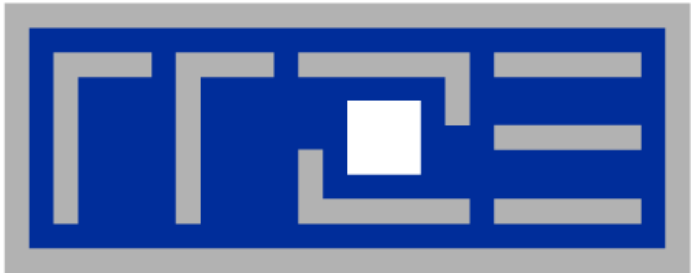
Friedrich-Alexander-Universität
Erlangen-Nürnberg



Ingredients for good parallel performance on multicore-based systems

Georg Hager[a] and Gerhard Wellein[a,b]

[a]HPC Services, Erlangen Regional Computing Center (RRZE)

[b]Department for Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg

PPoPP11 Tutorial
Feb 13th, 2011, San Antonio, TX

Tutorial outline

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **New chances with multicore hardware**
  - Wavefront parallelization of stencil codes
  - Explicit comm/calc overlap in sparse MVM
- **Summary**
- **Appendix**

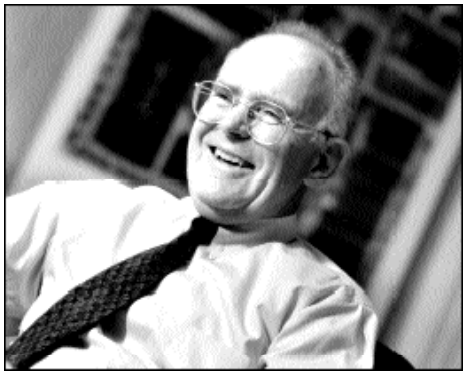
Tutorial outline

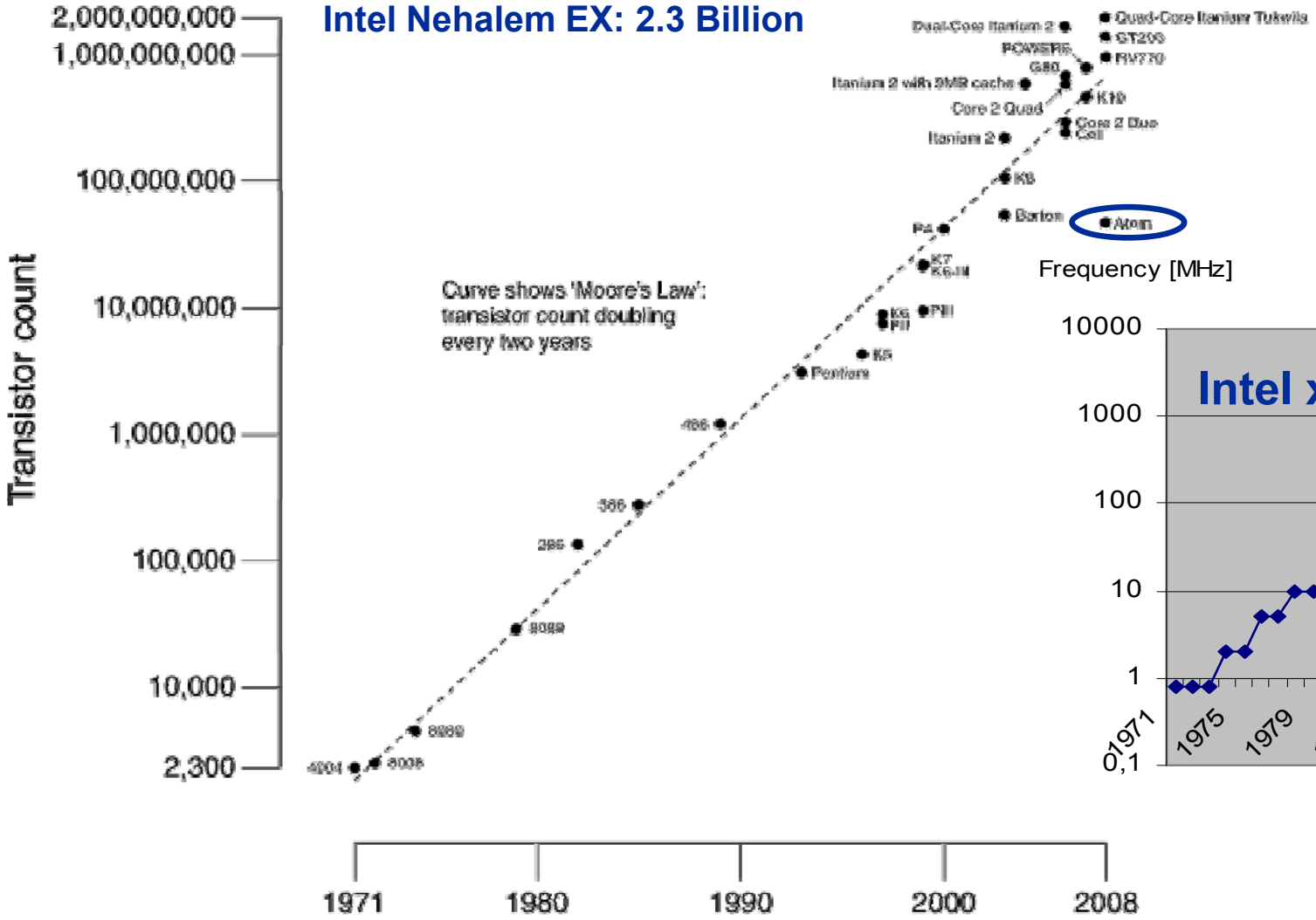
- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **New chances with multicore hardware**
  - Wavefront parallelization of stencil codes
  - Explicit comm/calc overlap in sparse MVM
- **Summary**
- **Appendix**

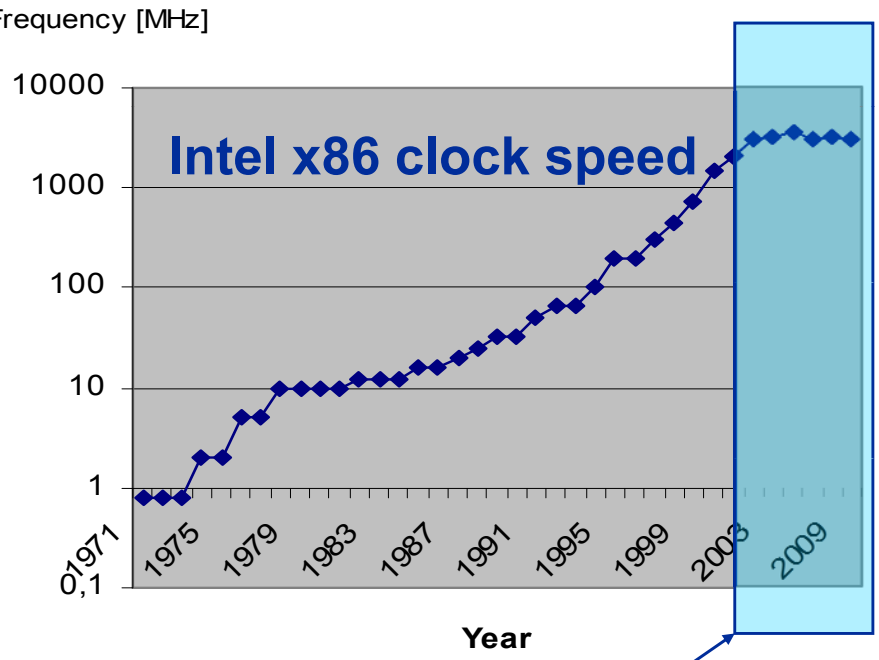
Welcome to the multi-/manycore era

The free lunch is over: But Moore's law continues

- **In 1965 Gordon Moore claimed:**
  #transistors on chip doubles every ≈24 months



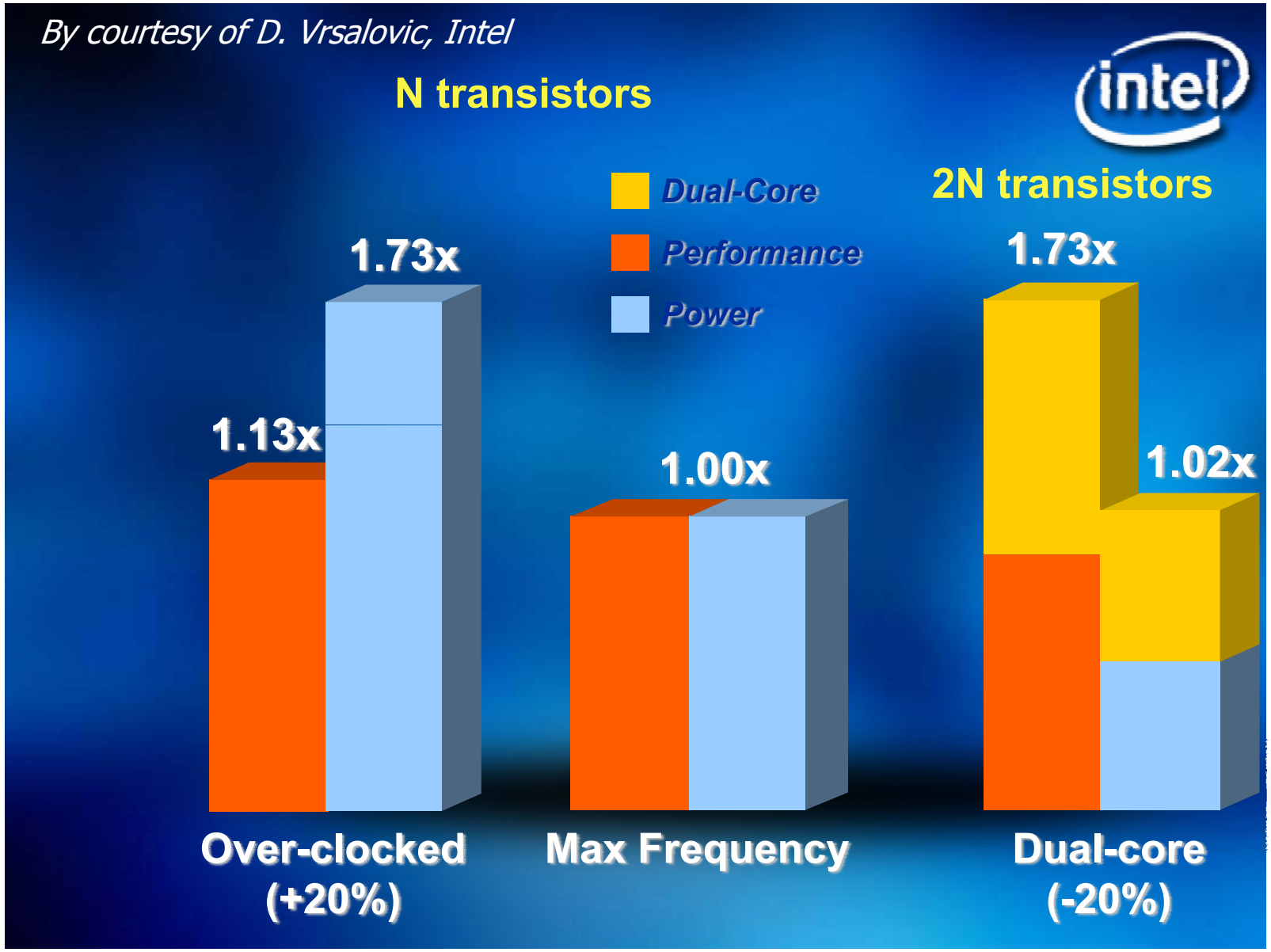




- We are living in the multicore era → Is really everyone aware of that?

Welcome to the multi-/manycore era

The game is over: But Moore's law continues



Power envelope:

Max. 95–130 W

Power consumption:

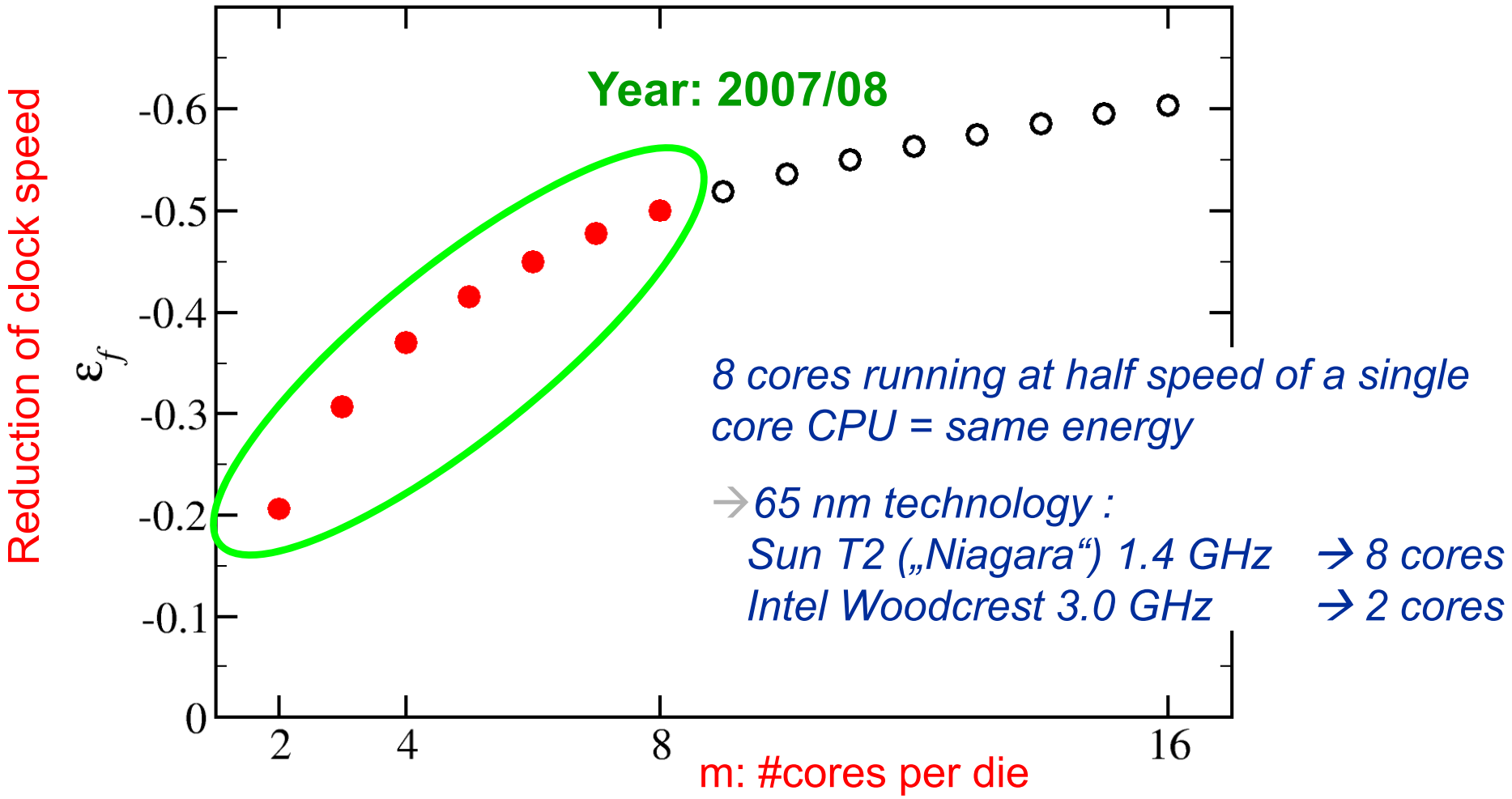
$P = f * (V_{core})^2$

V_{core} ~ 0.9–1.2 V

Same process technology:

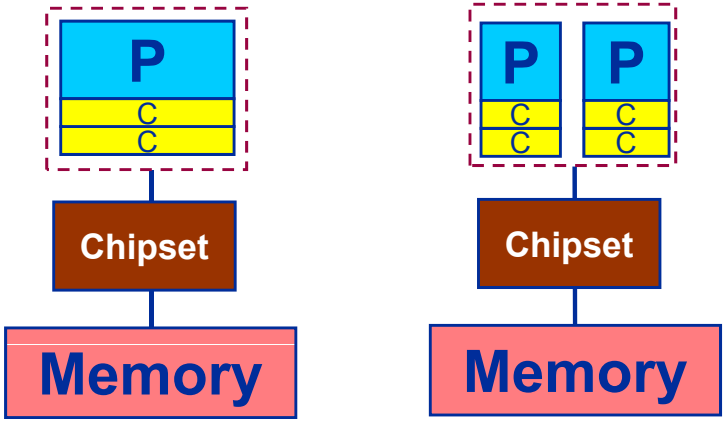
$P \sim f^3$

- **Required relative frequency reduction to run m cores (m times transistors) on a die at the same power envelope**

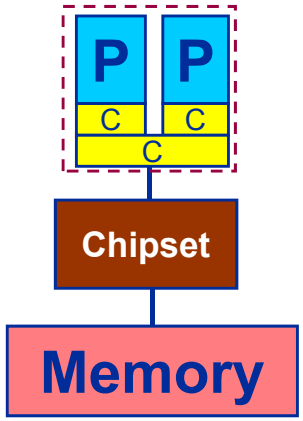


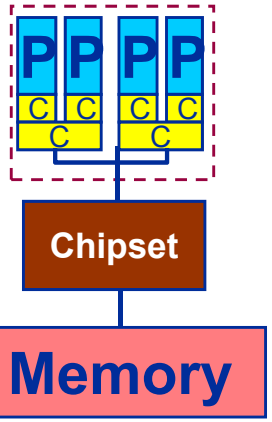
The x86 multicore evolution so far

Intel Single-Dual-/Quad-/Hexa-/-Cores (one-socket view)



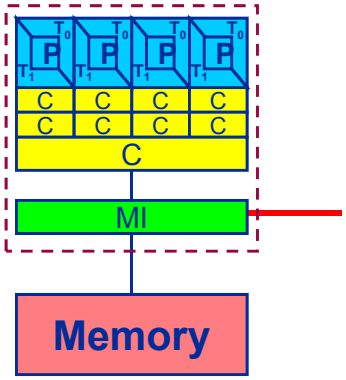
Woodcrest
"Core2 Duo"





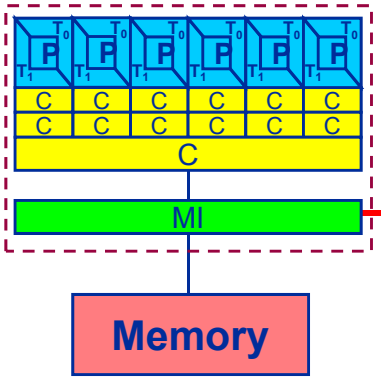
Harpertown
"Core2 Quad"

Hyperthreading/SMT is back!



Nehalem EP
"Core i7"

45 nm



Westmere EP

32 nm

2011:
"Sandy Bridge"

SSE → AVX

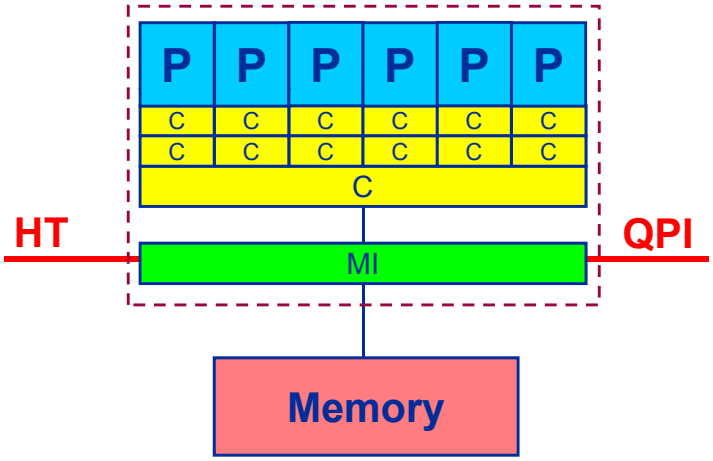
128 Bit → 256 Bit

Shared outer-level cache

- Fast data transfer
- Fast thread synchronisation

- Data Coherency!
- Increased intra-cache traffic?
- Scalable bandwidth?
- MPI parallelization?

AMD Opteron Istanbul
6 cores @ 2.8 GHz
L1: 64 KB
L2: 512 KB
L3: 6 MB
2 X DDR2-800 → 12.8 GB/s
HT2000 → 8 GB/s/dir

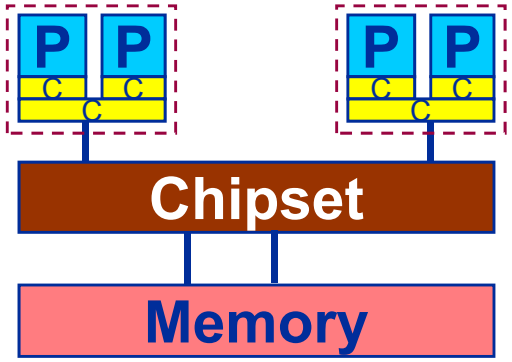


Memory bottleneck!

Intel Xeon Westmere
6 cores @ 2.93 GHz
L1: 32 KB
L2: 256 KB
L3: 12MB
3 X DDR3-1333 → 31.8 GB/s
2 X QPI6.4 → 12.8 GB/s/dir

Dual-socket Intel "Core2" node:

Yesterday



Uniform Memory Architecture (UMA):

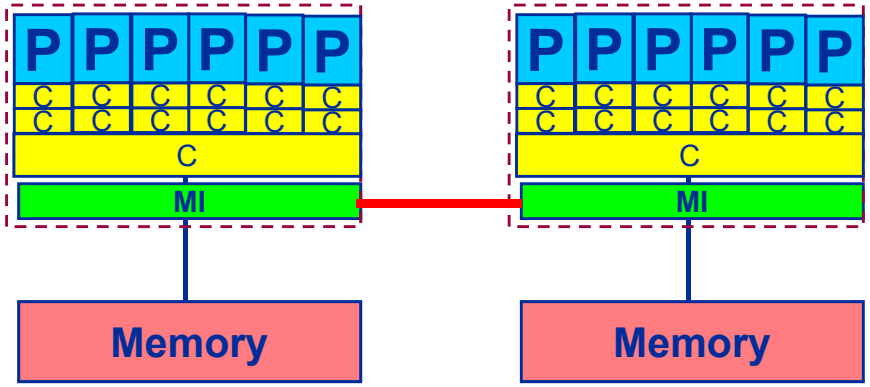
Flat memory ; symmetric MPs

But: system "anisotropy"

Shared Address Space within the node!

Dual-socket AMD (Istanbul) / Intel (Westmere) node:

Today

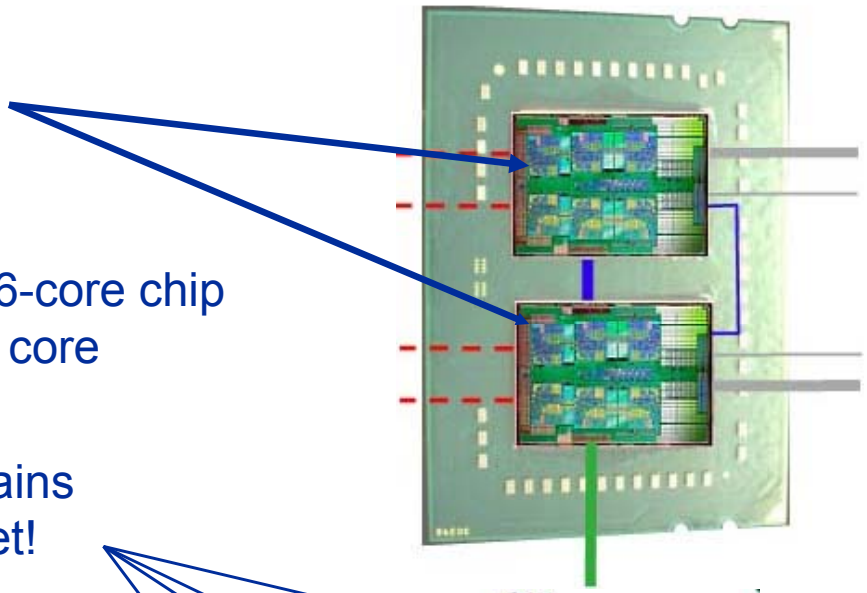


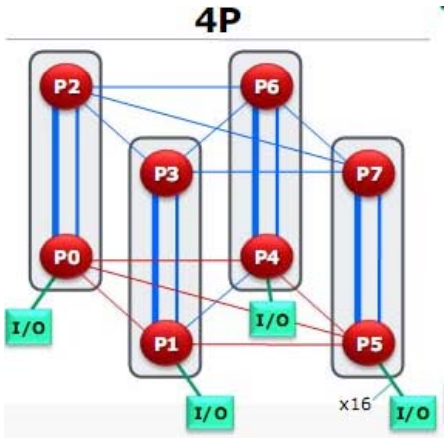
Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

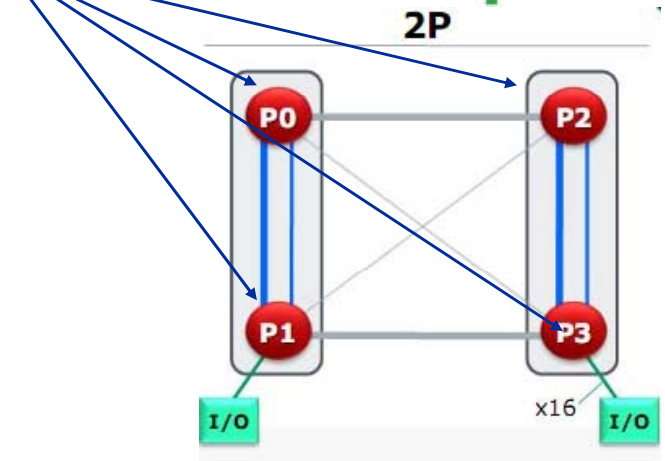
HT / QPI provide scalable bandwidth at the expense of ccNUMA architectures:
Where does my data finally end up?

Back to the 2-chip-per-case age:
AMD Magny-Cours – a 2x6-core socket

- **AMD: "Magny-Cours"**
  - 12-core socket comprising two 6-core chips connected via 1.5 HT links
  - Main memory access: → 2 DDR3-Channels per 6-core chip
    → 1/3 DDR3-Channel per core
  - 2 socket server → 4 memory locality domains
    → ccNUMA within a socket!
  - 4 socket server:







  - Network balance (QDR+2P Magny Cours) ~ 240 GF/s / 3 GB/s = 80 F/B
    (2003: Intel Xeon DP 2.66 GHz + GBit ~ 10 GF/s / 0.12 GB/s = 80 F/B)

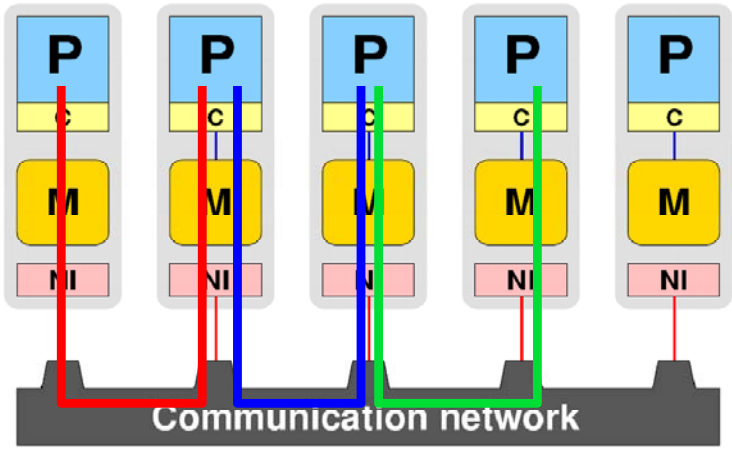
Parallel programming models
on multicore multisocket nodes

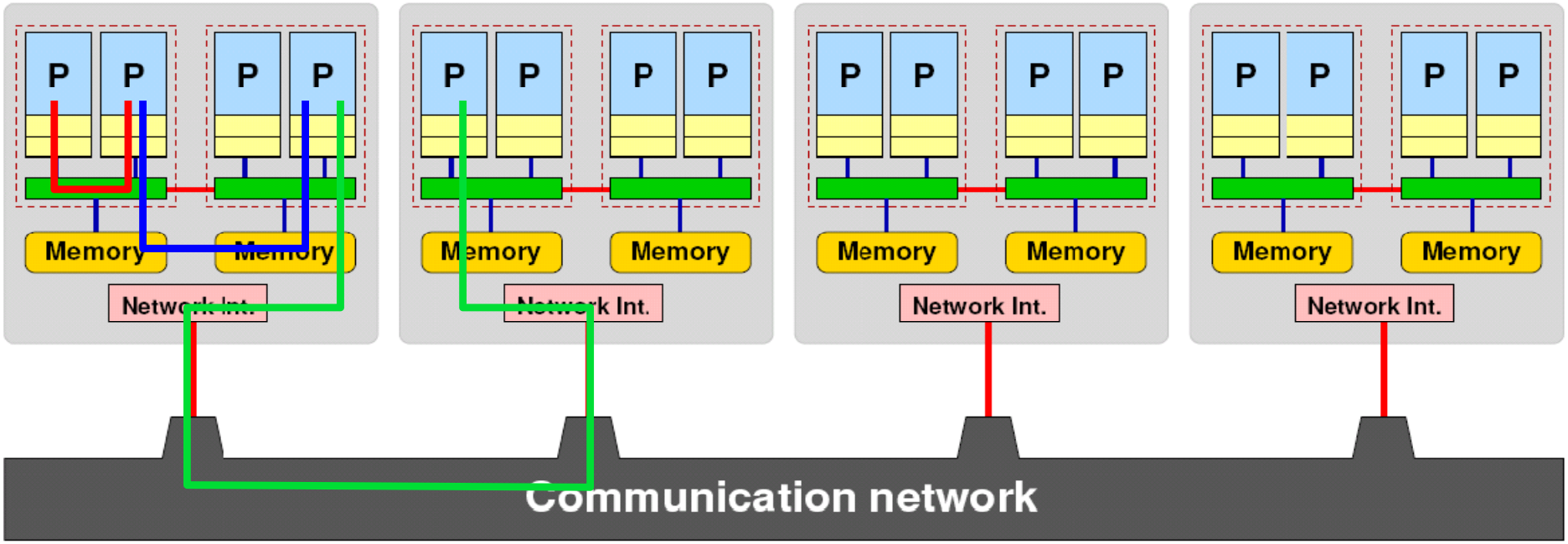
- **Shared-memory (intra-node)**
  - Good old MPI (current standard: 2.2)
  - OpenMP (current standard: 3.0)
  - POSIX threads
  - Intel Threading Building Blocks
  - Cilk++, OpenCL, StarSs,… you name it
- **Distributed-memory (inter-node)**
  - MPI (current standard: 2.2)
  - PVM (gone)
- **Hybrid**
  - Pure MPI
  - MPI+OpenMP
  - MPI + any shared-memory model

All models require awareness of topology and affinity issues for getting best performance out of the machine!

Parallel programming models:
Pure MPI

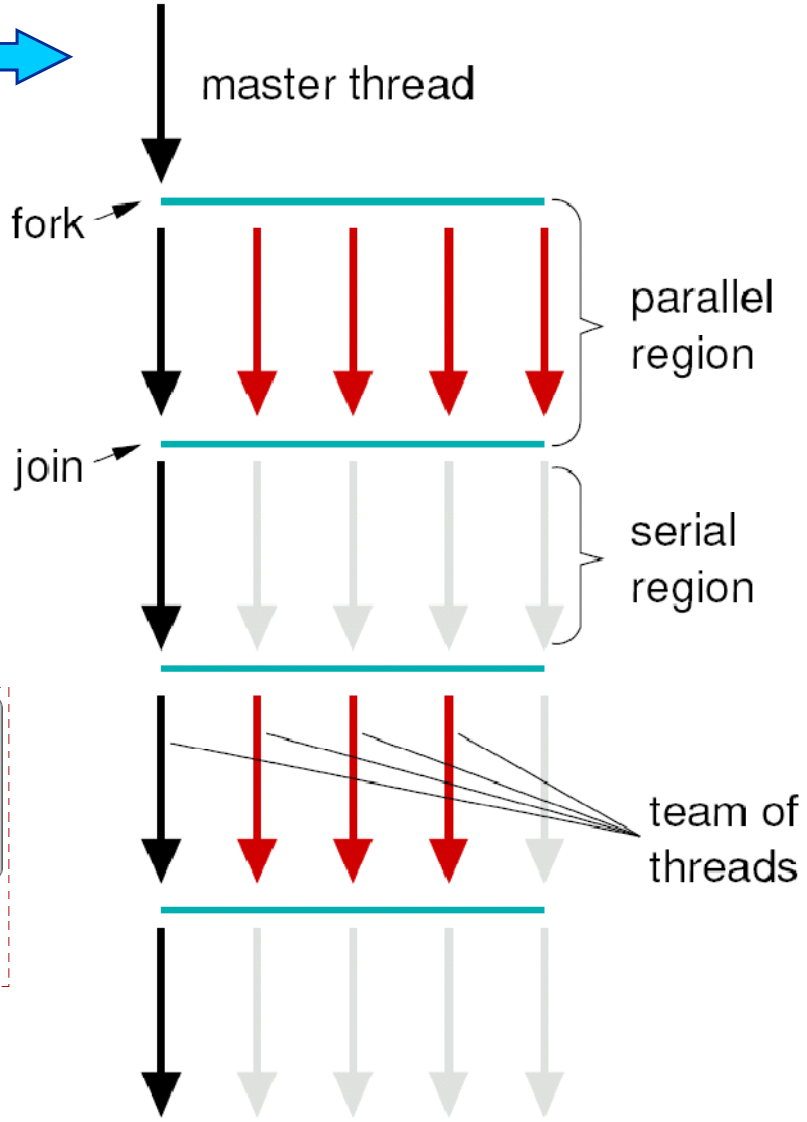
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

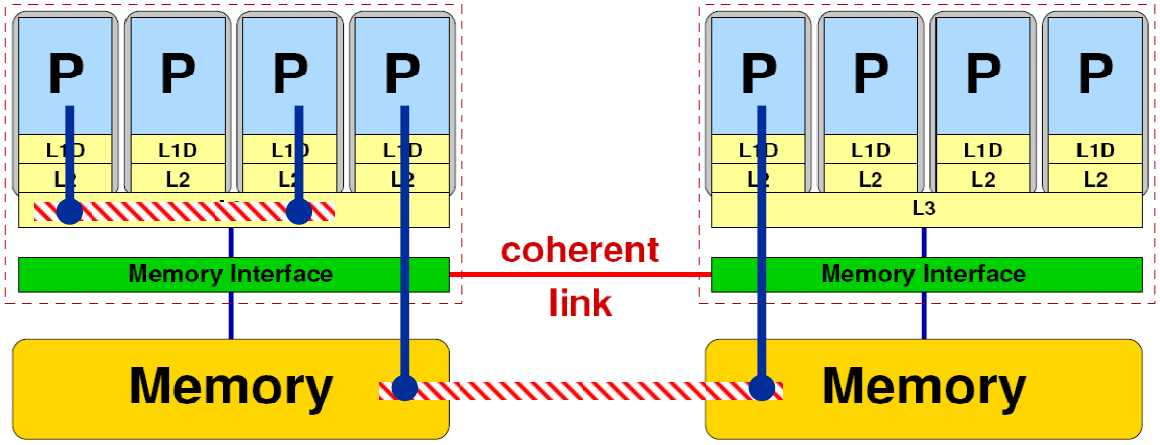




Parallel programming models:
Pure threading on the node

- **Machine structure is invisible to user**
  - → Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,…) should know about the details
- **Performance issues**
  - Synchronization overhead
  - Memory access
  - Node topology





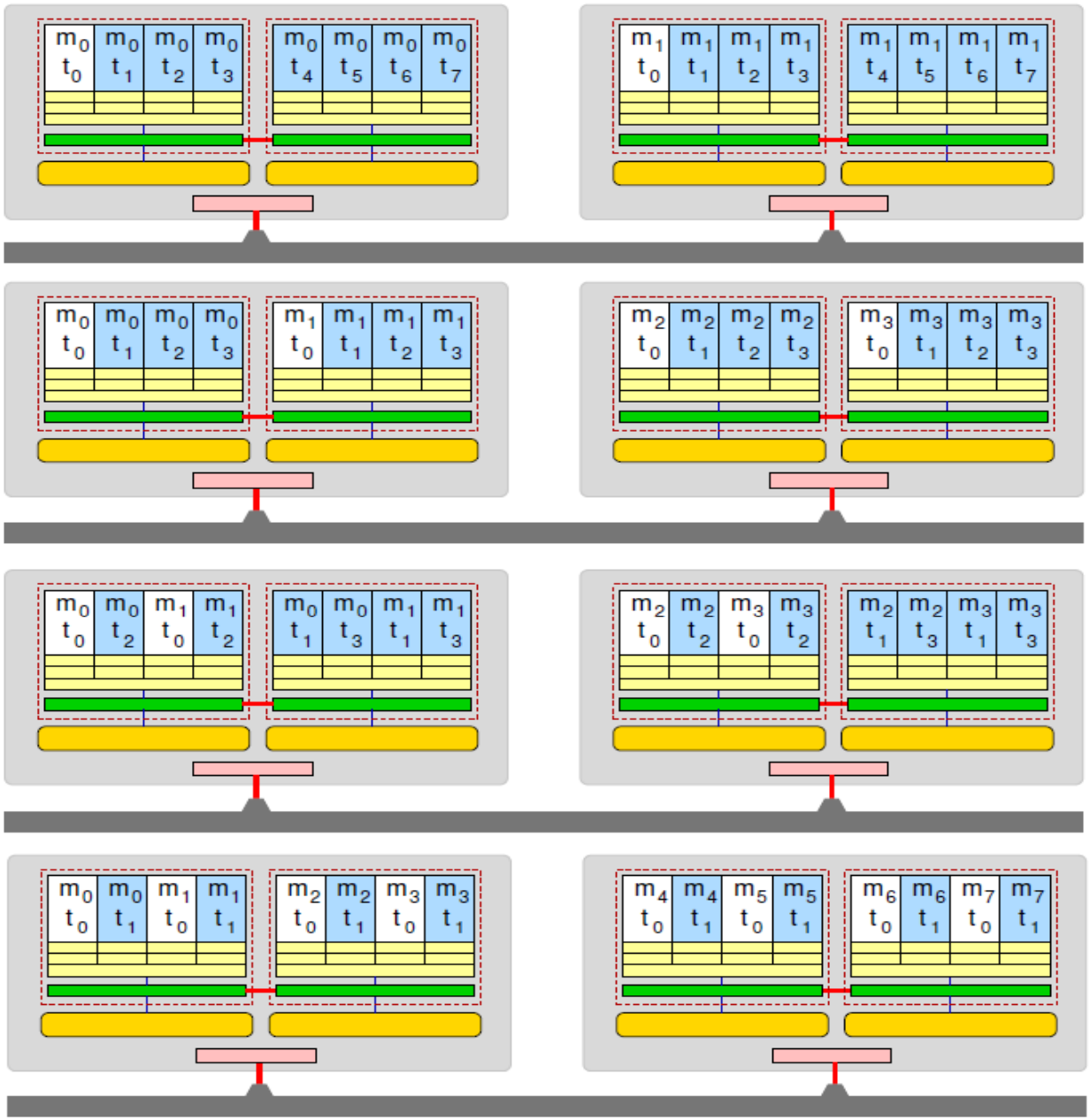
Parallel programming models:
Hybrid MPI+OpenMP on a multicore multisocket cluster

One MPI process / node

One MPI process / socket:
OpenMP threads on same socket: **"blockwise"**

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

Two MPI processes / socket
OpenMP threads on same socket



Section summary: What to take home

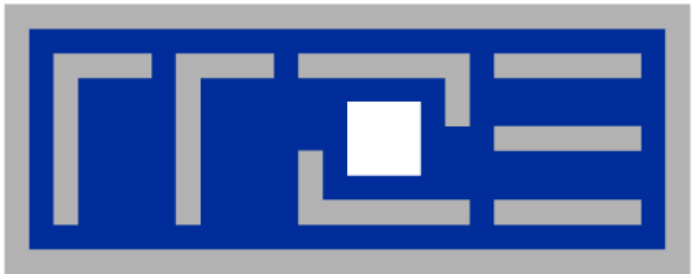
- **Multicore is here to stay**
  - Shifting complexity form hardware back to software
- **Increasing core counts**
  - 4-12 today, 16-32 tomorrow?
  - x2 or x4 per cores node
- **Shared vs. separate caches**
  - Complex chip/node topologies

- **UMA is practically gone; ccNUMA will prevail**
  - “Easy” bandwidth scalability, but programming implications (see later)
  - Bandwidth bottleneck prevails on the socket

- **Programming models that take care of those changes are still in heavy flux**
  - We are left with MPI and OpenMP for now
  - This is complex enough, as we will see…

Tutorial outline

- Introduction
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- Impact of processor/node topology on program performance
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- New chances with multicore hardware
  - Wavefront parallelization of stencil codes
  - Explicit comm/calc overlap in sparse MVM
- Summary
- Appendix



Probing node topology

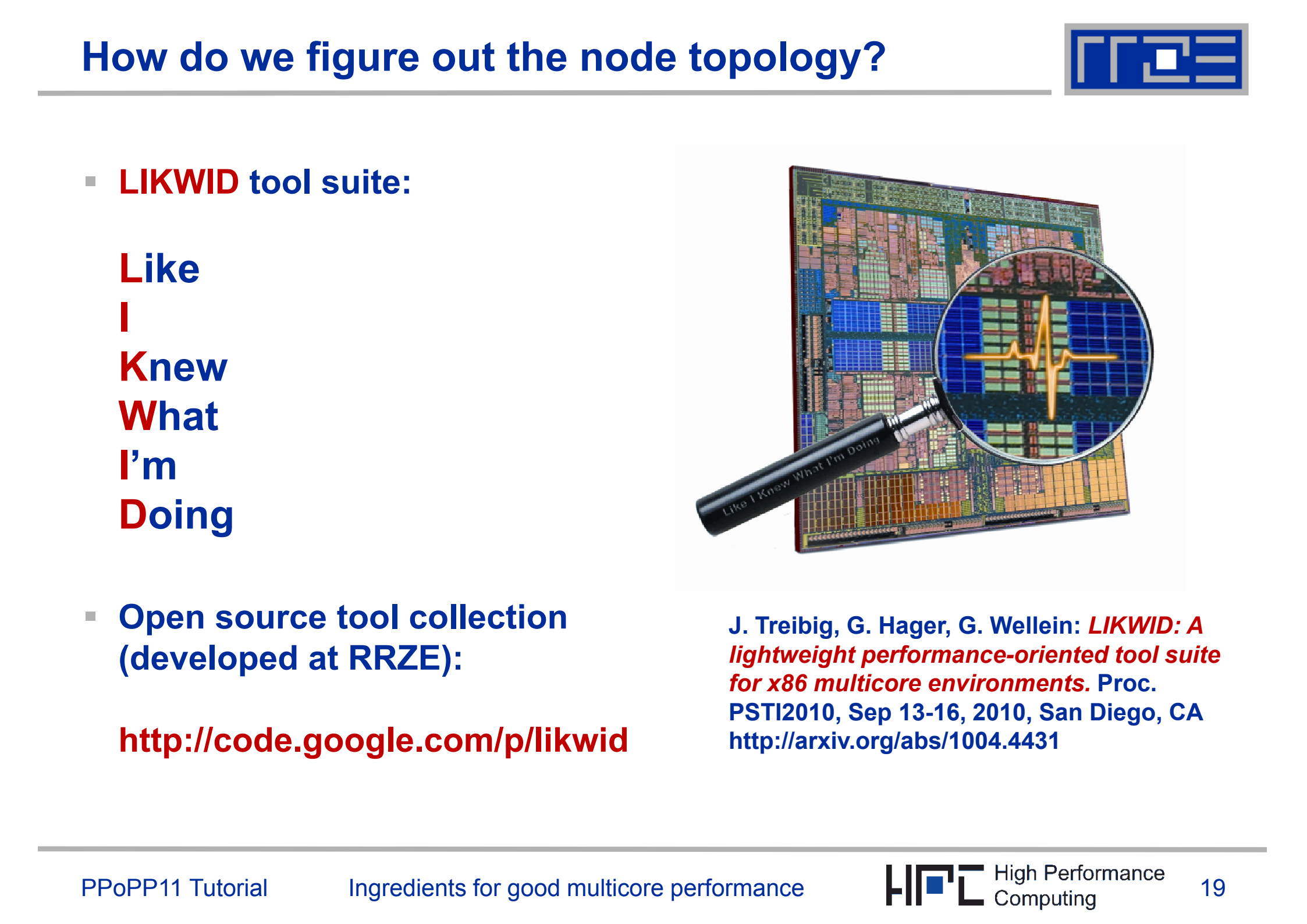
- Standard tools
- likwid-topology
- hwloc

How do we figure out the node topology?

- **Topology =**
  - Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads (“logical cores”) share a physical core?
- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers may change across kernels and BIOSes even on identical hardware
  - `numactl --hardware` prints ccNUMA node information ➔
  - Information on caches is harder to obtain

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: 8192 MB
node 3 free: 7840 MB
```

How do we figure out the node topology?



- **LIKWID tool suite:**

  Like
  I
  Knew
  What
  I'm
  Doing

- **Open source tool collection (developed at RRZE):**

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

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

Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - Supports current 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-bench: Low-level bandwidth benchmark generator tool
  - likwid-mpirun: mpirun wrapper script for easy LIKWID integration (alpha)

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 (more under development):**
  - Intel Core 2 (45nm + 65 nm)
  - Intel Nehalem + Westmere (Sandy Bridge in alpha phase)
  - AMD K10 (Quadcore, Hexacore, Magny Cours)
  - AMD K8
  - Linux OS, Windows port in alpha phase for likwid-pin

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
-------------------------------------------------------------
```

Thread-to-core mapping

Output of 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 )
--------------------------------------------------------------
**************************************************************
NUMA Topology
**************************************************************
NUMA domains: 2
--------------------------------------------------------------
Domain 0:
 Processors:  0 1 2 3 4 5 6 7
Memory: 5182.37 MB free of total 6132.83 MB
--------------------------------------------------------------
Domain 1:
 Processors:  8 9 10 11 12 13 14 15
Memory: 5568.5 MB free of total 6144 MB
--------------------------------------------------------------
```

ccNUMA domain info (analogous to numactl –hardware)

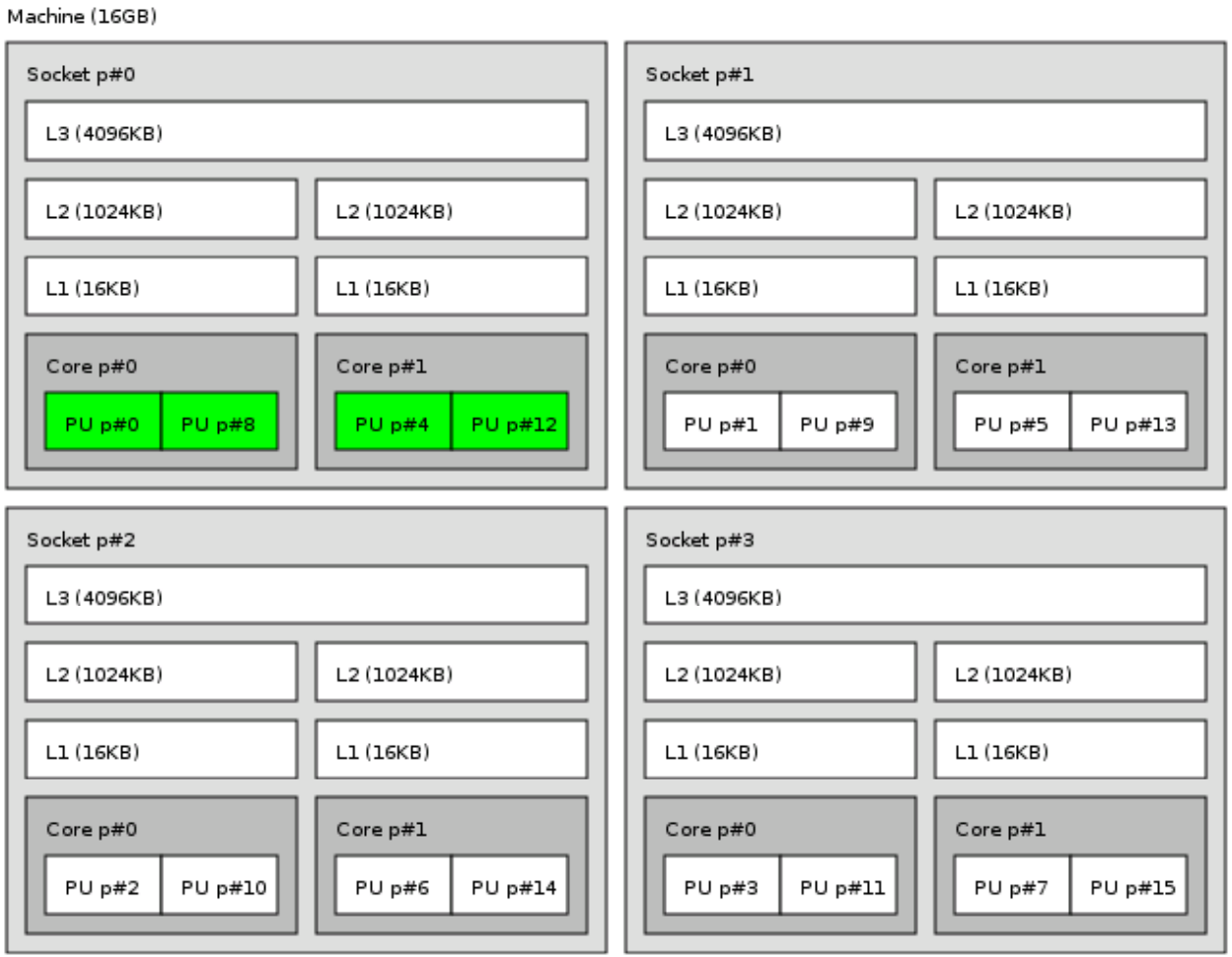
Output of likwid-topology

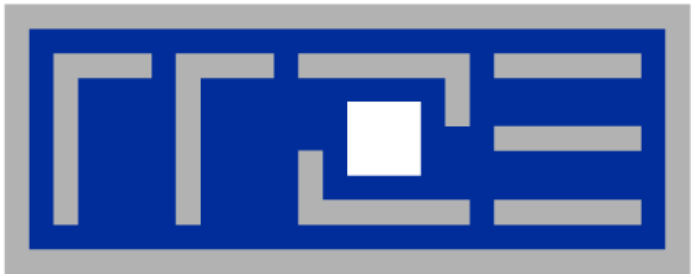
- … and also try the ultra-cool `-g` option!

```
Socket 0:
+-------------------------------------+
| +------+ +------+ +------+ +------+ |
| |  0  1| |  2  3| |  4  5| |  6  7| |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| |  32kB| |  32kB| |  32kB| |  32kB| |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| | 256kB| | 256kB| | 256kB| | 256kB| |
| +------+ +------+ +------+ +------+ |
| +---------------------------------+ |
| |                8MB              | |
| +---------------------------------+ |
+-------------------------------------+
Socket 1:
+-------------------------------------+
| +------+ +------+ +------+ +------+ |
| |  8  9| |10  11| |12  13| |14  15| |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| |  32kB| |  32kB| |  32kB| |  32kB| |
| +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ |
| | 256kB| | 256kB| | 256kB| | 256kB| |
| +------+ +------+ +------+ +------+ |
| +---------------------------------+ |
| |                8MB              | |
| +---------------------------------+ |
+-------------------------------------+
```

hwloc

- Alternative: http://www.open-mpi.org/projects/hwloc/
- Successor to (and extension of) PLPA, part of OpenMPI development
- Comprehensive API and command line tool to extract topology info
- Supports several OSs and CPU types
- Pinning API available





Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin

Generic thread/process-core affinity under Linux

- `taskset [OPTIONS] [MASK | -c LIST ] \`
  `[PID | command [args]...]`

- **binds processes/threads to a *set of CPUs*. Examples:**

```
taskset -c 0,2 mpirun -np 2 ./a.out # doesn't always work
taskset 0x0006 ./a.out
taskset -c 4 33187
```

- **Processes/threads can still move within the set!**
- **Alternative: let process/thread bind itself by executing syscall**

```
#include <sched.h>
int sched_setaffinity(pid_t pid, unsigned int len,
                      unsigned long *mask);
```

- **Disadvantage: which CPUs should you bind to on a non-exclusive machine?**

- **Still of value on multicore/multisocket cluster nodes, UMA or ccNUMA**

Generic thread/process-core affinity under Linux

- **Complementary tool: `numactl`**

  Example: `numactl --physcpubind=0,1,2,3 command [args]`
  Bind process to specified physical core numbers

  Example: `numactl --cpunodebind=1 command [args]`
  Bind process to specified ccNUMA node(s)

- **Many more options (e.g., interleave memory across nodes)**
  - → see section on ccNUMA optimization

- **Diagnostic command (see earlier):**
  `numactl --hardware`

- **Again, this is not suitable for a shared machine**

Thread/Process-core affinity ("pinning") options

- **Highly OS-dependent system calls**
  - But available on all systems

    Linux: `sched_setaffinity()`, PLPA (see below) → hwloc
    Solaris: `processor_bind()`
    Windows: `SetThreadAffinityMask()`
    …
- **Support for "semi-automatic" pinning in some compilers/environments**
  - Intel compilers > V9.1 (`KMP_AFFINITY` environment variable)
  - PGI, Pathscale, GNU
  - SGI Altix `dplace` (works with logical CPU numbers!)
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
- **Affinity awareness in MPI libraries**
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - …

Example for program-controlled affinity: Using PLPA under Linux!

Explicit Process/Thread Binding With PLPA on Linux: http://www.open-mpi.org/software/plpa/

- **Portable Linux Processor Affinity**
- **Wrapper library for `sched_*affinity()` functions**
  - Robust against changes in kernel API
- **Example for pure OpenMP: Pinning of threads**

```
#include <plpa.h>
...
#pragma omp parallel
  {
#pragma omp critical
  {
    if(PLPA_NAME(api_probe)()!=PLPA_PROBE_OK) {
        cerr << "PLPA failed!" << endl; exit(1);
  }
    plpa_cpu_set_t msk;
    PLPA_CPU_ZERO(&msk);
    int cpu = omp_get_thread_num();
    PLPA_CPU_SET(cpu,&msk);
    PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &msk);
  }
```

Pinning available?

Which core to run on?

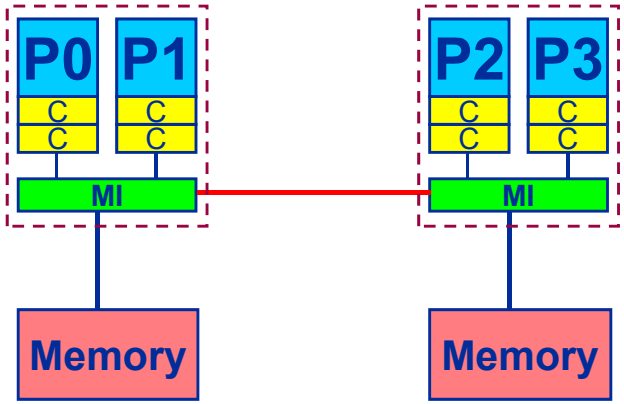
Pin “me”

Care about correct core numbering! 0…N-1 is not always contiguous! If required, reorder by a map:
`cpu = map[cpu];`

- **Similar for pure MPI and MPI+OpenMP hybrid code**

Process/Thread Binding With PLPA

- **Example for pure MPI: Process pinning**
  - Bind MPI processes to cores in a cluster of 2x2-core machines




```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
int mask = (rank % 4);
PLPA_CPU_SET(mask,&msk);
PLPA_NAME(sched_setaffinity)((pid_t)0,
                             sizeof(cpu_set_t), &msk);
```

- **Hybrid case:**

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
#pragma omp parallel
  {
    plpa_cpu_set_t msk;
    PLPA_CPU_ZERO(&msk);
    int cpu = (rank % MPI_PROCESSES_PER_NODE)*omp_num_threads
                + omp_get_thread_num();
    PLPA_CPU_SET(cpu,&msk);
    PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &msk);
  }
```

Likwid-pin
Overview

- Inspired by and based on `ptoverride` (Michael Meier, RRZE) and `taskset`
- Pins processes and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify skip mask (shepherd threads should not be pinned)
- Based on combination of wrapper tool together with overloaded pthread library
- Can also be used as a superior replacement for taskset
- Supports logical core numbering within a node and within an existing CPU set
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system
- Configurable colored output

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

Likwid-pin

Example: Intel OpenMP

- **Running the STREAM benchmark with likwid-pin:**

```
$ export OMP_NUM_THREADS=4
$ likwid-pin -s 0x1 -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
[... some STREAM output omitted ...]
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
[pthread wrapper] PIN_MASK: 0->1  1->4  2->5
[pthread wrapper] SKIP MASK: 0x1
[pthread wrapper 0] Notice: Using libpthread.so.0
        threadid 1073809728 -> SKIP
[pthread wrapper 1] Notice: Using libpthread.so.0
        threadid 1078008128 -> core 1 - OK
[pthread wrapper 2] Notice: Using libpthread.so.0
        threadid 1082206528 -> core 4 - OK
[pthread wrapper 3] Notice: Using libpthread.so.0
        threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]
```

Main PID always pinned

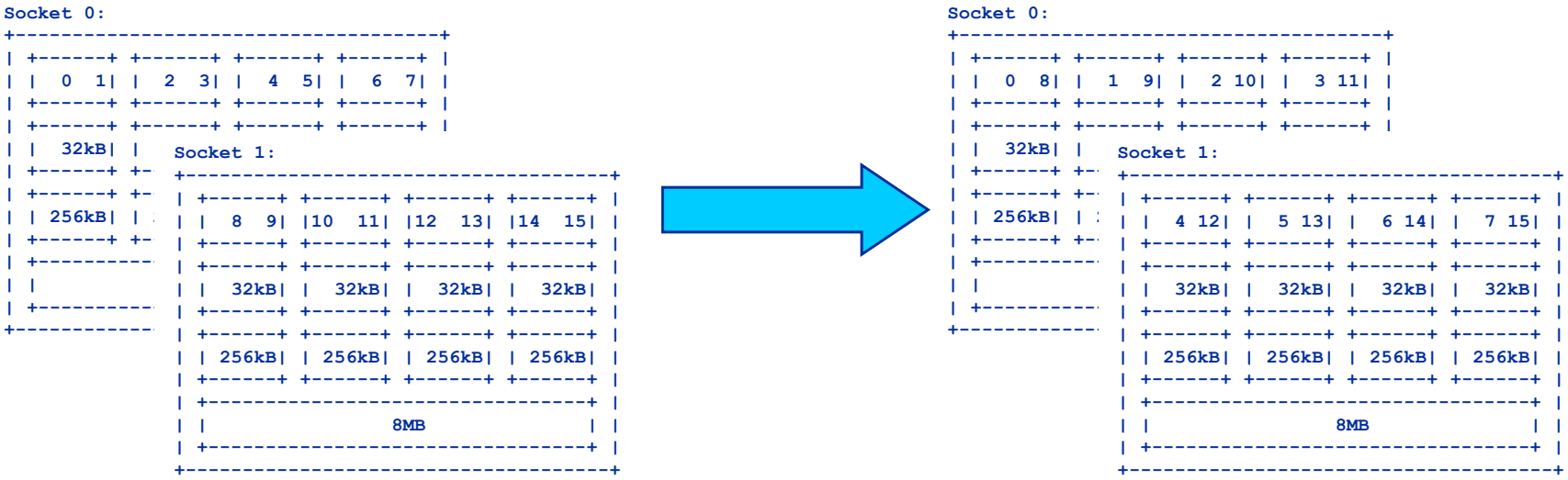
Skip shepherd thread

Pin all spawned threads in turn

Likwid-pin

Using logical core numbering

- **Core numbering may vary from system to system even with identical hardware**
  - Likwid-topology delivers this information, which can then be fed into likwid-pin
- **Alternatively, likwid-pin can abstract this variation and provide a purely logical numbering (physical cores first)**



- Across all cores in the node:
  `likwid-pin -c N:0-7 ./a.out`
- Across the cores in each socket and across sockets in each node:
  `likwid-pin -c S0:0-3@S1:0-3 ./a.out`

Likwid-pin

Using logical core numbering

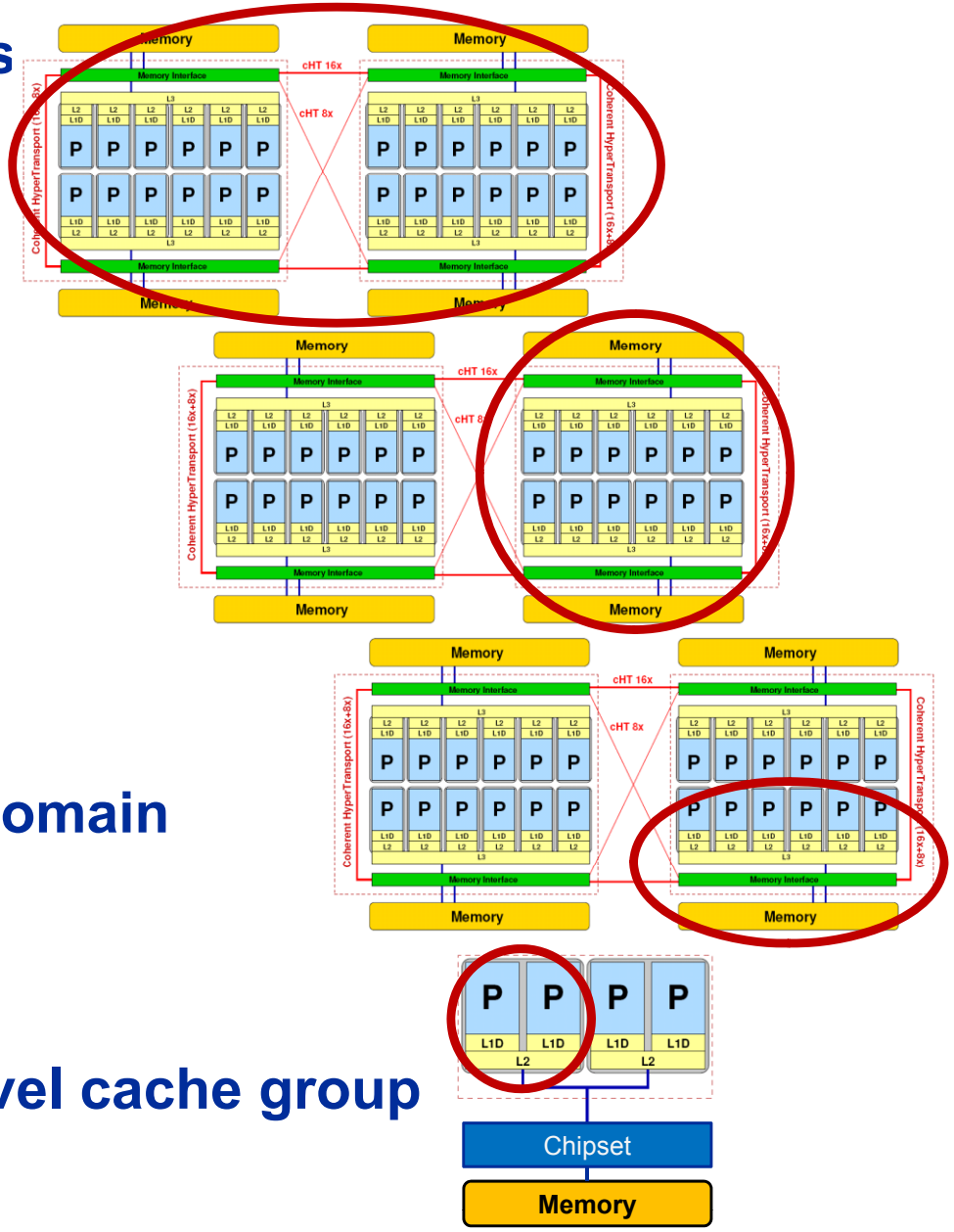
- **Possible unit prefixes**

N **node**

S **socket**

M **NUMA domain**

C **outer level cache group**



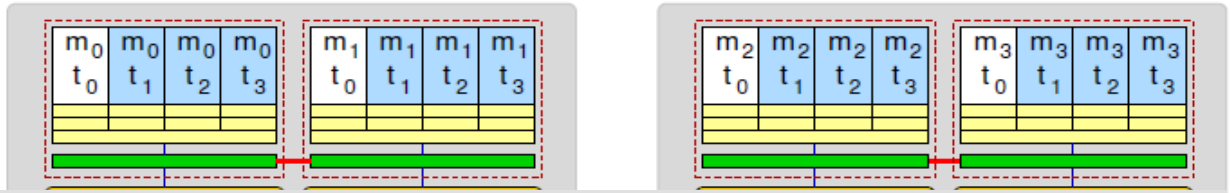
More examples: Hybrid MPI+OpenMP

Using Intel MPI+compiler & home-grown mpirun wrapper

One MPI process per node (w/ explicit logical numbering)

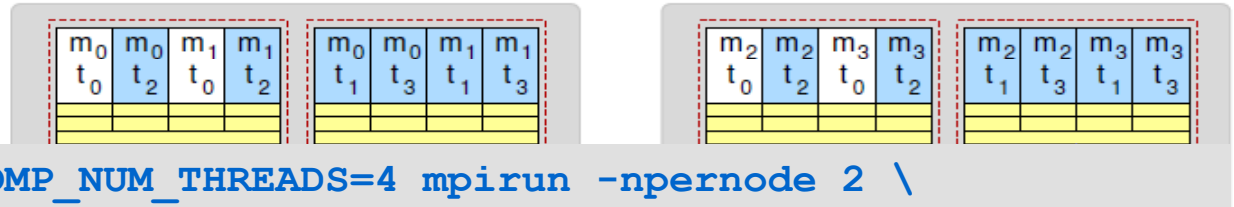

```
env OMP_NUM_THREADS=8 mpirun -pernode \
          likwid-pin -t intel -c N:0-7 ./a.out
```

One MPI process per socket (no pinning inside socket required)



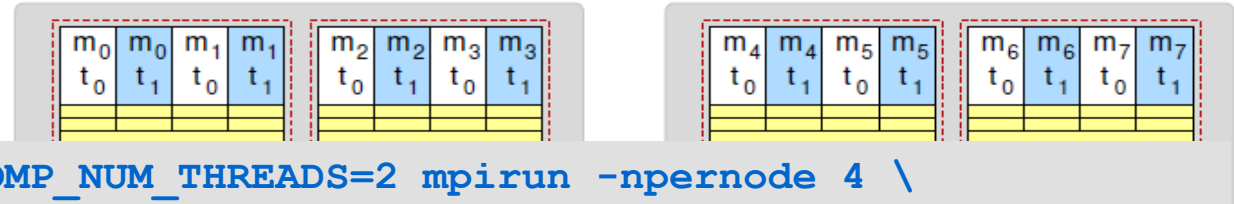

```
env OMP_NUM_THREADS=4 mpirun -npernode 2 \
        -pin "0,1,2,3_4,5,6,7" ./a.out
```

OpenMP threads pinned "round robin" across cores (logical core numbers due to cpu set established by mpirun)



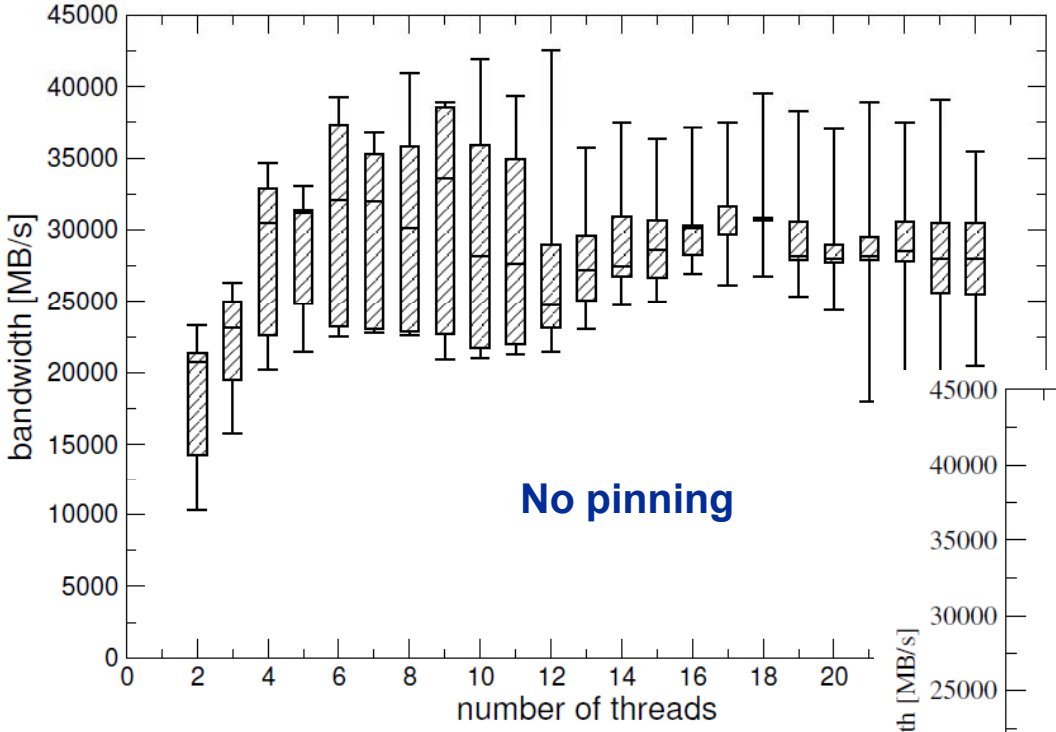

```
env OMP_NUM_THREADS=4 mpirun -npernode 2 \
        -pin "0,1,4,5_2,3,6,7" \
        likwid-pin -t intel -c 0,2,1,3 ./a.out
```

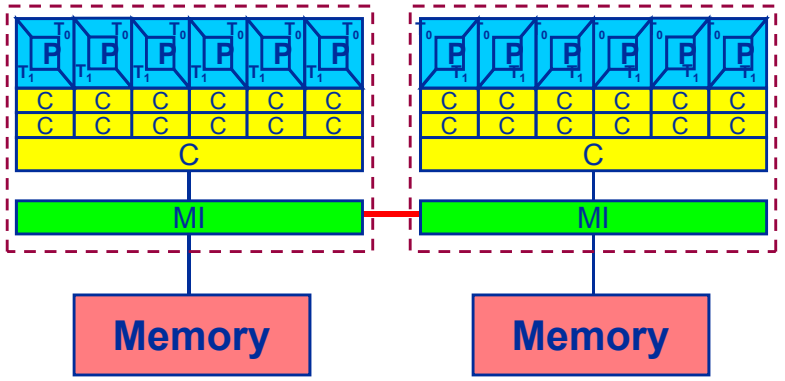
Two MPI processes per socket (dito)

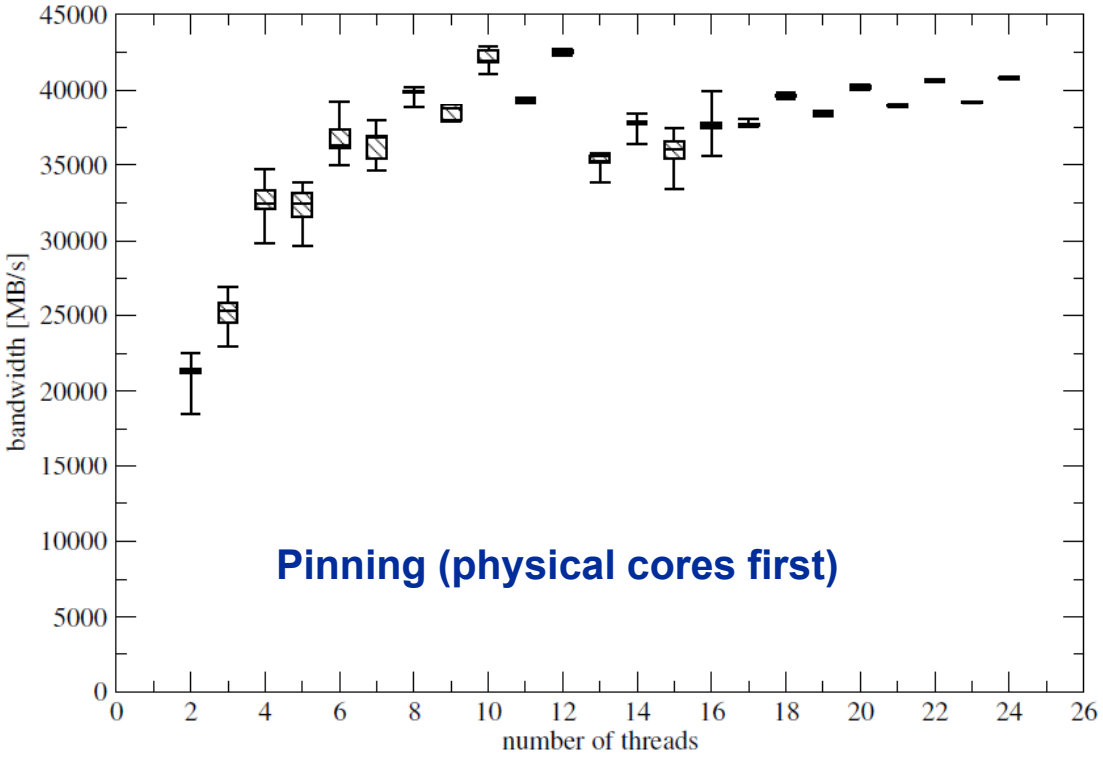



```
env OMP_NUM_THREADS=2 mpirun -npernode 4 \
        -pin "0,1_2,3_4,5_6,7" \
        likwid-pin -t intel -c 0,1 ./a.out
```

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







There are several reasons for caring about affinity:

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

Monitoring the Binding

- How can we see whether the measures for binding are really effective?
  - `sched_getaffinity(), ...`
  - `top:`

```
top - 16:05:03 up 24 days,  7:24, 32 users,  load average: 5.47, 4.92, 3.52
Tasks: 419 total,   4 running, 415 sleeping,   0 stopped,   0 zombie
Cpu(s):  95.7% us,  1.1% sy,  1.6% ni, 0.0% id,  1.4% wa,  0.0% hi,  0.2% si
Mem:   8157028k total,  8131252k used,    25776k free,     2772k buffers
Swap:  8393848k total,    93168k used,  8300680k free,  7160040k cached

  PID USER      PR  VIRT  RES  SHR  NI P S %CPU %MEM    TIME COMMAND
23914 unrz55    25  277m 223m 2660   0 2 R 99.9  2.8  23:42 dmrg_0.26_WOODY
24284 unrz55    16  8580 1556  928   0 2 R  0.2  0.0   0:00 top
 4789 unrz55    15 40220 1452 1448   0 0 S  0.0  0.0   0:00 sshd
 4790 unrz55    15  7900  552  548   0 3 S  0.0  0.0   0:00 tcsh
```

physical CPU ID

  - Press “H” for showing separate threads

Probing performance behavior

- **How do we find out about the performance requirements of a parallel code?**
  - Profiling via advanced tools is often overkill
- **A coarse overview is often sufficient**
  - likwid-perfctr (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
  - Simple end-to-end measurement of hardware performance metrics
  - “Marker” API for starting/stopping counters
  - Multiple measurement region support
  - Preconfigured and extensible metric groups, list with **`likwid-perfctr -a`**

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
FLOPS_X87: X87 MFlops/s
L2: L2 cache bandwidth in MBytes/s
L2CACHE: L2 cache miss rate/ratio
L3: L3 cache bandwidth in MBytes/s
L3CACHE: L3 cache miss rate/ratio
MEM: Main memory bandwidth in MBytes/s
TLB: TLB miss rate/ratio
```

likwid-perfctr

Example usage with preconfigured metric group

```
$ env OMP_NUM_THREADS=4 likwid-perfctr -c 0-3 -g FLOPS_DP likwid-pin -c 0-3 -s 0x1 ./stream.exe
-------------------------------------------------------------
CPU type:       Intel Core Lynnfield processor
CPU clock:      2.93 GHz
-------------------------------------------------------------
Measuring group FLOPS_DP
-------------------------------------------------------------
YOUR PROGRAM OUTPUT
+-----------------------------------+------------+------------+------------+------------+
|                Event              |   core 0   |   core 1   |   core 2   |   core 3   |
+-----------------------------------+------------+------------+------------+------------+
|         INSTR_RETIRED_ANY         | 1.97463e+08 | 2.31001e+08 | 2.30963e+08 | 2.31885e+08 |
|       CPU_CLK_UNHALTED_CORE       | 9.56999e+08 | 9.58401e+08 | 9.58637e+08 | 9.57338e+08 |
|  FP_COMP_OPS_EXE_SSE_FP_PACKED    | 4.00294e+07 | 3.08927e+07 | 3.08866e+07 | 3.08904e+07 |
|  FP_COMP_OPS_EXE_SSE_FP_SCALAR    |     882     |      0      |      0      |      0      |
| FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION |     0     |      0      |      0      |      0      |
| FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION | 4.00303e+07 | 3.08927e+07 | 3.08866e+07 | 3.08904e+07 |
+-----------------------------------+------------+------------+------------+------------+

+-------------------------+------------+---------+----------+----------+
|          Metric         |   core 0   | core 1  |  core 2  |  core 3  |
+-------------------------+------------+---------+----------+----------+
|       Runtime [s]       |  0.326242  | 0.32672 | 0.326801 | 0.326358 |
|           CPI           |  4.84647   | 4.14891 | 4.15061  | 4.12849  |
| DP MFlops/s (DP assumed) |  245.399   | 189.108 | 189.024  | 189.304  |
|      Packed MUOPS/s     |  122.698   | 94.554  | 94.5121  | 94.6519  |
|      Scalar MUOPS/s     | 0.00270351 |    0    |    0     |    0     |
|        SP MUOPS/s       |     0      |    0    |    0     |    0     |
|        DP MUOPS/s       |  122.701   | 94.554  | 94.5121  | 94.6519  |
+-------------------------+------------+---------+----------+----------+
```

Always measured

Configured metrics (this group)

Derived metrics

Things to look at

- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Shared cache BW saturation
- Flop/s, loads and stores per flop metrics
- CPI metric
- # of instructions, branches, mispredicted branches

Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
- In-socket performance saturation may have various reasons
- Cache miss metrics are overrated
  - If I really know my code, I can *calculate* the misses
  - Runtime and resource utilization is much more important

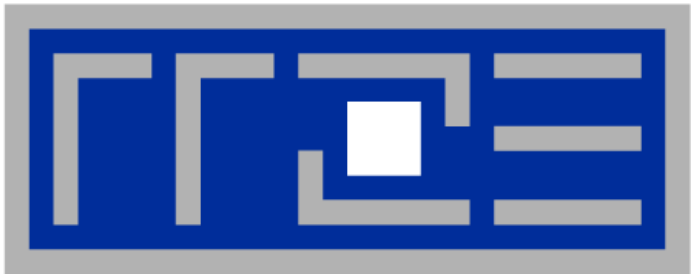
Section summary: What to take home

- **Figuring out the node topology is usually the hardest part**
  - Virtual/physical cores, cache groups, cache parameters
  - This information is usually scattered across many sources
- **LIKWID-topology**
  - One tool for all topology parameters
  - Supports Intel and AMD processors under Linux (currently)

- **Generic affinity tools**
  - Taskset, numactl do not pin individual threads
  - Manual (explicit) pinning from within code
- **LIKWID-pin**
  - Binds threads/processes to cores
  - Optional abstraction of strange numbering schemes (logical numbering)
- **LIKWID-perfctr**
  - End-to-end hardware performance metric measurement
  - Finds out about basic architectural requirements of a program

Tutorial outline

- Introduction
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- Multicore performance tools
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- New chances with multicore hardware
  - Wavefront parallelization of stencil codes
  - Explicit comm/calc overlap in sparse MVM
- Summary
- Appendix



General remarks on the performance properties of multicore multisocket systems

The parallel vector triad benchmark

A "swiss army knife" for microbenchmarking

- **Simple streaming benchmark:**

```
for(int j=0; j < NITER; j++){
#pragma omp parallel for
  for(i=0; i < N; ++i)
    a[i]=b[i]+c[i]*d[i];
    if(OBSCURE)
      dummy(a,b,c,d);
}
```

- **Report performance for different N**
- **Choose NITER so that accurate time measurement is possible**

The parallel vector triad benchmark

Optimal code on x86 machines

```
timing(&wct_start, &cput_start);
#pragma omp parallel private(j)
{
  for(j=0; j<niter; j++){
    if(size > CACHE_SIZE>>5) {
#pragma omp parallel for
#pragma vector always
#pragma vector aligned
#pragma vector nontemporal
      for(i=0; i<size; ++i)
        a[i]=b[i]+c[i]*d[i];
    } else {
#pragma omp parallel for
#pragma vector always
#pragma vector aligned
      for(i=0; i<size; ++i)
        a[i]=b[i]+c[i]*d[i];
    }
    if(a[5]<0.0)
      cout << a[3] << b[5] << c[10] << d[6];
  }
}
timing(&wct_end, &cput_end);
```

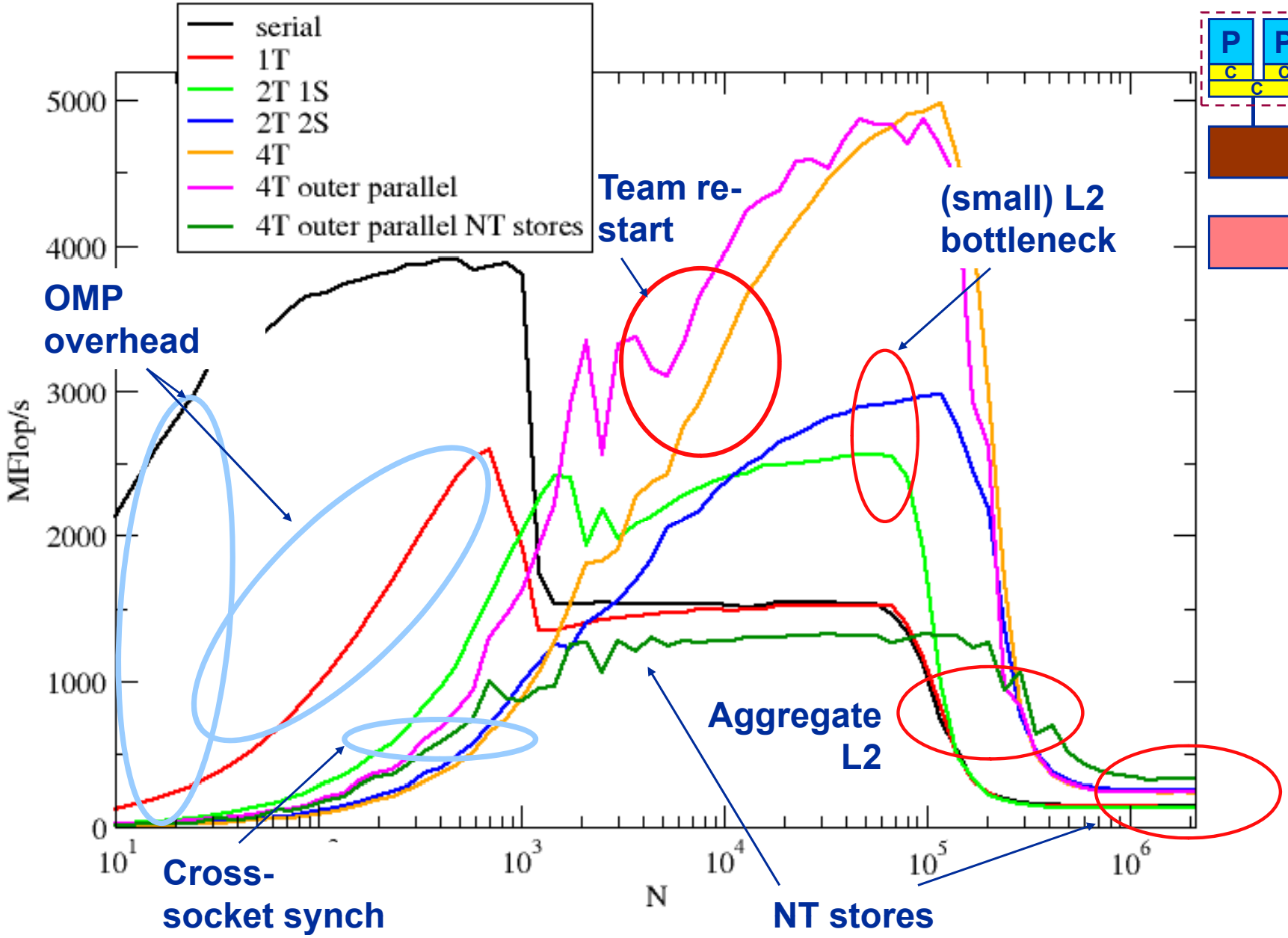
```
// size = multiple of 8
int vector_size(int n){
  return int(pow(1.3,n))&(-8);
}
```

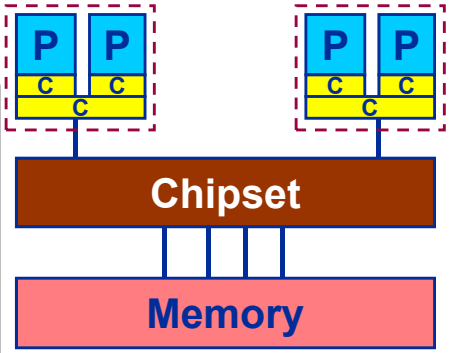
Large-N version (NT)

Small-N version (noNT)

The parallel vector triad benchmark

Performance results on Xeon 5160 node

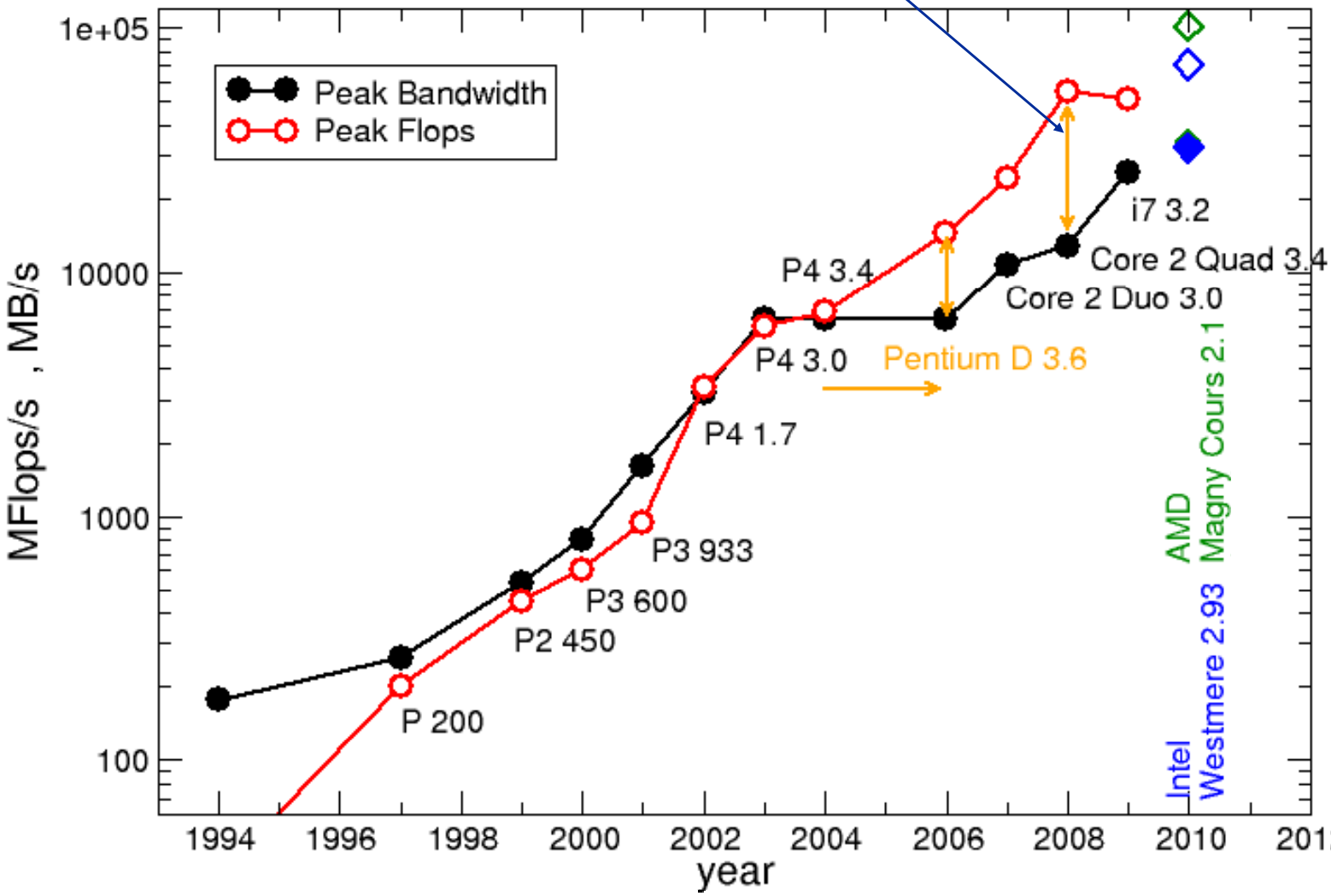




Bandwidth limitations: Memory

Some problems get even worse….

- System balance = PeakBandwidth [MByte/s] / PeakFlops [MFlop/s]
  Typical balance ~ 0.25 Byte / Flop → 4 Flop/Byte → 32 Flop/double

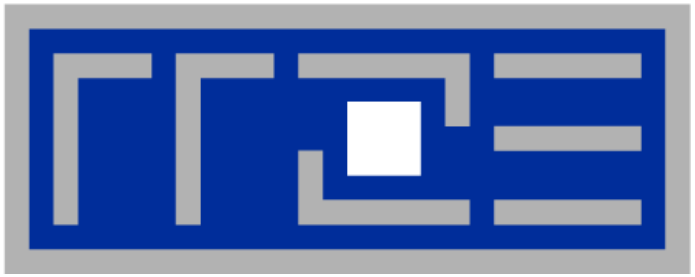


Balance values:

Scalar product:
1 Flop/double
→ 1/32 Peak

Dense
Matrix·Vector:
2 Flop/double
→ 1/16 Peak

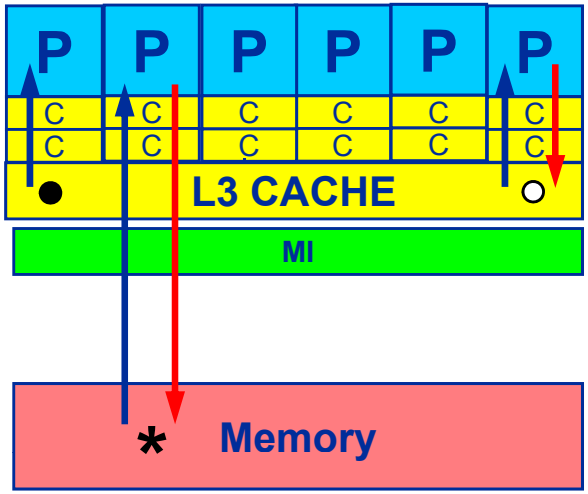
Large
MatrixMatrix
(BLAS3)

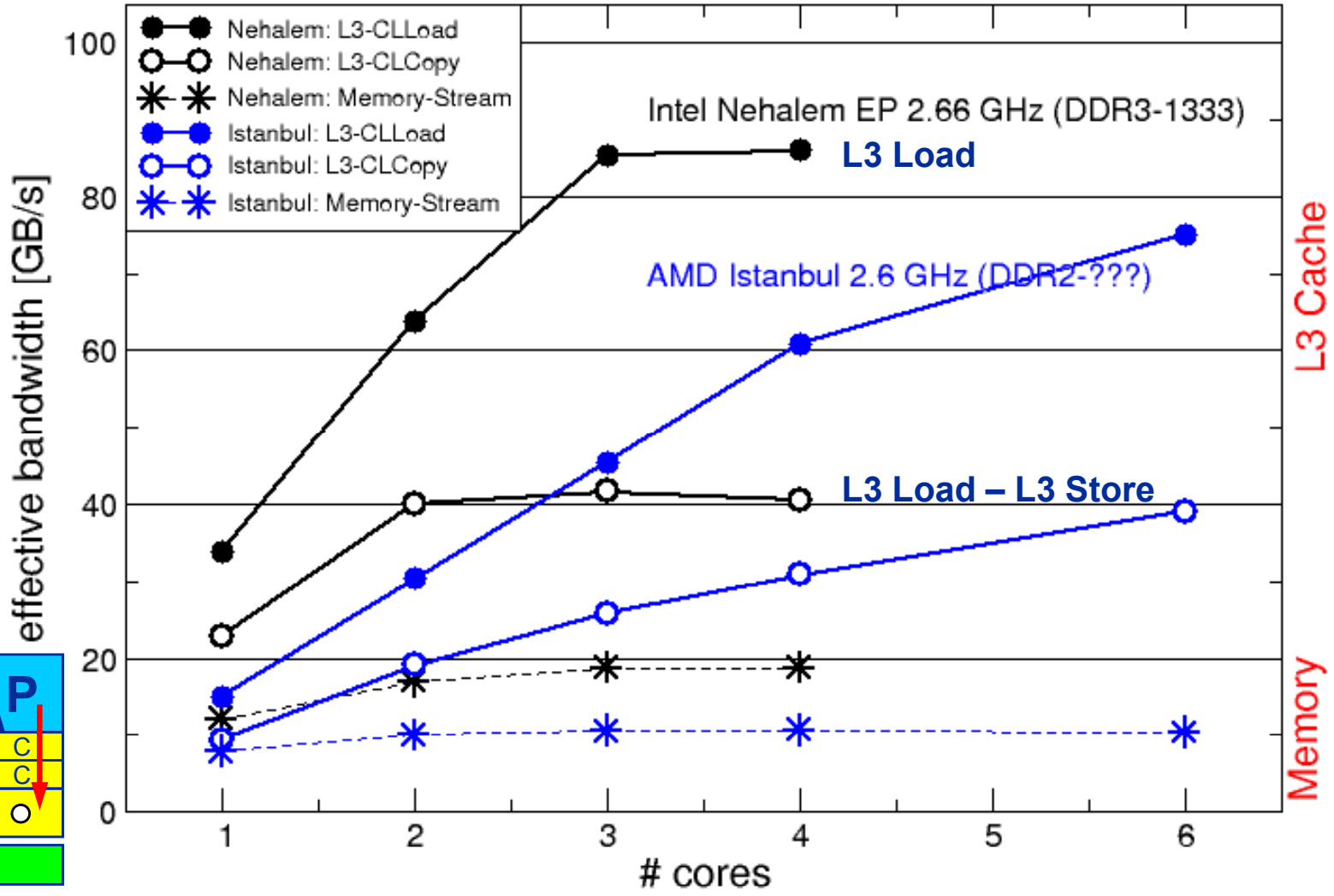


Bandwidth saturation effects in cache and memory

Bandwidth limitations: Memory and cache

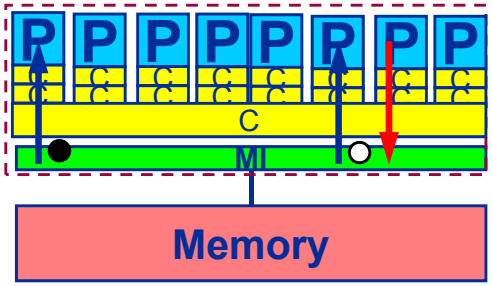
Scalability of shared data paths on a socket

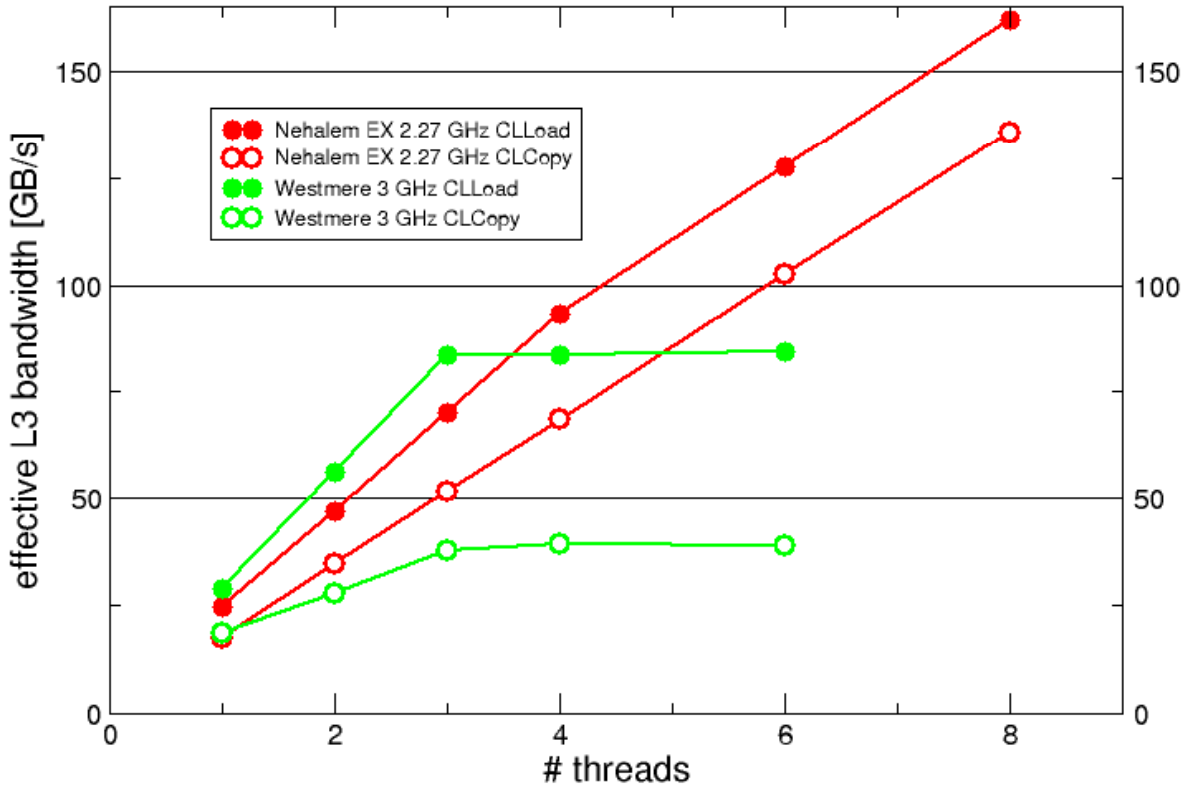




Bandwidth limitations: Outer-level cache

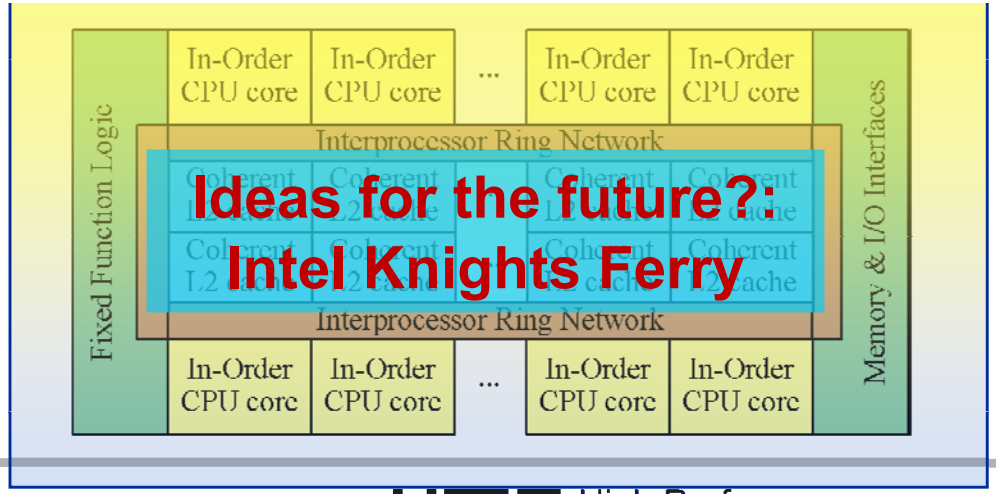
L3 bandwidth may scale a bit better in future systems…





- Intel Nehalem EX
  - **8-core chip**; 24 MB L3
  - 4 DDR3-channels per socke
  - 4 sockets EA system: 128 GB DDR3

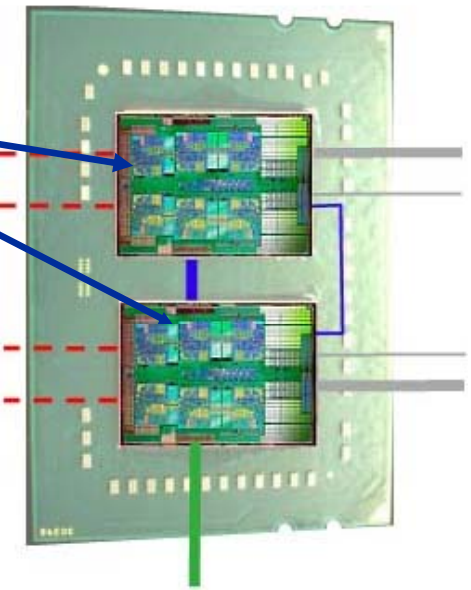
- Nehalem EX: New L3 design
  - 8 segments connected by ring
  - Scalable bandwidth
  - Lesson learned from “Larrabee”?
  - Has been retained in “Sandy Bridge”

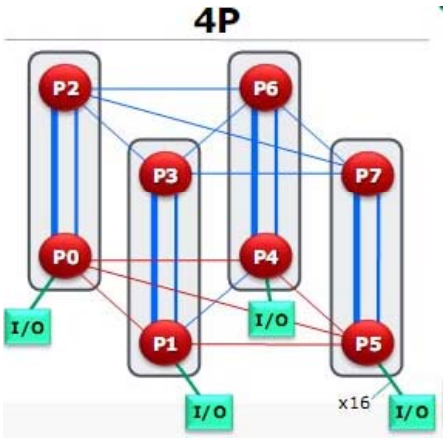


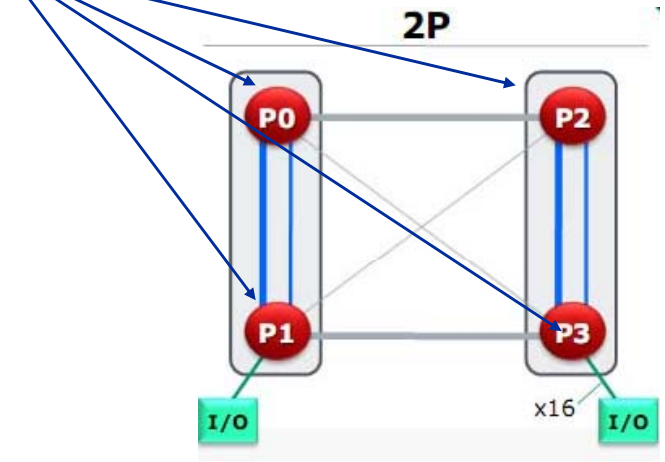
Ameliorating bandwidth limitations by on-socket ccNUMA

AMD Magny-Cours – a ccNUMA 12-core socket

- **AMD "Magny-Cours" available as 8-core or 12-core !**
  - 12-core socket implemented as two 6-core chips connected via 1.5 HT links
  - Main memory access: → 2 DDR3-Channels per 6-core chip
    → 1/3 DDR3-Channel per core
  - 2 socket server → 4 memory locality domains
    → ccNUMA within a socket!
  - 4 socket server:







  - Network balance (QDR+2P Magny Cours) ~ 240 GF/s / 3 GB/s = 80 F/B
    (2003: Intel Xeon DP 2.66 GHz + GBit ~ 10 GF/s / 0.12 GB/s = 80 B/F)

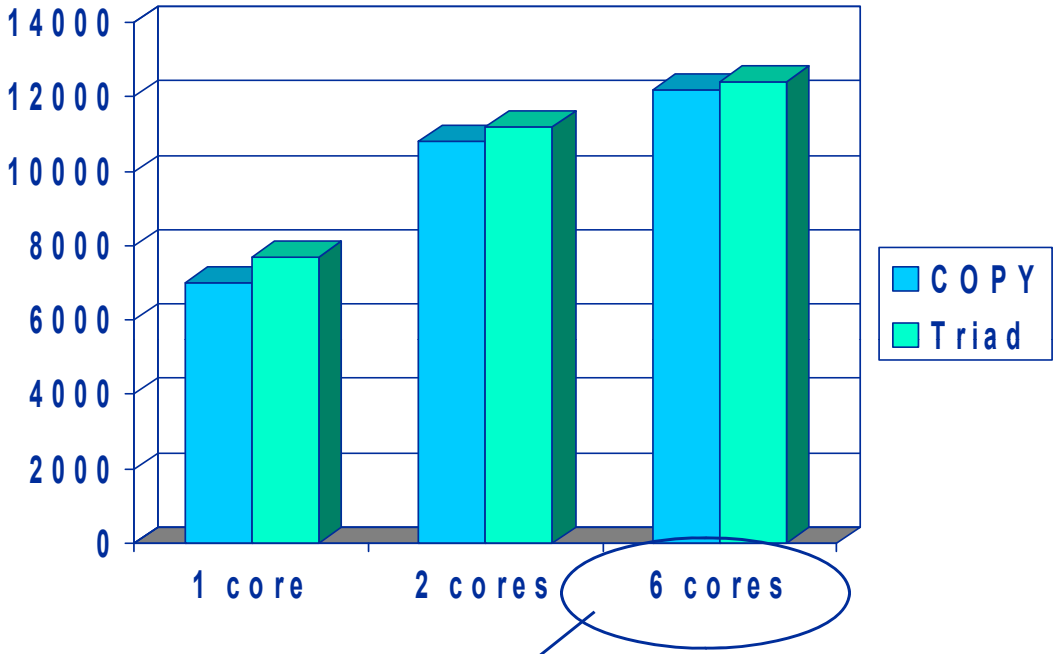
Ameliorating bandwidth limitations by on-socket ccNUMA

AMD Magny-Cours – a ccNUMA 12-core socket

- **Test system configuration:**
  - 2 x AMD Opteron 6172 (2x6 cores; 2x6MB L3; 2.1 GHz)
  - 64 GB DDR3-1333 MHz
- **Stream (triad w/ NT stores):**

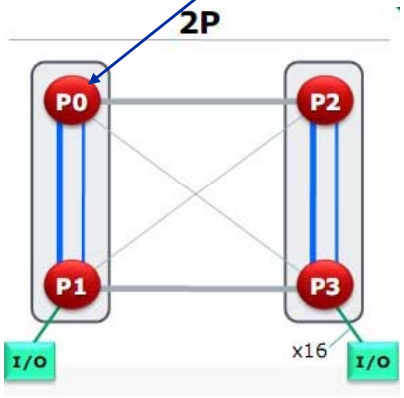
1 socket (12 cores): 24.8 GB/s

2 sockets: 49.7 GB/s



- **Local vs. remote data access**

Local / remote	Single thread (triad)
P0 → LD0	7,8 GB/s
P0 → LD1	5,1 GB/s
P0 → LD2	5,1 GB/s
P0 → LD3	3,0 GB/s



Case study: Sparse matrix-vector multiply

- **Important kernel in many applications (matrix diagonalization, solving linear systems)**
- **Strongly memory-bound for large data sets**
  - Streaming, with partially indirect access:

```
!$OMP parallel do
do i = 1,Nr
 do j = row_ptr(i), row_ptr(i+1) - 1
  c(i) = c(i) + val(j) * b(col_idx(j))
 enddo
enddo
!$OMP end parallel do
```

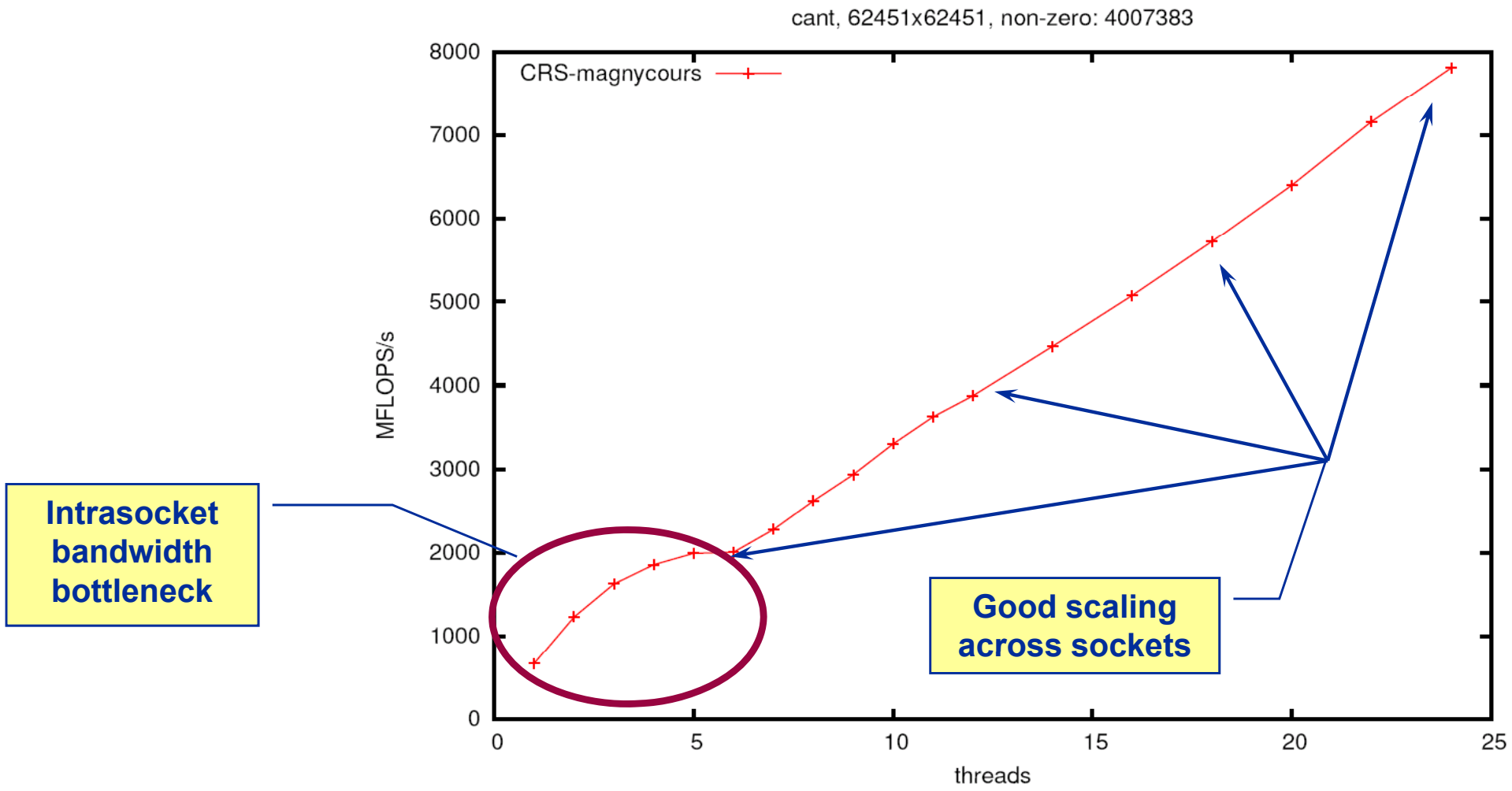
  - Usually many spMVMs required to solve a problem

- **Case study: Performance data on one 24-core AMD Magny Cours node**

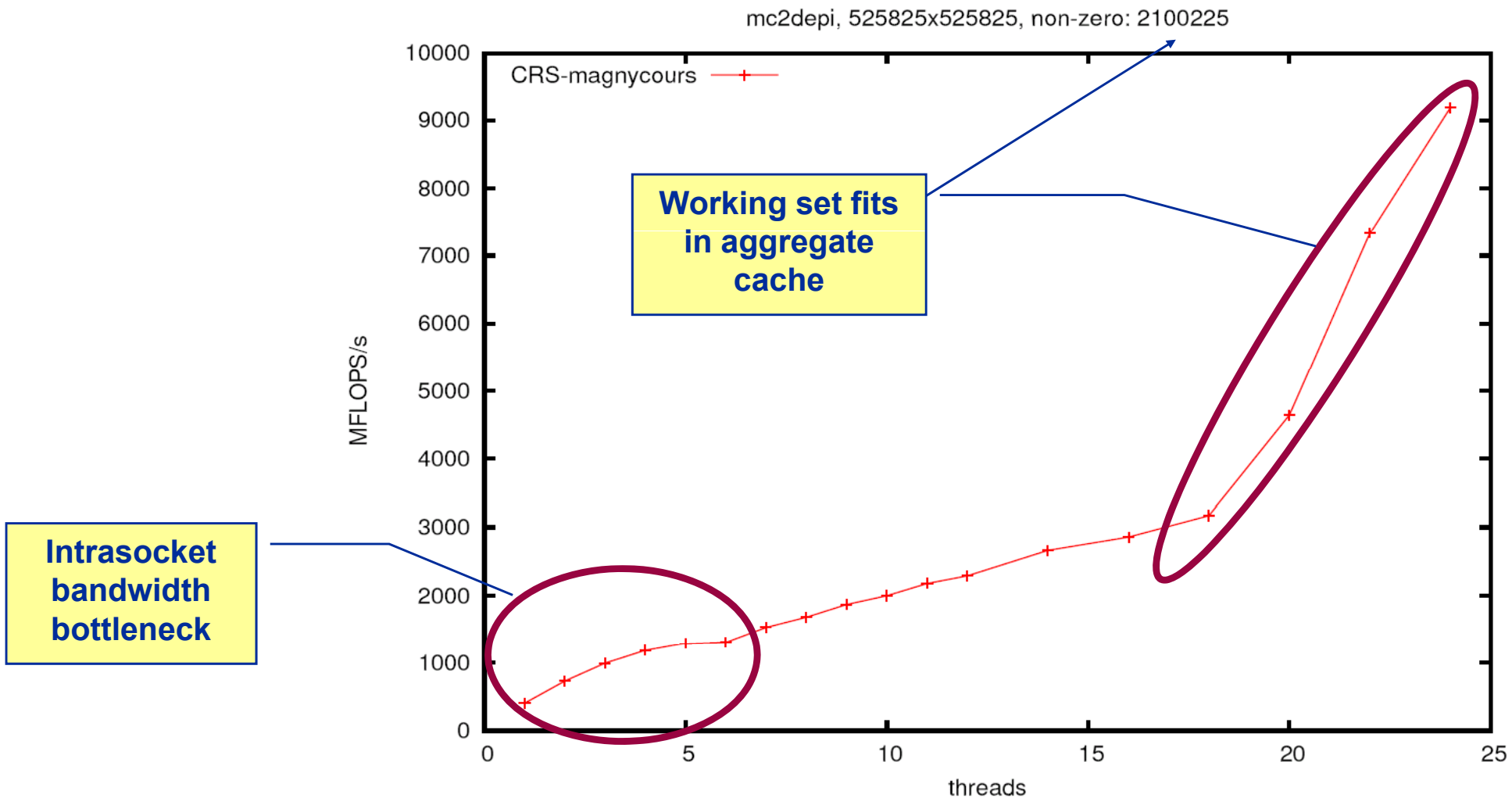
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

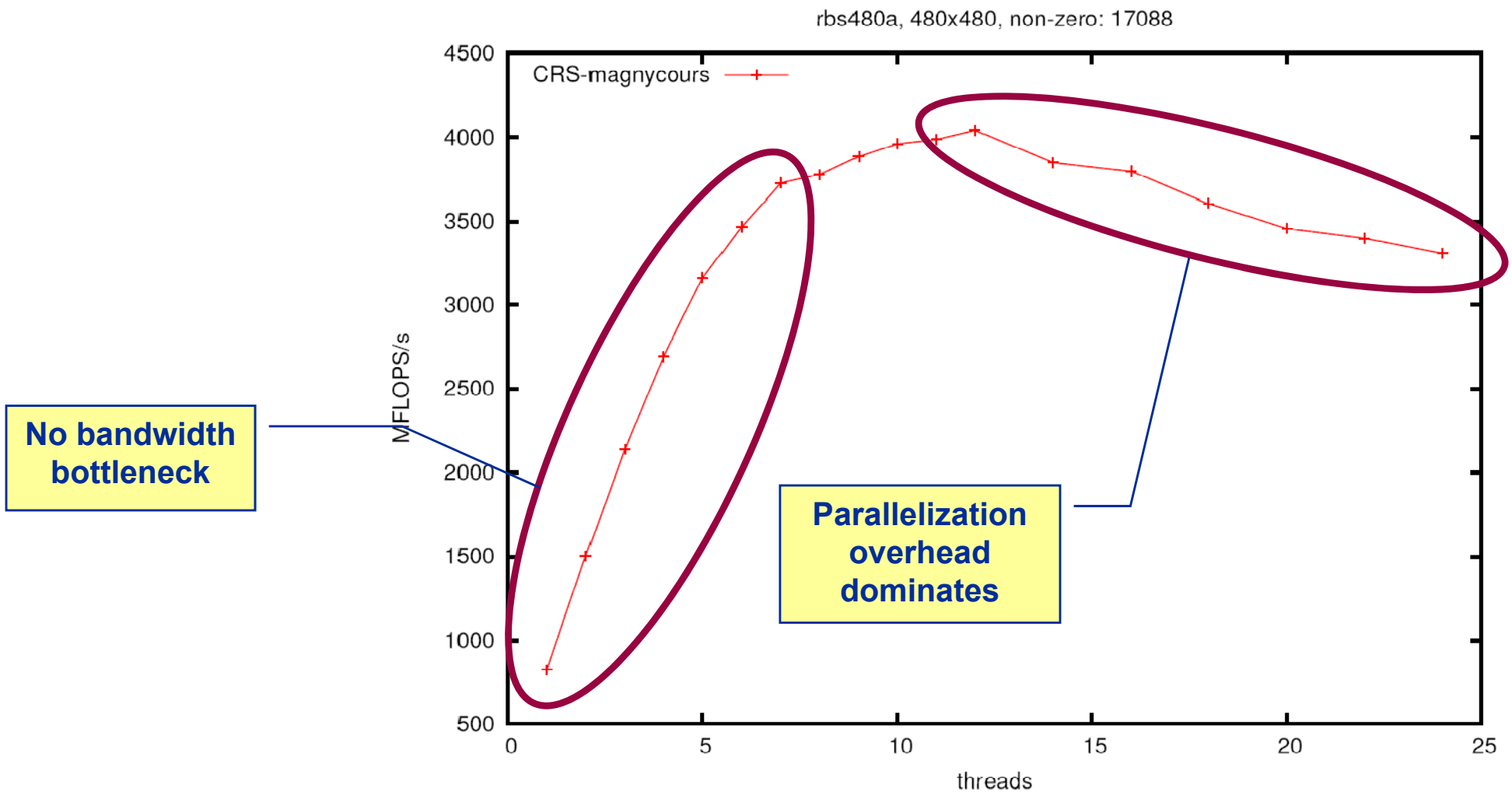
- **Case 1: Large matrix**

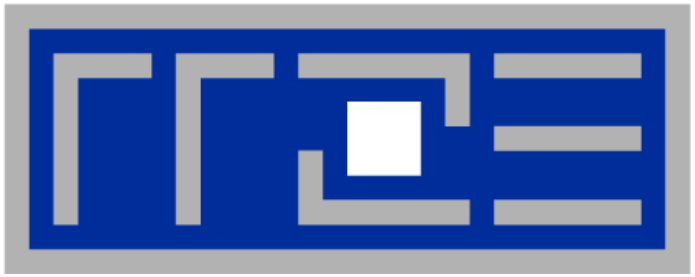


- **Case 2: Medium size**



- **Case 3: Small size**





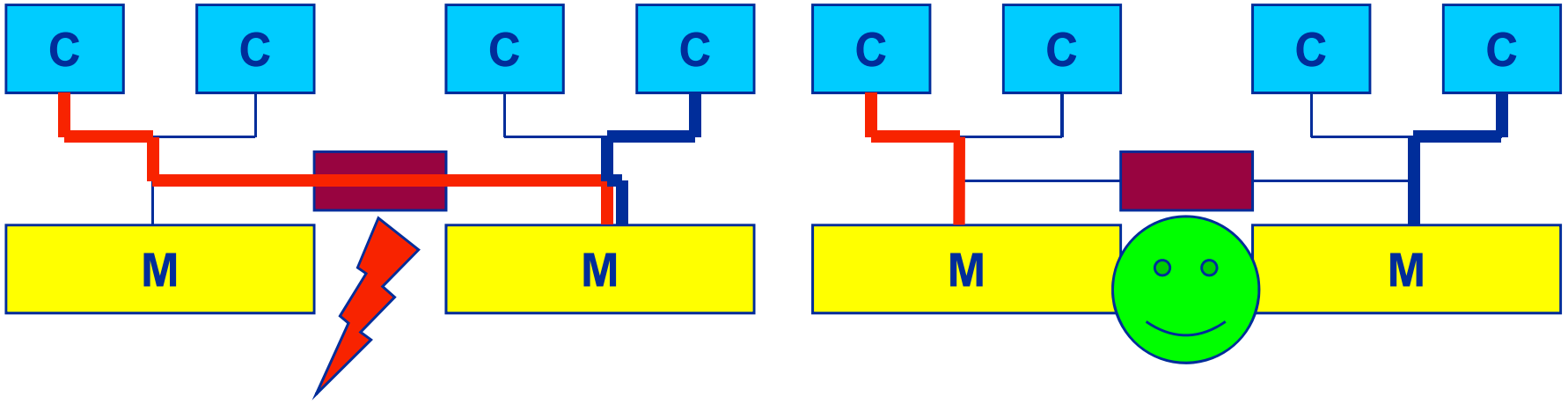
Efficient parallel programming on ccNUMA nodes

- Performance characteristics of ccNUMA nodes
- First touch placement policy
- C++ issues
- ccNUMA locality and dynamic scheduling
- ccNUMA locality beyond first touch

ccNUMA performance problems

"The other affinity" to care about

- **ccNUMA:**
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
- **How do we make sure that memory access is always as "local" and "distributed" as possible?**

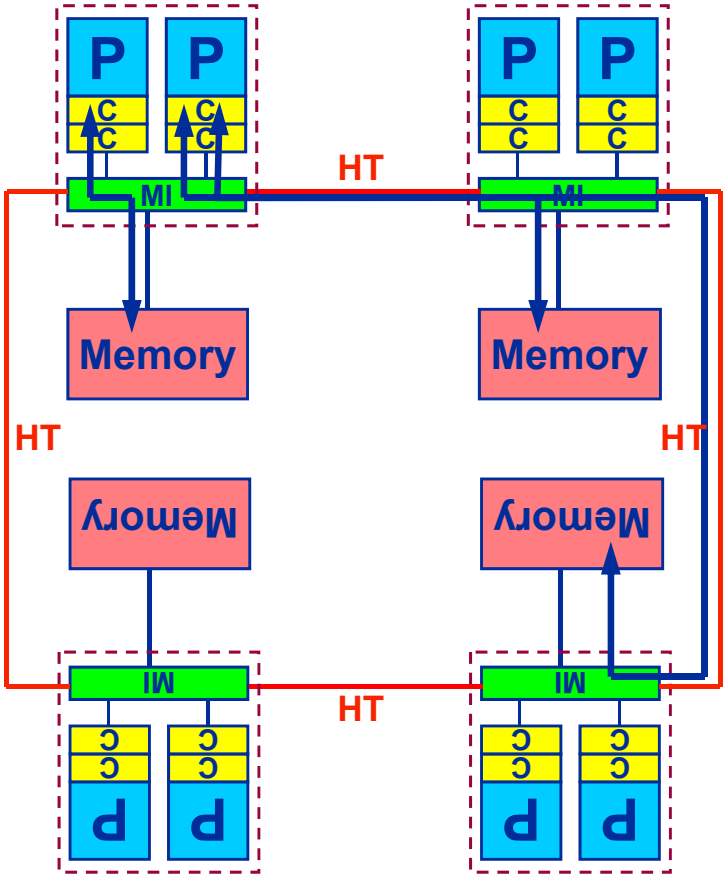


  - Page placement is implemented in units of OS pages (often 4kB, possibly more)

Example: HP DL585 G5

4-socket ccNUMA Opteron 8220 Server

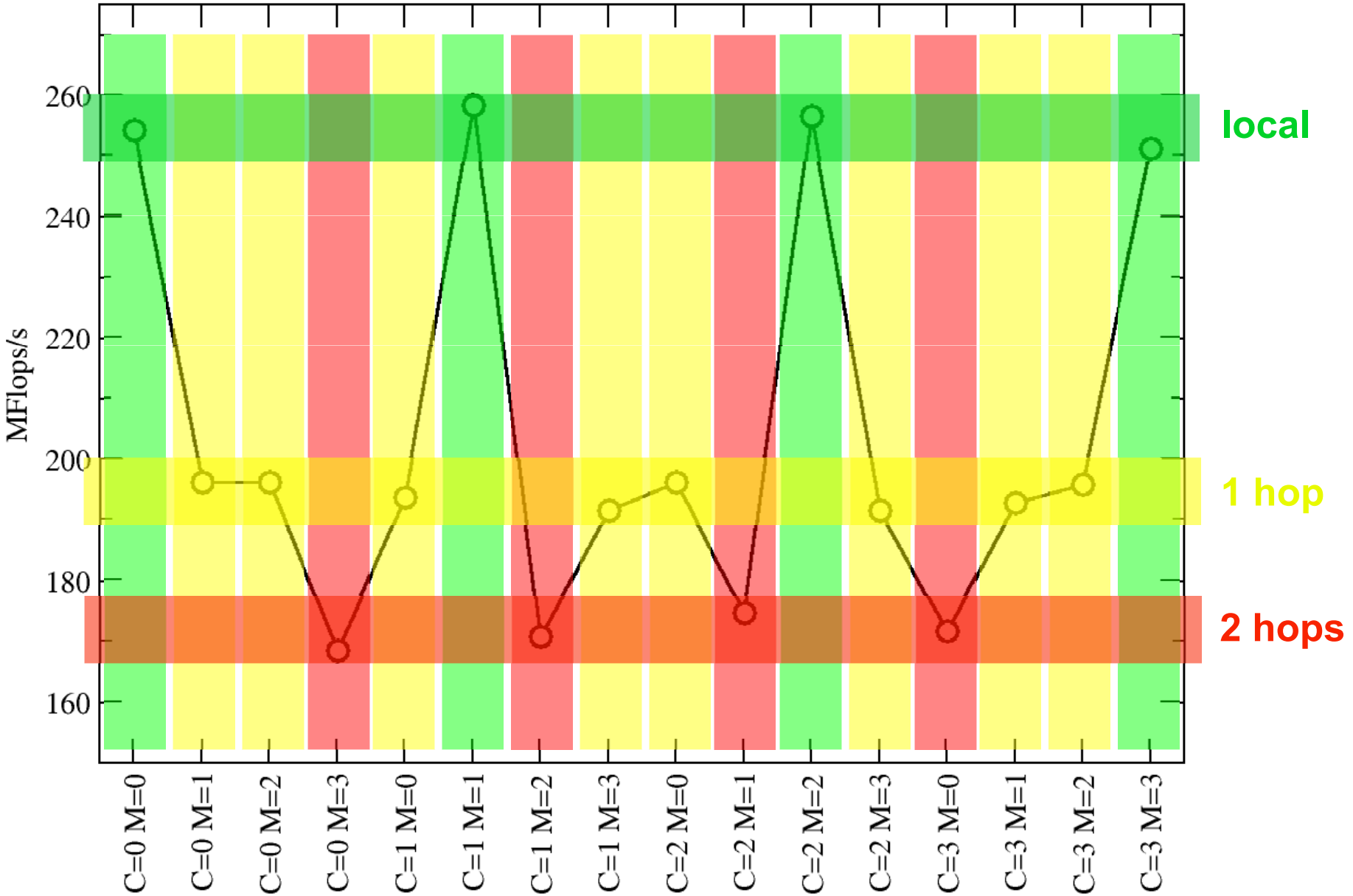
- **CPU**
  - 64 kB L1 per core
  - 1 MB L2 per core
  - No shared caches
  - On-chip memory controller (MI)
  - 10.6 GB/s local memory bandwidth
- **HyperTransport 1000 network**
  - 4 GB/s per link per direction
- **3 distance categories for core-to-memory connections:**
  - same LD
  - 1 hop
  - 2 hops



- **Q1: What are the real penalties for non-local accesses?**
- **Q2: What is the impact of contention?**

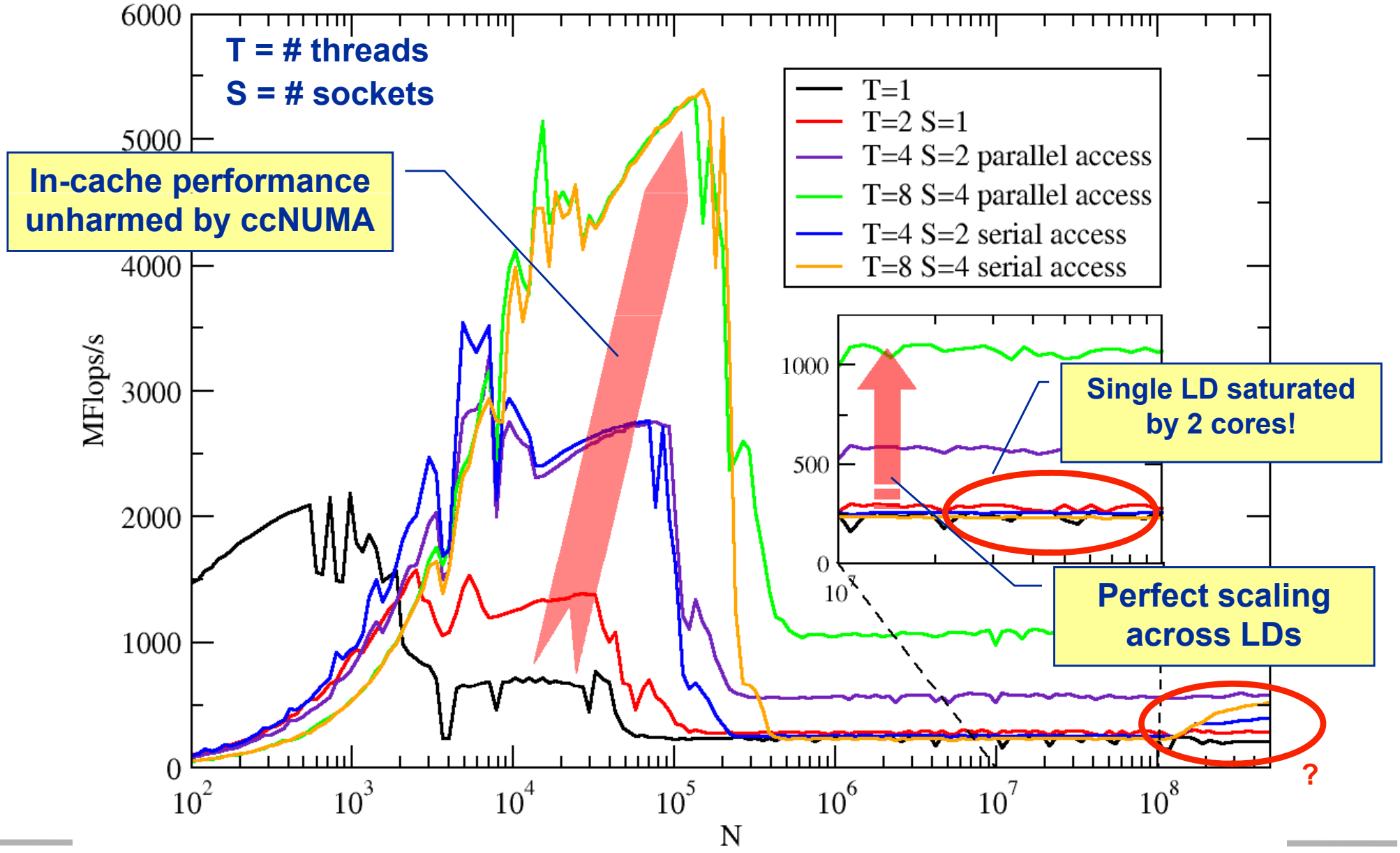
Effect of non-local access on HP DL585 G5:

Serial vector triad `A(:)=B(:)+C(:)*D(:)`



Contention vs. parallel access on HP DL585 G5:

OpenMP vector triad `A(:)=B(:)+C(:)*D(:)`



ccNUMA locality tool numactl:

How do we enforce some locality of access?

- **`numactl` can influence the way a binary maps its memory pages:**

```
numactl --membind=<nodes> a.out     # map pages only on <nodes>
        --preferred=<node>  a.out   # map pages on <node>
                                    # and others if <node> is full
        --interleave=<nodes> a.out  # map pages round robin across
                                    # all <nodes>
```

- **Examples:**

```
env OMP_NUM_THREADS=2 numactl --membind=0 -cpunodebind=1 ./stream

env OMP_NUM_THREADS=4 numactl --interleave=0-3 \
                              likwid-pin -c N:0,4,8,12 ./stream
```

- **But what is the default without `numactl`?**

ccNUMA default memory locality

- **"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
  - This might be a problem, see later
- **Caveat: "touch" means "write", not "allocate"**
- **Example:**

```
double *huge = (double*)malloc(N*sizeof(double));

for(i=0; i<N; i++) // or i+=PAGE_SIZE
   huge[i] = 0.0;
```

Memory not mapped here yet

Mapping takes place here

- **It is sufficient to touch a single item to map the entire page**

Coding for Data Locality

- **The programmer must ensure that memory pages get mapped locally in the first place (and then prevent migration)**
  - Rigorously apply the "Golden Rule"
    - I.e. we have to take a closer look at initialization code
  - Some non-locality at domain boundaries may be unavoidable
  - Stack data may be another matter altogether:

```
void f(int s) {              // called many times with different s
  double a[s];               // c99 feature
  // where are the physical pages of a[] now???
  …
}
```

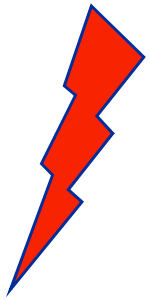
  - Fine-tuning is possible (see later)

- **Prerequisite: Keep threads/processes where they are**
  - Affinity enforcement (pinning) is key (see earlier section)

Coding for ccNUMA data locality

- Simplest case: explicit initialization

```
integer,parameter :: N=1000000
real*8 A(N), B(N)



A=0.d0


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

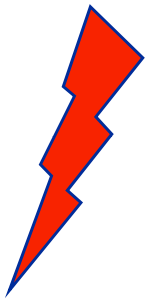


```
integer,parameter :: N=1000000
real*8 A(N),B(N)

!$OMP parallel do schedule(static)
do i = 1, N
  A(i)=0.d0
end do

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

Coding for Data Locality

- **Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so "localize" arrays before I/O**

```
integer,parameter :: N=1000000
real*8 A(N), B(N)

READ(1000) A
!$OMP parallel do
do I = 1, N
   B(i) = function ( A(i) )
end do
```



```
integer,parameter :: N=1000000
real*8 A(N),B(N)

!$OMP parallel do schedule(static)
do I = 1, N
A(i)=0.d0
end do
READ(1000) A
!$OMP parallel do schedule(static)
do I = 1, N
B(i) = function ( A(i) )
end do
```

Coding for Data Locality

- **Required condition: OpenMP loop schedule of initialization must be the same as in all computational loops**
  - Best choice: `static`! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
    - Guaranteed by OpenMP 3.0 only for loops in the same enclosing parallel region
    - In practice, it works with any compiler even across regions
  - If dynamic scheduling/tasking is unavoidable, more advanced methods may be in order

- **How about global objects?**
  - Better not use them
  - If communication vs. computation is favorable, might consider properly placed copies of global data
  - In C++, STL allocators provide an elegant solution

Coding for Data Locality:
Placement of static arrays or arrays of objects

- **Speaking of C++: Don't forget that constructors tend to touch the data members of an object. Example:**

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  inline D operator+(const D& o) throw() {
    return D(d+o.d);
  }
  inline D operator*(const D& o) throw() {
    return D(d*o.d);
  }
...
};
```

→ **placement problem with**

```
D* array = new D[1000000];
```

Coding for Data Locality:
Parallel first touch for arrays of objects

- **Solution: Provide overloaded `new` operator or special function that places the memory before constructors are called (PAGE_BITS = base-2 log of pagesize)**

```
template <class T> T* pnew(size_t n) {
  size_t st = sizeof(T);
  int ofs,len=n*st;
  int i,pages = len >> PAGE_BITS;
  char *p = new char[len];
#pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
      ofs = static_cast<size_t>(i) << PAGE_BITS;
      p[ofs]=0;
    }
#pragma omp parallel for schedule(static) private(ofs)
    for(ofs=0; ofs<n; ++ofs) {
      new(static_cast<void*>(p+ofs*st)) T;
    }
  return static_cast<T*>(m);
}
```

parallel first touch

placement new!

Coding for Data Locality:
NUMA allocator for parallel first touch in `std::vector<>`


```
template <class T> class NUMA_Allocator {
public:
  T* allocate(size_type numObjects, const void
                *localityHint=0) {
    size_type ofs,len = numObjects * sizeof(T);
    void *m = malloc(len);
    char *p = static_cast<char*>(m);
    int i,pages = len >> PAGE_BITS;
#pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
      ofs = static_cast<size_t>(i) << PAGE_BITS;
      p[ofs]=0;
    }
    return static_cast<pointer>(m);
  }
...
};
```

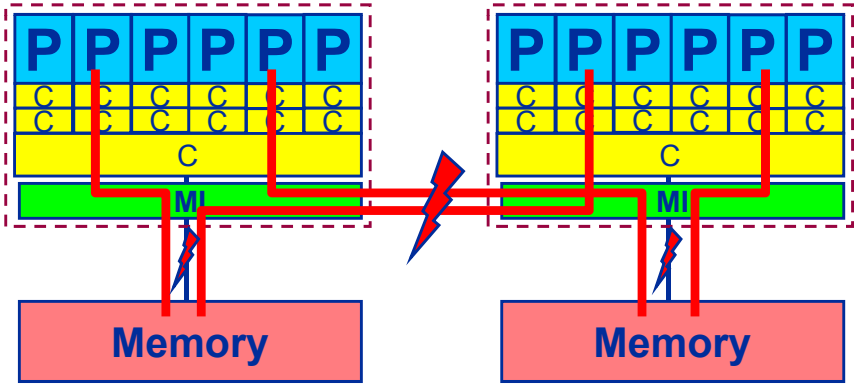
Application:

```
vector<double,NUMA_Allocator<double> > x(1000000)
```

Memory Locality Problems

- **Locality of reference is key to scalable performance on ccNUMA**
  - Less of a problem with distributed memory (MPI) programming, but see below
- **What factors can destroy locality?**

- **MPI programming:**
  - Processes lose their association with the CPU the mapping took place on originally
  - OS kernel tries to maintain strong affinity, but sometimes fails
- **Shared Memory Programming (OpenMP,…):**
  - Threads losing association with the CPU the mapping took place on originally
  - Improper initialization of distributed data
- **All cases:**
  - Other agents (e.g., OS kernel) may fill memory with data that prevents optimal placement of user data



Diagnosing Bad Locality

- **If your code is cache-bound, you might not notice any locality problems**
- **Otherwise, bad locality limits scalability at very low CPU numbers (whenever a node boundary is crossed)**
  - If the code makes good use of the memory interface
  - But there may also be a general problem in your code…
- **Consider using performance counters**
  - LIKWID-perfCtr can be used to measure nonlocal memory accesses
  - Example for Intel Nehalem (Core i7):

```
env OMP_NUM_THREADS=8 likwid-perfCtr -g MEM -c 0-7 \
                      likwid-pin -t intel -c 0-7 ./a.out
```

Using performance counters for diagnosing bad ccNUMA access locality

- **Intel Nehalem EP node:**

"Uncore" events only counted once per socket

Event	core 0	core 1	core 2	core 3	core 4	core 5
INSTR_RETIRED_ANY	5.20725e+08	5.24793e+08	5.21547e+08	5.23717e+08	5.28269e+08	5.29083e+08
CPU_CLK_UNHALTED_CORE	1.90447e+09	1.90599e+09	1.90619e+09	1.90673e+09	1.90583e+09	1.90746e+09
UNC_QMC_NORMAL_READS_ANY	8.17606e+07	0	0	0	8.07797e+07	0
UNC_QMC_WRITES_FULL_ANY	5.53837e+07	0	0	0	5.51052e+07	0
UNC_QHL_REQUESTS_REMOTE_READS	6.84504e+07	0	0	0	6.8107e+07	0
UNC_QHL_REQUESTS_LOCAL_READS	6.82751e+07	0	0	0	6.76274e+07	0

RDTSC timing: 0.827196 s

Metric	core 0	core 1	core 2	core 3	core 4	core 5	core 6	core 7
Runtime [s]	0.714167	0.714733	0.71481	0.715013	0.714673	0.715286	0.71486	0.71515
CPI	3.65735	3.63188	3.65488	3.64076	3.60768	3.60521	3.59613	3.60184
Memory bandwidth [MBytes/s]	10610.8	0	0	0	10513.4	0	0	0
Remote Read BW [MBytes/s]	5296	0	0	0	5269.43	0	0	0

Half of read BW comes from other socket!

If all fails…

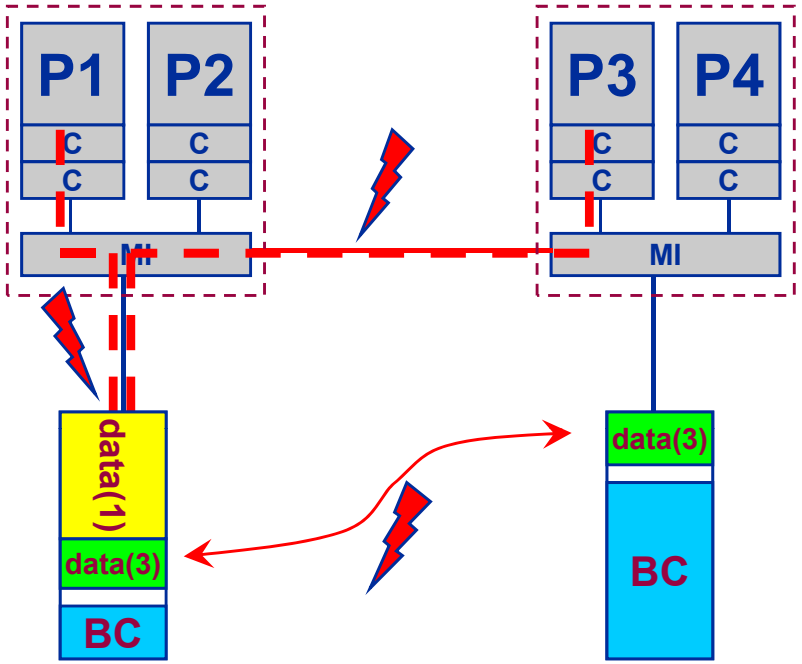
- **Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?**
  - Program has erratic access patters → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

```
top - 14:18:25 up 92 days,  6:07,  2 users,  load average: 0.00, 0.02, 0.00
Mem:   4065564k total,  1149400k used,  2716164k free,    43388k buffers
Swap:  2104504k total,     2656k used,  2101848k free,  1038412k cached
```

ccNUMA problems beyond first touch:
Buffer cache

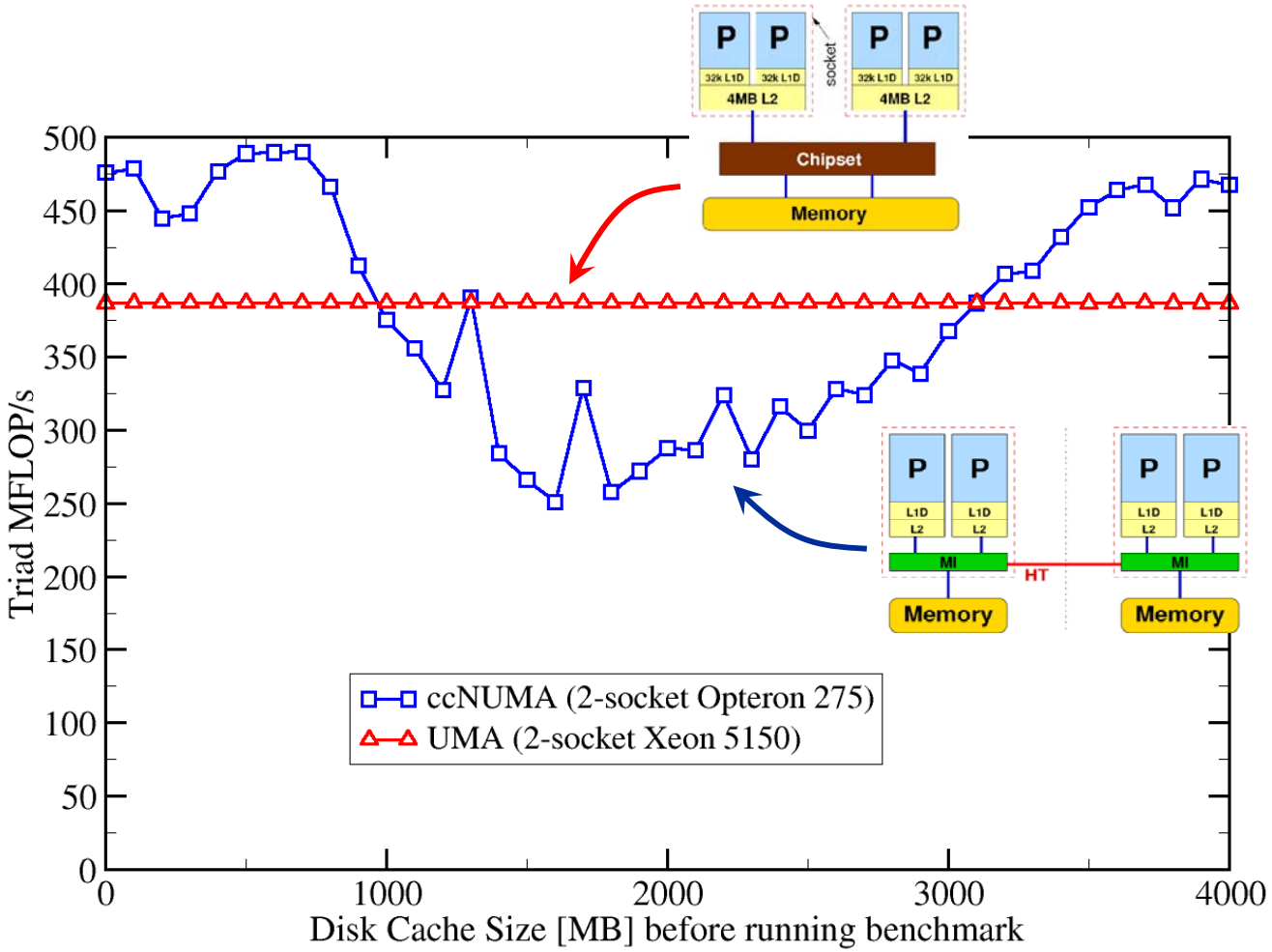
- **OS uses part of main memory for disk buffer (FS) cache**
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - → non-local access!
  - "sync" is not sufficient to drop buffer cache blocks



- **Remedies**
  - Drop FS cache pages after user job has run (admin's job)
  - User can run "sweeper" code that allocates and touches all physical memory before starting the real application
  - Linux: There is no way to limit the buffer cache size in standard kernels

ccNUMA problems beyond first touch:
Buffer cache

- **Real-world example: ccNUMA vs. UMA and the Linux buffer cache**
- **Compare two 4-way systems: AMD Opteron ccNUMA vs. Intel UMA, 4 GB main memory**
- **Run 4 concurrent triads (512 MB each) after writing a large file**
- **Report performance vs. file size**
- **Drop FS cache after each data point**



ccNUMA placement and erratic access patterns

- Sometimes access patterns are just not nicely grouped into contiguous chunks:

```
double precision :: r, a(M)
!$OMP parallel do private(r)
do i=1,N
  call RANDOM_NUMBER(r)
  ind = int(r * M) + 1
  res(i) = res(i) + a(ind)
enddo
!OMP end parallel do
```

- Or you have to use tasking/dynamic scheduling:

```
!$OMP parallel
!$OMP single
do i=1,N
  call RANDOM_NUMBER(r)
  if(r.le.0.5d0) then
!$OMP task
    call do_work_with(p(i))
!$OMP end task
  endif
enddo
!$OMP end single
!$OMP end parallel
```

- In both cases page placement cannot easily be fixed for perfect parallel access

ccNUMA placement and erratic access patterns

- **Worth a try: Interleave memory across ccNUMA domains to get at least some parallel access**

  1. Explicit placement:

```
!$OMP parallel do schedule(static,512)
do i=1,M
  a(i) = …
enddo
!$OMP end parallel do
```

Observe page alignment of array to get proper placement!

  2. Using global control via `numactl`:

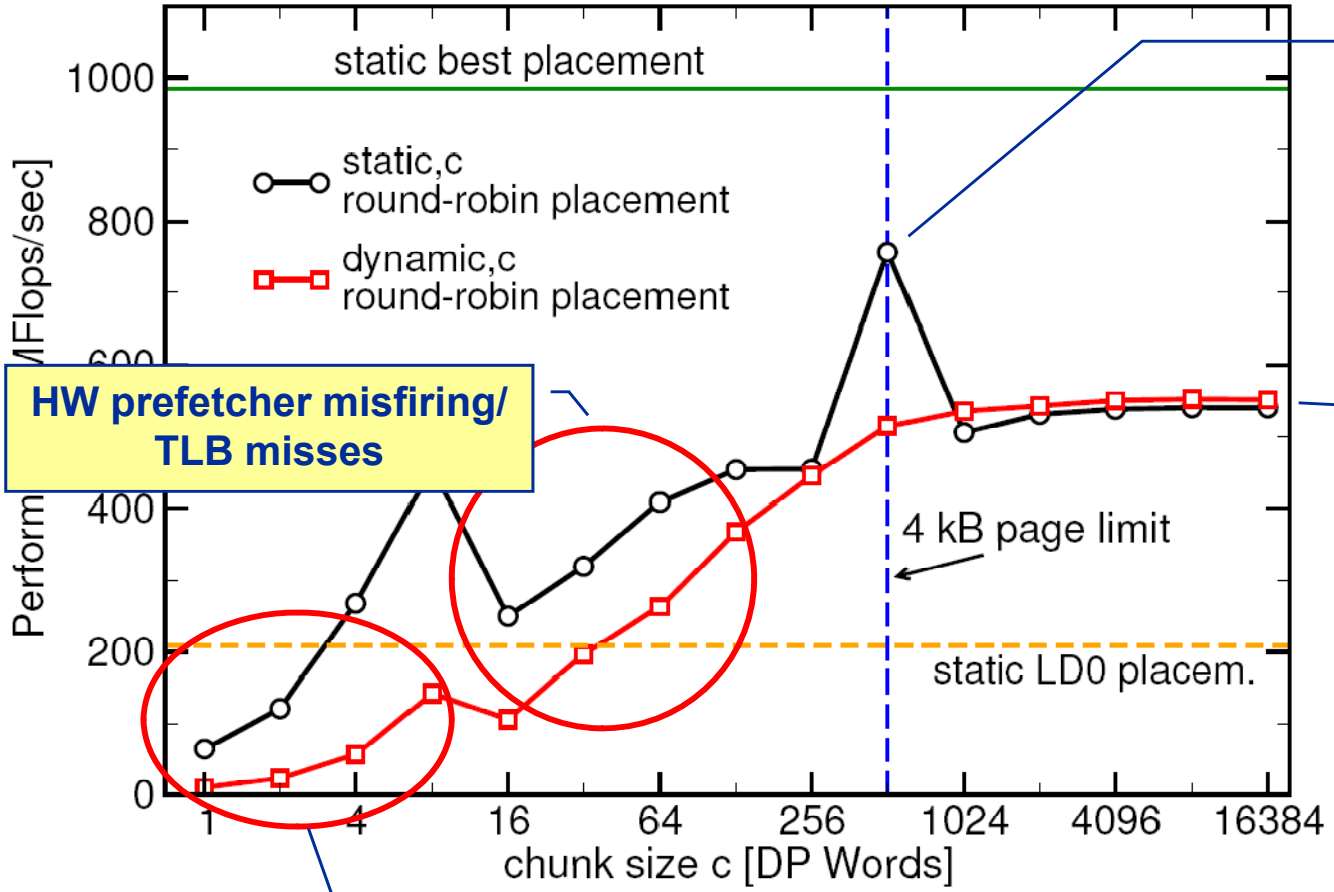
```
numactl --interleave=0-3 ./a.out
```

This is for all memory, not just the problematic arrays!

- **Fine-grained program-controlled placement via `libnuma` (Linux) using, e.g., `numa_alloc_interleaved_subset()`, `numa_alloc_interleaved()` and others**

Performance impact of round-robin page placement with dynamic scheduling/tasking

- OpenMP vector triad benchmark `A(:)=B(:)+C(:)*D(:)` with large array lengths on a 4-LD ccNUMA machine
- Round-robin page placement (see previous slide)
- Static vs. dynamic loop scheduling, varying chunk size

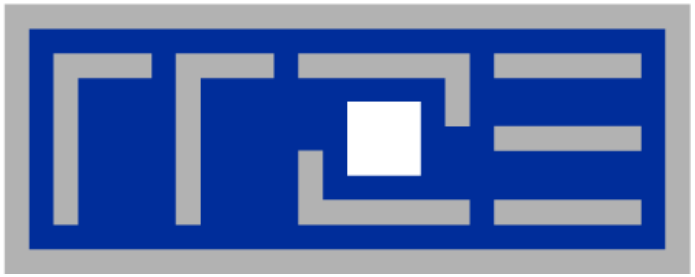


Static loop schedule matches initialization, but no page alignment of arrays

HW prefetcher misfiring/ TLB misses

Asymptotic limit: 75% of all page accesses are nonlocal

Full cache line transfer, partial access



OpenMP performance issues on multicore

Synchronization (barrier) overhead

Work distribution overhead

```
!$OMP PARALLEL …
…
!$OMP BARRIER
!$OMP DO
…
!$OMP ENDDO
!$OMP END PARALLEL
```

Threads are synchronized at **explicit** AND **implicit** barriers.

Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

On x86 systems there is no hardware support for synchronization.

- **Tested synchronization constructs**
  - `OpenMP` Barrier
  - `pthreads` Barrier
  - **Spin waiting loop** software solution
- **Test machines (Linux OS):**
  - Intel Core 2 Quad Q9550 (2.83 GHz)
  - Intel Core i7 920 (2.66 GHz)

Thread synchronization overhead

Barrier overhead in CPU cycles: 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

Barrier overhead: OpenMP icc vs. gcc

gcc obviously uses a pthreads barrier for the OpenMP barrier:

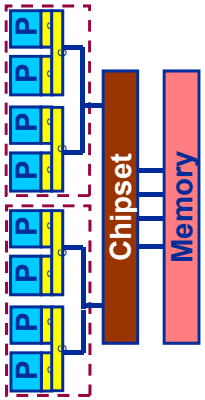
2 Threads	Q9550 (shared L2)	i7 920 (shared L3)
gcc 4.3.3	22603	7333
icc 11.0	399	469

4 Threads	Q9550	i7 920 (shared L3)
gcc 4.3.3	64143	10901
icc 11.0	977	814

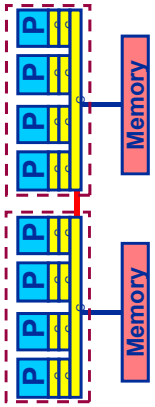
→ **Affinity enforcement is vital for getting small, reproducible sync overhead!**

Thread synchronization overhead

Barrier 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

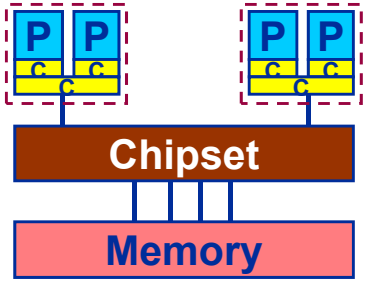
- SMT can be a big performance problem for synchronizing threads
  - Well known for a long time → see below
- Roll-your-own sync mechanism may be better sometimes, but good compilers do a good job, too

Work distribution overhead

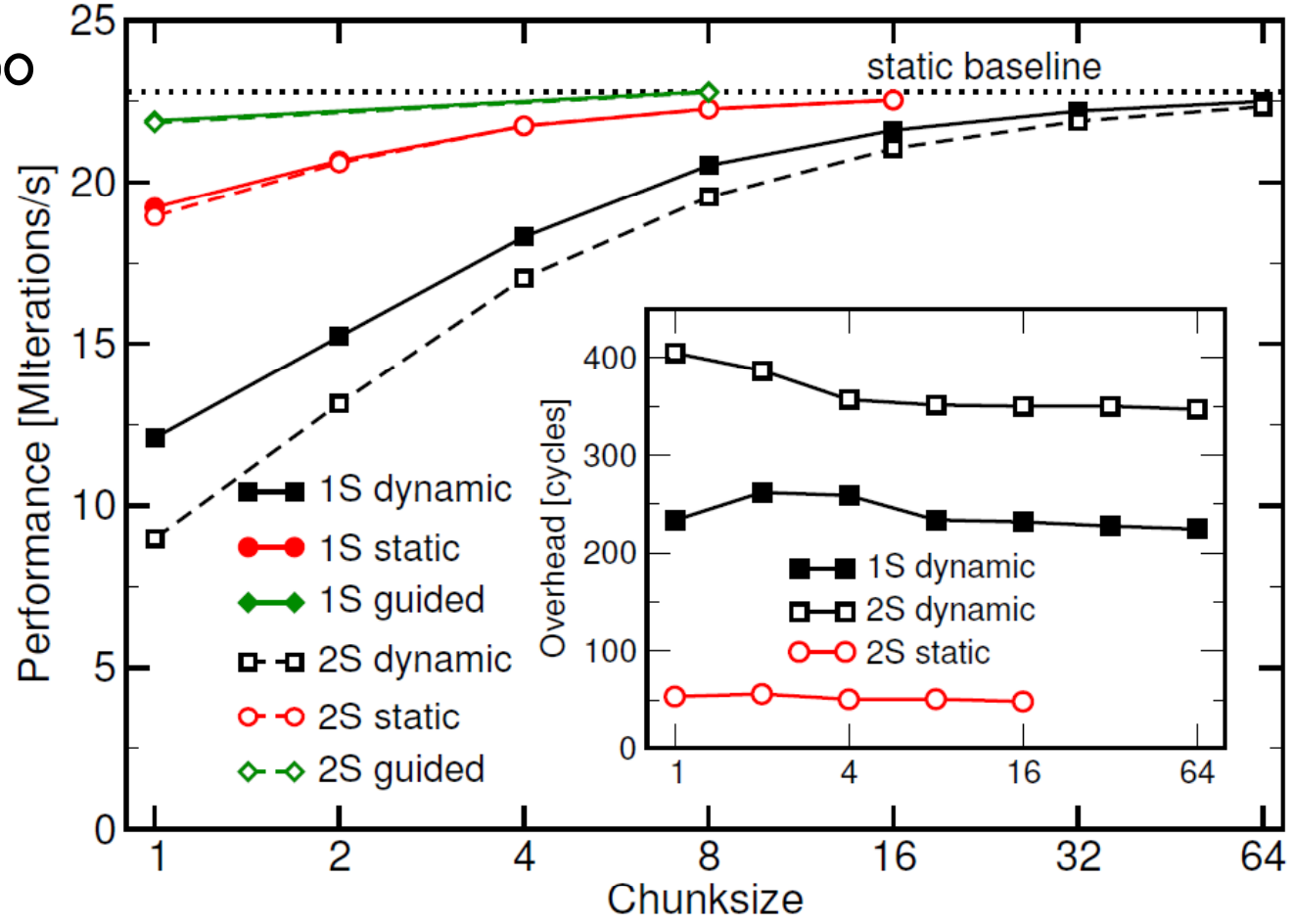
Influence of thread-core affinity

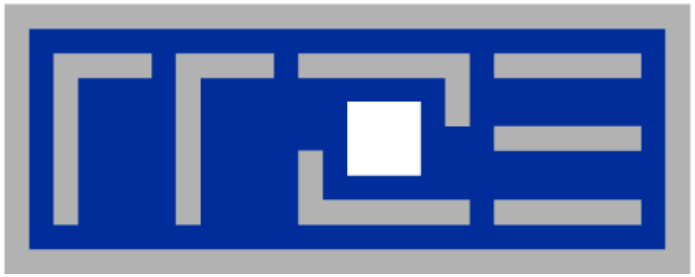
- **Overhead microbenchmark:**

```
!$OMP PARALLEL DO SCHEDULE(RUNTIME) REDUCTION(+:s)
do i=1,N
  s = s + compute(i)
enddo
!$OMP END PARALLEL DO
```



- **Choose N large so that synchronization overhead is negligible**
- **compute() implements purely computational workload**
  - ➔ **no bandwidth effects**
- **Run with 2 threads**



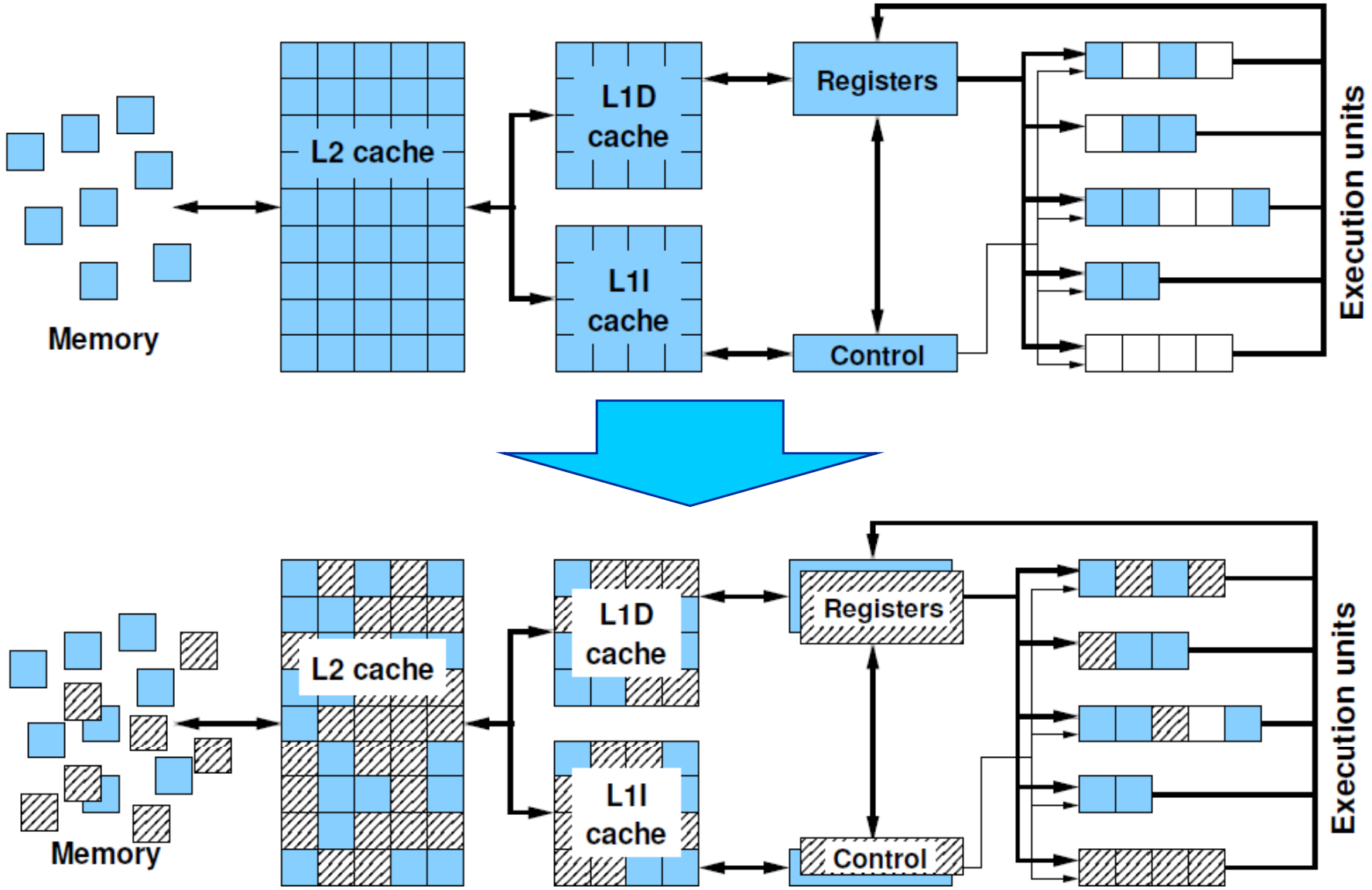


Simultaneous multi-threading

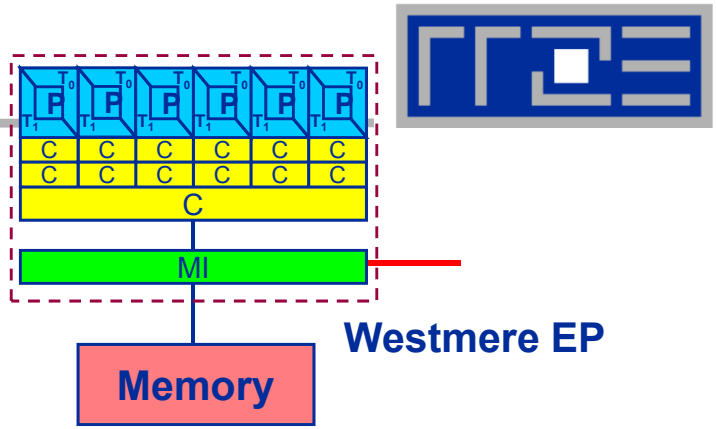
Principles and performance impact

SMT Makes a single physical core appear as two or more "logical" cores → multiple threads/processes run concurrently

- **SMT principle (2-way example):**



SMT impact



- **SMT adds another layer of topology (inside the physical core)**

- **Possible benefit: Better pipeline throughput**
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread's executing instructions:

Thread 0:

```
do i=1,N
  a(i) = a(i-1)*c
enddo
```

Thread 1:

```
do i=1,N
  b(i) = func(i)*d
enddo
```

Dependency → pipeline stalls until previous MULT is over

Unrelated work in other thread can fill the pipeline bubbles

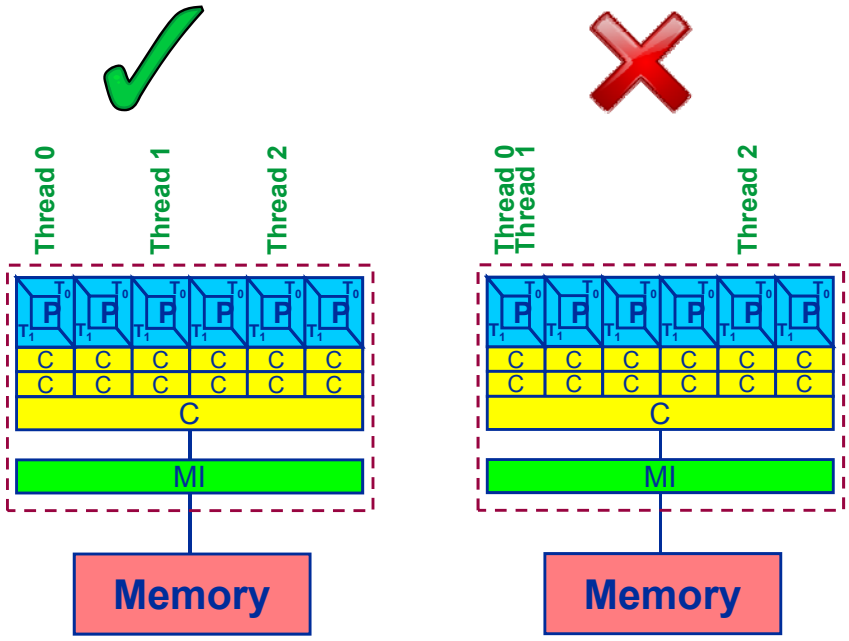
  - Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = func(i)*d
enddo
```

SMT impact

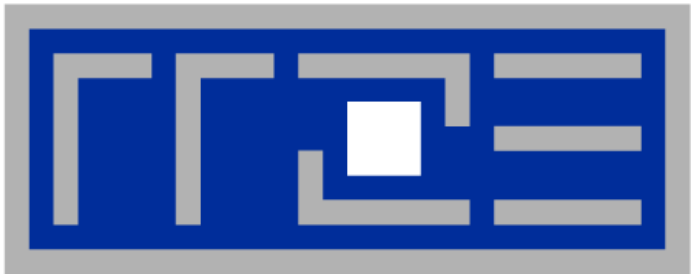
- **SMT is primarily suited for increasing processor throughput**
  - With multiple threads/processes running concurrently
- **Scientific codes tend to utilize chip resources quite well**
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- **SMT is an important topology issue**
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.



SMT: When it may help, and when not

Functional parallelization	✓ ✗
FP-only parallel loop code	✗
Frequent thread synchronization	✗
Code sensitive to cache size	✗
Strongly memory-bound code	✗
Independent pipeline-unfriendly instruction streams	✓



Understanding MPI communication in multicore environments

Intra-node vs. inter-node MPI

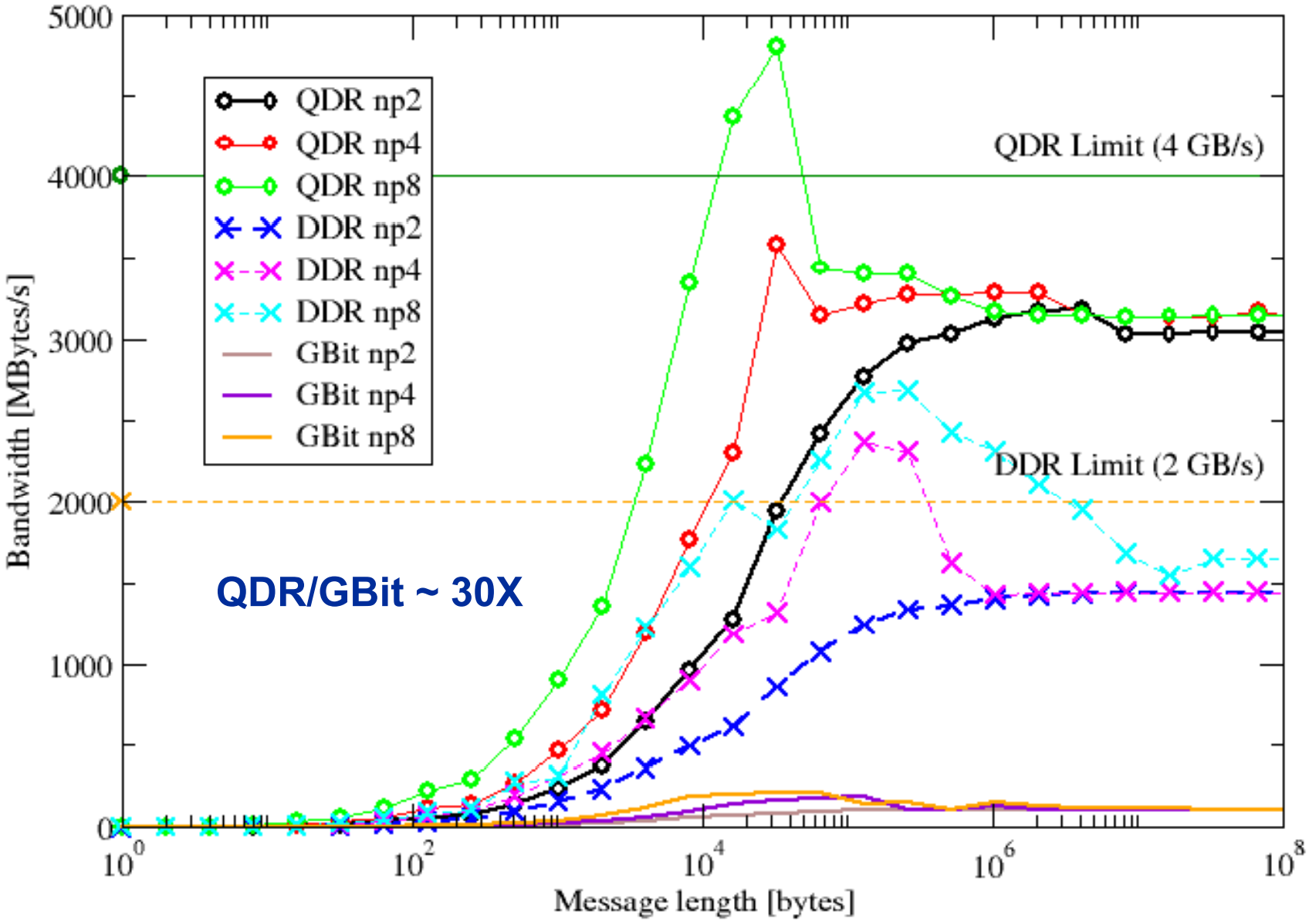
MPI Cartesian topologies and rank-subdomain mapping

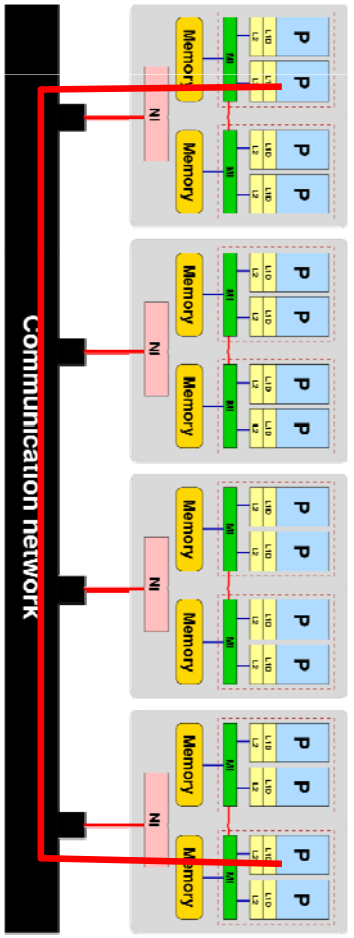
Intranode MPI

- **Common misconception: Intranode MPI is infinitely fast compared to internode**
- **Reality**
  - Intranode latency is much smaller than internode
  - Intranode asymptotic bandwidth is surprisingly comparable to internode
  - Difference in saturation behavior
- **Other issues**
  - Mapping between ranks, subdomains and cores with Cartesian MPI topologies
  - Overlapping intranode with internode communication

MPI and Multicores

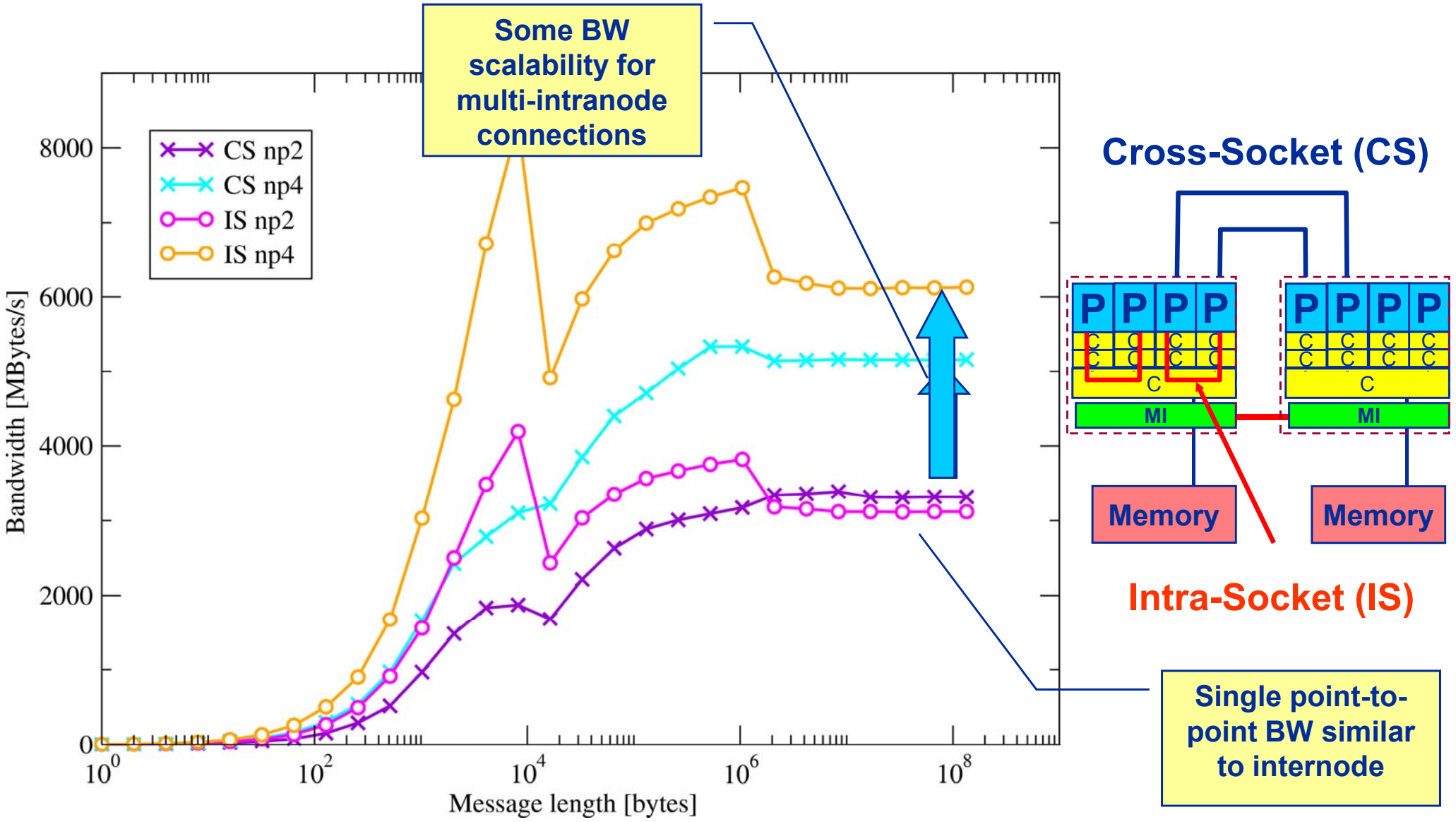
Clusters: Unidirectional internode Ping-Pong bandwidth





MPI and Multicores

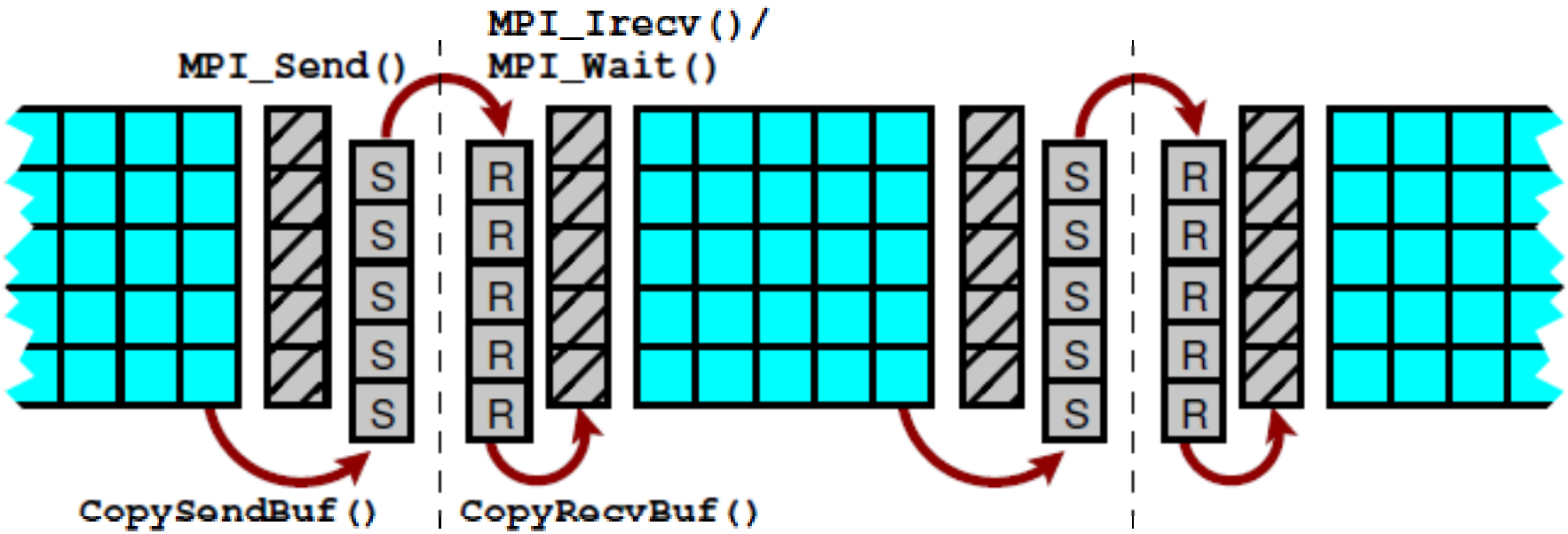
Clusters: Unidirectional intranode Ping-Pong bandwidth



Mapping problem for most efficient communication paths!?

“Best possible” MPI:
Minimizing cross-node communication

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

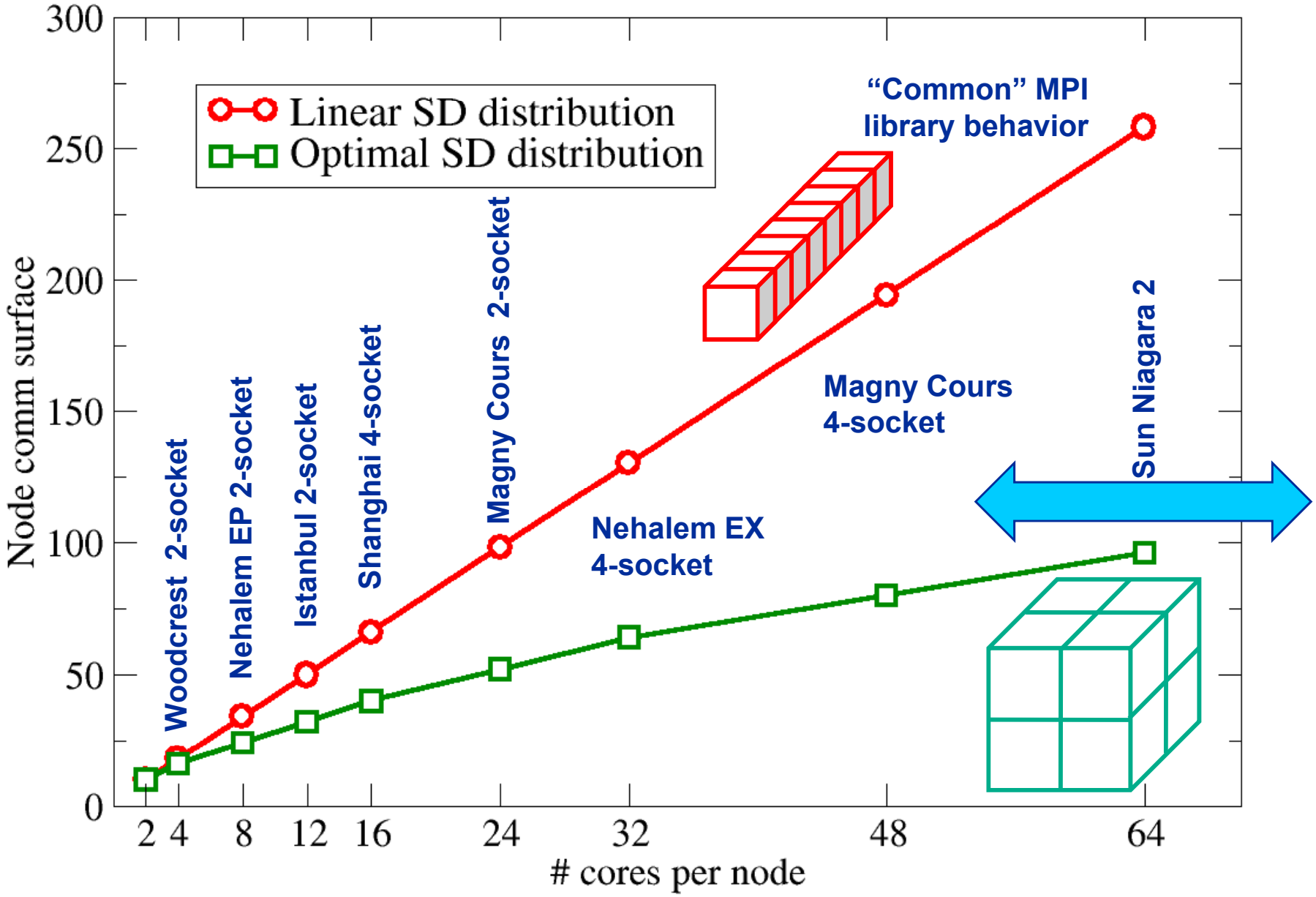


- **Goal: Reduce inter-node halo traffic**
- **Subdomains exchange halo with neighbors**
  - Populate a node's ranks with “maximum neighboring” subdomains
  - This minimizes a node's communication surface

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

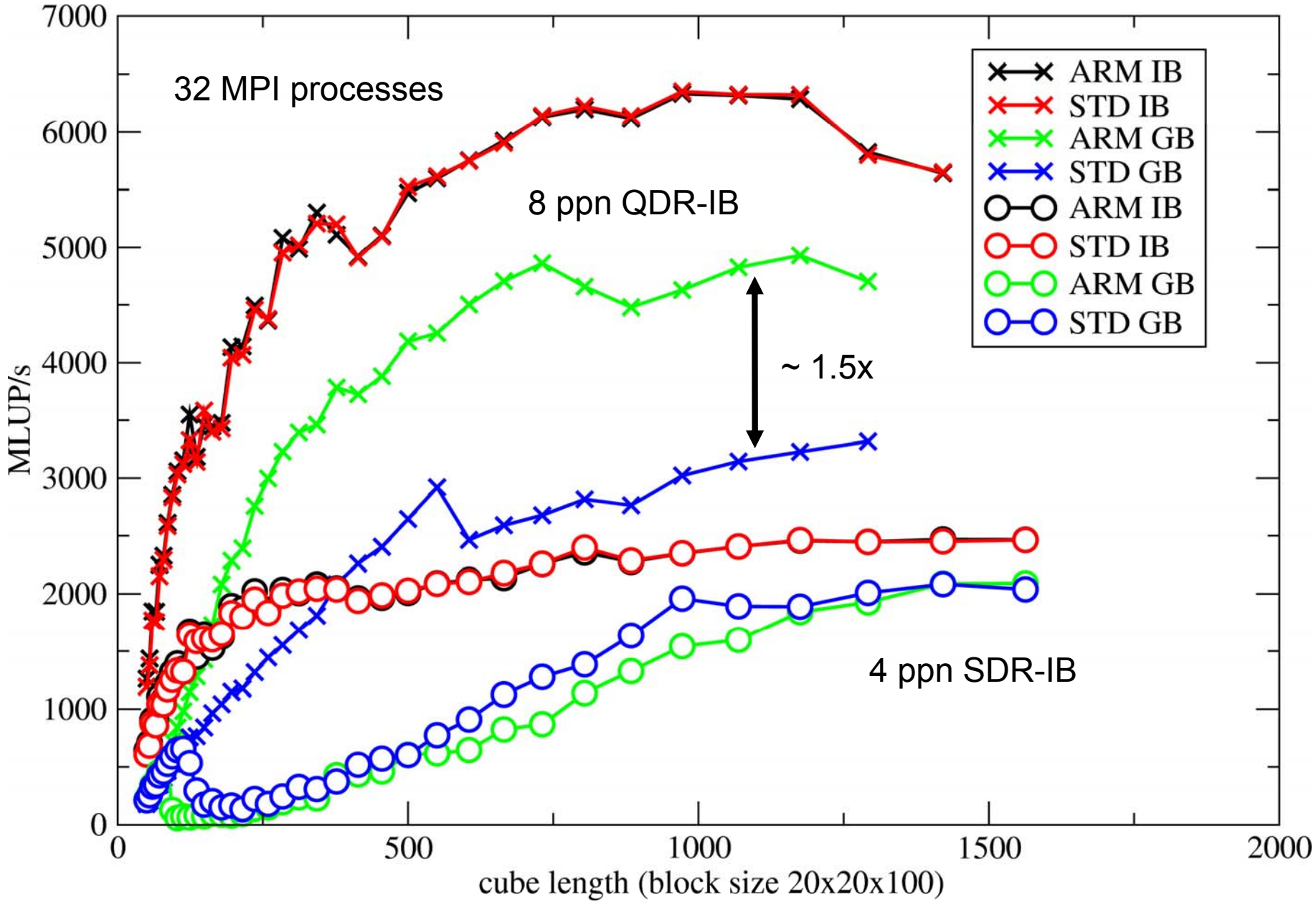
MPI rank-subdomain mapping in Cartesian topologies:

A 3D stencil solver and the growing number of cores per node



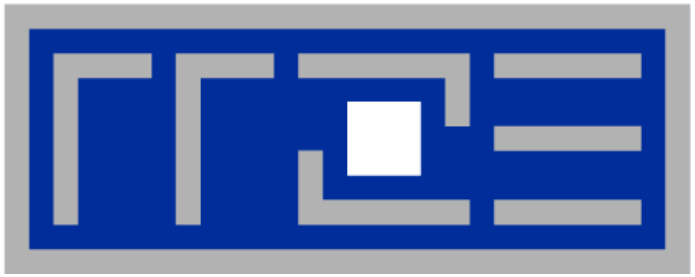
MPI rank-subdomain mapping:

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



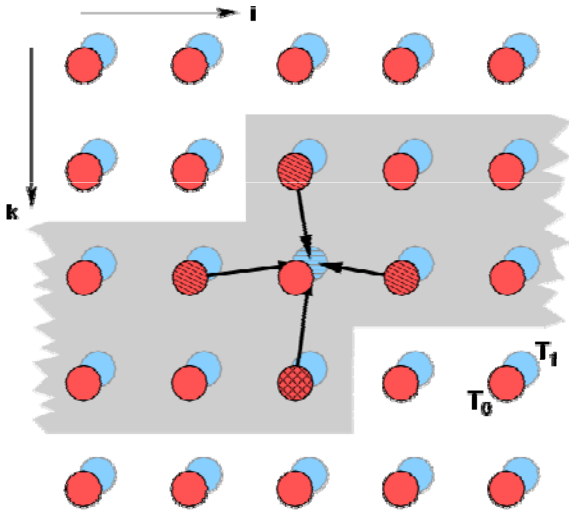
Section summary: What to take home

- **Bandwidth saturation is a reality, in cache and memory**
  - Use knowledge to choose the "right" number of threads/processes per node
  - You must know where those threads/processes should run
  - You must know the architectural requirements of your application
- **ccNUMA architecture must be considered for bandwidth-bound code**
  - Topology awareness, again
  - First touch page placement
  - Problems with dynamic scheduling and tasking: Round-robin placement is the "cheap way out"
- **OpenMP overhead**
  - Barrier (synchronization) often dominates the loop overhead
  - Work distribution and sync overhead is strongly topology-dependent
  - Strong influence of compiler
  - Synchronizing threads on "logical cores" (SMT threads) may be expensive
- **Intranode MPI**
  - May not be as fast as you think…
  - Becomes more important as core counts increase
  - May not be handled optimally by your MPI library



Interlude:
What can software do for you?

Automatic parallelization for moderate processor counts has been known for more than 15 years – simple testbed for modern multicores:




```
allocate( x(0:N+1,0:N+1,0:N+1) )
allocate( y(0:N+1,0:N+1,0:N+1) )
x=0.d0
y=0.d0
…
… somewhere in a subroutine …
do k = 1,N
  do j = 1,N
    do i = 1,N
      y(i,j,k) = b*(x(i-1,j,k)+x(i+1,j,k)+ x(i,j-1,k)+
                    x(i,j+1,k)+x(i,j,k-1)+x(i,j,k+1))
    enddo
  enddo
enddo
```

Simple 3D 7-point stencil update("Jacobi")

Performance Metric: Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs: 6 FLOP/LUP * MLUPs
Equivalent GByte/s: 24 Byte/LUP * MLUPs

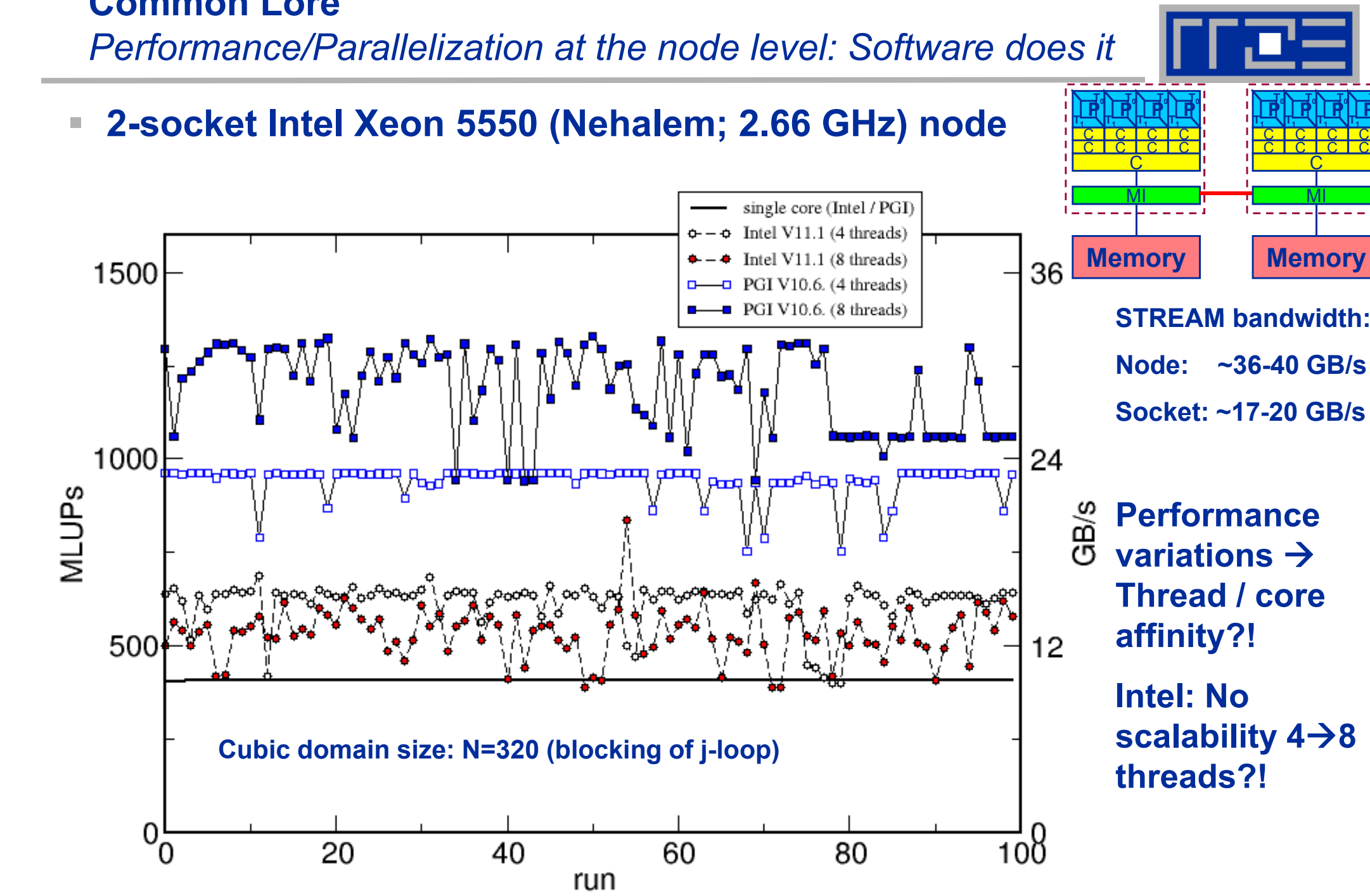
- **Intel Fortran compiler:**
  `ifort -O3 -xW -parallel -par-report2 …`

  - Version 9.1 (admittedly an older one…)
    - Innermost i-loop is SIMD vectorized, which prevents compiler from auto-parallelization: `serial loop: line 141: not a parallel candidate due to loop already vectorized`
    - No other loop is parallelized…

  - Version 11.1
    - Outermost k-loop is parallelized: `Jacobi_3D.F(139): (col. 10) remark: LOOP WAS AUTO-PARALLELIZED.`
    - Innermost i-loop is vectorized.
    - Most other loop structures are ignored by "parallelizer", e.g. `x=0.d0` and `y=0.d0`: `Jacobi_3D.F(37): (col. 16) remark: loop was not parallelized: insufficient computational work`

- **PGI compiler (V 10.6)**
  `pgf90 -tp nehalem-64 -fastsse -Mconcur -Minfo=par,vect`
  - Performs outer loop parallelization of k-loop
    ```
    139, Parallel code generated with block distribution if
    trip count is greater than or equal to 33
    ```
  - and vectorization of inner i-loop:
    ```
    141, Generated 4 alternate loops for the loop Generated
    vector sse code for the loop
    ```
  - Also the array instructions (**`x=0.d0; y=0.d0`**) used for initialization are parallelized:
    ```
    37, Parallel code generated with block distribution if
    trip count is greater than or equal to 50
    ```
  - Version 7.2. does the same job but some switches must be adapted

- **gfortran: No automatic parallelization feature so far (?!)**



- **2-socket Intel Xeon 5550 (Nehalem; 2.66 GHz) node**

STREAM bandwidth:

Node: ~36-40 GB/s

Socket: ~17-20 GB/s

Performance variations → Thread / core affinity?!

Intel: No scalability 4→8 threads?!

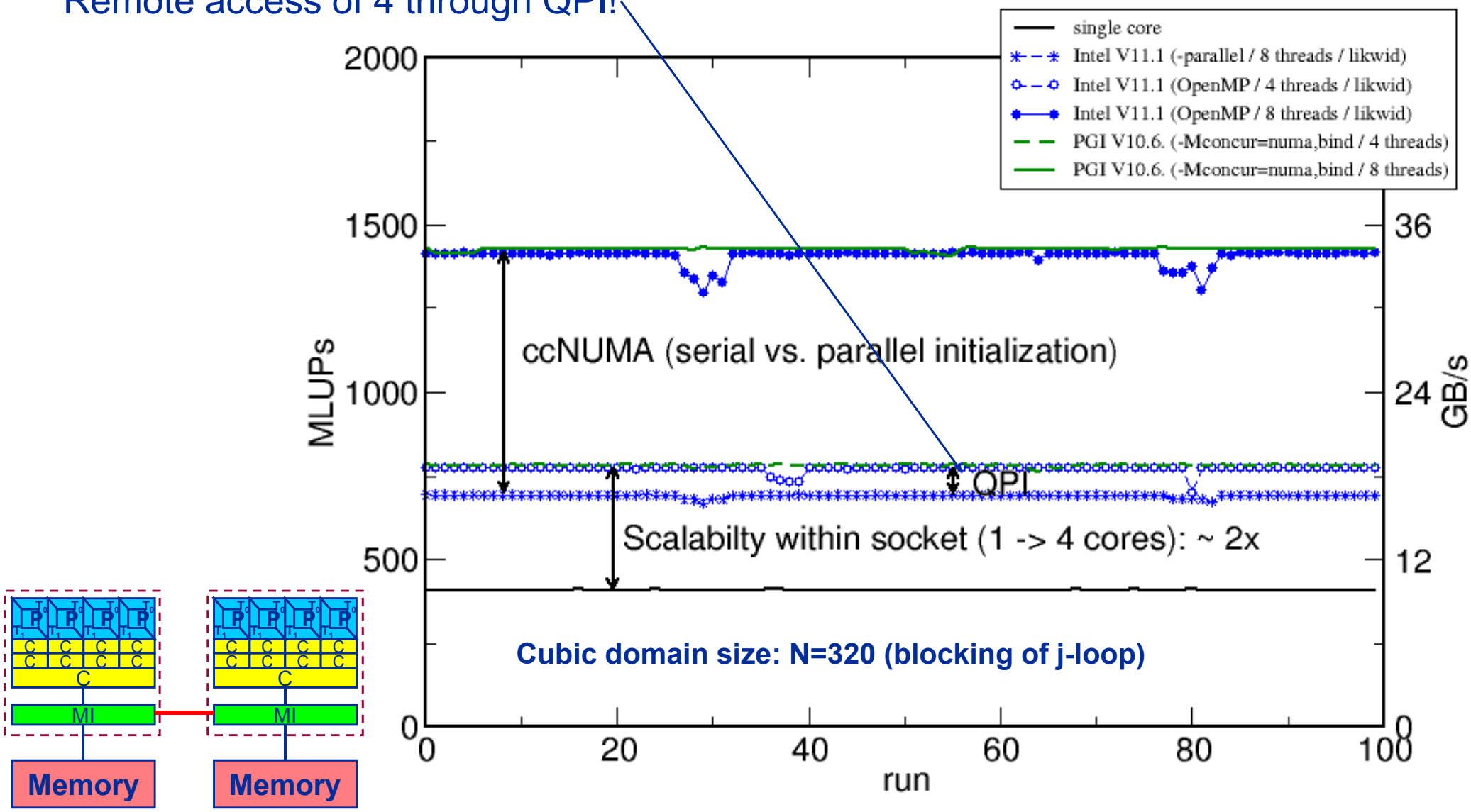
Controlling thread affinity / binding
Intel / PGI compilers

- **Intel compiler controls thread-core affinity via `KMP_AFFINITY` environment variable**
  - **`KMP_AFFINITY="granularity=fine,compact,1,0"`** is packs the threads in a blockwise fashion ignoring the SMT threads. (equivalent to **`likwid-pin -c 0-7`**)
  - Add **"`verbose`"** to get information at runtime
  - Cf. extensive Intel documentation
  - Disable when using other tools (automatic w/ LIKWID): **`KMP_AFFINITY=disabled`**
  - Builtin affinity does not work on non-Intel hardware

- **PGI compiler offers compiler options:**
  - **`Mconcur=bind`** (binds threads to cores; link time option)
  - **`Mconcur=numa`** (prevents OS from process / thread migration; link time option)
  - No manual control of thread-core affinity
  - **Interaction LIKWID ←→ PGI ?!**

Thread binding and ccNUMA effects
7-point 3D stencil on 2-socket Intel Nehalem system

- Performance drops if 8 threads instead of 4 access a single memory domain: Remote access of 4 through QPI!



Thread binding and ccNUMA effects
7-point 3D stencil on 2-socket AMD Magny-Cours system

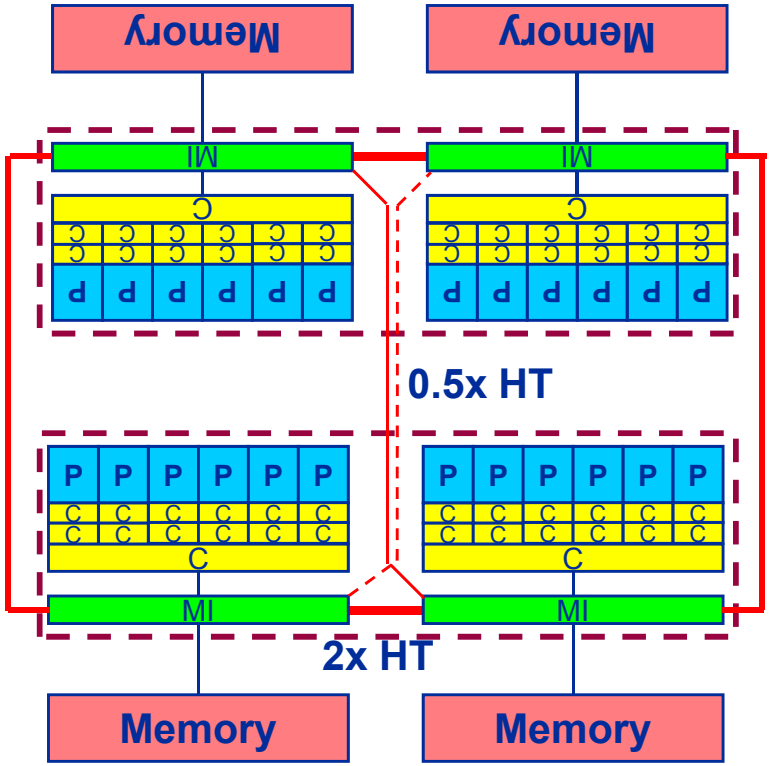
- 12-core Magny-Cours: A single socket holds two tightly HT-connected 6-core chips → 2-socket system has 4 data locality domains

Cubic domain size: N=320 (blocking of j-loop)

`OMP_SCHEDULE="static"`

Performance [MLUPs]

#threads	#L3 groups	#sockets	Serial Init.	Parallel Init.
1	1	1	221	221
6	1	1	512	512
12	2	1	347	1005
24	4	2	286	1860



3 levels of HT connections:

1.5x HT – 1x HT – 0.5x HT

Based on Jacobi performance results one could claim victory, but now we increase complexity a bit: Gauss-Seidel instead of Jacobi

```
… somewhere in a subroutine …
do k = 1,N
  do j = 1,N
       do i = 1,N
          x(i,j,k) = b*(x(i-1,j,k)+x(i+1,j,k)+ x(i,j-1,k)+
                        x(i,j+1,k)+x(i,j,k-1)+ x(i,j,k+1))
    enddo
  enddo
enddo
```

3D 7-point stencil update("Gauss-Seidel") with loop-carried dependencies

Performance Metric:	Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs:	6 FLOP/LUP * MLUPs
Equivalent GByte/s:	16 Byte/LUP * MLUPs

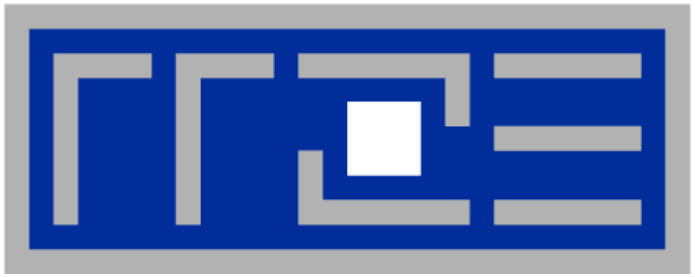
Performance of Gauß-Seidel should be up to 1.5x faster than Jacobi if main memory bandwidth is the limitation

- **State of the art compilers do not parallelize the Gauss-Seidel smoother:**
  `loop was not parallelized: existence of parallel dependence`
- **That's true but there are simple ways to remove the dependency even for the lexicographic Gauss-Seidel**
- **10 yrs+ ago Hitachi's compiler supported "pipeline parallel processing" (cf. later slides for more details on this technique)!**
- **There seem to be major problems to optimize even the serial code**
  - 1 Intel Xeon X5550 (2.66 GHz) core
  - Reference: Jacobi
    430 MLUPs
  - **Target Gauss-Seidel:**
    645 MLUPs

Intel V9.1.	**290 MLUPs**
Intel V11.1.072	**345 MLUPs**
pgf90 V10.6.	**149 MLUPs**
pgf90 V7.2.1	**149 MLUPs**

Tutorial outline

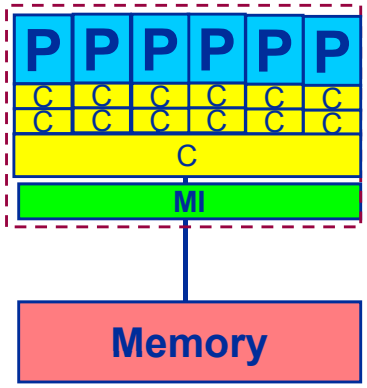
- Introduction
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- Multicore performance tools
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- Impact of processor/node topology on program performance
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **New chances with multicore hardware**
  - Wavefront parallelization of stencil codes
  - Explicit comm/calc overlap in sparse MVM
- Summary
- Appendix



New chances with multicore hardware

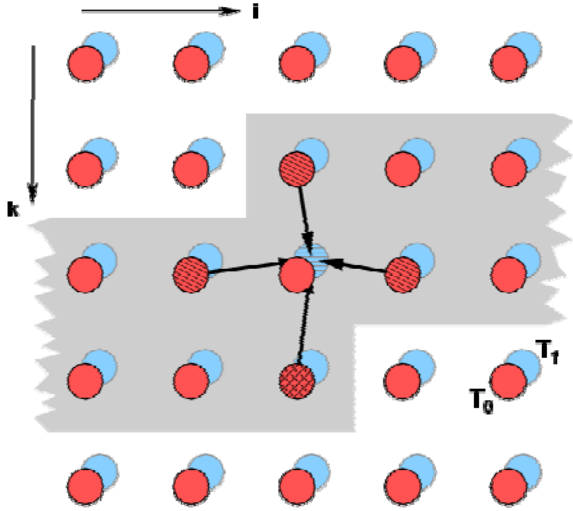
Leveraging shared caches:
Wavefront parallelization of stencil codes

Multicore processors are still mostly programmed the same way as classic n-way SMP single-core compute nodes!



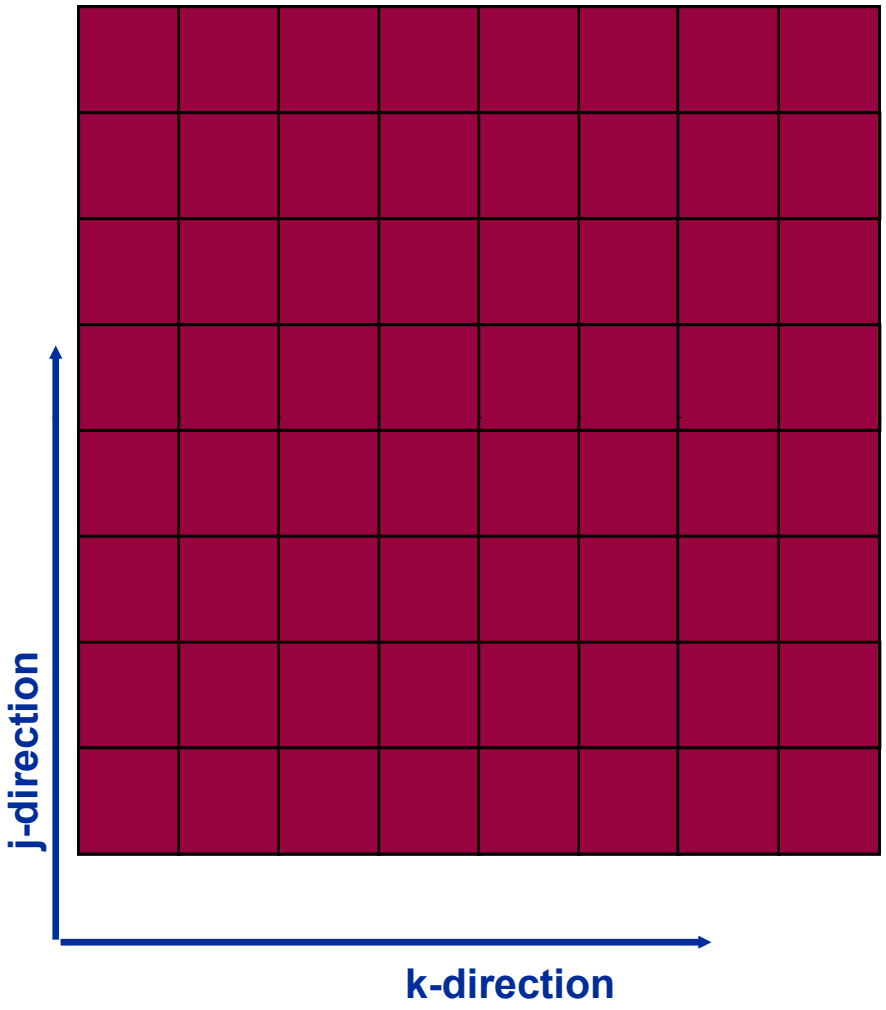
Simple 3D Jacobi stencil update (sweep):

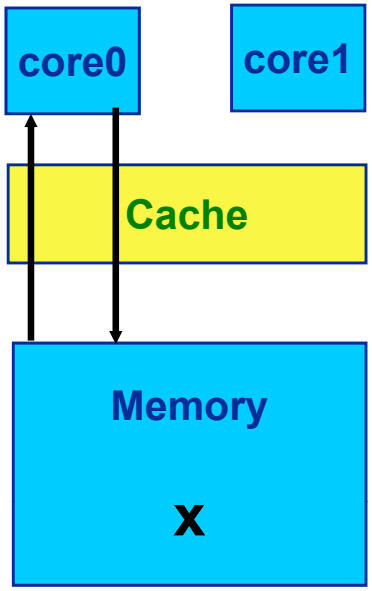
```
do k = 1 , Nk
  do j = 1 , Nj
       do i = 1 , Ni
          y(i,j,k) = a*x(i,j,k) + b*
                     (x(i-1,j,k)+x(i+1,j,k)+
                      x(i,j-1,k)+x(i,j+1,k)+
                      x(i,j,k-1)+x(i,j,k+1))
    enddo
  enddo
enddo
```



Performance Metric: Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs: 8 FLOP/LUP * MLUPs

Multicore awareness
Standard sequential implementation

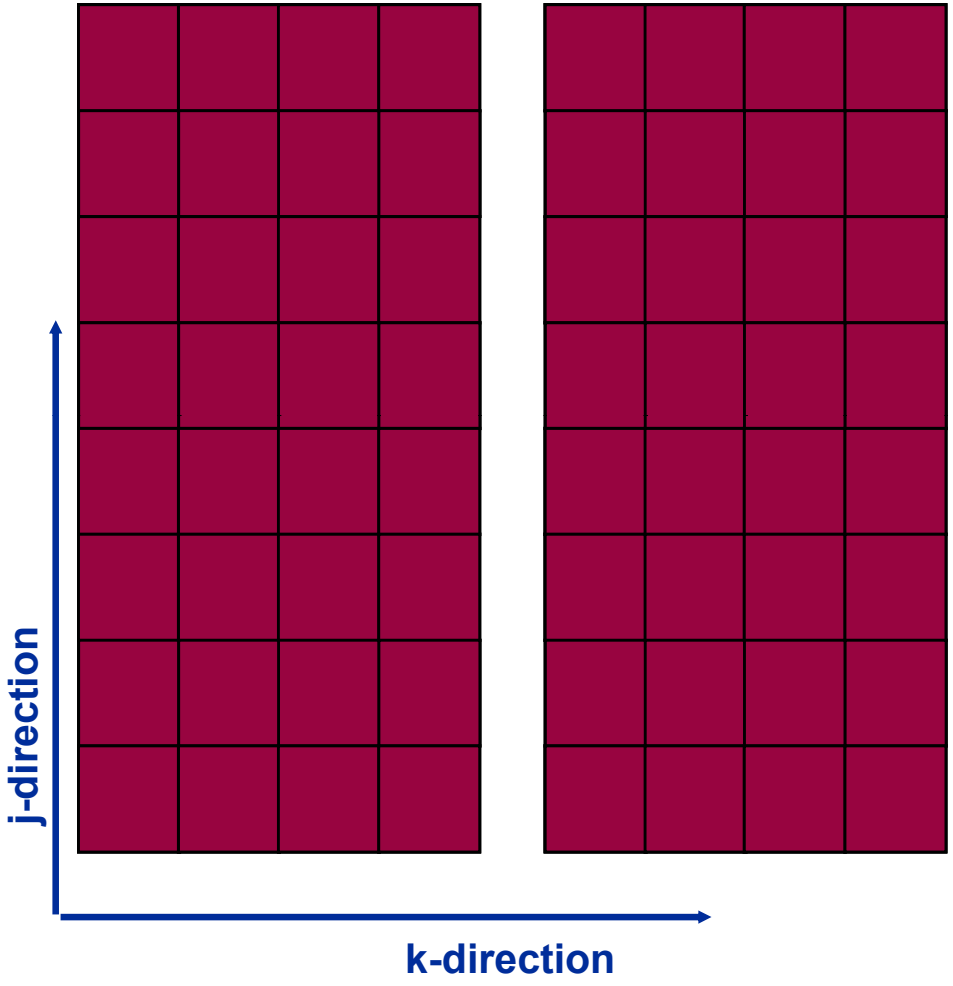


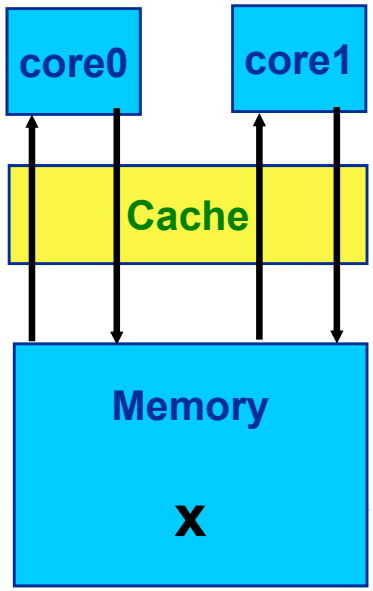



```
do t=1,tMax

   do k=1,N
      do j=1,N
         do i=1,N
            y(i,j,k) = …
         enddo
      enddo
   enddo

enddo
```

Multicore awareness

Classical Approaches: Parallelize!

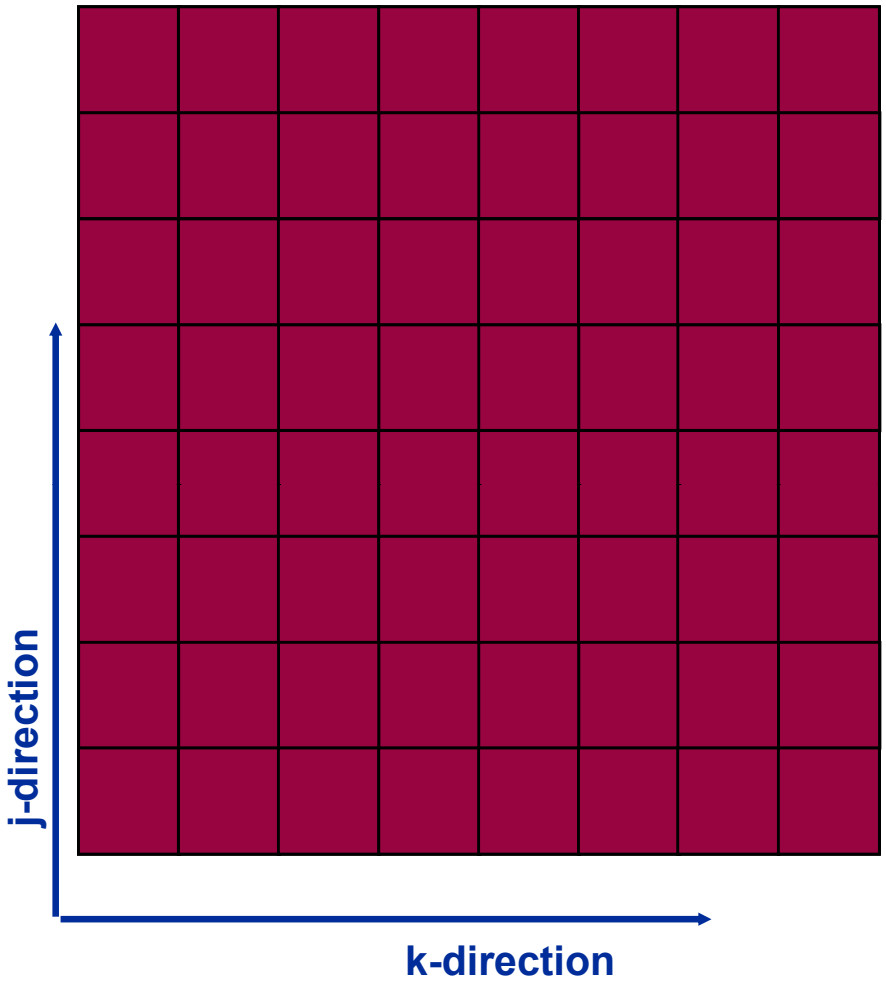


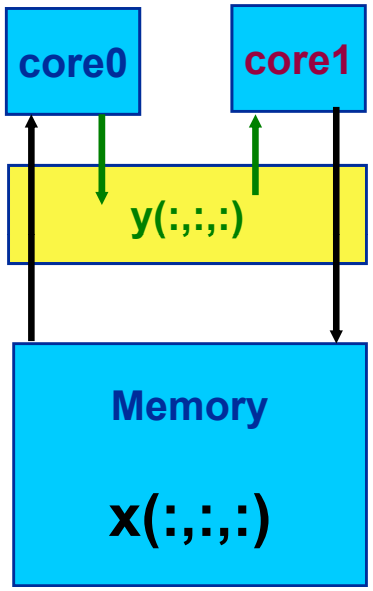



```
do t=1,tMax
!$OMP PARALLEL DO private(…)
   do k=1,N
      do j=1,N
         do i=1,N
            y(i,j,k) = …
         enddo
      enddo
   enddo
!$OMP END PARALLEL DO
enddo
```

Multicore awareness

Parallelization – reuse data in cache between threads





Do not use domain decomposition!

Instead shift 2nd thread by three i-j planes and proceed to the same domain
→ 2nd thread loads input data from shared OL cache!

Sync threads/cores after each k-iteration!

"Wavefront Parallelization (WFP)"

core0: $x(:,:,k-1:k+1)_{t}$ → $y(:,:,k)_{t+1}$

core1: $y(:,:,(k-3):(k-1))_{t+1}$ → $x(:,:,k-2)_{t+2}$

Multicore awareness

WF parallelization – reuse data in cache between threads

Use small ring buffer

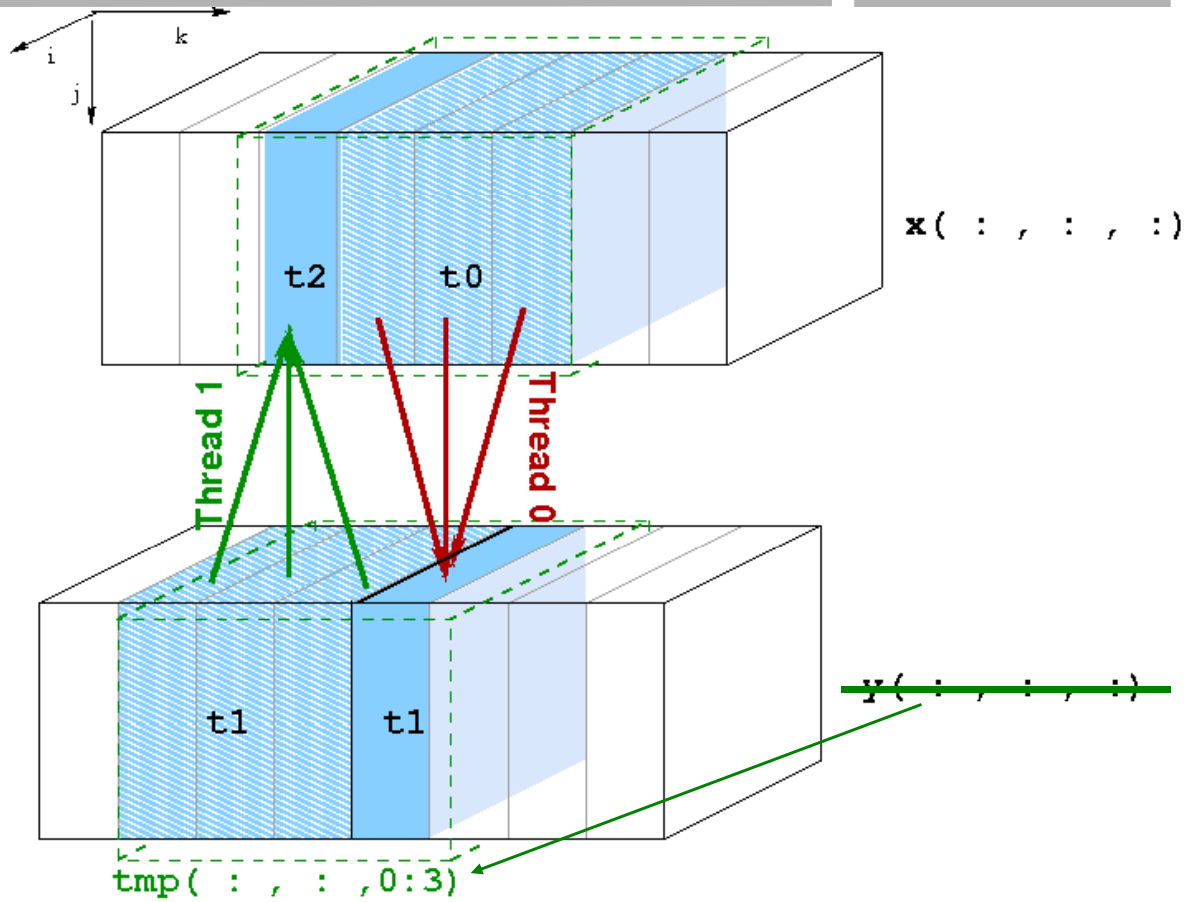
`tmp(:,:,0:3)`

which fits into the cache

Save main memory data transfers for `y(:,:,:)` !

16 Byte / 2 LUP !

8 Byte / LUP !



Compare with optimal baseline (nontemporal stores on y):
Maximum speedup of 2 can be expected

(assuming infinitely fast cache and
no overhead for OMP BARRIER after each k-iteration)

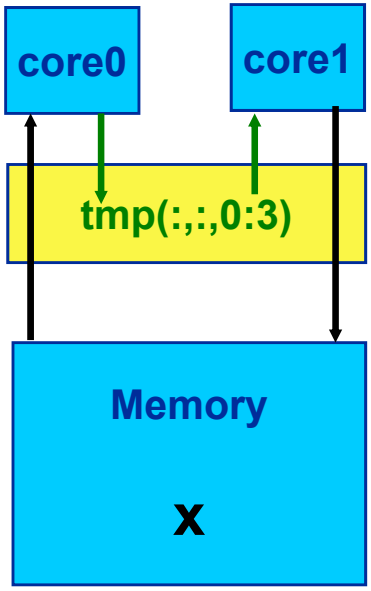
Thread 0: `x(:,:,k-1:k+1)`$_t$ → *`tmp(:,:,mod(k,4))`*

Thread 1: *`tmp(:,:,mod(k-3,4):mod(k-1,4))`* → `x(:,:,k-2)`$_{t+2}$

Performance model including finite cache bandwidth (B_C)

Time for 2 LUP:

$$T_{2LUP} = 16\ \text{Byte}/B_M + x * 8\ \text{Byte} / B_C = T_0 (1 + x/2 * B_M/B_C)$$



Minimum value: x =2

Speedup vs. baseline:

$$S_W = 2*T_0/T_{2LUP} = 2 / (1 + B_M/B_C)$$

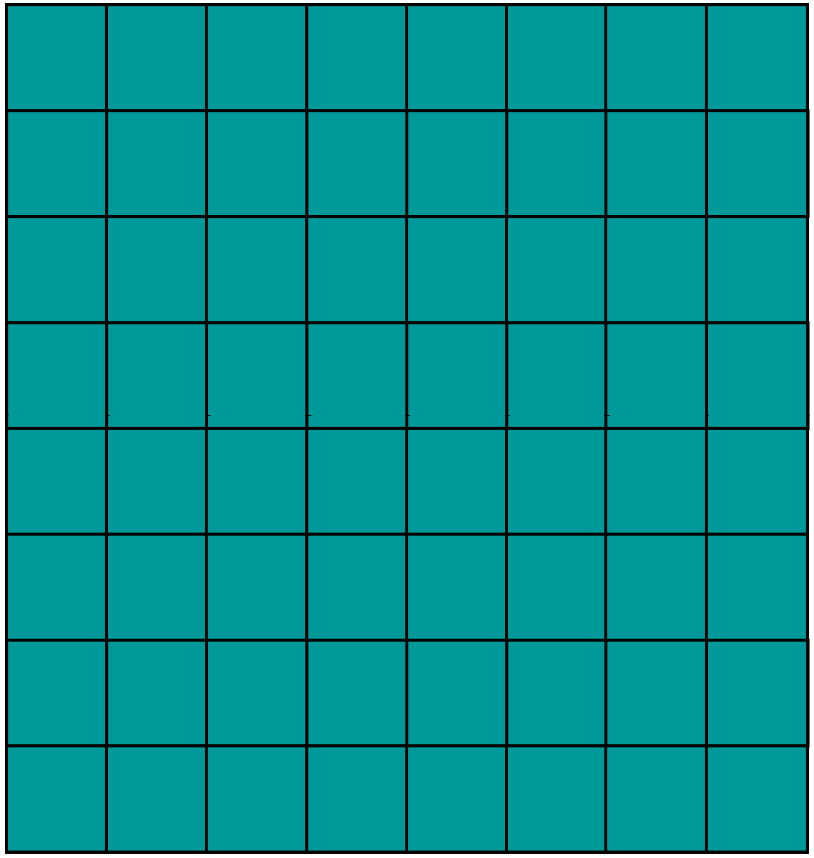
B_C and B_M are measured in saturation runs:

Clovertown: $B_M/B_C = 1/12$ → $S_W = 1.85$

Nehalem : $B_M/B_C = ¼$ → $S_W = 1.6$

Jacobi solver

WFP: Propagating four wavefronts on native quadcores (1x4)

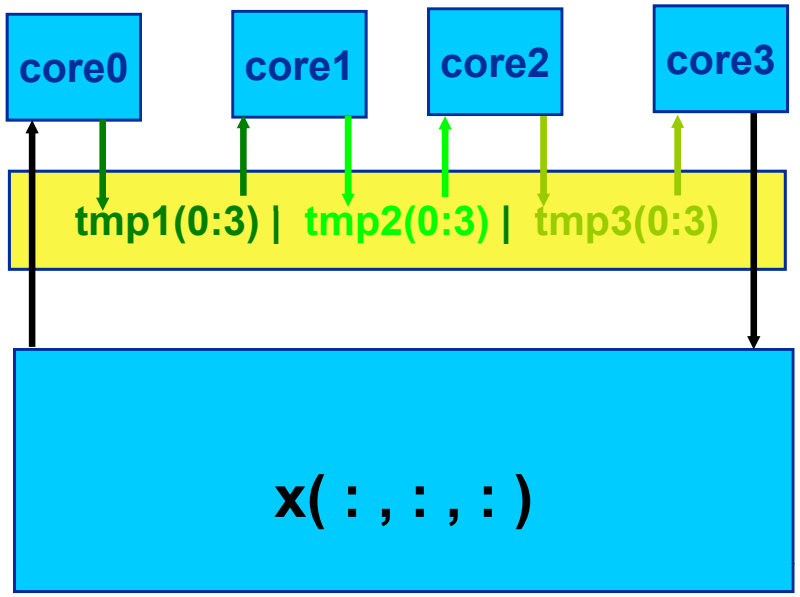


Running `tb` wavefronts requires `tb-1` temporary arrays `tmp` to be held in cache!

Max. performance gain (vs. optimal baseline): `tb = 4`

Extensive use of cache bandwidth!

1 x 4 distribution



Jacobi solver

WF parallelization: New choices on native quad-cores

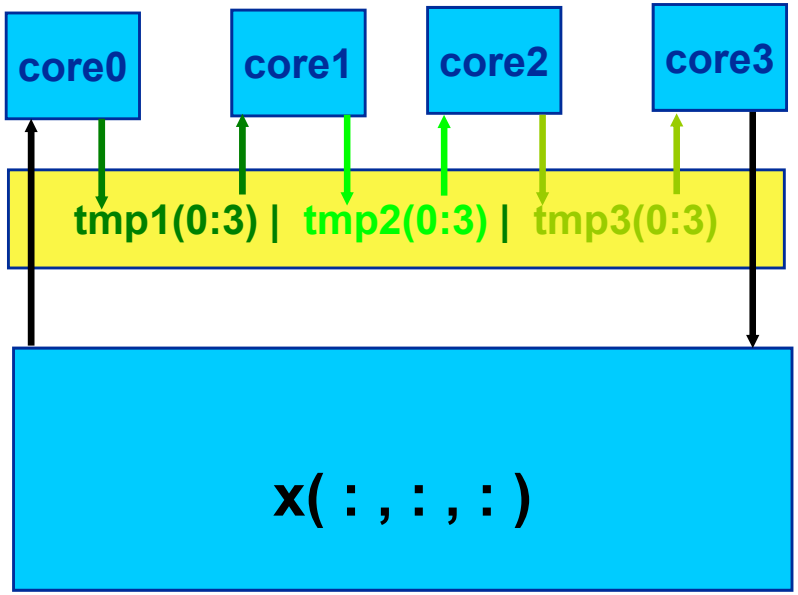
Thread 0: $x(:,:,k-1:k+1)_t$ → *tmp1(mod(k,4))*

Thread 1: *tmp1(mod(k-3,4):mod(k-1,4))* → *tmp2(mod(k-2,4))*

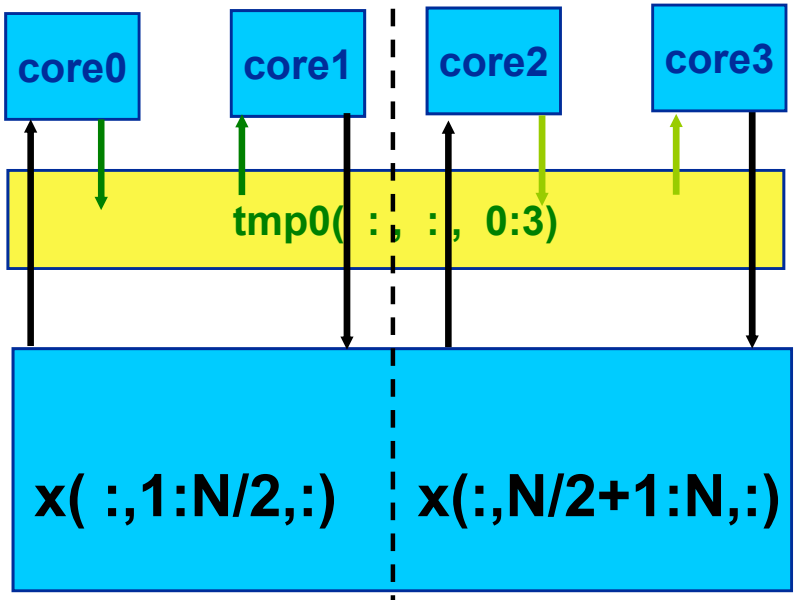
Thread 2: *tmp2(mod(k-5,4:mod(k-3,4))* → *tmp3(mod(k-4,4))*

Thread 3: *tmp3(mod(k-7,4):mod(k-5,4))* → $x(:,:,k-6)_{t+4}$

1 x 4 distribution

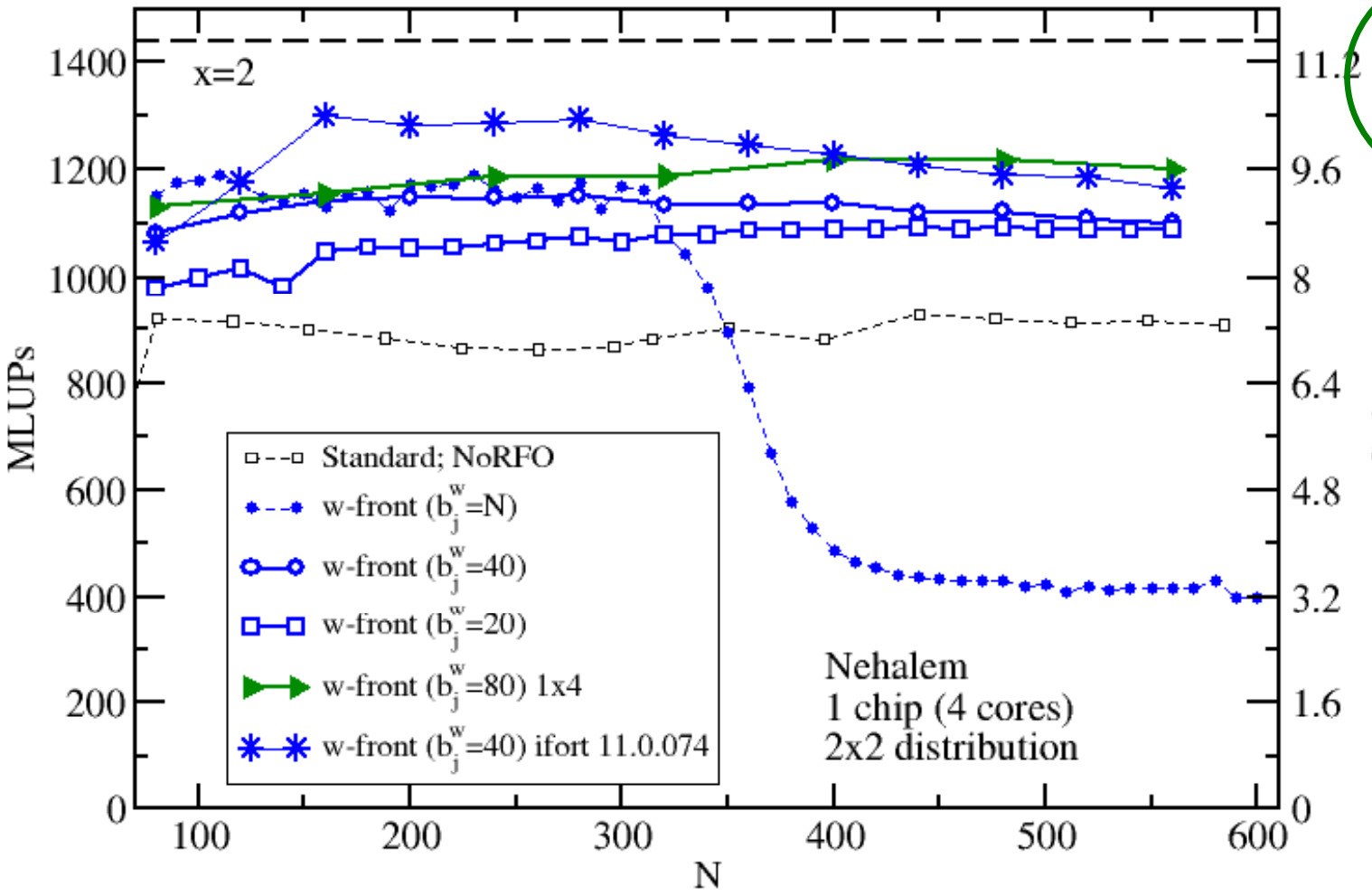


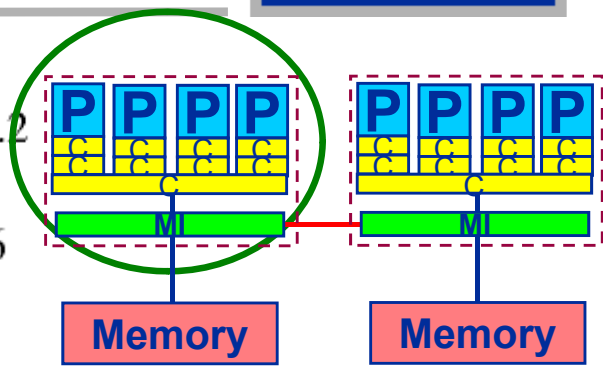
2 x 2 distribution



Jacobi solver

Wavefront parallelization: L3 group Nehalem





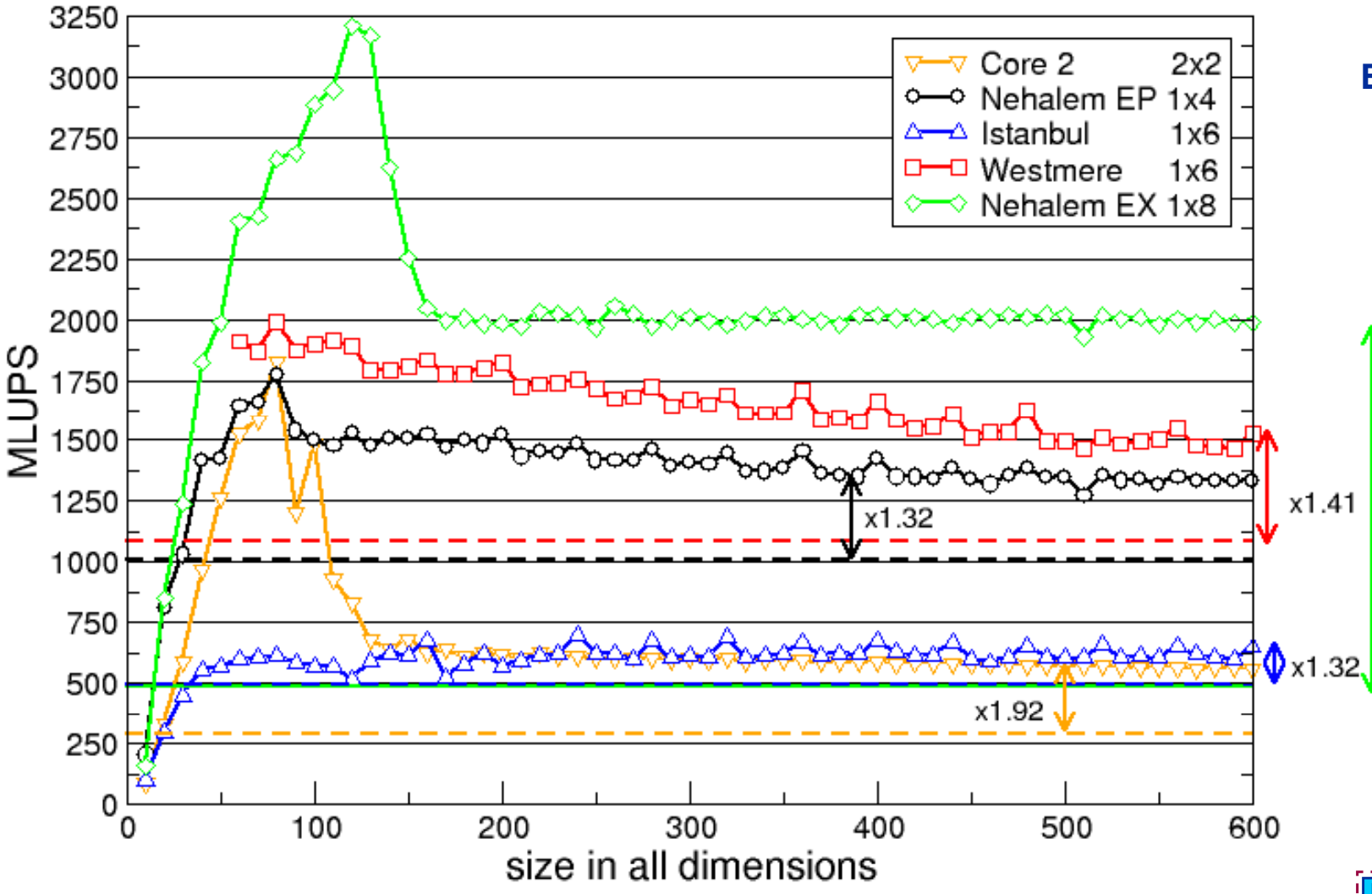
400^3 bj=40	MLUPs
1 x 2	786
2 x 2	1230
1 x 4	1254

Performance model indicates some potential gain → new compiler tested.

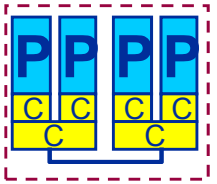
Only marginal benefit when using 4 wavefronts → A single copy stream does not achieve full bandwidth

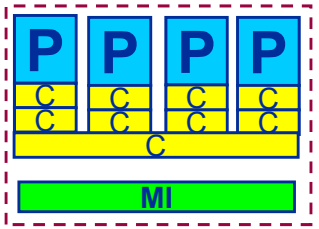
Multicore-aware parallelization

Wavefront – Jacobi on state-of-the art multicores

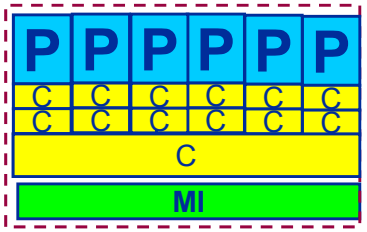


B_{olc} ~ 10

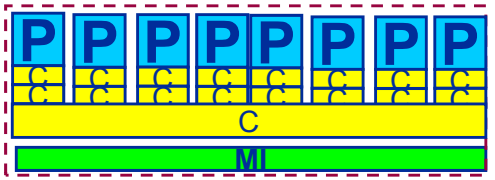




B_{olc} ~ 2-3

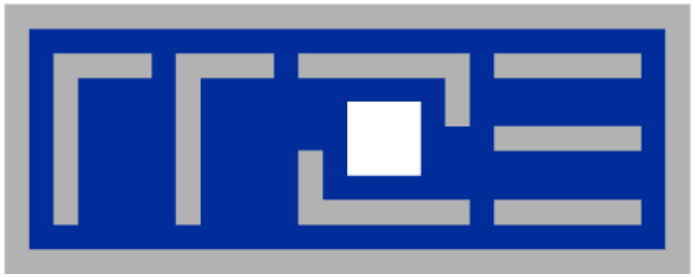


B_{olc} ~ 10



Compare against optimal baseline!

Performance gain ~ B_{olc} = L3 bandwidth / memory bandwidth



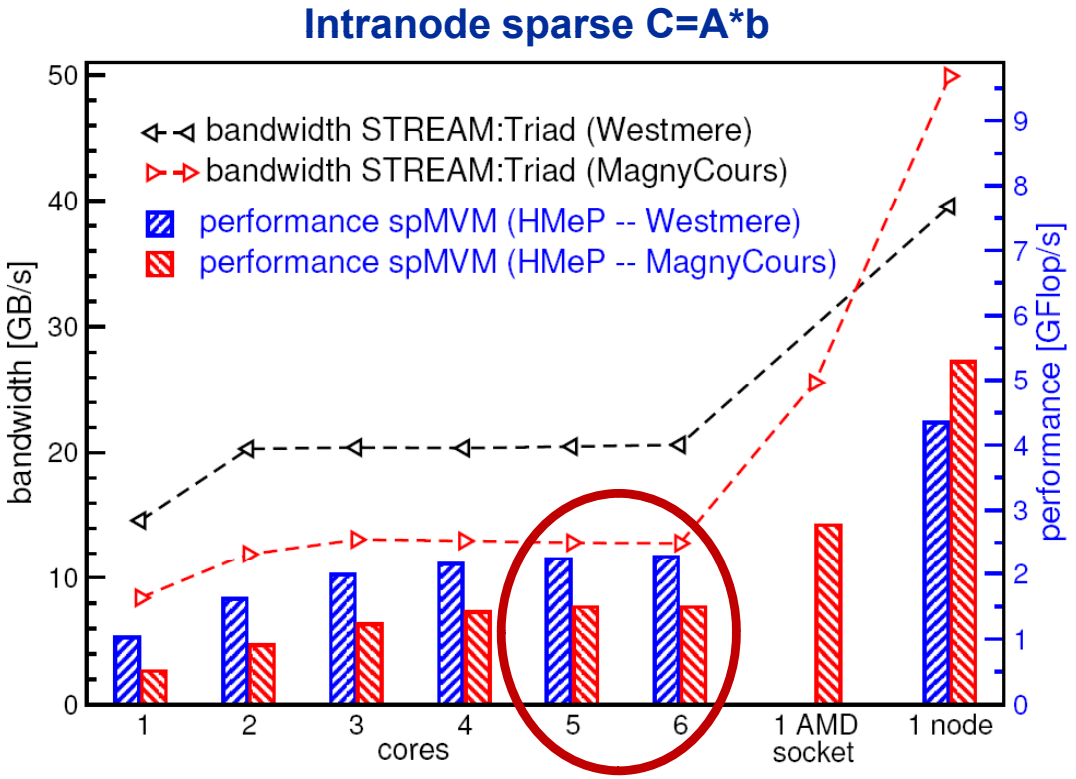
New chances with multicore hardware

Using spare cores:
Overlapping computation and communication
in hybrid sparse matrix-vector multiplication

Using cores for functional parallelism

Communication/computation overlap in sparse MVM

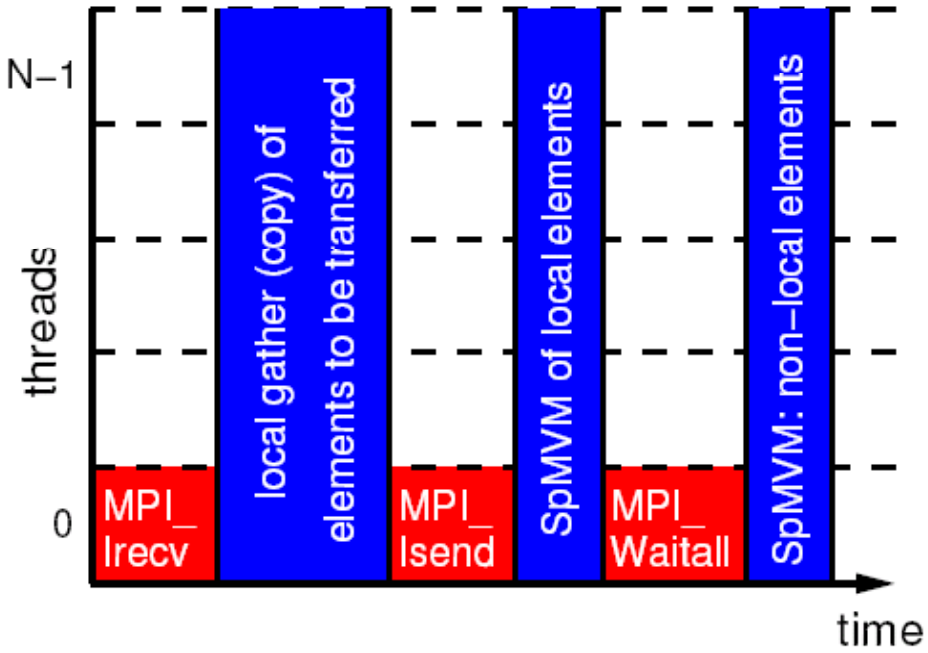
- **Sparse MVM is the dominant operation in many algorithms**
  - Sparse eigenvalue solvers
  - Sparse linear systems solvers
- **Data storage format is crucial for performance properties**
  - Most useful general format: Compressed Row Storage (CRS)
  - SpMVM is easily parallelizable in shared and distributed memory
- **For large problems, spMVM is inevitably memory-bound**
  - Intra-LD saturation effect on modern multicores
- **MPI-parallel spMVM is often communication-bound**
- **Can surplus cores be put to good use?**



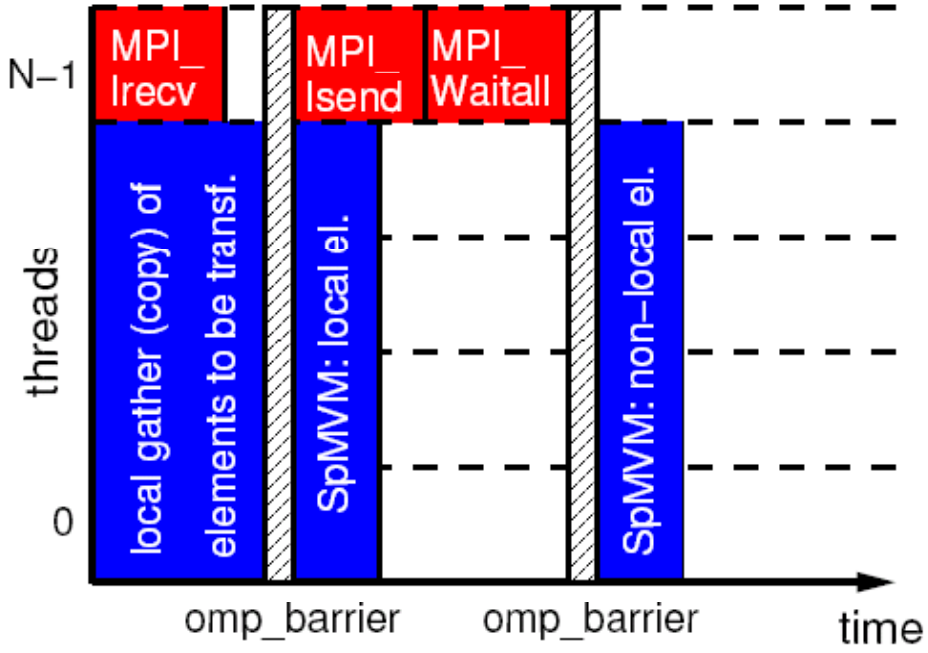
Using cores for functional parallelism

Communication/computation overlap in sparse MVM

- **Naïve vs. explicit communication vs. computation overlap**



"Vector mode with naïve overlap"



"Task mode"

- **Naïve overlap using nonblocking MPI does not work with most implementations**
- **MPI progress is limited to phases where MPI is executing**
- **Performance is similar to baseline with blocking MPI, but…**

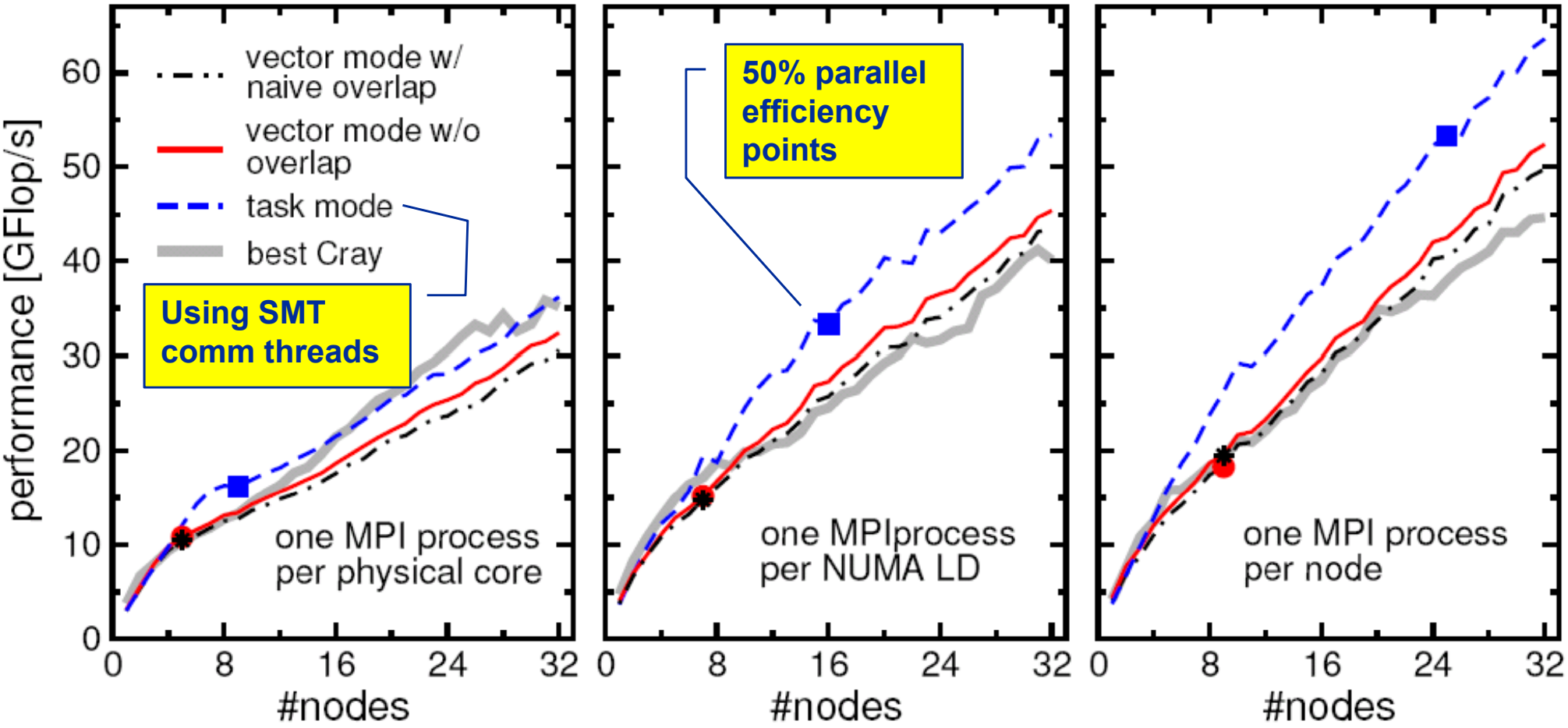
- **Explicit overlap sacrifices one thread (core) for communication**
- **Local MVM and nonlocal communication are explicitly asynchronous**
- **Many variations possible OpenMP tasking, manual work distribution,…**

Both variants need to write twice to the result vector!

Using cores for functional parallelism

Communication/computation overlap in sparse MVM

- **Performance results (strong scaling) on Westmere-based QDR IB cluster (vs. Cray XE6)**
  - HMeP matrix (Holstein-Hubbard Model, $9.2 \cdot 10^7$ nonzeros)

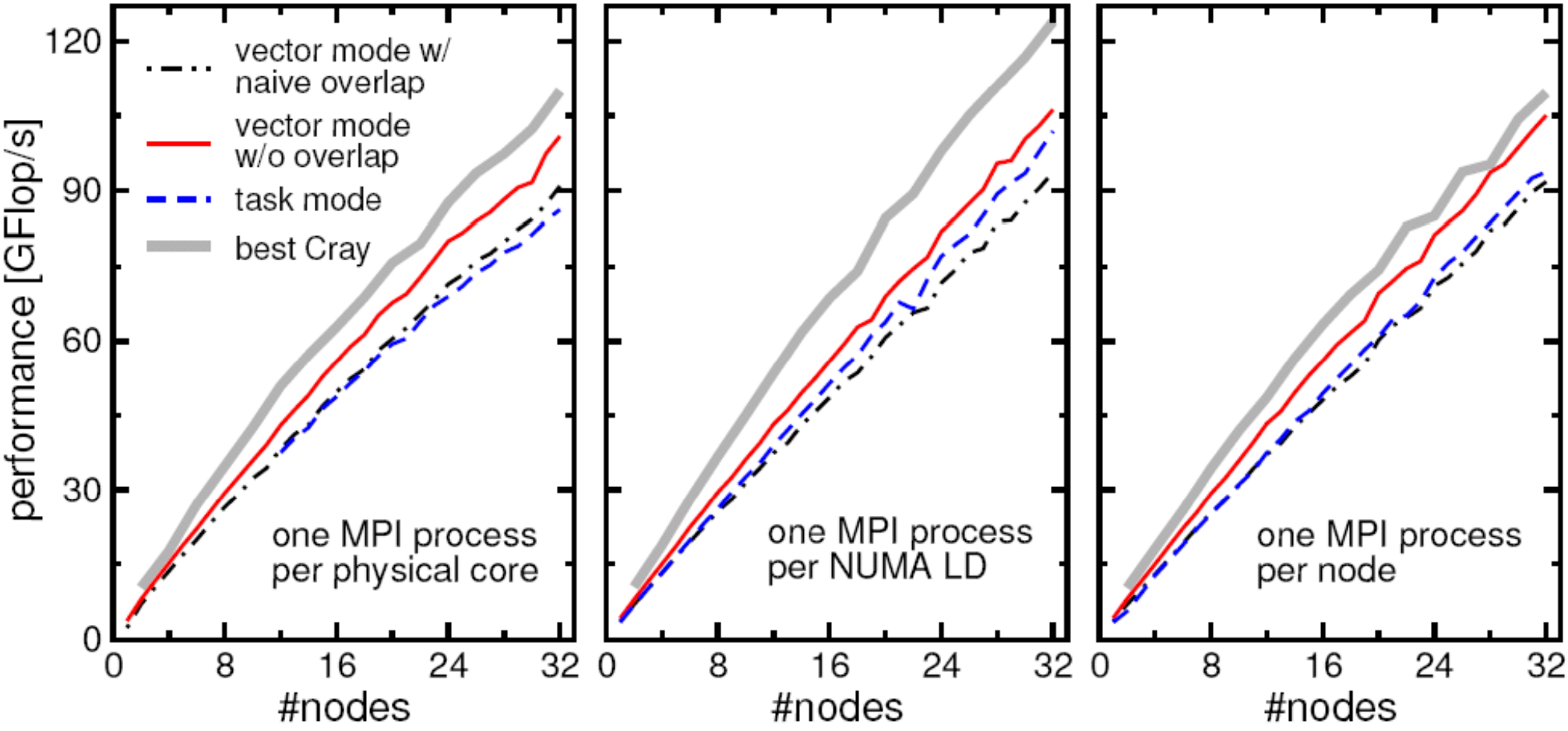


G. Schubert, G. Hager, H. Fehske and G. Wellein: *Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming*. Submitted. arXiv:1101.0091

Using cores for functional parallelism

Communication/computation overlap in sparse MVM

- **Performance results (strong scaling) on Westmere-based QDR IB cluster (vs. Cray XE6)**
  - sAMG matrix (Poisson problem on complex geometry, $1.6 \cdot 10^8$ nonzeros)



  - If communication is not the problem, overlap cannot pay off

Section summary: What to take home

- **Shared caches are *the* interesting new feature on current multicore chips**
  - Shared caches provide opportunities for fast synchronization (see sections on OpenMP and intra-node MPI performance)
  - Parallel software should leverage shared caches for performance
  - One approach: Shared cache reuse by WFP
- **WFP technique can easily be extended to many regular stencil based iterative methods, e.g.**
  - Gauß-Seidel (→ done)
  - Lattice-Boltzmann flow solvers (→ work in progress)
  - Multigrid-smoother (→ work in progress)
- **Surplus cores on multicore chips can be used for various purposes if they don't pay off for pure computation**
  - Explicit communication/computation overlap (example: sparse MVM) as an example of functional decomposition

Tutorial outline

- Introduction
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- Multicore performance tools
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- Impact of processor/node topology on program performance
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- New chances with multicore hardware
  - Wavefront parallelization of stencil codes
  - Explicit comm/calc overlap in sparse MVM
- **Summary**
- **Appendix**

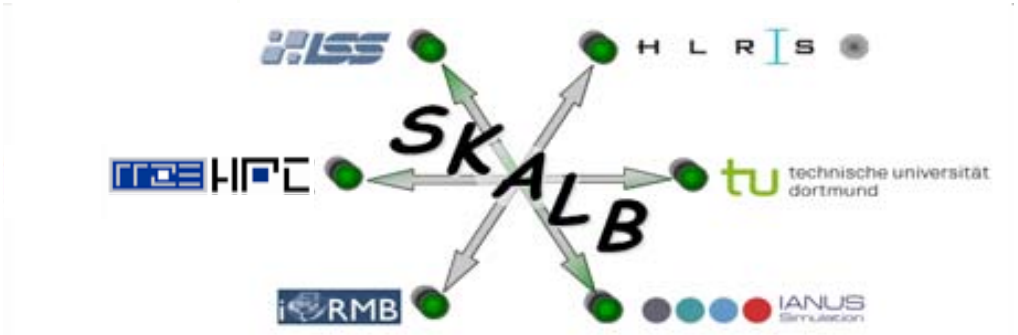
Summary & Conclusions

- **Multicore/multisocket topology needs to be considered:**
  - OpenMP performance
  - MPI communication parameters
  - Shared resources
- **Be aware of the architectural requirements of your code**
  - Bandwidth vs. compute
  - Synchronization
  - Communication
- **Use appropriate tools**
  - Node topology: likwid-pin, hwloc
  - Affinity enforcement: likwid-pin
  - Simple profiling: likwid-perfCtr
- **Try to leverage the new architectural feature and the abundant transistors of modern multicore chips**
  - Shared caches
  - Unused cores

Thank you

Jan Treibig
Markus Wittmann
Gerald Schubert
Michael Meier
Thomas Zeiser
Johannes Habich
Holger Stengel

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung
Grant # 01IH08003A (project SKALB)



Project OMI4PAPPS

Appendix: References

Books:

- G. Hager and G. Wellein: Introduction to High Performance Computing for Scientists and Engineers. CRC Computational Science Series, 2010. ISBN 978-1439811924
- R. Chapman, G. Jost and R. van der Pas: Using OpenMP. MIT Press, 2007. ISBN 978-0262533027
- S. Akhter: Multicore Programming: Increasing Performance Through Software Multi-threading. Intel Press, 2006. ISBN 978-0976483243

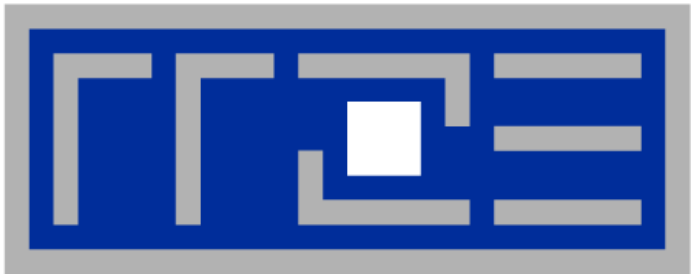
Papers:

- J. Treibig, G. Hager and G. Wellein: Multicore architectures: Complexities of performance prediction and the impact of cache topology. To appear. http://arxiv.org/abs/0910.4865
- 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. DOI:10.1109/COMPSAC.2009.82
- 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.

References

Papers continued:

- M. Wittmann, G. Hager, J. Treibig and G. Wellein: Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. Parallel Processing Letters **20** (4), 359-376 (2010). DOI: 10.1142/S0129626410000296. http://arxiv.org/abs/1006.3148
- J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proc. PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010. http://arxiv.org/abs/1004.4431
- G. Schubert, G. Hager and H. Fehske: Performance limitations for sparse matrix-vector multiplications on current multicore environments. To appear. http://arxiv.org/abs/0910.4836
- G. Hager, G. Jost, and R. Rabenseifner: Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In: Proceedings of the Cray Users Group Conference 2009 (CUG 2009), Atlanta, GA, USA, May 4-7, 2009.
- G. Schubert, G. Hager, H. Fehske and G. Wellein: Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming. Submitted. http://arxiv.org/abs/1101.0091



Advanced OpenMP: Pipeline parallel processing → Eliminating recursion

Parallelizing a 3D Gauss-Seidel solver

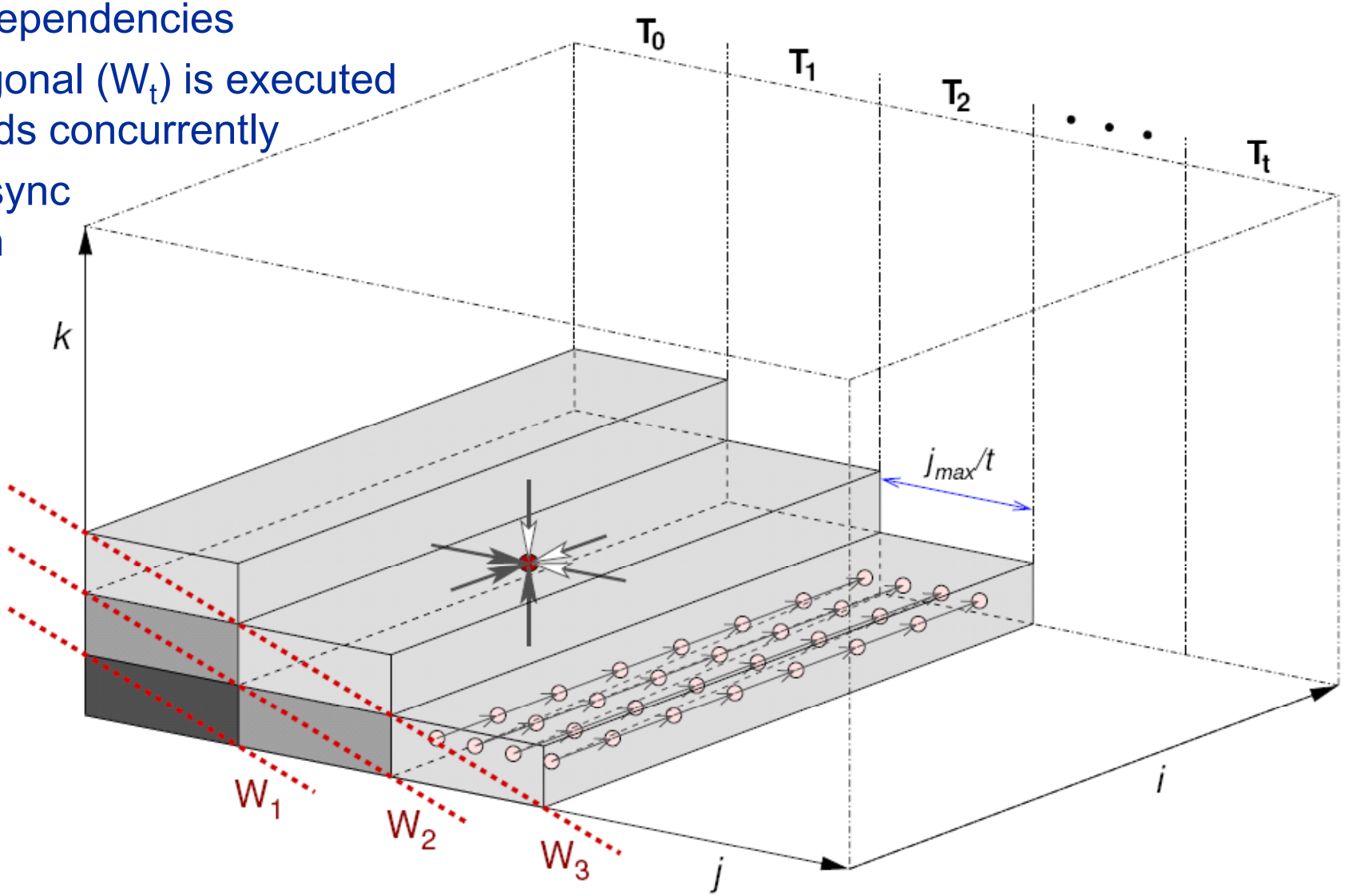
The Gauss-Seidel algorithm in 3D

```
double precision, parameter :: osth=1/6.d0
do it=1,itmax    ! number of iterations (sweeps)
  ! not parallelizable right away
  do k=1,kmax
    do j=1,jmax
      do i=1,imax
        phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
                     + phi(i,j-1,k) + phi(i,j+1,k)
                     + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
      enddo
    enddo
  enddo
enddo
```

- **Not parallelizable by compiler or simple directives because of loop-carried dependency**
- **Is it possible to eliminate the dependency?**

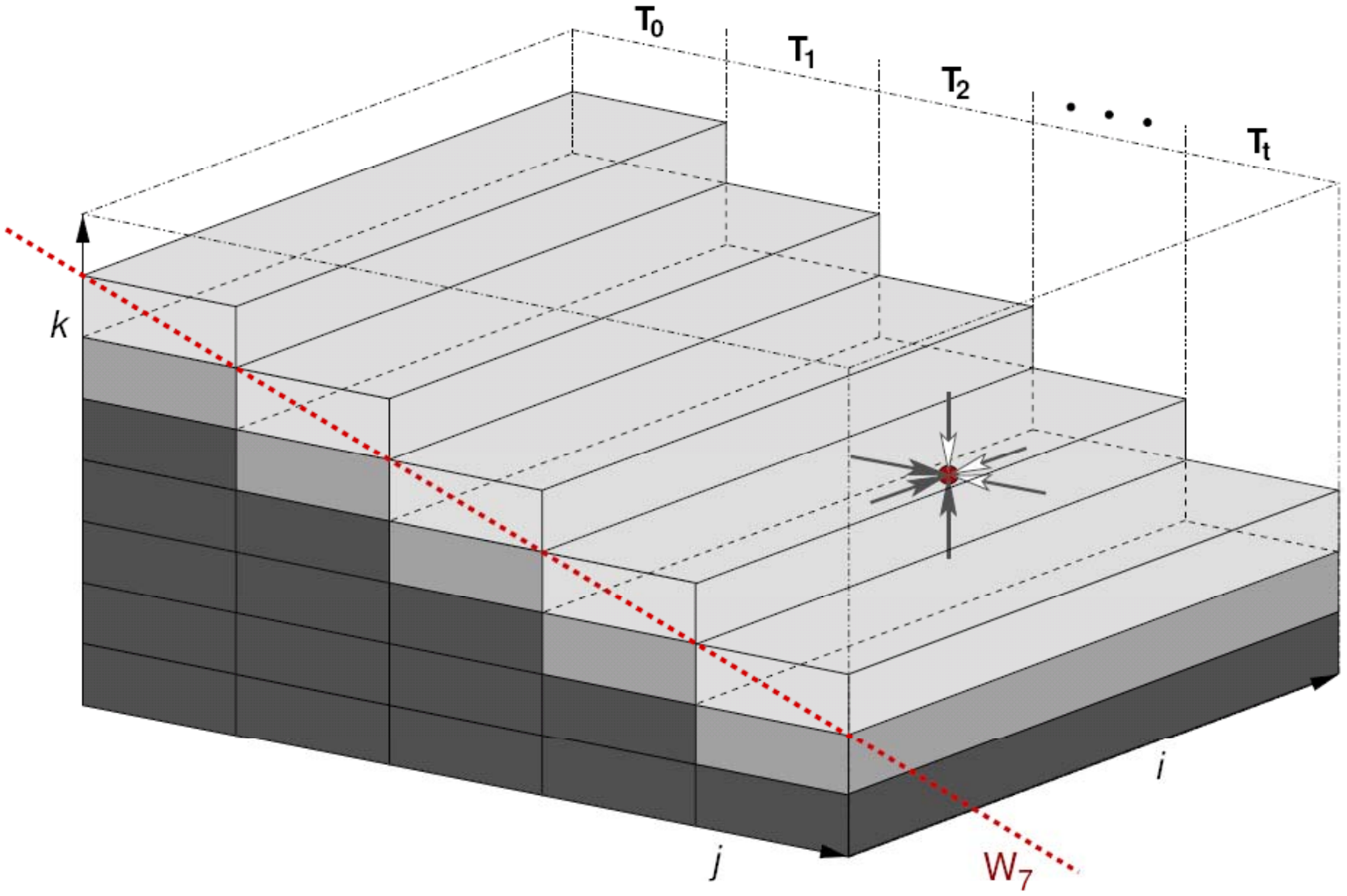
3D Gauss-Seidel parallelized

- **Pipeline parallel principle: Wind-up phase**
  - Parallelize middle j-loop and shift thread execution in k-direction to account for data dependencies
  - Each diagonal (W_t) is executed by t threads concurrently
  - Threads sync after each k-update



3D Gauss-Seidel parallelized

- **Full pipeline: All threads execute**



3D Gauss-Seidel parallelized: The code

```
!$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
  threadID=OMP_GET_THREAD_NUM()
!$OMP SINGLE
  numThreads=OMP_GET_NUM_THREADS()
!$OMP END SINGLE
  jStart=jmax/numThreads*threadID
  jEnd=jStart+jmax/numThreads ! jmax is amultiple of numThreads
  do l=1,kmax+numThreads-1
    k=l-threadID
    if((k.ge.1).and.(k.le.kmax)) then
      do j=jStart,jEnd          ! this is the actual parallel loop
        do i=1,iMax
          phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
                       + phi(i,j-1,k) + phi(i,j+1,k)
                       + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
        enddo
      enddo
    endif
!$OMP BARRIER
  enddo
!$OMP END PARALLEL
```

Parallel 3D Gauß-Seidel

- **Gauß-Seidel can also be parallelized using a red-black scheme**

- **But data dependency is representative for several linear (sparse) solvers Ax=b arising from regular discretization, e.g. Stone's Strong Implicit (SIP) solver based on incomplete A ~ LU factorization**
  - Still used in many CFD FV codes (→ RRZE report)
  - L & U: Each contains 3 nonzero off-diagonals only!
  - Solving Lx=b or Ux=c has loop carried data dependencies similar to GS → PPP

Presenter Biography

- Georg Hager holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems.

  See his blog at http://blogs.fau.de/hager for current activities, publications, and talks.

Abstract

- **Tutorial: Ingredients for Good Parallel Performance on Multicore-based systems**
- **Presenter: Georg Hager**
- **Authors: Georg Hager, Gerhard Wellein**

- **ABSTRACT:**

This tutorial covers program optimization techniques for multi-core processors and the systems they are used in. It concentrates on the dominating parallel programming paradigms, MPI and OpenMP. We start by giving an architectural overview of multicore processors. Peculiarities like shared vs. separate caches, bandwidth bottlenecks, and ccNUMA characteristics are pointed out. We show typical performance features like synchronization overhead, intranode MPI bandwidths and latencies, ccNUMA locality, and bandwidth saturation (in cache and memory) in order to pinpoint the influence of system topology and thread affinity on the performance of typical parallel programming constructs. Multiple ways of probing system topology and establishing affinity, either by explicit coding or separate tools, are demonstrated. Finally we elaborate on programming techniques that help establish optimal parallel memory access patterns and/or cache reuse, with an emphasis on leveraging shared caches for improving performance.