Performance-oriented programming on multicore-based clusters with MPI, OpenMP, and hybrid MPI/OpenMP

Georg Hager[(a)], Gabriele Jost[(b)], Rolf Rabenseifner[(c)], Jan Treibig[(a)], and Gerhard Wellein[(a,d)]

[(a)]HPC Services, Erlangen Regional Computing Center (RRZE)
[(b)]Texas Advanced Computing Center (TACC), University of Texas, Austin
[(c)]High Performance Computing Center Stuttgart (HLRS)
[(d)]Department for Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg

ISC11 Tutorial, June 19th, 2011, Hamburg, Germany

http://blogs.fau.de/hager/tutorials/isc11/

Tutorial outline (1)

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

Tutorial outline (2)

- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
  - Practical “How-tos” for hybrid
- **Online demo: likwid tools (2)**
  - Advanced pinning
  - Making bandwidth maps
  - Using likwid-perfctr to find NUMA problems and load imbalance
  - likwid-perfctr internals
  - likwid-perfscope
- **Case studies for hybrid MPI/OpenMP**
  - Overlap for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - PIR3D – hybridization of a full scale CFD code
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

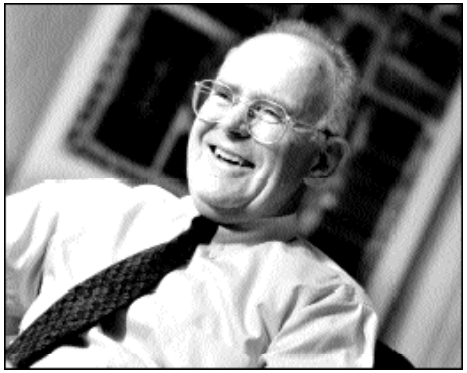
Tutorial outline

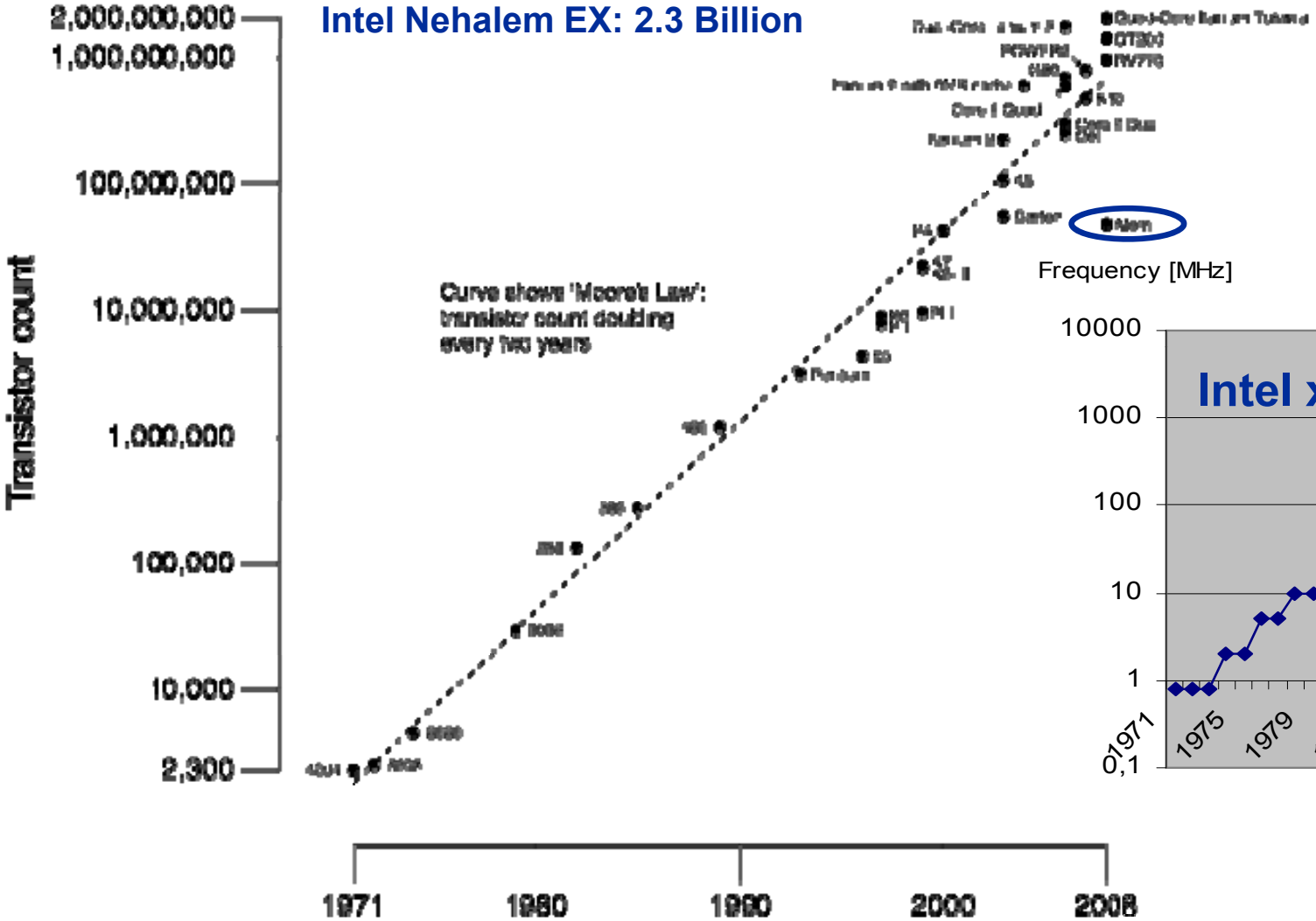
- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

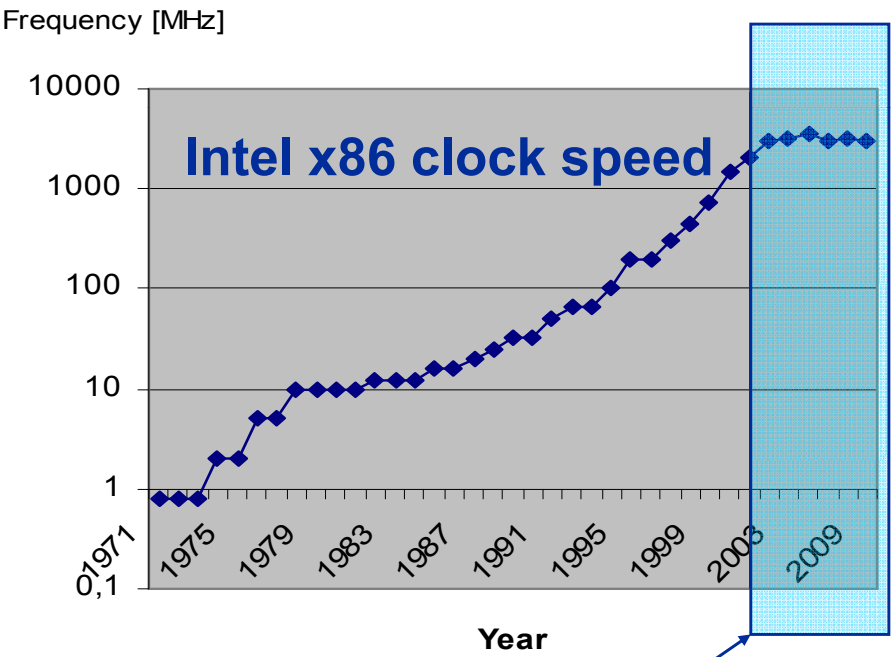
Welcome to the multi-/manycore era

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

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



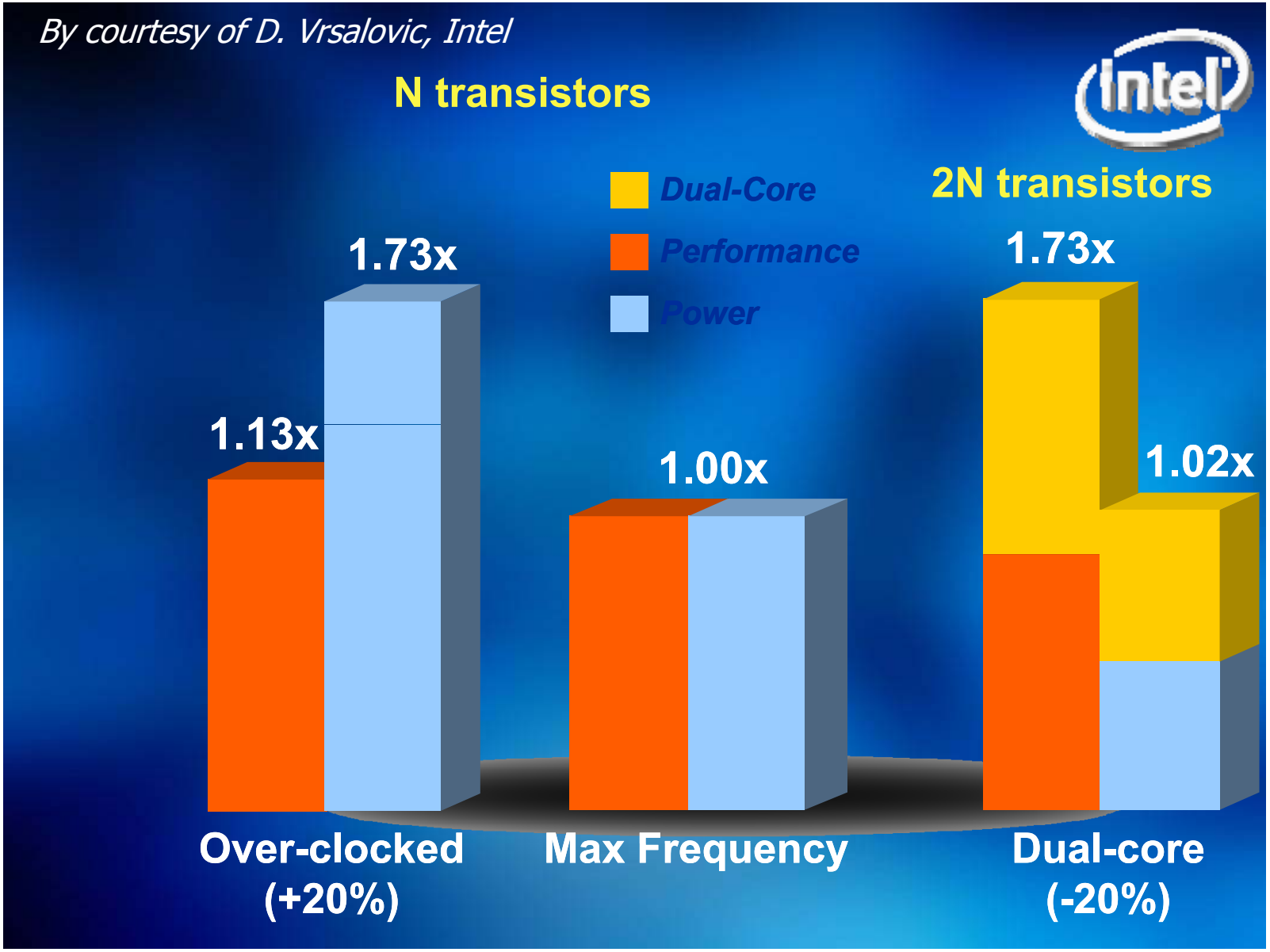




- We are living in the multicore era → Is really everyone aware of that?

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:

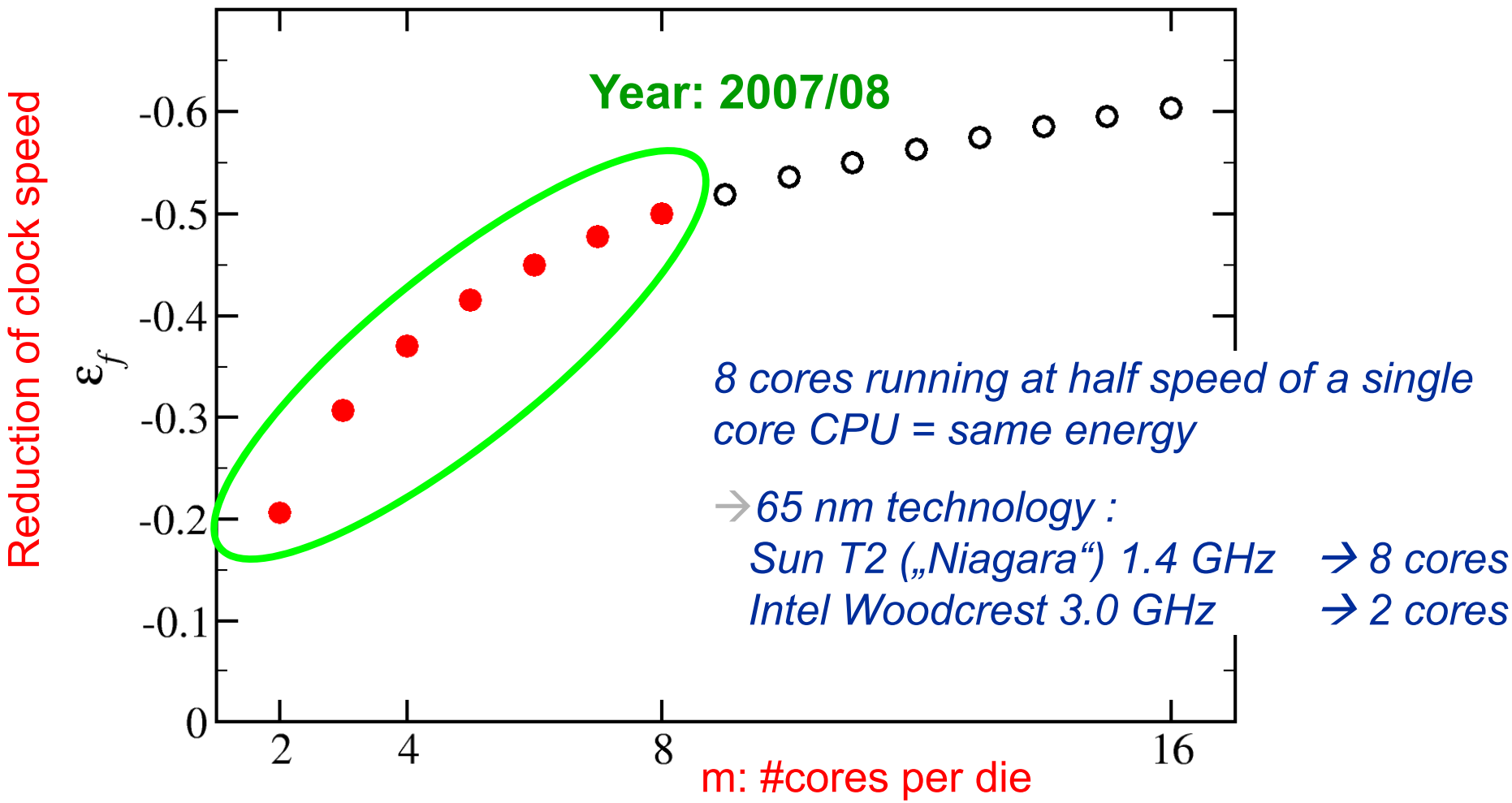
$P \sim f^3$

Welcome to the multi-/many-core era

The game is over: But Moore's law continues

SKIPPED

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



Trading single thread performance for parallelism

- Power consumption limits clock speed: $P \sim f^2$ (worst case $\sim f^3$)
- Core supply voltage approaches a lower limit: $V_C \sim 1V$
- TDP approaches economical limit: TDP ~ 80 W,…,130 W

P5 / 80586 (1993)	Pentium3 (1999)	Pentium4 (2003)	Core i7–960 (2009)
66 MHz	600 MHz	2800 MHz	3200 MHz
16 W @ V_C = 5 V	23 W @ V_C = 2 V	68 W @ V_C = 1.5 V	130 W @ V_C = 1.3
800 nm / 3 M	250 nm / 28 M	130 nm / 55 M	45 nm / 730 M
			Quad-Core

TDP /
Core supply voltage

Process technology /
Number of transistors in million

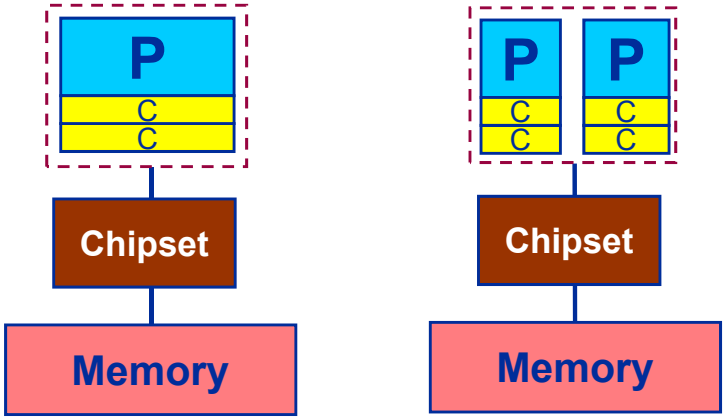
- Moore's law is still valid…
  → more cores + new on-chip functionality (PCIe, GPU)

Be prepared for more cores with less complexity and slower clock!

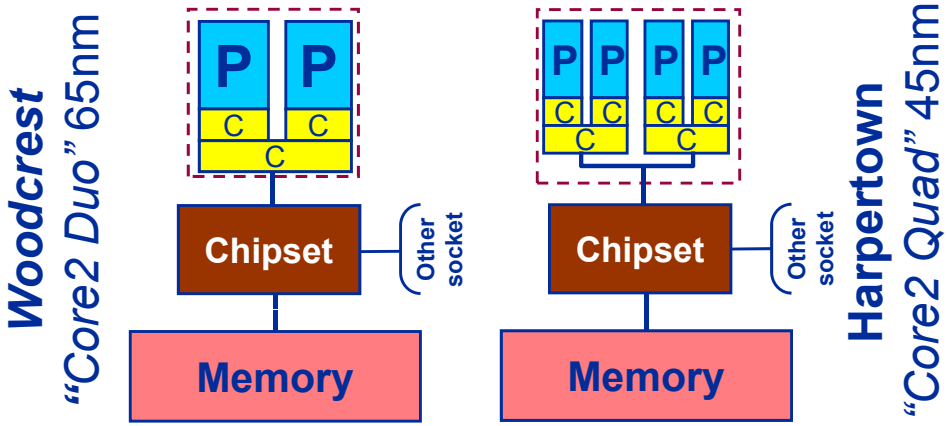
The x86 multicore evolution so far

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

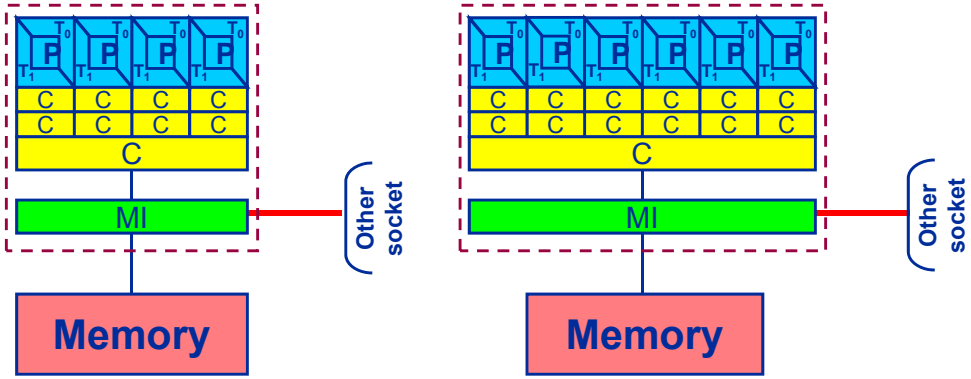
2005: "Fake" dual-core



2006: True dual-core



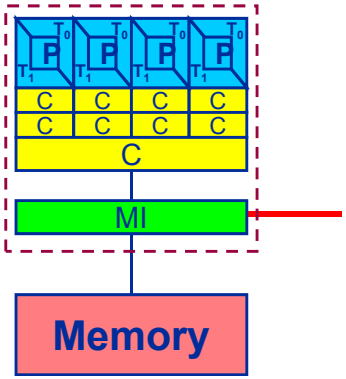
2008: Hyperthreading/SMT is back!



Nehalem EP
"Core i7"
45nm

Westmere EP
"Core i7"
32nm

2010/11: Wider SIMD units
SSE → AVX
128 Bit → 256 Bit



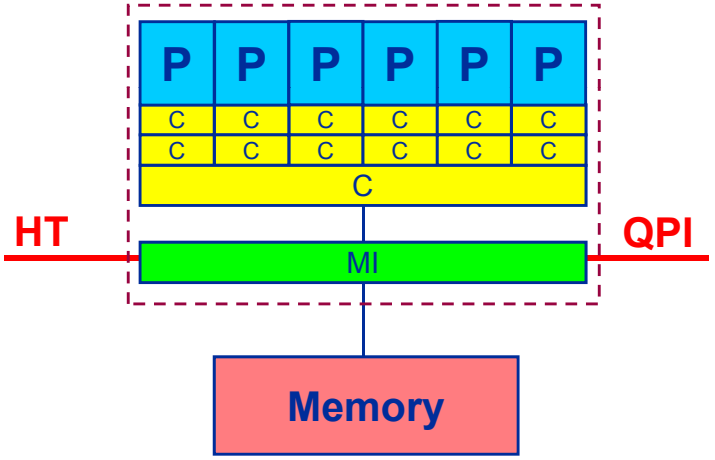
Sandy Bridge (Desktop)
"Core i7"
32nm

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!

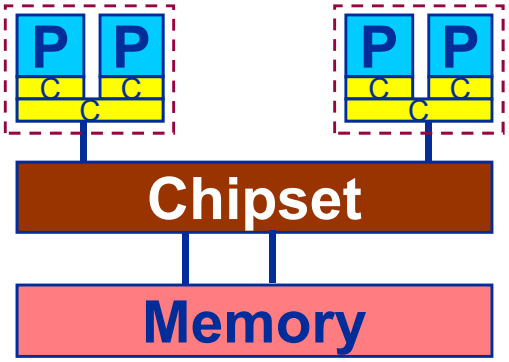
Intel Xeon Westmere
6 cores @ 2.93 GHz
L1: 32 KB
L2: 256 KB
L3: 12MB
3 X DDR3-1333 → 31.8 GB/s
2 X QPI6.4 → 12.8 GB/s/dir

From UMA to ccNUMA

Basic architecture of commodity compute cluster nodes

Dual-socket Intel "Core2" node:

Yesterday



Uniform Memory Architecture (UMA):

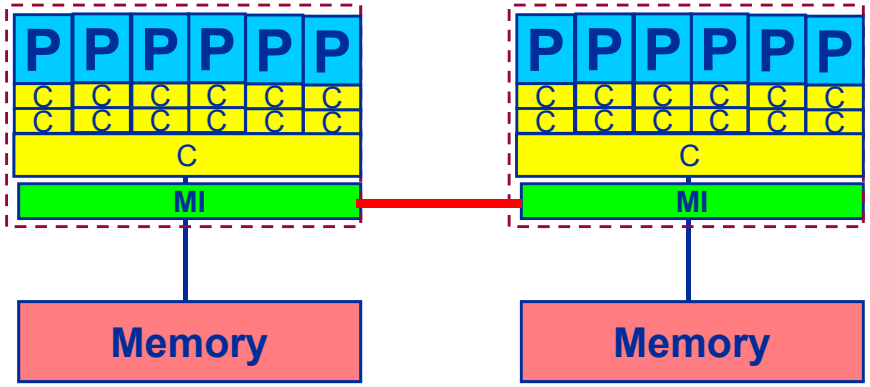
Flat memory ; symmetric MPs

But: system "anisotropy"

Shared Address Space within the node!

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

Today

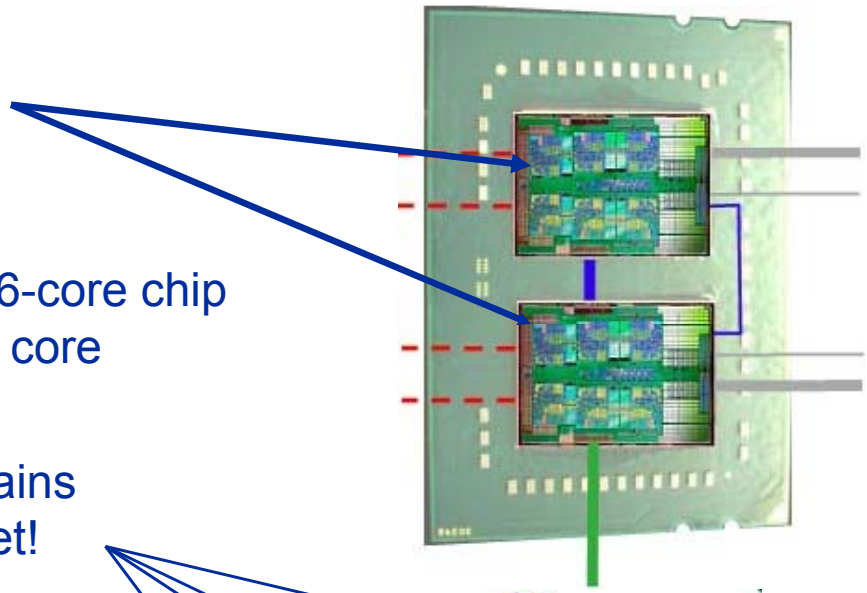


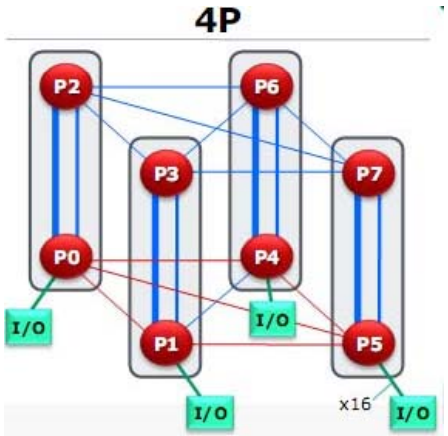
Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

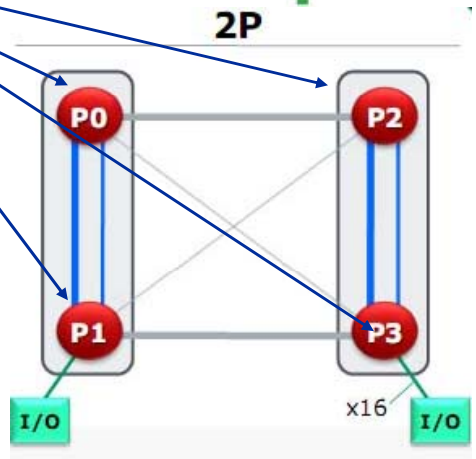
HT / QPI provide scalable bandwidth at the expense of ccNUMA architectures: *Where does my data finally end up?*

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:





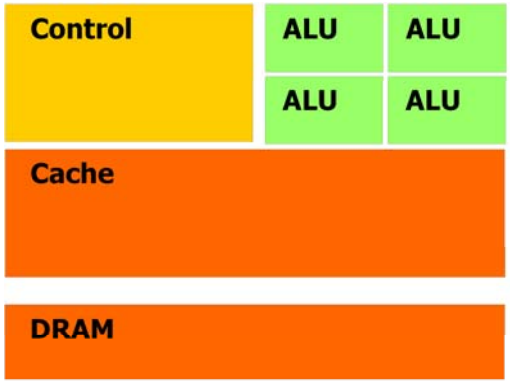


  - Network balance (QDR+2P Magny Cours) ~ 240 GF/s / 3 GB/s = 80 Bytes/Flop
    (2003: Intel Xeon DP 2.66 GHz + GBit ~ 10 GF/s / 0.12 GB/s = 80 Bytes/Flop)

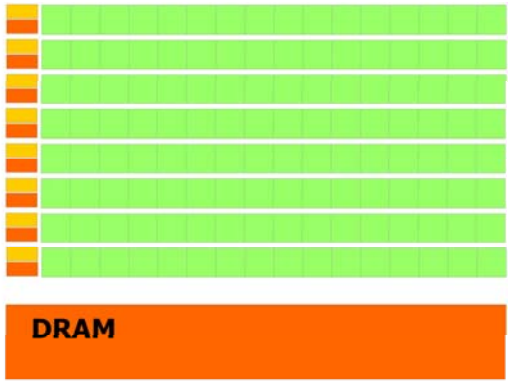
Trading single thread performance for parallelism:
GPGPUs vs. CPUs

GPU vs. CPU
light speed estimate:

1. **Compute bound: 4-5 X**
2. **Memory Bandwidth: 2-5 X**



CPU



GPU

	Intel Core i5 – 2500 (“Sandy Bridge”)	Intel X5650 DP node (“Westmere”)	NVIDIA C2070 (“Fermi”)
Cores@Clock	4 @ 3.3 GHz	2 x 6 @ 2.66 GHz	448 @ 1.1 GHz
Performance+/core	52.8 GFlop/s	21.3 GFlop/s	2.2 GFlop/s
Threads@stream	4	12	8000 +
Total performance+	210 GFlop/s	255 GFlop/s	1,000 GFlop/s
Stream BW	17 GB/s	41 GB/s	90 GB/s (ECC=1)
Transistors / TDP	1 Billion* / 95 W	2 x (1.17 Billion / 95 W)	3 Billion / 238 W

+ Single Precision

** Includes on-chip GPU and PCI-Express*

Complete compute device

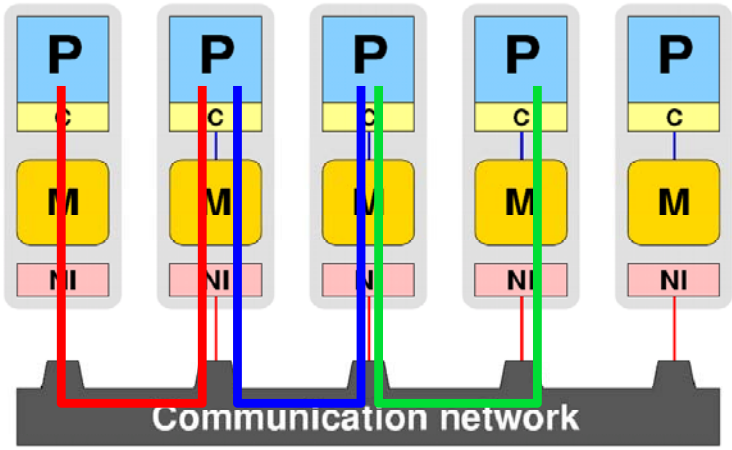
Parallel programming models
on multicore multisocket nodes

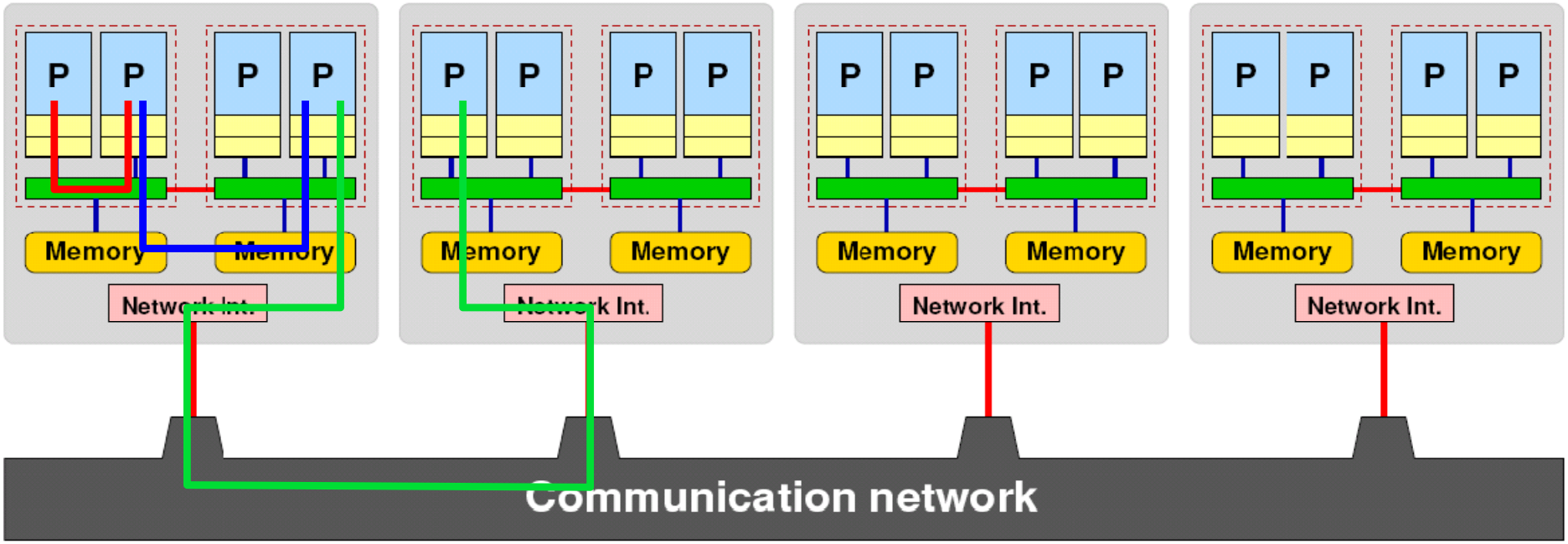
- **Shared-memory (intra-node)**
  - Good old MPI (current standard: 2.2)
  - OpenMP (current standard: 3.0)
  - POSIX threads
  - Intel Threading Building Blocks
  - Cilk++, OpenCL, StarSs,… you name it
- **Distributed-memory (inter-node)**
  - MPI (current standard: 2.2)
  - PVM (gone)
- **Hybrid**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

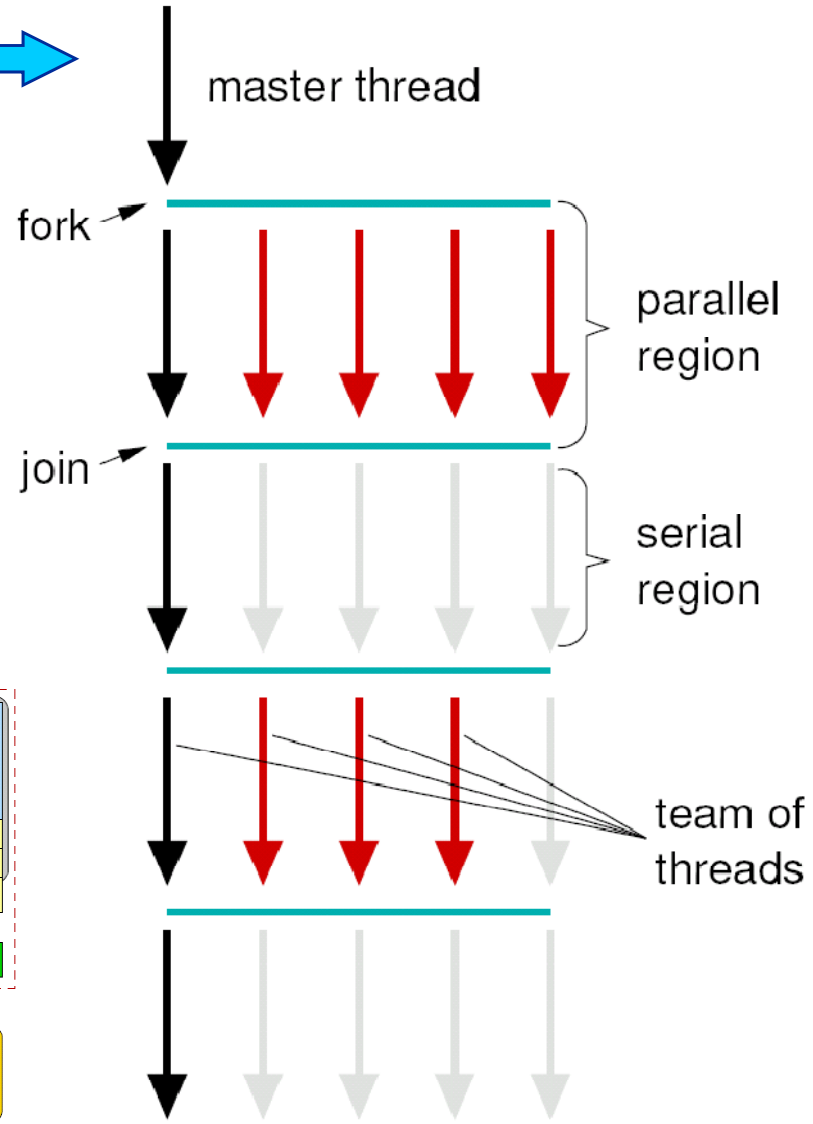
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

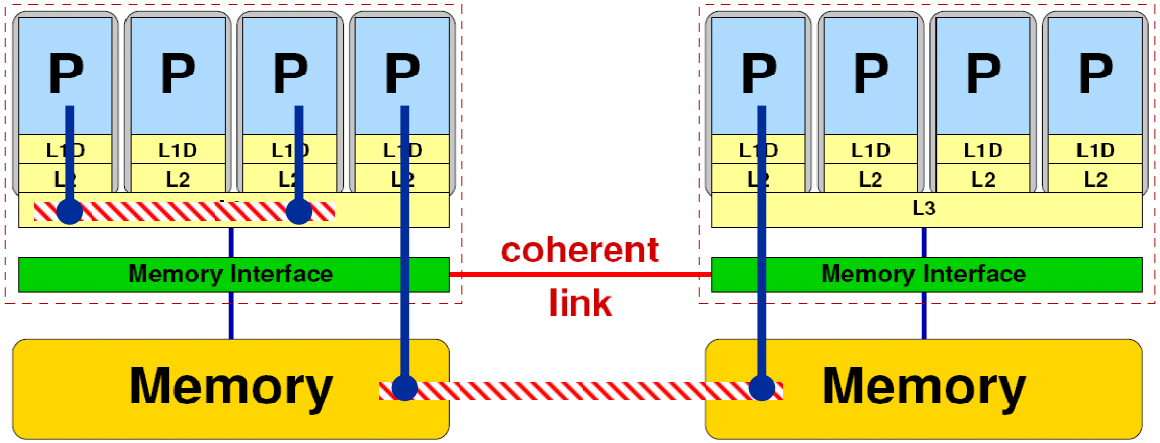




Parallel programming models:
Pure threading on the node

- **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 per socket (package)**
  - 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
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

Probing node topology

- Standard tools
- likwid-topology
- hwloc

How do we figure out the node topology?

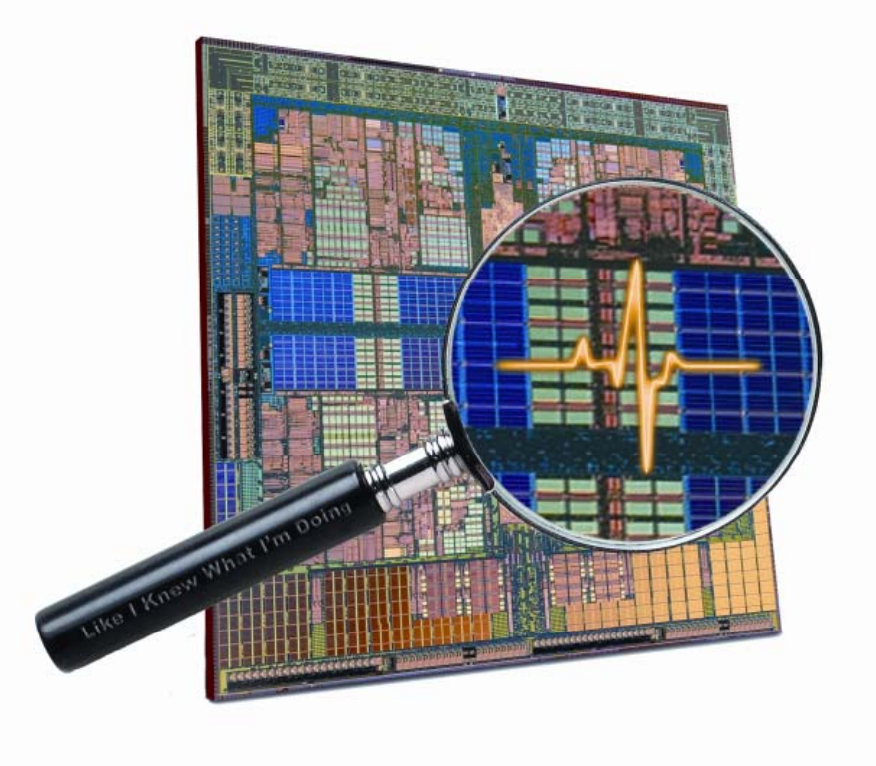
- **Topology =**
  - Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads ("logical cores") share a physical core?
- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers 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.*** **Accepted for 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 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-mpirun**: mpirun wrapper script for easy LIKWID integration
  - **likwid-bench**: Low-level bandwidth benchmark generator tool

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 beta phase)
  - AMD K10 (Quadcore and Hexacore)
  - AMD K8
  - Linux OS

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

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

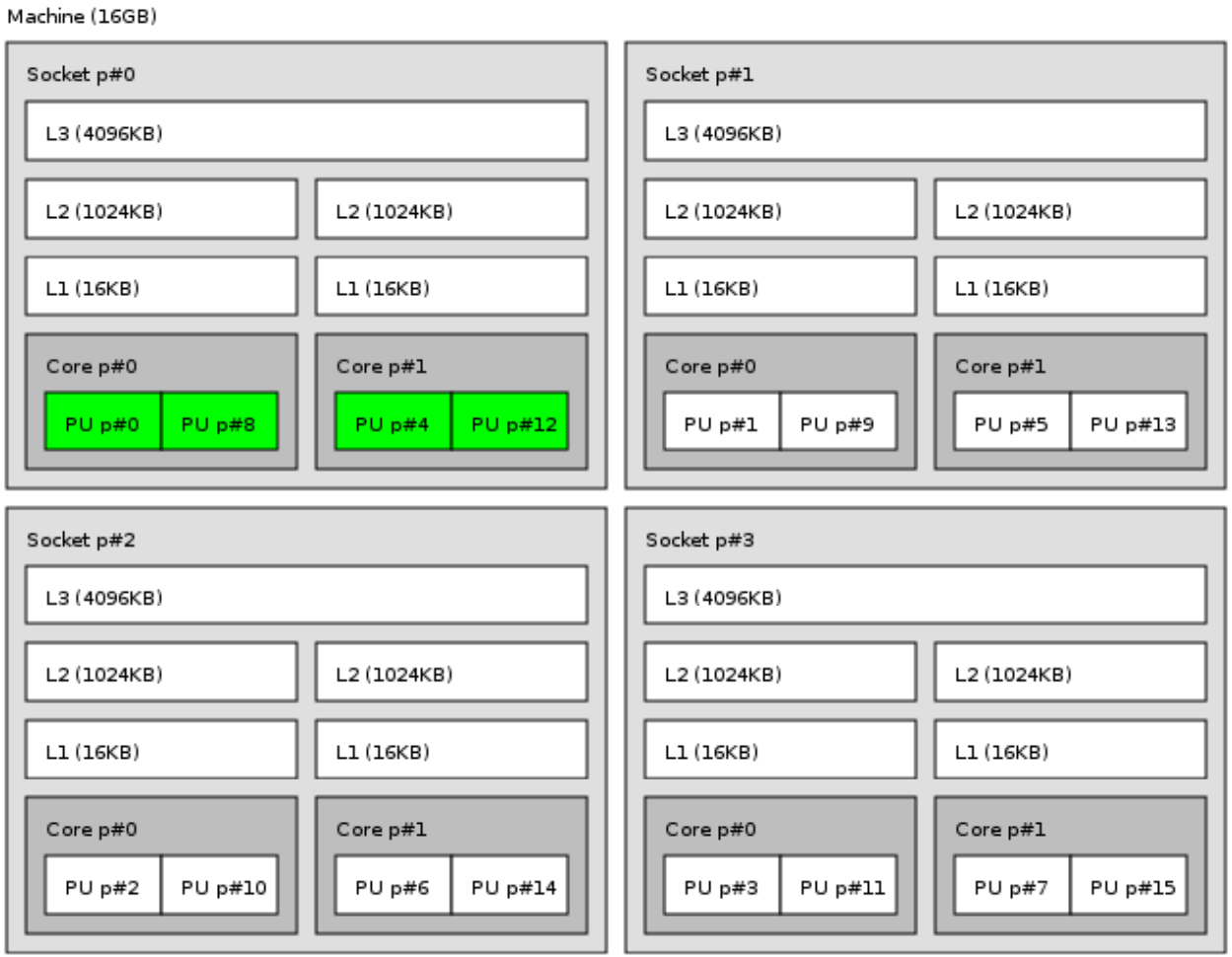
Output of likwid-topology

- … 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

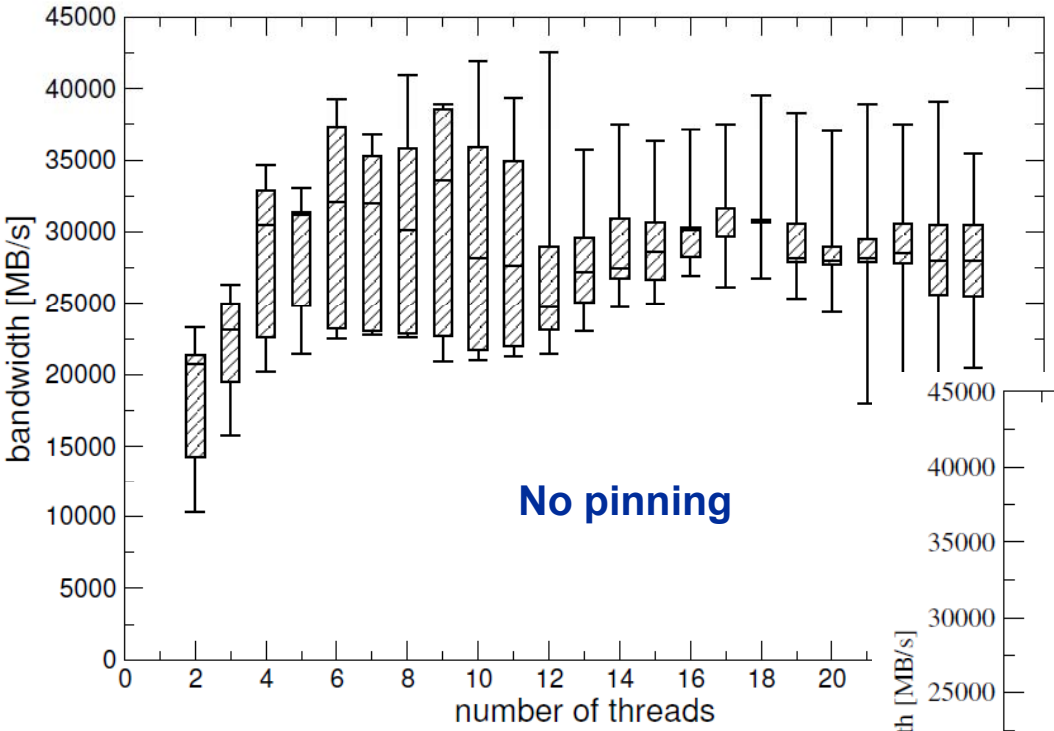
- Alternative: http://www.open-mpi.org/projects/hwloc/
- Successor to (and extension of) PLPA, part of OpenMPI development
- Comprehensive API and command line tool to extract topology info
- Supports several OSs and CPU types
- Pinning API available

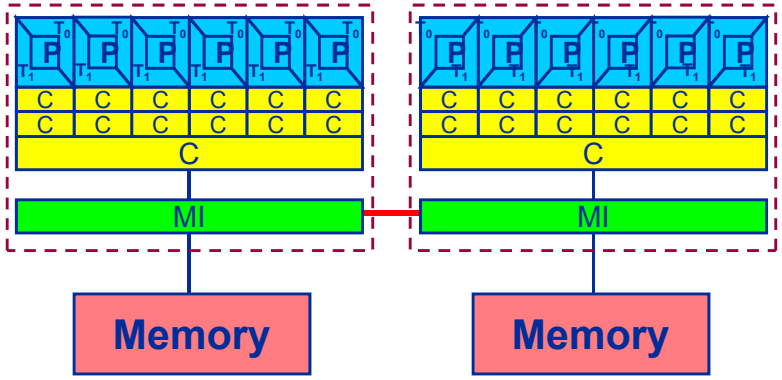


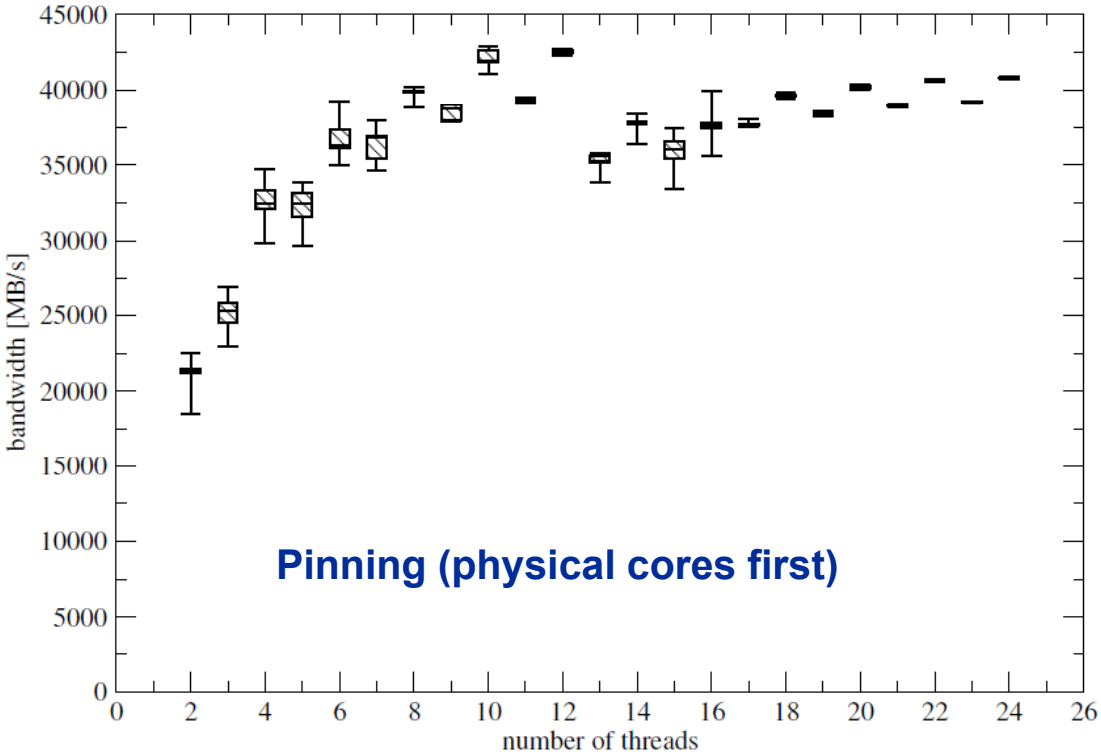
Enforcing thread/process-core affinity under the Linux OS

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

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**

Generic thread/process-core affinity under Linux

Overview

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

- `taskset` 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**

More 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[illegible]trolled affinity: Using P[illegible]nder Linux!

SKIPPED

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

SKIPPED

- **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”

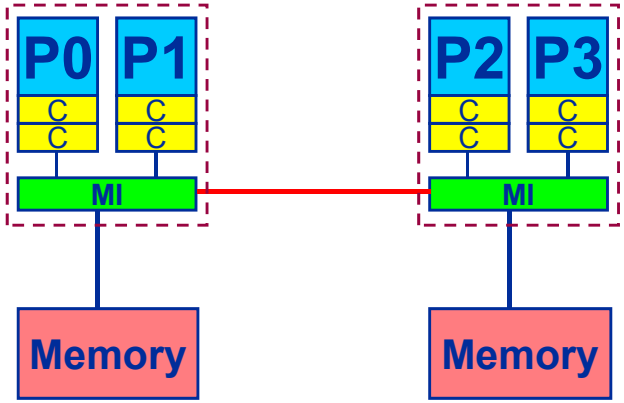
Care about correct core numbering! 0…N-1 is not always contiguous! If required, reorder by a map:
`cpu = map[cpu];`

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

Process/Thread Binding With PLPA

SKIPPED

- **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`

- **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)**

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

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

- Across all cores in the node:

```
OMP_NUM_THREADS=8 likwid-pin -c N:0-7 ./a.out
```

- Across the cores in each socket and across sockets in each node:

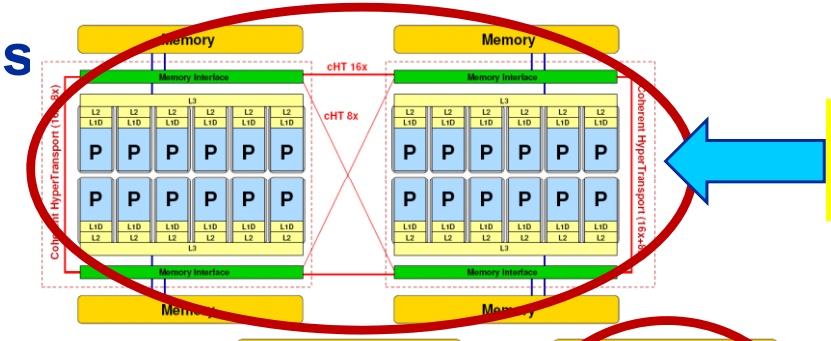
```
OMP_NUM_THREADS=8 likwid-pin -c S0:0-3@S1:0-3 ./a.out
```

Likwid-pin

Using logical core numbering

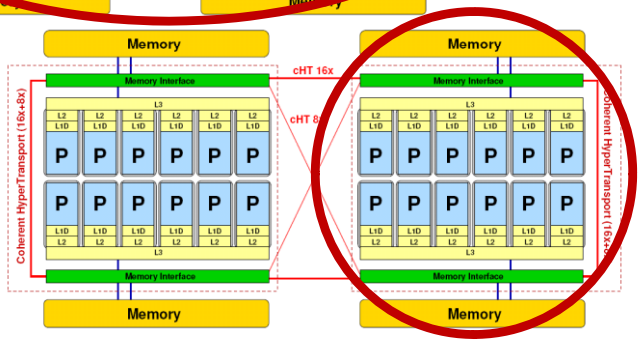
- **Possible unit prefixes**

N **node**

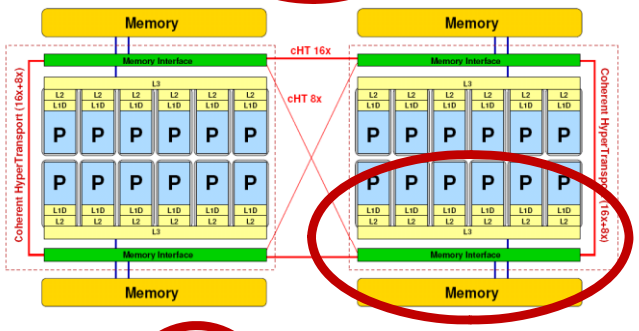


Default if –c is not specified!

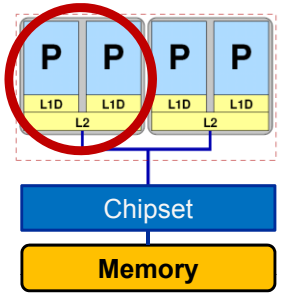
S **socket**



M **NUMA domain**



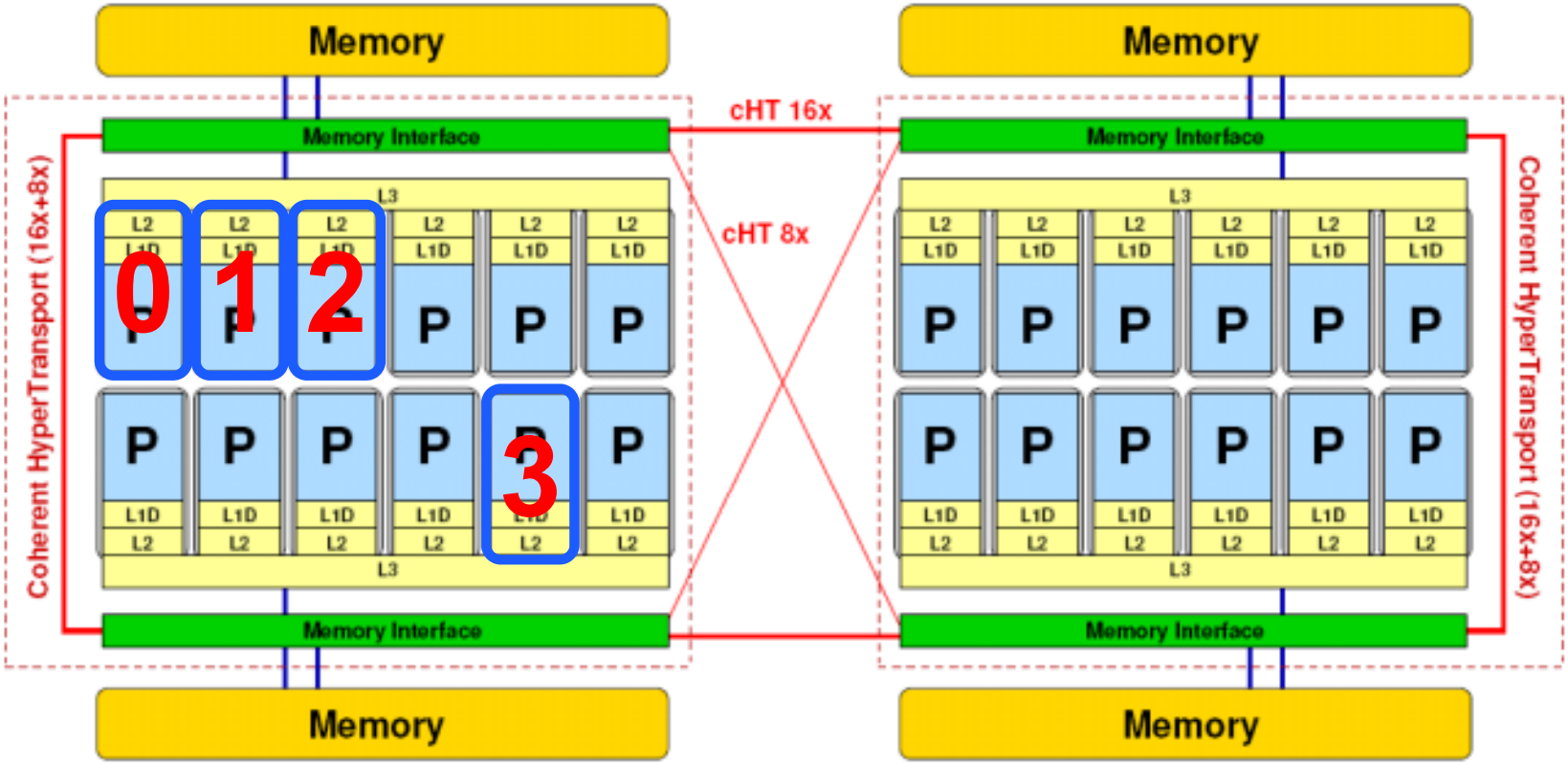
C **outer level cache group**



Likwid-pin

Using logical core numbering

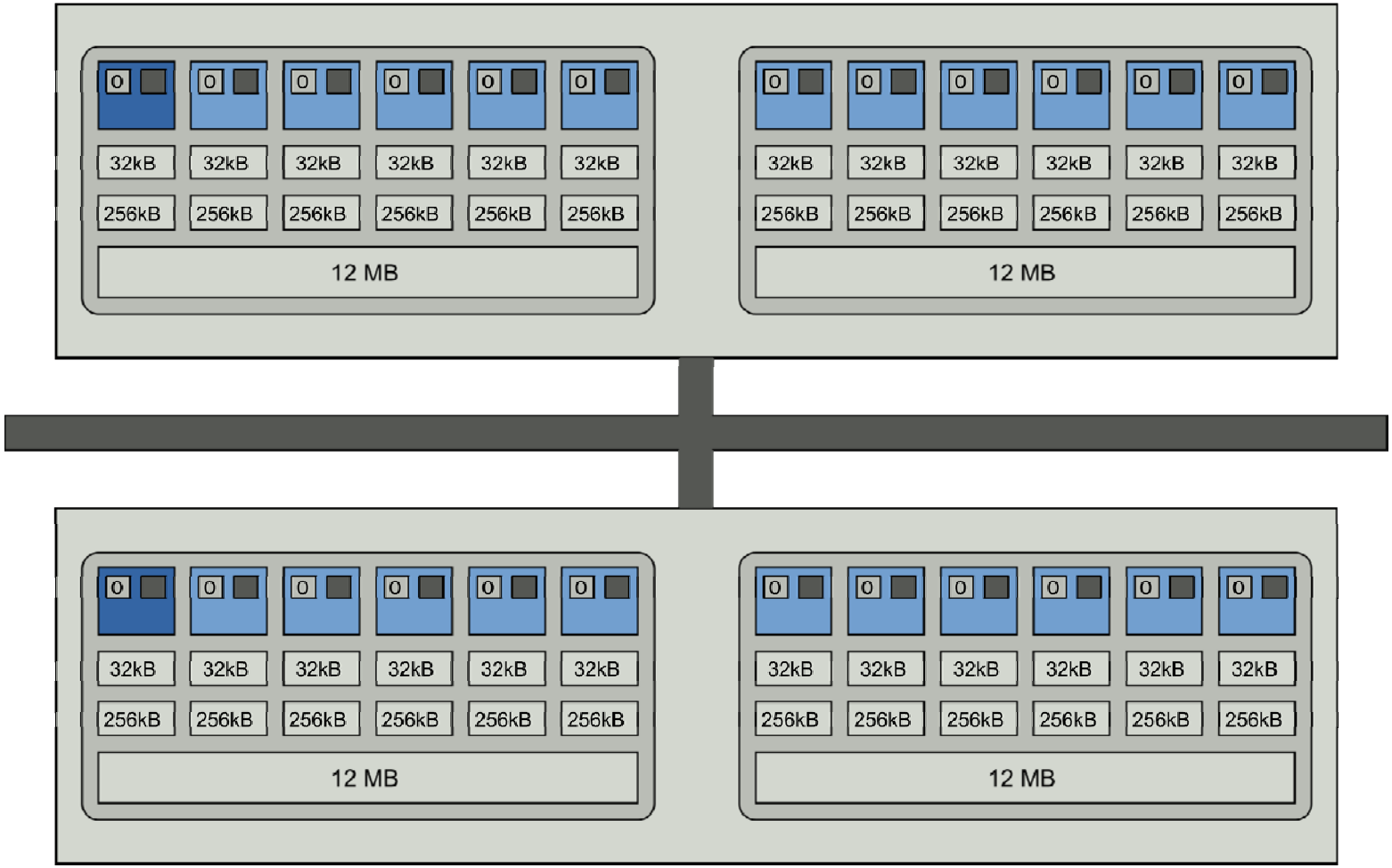
- **… and: Logical numbering inside a pre-existing cpuset:**



- `OMP_NUM_THREADS=4 likwid-pin -c L:0-3 ./a.out`

Examples for hybrid pinning with likwid-mpirun: 1 MPI process per node

```
OMP_NUM_THREADS=12 likwid-mpirun -np 2 -pin N:0-11  ./a.out
```

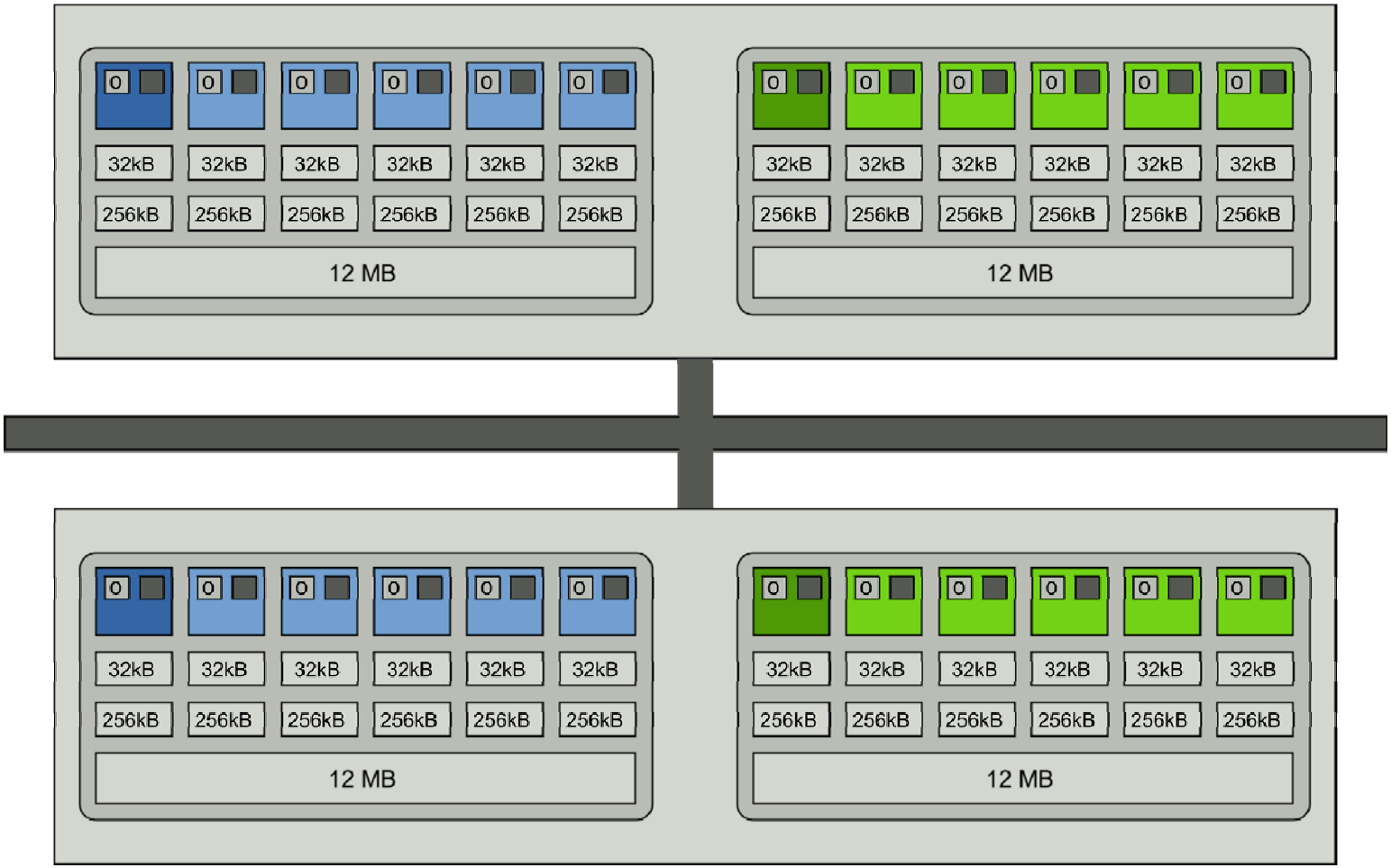


Intel MPI+compiler:

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

Examples for hybrid pinning with likwid-mpirun: 1 MPI process per socket

```
OMP_NUM_THREADS=6  likwid-mpirun -np 4 -pin S0:0-5_S1:0-5 ./a.out
```



Intel MPI+compiler:

```
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \
     -env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out
```

Monitoring the Binding

SKIPPED!

- 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
- SIMD vectorization
- 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 often *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
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

Live demo:

LIKWID tools

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
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

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);
```

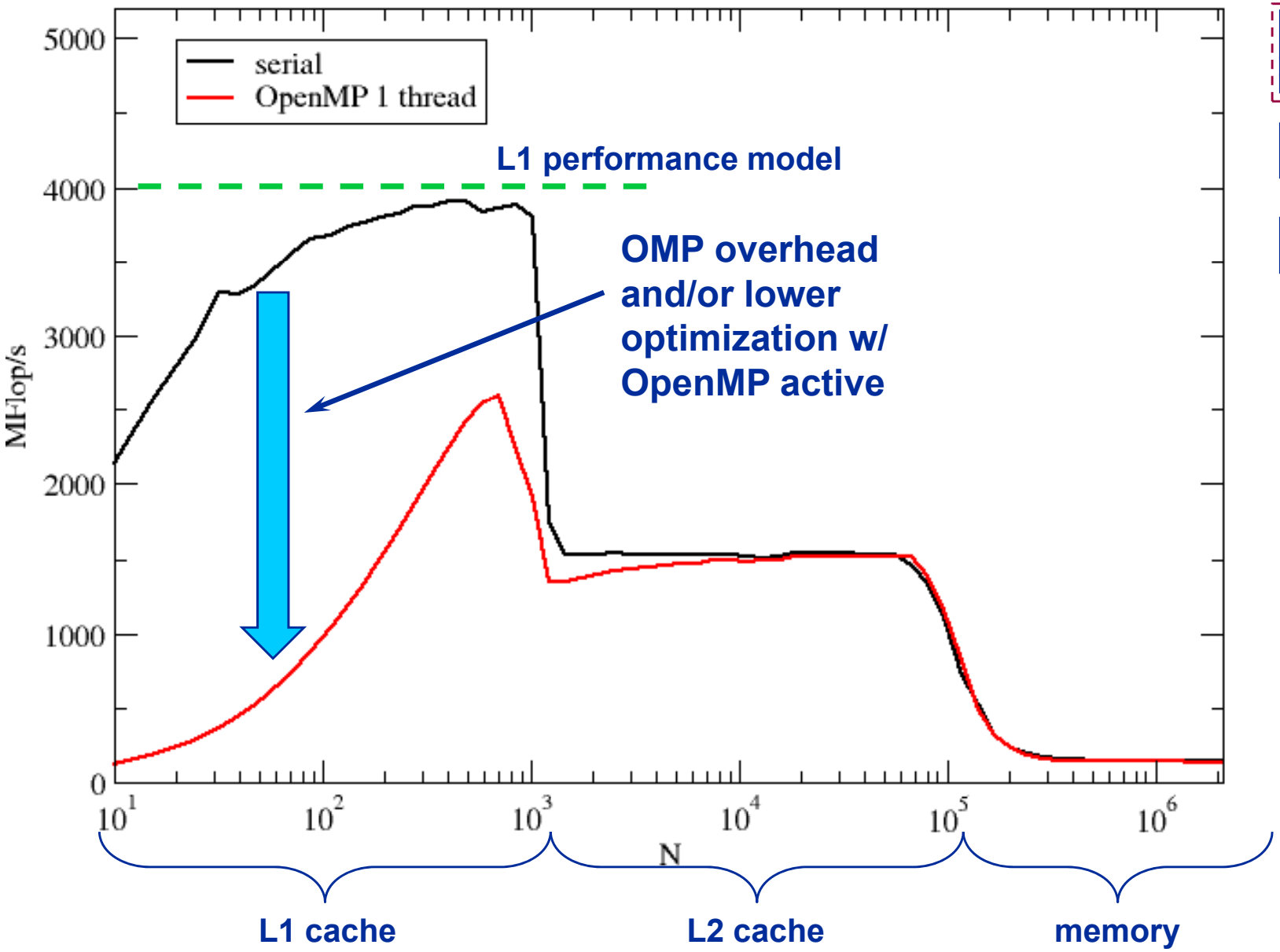
```
// size = multiple of 8
int vector_size(int n){
  return int(pow(1.3,n))&(-8);
}
```

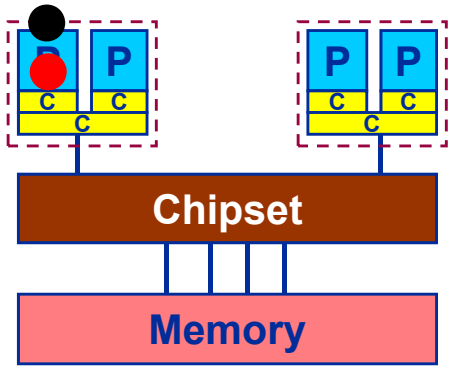
Large-N version (NT)

Small-N version (noNT)

The parallel vector triad benchmark

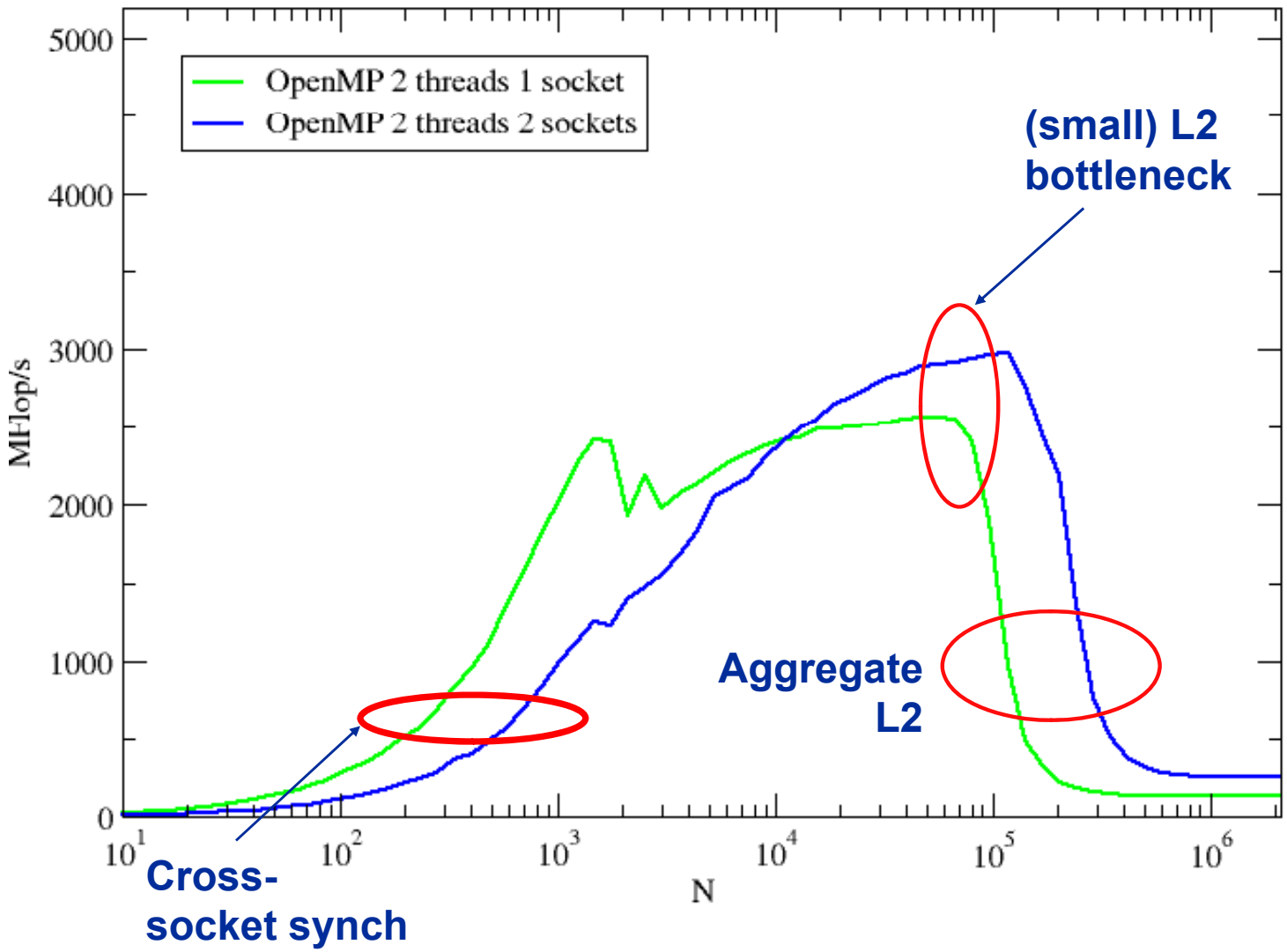
Performance results on Xeon 5160 node

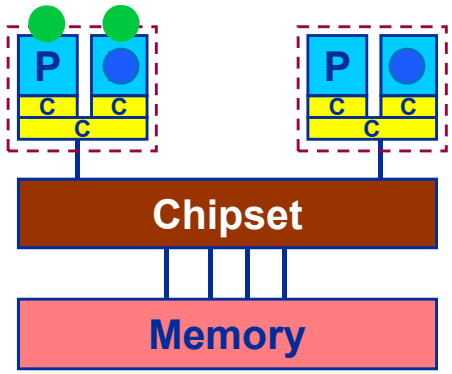




The parallel vector triad benchmark

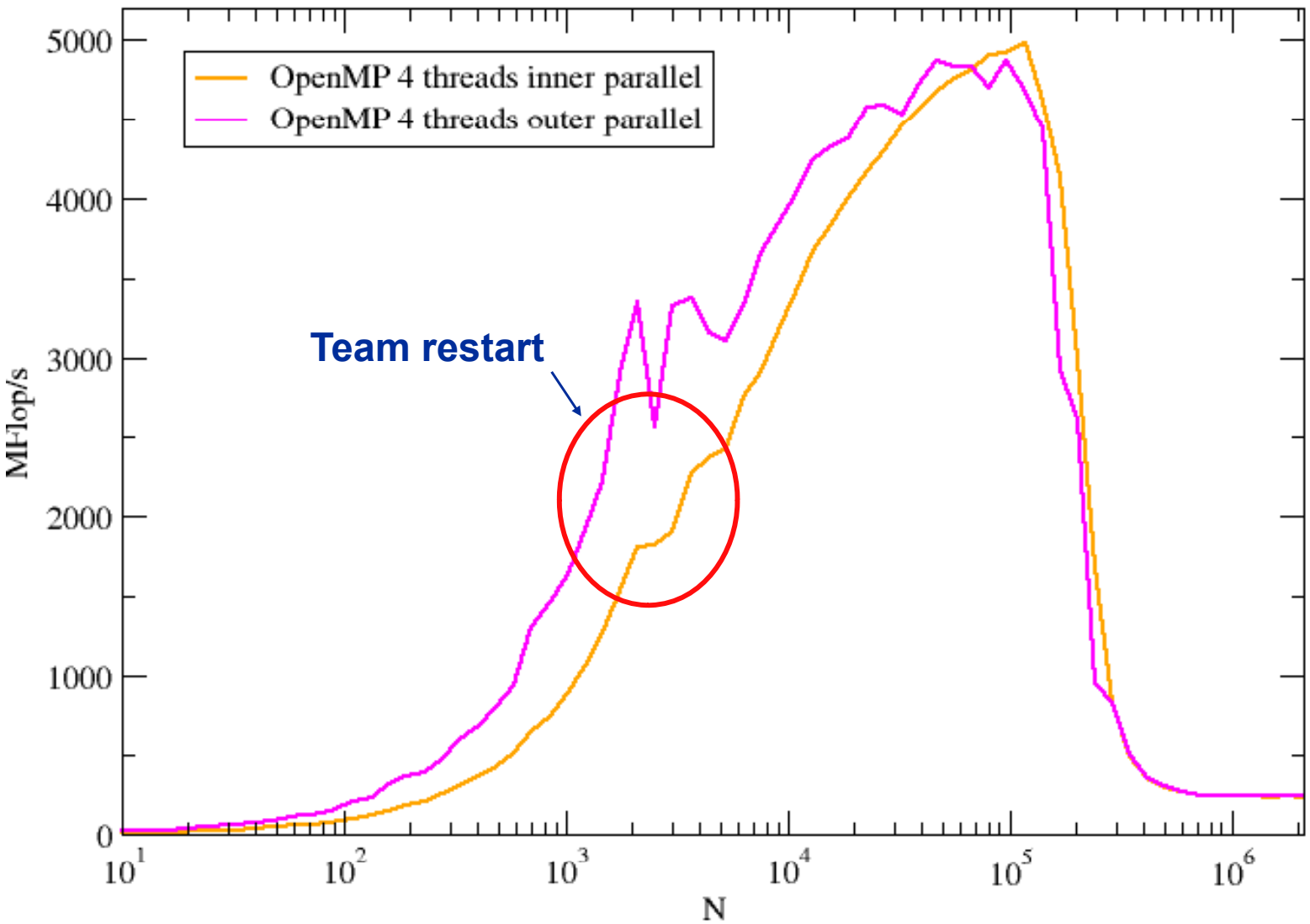
Performance results on Xeon 5160 node

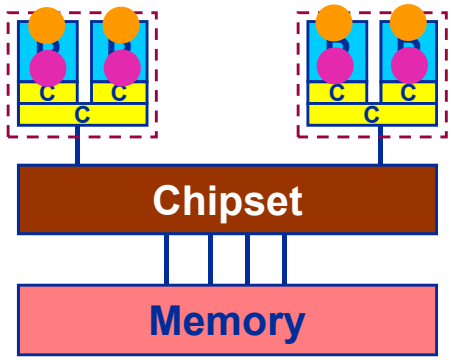




The parallel vector triad benchmark

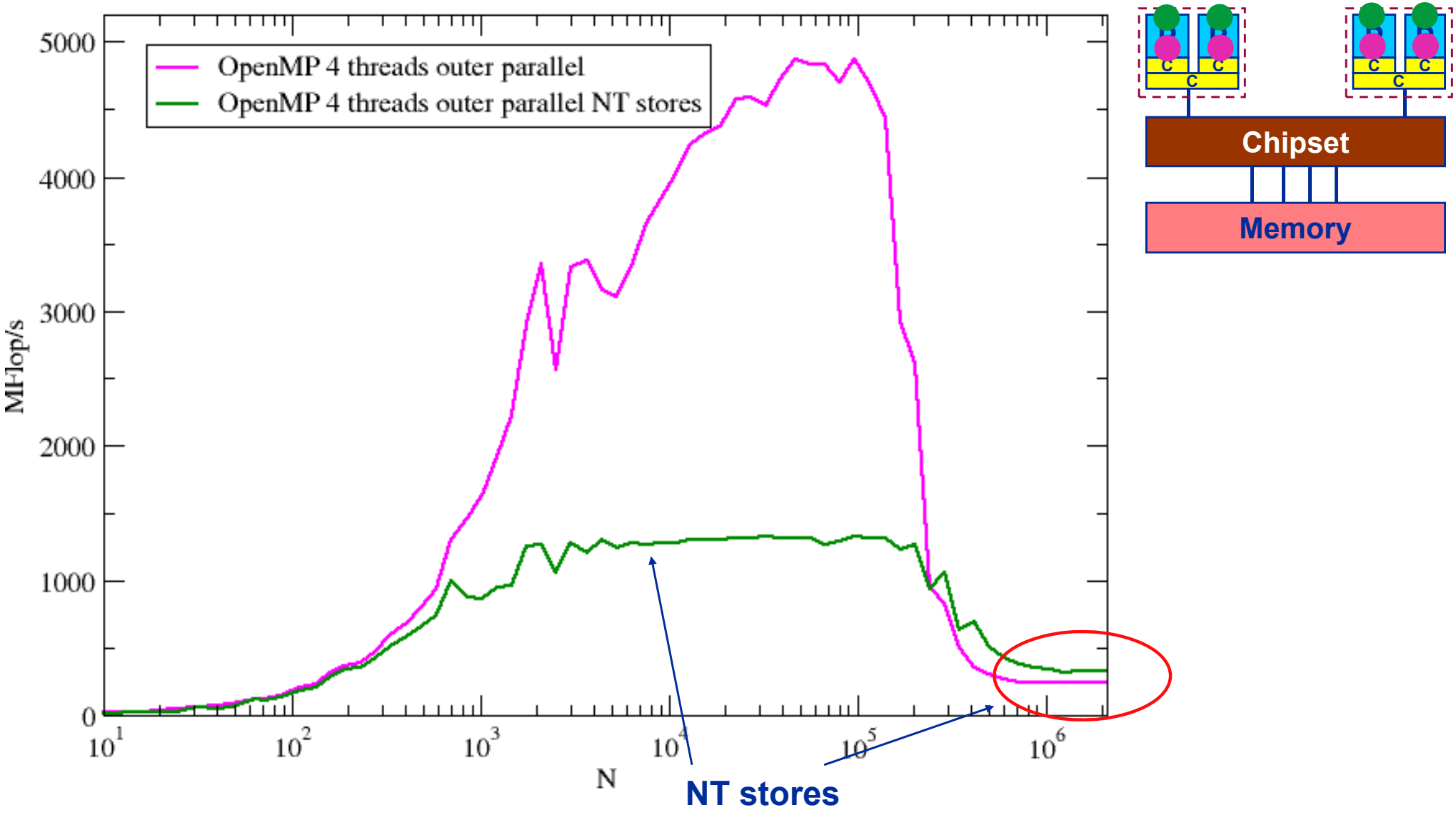
Performance results on Xeon 5160 node





The parallel vector triad benchmark

Performance results on Xeon 5160 node



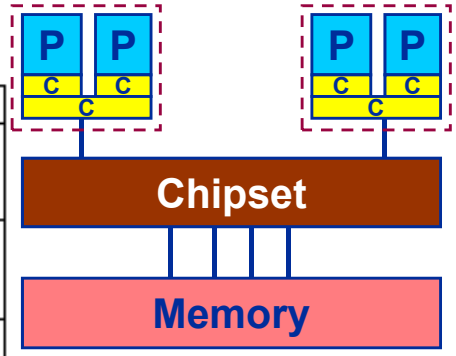
The parallel vector triad benchmark

Performance results on Xeon 5160 node

serial
1T
2T 1S
2T 2S
4T
4T outer parallel
4T outer parallel NT stores

MFlop/s

N

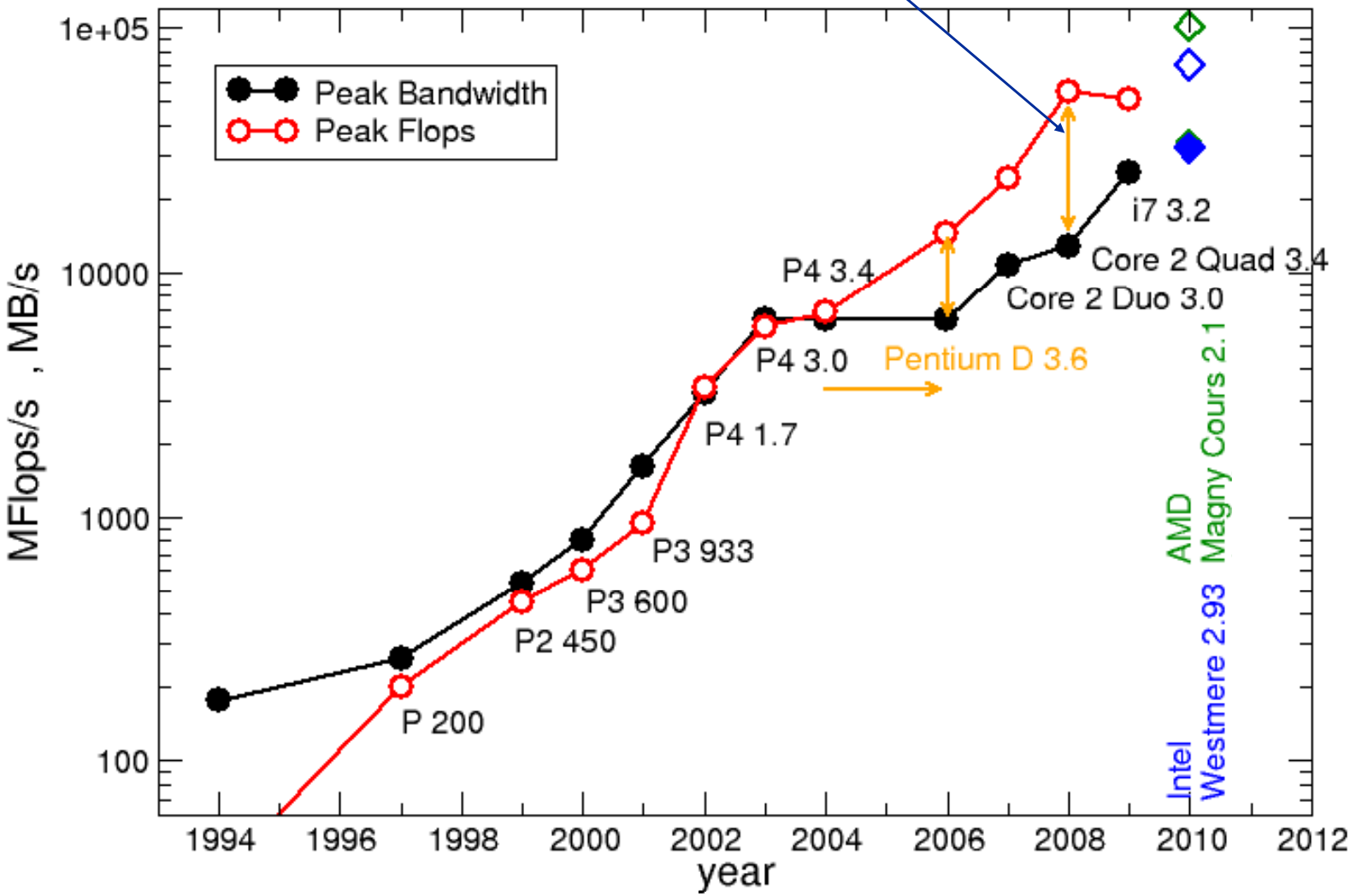


Memory BW saturation

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

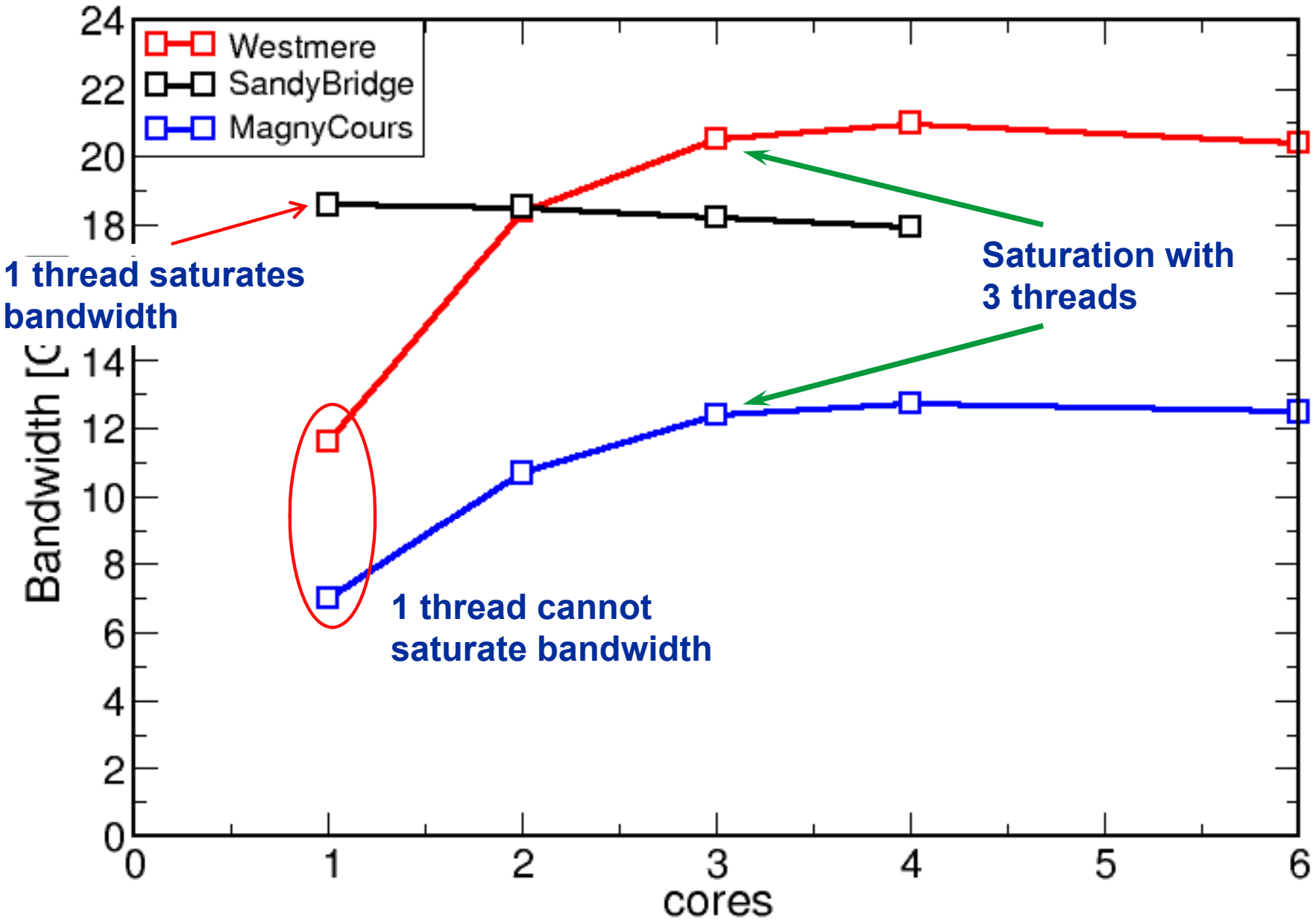
Large MatrixMatrix (BLAS3)

Bandwidth saturation effects in cache and memory

Low-level benchmark results

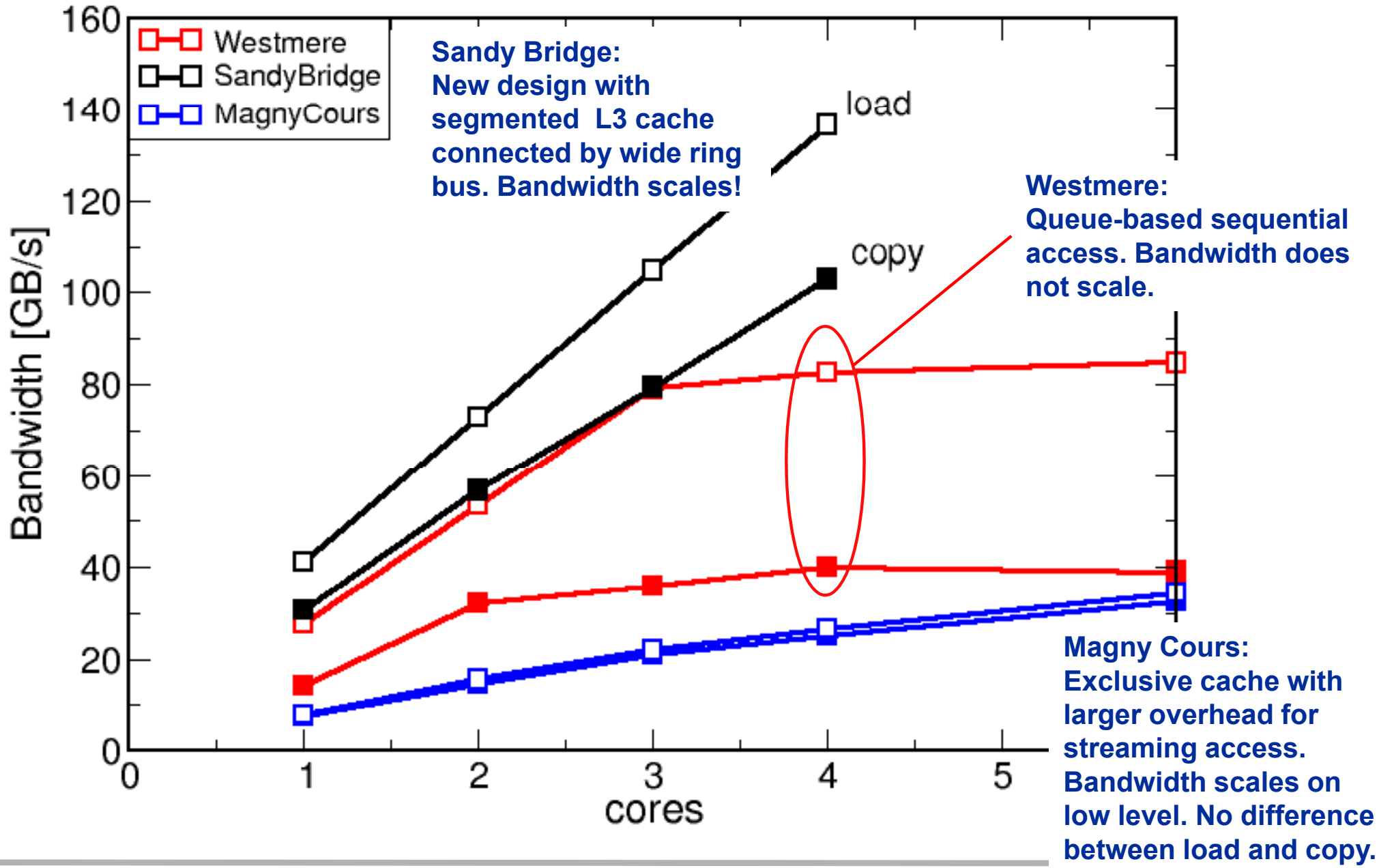
Bandwidth limitations: Main Memory

Scalability of shared data paths inside NUMA domain (`A(:)=B(:)`)



Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache



Case study: OpenMP-parallel sparse matrix-vector multiplication in depth

A simple (but sometimes not-so-simple) example for bandwidth-bound code and saturation effects in memory

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
```

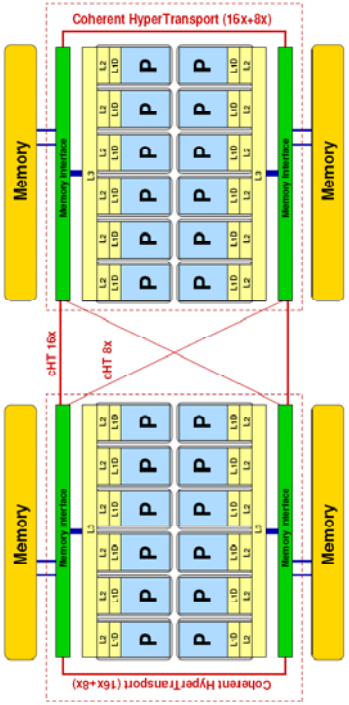
  - Usually many spMVMs required to solve a problem

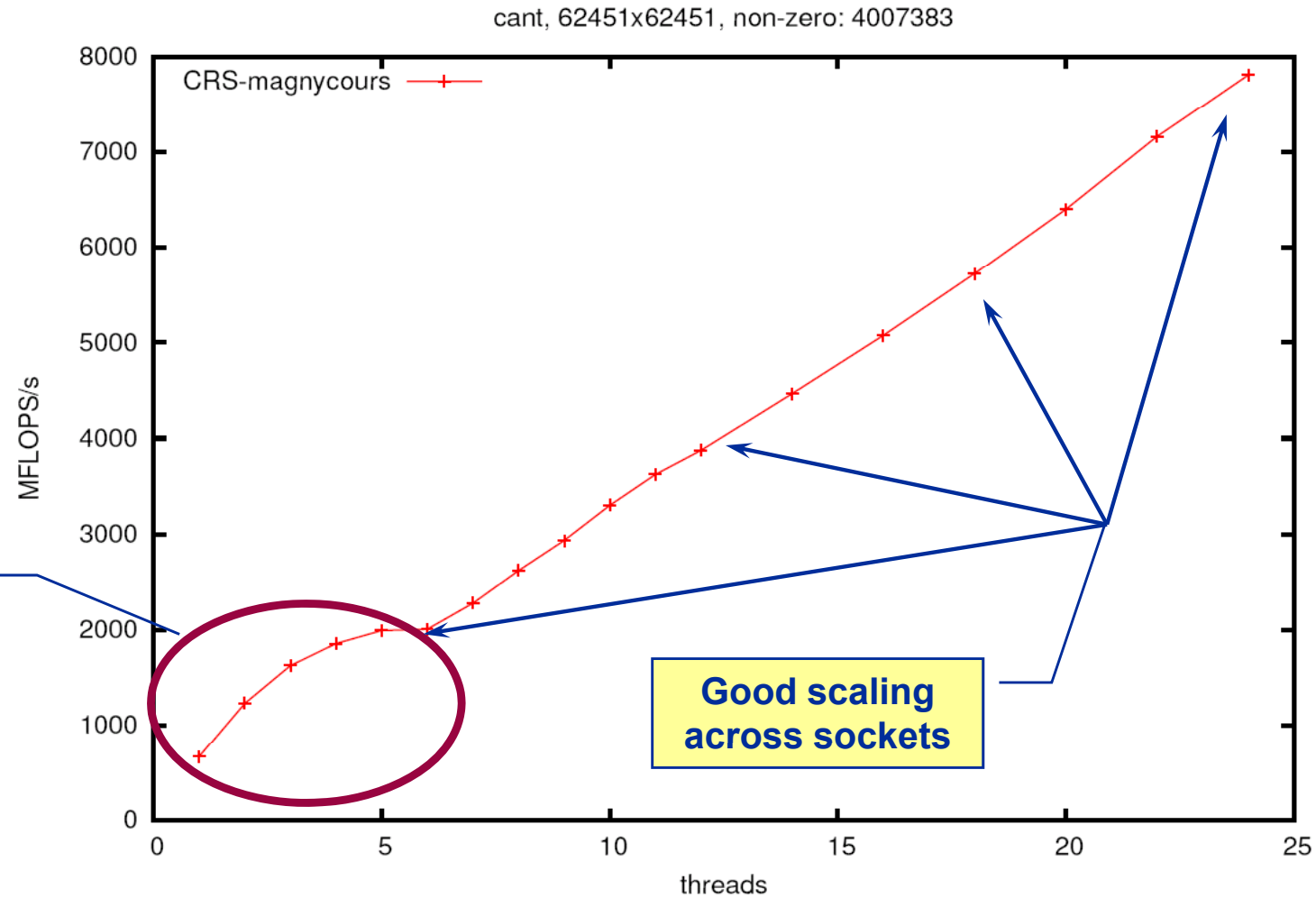
- **Following slides: Performance data on one 24-core AMD Magny Cours node**

Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- **Case 1: Large matrix**





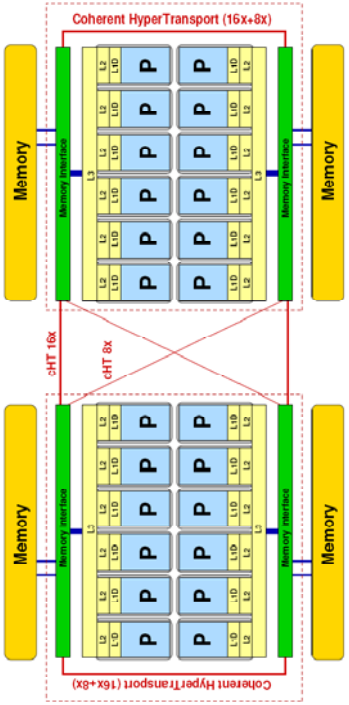
Intrasocket bandwidth bottleneck

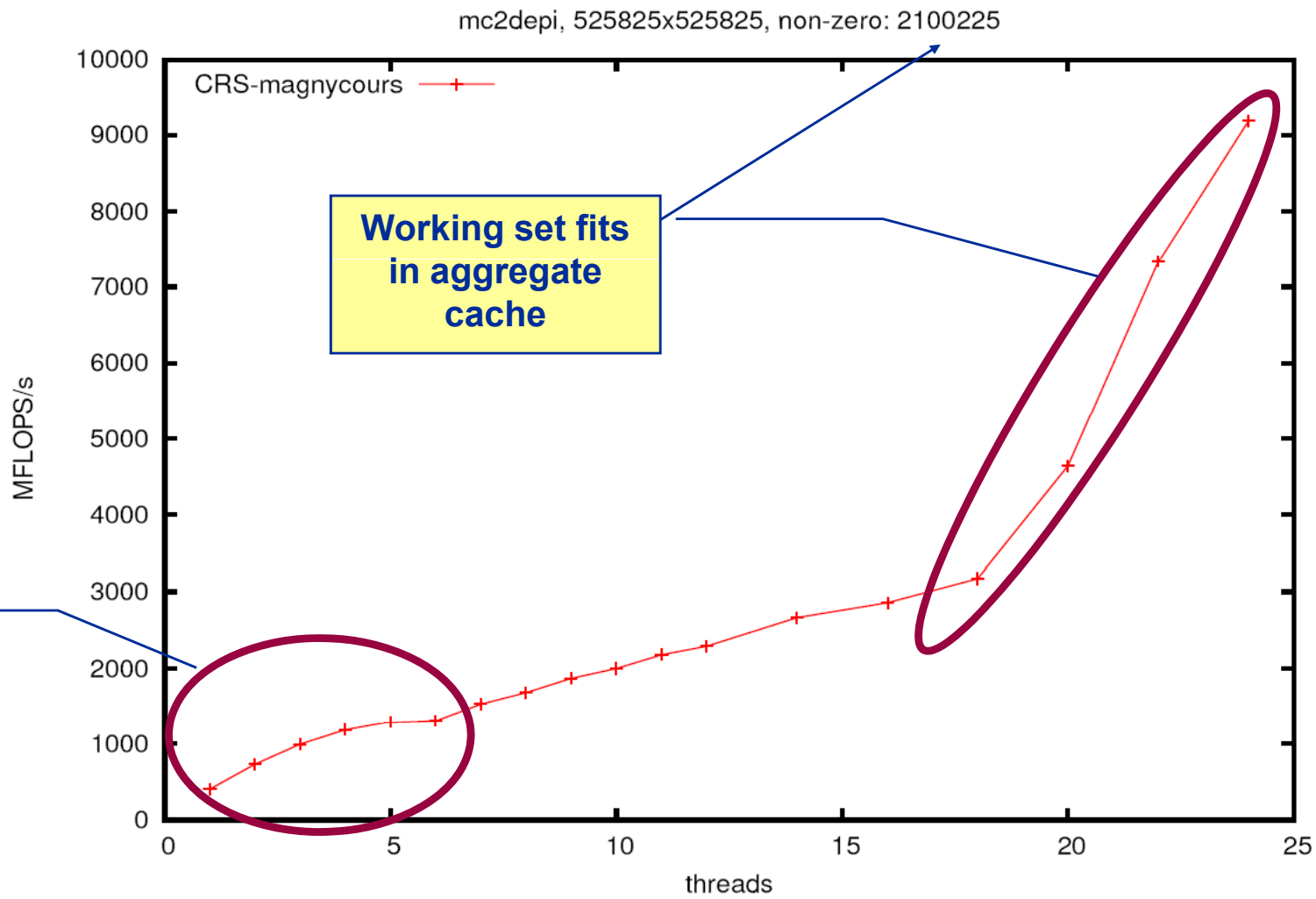
Good scaling across sockets

Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- **Case 2: Medium size**

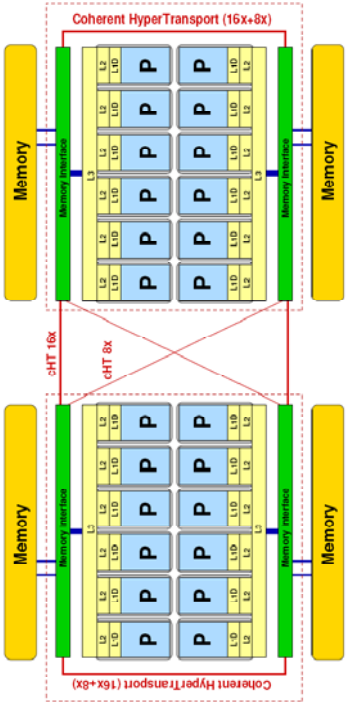


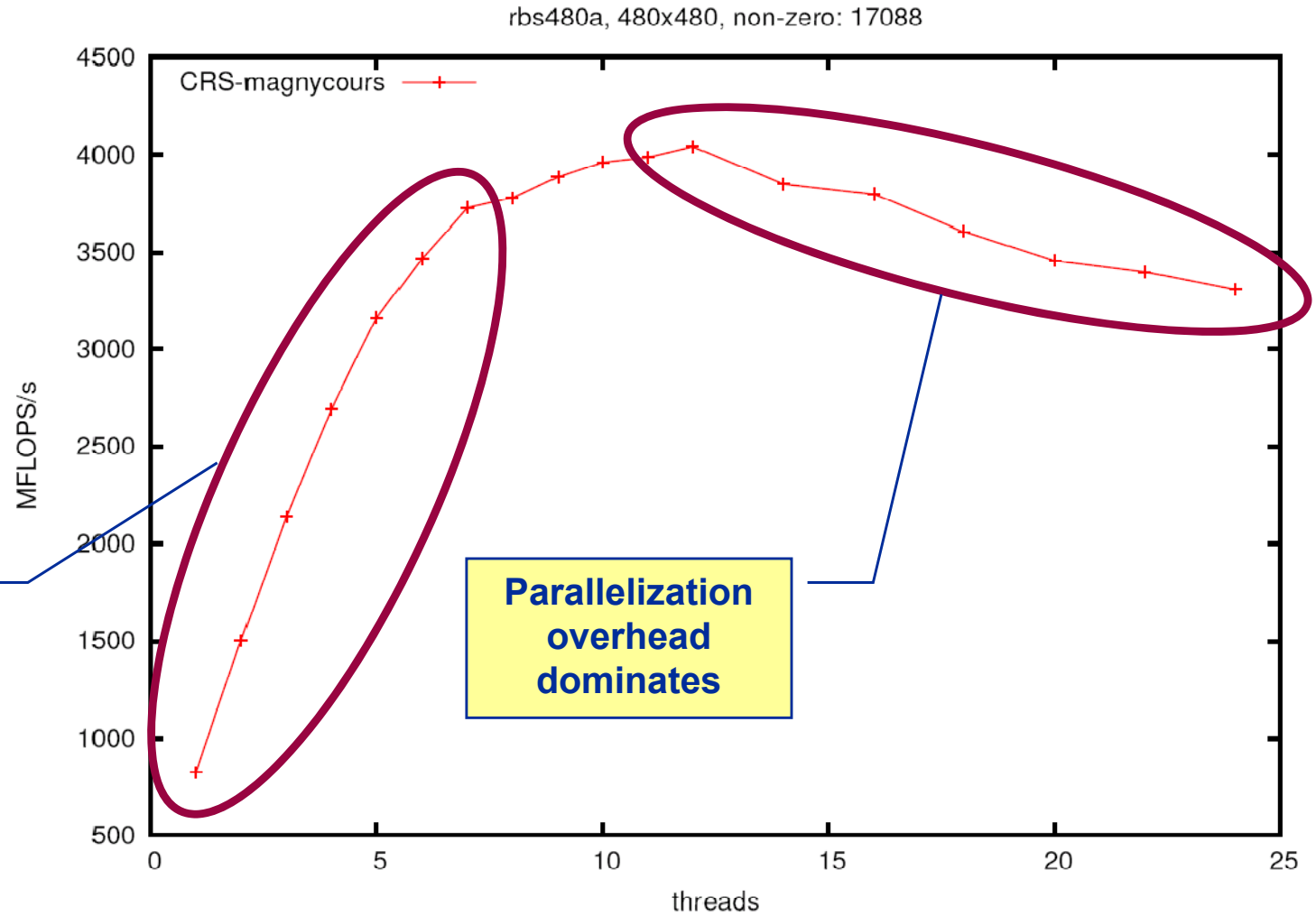


Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- **Case 3: Small size**





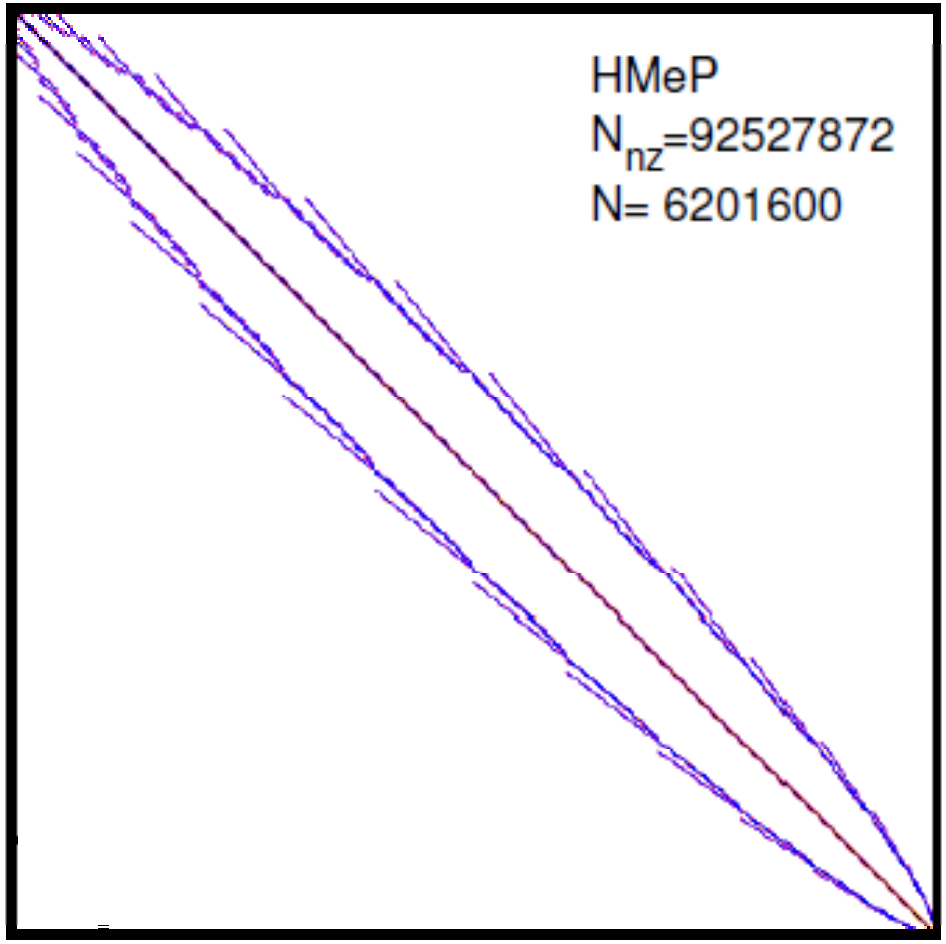
No bandwidth bottleneck

Parallelization overhead dominates

Bandwidth-bound parallel algorithms:
Sparse MVM

- **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**
  - See hybrid part for what we can do about this…

SpMVM node performance model

- **Double precision CRS:**

```
do i = 1,N_r
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

8 8 8 8 4 8

- **DP CRS code balance**

$$B_{\mathrm{CRS}} = \left(\frac{12 + 24/N_{\mathrm{nzr}} + \kappa}{2}\right) \frac{\mathrm{bytes}}{\mathrm{flop}} = \left(6 + \frac{12}{N_{\mathrm{nzr}}} + \frac{\kappa}{2}\right) \frac{\mathrm{bytes}}{\mathrm{flop}} .$$

  - κ quantifies extra traffic for loading RHS more than once
  - Predicted Performance = streamBW/B_{CRS}
  - Determine κ by measuring performance and actual memory BW

G. Schubert, G. Hager, H. Fehske and G. Wellein: Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming. Workshop on Large-Scale Parallel Processing (LSPP 2011), May 20th, 2011, Anchorage, AK. Preprint: arXiv:1101.0091

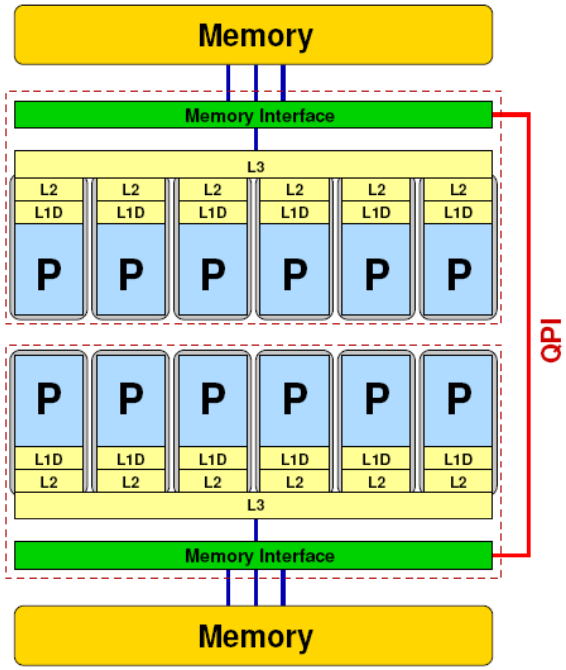
Test matrices: Sparsity patterns

- **Analysis for HMeP matrix ($N_{nzr} \approx 15$) on Nehalem EP socket**
  - BW used by spMVM kernel = 18.1 GB/s → should get ≈ 2.66 Gflop/s spMVM performance
  - Measured spMVM performance = 2.25 Gflop/s
  - Solve 2.25 Gflop/s = BW/B_{CRS} for $\kappa \approx 2.5$

    → 37.5 extra bytes per row
    → RHS is loaded ≈6 times from memory, but each element is used $N_{nzr} \approx 15$ times
    → about 25% of BW goes into RHS

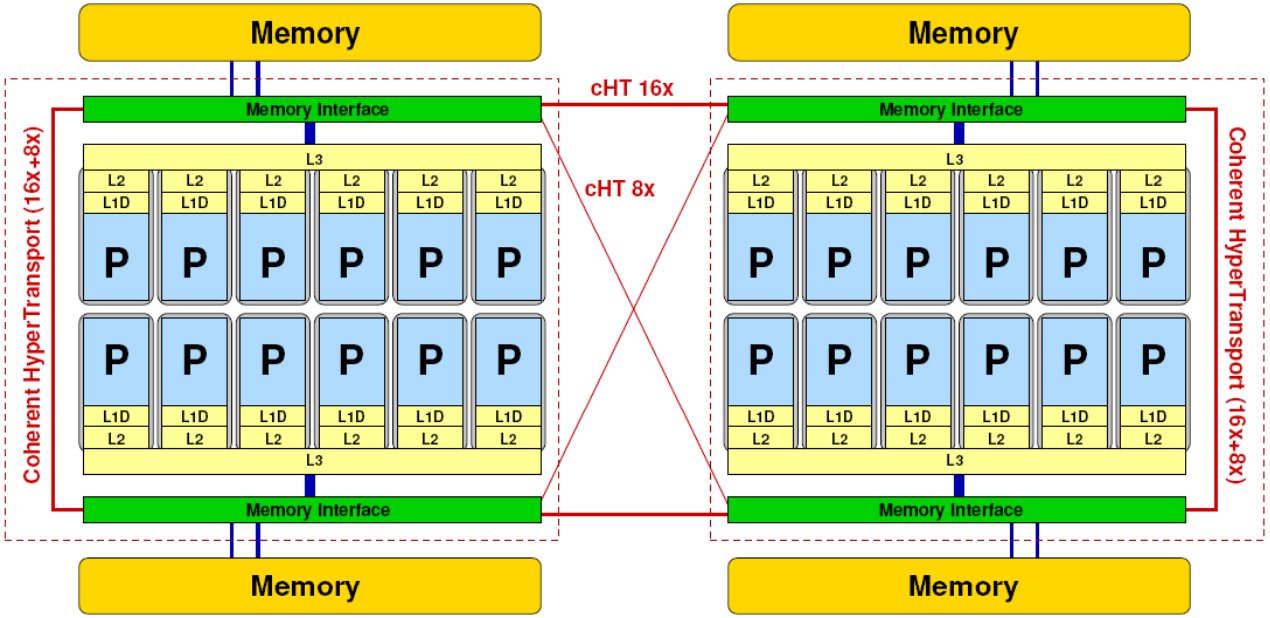
- **Special formats that exploit features of the sparsity pattern are not considered here**
  - Symmetry
  - Dense blocks
  - Subdiagonals (possibly w/ constant entries)

Test systems

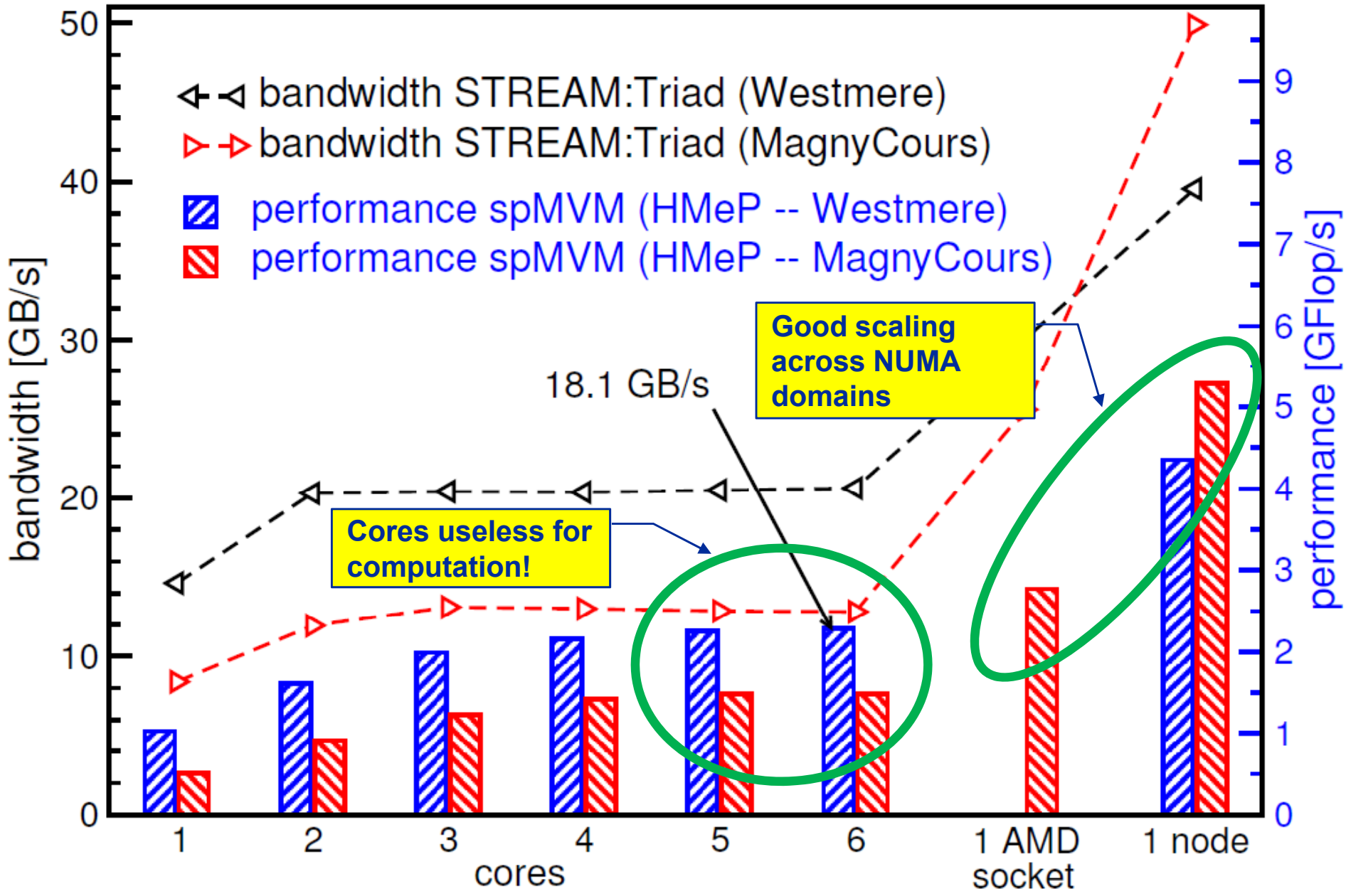


- **Intel Westmere EP (Xeon 5650)**
- **STREAM triad BW: 20.6 GB/s per domain**
- **QDR InfiniBand fully nonblocking fat-tree interconnect**

- **AMD Magny Cours (Opteron 6172)**
- **STREAM triad BW: 12.8 GB/s per domain**
- **Cray Gemini interconnect**



Node-level performance for HMeP: Westmere EP (Xeon 5650) vs. Cray XE6 Magny Cours (Opteron 6172)



OpenMP sparse MVM:
Take-home messages

- **Yes, sparse MVM is usually memory-bound**
- **This statement is insufficient for a full understanding of what's going on**
  - Nonzeros (matrix data) may not take up 100% of bandwidth
  - We can figure out easily how often the RHS has to be loaded
- **A lot of research is put into bandwidth reduction optimizations for sparse MVM**
  - Symmetries, dense subblocks, subdiagonals,…
- **Bandwidth saturation → using all cores may not be required**
  - There are free resources – what can we do with them?
    - Turn off/reduce clock frequency
    - Put to better use → see hybrid case studies

Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

First touch placement policy

C++ issues

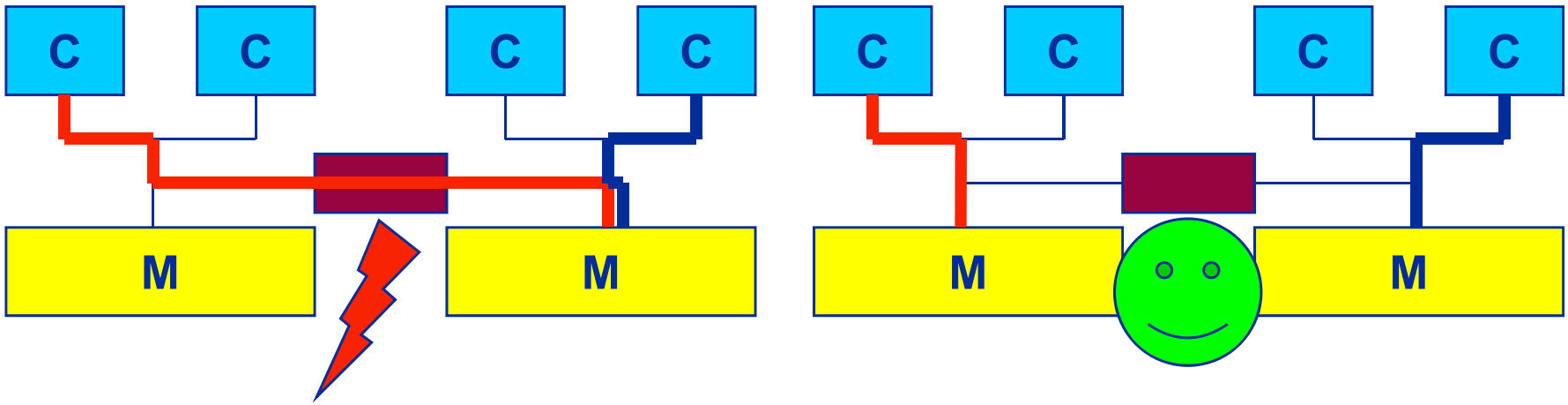
ccNUMA locality and dynamic scheduling

ccNUMA locality beyond first touch

ccNUMA performance problems

"The other affinity" to care about

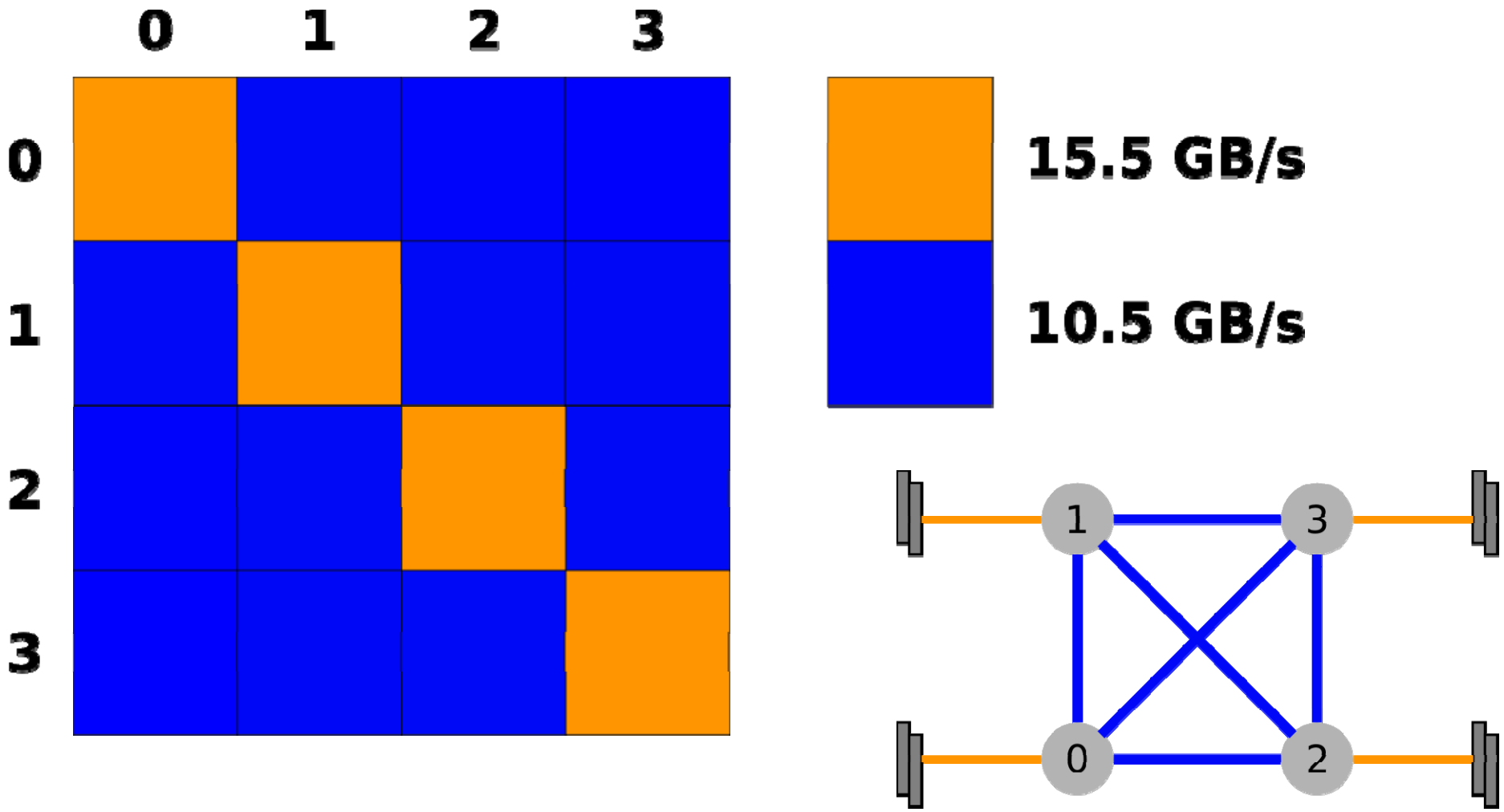
- **ccNUMA:**
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
- **How do we make sure that memory access is always as "local" and "distributed" as possible?**



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

Intel Nehalem EX 4-socket system

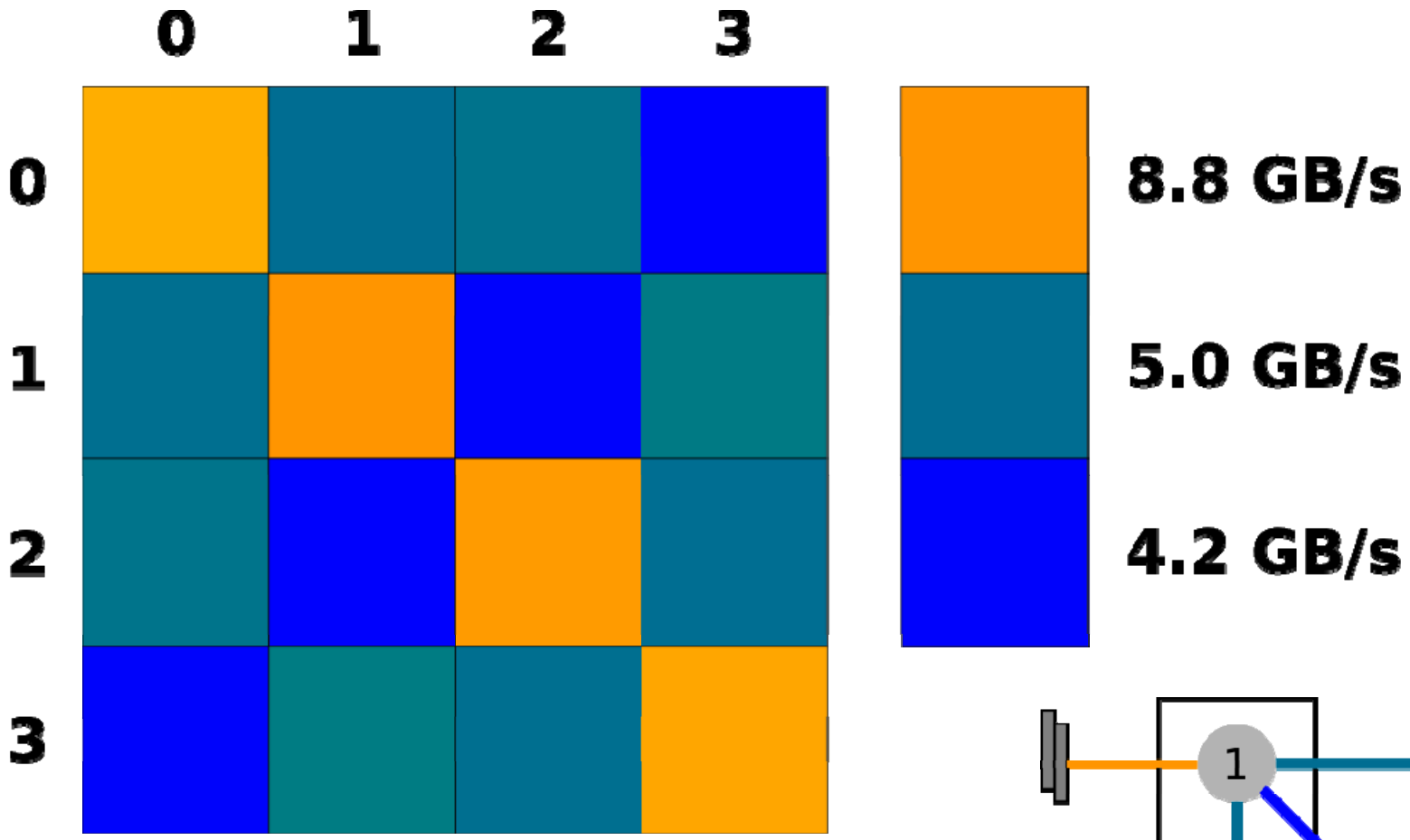
ccNUMA bandwidth map

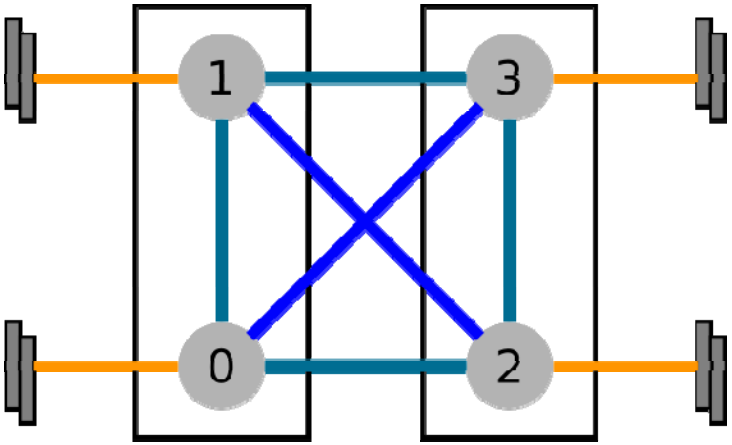


Bandwidth map created with likwid-bench. All cores used in one NUMA domain, memory is placed in a different NUMA domain.
Test case: simple copy `A(:)=B(:)`, large arrays

AMD Magny Cours 2-socket system

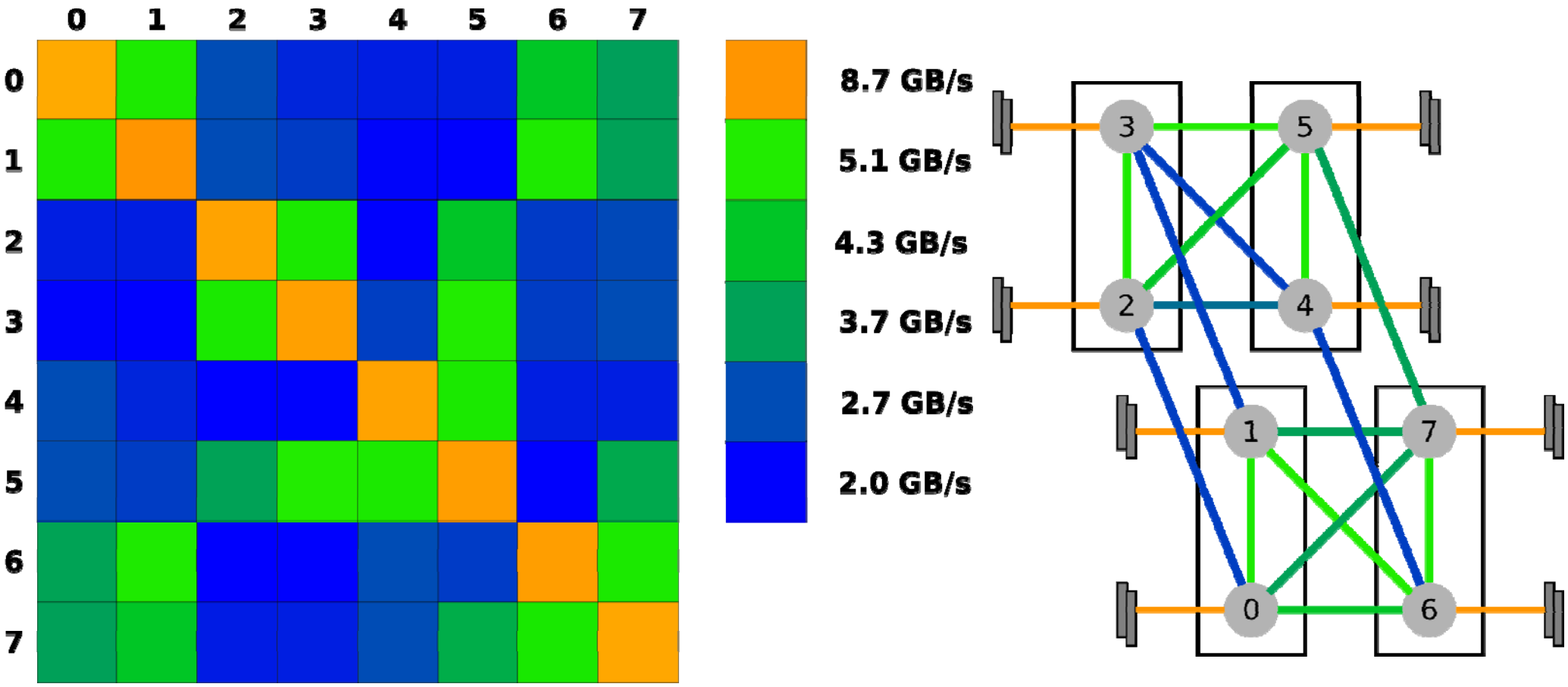
4 chips, two sockets





AMD Magny Cours 4-socket system

Topology at its best?



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???
  …
}
```

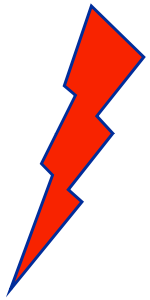
  - Fine-tuning is possible (see later)

- **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 (see hidden slides)

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

optional

- **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

optional

- **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<>`

optional

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

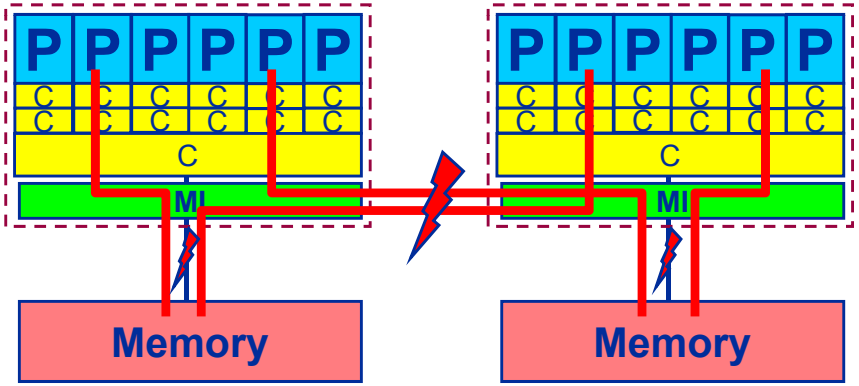
Application:

```
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…

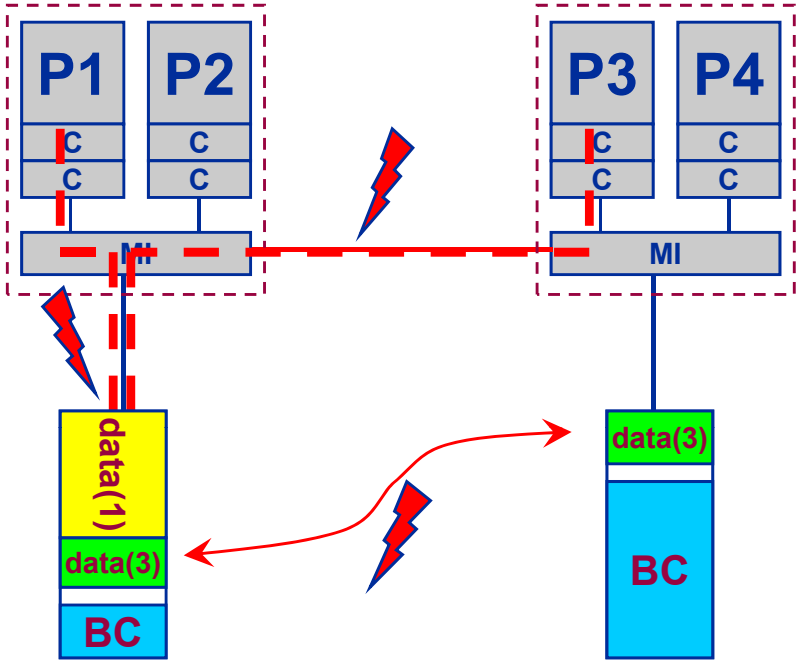
- **Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?**
  - Program has erratic access patters → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```
# 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
  - `numactl` tool can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer cache size in standard kernels

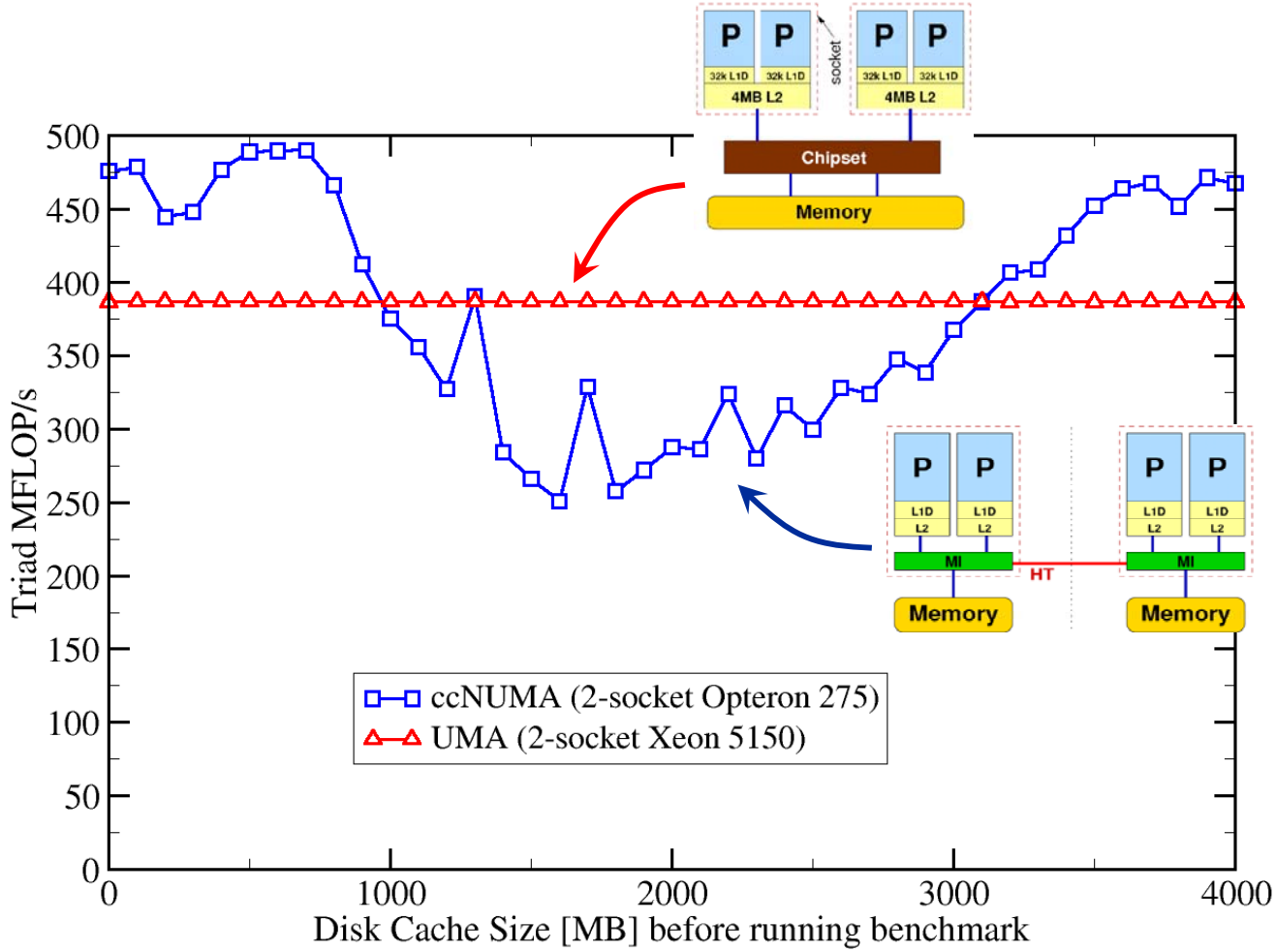
ccNUMA problems beyond first touch:
Buffer cache

- **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 perfor-mance 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`:

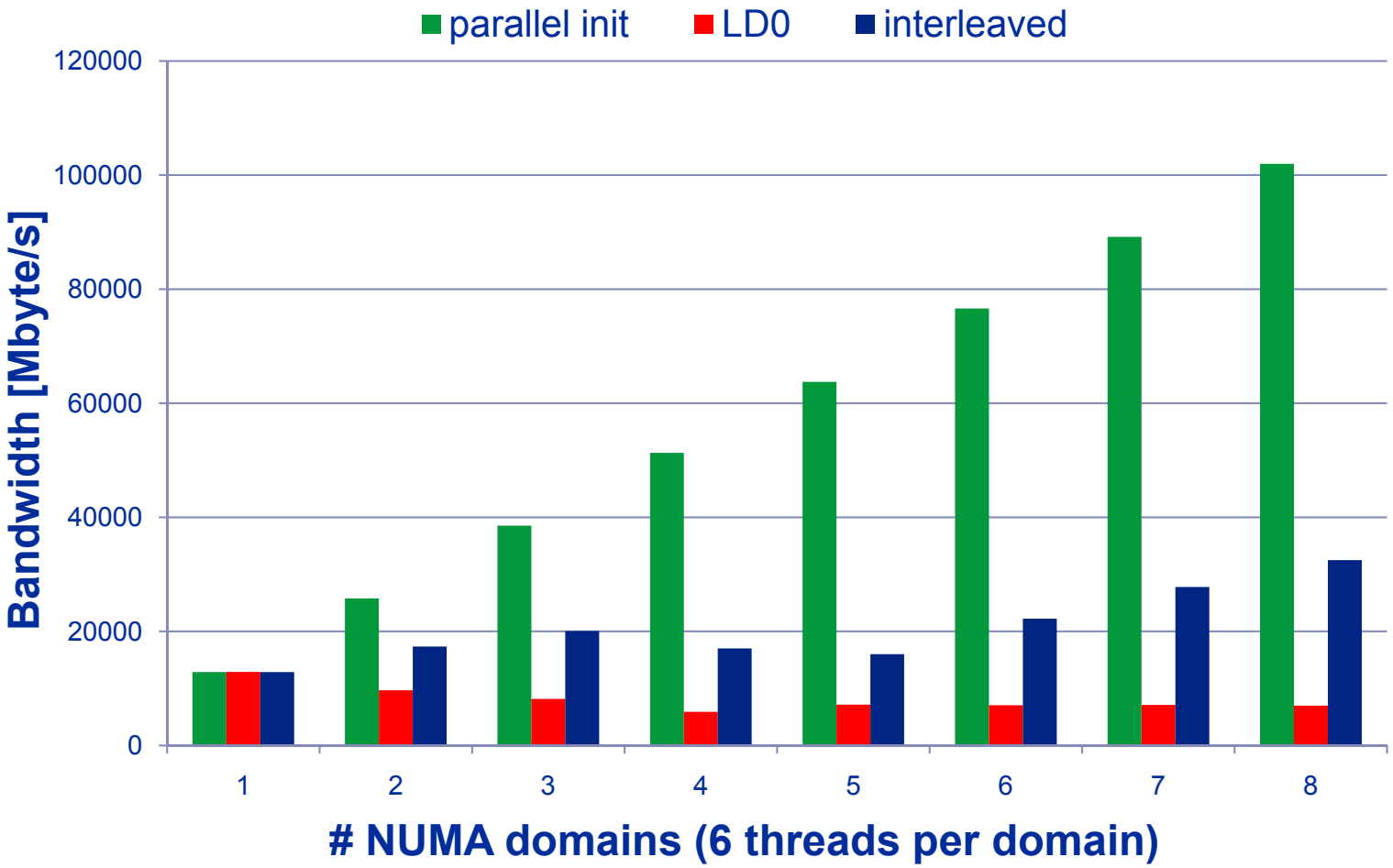
```
numactl --interleave=0-3 ./a.out
```

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**

The curse and blessing of interleaved placement: OpenMP STREAM triad on 4-socket (48 core) Magny Cours node

- **Parallel init**: Correct parallel initialization
- **LD0**: Force data into LD0 via `numactl -m 0`
- **Interleaved**: `numactl --interleave <LD range>`



OpenMP performance issues on multicore

Synchronization (barrier) overhead

Work distribution overhead

Welcome to the multi-/many-core era

Synchronization of threads may be expensive!

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

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP progams.

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

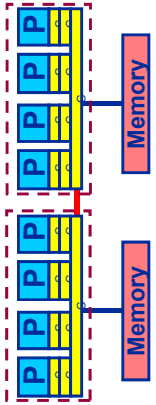
Barrier overhead in CPU cycles: pthreads vs. OpenMP vs. spin loop

4 Threads	Q9550	i7 920 (shared L3)
pthreads_barrier_wait	42533	9820
omp barrier (icc 11.0)	977	814
gcc 4.4.3	41154	8075
Spin loop	1106	475

pthreads → OS kernel call

Spin loop does fine for shared cache sync

OpenMP & Intel compiler



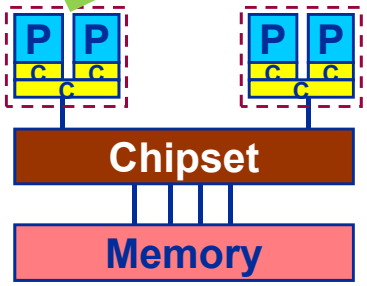
Nehalem 2 Threads	Shared SMT threads	shared L3	different socket
pthreads_barrier_wait	23352	4796	49237
omp barrier (icc 11.0)	2761	479	1206
Spin loop	17388	267	787

SMT can be a big performance problem for synchronizing threads

Work distribution overhead

Influence of thread-core affinity

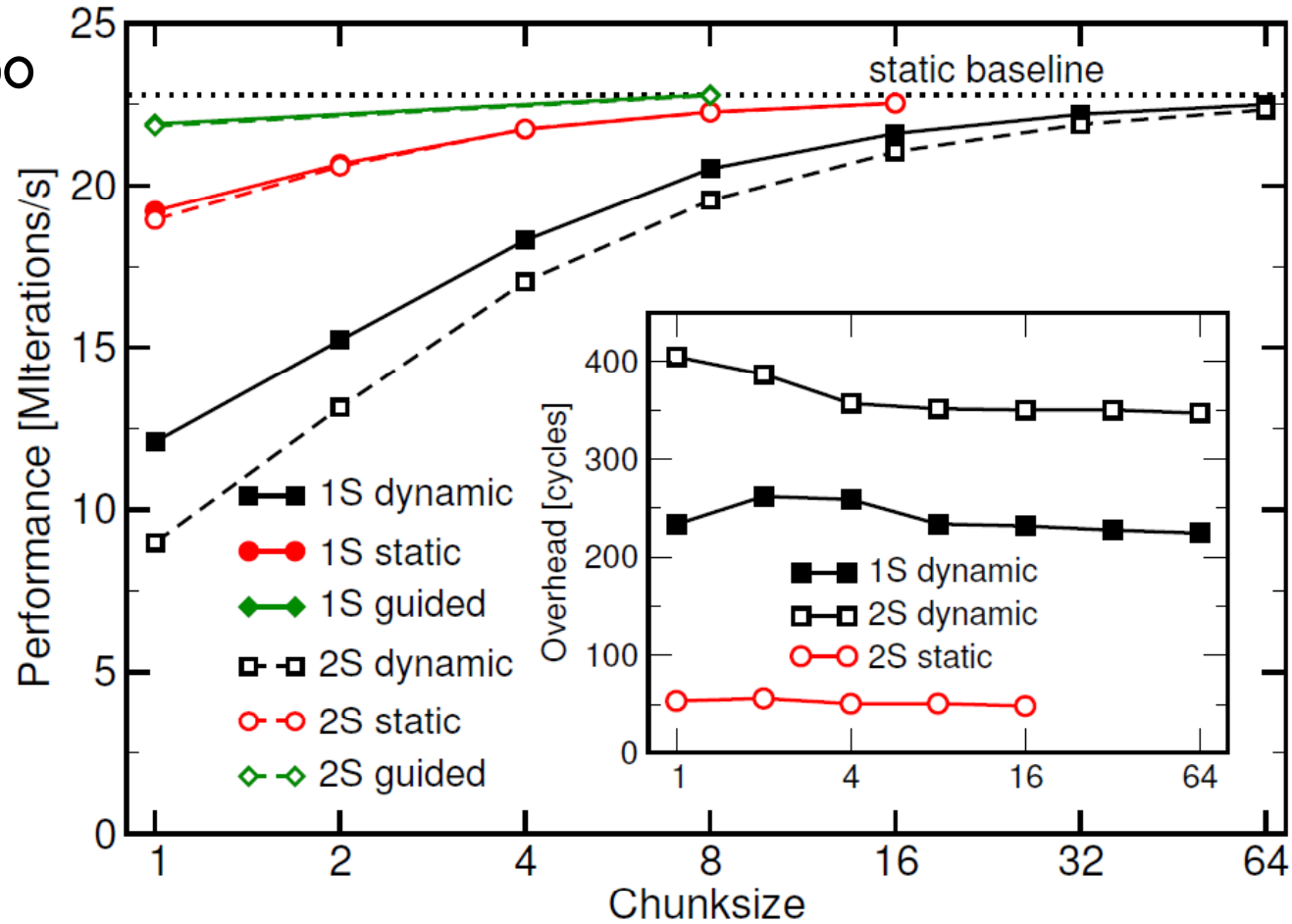
SKIPPED



- **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**



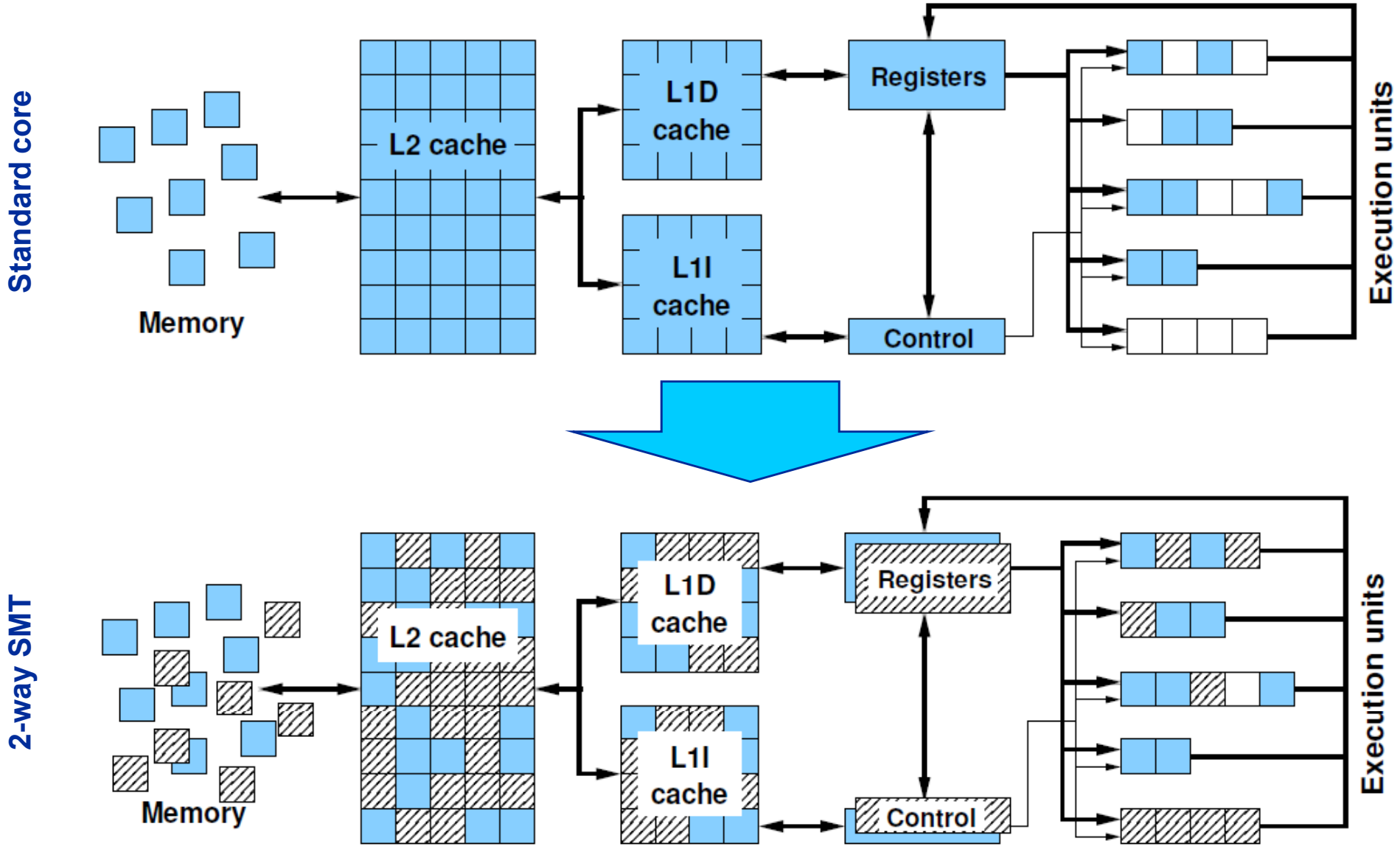
Simultaneous multithreading (SMT)

Principles and performance impact

Facts and fiction

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

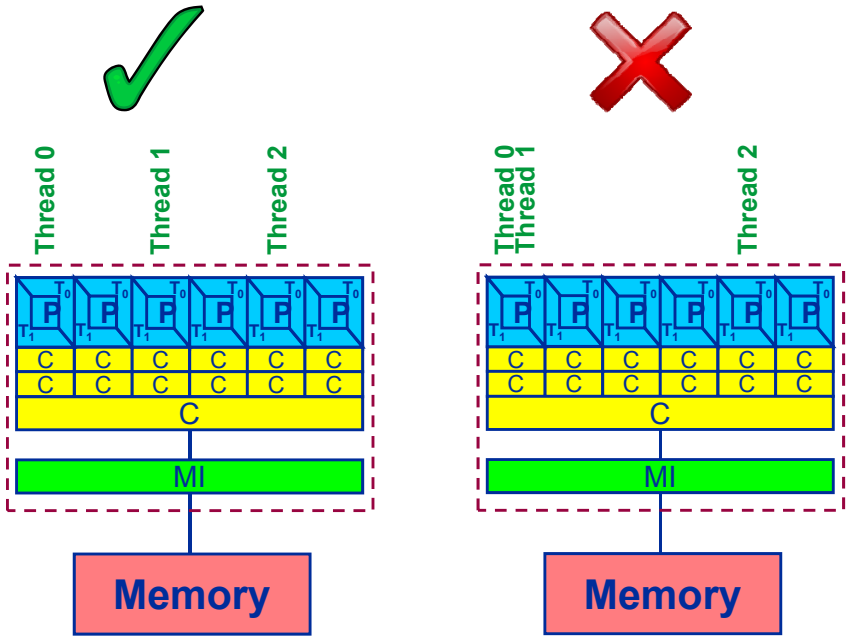
- **SMT principle (2-way example):**



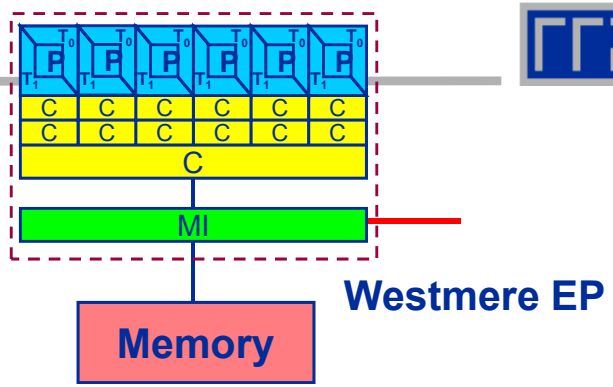
SMT impact

- **SMT 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 impact



- **SMT adds another layer of topology (inside the physical core)**
- **Caveat: SMT threads share all caches!**
- **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

- **Interesting case: SMT as an alternative to outer loop unrolling**

Original code (badly pipelined)

```
do i=1,N
  ! Iterations of j loop indep.
  do j=1,M
  !
  ! very complex loop body with
  ! many flops and massive
  ! register dependencies
  !
  enddo
enddo
```

"Optimized" code

```
do i=1,N,2
  ! Iterations of j loop indep.
  do j=1,M
  !
  ! loop body, 2 copies
  ! interleaved → better
  ! pipeline utilization
  !
  enddo
enddo
```

- **This does not work!**
  - Massive register use forbids outer loop unrolling: Register shortage/spill
- **Remedy: Parallelize one of the loops across virtual cores!**
  - Each virtual core has its own register set, so SMT will fill the pipeline bubbles

J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein: *Pushing the limits for medical image reconstruction on recent standard multicore processors*. Submitted. Preprint: arXiv:1104.5243

SMT myths: Facts and fiction

- **Myth: "If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement."**
- **Truth: A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.**

- **Myth: "If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory."**
- **Truth: If all SMT threads wait for memory, nothing is gained. SMT can help here only if the additional threads execute code that is *not* waiting for memory.**

- **Myth: "SMT can help bridge the latency to memory (more outstanding references)."**
- **Truth: Outstanding loads are a shared resource across all SMT threads. SMT will not help.**

SMT: When it may help, and when not

Functional parallelization (see hybrid case studies)	✓ ✗
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

Intranode vs. internode MPI

MPI Cartesian topologies and rank-subdomain mapping

Intranode MPI

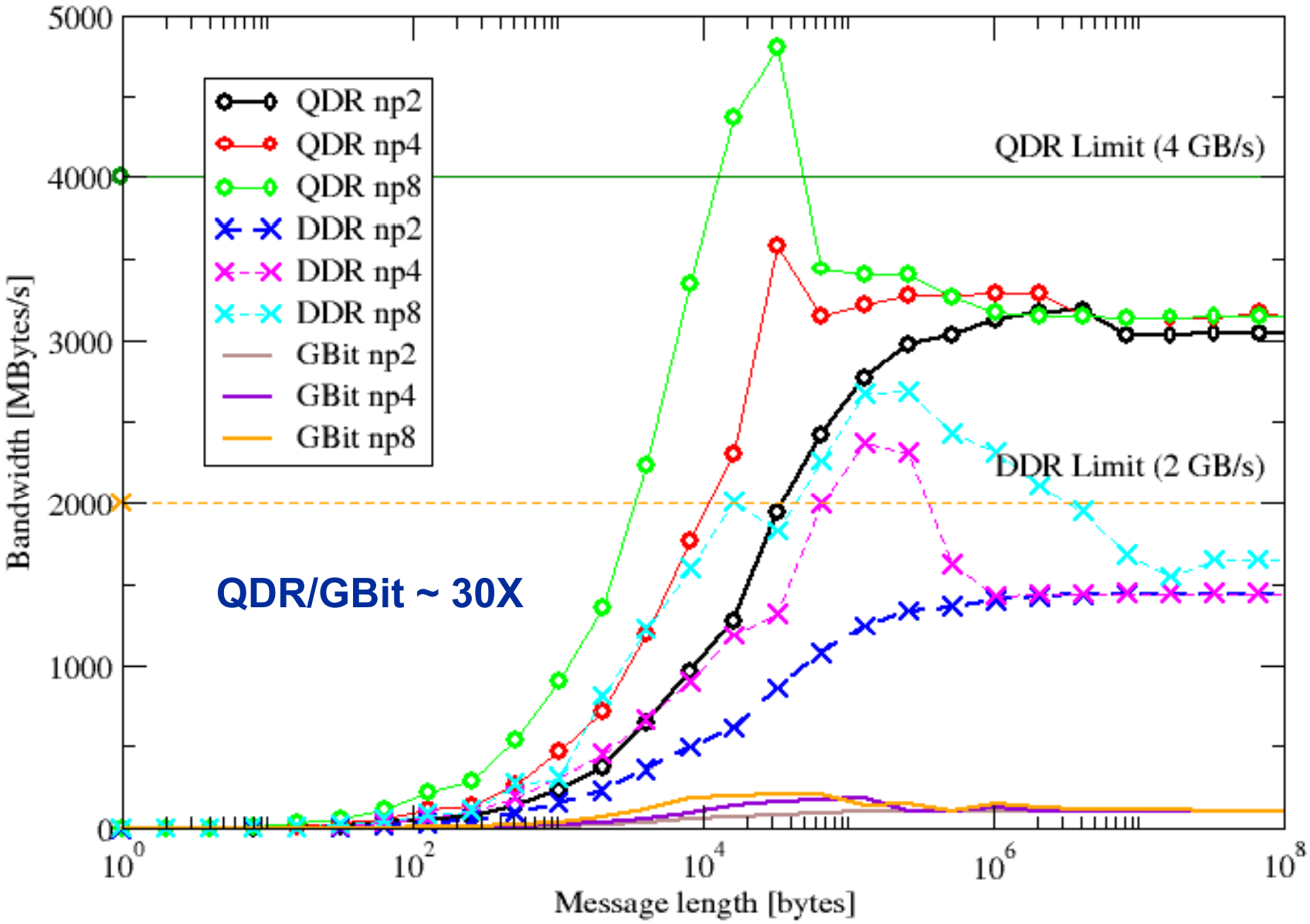
- **Common misconception: Intranode MPI is infinitely fast compared to internode**

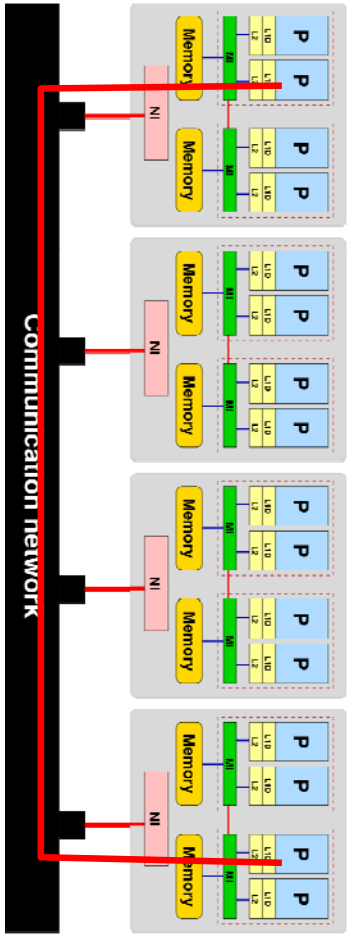
- **Reality**
  - Intranode latency is much smaller than internode
  - Intranode asymptotic bandwidth is surprisingly comparable to internode
  - Difference in saturation behavior

- **Other issues**
  - Mapping between ranks, subdomains and cores with Cartesian MPI topologies
  - Overlapping intranode with internode communication

MPI and Multicores

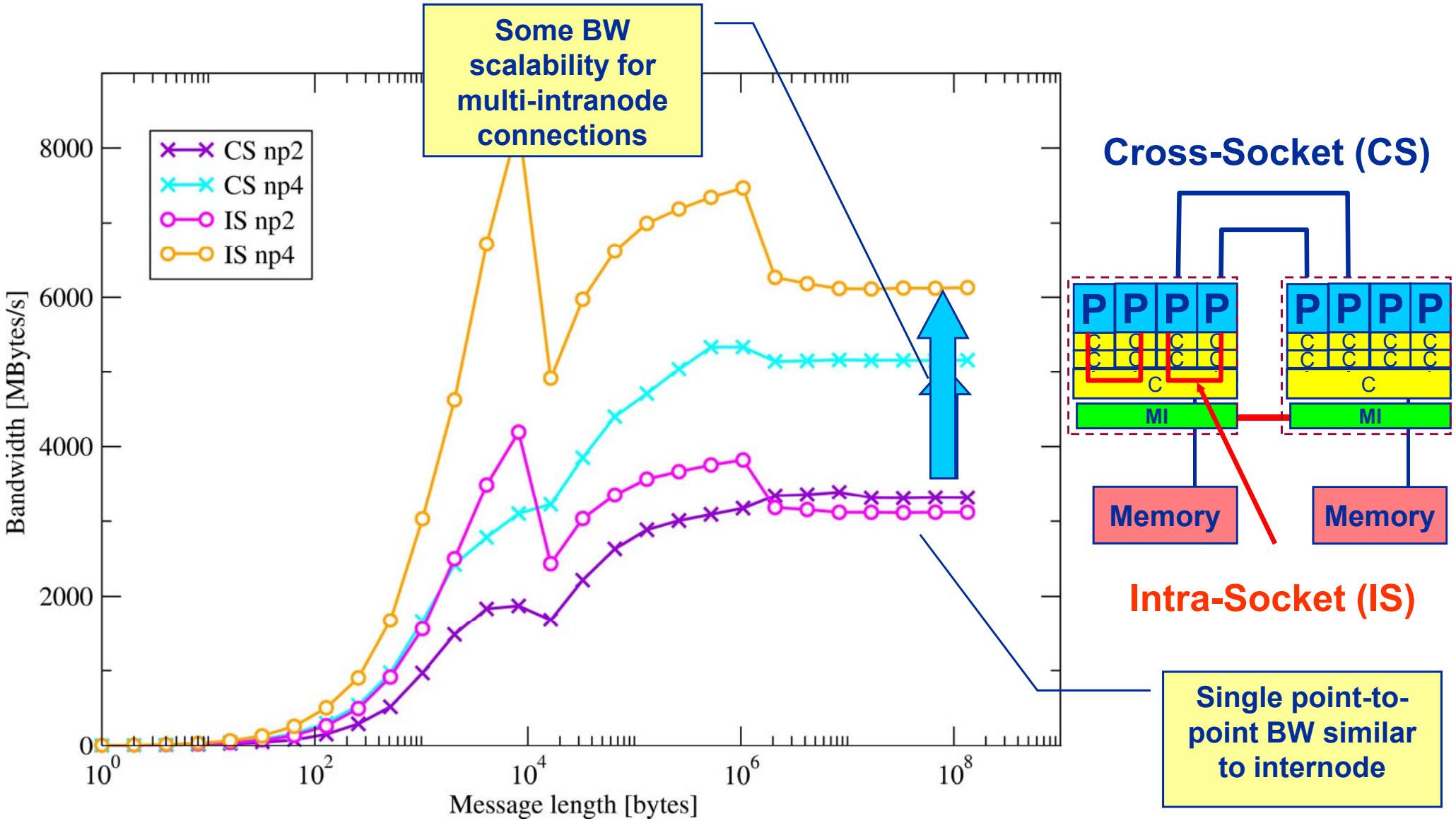
Clusters: Unidirectional internode Ping-Pong bandwidth





MPI and Multicores

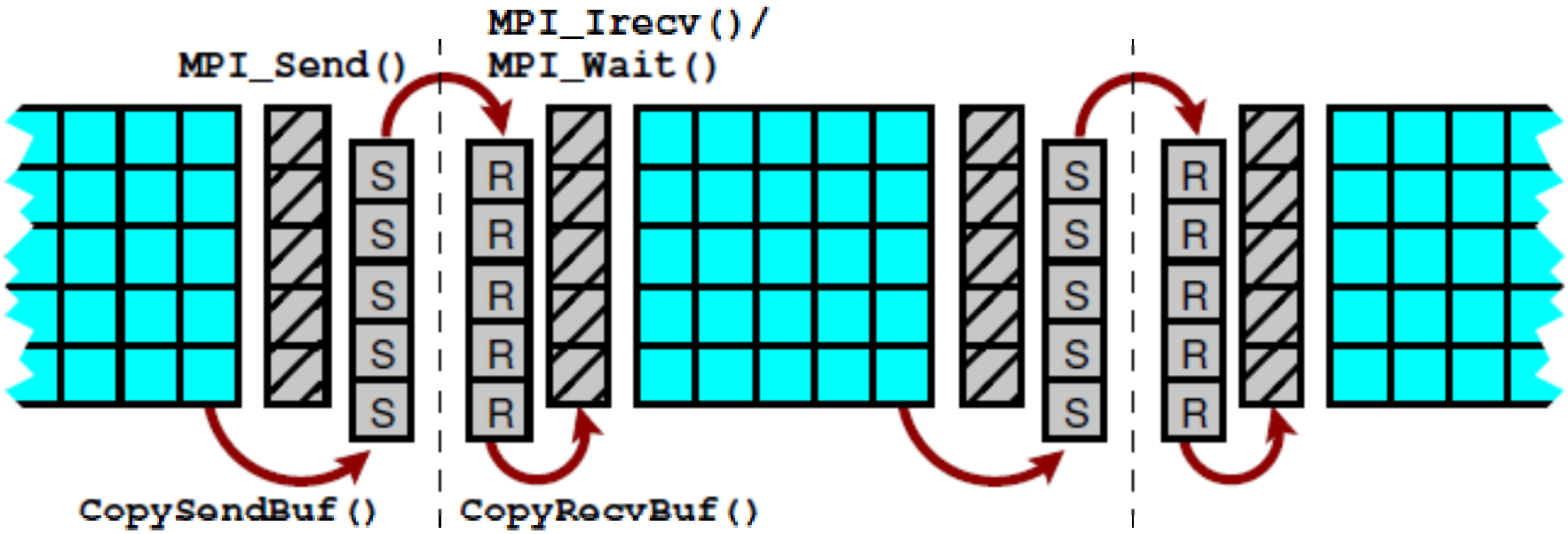
Clusters: Unidirectional intranode Ping-Pong bandwidth



Mapping problem for most efficient communication paths!?

"Best possible" MPI:
Minimizing cross-node communication

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

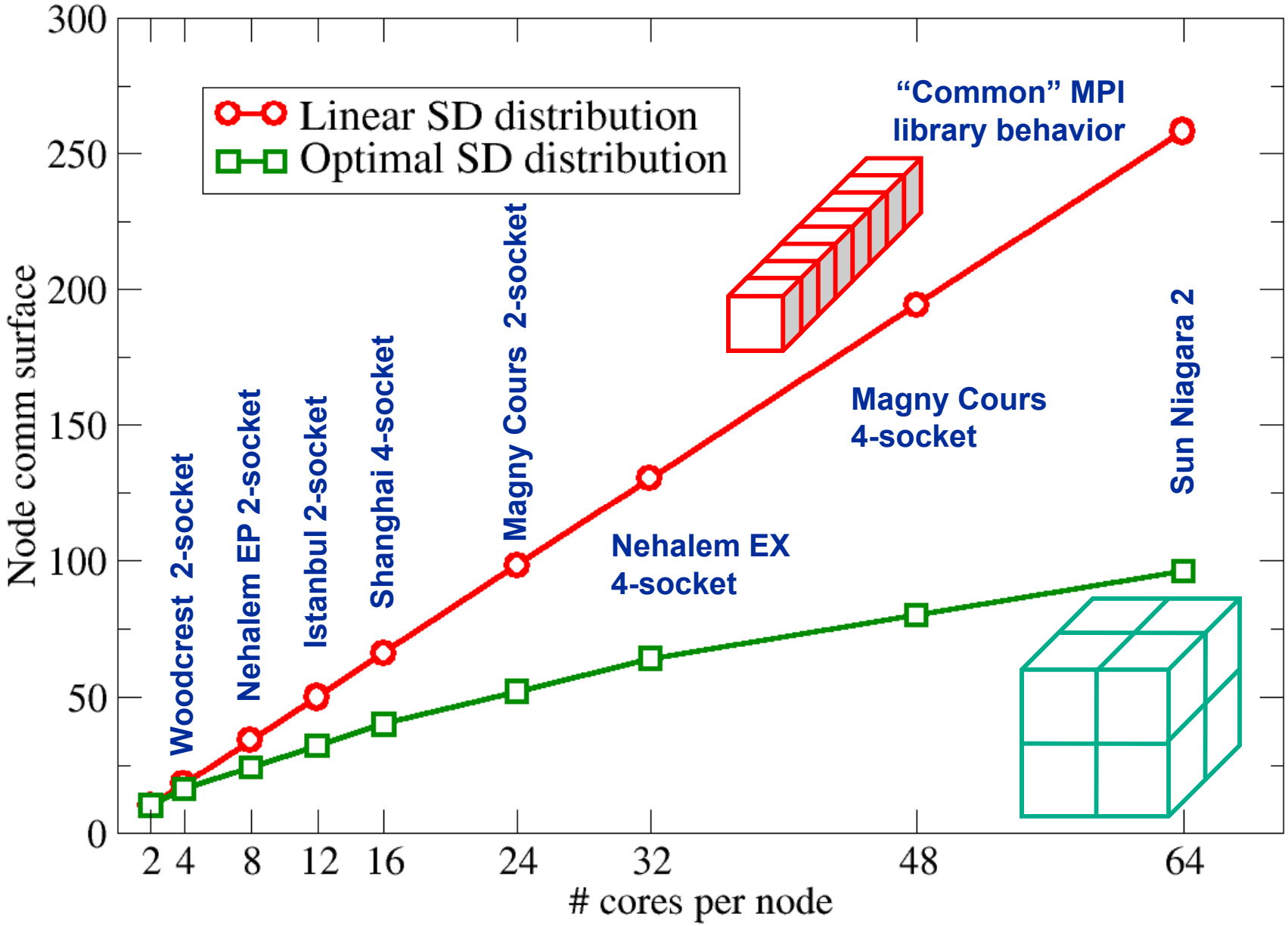


- **Goal: Reduce inter-node halo traffic**
- **Subdomains exchange halo with neighbors**
  - Populate a node's ranks with "maximum neighboring" subdomains
  - This minimizes a node's communication surface

- **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



For more details see hybrid part!

Section summary: What to take home

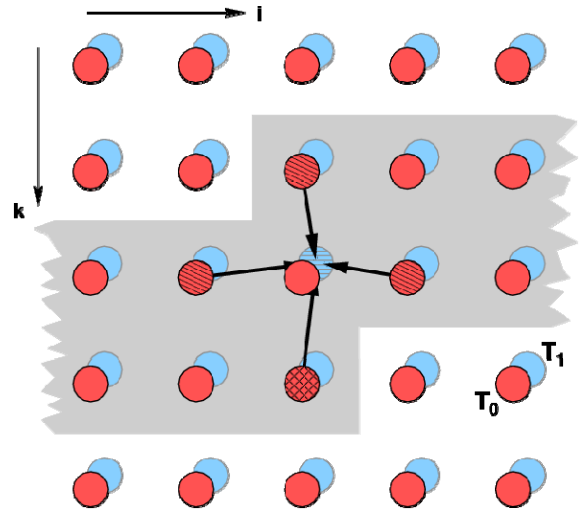
- **Bandwidth saturation is a reality, in cache and memory**
  - Use knowledge to choose the "right" number of threads/processes per node
  - You must know where those threads/processes should run
  - You must know the architectural requirements of your application
- **ccNUMA architecture must be considered 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

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
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

Automatic shared-memory parallelization: What can the compiler do for you?

Automatic parallelization for moderate processor counts is 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. (the latest one…)
    - 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`**

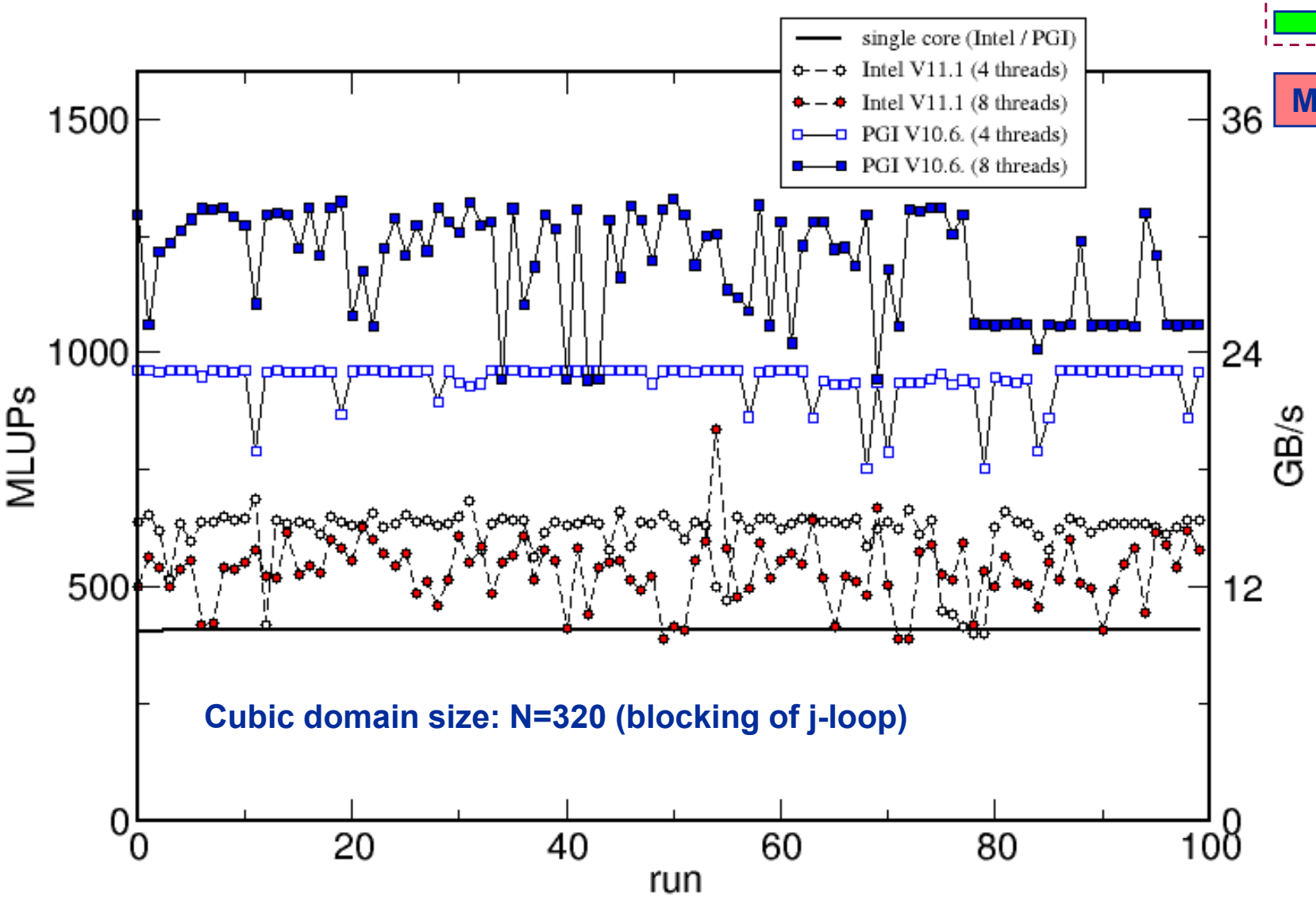
- **PGI compiler (V 10.6)**
  `pgf90 -tp nehalem-64 -fastsse -Mconcur -Minfo=par,vect`
  - Performs outer loop parallelization of k-loop
    ```
    139, Parallel code generated with block distribution if
    trip count is greater than or equal to 33
    ```
  - and vectorization of inner i-loop:
    ```
    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

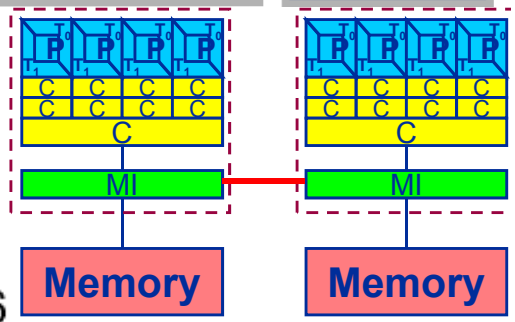
- **gfortran: No automatic parallelization feature so far (?!)**

Common Lore

Performance/Parallelization at the node level: Software does it

- **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?!

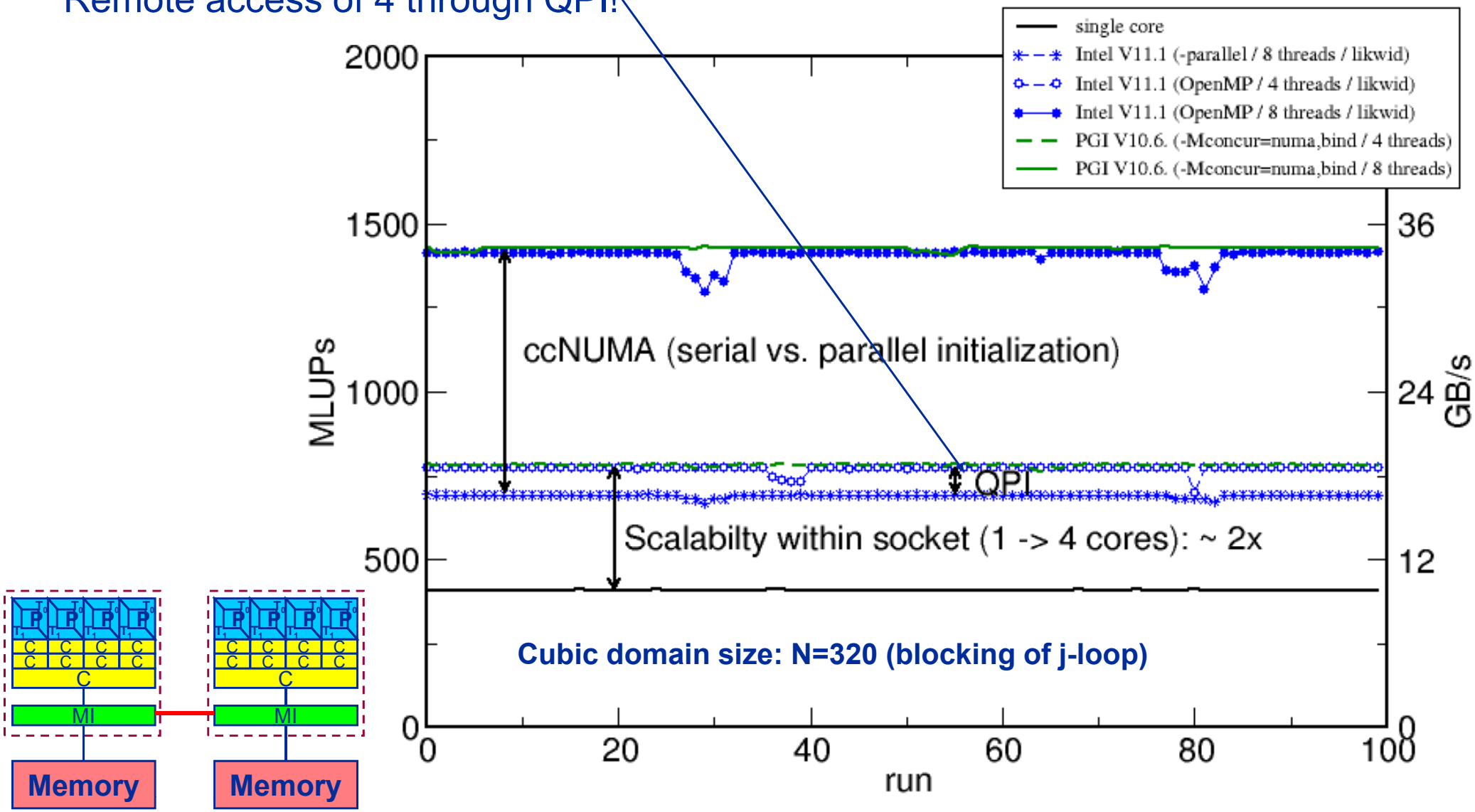
Controlling thread affinity / binding
Intel / PGI compilers

- **Intel compiler controls thread-core affinity via `KMP_AFFINITY` environment variable**
  - `KMP_AFFINITY="granularity=fine,compact,1,0"` 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, e.g. 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 about 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

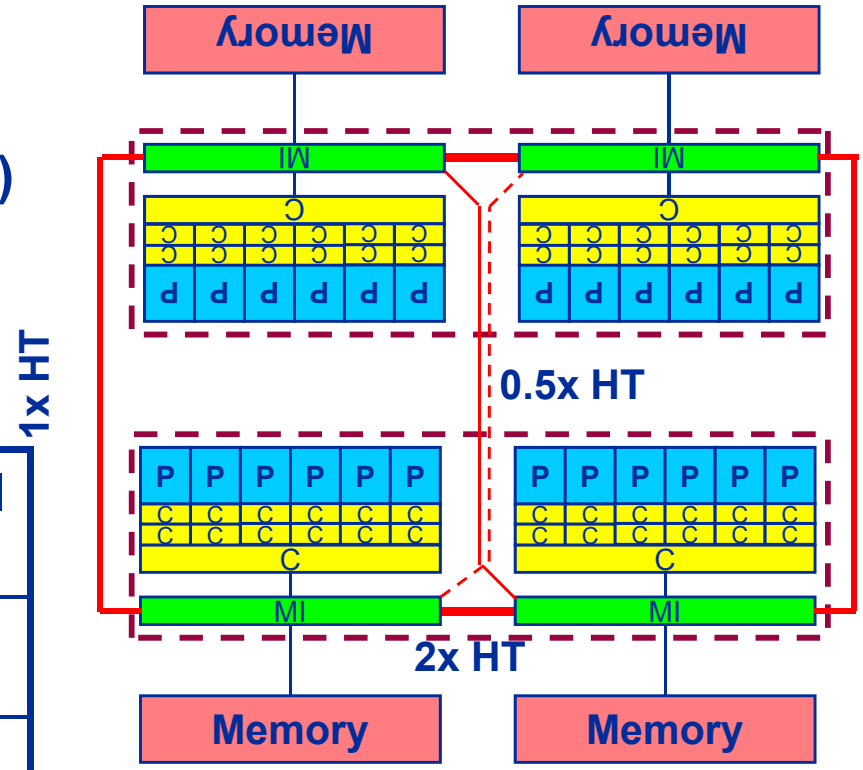
- 12-core Magny-Cours: A single socket holds two tightly HT-connected 6-core chips → 2-socket system has 4 data locality domains

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 increase complexity a bit, e.g. simple 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
```

A bit more complex 3D 7-point stencil update(„Gauss-Seidel“)

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

Common Lore

Performance/Parallelization at the node level: Software does it

- **State of the art compilers do not parallelize Gauß-Seidel iteration scheme:** `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+ 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**

optional

Advanced OpenMP: Eliminating recursion

Parallelizing a 3D Gauss-Seidel solver by pipeline parallel processing

The Gauss-Seidel algorithm in 3D

optional

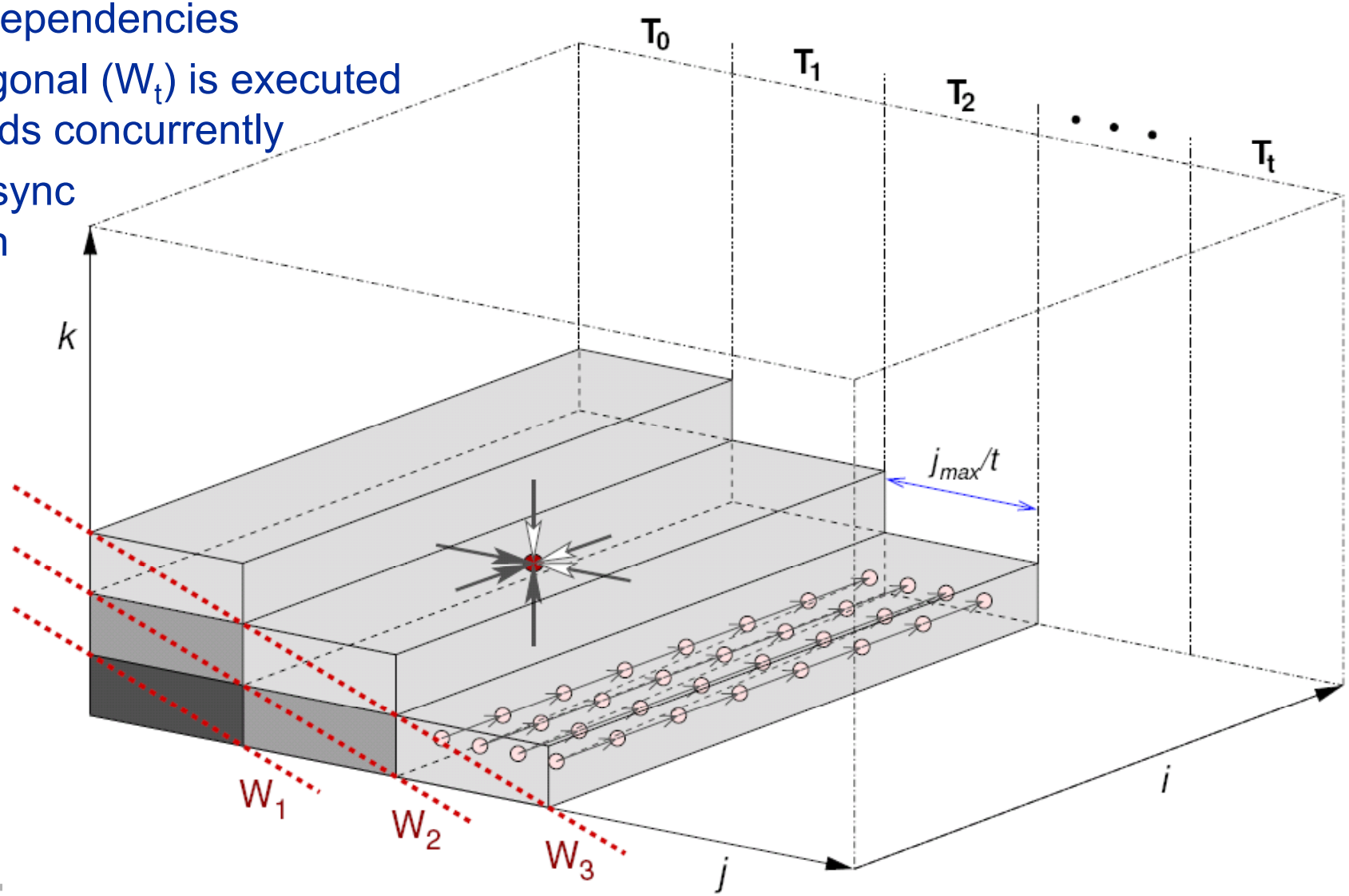
```
double precision, parameter :: osth=1/6.d0
do it=1,itmax    ! number of iterations (sweeps)
  ! not parallelizable right away
  do k=1,kmax
    do j=1,jmax
      do i=1,imax
        phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
                     + phi(i,j-1,k) + phi(i,j+1,k)
                    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

optional

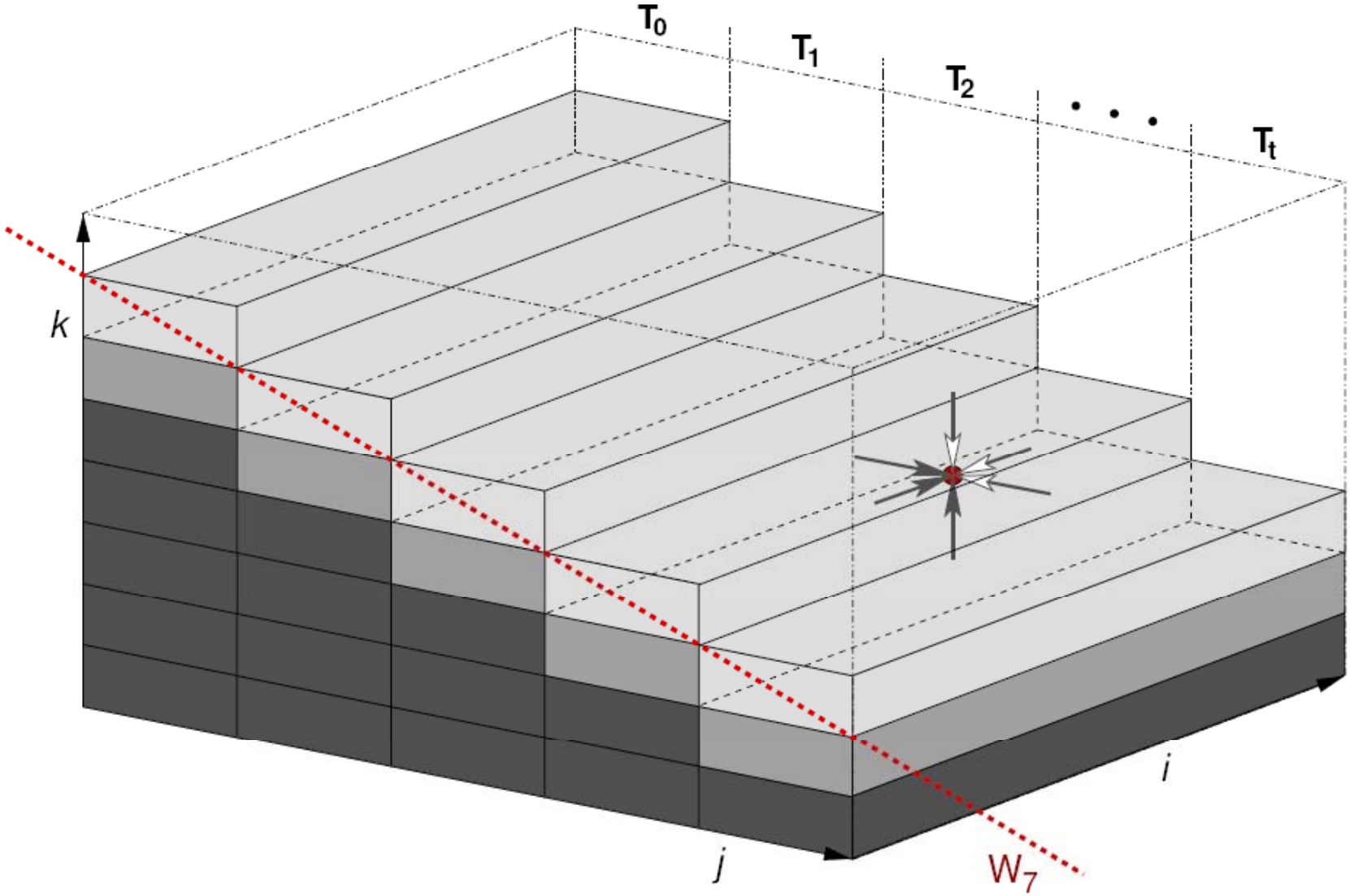
- **Pipeline parallel principle: Wind-up phase**
  - Parallelize middle j-loop and shift thread execution in k-direction to account for data dependencies
  - Each diagonal (W_t) is executed by t threads concurrently
  - Threads sync after each k-update



3D Gauss-Seidel parallelized

optional

- **Full pipeline: All threads execute**



3D Gauss-Seidel parallelized: The code

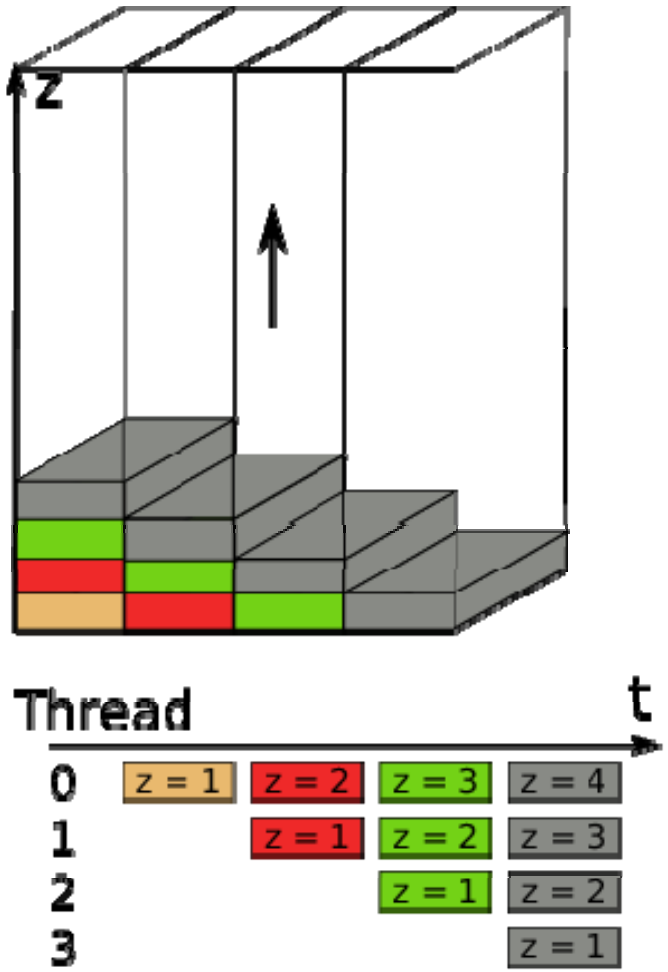
optional

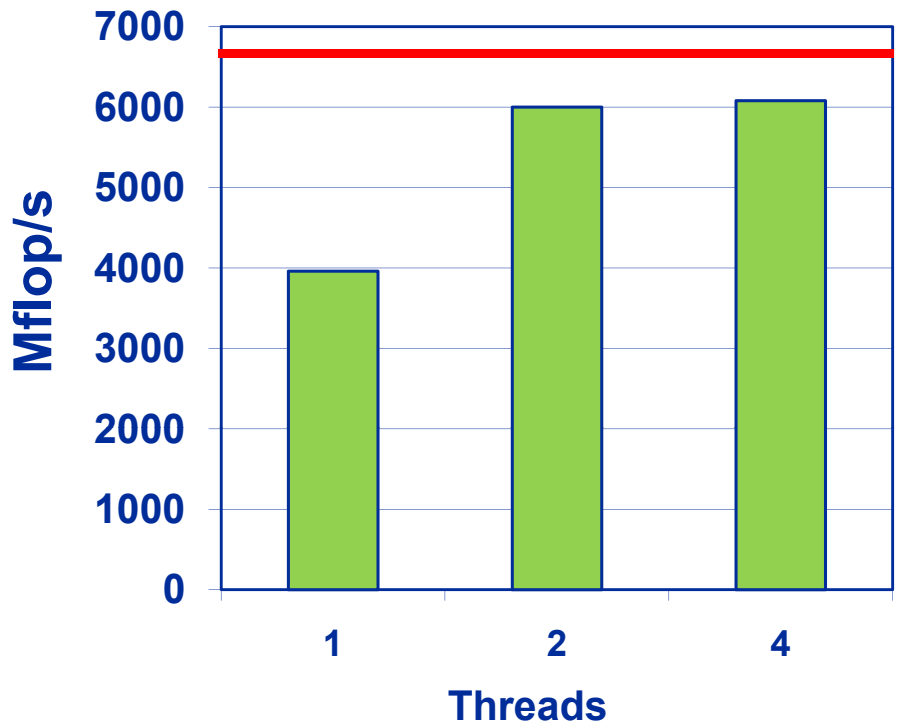
```
!$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
  threadID=OMP_GET_THREAD_NUM()
!$OMP SINGLE
  numThreads=OMP_GET_NUM_THREADS()
!$OMP END SINGLE
  jStart=jmax/numThreads*threadID
  jEnd=jStart+jmax/numThreads ! jmax is amultiple of numThreads
  do l=1,kmax+numThreads-1
    k=l-threadID
    if((k.ge.1).and.(k.le.kmax))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

Global OpenMP barrier for thread sync – better solutions exist! (see hybrid part)

3D Gauss-Seidel parallelized: Performance results

optional





Performance model: 6750 Mflop/s (based on 18 GB/s STREAM bandwidth)

Intel Core i7-2600 ("Sandy Bridge")

3.4 GHz; 4 cores

Optimized Gauss-Seidel kernel! See:

J. Treibig, G. Wellein and G. Hager: *Efficient multicore-aware parallelization strategies for iterative stencil computations*. Journal of Computational Science **2** (2011) 130-137. DOI: 10.1016/j.jocs.2011.01.010, Preprint: arXiv:1004.1741

Parallel 3D Gauss-Seidel

optional

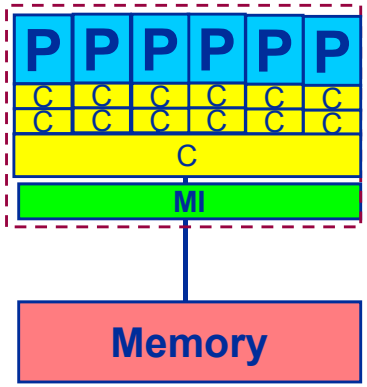
- **Gauss-Seidel can also be parallelized using a red-black scheme**

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

Wavefront-parallel temporal blocking for stencil algorithms

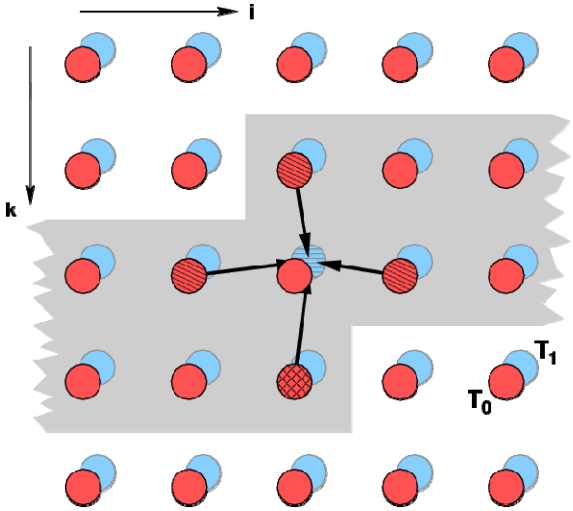
One example for truly “multicore-aware” programming

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



Simple 3D Jacobi stencil update (sweep):

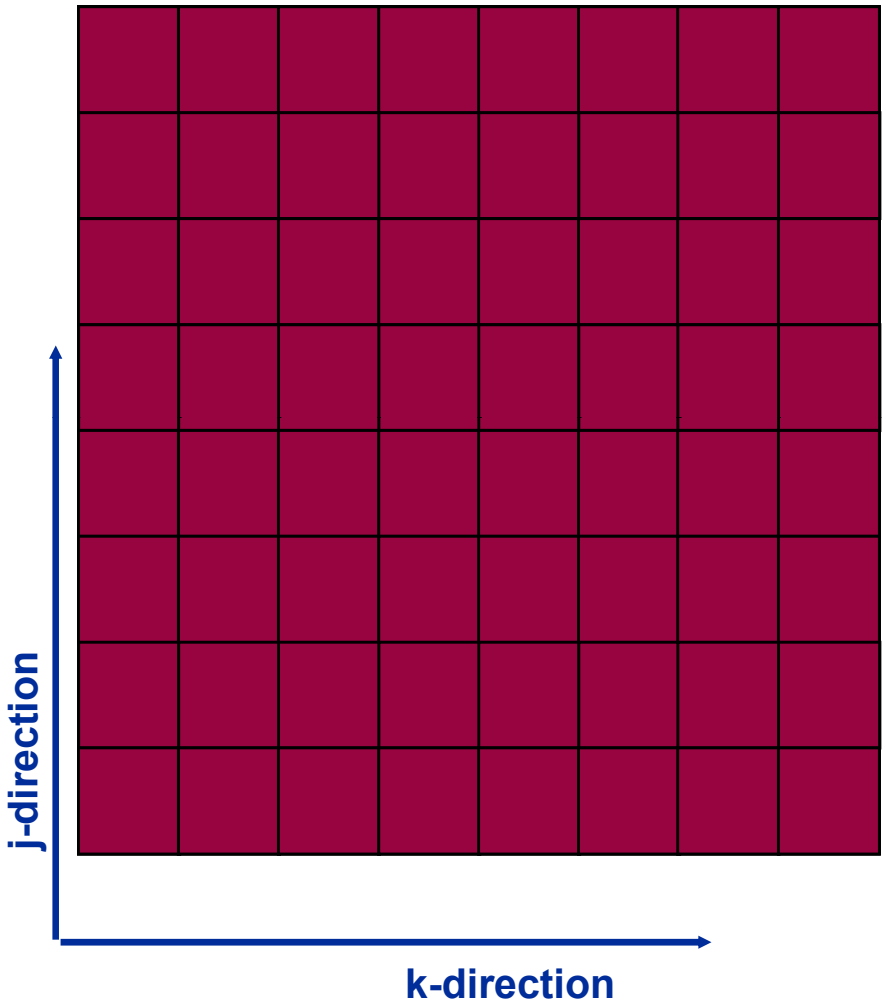
```
do k = 1 , Nk
  do j = 1 , Nj
      do i = 1 , Ni
          y(i,j,k) = a*x(i,j,k) + b*
                     (x(i-1,j,k)+x(i+1,j,k)+
                      x(i,j-1,k)+x(i,j+1,k)+
                      x(i,j,k-1)+x(i,j,k+1))
     enddo
  enddo
enddo
```

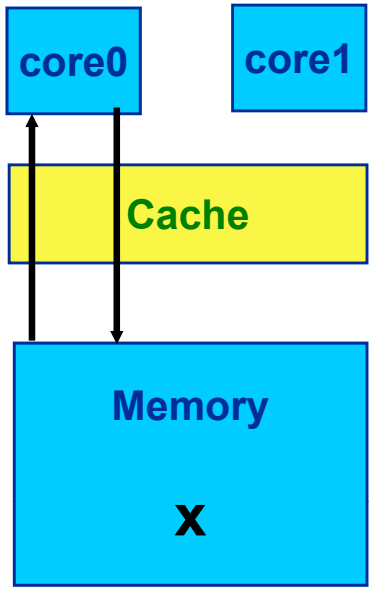


Performance Metric: Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs: 8 FLOP/LUP * MLUPs

Multicore awareness

Standard sequential implementation

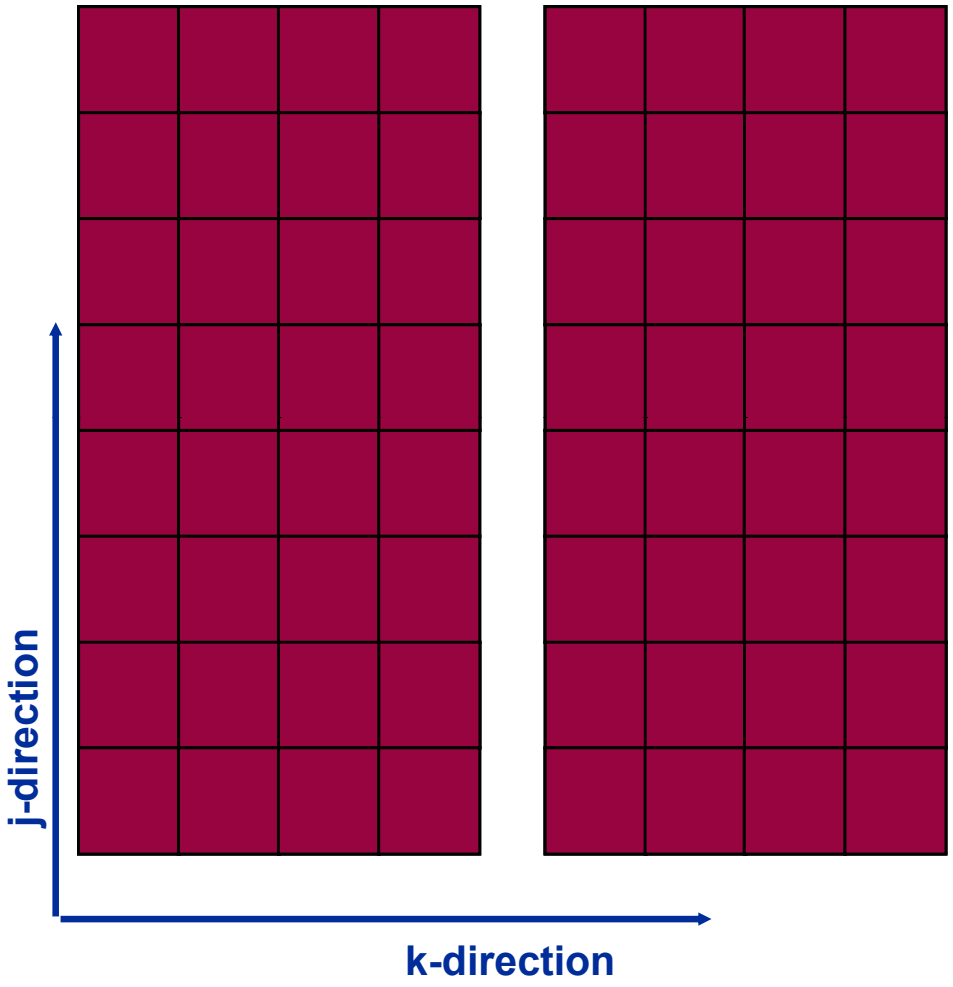


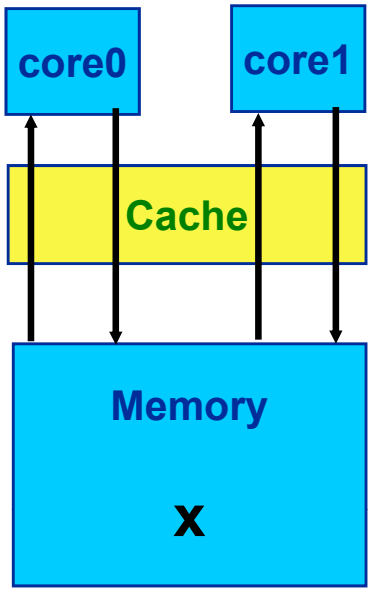



```
do t=1,tMax

   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!

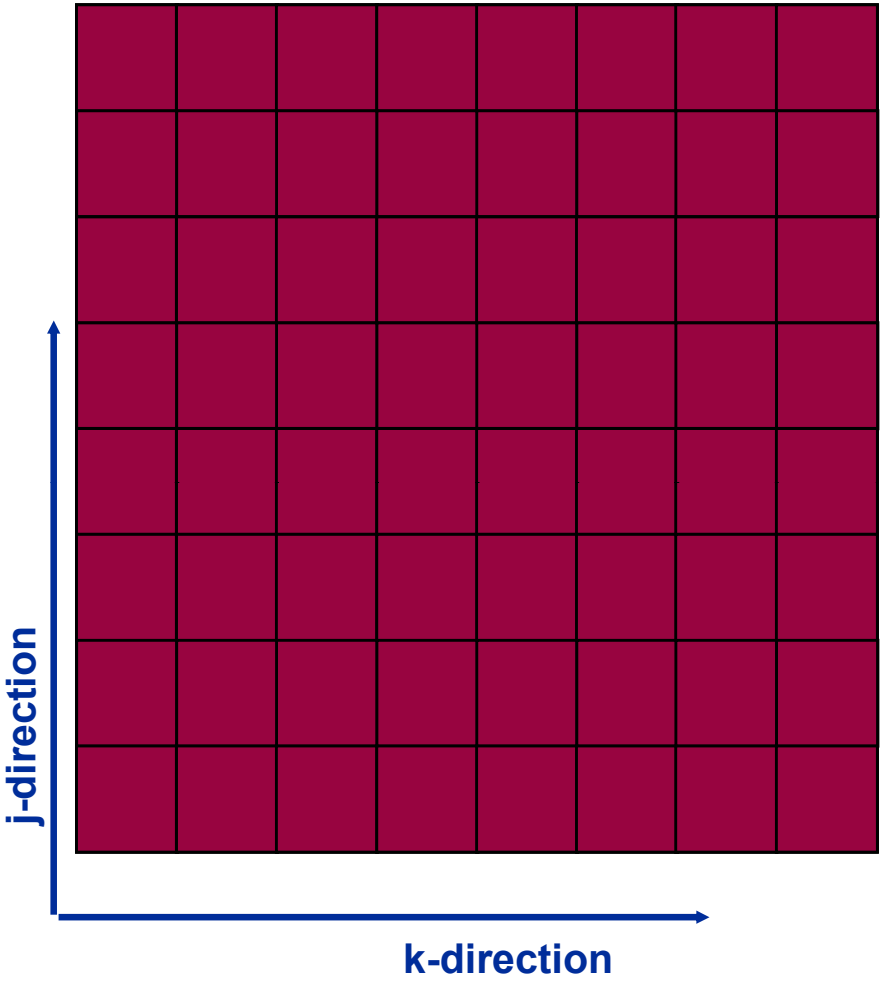


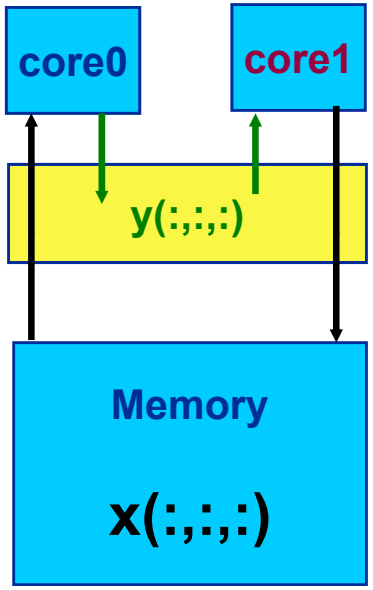



```
do t=1,tMax
!$OMP PARALLEL DO private(…)
  do k=1,N
     do j=1,N
        do i=1,N
           y(i,j,k) = …
        enddo
     enddo
  enddo
!$OMP END PARALLEL DO
enddo
```

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

↓

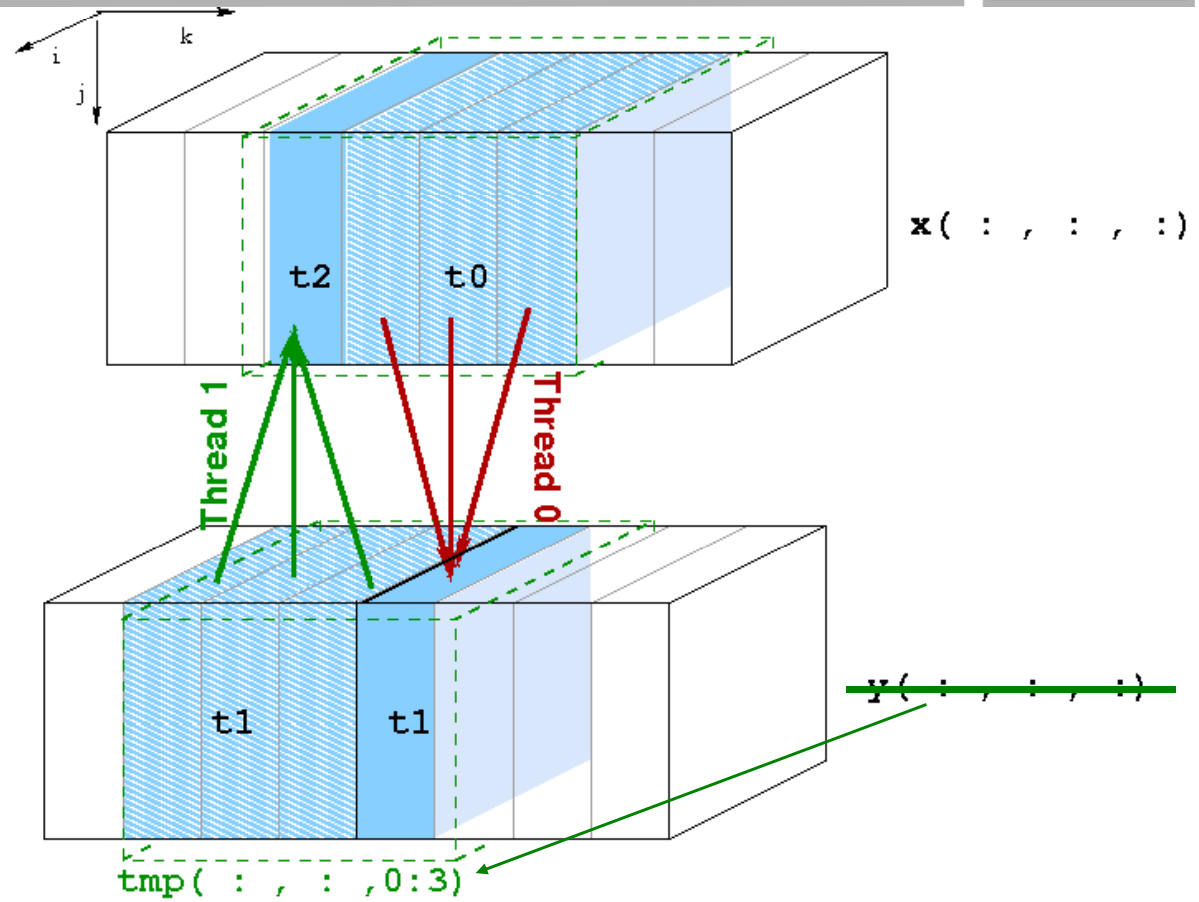
Save main memory data transfers for `y(:,:,:)` !

↓

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)

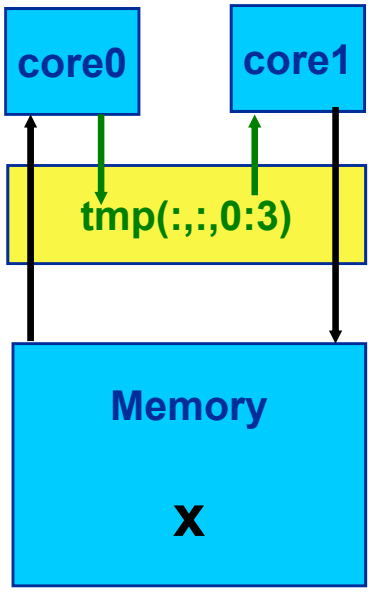
Thread 0: `x(:,:,k-1:k+1)`$_t$ → *`tmp(:,:,mod(k,4))`*

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

Speed-Up vs. baseline: $\mathbf{S_W = 2*T_0/T_{2LUP}}$
$\mathbf{= 2 / (1 + B_M/B_C)}$

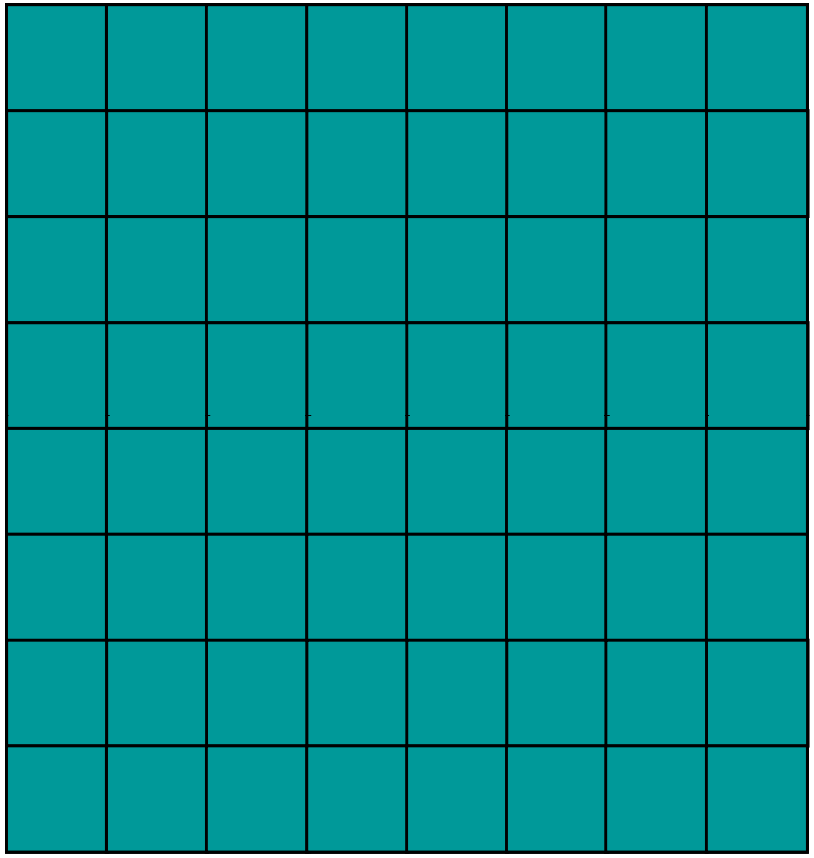
B_C and B_M are measured in saturation runs:

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

Nehalem : $B_M/B_C = 1/4$ → $S_W = 1.6$

Jacobi solver

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

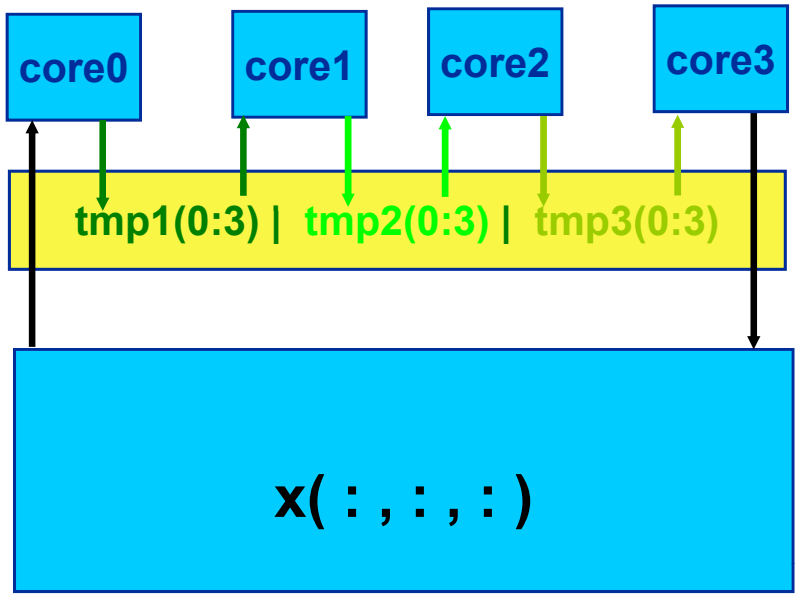


Running **`tb`** wavefronts requires **`tb-1`** temporary arrays **`tmp`** to be held in cache!

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

Extensive use of cache bandwidth!

1 x 4 distribution



Jacobi solver

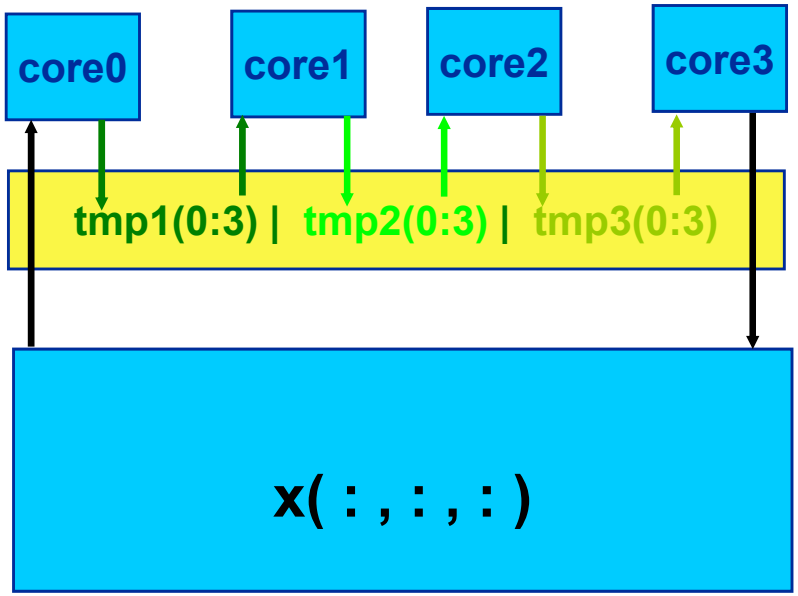
WF parallelization: New choices on native quad-cores

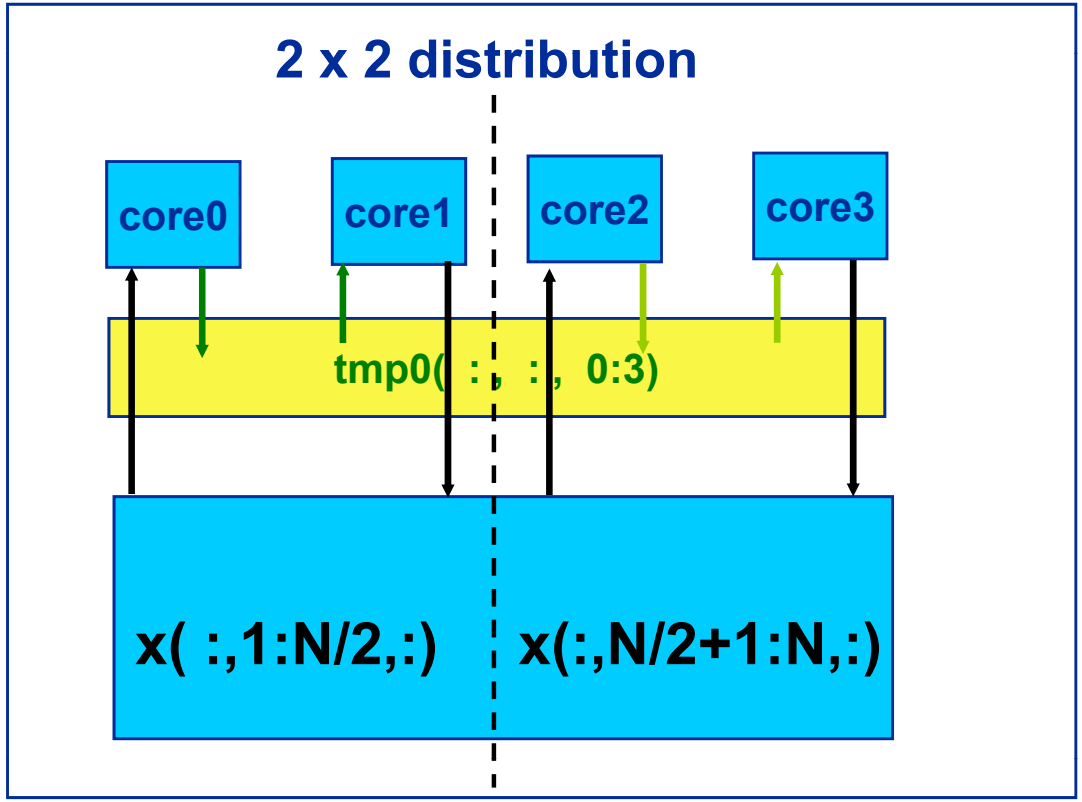
Thread 0: `x(:,:,k-1:k+1)`$_t$ → *tmp1(mod(k,4))*

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))*

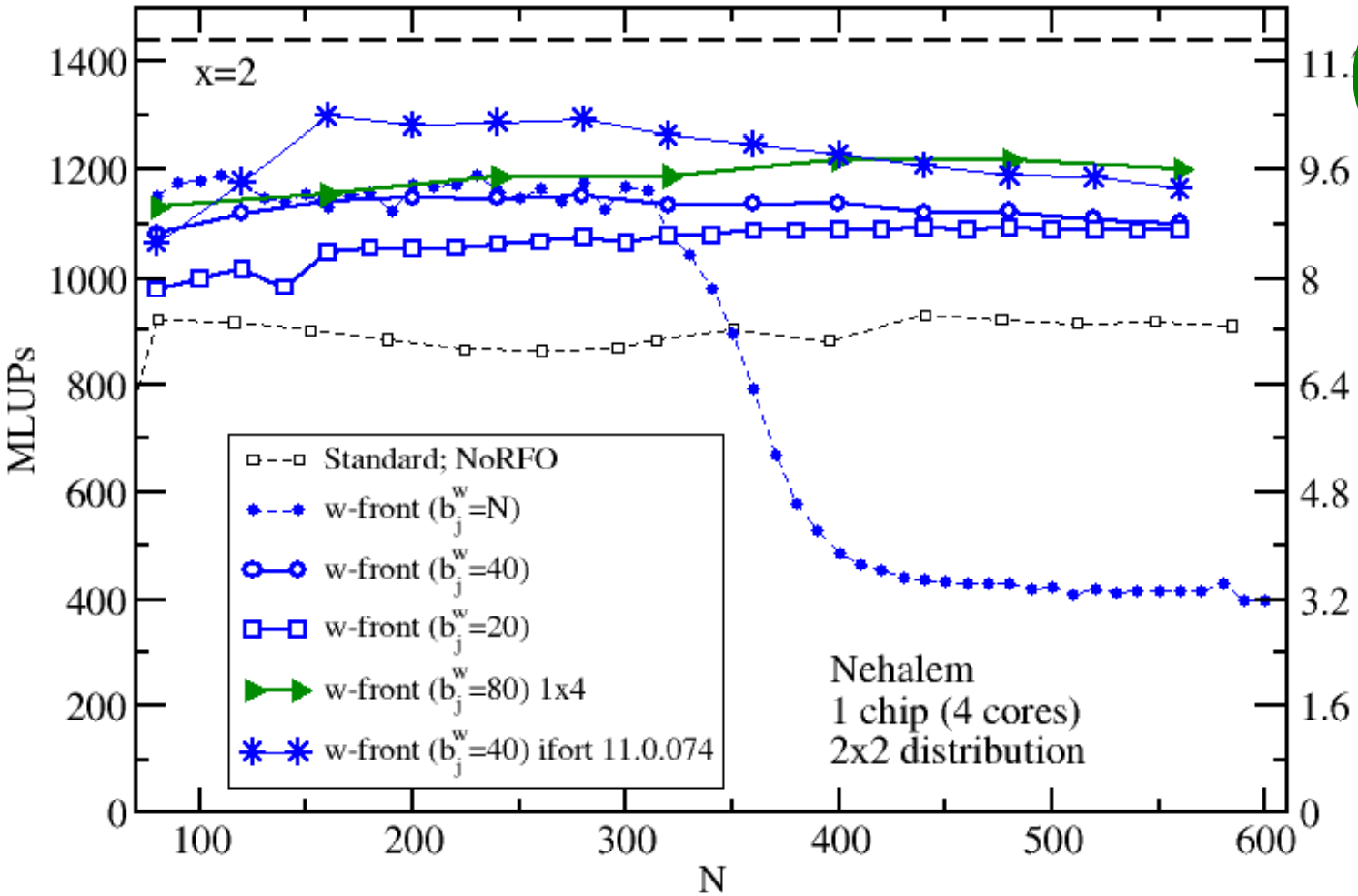
Thread 3: *tmp3(mod(k-7,4):mod(k-5,4))* → `x(:,:,k-6)`$_{t+4}$

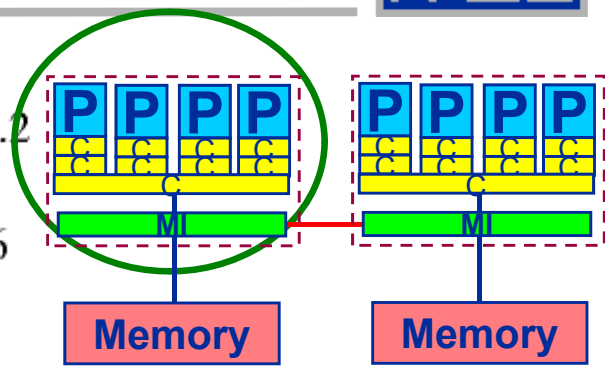




Jacobi solver

Wavefront parallelization: L3 group Nehalem





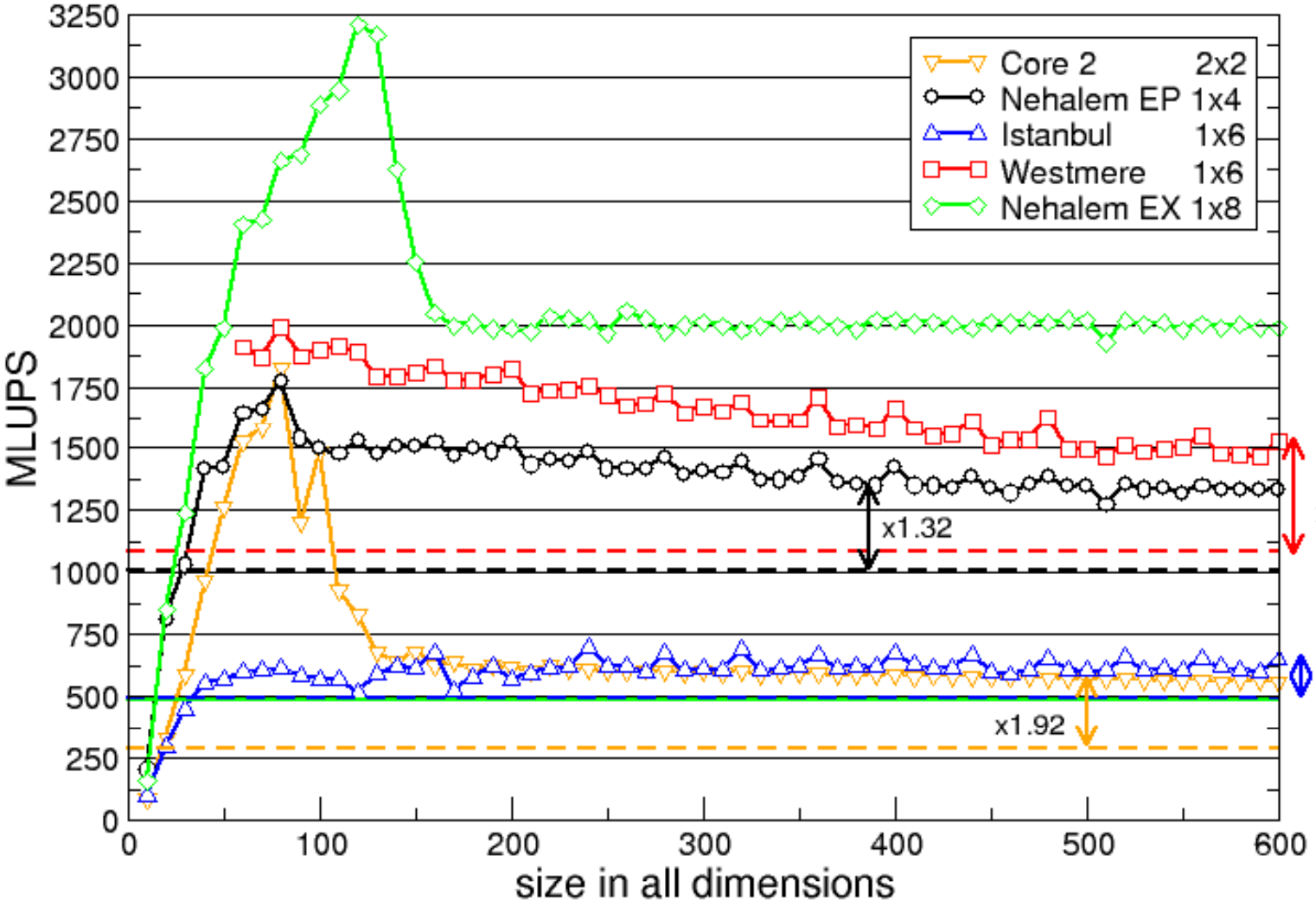
400^3 bj=40	MLUPs
1 x 2	786
2 x 2	1230
1 x 4	1254

Performance model indicates some potential gain → new compiler tested.

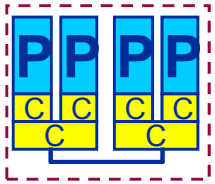
Only marginal benefit when using 4 wavefronts → A single copy stream does not achieve full bandwidth

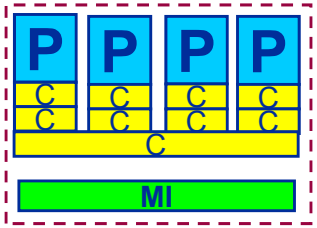
Multicore-aware parallelization

Wavefront – Jacobi on state-of-the art multicores

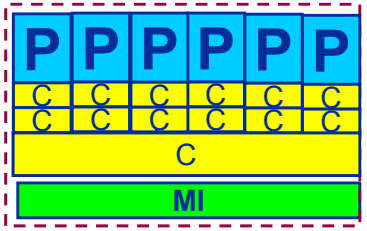


B_{olc} ~ 10

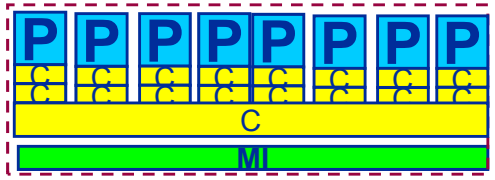




B_{olc} ~ 2-3



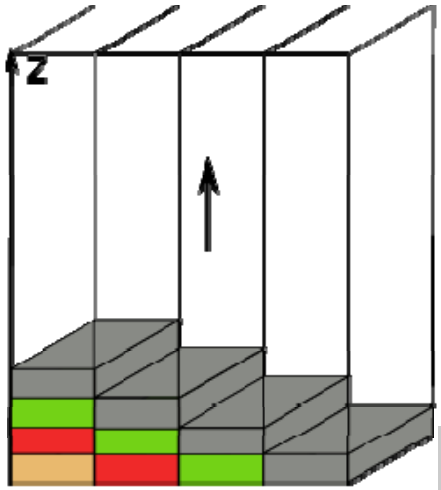
B_{olc} ~ 10



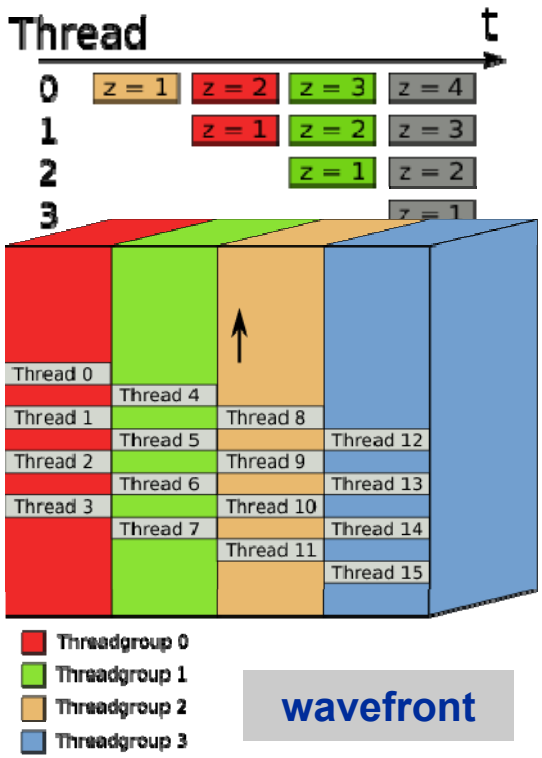
Compare against optimal baseline!

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

Multicore-specific features – Room for new ideas:
Wavefront parallelization of Gauss-Seidel solver

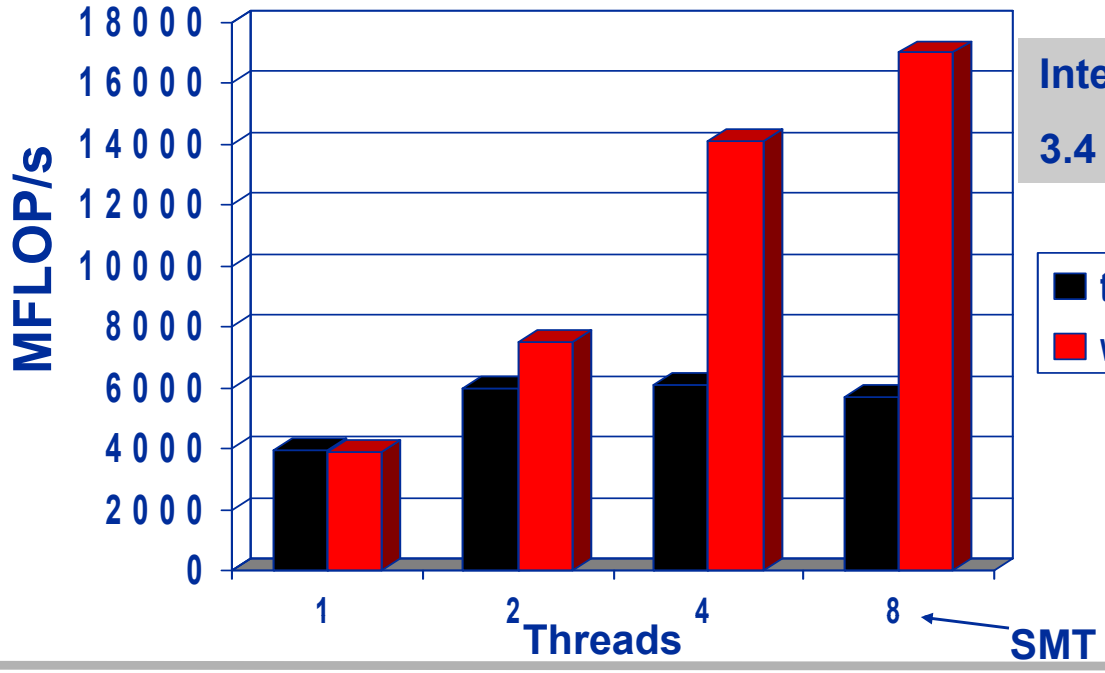


threaded



wavefront

- **Shared caches in Multi-Core processors**
  - Fast thread synchronization
  - Fast access to shared data structures
- **FD discretization of 3D Laplace equation:**
  - Parallel lexicographical Gauß-Seidel using pipeline approach ("threaded")
  - Combine threaded approach with wavefront technique ("wavefront")



Intel Core i7-2600

3.4 GHz; 4 cores

Section summary: What to take home

- **Auto-parallelization may work for simple problems, but it won't make us jobless in the near future**
  - There are enough loop structures the compiler does not understand

- **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)

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
- **Online demo: likwid tools (1)**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Bandwidth saturation effects
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI
- **Case studies for shared memory**
  - Automatic parallelization
  - Pipeline parallel processing for Gauß-Seidel solver
  - Wavefront temporal blocking of stencil solver
- **Summary: Node-level issues**

Summary & Conclusions on node-level issues

- **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
  - Lowlevel benchmarking: likwid-bench
- **Try to leverage *the* new architectural feature of modern multicore chips**
  - Shared caches!

Tutorial outline (2)

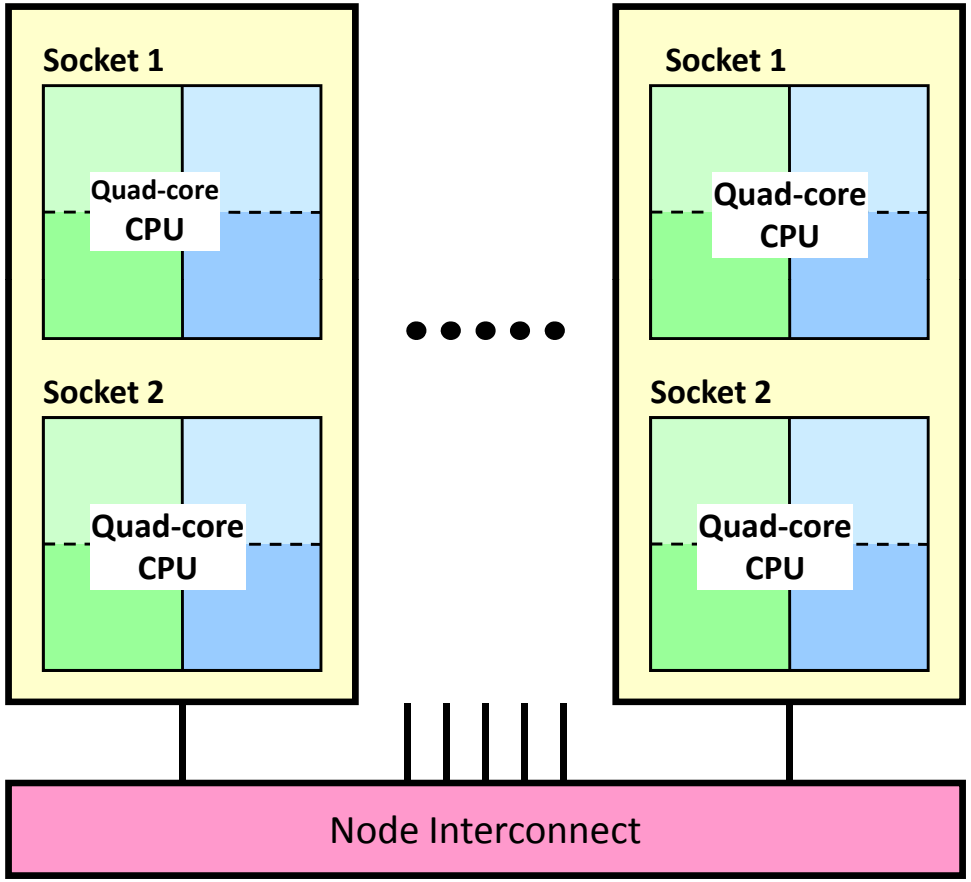
- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
  - Practical “How-tos” for hybrid
- **Online demo: likwid tools (2)**
  - Advanced pinning
  - Making bandwidth maps
  - Using likwid-perfctr to find NUMA problems and load imbalance
  - likwid-perfctr internals
  - likwid-perfscope
- **Case studies for hybrid MPI/OpenMP**
  - Overlap for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - PIR3D – hybridization of a full scale CFD code
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

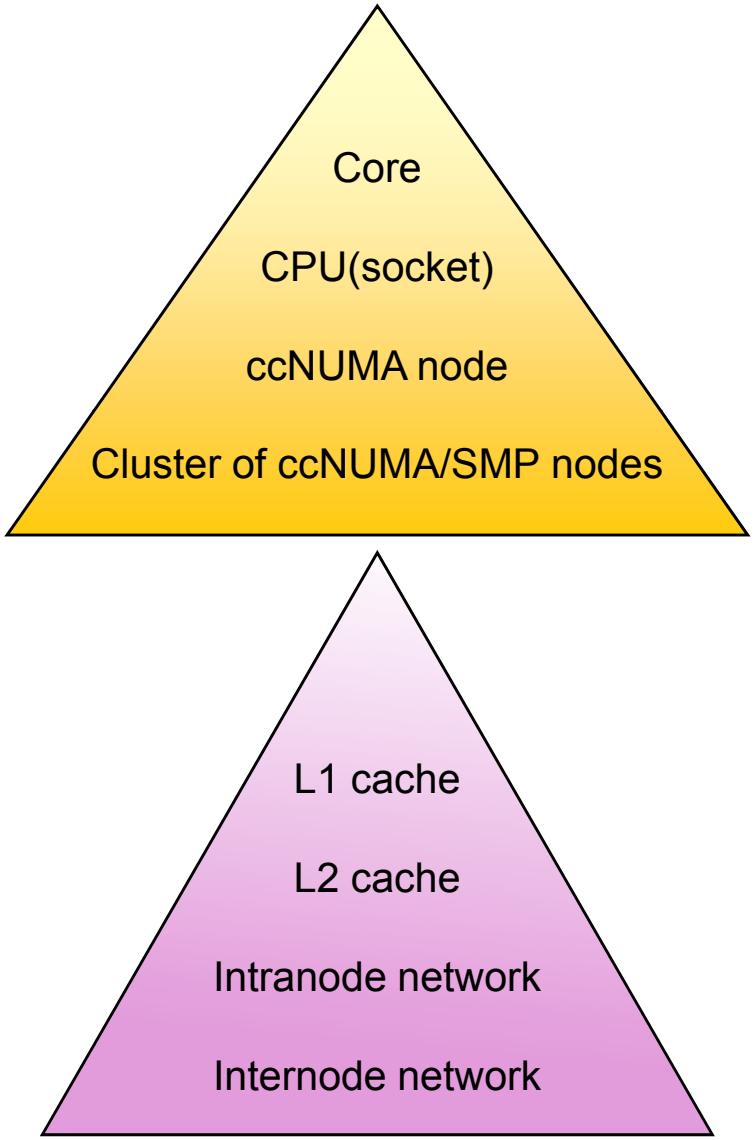
Tutorial outline

- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
  - Practical "How-tos" for hybrid
- **Online demo: likwid tools (2)**
  - Advanced pinning
  - Making bandwidth maps
  - Using likwid-perfctr to find NUMA problems and load imbalance
  - likwid-perfctr internals
  - likwid-perfscope
- **Case studies for hybrid MPI/OpenMP**
  - Overlap for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - PIR3D – hybridization of a full scale CFD code
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

Clusters of Multicore Nodes

- **Can hierarchical hardware benefit from a hierarchical programming model?**

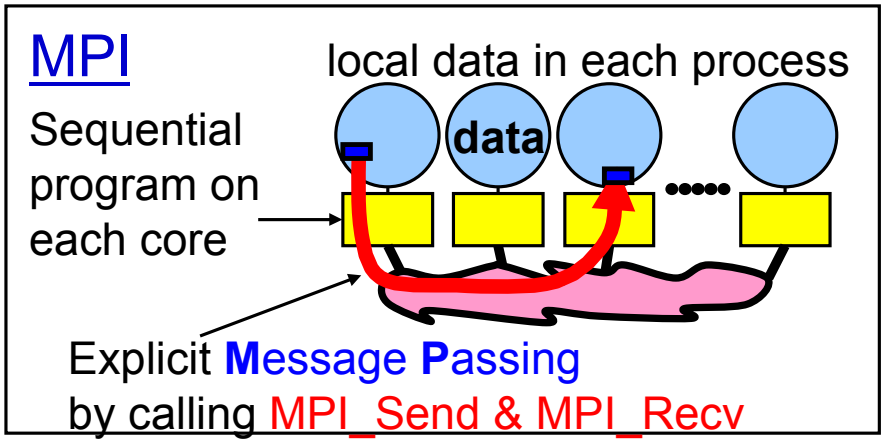


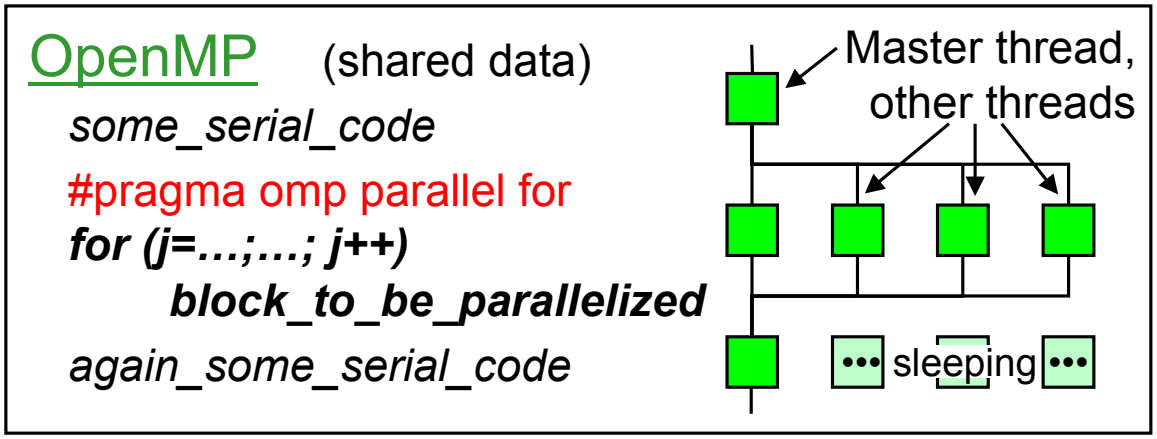


MPI vs. OpenMP

Programming Models for SMP Clusters

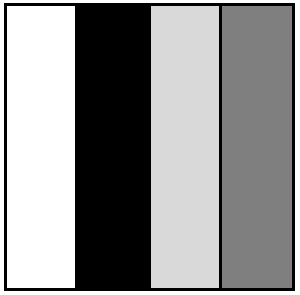
- **Pure MPI (one process on each core)**
- **Hybrid MPI+OpenMP**
  - Shared memory OpenMP
  - Distributed memory MPI
- **Other: Virtual shared memory systems, PGAS, HPF, …**
- **Often hybrid programming (MPI+OpenMP) slower than pure MPI**
  - Why?





MPI Parallelization of Jacobi Solver

- Initialize MPI
- Domain decomposition
- Compute local data
- Communicate shared data



1D partitioning

```
...
CALL MPI_INIT(ierr)
! Compute number of procs and myrank
CALL MPI_COMM_SIZE(comm, p, ierr)
CALL MPI_COMM_RANK(comm, myrank, ierr)
!Main Loop
DO WHILE(.NOT.converged)
   ! compute
   DO j=1, m_local
      DO i=1, n
         BLOC(i,j)=0.25*(ALOC(i-1,j)+
                         ALOC(i+1,j)+
                         ALOC(i,j-1)+
                         ALOC(i,j+1))
      END DO
   END DO
 ! Communicate
      CALL MPI_SENDRECV(BLOC(1,1),n,
        MPI_REAL, left, tag, ALOC(1,0),n,
        MPI_REAL, left, tag, comm,
        status, ierr)
```

OpenMP Parallelization of Jacobi Solver

```
!Main Loop
DO WHILE(.NOT.converged)
   ! Compute
!$OMP PARALLEL SHARED(A,B) PRIVATE(J,I)
!$OMP DO
   DO j=1, m
      DO i=1, n
         B(i,j)=0.25*(A(i-1,j)+
                      A(i+1,j)+
                      A(i,j-1)+
                      A(i,j+1))
      END DO
   END DO
!$OMP END DO
!$OMP DO
   DO j=1, m
      DO i=1, n
         A(i,j) = B(i,j)
      END DO
   END DO
!$OMP END DO
!$OMP END PARALLEL
...
```

implicit removable barrier

Comparison of MPI and OpenMP

MPI

- **Memory Model**
  - Data private by default
  - Data accessed by multiple processes needs to be explicitly communicated
- **Program Execution**
  - Parallel execution starts with MPI_Init, continues until MPI_Finalize
- **Parallelization Approach**
  - Typicall coarse grained, based on domain decomposition
  - Explicitly programmed by user
  - All-or-nothing approach
- **Scalability possible across the whole cluster**
- **Performance: Manual parallelization allows high optimization**

OpenMP

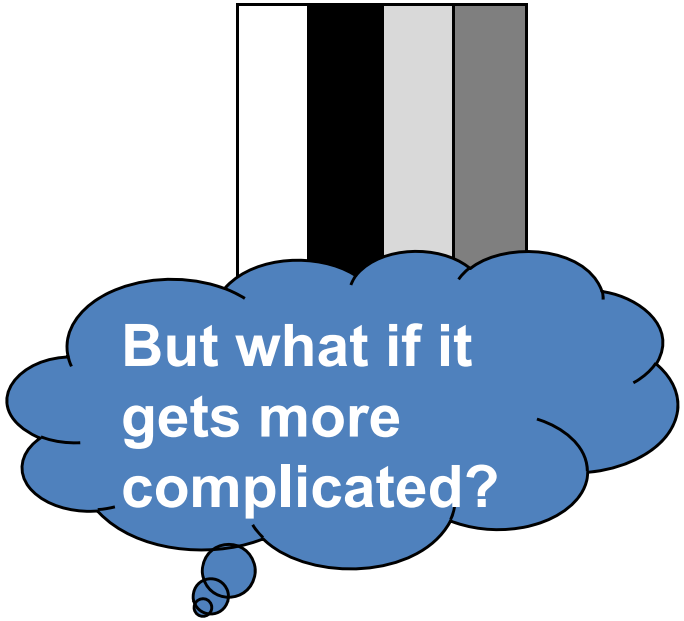
- **Memory Model**
  - Data shared by default
  - Access to shared data requires explicit synchronization
  - Private data needs to be explicitly declared
- **Program Execution**
  - Fork-Join Model
- **Parallelization Approach:**
  - Typically fine grained on loop level
  - Based on compiler directives
  - Incremental approach
- **Scalability limited to one shared memory node**
- **Performance dependent on compiler quality**

Combining MPI and OpenMP: Jacobi Solver

- **Simple Jacobi Solver Example**
  - MPI parallelization in j dimension
  - OpenMP on i-loops
- **All calls to MPI outside of parallel regions**

```
!Main Loop
DO WHILE(.NOT.converged)
   ! compute
   DO j=1, m_loc
!$OMP PARALLEL DO
      DO i=1, n
         BLOC(i,j)=0.25*(ALOC(i-1,j)+
                         ALOC(i+1,j)+
                         ALOC(i,j-1)+
                         ALOC(i,j+1))
      END DO
!$OMP END PARALLEL DO
   END DO
   DO j=1, m
!$OMP PARALLEL DO
      DO i=1, n
         ALOC(i,j) = BLOC(i,j)
      END DO
!$OMP END PARALLEL DO
   END DO
   CALL MPI_SENDRECV (ALOC,…
   CALL MPI_SENDRECV (BLOC,…
...
```

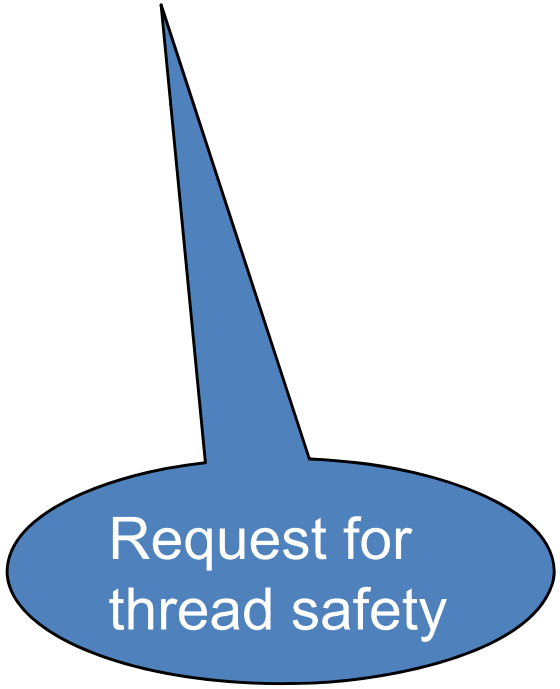
local length might be small for many MPI procs



Support of Hybrid Programming

MPI

- **MPI-2:**
  - `MPI_Init_Thread`



OpenMP

- **API only for one execution unit, which is one MPI process**
- **For example: No means to specify the total number of threads across several MPI processes.**

Thread safety quality of MPI libraries

MPI2 MPI_Init_thread

Syntax:

```
call MPI_Init_thread(                                  irequired,     iprovided, ierr)
int  MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
```

Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only main thread will make MPI calls (calls are ’ ’ funneled" to main thread). Default
MPI_THREAD_SERIALIZED	Process may be multi-threaded, any thread can make MPI calls, but threads cannot execute MPI calls concurrently (all MPI calls must be ’ ’ serialized").
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, no restrictions.

If supported, the call will return provided = required.
Otherwise, the highest supported level will be provided.

Funneling through OMP Master

- **Fortran**

```
include 'mpif.h'
program hybmas

call mpi_init_thread(MPI_THREAD_FUNNELED,
                     ...)

!$OMP parallel

   !$OMP barrier
   !$OMP master

     call MPI_<whatever>(…,ierr)
   !$OMP end master

   !$OMP barrier

!$OMP end parallel
end
```

$OMP master does not have implicit barrier

- **C**

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;
 ierr = MPI_Init_thread (...,
        MPI_THREAD_FUNNELED,...);
#pragma omp parallel
{
   #pragma omp barrier
   #pragma omp master
   {
     ierr=MPI_<Whatever>(…);
   }

   #pragma omp barrier

}
}
```

Overlapping Communication and Work

- **Fortran**

```
include 'mpi.h'
program hybover

call mpi_init_thread(MPI_THREAD_FUNNELED,
                     ...)

!$OMP parallel
   if (ithread .eq. 0) then
      call MPI_<whatever>(…,ierr)
   else
      <work>
   endif

!$OMP end parallel
end
```

- **C**

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, I;
 ierr=MPI_Init_thread(...,
         MPI_THREAD_FUNNELED,...);

#pragma omp parallel
{
   if (thread == 0){
      ierr=MPI_<Whatever>(…);
   }
   else {
      <work>
   }

}
}
```

Funneling through OMP SINGLE

- **Fortran**

```
include 'mpif.h'
program hybsing
call
mpi_init_thread(MPI_THREAD_FUNNELED,
                ...)
!$OMP parallel

   !$OMP barrier
   !$OMP single

     call MPI_<whatever>(…,ierr)
   !$OMP end single

   !!!$OMP barrier

!$OMP end parallel
end
```

- **C**

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;
mpi_init_thread(…,
          MPI_THREAD_FUNNELED,...)
#pragma omp parallel
{
   #pragma omp barrier
   #pragma omp single
   {
     ierr=MPI_<Whatever>(…)
   }

   //#pragma omp barrier

}
```

$OMP single has an implicit barrier

Thread-rank Communication

```
call mpi_init_thread( … MPI_THREAD_MULTIPLE, iprovided,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,  irank, ierr)
call mpi_comm_size(MPI_COMM_WORLD, nranks, ierr)

!$OMP parallel private(i, ithread, nthreads)

  nthreads = OMP_GET_NUM_THREADS()
  ithread  = OMP_GET_THREAD_NUM()
  call pwork(ithread, irank, nthreads, nranks…)
  if(irank == 0) then
   call mpi_send(ithread,1,MPI_INTEGER, 1, ithread,MPI_COMM_WORLD, ierr)
  else
   call mpi_recv(      j,1,MPI_INTEGER, 0, ithread,MPI_COMM_WORLD,
                                                        istatus,ierr)
   print*, "Yep, this is ",irank," thread ", ithread,
           " I received from ", j
  endif

!$OMP END PARALLEL
end
```

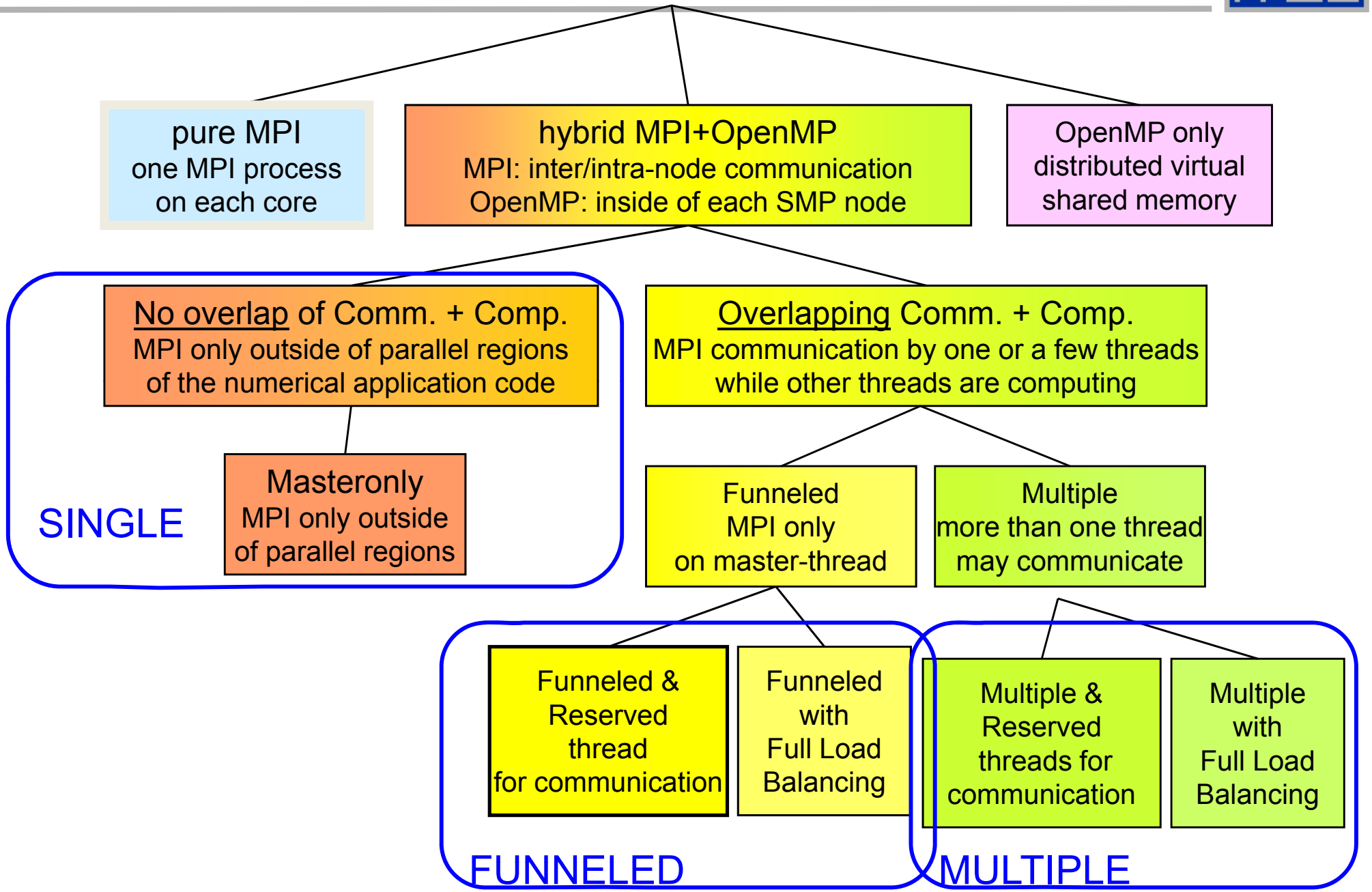
Communicate between ranks.

Threads use tags to differentiate.

Strategies/options for Combining MPI with OpenMP

Topology and Mapping Problems
Potential Opportunities

Different Strategies to Combine MPI and OpenMP



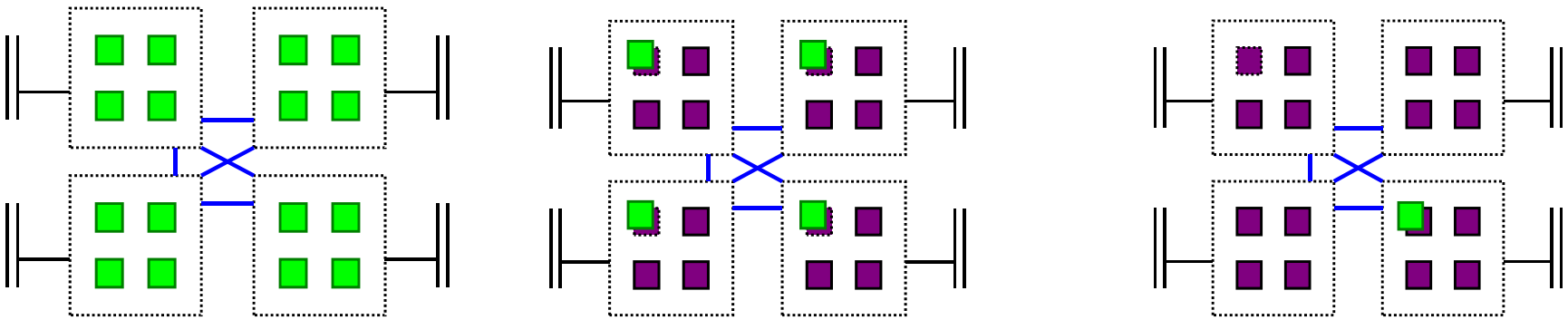
Modes of Hybrid Operation

Pure MPI …… Mixed ……. Fully Hybrid

16 MPI Tasks

4 MPI Tasks
4Threads/Task

1 MPI Task
16 Threads/Task



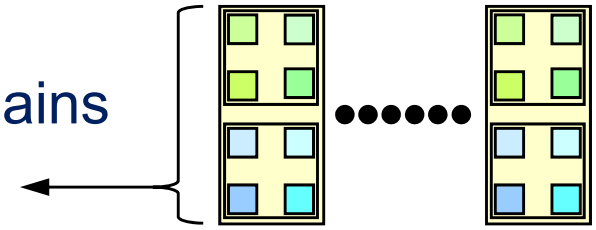
Master Thread of MPI Task
MPI Task on Core
Master Thread of MPI Task
Slave Thread of MPI Task

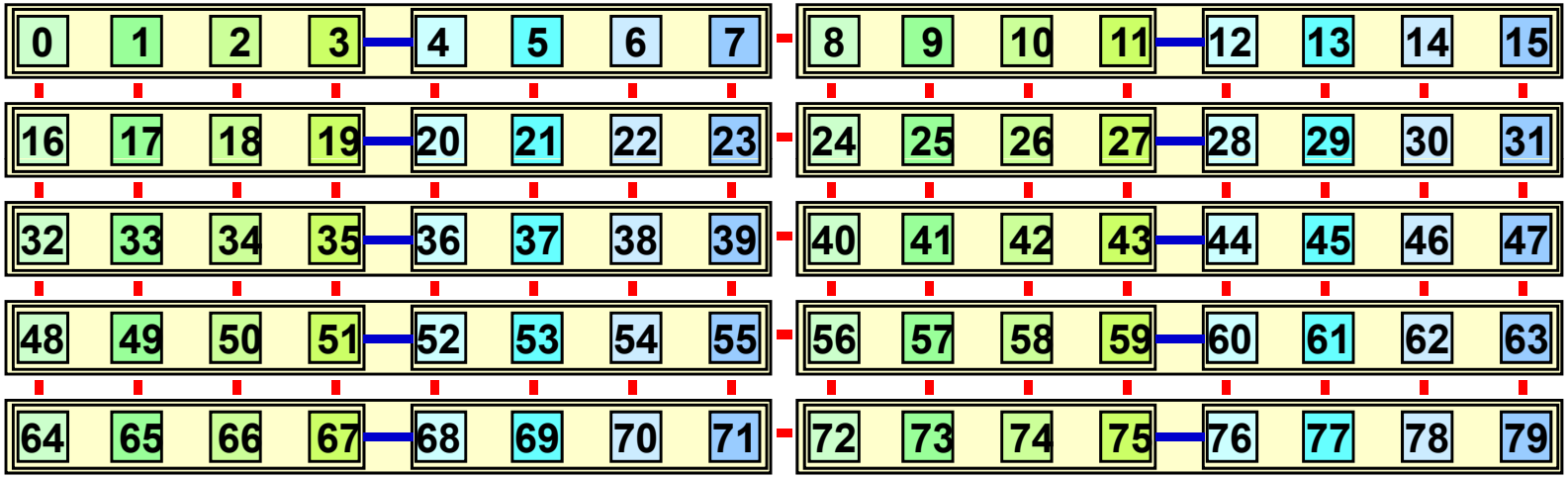
The Topology Problem with

pure MPI
one MPI process
on each core

Application example on 80 cores:

- Cartesian application with 5 x 16 = 80 sub-domains
- On system with 10 x dual socket x quad-core





\+ 17 x inter-node connections per node

− 1 x inter-socket connection per node

Sequential ranking of MPI_COMM_WORLD

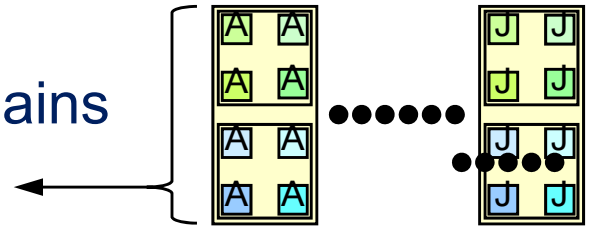
Does it matter?

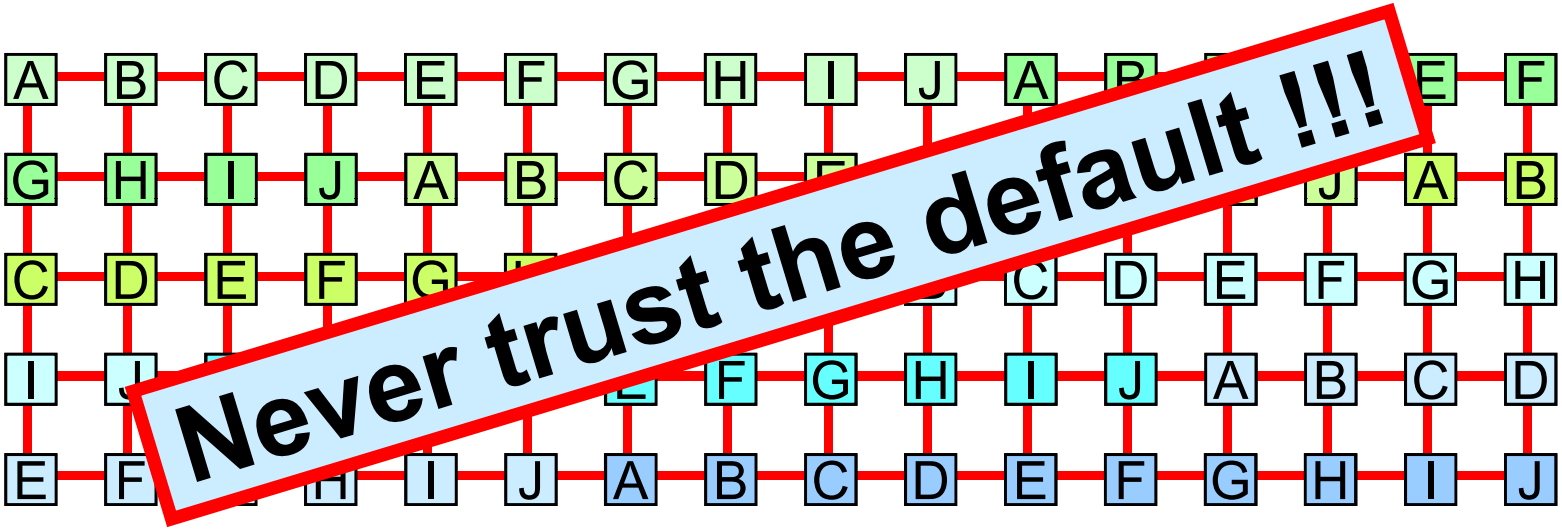
The Topology Problem with

pure MPI
one MPI process
on each core

Application example on 80 cores:

- Cartesian application with 5 x 16 = 80 sub-domains
- On system with 10 x dual socket x quad-core





\+ 32 x inter-node connections per node

− 0 x inter-socket connection per node

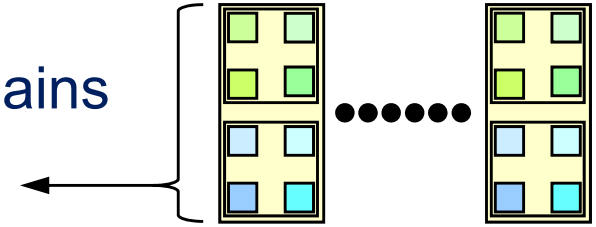
Round robin ranking of MPI_COMM_WORLD

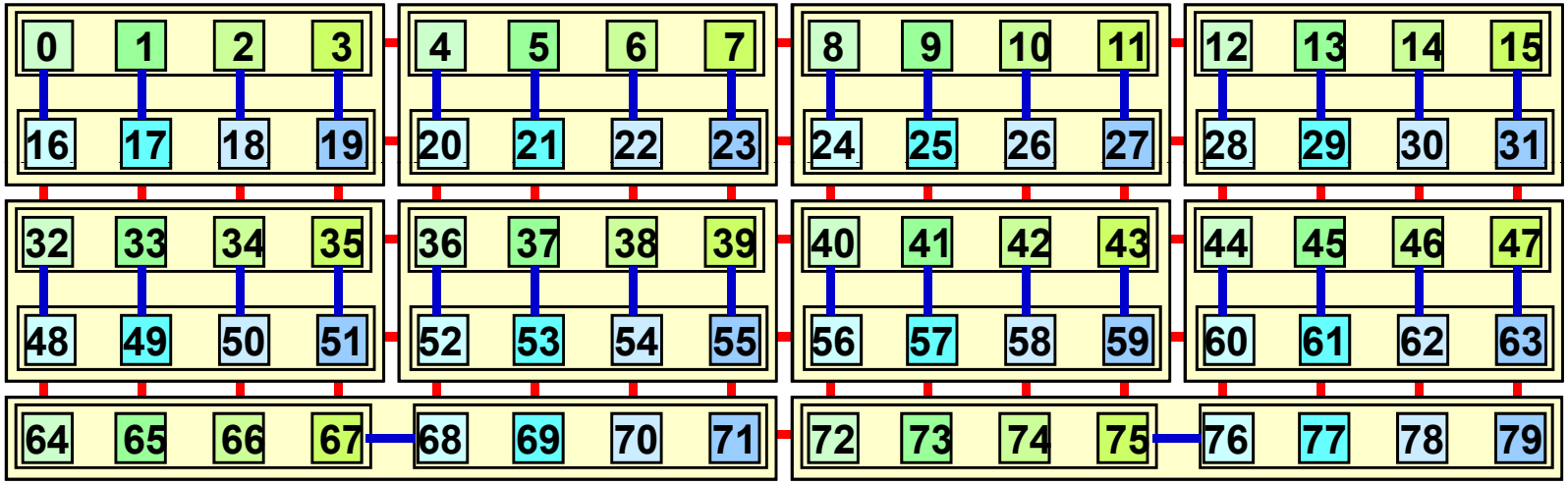
The Topology Problem with pure MPI

one MPI process on each core

Application example on 80 cores:

- Cartesian application with 5 x 16 = 80 sub-domains
- On system with 10 x dual socket x quad-core





+ 12 x inter-node connections per node
+ 4 x inter-socket connection per node

Two levels of domain decomposition

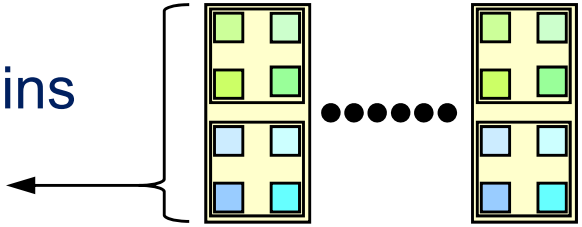
Bad affinity of cores to thread ranks

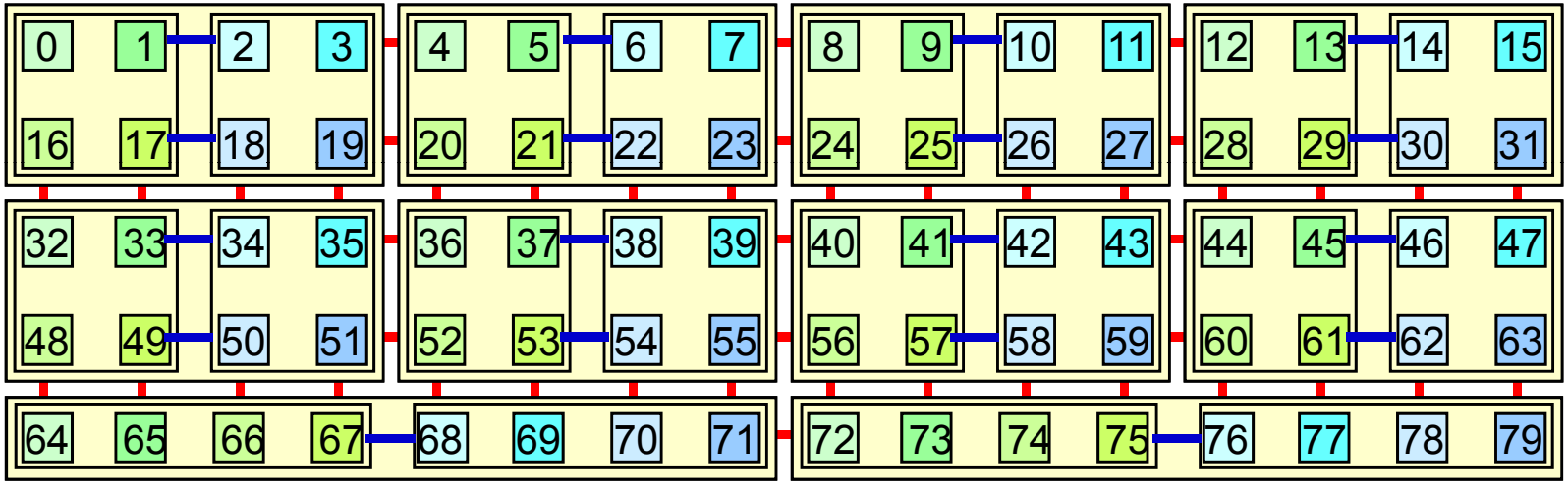
The Topology Problem with

pure MPI
one MPI process
on each core

Application example on 80 cores:

- Cartesian application with 5 x 16 = 80 subdomains
- On system with 10 x dual socket x quad-core





\+ 12 x inter-node connections per node

\+ 2 x inter-socket connection per node

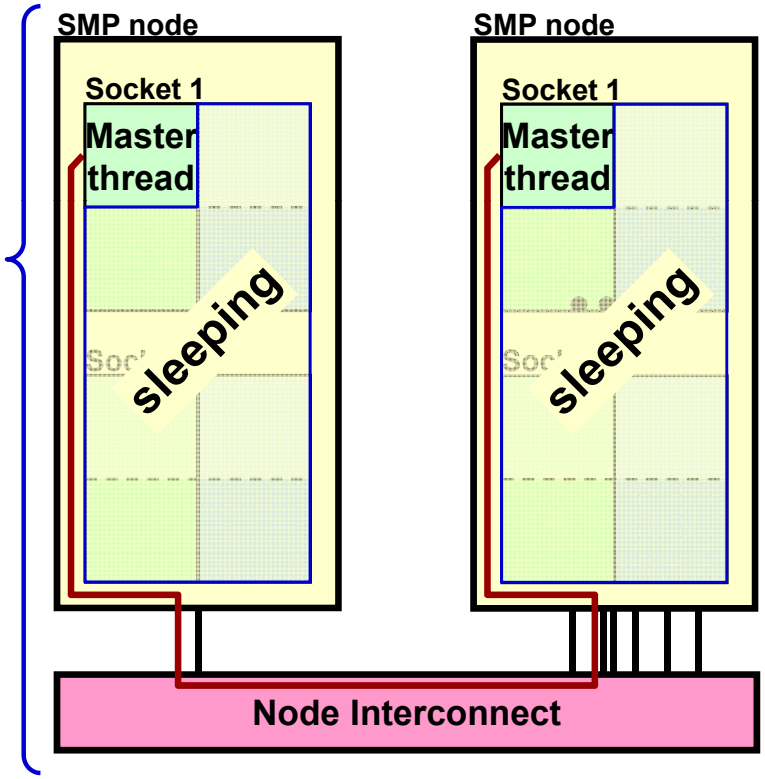
Two levels of domain decomposition

Good affinity of cores to thread ranks

Hybrid Mode: Sleeping threads and network saturation with

Masteronly
MPI only outside of parallel regions

```
for (iteration ….)
{
  #pragma omp parallel
    numerical code
  /*end omp parallel */

  /* on master thread only */
    MPI_Send (original data
      to halo areas
      in other SMP nodes)
    MPI_Recv (halo data
      from the neighbors)
} /*end for loop
```



Problem 1:
- **Can the master thread saturate the network?**

Solution:
- Use mixed model, i.e., several MPI processes per SMP node

Problem 2:
- **Sleeping threads are wasting CPU time**

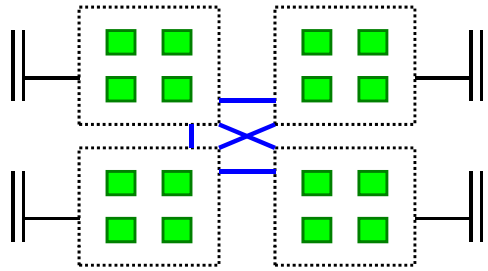
Solution:
- If funneling is suported use overlap of computation and communication

Problem 1&2 together:
- Producing more idle time through lousy bandwidth of master thread

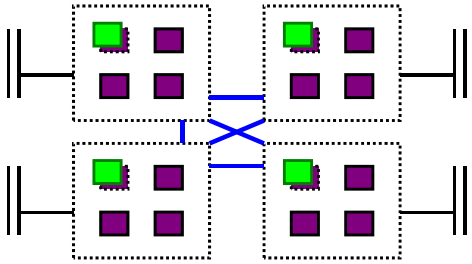
Pure MPI and Mixed Model

- **Problem:**
  - Contention for network access
  - MPI library must use appropriate fabrics / protocol for intra/inter-node communication
  - Intra-node bandwidth higher than inter-node bandwidth
  - MPI implementation may cause unnecessary data copying → waste of memory bandwidth
  - Increase memory requirements due to MPI buffer space
  - Mixed Model:
    - Need to control process and thread placement
    - Consider cache hierarchies to optimize thread execution

16 MPI Tasks



4 MPI Tasks
4Threads/Task



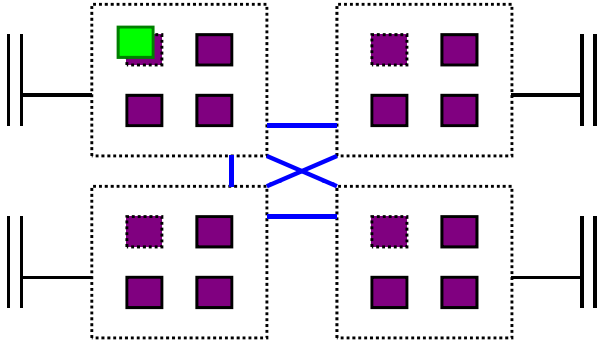
... but maybe not as much as you think!

Fully Hybrid Model

Problem 1: Can the master thread saturate the network?

Problem 2: Many Sleeping threads are wasting CPU time during communication

1 MPI Task
16Threads/Task



Problem 1&2 together:

- Producing more idle time through lousy bandwidth of master thread

Possible solutions:

- Use mixed model (several MPI per SMP)?
- If funneling is supported: Overlap communication/computation?
- Both of the above?

Problem 3:

- Remote memory access impacts the OpenMP performance

Possible solution:

- Control memory page placement to minimize impact of remote access

Other challenges for Hybrid Programming

- **Multicore / multisocket anisotropy effects**
  - Bandwidth bottlenecks, shared caches
  - Intra-node MPI performance
    - Core ↔ core vs. socket ↔ socket
  - OpenMP loop overhead depends on mutual position of threads in team
- **Non-Uniform Memory Access:**
  - Not all memory access is equal
- **ccNUMA locality effects**
  - Penalties for inter-LD access
  - Impact of contention
  - Consequences of file I/O for page placement
  - Placement of MPI buffers
- **Where do threads/processes and memory allocations go?**
  - Scheduling Affinity and Memory Policy can be changed within code with (sched_get/setaffinity, get/set_memory_policy)

Example: Sun Constellation Cluster Ranger (TACC)

Highly hierarchical

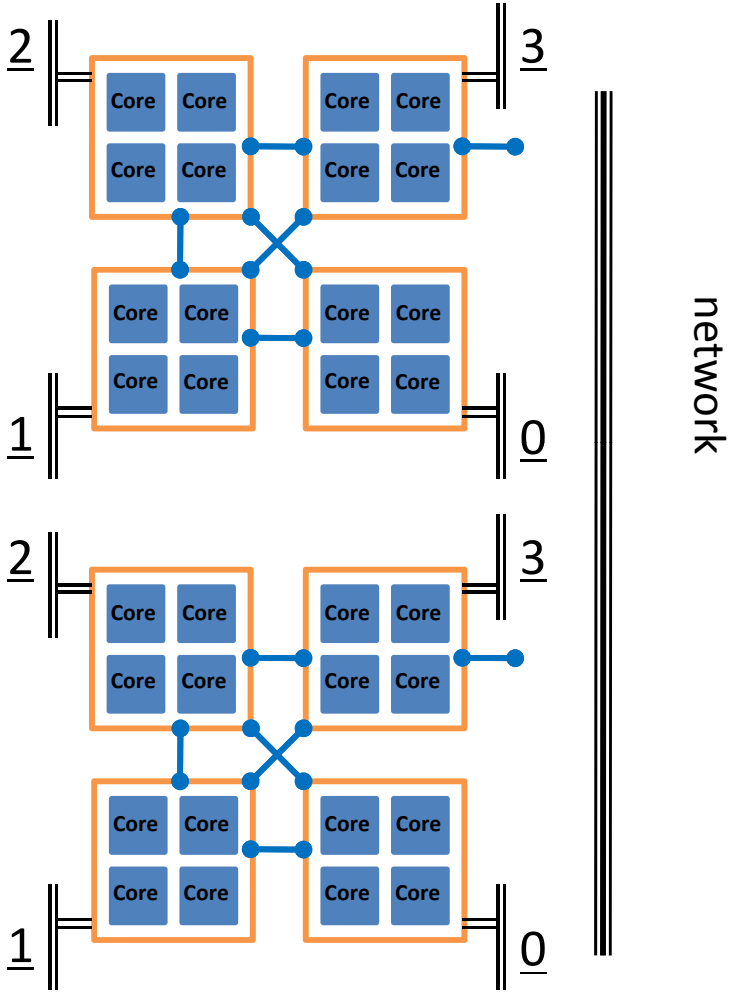
- **Shared Memory:**
  - 16 way cache-coherent, Non-uniform memory access (ccNUMA) node
- **Distributed Memory:**
  - Network of ccNUMA nodes
    - Core-to-Core
    - Socket-to-Socket
    - Node-to-Node
    - Chassis-to-chassis

Unsymmetric:

2 Sockets have 3 HT connected to neighbors

1 Socket has 2 connections to neighbors, 1 to network

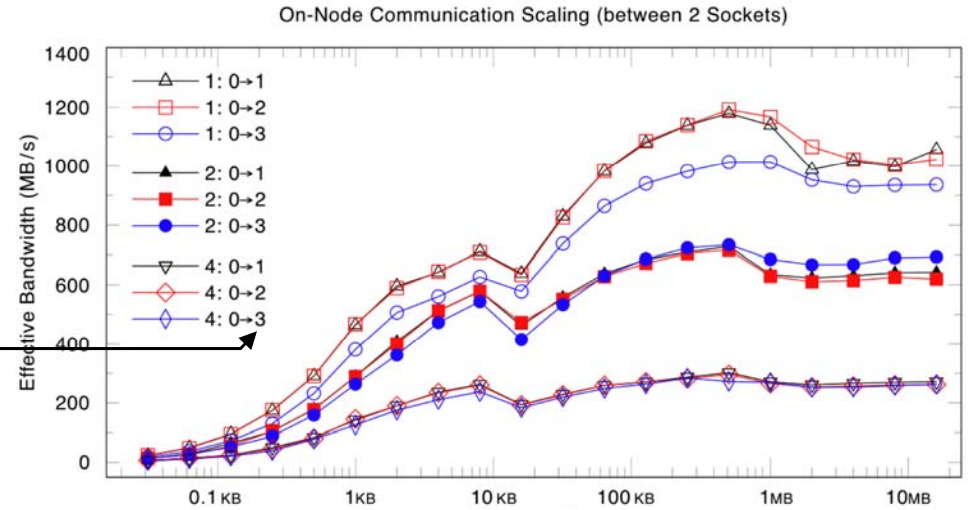
1 Socket has 2 connections to neighbors

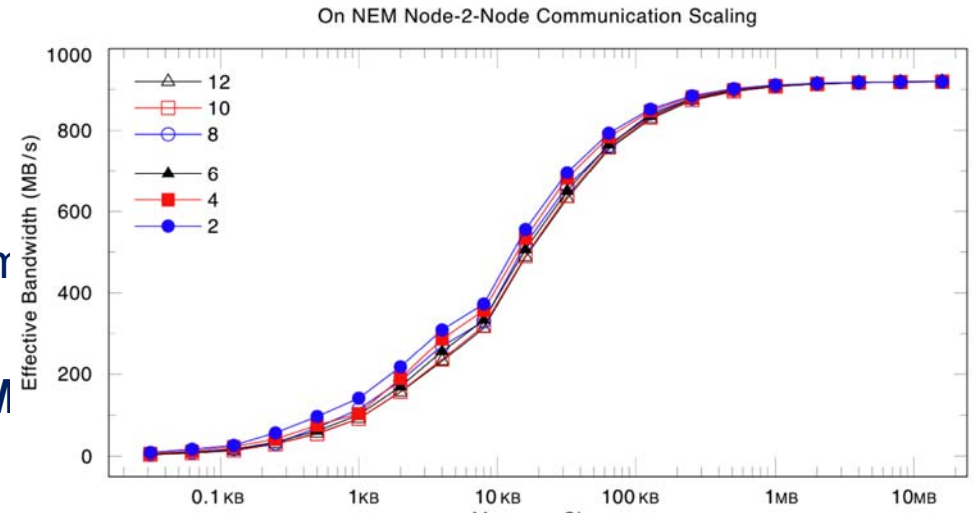


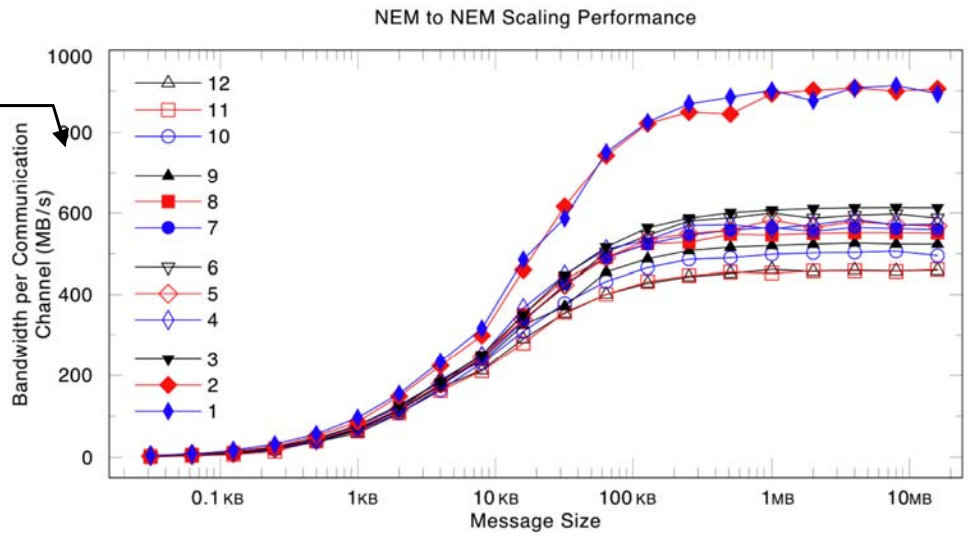
MPI ping-pong microbenchmark results on Ranger

- **Inside one node: Ping-pong socket 0 with 1, 2, 3 and 1, 2, or 4 simultaneous comm. (quad-core)**
  - Missing Connection: Communication between socket 0 and 3 is slower
  - Maximum bandwidth: 1 x 1180, 2 x 730, 4 x 300 MB/s
- **Node-to-node inside one chassis with 1-6 node-pairs (= 2-12 procs)**
  - Perfect scaling for up to 6 simultaneous comm
  - Max. bandwidth : 6 x 900 MB/s
- **Chassis to chassis (distance: 7 hops) with 1 M simultaneous communication links**
  - Max: 2 x 900 up to 12 x 450 MB/s

Exploiting Multi-Level Parallelism on the Sun Constellation System", L. Koesterke, et al., TACC, TeraGrid08 Paper







Overlapping Communication and Work

- One core can saturate the PCIe ←→network bus. Why use all to communicate?
- Communicate with one or several cores.
- Work with others during communication.
- Need at least MPI_THREAD_FUNNELED support.
- Can be difficult to manage and load balance!

Overlapping communication and computation

Three problems

1. **The application problem:**
   - one must separate application into:
     - code that can run before the halo data is received
     - code that needs halo data

   ➔ **very hard to do !!!**

2. **The thread-rank problem:**
   - comm. / comp. via thread-rank
   - cannot use worksharing directives

   ➔ **loss of major OpenMP support**
   (see next slide)

3. **The load balancing problem**

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

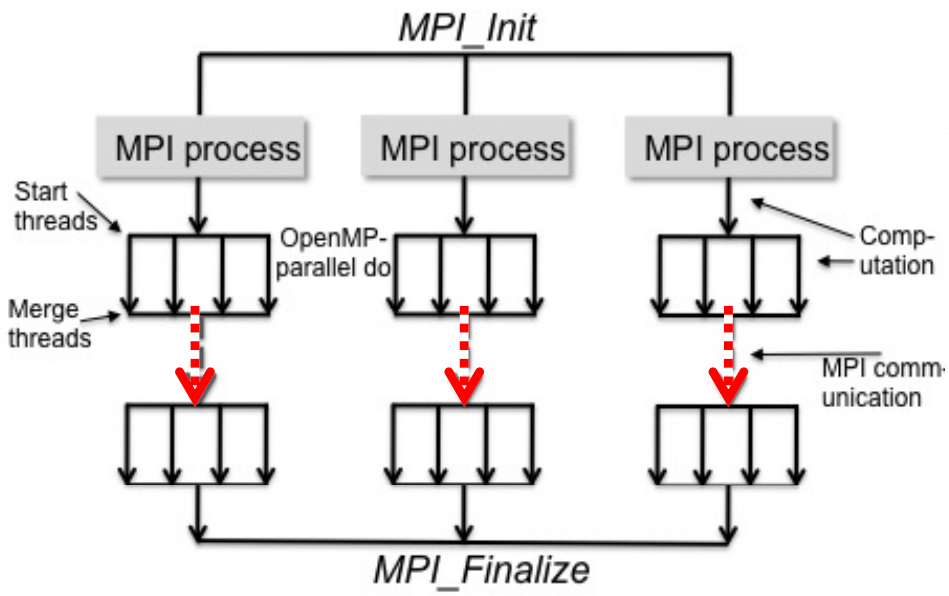
```
if (my_thread_rank < 1) {
  MPI_Send/Recv….
} else {
  my_range = (high-low-1)/(num_threads-1)+1;
  my_low = low + (my_thread_rank+1)*my_range;
  my_high=high+ (my_thread_rank+1+1)*my_range;
  my_high = max(high, my_high)
  for (i=my_low; i<my_high; i++) {
        ...
  }
}
```

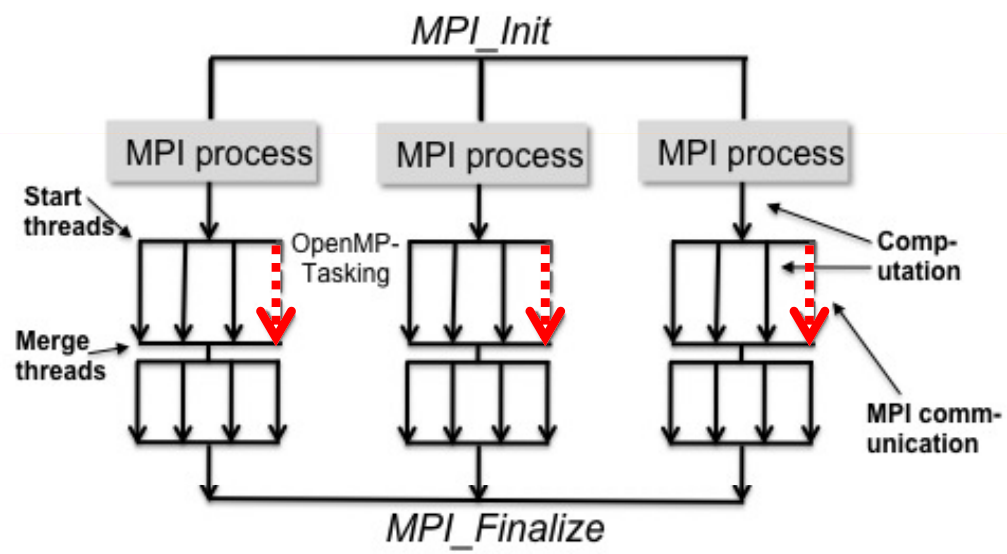
New in OpenMP 3.0: TASK Construct

- Purpose is to support the OpenMP parallelization of while loops
- Tasks are spawned when `!$omp task` or `#pragma omp task` is encountered
- Tasks are executed in an undefined order
- Tasks can be explicitly waited for by the use of `!$omp taskwait`
- Shows good potential for overlapping computation with communication and/or IO (see examples later on)

```
#pragma omp parallel  {
#pragma omp single private(p)
{
   p = listhead ;
      while (p) {
     #pragma omp task
        process (p);
     p=next (p) ;
}  // Implicit taskwait
```

Case study: Communication and Computation in Gyrokinetic Tokamak Simulation (GTS) shifter





A. Koniges et. al.: *Application Acceleration on Current and Future Cray Platforms*. Presented at CUG 2010, Edinburgh, GB, May 24-27, 2010.
R. Preissl, et. al.: *Overlapping communication with computation using OpenMP tasks on the GTS magnetic fusion code*. Scientific Programming, IOS Press, Vol. 18, No. 3-4 (2010)

OpenMP Tasking Model gives a new way to achieve more parallelism form hybrid computation.

Slides courtesy of Alice Koniges, NERSC, LBNL

Communication and Computation in Gyrokinetic Tokamak Simulation (GTS) shift routine

```
do iterations=1,N
!compute particles to be shifted
  !$omp parallel do
  shift_p=particles_to_shift(p_array);

!communicate amount of shifted
! particles and return if equal to 0
  shift_p=x+y
  MPI_ALLREDUCE(shift_p,sum_shift_p);
  if(sum_shift_p==0) { return; }

!pack particle to move right and left
  !$omp parallel do
  do m=1,x
    sendright(m)=p_array(f(m));
  enddo
  !$omp parallel do
  do n=1,y
    sendleft(n)=p_array(f(n));
  enddo
```

```
1   der remaining particles: fill holes
3   _hole(p_array);

5   number of particles to move right
    _SENDRECV(x,length=2,..);
7   to right and receive from left
    _SENDRECV(sendright,length=g(x),..);
    number of particles to move left
9   _SENDRECV(y,length=2,..);
    to left and receive from right
11  _SENDRECV(sendleft,length=g(y),..);

13  ng shifted particles from right
    np parallel do
15  n=1,x
    _array(h(m))=sendright(m);
17  do
    ng shifted particles from left
19  np parallel do
    n=1,y
21  _array(h(n))=sendleft(n);
    enddo
}
```

INDEPENDENT

INDEPENDENT

SEMI-INDEPENDENT

23 25 27 29 31 33 35 37 39 41 43

Slides courtesy of Alice Koniges, NERSC, LBNL

Overlapping can be achieved with OpenMP tasks (2nd part)

```
!$omp parallel
!$omp master
  !$omp task
  fill_hole(p_array);
  !$omp end task

  MPI_SENDRECV(x,length=2,..);
  MPI_SENDRECV(sendright,length=g(x),..);
  MPI_SENDRECV(y,length=2,..);
!$omp end master
!$omp end parallel
}
```

Particle reordering of the remaining

```
!$omp parallel
!$omp master
!adding shifted particles from right
   do m=1,x-stride,stride
     !$omp task
     do mm=0,stride-1,1
        p_array(h(m))=sendright(m);
     enddo
     !$omp end task
   enddo
   !$omp task
   do m=m,x
      p_array(h(m))=sendright(m);
   enddo
   !$omp end task

  MPI_SENDRECV(sendleft,length=g(y),..);
!$omp end master
!$omp end parallel

!adding shifted particles from left
!$omp parallel do
do n=1,y
   p_array(h(n))=sendleft(n);
enddo
```

Overlapping remaining MPI_Sendrecv

Slides, courtesy of Alice Koniges, NERSC, LBNL

Overlapping can be achieved with OpenMP tasks (1st part)

```
integer stride=1000
!$omp parallel
!$omp master
!pack particle to move right
  do m=1,x-stride ,stride
    !$omp task
    do mm=0,stride -1,1
      sendright(m+mm)=p array(f(m+mm));
    enddo
    !$omp end task
  enddo
  !$omp task
  do m=m,x
    sendright(m)=p_array(f(m));
  enddo
  !$omp end task
```

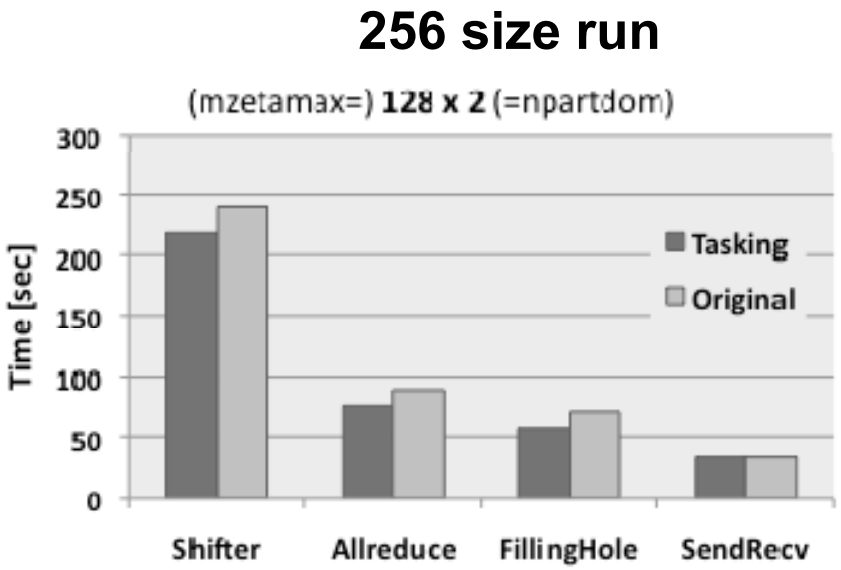
```
!pack particle to move left
  do n=1,y-stride ,stride
    !$omp task
    do nn=0,stride -1,1
      sendleft(n+nn)=p_array(f(n+nn));
    enddo
    !$omp end task
  enddo
  !$omp task
  do n=n,y
    sendleft(n)=p_array(f(n));
  enddo
  !$omp end task
  MPI_ALLREDUCE(shift_p ,sum_shift_p);
!$omp end master
!$omp end parallel
if(sum_shift_p==0) { return; }
```

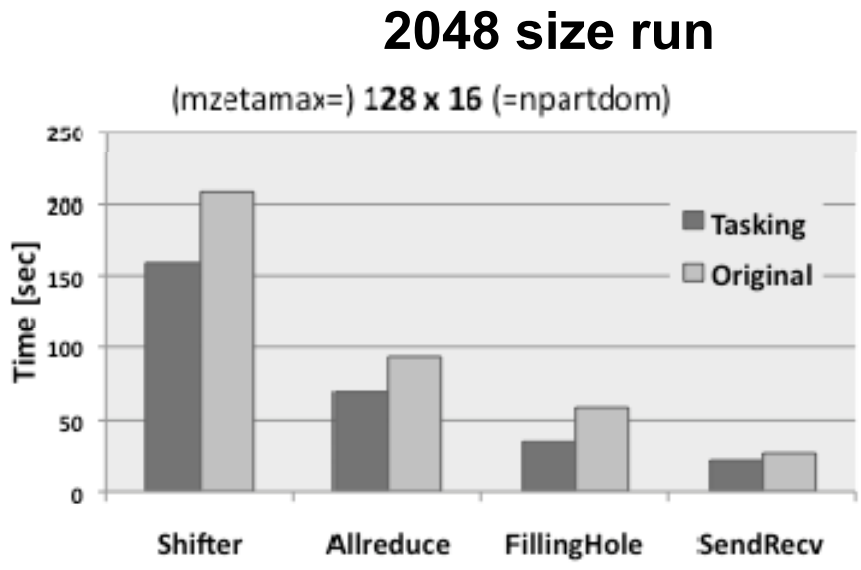
Overlapping MPI_Allreduce with particle work

- Overlap: Master thread encounters (!$omp master) tasking statements and creates work for the thread team for deferred execution. MPI Allreduce call is immediately executed.
- MPI implementation has to support at least MPI_THREAD_FUNNELED
- Subdividing tasks into smaller chunks to allow better load balancing and scalability among threads.

Slides, courtesy of Alice Koniges, NERSC, LBNL

OpenMP tasking version outperforms original shifter, especially in larger poloidal domains





- **Performance breakdown of GTS shifter routine using 4 OpenMP threads per MPI pro-cess with varying domain decomposition and particles per cell on Franklin Cray XT4.**
- **MPI communication in the shift phase uses a toroidal MPI communicator (constantly 128).**
- **Large performance differences in the 256 MPI run compared to 2048 MPI run!**
- **Speed-Up is expected to be higher on larger GTS runs with hundreds of thousands CPUs since MPI communication is more expensive.**

Slides, courtesy of
Alice Koniges, NERSC, LBNL

Other Hybrid Programming Opportunities

- **Exploit hierarchical parallelism within the application:**
  - Coarse-grained parallelism implemented in MPI
  - Fine-grained parallelism on loop level exploited through OpenMP
- **Increase parallelism if coarse-grained parallelism is limited**
- **Improve load balancing, e.g. by restricting # MPI processes or assigning different # threads to different MPI processes**
- **Lower the memory requirements by restricting the number of MPI processes**
  - Lower requirements for replicated data
  - Lower requirements for MPI buffer space
- **Examples for all of this will be presented in the case studies**

Practical “How-Tos” for hybrid

How to compile, link and run

- **Compiler usually invoked via a wrapper script, e.g., "mpif90", "mpicc"**
- **Use appropriate compiler flag to enable OpenMP directives/pragmas:**

  `-openmp` (Intel), `-mp` (PGI), `-qsmp=omp` (IBM)
- **Link with MPI library**
  - Usually wrapped in MPI compiler script
  - If required, specify to link against thread-safe MPI library (Often automatic when OpenMP or auto-parallelization is switched on)

- **Running the code**
  - Highly nonportable! Consult system docs! (if available…)
  - If you are on your own, consider the following points
  - Make sure `OMP_NUM_THREADS` etc. is available on all MPI processes
    - E.g., start "env VAR=VALUE … <YOUR BINARY>" instead of your binary alone
  - Figure out how to start less MPI processes than cores on your nodes

Compiling/Linking Examples (1)

- PGI (Portland Group compiler)
  - `mpif90 -fast -mp`
- Pathscale :
  - `mpif90 -Ofast -openmp`
- IBM Power 6:
  - `mpxlf_r -O4 -qarch=pwr6 -qtune=pwr6 -qsmp=omp`
- Intel Xeon Cluster:
  - `mpif90 -openmp -O2`

High optimization level is required because enabling OpenMP interferes with compiler optimization

Compile/Run/Execute Examples (2)

- NEC SX9
  - NEC SX9 compiler
  - **`mpif90 -C hopt -P openmp`** … # –ftrace for profiling info
  - Execution:

```
$ export OMP_NUM_THREADS=<num_threads>
$ MPIEXPORT="OMP_NUM_THREADS"
$ mpirun -nn <# MPI procs per node> -nnp <# of nodes> a.out
```

- Standard x86 cluster:
  - Intel Compiler
  - **`mpif90 -openmp`** …
  - Execution (handling of **`OMP_NUM_THREADS`**, see next slide):

```
$ mpirun_ssh -np <num MPI procs> -hostfile machines a.out
```

Handling OMP_NUM_THREADS

- **without any support by mpirun:**
  - **Problem** (e.g. with mpich-1): mpirun has no features to export environment variables to the via ssh automatically started MPI processes
  - **Solution:**
    ```
    export OMP_NUM_THREADS=<# threads per MPI process>
    ```
    in ~/.bashrc (if a bash is used as login shell)
  - **Problem**: Setting OMP_NUM_THREADS individually for the MPI processes:
  - **Solution:**
    ```
    test -s ~/myexports && . ~/myexports
    ```
    in your ~/.bashrc
    ```
    echo '$OMP_NUM_THREADS=<# threads per MPI process>' >
    ~/myexports
    ```
    before invoking mpirun. **Caution: Several invocations of mpirun cannot be executed at the same time with this trick!**
- **with support, e.g. by OpenMPI –x option:**
  ```
  export OMP_NUM_THREADS= <# threads per MPI process>
  mpiexec -x OMP_NUM_THREADS -n <# MPI processes> ./a.out
  ```

Example: Constellation Cluster Ranger (TACC)

- **Sun Constellation Cluster:**
  - `mpif90 -fastsse -tp barcelona-64 -mp` …
  - SGE Batch System
  - `ibrun numactl.sh a.out`
  - Details see TACC Ranger User Guide (www.tacc.utexas.edu/services/userguides/ranger/#numactl)

```
#!/bin/csh
#$ -pe 2way 512
setenv OMP_NUM_THREADS 8
ibrun  numactl.sh  bt-mz-64.exe
```

2 MPI Procs per node
512 cores total

Example: Cray XT5

Cray XT5:

- 2 quad-core AMD Opteron per node
- `ftn -fastsse -mp` (PGI compiler)

Maximum of 8 threads per MPI process on XT5

```
#!/bin/csh
#PBS -q standard
#PBS -l mppwidth=512
#PBS -l walltime=00:30:00
module load xt-mpt
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 8
aprun -n 64 -N 1 -d 8./bt-mz.64
setenv OMP_NUM_THREADS 4
aprun -n 128 -S 1 -d 4 ./bt-mz.128
```

8 threads per MPI Process

Number of MPI Procs per Node:
1 Proc per node with up to 8 threads each

4 threads per MPI Process

Number of MPI Procs per Numa Node:
1 Proc per Numa Node => 2 Procs per Node

Example: Different Number of MPI Processes per Node (XT5)

- **Usage Example:**
  - Different Components of an application require different resources, eg. Community Climate System Model (CCSM)

```
aprun -n 8 -S 4 -d 1 ./ccsm.exe: -n 4 -S 2 -d 2 ccsm.exe : \
 -n 2 -S 1 -d 4  .ccsm.exe: -n 2 -N 1 -d 8 ./ccsm.exe
```

8 MPI Procs with 1 thread
4 MPI Procs with 2 threads
2 MPI Procs with 4 threads
2 MPI Procs with 8 threads

```
export MPICH_RANK_REORDER_DISPLAY=1
```

```
PE_0]: rank 0 is on nid00205 [PE_0]:
rank 1 is on nid00205 [PE_0]: rank 2
is on nid00205 [PE_0]: rank 3 is on
nid00205 [PE_0]: rank 4 is on
nid00205 [PE_0]: rank 5 is on
nid00205 [PE_0]: rank 6 is on
nid00205 [PE_0]: rank 7 is on
nid00205 [PE_0]: rank 8 is on
nid00208 [PE_0]: rank 9 is on
nid00208 [PE_0]: rank 10 is on
nid00208 [PE_0]: rank 11 is on
nid00208 [PE_0]: rank 12 is on
nid00209 [PE_0]: rank 13 is on
nid00209 [PE_0]: rank 14 is on
nid00210 [PE_0]: rank 15 is on
nid00211
```

Example : IBM Power 6

- Hardware: 4.7GHz Power6 Processors, 150 Compute Nodes, 32 Cores per Node, 4800 Compute Cores
- **`mpxlf_r -O4 -qarch=pwr6 -qtune=pwr6 -qsmp=omp`**

enable OpenMP

Crucial for full optimization in presence of OpenMP directives

```
#!/bin/csh
#PBS -N bt-mz-16x4
#PBS -m be
#PBS -l walltime=00:35:00
#PBS -l select=2:ncpus=32:mpiprocs=8:ompthreads=4
#PBS -q standard
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 4
poe ./bin/bt-mz.B.16
```

Example : Intel Linux Cluster

ScaliMPI

```
#!/bash
#PBS -q standard
#PBS -l select=16:ncpus=4
#PBS -l walltime=8:00:00
#PBS -j oe
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=2
mpirun -np 32 -npn 2 -affinity_mode none ./bt-mz.C.32
```

Place 2 MPI Procs per node

Use more than one core per MPI Proc

OpenMPI

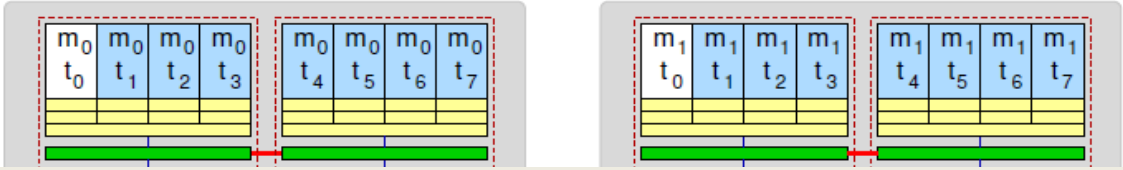
```
#!/bash
#PBS -q standard
#PBS -l select=16:ncpus=4
#PBS -l walltime=8:00:00
#PBS -j oe
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=2
mpirun -np 32 -bynode ./bt-mz.C.32
```

Processes placed round-robin on nodes

Topology choices with MPI/OpenMP:

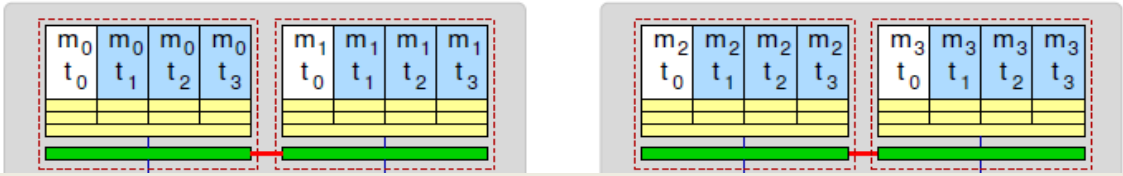
More examples using Intel MPI+compiler & home-grown mpirun (@RRZE)

One MPI process per node




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

One MPI process per socket




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

OpenMP threads pinned "round robin" across cores in node


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

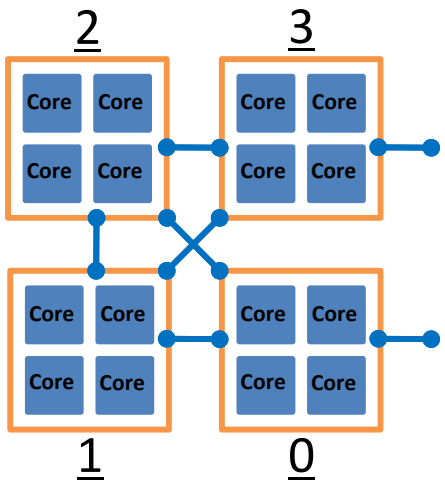
Two MPI processes per socket


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

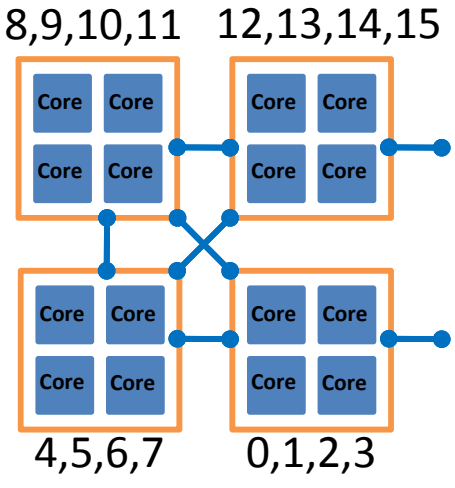
NUMA Control: Process and Memory Placement

- **Affinity and Policy can be changed externally through `numactl` at the socket and core level.**

```
Command:          numactl    <options>    ./a.out
```



Socket References

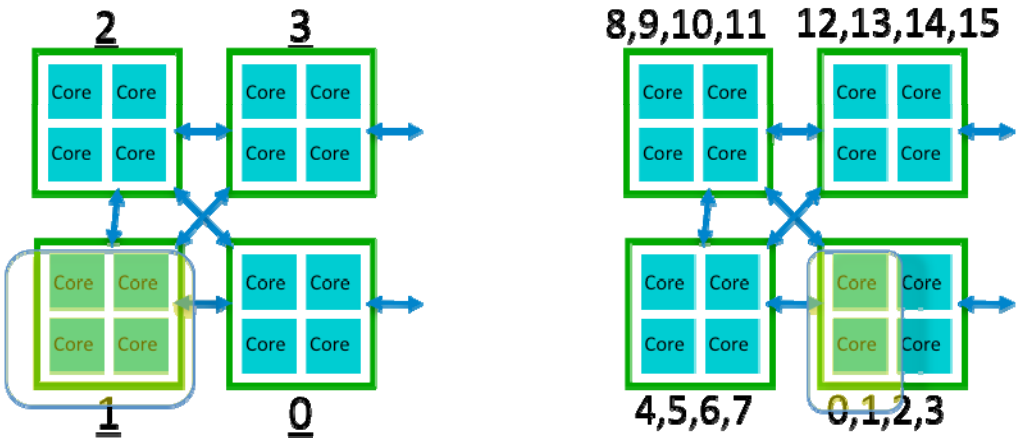


Core References

NUMA Control: Process Placement

- Affinity and Policy can be changed externally through numactl at the socket and core level.

Command: numactl <options> ./a.out



Caution: socket numbering system dependent!

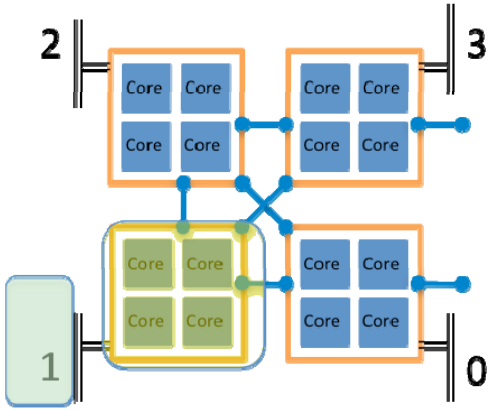
Socket References

Example: numactl -N 1 ./a.out

Core References

Example: numactl –c 0,1 ./a.out

NUMA Operations: Memory Placement



Memory: Socket References

Memory allocation:

- MPI
  - local allocation is best
- OpenMP
  - Interleave best for large, completely shared arrays that are randomly accessed by different threads
  - local best for private arrays
- Once allocated, a memory-structure is fixed

Example: numactl -N 1 -l ./a.out

06/19/09, Author: Gabriele Jost

3/23/11

Tutorial on Hybrid Programming PRACE Spring School 2011: Case Studies

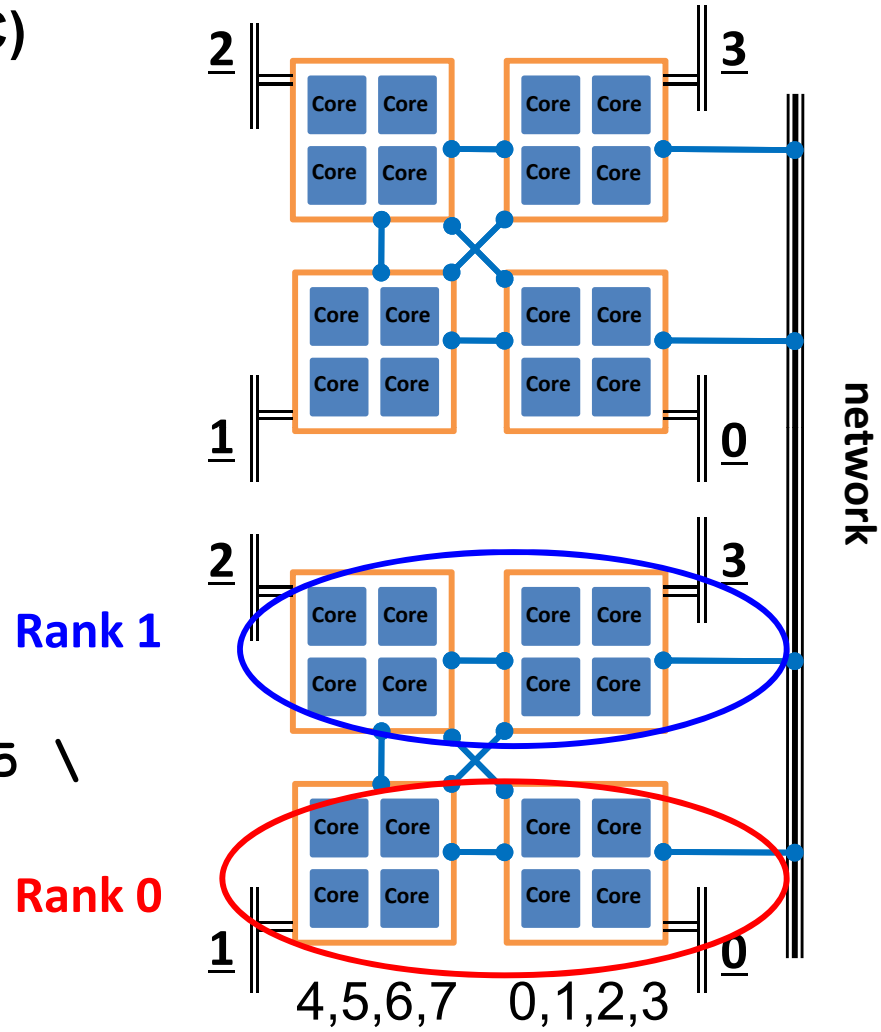
12

Example: Numactl on Ranger Cluster (TACC)

Running BT-MZ Class D 128 MPI Procs, 8 threads each, 2 MPI on each node on Ranger (TACC)

Use of `numactl` for affinity:

```
if [ $localrank == 0 ]; then
exec numactl \
   --physcpubind=0,1,2,3,4,5,6,7  \
   -m 0,1 $*
elif [ $localrank == 1 ]; then
exec numactl \
   --physcpubind=8,9,10,11,12,13,14,15 \
   -m 2,3 $*
fi
```



Example: numactl on Lonestar Cluster at TACC

```
CPU type: Intel Core Westmere processor
*************************************
Hardware Thread Topology
*************************************
Sockets:                    2
Cores per socket:           6
Threads per core:           1


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

Half of the threads access remote memory

Running NPB BT-MZ Class D 128 MPI Procs, 6 threads each 2MPI per node

Pinning A:

```
if [ $localrank == 0 ]; then
exec numactl --physcpubind=0,1,2,3,4,5 \
    -m 0 $*
elif [ $localrank == 1 ]; then
exec numactl \
    --physcpubind=6,7,8,9,10,11 \
    -m 1 $*
fi
```

610 Gflop/s

Running 128 MPI Procs, 6 threads each

Pinning B:

```
if [ $localrank == 0 ]; then
exec numactl --physcpubind=0,2,4,6,8,10 \
    -m 0 $*
elif [ $localrank == 1 ]; then
exec numactl -physcpubind=1,3,5,7,9,11 \
    -m 1 $*
fi
```

900 Gflop/s

Lonestar Node Topology

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

likwid-topology output

Performance Statistics

- **Important MPI Statistics:**
  - Time spent in communication
  - Time spent in synchronization
  - Amount of data communicated, length of messages, number of messages
  - Communication pattern
  - Time spent in communication vs computation
  - Workload balance between processes
- **Important OpenMP Statistics:**
  - Time spent in parallel regions
  - Time spent in work-sharing
  - Workload distribution between threads
  - Fork-Join Overhead
- **General Statistics:**
  - Time spent in various subroutines
  - Hardware Counter Information (CPU cycles, cache misses, TLB misses, etc.)
  - Memory Usage
- **Methods to Gather Statistics:**
  - Sampling/Interrupt based via a profiler
  - Instrumentation of user code
  - Use of instrumented libraries, e.g. instrumented MPI library

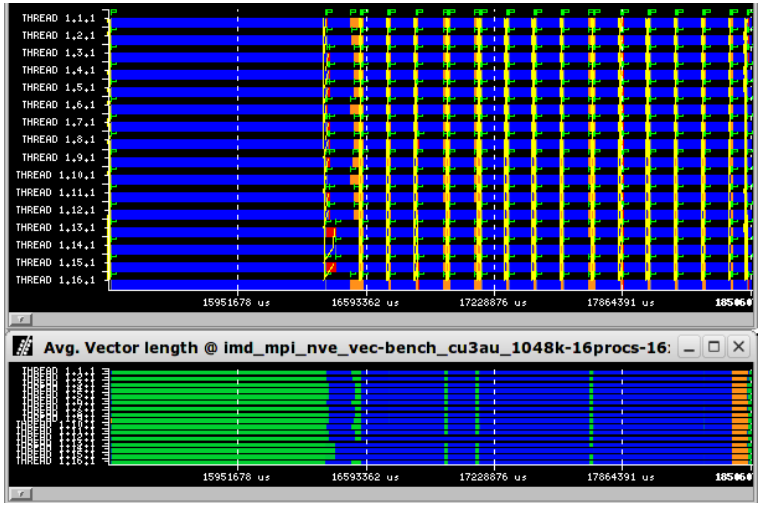
Examples of Performance Analysis Tools

- **Vendor Supported Software:**
  - CrayPat/Cray Apprentice2: Offered by Cray for the XT Systems.
  - pgprof: Portland Group Performance Profiler
  - Intel Tracing Tools
  - IBM xprofiler
- **Public Domain Software:**
  - PAPI (Performance Application Programming Interface):
    - Support for reading hardware counters in a portable way
    - Basis for many tools
    - http://icl.cs.utk.edu/papi/
  - TAU:
    - Portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++ and others
    - University of Oregon, http://www.cs.uoregon.edu/research/tau/home.php
  - IPM (Integrated Performance Monitoring):
    - Portable profiling infrastructure for parallel codes
    - Provides a low-overhead performance summary of the computation
    - http://ipm-hpc.sourceforge.net/
  - Scalasca:
    - http://icl.cs.utk.edu/scalasca/index.html
  - Paraver:
    - Barcelona Supersomputing Center
    - http://www.bsc.es/plantillaA.php?cat_id=488

see Case Studies

Performance Tools Support for Hybrid Code

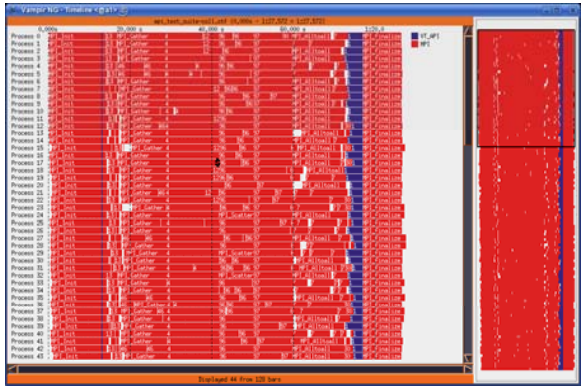
- **Paraver** tracing is done with linking against (closed-source) `omptrace` or `ompitrace`



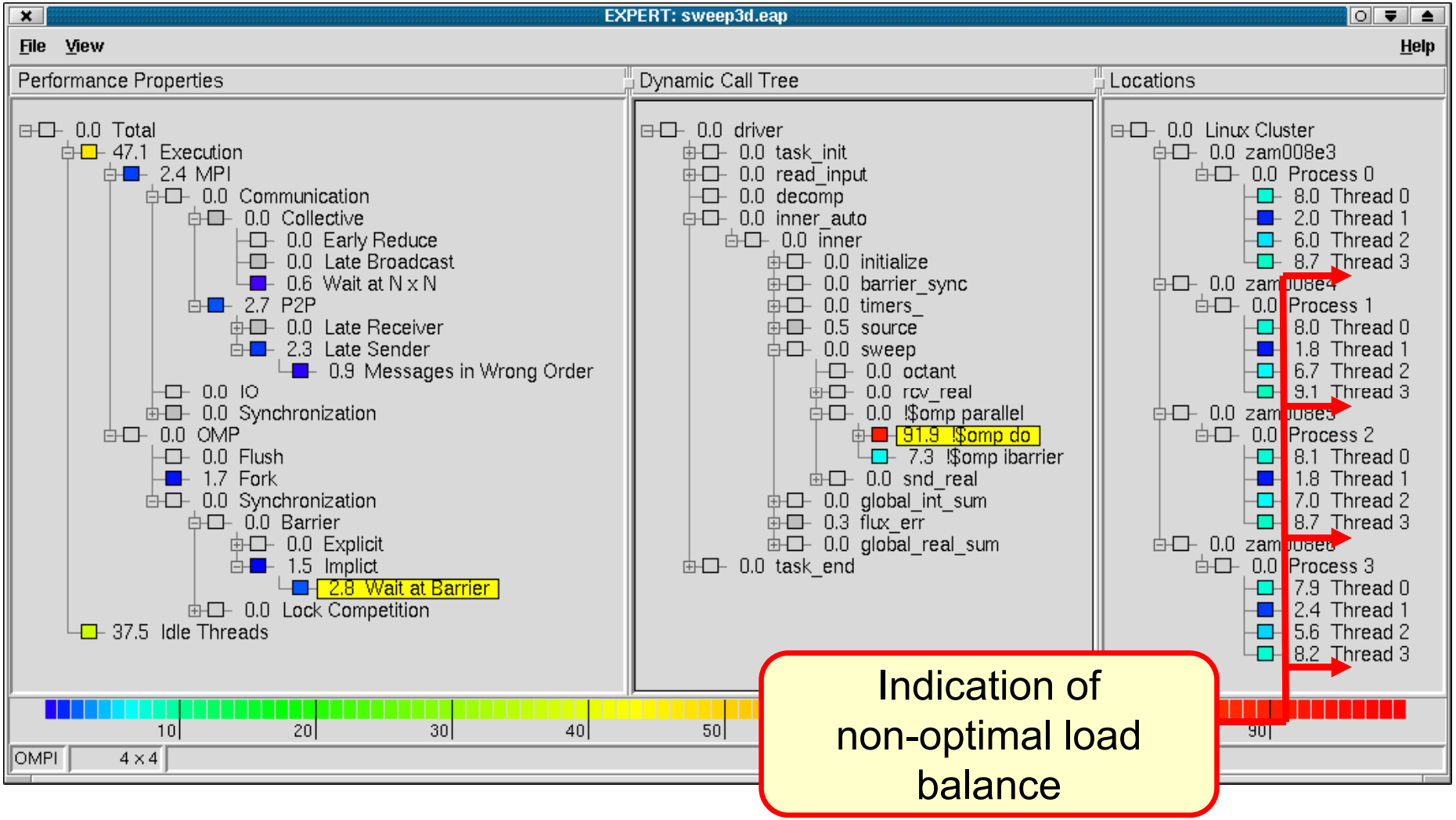
- **For Vampir/Vampirtrace performance analysis:**

```
./configure -enable-omp \
  -enable-hyb \
  -with-mpi-dir=/opt/OpenMPI/1.3-icc \
  CC=icc F77=ifort FC=ifort
```

(Attention: does not wrap `MPI_Init_thread`!)



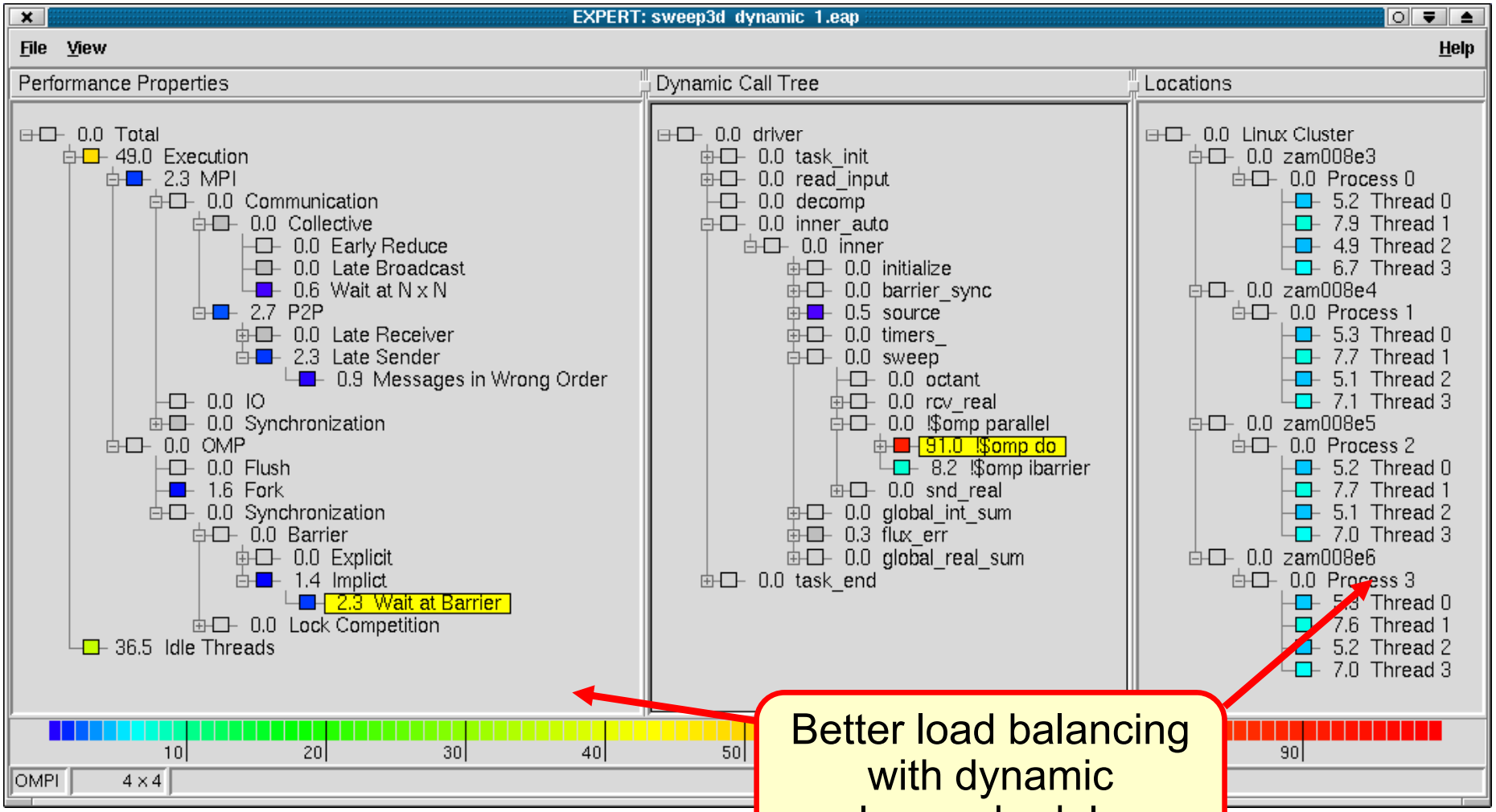
Scalasca – Example "Wait at Barrier"



Indication of non-optimal load balance

Screenshots, courtesy of KOJAK JSC, FZ Jülich

Scalasca – Example "Wait at Barrier", Solution



Screenshots, courtesy of KOJAK JSC, FZ Jülich

MPI/OpenMP hybrid "how-to": Take-home messages

- **Be aware of inter/intra-node MPI behavior:**
  - available shared memory vs resource contention
- **Observe the topology dependence of**
  - Inter/Intra-node MPI
  - OpenMP overheads
- **Enforce proper thread/process to core binding, using appropriate tools (whatever you use, but use SOMETHING)]**
- **OpenMP processes on ccNUMA nodes require correct page placement**

Tutorial outline

- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
  - Practical "How-tos" for hybrid
- **Online demo: likwid tools (2)**
  - Advanced pinning
  - Making bandwidth maps
  - Using likwid-perfctr to find NUMA problems and load imbalance
  - likwid-perfctr internals
  - likwid-perfscope
- **Case studies for hybrid MPI/OpenMP**
  - Overlap for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - PIR3D – hybridization of a full scale CFD code
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

Live demo:

LIKWID tools – advanced topics

Tutorial outline

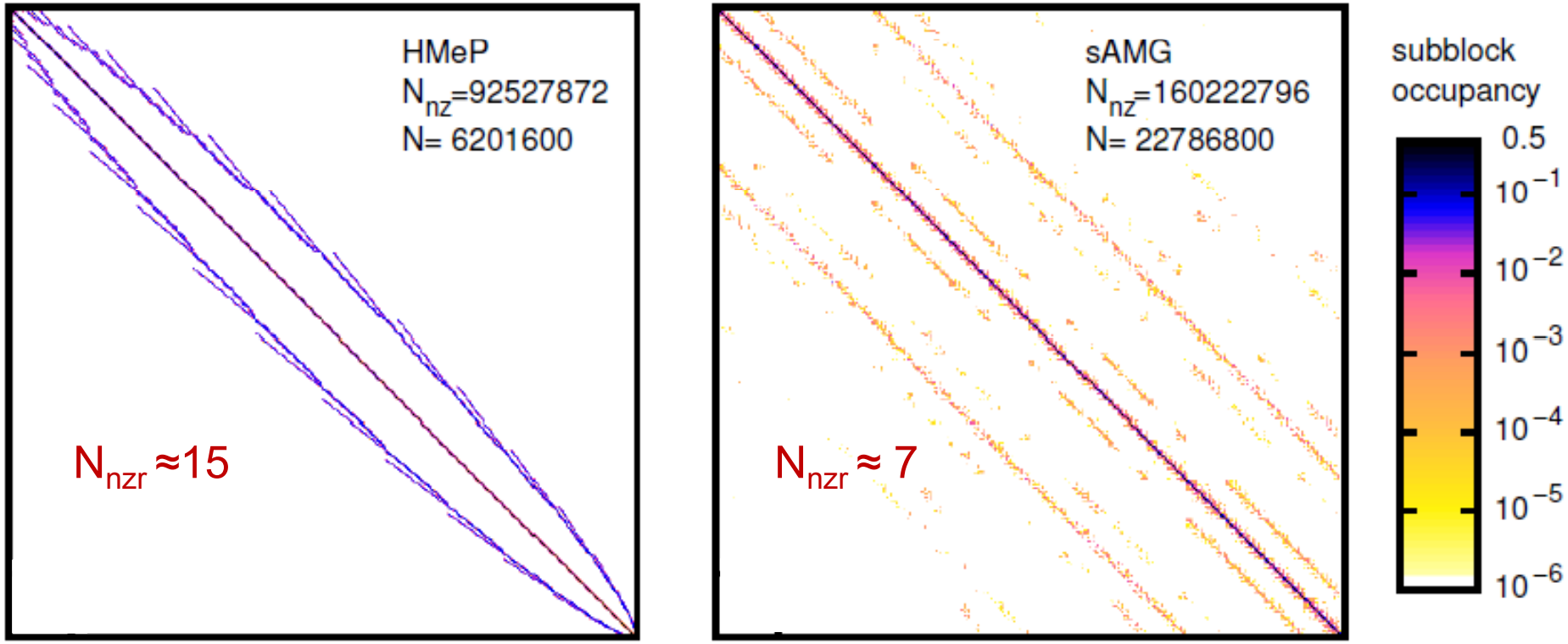
- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
  - Practical “How-tos” for hybrid
- **Online demo: likwid tools (2)**
  - Advanced pinning
  - Making bandwidth maps
  - Using likwid-perfctr to find NUMA problems and load imbalance
  - likwid-perfctr internals
  - likwid-perfscope
- **Case studies for hybrid MPI/OpenMP**
  - Overlap for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - PIR3D – hybridization of a full scale CFD code
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

Case study:
MPI/OpenMP hybrid parallel sparse matrix-vector multiplication

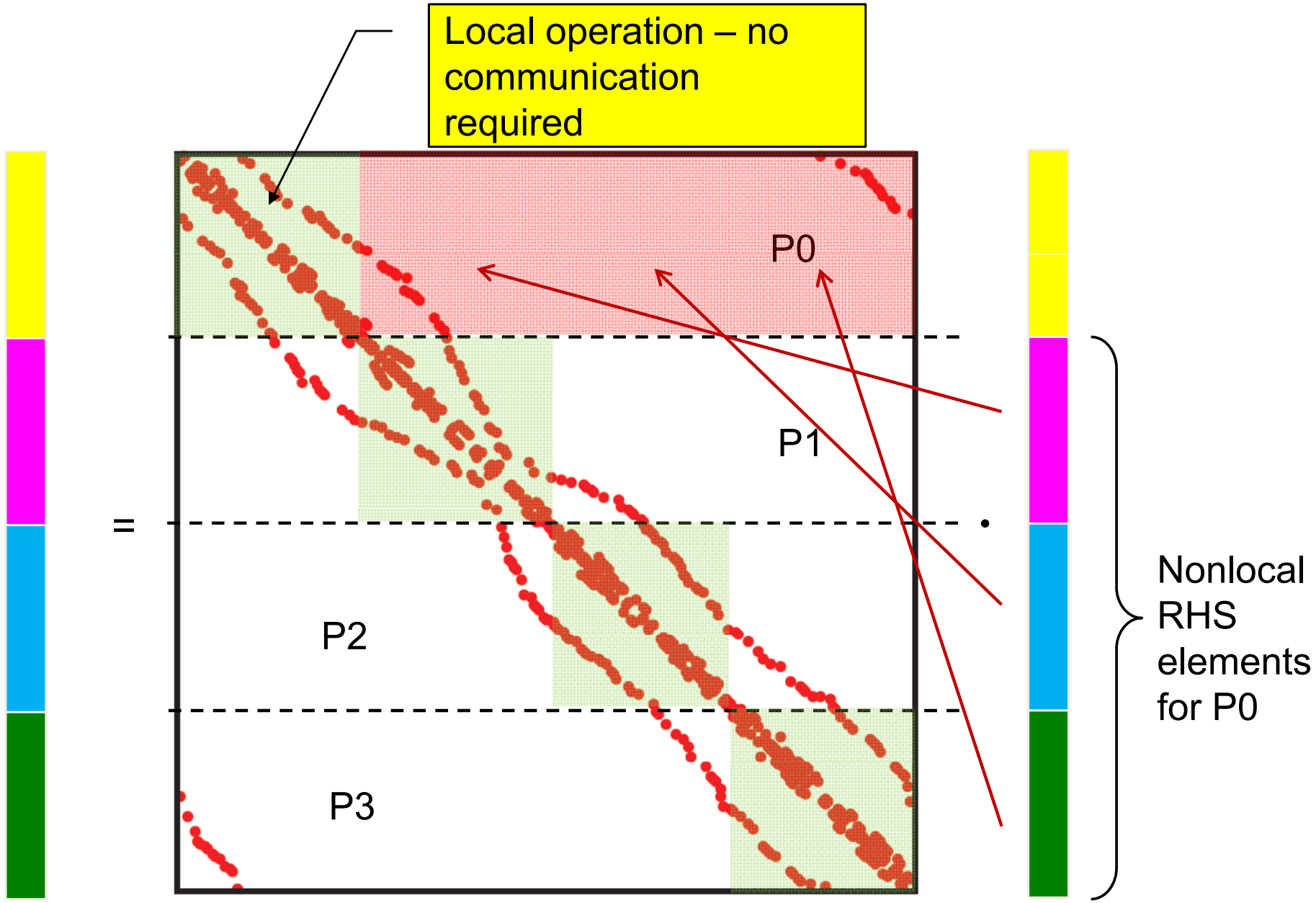
A case for explicit overlap of communication and computation

SpMVM test cases

- **Matrices in our test cases: $N_{nzr} \approx 7 \ldots 15$ → RHS and LHS do matter!**
  - HM: Hostein-Hubbard Model (solid state physics), 6-site lattice, 6 electrons, 15 phonons, $N_{nzr} \approx 15$
  - sAMG: Adaptive Multigrid method, irregular discretization of Poisson stencil on car geometry, $N_{nzr} \approx 7$



Distributed-memory parallelization of spMVM

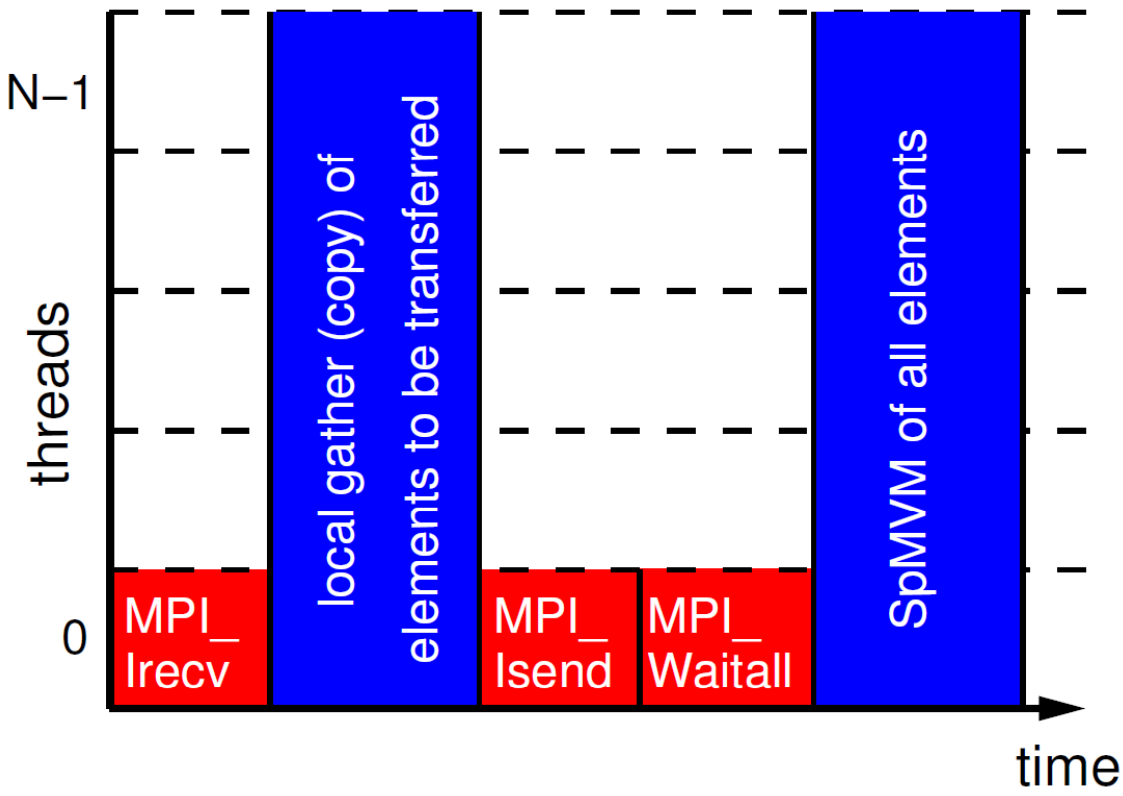


Distributed-memory parallelization of spMVM

- **Variant 1: "Vector mode" without overlap**

- **Standard concept for "hybrid MPI+OpenMP"**
- **Multithreaded computation (all threads)**

- **Communication only outside of computation**



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

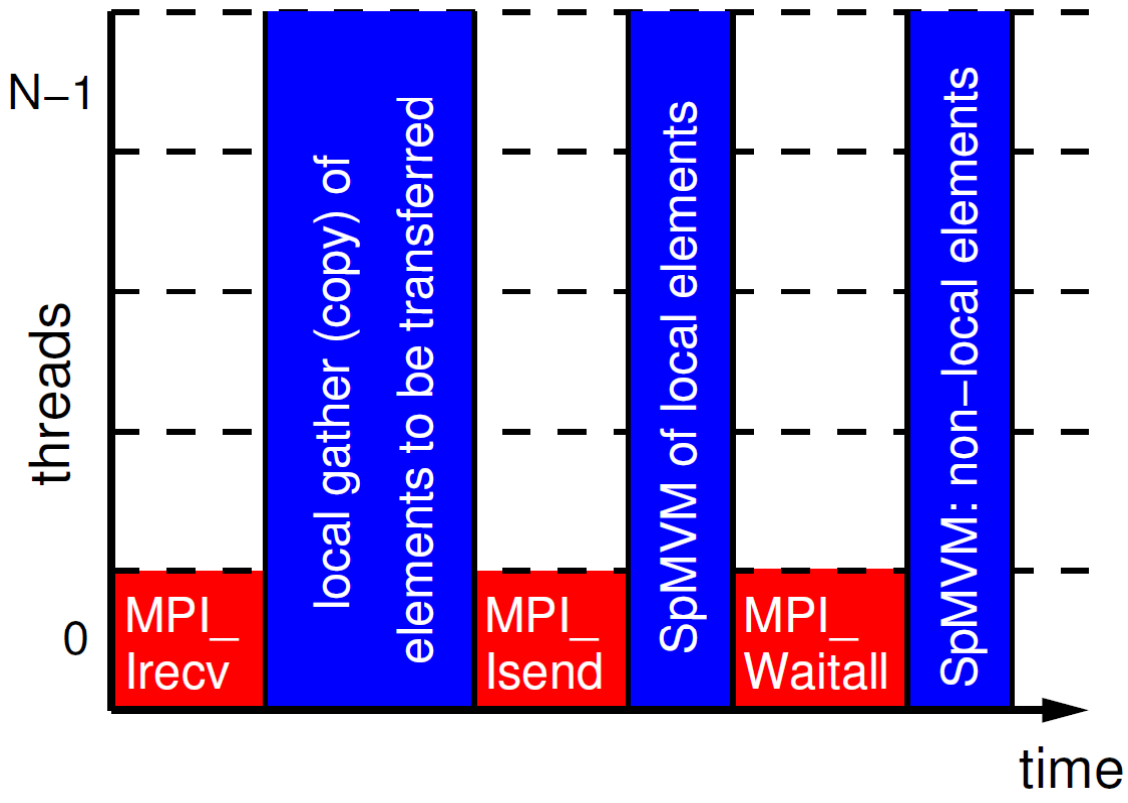
G. Hager, G. Jost, and R. Rabenseifner: *Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes.* In: Proceedings of the Cray Users Group Conference 2009 (CUG 2009), Atlanta, GA, USA, May 4-7, 2009. PDF

Distributed-memory parallelization of spMVM

- **Variant 2: "Vector mode" with naïve overlap ("good faith hybrid")**

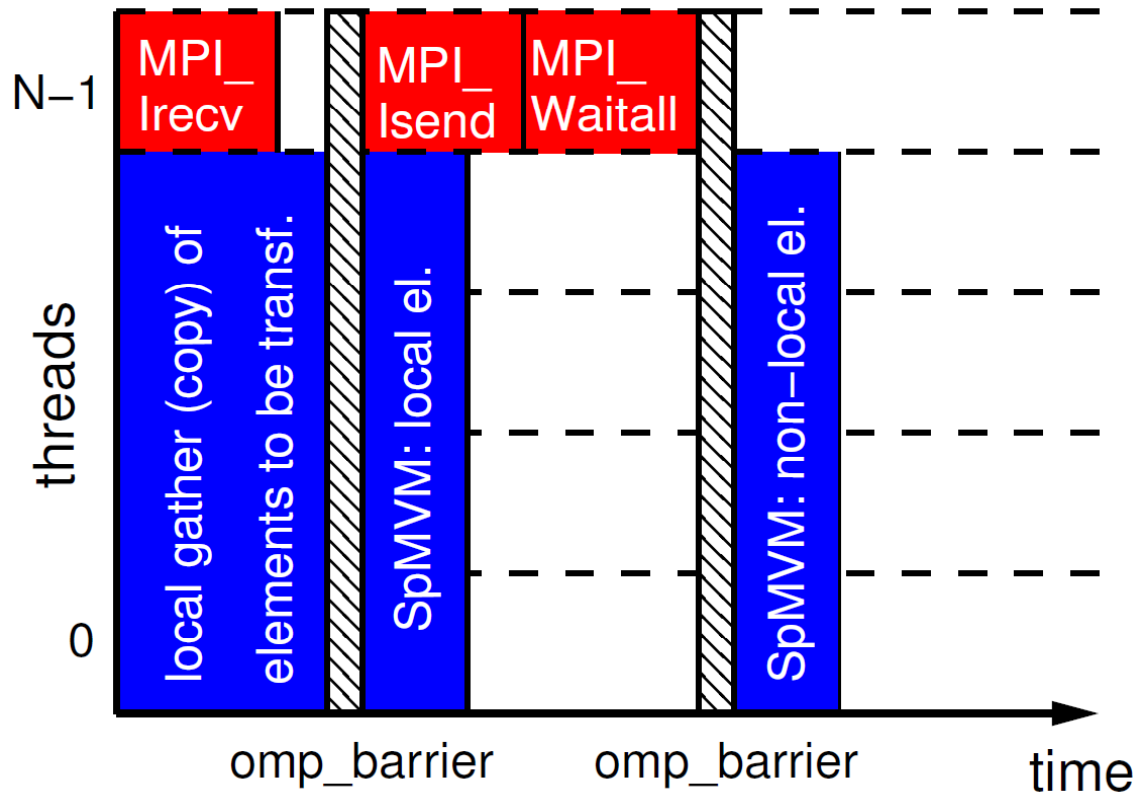
- **Relies on MPI to support asynchronous nonblocking point-to-point**
- **Multithreaded computation (all threads)**

- **Still simple programming**
- **Drawback: Result vector is written twice to memory**
  - modified performance model



Distributed-memory parallelization of spMVM

- **Variant 3: "Task mode" with dedicated communication thread**
- **Explicit overlap, more complex to implement**
- **One thread missing in team of compute threads**
  - But that doesn't hurt here…
  - Using tasking seems simpler but may require some work on NUMA locality
- **Drawbacks**
  - Result vector is written twice to memory
  - No simple OpenMP worksharing (manual, tasking)

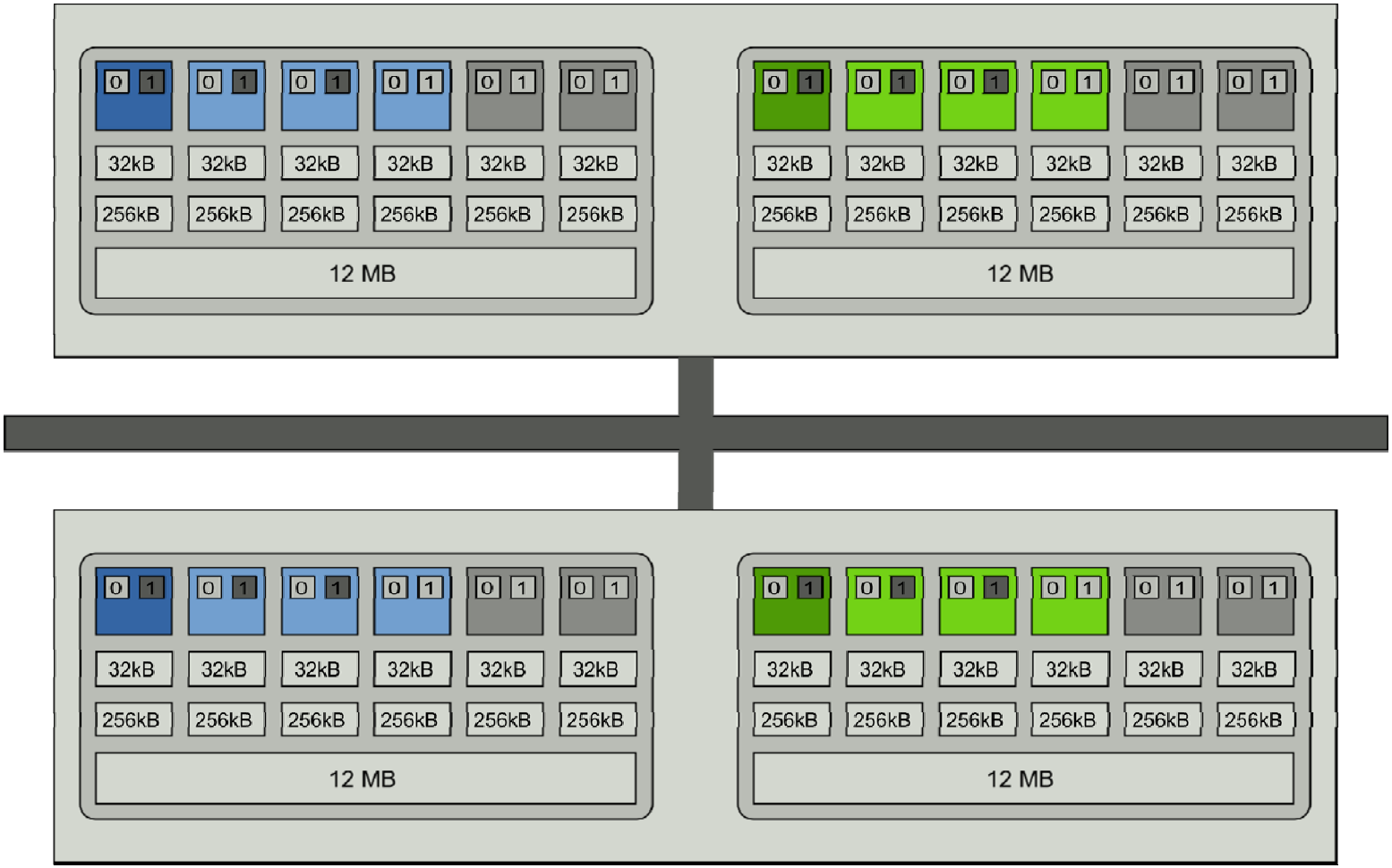


R. Rabenseifner and G. Wellein: *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.* International Journal of High Performance Computing Applications 17, 49-62, February 2003. DOI:10.1177/1094342003017001005

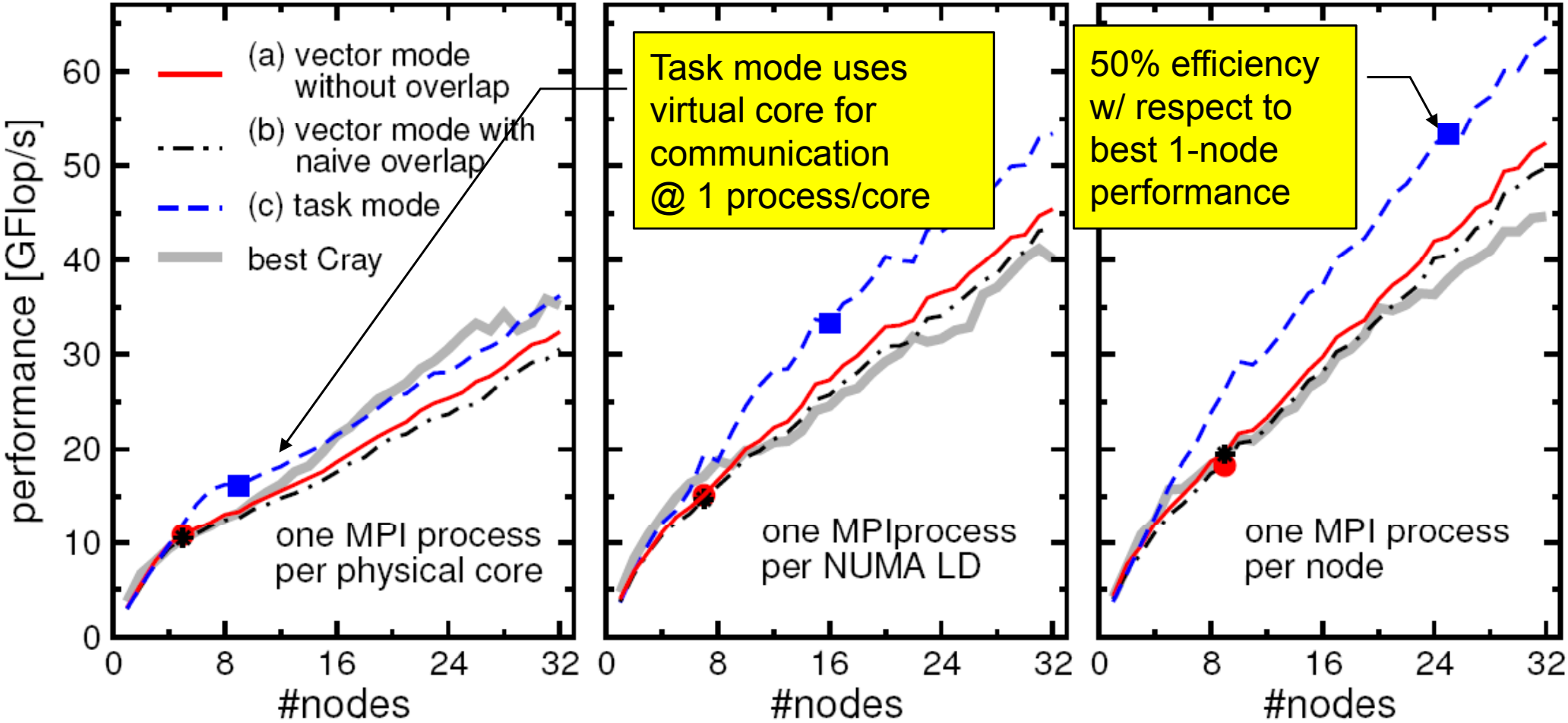
M. Wittmann and G. Hager: *Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems.* Technical report. Preprint:arXiv:1101.0093

Advanced hybrid pinning: One MPI process per socket, communication thread on virtual core (SMT)

```
OMP_NUM_THREADS=5 likwid-mpirun -np 4 -pin S0:0-3,9_S1:0-3,9 ./a.out
```

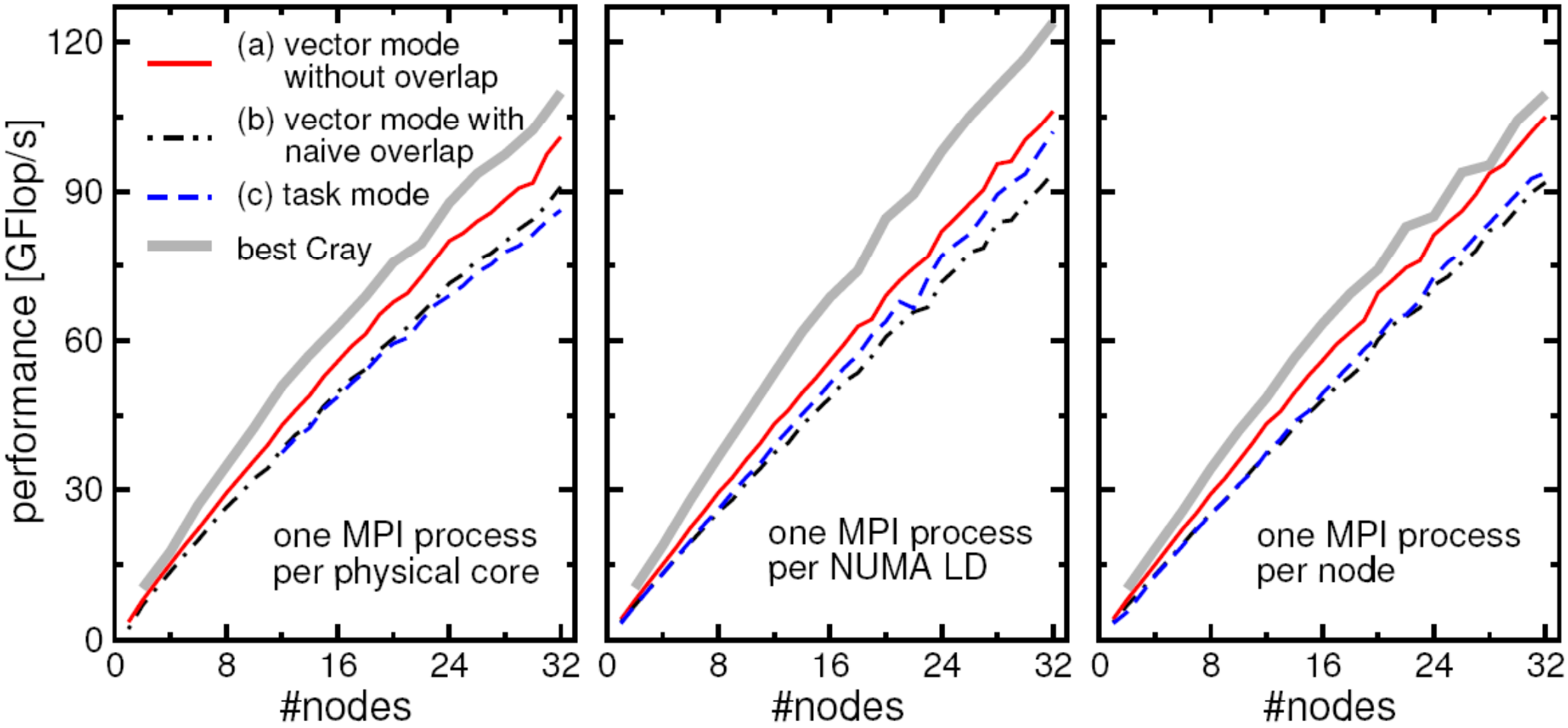


Results HMeP (strong scaling) on Westmere-based QDR IB cluster (vs. Cray XE6)



- Dominated by communication (and some load imbalance for large #procs)
- Single-node Cray performance cannot be maintained beyond a few nodes
- Task mode pays off esp. with one process (12 threads) per node
- Task mode overlap (over-)compensates additional LHS traffic

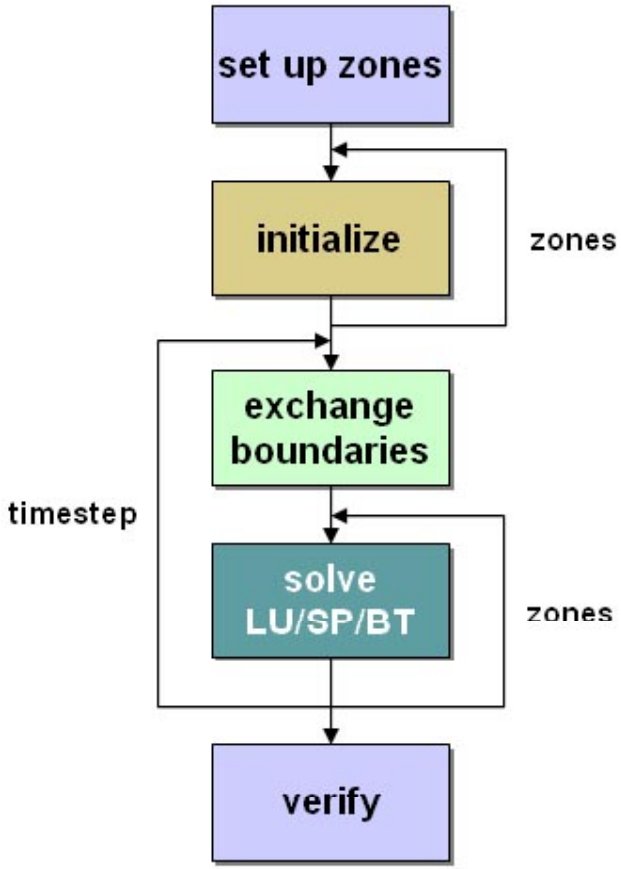
Results sAMG



- **Much less communication-bound**
- **XE6 outperforms Westmere cluster, can maintain good node performance**
- **Hardly any discernible difference as to # of threads per process**
- **If pure MPI is good enough, don't bother going hybrid!**

Case study:
The Multi-Zone NAS Parallel Benchmarks (NPB-MZ)

The Multi-Zone NAS Parallel Benchmarks



	MPI/OpenMP	MLP	Nested OpenMP
Time step	sequential	sequential	sequential
inter-zones	MPI Processes	MLP Processes	OpenMP
exchange boundaries	Call MPI	data copy+ sync.	OpenMP
intra-zones	OpenMP	OpenMP	OpenMP

- Multi-zone versions of the NAS Parallel Benchmarks LU,SP, and BT
- Two hybrid sample implementations
- Load balance heuristics part of sample codes
- www.nas.nasa.gov/Resources/Software/software.html

MPI/OpenMP BT-MZ

```
call omp_set_numthreads (weight)
do step = 1, itmax
  call exch_qbc(u, qbc, nx,…)
```

call mpi_send/recv

```
do zone = 1, num_zones
    if (iam .eq. pzone_id(zone)) then
        call zsolve(u,rsd,…)
      end if
    end do

end do
  ...
```

```
subroutine zsolve(u, rsd,…)
  ...
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP& PRIVATE(m,i,j,k...)
  do k = 2, nz-1
!$OMP DO
    do j = 2, ny-1
      do i = 2, nx-1
        do m = 1, 5
       u(m,i,j,k)=
              dt*rsd(m,i,j,k-1)
        end do
      end do
    end do
!$OMP END DO nowait
  end do
  ...
!$OMP END PARALLEL
```

MPI/OpenMP LU-MZ

```
call omp_set_numthreads (weight)
do step = 1, itmax
  call exch_qbc(u, qbc, nx,…)
```

call mpi_send/recv

```
do zone = 1, num_zones
    if (iam .eq. pzone_id(zone)) then
        call ssor
      end if
    end do

end do
  ...
```

Pipelined Thread Execution in SSOR

```
subroutine  ssor
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP& PRIVATE(m,i,j,k...)
  call sync1 (…)
  do k = 2, nz-1
!$OMP DO
    do j = 2, ny-1
      do i = 2, nx-1
        do m = 1, 5
      rsd(m,i,j,k)=
        dt*rsd(m,i-1,j-1,k-1)
        end do
      end do
    end do
!$OMP END DO nowait
  end do
  call sync2 (…)
  ...
!$OMP END PARALLEL
  ...
```

```
subroutine sync1
 …neigh = iam -1
 do while (isync(neigh) .eq. 0)
!$OMP FLUSH(isync)
 end do
 isync(neigh) = 0
!$OMP FLUSH(isync)
 …
 subroutine sync2
 …
 neigh = iam -1
 do while (isync(neigh) .eq. 1)
!$OMP FLUSH(isync)
 end do
 isync(neigh) = 1
!$OMP FLUSH(isync)
```

"PPP without global sync" – cf. Gauss-Seidel example in OpenMP section!

Benchmark Characteristics

- **Aggregate sizes:**
  - Class D: 1632 x 1216 x 34 grid points
  - Class E: 4224 x 3456 x 92 grid points
- **BT-MZ: (Block tridiagonal simulated CFD application)**
  - Alternative Directions Implicit (ADI) method
  - #Zones: 1024 (D), 4096 (E)
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance
- **LU-MZ: (LU decomposition simulated CFD application)**
  - SSOR method (2D pipelined method)
  - #Zones: 16 (all Classes)
  - Size of the zones identical:
    - no load-balancing required
    - limited parallelism on outer level
- **SP-MZ: (Scalar Pentadiagonal simulated CFD application)**
  - #Zones: 1024 (D), 4096 (E)
  - Size of zones identical
    - no load-balancing required

Expectations:

Pure MPI: Load balancing problems!

Good candidate for MPI+OpenMP

Limited MPI Parallelism: → MPI+OpenMP increases Parallelism

Load-balanced on MPI level: Pure MPI should perform best

Benchmark Architectures

- Sun Constellation (Ranger)
- Cray XT5
- Cray XE6
- IBM Power 6
- Some miscellaneous others

Sun Constellation Cluster Ranger

- **Located at the Texas Advanced Computing Center (TACC), University of Texas at Austin (http://www.tacc.utexas.edu)**
- **3936 Sun Blades, 4 AMD Quad-core 64bit 2.3GHz processors per node (blade), 62976 cores total**
- **InfiniBand Switch interconnect**
- **Sun Blade x6420 Compute Node:**
  - 4 Sockets per node
  - 4 cores per socket
  - HyperTransport System Bus
  - 32GB memory
- http://services.tacc.utexas.edu/index.php/ranger-user-guide

- **Compilation:**
  - PGI pgf90 7.1
  - `mpif90 -tp barcelona-64 -r8 -mp`

Enable OpenMP!

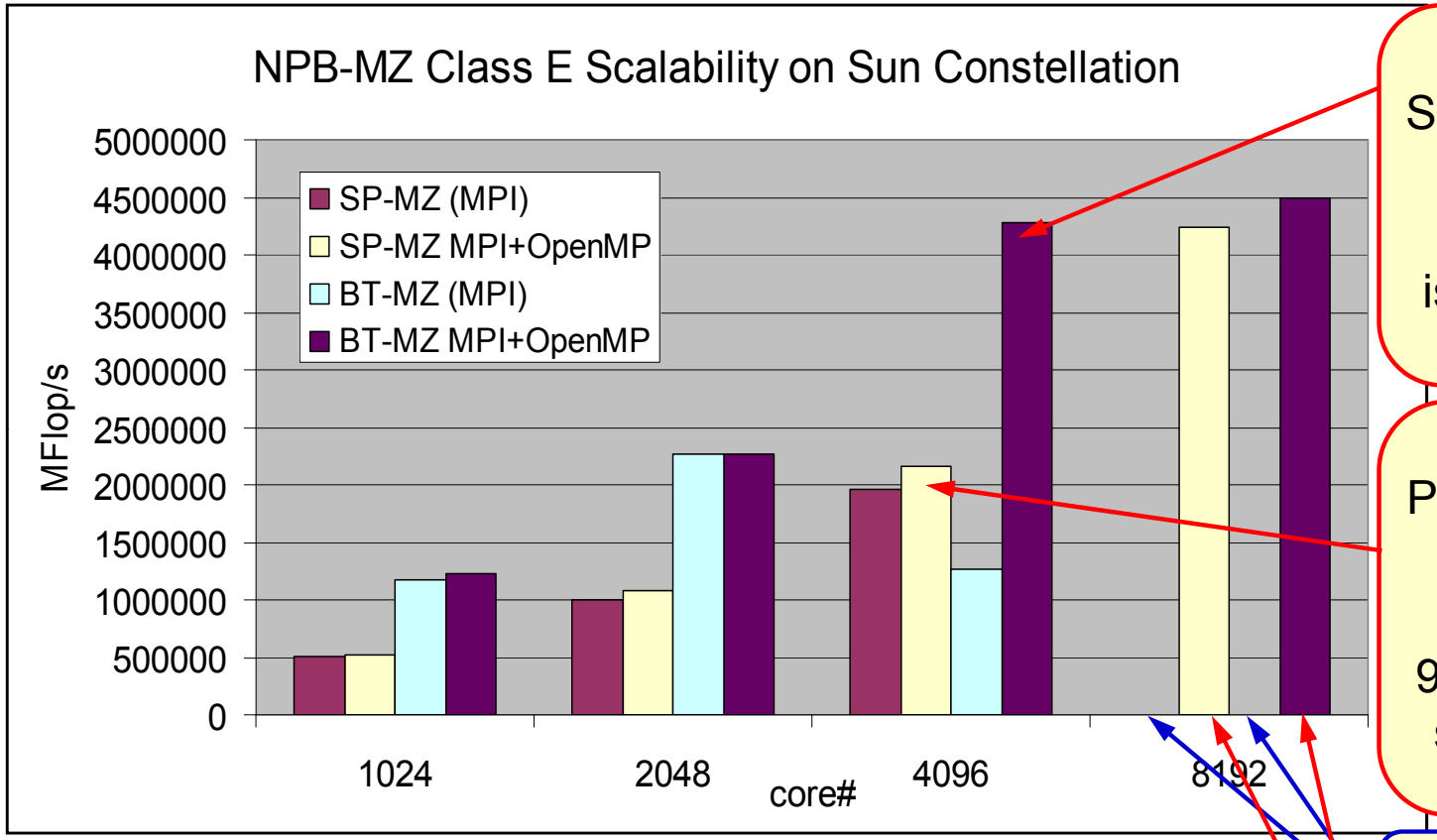
- **Cache optimized benchmarks**
- **Execution:**
  - MPI is MVAPICH
  - `setenv OMP_NUM_THREADS \`
    `nthreads`
  - `ibrun tacc_affinity bt-mz.exe`

Set number of threads!

- **numactl controls**
  - Socket affinity: select sockets to run
  - Core affinity: select cores within socket
  - Memory policy:where to allocate memory
  - http://www.halobates.de/numaapi3.pdf

Control process and memory affinity!

NPB-MZ Class E Scalability on Ranger



BT

Significant improvement (235%):

Load balancing issues solved with MPI+OpenMP

SP

Pure MPI is already load-balanced.

But hybrid 9.6% faster, due to smaller message rate at NIC

8192 max # of MPI procs

Hybrid:

SP: still scales

BT: does not scale

- **Performance in Mflop/s**
- **We report pure MPI and the highest achieved hybrid performance**
- **MPI/OpenMP outperforms pure MPI**
- **Use of numactl essential to achieve scalability**

Numactl – Pitfalls: Using Threads across Sockets

bt-mz.1024x8 yields best workload balance BUT:

```
#$ -pe 2way 8192    # in batch script!

export OMP_NUM_THREADS=8 # in batch script
```

In tacc_affinity:

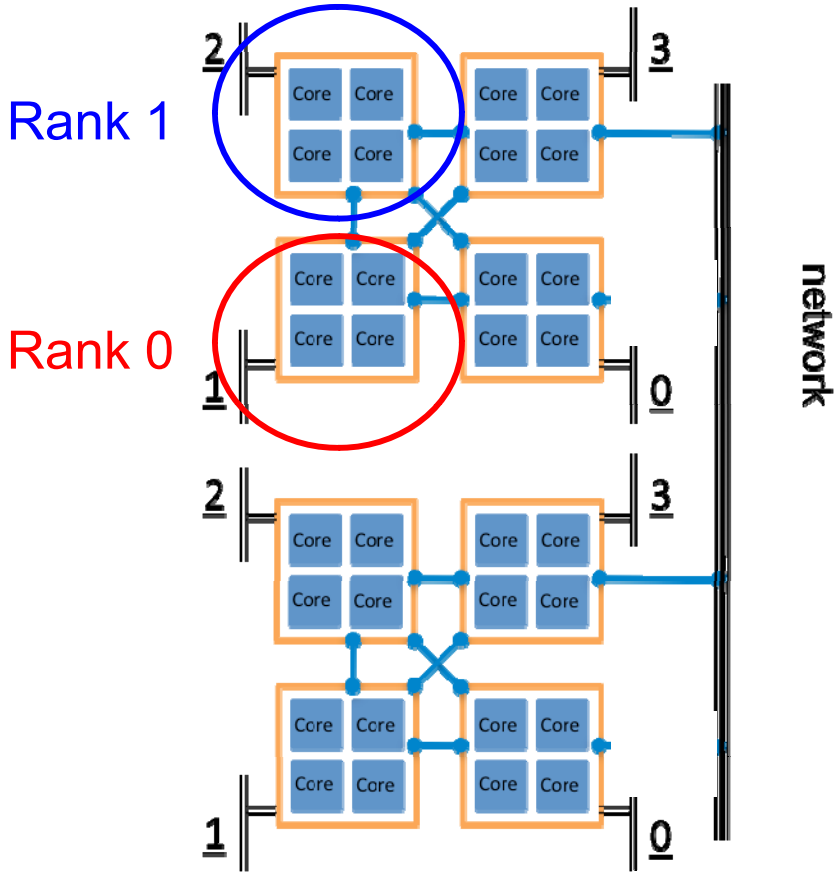
```
my_rank=$PMI_RANK
local_rank=$(( $my_rank % $myway ))
numnode=$(( $local_rank + 1 ))
```

<u>In original tacc_affinity:</u>

```
numactl -N $numnode -m $numnode $*
```

Bad performance!

- Processes bound to just one socket
- Each process runs 8 threads on 4 cores
- Memory allocated on one socket



Numactl – Pitfalls: Using Threads across Sockets

bt-mz.1024x8

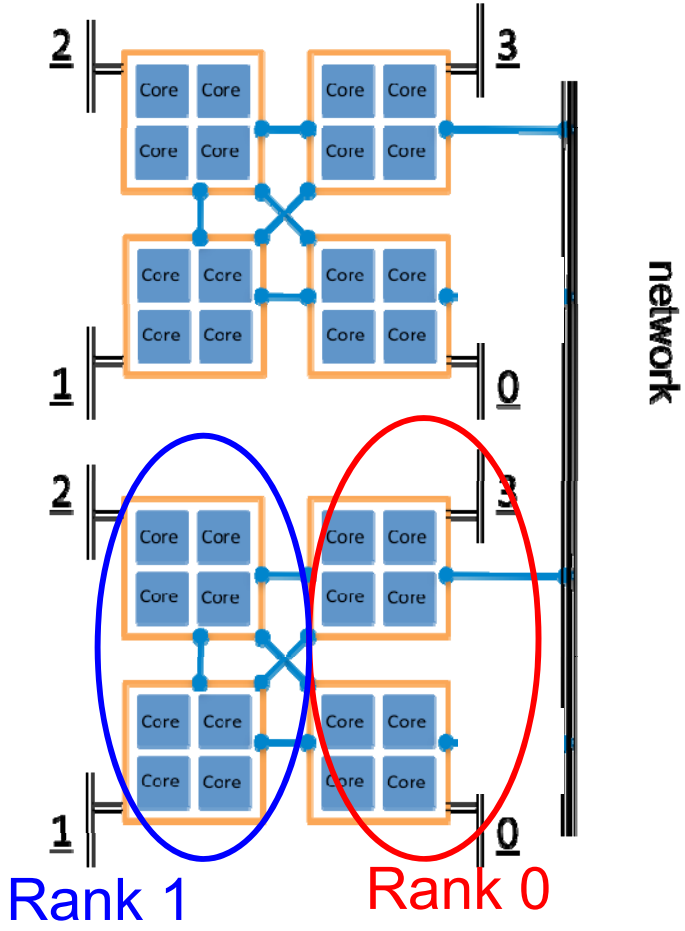
```
export OMP_NUM_THREADS=8

my_rank=$PMI_RANK
local_rank=$(( $my_rank % $myway ))
numnode=$(( $local_rank + 1 ))
```

Original:

```
numactl -N $numnode -m $numnode $*
```

Modified:

```
if [ $local_rank -eq 0 ]; then
    numactl -N 0,3 -m 0,3 $*
else
    numactl -N 1,2 -m 1,2 $*
fi
```



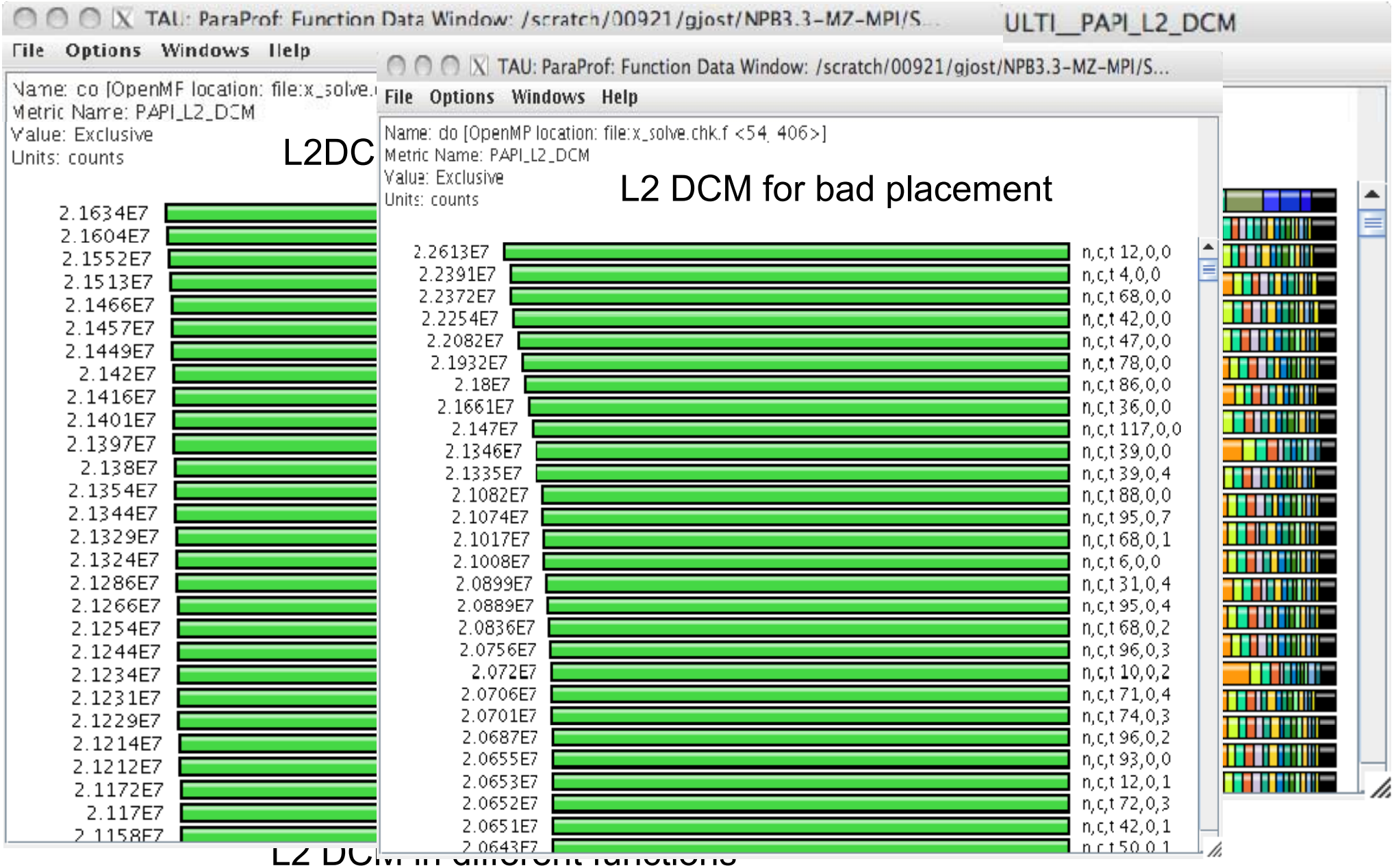
Achieves Scalability!

- Process uses cores and memory across 2 sockets
- Suitable for 8 threads

Using TAU on Ranger

- `module load papi kojak pdtoolkit tau`
- **Compilation:**
  - Use a TAU Makefile which supports profiling of MPI and OpenMP, eg:
  - `export TAU_MAkEFILE=$TAU_LIB/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari`
  - Use `tau_f90.sh` to compile and link.
- **Execution :**
  - `export COUNTER1=GET_TIME_OF_DAY`
  - `export COUNTER2=PAPI_FP_OPS`
  - `export COUNER3=PAPI_L2_DCM`
  - `ibrun a.out /bt-mz.exe`
- **Generates performance statisitics:**
  - `MULTI_LINUX_TIMERS`
  - `MULTI_PAPI_FP_OPS`
  - `MULTI_PAPI_L2_DCM`
- **View with `paraprof` (GUI) or `pprof` (text based)**

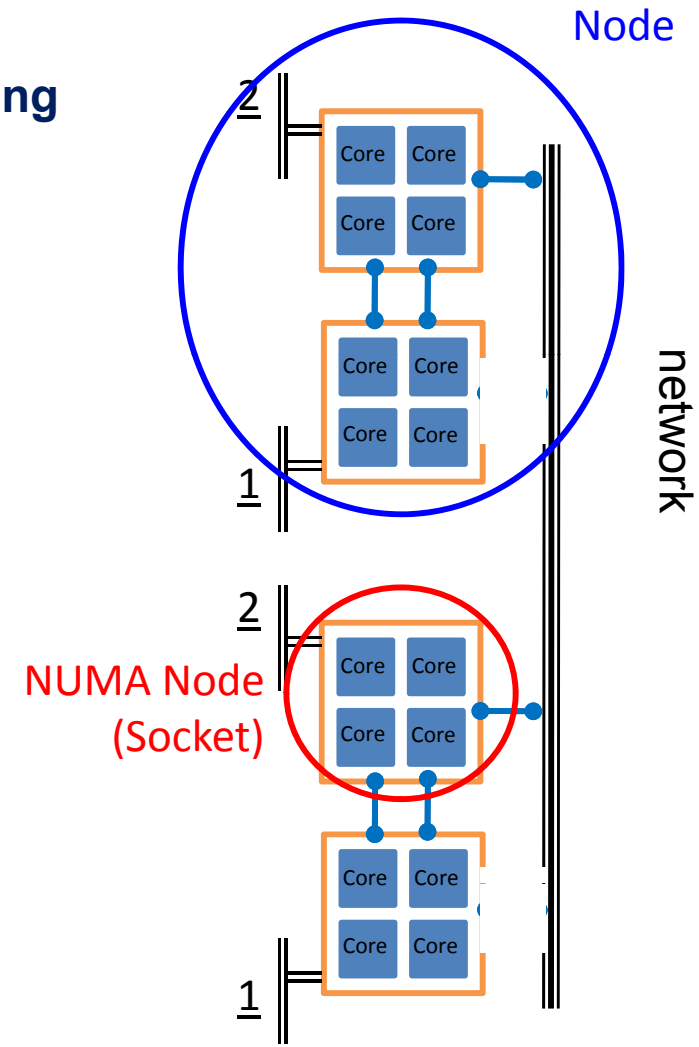
BT-MZ TAU Performance Statistics



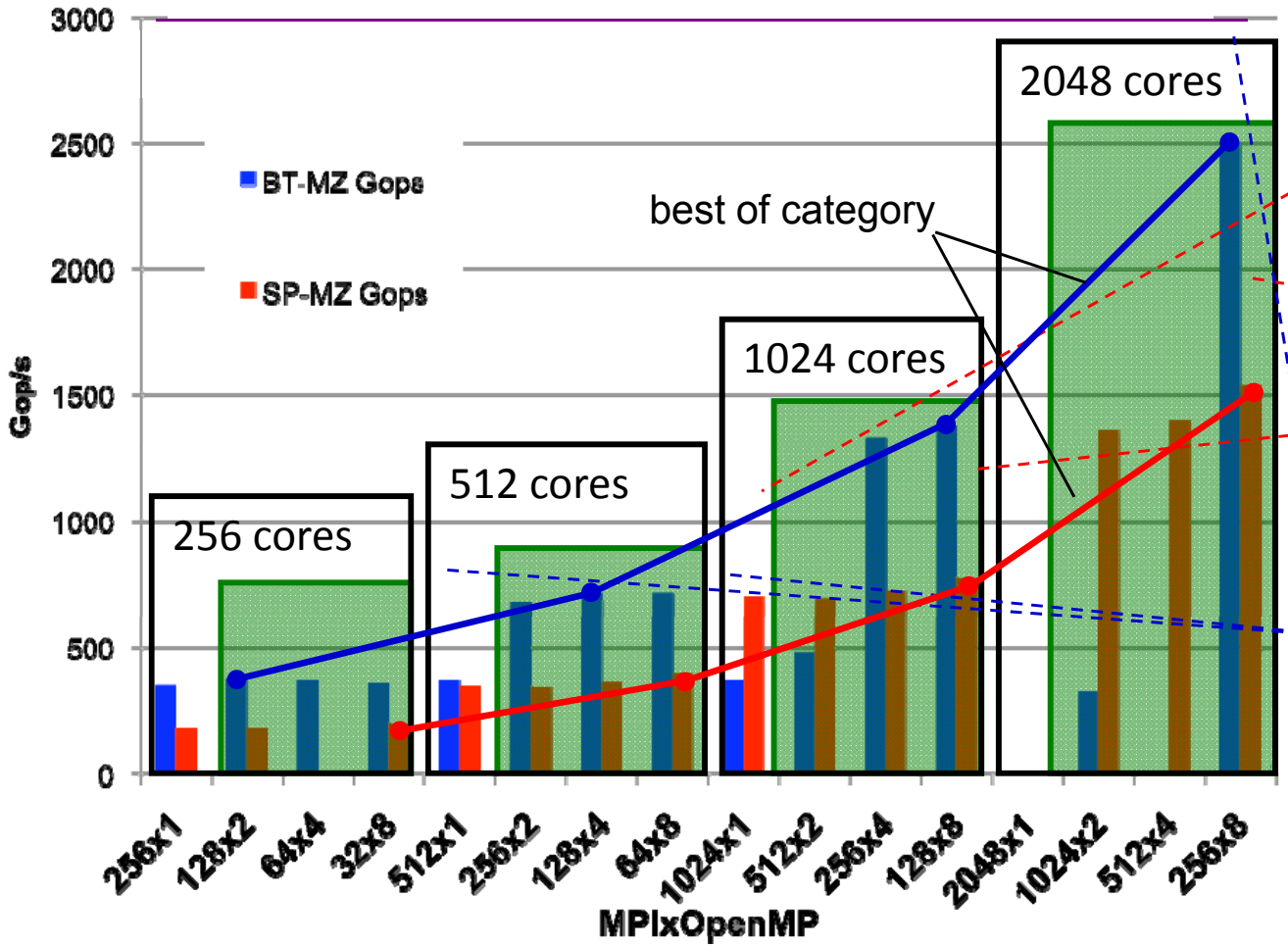
L2 DCM in different functions

Cray XT5

- **Results obtained by the courtesy of the HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)**
- **Cray XT5 is located at the Arctic Region Supercomputing Center (ARSC) (http://www.arsc.edu/resources/pingo)**
  - 432 Cray XT5 compute nodes with
    - 32 GB of shared memory per node (4 GB per core)
    - 2 quad core 2.3 GHz AMD Opteron processors per node.
    - 1 Seastar2+ Interconnect Module per node.
  - Cray Seastar2+ Interconnect between all compute and login nodes



Cray XT5: NPB-MZ Class D Scalability



Results reported for Class D on 256-2048 cores

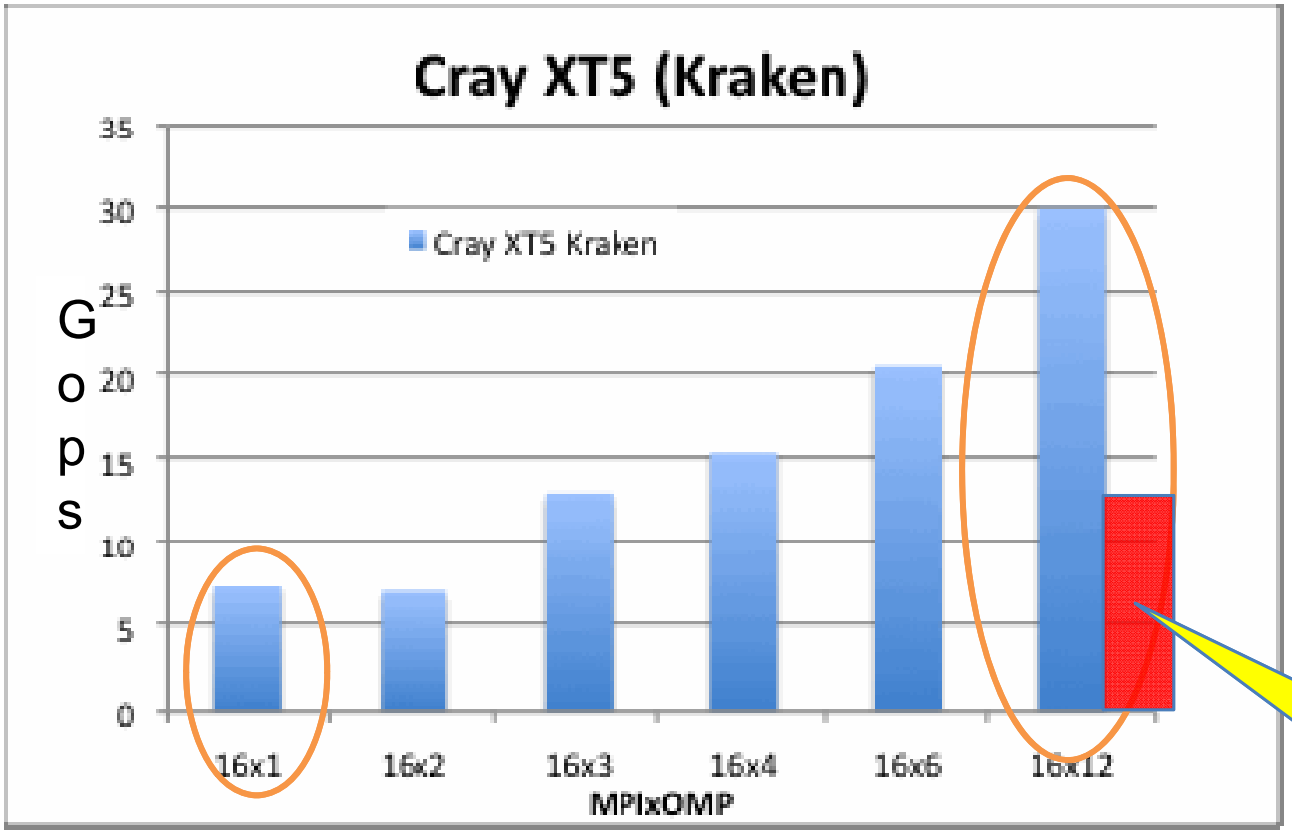
Expected: #MPI processes limited to 1024

- SP-MZ pure MPI scales up to 1024 cores
- SP-MZ MPI/OpenMP scales to 2048 cores
- SP-MZ MPI/OpenMP outperforms pure MPI for 1024 cores
- BT-MZ MPI does not scale
- BT-MZ MPI/OpenMP scales to 2048 cores, outperforms pure MPI

Unexpected!

Expected: Load Imbalance for pure MPI

LU-MZ Class D



- Kraken: Cray XT5 TeraGrid system at NICS/ University of Tennessee
- Two 2.6 GHz six-core AMD Opteron processors (Istanbul) per node
- 12-way SMP system
- 16 GB of memory per node
- Cray SeaStar2+ interconnect
- Intel compiler available!

16x1 on 192 cores:
2x speed-up vs 16x1 on 16 cores
BUT: 11 idle cores per node!

- **Pure MPI limited to 16 processes**
- **Hybrid MPI/OpenMP improves scalability considerably**

CrayPat Performance Analysis (1)

- `module load perftools`

- **Compilation (PrgEnv-pgi):**
  - `ftn -fastsse -tp barcelona-64 -r8 -mp=nonuma,[trace ]`
- **Instrument:**
  - `pat_build -w [ -T TraceOmp], -g mpi,omp bt.exe bt.exe.inst`
- **Execution :**
  - `export PAT_RT_HWPC={0,1,2,..}`
  - `export OMP_NUM_THREADS=4`
  - `aprun -n NPROCS -S 1 -d 4 ./bt.exe.inst`
- **Generate report:**
  -
```
pat_report \
-O load_balance,thread_times,program_time,mpi_callers \
-O profile_pe.th <tracefile>
```

CrayPat Performance Analysis (2)

- **How to obtain guidance for profiling instrumentation:**

1. Sampling-based profile with instrumentation suggestions:
   ```
   pat_build  -O apa a.out
   ```

2. Execution:
   ```
   aprun  -n NPROCS -S 1 -d 4  ./a.out+apa
   ```

3. Generate report:
   ```
   pat_report tracefile.xf
   ```

4. This will produce a file **`tracefile.apa`** with instrumentation suggestions

Cray XT5: BT-MZ 32x4 Function Profile

```
42
43  !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(n,m,k,i,j,ksize)
44  !$OMP&  SHARED(dz5,dz4,dz3,dz2,dz1,tz2,tz1,dt,c1345,c4,c3,con43,c3c4,c1,
45  !$OMP&         c2,nx,ny,nz)
46        ksize = nz-1
47
48  c---------------------------------------------------------------------
49  c     Compute the indices for storing the block-diagonal matrix;
50  c     determine c (labeled f) and s jacobians
51  c---------------------------------------------------------------------
52  !$OMP DO
53        do j = 1, ny-2
54           do i = 1, nx-2
55              do k = 0, ksize
56
57                 tmp1 = 1.d0 / u(1,i,j,k)
58                 tmp2 = tmp1 * tmp1
59                 tmp3 = tmp1 * tmp2
60
61                 fjac(1,1,k) = 0.d0
62                 fjac(1,2,k) = 0.d0
63                 fjac(1,3,k) = 0.d0
64                 fjac(1,4,k) = 1.d0
65                 fjac(1,5,k) = 0.d0
```

```
nt=1
n
ead='HIDE'
----------
e_.LOOP@li.43
e_.LOOP@li.43
e_.LOOP@li.46
e_rhs_.MASTER@li.291
e_rhs_.LOOP@li.187
e_rhs_.LOOP@li.53
e_rhs_.LOOP@li.76
e_rhs_.LOOP@li.28
e_rhs_.LOOP@li.297
lize_.LOOP@li.40
e_rhs_.LOOP@li.381
||   1.2% | 0.016753 | 0.003972 |  19.3% |   168 |add_.LOOP@li.22
||=================================================================
|   2.1% | 0.030491 |       -- |     -- |  1040 |MPI
||-----------------------------------------------------------------
||   1.8% | 0.026193 | 0.111613 |  81.6% |   105 |mpi_waitall_
|=================================================================
```

Cray XT5: BT-MZ Load Balance 32x4 vs 128x1

```
Table 2:  Load Balance across PE's by FunctionGroup

 Time % |     Time | Calls |Experiment=1
        |          |       |Group
        |          |       | PE[mmm]
        |          |       |  Thread

 100.0% | 1.782603 | 18662 |Total
|-------------------------------------
|  86.1% | 1.535163 |  7783 |USER
||------------------------------------
||   2.7% | 1.535987 |  6813 |pe.0
|||-----------------------------------
3||   0.7% | 1.535987 |  6188 |thread.1
3||   0.7% | 1.535871 |  6188 |thread.3
3||   0.7% | 1.535829 |  6188 |thread.2
3||   0.7% | 1.466954 |  6813 |thread.0
|||===================================
||   2.7% | 1.535147 |  7783 |pe.18
|||-----------------------------------
3||   0.7% | 1.535147 |  7072 |thread.1
3||   0.7% | 1.534995 |  7072 |thread.3
3||   0.7% | 1.534968 |  7072 |thread.2
3||   0.6% | 1.290502 |  7783 |thread.0
|||===================================
||   2.7% | 1.534239 |  7783 |pe.16
|||-----------------------------------
3||   0.7% | 1.534239 |  7072 |thread.1
3||   0.7% | 1.534101 |  7072 |thread.3
3||   0.7% | 1.534076 |  7072 |thread.2
3||   0.6% | 1.268085 |  7783 |thread.0
||===================================
```

bt-mz-C.32x4

```
Table 2:  Load Balance across PE's by FunctionGroup

 Time % |      Time | Calls |Group
        |           |       | PE[mmm]

 100.0% | 24.277514 | 38258 |Total
|-------------------------------------
|  54.2% | 13.166225 |  4545  MPI
||------------------------------------
||   0.5% | 16.454993 |  4846 |pe.91
||   0.5% | 14.058598 |  2434 |pe.29
||   0.0% |  0.285479 |  2434 |pe.0
||====================================
|  44.3% | 10.094000 | 17003  USER
||------------------------------------
||   0.7% | 23.205797 |  9093 |pe.0
||   0.5% | 10.084200 | 26875 |pe.110
||   0.3% |  8.070997 | 17983 |pe.91
|=====================================
```

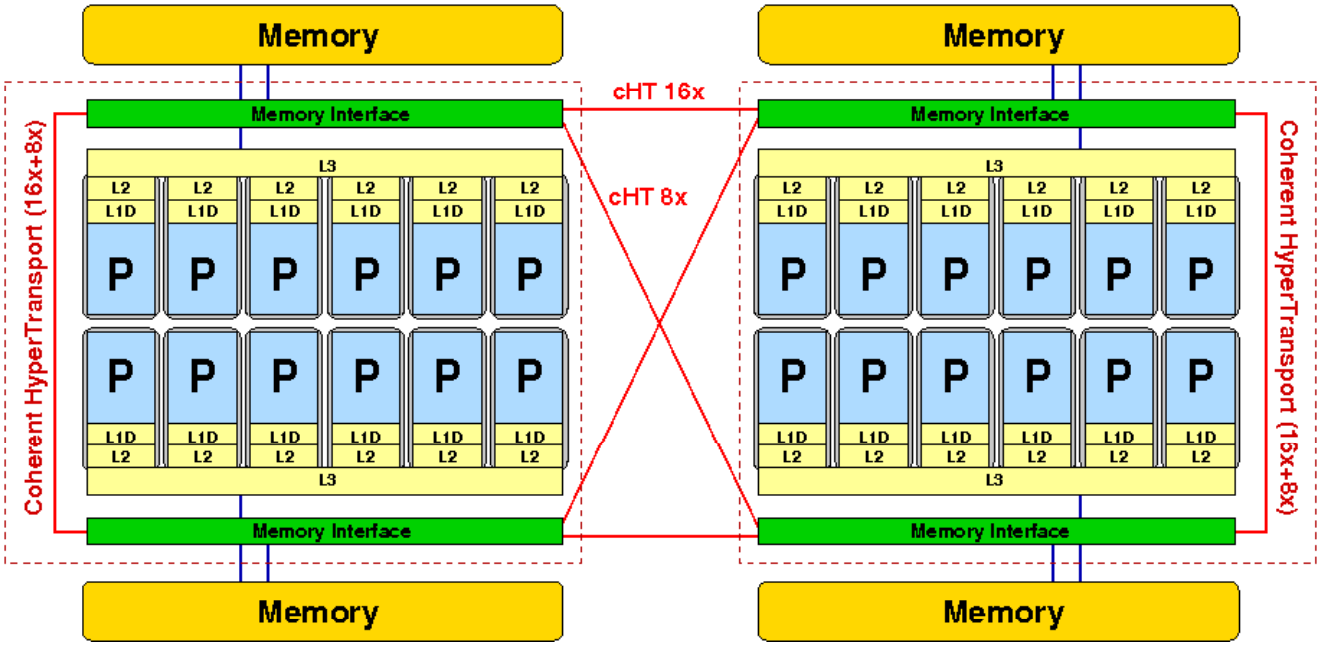
bt-mz-C.128x1

- maximum, median, minimum PE are shown
- bt-mz.C.128x1 shows large imbalance in User and MPI time
- bt-mz.C.32x4 shows well balanced times

Cray XE6 (Hector)

- Located at EPCC, Edinburgh, Scotland, UK National Supercomputing Services, Hector Phase 2b (http://www.hector.ac.uk)
- 1856 XE6 compute nodes.
- Around 373 Tflop/s theoretical peak performance
- Each node contains two AMD 2.1 GHz 12-core processors for a total of 44,544 cores
- 32 GB of memory per node
- 24-way shared memory system, four ccNUMA domains
- Cray Gemini interconnect

Node layout:



Graphical likwid-topology output Cray XE6 (Hector)

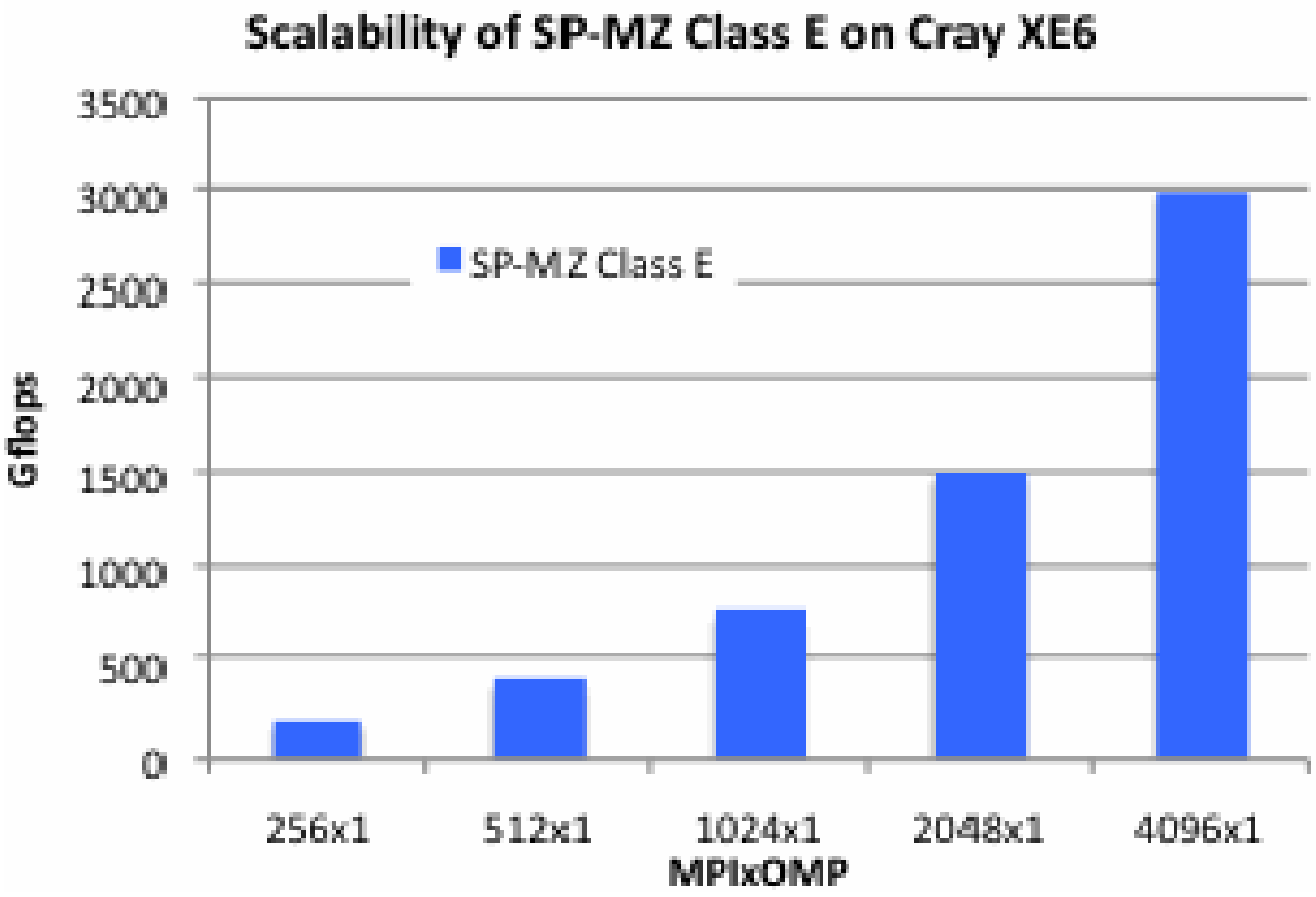
```
CPU type:          AMD Magny Cours processor
Hardware Thread Topology
Sockets:                    2
Cores per socket:           12
Threads per core:           1
```

no SMT

4 NUMA domains

```
Socket 0:
+-------------------------------------------------------------------------------+
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| |   0   | |   1   | |   2   | |   3   | |   4   | |   5   | |   6   | |   7   | |   8   | |   9   | |  10   | |  11   | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------------------------------------------------------+ +-------------------------------------------------------+ |
| |                          5MB                          | |                          5MB                          | |
| +-------------------------------------------------------+ +-------------------------------------------------------+ |
+-------------------------------------------------------------------------------+
Socket 1:
+-------------------------------------------------------------------------------+
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| |  12   | |  18   | |  19   | |  20   | |  21   | |  22   | |  23   | |  13   | |  14   | |  15   | |  16   | |  17   | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | | 64kB  | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | | 512kB | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------------------------------------------------------+ +-------------------------------------------------------+ |
| |                          5MB                          | |                          5MB                          | |
| +-------------------------------------------------------+ +-------------------------------------------------------+ |
+-------------------------------------------------------------------------------+
```

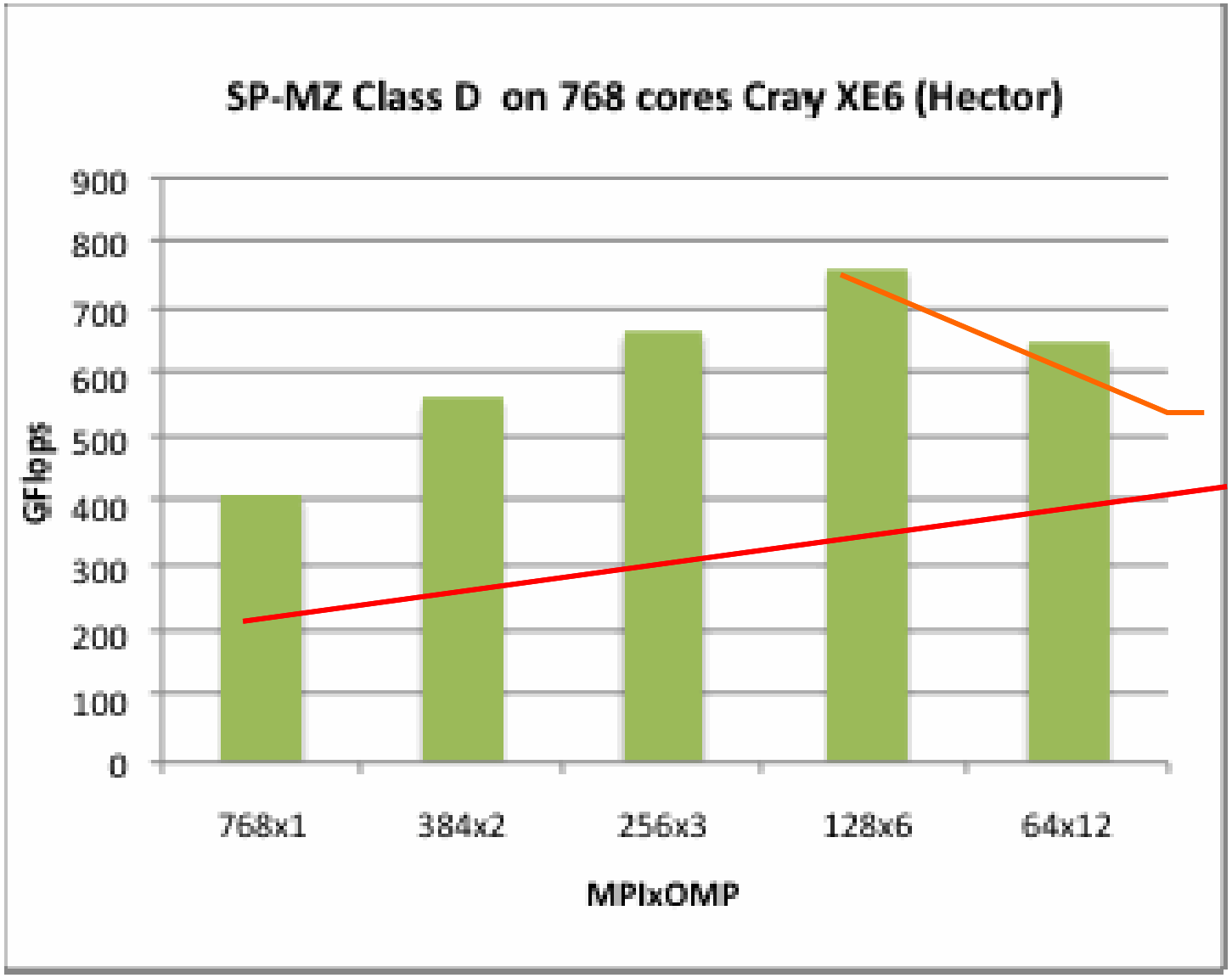
SP-MZ Class E Pure MPI Scalability on Cray XE6



Good Scalability for Pure MPI!
No need for hybrid approach

Observations:
- #used cores divides #zones
- Not all allocated cores are used
  - 24-way nodes → <24 idle cores

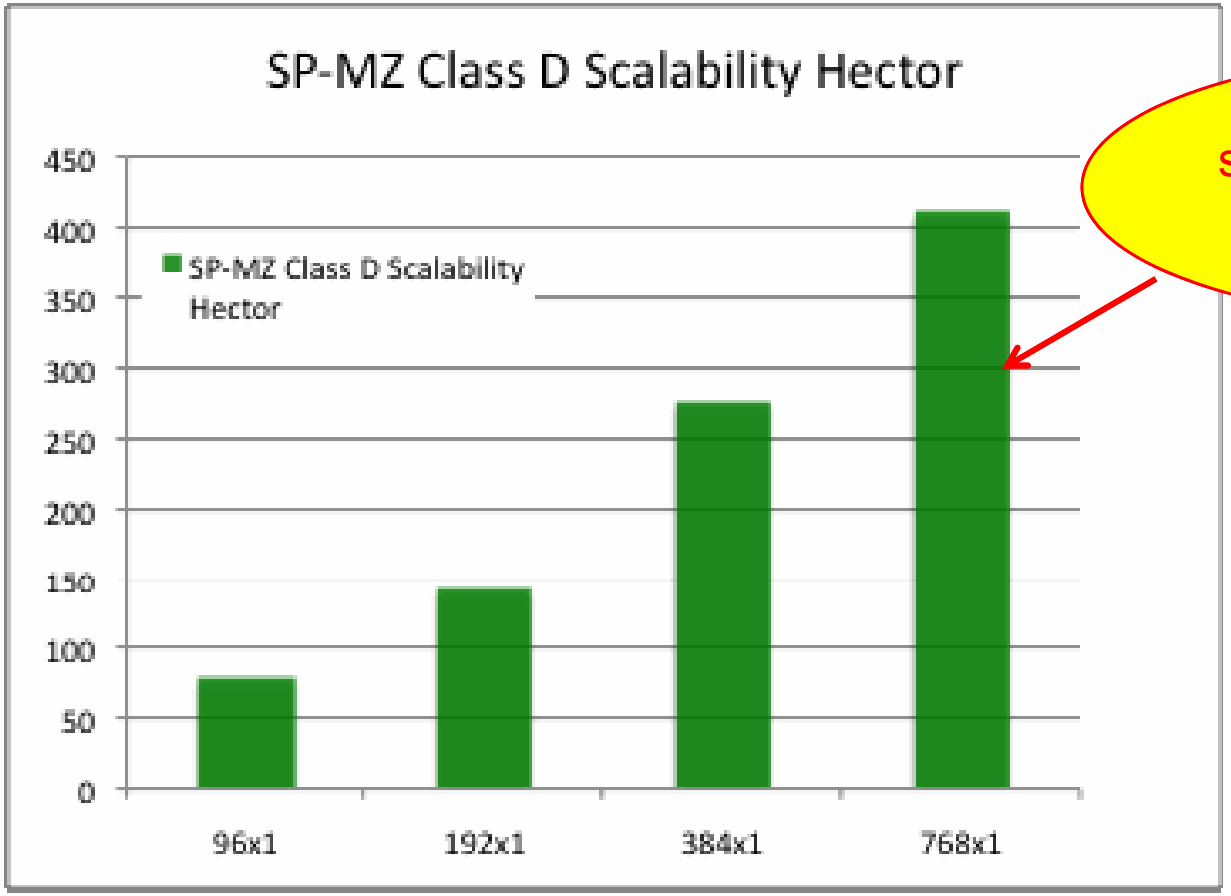
SP-MZ Class D Hybrid MPI/OpenMP Performance Cray XE6



- #cores does not divide #zones!
- Hybrid approach yields performance gain due to better load balancing

SP-MZ Class D Hybrid MPI/OpenMP Scalability Cray XE6

SKIPPED



Pure MPI does not scale from 384 to 768. Due to bad load balancing

Craypat Statistics for SP-MZ Class D

SKIPPED

768 MPI procs

```
MPI Message Stats by Caller
    MPI Msg |    MPI | MsgSz | 16B<= | 256B<= | 64KB<= | 1MB<= |Experiment=1
      Bytes |    Msg |  <16B | MsgSz |  MsgSz |  MsgSz | MsgSz |Function
            | Count  | Count | <256B |   <4KB |   <1MB | <16MB | Caller
  2616644.0 |    6.1 |   1.0 |   0.2 |    0.2 |    3.7 |   0.9 |Total
|-----------------------------------------------------------------------
|  2616533.0 |    4.6 |    -- |    -- |     -- |    3.7 |   0.9 |MPI_ISEND
|            |        |       |       |        |        |       | exch_qbc_
3            |        |       |       |        |        |       |  MAIN_
||||---------------------------------------------------------------------
4||| 26329600.0 |   44.0 |    -- |    -- |     -- |   33.0 |  11.0 |pe.33
4|||        0.0 |     -- |    -- |    -- |     -- |     -- |    -- |pe.610
4|||        0.0 |     -- |    -- |    -- |     -- |     -- |    -- |pe.242
||||=====================================================================
==========
```

384 MPI procs

```
 MPI Msg |    MPI | MsgSz | 16B<= | 256B<= | 4KB<= | 64KB<= |Experiment=1
     Bytes |    Msg |  <16B | MsgSz |  MsgSz | MsgSz |  MsgSz |Function
           | Count  | Count | <256B |   <4KB | <64KB |   <1MB | Caller
 6156152.0 |   57.8 |   8.0 |   2.0 |    2.0 |   3.7 |   42.2 |Total
|-----------------------------------------------------------------------
| 6152960.0 |   45.8 |    -- |    -- |     -- |   3.7 |   42.2 |MPI_ISEND
|           |        |       |       |        |       |        | exch_qbc_
3           |        |       |       |        |       |        |  MAIN_
||||---------------------------------------------------------------------
4||| 7180800.0 |   44.0 |    -- |    -- |     -- |    -- |   44.0 |pe.127
4||| 7180800.0 |   55.0 |    -- |    -- |     -- |  11.0 |   44.0 |pe.54
4||| 4421120.0 |   44.0 |    -- |    -- |     -- |  22.0 |   22.0 |pe.4
||||
```

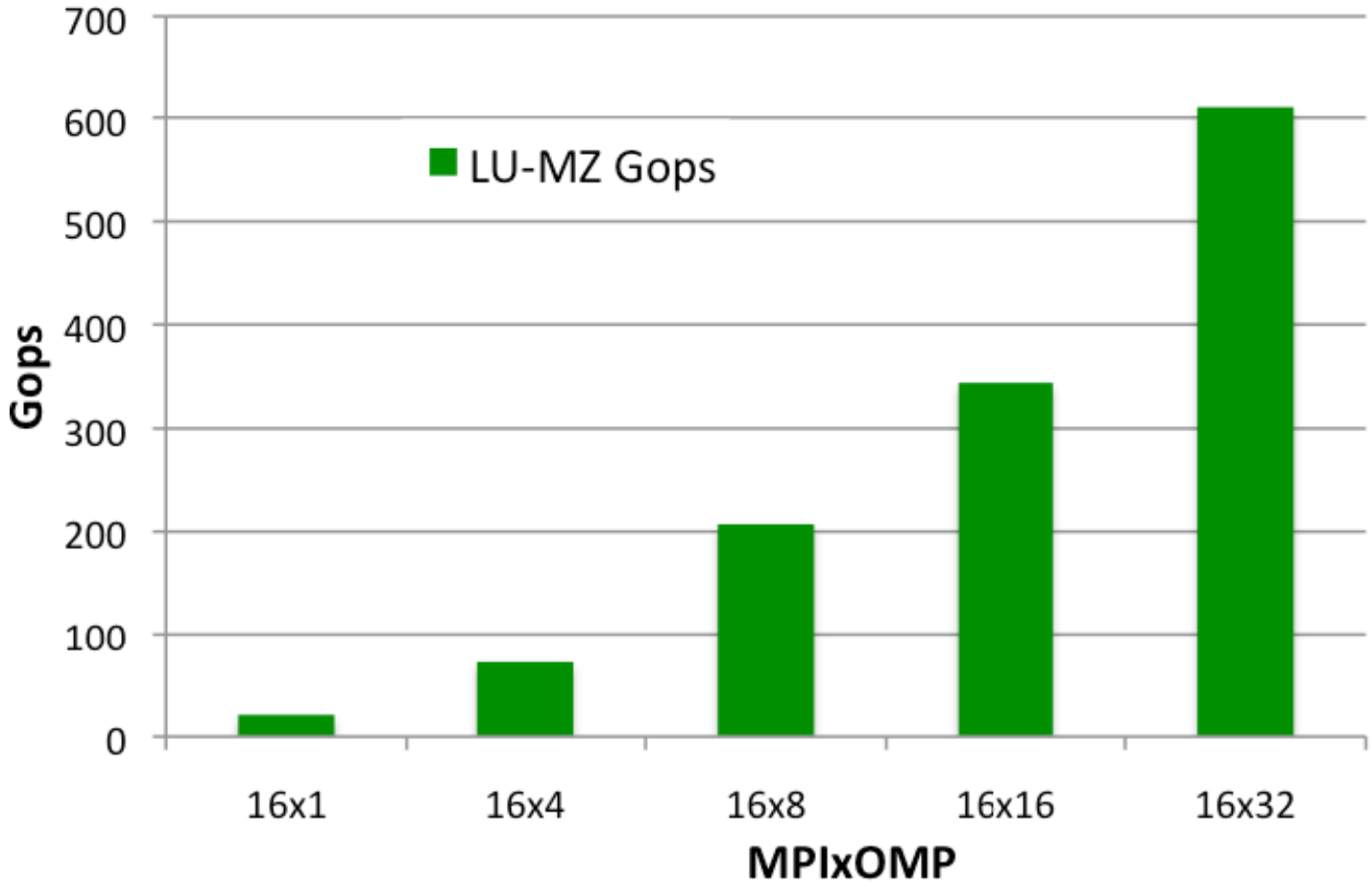
IBM Power 6

- **Results obtained by the courtesy of the HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)**
- **The IBM Power 6 System is located at (http://www.navo.hpc.mil/davinci_about.html)**
- **150 Compute Nodes**
- **32 4.7 GHz Power6 Cores per Node (4800 cores total)**
- **64 GBytes of memory per node**
- **QLOGIC Infiniband DDR interconnect**
- **IBM MPI: MPI 1.2 + MPI-IO**
  - `mpxlf_r -O4 -qarch=pwr6 -qtune=pwr6 -qsmp=omp`

Flag was essential to achieve full compiler optimization in presence of OMP directives!

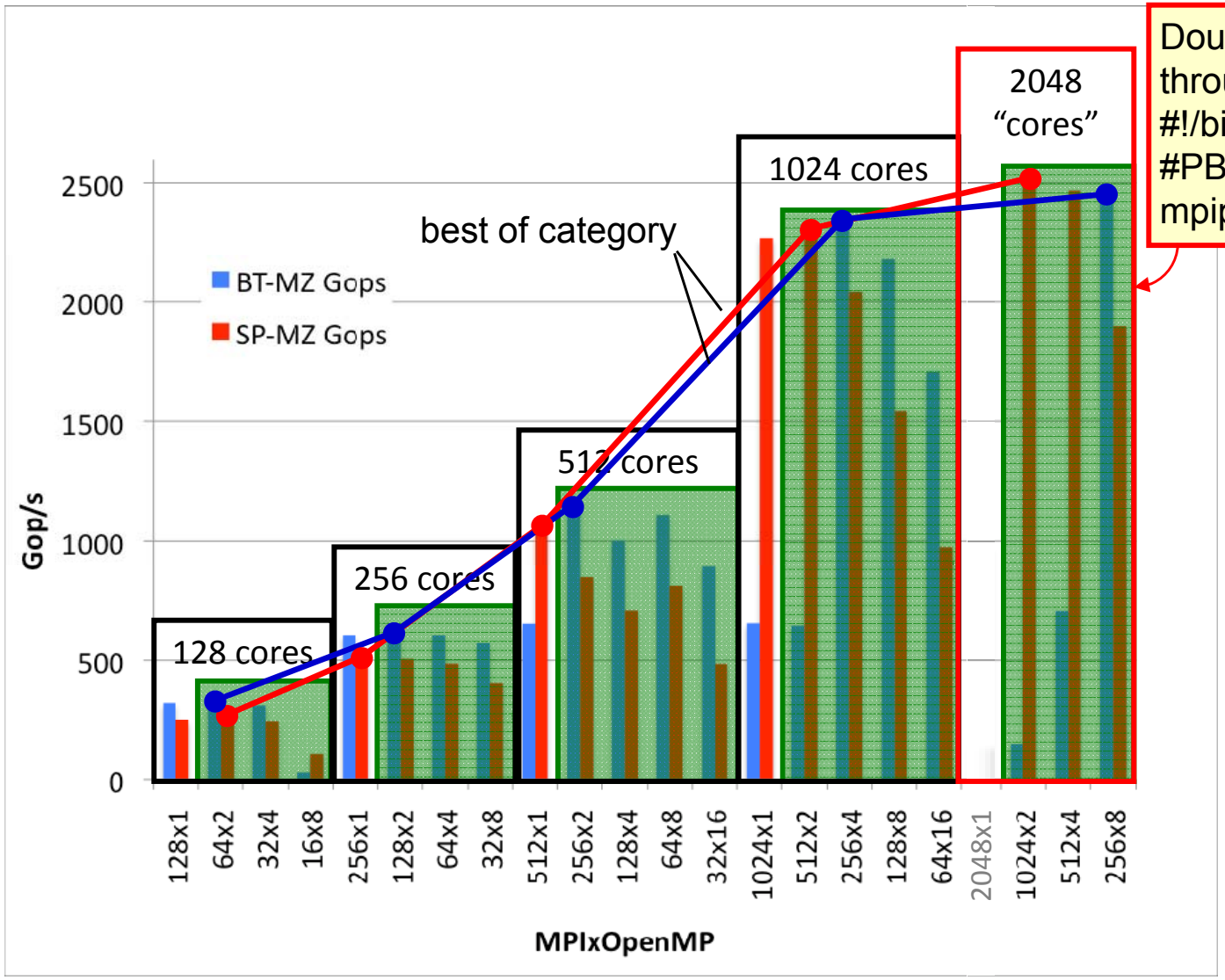
- **Execution:**
  - `poe launch $PBS_O_WORKDIR/sp.C.16x4.exe`

LU-MZ Class D on Power6



- LU-MZ significantly benefits from hybrid mode:
  - Pure MPI limited to 16 cores, due to #zones = 16

NPB-MZ Class D on IBM Power 6: Exploiting SMT for 2048 Core Results



Doubling the number of threads through hyperthreading (SMT):
#!/bin/csh
#PBS -l select=32:ncpus=64:
mpiprocs=NP:ompthreads=NT

- Results for 128-2048 cores
- Only 1024 cores were available for the experiments
- BT-MZ and SP-MZ show benefit from Simultaneous Multithreading (SMT): 2048 threads on 1024 cores

Performance Analysis with gprof on IBM Power 6

- **Compilation:**
  - `mpxlf_r -O4 -qarch=pwr6 -qtune=pwr6 -qsmp=omp -pg`
- **Execution :**
  - `export OMP_NUM_THREADS 4`
  - `poe launch $PBS_O_WORKDIR./sp.C.16x4.exe`
  - Generates a file `gmount.MPI_RANK.out` for each MPI Process
- **Generate report:**
  - `gprof sp.C.16x4.exe gmon*`

```
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
16.7     117.94   117.94   205245     0.57     0.57  .@10@x_solve@OL@1 [2]
14.6     221.14   103.20   205064     0.50     0.50  .@15@z_solve@OL@1 [3]
12.1     307.14    86.00   205200     0.42     0.42  .@12@y_solve@OL@1 [4]
 6.2     350.83    43.69   205300     0.21     0.21  .@8@compute_rhs@OL@1@OL@6 [5]
```

Conclusions:

- **BT-MZ:**
  - Inherent workload imbalance on MPI level
  - #nprocs = #nzones yields poor performance
  - #nprocs < #zones => better workload balance, but decreases parallelism
  - Hybrid MPI/OpenMP yields better load-balance, maintains amount of parallelism
- **SP-MZ:**
  - No workload imbalance on MPI level, pure MPI should perform best
  - MPI/OpenMP outperforms MPI on some platforms due contention to network access within a node
- **LU-MZ:**
  - Hybrid MPI/OpenMP increases level of parallelism

- **"Best of category"**
  - Depends on many factors
  - Hard to predict
  - Good thread affinity is essential

Parallelization of a 3-D Flow Solver for Multi-Core Node Clusters: Experiences Using Hybrid MPI/OpenMP In the Real World

Dr. Gabriele Jost[1]
gjost@tacc.utexas.edu

Robert E. Robins[2)]
bob@nwra.com

1)Texas Advanced Computing Center, The University of Texas at Austin, TX

2)NorthWest Research Associates, Inc., Redmond, WA

Published in Scientific Programming, Vol. 18, No. 3-4 /2010 pp 127-138, IOS Press DOI 10.3233/SPR-2010-0308

Acknowledgements:

- **NWRA, NASA, ONR**
- **DoD HPCMP, in particular**
- **U.S. Army Engineering Research and Development Center, http://www.erdc.hpc.mil**
- **The Navy DoD Supercomputing Resource Center, http://www.navo.hpc.mil**

Numerical Approach

- Solve 3-D (or 2-D) Boussinesq equations for incompressible fluid (ocean or atmosphere)
- FFT's for horizontal derivatives (periodic BC)
- Higher-order compact scheme for vertical derivatives
- 2nd order Adams-Bashforth time-stepping
  (projection method to ensure incompressibility – requires solution to Poisson's Equation at every time step)
- Sub-grid scale model
- Periodic smoothing to control small-scale energy – compact approach in vertical, FFT approach in horizontal

```
Start Time-Step Loop
    CALL DCALC (calculate time derivatives)
    DO ADVECTION LOOP
    CALL DMOVE (derivs_2 => derivs_1)
    CALL PCALC (solve Poisson's equation)
    DO PROJECTION LOOP
    CALL TAPER (apply boundary conditions)
End Time-Step Loop
```

Multiple z-and y- derivatives in x
Multiple x-derivatives in y-plane

2D FFTs in z-plane

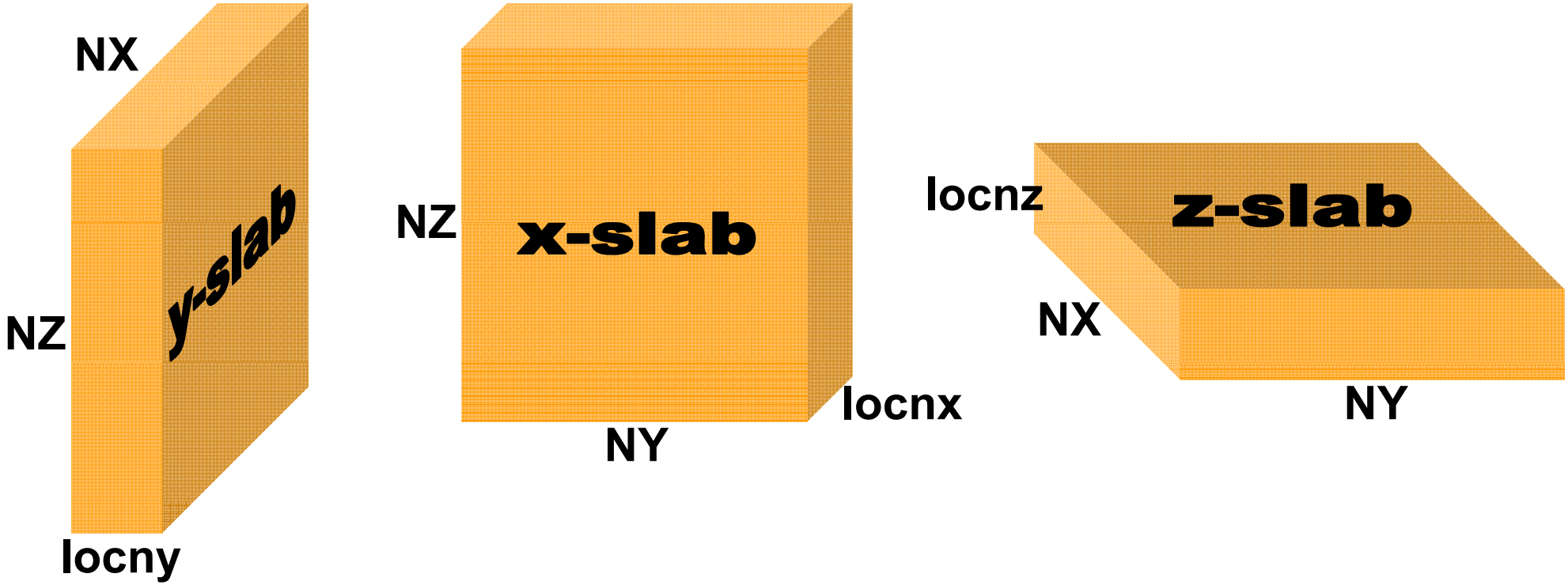
Development of MPI Parallelization

- Initial code developed for vector processors
- MPI Version: Aim for portability and scalability on clusters of SMPs

- 1D domain decomposition (based on scalar/vector code structure):
  - x-slabs to do z- and y-derivatives, y-slabs to do x-derivatives, z-slabs for Poisson solver
- Each processor contains
  - x-slab (#planes=locnx=NX/nprocs)
  - y-slab (#planes=locny=NY/nprocs)
  - z-slab (#planes=locnz=NZ/nprocs)
  - for each variable

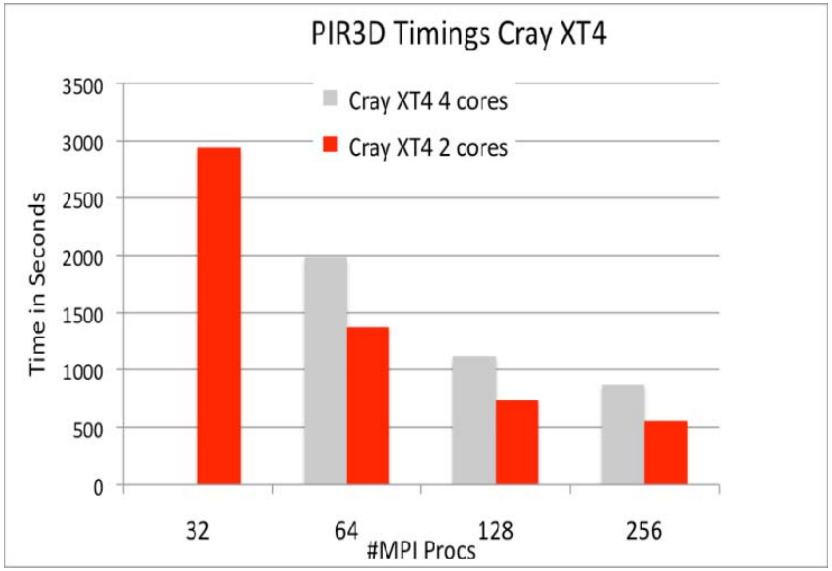
- Redistribution of data (swapping) required during execution
- Basic structure of code was be preserved

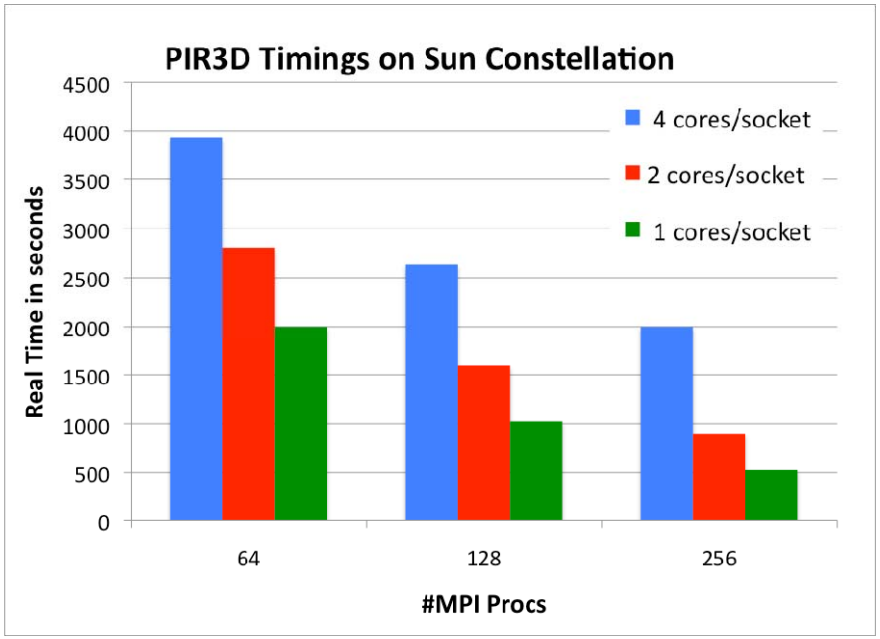
Domain Decomposition for Parallel Derivative Computations

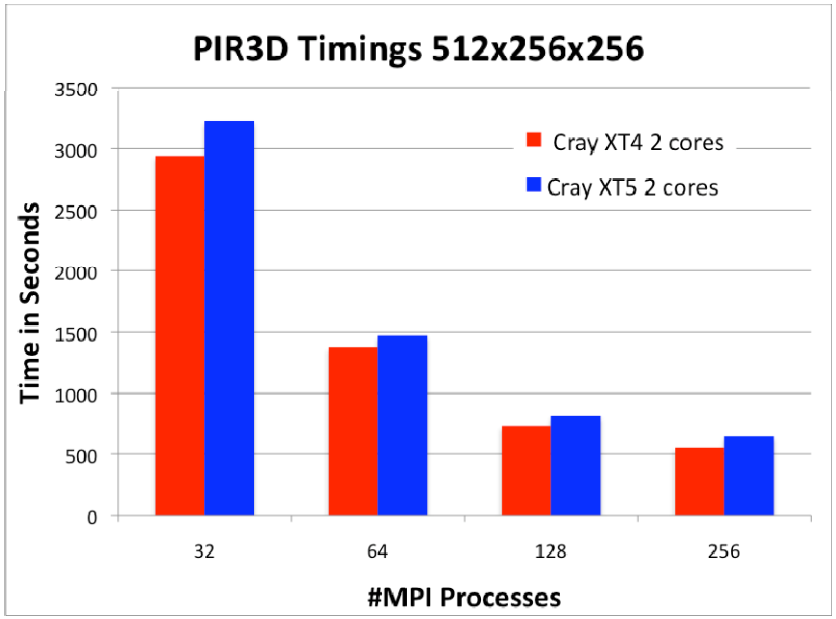


locn[xyz] = N[XYZ] / nprocs

Initial PIR3D Timings Case 512x256x256







- Problem Size 512x256x256
- Cray XT4: 4 cores per node
- Cray XT5: 8 cores per node
- Sun Constellation: 16 cores per node
- Significant time decrease when using 2 cores per socket rather than 4

- BUT: Using only 2 cores:
  - Increases resource requirement (#cores/nodes)
  - Leaves half of the requested cores idle

PIR3D Performance

- **What causes performance decrease when using all cores per socket?**

  - Some increase in User CPU Time
  - Significant increase in MPI time
  - Swapping requires global all-to-all type communication

CrayPat Performance Statistics for Cray XT5

4 cores per socket

Table 1: Profile by Function

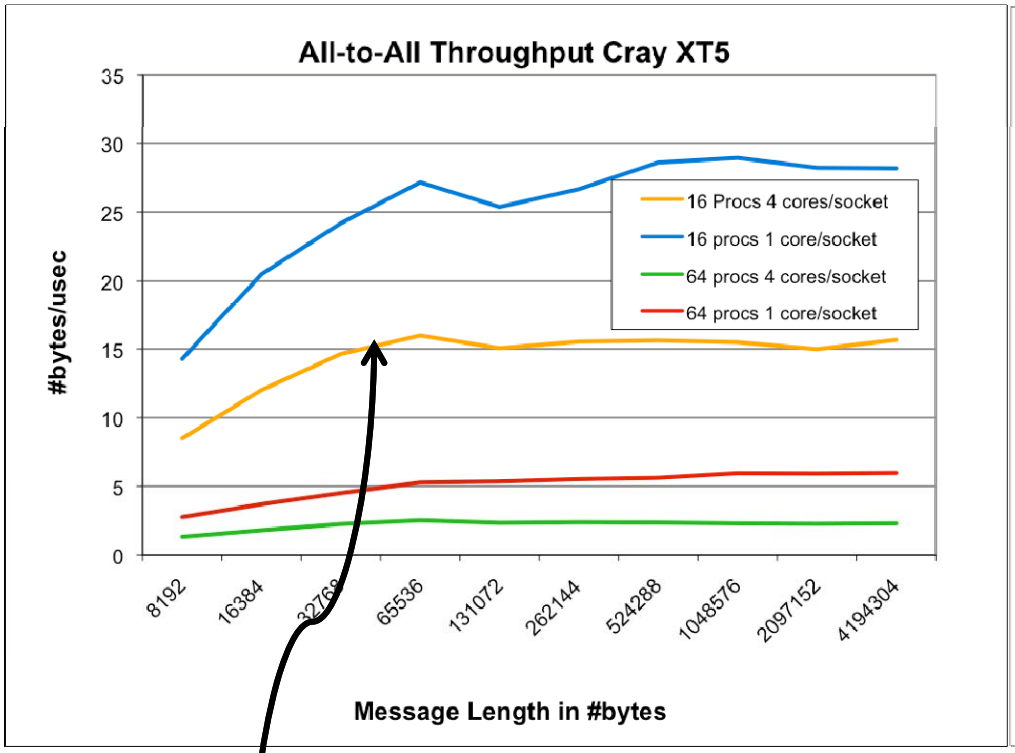
Samp %	Samp	Group
40.9%	279404	USER
8.4%	57437	dcalc_
5.0%	34240	getdiv_
4.1%	28323	rvcalc_
4.0%	27202	csfft_
2.3%	15693	swapyx_
1.5%	10051	swapxy_
29.9%	204411	MPI
16.1%	109624	mpi_waitall_
4.1%	28253	mpi_send_
3.9%	26565	mpi_ibsend_
3.3%	22363	mpi_irecv_
1.9%	13100	mpi_bsend_
29.1%	198881	ETC
6.9%	46856	dgtts2_
4.6%	31179	_c_mcopy8
4.3%	29496	daxpy_k
1.5%	10027	hc2cbdftv_8
1.2%	8117	dgbmv_n

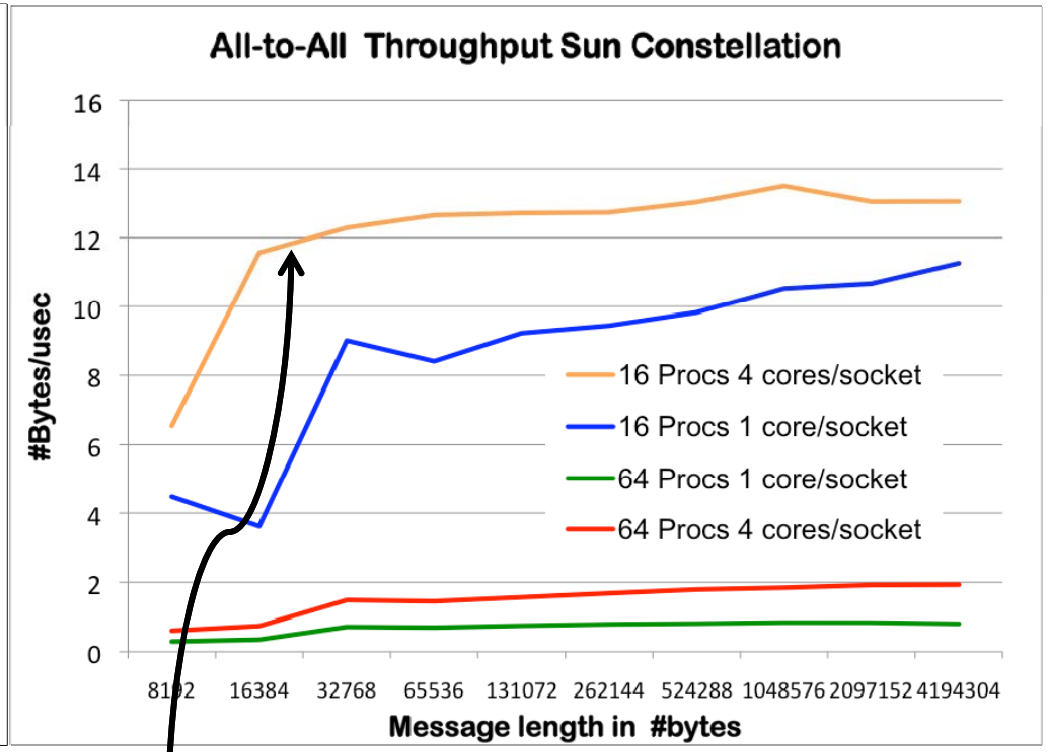
1 core per socket

Table 1: Profile by Function

Samp %	Samp	Group
100.0%	442157	Total
40.7%	179890	USER
10.9%	48416	dcalc_
4.9%	21543	getdiv_
4.3%	19064	rvcalc_
3.1%	13795	csfft_
2.6%	11531	swapyx_
1.6%	6941	swapxy_
1.3%	5679	scfft_
38.2%	169084	ETC
10.4%	46117	dgtts2_
6.7%	29648	daxpy_k
4.3%	18820	__c_mcopy8
2.1%	9194	hc2cbdftv_8
1.8%	8108	dgbmv_n
21.1%	93183	MPI
7.3%	32290	mpi_waitall_
4.7%	20944	mpi_ibsend_
3.5%	15558	mpi_irecv_
2.5%	10862	mpi_send_
2.2%	9755	mpi_bsend_

All-to-All Throughput





Inter-Node Communication requires network access.

Intra-Node Communication only! No network access required.

Limitations of PIR3D MPI Implementation

- **Global MPI communication yields resource contention within a node (access to network)**
  - Mitigate by using fewer MPI processes than cores per node
- **#MPI Procs restricted to shortest dimension due to 1D domain decomposition**
  - Possible solution: Use 3D Domain Composition, but would mean considerable implementation effort
- **Memory requirements may restrict run to use at most 1 core/socket**
  - 3D Data is distributed, each MPI Proc only holds a slab
  - 2D Work arrays are replicated
  - Necessary to use fewer MPI Procs than cores per node

All-the-cores-all-the-time: How can OpenMP help?

OpenMP Parallelization of PIR3D (1)

- **Motivation:**
  - Increase performance by taking advantage of idle cores within one shared memory node
- **OpenMP Parallelization strategy:**
  - Identify most time consuming routines
  - Place OpenMP directives on the time consuming loops
  - Only place directives on loops across undistributed dimension
  - MPI calls only occur outside of parallel regions: No thread safety is required for MPI library

```
DO 2500 IX=1,LOCNX
….
!$omp parallel do private(iy,rvsc)
 DO 2220 IZ=1,NZ
   DO 2220 IY=1,NY
      VYIX(IY,IZ) = YF(IY,IZ)
      VY_X(IZ,IY,IX) = YF(IY,IZ)
      RVSC = RVISC_X(IZ,IY,IX)
      DVY2_X(IZ,IY,IX) =
        DVY2_X(IZ,IY,IX) -
        (VYIX(IY,IZ)+VBG(IZ)) *
        YDF(IY,IZ)+RVSC*YDDF(IY,IZ)
2220 CONTINUE
!$omp end parallel do
….
2500 CONTINUE
```

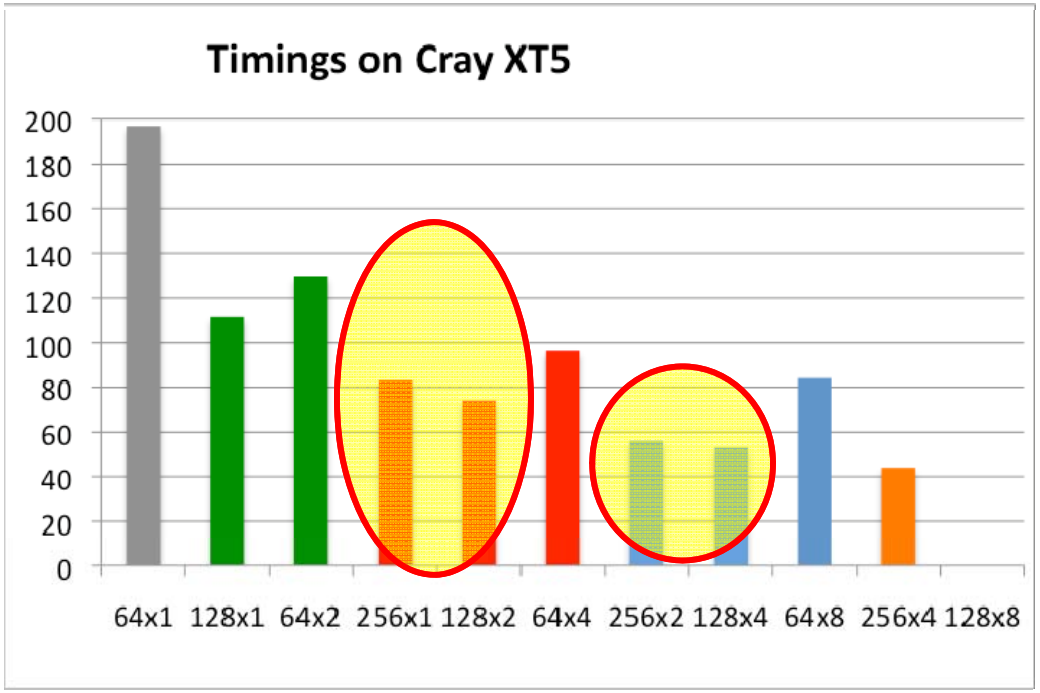
OpenMP Parallelization of PIR3D (2)

- Thread safe LAPACK and FFTW routines required
- FFTW initialization routine not thread safe: Execute outside of parallel region

- Limitation of current OpenMP parallelization:
  - Only a small subset of routines have been parallelized
  - Computation time distributed across a large number of routines

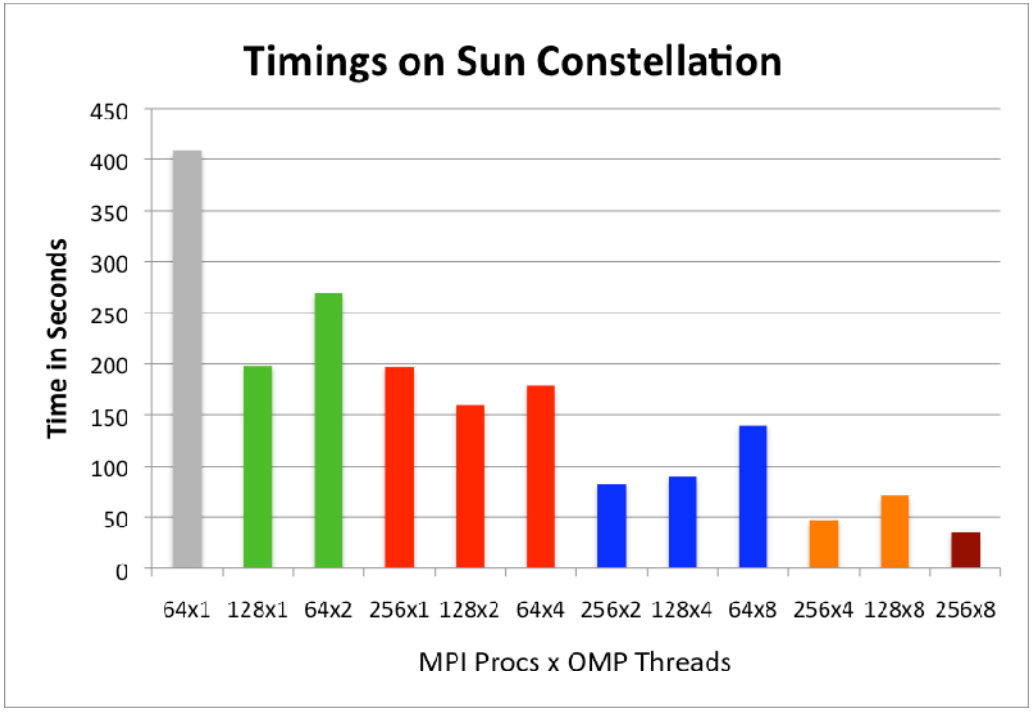
```
      subroutine csfftm(isign,ny,…)
      implicit none
      integer isign, n, m,
      integer i, ny
      integer omp_get_num_threads
      real work, tabl
      real a(1:m2,1:m)
      complex f(1:m1,1:m)
!$omp parallel if(isign.ne.0)
!$omp do
      do i = 1, m
         CALL csfft (isign,ny,…)
      end do
!$omp end do
!$omp end parallel
      return
      end
```

Hybrid Timings for Case 512x256x256

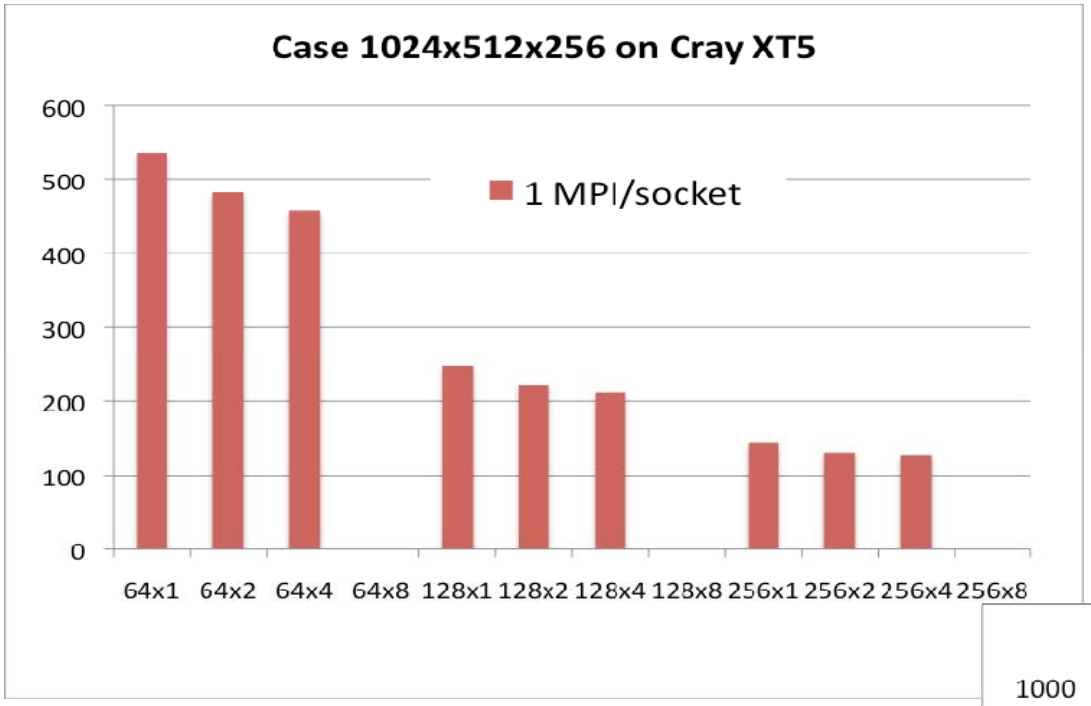


- Use all 4 cores/per socket
- Benefits of OpenMP:
  - Increase the number of usable cores
  - 128x2 outperforms 256x1 on 256 cores,128x4 better than 256x2 on 512 cores

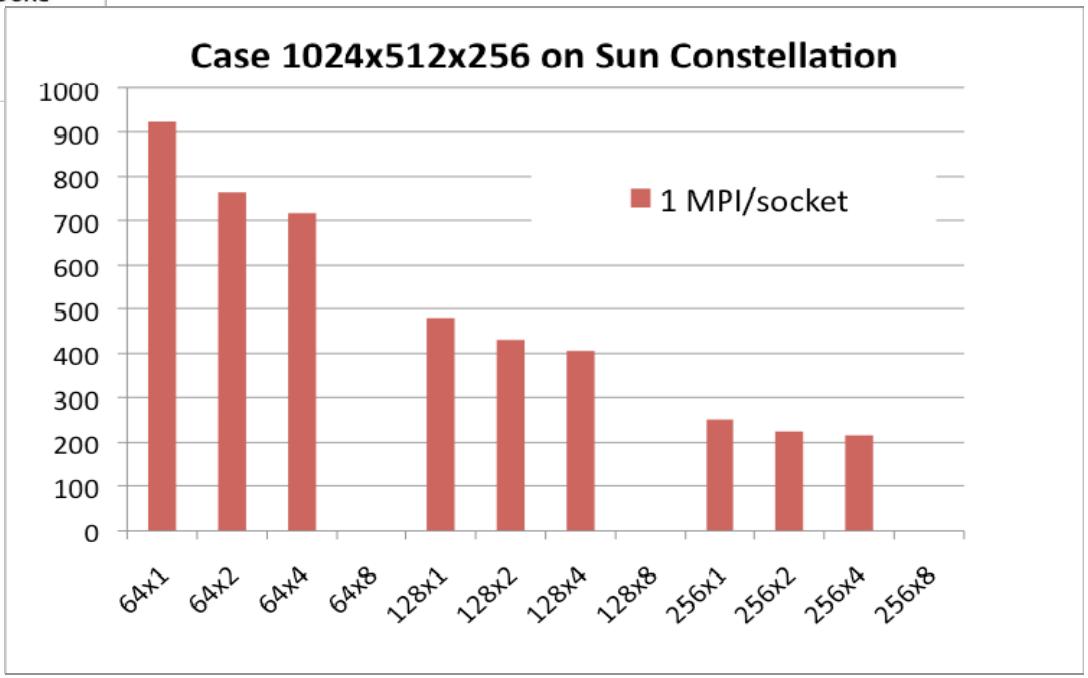
But: Most of the performance due to "spacing" of MPI. About 12% improvement due to OpenMP

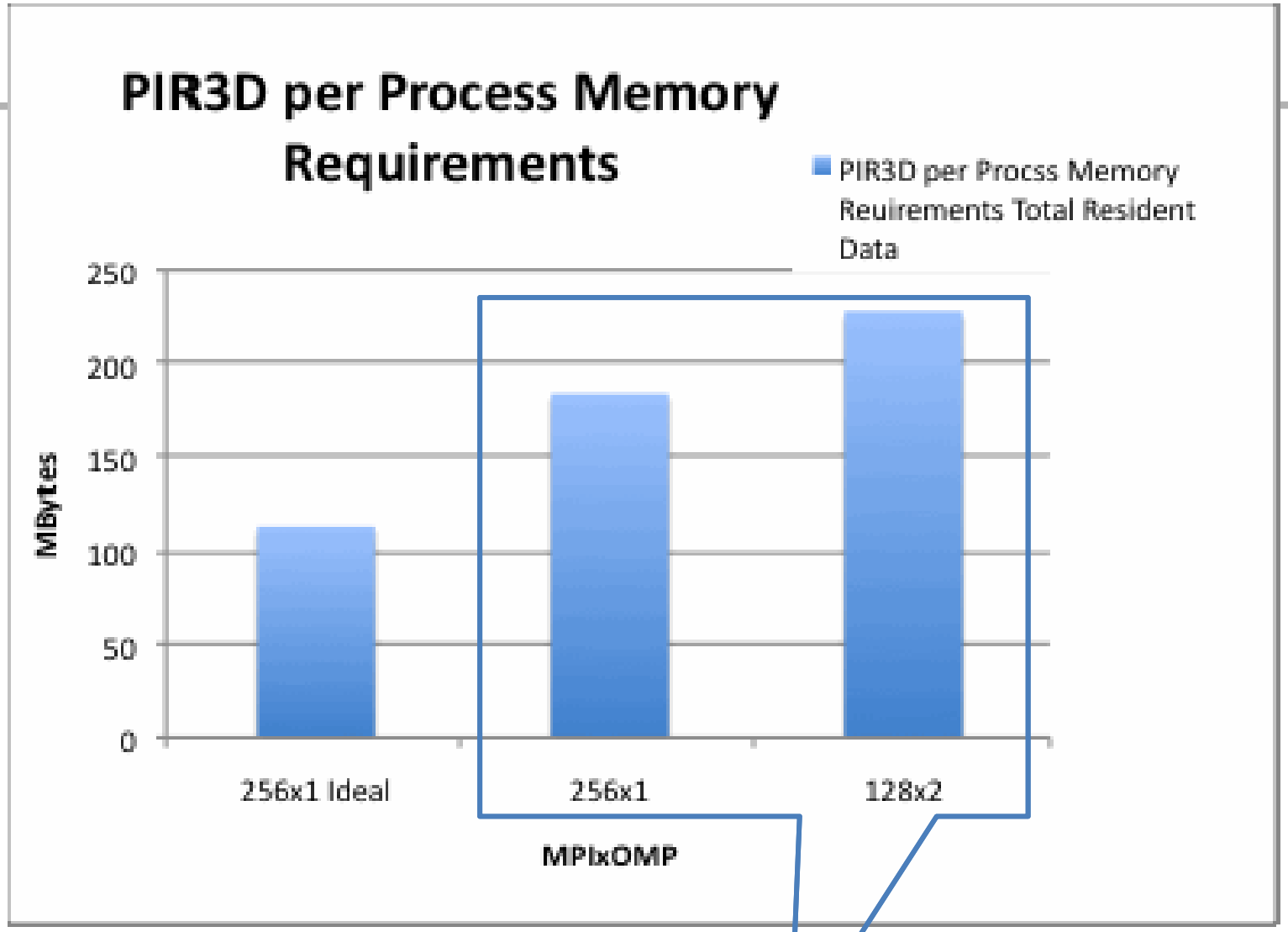


Hybrid Timings for Case 1024x512x256



- Only 1 MPI Process per socket due to memory consumption
- 14%-10% performance increase on Cray XT5
- 13% to 22% performance increase on Sun Constellation





Includes distributed and replicated data and MPI buffers for problem size 256x512x256

Conclusions for PIR3D

- **Hybrid OpenMP parallelization of PIR3D was beneficial**
  - Easy to implement when aiming for moderate speedup
  - Reduce MPI time for global communication:
    - Lower number of MPI processors to mitigate network contention
  - Take advantage of idle cores allocated for memory requirements
  - Lower memory requirements (e.g., replicated data, MPI buffers)

- **Issues when using OpenMP:**
  - Runtime libraries: Are they thread-safe? Are they multi-threaded? Are they compatible with OpenMP?
  - Easy for moderate scalability (4-8 threads), **But** for 10's or 100's of threads?
  - Are there sufficient parallelizable loops? Only moderate speed-up if not enough parallelizable loops
  - **Good scalability may require to parallelize many loops!**

- **Issues when running hybrid codes:**
  - Placement of MPI processes and OpenMP threads onto available cores is:
    - critical for good performance
    - highly system dependent

Tutorial outline

- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
  - Practical “How-tos” for hybrid
- **Online demo: likwid tools (2)**
  - Advanced pinning
  - Making bandwidth maps
  - Using likwid-perfctr to find NUMA problems and load imbalance
  - likwid-perfctr internals
  - likwid-perfscope
- **Case studies for hybrid MPI/OpenMP**
  - Overlap for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - PIR3D – hybridization of a full scale CFD code
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

Elements of Successful Hybrid Programming

- **System Requirements:**
  - Some level of shared memory parallelism, such as within a multi-core node
  - Runtime libraries and environment to support both models
    - Thread-safe MPI library
    - Compiler support for OpenMP directives, OpenMP runtime libraries
  - Mechanisms to map MPI processes and threads onto cores and nodes
- **Application Requirements:**
  - Expose multiple levels of parallelism
    - Coarse-grained and fine-grained
    - Enough fine-grained parallelism to allow OpenMP scaling to the number of cores per node
- **Performance:**
  - Highly dependent on optimal process and thread placement
  - No standard API to achieve optimal placement
  - Optimal placement may not be known beforehand (i.e. optimal number of threads per MPI process) or requirements may change during execution
  - Memory traffic yields resource contention on multicore nodes
  - Cache optimization more critical than on single core nodes

Recipe for Successful Hybrid Programming

- **Familiarize yourself with the layout of your system:**
  - Blades, nodes, sockets, cores?
  - Interconnects?
  - Level of Shared Memory Parallelism?
- **Check system software**
  - Compiler options, MPI library, thread support in MPI
  - Process placement
- **Analyze your application:**
  - Architectural requirements (code balance, pipelining, cache space)
  - Does MPI scale? If yes, why bother about hybrid? If not, why not?
    - Load imbalance → OpenMP might help
    - Too much time in communication? Workload too small?
  - Does OpenMP scale?
- **Performance Optimization**
  - Optimal process and thread placement is important
  - Find out how to achieve it on your system
  - Cache optimization critical to mitigate resource contention
  - Creative use of surplus cores: Overlap, functional decomposition,…

Hybrid Programming: Does it Help?

- **Hybrid Codes provide these opportunities:**
  - Lower communication overhead
    - Few multithreaded MPI processes vs many single-threaded processes
    - Fewer number of calls and smaller amount of data communicated
  - Lower memory requirements
    - Reduced amount of replicated data
    - Reduced size of MPI internal buffer space
    - May become more important for systems of 100's or 1000's cores per node
  - Provide for flexible load-balancing on coarse and fine grain
    - Smaller #of MPI processes leave room to assign workload more even
    - MPI processes with higher workload could employ more threads
  - Increase parallelism
    - Domain decomposition as well as loop level parallelism can be exploited
    - Functional parallelization

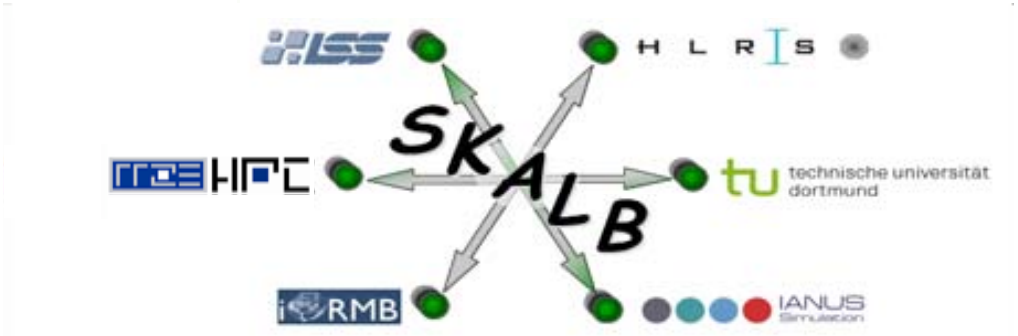
YES, IT CAN!

Thank you

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

Grant # 01IH08003A
(project SKALB)



Project OMI4PAPPS

Appendix

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 for Bandwidth-Limited Loop Kernels on Multi-Core Architectures. DOI: 10.1007/978-3-642-13872-0_1, Preprint: arXiv: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, 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. Preprint: arXiv:1006.3148
- R. Preissl et al.: Overlapping communication with computation using OpenMP tasks on the GTS magnetic fusion code. Scientific Programming, Vol. 18, No. 3-4 (2010). DOI: 10.3233/SPR-2010-0311

References

Papers continued:

- 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. DOI: 10.1109/ICPPW.2010.38. Preprint: arXiv:1004.4431
- G. Schubert, G. Hager, H. Fehske and G. Wellein: Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming. Accepted for the Workshop on Large-Scale Parallel Processing (LSPP 2011), May 20th, 2011, Anchorage, AK. Preprint: arXiv:1101.0091
- G. Schubert, G. Hager and H. Fehske: Performance limitations for sparse matrix-vector multiplications on current multicore environments. Proc. HLRB/KONWIHR Workshop 2009. DOI: 10.1007/978-3-642-13872-0_2 Preprint: arXiv: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. PDF
- R. Rabenseifner and G. Wellein: Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. International Journal of High Performance Computing Applications **17**, 49-62, February 2003. DOI:10.1177/1094342003017001005
- G. Jost and R. Robins: Parallelization of a 3-D Flow Solver for Multi-Core Node Clusters: Experiences Using Hybrid MPI/OpenMP In the Real World. Scientific Programming, Vol. 18, No. 3-4 (2010) pp. 127-138. DOI 10.3233/SPR-2010-0308

Presenter Biographies

Georg Hager (georg.hager@rrze.uni-erlangen.de) holds a PhD in computational physics from the University of Greifswald, Germany. 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, talks, and teaching.

Gabriele Jost (gjost@tacc.utexas.edu) received her doctorate in applied mathematics from the University of Göttingen, Germany. She has worked in software development, benchmarking, and application optimization for various vendors of high performance computer architectures. She also spent six years as a research scientist in the Parallel Tools Group at the NASA Ames Research Center in Moffett Field, California. Her projects included performance analysis, automatic parallelization and optimization, and the study of parallel programming paradigms. She is now a Research Scientist at the Texas Advanced Computing Center (TACC), working remotely from Monterey, CA on all sorts of projects related to large scale parallel processing for scientific computing.

Jan Treibig (jan.treibig@rrze.uni-erlangen.de) holds a PhD in Computer Science from the University of Erlangen-Nuremberg, Germany. From 2006 to 2008 he was a software developer and quality engineer in the embedded automotive software industry. Since 2008 he is a research scientist in the HPC Services group at Erlangen Regional Computing Center (RRZE). His main research interests are low-level and architecture-specific optimization, performance modeling, and tooling for performance-oriented software developers. Recently he has founded a spin-off company, "LIKWID High Performance Programming."

Gerhard Wellein (gerhard.wellein@rrze.uni-erlangen.de) holds a PhD in solid state physics from the University of Bayreuth, Germany and is a professor at the Department for Computer Science at the University of Erlangen-Nuremberg. He leads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering programs. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.

Abstract

- **Tutorial:** **Performance-oriented programming on multicore-based clusters with MPI, OpenMP, and hybrid MPI/OpenMP**

- **Presenters:** Georg Hager, Gabriele Jost, Jan Treibig, Gerhard Wellein
- **Authors:** Georg Hager, Gabriele Jost, Rolf Rabenseifner, Jan Treibig, Gerhard Wellein

- **Abstract:** Most HPC systems are clusters of multicore, multisocket nodes. These systems are highly hierarchical, and there are several possible programming models; the most popular ones being shared memory parallel programming with OpenMP within a node, distributed memory parallel programming with MPI across the cores of the cluster, or a combination of both. Obtaining good performance for all of those models requires considerable knowledge about the system architecture and the requirements of the application. The goal of this tutorial is to provide insights about performance limitations and guidelines for program optimization techniques on all levels of the hierarchy when using pure MPI, pure OpenMP, or a combination of both.
  We cover peculiarities like shared vs. separate caches, bandwidth bottlenecks, and ccNUMA locality. Typical performance features like synchronization overhead, intranode MPI bandwidths and latencies, ccNUMA locality, and bandwidth saturation (in cache and memory) are discussed in order to pinpoint the influence of system topology and thread affinity on the performance of parallel programming constructs. Techniques and tools for establishing process/thread placement and measuring performance metrics are demonstrated in detail. We also analyze the strengths and weaknesses of various hybrid MPI/OpenMP programming strategies. Benchmark results and case studies on several platforms are presented.