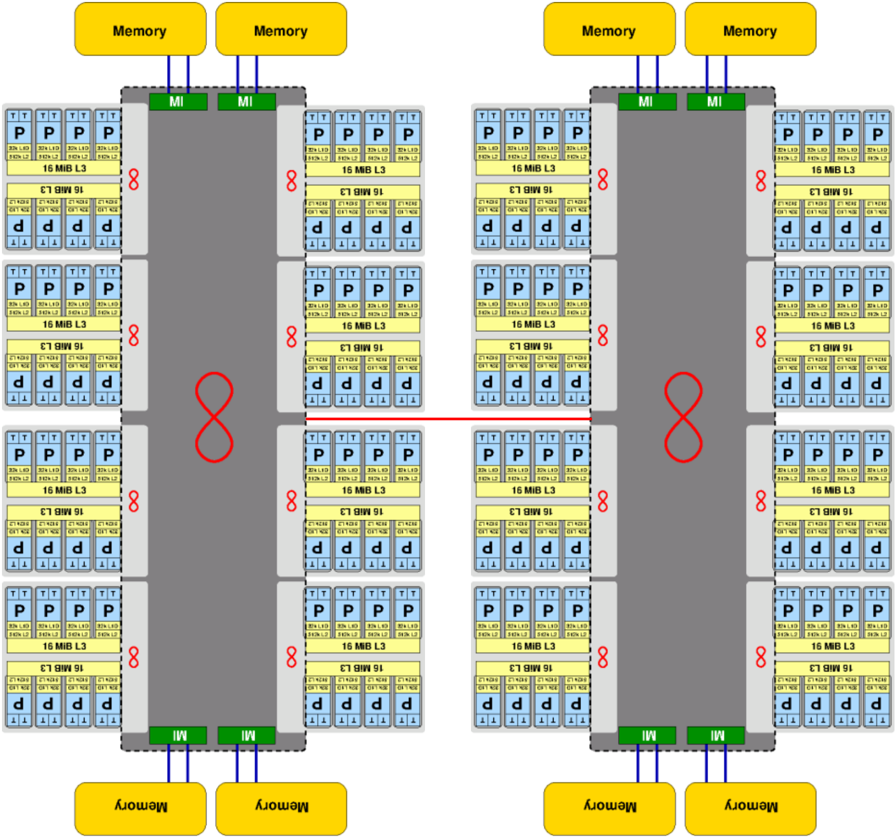
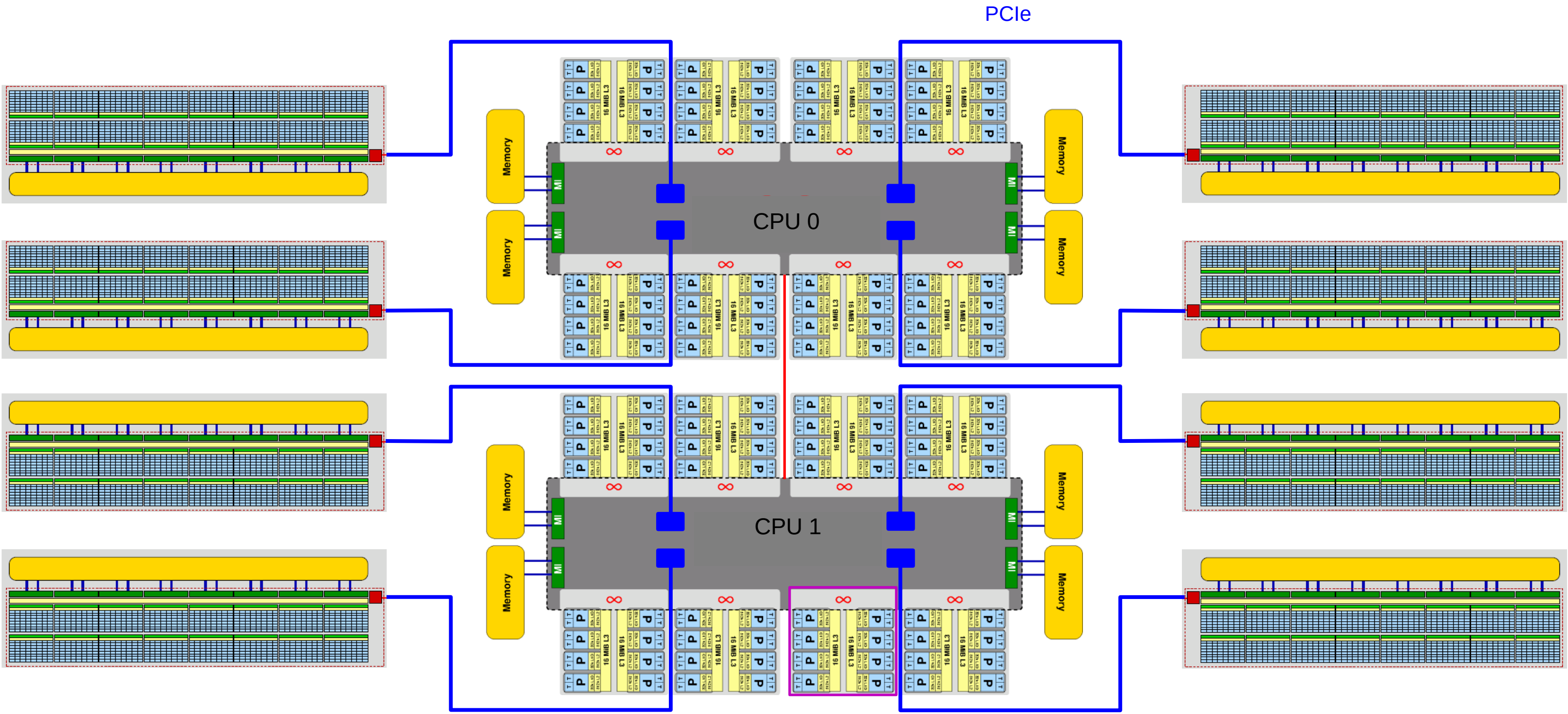
Hybrid MPI+OpenMP programming – an outlook

Modern CPU nodes are strongly hierarchical

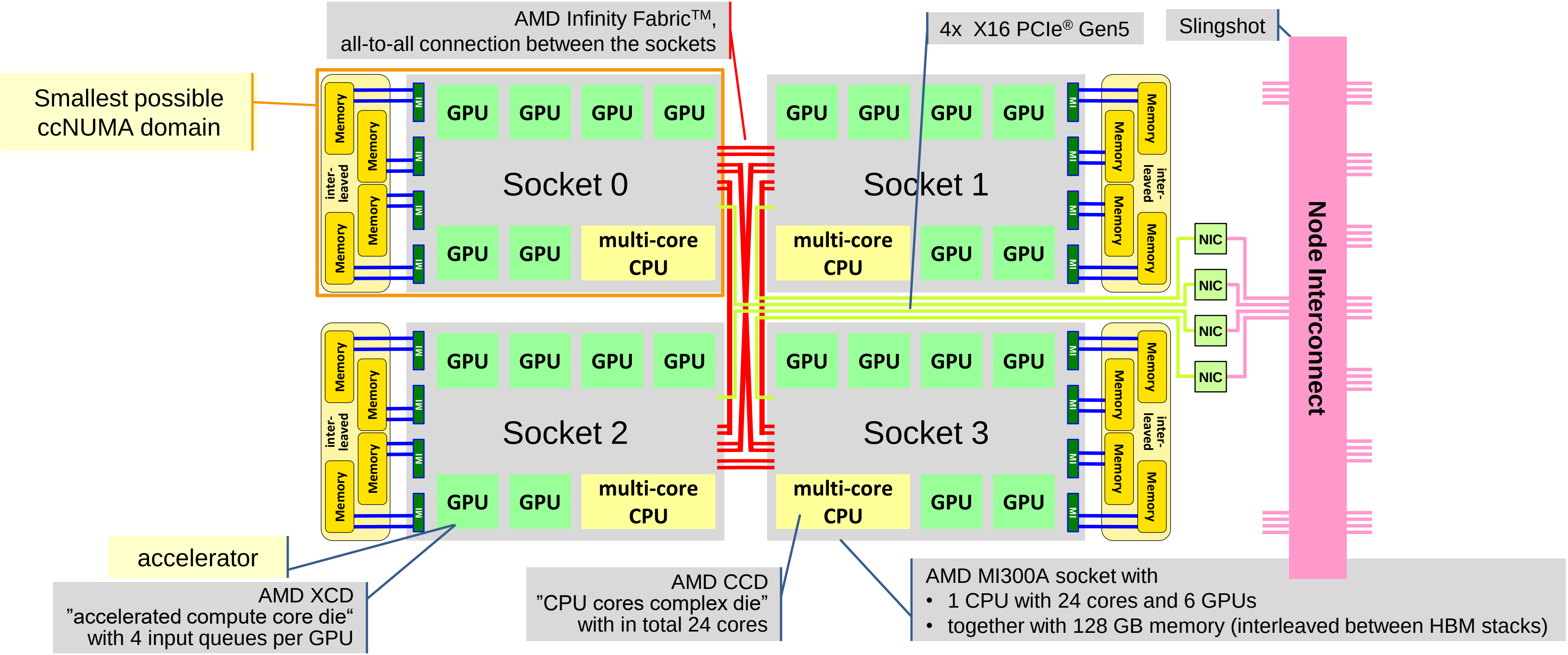


Adding accelerators complicates matters



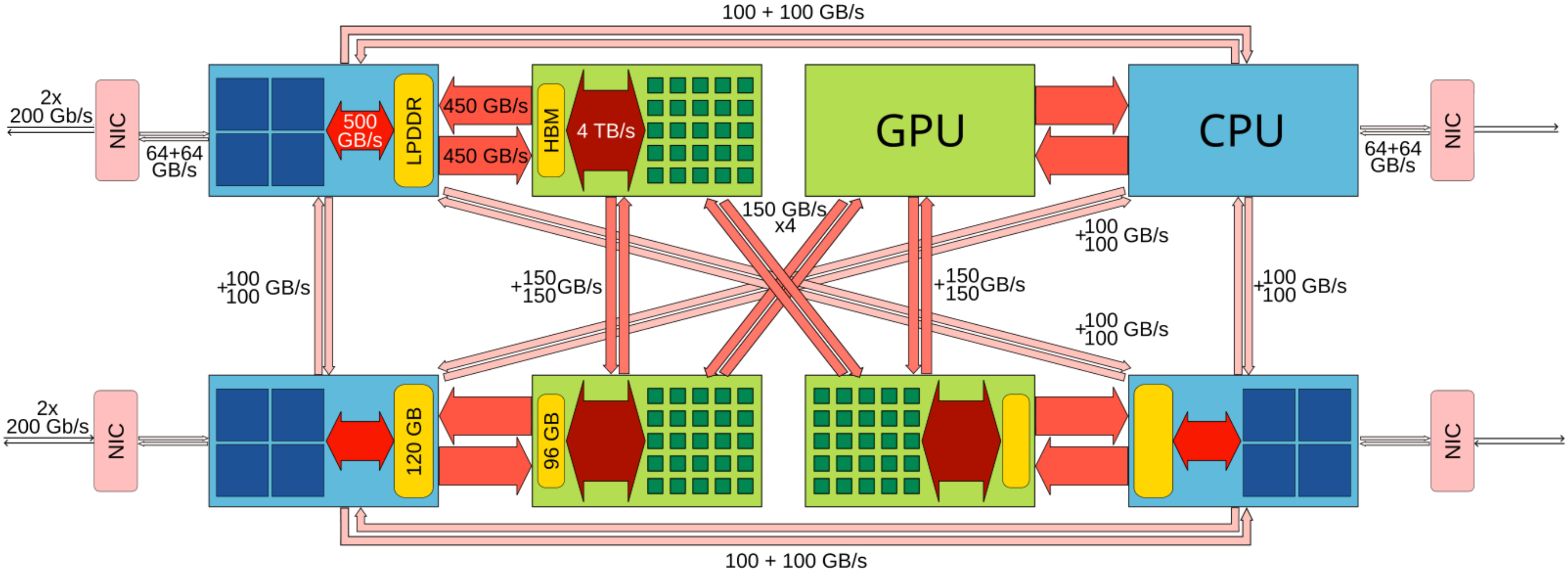
New kids on the block: shared memory accelerators

HLRS "Hunter" (4x MI300A)



New kids on the block: shared memory accelerators

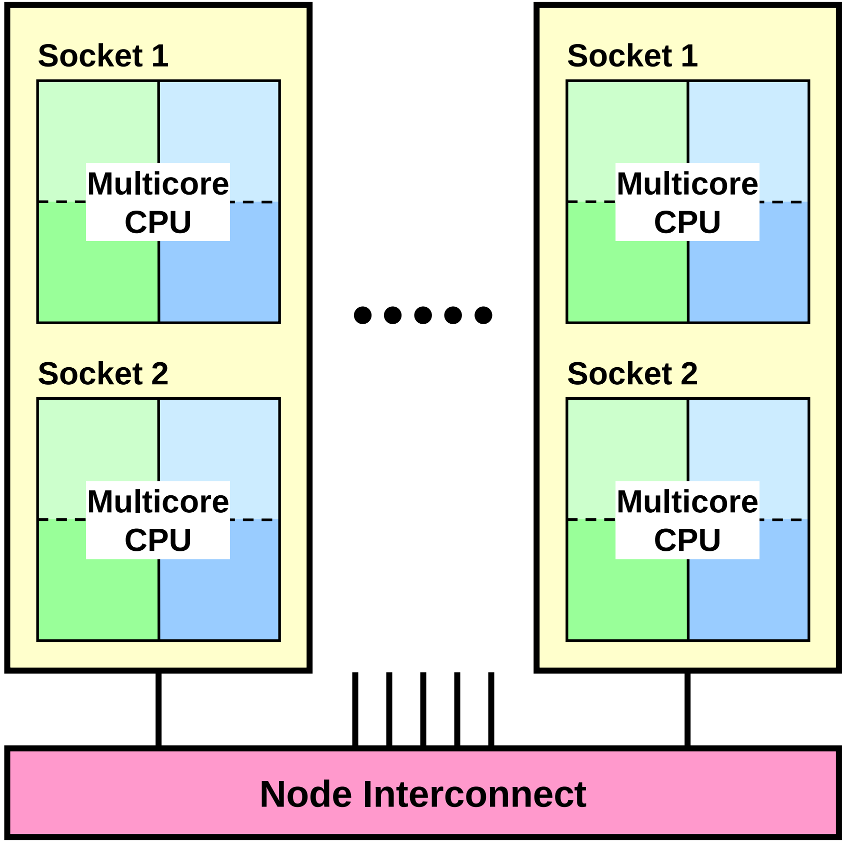
JSC JUPITER Booster (4x NVIDIA Grace-Hopper)



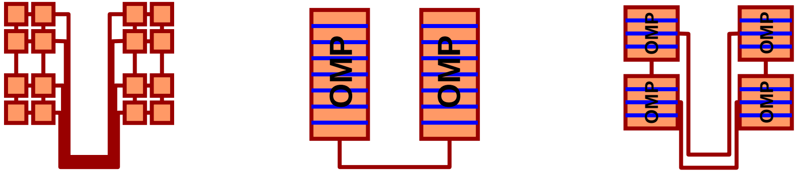
Question

Does it make sense to combine two separate programming models in order to address the hybrid parallelism in modern compute nodes?

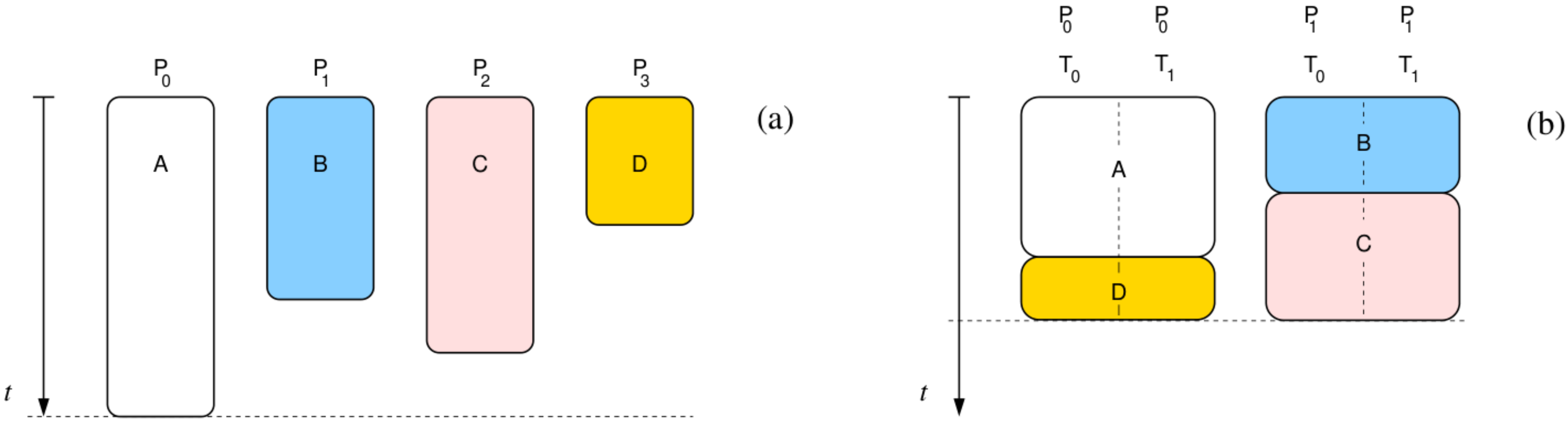
MPI+OpenMP hybrid programming



- Part of the modern cluster topology is accessible to shared-memory parallelization
- OpenMP is the typical choice for that
- Idea: Combine threading on the node level with MPI across nodes
- But how? And are there good arguments to do it at all?
- Lots of choices…

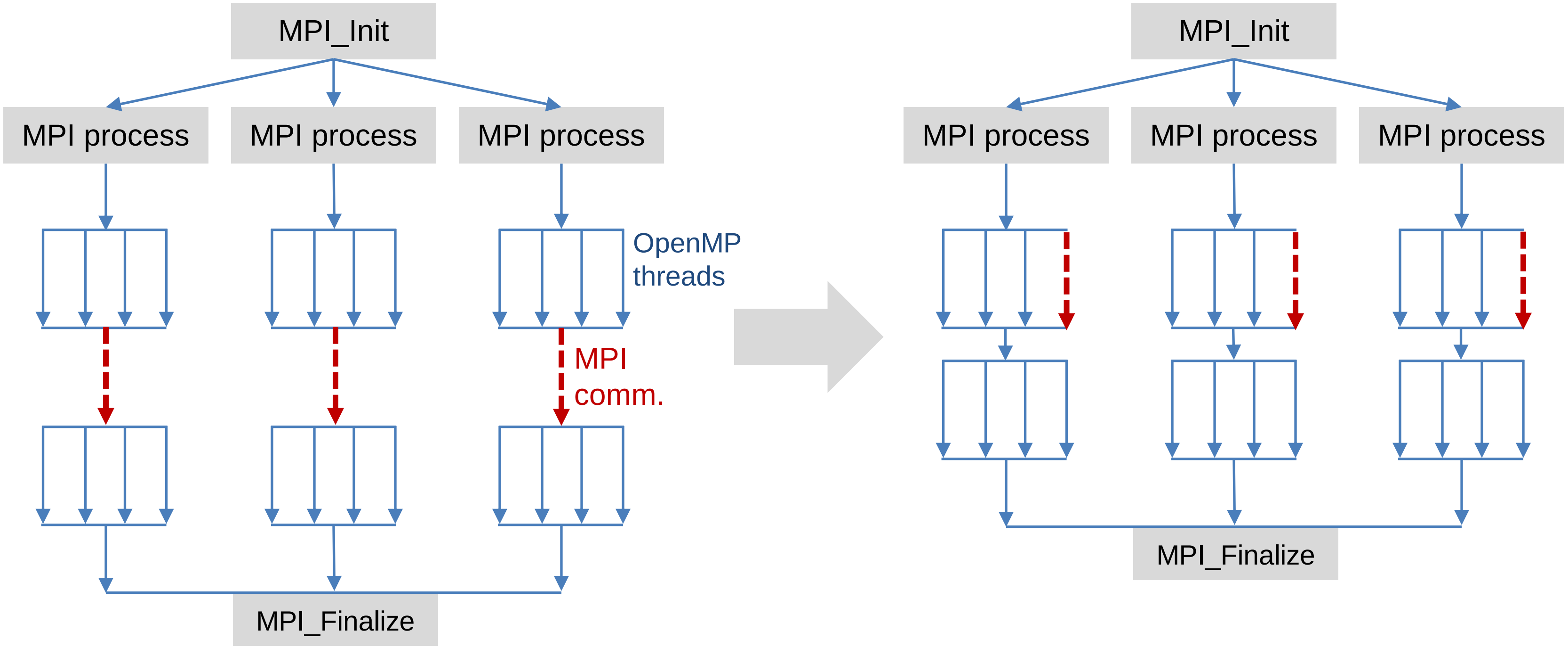


Benefit #1: Improved load balancing



- Making MPI processes “fatter” opens opportunities for better balancing of load
- OpenMP provides inherent load balancing facilities (dynamic scheduling, tasking)

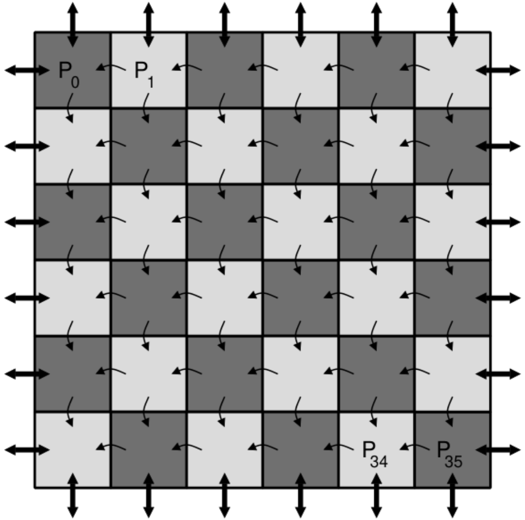
Benefit #2: Overlapping communication



- OpenMP threads can be used to carry out communication in a truly asynchronous manner

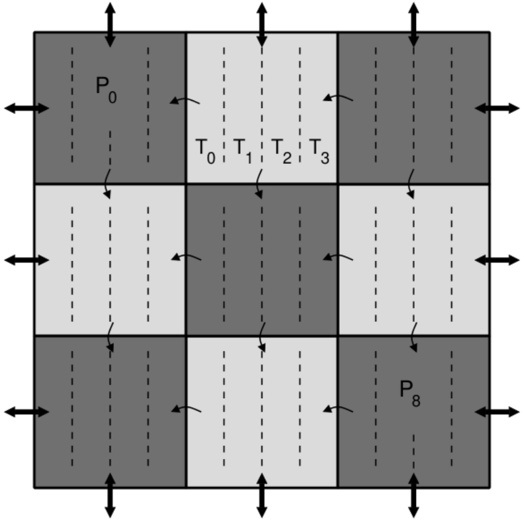
Benefit #3: Reduction of communication overhead

36 processes/node
60 intranode exchanges
24 internode exchanges



(a)

9 processes/node (4 threads each)
12 intranode exchanges
12 internode exchanges



(b)

- Less overall surface area, fewer messages

Other benefits

- Can exploit additional levels of parallelism
  - Parallelize lower-lying loop structures
  - But: OpenMP also incurs overhead
- Might consume less memory due to less replicated data
  - Many MPI programs allocate memory on all processes that is only needed on one
  - Some data might be shared (tables, …)
- Can possibly improve rate of solver convergence
  - MPI processes may break dependencies in solver

    D. Kaushik et al. *Understanding the performance of hybrid MPI/OpenMP programming model for implicit CFD codes*. Proc. Parallel CFD 2009

Enabling thread interoperability in MPI

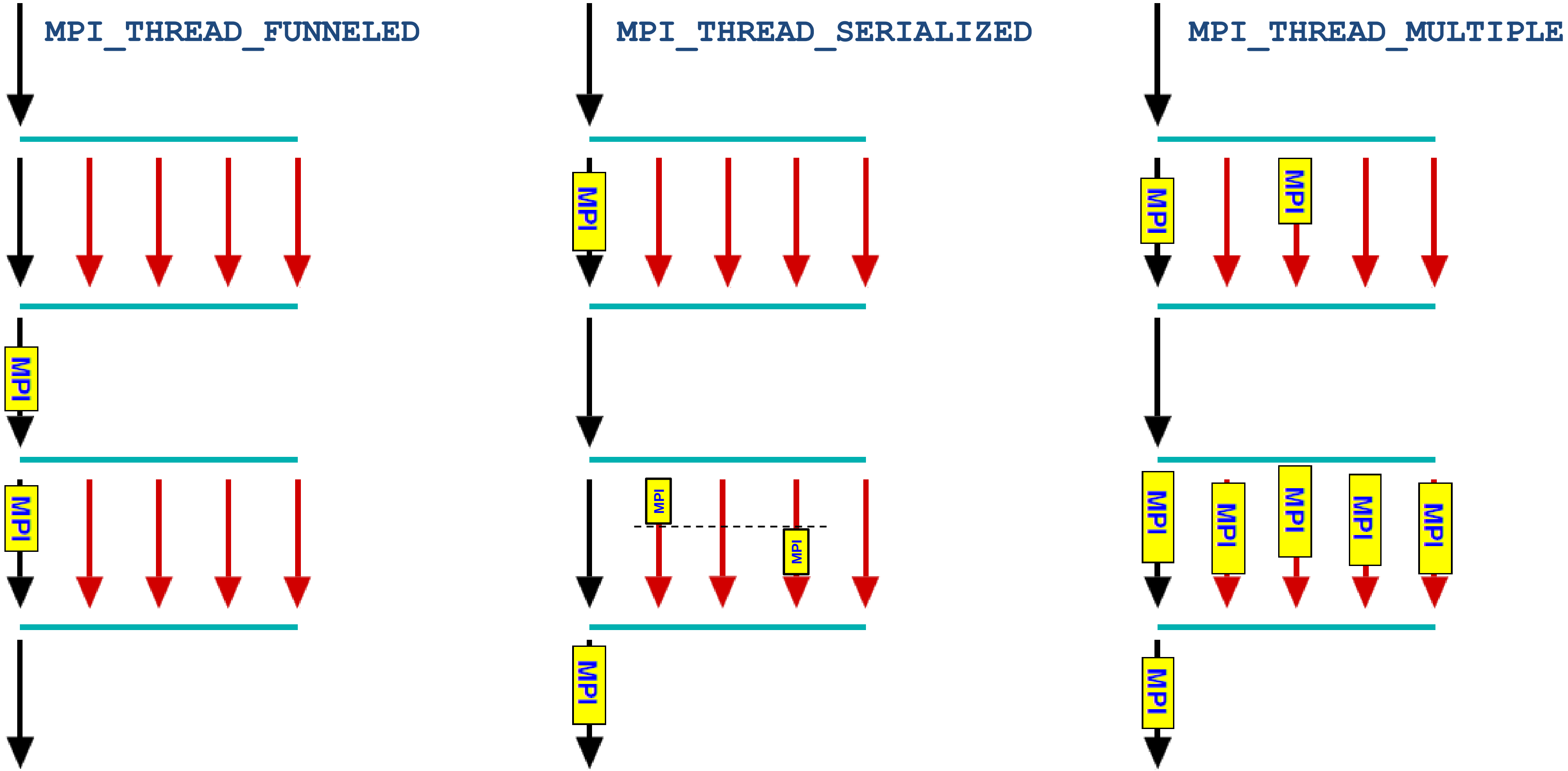
- Use **`MPI_Init_thread()`** instead of **`MPI_Init()`** for initialization

```
int MPI_Init_thread(int * argc, char ** argv[],
                    int thread_level_required,   // input
                    int * thread_level_provided);// output
```

- REQUIRED values (increasing order):
  - **`MPI_THREAD_SINGLE`** Only one thread will execute
  - **`MPI_THREAD_FUNNELED`** Only master thread will make MPI-calls
  - **`MPI_THREAD_SERIALIZED`** Multiple threads may make MPI-calls, but only one at a time
  - **`MPI_THREAD_MULTIPLE`** Multiple threads may call MPI, with no restrictions
- returned **`provided`** may be less or more than **`required`** by the application

Minimum required for *any* threading with MPI

Thread interoperability levels



Compile, link, run

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

Compiling from a single source

Make use of predefined symbols!

```
#ifdef _OPENMP  # _OpenMP defined when OpenMP is active
       // all that is special for OpenMP
#endif

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

 rank = 0;
 size = 1;

#ifdef USE_MPI
       MPI_Init(...);
       MPI_Comm_rank(..., &rank);
       MPI_Comm_size(..., &size);
#endif
```

Compile, link, run

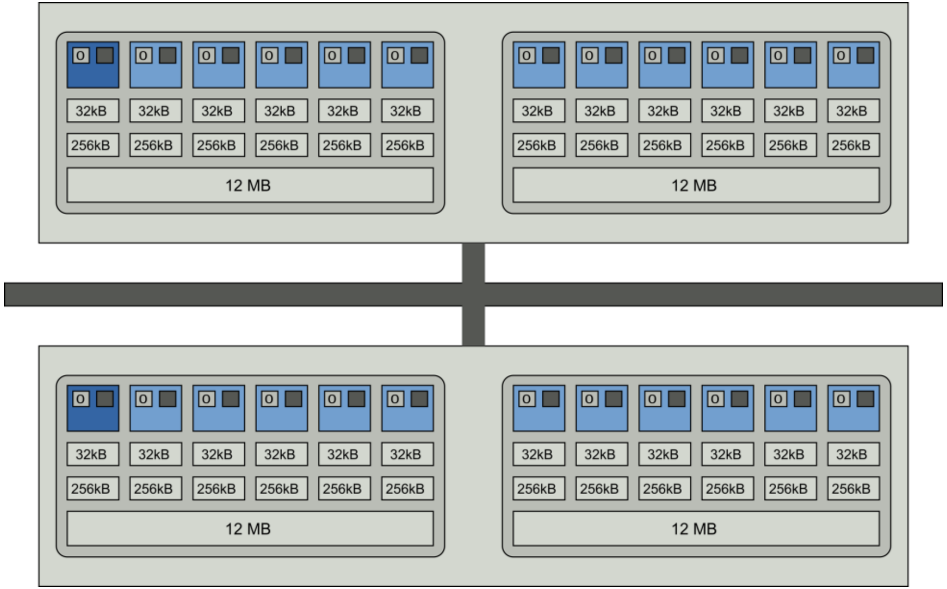
- Examples

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

  - Intel Ivy Bridge (10-core 2-socket) cluster, Intel MPI/OpenMP
    - `mpiifort -qopenmp ...`
    - ```
      OMP_NUM_THREADS=10 mpirun -ppn 2 -np 4 \
           -env I_MPI_PIN_DOMAIN socket \
           -env KMP_AFFINITY scatter ./a.out
      ```

Thread and process binding

- Highly nonportable → many options
- Example: Fully hybrid on dual-socket 6-core cluster



LIKWID:

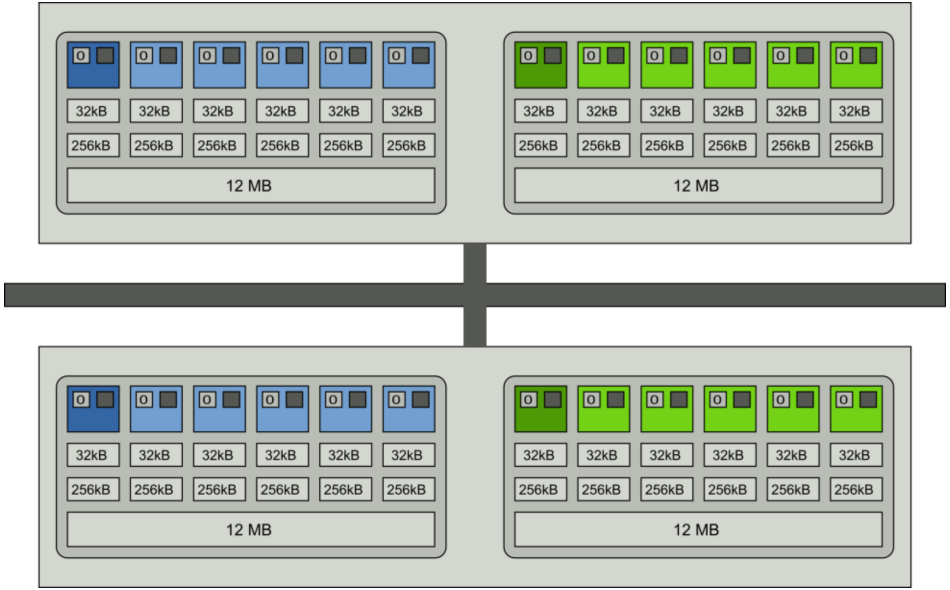
```
likwid-mpirun -np 2 -pin N:0-11  ./a.out
```

Intel MPI+compiler:

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

Thread and process binding

- Example: Mixed mode (1 process with 6 threads per socket) on dual-socket 6-core cluster



LIKWID:

```
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
```

Pure MPI – pros and cons

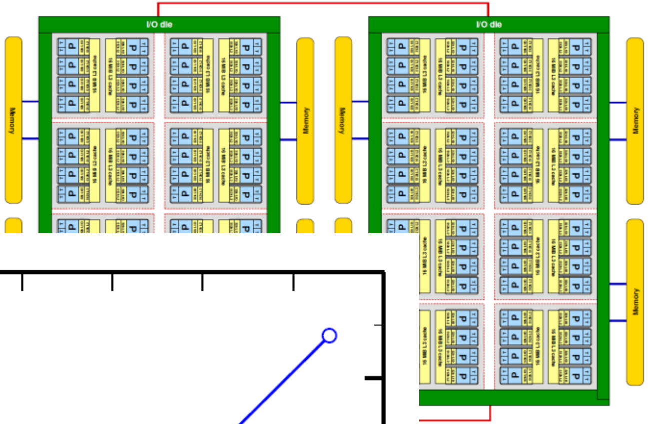
Pros

- Simpler programming, easier affinity enforcement
- May need multiple processes to saturate network bandwidth
- No thread safety concerns
- Only one level of Amdahl's
- Only one bag of overheads
- No (?) ccNUMA page placement problems

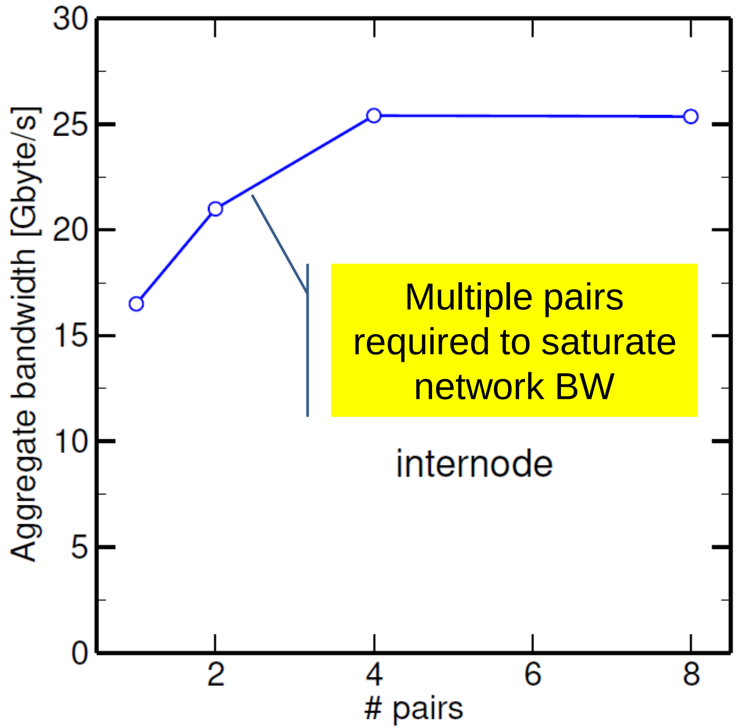
Cons

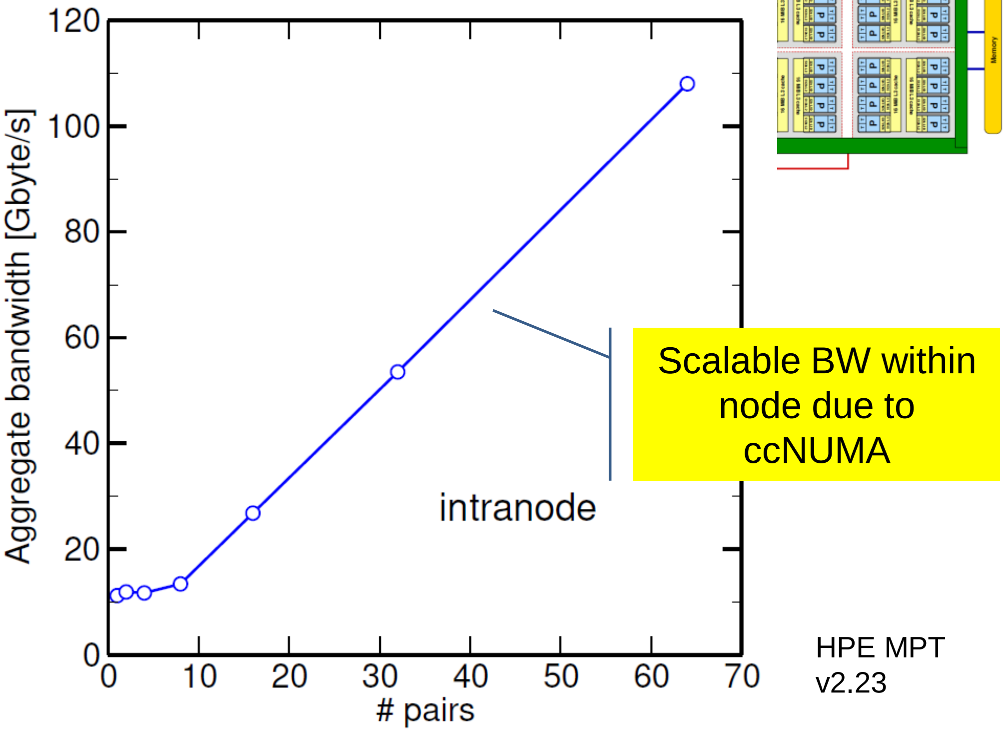
- Hard to exploit multiple levels of parallelism
- Replicated data can get out of hand
- Lots of processes → lots of messages
- Load balancing is difficult
- No guaranteed communication overlap

Effective communication bandwidth saturation



"Multi-mode" Ping-Pong test on Hawk @ HLRS

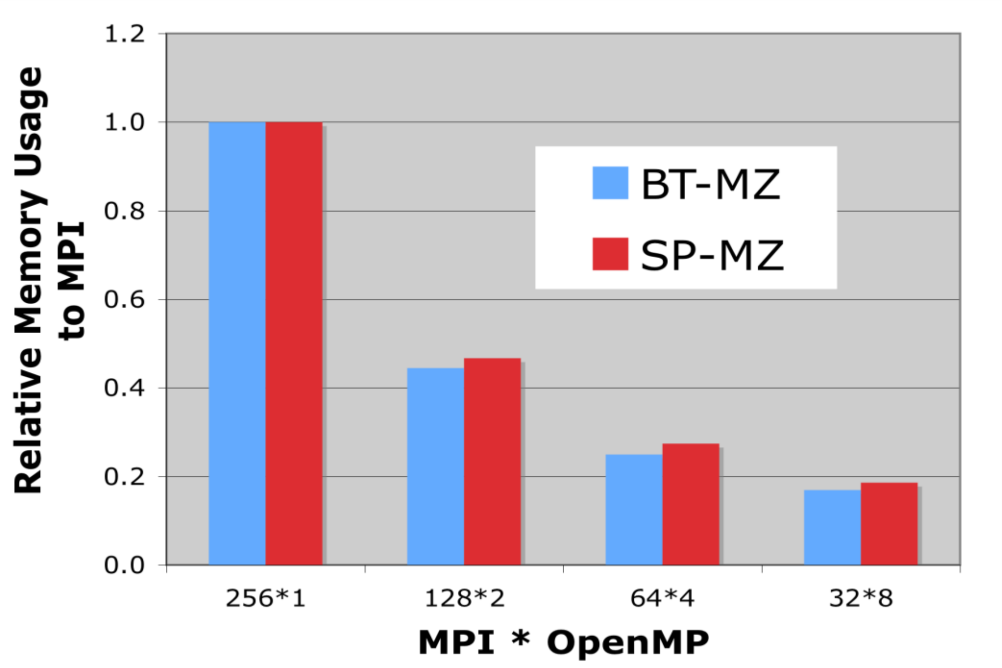




HPE MPT v2.23

Saving memory with hybrid MPI+OpenMP

- Case study: NAS Parallel Benchmarks,
  two variants (BT-MZ, SP-MZ) on Cray XT5
- Massive data replication among MPI ranks
- > 5x memory savings with 8 threads per rank

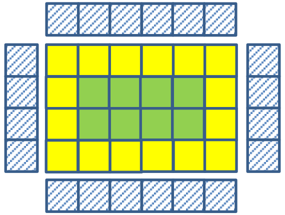


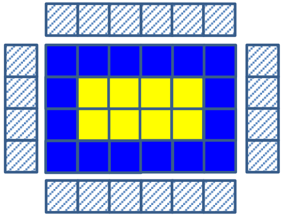
Hongzhang Shan, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, Nicholas J. Wright:
Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms.
Proceedings, CUG 2010, Edinburgh, GB, May 24-27, 2010.

Communication/computation overlap

- Naïve approach: nonblocking MPI calls
- Example: Cartesian domain decomposition with halos

```
for(iterations) {
  MPI_Isend(halo data to neighbors)
  MPI_Irecv(halo data from neighbors)
  for(bulk grid points) {
     update bulk (local domain),
     i.e., all points that do not need the halo
  }
  MPI_Waitall(...)
  for(boundary points) {
     update points that need the halo
  }
}
```



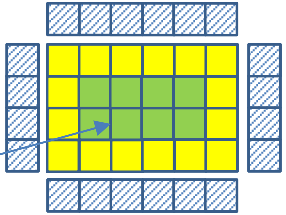


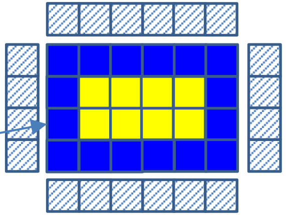
- But: truly asynchronous communication is not guaranteed. It may still happen only in **`MPI_Waitall()`**.

Explicit communication overlap with MPI+OpenMP: the idea

```
if (my_thread_rank < 1) {
   MPI_Send/Recv….
     i.e., communicate all halo data
} else {
   Execute those parts of the application
     that do not need halo data
     (on non-communicating threads)
}

Execute those parts of the application
  that  need halo data
  (on all threads)
```





Explicit communication overlap with MPI+OpenMP

Three problems with standard loop worksharing:

- Application problem: separate application into two parts (“bulk” vs. “boundary”)
  → may be hard to do
- Sub-teams problem: split OpenMP team into communicating & computing sub-teams
  → convenient worksharing directives not applicable
- Load balancing must be done manually

… but is it really so bad?

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

OpenMP taskloop to the rescue?

- `#pragma omp taskloop [clauses]`
  `for-loop`

  breaks loop into chunks and makes them tasks
- Can be combined with “normal” tasks

→ As long as tasking is OK for the “bulk,” this solves at least two of the three problems

→ Issues: ccNUMA placement, overhead

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      communicate(halo);
      compute(boundary);
    }
    #pragma omp taskloop \
            grain_size(100)
    for(<bulk_points>) {
       update_bulk(...);
    }
  }
}
```

Hybrid MPI+OpenMP conclusions

- Do not be fooled by lore and anecdotal evidence
- The benefit of going hybrid (starting from MPI) depends heavily on the particular code

- Main advantages: Explicit communication overlap, “easier” load balancing, less intra-node MPI, fewer messages
- Main challenges: OpenMP overhead, ccNUMA, affinity

- There must be good reasons to embark on a massive refactoring effort

Course

- Hybrid Programming in HPC – MPI+X
  - Georg Hager (NHR@FAU), Tobias Haas (HLRS), Claudia Blaas-Schenner (ASC, TU Wien)
  - Three-day tutorial with hands-on exercises
  - https://tiny.cc/MPIX-HLRS
  - Next course: Beginning of 2027 (?)

HLRS
High-Performance Computing Center | Stuttgart

ASC | Austrian Scientific Computing

NHR FAU