Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

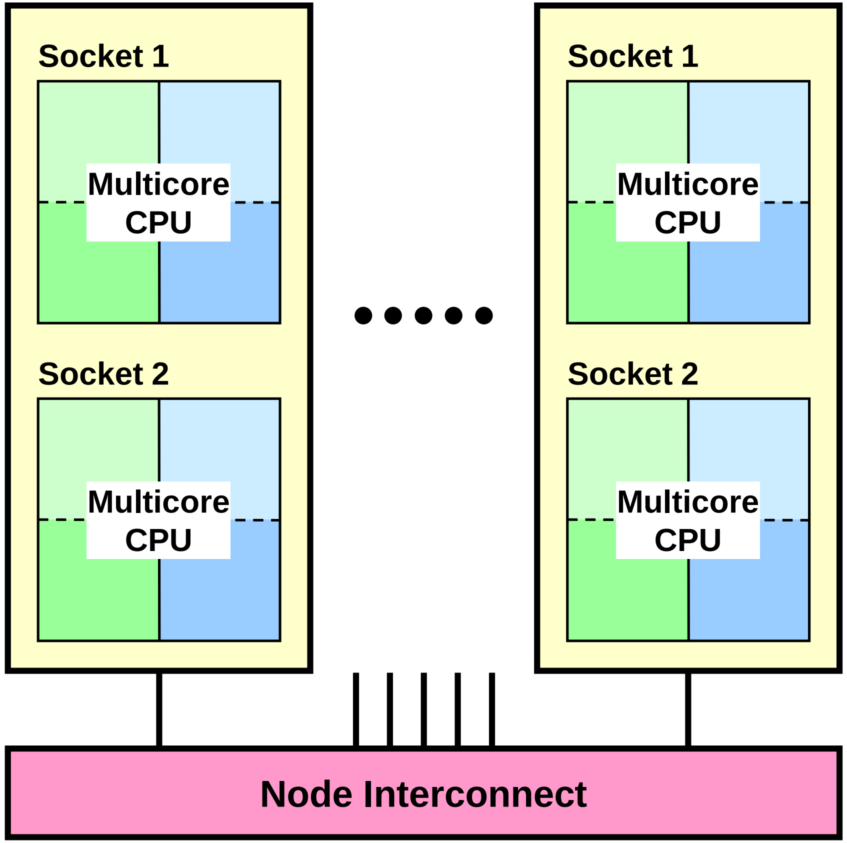
Lecture 13: MPI+OpenMP hybrid programming

(some material by Rolf Rabenseifner, HLRS, and Claudia Blaas-Schenner, TU Wien)

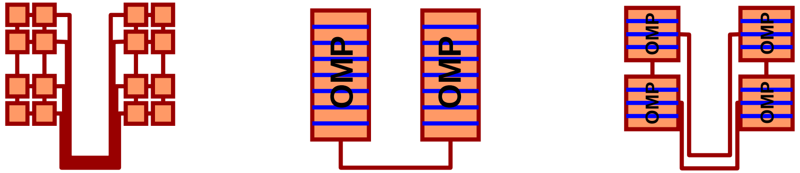
MPI+OpenMP hybrid programming

The basics

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…



Why MPI+OpenMP? – the fiction

- It “fits” the hierarchical structure of modern compute nodes – threading for multicore, MPI for internode communication
  - Not always. OpenMP opens its own can of worms, and you have to know how to deal with it (ccNUMA, overhead, affinity).
- It reduces the communication volume and number of messages
  - Not always. MPI communication can also be optimized in MPI-only programs, and the inter-node communication volume may be the same.
- OpenMP is more lightweight and thus more efficient then MPI on the node level
  - Not generally. This depends entirely on the code. Also, compare a full-node OpenMP barrier with an MPI latency…

Summary: There is no definite answer. It’s complicated.

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
```

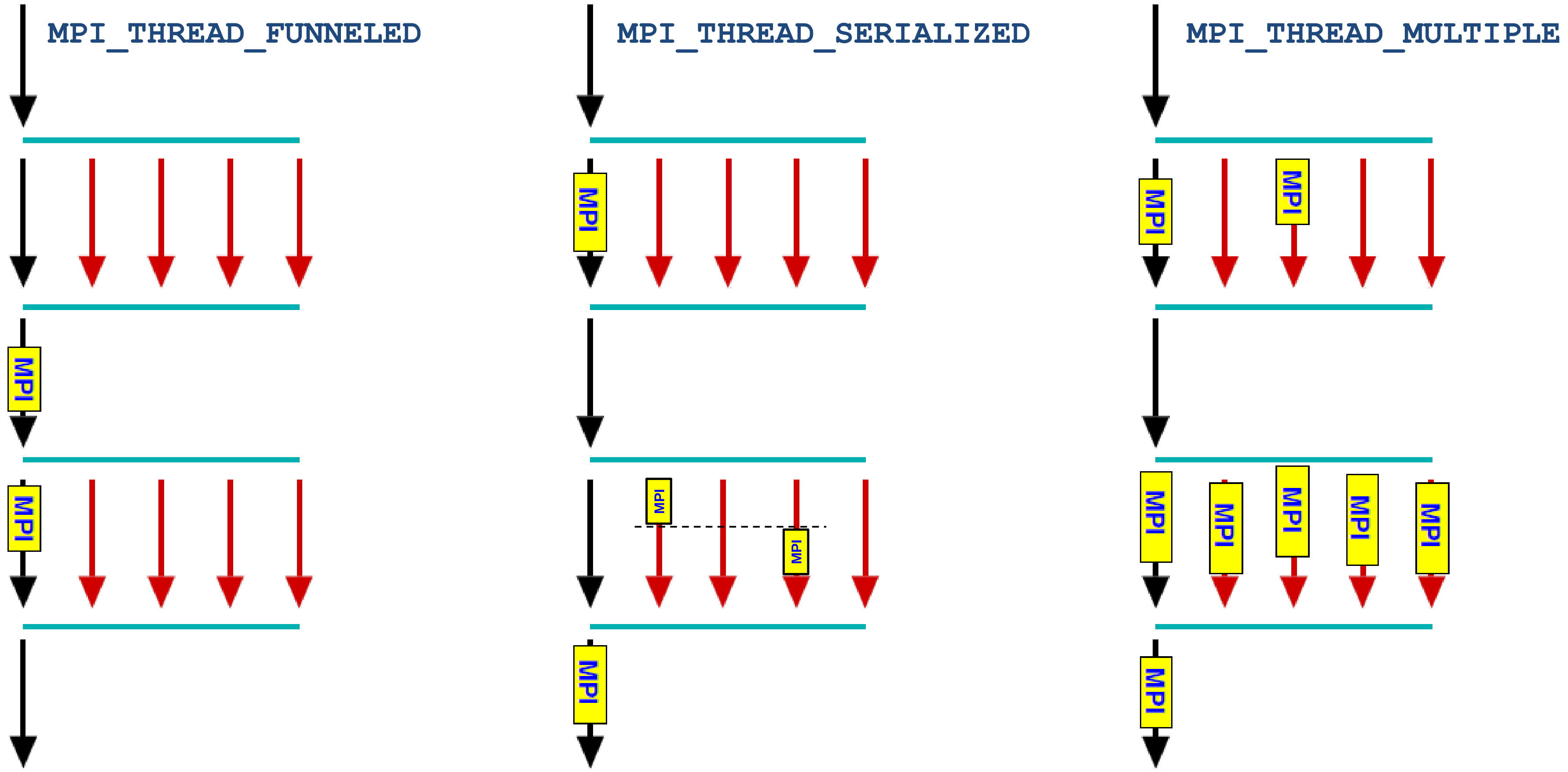
- REQUIRED values (increasing order):

– **`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

Minimum required for *any* threading with MPI

- returned **`provided`** may be less or more than **`required`** by the application

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

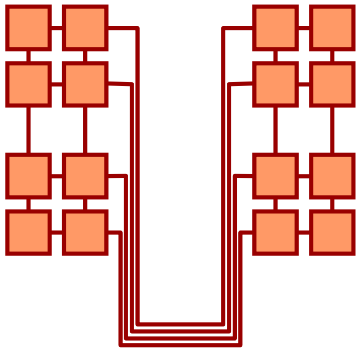
- Examples
  - Cray XC40 (2 NUMA domains w/ 12 cores each):
    - `ftn -h omp ...`
    - `export OMP_NUM_THREADS=12`

    -
      ```
      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
      ```

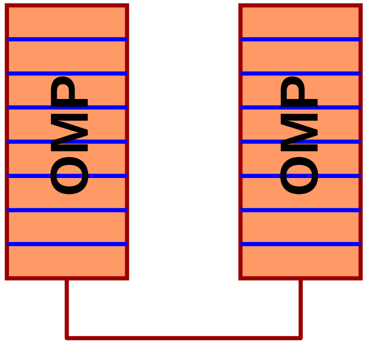
Some nomenclature

Pure MPI



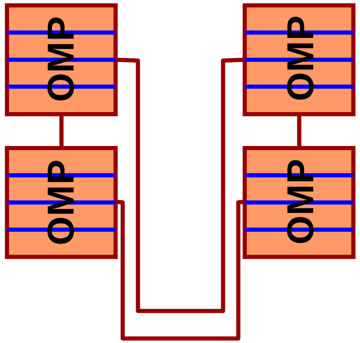
- 1 MPI process per core
- No threading

Fully hybrid



- 1 MPI process per node
- OpenMP only within a node

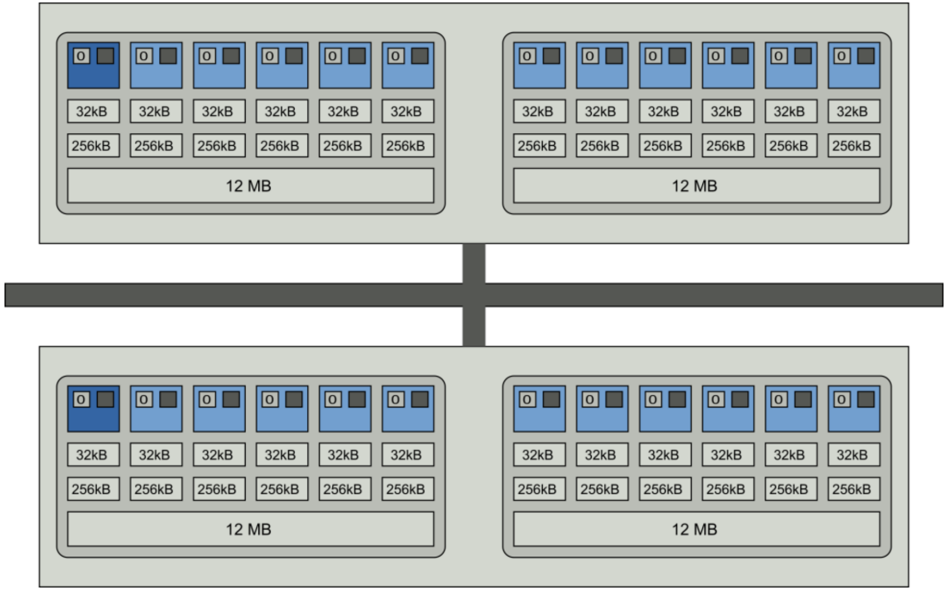
Mixed mode



- >1 MPI processes per node
- >1 OpenMP threads per process

Thread and process binding

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



LIKWID:

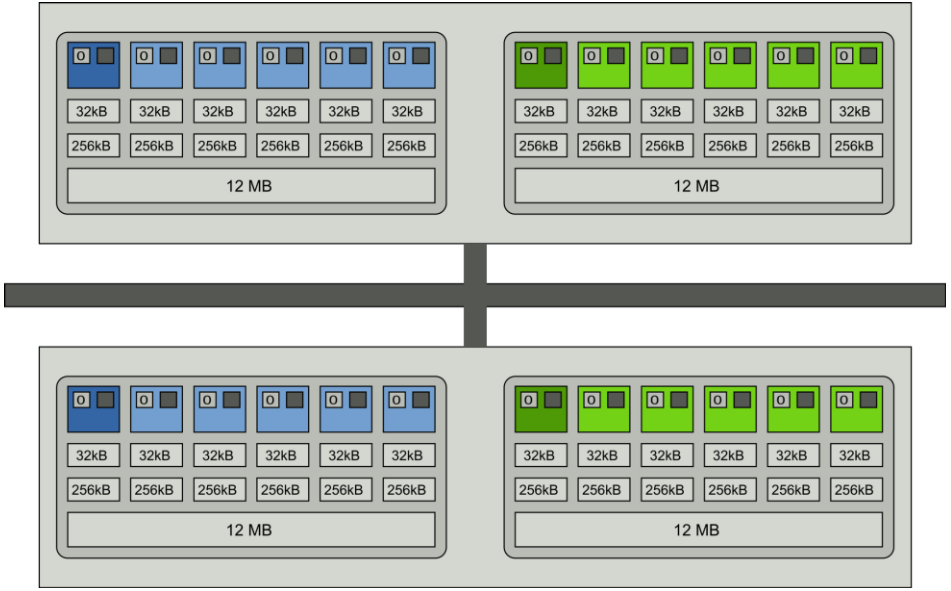
```
likwid-mpirun -np 2 -pin N:0-11  ./a.out
```

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

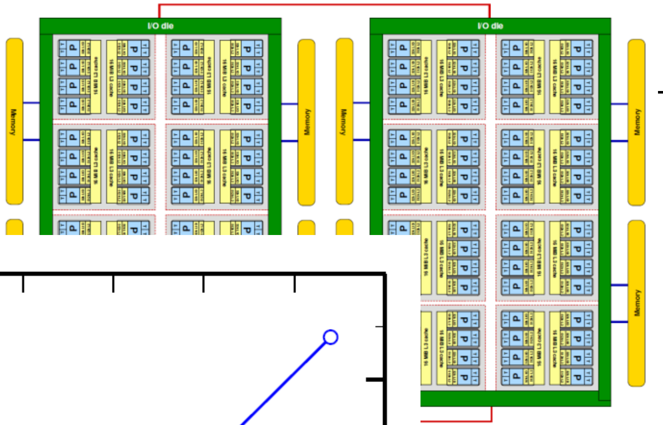
- Simpler programming, easier affinity enforcement
- May need multiple processes to saturate network bandwidth
- No thread safety concerns
- Only one level of Amdahl's
- Only one bag of overheads
- No (?) ccNUMA page placement problems

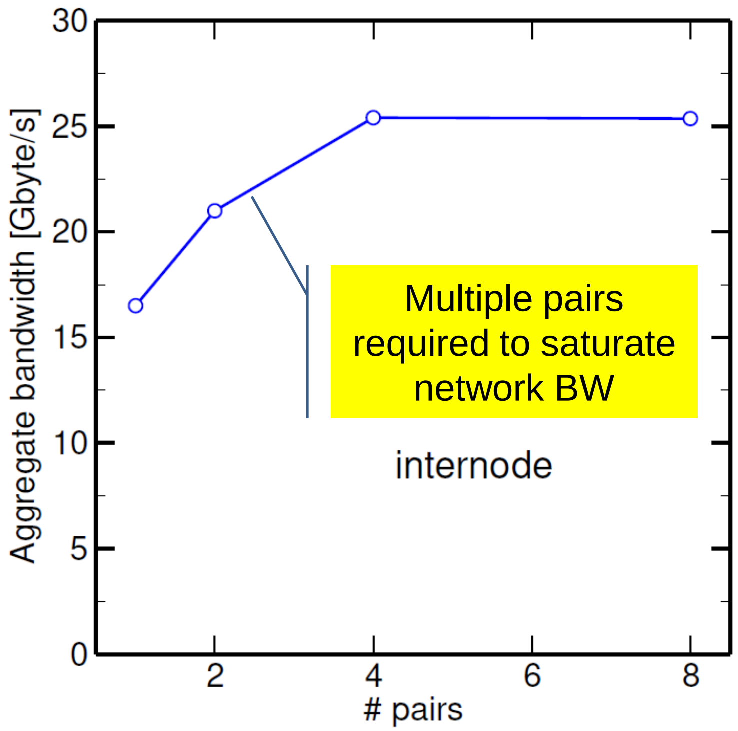
Cons

- 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

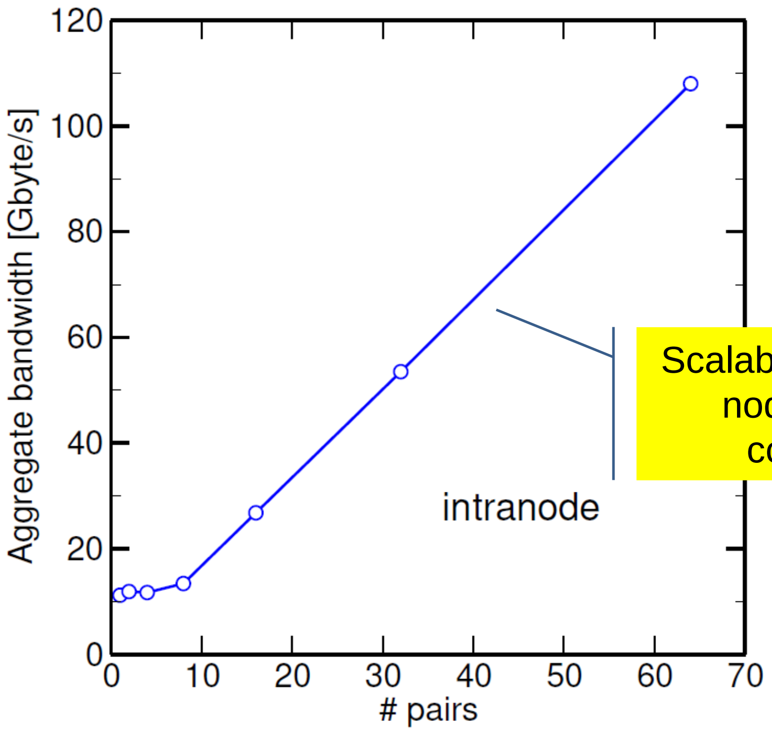
Effective communication bandwidth saturation

“Multi-mode” Ping-Pong test on Hawk @ HLRS





Multiple pairs required to saturate network BW



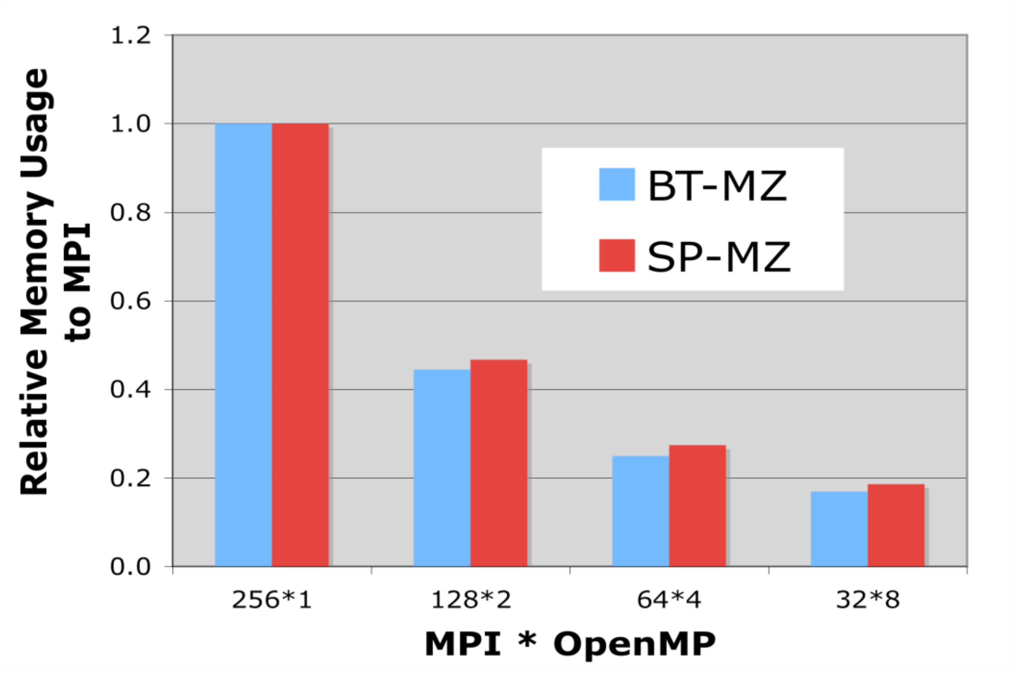
Scalable BW within node due to ccNUMA

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

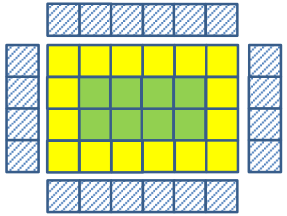


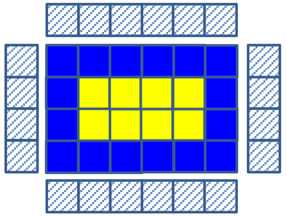
Hongzhang Shan, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, Nicholas J. Wright: *Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms*. Proceedings, CUG 2010, Edinburgh, GB, May 24-27, 2010.

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
  }
}
```

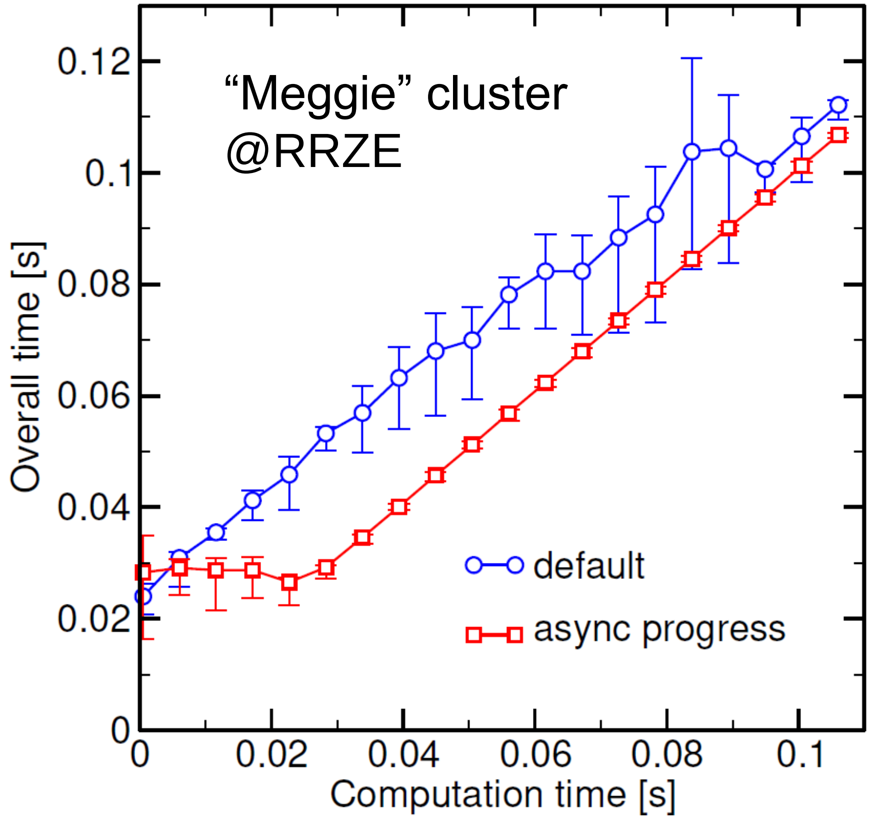




Communication overlap: the problem

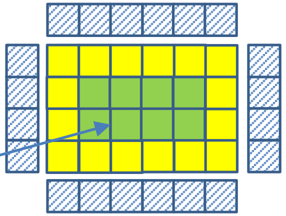
- Remember the “non-blocking MPI overlap benchmark”?
- Asynchronous communication is not guaranteed by non-blocking MPI

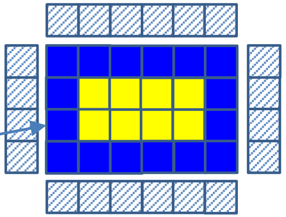
→ Hybrid MPI+OpenMP provides a solution



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

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Sparse matrix-vector multiplication

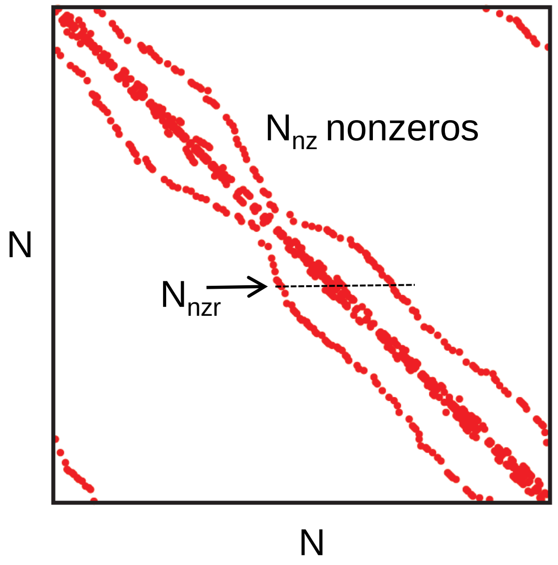
A case study for hybrid programming with MPI and OpenMP

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, Preprint: arXiv:1106.5908

High Performance Computing

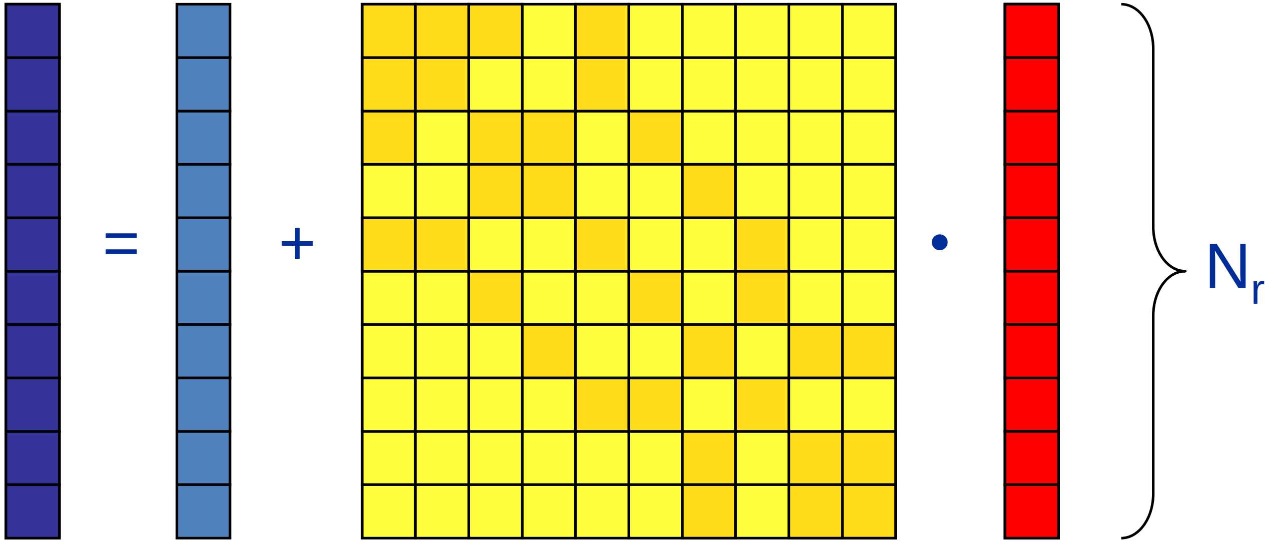
Sparse matrices

- "Sparse" matrix ≅ "N_{nz} grows slower than quadratically with N"
  - N_{nzr} = avg. # nonzeros per row
- A different sparsity pattern ("fingerprint") for each problem
  - Even changes with different numbering of DoFs
- Performance of spMVM c = A·b
  - Always memory-bound for large N_{nz}
  - Usage of memory BW divided between nonzeros and RHS/LHS vectors
  - Sparsity pattern has strong impact
  - Storage format, too
- Storage formats
  - Compressed Row Storage (CRS): Best for modern cache-based µP
  - Jagged Diagonals Storage (JDS): Best for vector(-like) architectures
  - Special formats exploit specific matrix properties



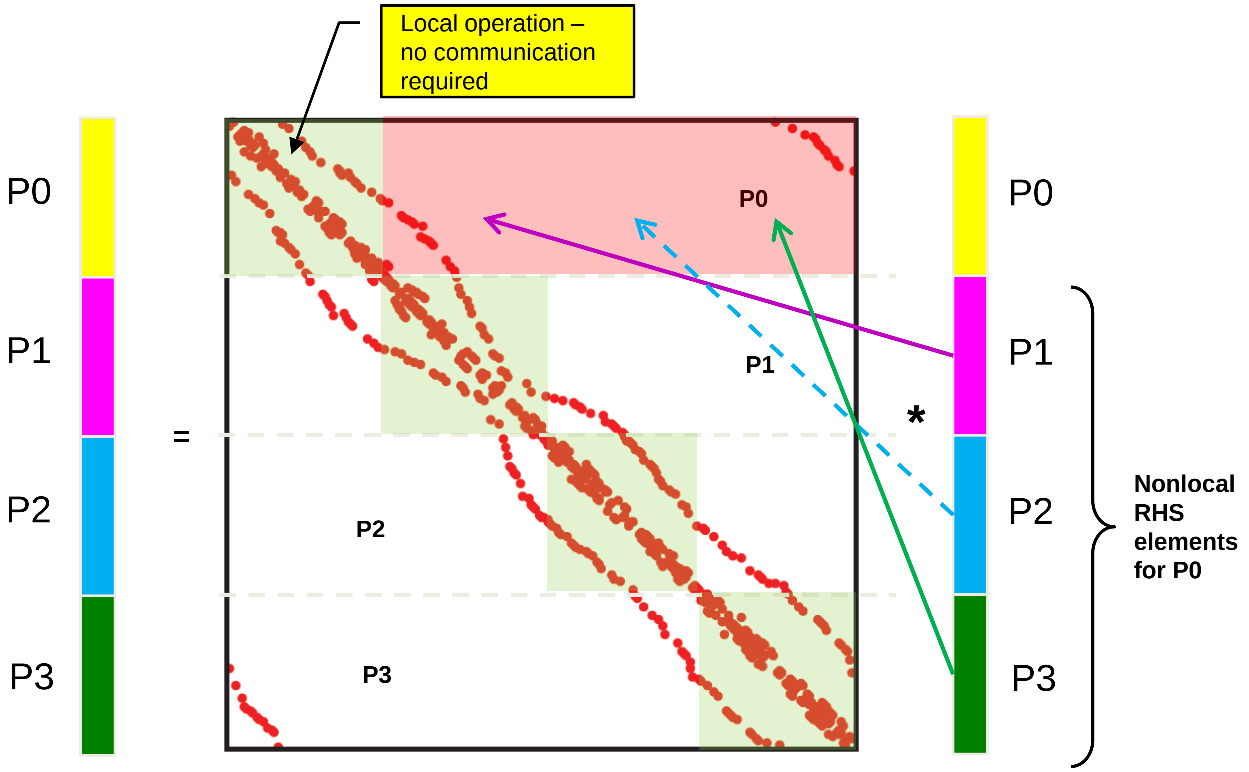
Sparse MVM

- Key ingredient in many algorithms
  - Eigenvalue solvers: Lanczos, Davidson, Jacobi-Davidson
  - Sparse linear systems solvers: Jacobi, GS, CG, and derivatives



General case: some indirect addressing required!

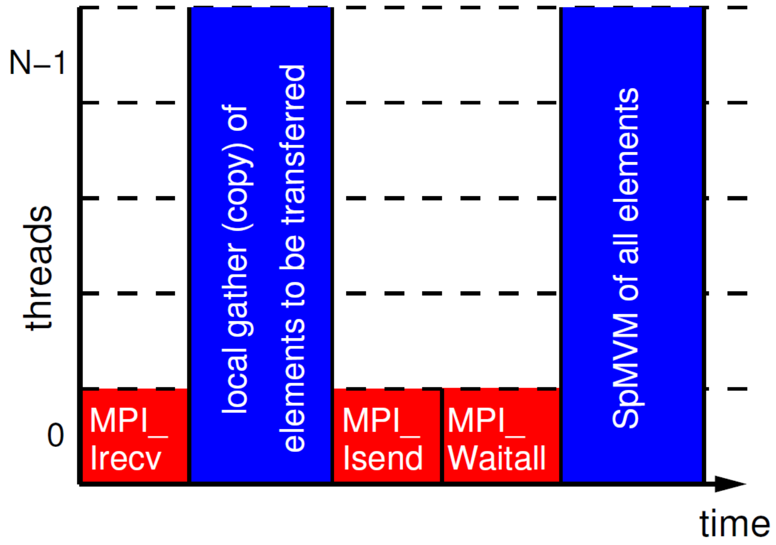
Distributed-memory sparse MVM


Local operation – no communication required
P0
P1
P2
P3
=
P0
P1
P2
P3
*
P0
P1
P2
P3
Nonlocal RHS elements for P0

SpMVM with MPI, variant 1

- “Vector mode” without overlap
- Multithreaded computation (all threads)
- Masteronly style; MPI communication only outside of computation

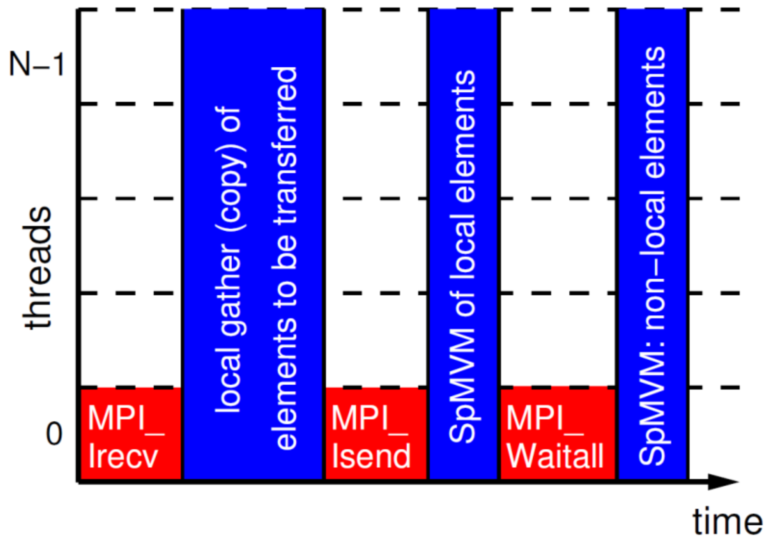
- Benefit of threaded MPI process only due to message aggregation and (probably) better load balancing



SpMVM with MPI, variant 2

- "Vector mode" with naïve overlap ("good faith hybrid")
- Relies on MPI to support async nonblocking PtP
- Multithreaded computation (all threads)
- Still simple programming

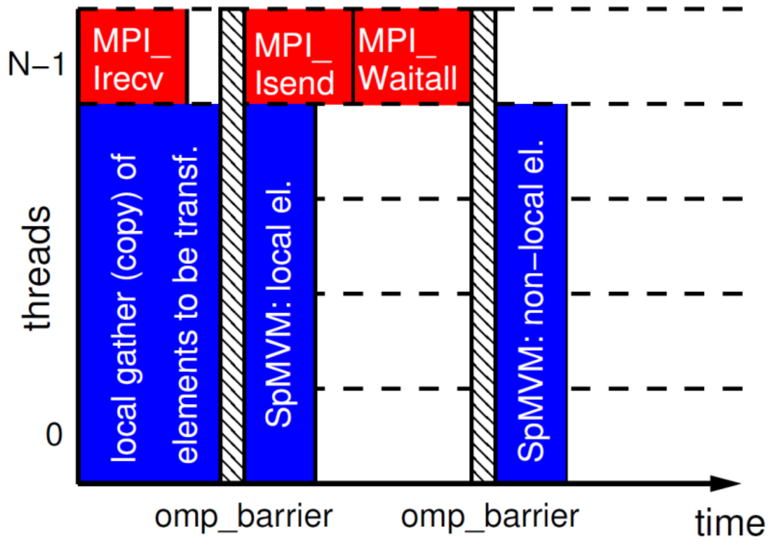
- Drawback: Result vector is written twice to memory
  - modified performance model


N–1
0
threads
MPI_ Irecv
local gather (copy) of elements to be transferred
MPI_ Isend
SpMVM of local elements
MPI_ Waitall
SpMVM: non–local elements
time

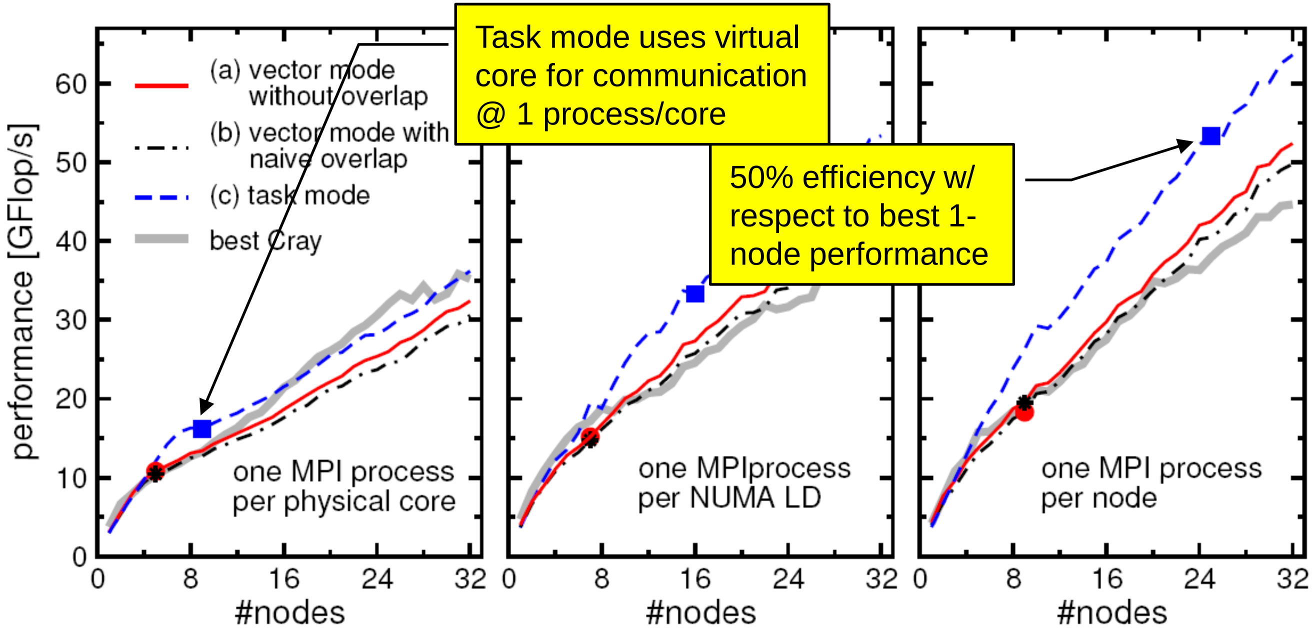
SpMVM with MPI, variant 3

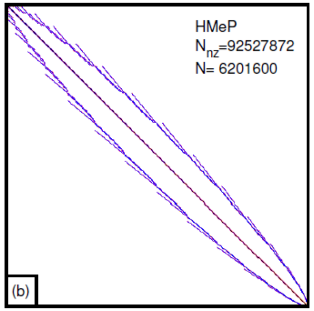
- "Task mode" with dedicated communication thread
- Explicit overlap, more complex to implement
- One thread missing in team of compute threads

- Drawbacks
  - Result vector is written twice to memory
  - No simple OpenMP worksharing; must revert to manual or tasking solutions



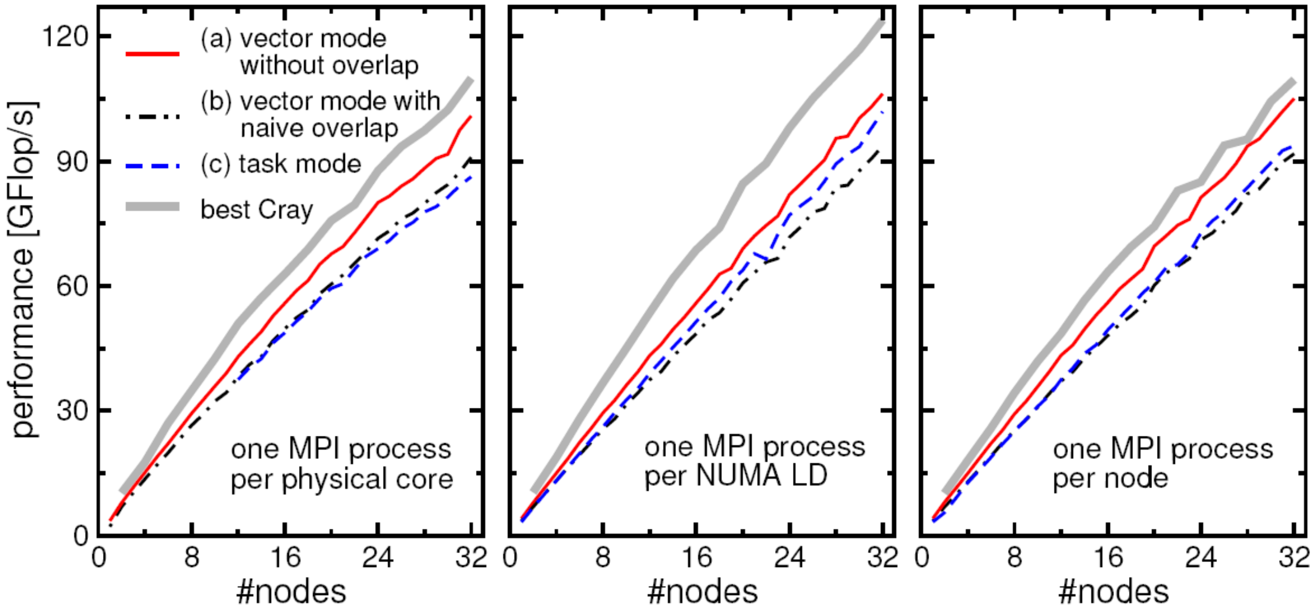
Results for HMeP matrix

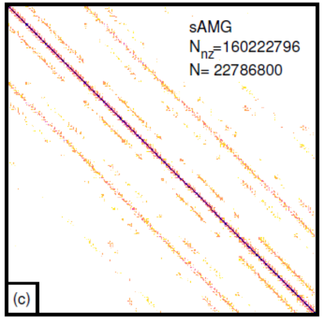




- Dual-socket 6-core cluster vs. Cray XE6
- Dominated by communication (and some load imbalance for large #procs)
- Task mode pays off esp. with one process (12 threads) per node
- Task mode overlap (over-)compensates additional LHS traffic

Results for sAMG matrix





- Much less communication-bound
- # of threads per process makes hardly any difference
- If pure MPI is good enough, don’t bother going hybrid!

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
- Main challenges: OpenMP overhead, ccNUMA

- If possible, use a performance model to check whether your MPI implementation is “good enough”