Hybrid Programming in HPC – MPI+X

Claudia Blaas-Schenner[1)]
claudia.blaas-schenner@tuwien.ac.at

Tobias Haas[2)]
tobias.haas@hlrs.de

Georg Hager[3)]
georg.hager@fau.de

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

ON-SITE & ONLINE COURSE @ HLRS Stuttgart, February 10-12, 2026

http://tiny.cc/MPIX-HLRS

General outline

Introduction

Programming Models and Optimizations

- MPI + MPI-3 shared memory
- MPI + OpenMP on multi/many-core
- MPI + Accelerators

Conclusions

Appendix

Introduction

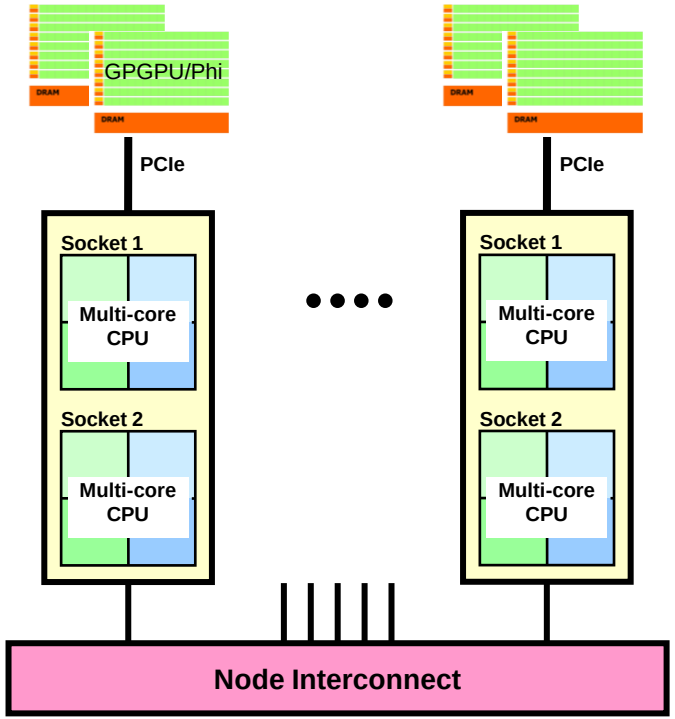
Hardware and programming models

Hardware Bottlenecks

Questions addressed in this tutorial

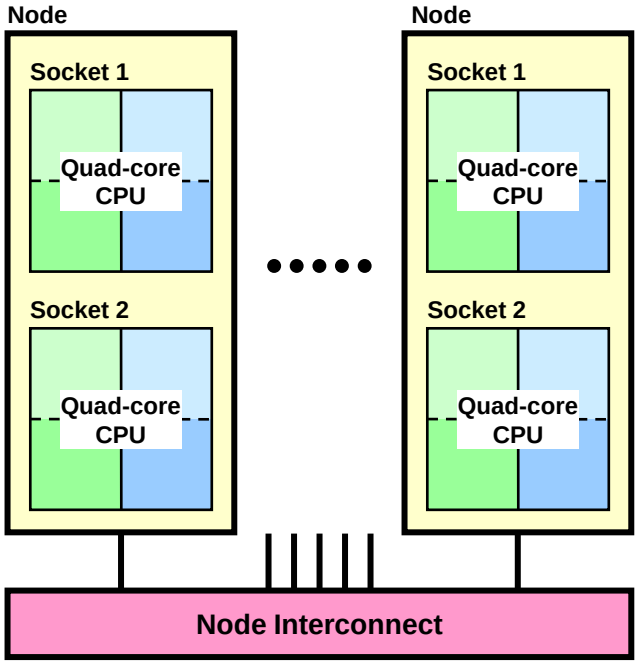
Remarks on Cost-Benefit Calculation

Hardware and programming models

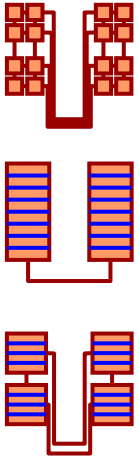


- MPI + threading
  - OpenMP
  - Cilk(+)
  - TBB (Threading Building Blocks)
- MPI + MPI shared memory
- MPI + accelerator
  - OpenMP offloading
  - OpenACC
  - CUDA
  - OpenCL, Kokkos, SYCL,…
- Pure MPI communication
  - Optimized node-to-node communication

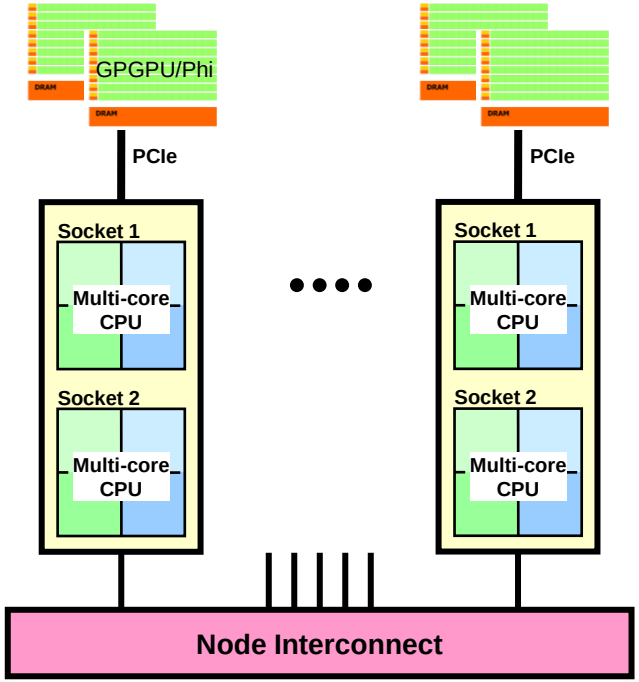
Options for running code on multicore clusters



- 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

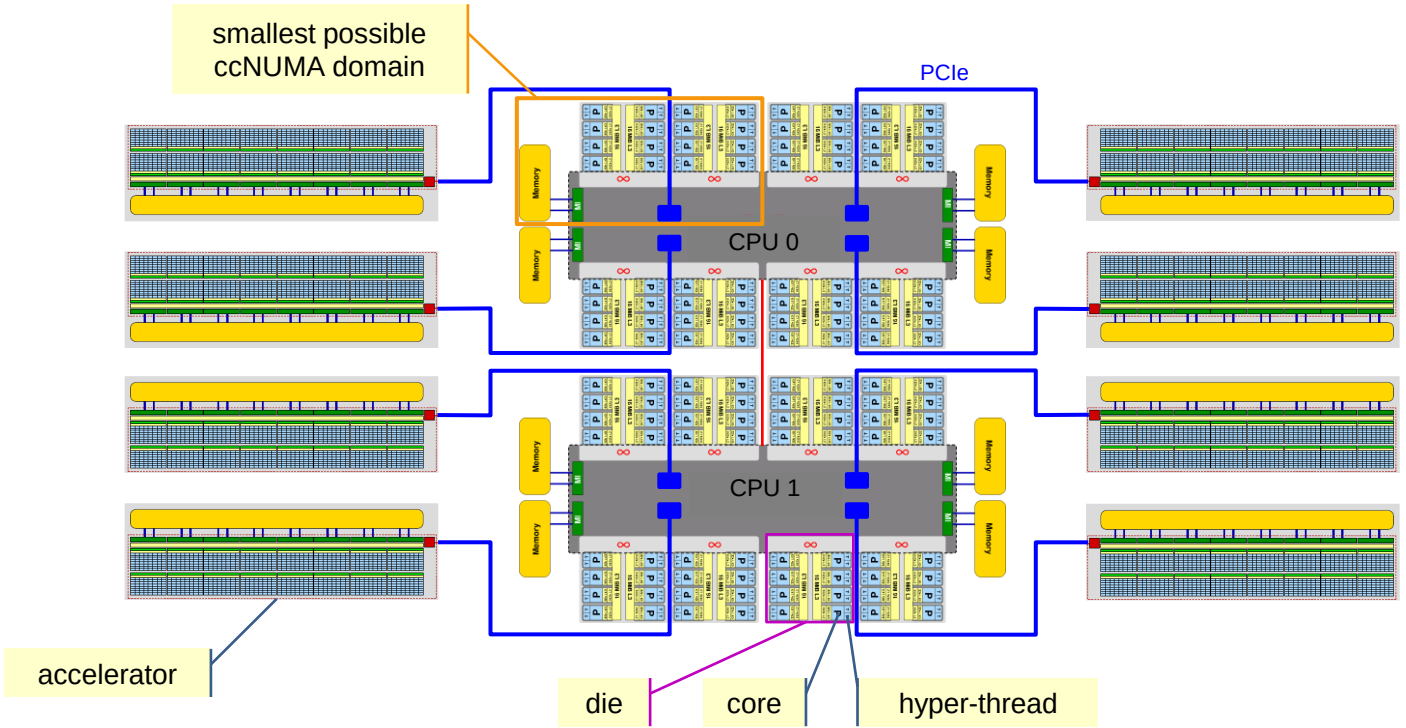
Hierarchical parallel programming

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

Where is the bottleneck?

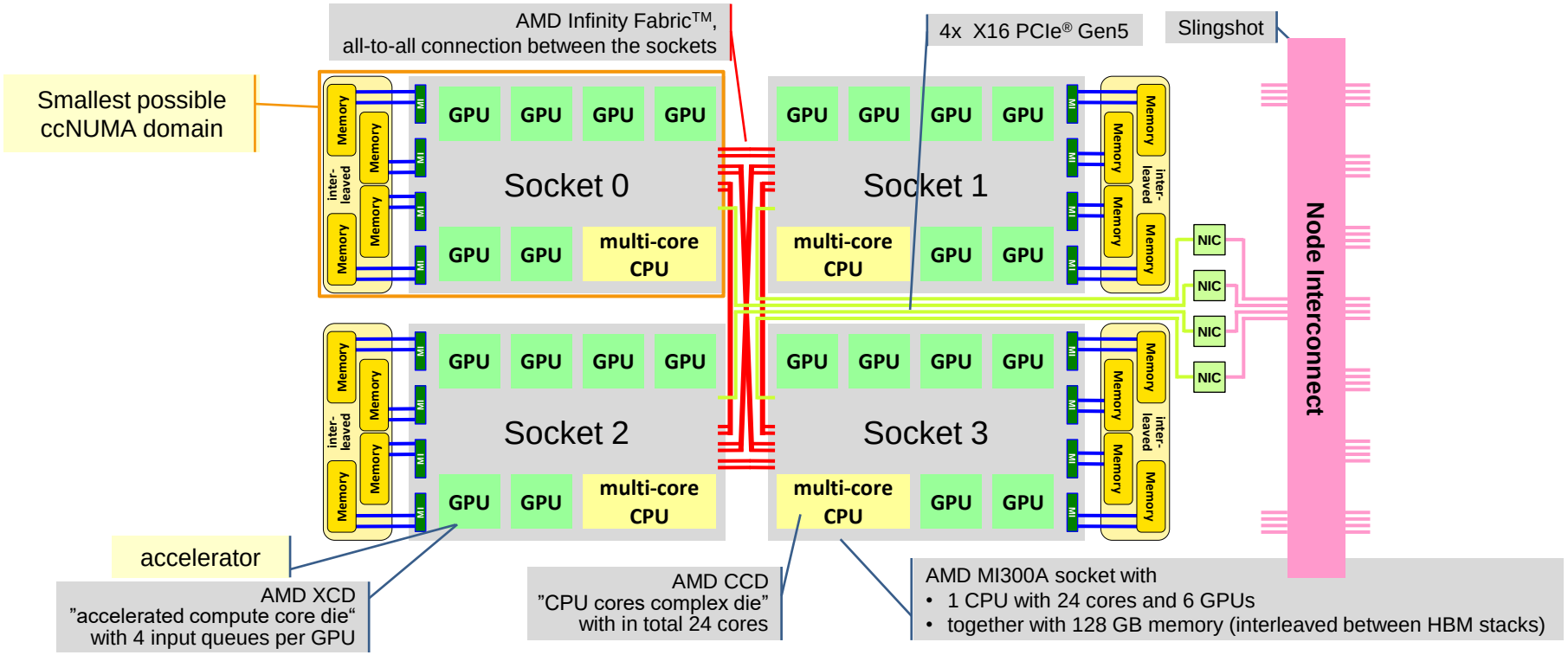
Dual-CPU ccNUMA + accelerator node architecture

Modern compute node with **separate memories** for CPUs and GPUs



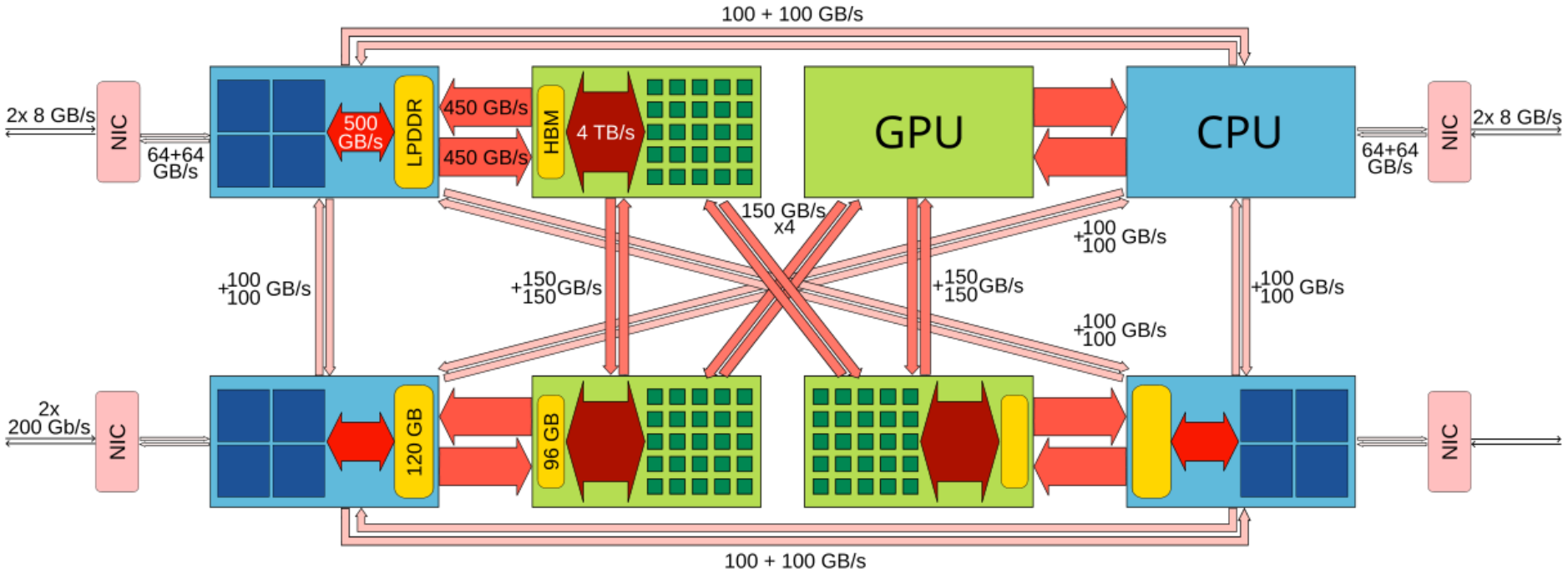
Accelerated node architecture with AMD MI300A APUs

with **common shared memory** for CPUs and GPUs (HLRS “Hunter”)



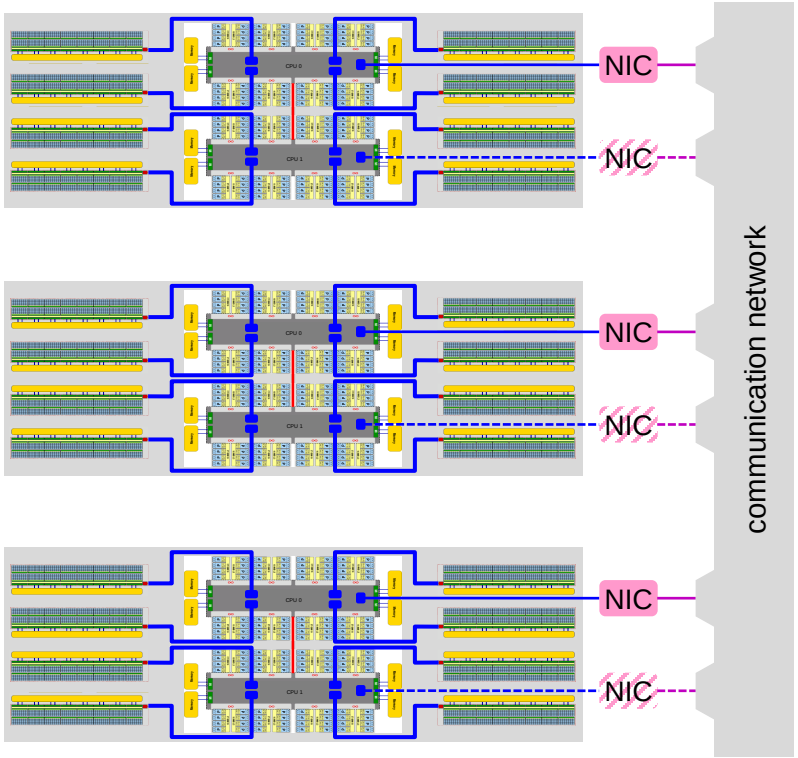
Accelerated node architecture with NVIDIA Grace-Hopper CPU/GPUs

with **common shared memory** for CPUs and GPUs (JSC JUPITER Booster)



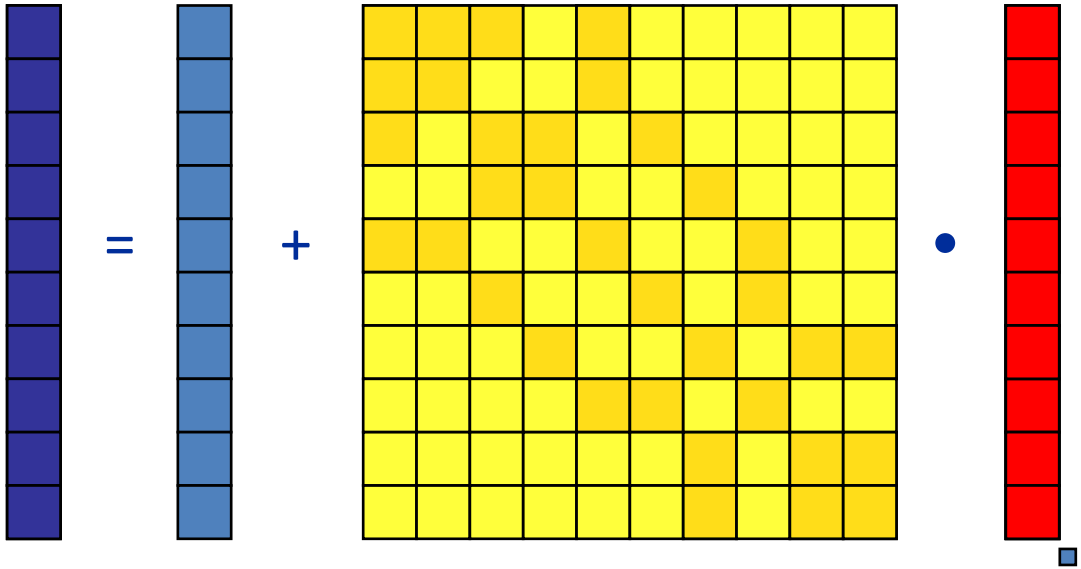
Hardware bottlenecks

- 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 performance 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?
- 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?

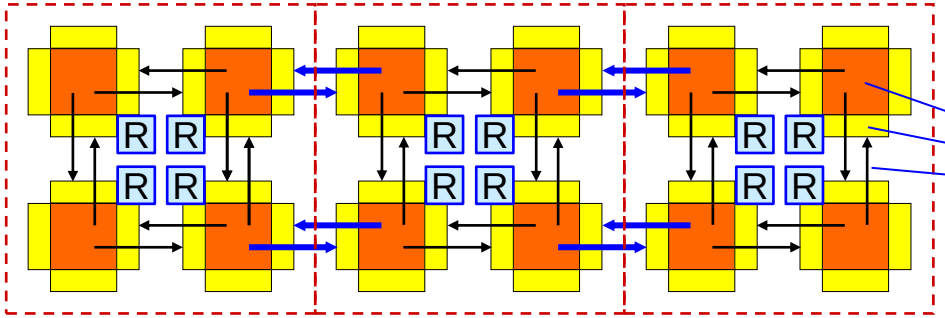
Programming models and optimizations

- **MPI + MPI-3.0 shared memory**
- **MPI + OpenMP on multi/many-core**
- **MPI + Accelerators**

Programming models
- MPI + MPI-3 shared memory

General considerations & uses cases
Re-cap: MPI_Comm_split & one-sided communication
How-to
Advantages & disadvantages, conclusions

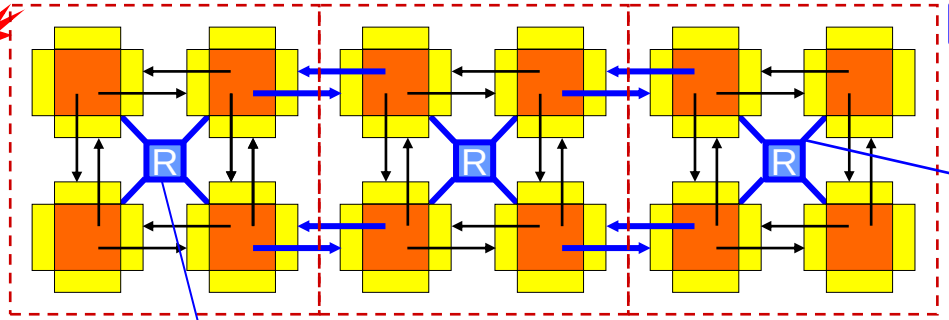
Use case A: Reducing memory requirements



R = Replicated data in each MPI process

Subdomain
halo
halo-communication

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



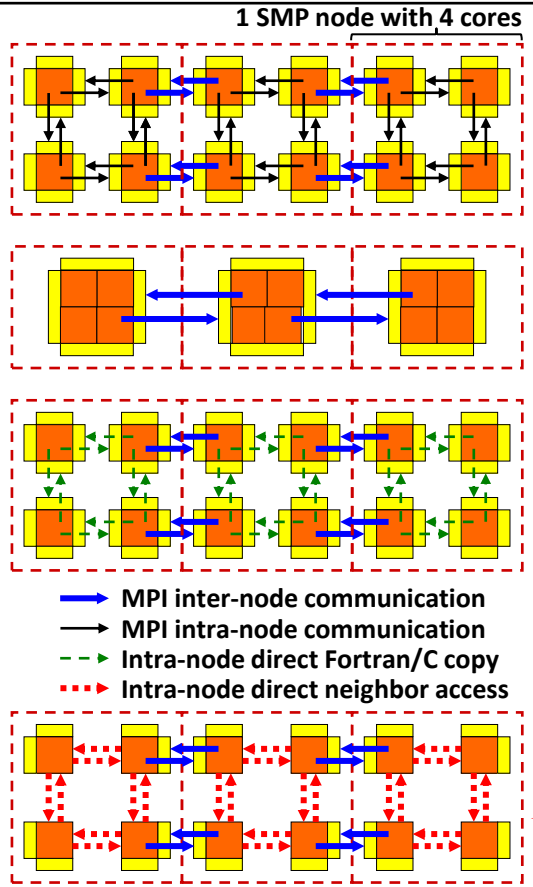
R = Shared memory
→ replicated data only once within each SMP node

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/Fortran/Python 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

Hybrid MPI + MPI-3 shared memory

- Further advantages
  - Using only one parallel programming model
  - Fewer 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

Programming models
- MPI + MPI-3.0 shared memory

Re-cap

- MPI_Comm_split
- One-sided communication

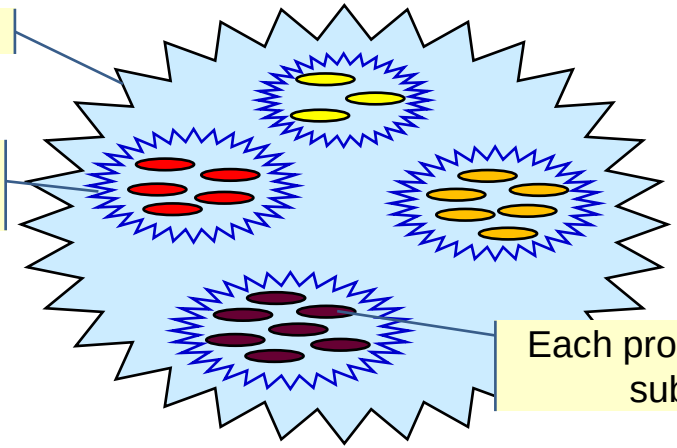
General considerations & uses cases
> **Re-cap: MPI_Comm_split & one-sided communication**
How-to
Advantages & disadvantages, conclusions

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

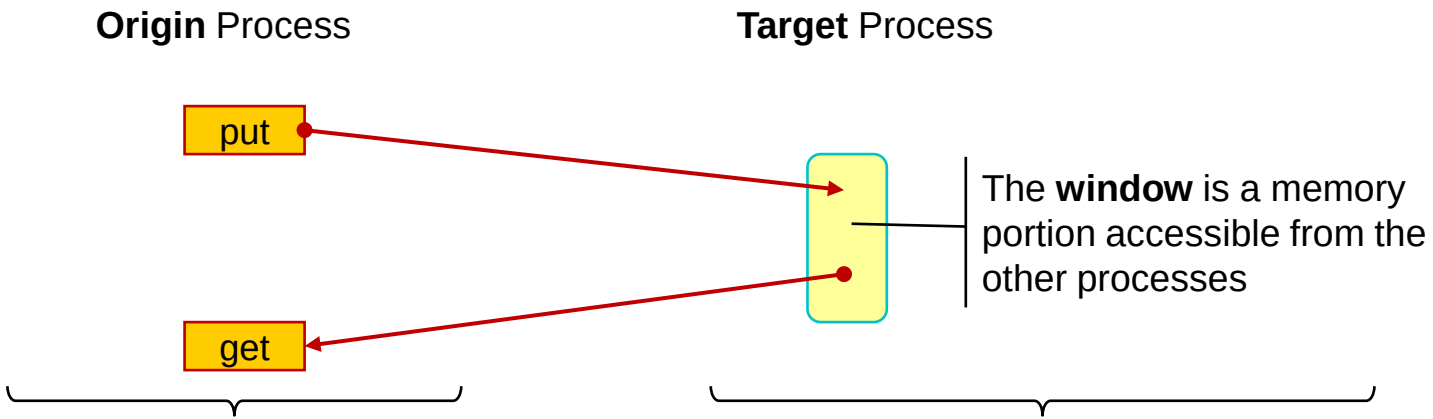
& MPI_Comm_split_type
→ shared memory

New in MPI-3.0

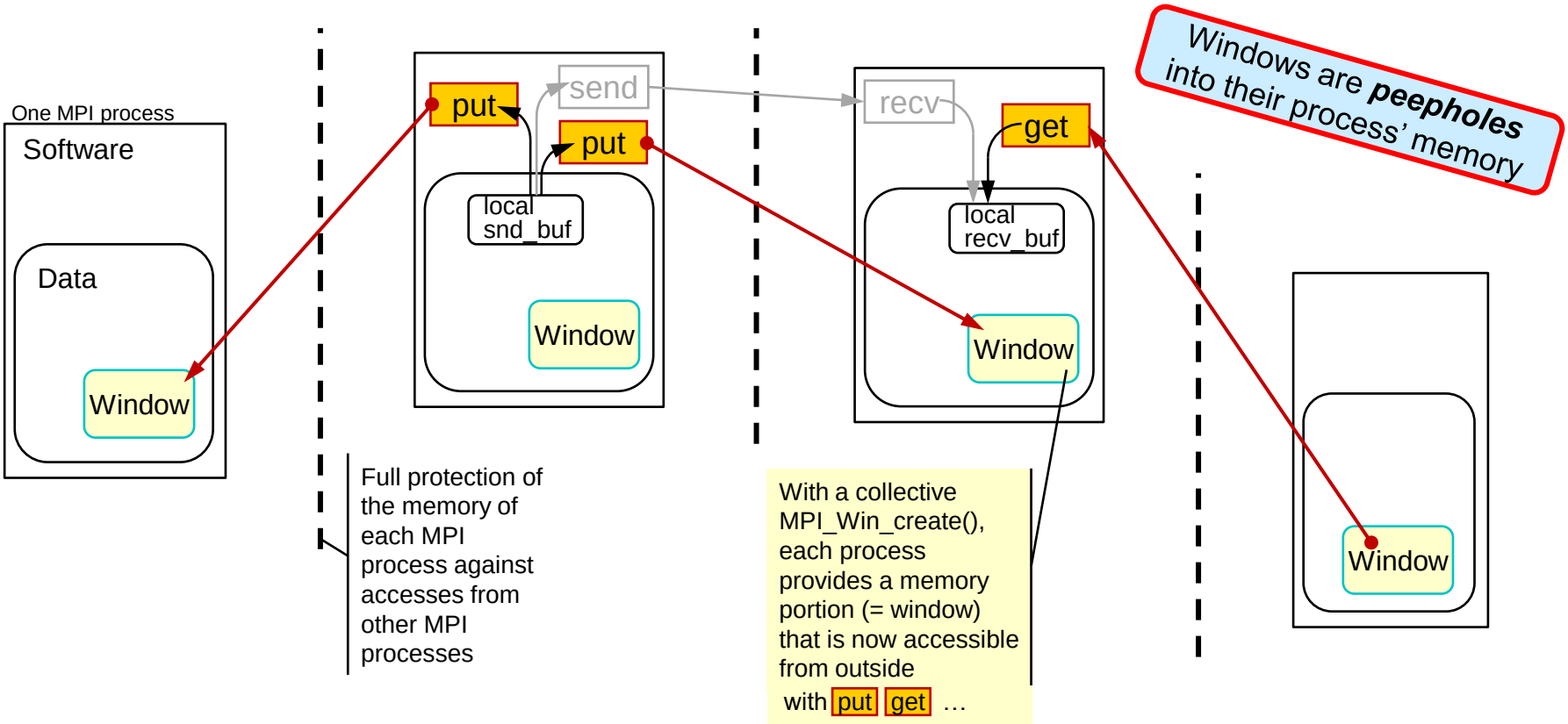


Re-cap: One-sided Communication

- 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



Programming models
- MPI + MPI-3.0 shared memory

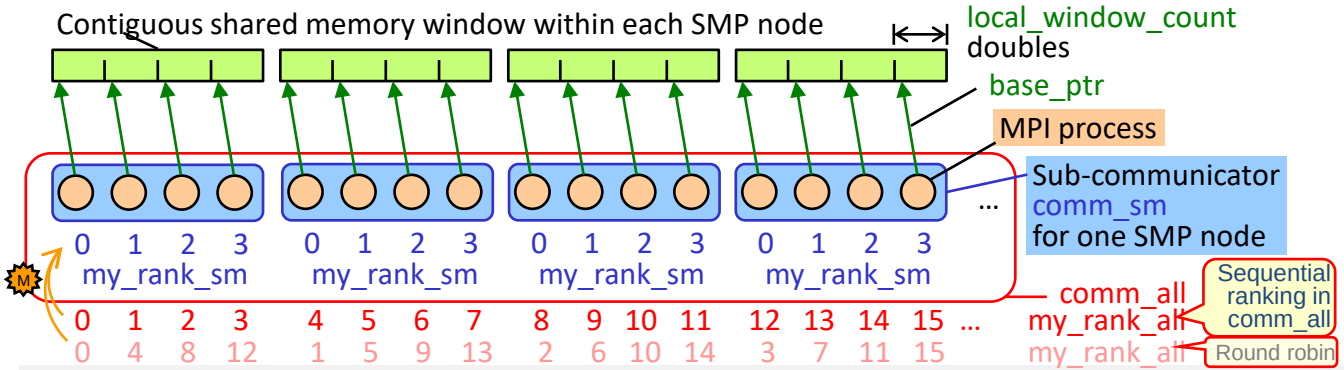
How-to

General considerations & uses cases
Re-cap: MPI_Comm_split & one-sided communication
> How-to
Advantages & disadvantages, conclusions

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_Aint /*IN*/ local_window_count=10;  double /*OUT*/  *base_ptr;
MPI_Comm  comm_all,  comm_sm;       int  my_rank_all,  my_rank_sm,  size_sm,  disp_unit;
MPI_Comm_rank (comm_all, &my_rank_all);
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0,
            MPI_INFO_NULL,  &comm_sm);
MPI_Comm_rank (comm_sm, &my_rank_sm);  MPI_Comm_size (comm_sm, &size_sm);
disp_unit = sizeof(double);  /* shared memory should contain doubles */
MPI_Win_allocate_shared ((MPI_Aint) local_window_count*disp_unit,  disp_unit,
            MPI_INFO_NULL,  comm_sm,  &base_ptr,  &win_sm);
```

collective call

collective call

Sequence in comm_sm as in comm_all

Window size in bytes

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

Ⓜ 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

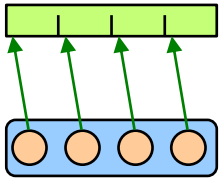
max_length = …

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=I6pEaUttWo8
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, …)

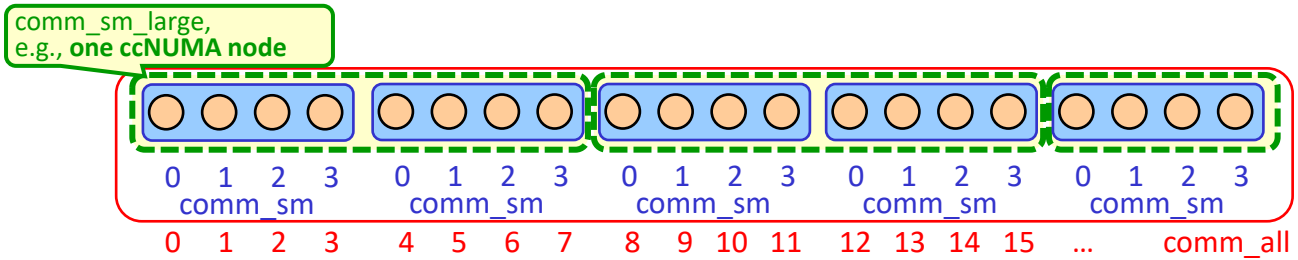
- Caution:
  Although each process in comm_sm accesses the same physical memory, the virtual start address of the whole array may be different in all processes!

  → **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
  - MPI_COMM_TYPE_RESOURCE_GUIDED and MPI_Get_hw_resource_info(&hw_info)

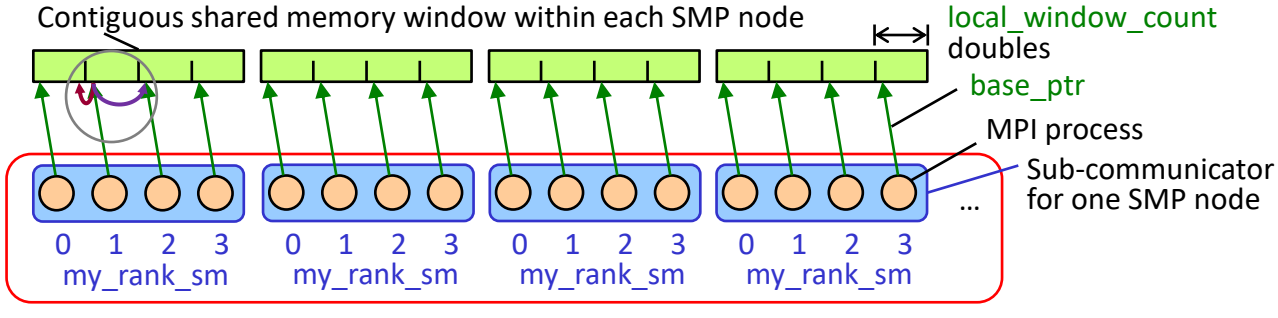
New in MPI-4.0

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

New in MPI-4.1

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 completed, 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] );
```

barrier-like synchronization

barrier-like synchronization

Local stores

Direct load access to remote window portion

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

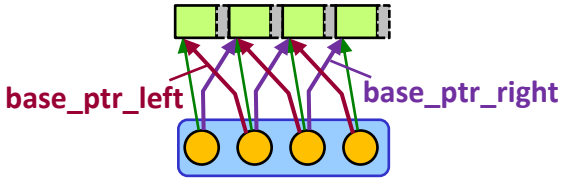
In Fortran, before and after the synchronization, on must add: CALL MPI_F_SYNC_REG (buffer) to guarantee that register copies of buffer are written back to memory, respectively read again from memory. The buffer should be declared as ASYNCHRONOUS, see course Chapter 10, slide "Fortran Problems with 1-Sided".

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**)

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

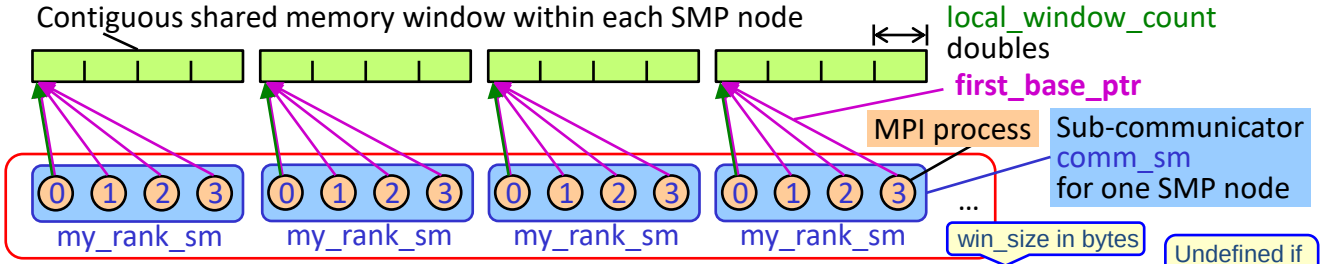


local call

```
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 completed, 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 ] );
```

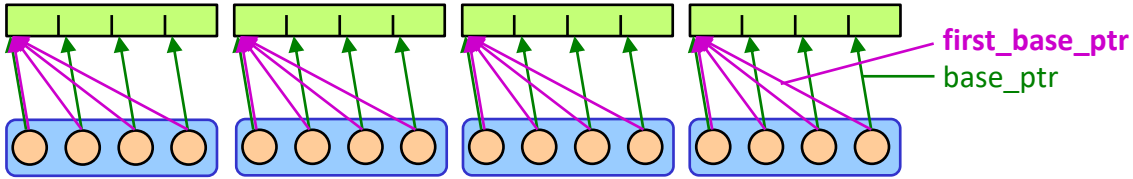

Thanks to Steffen Weise (TU Freiberg) for testing and correcting the example codes.

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




```
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.

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
> Advantages & disadvantages, conclusions

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
- 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)
- MPI shared memory designed for use in CPU code, not GPUs

Further reading on shared memory synchronization
- Wikipedia: Memory barrier. https://en.wikipedia.org/wiki/Memory_barrier
- Wikipedia: Runtime memory ordering https://en.wikipedia.org/wiki/Memory_ordering#Runtime_memory_ordering

(and courtesy to Dave Goodell):
- Paul E. McKenney (ed.). Is Parallel Programming Hard, And, If So, What Can You Do About It? First Edition, Linux Technology Center, IBM Beaverton, March 10, 2014. https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.pdf

On compiler optimization problems (courtesy to Bill Gropp):
- Hans-J. Boehm. Threads Cannot be Implemented as a Library. HP Laboratories Palo Alto, report HPL-2004-2092004, 2004. https://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf
- Sarita V. Adve, Hans-J. Boehm: You don't know Jack About Shared Variables or Memory Models. https://queue.acm.org/detail.cfm?id=2088916

Programming models – MPI + OpenMP

- 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: Simple 2D stencil smoother
  - Case study: The Multi-Zone NAS Parallel Benchmarks (skipped)
  - 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

General considerations

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);
```

- 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

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 */
```

- MPI calls within omp single
  → **`MPI_THREAD_SERIALIZED`** is required
- 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

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

```
mpiicx          -DUSE_MPI -qopenmp      ...
icx                       -qopenmp      ...
```

- Intel MPI + Intel Fortran

```
mpiifx    -fpp -DUSE_MPI -qopenmp      ...
ifx       -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), one process (12 threads) per socket
  - `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, one process (36 threads) per socket
  - `mpiifx -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

- Intel Ice Lake (36-core 2-socket) cluster, Intel MPI/OpenMP + likwid-mpirun, one process (36 threads) per socket
  - `mpiifx -qopenmp ...`
  - `likwid-mpirun -np 4 -pin S0:0-35_S1:0-35 ./a.out`

- Intel Skylake (24-core 2-socket) cluster, GCC + OpenMPI 4.1, one process (24 threads) per socket
  - `mpif90 -fopenmp ...`
  - ```
    OMP_NUM_THREADS=24 OMP_PLACES=cores OMP_PROC_BIND=close \
      mpirun --map-by ppr:1:socket:PE=24  ./a.out
    ```
  - Dito, two processes per socket (12 threads each)

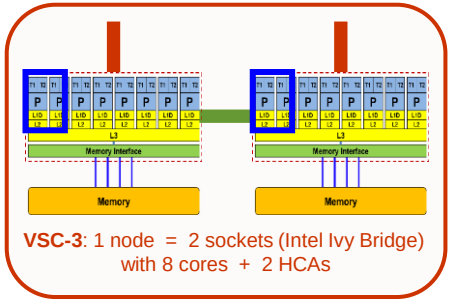
    ```
    OMP_NUM_THREADS=12 OMP_PLACES=cores OMP_PROC_BIND=close \
      mpirun --map-by ppr:2:socket:PE=12  ./a.out
    ```
    ```

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  | | 7 23  | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |  32kB | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 256kB | | 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

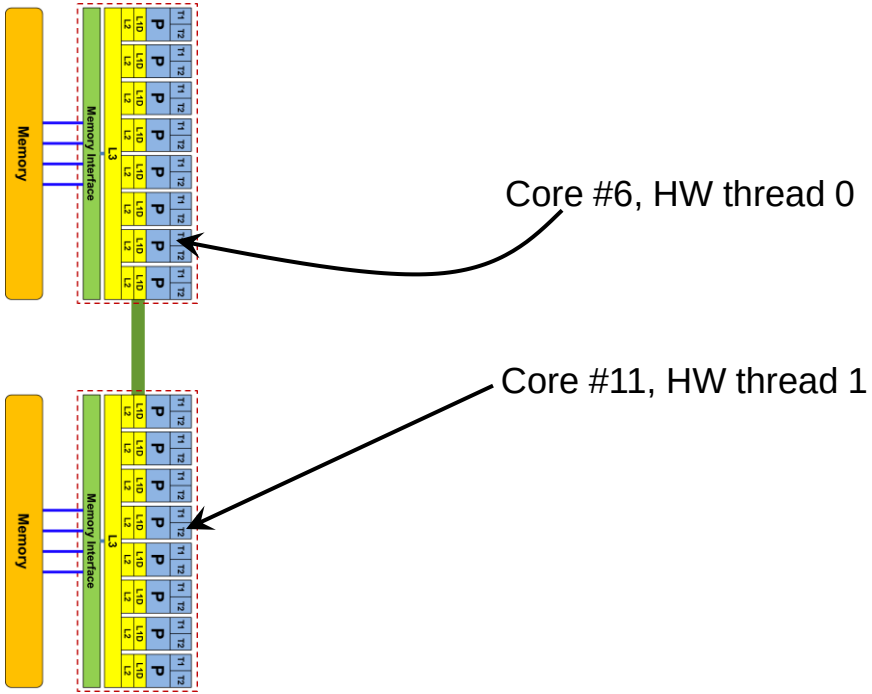
Hello hybrid!

Programming models - MPI + OpenMP

System topology, ccNUMA, and memory bandwidth

What is “topology”?

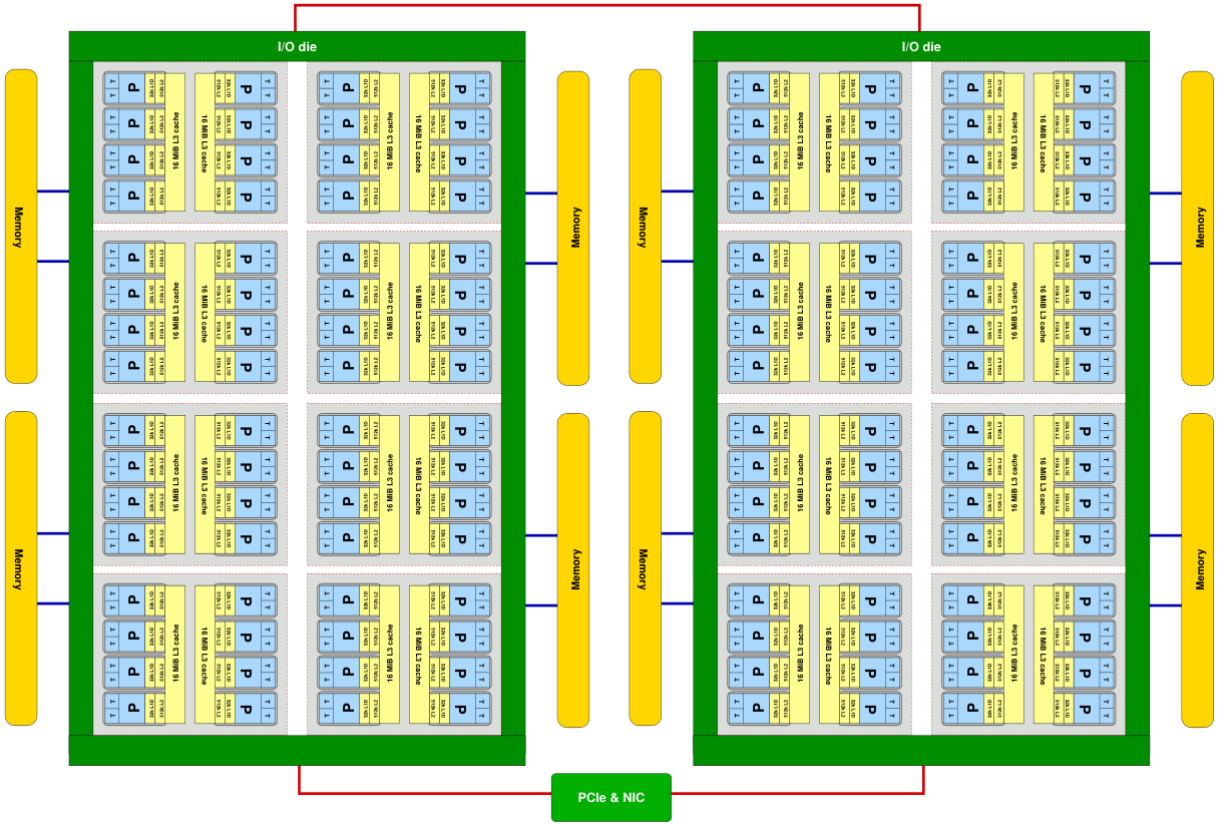
Where in the machine does core (or hardware thread) #n reside?



Why is this important?

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

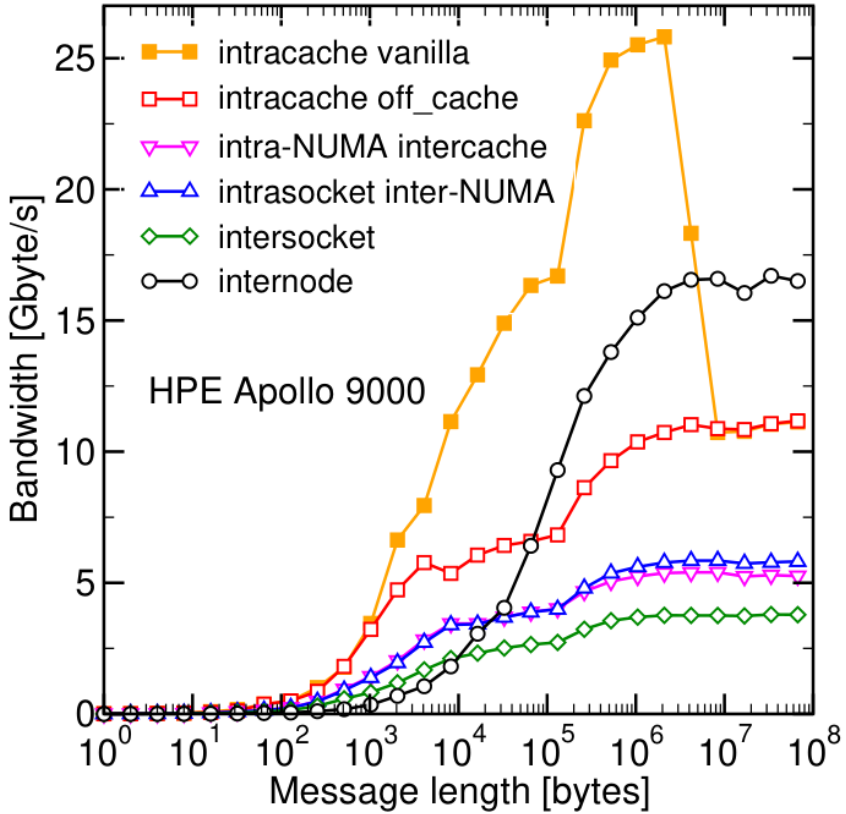
HLRS Hawk node layout



MPI communication: Ping-Pong on Hawk

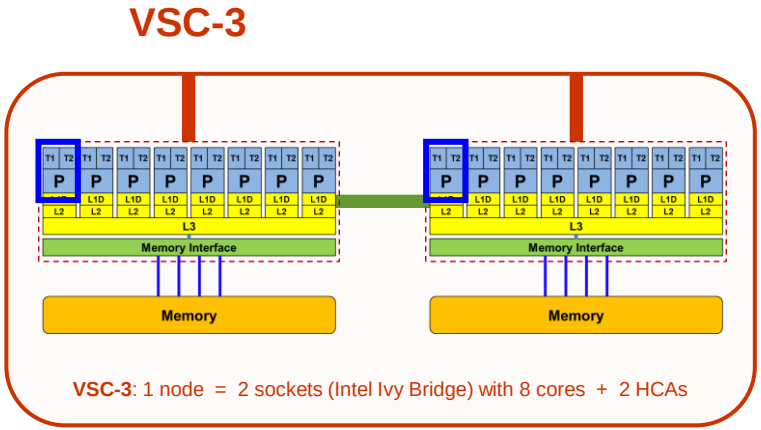
```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
  E = get_walltime()
  GBYTES = 2*N/(E-S)/1.d9 ! Gbyte/s rate
  TIME = (E-S)/2*1.d6     ! transfer time
else
  targetID = 0
  call Receive_message(buffer,N,targetID)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

	latency [μs]
intracache	0.14
intra-NUMA	0.36
inter-NUMA	0.41
intersocket	0.60
internode	1.7



Compute nodes – data access in 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)

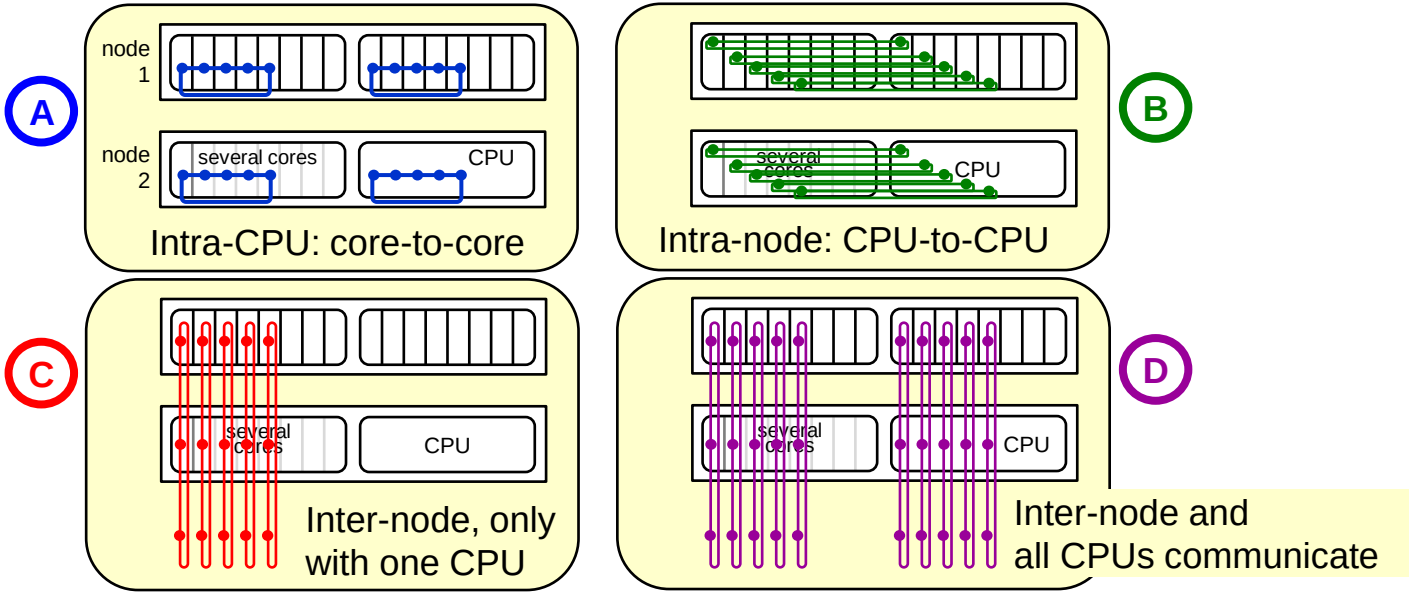


Benchmark: multiple communicating rings

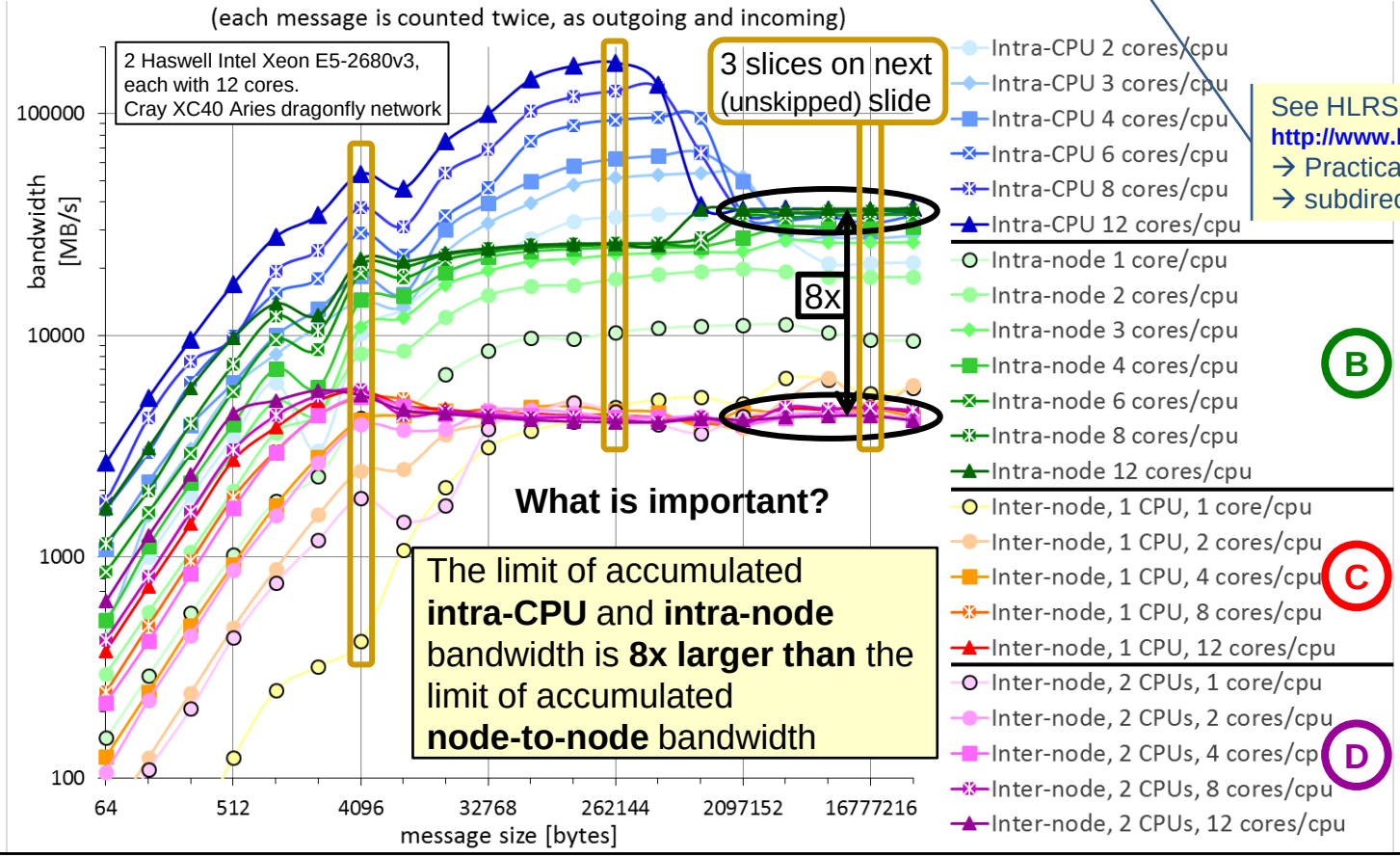
Benchmark halo_irecv_send_multiplelinks_toggle.c

- 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

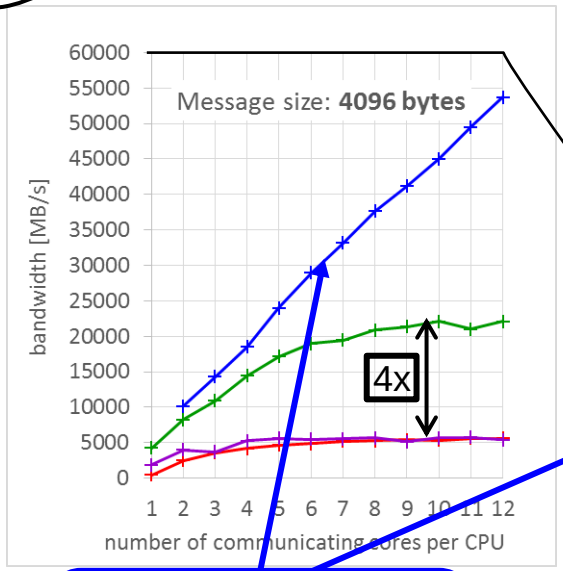


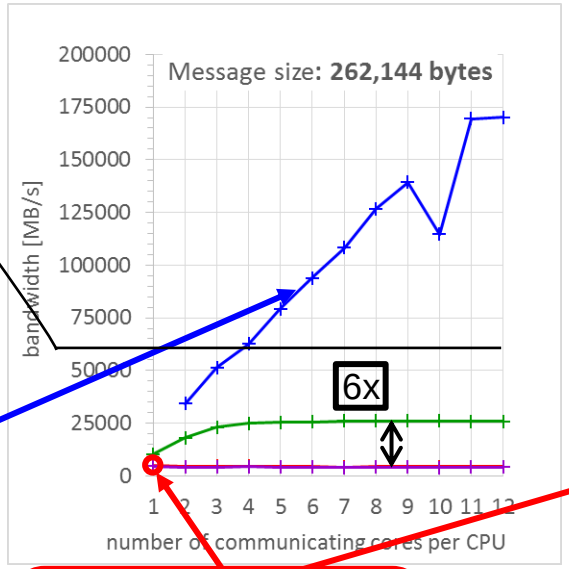
See HLRS online courses
http://www.hlrs.de/training/self-study-materials
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

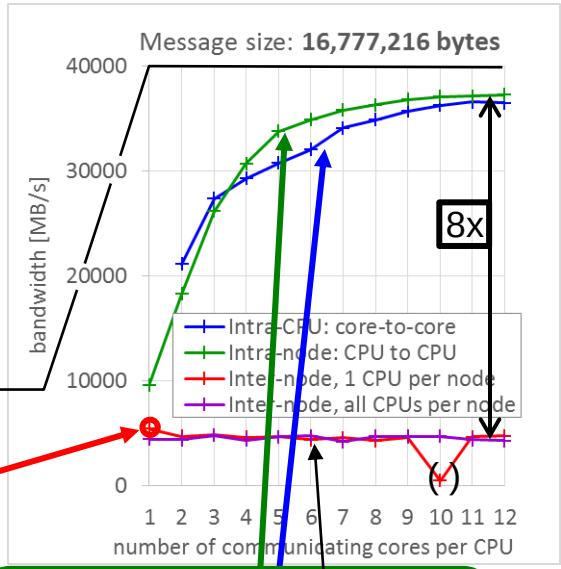
Measurement with halo_irecv_send_multiplelinks_toggle.c on 4 nodes of Cray XC40 hazelhen.hww.de, June 15, 2018, HLRS, by Rolf Rabenseifner (protocol 10)

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**

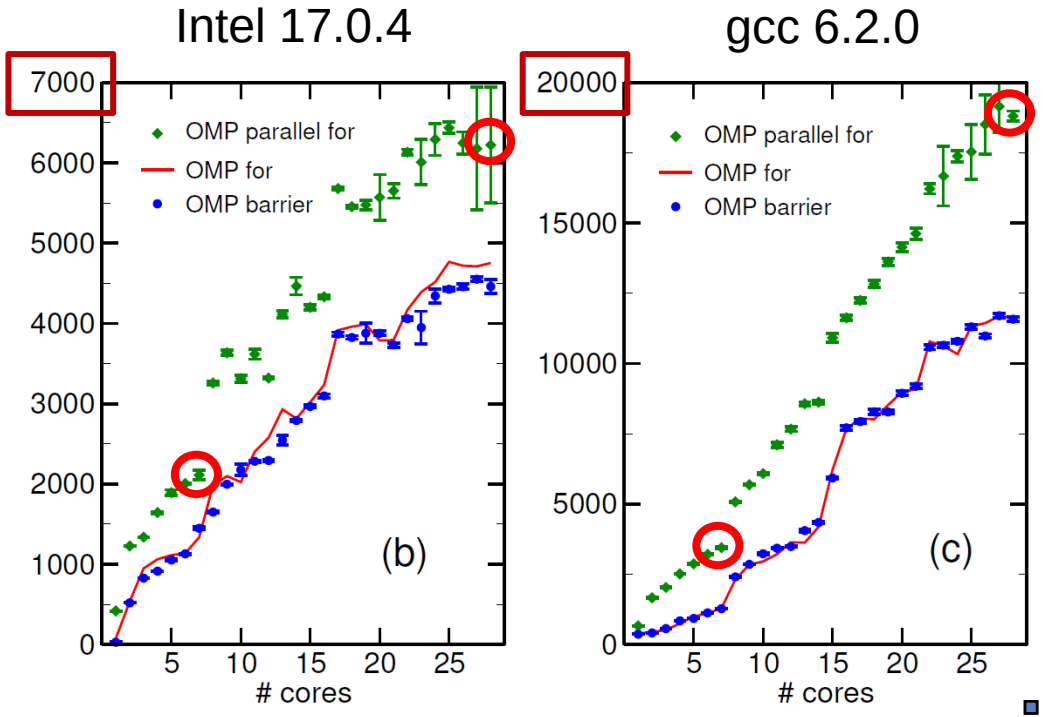
Result: The limit of accumulated **intra-CPU** and **intra-node** bandwidth is **8x larger than** the limit of accumulated **node-to-node** bandwidth

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

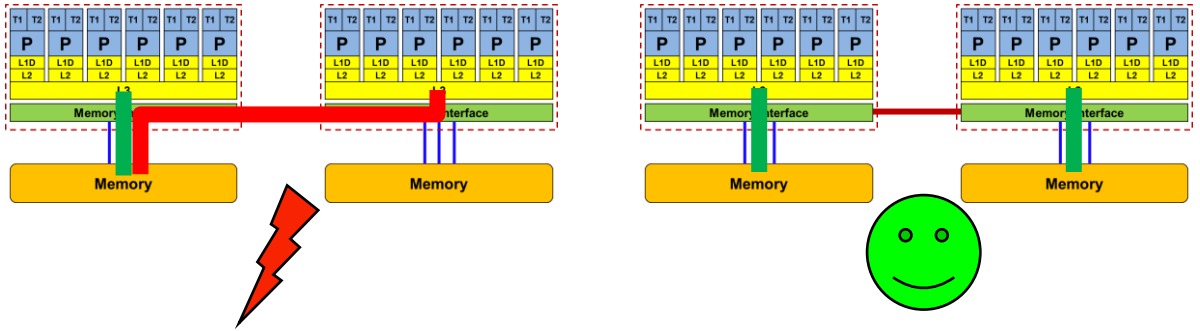


Programming models - MPI + OpenMP

Memory placement on ccNUMA systems

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)



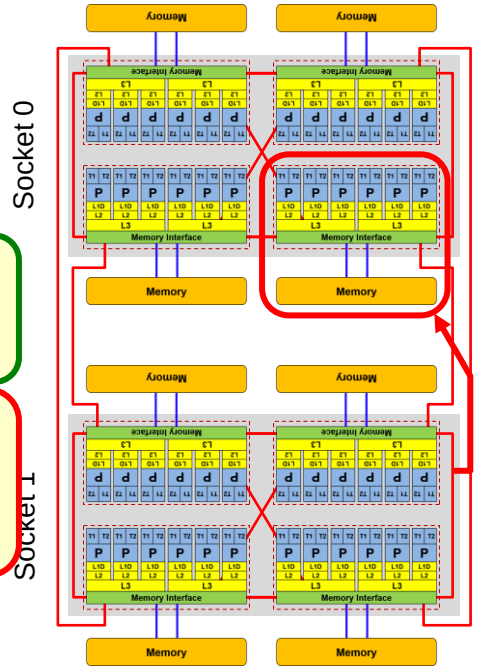
How much bandwidth does non-local access cost?

- 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	10.6	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”
- 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

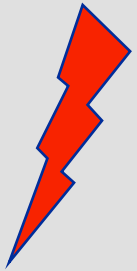
```
integer,parameter :: N=100000000
double precision A(N), B(N)



A=0.d0



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



```
integer,parameter :: N=100000000
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
- 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
- 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!

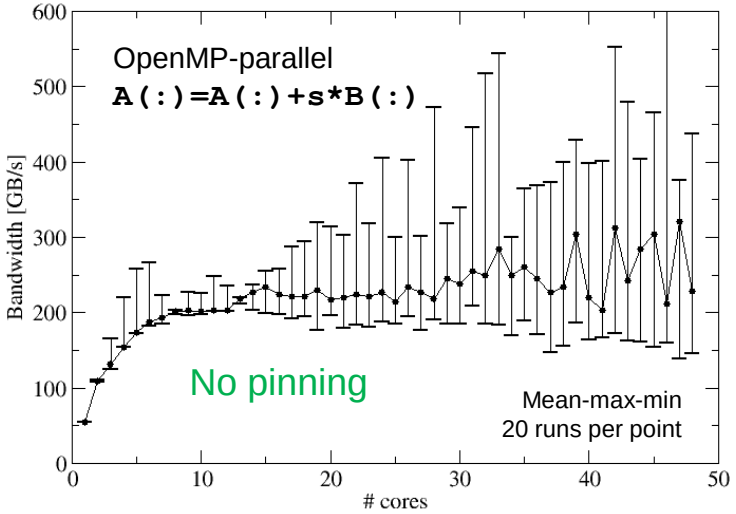
Programming models - MPI + OpenMP

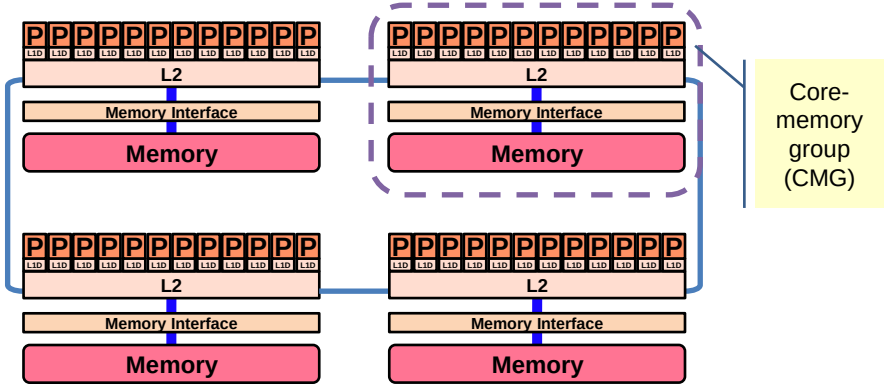
Topology and affinity on multicore

Thread/Process Affinity (“Pinning”)

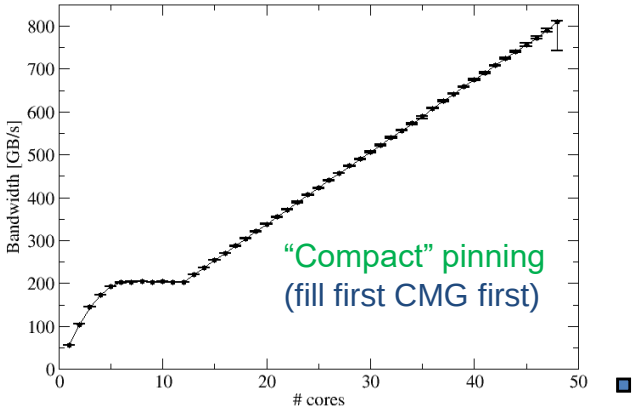
- Highly OS-dependent system calls
  - But available on all OSs
  - Non-portable
- Support for user-defined pinning for OpenMP threads in all compilers
  - Compiler specific
  - Standardized in OpenMP (places)
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`**
- Affinity awareness in all MPI libraries
  - Not defined by the MPI standard (as of 5.0)
  - 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



There are several reasons for caring about affinity:

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

skipped

likwid-pin

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

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

- 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

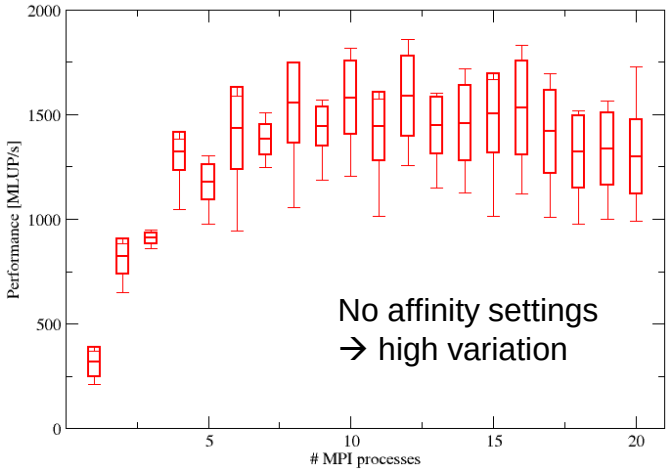
```
OMP_NUM_THREADS=8
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

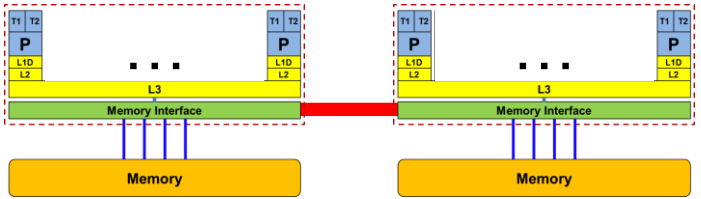
Always prefer abstract places instead of HW thread IDs!

Pinning of MPI processes

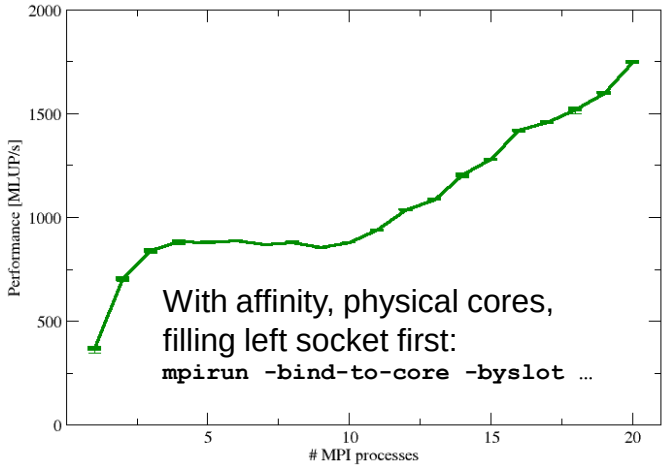
- Highly system dependent!
- Intel MPI: env variable **`I_MPI_PIN_DOMAIN`**
- OpenMPI: choose between several mpirun options, e.g.,
  -bind-to-core, -bind-to-socket, -bycore, -byslot …
- Cray's aprun: pinning by default
- Platform-independent tools: likwid-mpirun
  (likwid-pin, numactl)

Anarchy vs. affinity with a heat equation solver





2x 10-core Intel Ivy Bridge, OpenMPI

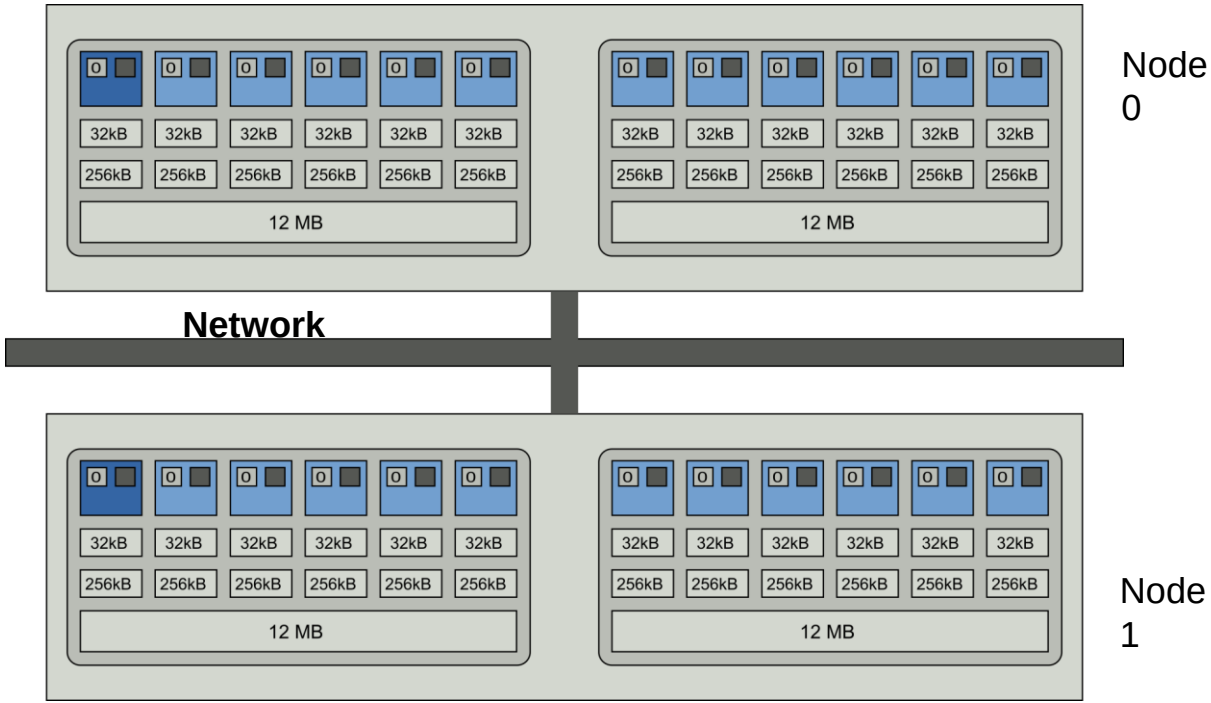


Reasons for caring about affinity:

- 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
```

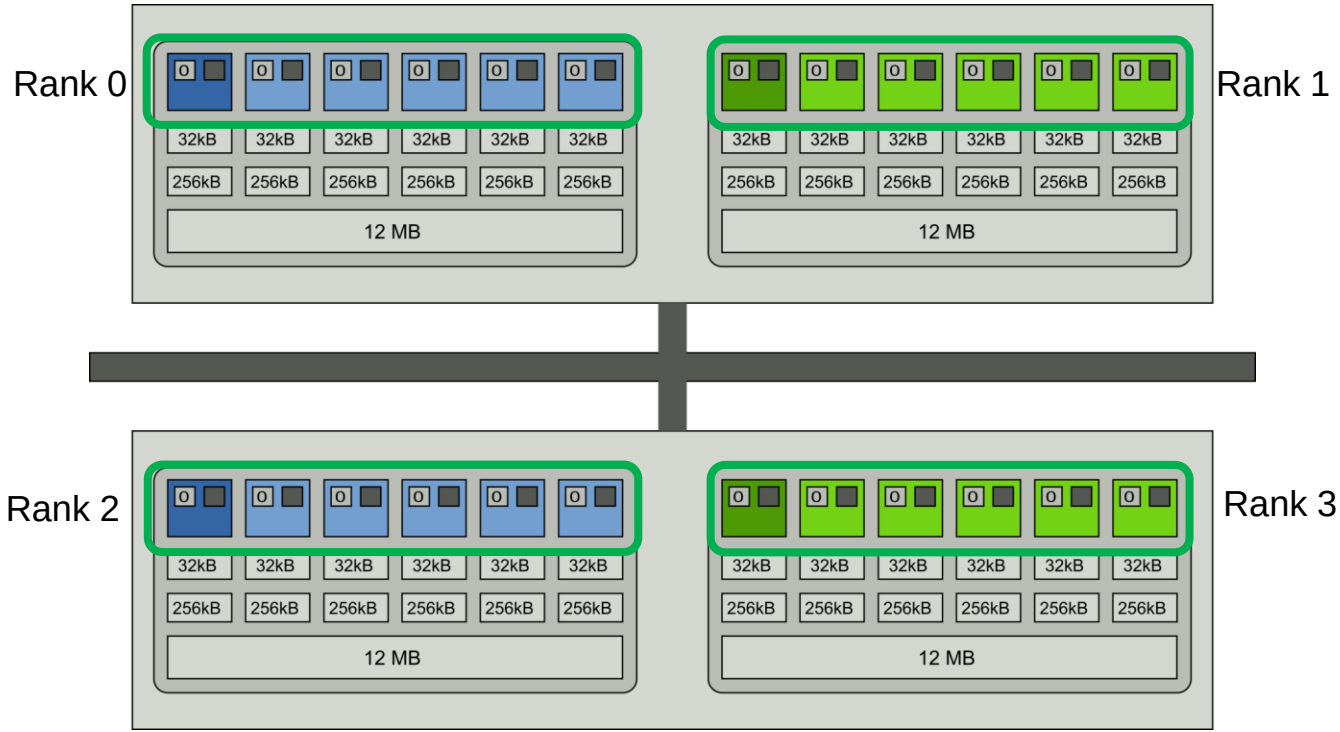


Intel MPI+compiler:

```
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)
- 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

http://tiny.cc/MPIX-HLRS

Programming models - MPI + OpenMP

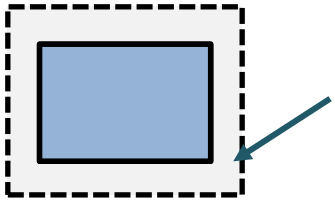
Case study: Simple 2D stencil smoother

Stencil smoother with ghost cell exchange

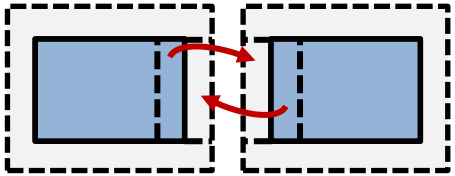
2D domain distributed to ranks (here 4 x 3), each rank gets one tile

rank 0	rank 1	rank 2	rank 3
rank 4	Rank 5	rank 6	rank 7
rank 8	rank 9	rank 10	rank 11

Each rank's tile is surrounded by ghost cells, representing the cells of the neighbors



After each sweep over a tile, perform ghost cell exchange, i.e., update ghost cells with new values of neighbor cells

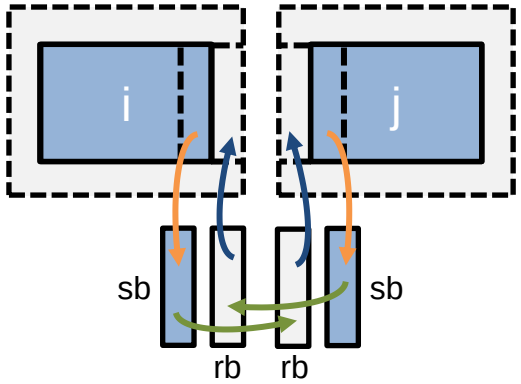


Possible implementation:

1. copy new data into contiguous send buffer (possibly optional)
2. send to corresponding neighbor, receive new data from same neighbor
3. copy received new data into ghost cells

```
MPI_Sendrecv(
sb, …, j,
rb, …, j, …)
```

step 2

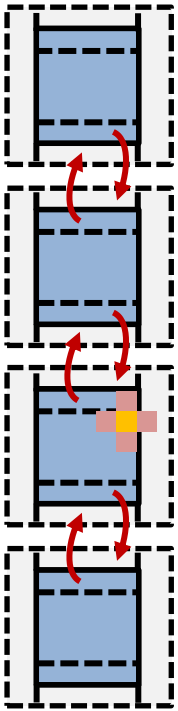



```
MPI_Sendrecv(
sb, …, i,
rb, …, i, …)
```

step 2

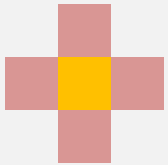
Jacobi 2D – 1D decomposition

Simple benchmark: 1D decomposition of grid along outer dimension



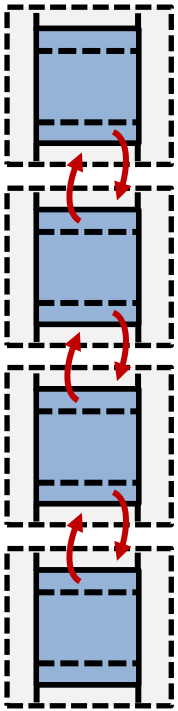
```
for (int iter = 0; iter < n_iterations; ++iter) {
  // ghost cell xchg
  exchange(domain, src_grid);
  // domain update
  relax(domain, src_grid, dst_grid);
  swap(src_grid, dst_grid);
}
```

```
void relax(...) {
...
  #pragma omp parallel for
  for (int y = start_y; y < end_y; ++y)
    for (int x = start_x; x < end_x; ++x)
      dst[y][x] = 0.25 * (src[y][x-1] + src[y][x+1]
                        + src[y-1][x] + src[y+1][x]);
}
```



Jacobi 2D – 1D decomposition

Simple benchmark: 1D decomposition of grid along outer dimension



```
void exchange(...) {
...
  // top neighbor xchg
  if (domain->comm_rank + 1 < domain->comm_size) {
    int top = domain->comm_rank + 1;
    MPI_Isend(&src[dim_y-1][0], dim_x,...,  &requests[0]);
    MPI_Irecv(grid->ghost_cells_top, dim_x, &requests[1]);
  }
  // bottom neighbor xchg
  if (domain->comm_rank > 0) {
    int bottom = domain->comm_rank - 1;
    MPI_Isend(&src[0][0], dim_x,...,            &requests[2]);
    MPI_Irecv(grid->ghost_cells_bottom,  dim_x, &requests[3]);
  }
  MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
}
```

No buffer copying necessary (halos are contiguous in memory)

Jacobi 2D – Benchmarking

Benchmark case

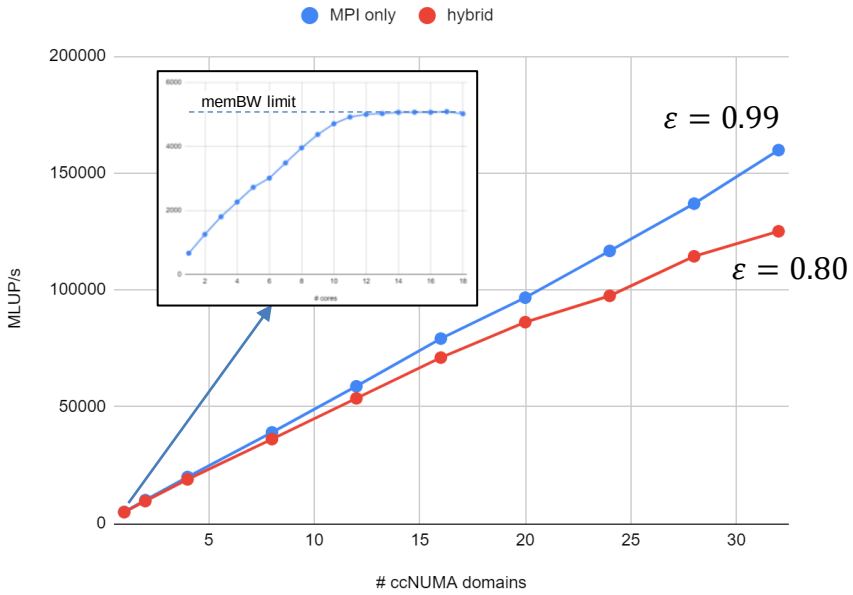
- Cluster: “Fritz” at NHR@FAU
  - 2x 36c Intel Ice Lake CPU per node
    Sub-NUMA Clustering (18 cores per NUMA domain)
  - Memory BW ~ 160 GB/s per socket (2 NUMA domains)
  - HDR-100 fat-tree interconnect
  - Intel compiler, Intel MPI

- Problem size 8000x8000 (working set ~ 1 GB)
  - Message size 64000 byte

Jacobi 2D – Benchmarking

- Up to 8 nodes (32 NUMA domains)
- MPI only vs. MPI+OpenMP
- Hybrid: 18 OpenMP threads per process, one process per NUMA domain

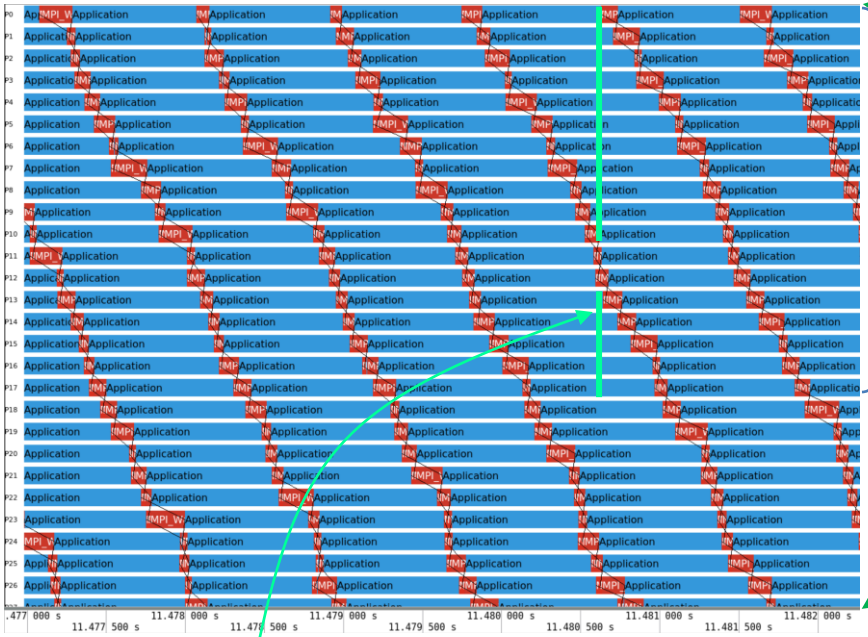
- Code behaves according to memory BW limitation on one NUMA domain
- MPI-only scales better
- Why???



Parallel efficiency $\varepsilon(N) = \frac{P(N)}{nP(1)}$, where $P(N)$ = performance with N NUMA domains

Jacobi 2D – Benchmarking

Intel Trace Analyzer view of MPI-only run (4 nodes, 288 processes)

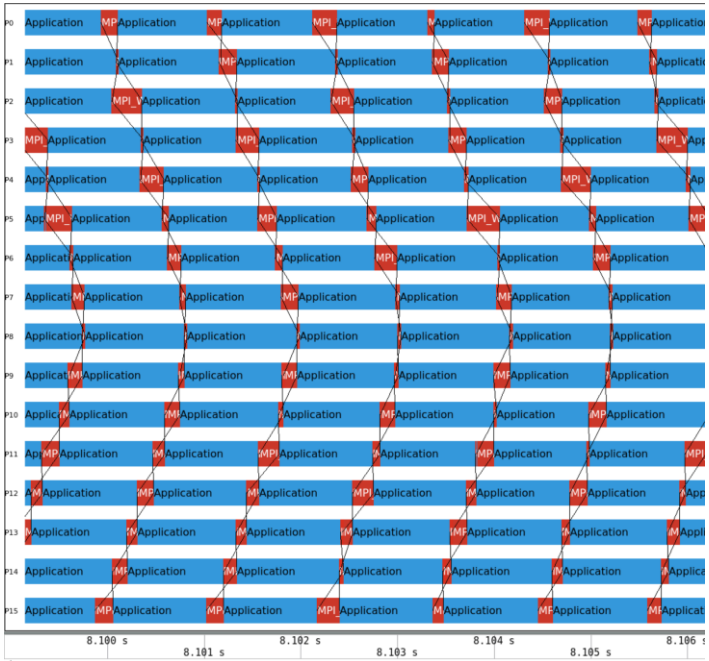


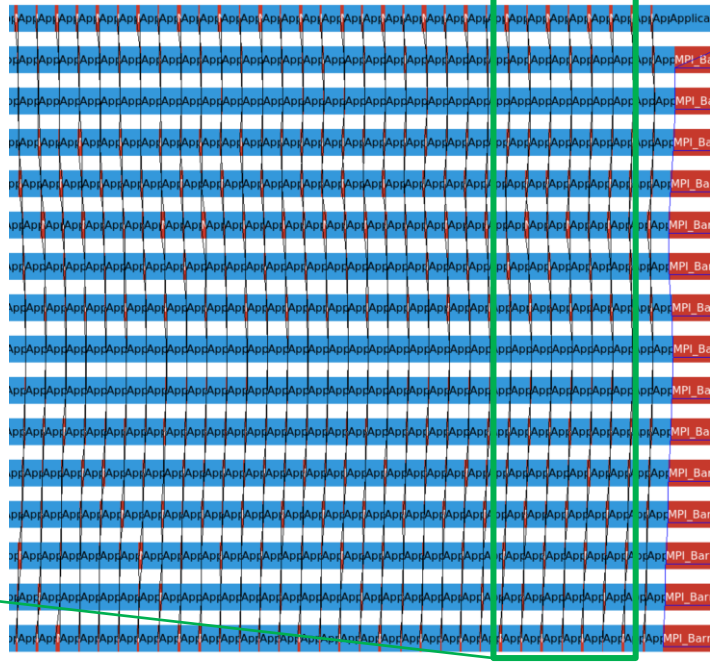
16 cores actively running code per NUMA domain

→ memory BW saturated!

Jacobi 2D – Benchmarking

Intel Trace Analyzer view of hybrid run (4 nodes, 16 processes)





Some skew across processes, but overhead is exposed → memory BW not saturated!

So why is pure MPI faster with the Jacobi code?

- The execution bottleneck is main memory bandwidth
- The execution is desynchronized across processes (no lock-step)
- As long as enough processes are actively working on a NUMA domain, the bottleneck is fully utilized → optimal performance
  - If a few cores spend time in MPI, nobody cares
  - MPI waiting times are overlapped with useful work across cores
- OpenMP forces the cores on a NUMA domain into lock-step → no desynchronization possible
  - MPI time is exposed as overhead → memory bandwidth not fully utilized
- Interested? More info:
  - Afzal et al., DOI: 10.1007/978-3-030-50743-5_20
  - Afzal et al., DOI: 10.1109/TPDS.2022.3221085, and references therein

Programming models - MPI + OpenMP

Hands-On #3

Masteronly hybrid Jacobi

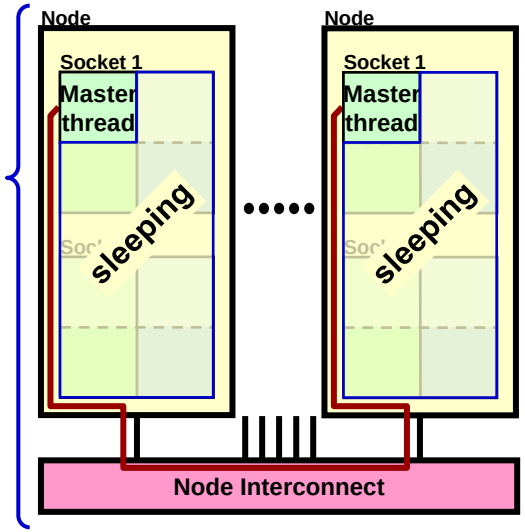
Programming models - MPI + OpenMP

Overlapping Communication and Computation

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*/
```

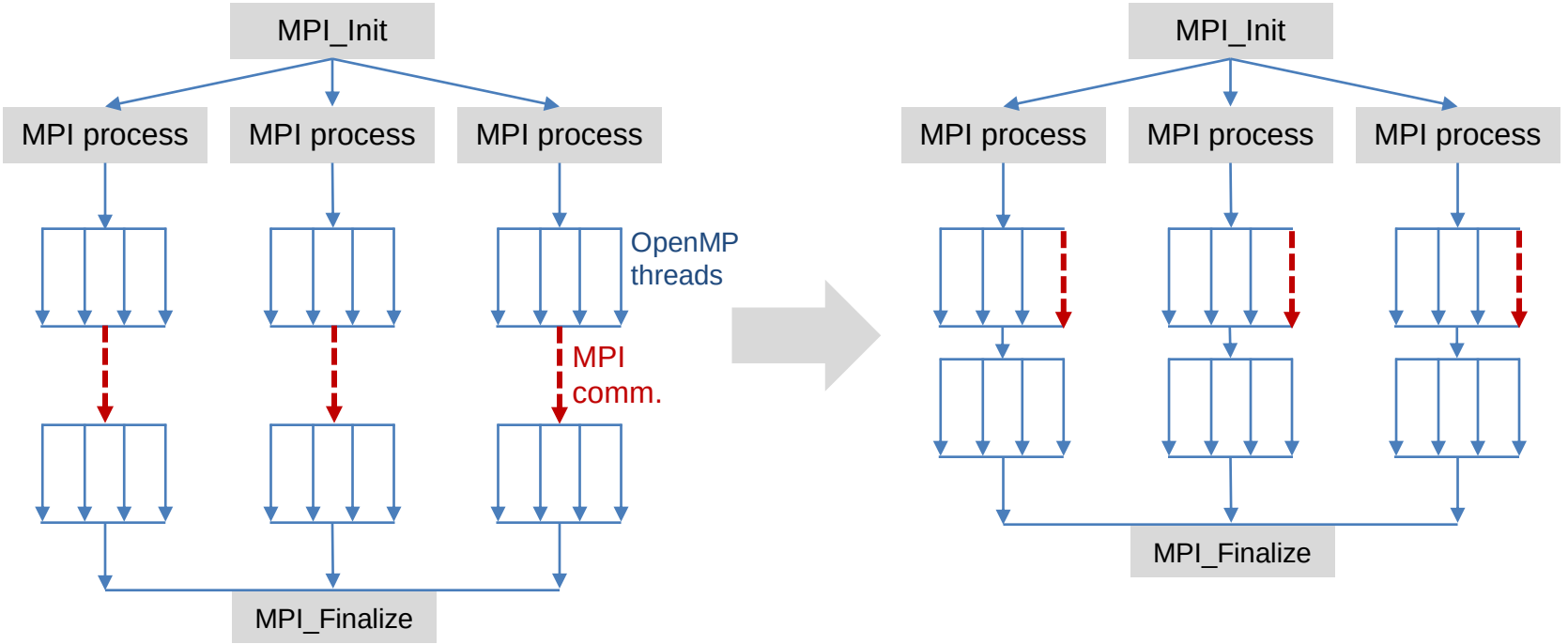


- Problem:
  - Sleeping threads are wasting CPU time
- Solution:
  - Overlapping of computation and communication
- Limited benefit:
  - Best case: reduces communication overhead from 50% to 0%
    → speedup of 2x
  - Usual case of 20% to 0%
    → speedup of 1.25x
  - Requires significant work → later

Nonblocking vs. threading for overlapped comm.

- Why not use nonblocking calls?
  - Nonblocking communication is important to prevent serializations and deadlocks, but asynchronous progress is 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

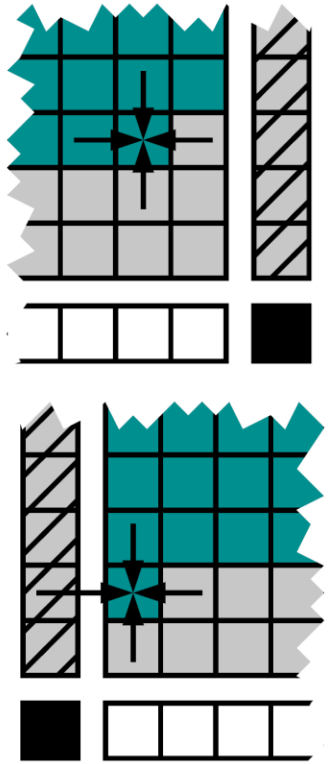


Explicit overlapping of communication and computation

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

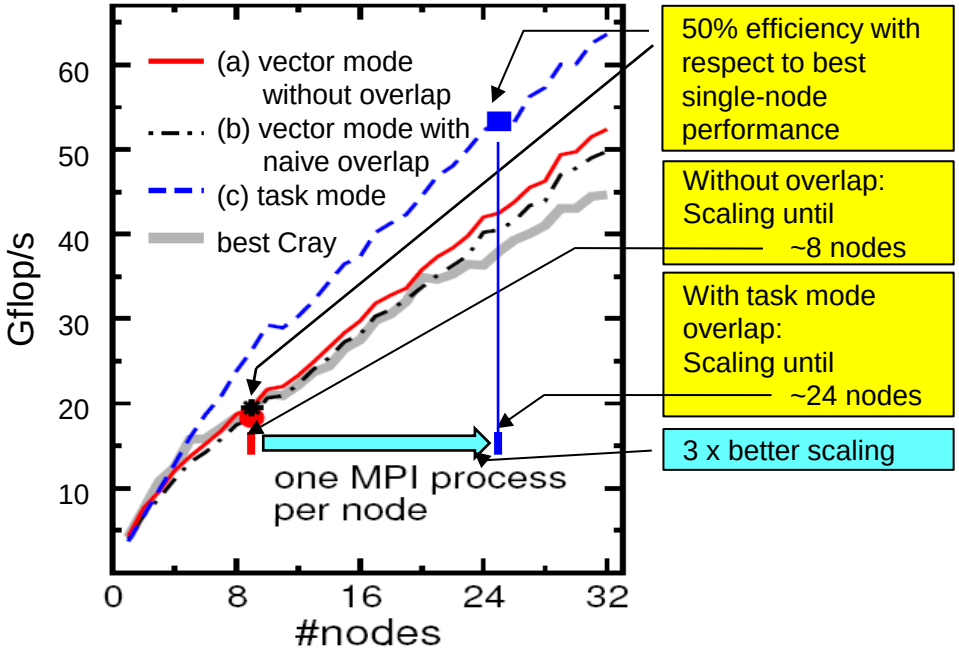
Three problems:

- 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

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

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

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*`)`

    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

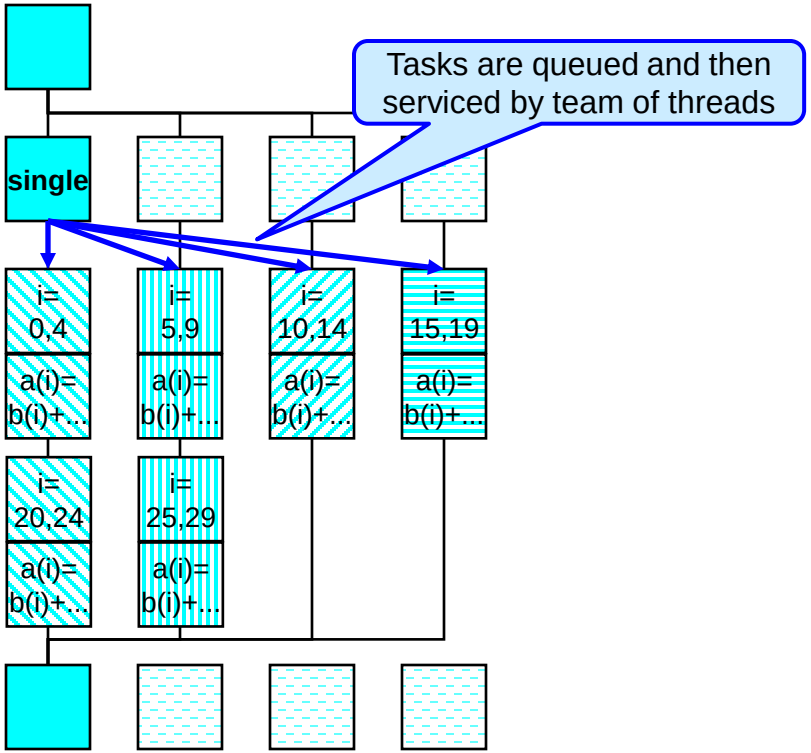
C/C++

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



skipped

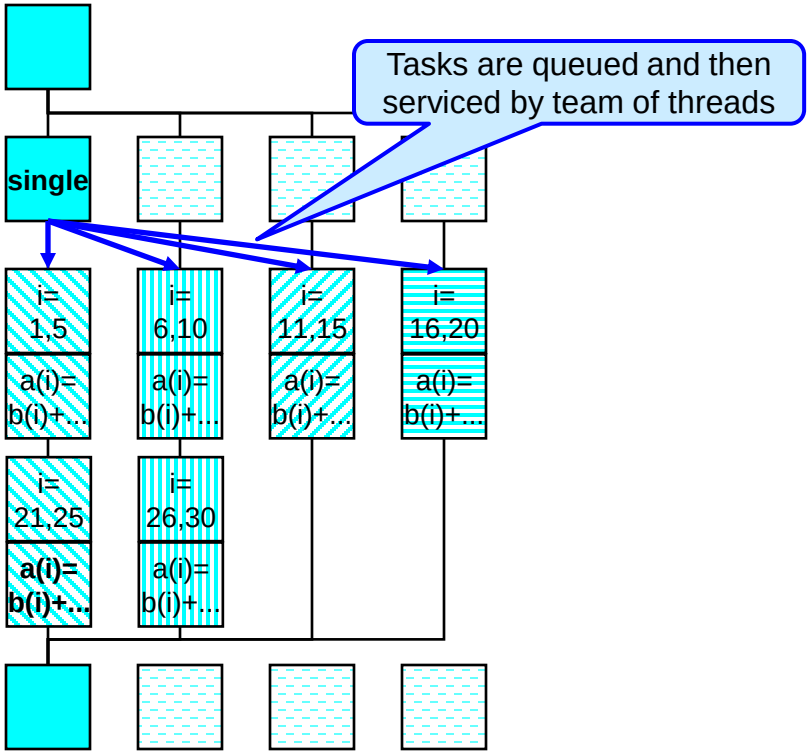
OpenMP `single` & `taskloop` Directives

Fortran

Fortran:

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

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



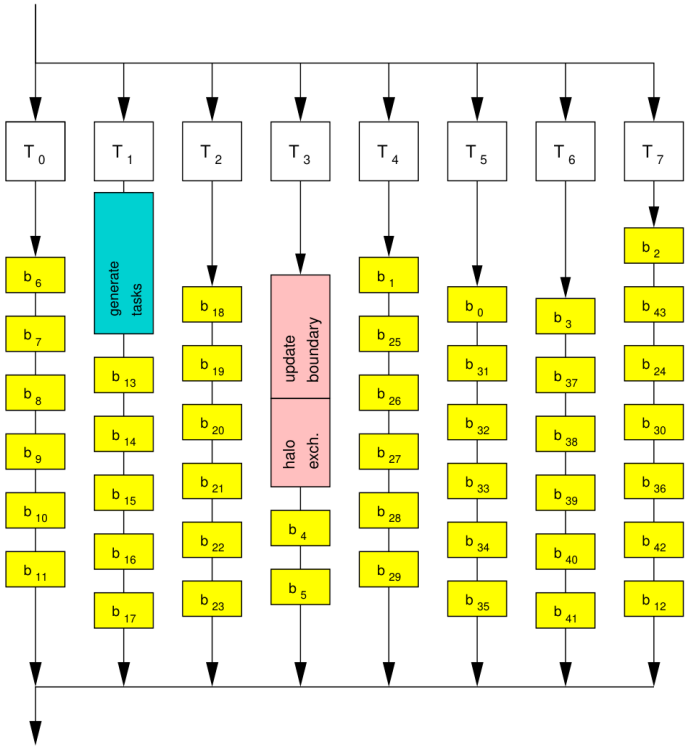
Comm. overlap with task & taskloop Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task[1)]
    { // 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



[1)] Adding a `priority(1)` clause may help that the MPI communication is not delayed by some numerical tasks generated by `#pragma omp taskloop`.

Programming models - MPI + OpenMP

Hands-On #4

Taskloop-based hybrid Jacobi

http://tiny.cc/MPIX-HLRS

Programming models - MPI + OpenMP

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 (if only one MPI process per node)
    (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)

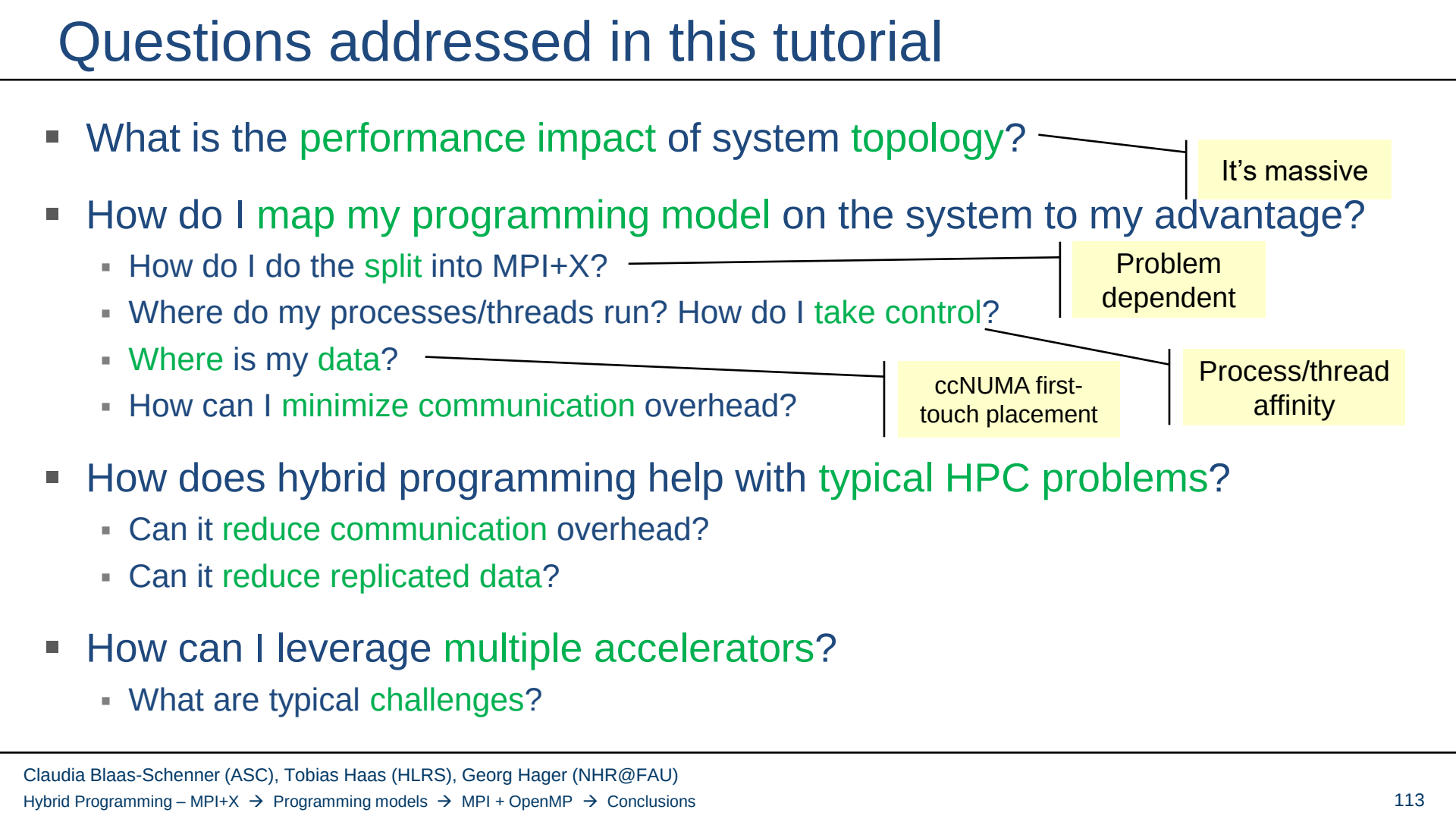
- Sleeping threads

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?



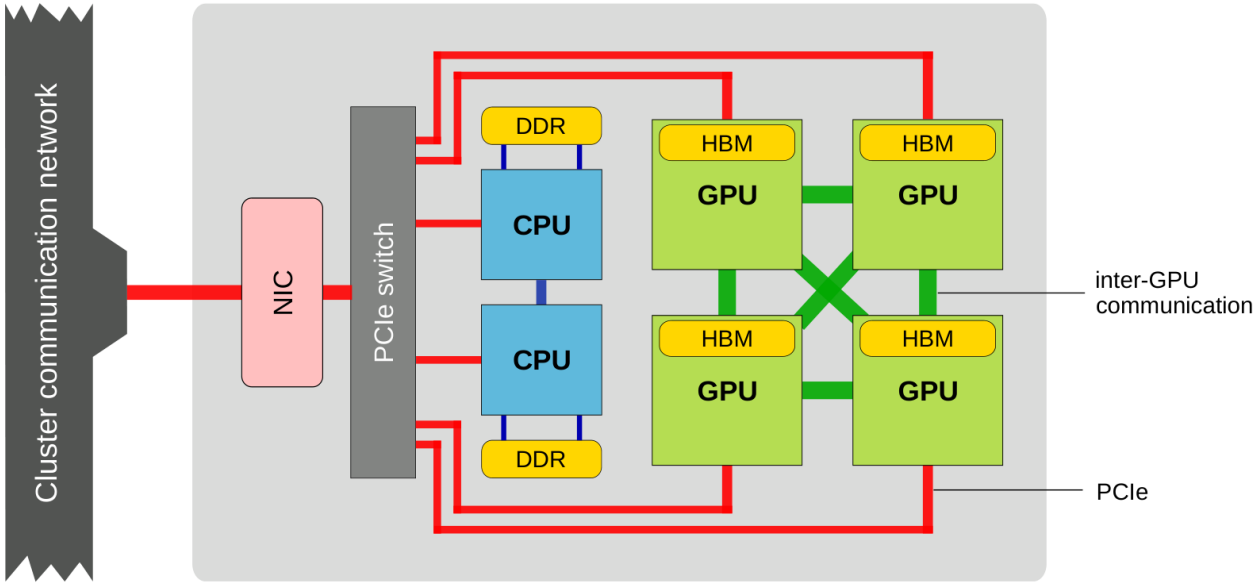
Programming models - MPI + Accelerator

Accelerated compute nodes and clusters
OpenMP offloading for accelerators
Case study: Accelerated stencil smoother
Advantages & main challenges, conclusions

Accelerated compute node design (I)

PCIe-attached accelerators

- No shared memory between host CPU and GPU
- Explicit copy operations or automatic “Unified Memory” required
- Multiple GPUs may still have high-performance interconnect among each other

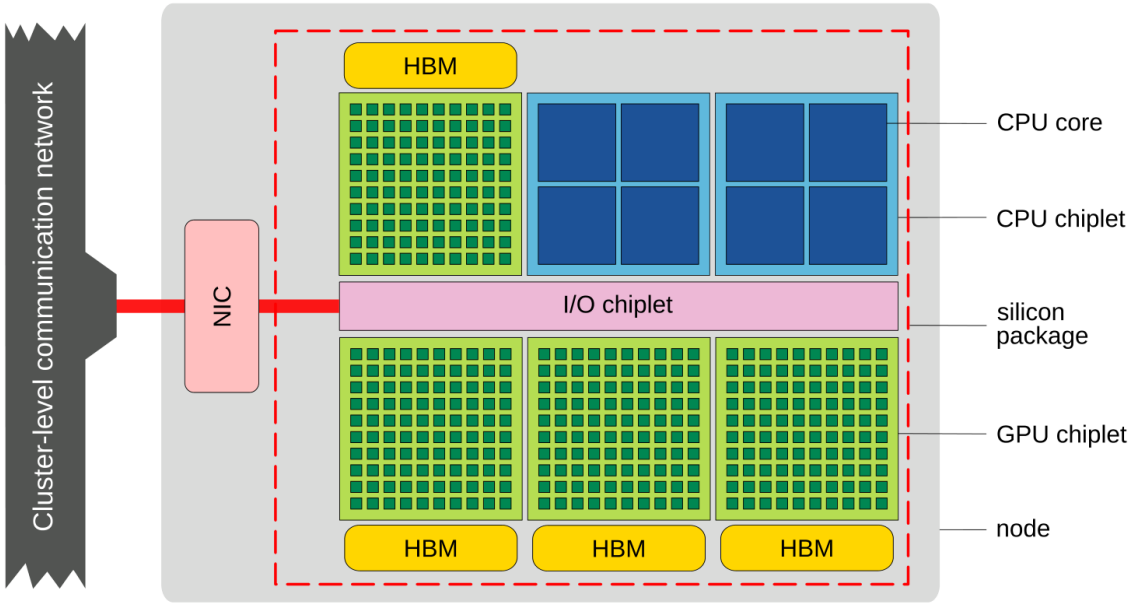


Accelerated compute node design (II)

Cache-coherent shared memory (UMA) between CPU and GPU

- Direct, shared access to HBM between CPU and GPU
- Possibly shared cache
- Rather weak CPU performance

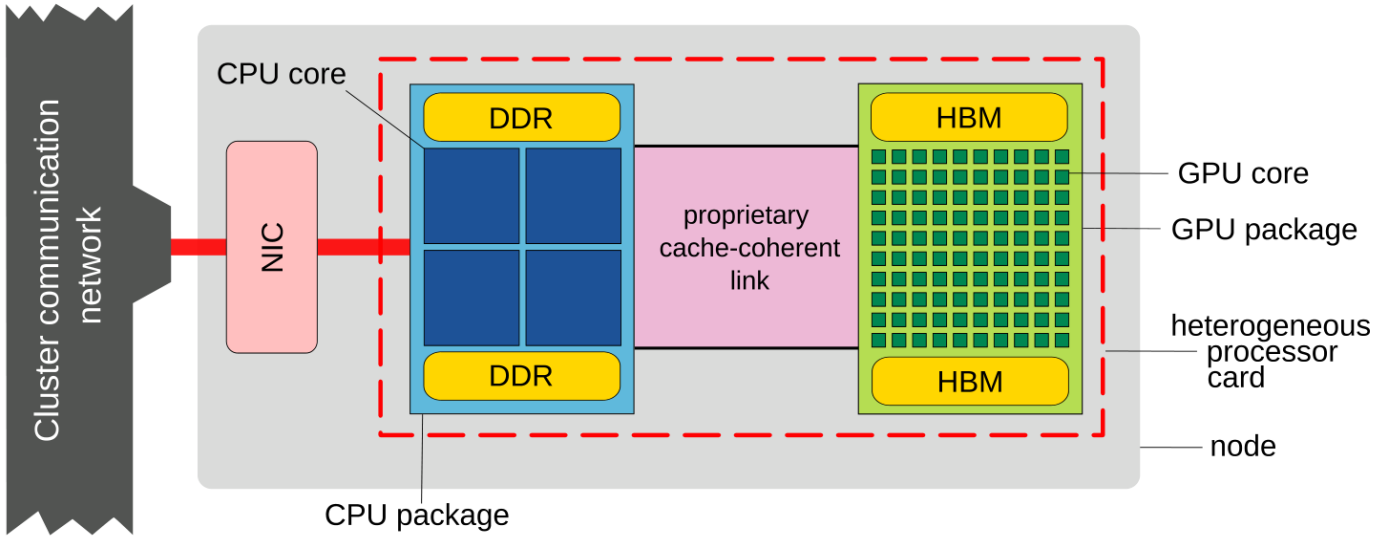
- Example: AMD MI300A



Accelerated compute node design (III)

Cache-coherent shared memory (ccNUMA) between CPU and GPU

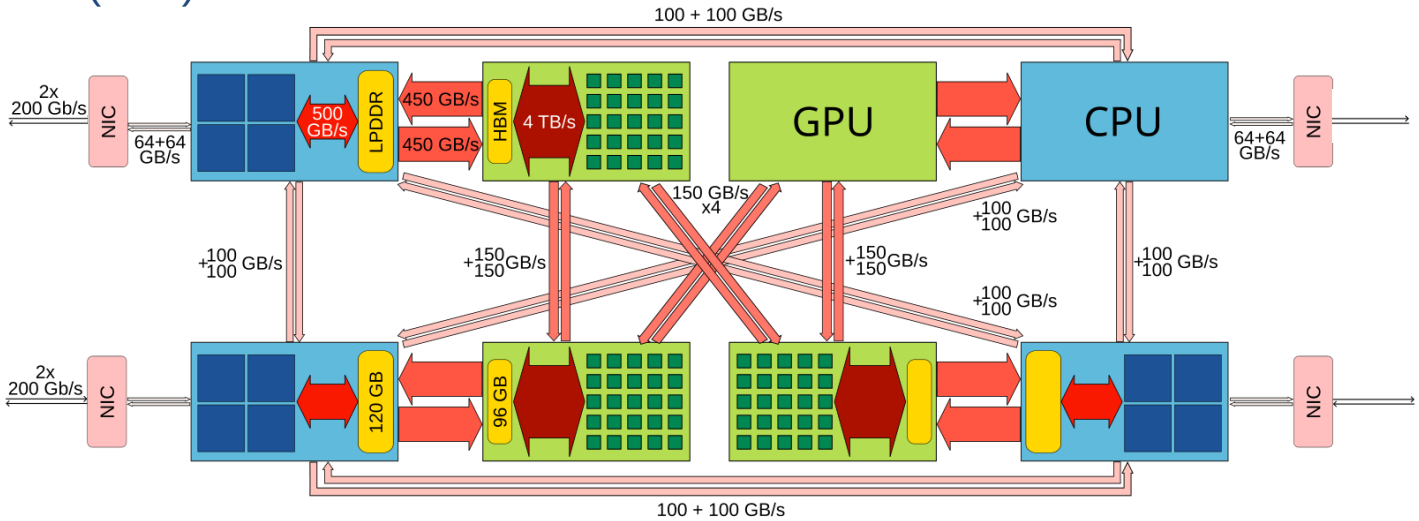
- Separate memory domains on CPU and GPU but cache-coherent shared memory
- Automatic page migration is possible
- Example:
  NVIDIA Grace-Hopper



Accelerated compute node design (IV)

Cache-coherent multi-accelerator node (ccNUMA+ccNUMA)

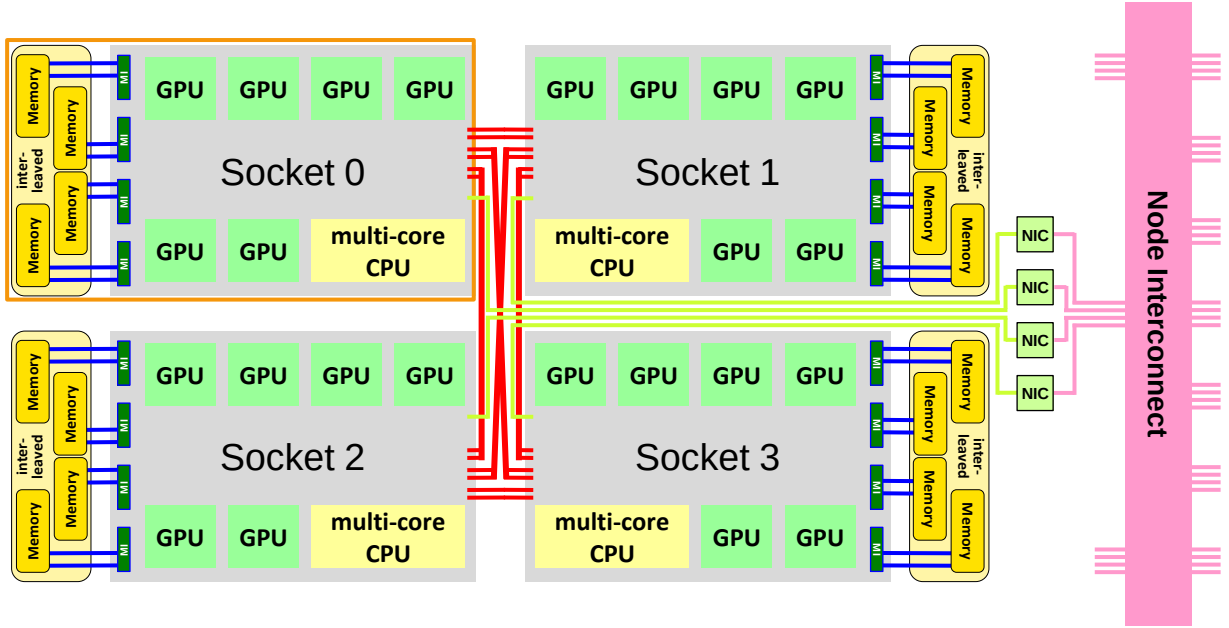
- Inter-accelerator bandwidth much lower than local CPU-GPU bandwidth
- Multiple NICs into communication network → topology matters!
- Example:
  JUPITER Booster node (JSC)



Accelerated compute node design (V)

Cache-coherent multi-accelerator node (UMA+ccNUMA)

- Inter-accelerator bandwidth much lower than local CPU-GPU bandwidth
- Multiple NICs into communication network → topology matters!
- Example:
  Hunter node (HLRS)



Accelerator programming: Bottlenecks reloaded

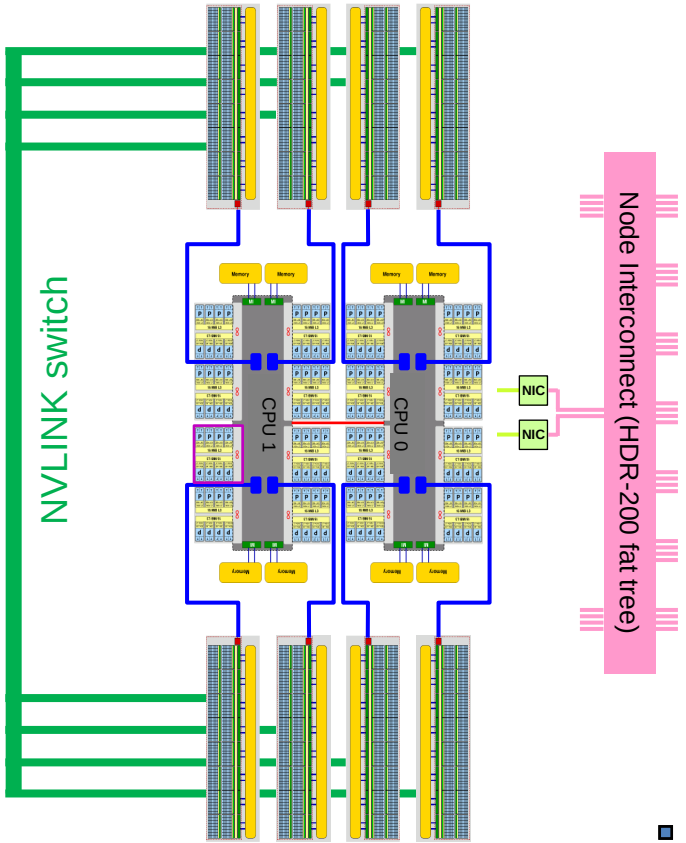
Example: 2-socket AMD “Zen3” (2x64 cores) node with eight NVIDIA A100 GPUs (PCIe 4) + NVLINK (“Alex” at NHR@FAU)

	per GPU	per CPU
DP peak performance	9.7 Tflop/s	2.0 Tflop/s
	0.13 B/F	0.08 B/F
eff. memory bandwidth	1300 Gbyte/s	160 Gbyte/s
inter-device BW (PCIe)	≈ 25 Gbyte/s (max.)	
inter-device BW (NVlink)	> 500 Gbyte/s	
Network BW (4x 100 Gbit/s)	2x 25 Gbyte/s (theor.)	

5x

8x

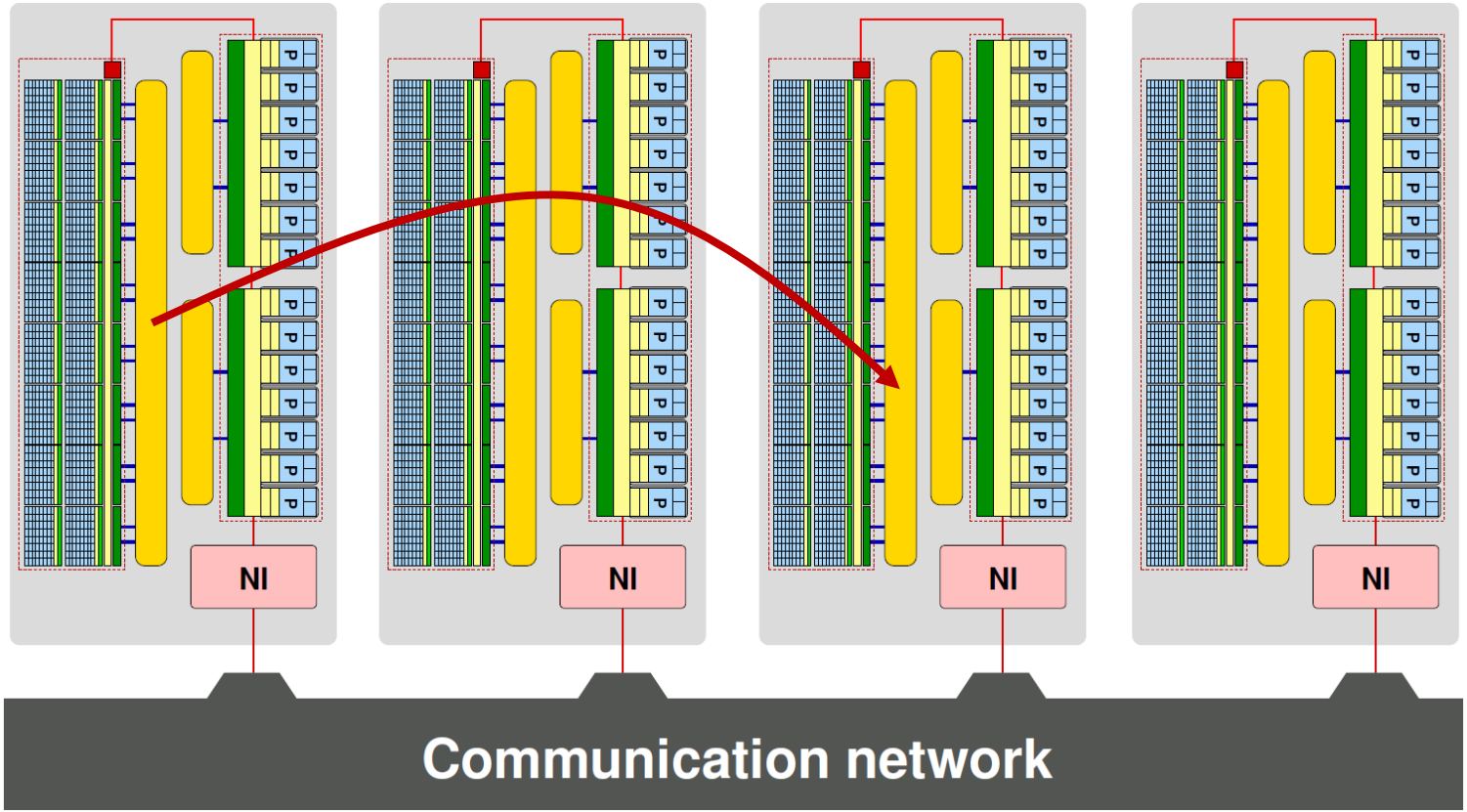
Machine balance



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

→ Basic estimates help

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



Questions to ask

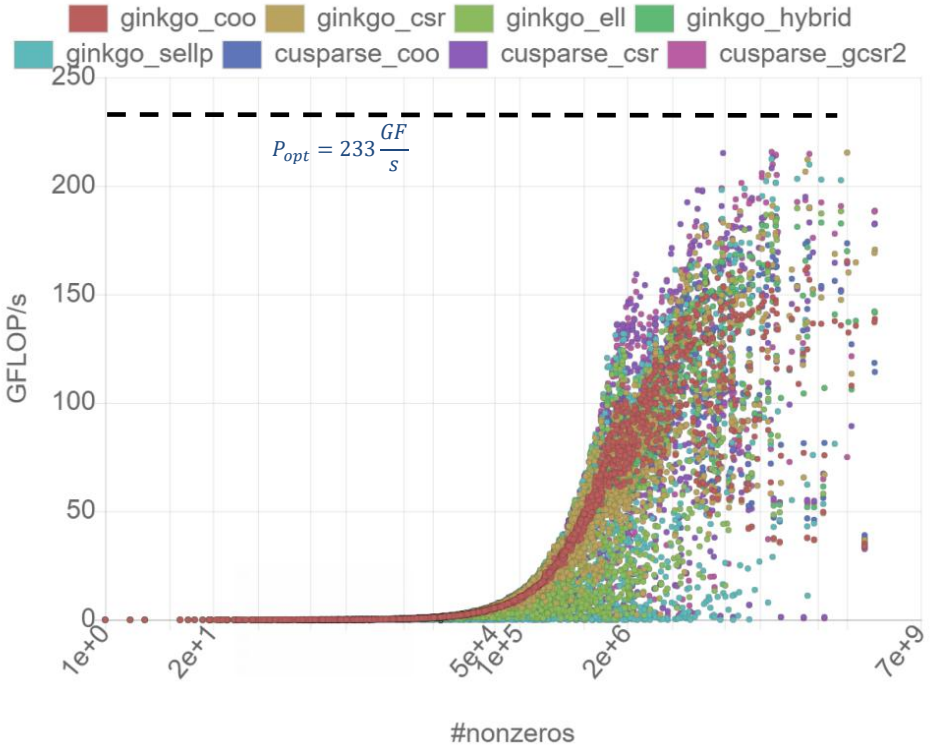
- Is the MPI implementation CUDA/GPU aware?
  - Yes: Can use device pointers in MPI calls
  - No: Explicit DtoH/HtoD buffer transfers required
  - Copying to consecutive halo buffers may still be necessary
- Is NVLink available?
  - Yes: Direct GPU-GPU MPI communication with MPI
    - Supported by: P100, 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 shared memory between CPU and GPU and across GPUs?
  - Memory placement may still by relevant

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!



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
SYCL, Kokkos,… (C++)
OpenACC
OpenMP offloading (4.0++)
special purpose
…

Which model/combination is the best???

→ the one that allows you to address the relevant hardware bottleneck(s)

*) PGAS = Partitioned Global Address space languages, CAF = Coarray Fortran, UPC = Unified Parallel C

Programming models
- MPI + Accelerator

OpenMP offloading for accelerators

(Slides adapted from material by M. Wittmann, NHR@FAU)

OpenMP offloading

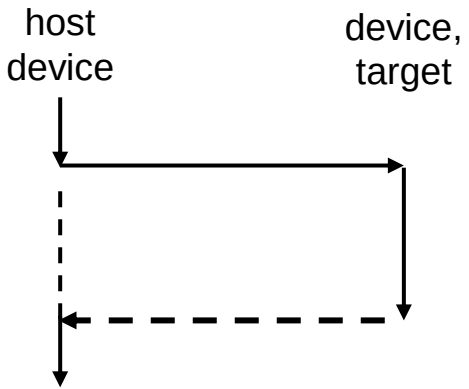
- OpenMP 4.0++ supports offloading of loops and regions of code from a host CPU to an attached accelerator in C, C++, and Fortran
- Set of compiler directives, runtime routines, and environment variables
- Simple programming model for using accelerators (GPGPUs and other many-core chips)

- 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 (teams, threads, SIMD)

Introduction

- Execute code on a device, typically an accelerator
  - OpenMP tries to abstract from the targeted device's architecture

- **target**: device where code and data is offloaded to
- execution always starts on the **host device**



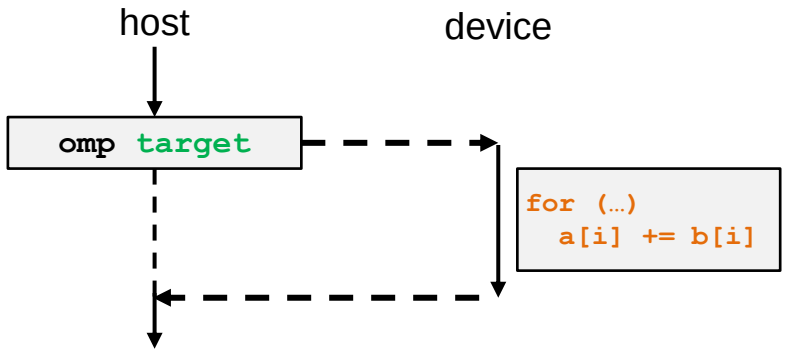
`target` construct

```
target [clauses…]
<structured block>
```

- execute associated structured block on the device
- on the target:
  - execution is initially single threaded
- on the host:
  - wait until offloaded code completes

- **`target`** construct cannot be nested inside another **`target`** construct

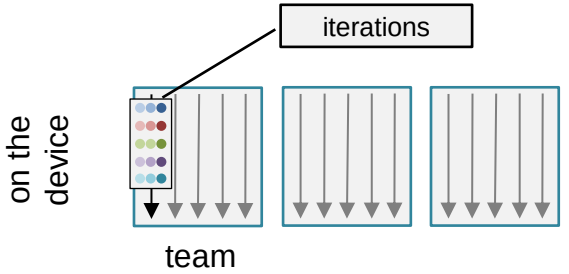
```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



Generating Parallelism

- **`target`** construct alone does not generate parallelism

```
#pragma omp target
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

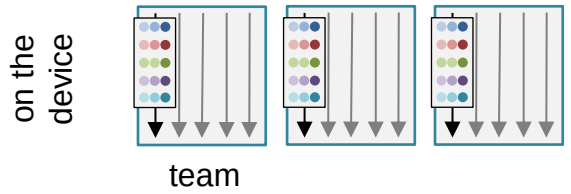


visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism

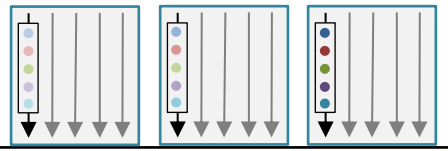
- `teams` construct
  - generates **league of teams**
  - A team has only one initial thread
  - Each team executes the same code
  - Number of teams: implementation defined
    - `num_teams(n)` clause

```
#pragma omp target teams
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- `distribute` construct
  - distributes iteration space of associated loop(s) over teams

```
#pragma omp target teams distribute
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

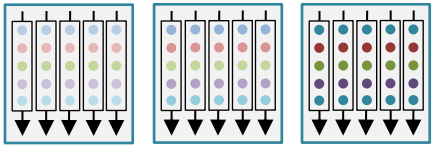


visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism

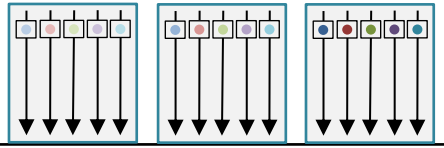
- `parallel` construct
  - gen. parallel region with multiple threads inside each team

```
#pragma omp target teams distribute \
                    parallel
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- worksharing loop
  - distribute team's iteration space over all threads inside a team

```
#pragma omp target teams distribute \
                    parallel for
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism

- **`simd`** construct
  - use SIMD lanes in each thread

```
#pragma omp target teams distribute \
                    parallel for simd
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

- how each directive maps to a GPU entity depends on the compiler

Generating Parallelism

- On GPUs, which construct maps to what entity depends on the compiler
  - might be that **`teams`** → threadblock/work group
  - might be that **`parallel`** [**`simd`**] → threads/work item

- Compilers with OpenMP offload support
  - GCC
  - LLVM/Clang
  - AMD
  - HPE/Cray
  - Nvidia
  - Intel

Generating Parallelism

Some possible combinations

```
omp target <sb>
omp target loop <ln>
omp target parallel <sb>
omp target parallel for/do <ln>
omp target parallel for/do simd <ln>
omp target simd <ln>
omp target teams <sb>
omp target teams loop <ln>
omp target teams distribute <ln>
omp target teams distribute parallel for/do <ln>
omp target teams distribute parallel for/do simd <ln>
omp target teams distribute simd <ln>
```

sb: structured block
ln: loop nest

skipped

`target teams` construct

- each team has a new initial thread
- teams are loosely coupled
  - in contrast to the `parallel` construct
- no synchronization across teams

```
#pragma omp target teams
{ … }
```

```
#pragma omp target
#pragma omp teams
{ … }
```

`target teams` must be a compound construct or directly nested

clauses:

- `num_teams(expr)` clause
  - no. of teams to create
  - if unspecified gen. no. of teams is implementation defined
- `thread_limit(expr)` clause
  - max. no. of active threads in a team
- `if(expr)` clause
  - evaluate to true: create teams
  - evaluate to false: create only 1 team
- `shared, private, firstprivate, default`:
  - usual meaning
- `reduction` clause: see later

skipped `distribute` construct

- distribute iterations of associated loop over teams
  - must be strictly nested inside a `teams` construct
  - iteration space must be the same for all teams
  - no implicit barrier at the end

```
#pragma omp target teams distribute
<loop>
```

`distribute` must be a compound construct or strictly nested

```
#pragma omp target teams
#pragma omp distribute
<loop>
```

- `dist_schedule(static[,chunk_size])` clause
  - if unspecified: implementation defined
  - w/o `chunk_size`: each team gets one equally sized chunk
- `collapse(n)` clause
  - same as for `for`/`do` construct
  - associate and collapse iteration space of n nested loops

skipped

`distribute` construct

- `private`, `firstprivate`, `lastprivate` clauses: usual meaning
- `order` clause: not handled here

- reproducible schedule:
  - `order(reproducible)`
  - `dist_schedule(static[,chunk_size]) order(...)` where order does not contain `unconstrained`

- avoid data races with `lastprivate`
  - `lastprivate` variables should not be accessed between end of `distribute` and `teams` construct

```
#pragma omp target teams
{
  #pragma omp distribute \
              lastprivate(lp)
  { <loop> }
  /* other code       */
  /* do not access lp */
}
```

A simpler solution: the `loop` construct

- Leave it to the compiler and the runtime to figure stuff out:

“prescriptive”

```
#pragma omp target teams distribute \
                  parallel for simd
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```


“descriptive”

```
#pragma omp target loop
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

- In practice, it might be good to add some prescriptivism:

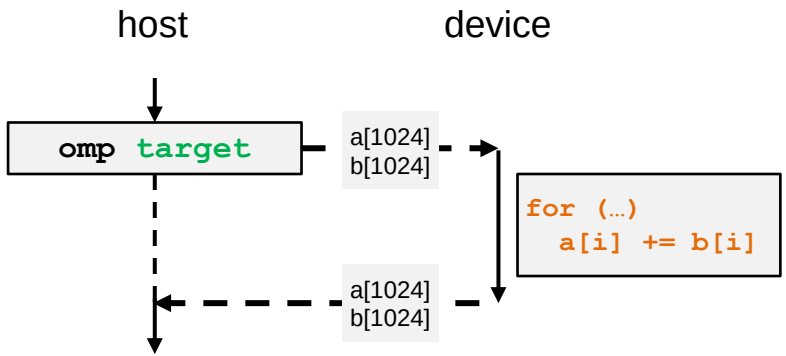
```
#pragma omp target teams loop \
                  num_teams(800) \
                  thread_limit(128)
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

Data Mapping

- host and device memory can be separate
- mapping of variables ensures
  - a variable is accessible on the target, e.g. by copy or allocation
  - a consistent memory view
- what can be mapped:
  - variables, array sections, members of structures
- mapping causes a presence check
  - copy to device only if not already present
- mapping attributes can be
  - implicit or explicit

```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```

here, implicit mapping attributes cause variables to be mapped, note a[1024], b[1024]



`map` clause

- map clause

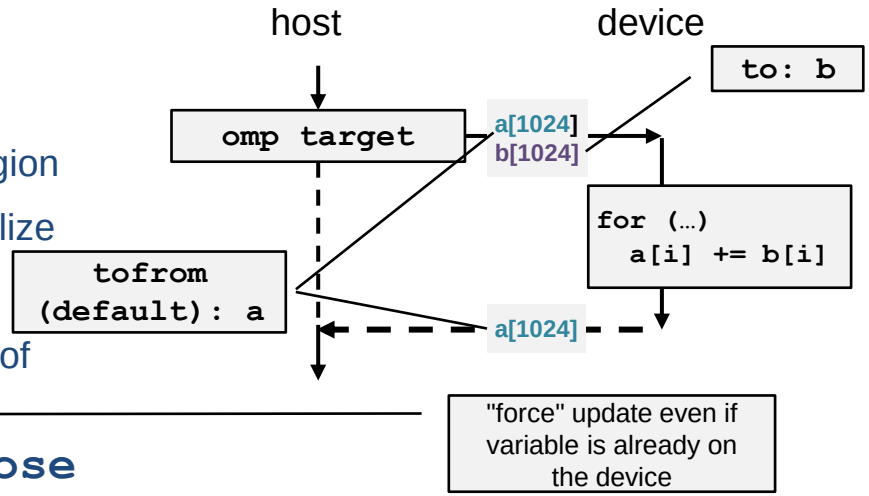
  `map([<mtm>,]<map-type>: <variables>)`

- `map-type`: how a variable is mapped

`tofrom`	default, copy to device on entry of target region and back at the end
`to`	copy to device on entry of target region
`from`	allocate on entry of target region, copy from device to host on exit of target region
`alloc`	on entry, allocate on device, but do not initialize
`release`	counterpart to `alloc`
`delete`	removes variable from device (independent of RC)

- `mtm`: `map-type-modifier: always, close`

```
int a[1024], b[1024];
/* init a and b */
#pragma omp target map(a) map(to:b)
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



Allocating on the Device

- map-type **`alloc`**
  - allocate variable/array on device
  - no initialization is performed
  - no copy back to host
- useful, e.g. when an array is only used on the device

`tmp` allocated on the device

```
int tmp[1024];

#pragma omp target map(alloc:tmp)
{
  for (int i = 0; i < 1024; ++i)
    tmp[i] = compute(i);

  for (int i = 0; i < 1024; ++i)
    work(tmp[i]);

  for (int i = 0; i < 1024; ++i)
    work2(tmp[i]);
}
```

`tmp` not copied back

How to map dynamically allocated arrays in C/C++

- map dynamically allocated arrays via array section syntax

`array[[lower-bound]:length]`

```
double * a = malloc(sizeof(double) * n_el);
double * b = malloc(sizeof(double) * n_el);
/* init a */

#pragma omp target map(to:a[:n_el]) \
                   map(alloc:b[:n_el])
for (int i = 0; i < n_el; ++i) {
  b[i] = a[i];
}
```

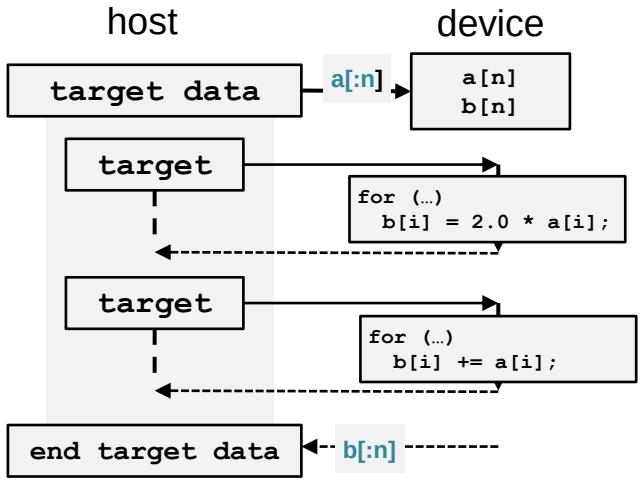
`target data` construct

```
target data [clauses]
<block>
```

- map data for the duration of the associated block to the DDE
  - `<block>` still executed on host
  - `<block>` typically includes multiple `target` regions
- clauses:
  - `map()` with `to`, `from`, `tofrom`, `alloc`
  - not covered: `device`, `if`, `use_device_addr`, `use_device_ptr`

```
#pragma omp target data map(to:a[:n]) \
                        map(from:b[:n])
{
  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] = 2.0 * a[i]; }

  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] += a[i]; }
}
```



`target update` Construct

`target update [clauses]`

- copy data between host and device
  - runs on the host
  - cannot appear inside a `target` construct
  - copy is always performed
    - in contrast to `target map(…)`
- clauses
  - `to(var-list)` copy vars. to device
  - `from(var-list)` copy vars. to host
  - not covered: device, if, nowait, depend

```
#pragma omp target data map(to:a[:n]) \
                        map(from:b[:n])
{
  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] = 2.0 * a[i]; }

  #pragma omp target update from(b[:n])
  /* do something with b */

  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] += a[i]; }
}
```

`enter data`/`exit data` directives

```
target enter data map(…)[clauses]
target exit data map(…) [clauses]
```

→ map data

→ unmap data

- **unstructured**
- can be called at any point on host
- at `exit data`: listed variables not present on the device are ignored
- clauses not covered: `device`, `if`, `depend`, `nowait`

allowed: to, alloc

```
double * vec_allocate(int n_el)
{
  double * a = malloc(…);
  #pragma omp target enter data \
              map(alloc:a[:n_el])
  return a;
}

void vec_free(double * a)
{
  #pragma omp target exit data \
              map(release:a[:n_el])
  free(a);
}
```

allowed: from, release, delete

`target data use_device_ptr` directive

```
target data use_device_ptr(<list>)
<structured-block>
```

- Indicates that list item is pointer to object with corresponding storage on device
- References to list items in structured block are converted to local pointer with device address
- "**In this block, use device addresses for these pointers**"
- Useful if functions need to be handed device pointers (e.g., GPU-aware MPI)

```
double * p = malloc(n);

#pragma omp target data map(p[:n])
{
 // call host func with device ptr
#pragma omp target data use_device_ptr(p)
   accel_func(p);
}
```

`device()` clause

- Clause for **`target`** directive
- Selects target device by ID
- device (int_expr)

```
int dev = ...;
#pragma omp target device(dev)
{
  for (int i = 0; i < N; ++i)
    a[i] += b[i];
}
```

- API function: **`omp_set_default_device(int)`**

- Essential for multi-GPU parallelism
  - Question: How to assign device IDs to threads and/or processes?

Programming models - MPI + Accelerator

Hands-On #5

OpenMP offloading

http://tiny.cc/MPIX-HLRS

Assigning GPUs to threads (1)

Simple setup: one thread per device (no MPI)

```
int num_dev = omp_get_num_devices();
int sz = N / num_dev; // assuming no remainder
#pragma omp parallel num_threads(num_dev)
{
  int tID = omp_get_thread_num();
  int istart = tID * sz;
  int iend = istart + sz;

  #pragma omp target loop device(tID)
  for (int i = istart; i < iend; ++i)
    a[i] += b[i];
}
```

Assigning GPUs to threads (2)

- One MPI process per device…
  - … assuming **num_dev <= procs_per_node**
  - and a linear mapping of MPI ranks to nodes

```
int dev_ID = my_rank % omp_get_num_devices();
int sz = N / mpi_procs; // assuming no remainder
int istart = my_rank * sz;
int iend = istart + sz;

#pragma omp target loop device(dev_ID)
for (int i = istart; i < iend; ++i)
  a[i] += b[i];
```

Assigning GPUs to threads (3)

- One MPI process per device…
  - … checking if sufficient GPUs are available on each node
  - and making no assumptions about rank mapping

```
MPI_Comm shared_node_comm; int node_rank;
MPI_Comm_split_type(MPI_COMM_WORLD , MPI_COMM_TYPE_SHARED ,
     0, MPI_INFO_NULL , &shared_node_comm);
MPI_Comm_rank(shared_node_comm, &node_rank);
if(node_rank >= omp_get_num_devices()){
  printf("No device for node rank %d\n", node_rank);
  MPI_Abort(MPI_COMM_WORLD, 17);
}
MPI_Comm_free(&shared_node_comm);
int sz = N / mpi_procs;
int istart = my_rank * sz; int iend = istart + sz;

#pragma omp target loop device(node_rank)
for (int i = istart; i < iend; ++i)
  a[i] += b[i];
```

Some useful API functions for offloading

- `int omp_get_num_devices()`
  Get number of available target devices (only from outside)
- `int omp_get_device_num()`
  Get ID of device on which calling thread is executing
- `void omp_set_default_device(int)`
  Set device for upcoming target regions
  Overriden by `device()` clause of target directive
- `int omp_get_num_teams()`
  Get number of teams in the current teams region (or 1 if called from outside)
- `int omp_get_team_num()`
  Get initial team ID of calling thread (or 0 if called from outside)

Programming models
- MPI + Accelerator

Case study: Accelerated stencil smoother

Accelerating an MPI/hybrid Jacobi 2D smoother

Domain sweep with offload directive (in **`relax()`** function):

```
#pragma omp target teams distribute parallel for collapse(2) \
   map(src[:n_cells]) map(dst[0:n_cells])
    for (int y = start_y; y < end_y; ++y)
      for (int x = start_x; x < end_x; ++x)
        dst[y * dim_x + x] =
          0.25 * (src[y*dim_x+x-1]   + src[y*dim_x+x+1]
                + src[(y-1)*dim_x+x] + src[(y+1)*dim_x+x]);
```

executed on device

- This alone would be sufficient to run the loop nest on the GPU
- **`map`** clause (in this form) copies arrays **`src[]`** and **`dst[]`** to the device before the loop and then back after
- Prize question: What is the expected performance in LUP/s (lattice site updates per second)? (Hint: it's abysmal)
- How can we do better?

Accelerating an MPI/hybrid Jacobi 2D smoother

Better solution: Copy arrays to device before the iteration loop and back after

```
  omp_set_default_device(my_rank % omp_get_num_devices());
#pragma omp target enter data \
  map(to:src[:n_cells]) map(to:dst[:n_cells])
  for (int iter = 0; iter < n_iterations; ++iter) {
    exchange(domain, src_grid);
    relax(domain, src_grid, dst_grid);
    swap(src_grid, dst_grid);
  }
#pragma omp target exit data \
  map(from:src[:n_cells]) map(from:dst[:n_cells])
```

Assuming ppn == num_devices, linear mapping

executed on host

- Entire “algorithm” is now on the GPU. Can we do even better?
- What about the halo communication?
  - Is the MPI implementation GPU aware?

Accelerating an MPI/hybrid Jacobi 2D smoother

Halo exchange without GPU-aware MPI: Update boundary cells from device (halos to device) before (after) communication

```
void exchange(...) {
...
#pragma omp target update \
  from(src[(dim_y - 1) * dim_x:dim_x],src[0:dim_x])
  // top neighbor xchg
  if (domain->comm_rank + 1 < domain->comm_size) {
    int top = domain->comm_rank + 1;
    MPI_Isend(&src[dim_y-1][0], dim_x,...,  &requests[0]);
    MPI_Irecv(grid->ghost_cells_top, dim_x, &requests[1]);
  }
  // bottom neighbor xchg
  if (domain->comm_rank > 0) {
    int bottom = domain->comm_rank - 1;
    MPI_Isend(&src[0][0], dim_x,...,             &requests[2]);
    MPI_Irecv(grid->ghost_cells_bottom,  dim_x, &requests[3]);
  }
  MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
#pragma omp target update \
  to(grid->ghost_cells_top[:dim_x],grid->ghost_cells_bottom[:dim_x])
}
```

executed on host

Accelerating an MPI/hybrid Jacobi 2D smoother

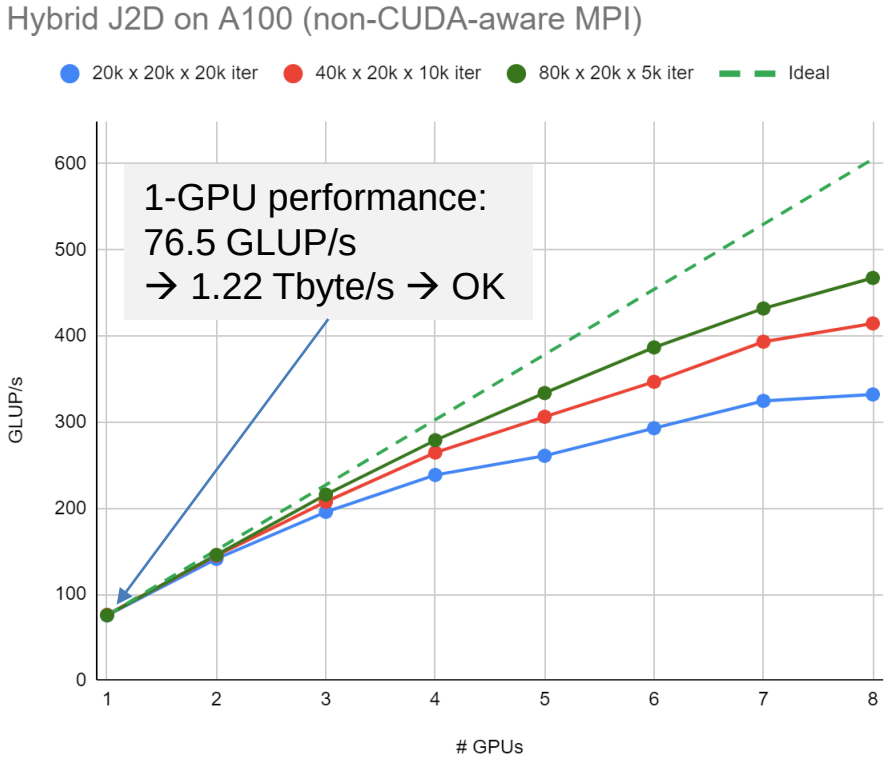
Halo exchange with GPU-aware MPI: Tell compiler to use device pointers in the region

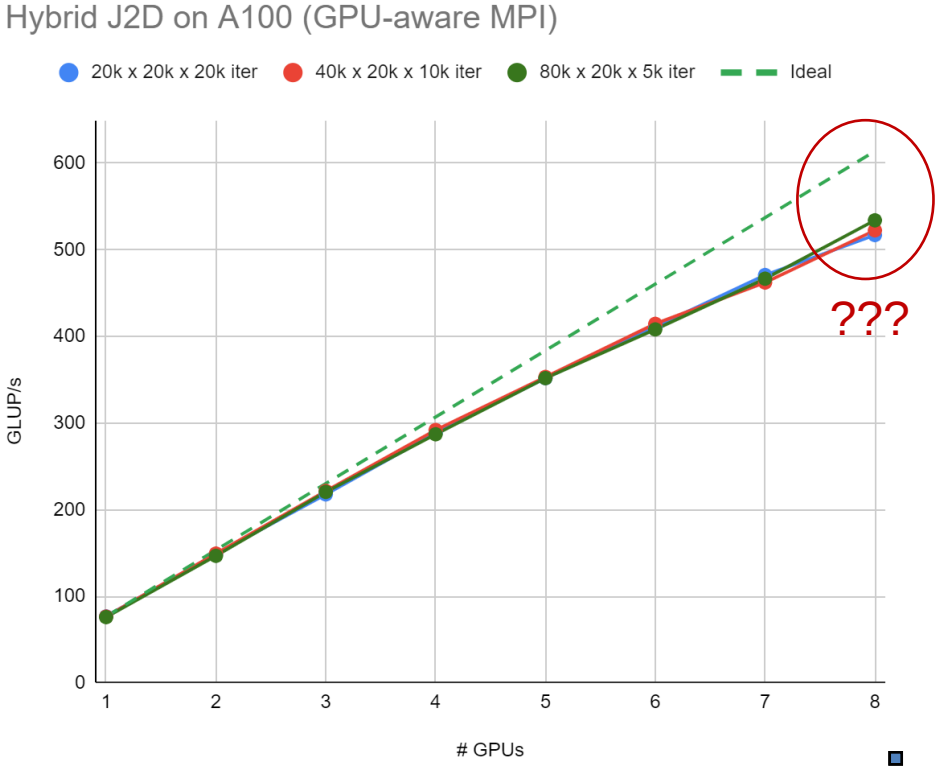
```
void exchange(...) {
...
  double *gct = grid->ghost_cells_top, *gcb=grid->ghost_cells_bottom;
#pragma omp target data \
  use_device_ptr(src, gct, gcb)
{
  // top neighbor xchg
  if (domain->comm_rank + 1 < domain->comm_size) {
    int top = domain->comm_rank + 1;
    MPI_Isend(&src[dim_y-1][0], dim_x,...,  &requests[0]);
    MPI_Irecv(gct, dim_x, &requests[1]);
  }
  // bottom neighbor xchg
  if (domain->comm_rank > 0) {
    int bottom = domain->comm_rank - 1;
    MPI_Isend(&src[0][0], dim_x,...,            &requests[2]);
    MPI_Irecv(gcb,  dim_x, &requests[3]);
  }
  MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
}
```

executed on host

J2D smoother multi-GPU scaling

“Alex” node (8x A100 40GB), nvhpc 23.7, OpenMPI 4.1.6



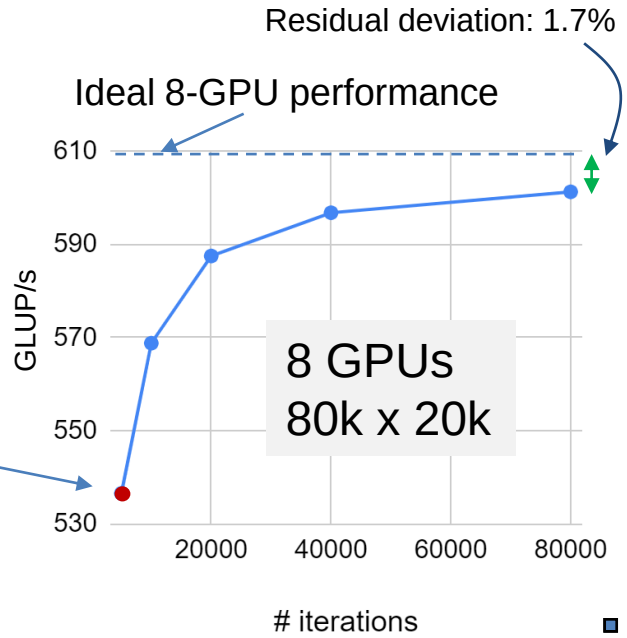


PCIe transfers

- Why is there no perfect scaling even at large problem sizes?
- Runtime at problem size 80k x 20k with 5k iterations: 15 s
- Transfer time of grids to and from accelerator:

$$T_{transfer} = \frac{80 \times 20 \times 10^6 \times 2 \times 16 \text{ byte}}{b_{PCI}} \approx 2 \text{ s}$$

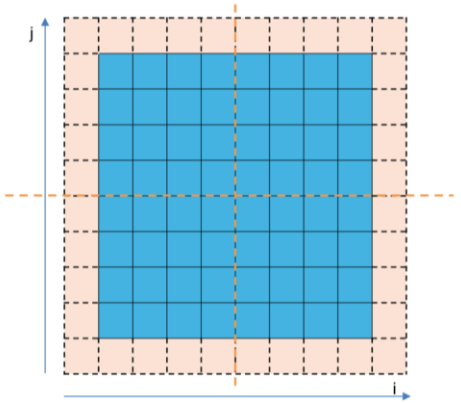
→ 13% of runtime goes into PCIe transfers

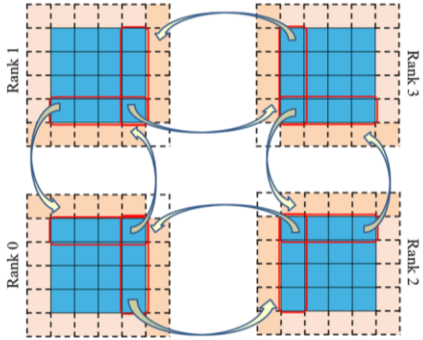


What about non-contiguous halos (and other types)?

- Optimal strategy depends heavily on MPI implementation
- Strided halo communication amounts to
  - MPI packing boundary cells into contiguous internal buffer
  - Sending the internal buffer to internal buffer on other process
  - Unpacking and storing data into discontinuous halo cells

- Is this done efficiently under the hood, even with GPUs?

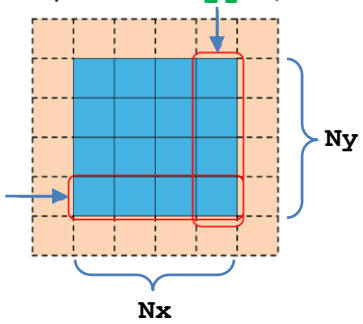




Artwork by Aditya Ujeniya

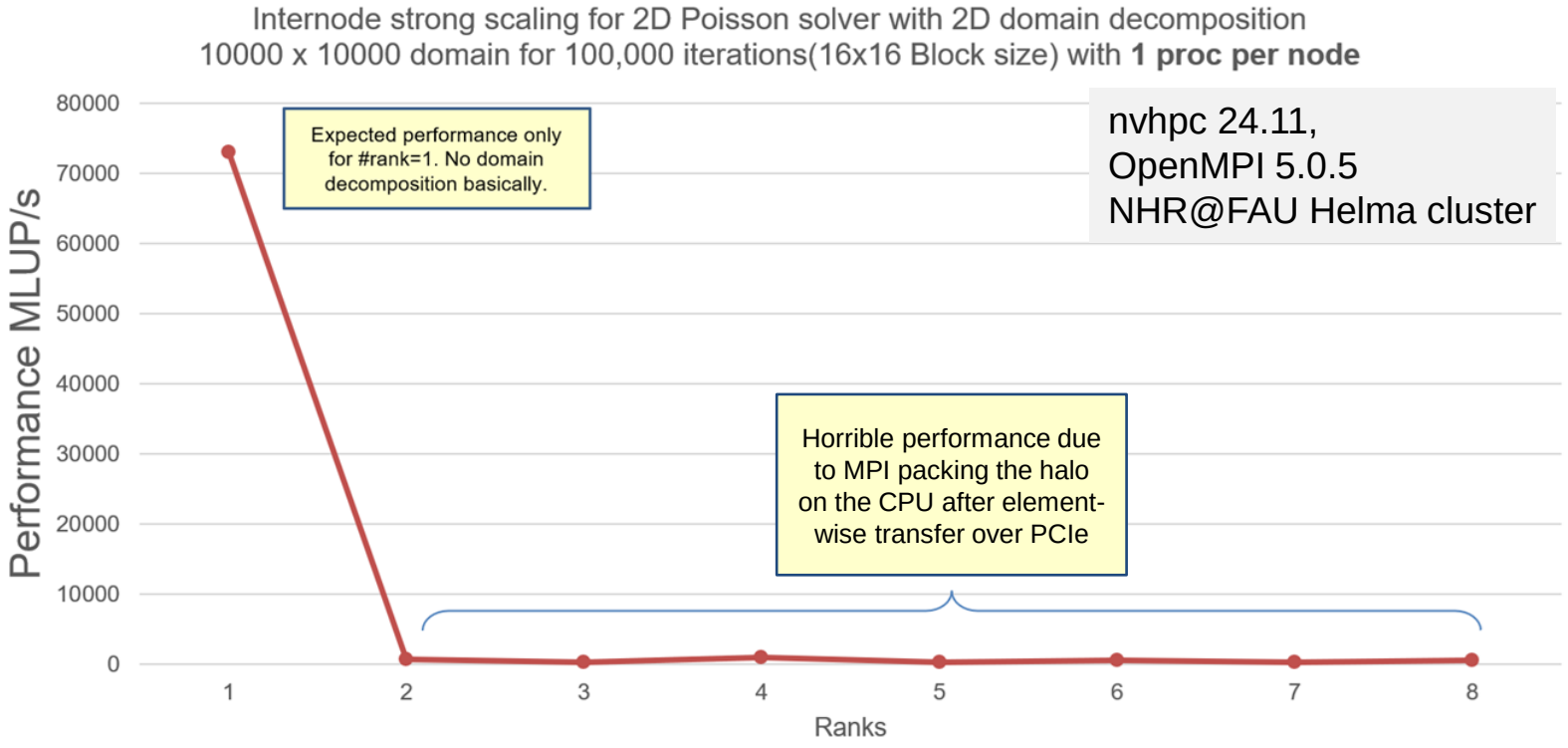
```
MPI_Type_vector(Ny, 1, Nx+2,
                MPI_DOUBLE, &newtype)
```

No type required



2D Poisson solver with (partly) non-contiguous halos

No.



https://blogs.fau.de/adityauj/2025/02/27/cuda-aware-mpi-part-3-solving-2d-poisson-equation-with-2d-domain-decomposition/

So what should we do if strided data types are slow?

```
void exchange(int Nx, int Ny, double data[Ny+2][Nx+2], ...) {
  double buf1[Ny], buf2[Ny], buf3[Ny], buf4[Ny];
  MPI_Request r[8]; MPI_Status statuses[8];

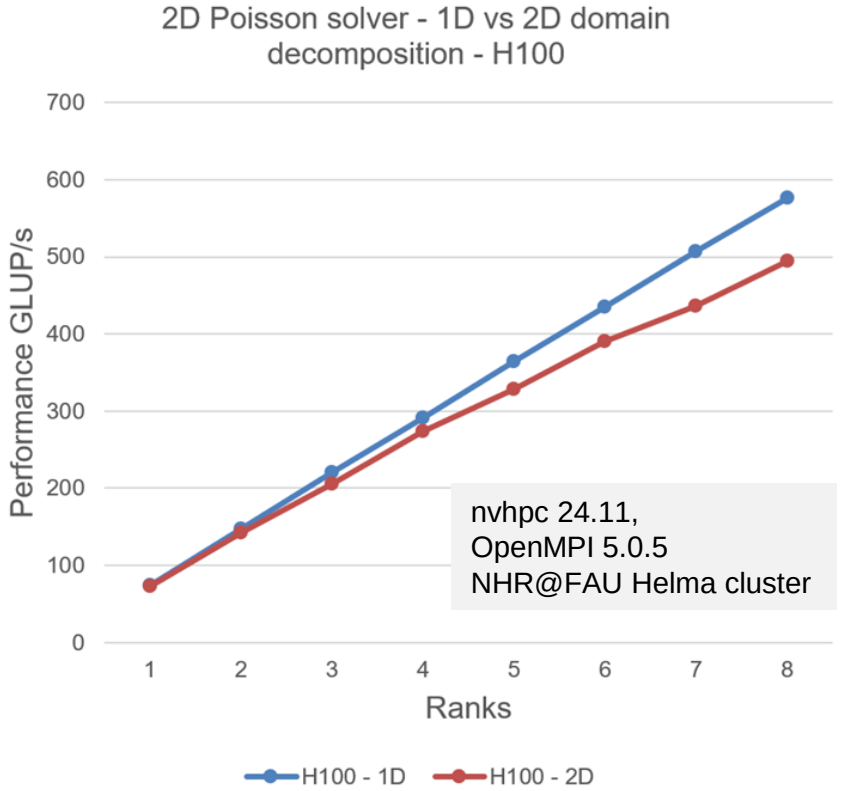
  #pragma omp target loop
  for(int i=1; i<=Ny; ++i)
  {  buf1[i-1] = data[i][Nx]; buf2[i-1] = data[i][1]; }
  MPI_Isend(buf1, Ny, MPI_DOUBLE, right, 0, MPI_COMM_WORLD, &r[0]);
  MPI_Isend(buf2, Ny, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &r[1]);
  MPI_Isend(&data[1][1], Nx, MPI_DOUBLE, top, 0, MPI_COMM_WORLD, &r[2]);
  MPI_Isend(&data[Ny][Nx], Nx, MPI_DOUBLE, bottom, 0, MPI_COMM_WORLD, &r[3]);
  MPI_Irecv(buf3, Ny, MPI_DOUBLE, right, 0, MPI_COMM_WORLD, &r[4]);
  MPI_Irecv(buf4, Ny, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &r[5]);
  MPI_Irecv(&data[0][1], Nx, MPI_DOUBLE, top, 0, MPI_COMM_WORLD, &r[6]);
  MPI_Irecv(&data[Ny+1][1], Nx, MPI_DOUBLE, bottom, 0, MPI_COMM_WORLD, &r[7]);
  MPI_Waitall(8, r, statuses);
  #pragma omp target loop
  for(int i=1; i<=Ny; ++i)
  {  data[i][Nx+1] = buf3[i-1]; data[i][0] = buf4[i-1]; }
}
```

Copy strided boundary cells into contig. buffers

Copy received halos into strided local halo cells

2D Poisson with explicit “halo consolidation”

- Caveat: Kernel launches for halo consolidation still incur latency
- The smaller the domains, the larger the (relative) penalty
- Might try to hide the communication/packing/unpacking cost



Programming models
- MPI + Accelerator

Optimizing OpenMP offload code

General considerations
OpenMP offloading for accelerators
Case study: Accelerated stencil smoother
> **Optimizations**
Advantages & main challenges, conclusions

Working with unified shared memory

- Proper USM support makes map clauses redundant

```
#pragma omp requires unified_shared_memory
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

for (int i=0; i<M; i++)
  in[i] = func(i);

#pragma omp target loop map(to:in[0:M]) map(tofrom:out[0:M])
for (int i=0; i<M; i++)
  out[i] = ... in[i] ...;

for (int i=0; i<M; i++)
… = out[i]; //reads GPU mem directly
```

Asynchronous offloading

Theory:

Converts offload region to OpenMP task

```
#pragma omp target loop nowait
for(int i=0; i<N; ++i) {
  a[i] = b[i] * 5.0;
}

#pragma omp parallel for
for(int i=0; i<N; ++i) {
  c[i] = sin(d[i]);
}

#pragma omp taskwait
```

Background execution of offload task

executed on host

Wait for all outstanding tasks to finish

Asynchronous offloading

- Practice
  - OpenMP runtimes spend a (spinning) thread for managing the offloaded region
  - OpenMP runtimes incur significant overhead for setting up the necessary data structures
  - Offload task is put into a queue and competes for resources like everyone else

- Consequences
  - The task may not even execute asynchronously
  - You probably sacrifice one core per offload region

Different async offloading variants

Variant 1: cross fingers and hope for the best ("free agent threads")

```
#pragma omp target loop nowait
for(int i=0; i<N; ++i) {
  a[i] = b[i] * 5.0;
}

#pragma omp parallel for
for(int i=0; i<N; ++i) {
  c[i] = sin(d[i]);
}

#pragma omp taskwait
```

Variant 2: curb your uneasiness with some extra threads

```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    #pragma omp target loop nowait
    for(int i=0; i<N; ++i) {
      a[i] = b[i] * 5.0;
    }
  }
  #pragma omp for
  for(int i=0; i<N; ++i) {
      c[i] = sin(d[i]);
  }
  #pragma omp taskwait
}
```

Different async offloading variants

Variant 3: explicit tasking for the offloaded region

```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    #pragma omp task
    {
      #pragma omp target loop nowait
      for(int i=0; i<N; ++i) {
        a[i] = b[i] * 5.0;
      }
    }
  }
  #pragma omp for
  for(int i=0; i<N; ++i) {
      c[i] = sin(d[i]);
  }
  #pragma omp taskwait
}
```

Variant 4: tasking all the way for overlap

```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    #pragma omp task
    {
      #pragma omp target loop nowait
      for(int i=0; i<N; ++i) {
        a[i] = b[i] * 5.0;
      }
    }
    #pragma omp taskloop grainsize(1000)
    for(int i=0; i<N; ++i) {
      c[i] = sin(d[i]);
    }
  }
  #pragma omp taskwait
}
```

Different async offloading variants

Variant 5: sections

```
#pragma omp parallel sections num_threads(2)
{
  #pragma omp section
    #pragma omp target loop
    for(int i=0; i<N; ++i) {
      a[i] = b[i] * 5.0;
    }
  #pragma omp section
    #pragma omp parallel for num_threads(n-1)
    for(int i=0; i<N; ++i) {
      c[i] = sin(d[i]);
    }
}
```

Programming models
- MPI + Accelerator

Advantages & main challenges, conclusions

General considerations
OpenMP offloading for accelerators
Case study: Accelerated stencil smoother
> Advantages & main challenges, conclusions

Conclusions from the accelerated J2D example

- OpenMP offloading glosses over many intricacies of the underlying hardware and software
  - It is good to have a performance model at hand
  - See also Performance Analysis with NVIDIA Tools on YouTube
- Data transfers are still the #1 performance limiter
  - Abstractions can easily lead to excessive overhead
- Observing performance behavior when varying parameters is useful
  - Problem size, number of iterations, resources, domain decomposition,…
  - “shake it and see what happens”
- GPU/CUDA-aware MPI can boost scalability
  - However, the future holds shared memory between host and device
    → GPU awareness will be obsolete

MPI+Accelerators: Main advantages

- Hybrid MPI/OpenMP can leverage accelerators and yield performance increase over pure MPI on multicore
- Compiler/pragma-based API provides relatively easy way to use coprocessors
- OpenMP extensions provide flexibility to use a wide range of heterogeneous coprocessors (GPU, APU, heterogeneous many-core types)
- Still some deficiencies in implementations of OpenMP offloading runtimes

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, AMD catching up

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, OpenMP feature support varies

Appendix

Acknowledgments

Abstract

Authors

Acknowledgments

- We want to thank
  - Gabriele Jost, Supersmith, Maximum Performance Software, USA
    - Co-author of several slides and previous tutorials
  - Gerhard Wellein, NHR@FAU
  - 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
  - Michael Klemm, AMD

Abstract

Hybrid Programming in HPC – MPI+X

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

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

Abstract: Most HPC systems are clusters of shared memory nodes and many use accelerators, e.g. GPUs. To use such systems efficiently, both memory consumption and communication time must 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 with and without accelerators. Multi-socket-multi-core systems, with and without accelerators, in highly parallel environments are given special consideration. In addition, we will review the shared memory programming interface introduced in MPI-3.0, which can be combined with inter-node MPI communication.

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: 2006…

2016-HY-VSC	https://asc.ac.at/training/2016/HY-VSC
ISC 2017	https://www.isc-hpc.com/agenda2017/sessiondetails23ac.html?t=session&o=510
2028-HY-VSC	https://asc.ac.at/training/2018/HY-VSC
2019-HY-LRZ	https://www.lrz.de/services/compute/courses/archive/2019/2019-01-28_hhyp1w18/
2020-HY-VSC	http://asc.ac.at/training/2020/HY-VSC
2020-HY-HLRS	http://www.hlrs.de/training/2020/HY-S
2021-HY-VSC	http://asc.ac.at/training/2021/HY-VSC
2022-HY-VSC	http://asc.ac.at/training/2022/HY-VSC
2022-HY-VSC-Dec	https://asc.ac.at/training/2022/HY-VSC-Dec
2022-HY-LRZ	http://www.hlrs.de/training/2022/HY-LRZ
2024-HY-HLRS	https://www.hlrs.de/training/2024/HY-HLRS
2025-HY-HLRS	https://www.hlrs.de/training/2025/HY-HLRS
2026-HY-HLRS	https://www.hlrs.de/training/2026/HY-HLRS

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 Austrian Scientific Computing (ASC), aka ASC Research Center at TU Wien (Vienna, Austria), where she founded and directs the ASC 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.

Claudia is recognized for her expertise in parallel computing and the Message-Passing Interface (MPI), serving as a chapter chair for Terms and Convention in the MPI Forum, which standardizes MPI.

Claudia is deeply engaged in various European projects that advance HPC and AI (AI Factory Austria AI:AT, EuroCC (1), 2, 3, EUMaster4HPC, EVITA), with a particular focus on addressing current needs in training and education.

Tobias Haas

Tobias Haas is a Research Scientist at the High-Performance Computing Center Stuttgart (HLRS). As a key member of the Training and Scalable Algorithms department, he is central to the organization and development of professional training programs designed for the scientific and industrial computing communities. His work focuses on high-level instruction in parallel programming and modern computational methods. This includes expertise in hybrid programming models like MPI+X, as well as the optimization of algorithms for high-performance computing (HPC) and Artificial Intelligence. He represents HLRS within international bodies such as the MPI Forum, where he serves as a chapter chair, contributing to the ongoing evolution of global message-passing standards. Additionally, he is involved in major research and infrastructure projects, including SiVeGCS.

Dr. Haas earned his doctorate in mathematics from the University of Stuttgart in 2019. His research background is rooted in mathematical physics, with a specific focus on the validity of amplitude equations for complex dispersive and dissipative systems. Today, he applies this mathematical rigor to help researchers leverage the power of supercomputers like Hawk and the upcoming Hunter system.

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 (not presenting)

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 he was responsible for the MPI-2.1 version of the standard. Since Dec. 2007, he was in the steering committee of the MPI-3 Forum, until the completion of MPI-4.1 in Nov. 2023. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at the TU Dresden. He was 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.