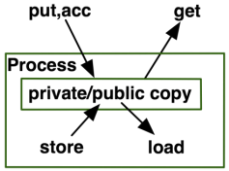


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)

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?

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

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."

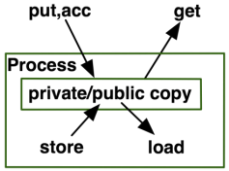


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   - **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**

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?

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

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."

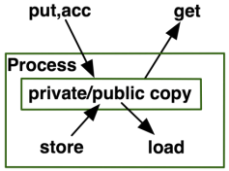


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   - **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**
   - **Any** (MPI) **synchronization** together with a **pair of C++11 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?

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

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."

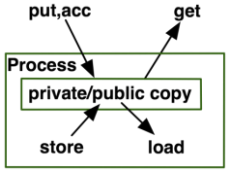


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   - **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**
   - **Any** (MPI) **synchronization** together with a **pair of C++11 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?
   - ***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.

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

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."

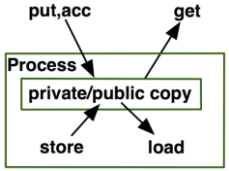


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   - **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**
   - **Any** (MPI) **synchronization** together with a **pair of C++11 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?
   - ***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.
   - ***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**?

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."

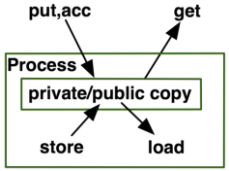


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   - **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**
   - **Any** (MPI) **synchronization** together with a **pair of C++11 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?
   - ***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.
   - ***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**?
   - Two processes access the **same shared variable** and at **least one process modifies** the variable **and the accesses are concurrent**.

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."

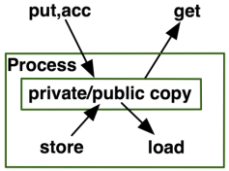


Figure: Courtesy of Torsten Hoefler

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)?
   - **Any MPI one-sided synchronization** (e.g., MPI_Win_fence, …_post/start, …, …_lock/unlock)
   - **Any** (MPI) **synchronization** together with a **pair of MPI_Win_sync**
   - **Any** (MPI) **synchronization** together with a **pair of C++11 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?
   - ***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.
   - ***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**?
   - Two processes access the **same shared variable** and at **least one process modifies** the variable **and the accesses are concurrent**.
   - 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?

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

Quiz on Virtual topologies

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

B. And for which use cases?

1. **Cartesian topologies**

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

Quiz on Virtual topologies

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

B. And for which use cases?

1. **Cartesian topologies**

2. **Distributed graph topologies and graph topologies**

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

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**

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

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?

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;**

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.