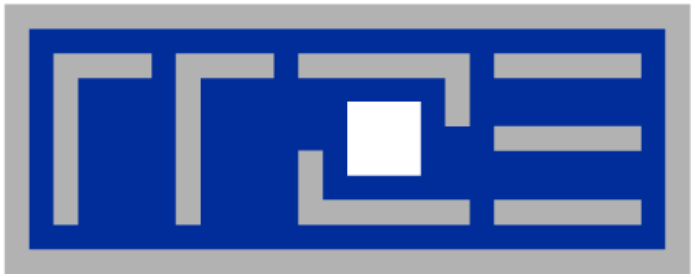


# Hybrid (i.e. MPI+OpenMP) applications (i.e. programming) on modern (i.e. multi-socket multi-NUMA-domain multi-core multi-cache multi-whatever) architectures: Things to consider


Georg Hager
Gerhard Wellein
Markus Wittmann
Holger Stengel
Jan Treibig

Erlangen Regional Computing Center (RRZE)


SIAM PP10, MS45

February 26th, 2010

# 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:**

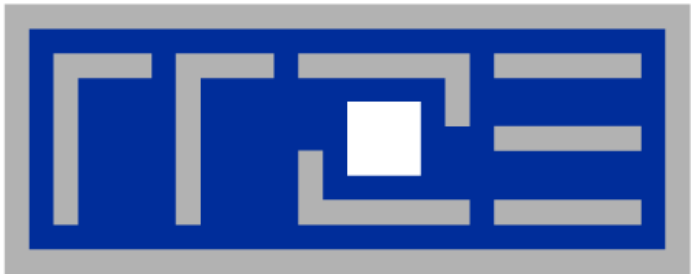
**"You have to compare apples to apples, i.e. the best hybrid code to the best pure MPI code"**

Needless to say, both may require significant optimization effort.

# Outline

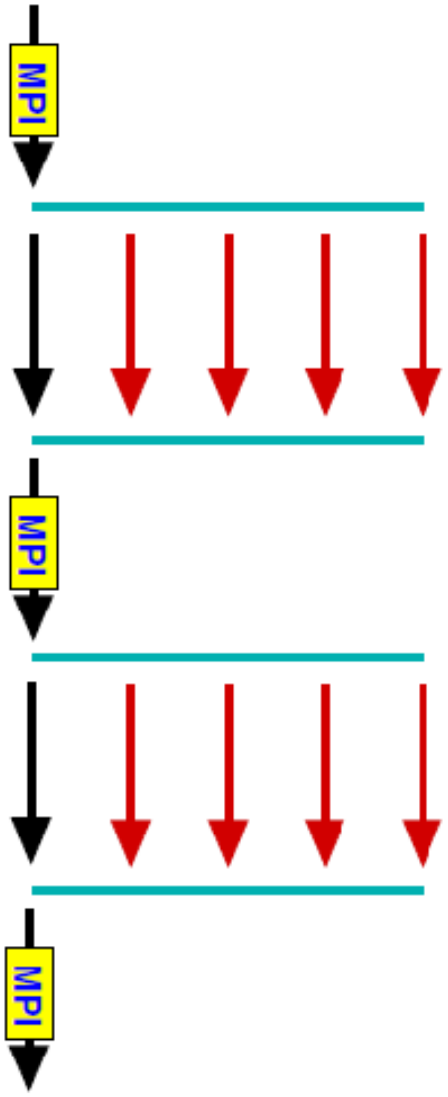
- **Hybrid programming benefits and taxonomy**
  - Vector mode, task mode
  - Thread-core mapping
- **“Best possible” MPI code**
  - Rank-subdomain mapping
  - Overlapping computation and communication via non-blocking MPI
  - Overlapping cross-node and intra-node communication
- **“Best possible” OpenMP code**
  - Cache re-use
  - Synchronization overhead
  - ccNUMA page placement
- **“Best possible” MPI+OpenMP hybrid code**
  - True comm/calc overlap
  - Load distribution issues
  - ccNUMA and task mode

**To see it all, visit one of the hybrid tutorials at SC, ISC, …!**



# Hybrid taxonomy and possible benefits

# Taxonomy of hybrid "modes":
*Several OpenMP threads per MPI process*

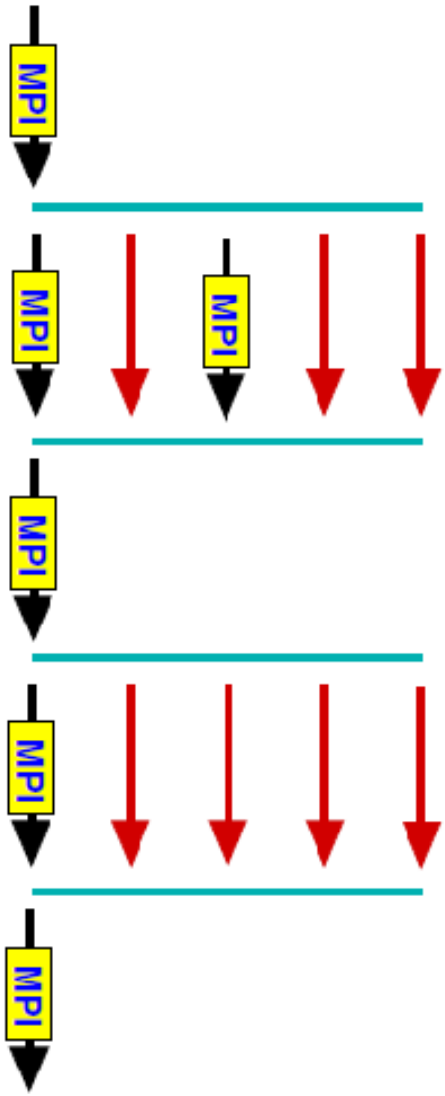



**Vector mode: MPI is called only outside OpenMP parallel regions. This is what many people mean when they say "hybrid"**

- 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**

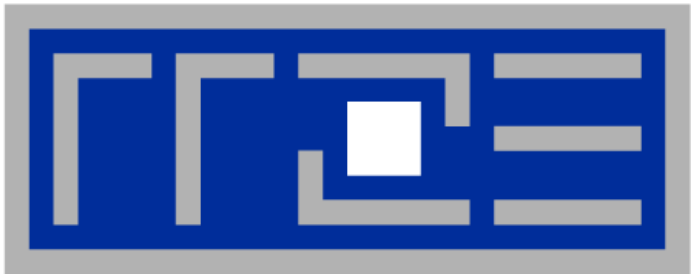
- This is functional parallelism on the thread level




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 | ✓ | ✓ |
| Overlapping communication and computation | | ✓ |
| Improved rate of convergence | ✓ | ✓ |
| Re-use of data in shared caches | ✓ | ✓ |
| Reduced MPI overhead | ✓ | ✓ |



# Hybrid mapping choices on current harware

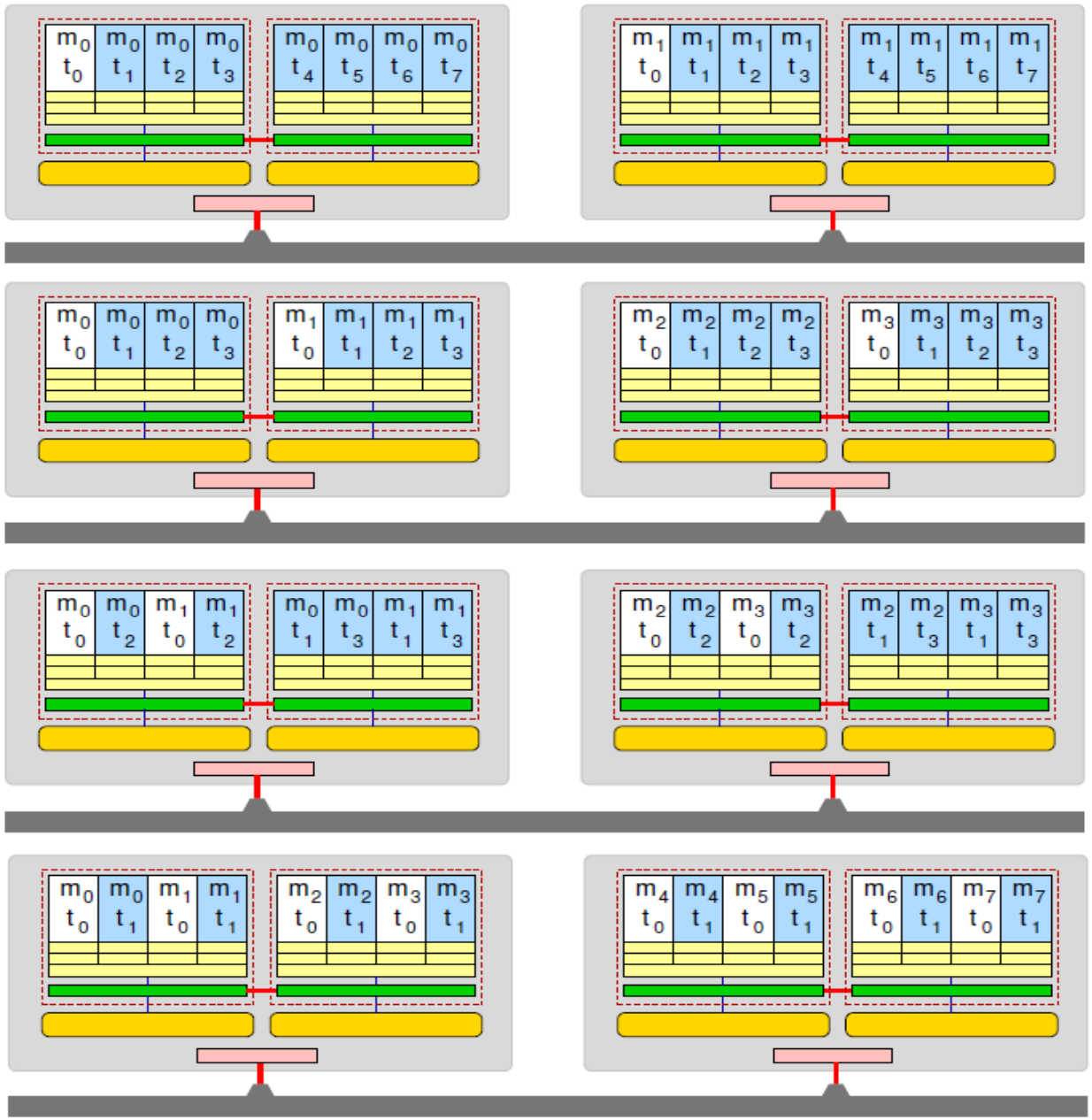
# Topology ("mapping") choices with MPI+OpenMP

**One MPI process per node**

**One MPI process per socket**

**OpenMP threads pinned "round robin" across cores in node**

**Two MPI processes per node**



# 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 combniations:

  Like
  I
  Knew
  What
  I'm
  Doing

- Open source tool collection (developed 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

# likwid-topology – Topology information

- **Based on cpuid information**
- **Functionality:**
  - Measured clock frequency
  - Thread topology
  - Cache topology
  - Cache parameters (-c command line switch)
  - ASCII art output (-g command line switch)
- **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

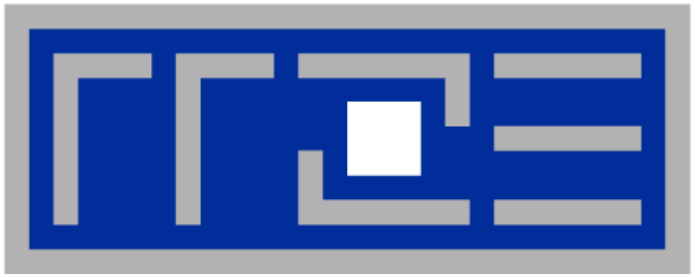
```
Socket 0: ( 0 1 2 3 4 5 6 7 )
Socket 1: ( 8 9 10 11 12 13 14 15 )
--------------------------------------------------------------

**************************************************************
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 and based on `ptoverride` (Michael Meier, RRZE) and `taskset`
- Pins process and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify skip mask (hybrid)
- Based on combination of wrapper tool together with overloaded pthread library
- Can also be used as replacement for taskset
- Configurable colored output

- Usage:
  - `likwid-pin  -c 0,2,4-6  ./myApp parameters`
  - `mpirun likwid-pin -s 0x3 -c 0,3,5,6  ./myApp parameters`



# MPI:
# Common problems (beyond the usual…)

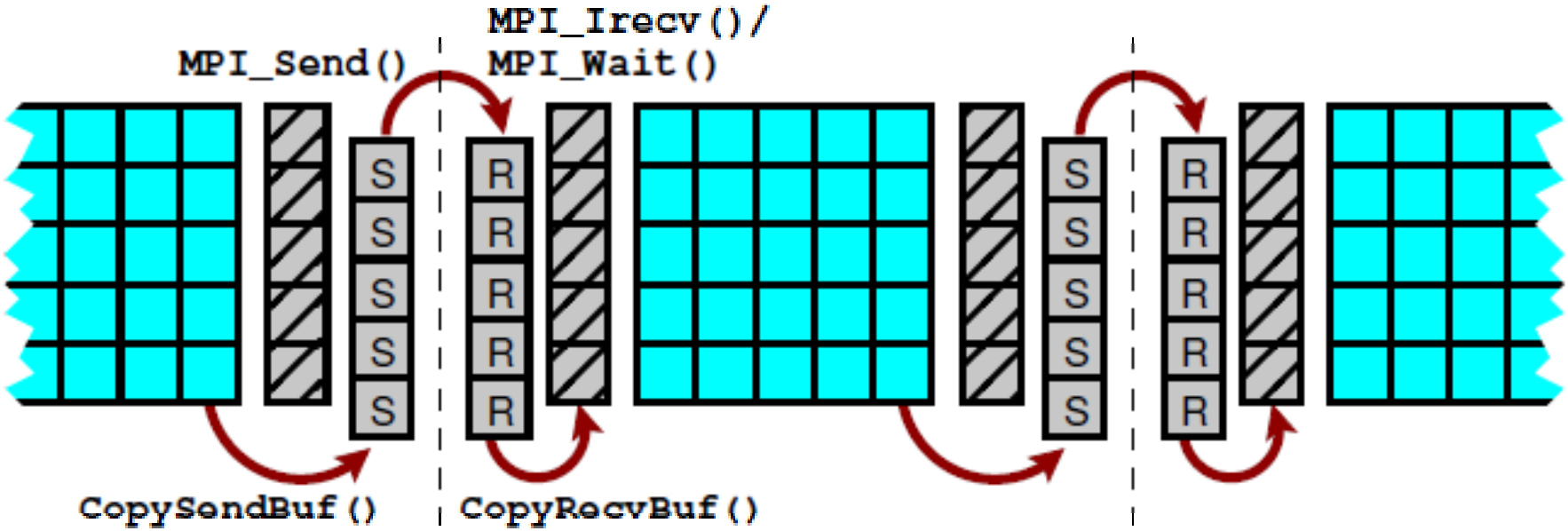
## Rank-subomain mapping

## Overlapping computation with communication

# “Best possible” MPI:
*Minimizing cross-node communication*



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

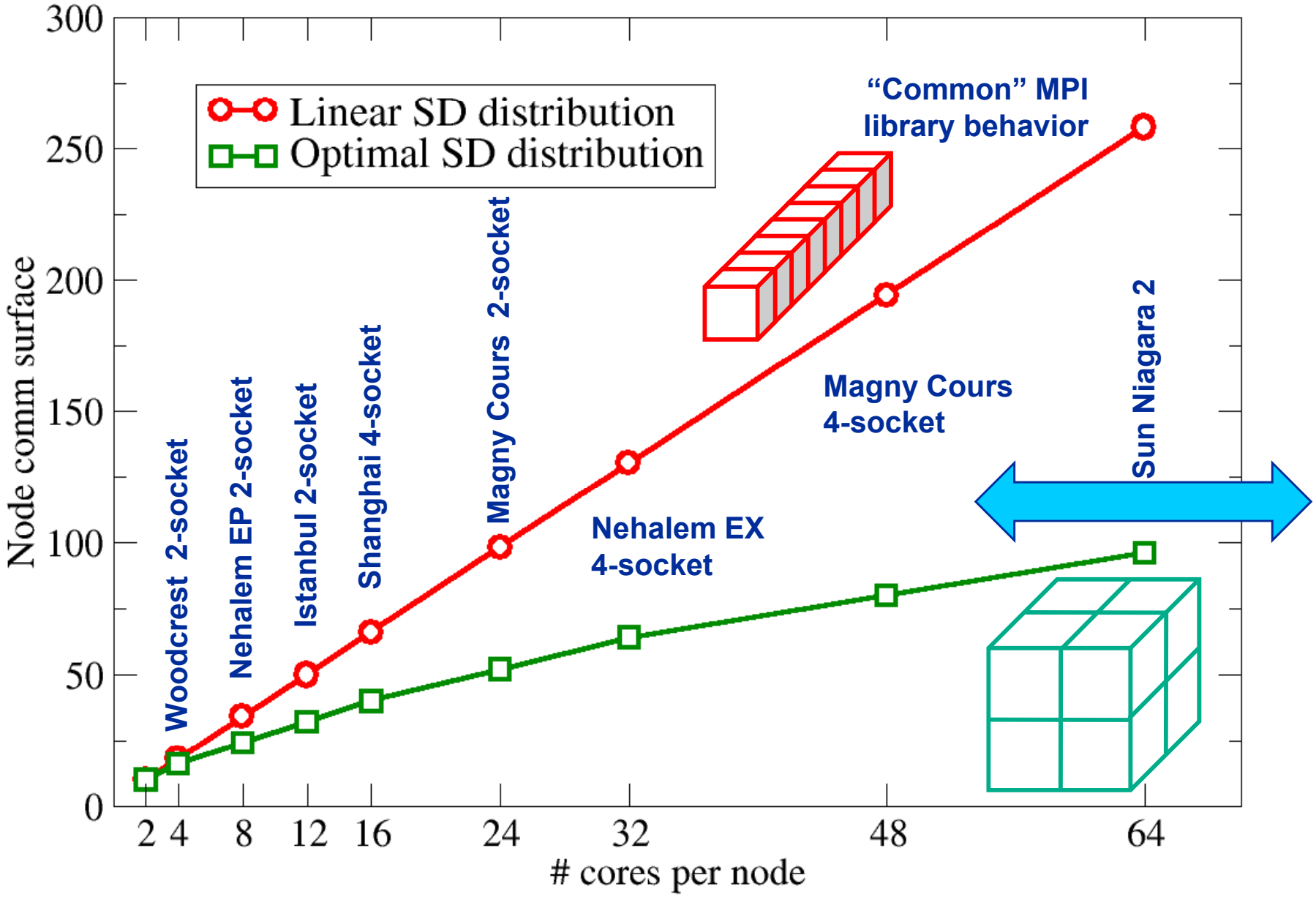



- **Goal: Reduce cross-node (CN) halo traffic**
- **Subdomains exchange halo with neighbors**
  - Populate a node's ranks with “maximum neighboring” subdomains
  - This minimizes a node's CN communication surface

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

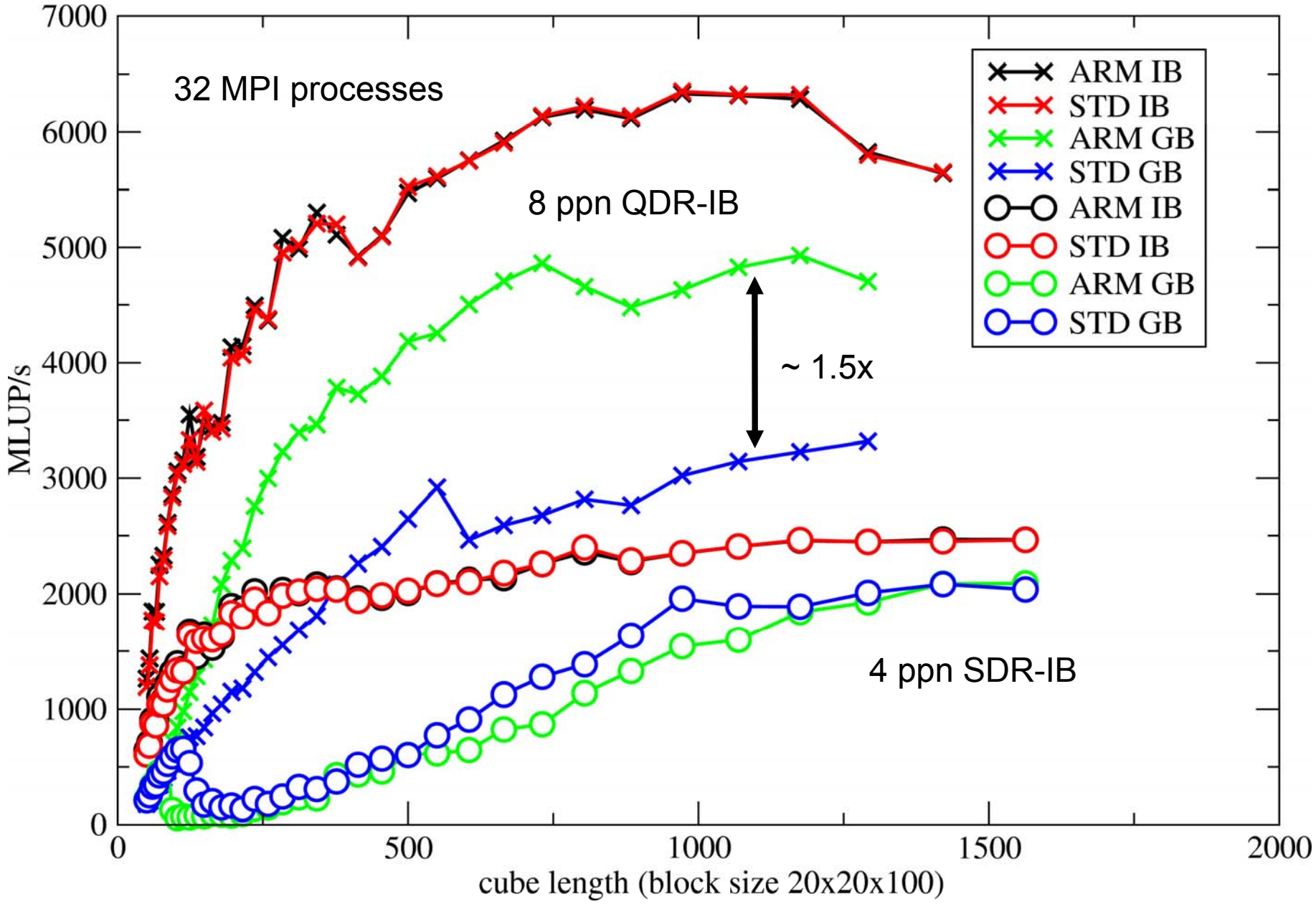
# MPI rank-subdomain mapping:

*3D stencil solver – theory*



# MPI rank-subdomain mapping:

*3D stencil solver – measurements for 8ppn and 4ppn GBE vs. IB*



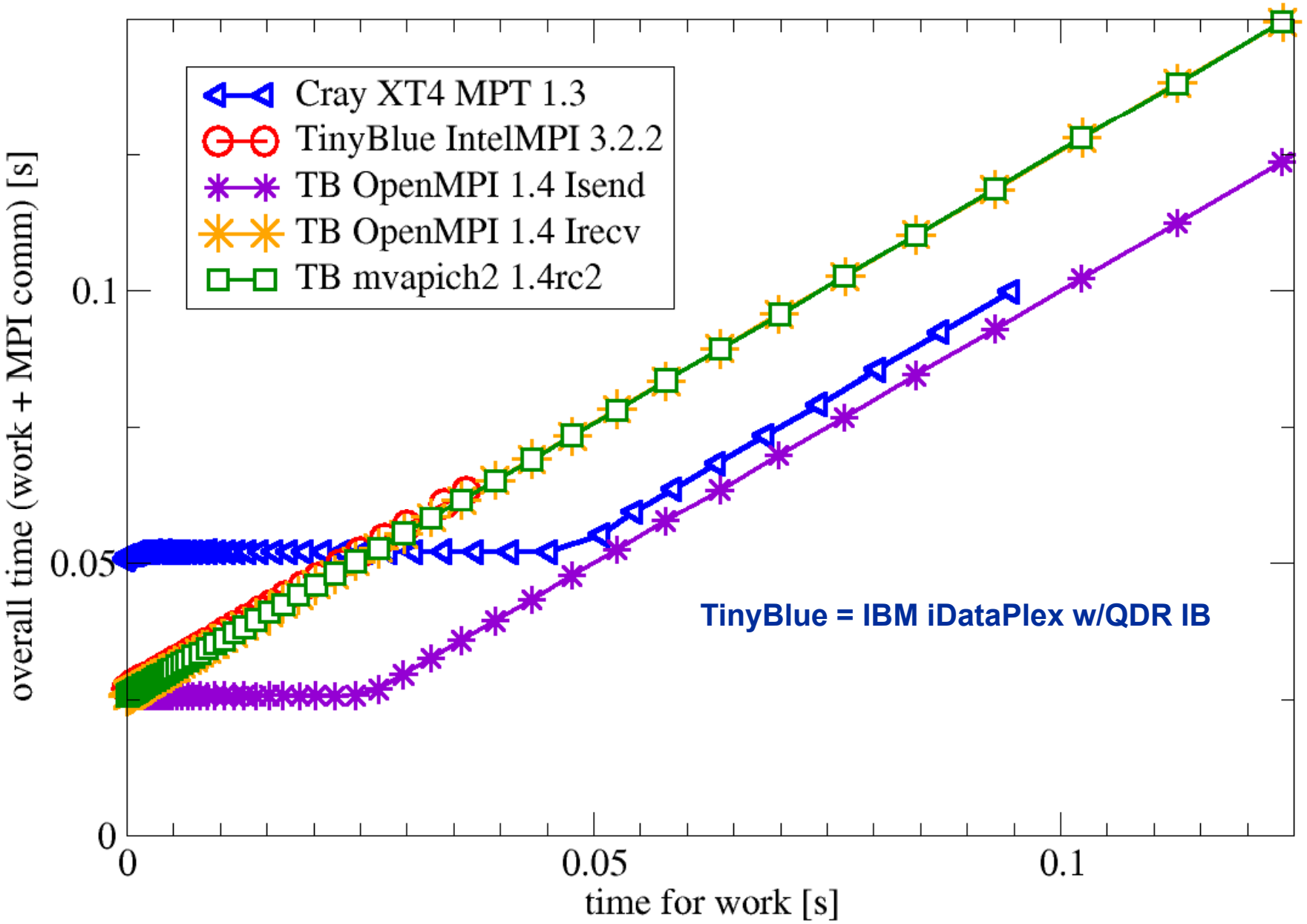
# Overlap of computation and non-blocking MPI:
*A simple test*

- 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);
```

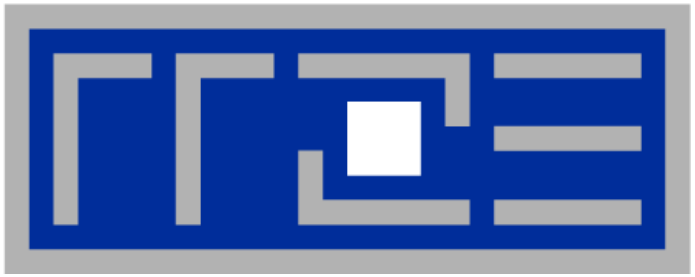
# Overlap of computation and non-blocking MPI:

*Results for different MPI versions*



# 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 ☺



# A word about barrier overhead in general…

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

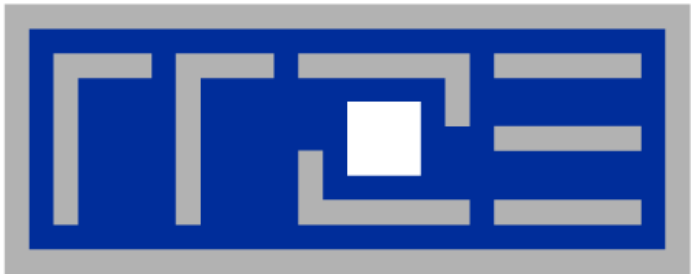
# Thread synchronization overhead

*Topology influence*

| Xeon E5420<br>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<br>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

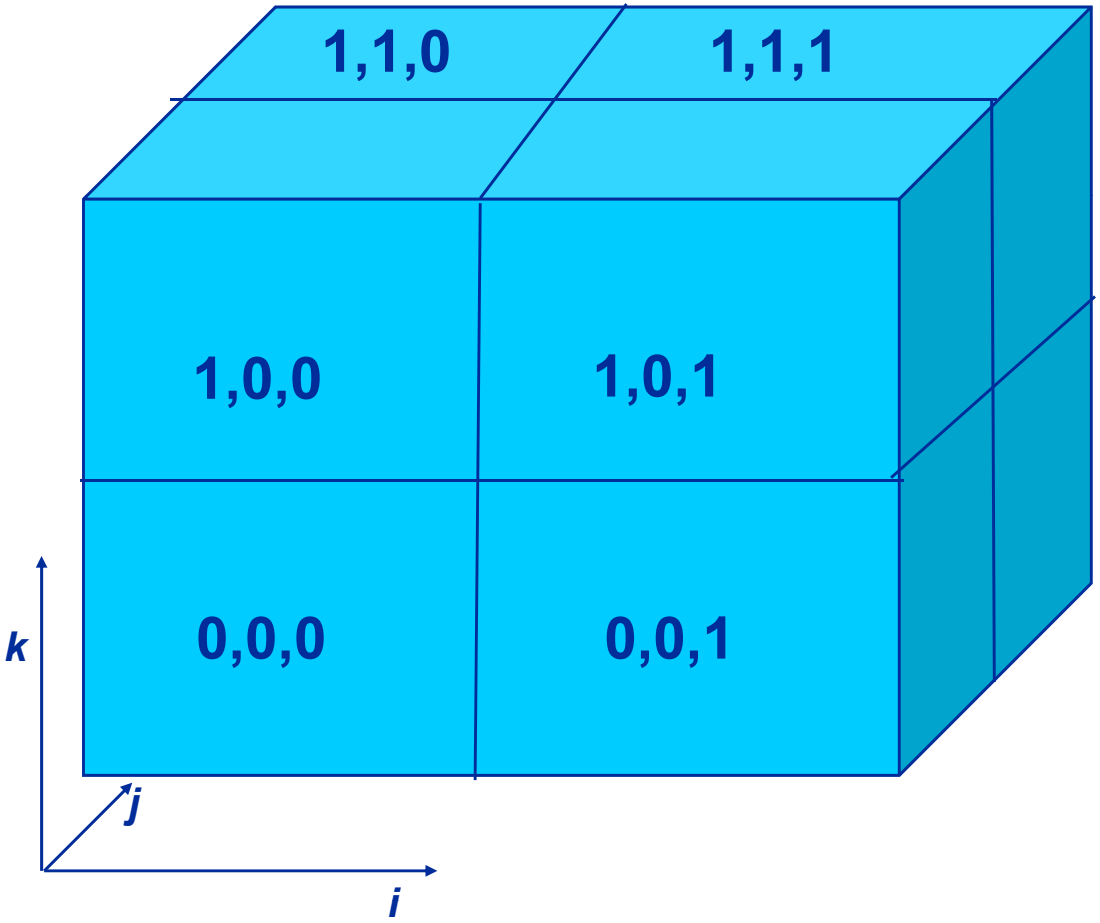


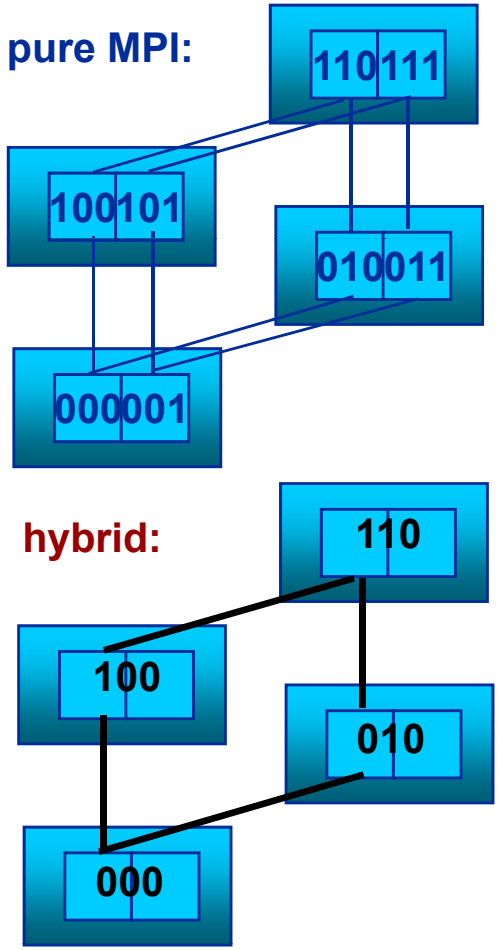
# Hybrid task mode in action

… 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



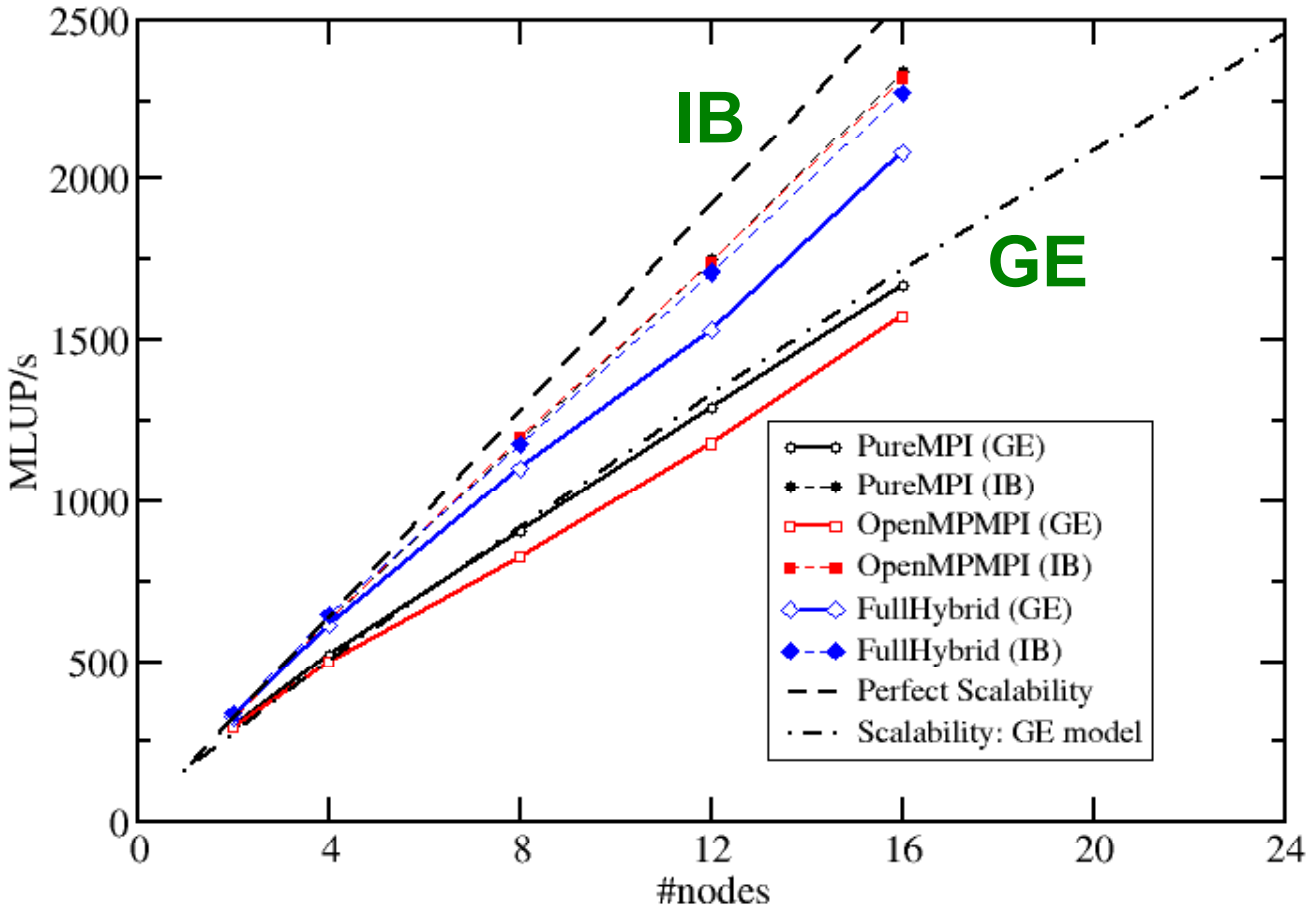



# Performance Data for 3D MPI/hybrid Jacobi

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



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




## Performance model

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

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

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

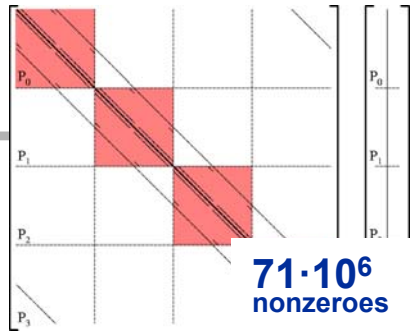
$P_0$ = 150 MLUP/s
BW(GE) = 100 MByte/s

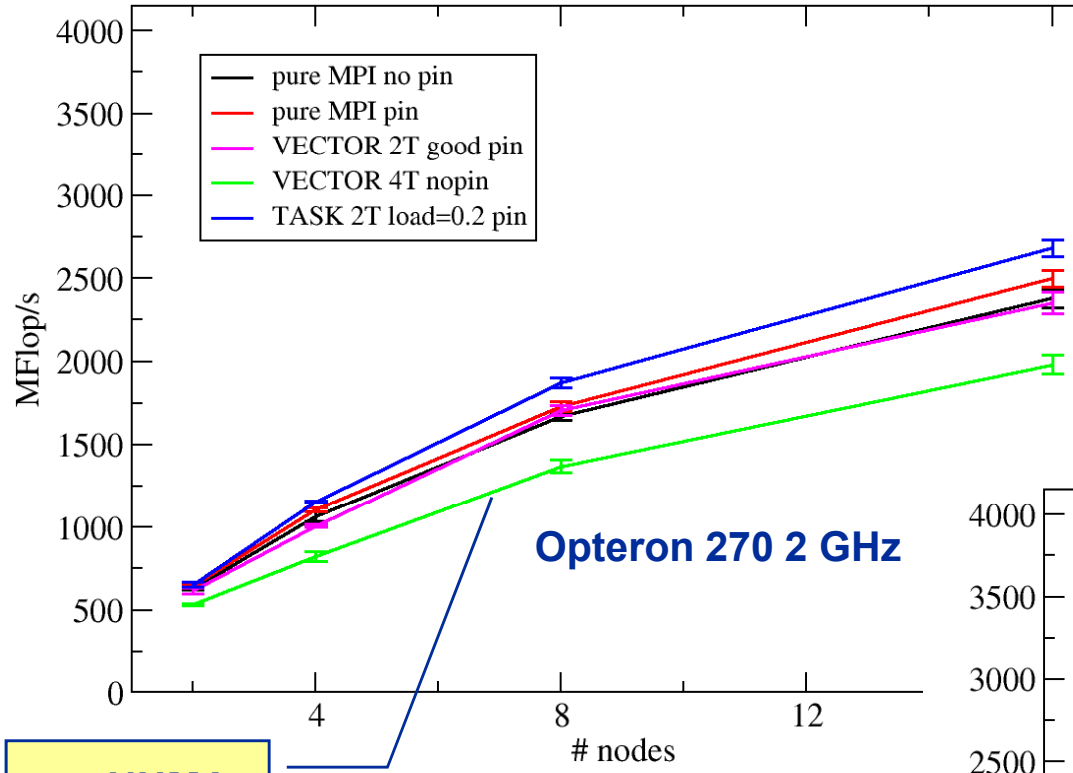
$V_{data}$ = Data volume of halo exchange

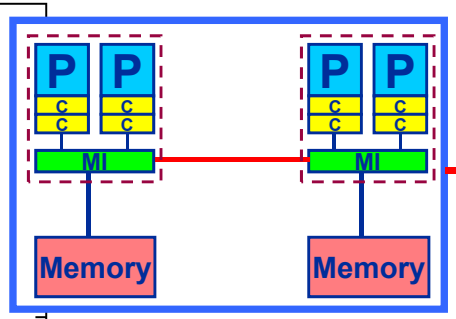
**Performance estimate (GE) for *n* nodes:**

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

# JDS Sparse MVM:
## *Performance and scalability on two different platforms*

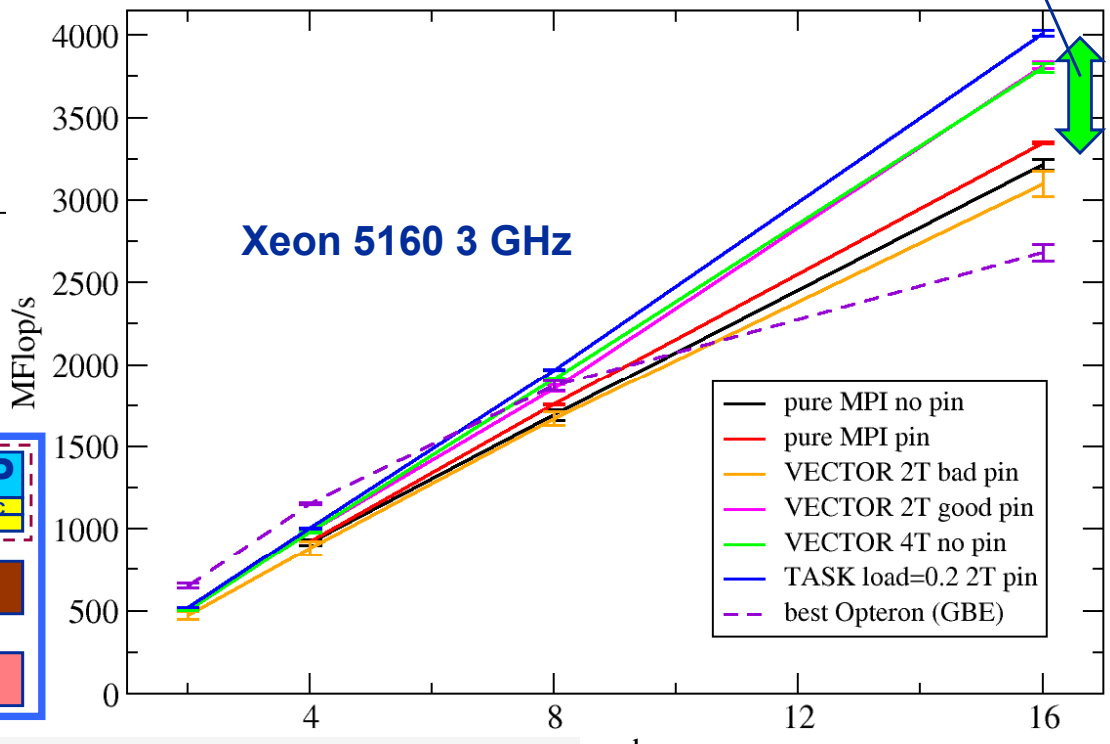


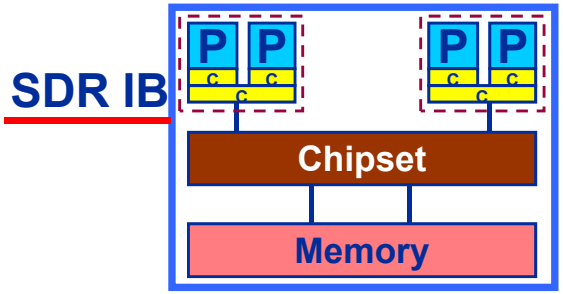




**hybrid advantage**

**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…

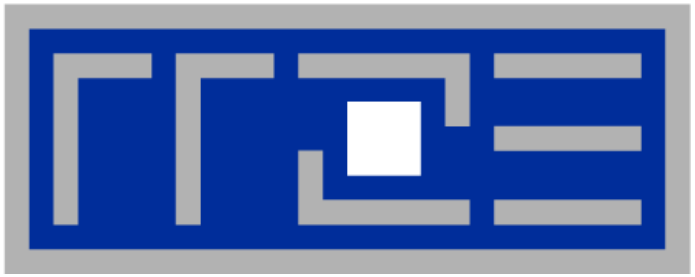
- **Nested parallelism must be supported by the compiler**
  - Probably less of a problem today
- **You don't know what actually happens when starting a new team**
  - ccNUMA page placement?
  - Thread-core affinity?
- **Alternatives:**
  - Use manual work distribution
    - This is somewhat clumsy, but well "wrappable"
    - More importantly, it is *static* (no advanced scheduling options)
  - Use OpenMP 3.0 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

# Conclusions

- **Hybrid MPI+OpenMP programming**
  - is not for the faint of heart
    - Know your basics about NUMA placement, chip/node topology, thread/core affinity
    - Leverage task mode to *really* overlap communication with computation
  - is sometimes unnecessary
    - If pure MPI scales OK, why bother?
  - may give you substantial performance boost
    - But: Try to figure out whether this is possible at all through profiling/tracing and apropriate performance models

# THANK YOU

BACKUP →

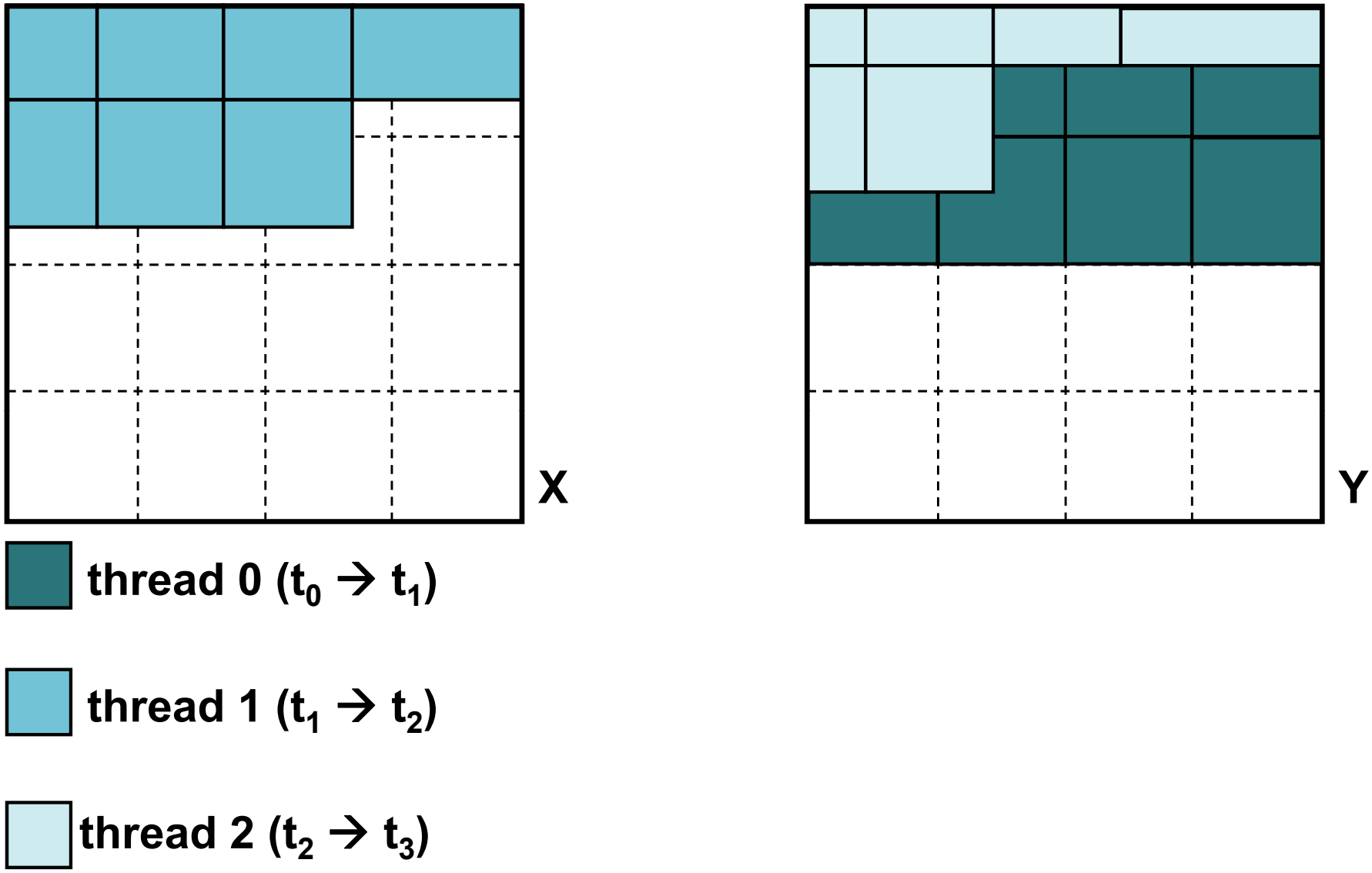


# Re-use of shared cache data and relaxed synchronization

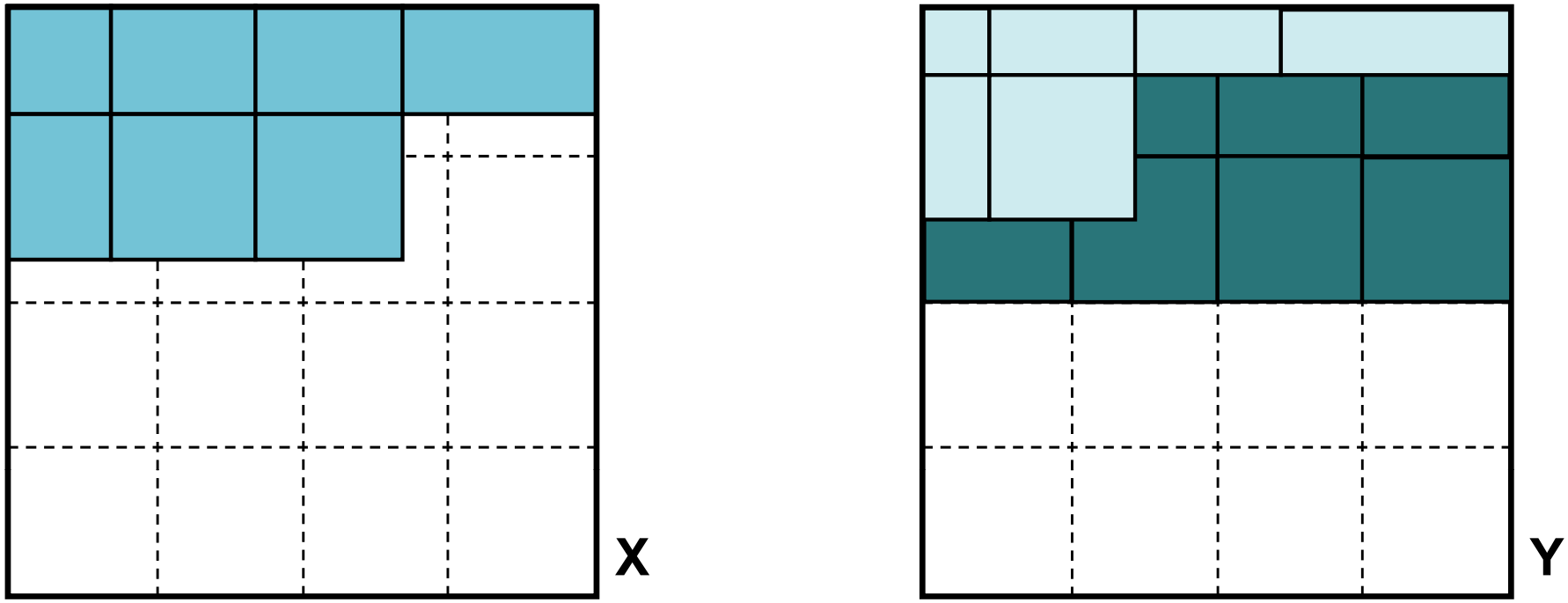
G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization.* Proc. COMPSAC 2009. Best Paper Award!

M. Wittmann, G. Hager and G. Wellein: *Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory.* Workshop on Large-Scale Parallel Processing (LSPP), IPDPS 2010, April 23rd, 2010, Atlanta, GA

# Pipelined temporal blocking



# Pipelined temporal blocking




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

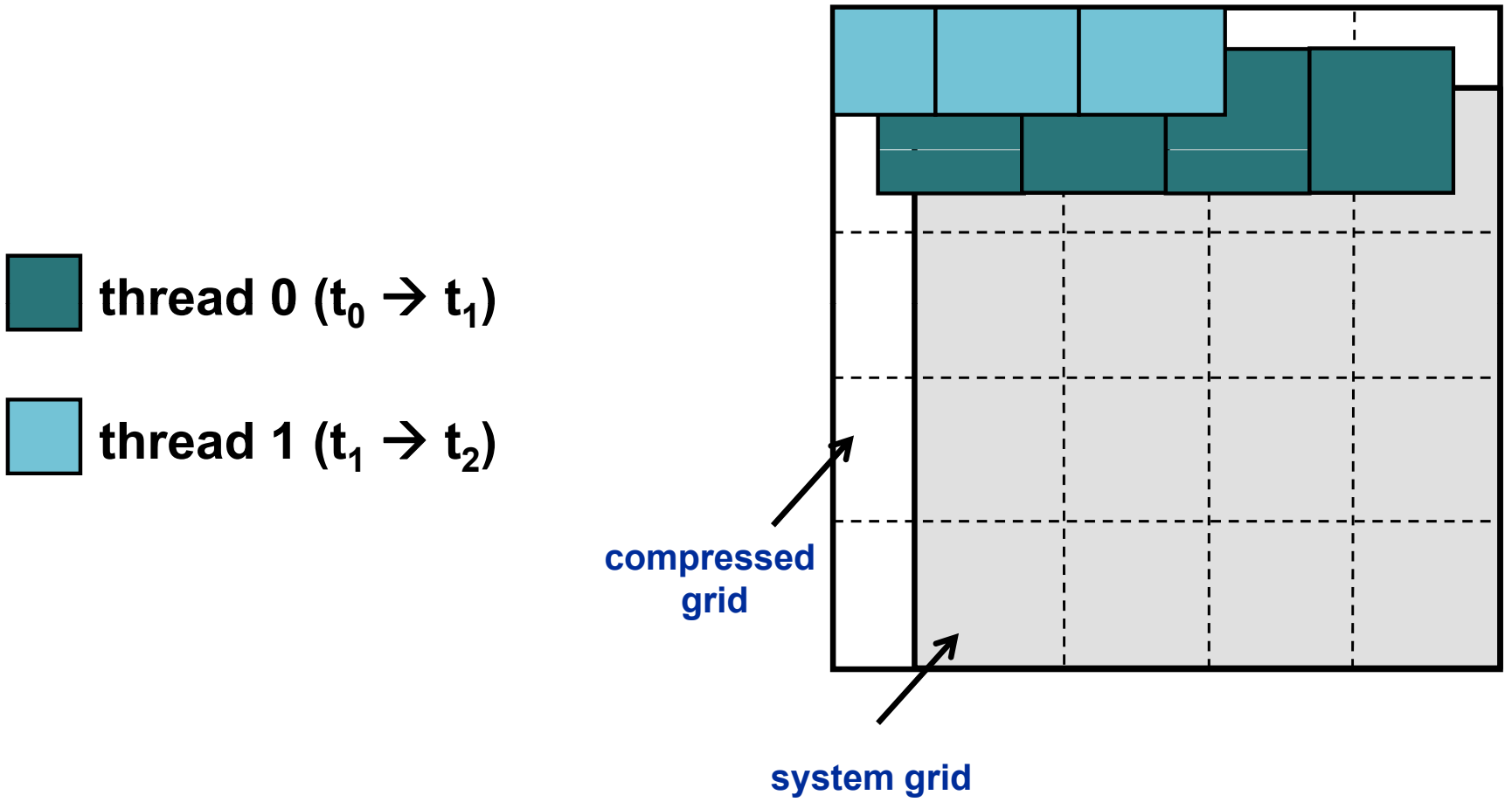
## Advantages

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

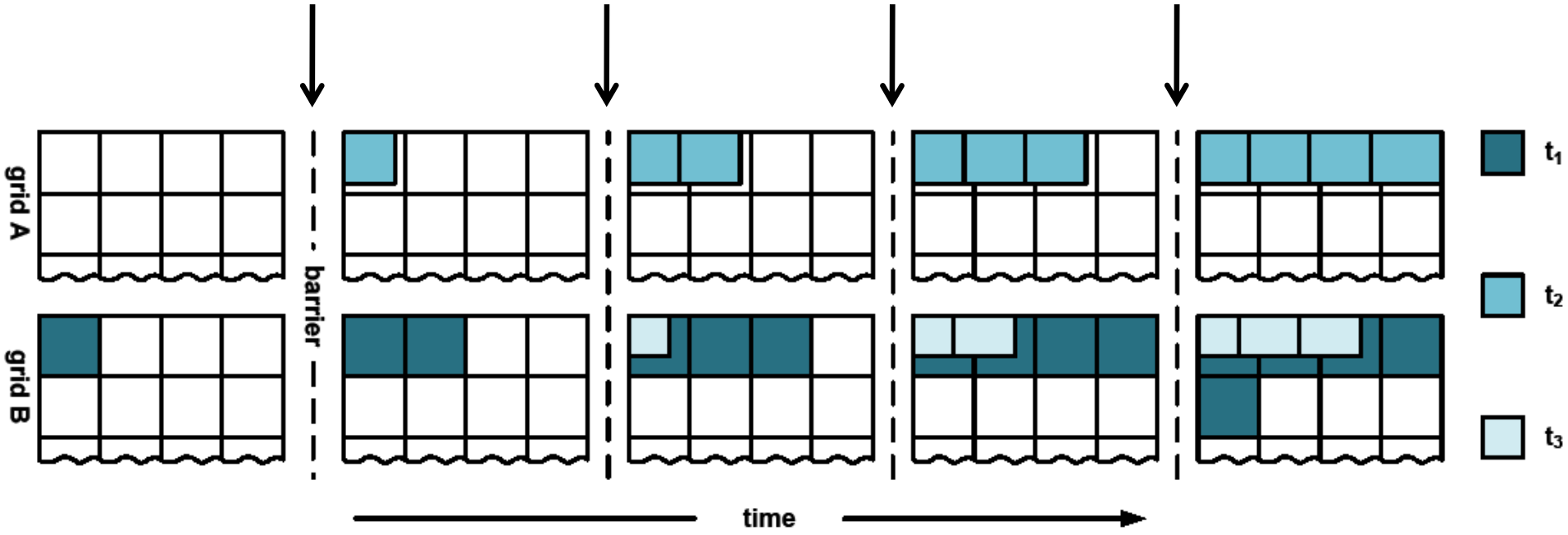
## Drawbacks

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

# Pipelined temporal blocking
## *with compressed Grid*

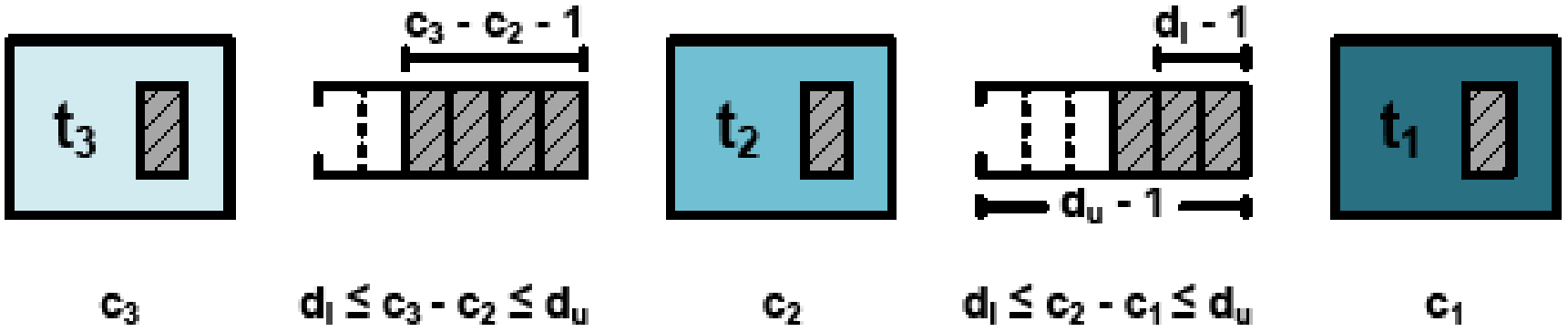


# Pipelined temporal blocking



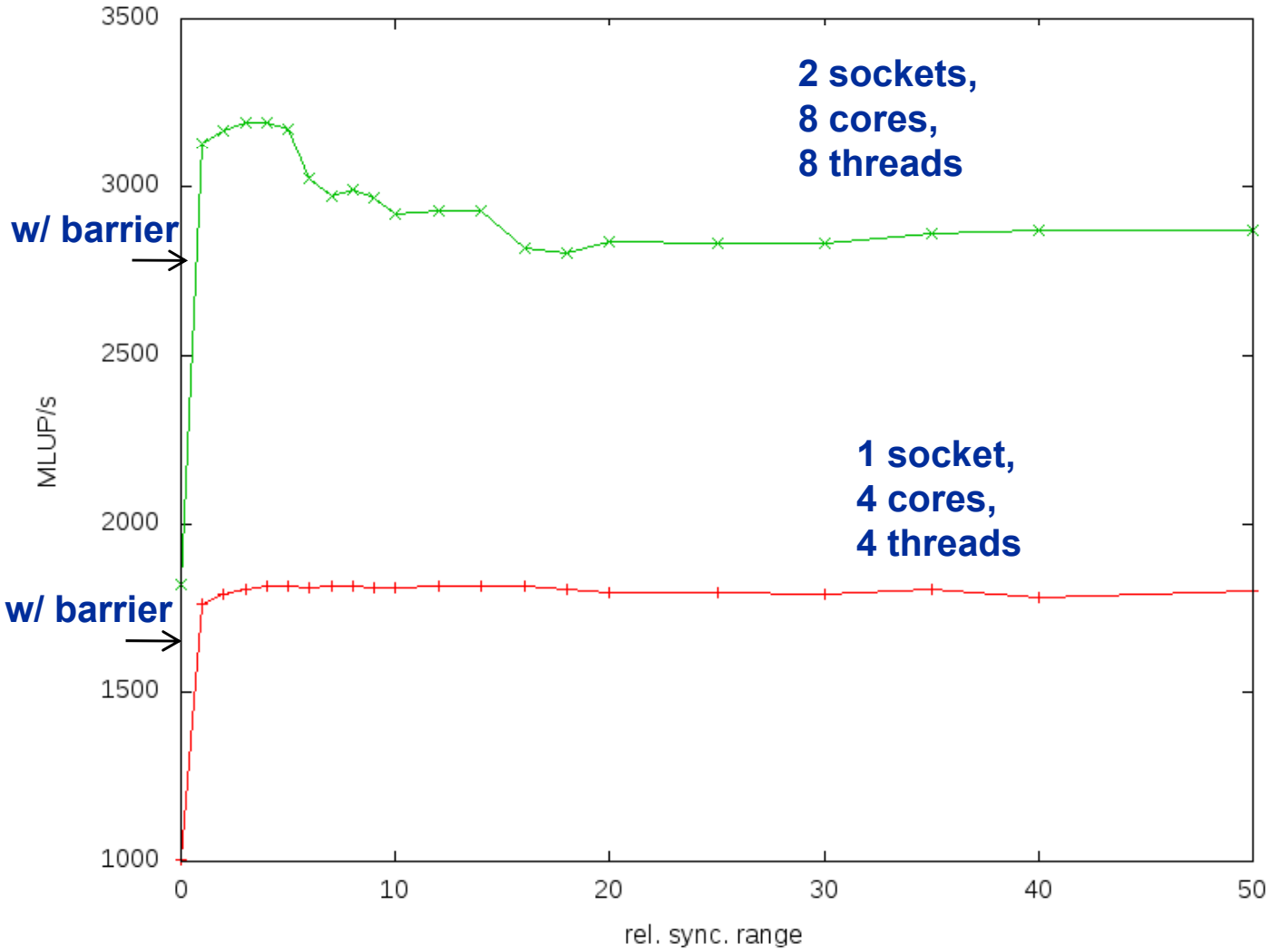

- All threads need to synchronize after finishing T iterations on their current tile
- Synchronization gets more expensive with increasing number of threads

# Relaxed Synchronisation

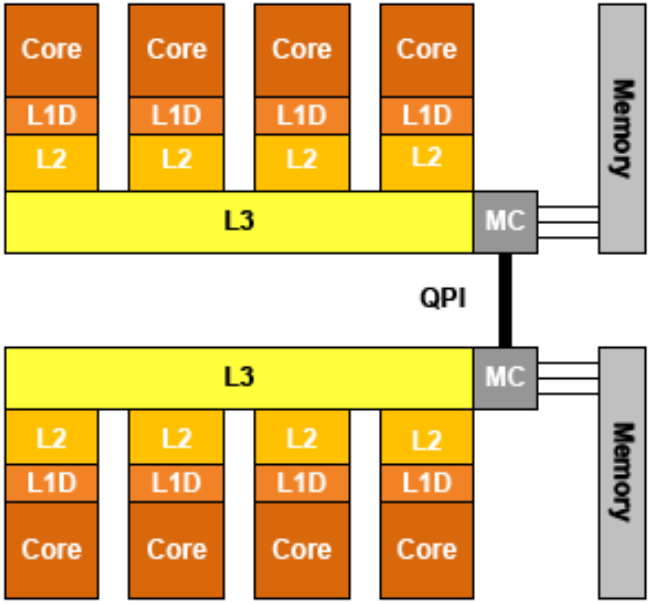



- Every thread $t_i$ only increments its own counter $c_i$
- Thread $t_i$ has a minimal distance $d_l$ to its preceding thread $t_{i-1}$
- Thread $t_i$ has a maximal distance $d_u$ to its following thread $t_{i+1}$
- Two threads have at least $d_l$ and at most $d_u$ tiles between them

# Performance with different looseness

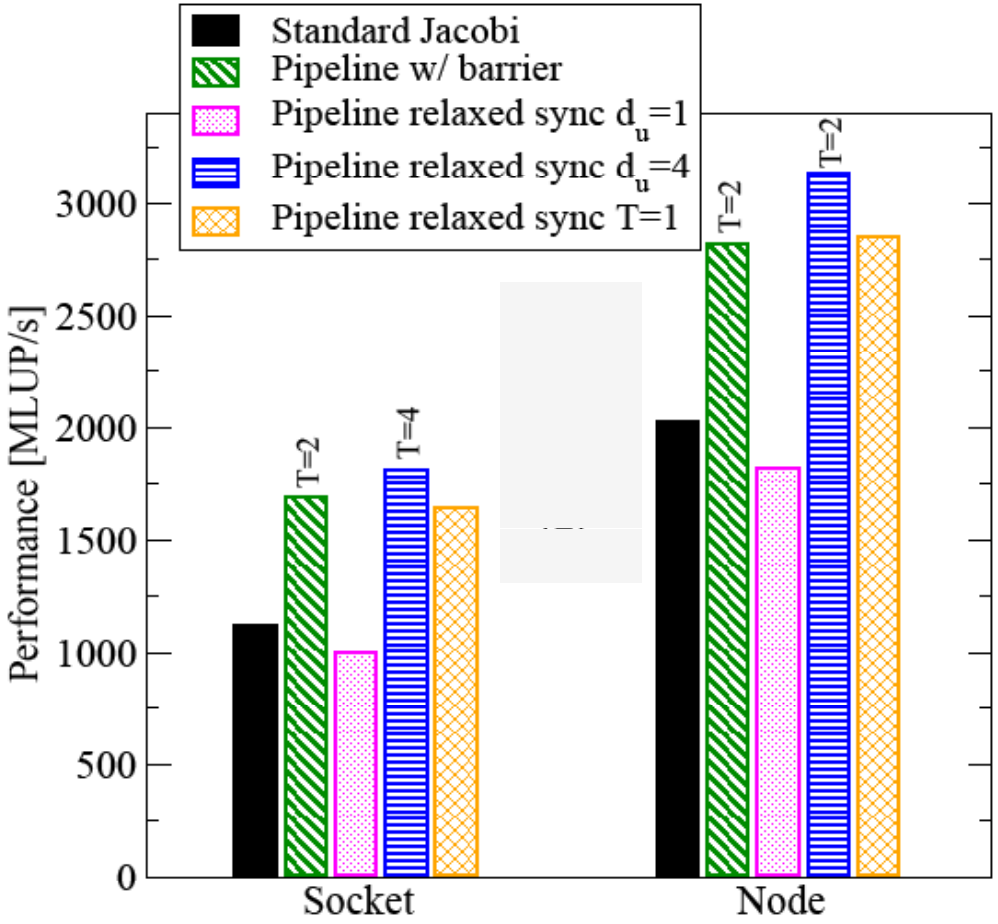



- Intel Nehalem 2.66 GHz
- 4 cores/socket
- 2 HW threads/core



# Performance Results on TinyBlue

## *Single Node*




**System size: $600^3$**