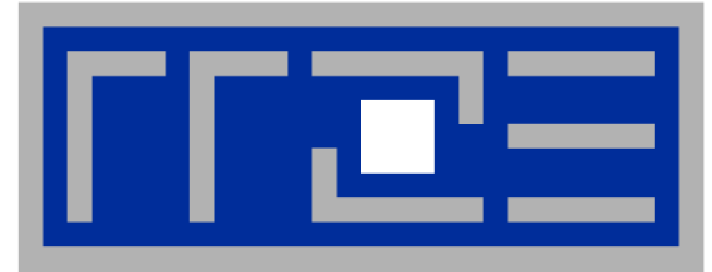
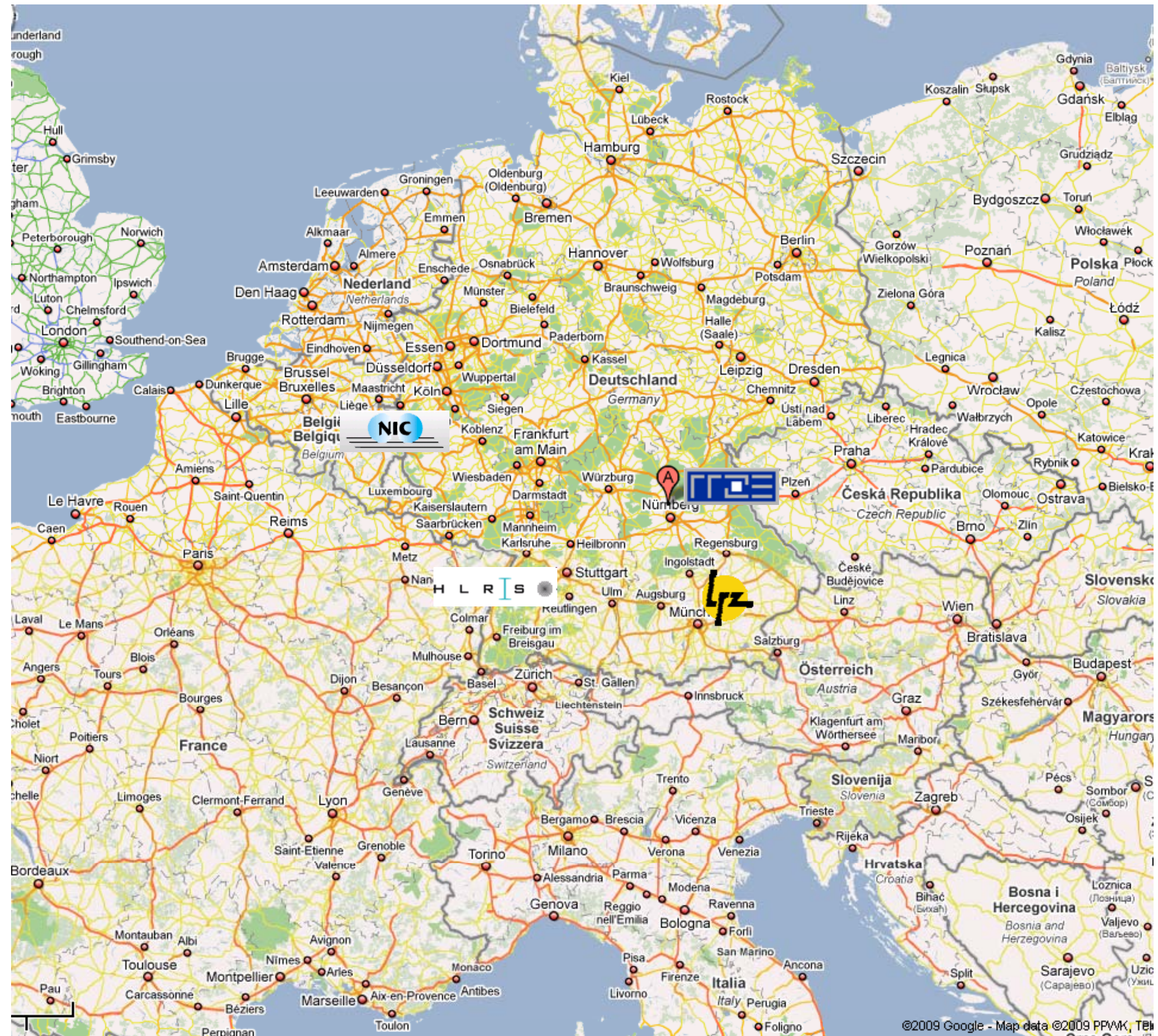


MPI+OpenMP hybrid computing (on modern multicore systems)

Georg Hager
Holger Stengel
Gerhard Wellein
Jan Treibig
Markus Wittmann
Michael Meier
Erlangen Regional Computing Center (RRZE), Germany

39th Speedup Workshop on High-Performance Computing
ETH Zürich, September 6-7, 2010

- **RRZE = Erlangen Regional Computing Center**
  - ≈ 100 employees and students,
    10 in HPC Services
  - 14 (+60) TFlop/s in clusters & some "hot silicon"
  - "IT Service Provider for FAU"
- **FAU = Friedrich-Alexander University of Erlangen-Nuremberg**
  - Second largest university in Bavaria
  - 26000 students
  - 12000 employees
  - 550 professors
  - 260 chairs



Statement

Common lore:

"An OpenMP+MPI hybrid code is never faster than a pure MPI code on the same hybrid hardware, except for obvious cases"

Our statement:

"You have to compare apples to apples, i.e. the best hybrid code to the best pure MPI code"

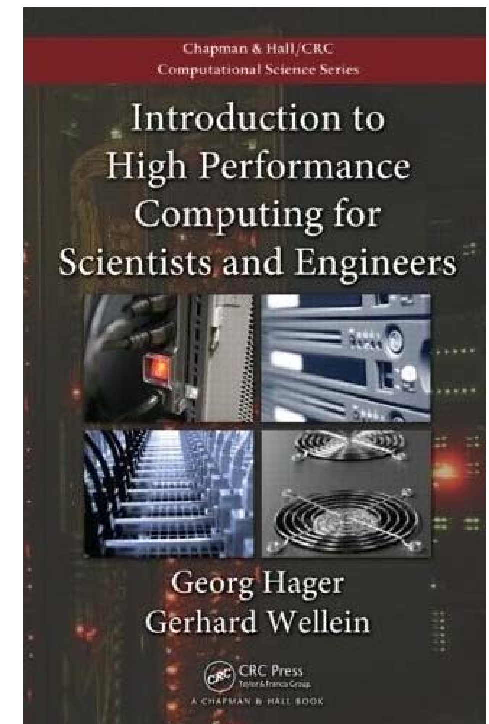
Needless to say, both may require significant optimization effort.

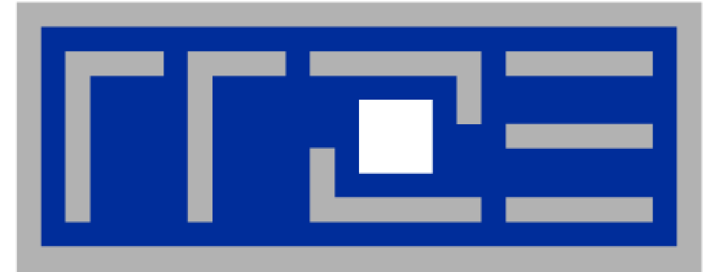
And remember: Using pure MPI on a current cluster must be called "hybrid computing" as well!

Outline

- **Hybrid programming benefits and taxonomy**
  - Vector mode, task mode
  - Topology awareness and thread-core mapping
- **“Best possible” MPI code**
  - Rank-subdomain mapping
  - Overlapping computation and communication via non-blocking MPI?
  - Overlapping cross-node and intra-node communication
  - Understanding intra-node MPI behavior
- **“Best possible” OpenMP code**
  - Synchronization overhead
  - ccNUMA page placement
- **“Best possible” MPI+OpenMP hybrid code**
  - True comm/calc overlap via hybrid task mode
  - ccNUMA and task mode
  - Hybrid parallel temporal blocking

To see more, visit one of the multicore/hybrid tutorials at SC10, PPoPP11, PARENG11





Hybrid taxonomy and possible benefits

Taxonomy of hybrid “modes”:
Several OpenMP threads per MPI process

Vector mode: MPI is called only outside OpenMP parallel regions. This is what many people mean when they say “hybrid”

- Similar to what we did on vector-parallel machines

Task mode: One or more threads in the parallel region are dedicated to special tasks, like doing communication in the background

- This is functional parallelism on the thread level

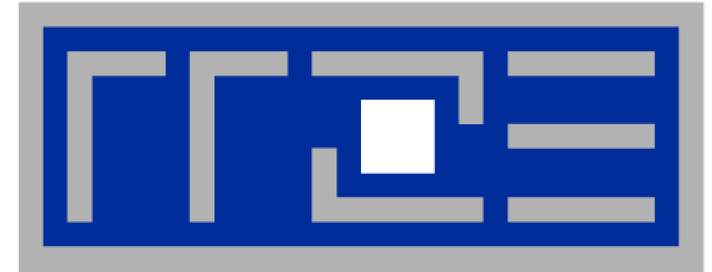
R. Rabenseifner and G. Wellein, *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures*. Int. J. High Perf. Comp. Appl. 17(1), 49-62 (2003)

Possible hybrid benefits

	Vector mode	Task mode
Improved/easier load balancing	✔	✔
Additional levels of parallelism	✔	✔
Reliable overlapping of communication and computation	✘	✔
Improved rate of convergence	✔	✔
Re-use of data in shared caches	✔	✔
Reduced MPI overhead	✔	✔

Possible hybrid drawbacks

	Vector mode	Task mode
OpenMP overheads	✓	✓
Node-level bulk-synchronous communication	✓	(✓)
Possible deficiencies in code optimization by compiler	✓	✓
ccNUMA placement problems	✓	✓ ✓
Nonability to saturate network interface	✓	(✓)
Complexities in thread/core affinity	✓	✓ ✓



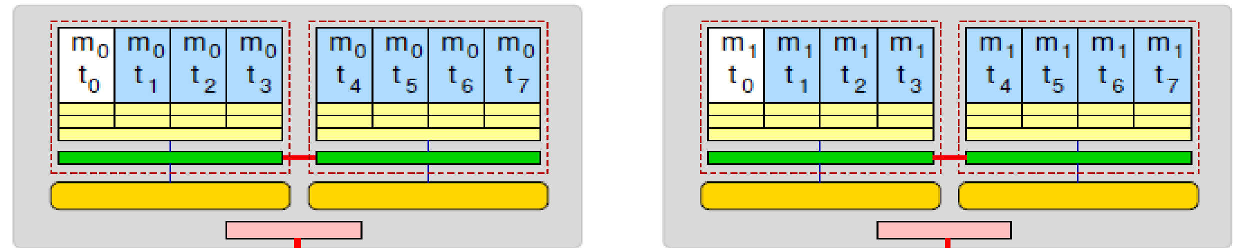
Hybrid mapping choices on current hardware

Choices for running programs on multicore/multisocket hardware

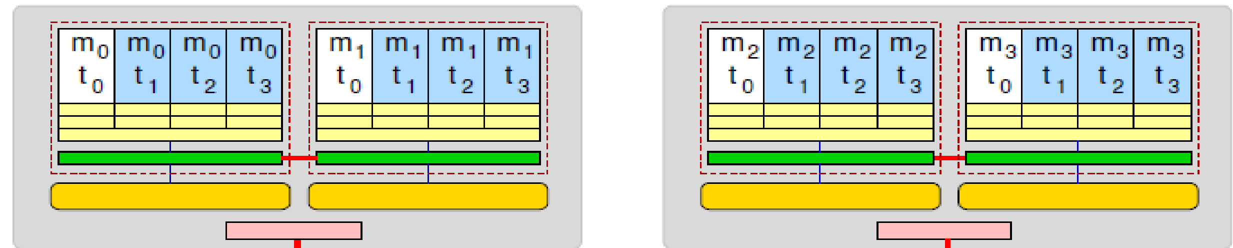
The LIKWID toolset, esp. likwid-topology and likwid-pin

Topology (“mapping”) choices with MPI+OpenMP

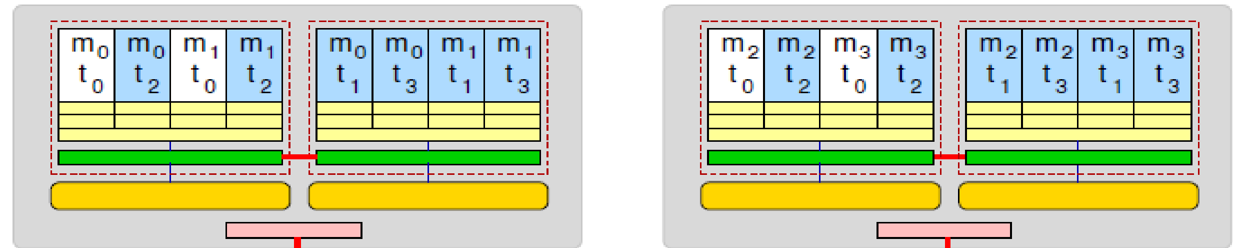
One MPI process per node



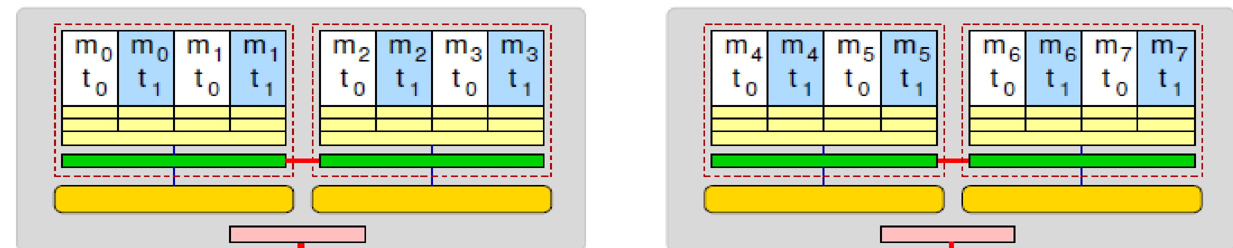
One MPI process per socket



OpenMP threads pinned “round robin” across cores in node



Two MPI processes per node



<commercial>

How do we figure out the topology?

- … and how do we enforce the mapping?
- Compilers and MPI libs may give you ways to do that

- But LIKWID supports all sorts of combinations:

  Like
  I
  Knew
  What
  I'm
  Doing

  J. Treibig, G. Hager, G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* Accepted for PSTI2010, Sep 13-16, 2010, San Diego, CA http://arxiv.org/abs/1004.4431

- Open source tool collection (developed by Jan Treibig at RRZE):

  http://code.google.com/p/likwid

Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard Linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- **Current tools:**
  - **likwid-topology: Print thread and cache topology**
  - **likwid-pin: Pin threaded application without touching code**
  - likwid-perfCtr: Measure performance counters
  - likwid-features: View and enable/disable hardware prefetchers (only for Intel Core2 at the moment)
  - likwid-bench: Bandwidth benchmark generator tool

likwid-topology – Topology information

- **Based on cpuid information**
- **Functionality:**
  - Measures clock frequency
  - Thread topology: numbering of logical cores
  - Cache topology: which HW threads share which cache level(s)
  - Cache parameters (-c command line switch)
  - ASCII art output (-g command line switch)
  - Physical and logical core numbering
- **Currently supported:**
  - Intel Core 2 (45nm + 65 nm)
  - Intel Nehalem
  - AMD K10 (Quadcore and Hexacore)
  - AMD K8

Output of likwid-topology

```
CPU name:       Intel Core i7 processor
CPU clock:      2666683826 Hz
*************************************************************
Hardware Thread Topology
*************************************************************
Sockets:                2
Cores per socket:       4
Threads per core:       2
-------------------------------------------------------------
HWThread        Thread          Core            Socket
0               0               0               0
1               1               0               0
2               0               1               0
3               1               1               0
4               0               2               0
5               1               2               0
6               0               3               0
7               1               3               0
8               0               0               1
9               1               0               1
10              0               1               1
11              1               1               1
12              0               2               1
13              1               2               1
14              0               3               1
15              1               3               1
-------------------------------------------------------------
```

likwid-topology continued

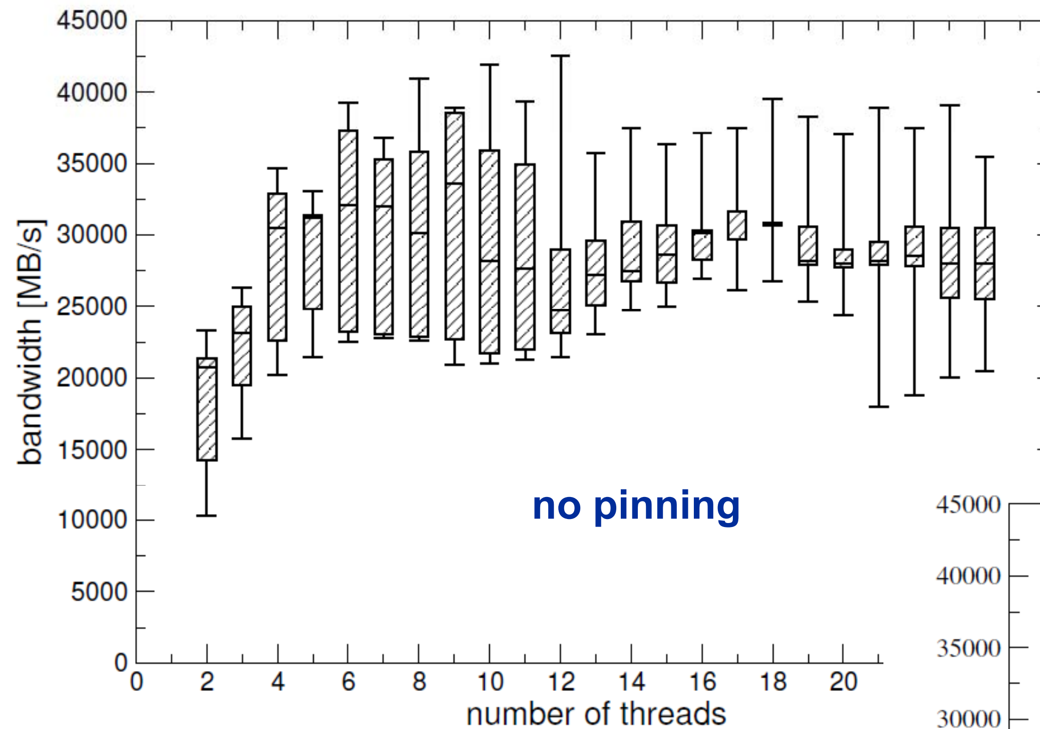
```
Socket 0: ( 0 1 2 3 4 5 6 7 )
Socket 1: ( 8 9 10 11 12 13 14 15 )
-------------------------------------------------------------

*************************************************************
Cache Topology
*************************************************************
Level:    1
Size:     32 kB
Cache groups:    ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 )
-------------------------------------------------------------
Level:    2
Size:     256 kB
Cache groups:    ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 )
-------------------------------------------------------------
Level:    3
Size:     8 MB
Cache groups:    ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 )
-------------------------------------------------------------
```

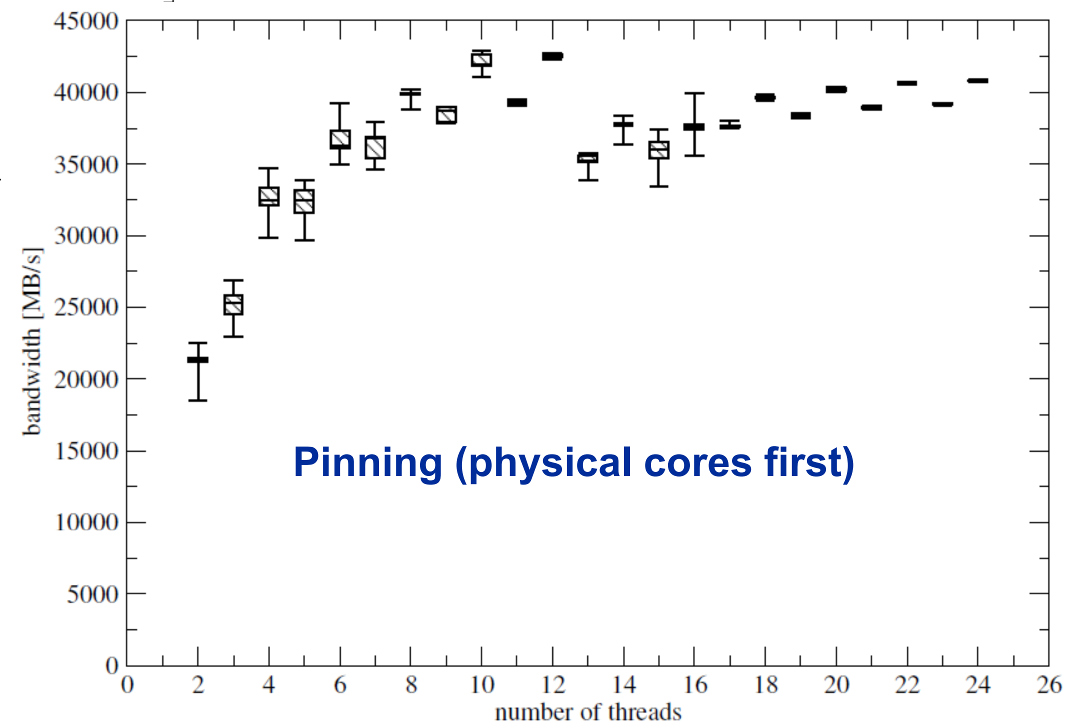
- … and also try the ultra-cool `-g` option!

likwid-pin

- Inspired by and based on `ptoverride` (Michael Meier, RRZE) and `taskset`
- Pins process and its threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify **skip mask** for **excluding auxiliary threads**
- Based on combination of wrapper tool together with overloaded pthread library
- Can also be used as **replacement for taskset**
- Defaults to logical core numbering if started inside a restricted set of cores
- Usage examples:
  - `likwid-pin -t intel -c 0,2,4-6  ./myApp`
  - `likwid-pin -c S0:0-2@S1:0-2 ./myApp`
  - `mpirun ... likwid-pin -s 0x3 -c 0,3,5,6  ./myApp`

Example: STREAM benchmark on 12-core Intel Westmere:
Anarchy vs. thread pinning

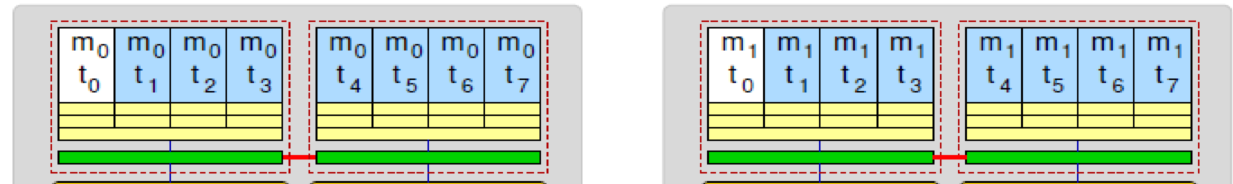




Topology ("mapping") choices with MPI+OpenMP:

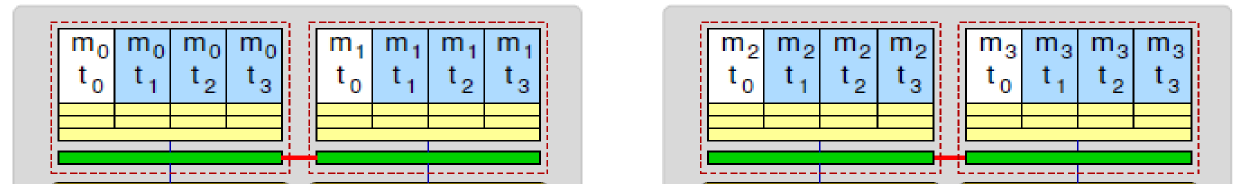
More examples using Intel MPI+compiler & home-grown mpirun

One MPI process per node



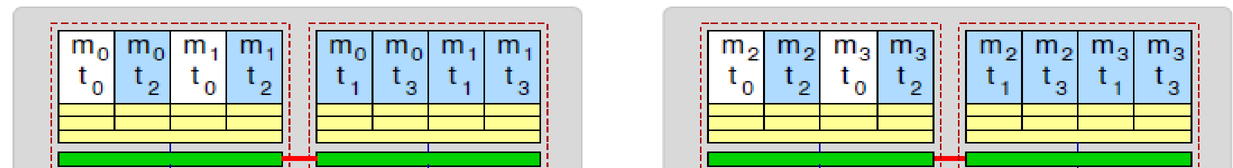

```
env OMP_NUM_THREADS=8 mpirun -pernode likwid-pin -t intel -c 0-7 ./a.out
```

One MPI process per socket



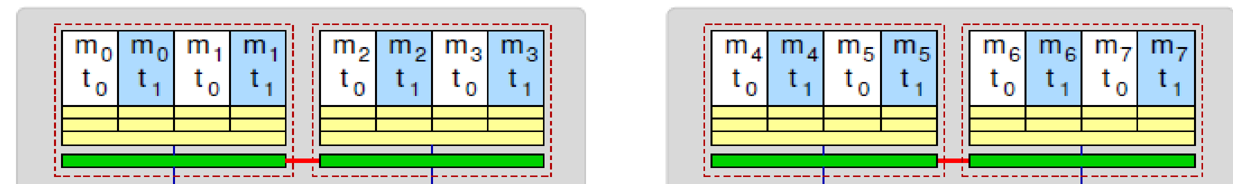

```
env OMP_NUM_THREADS=4 mpirun -npernode 2 -pin "0,1,2,3_4,5,6,7" ./a.out
```

OpenMP threads pinned "round robin" across cores in node



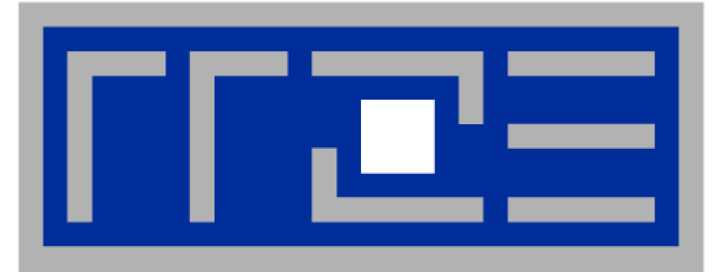

```
env OMP_NUM_THREADS=4 mpirun -npernode 2 -pin "0,1,4,5_2,3,6,7"
    likwid-pin -t intel -c 0,2,1,3 ./a.out
```

Two MPI processes per node




```
env OMP_NUM_THREADS=2 mpirun -npernode 4 -pin "0,1_2,3_4,5_6,7"
    likwid-pin -t intel -c 0,1 ./a.out
```

</commercial>



MPI:
Common problems (beyond the usual…)

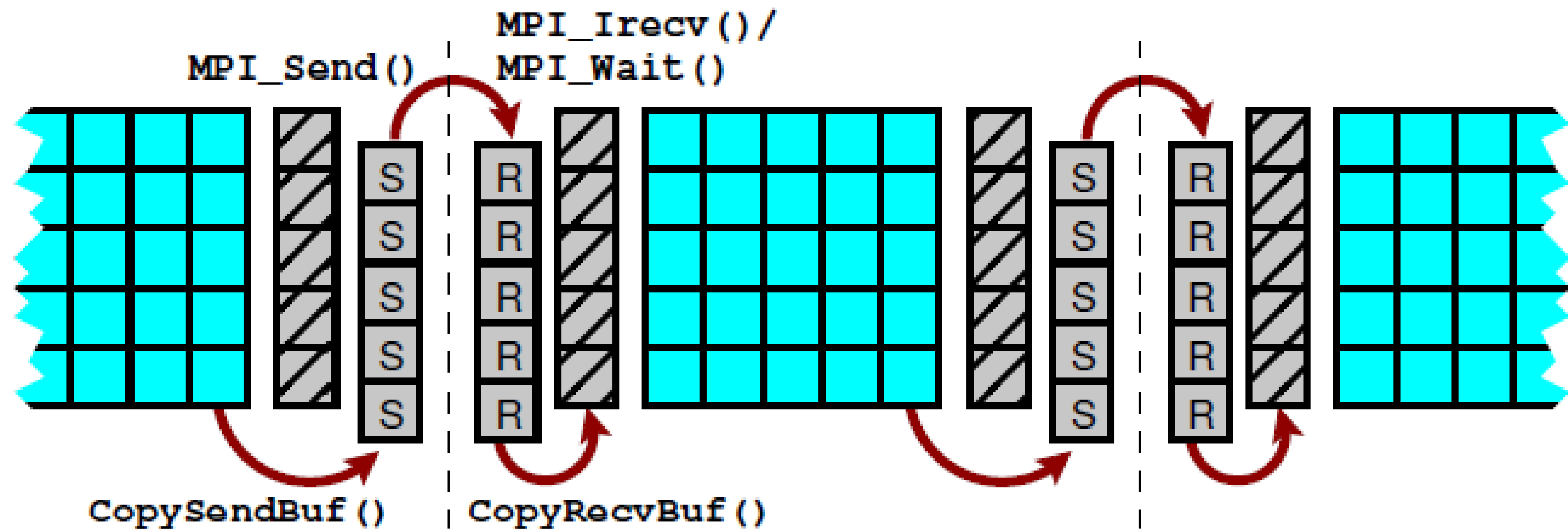
Rank-subdomain mapping

Overlapping computation with communication

Intranode communication characteristics

"Best possible" MPI:
Minimizing cross-node communication

- **Example: Stencil solver with halo exchange**



- **Goal: Reduce internode halo traffic**
- **Subdomains exchange halo with neighbors**
  - Populate a node's ranks with "maximum neighboring" subdomains
  - This minimizes a node's communication surface

- **Shouldn't `MPI_CART_CREATE` (w/ reorder) take care of this for me?**

MPI rank-subdomain mapping:
3D stencil solver – theory

Linear SD distribution
Optimal SD distribution

"Common" MPI library behavior

Woodcrest 2-socket
Nehalem EP 2-socket
Istanbul 2-socket
Shanghai 4-socket
Magny Cours 2-socket
Nehalem EX 4-socket
Magny Cours 4-socket
Sun Niagara 2

Node comm surface: 0, 50, 100, 150, 200, 250, 300

cores per node: 2, 4, 8, 12, 16, 24, 32, 48, 64

MPI rank-subdomain mapping:
3D stencil solver – measurements for 8ppn and 4ppn GBE vs. IB

32 MPI processes

8 ppn QDR-IB + GBE

~ 1.5x

4 ppn SDR-IB + GBE

Legend: ARM IB (x), STD IB (x), ARM GB (x), STD GB (x), ARM IB (o), STD IB (o), ARM GB (o), STD GB (o)

MLUP/s: 0, 1000, 2000, 3000, 4000, 5000, 6000, 7000

cube length (block size 20x20x100): 0, 500, 1000, 1500, 2000

Overlap of computation and nonblocking MPI:
A simple test

- **CN communication buffer `buf`: 80 MB**
- **`do_work()` does intra-register work for some amount of time**

```
MPI_Barrier(MPI_COMM_WORLD);
if(rank==0) {
    stime = MPI_Wtime();
    MPI_Irecv/Isend(buf,bufsize,MPI_DOUBLE,1,0,MPI_COMM_WORLD,request);
    delayTime = do_work(Length);
    MPI_Wait(request,status);
    etime = MPI_Wtime();
    cout << delayTime << " " << etime-stime << endl;
} else {
    MPI_Send(buf,bufsize,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
```

Overlap of computation and nonblocking MPI:

Results for different MPI versions and systems

Cray XT4 MPT 1.3
TinyBlue IntelMPI 3.2.2
TB OpenMPI 1.4 Isend
TB OpenMPI 1.4 Irecv
TB mvapich2 1.4rc2

overall time (work + MPI comm) [s]

0
0.05
0.1

0
0.05
0.1

time for work [s]

TinyBlue = IBM iDataPlex w/QDR IB

80MB / 1.6GB/s = 0.05s

IMB Ping-Pong: Latency

Intranode vs. internode on Woodcrest DDR-IB cluster (Intel MPI 3.1)

Latency [µs]

	Latency [µs]
IB internode	3,24
IB intranode 2S	0,55
IB intranode 1S	0,31

P P P P
C C C C
C C
Chipset
Memory

Affinity matters!

IMB Ping-Pong: Bandwidth Characteristics

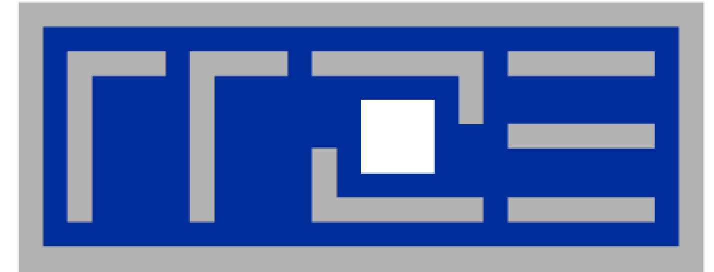
Intra-node vs. Inter-node on Woodcrest DDR-IB cluster *(Intel MPI 3.1)*

Affinity matters!

MPI take-home messages

- **MPI may not do the best it could when mapping your ranks to your subdomains**
  - Even if all it would take is to know how many processes run on a node
- **MPI may not provide truly asynchronous communication with non-blocking point-to-point calls**
  - Very common misconception
  - Check your system using low-level benchmarks
  - Task mode hybrid can save you ☺
- **MPI intranode characteristics are worth investigating**
  - Latency is good, but bandwidth may not be what you expect
  - Overlapping intranode with internode traffic should not be taken for granted

H. Stengel: *Parallel programming on hybrid hardware: Models and applications*. Master's thesis, Ohm University of Applied Sciences/RRZE, Nuremberg, 2010



A word about barrier overhead for OpenMP …

J. Treibig, G. Hager and G. Wellein: *Multi-core architectures: Complexities of performance prediction and the impact of cache topology*. To appear.

http://arxiv.org/abs/0910.4865

Thread synchronization overhead

pthreads vs. OpenMP vs. Spin loop

2 Threads	Q9550 (shared L2)	I7 920 (shared L3)
pthreads_barrier_wait	**23739**	6511
omp barrier (icc 11.0)	399	469
Spin loop	**231**	270

4 Threads	Q9550	I7 920 (shared L3)
pthreads_barrier_wait	**42533**	9820
omp barrier (icc 11.0)	977	814
Spin loop	1106	**475**

pthreads → OS kernel call

Spin loop does fine for shared cache sync

OpenMP & Intel compiler

Thread synchronization overhead
OpenMP: icc vs. gcc

gcc obviously uses pthreads barrier to for OpenMP barrier.

2 Threads	Q9550 (shared L2)	I7 920 (shared L3)
gcc 4.3.3	**22603**	7333
icc 11.0	**399**	469

4 Threads	Q9550	I7 920 (shared L3)
gcc 4.3.3	**64143**	10901
icc 11.0	977	**814**

Correct pinning of threads:

- Manual pinning in source code or
- likwid-pin: **http://code.google.com/p/likwid/**
- Prevent icc compiler from pinning → KMP_AFFINITY=disabled

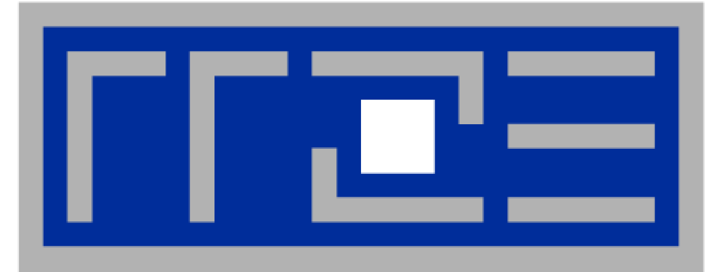
Thread synchronization overhead
Topology influence

Xeon E5420 2 Threads	shared L2	same socket	different socket
pthreads_barrier_wait	5863	27032	**27647**
omp barrier (icc 11.0)	576	760	**1269**
Spin loop	259	485	11602

Nehalem 2 Threads	Shared SMT threads	shared L3	different socket
pthreads_barrier_wait	**23352**	4796	49237
omp barrier (icc 11.0)	**2761**	479	1206
Spin loop	17388	267	787

- Spin waiting loops are not suited for SMT
  - Well known for a long time…
- Roll-your-own barrier may be better than compiler, but take care

Affinity matters!

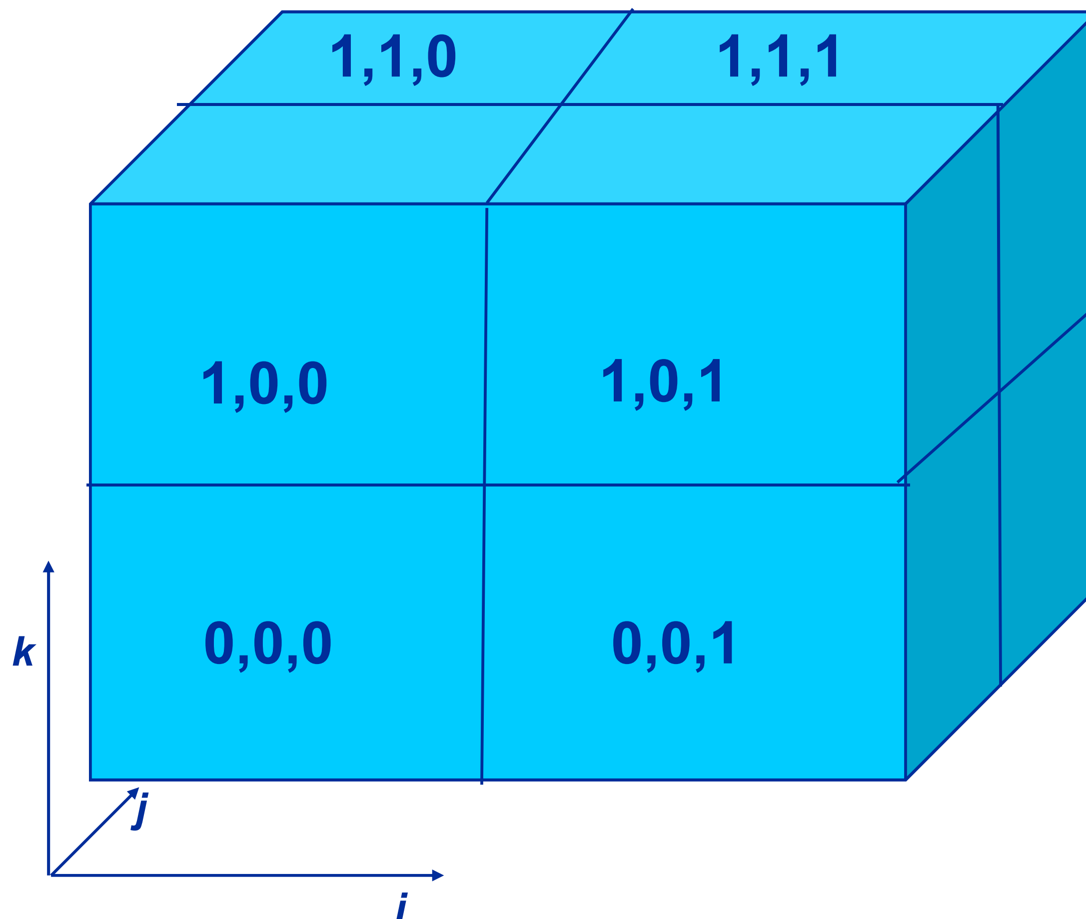


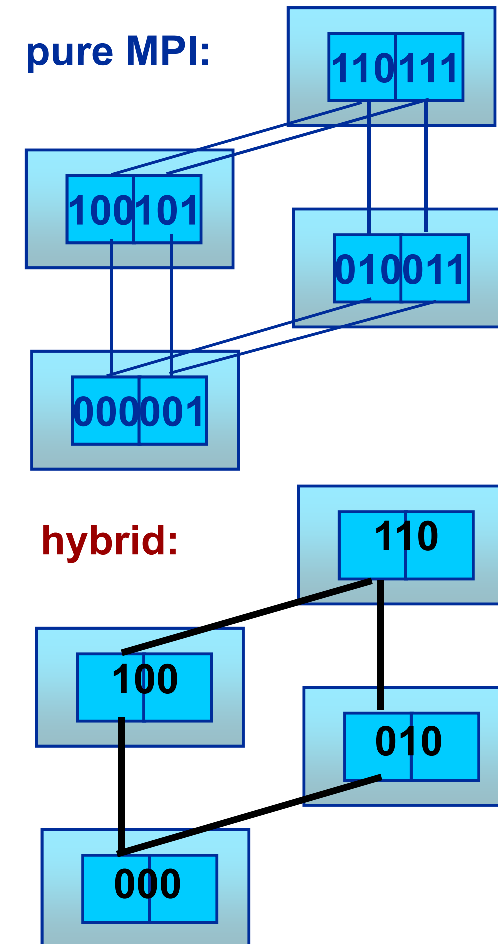
Hybrid task mode in action

… and when it makes sense to consider it at all

MPI/OpenMP Parallelization – 3D Jacobi

- Cubic 3D computational domain with periodic BCs in all directions
- Use single-node IB/GE cluster with one dual-core chip per node
- Homogeneous distribution of workload, e.g. on 8 procs

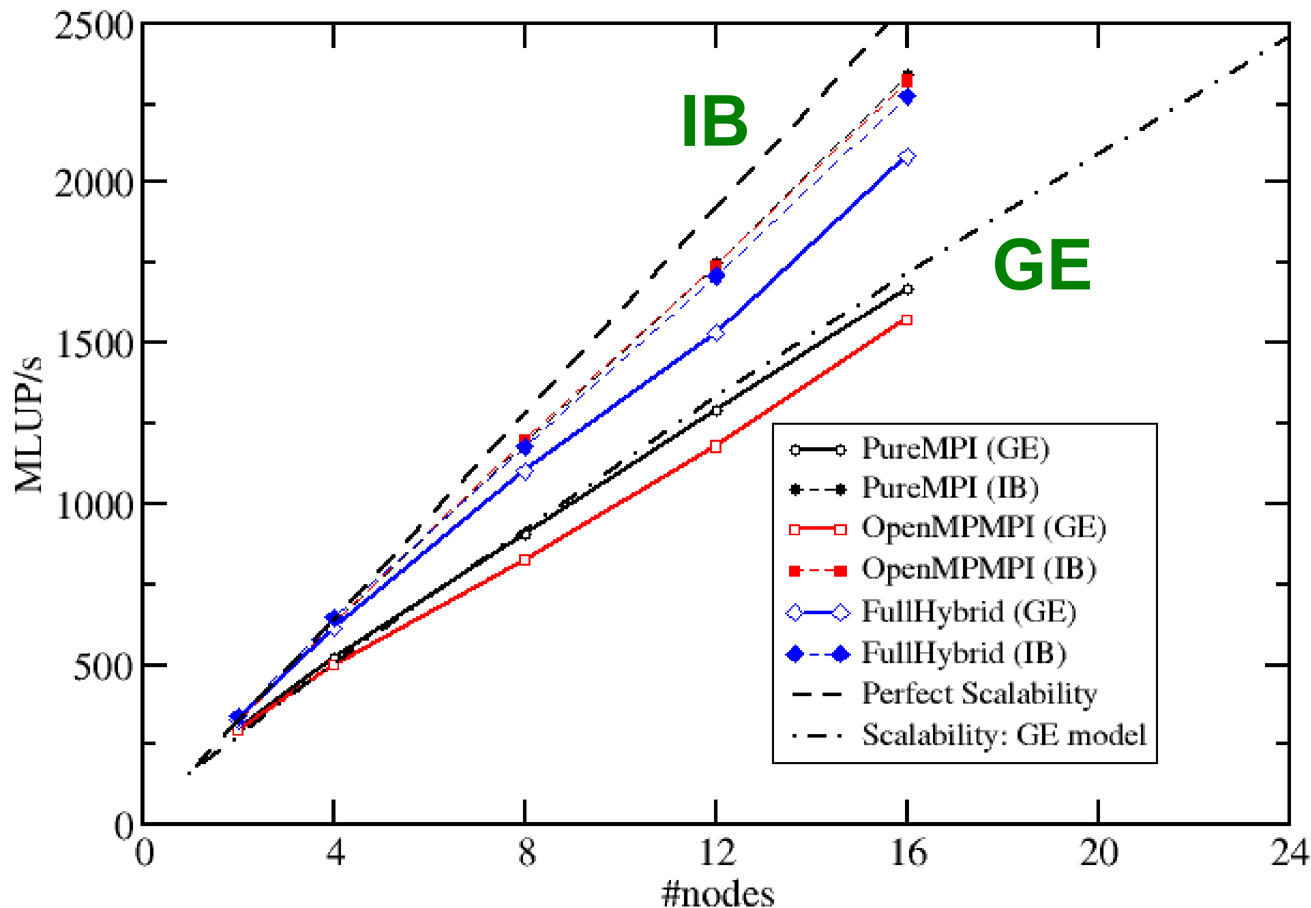




Performance Data for 3D MPI/hybrid Jacobi

Strong scaling, $N^3 = 480^3$

Hybrid: Thread 0: Communication + boundary cell updates
Thread 1: Inner cell updates



Performance model

$T = T_{COMM} + T_{COMP}$

$T_{COMP} = N^3 / P_0$

$T_{COMM} = V_{data} / BW$

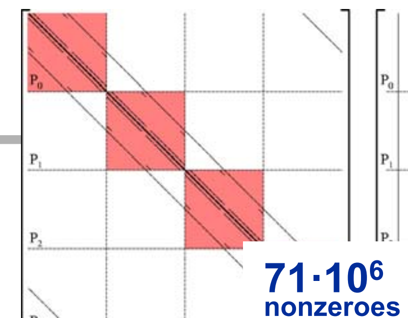
P_0 = 150 MLUP/s
BW(GE) = 100 MByte/s

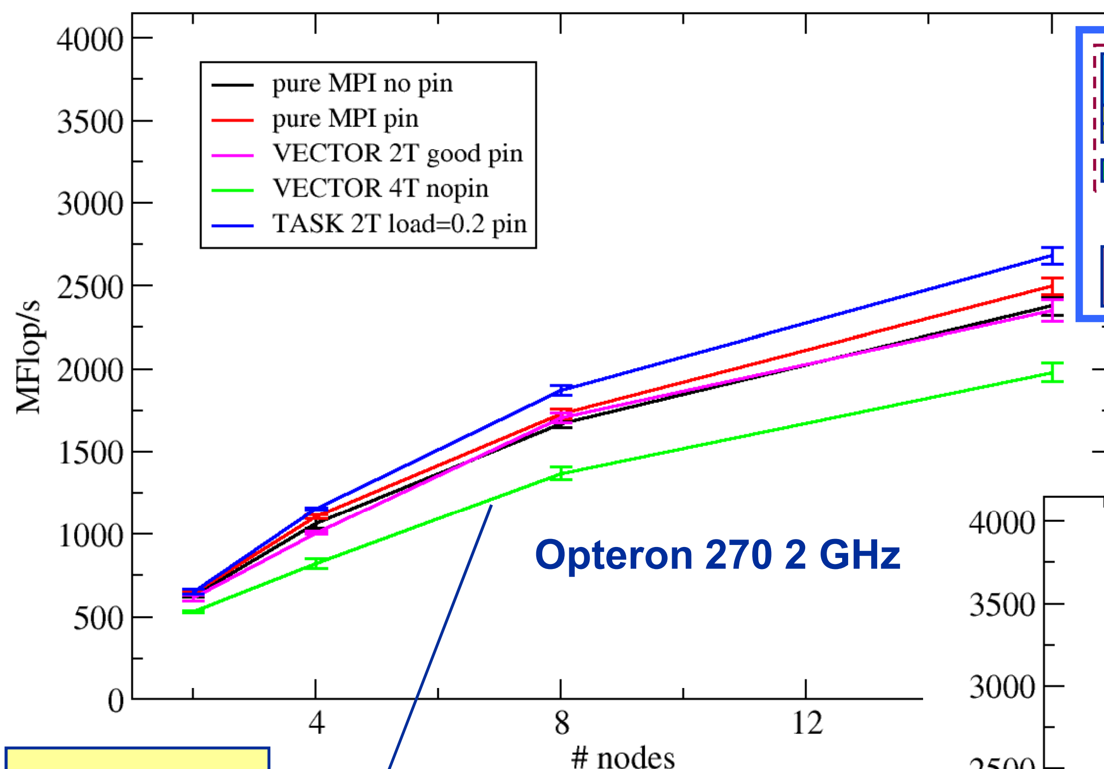
V_{data} = Data volume of halo exchange

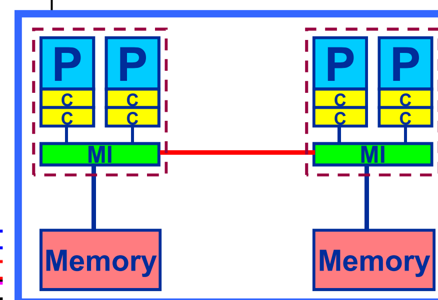
Performance estimate (GE) for *n* nodes:

$$P(n) = N^3 / ((T_{COMP}/n) + T_{COMM}(n))$$

JDS Sparse MVM:
Performance and scalability on two different platforms

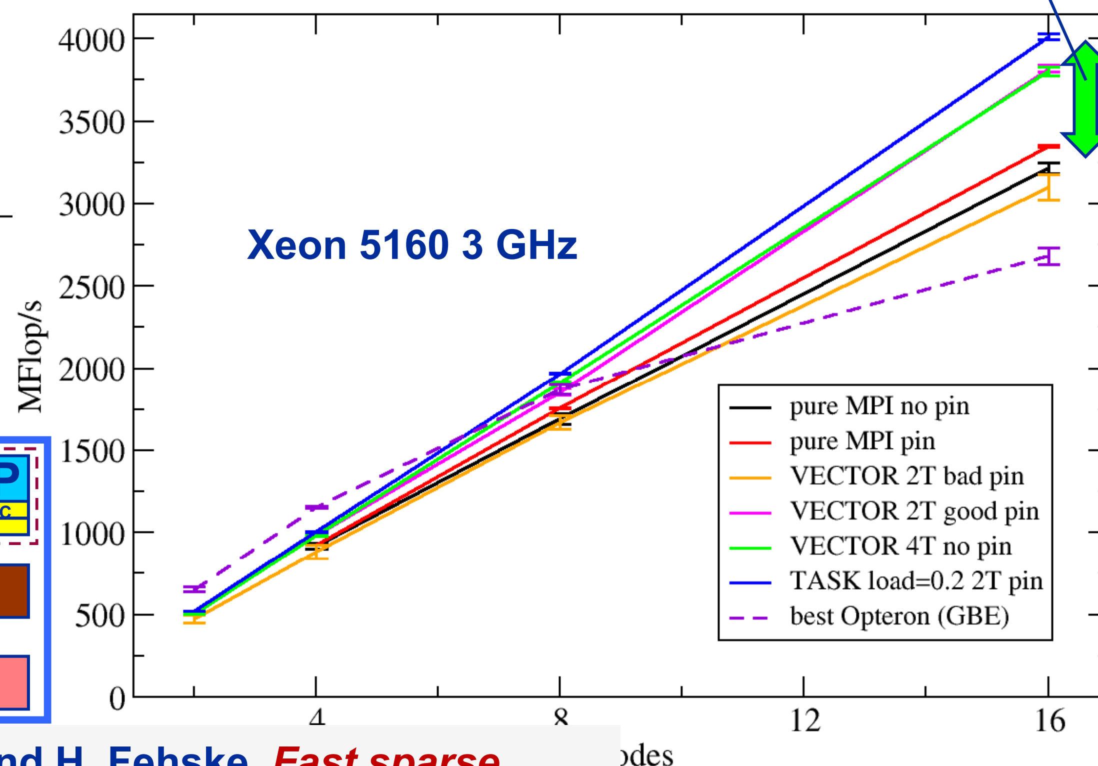


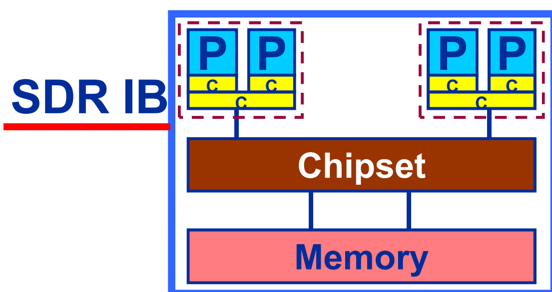




hybrid advantage

no NUMA placement!





G. Wellein and G. Hager and A. Basermann and H. Fehske, ***Fast sparse matrix-vector multiplication for TFlops computers***. **Proc. VECPAR2002, LNCS 2565**

The obvious question…

- **How do you distribute loop iterations if one thread of your team is missing?**
  - Straightforward answer: Use nested parallelism

```
#pragma omp parallel num_threads(2)
{
  if(!omp_get_thread_num()) {
    // do comm thread stuff here
  }
  else {
    #pragma omp parallel num_threads(7)
    {
      #pragma omp for
      // do work threads stuff here
    }
  }
}
```

However…

- **Nested parallelism must be supported by the compiler**
  - Probably less of a problem today
- **You don’t know what actually happens when starting a new team**
  - ccNUMA page placement?
  - Thread-core affinity?

- **Alternatives:**
  - Use manual work distribution
    - This is somewhat clumsy, but well “wrappable”
    - More importantly, it is *static* (no advanced scheduling options, but also less overhead)
  - Use “tasking” constructs
    - Dynamic scheduling (with all its advantages and drawbacks)
      M. Wittmann and G. Hager: *A proof of concept for optimizing task parallelism by locality queues*. http://arxiv.org/abs/0902.1884
    - Communication thread can participate in worksharing activities after communication is over

Hybrid task mode via “tasking” constructs (1)

- OpenMP 3.0 tasking

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      MPI_Isend(…);
      MPI_Irecv(…);
      MPI_Waitall(…);
    }
    for(i=0; i<no_of_tasks; ++i) {
      #pragma omp task
      {
         // ... do work
      }
    } // end task loop
  } // end single
} // end parallel → implicit barrier
```

Hybrid task mode via “tasking” constructs (2)

- Dynamic loop scheduling (no implicit barrier at the start of a workshared loop!)

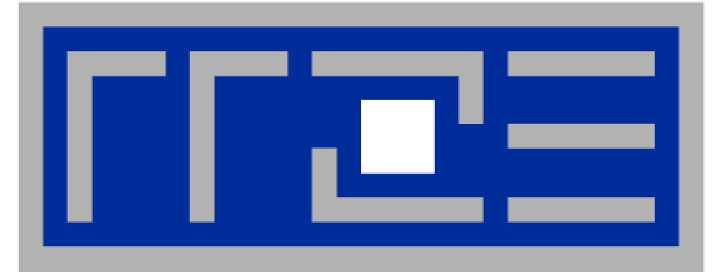
```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    MPI_Isend(…);
    MPI_Irecv(…);
    MPI_Waitall(…);
  } // end single
  #pragma omp for schedule(dynamic,cs) nowait
    for(i=0; i<no_of_tasks; ++i) {
      // ... do work
    } // end task loop
} // end parallel → implicit barrier
```

Hybrid OpenMP+MPI take-home messages

- **Hybrid task mode is almost mandatory if communication has a significant impact on runtime**
  - True overlap of communication with computation
    - Know your basics about NUMA placement, chip/node topology, thread/core affinity

- **Hybrid (task or vector mode) is sometimes unnecessary**
  - If pure MPI scales OK, why bother?

- **But: Try to figure out possible benefits through**
  - Profiling/tracing
  - Appropriate performance models
  - Awareness of the basic limitations of the underlying architecture



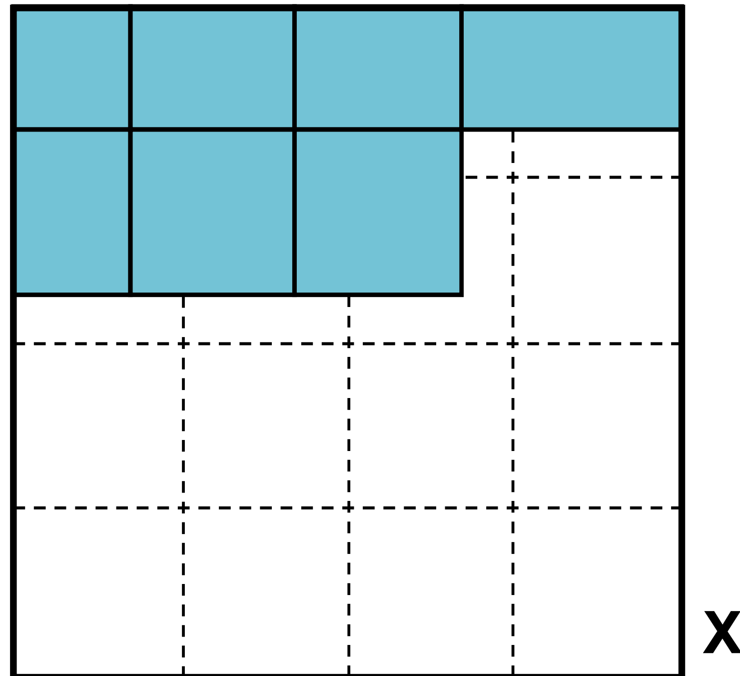
Case study: Re-use of shared cache data and relaxed synchronization with a temporally blocked Jacobi solver

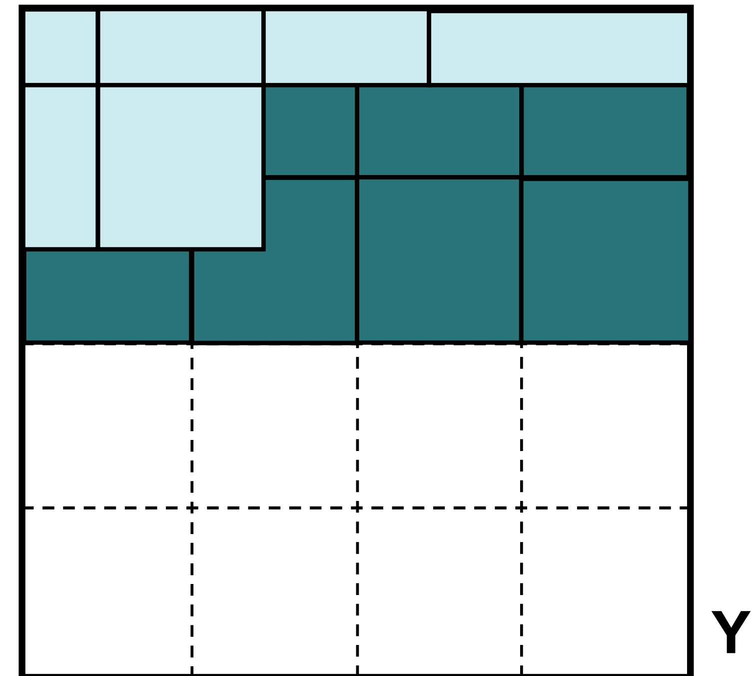
G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization*. Proc. COMPSAC 2009. Best Paper Award!

J. Treibig, G. Wellein and G. Hager: *Efficient multicore-aware parallelization strategies for iterative stencil computations.* Submitted. http://arxiv.org/abs/1004.1741

M. Wittmann, G. Hager, J. Treibig and G. Wellein: *Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters*. Accepted for Parallel Processing Letters, December 2010. http://arxiv.org/abs/1006.3148

Pipelined temporal blocking



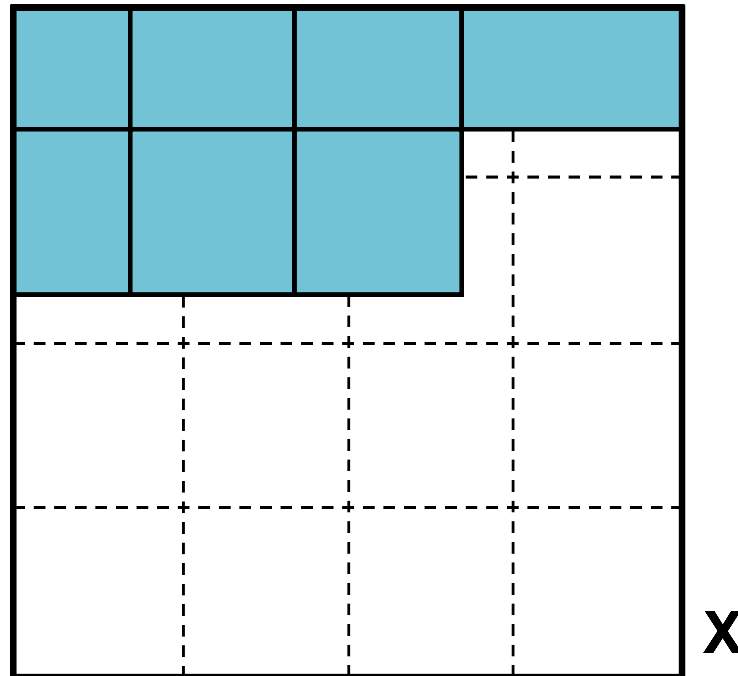


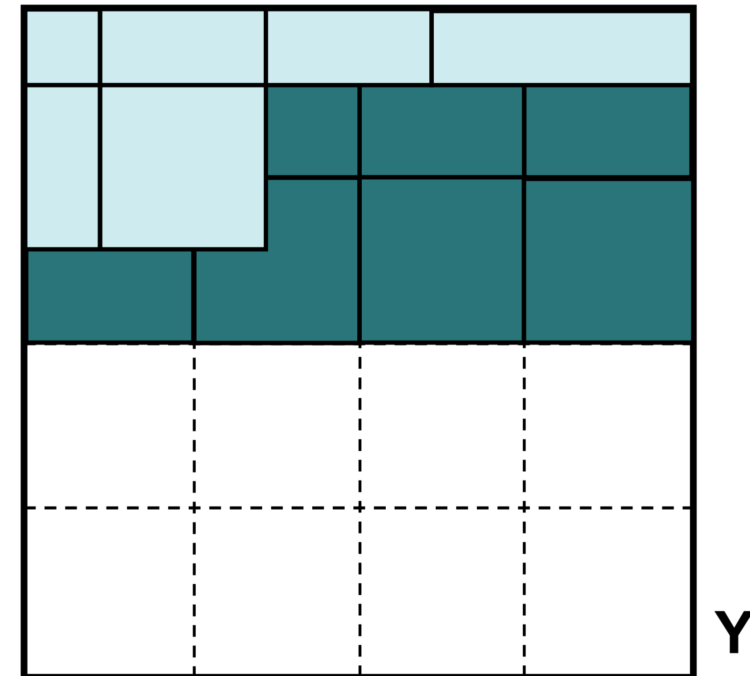
thread 0 ($t_0 \rightarrow t_1$)

thread 1 ($t_1 \rightarrow t_2$)

thread 2 ($t_2 \rightarrow t_3$)

Pipelined temporal blocking





One long pipeline (all cores of a node) advances through the lattice, each update is shifted by (-1,-1,-1)

Advantages

- Freestyle spatial blocking
- No explicit boundary copies
- Multiple updates per core

Drawbacks

- Shift reduces cache reuse
- Huge parameter space
- Boundary tiles

Temporal Blocking w/ PPP on Nehalem EP (Core i7)

core 0 core 1 core 2 core 3

L1 L1 L1 L1

L2 L2 L2 L2

IV 1st upd.

III 2nd upd.

L3

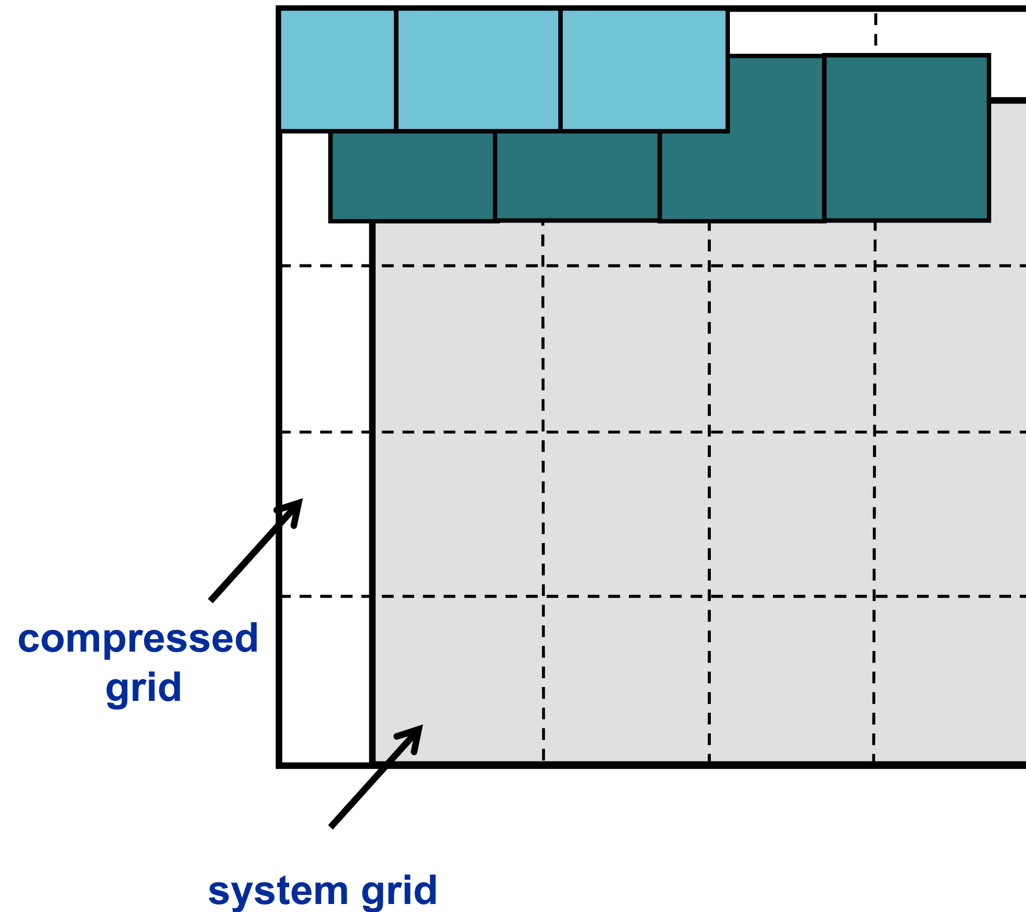
II 3rd upd.

I 4th upd.

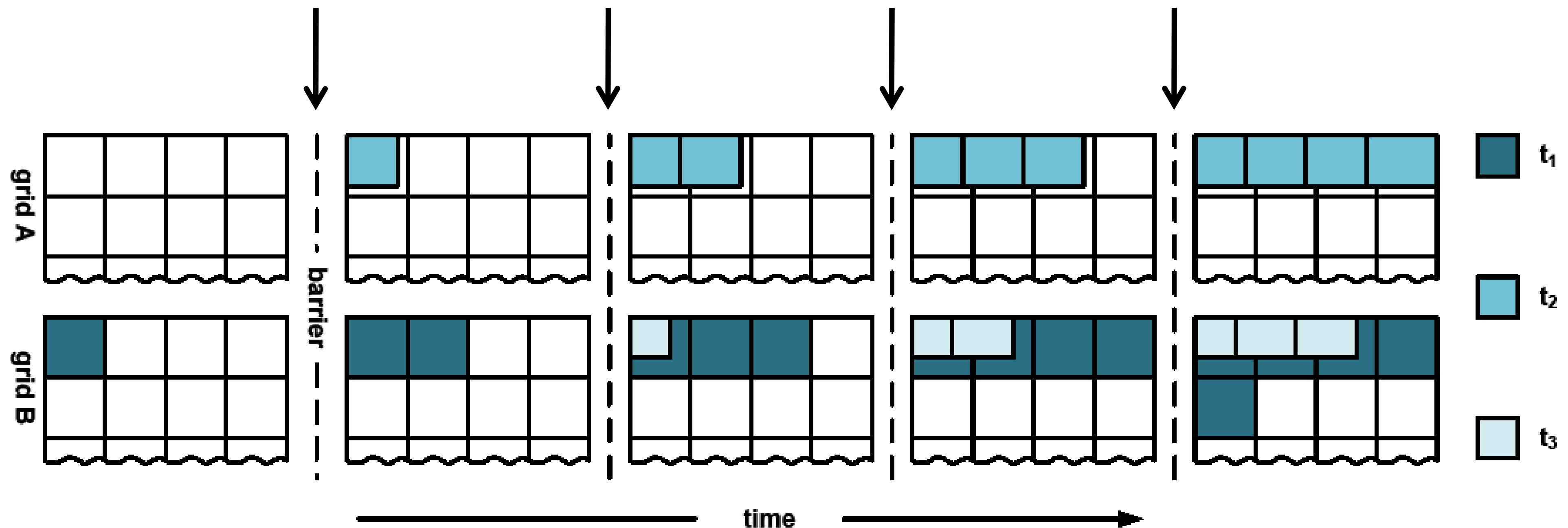
memory

Save six memory accesses by reusing blocks from shared cache

Pipelined temporal blocking
with compressed Grid

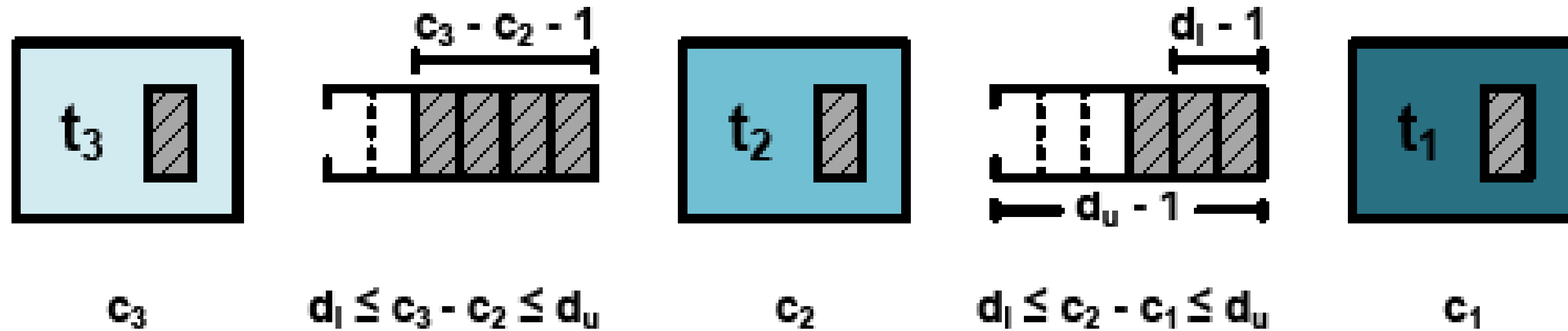


Pipelined temporal blocking



- **All threads need to synchronize after finishing T iterations on their current tile**
- **Synchronization gets more expensive with increasing number of threads**

Relaxed Synchronisation



- Every thread t_i only increments its own counter c_i
- Thread t_i has a minimal distance d_l to its preceding thread t_{i-1}
- Thread t_i has a maximal distance d_u to its following thread t_{i+1}
- Two threads have at least d_l and at most d_u tiles between them

Performance with different looseness

3500
3000
2500
2000
1500
1000

MLUP/s

w/ barrier

w/ barrier

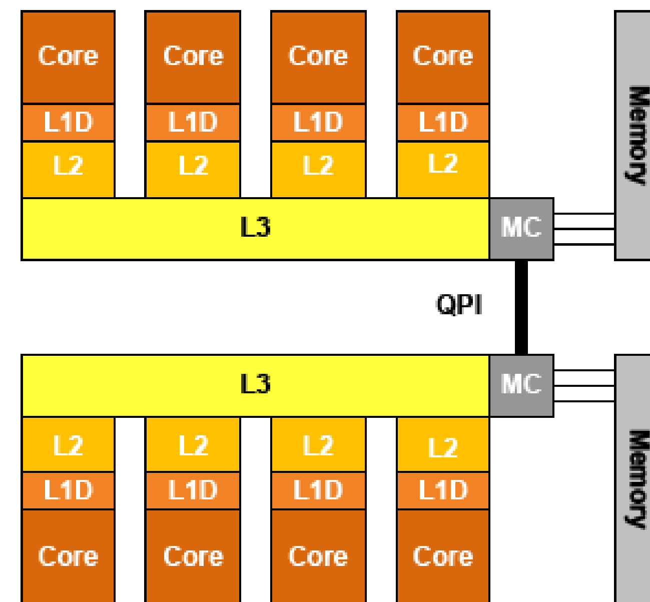
2 sockets,
8 cores,
8 threads

1 socket,
4 cores,
4 threads

0 10 20 30 40 50

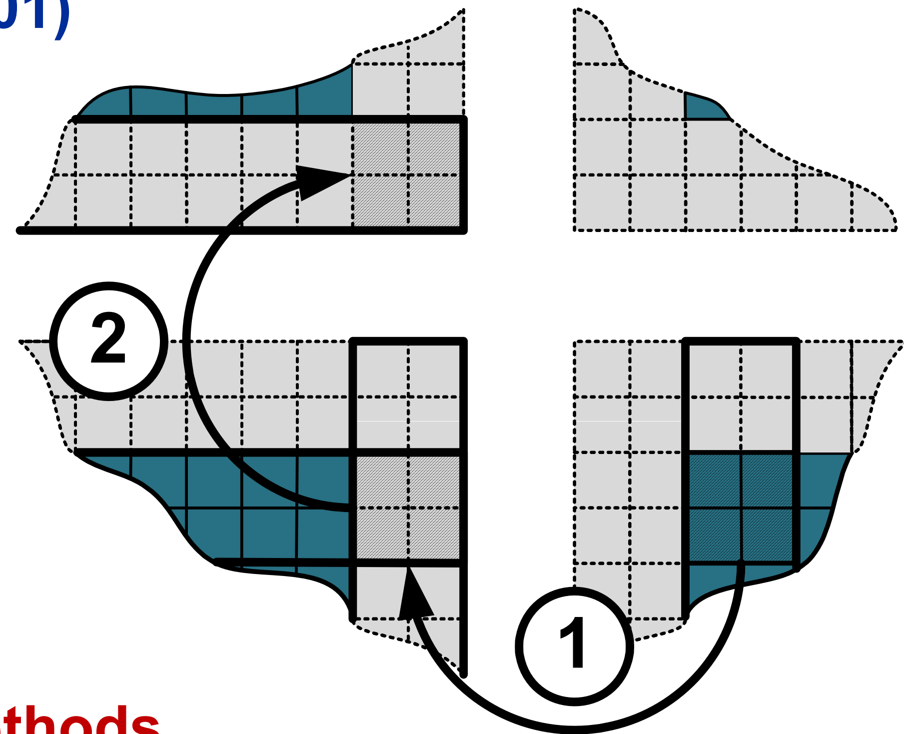
rel. sync. range

- Intel Nehalem 2.66 GHz
- 4 cores/socket
- 2 HW threads/core



Hybrid temporally blocked computations via multi-layer halos

- **Temporal blocking requires multi-layer halos**
- **Using diagonal communication elimination (DCE) (Ding/He SC 2001)**
- **Exchanging halo with neighbors done only along the coordinate directions**



- **More complex stencils, e.g. occurring at lattice Boltzmann methods, need more attention for deciding which data to communicate**

Impact of Multi-layer Halo on Performance

- **Assumptions for model:**
- No overlap between communication and computation
- QDR InfiniBand
  - 3.2 GB/s
  - 1.8 µs latency
- Node performance 2 GLUP/s

Reduced latency by message aggregation

Degrade due to halo work

No impact for large domain sizes

strong scaling

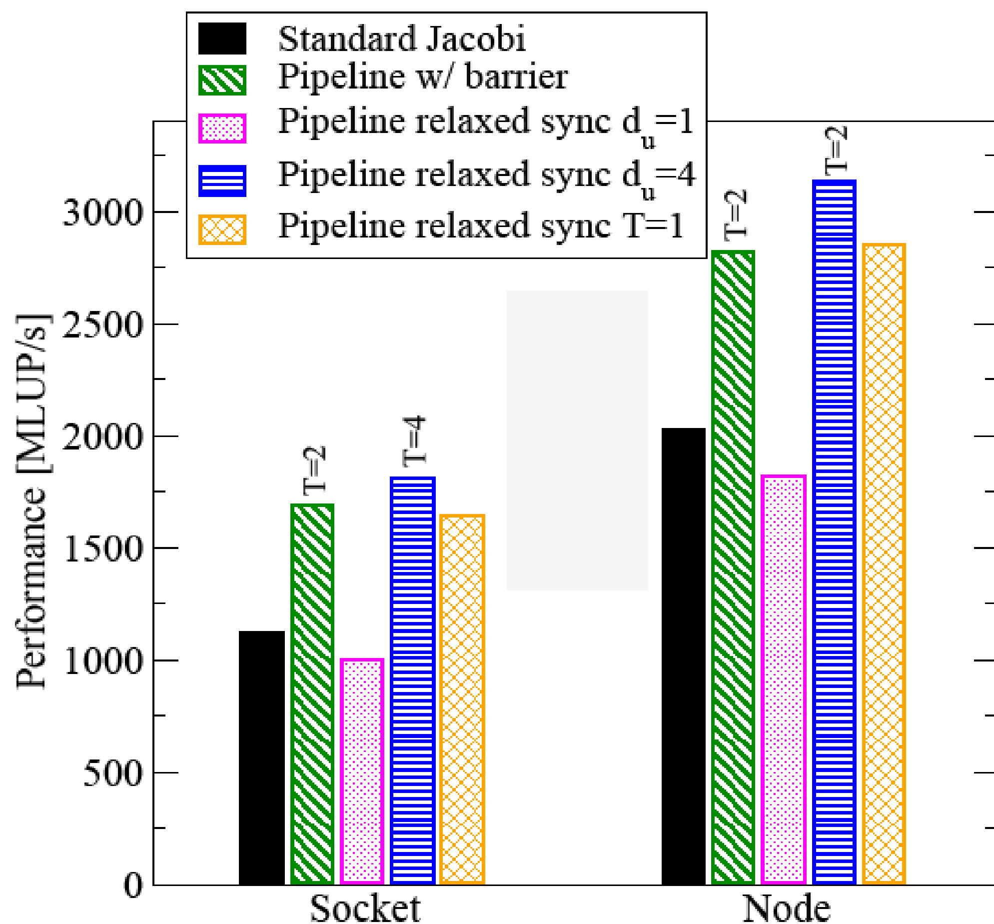
h=2
h=4
h=8
h=16
h=32

Ratio comp./overall time

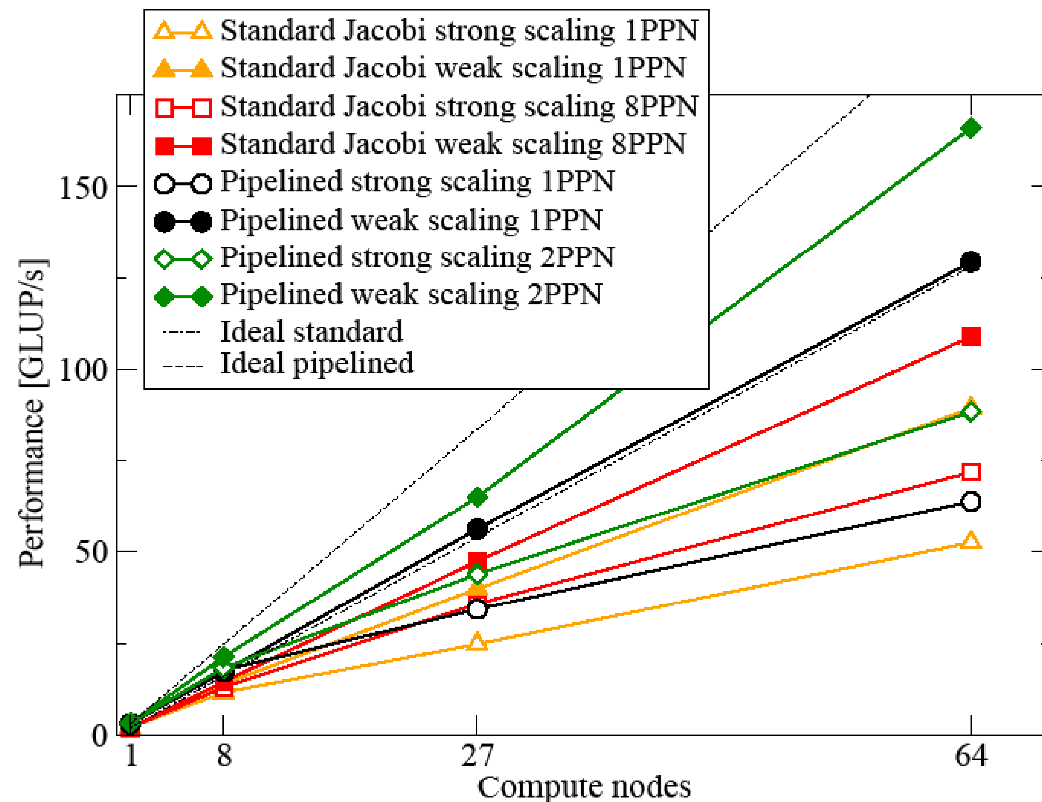
L

Performance Results on NHL EP QDR IB cluster

Single-node and multinode

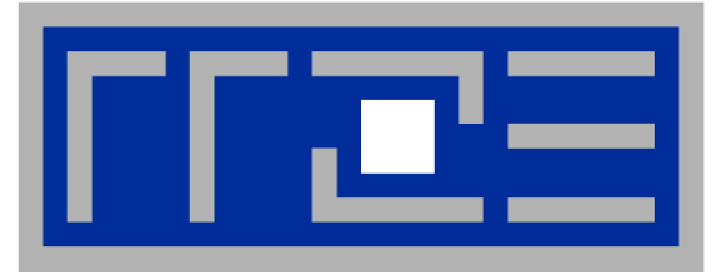


System size: 600^3 (per node)

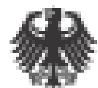


Conclusions

- **Whatever you do, be aware of the limitations the hardware puts on your code's performance**
  - Apply performance models whenever possible
- **Investigate and apply proper thread/core affinity**
  - Use LIKWID or the MPI/compiler facilities or anything, *but use it!*
- **Intranode MPI effects may be important**
- **If MPI performs/scales ok, don't bother using MPI+OpenMP**
- **However, if you can leverage new features it may still be worth looking into**
  - Shared caches are *the* interesting property of modern CPUs
  - Load balancing, new levels of parallelism, convergence,…
- **Be aware of the typical OpenMP pitfalls**
  - Synchronization and work distribution overheads are most prominent
  - … and they are really topology-dependent



THANK YOU



Bundesministerium
für Bildung
und Forschung

Supported by
BMBF, grant No 01IH08003A
(project SKALB)

Supported by
KONWIHR, project
OMI4PAPPS