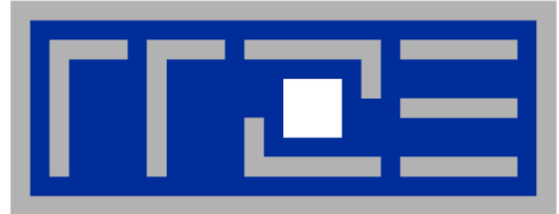


Performance-oriented programming on multicore-based clusters with MPI, OpenMP, and hybrid MPI/OpenMP

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), Germany
[b]Advanced Micro Devices (AMD), USA
[c]High Performance Computing Center Stuttgart (HLRS), Germany
[d]Department for Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg, Germany

ISC12 Tutorial, June 17th, 2012, Hamburg, Germany

http://blogs.fau.de/hager/tutorials/isc12/

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**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Microbenchmarking with simple parallel loops
  - Bandwidth saturation effects in cache and memory
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - ccNUMA effects and how to circumvent performance penalties
  - Simultaneous multithreading (SMT)
- **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
- **Case studies for hybrid MPI/OpenMP**
  - Overlap of communication and computation for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - Hybrid computing with accelerators and compiler directives
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

Tutorial outline (1)

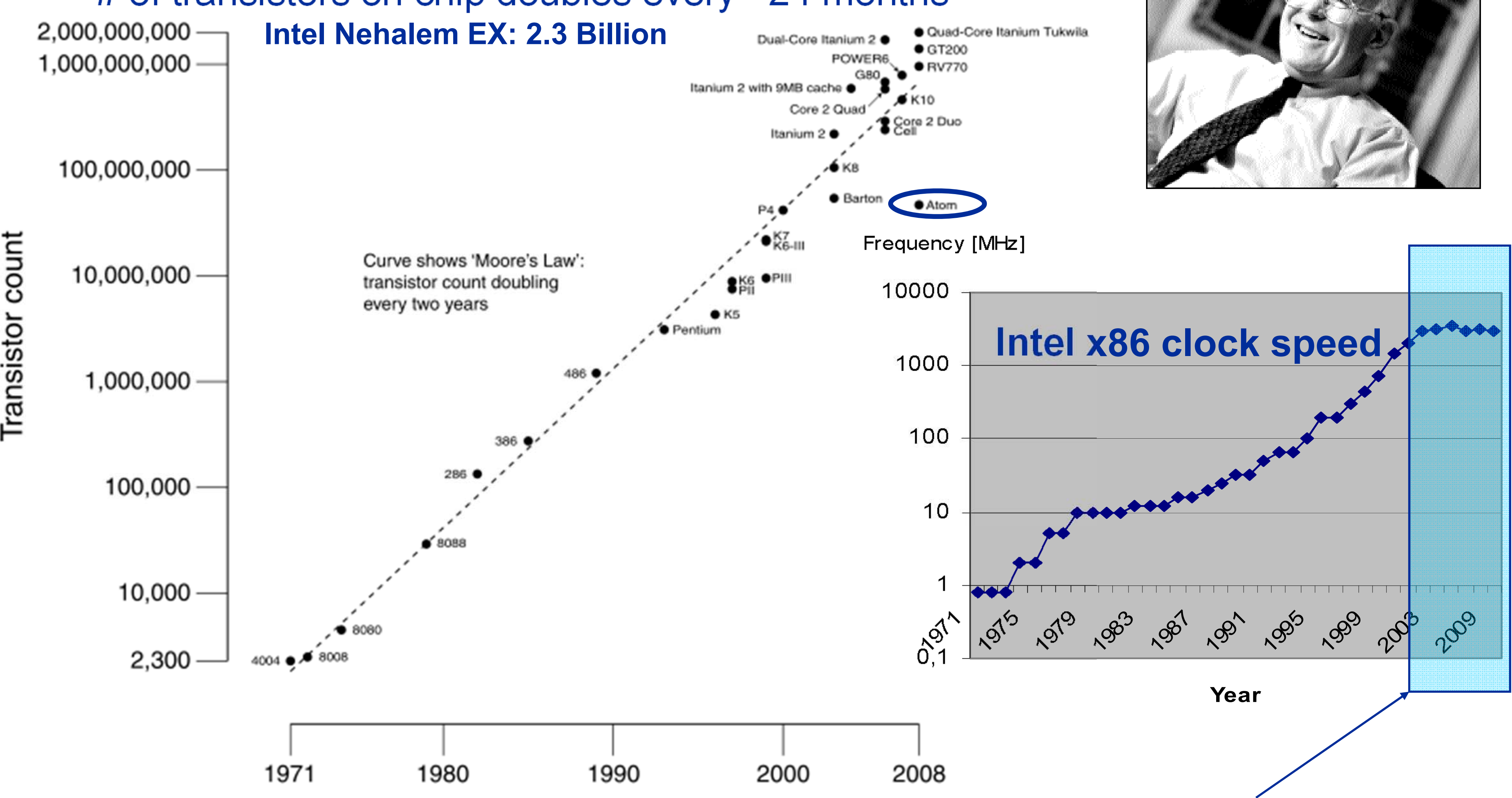
- Introduction
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models
- Multicore performance tools
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements
- Online demo: likwid tools
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- Impact of processor/node topology on performance
  - Microbenchmarking with simple parallel loops
  - Bandwidth saturation effects in cache and memory
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - ccNUMA effects and how to circumvent performance penalties
  - Simultaneous multithreading (SMT)
- 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

By courtesy of D. Vrsalovic, Intel

intel

N transistors

2N transistors

Dual-Core

Performance

Power

1.13x

1.73x

1.00x

1.73x

1.02x

Over-clocked (+20%)

Max Frequency

Dual-core (-20%)

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

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

8 cores running at half speed of a single core CPU = same energy

→ 65 nm technology :
Sun T2 („Niagara") 1.4 GHz → 8 cores
Intel Woodcrest 3.0 GHz → 2 cores

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

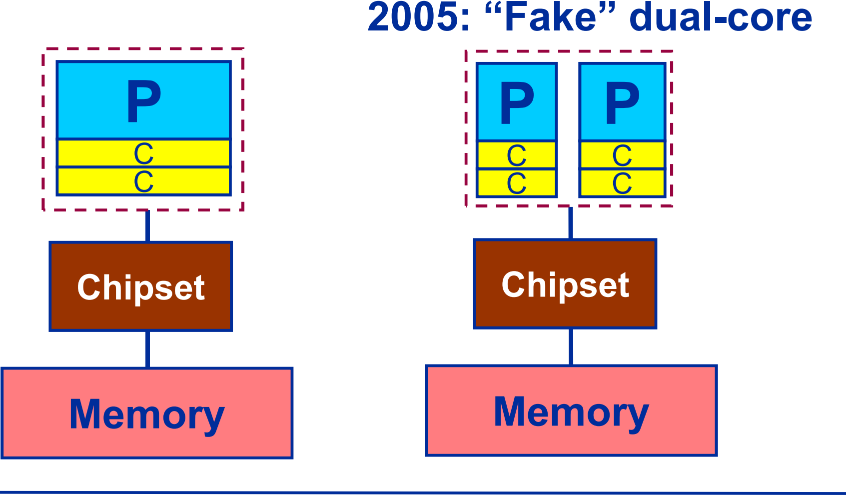
Process technology / Number of transistors in million

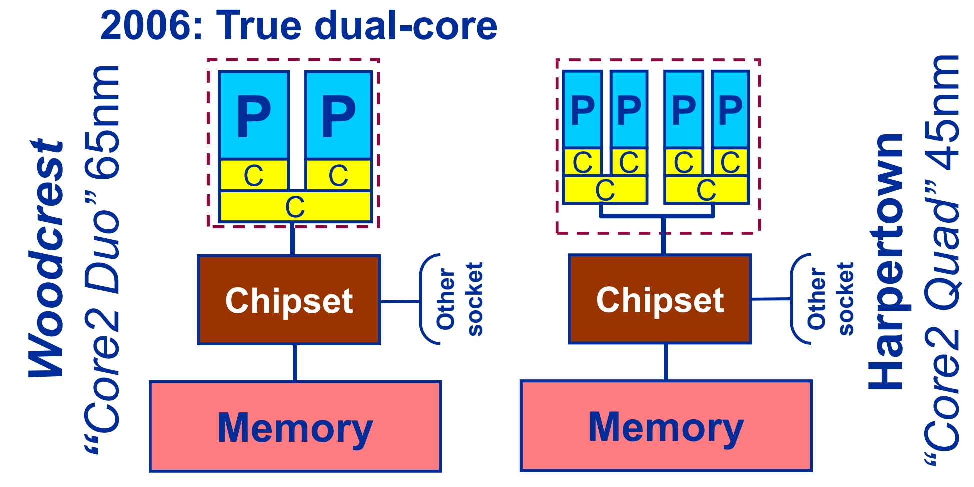
- **Moore's law is still valid…**
  → more cores + new on-chip functionality (PCIe, GPU)

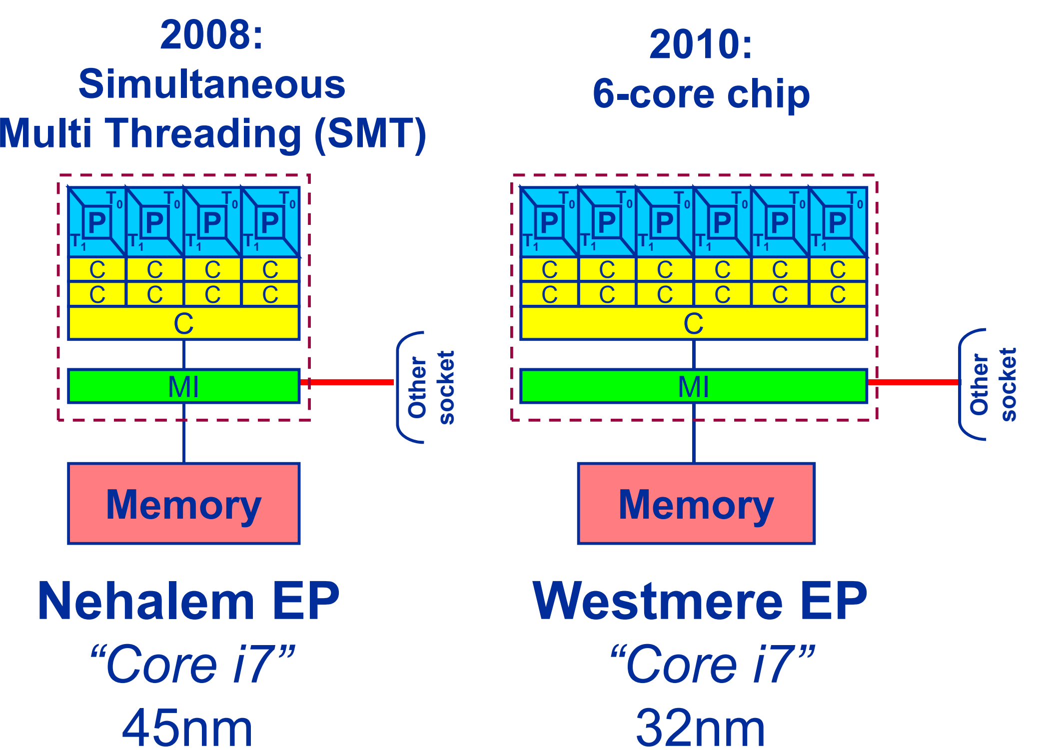
Be prepared for more cores with less complexity and slower clock!

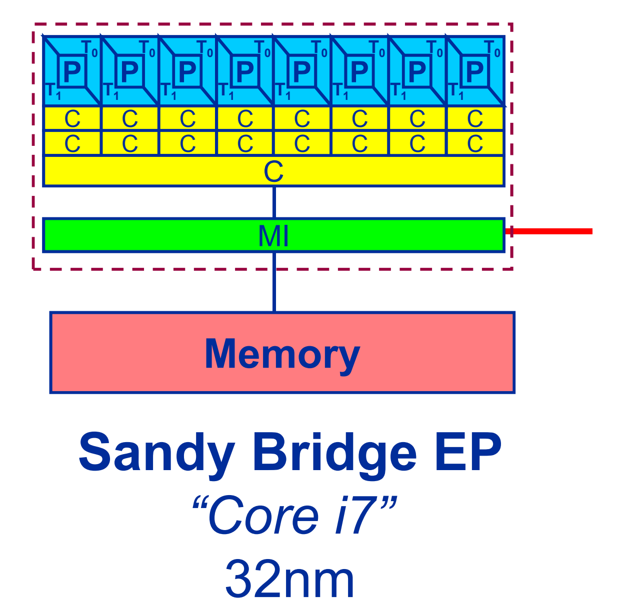
The x86 multicore evolution so far

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







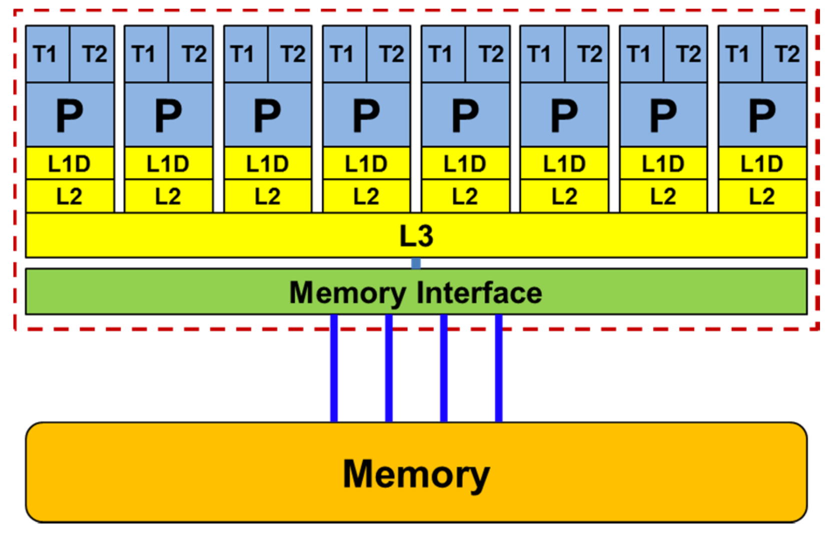


Nehalem EP
"Core i7"
45nm

Westmere EP
"Core i7"
32nm

Sandy Bridge EP
"Core i7"
32nm

There is no longer a single driving force for chip performance!



Intel Xeon
"Sandy Bridge EP" socket
4,6,8 core variants available

Floating Point (FP) Performance:

$$P = n_{core} * F * S * \nu$$

n_{core} number of cores: 8

F FP instructions per cycle: 2
(1 MULT and 1 ADD)

S FP ops / instruction: 4 (dp) / 8 (sp)
(256 Bit SIMD registers – "AVX")

ν Clock speed : 2.5 GHz

TOP500 rank 1 (1996)

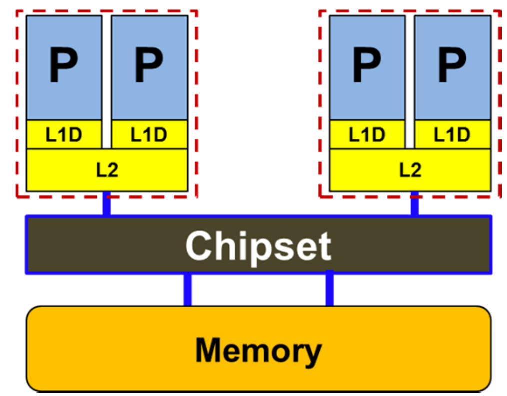
P = 160 GF/s (dp) / 320 GF/s (sp)

But: P=5 GF/s (dp) for serial, non-SIMD code

From UMA to ccNUMA

Basic architecture of commodity Intel-based compute cluster nodes

Yesterday (2006): Dual-socket Intel "Core2" node:

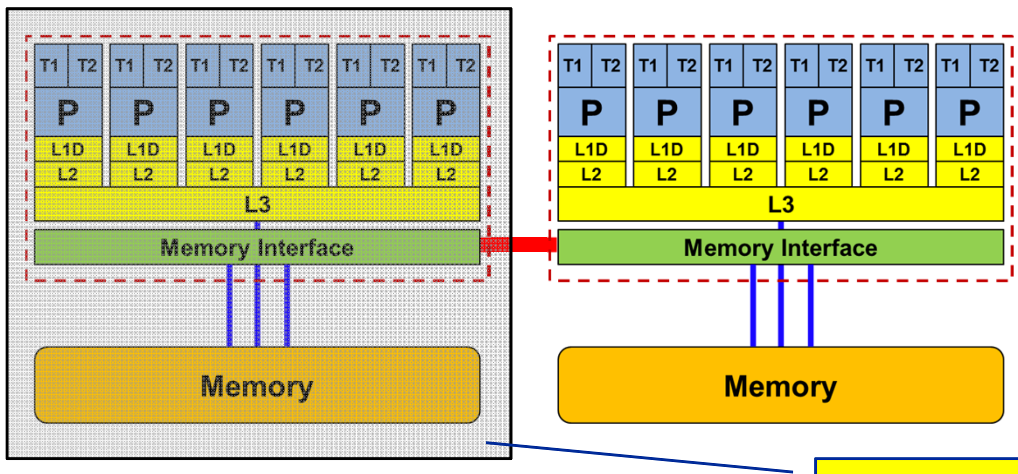


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system "anisotropy"

Today: Dual-socket Intel "Core i7" node:



Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

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

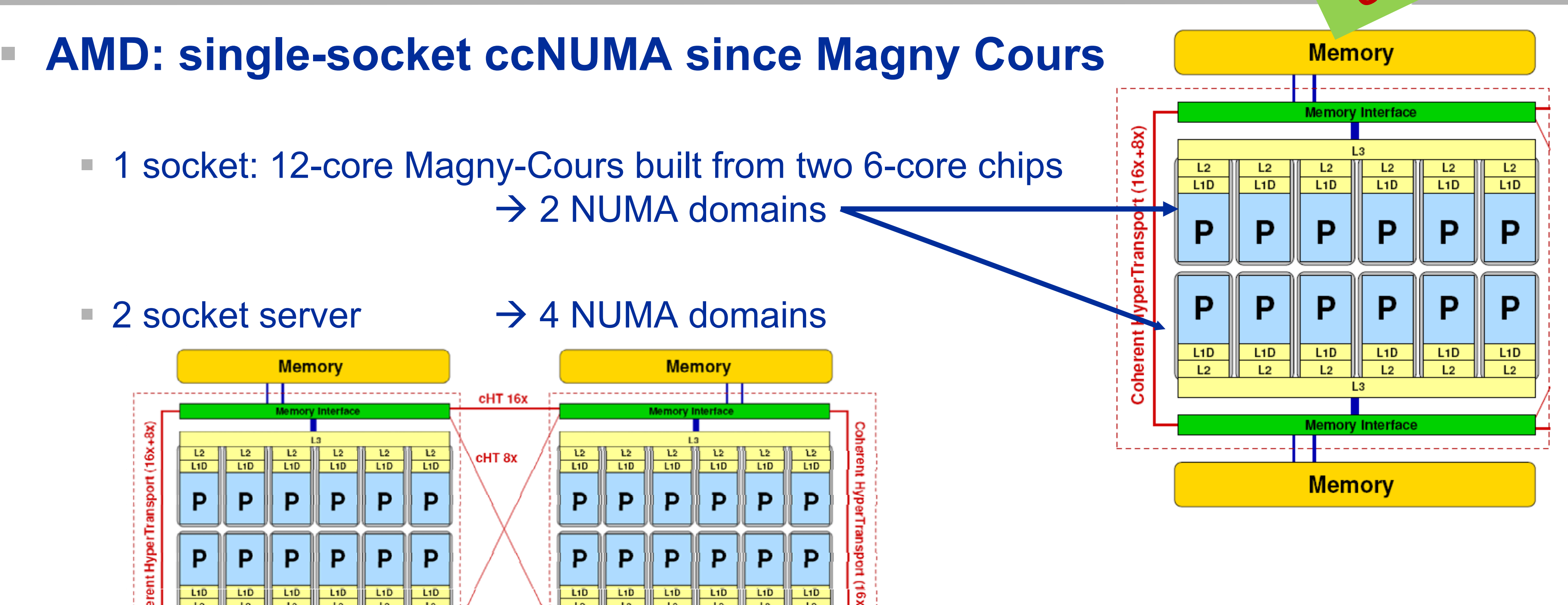
NUMA locality domain (LD)

On AMD it is even more complicated → ccNUMA within a socket!

Back to the 2-chip-per-case age

12 core AMD Magny-Cours – a 2x6-core ccNUMA socket

- **AMD: single-socket ccNUMA since Magny Cours**
  - 1 socket: 12-core Magny-Cours built from two 6-core chips → 2 NUMA domains
  - 2 socket server → 4 NUMA domains
  - 4 socket server: → 8 NUMA domains



- **WHY? → Shared resources are hard two scale: 2 x 2 memory channels vs. 1 x 4 memory channels per socket**

Another flavor of "SMT"
AMD Interlagos / Bulldozer

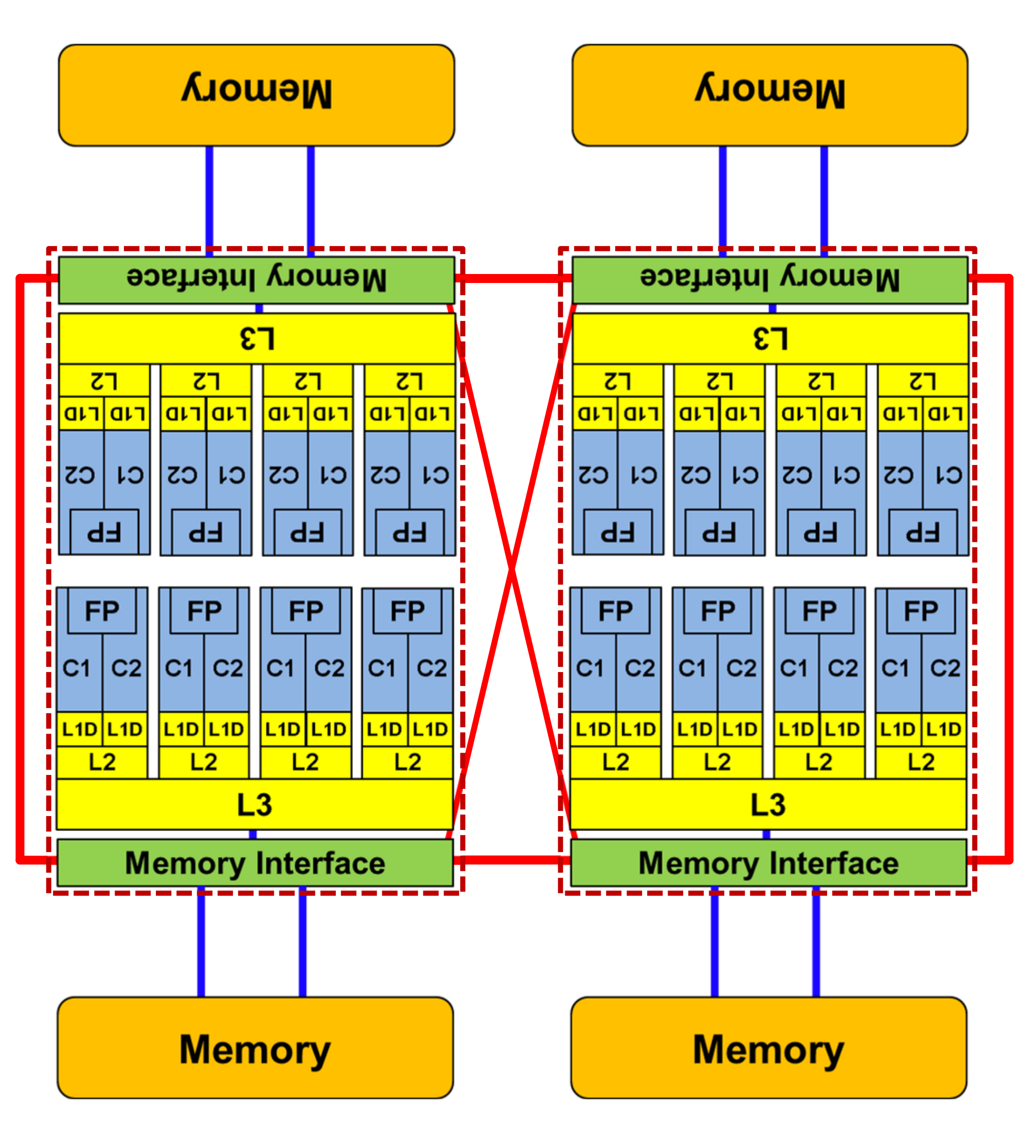
- Up to 16 cores (8 Bulldozer modules) in a single socket
- Max. 2.6 GHz (+ Turbo Core)
- P_{max} = (2.6 x 8 x 8) GF/s
  = 166.4 GF/s

Each Bulldozer module:
- 2 "lightweight" cores
- 1 FPU: 4 MULT & 4 ADD (double precision) / cycle
- Supports AVX
- Supports FMA4

2 DDR3 (shared) memory channel > 15 GB/s

2 NUMA domains per socket

32-core dual socket "Interlagos" node



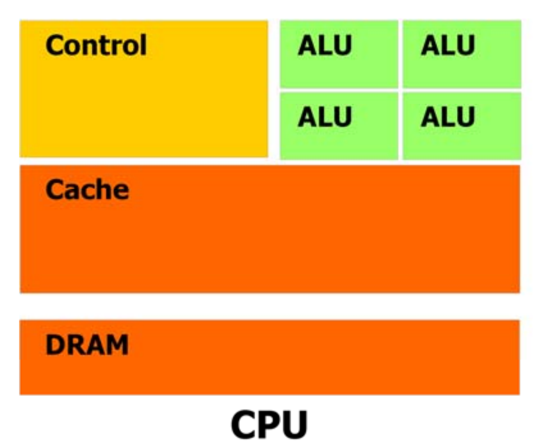
- Two 8- (integer-) core chips per socket
- Separate DDR3 memory interface per chip
  - ccNUMA on the socket!
- Shared FP unit per pair of integer cores ("module")
  - "256-bit" FP unit
  - SSE4.2, AVX, FMA4
- 16 kB L1 data cache per core
- 2 MB L2 cache per module
- 8 MB L3 cache per chip (6 MB usable)

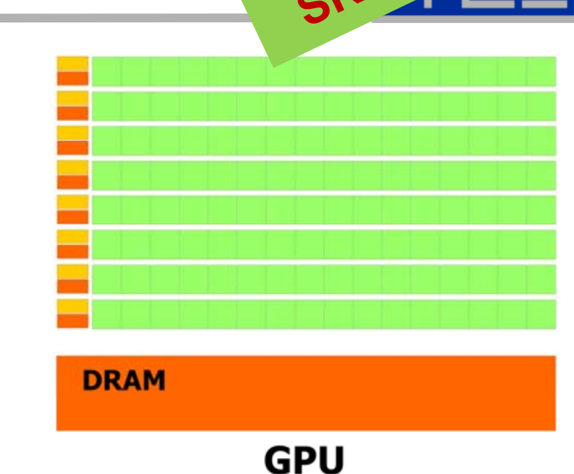
Trading single thread performance for parallelism:
GPGPUs vs. CPUs – speedup mythbusting

SKIPPED

GPU vs. CPU
light speed estimate:

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





	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*

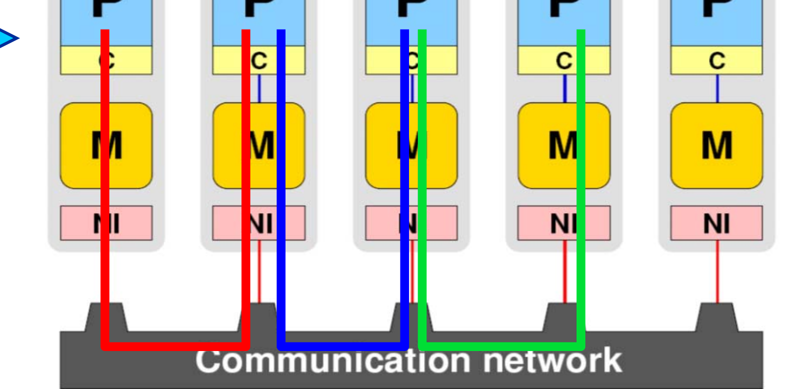
Complete compute device

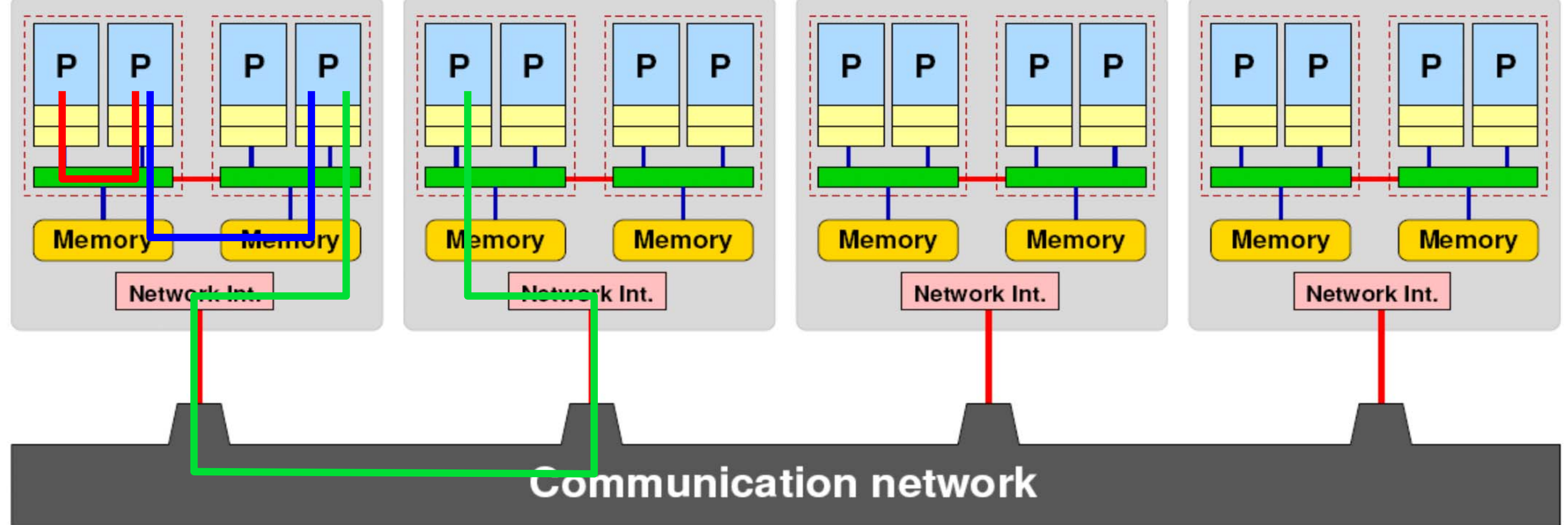
Parallel programming models
on multicore multisocket nodes

- **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!

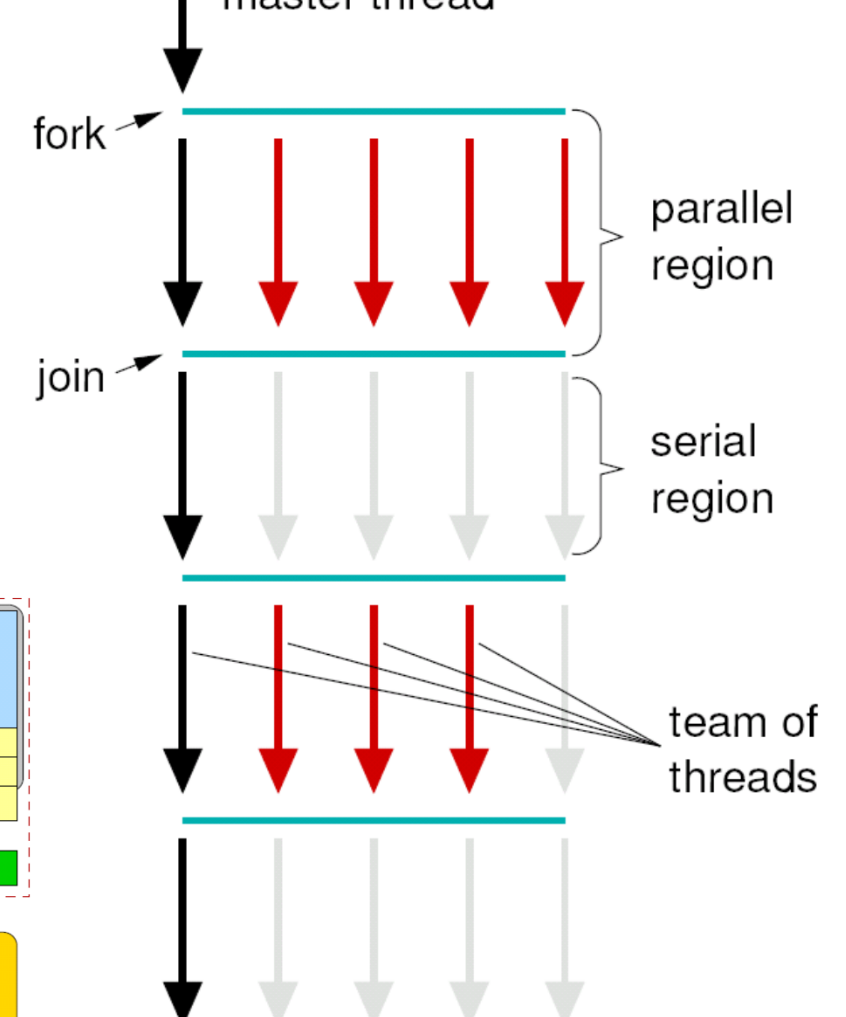
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

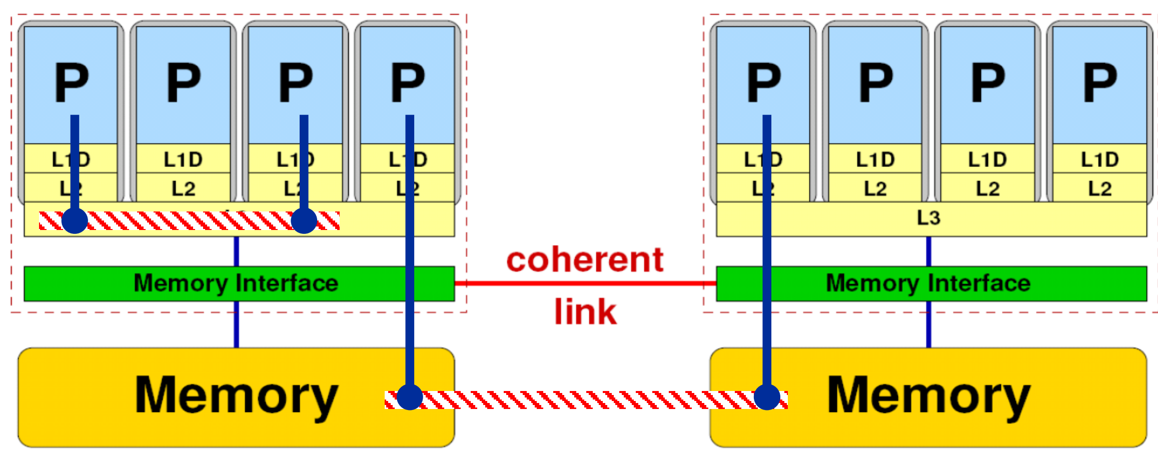




Parallel programming models:
Pure threading on the node

- **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

Covered in more detail in the hybrid part

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 (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**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- Impact of processor/node topology on performance
  - Microbenchmarking with simple parallel loops
  - Bandwidth saturation effects in cache and memory
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - ccNUMA effects and how to circumvent performance penalties
  - Simultaneous multithreading (SMT)
- Summary: Node-level issues

H L R I S

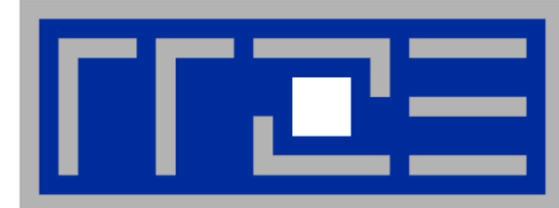
LIKWID: Lightweight Performance Tools

Contribution

- **Lightweight command line tools for Linux**
- **Help to face the challenges without getting in the way**
- **Focus on X86 architecture**
- **Philosophy:**
  - Simple
  - Efficient
  - Portable
  - Extensible

Open source project (GPL v2):

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



Scenario 1: Dealing with node topology and thread affinity

likwid-topology

likwid-pin

likwid-mpirun

likwid-topology

Single source of node information

- **Node information is usually scattered in various places**
- **`likwid-topology` provides all information in a single reliable source**
- **All information is based on cpuid directly**
- **Features:**
  - Thread topology
  - Cache topology
  - ccNUMA topology
  - Detailed cache parameters (`-c` command line switch)
  - Processor clock (measured)
  - ASCII art output (`-g` command line switch)

Usage: likwid-topology

```
-------------------------------------------------------------
CPU type:       Intel Core Westmere processor
*************************************************************
Sockets:                        2
Cores per socket:               6
Threads per core:               2
-------------------------------------------------------------
HWThread        Thread          Core            Socket
0               0               0               0
1               0               1               0
2               0               2               0
-------------------------------------------------------------
Socket 0: ( 0 12 1 13 2 14 3 15 4 16 5 17 )
Socket 1: ( 6 18 7 19 8 20 9 21 10 22 11 23 )
-------------------------------------------------------------
Cache Topology
-------------------------------------------------------------
Level:          3
Size:           12 MB
Type:           Unified cache
Associativity:  16
Number of sets: 12288
Cache line size: 64
Non Inclusive cache
Shared among 12 threads
Cache groups:   ( 0 12 1 13 2 14 3 15 4 16 5 17 ) ( 6 18 7 19 8
-------------------------------------------------------------
NUMA Topology
NUMA domains: 2
-------------------------------------------------------------
Domain 0:
 Processors:  0 1 2 3 4 5 12 13 14 15 16 17
Memory: 11615.9 MB free of total 12276.3 MB
-------------------------------------------------------------
Domain 1:
 Processors:  6 7 8 9 10 11 18 19 20 21 22 23
Memory: 12013.9 MB free of total 12288 MB
-------------------------------------------------------------
```

```
+-------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| | 0   12 | | 1   13 | | 2   14 | | 3   15 | | 4   16 | | 5   17 | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| | 256kB  | | 256kB  | | 256kB  | | 256kB  | | 256kB  | | 256kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-----------------------------------------------------------+ |
| |                           12MB                            | |
| +-----------------------------------------------------------+ |
+-------------------------------------------------------------+

+-------------------------------------------------+
| +------+ +------+ +------+ +------+ +------+ +------+ |
| |    0 | |   12 | |    1 | |   13 | |    2 | |   14 | |
| +------+ +------+ +------+ +------+ +------+ +------+ |
| +------+ +------+ +------+ +------+ +------+ +------+ |
| | 32kB| | 32kB| | 32kB| | 32kB| | 32kB| | 32kB| |
| +------+ +------+ +------+ +------+ +------+ +------+ |
| +--------------+ +--------------+ +--------------+ |
| |      3MB     | |      3MB     | |      3MB     | |
| +--------------+ +--------------+ +--------------+ |
| +-----------------------------------------------+ |
| |                     16MB                      | |
| +-----------------------------------------------+ |
+-------------------------------------------------+
```

Information can also be queried via API.

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

No pinning

Pinning (physical cores first)

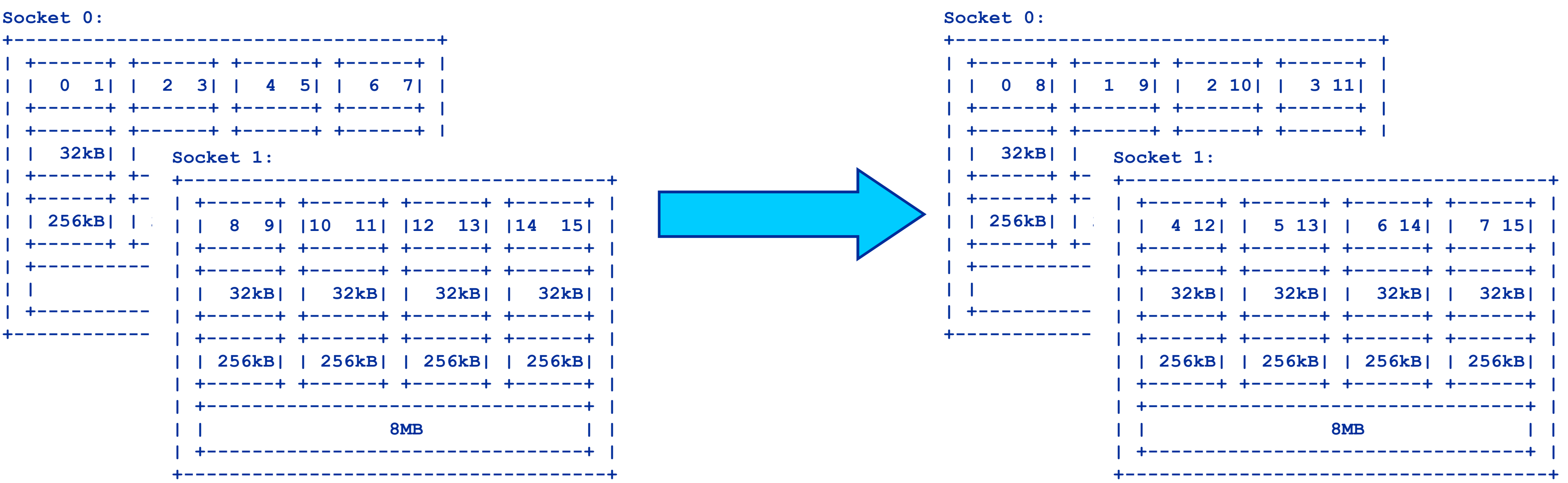
There are several reasons for caring about affinity:

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

Likwid-pin

Using logical core numbering

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



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

Likwid-pin

Using logical core numbering

- **Possible unit prefixes**

N	**node**
S	**socket**
M	**NUMA domain**
C	**outer level cache group**

Default if –c is not specified!

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
```

likwid-pin

- Pins process and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify skip mask (hybrid OpenMP/MPI)
- Can also be used as replacement for taskset

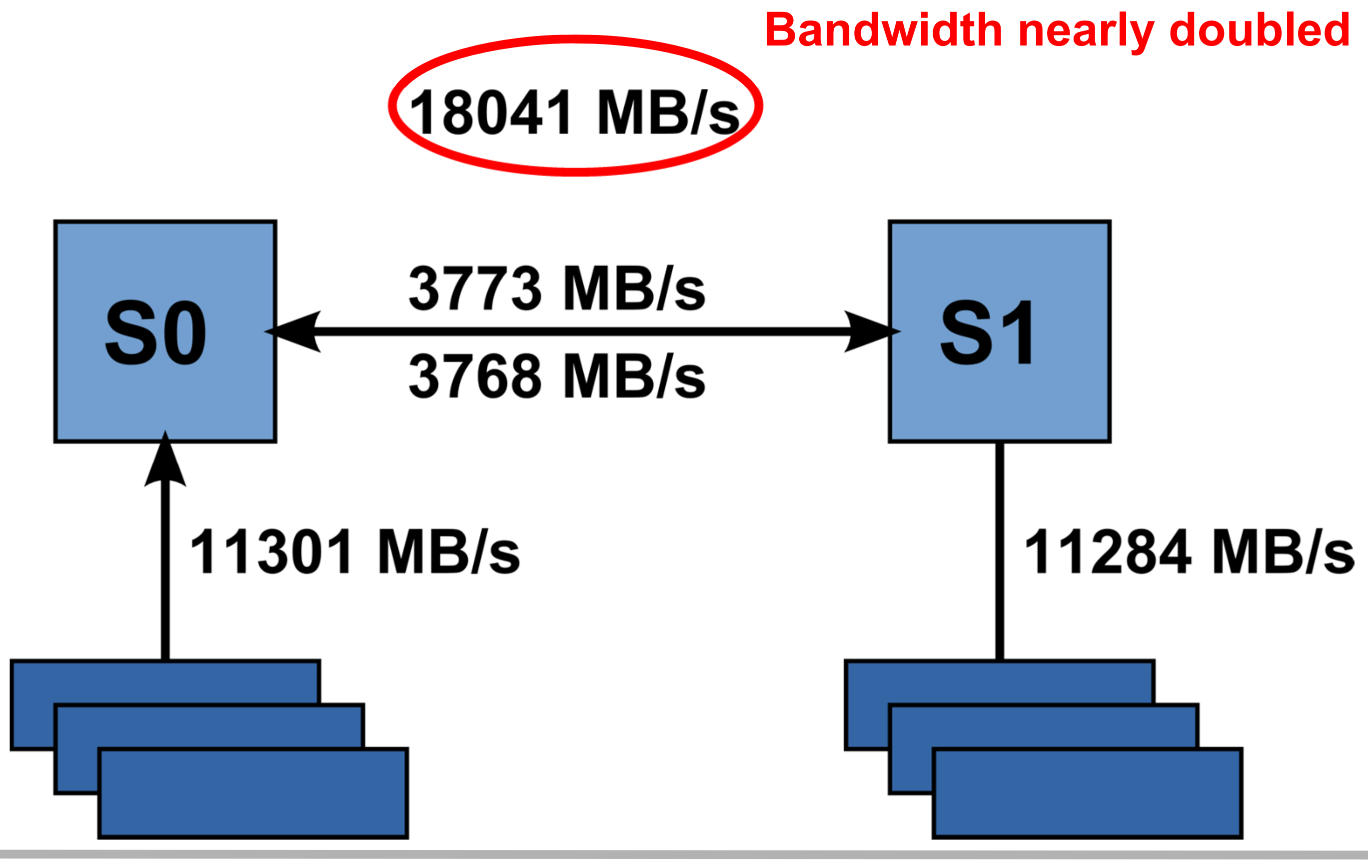
Supported usage modes:

- Physical numbering: `likwid-pin -c 0,2,5-8`
- Logical numbering (node): `likwid-pin -c N:3-7`
- Logical numbering (socket): `likwid-pin -c S0:0,2@S2:0-3`
- Logical numbering (NUMA): `likwid-pin -c M0:1-3@M2:1-3`

All logical numberings use physical cores first.

- **Effective improvement without any code change possible**
- **Memory policy is set to interleave with likwid-pin:**

```
likwid-pin -c N:0-7 -i likwid-bench -g 2 -i 1000 -t
copy -w S0:500MB:4 -w S1:500MB:4-0:S0,1:S0
```



- **In the long run a unified standard is needed**
- **Till then, likwid-mpirun provides a portable/flexible solution**
- **The examples here are for Intel MPI/OpenMP programs, but are also applicable to other threading models**

Pure MPI:

```
$ likwid-mpirun -np 16 -nperdomain S:2 ./a.out
```

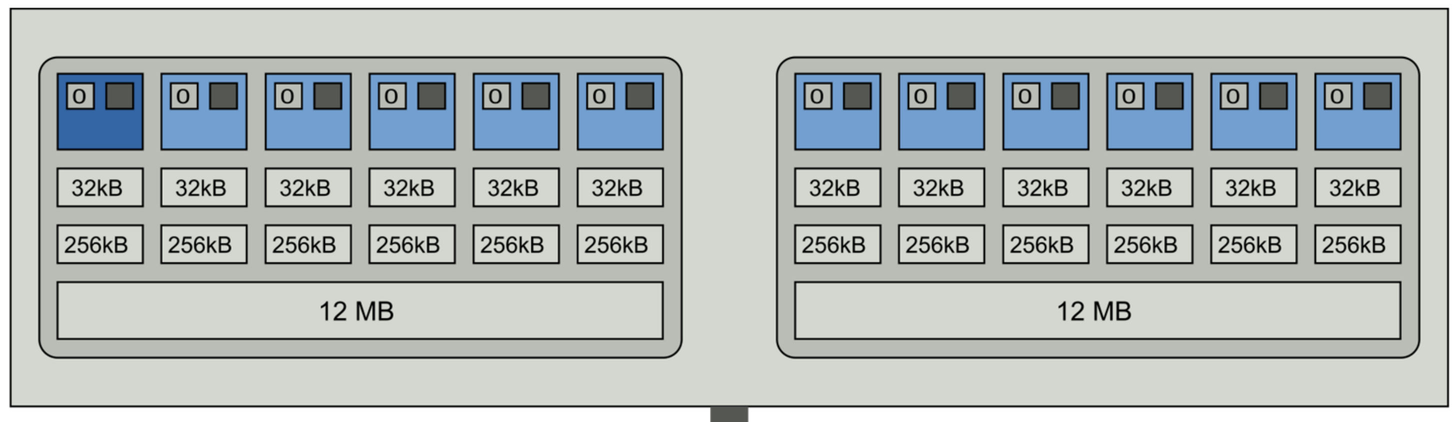
Hybrid:

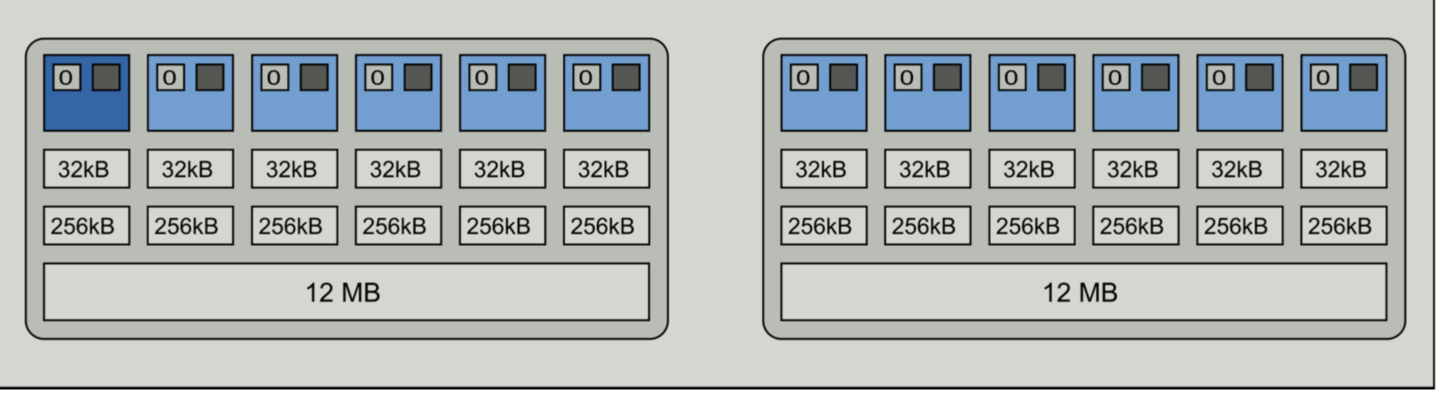
```
$ likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
```

likwid-mpirun

1 MPI process per node

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





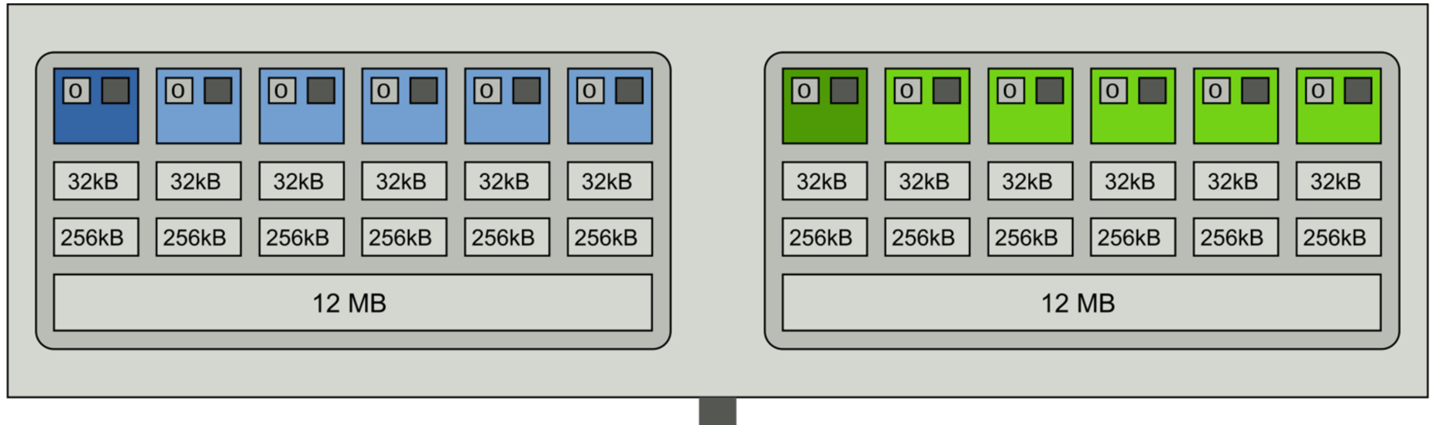
Intel MPI+compiler:

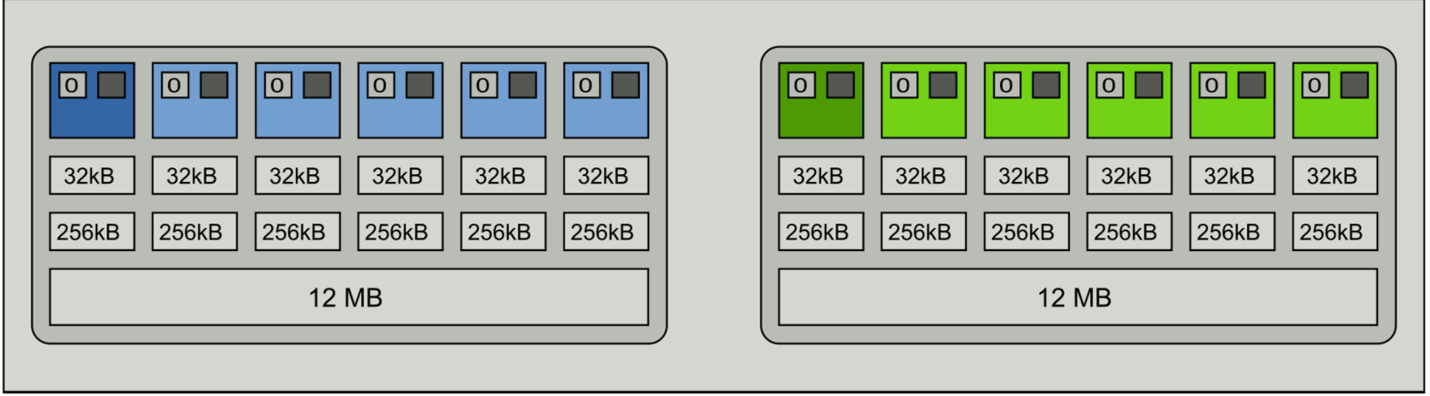
```
OMP_NUM_THREADS=12 mpirun -ppn 1 -np 2 -env KMP_AFFINITY scatter ./a.out
```

likwid-mpirun

1 MPI process per socket

```
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
```

- **likwid-mpirun can optionally set up likwid-perfctr for you**

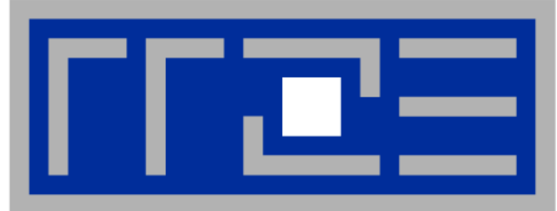
```
$ likwid-mpirun -np 16 -nperdomain S:2 -perf FLOPS_DP \
     -marker -mpi intelmpi  ./a.out
```

- **likwid-mpirun generates an intermediate perl script which is called by the native MPI start mechanism**
- **According the MPI rank the script pins the process and threads**

- **If you use perfctr after the run for each process a file in the format `Perf-<hostname>-<rank>.txt`**

  Its output which contains the perfctr results.

- **In the future analysis scripts will be added which generate reports of the raw data (e.g. as html pages)**



Scenario 2: Hardware performance monitoring and Node performance characteristics

likwid-perfctr

likwid-bench

likwid-powermeter

- **A coarse overview of hardware performance monitoring data is often sufficient**
  - likwid-perfctr (similar to "perfex" on IRIX, "hpmcount" on AIX, "lipfpm" on Linux/Altix, "craypat" on Cray systems)
  - Simple end-to-end measurement of hardware performance metrics
  - Operating modes:
    - Wrapper
    - Stethoscope
    - Timeline
    - Marker API
  - 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 for Wrapper mode

```
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -t intel -g FLOPS_DP  ./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  |
+--------------------------+------------+---------+----------+----------+
```

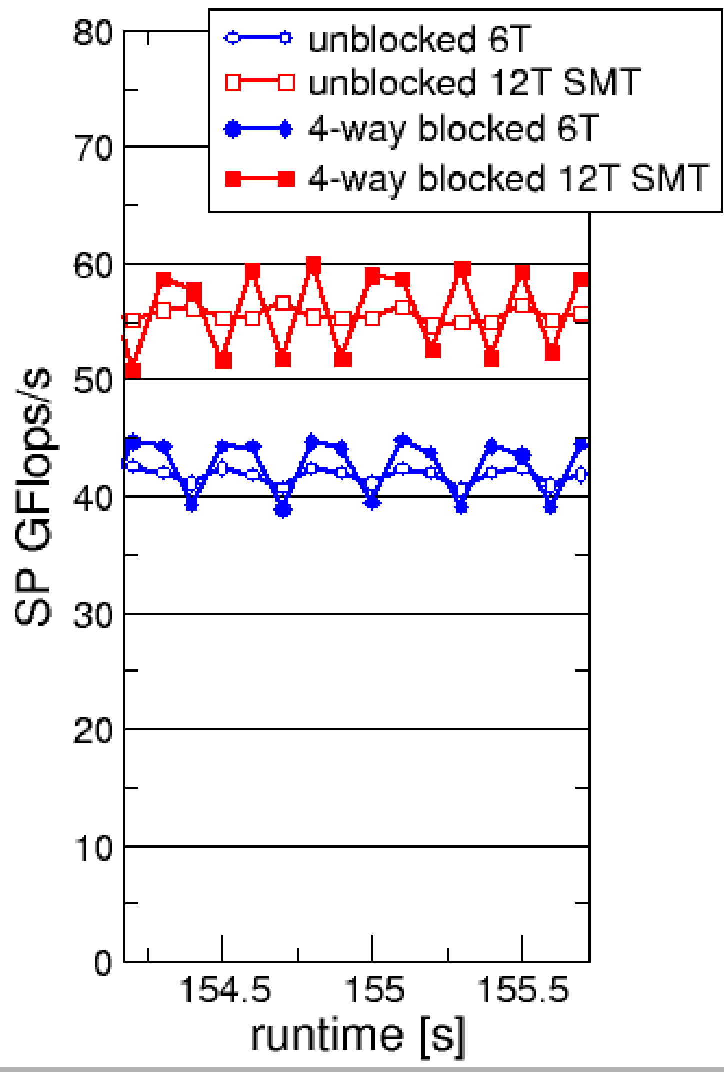
- **likwid-perfctr measures what happens on the cores; no connection to the running binary/ies exists**
- **This allows to listen on what currently happens without any overhead:**

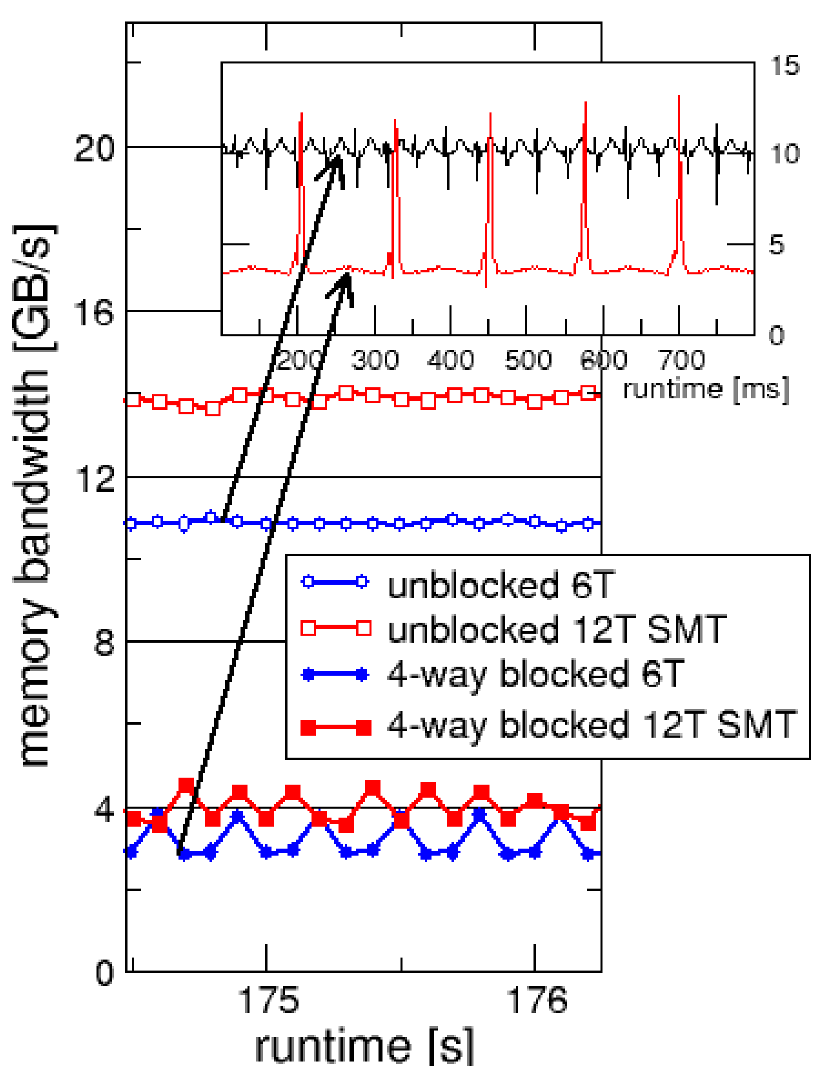
```
$ likwid-perfctr -c N:0-11 -g FLOPS_DP  -s 10
```

- **It can be used as cluster/server monitoring tool**
- **A frequent use is to measure a certain part of a long running parallel application from outside**

- **likwid-perfctr supports time resolved measurements of full node:**

```
$ likwid-perfctr -c N:0-11 -g MEM -d 50ms  > out.txt
```





likwid-perfctr
Marker API

- To measure only parts of an application a marker API is available.
- The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr application.
- Multiple named regions can be measured
- Results on multiple calls are accumulated
- Inclusive and overlapping Regions are allowed

```
likwid_markerInit();  // must be called from serial region

likwid_markerStartRegion("Compute");
. . .
likwid_markerStopRegion("Compute");


likwid_markerStartRegion("postprocess");
. . .
likwid_markerStopRegion("postprocess");


likwid_markerClose();  // must be called from serial region
```

likwid-perfctr

Group files

```
SHORT PSTI
EVENTSET
FIXC0 INSTR_RETIRED_ANY
FIXC1 CPU_CLK_UNHALTED_CORE
FIXC2 CPU_CLK_UNHALTED_REF
PMC0  FP_COMP_OPS_EXE_SSE_FP_PACKED
PMC1  FP_COMP_OPS_EXE_SSE_FP_SCALAR
PMC2  FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION
PMC3  FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION
UPMC0  UNC_QMC_NORMAL_READS_ANY
UPMC1  UNC_QMC_WRITES_FULL_ANY
UPMC2 UNC_QHL_REQUESTS_REMOTE_READS
UPMC3 UNC_QHL_REQUESTS_LOCAL_READS
METRICS
Runtime [s] FIXC1*inverseClock
CPI  FIXC1/FIXC0
Clock [MHz]  1.E-06*(FIXC1/FIXC2)/inverseClock
DP MFlops/s (DP assumed) 1.0E-06*(PMC0*2.0+PMC1)/time
Packed MUOPS/s   1.0E-06*PMC0/time
Scalar MUOPS/s 1.0E-06*PMC1/time
SP MUOPS/s 1.0E-06*PMC2/time
DP MUOPS/s 1.0E-06*PMC3/time
Memory bandwidth [MBytes/s] 1.0E-06*(UPMC0+UPMC1)*64/time;
Remote Read BW [MBytes/s] 1.0E-06*(UPMC2)*64/time;
LONG
Formula:
DP MFlops/s =  (FP_COMP_OPS_EXE_SSE_FP_PACKED*2 +  FP_COMP_OPS_EXE_SSE_FP_SCALAR)/ runtime.
```

- Groups are architecture specific
- They are defined in simple text files
- During recompile the code is generated
- likwid-perfctr -a outputs list of groups
- For every group an extensive documentation is available

Likwid supports to specify an output file with placeholder for:

- %j - PBS_JOBID taken from environment
- %r - MPI Rank as specified by newer Intel MPI versions
- %h - hostname
- %p - process id

Example:

```
likwid-perfctr -C L:0 -g FLOPS_DP -o test_%h_%p.txt  ./a.out
```

Depending on the file suffix an optional converter script is called:

- txt Direct output without conversion
- csv Convert to comma separated values format
- xml Convert to xml format

Useful for integration in other tool chains or automated frameworks.

- **Implemented completely in user space (uses msr kernel module)**
- **For security-sensitive environments a small proxy application managing a controlled access to the msr device files is available**
- **Supported processors:**
  - Intel Core 2
  - Intel Nehalem /Westmere (all variants) supporting Uncore events
  - Intel NehalemEX/WestmereEX (with Uncore)
  - Intel Sandy Bridge (without Uncore)
  - AMD K8/K10
  - AMD Interlagos
- **likwid-perfctr allows to specify arbitrary event sets on the command line:**

```
$ likwid-perfctr -c N:0-11 -g
  INSTR_RETIRED_ANY:FIXC0,CPU_CLK_UNHALTED_CORE:FIXC1,\
  FP_COMP_OPS_EXE_SSE_FP_PACKED:PMC0,\
  UNC_L3_LINES_IN_ANY:UPMC0 -s 10
```

- likwid-perfctr can be used with MPI if processes are pinned
- For hybrid usage you can pin logically inside a cpuset
- To distinguish the output it can be written to separate files

```
$ likwid-perfctr -C L:0 -g FLOPS_DP -o myTag_%r_%h ./app
```

- There are efforts to add likwid support in Scalasca (and Vampir ?)
- likwid-mpirun provides integrates perfctr support

likwid-bench
Microbenchmarking application/platform

- To know the performance properties of a machine is essential for any optimization effort
- Microbenchmarking is an important method to gain this information

- Extensible, flexible benchmarking framework
- Rapid development of low-level kernels
- Already includes many ready to use threaded benchmark kernels

- Benchmarking runtime cares for:
  - Thread management and placement
  - Data allocation and NUMA-aware initialization
  - Timing and result presentation

likwid-bench Example

- Implement micro benchmark in abstract assembly
- Add meta information
- The benchmark file is automatically converted, compiled and added to the benchmark application

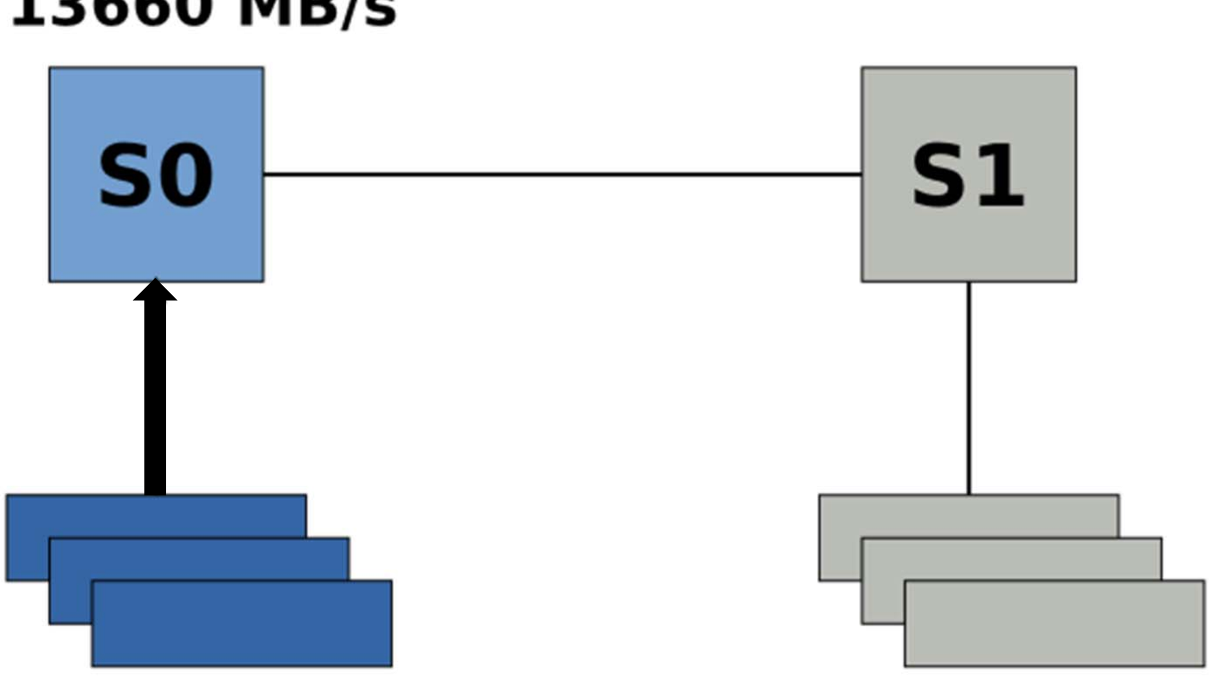
```
$likwid-bench -t clcopy -g 1 -i 1000 -w S0:1MB:2
$likwid-bench -t load -g 2 -i 100 -w S1:1GB  -w S0:1GB-0:S1,1:S0
```

```
STREAMS 2
TYPE DOUBLE
FLOPS 0
BYTES 16
LOOP  32
movaps       FPR1, [STR0 + GPR1 * 8 ]
movaps       FPR2, [STR0 + GPR1 * 8 + 64 ]
movaps       FPR3, [STR0 + GPR1 * 8 + 128 ]
movaps       FPR4, [STR0 + GPR1 * 8 + 192 ]
movaps       [STR1 + GPR1 * 8 ], FPR1
movaps       [STR1 + GPR1 * 8 + 64 ], FPR2
movaps       [STR1 + GPR1 * 8 + 128 ], FPR3
movaps       [STR1 + GPR1 * 8 + 192 ], FPR4
```

Measuring node bandwidths 1

- 1 thread group on socket 0

```
likwid-bench -g 1 -i 50 -t copy -w S0:1GB:6
```



Detecting NUMA problems 4

- 1 thread group with 6 threads on socket 0
- Memory placed on socket 1

```
likwid-bench -g 1 -i 50 -t copy -w S0:1GB:6-0:S1,1:S1
```

9517 MB/s

S0

S1

Measuring energy consumption
likwid-powermeter

- Implements Intel RAPL interface (Sandy Bridge)
- RAPL (Running average power limit)

```
-------------------------------------------------------------
CPU name:        Intel Core SandyBridge processor
CPU clock:       3.49 GHz
-------------------------------------------------------------
Base clock:      3500.00 MHz
Minimal clock:   1600.00 MHz
Turbo Boost Steps:
C1 3900.00 MHz
C2 3800.00 MHz
C3 3700.00 MHz
C4 3600.00 MHz
-------------------------------------------------------------
Thermal Spec Power: 95 Watts
Minimum  Power: 20 Watts
Maximum  Power: 95 Watts
Maximum  Time Window: 0.15625 micro sec
-------------------------------------------------------------
```

Likwid-perfctr example

RAPL Integration

```
$ likwid-perfctr -c S1:0-3 -g ENERGY -m likwid-bench  \
      -g 1 -i 50 -t stream_avx  -w S1:1GB:4
```

Shortened output:

Metric	core 8	core 9	core 10	core 11
Runtime [s]	2.39535	2.39481	2.39494	2.39493
Runtime rdtsc [s]	2.03051	2.03051	2.03051	2.03051
Clock [MHz]	3192.14	3192.13	3192.14	3192.12
CPI	10.0977	10.1713	10.2047	10.2526
Energy [J]	146	0	0	0
Power [W]	71.9031	0	0	0

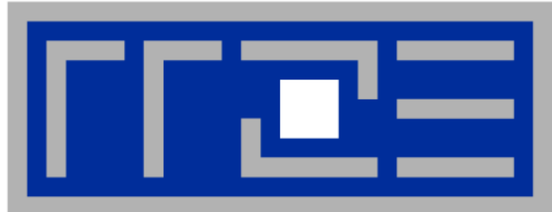
Live demo:

LIKWID tools

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**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Microbenchmarking with simple parallel loops
  - Bandwidth saturation effects in cache and memory
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - ccNUMA effects and how to circumvent performance penalties
  - Simultaneous multithreading (SMT)
- **Summary: Node-level issues**

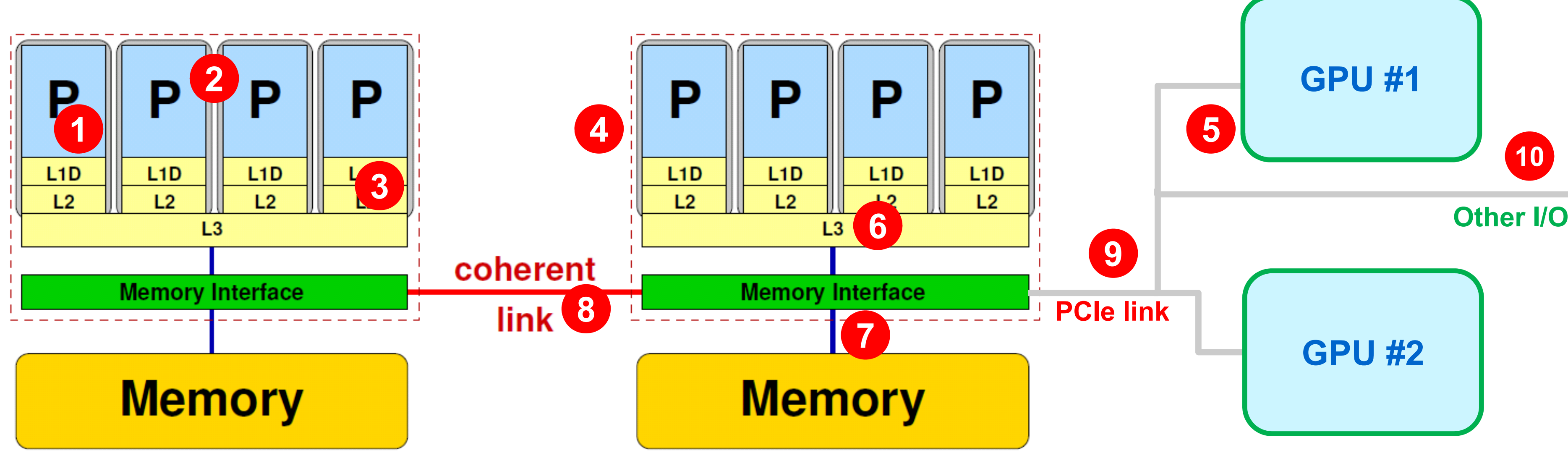
H L R I S



General remarks on the performance properties of multicore multisocket systems

Parallelism in modern computer systems

- Parallel and shared resources within a shared-memory node



Parallel resources:

- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / memory domains 4
- Multiple accelerators 5

Shared resources:

- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

How does your application react to all of those details?

Simple streaming benchmark:

```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
```

Prevents smarty-pants compilers from doing "clever" stuff

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!

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);
```

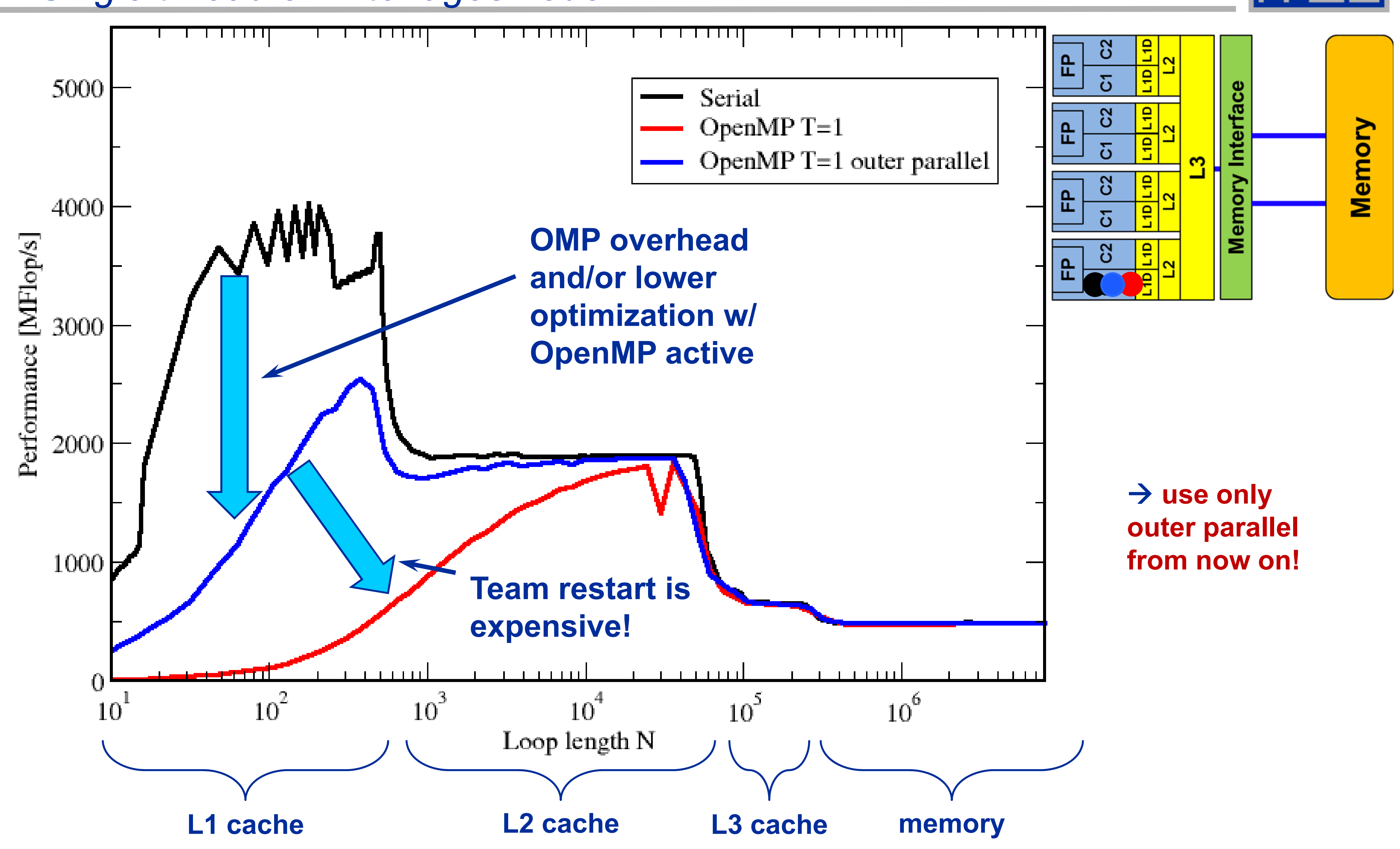
(In both `#pragma omp parallel for` lines, "parallel" is struck through.)

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

Large-N version (NT)

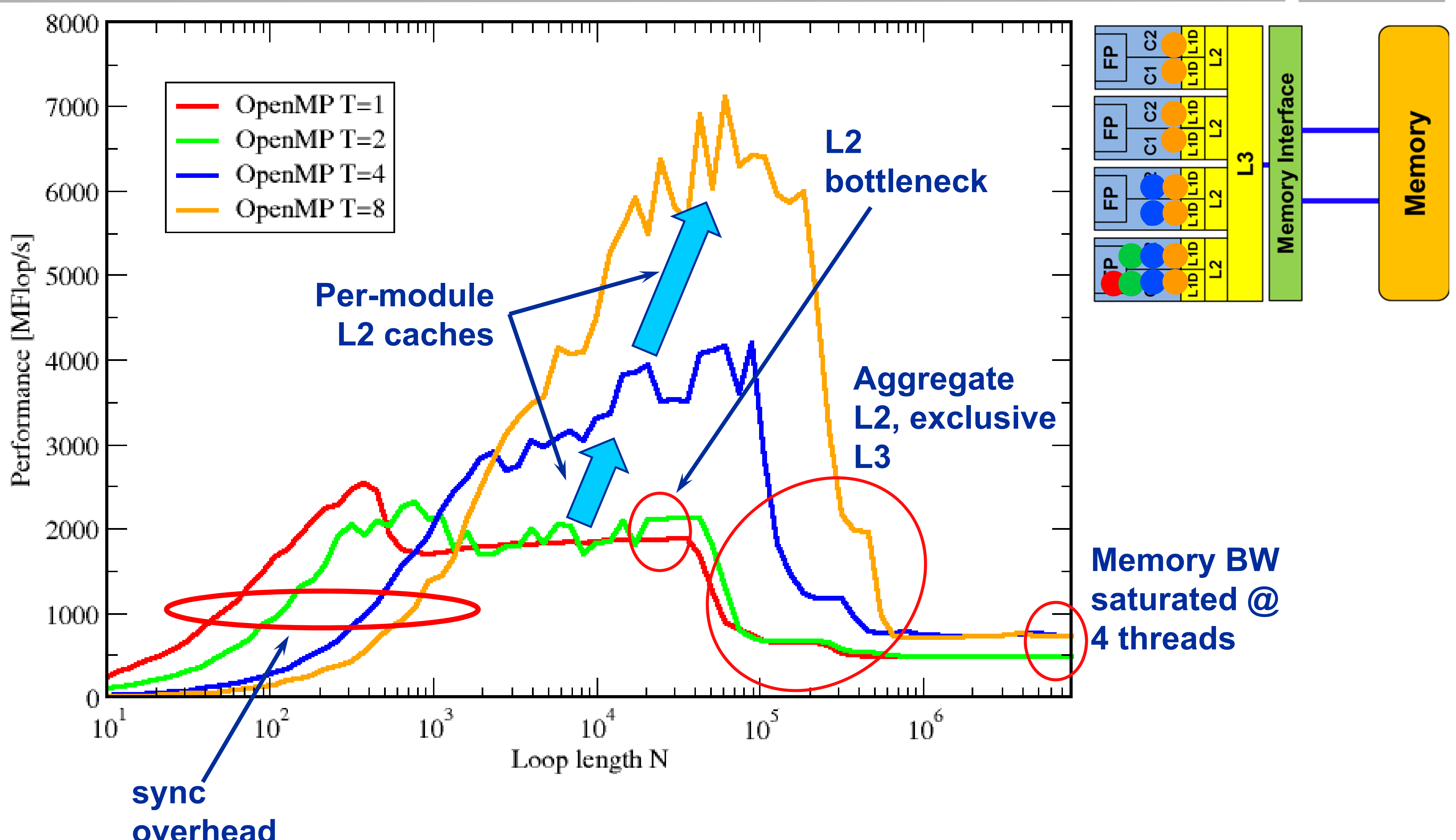
Small-N version (noNT)

The parallel vector triad benchmark
Single thread on Interlagos node



The parallel vector triad benchmark

Intra-chip scaling on Interlagos node



The parallel vector triad benchmark

Nontemporal stores on Interlagos node

Performance [MFlop/s]

8000
7000
6000
5000
4000
3000
2000
1000
0

OpenMP T=8
OpenMP T=8 NT

Loop length N

10^1 10^2 10^3 10^4 10^5 10^6

NT stores hazardous if data in cache

slow L3

25% speedup for vector triad in memory via NT stores

FP
L1D
L2
L3
Memory Interface
Memory

The parallel vector triad benchmark

Topology dependence on Interlagos node

Performance [MFlop/s]

Loop length N

- OpenMP T=8
- OpenMP T=8 S=1 C=2
- OpenMP T=8 S=2 C=2

Memory
Memory Interface
L3
L2
L1D
C1 C2
FP

more aggregate L3 with more chips

bandwidth scalability across memory interfaces

sync overhead nearly topology-independent @ constant thread count

The parallel vector triad benchmark
Inter-chip scaling on Interlagos node

OpenMP T=8
OpenMP T=16
OpenMP T=24
OpenMP T=32

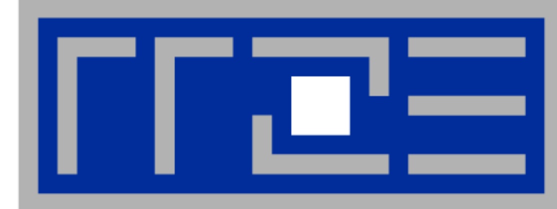
Performance [MFlop/s]

Loop length N

sync overhead grows with core/chip count

bandwidth scalability across memory interfaces

H L R I S



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(:))**

Westmere
Sandy Bridge
Interlagos

Saturation with 3 threads

Saturation with 2 threads

1 thread cannot saturate bandwidth

Saturation with 4 threads

1 NUMA domain | 2 NUMA domains

Bandwidth [GByte/s]

cores

Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache

Bandwidth [GByte/s]

300
250
200
150
100
50
0

0
4
8
12
16

cores

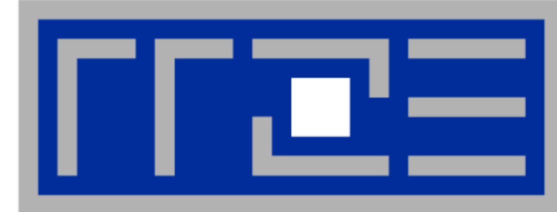
Westmere (L3)
Sandy Bridge (L3)
Interlagos (L3)
Interlagos (L2)

Intel SB:
New scalable
L3 design

AMD:
Optimize for L2 cache!

1 NUMA domain
2 NUMA domains

H L R I S



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**

- **Case 1: Large matrix**

cant, 62451x62451, non-zero: 4007383

CRS-magnycours

MFLOPS/s

threads

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**

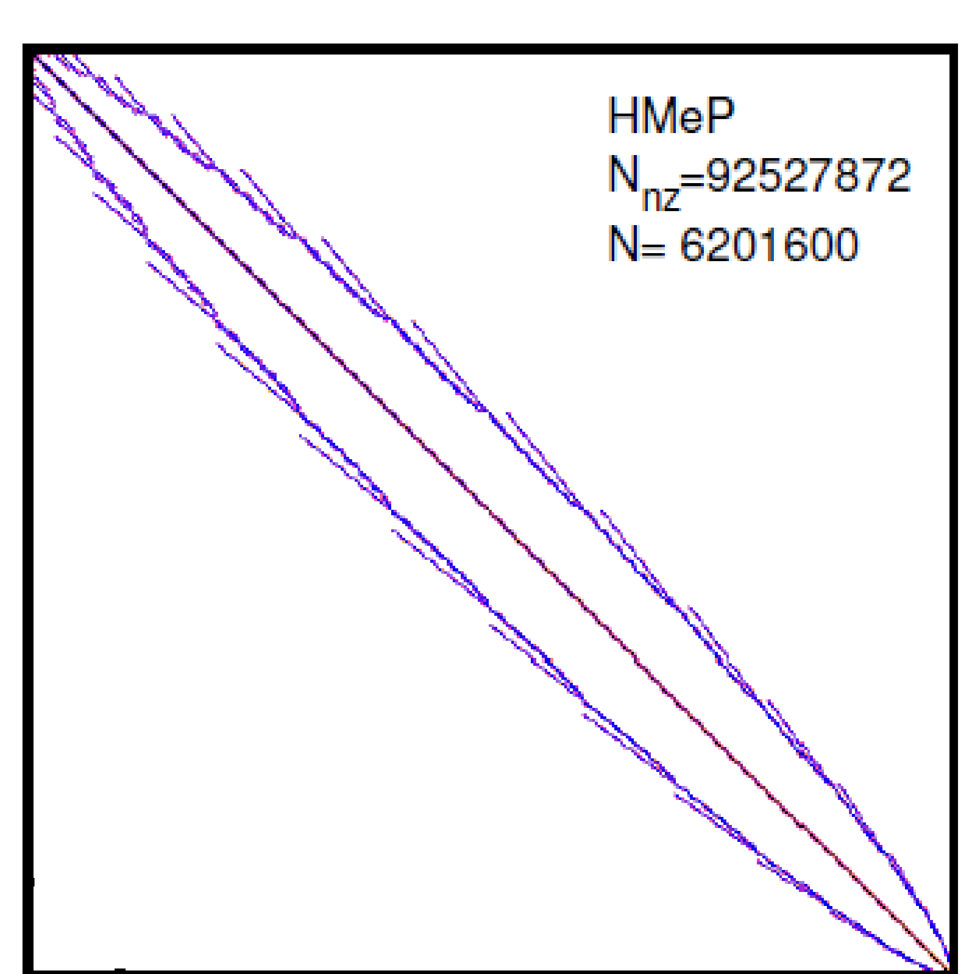
No bandwidth bottleneck

Parallelization overhead dominates

rbs480a, 480x480, non-zero: 17088

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


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

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

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

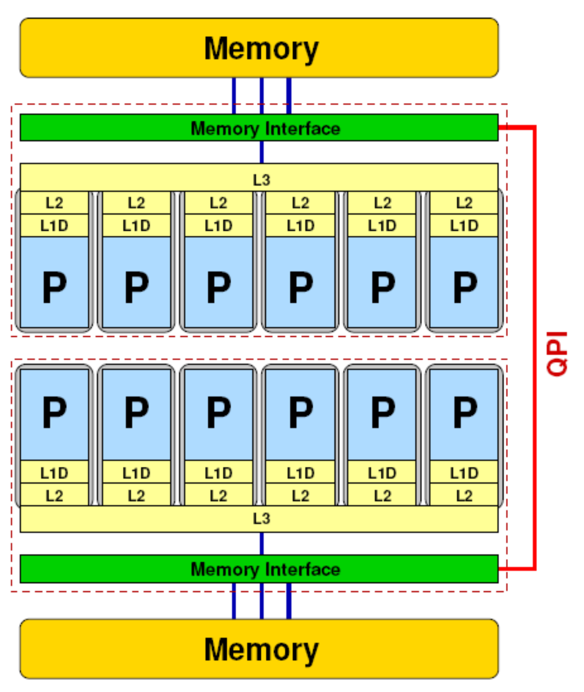
Test matrices: Sparsity patterns

- **Analysis for HMeP matrix (N_{nzr}≈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}≈15 times
  → about 25% of BW goes into RHS

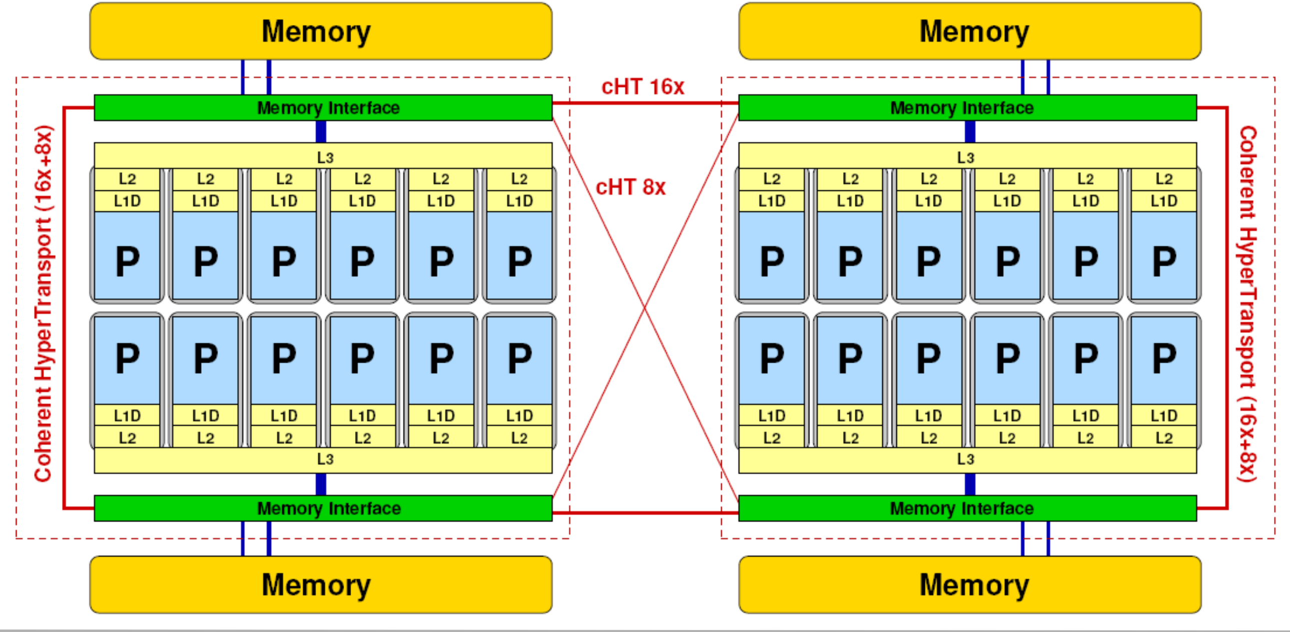
- **Special formats that exploit features of the sparsity pattern are not considered here**
  - Symmetry
  - Dense blocks
  - Subdiagonals (possibly w/ constant entries)

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)

bandwidth STREAM:Triad (Westmere)
bandwidth STREAM:Triad (MagnyCours)
performance spMVM (HMeP -- Westmere)
performance spMVM (HMeP -- MagnyCours)

bandwidth [GB/s]
50
40
30
20
10
0

performance [GFlop/s]
9
8
7
6
5
4
3
2
1
0

1
2
3
4
5
6
cores
1 AMD socket
1 node

18.1 GB/s

Good scaling across NUMA domains

Cores useless for computation!

OpenMP sparse MVM:
Take-home messages

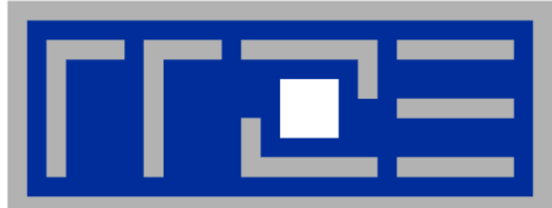
- **Yes, sparse MVM is usually memory-bound**

- **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

H L R I S



Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

First touch placement policy

C++ issues

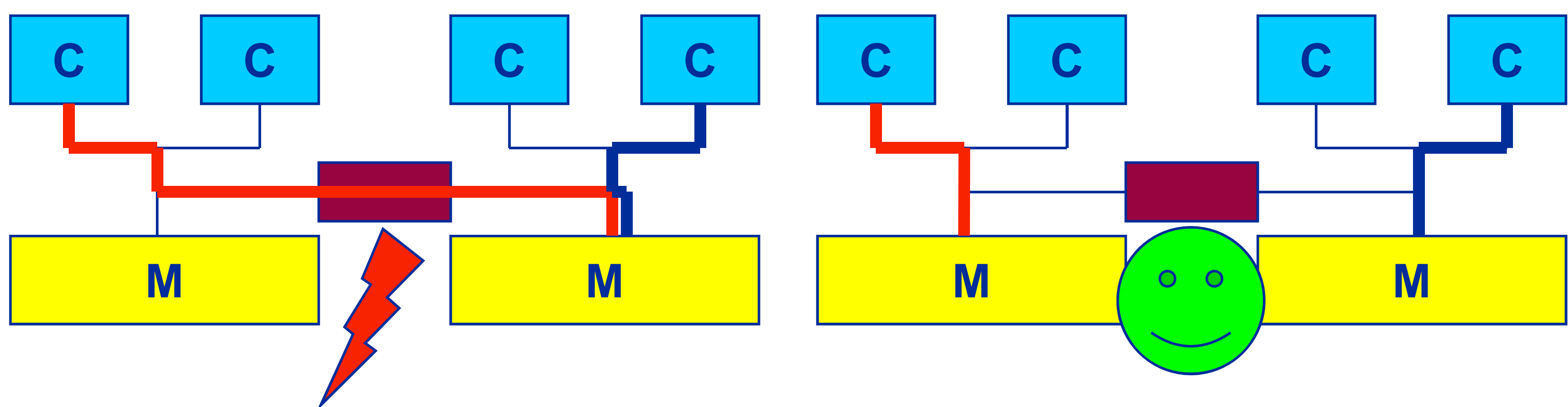
ccNUMA locality and dynamic scheduling

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

	0	1	2	3
0				
1				
2				
3				

8.8 GB/s

5.0 GB/s

4.2 GB/s

AMD Magny Cours 4-socket system

Topology at its best?

0 1 2 3 4 5 6 7

8.7 GB/s
5.1 GB/s
4.3 GB/s
3.7 GB/s
2.7 GB/s
2.0 GB/s

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;
```


- **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

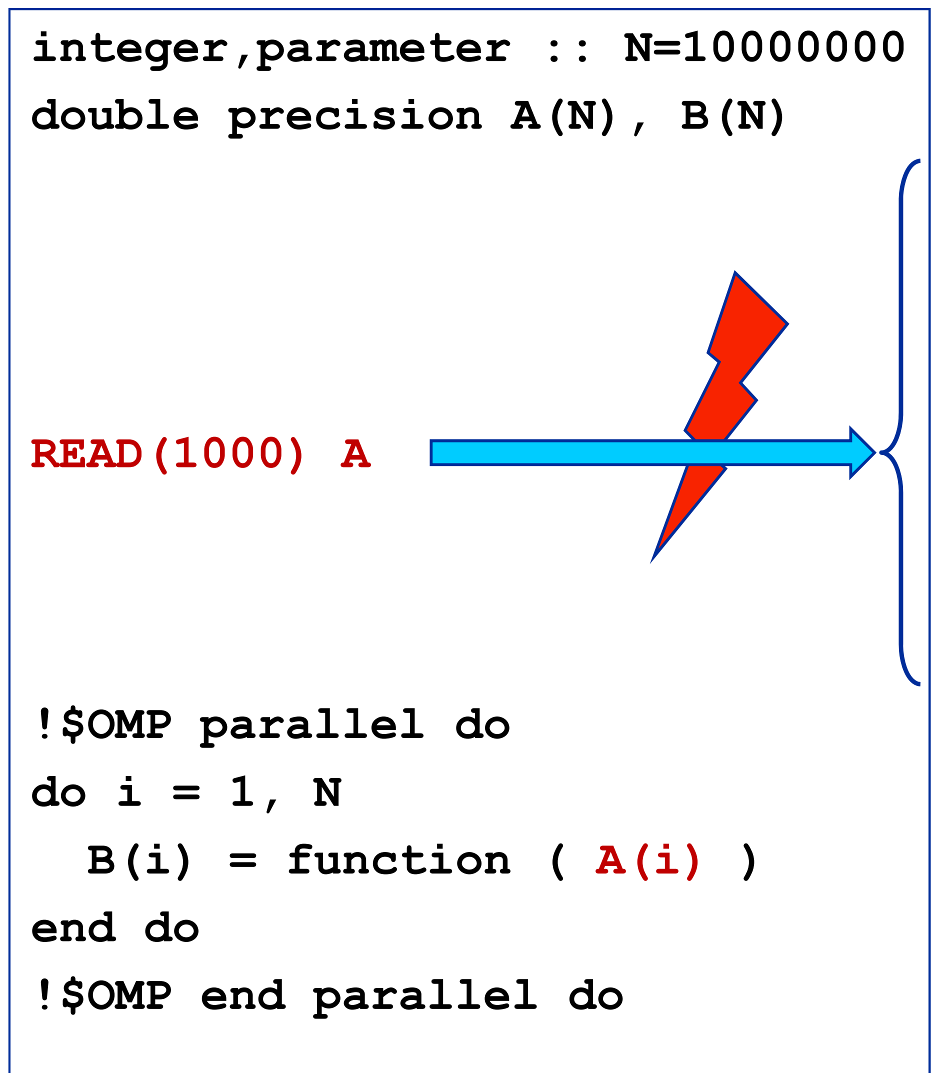
- **Most simple case: explicit initialization**

```
integer,parameter :: N=10000000
double precision A(N), B(N)


A=0.d0


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

```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Coding for ccNUMA data locality

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




```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

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

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

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

→ **placement problem with**
`D* array = new D[1000000];`

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

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

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

parallel first touch

placement `new`!

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

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

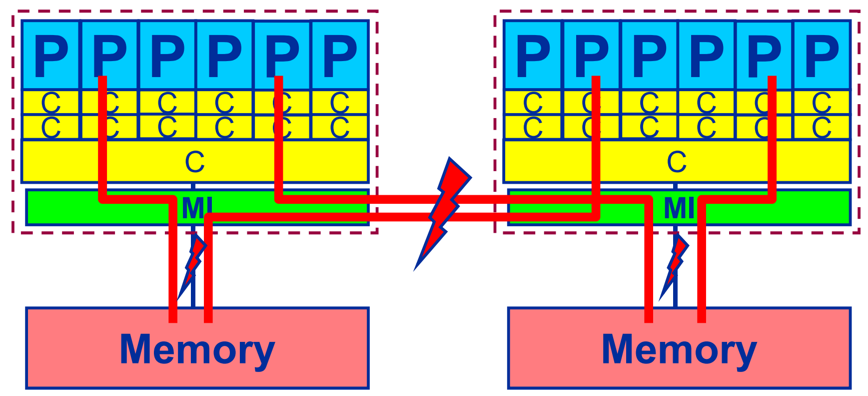
Application:

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

Memory Locality Problems

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

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



Diagnosing Bad Locality

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

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

Using performance counters for diagnosing bad ccNUMA access locality

- Intel Nehalem EP node:

Uncore events only counted once per socket

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

RDTSC timing: 0.827196 s

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

Half of read BW comes from other socket!

ccNUMA placement and erratic access patterns

OPTIONAL

- 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

OPTIONAL

- **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>**

parallel init | LD0 | interleaved

Bandwidth [Mbyte/s]

120000
100000
80000
60000
40000
20000
0

1 2 3 4 5 6 7 8

 # NUMA domains (6 threads per domain)

If all fails…

OPTIONAL

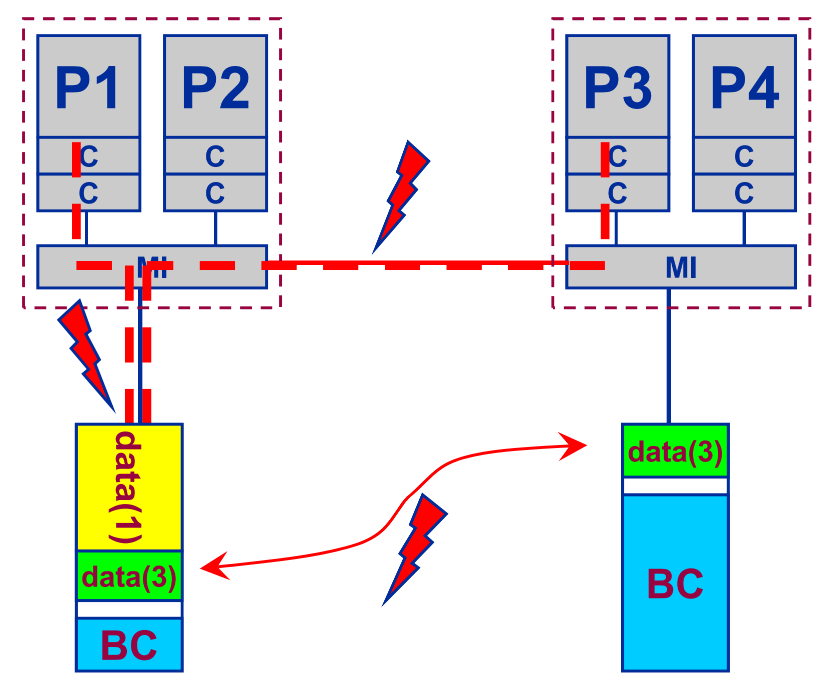
- **Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?**
  - Program has erratic access patters → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```
# 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



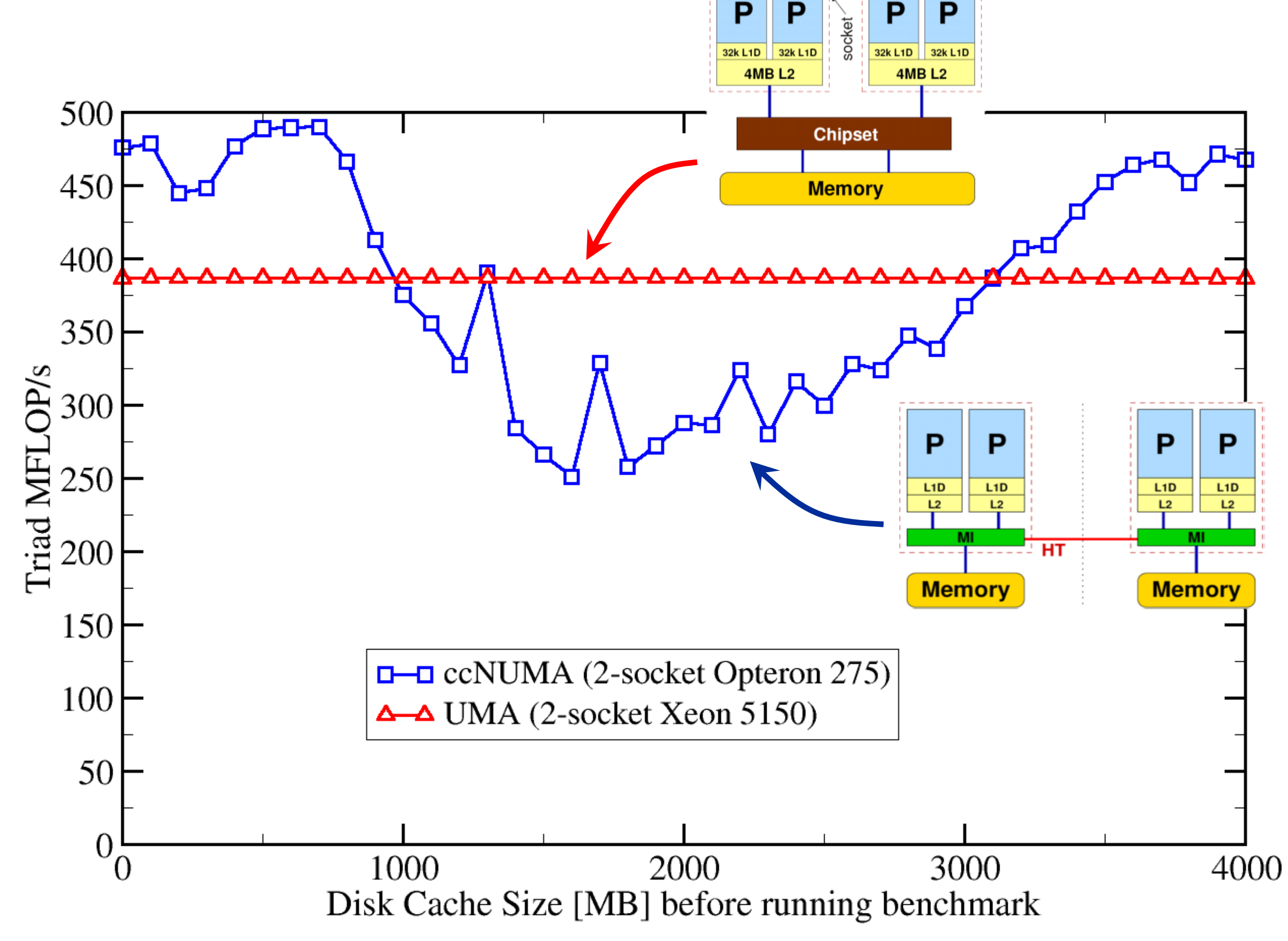
- **Remedies**
  - Drop FS cache pages after user job has run (admin's job)
  - User can run "sweeper" code that allocates and touches all physical memory before starting the real application
  - `numactl` tool can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer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

ccNUMA problems beyond first touch:
Buffer cache

OPTIONAL

- Real-world example: ccNUMA vs. UMA and the Linux buffer cache
- Compare two 4-way systems: AMD Opteron ccNUMA vs. Intel UMA, 4 GB main memory

- Run 4 concurrent triads (512 MB each) after writing a large file
- Report performance vs. file size
- Drop FS cache after each data point



H L R I S

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.

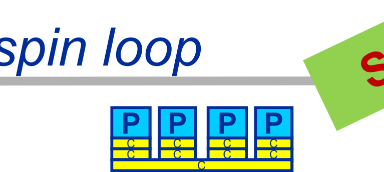
Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

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

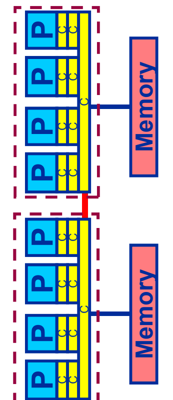


4 Threads	Q9550	i7 920 (shared L3)
pthreads_barrier_wait	42533	9820
omp barrier (icc 11.0)	977	814
gcc 4.4.3	41154	8075
Spin loop	1106	475

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

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**

P P P P
Chipset
Memory

static baseline
Performance [MIterations/s]
Chunksize
1S dynamic
1S static
1S guided
2S dynamic
2S static
2S guided
Overhead [cycles]
1S dynamic
2S dynamic
2S static

HLRS

Simultaneous multithreading (SMT)

Principles and performance impact
SMT vs. independent instruction streams
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):**

Standard core

Memory

L2 cache

L1D cache

L1I cache

Registers

Control

Execution units

2-way SMT

Memory

L2 cache

L1D cache

L1I cache

Registers

Control

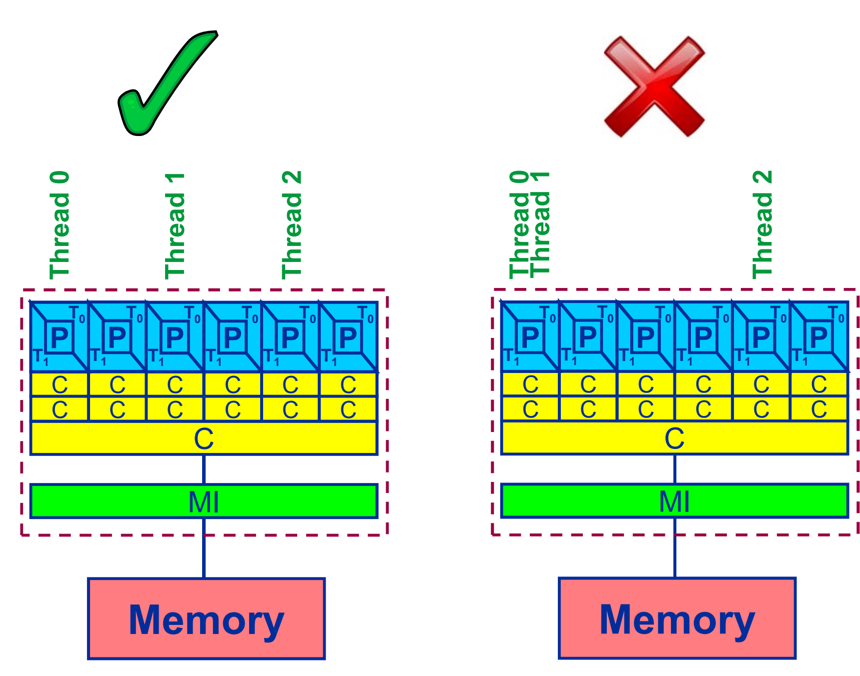
Execution units

SMT impact

OPTIONAL

- **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.



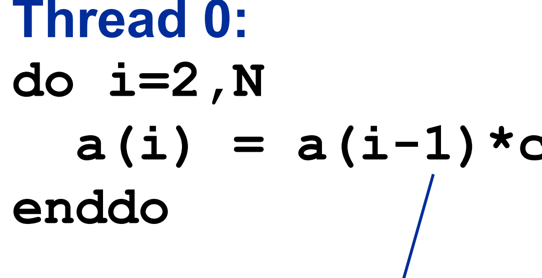
SMT impact

OPTIONAL

Westmere EP

Memory

- **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 1:

```
do i=2,N
  b(i) = b(i-1)*d+s
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=2,N
  a(i) = a(i-1)*c
  b(i) = b(i-1)*d+s
enddo
```

Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

MULT Pipeline depth: 5 stages → 1 F / 5 cycles for recursive update

Fill bubbles via:

- SMT
- Multiple streams

Thread 0:	Thread 1:

`do i=1,N`
`A(i)=A(i-1)*c`
`B(i)=B(i-1)*d`
`enddo`
	`do i=1,N`
	`A(i)=A(i-1)*c`
	`B(i)=B(i-1)*d`
	`enddo`

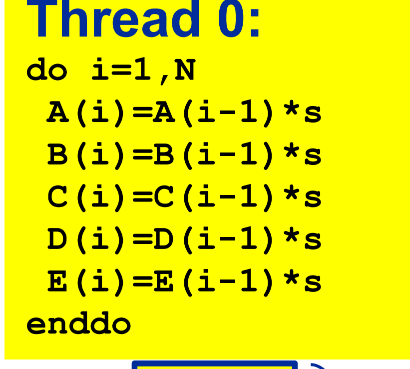
Simultaneous recursive updates with SMT

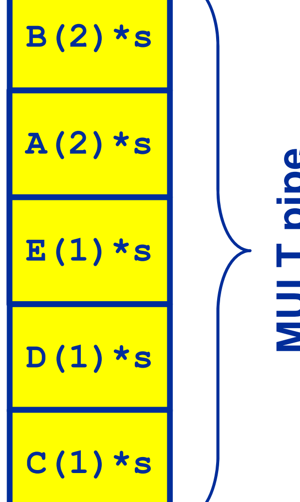
OPTIONAL

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

MULT Pipeline depth: 5 stages → 1 F / 5 cycles for recursive update

4000
3000
2000
1000
0
MF/s
10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7
N
1 F/cycle
Full MULT pipeline
A(i)=A(i-1)*s
...
E(i)=E(i-1)*s
SMT
A(i)=A(i-1)*s B(i)=B(i-1)*s
A(i)=A(i-1)*s B(i)=B(i-1)*s
SMT
A(i)=A(i-1)*s A(i)=A(i-1)*s
1/5 F/cycle
A(i)=A(i-1)*s
MULT pipeline execution speed





5 independent updates on a single thread do the same job!

Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

Pure update benchmark can be vectorized → 2 F / cycle (store limited)

MF/s

7000
6000
5000
4000
3000
2000
1000
0

10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7

N

2 F/cycle

A(i)=A(i)*s [SIMD]

Full MULT pipeline

1 F/cycle

5 rec. streams

SMT: 2 rec. streams

SMT: 1 rec. stream

A(i)=A(i-1)*s

1/5 F/cycle

MULT pipeline execution speed

Recursive update:

- SMT can fill pipeline bubles
- A single thread can do so as well
- Bandwidth does not increase through SMT
- **SMT can not replace SIMD!**

SMT myths: Facts and fiction (1)

OPTIONAL

- **Myth: “If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement.”**

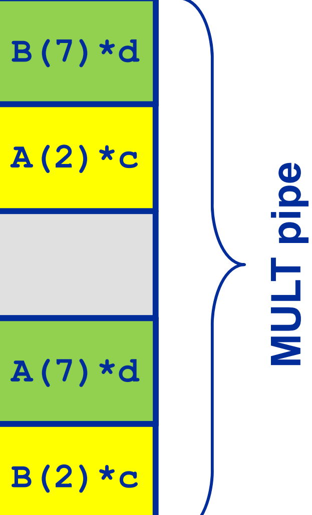
- **Truth**
  1. **A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.**
  2. **If a pipeline is already full, SMT will not improve its utilization**

Thread 0:

```
do i=1,N
A(i)=A(i-1)*c
B(i)=B(i-1)*d
enddo
```

Thread 1:

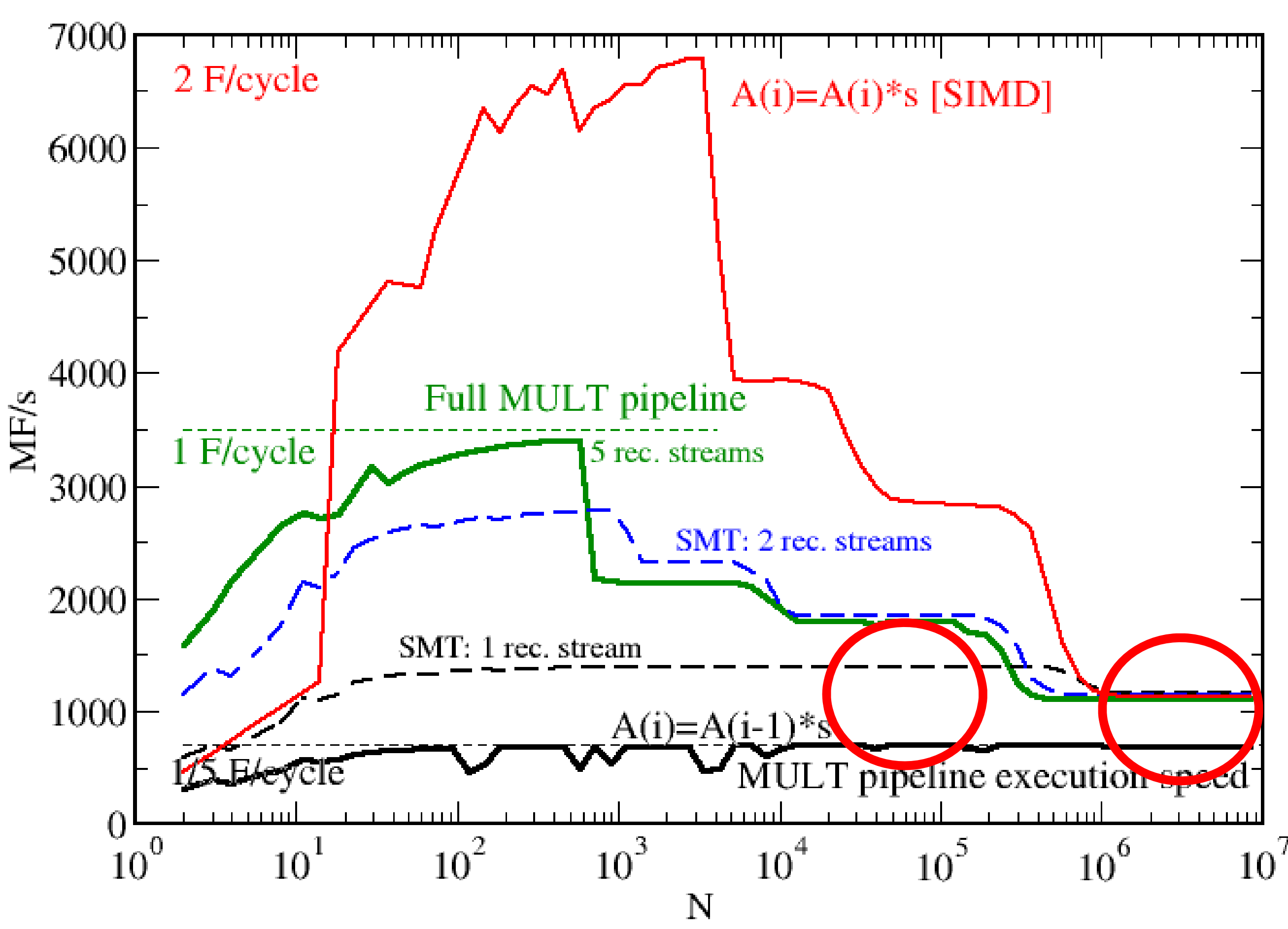
```
do i=1,N
A(i)=A(i-1)*c
B(i)=B(i-1)*d
enddo
```



SMT myths: Facts and fiction (2)

OPTIONAL

- **Myth: "If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory."**
- **Truth:**
  1. **If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.**
  2. **If the maximum memory bandwidth is not reached, SMT may help since it can fill bubbles in the LOAD pipeline.**

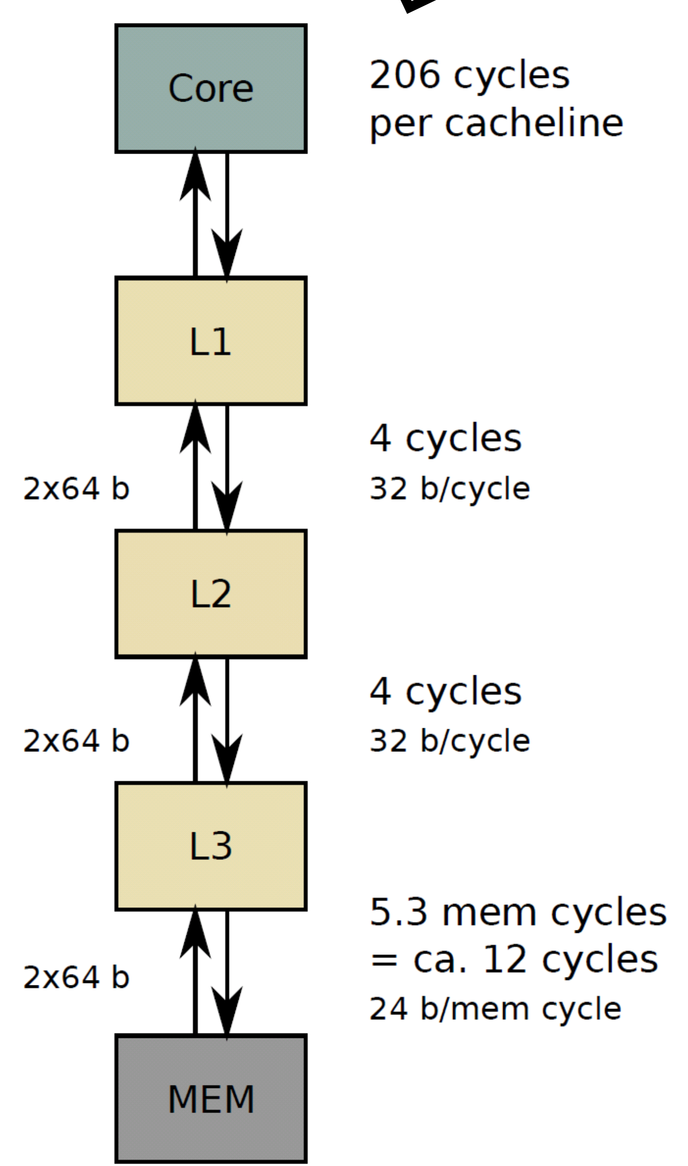


SMT myths: Facts and fiction (3)

OPTIONAL

- **Myth: "SMT can help bridge the latency to memory (more outstanding references)."**

- **Truth:**
  Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets "wasted" in the cache hierarchy.



SMT: When it may help, and when not

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

H L R S

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

QDR np2
QDR np4
QDR np8
DDR np2
DDR np4
DDR np8
GBit np2
GBit np4
GBit np8

QDR Limit (4 GB/s)

DDR Limit (2 GB/s)

QDR/GBit ~ 30X

Bandwidth [MBytes/s]

0
1000
2000
3000
4000
5000

Message length [bytes]

10^0 10^2 10^4 10^6 10^8

Communication network

Memory
NI
P

MPI and Multicores

Clusters: Unidirectional intranode Ping-Pong bandwidth

Some BW scalability for multi-intranode connections

CS np2
CS np4
IS np2
IS np4

Bandwidth [MBytes/s]

Message length [bytes]

Cross-Socket (CS)

P P P P
C C C C
C C C C
C
MI

Memory

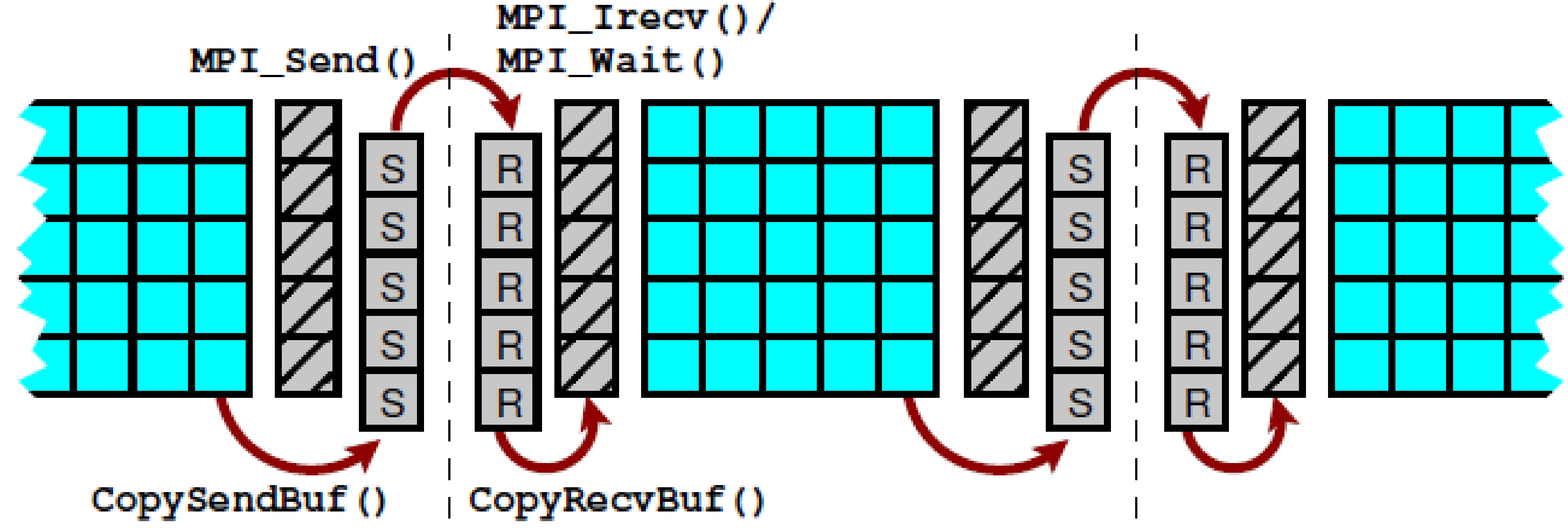
Intra-Socket (IS)

Single point-to-point BW similar to internode

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

Node comm surface
300
250
200
150
100
50
0
2 4 8 12 16 24 32 48 64
cores per node
Linear SD distribution
Optimal SD distribution
"Common" MPI library behavior
Woodcrest 2-socket
Nehalem EP 2-socket
Istanbul 2-socket
Shanghai 4-socket
Magny Cours 2-socket
Nehalem EX 4-socket
Magny Cours 4-socket
Sun Niagara 2

For more details see hybrid part!

Summary: Multicore performance properties

- **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 is ubiquitous**
  - 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

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**

H L R I S

Wavefront-parallel temporal blocking for stencil algorithms

One example for truly "multicore-aware" programming

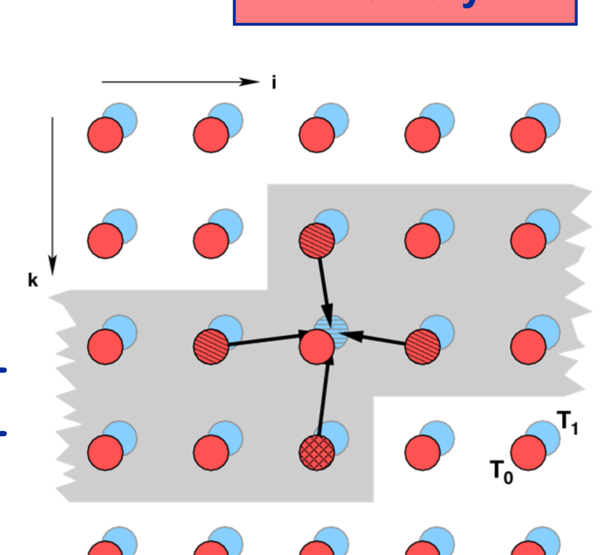
SKIPPED

P P P P P P
C C C C C C
C C C C C C
C
MI
Memory

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

Simple 3D Jacobi stencil update (sweep):

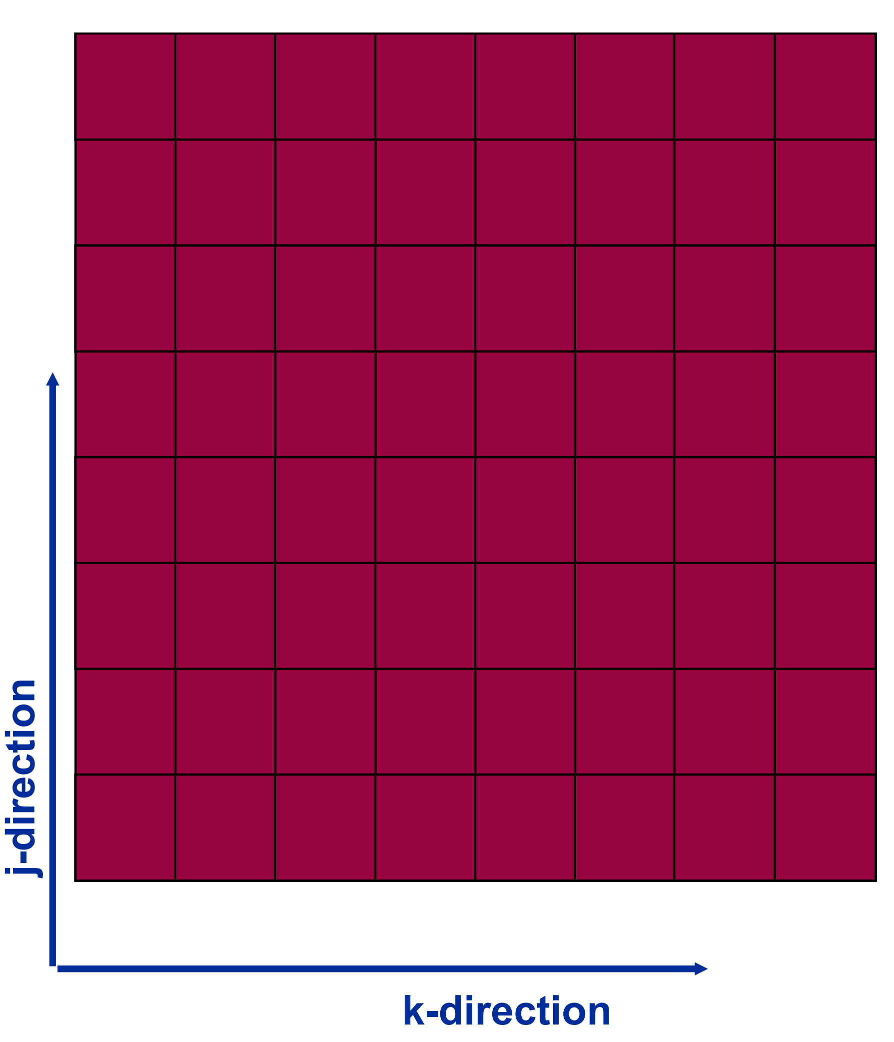
```
do k = 1 , Nk
  do j = 1 , Nj
       do i = 1 , Ni
          y(i,j,k) = a*x(i,j,k) + b*
                     (x(i-1,j,k)+x(i+1,j,k)+
                      x(i,j-1,k)+x(i,j+1,k)+
                      x(i,j,k-1)+x(i,j,k+1))
    enddo
  enddo
enddo
```

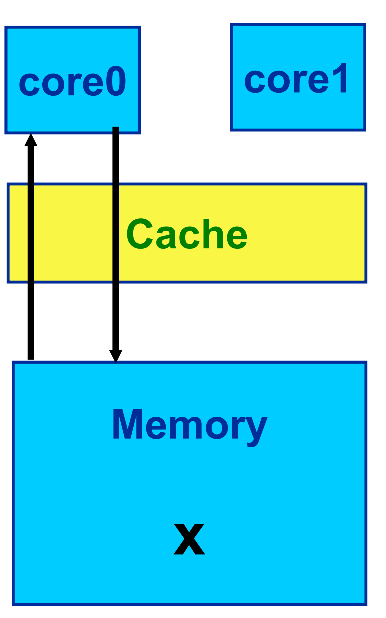


Performance Metric: Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs: 8 FLOP/LUP * MLUPs

Multicore awareness

Standard sequential implementation

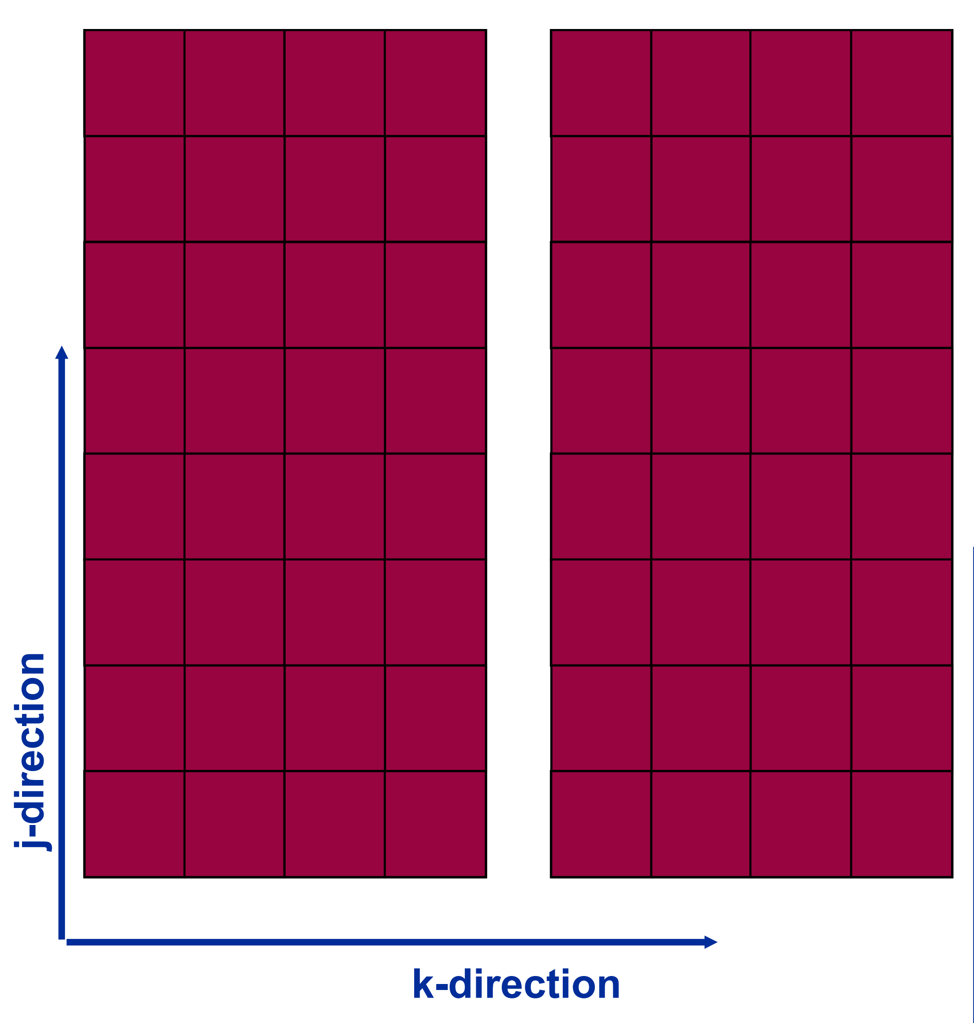


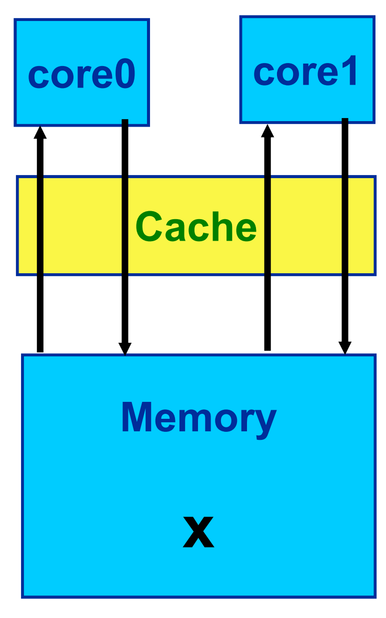



```
do t=1,tMax

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

enddo
```

Multicore awareness

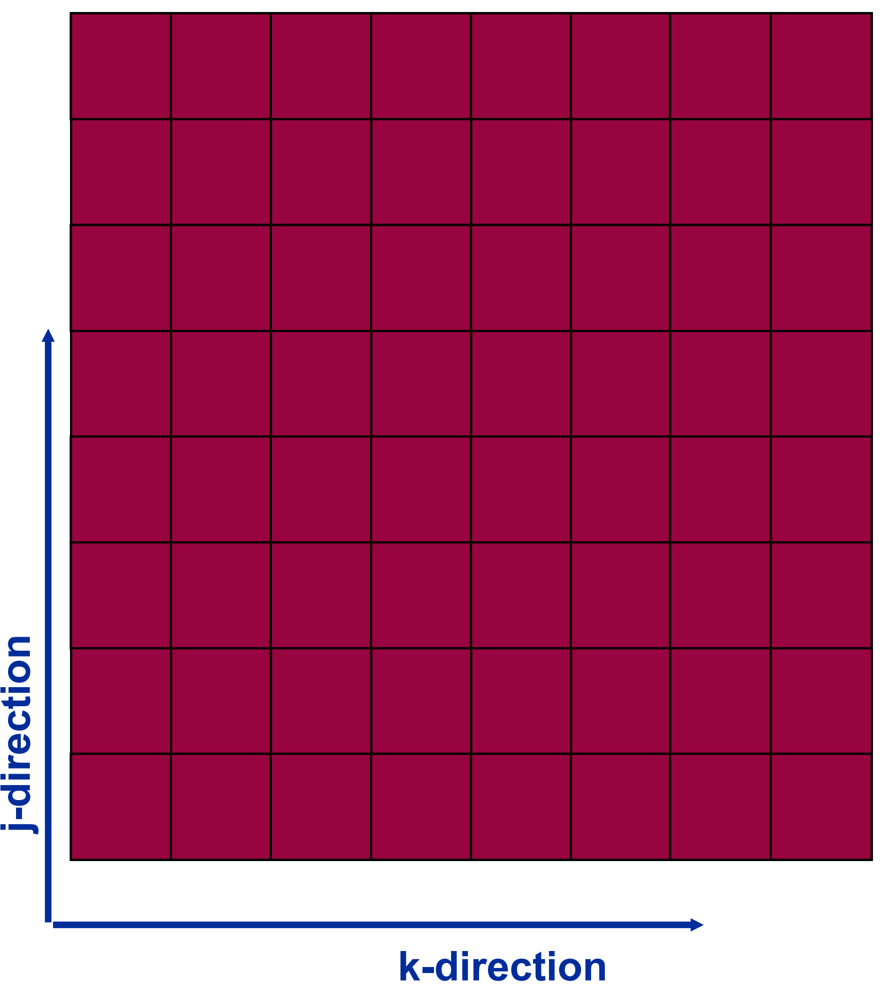
Classical Approaches: Parallelize!

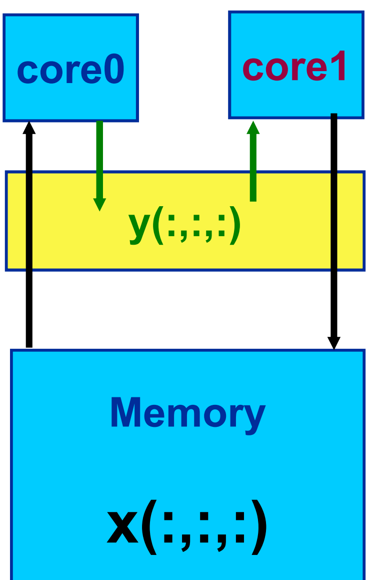





```
do t=1,tMax
!$OMP PARALLEL DO private(…)
  do k=1,N
     do j=1,N
        do i=1,N
           y(i,j,k) = …
        enddo
     enddo
  enddo
!$OMP END PARALLEL DO
enddo
```

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

↓

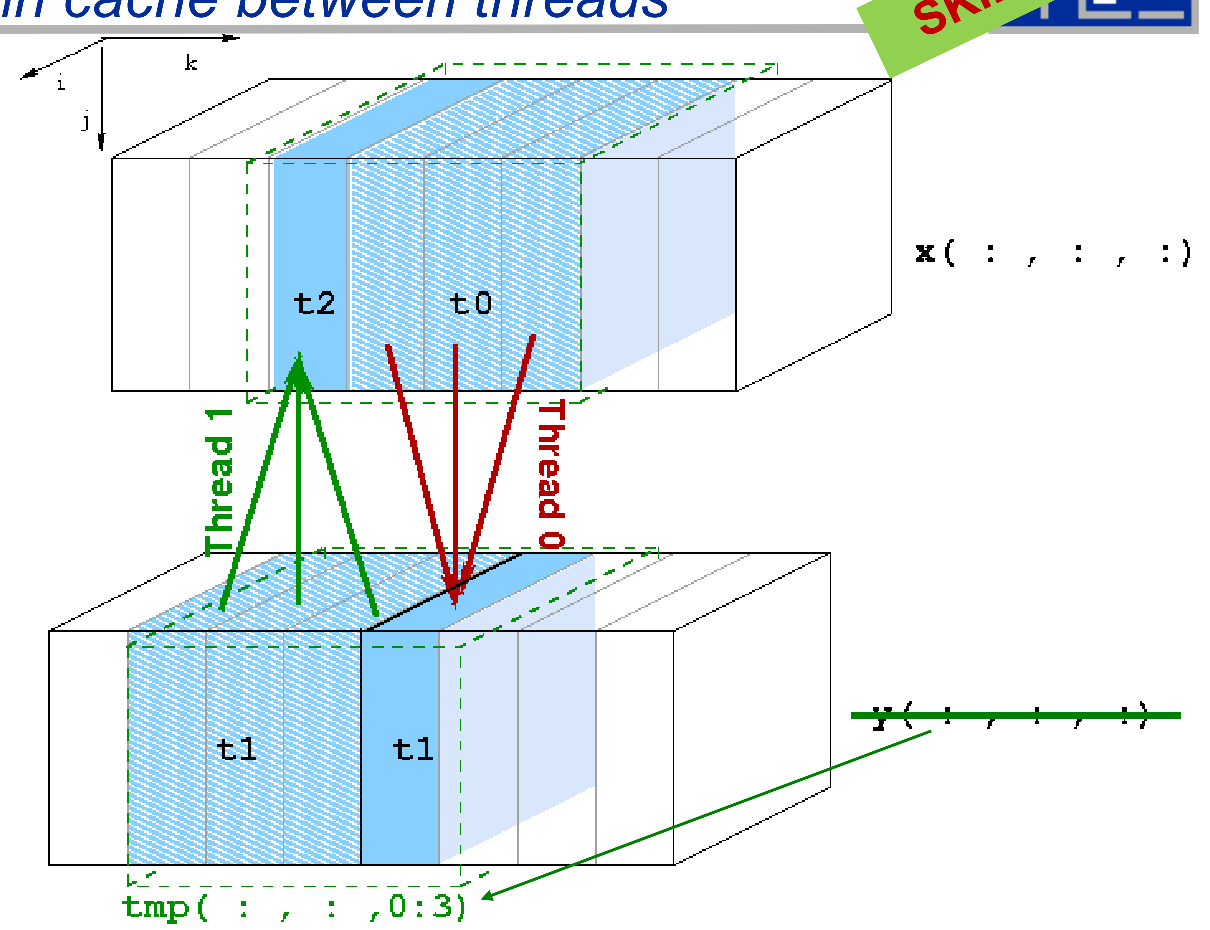
Save main memory data transfers for `y(:,:,:)` !

↓

16 Byte / 2 LUP !

↓

8 Byte / LUP !



SKIPPED

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)

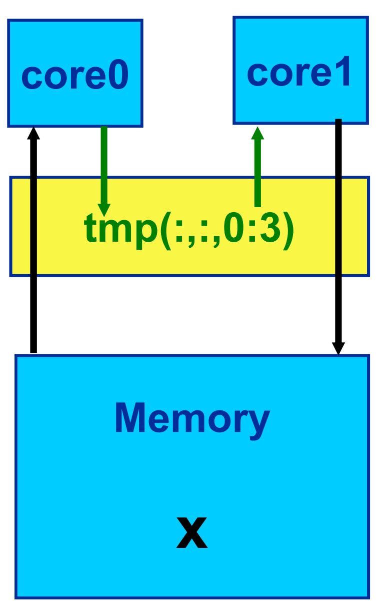
Thread 0: `x(:,:,k-1:k+1)`$_t$ → *`tmp(:,:,mod(k,4))`*

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:

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



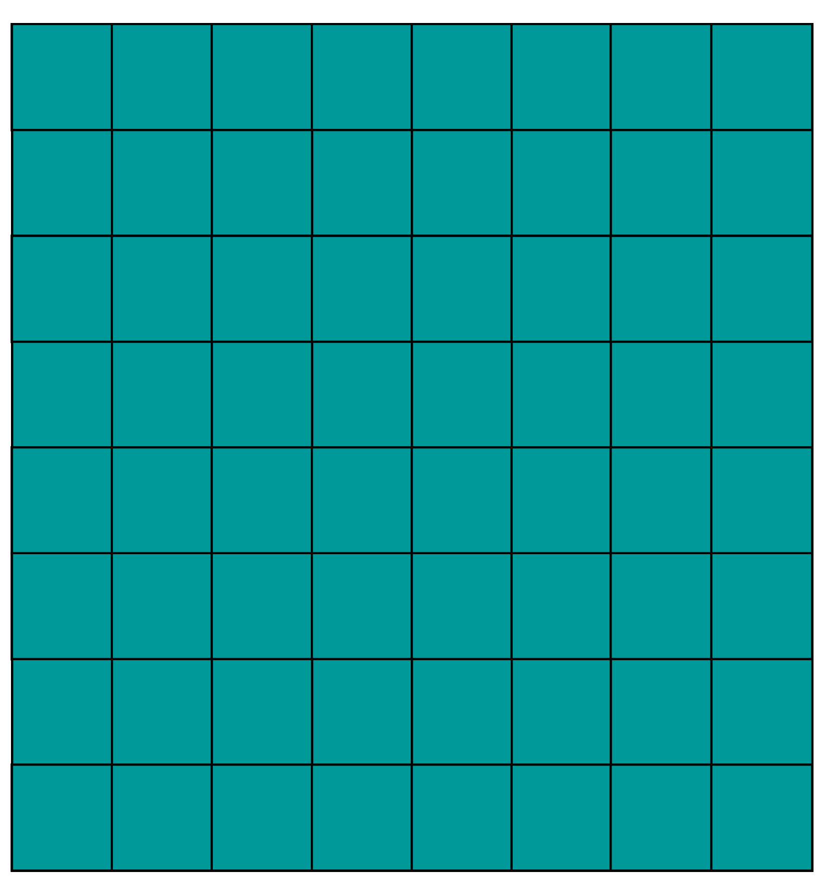
Minimum value: $x = 2$

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

B_C and B_M are measured in saturation runs:

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

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

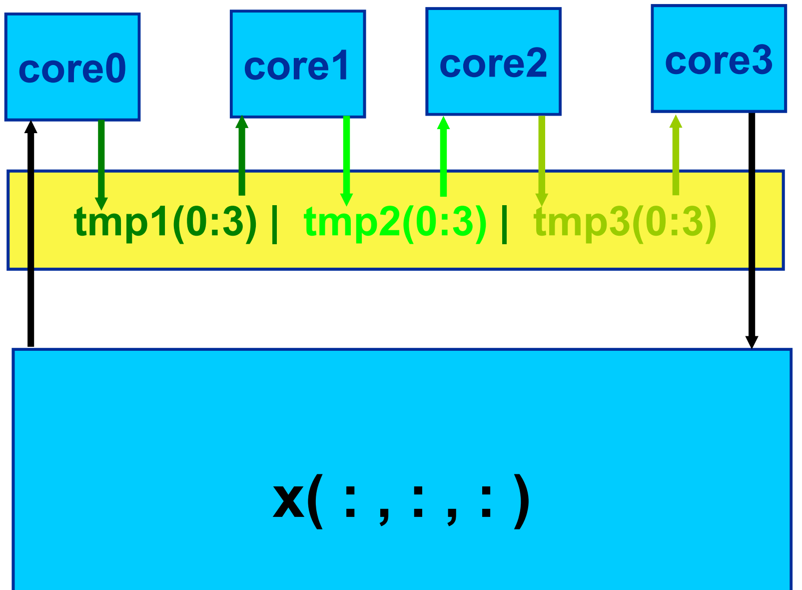


Running `tb` wavefronts requires `tb-1` temporary arrays `tmp` to be held in cache!

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

Extensive use of cache bandwidth!

1 x 4 distribution



Jacobi solver

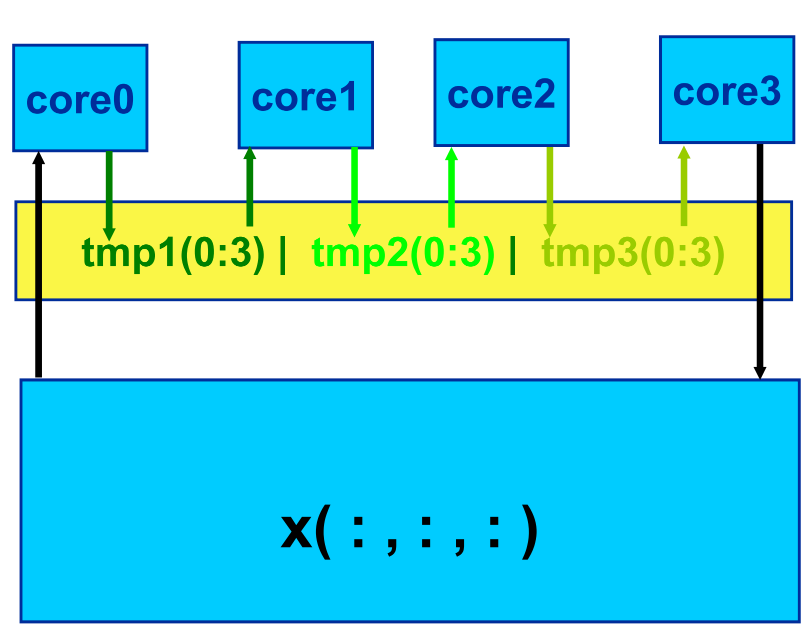
WF parallelization: New choices on native quad-cores

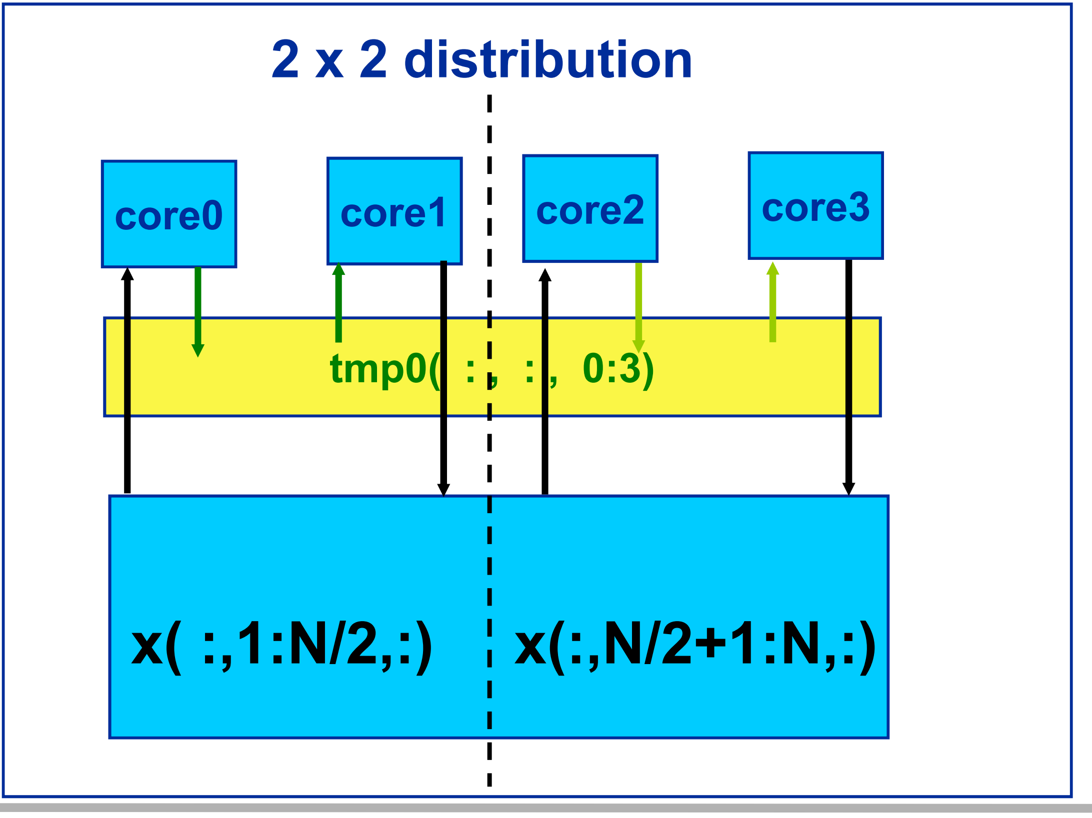
Thread 0: `x(:,:,k-1:k+1)`$_t$ → *`tmp1(mod(k,4))`*

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

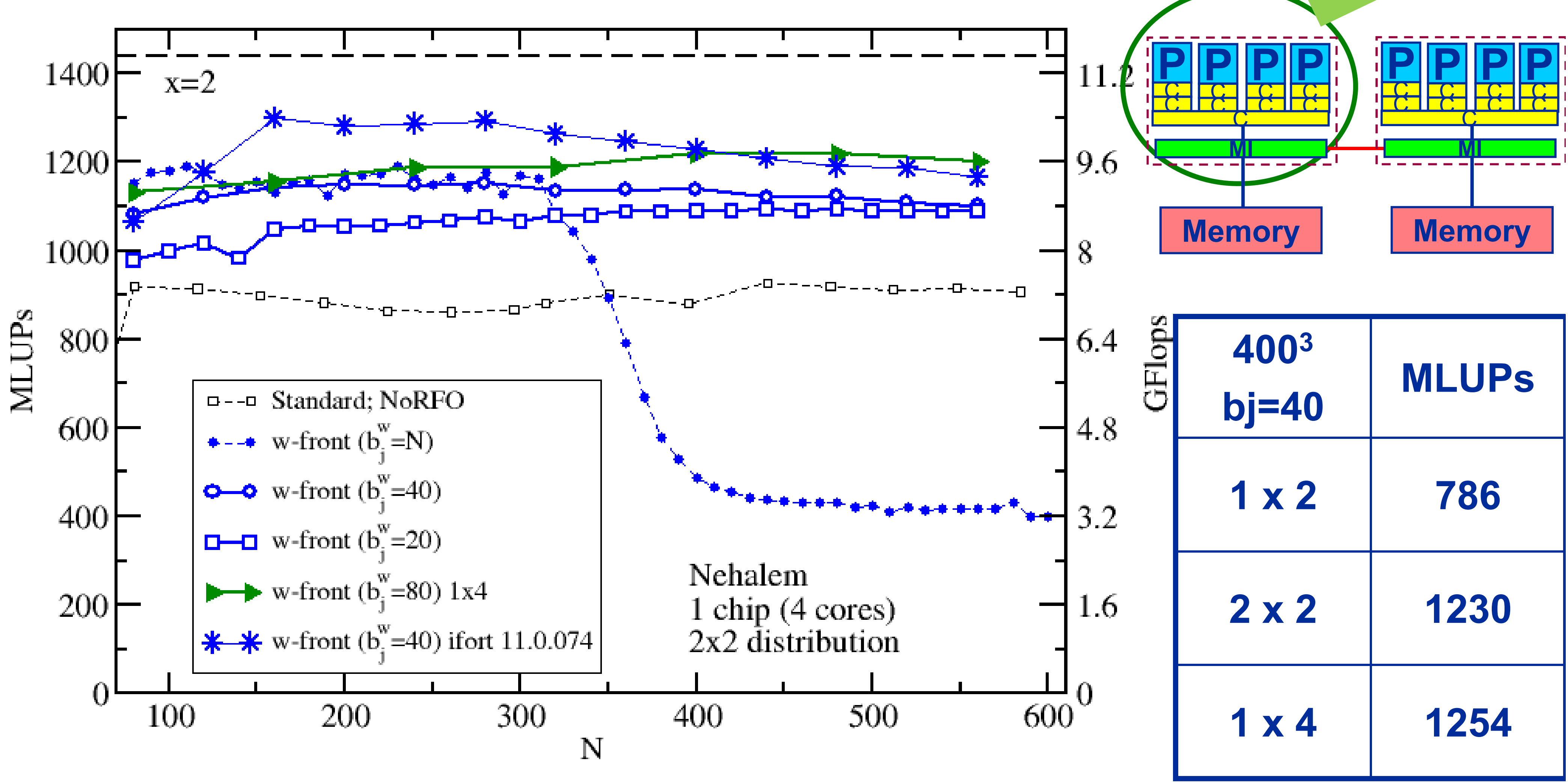
Thread 2: *`tmp2(mod(k-5,4:mod(k-3,4))`* → *`tmp3(mod(k-4,4))`*

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





Jacobi solver
Wavefront parallelization: L3 group Nehalem



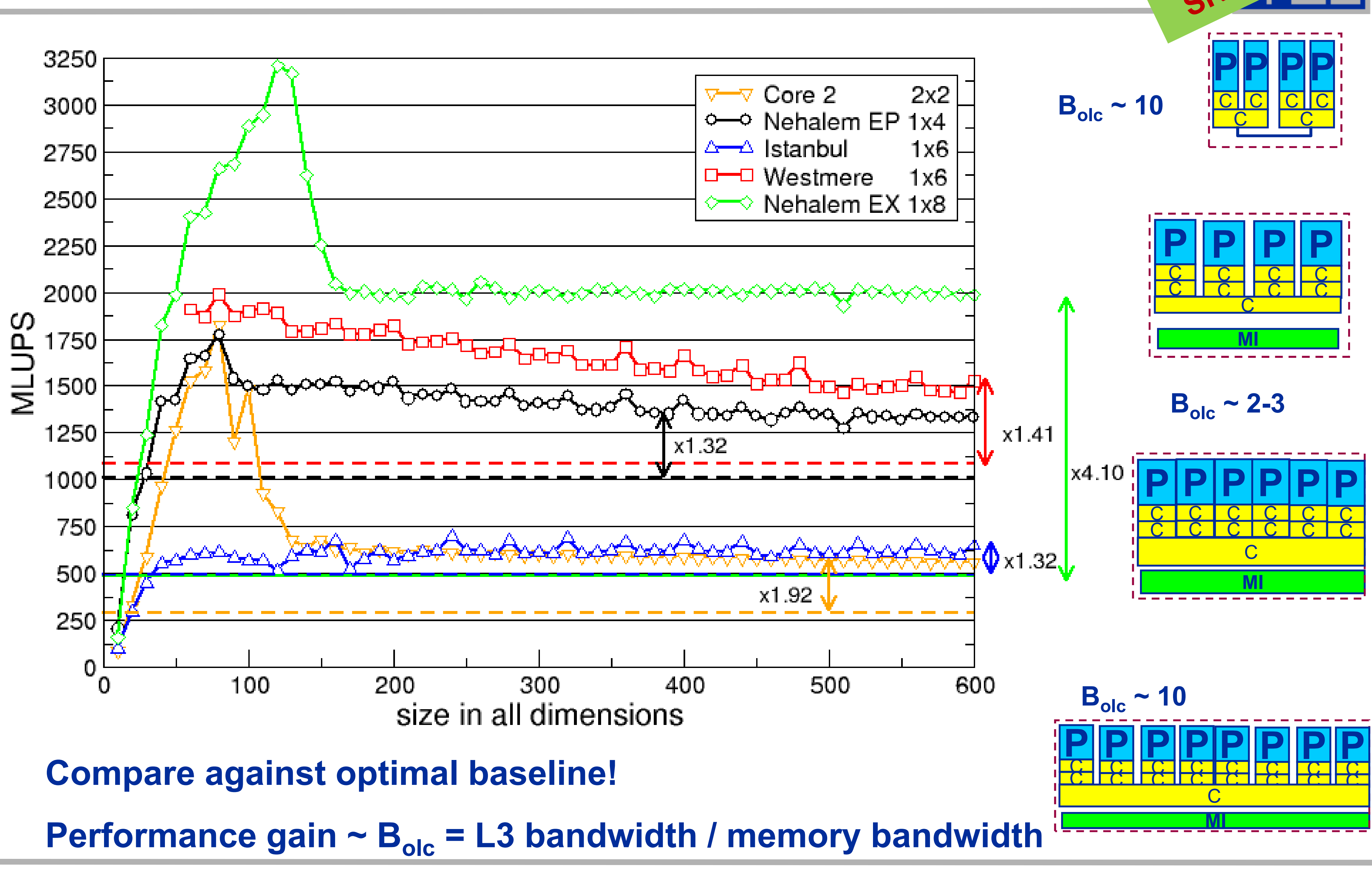
400^3 bj=40	MLUPs
1 x 2	786
2 x 2	1230
1 x 4	1254

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

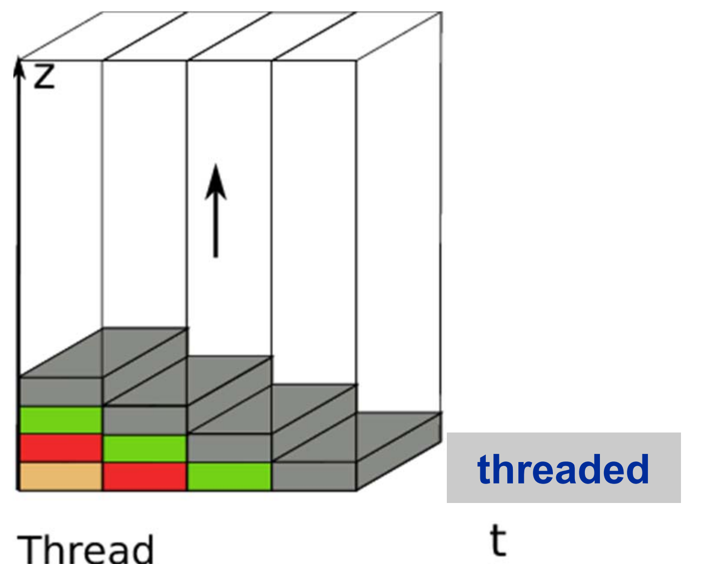
Wavefront – Jacobi on state-of-the art multicores

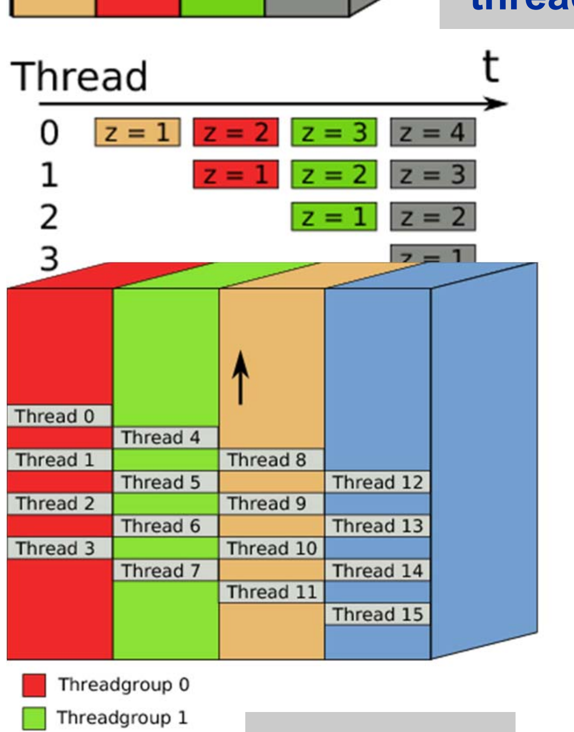


Compare against optimal baseline!

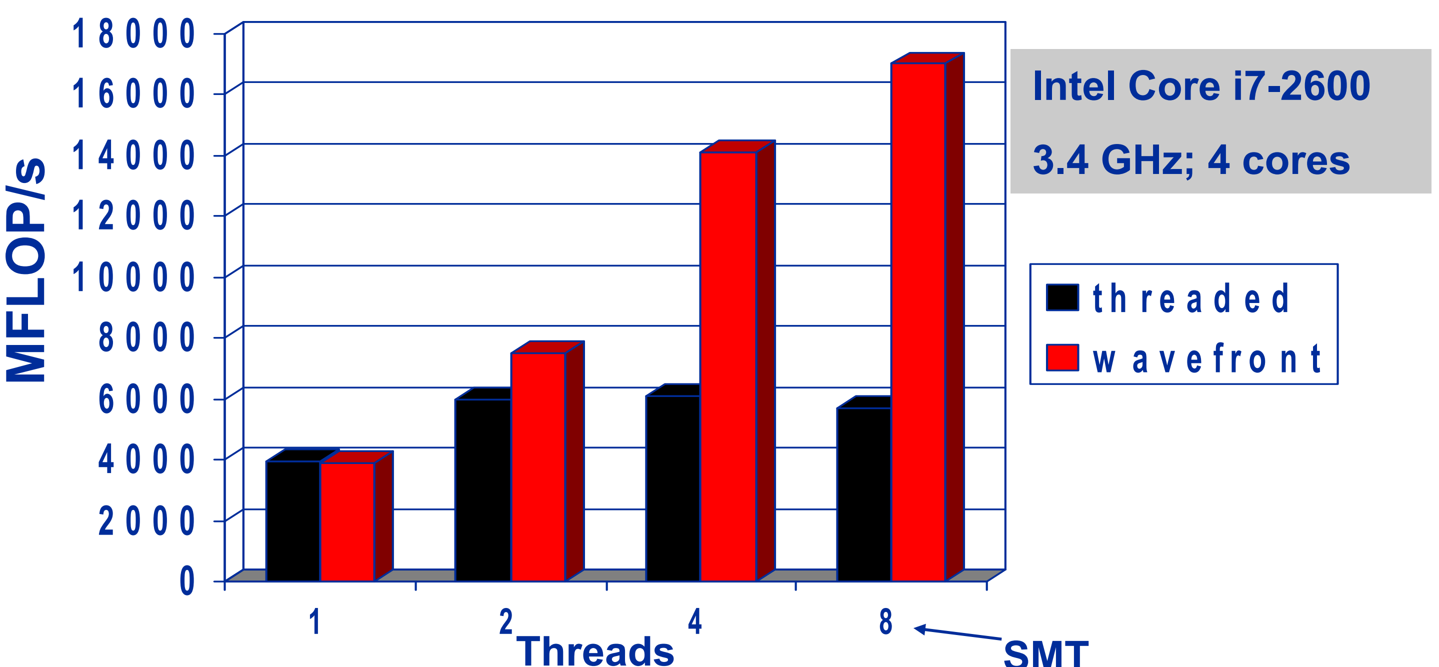
Performance gain ~ B_{olc} = L3 bandwidth / memory bandwidth

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





- **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")



Section summary: What to take home

- **Shared caches are *the* interesting new feature on current multicore chips**
  - Shared caches provide opportunities for fast synchronization (see sections on OpenMP and intra-node MPI performance)
  - Parallel software should leverage shared caches for performance
  - One approach: Shared cache reuse by WFP

- **WFP technique can easily be extended to many regular stencil based iterative methods, e.g.**
  - Gauß-Seidel (→ done)
  - Lattice-Boltzmann flow solvers (→ work in progress)
  - Multigrid-smoother (→ work in progress)

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**
  - topology
  - pin
  - Monitoring the binding
  - perfctr basics and best practices
- **Impact of processor/node topology on performance**
  - Microbenchmarking with simple parallel loops
  - Bandwidth saturation effects in cache and memory
  - Case study: OpenMP sparse MVM as an example for bandwidth-bound code
  - ccNUMA effects and how to circumvent performance penalties
  - Simultaneous multithreading (SMT)
- **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

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
- **Case studies for hybrid MPI/OpenMP**
  - Overlap of communication and computation for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - Hybrid computing with accelerators and compiler directives
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

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

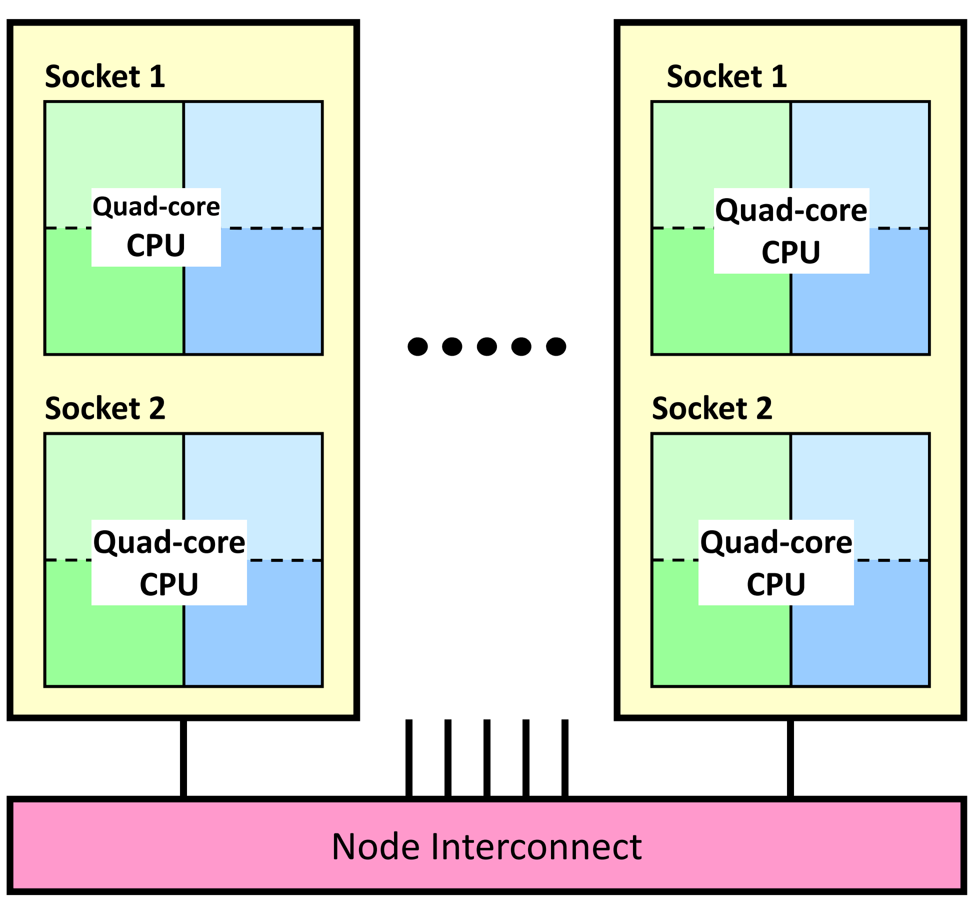
- **Case studies for hybrid MPI/OpenMP**
  - Overlap of communication and computation for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - Hybrid computing with accelerators and compiler directives

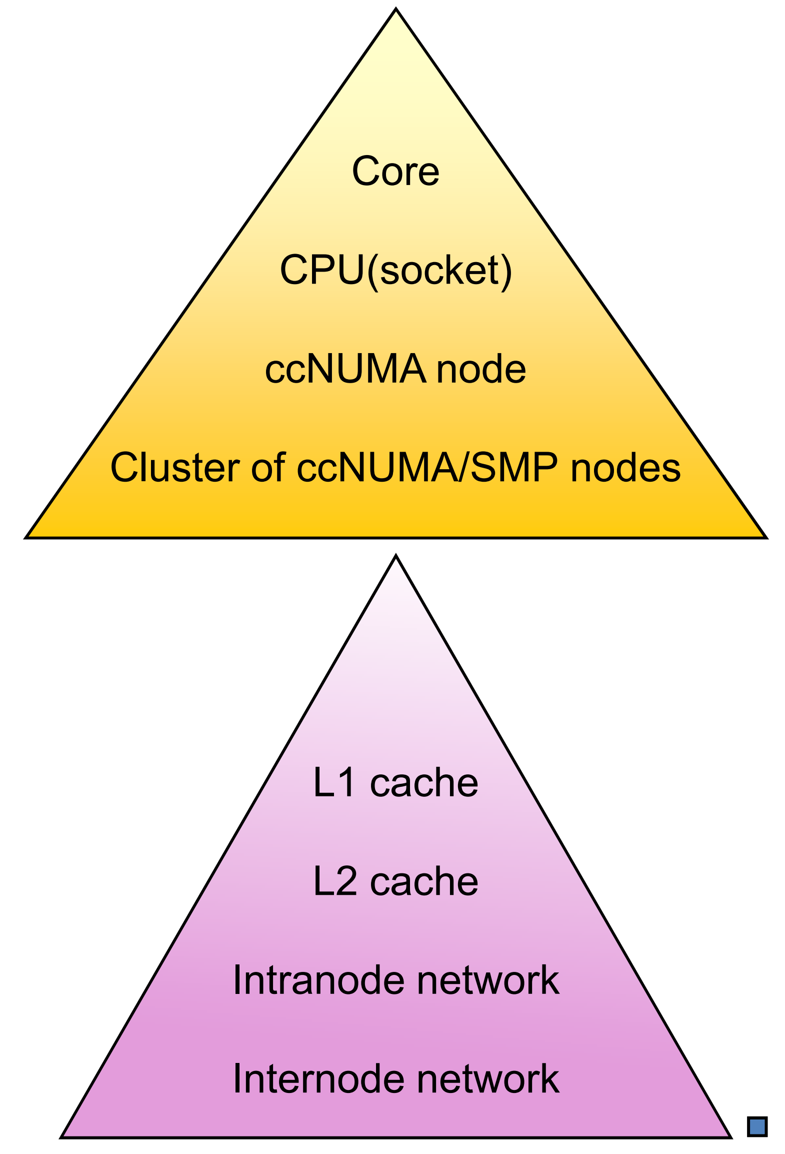
- **Summary: Opportunities and Pitfalls of Hybrid Programming**

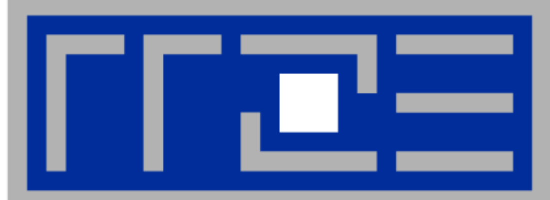
- **Overall summary and goodbye**

Clusters of Multicore Nodes

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



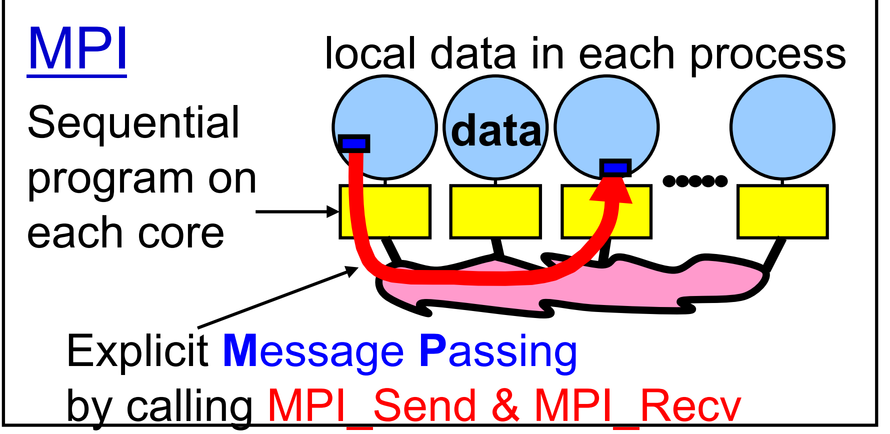


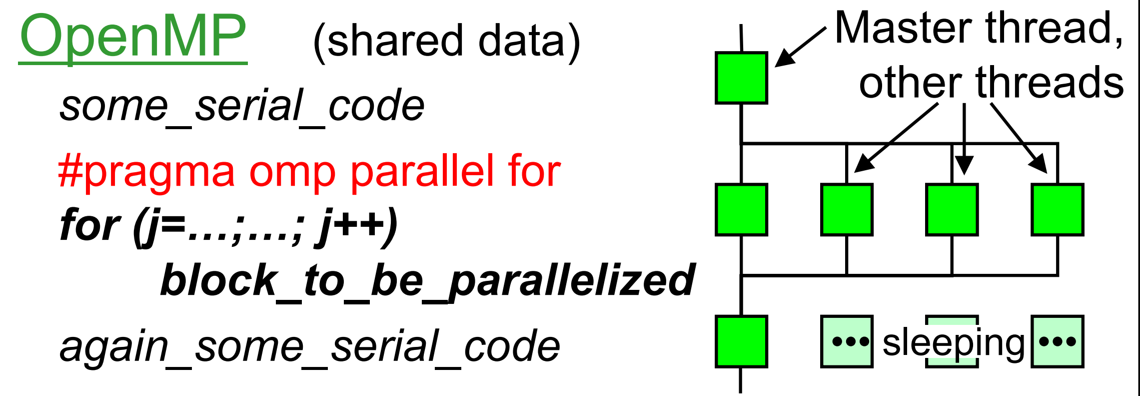


MPI vs. OpenMP

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?
  - Are there “safe bets” where it should really be faster?
  - Do you really understand what your code is doing???

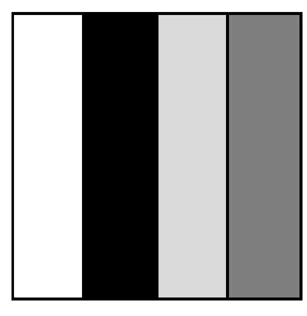




MPI Parallelization of Jacobi Solver

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

```
...
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)
```



1D partitioning

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 (→ `!$OMP END DO`)

Comparison of MPI and OpenMP

MPI

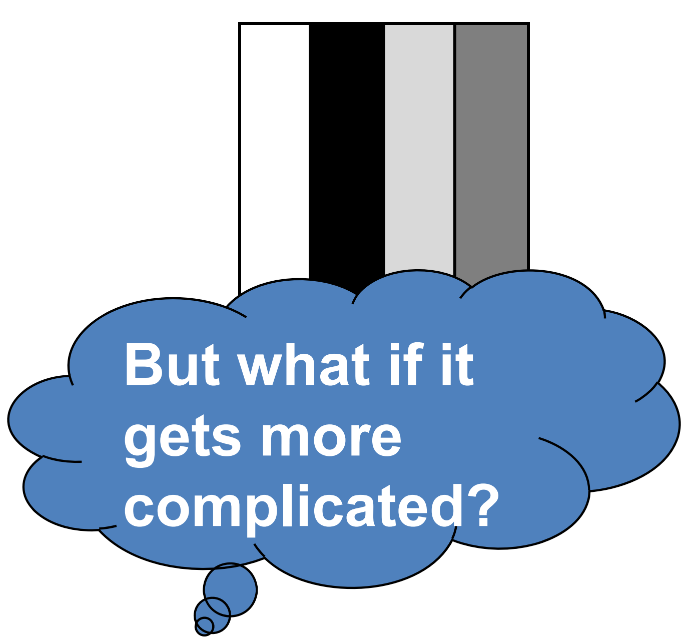
- **Memory Model**
  - Data private by default
  - Data accessed by multiple processes needs to be explicitly communicated
- **Program Execution**
  - Parallel execution starts with MPI_Init, continues until MPI_Finalize
- **Parallelization Approach**
  - Typicall coarse grained, based on domain decomposition
  - 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**



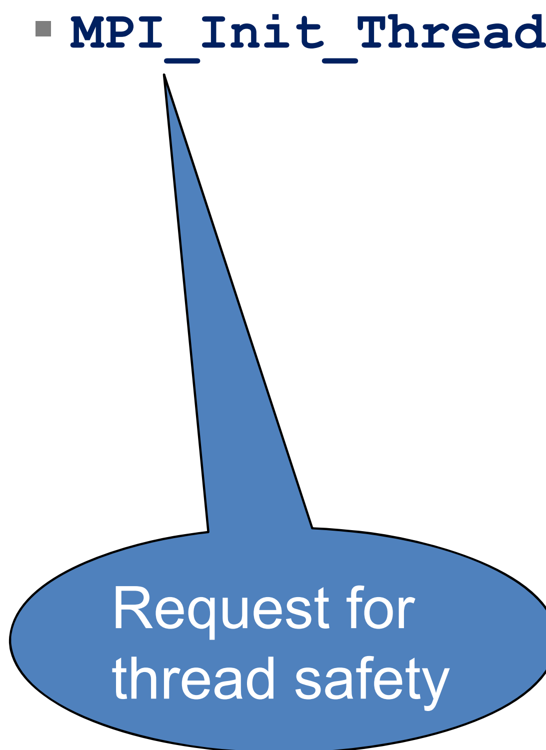

```
!Main Loop
DO WHILE(.NOT.converged)
   ! compute
   DO j=1, m_loc
!$OMP PARALLEL DO
      DO i=1, n
         BLOC(i,j)=0.25*(ALOC(i-1,j)+
                         ALOC(i+1,j)+
                         ALOC(i,j-1)+
                         ALOC(i,j+1))
      END DO
!$OMP END PARALLEL DO
   END DO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

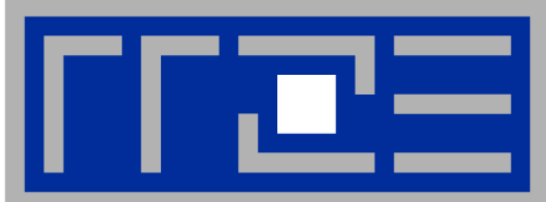
MPI

- **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, if possible, a higher level (stronger support).
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 parallel work>
   !$OMP barrier
   !$OMP master

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

!$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
{
   <OMP parallel work>
   #pragma omp barrier
   #pragma omp master
   {
     ierr=MPI_<whatever>(…);
   }
#pragma omp barrier

}
}
```

$OMP master does not have implicit barrier

Overlapping Communication and Work

- **Fortran**

```
include 'mpif.h'
program hybover

call mpi_init_thread(MPI_THREAD_FUNNELED,
                     ...)

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

}
}
```

Funneling through OMP SINGLE

- Fortran

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

    <OMP parallel work>
   !$OMP barrier
   !$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_SERIALIZED,...)
#pragma omp parallel
{
   <OMP parallel work>
   #pragma omp barrier
   #pragma omp single
   {
    ierr=MPI_<Whatever>(…)
   }

   //#pragma omp barrier
```

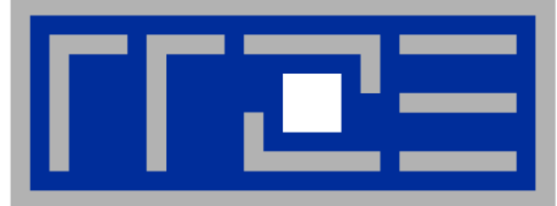
$OMP single has an implicit barrier

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

pure MPI
one MPI process
on each core

hybrid MPI+OpenMP
MPI: inter/intra-node communication
OpenMP: inside of each SMP node

OpenMP only
distributed virtual
shared memory

SINGLE

No overlap of Comm. + Comp.
MPI only outside of parallel regions
of the numerical application code

Masteronly
MPI only outside
of parallel regions

Overlapping Comm. + Comp.
MPI communication by one or a few threads
while other threads are computing

Funneled
MPI only
on master-thread

Multiple
more than one thread
may communicate

FUNNELED

Funneled &
Reserved
thread
for communication

Funneled
with
Full Load
Balancing

MULTIPLE

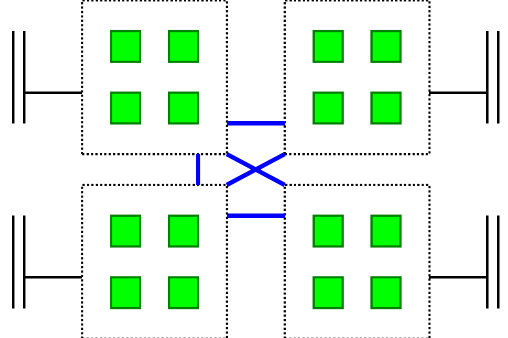
Multiple &
Reserved
threads for
communication

Multiple
with
Full Load
Balancing

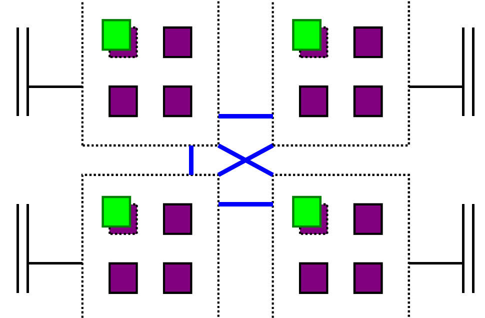
Modes of Hybrid Operation

Pure MPI …… Mixed ………. Fully Hybrid

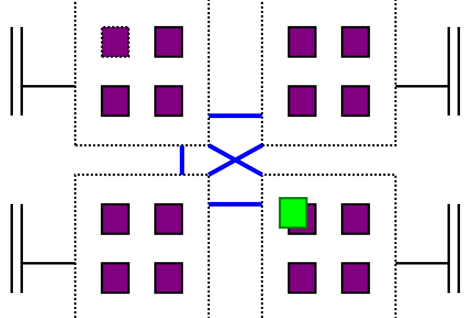
16 MPI processes
(i.e. 1 MPI process
per core)



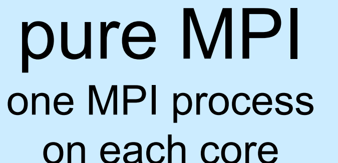
4 MPI processes
4 threads/process
(i.e. 1 MPI process
per NUMA domain)



1 MPI process
16 threads/process
(i.e. 1 MPI process
per ccNUMA node)

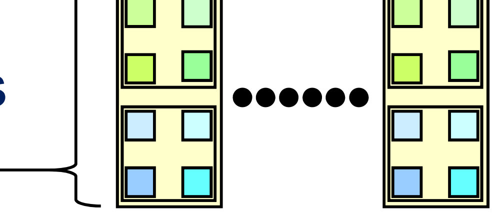


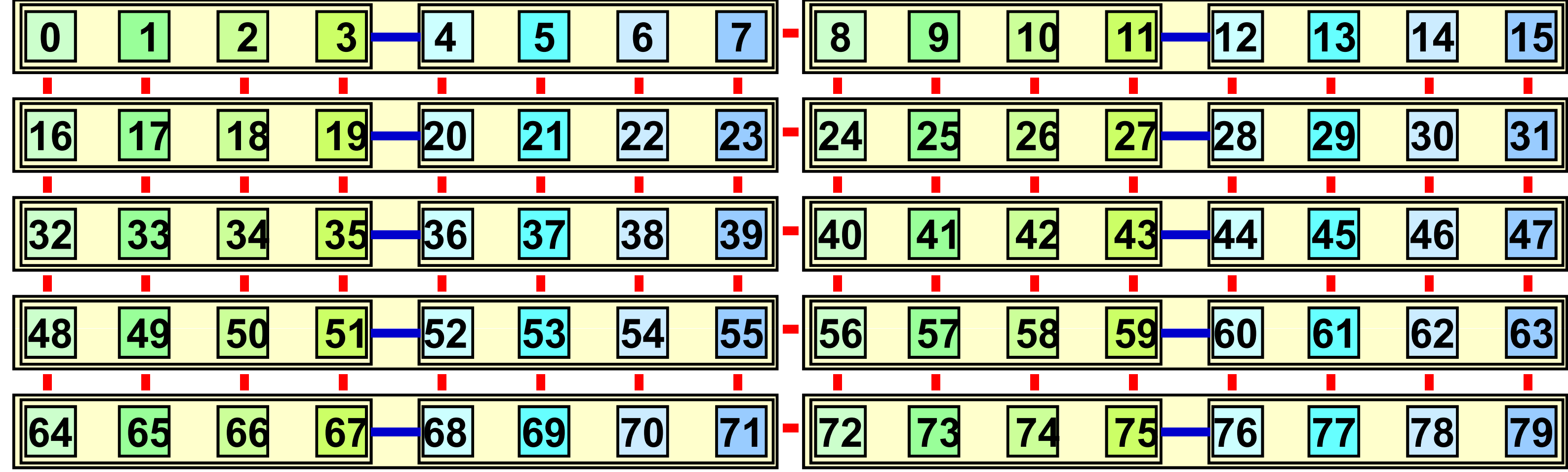
Master Thread of MPI Process
- MPI Process on Core
- Master Thread of MPI Process
- Slave Thread of MPI Process



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

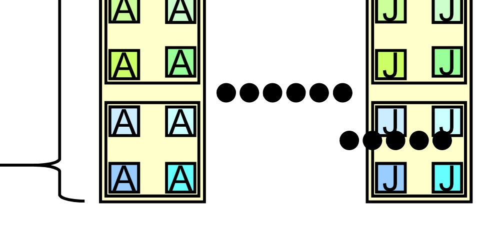
Does it matter?

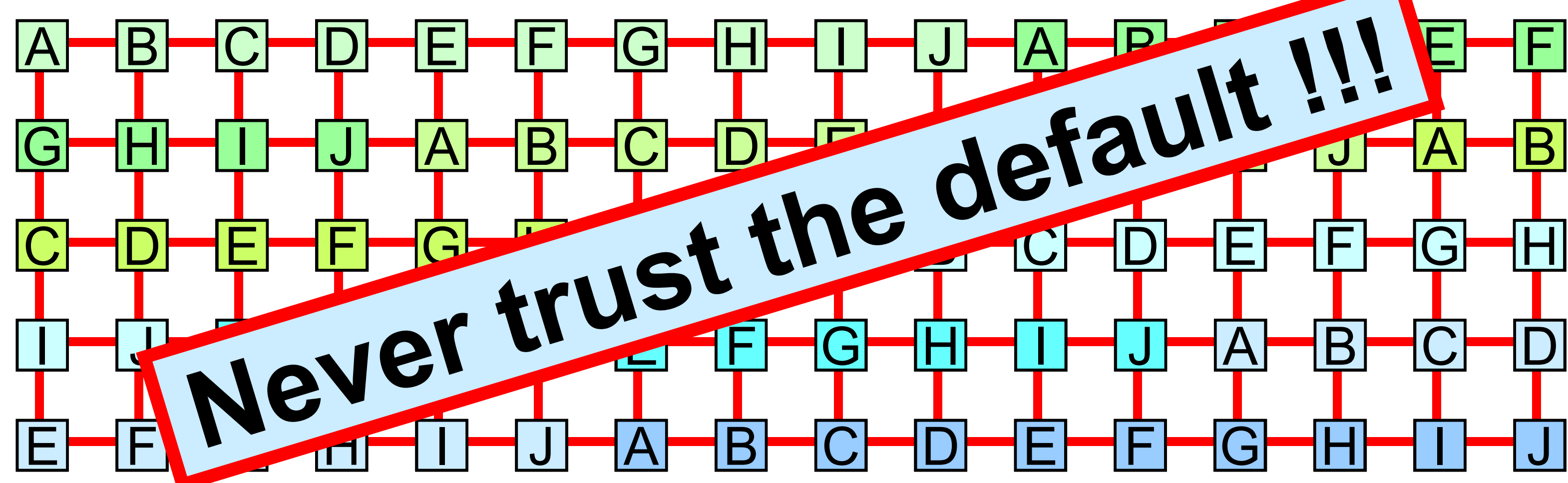
The Topology Problem with

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

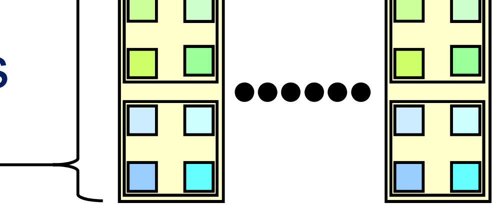
Round robin ranking of MPI_COMM_WORLD

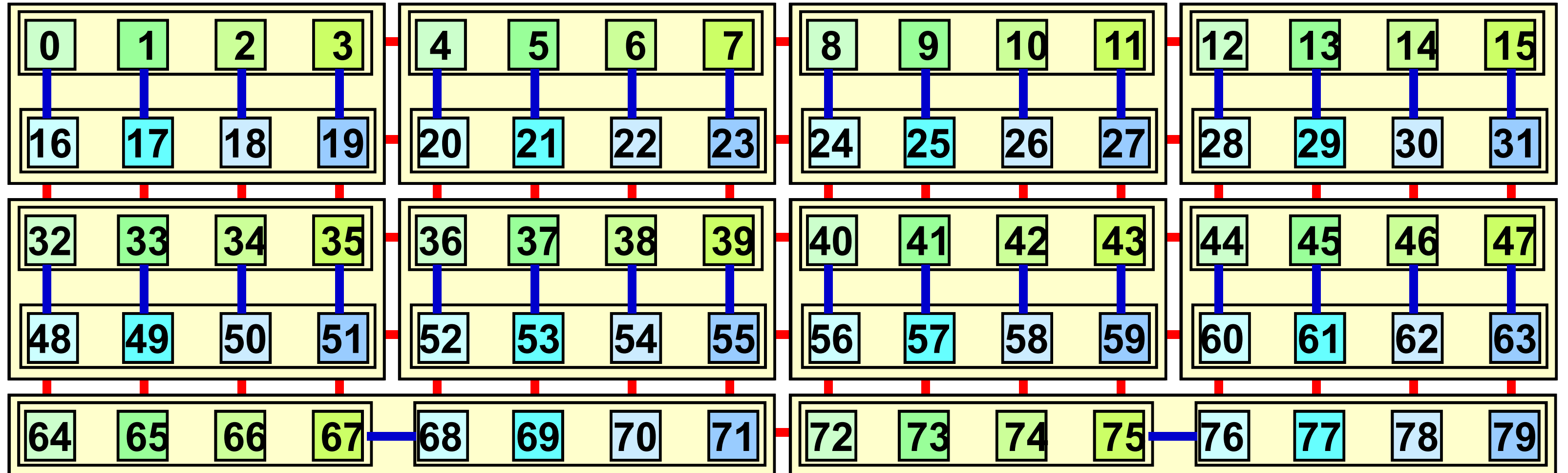
The Topology Problem with

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



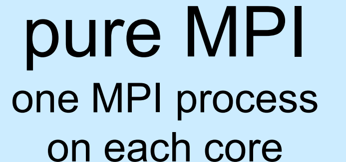


12 x inter-node connections per node

4 x inter-socket connection per node

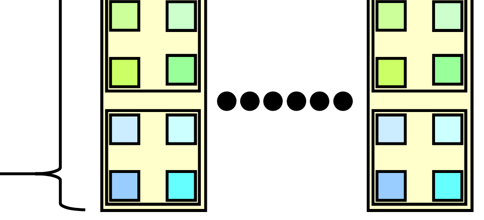
Two levels of domain decomposition

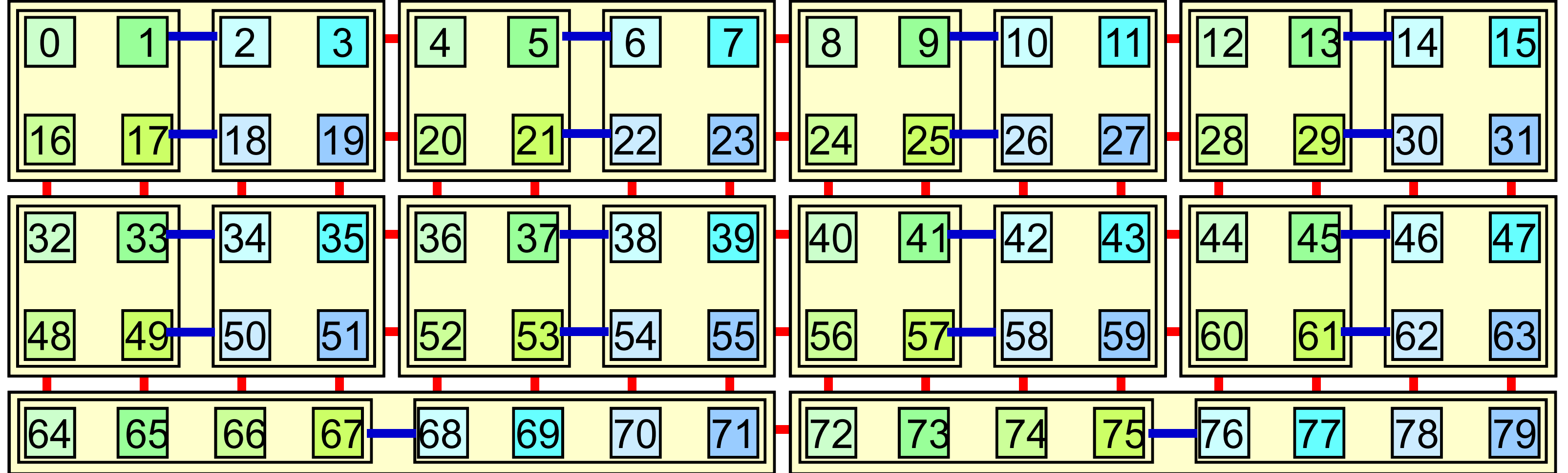
Bad affinity of cores to thread ranks



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

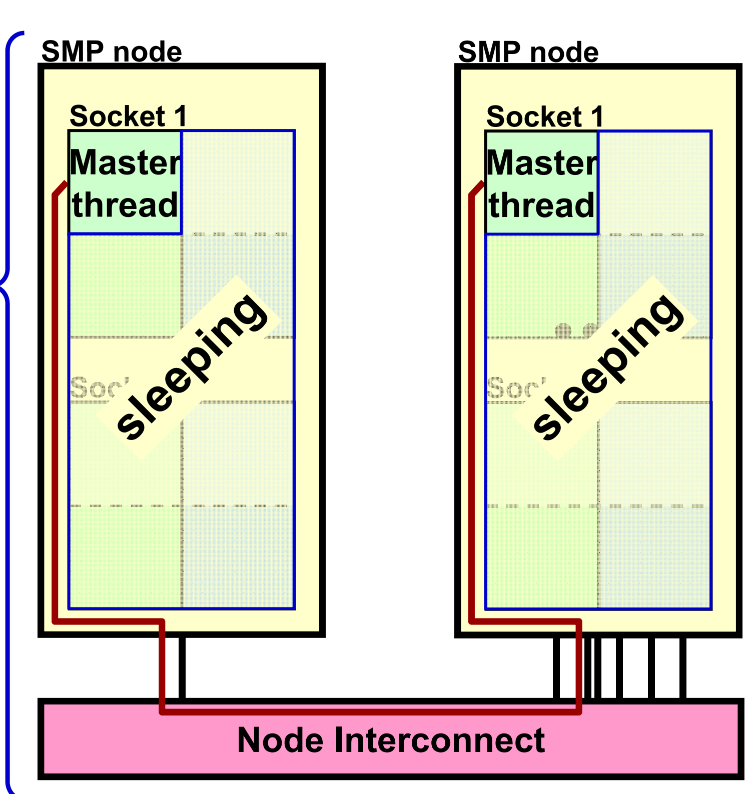
Two levels of domain decomposition

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**

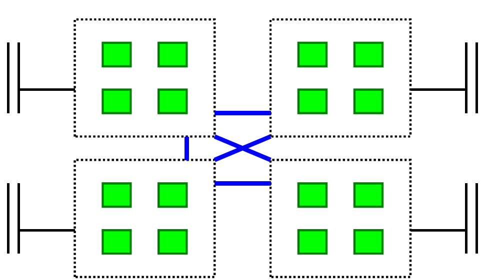
Solution:
- If funneling is supported use overlap of computation and communication

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

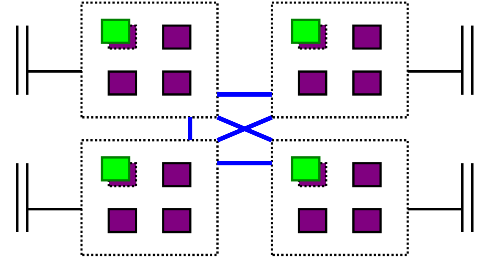
Pure MPI and Mixed Model

- **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

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



4 MPI Processes
4Threads/Process



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 Process
16Threads/Process

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

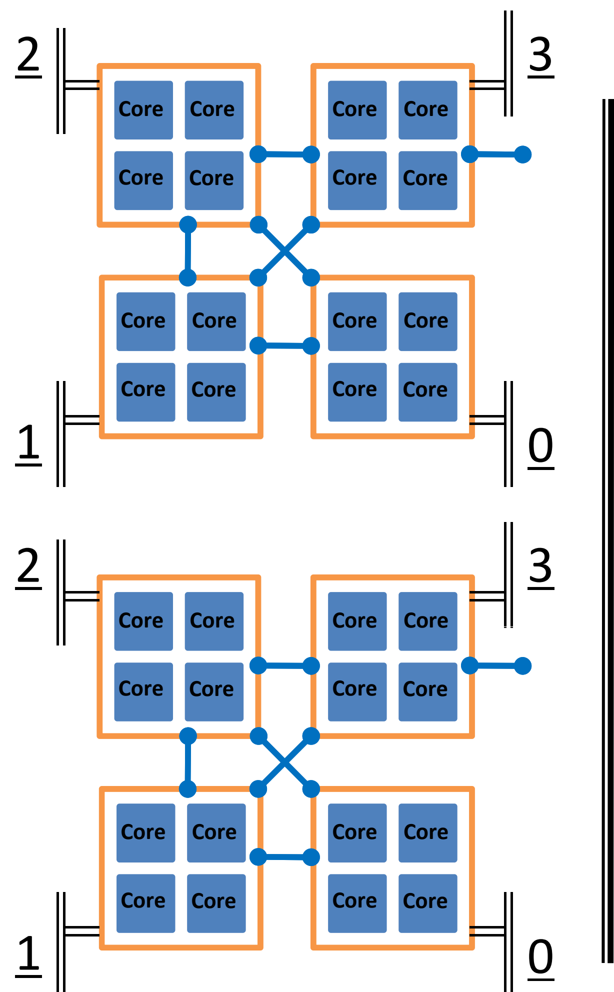
Other challenges for Hybrid Programming on multicore systems → see also first part of tutorial!

- **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 access across NUMA domain boundaries
  - 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)
  - Tools are available: taskset, numactl, LIKWID

Example for anisotropy effects:
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

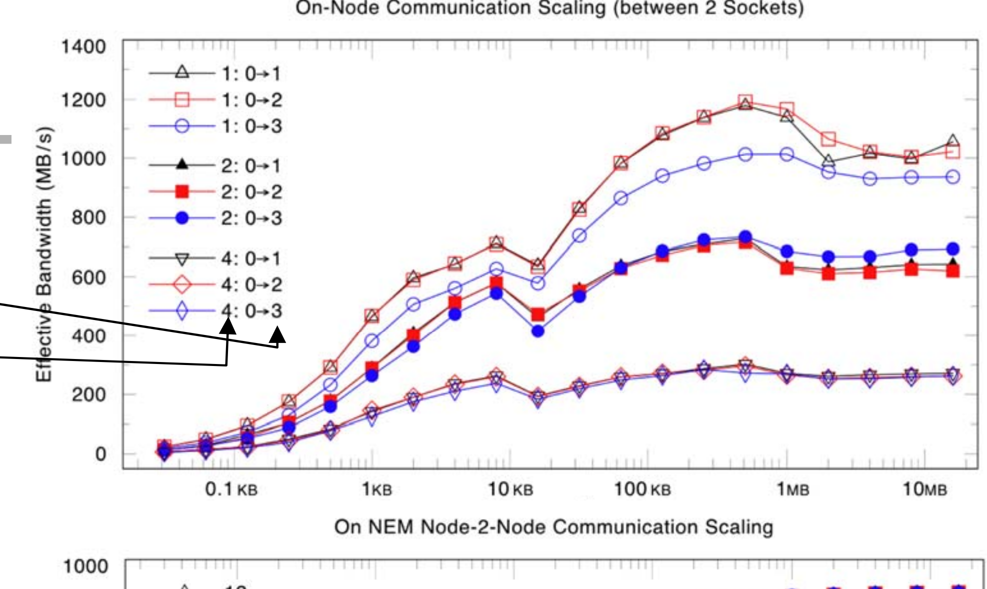
1 Socket has 2 connections to neighbors, 1 to network

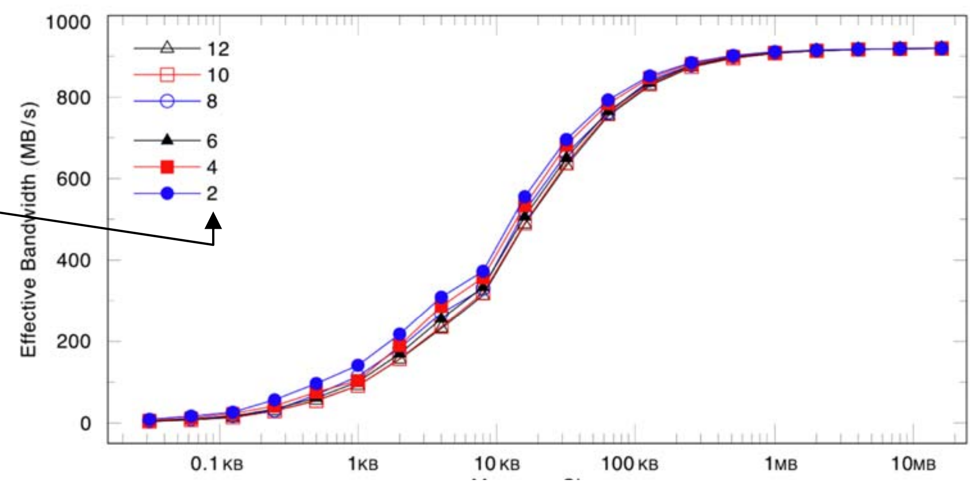
1 Socket has 2 connections to neighbors

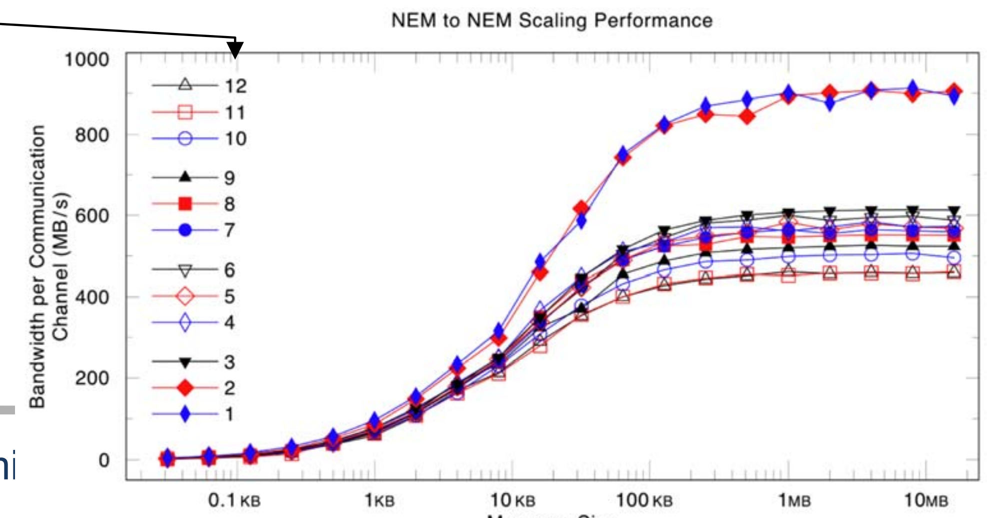
MPI ping-pong microbenchmark results on Ranger

- **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 communications
  - ➢ Max. bandwidth : 6 x 900 MB/s
- **Chassis to chassis (distance: 7 hops) with 1 MPI process per node and 1-12 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

Overlapping Communication and Computation

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

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**

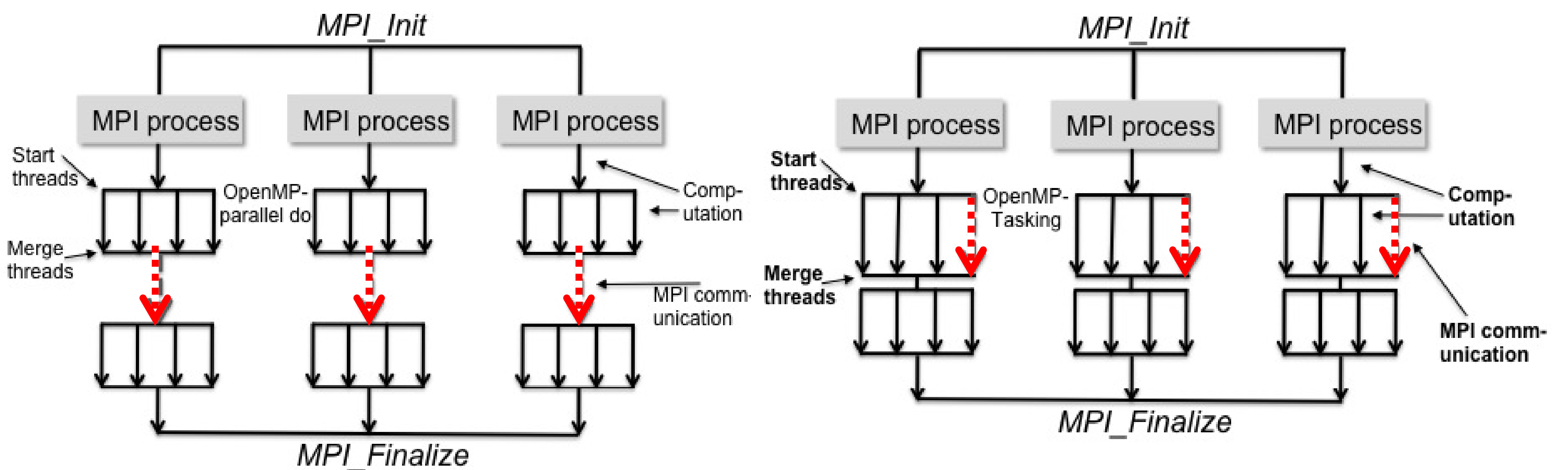
```
if (my_thread_rank < 1) {
  MPI_Send/Recv….
} else {
  my_range = (high-low-1)/(num_threads-1)+1;
  my_low = low + (my_thread_rank+1)*my_range;
  my_high=high+ (my_thread_rank+1+1)*my_range;
  my_high = max(high, my_high)
  for (i=my_low; i<my_high; i++) {
        ...
  }
}
```

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

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

```
do iterations=1,N                                          1
!compute particles to be shifted
  !$omp parallel do                                        3
  shift_p=particles_to_shift(p_array);
                                                           5
!communicate amount of shifted
! particles and return if equal to 0                       7
  shift_p=x+y
  MPI_ALLREDUCE(shift_p,sum_shift_p);                      9
  if(sum_shift_p==0) { return; }
                                                           11
!pack particle to move right and left
  !$omp parallel do                                        13
  do m=1,x
    sendright(m)=p_array(f(m));                            15
  enddo
  !$omp parallel do                                        17
  do n=1,y
    sendleft(n)=p_array(f(n));                             19
  enddo
                                                           21
!reorder remaining particles: fill holes
  fill_hole(p_array);                                      23

!send number of particles to move right                    25
  MPI_SENDRECV(x,length=2,..);
!send to right and receive from left                       27
  MPI_SENDRECV(sendright,length=g(x),..);
!send number of particles to move left                     29
  MPI_SENDRECV(y,length=2,..);
!send to left and receive from right                       31
  MPI_SENDRECV(sendleft,length=g(y),..);
                                                           33
!adding shifted particles from right
  !$omp parallel do                                        35
  do m=1,x
    p_array(h(m))=sendright(m);                            37
  enddo
!adding shifted particles from left                        39
  !$omp parallel do
  do n=1,y                                                 41
    p_array(h(n))=sendleft(n);
  enddo                                                    43
}
```

INDEPENDENT (lines 5–10 and 11–20)

INDEPENDENT (lines 22–23 and 24–33)

SEMI-INDEPENDENT (lines 24–33 and 34–43)

GTS shift routine

Work on particle array (packing for sending, reordering, adding after sending) can be overlapped with **data independent** MPI communication using **OpenMP tasks.**

Slides courtesy of Alice Koniges, NERSC, LBNL

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

```
integer stride=1000
!$omp parallel                                  2
!$omp master
!pack particle to move right                    4
  do m=1,x-stride ,stride
    !$omp task                                  6
    do mm=0,stride -1,1
      sendright(m+mm)=p_array(f(m+mm));         8
    enddo
    !$omp end task                              10
  enddo
  !$omp task                                    12
  do m=m,x
    sendright(m)=p_array(f(m));                 14
  enddo
  !$omp end task                                16
```

```
!pack particle to move left
  do n=1,y-stride ,stride                       18
    !$omp task
    do nn=0,stride -1,1                         20
      sendleft(n+nn)=p_array(f(n+nn));
    enddo                                       22
    !$omp end task
  enddo                                         24
  !$omp task
  do n=n,y                                      26
    sendleft(n)=p_array(f(n));
  enddo                                         28
  !$omp end task
  MPI_ALLREDUCE(shift_p ,sum_shift_p);          30
!$omp end master
!$omp end parallel                              32
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

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

```
!$omp parallel                                  1
!$omp master
  !$omp task                                    3
  fill_hole(p_array);
  !$omp end task                                5

  MPI_SENDRECV(x,length=2,..);                  7
  MPI_SENDRECV(sendright,length=g(x),..);
  MPI_SENDRECV(y,length=2,..);                  9
!$omp end master
!$omp end parallel                              11
}
```

Particle reordering of remaining particles (above) and adding sent particles into array (right) & sending or receiving of shifted particles can be independently executed.

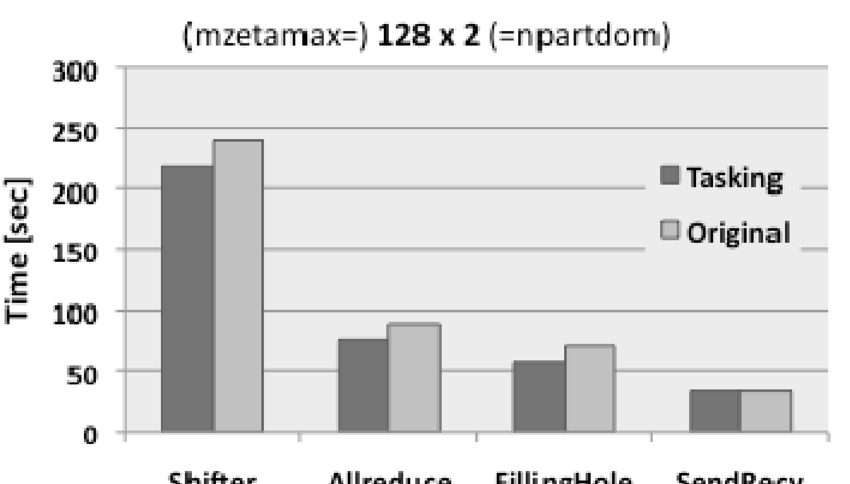
```
!$omp parallel
!$omp master                                    2
!adding shifted particles from right
  do m=1,x-stride,stride                        4
    !$omp task
    do mm=0,stride-1,1                          6
      p_array(h(m))=sendright(m);
    enddo                                       8
    !$omp end task
  enddo                                         10
  !$omp task
  do m=m,x                                      12
    p_array(h(m))=sendright(m);
  enddo                                         14
  !$omp end task
                                                16
  MPI_SENDRECV(sendleft,length=g(y),..);
!$omp end master                                18
!$omp end parallel
                                                20
!adding shifted particles from left
!$omp parallel do                               22
do n=1,y
  p_array(h(n))=sendleft(n);                    24
enddo
```

Overlapping remaining MPI_Sendrecv

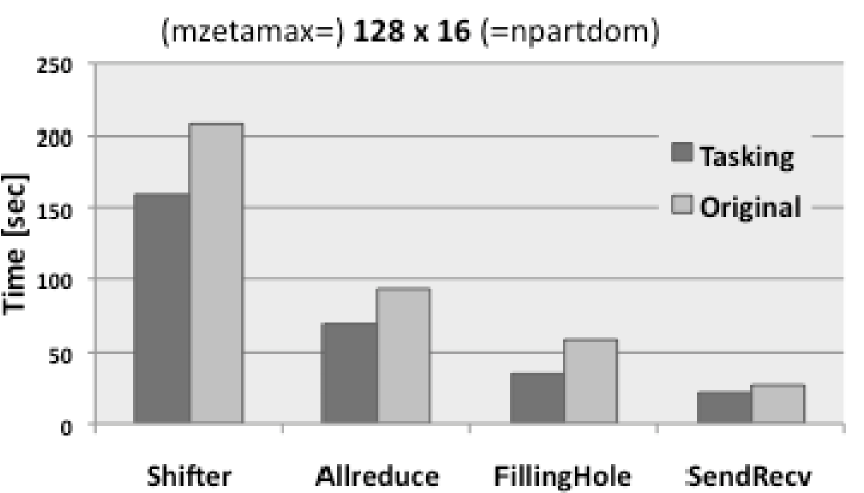
Slides, courtesy of Alice Koniges, NERSC, LBNL

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

256 size run



2048 size run



- **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

See case studies!

… maybe one of the major reasons for using hybrid MPI/OpenMP

Case study: MPI+OpenMP memory usage of NPB

Relative Memory Usage to MPI

1.2
1.0
0.8
0.6
0.4
0.2
0.0

BT-MZ
SP-MZ

256*1
128*2
64*4
32*8

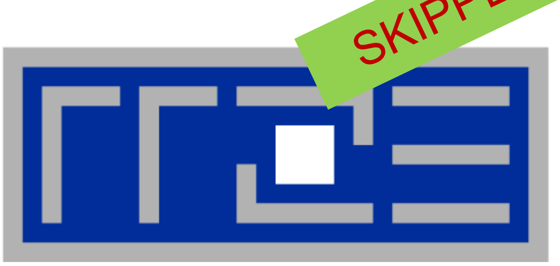
MPI * OpenMP

Always same number of cores

Using more OpenMP threads could reduce the memory usage **substantially**, up to **five** times on Hopper Cray XT5 (eight-core nodes).

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

Slide, courtesy of
Alice Koniges, NERSC, LBLN



Practical "How-To" 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
```

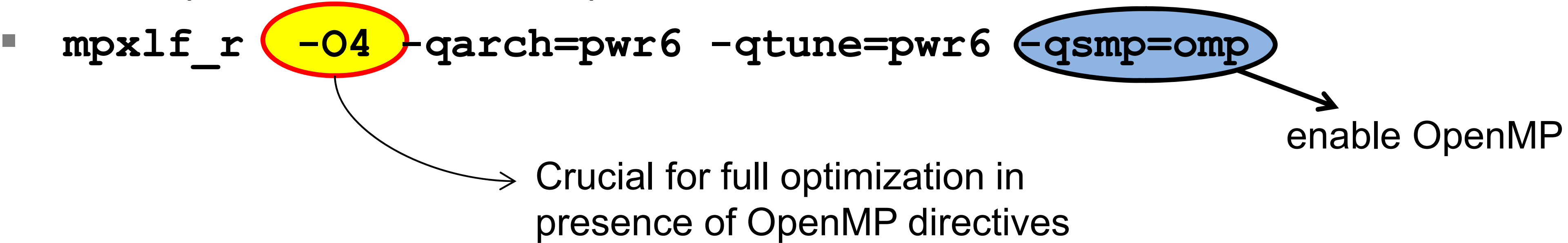
8 MPI Procs with 1 thread
4 MPI Procs with 2 threads
2 MPI Procs with 4 threads
2 MPI Procs with 8 threads

```
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`




```
#!/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

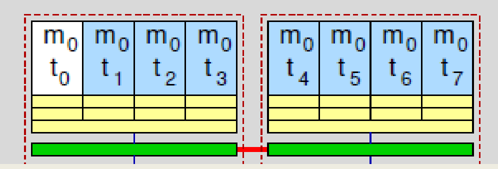
```
#!/bash
#PBS -q standard
#PBS -l select=16:ncpus=4
#PBS -l walltime=8:00:00
#PBS -j oe
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=2
mpirun -np 32 -bynode ./bt-mz.C.32
```

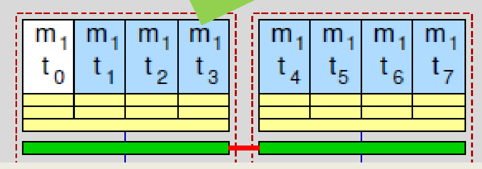
Processes placed round-robin on nodes

Topology choices with MPI/OpenMP:
More examples using Intel MPI+compiler & home-grown mpirun (@RRZE)

SKIPPED

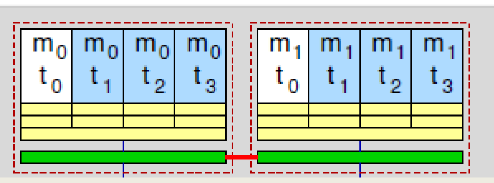
One MPI process per node

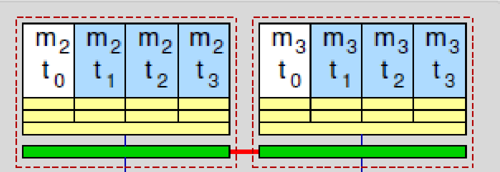





```
env OMP_NUM_THREADS=8 mpirun -pernode \
          likwid-pin -t intel -c N:0-7 ./a.out
```

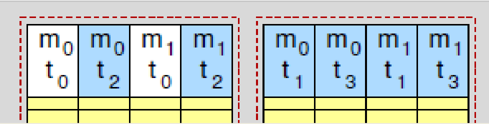
One MPI process per socket

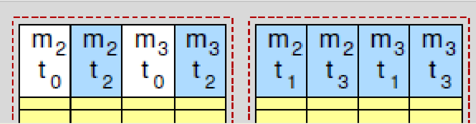





```
env OMP_NUM_THREADS=4 mpirun -npernode 2 \
        -pin "0,1,2,3_4,5,6,7" ./a.out
```

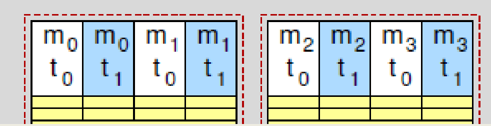
OpenMP threads pinned "round robin" across cores in node

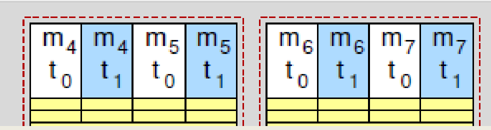





```
env OMP_NUM_THREADS=4 mpirun -npernode 2 \
        -pin "0,1,4,5_2,3,6,7" \
        likwid-pin -t intel -c L:0,2,1,3 ./a.out
```

Two MPI processes per socket



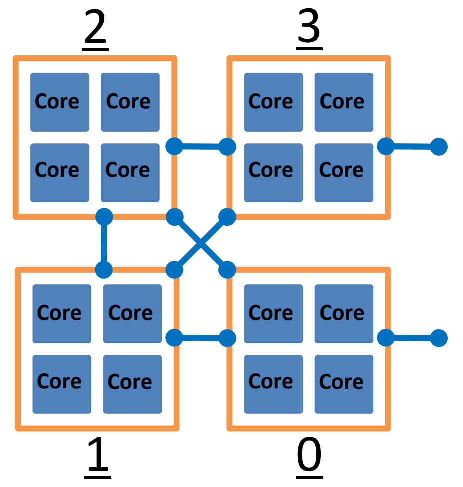



```
env OMP_NUM_THREADS=2 mpirun -npernode 4 \
        -pin "0,1_2,3_4,5_6,7" \
        likwid-pin -t intel -c L:0,1 ./a.out
```

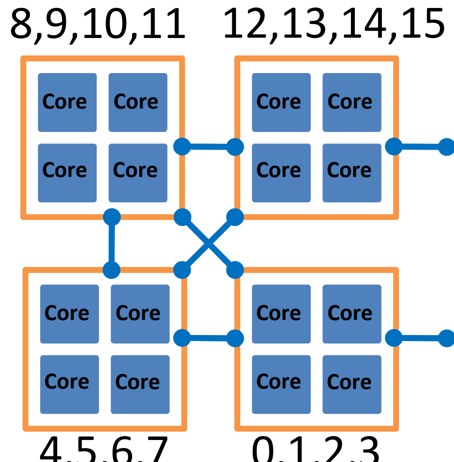
NUMA Control: Process and Memory Placement

- **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

Socket References

Example: numactl -N 1 ./a.out

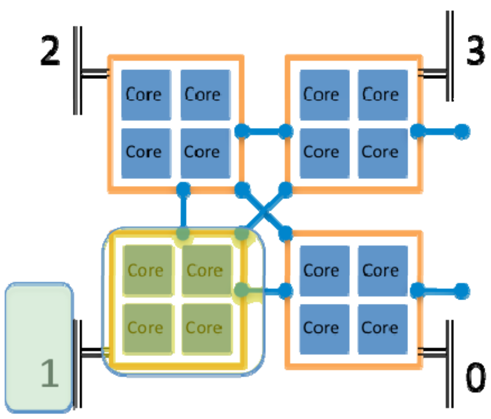
Core References

Example: numactl –c 0,1 ./a.out

Tutorial on Hybrid Programming PRACE Spring School 2011: Case Studies

3/23/11 11

NUMA Operations: Memory Placement

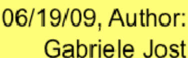


Memory: Socket References

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



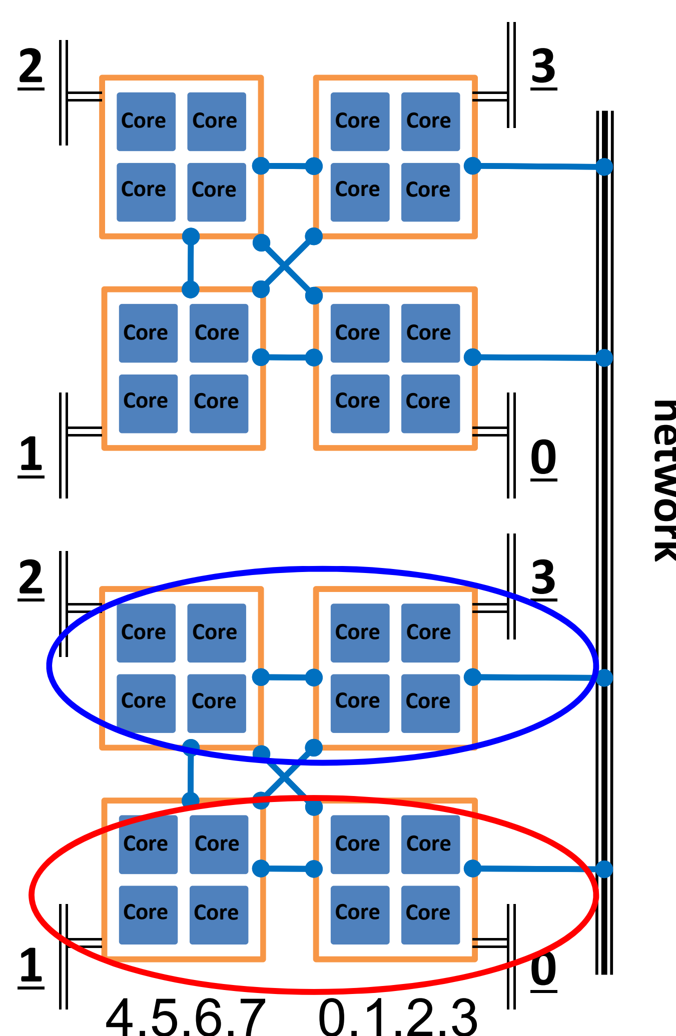
3/23/11 Tutorial on Hybrid Programming PRACE Spring School 2011: Case Studies 12

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

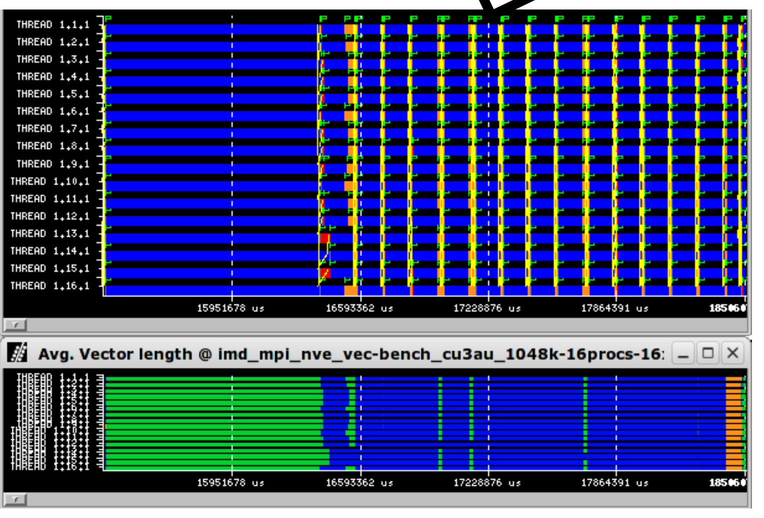
- **Vendor Supported Software:**
  - CrayPat/Cray Apprentice2: Offered by Cray for the XT Systems.
  - pgprof: Portland Group Performance Profiler
  - Intel Tracing Tools
  - IBM xprofiler
- **Public Domain Software:**
  - PAPI (Performance Application Programming Interface):
    - Support for reading 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

OPTIONAL

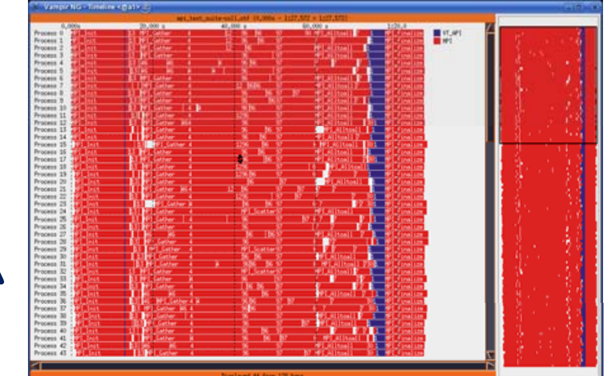
- **Paraver tracing is done with linking against (closed-source) `omptrace` or `ompitrace`**



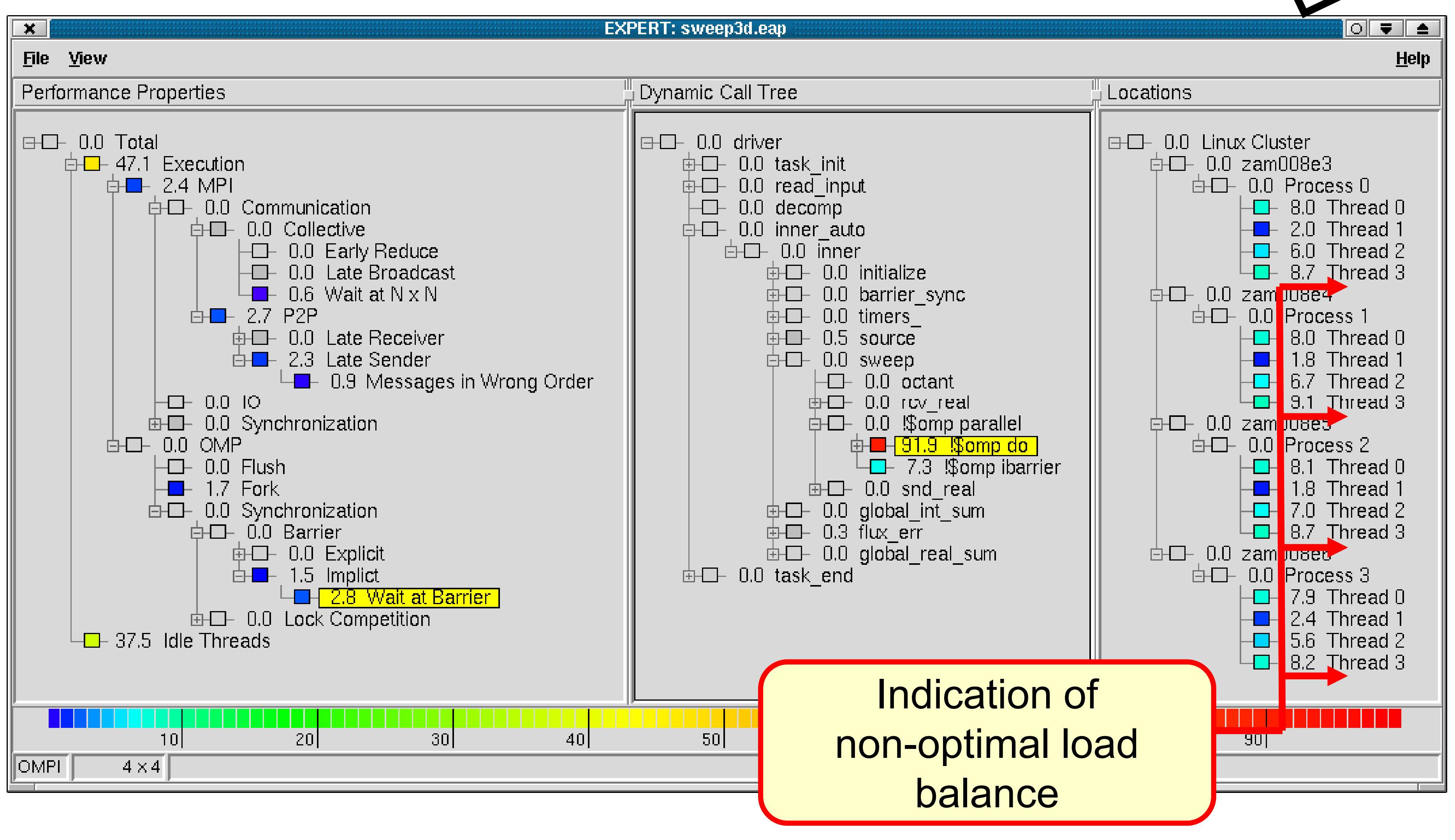
- **For Vampir/Vampirtrace performance analysis:**

```
./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`!)



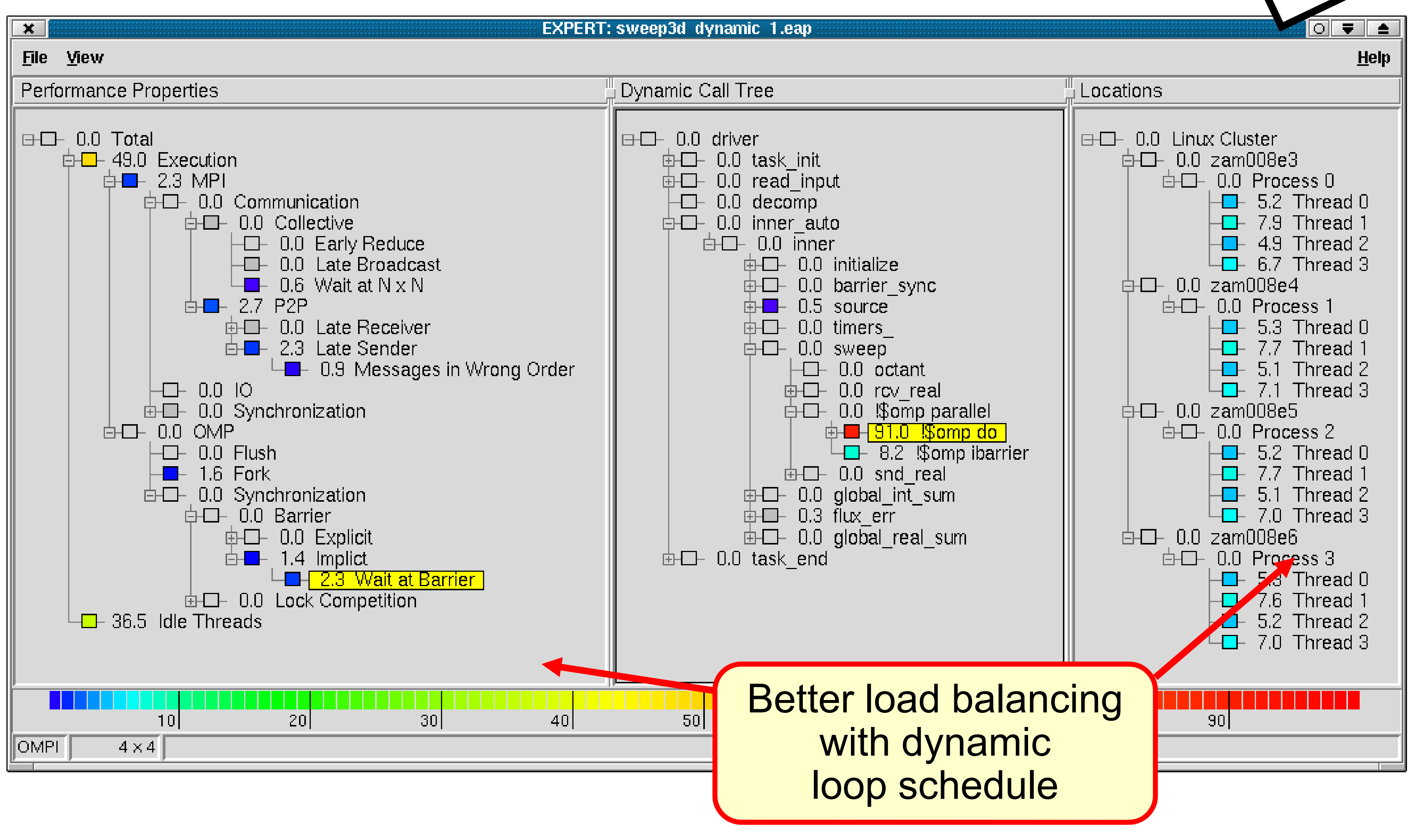
Scalasca – Example "Wait at Barrier"



Screenshots, courtesy of KOJAK JSC, FZ Jülich

Scalasca – Example “Wait at Barrier”, Solution

OPTIONAL



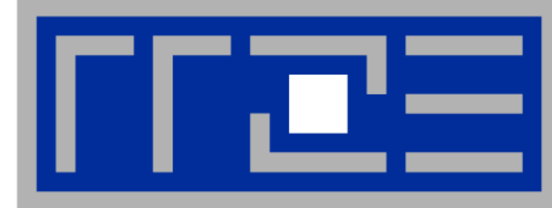
Screenshots, courtesy of KOJAK JSC, FZ Jülich

Hybrid MPI/OpenMP: 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**
  - Alternative: Do not let MPI processes span multiple NUMA domains

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
- **Case studies for hybrid MPI/OpenMP**
  - Overlap of communication and computation for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - Hybrid computing with accelerators and compiler directives
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

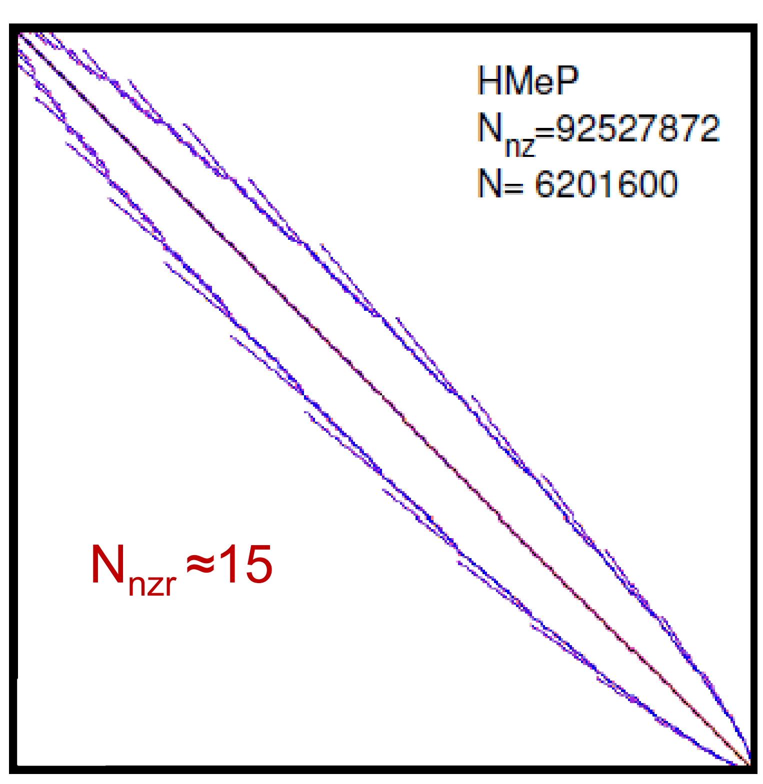


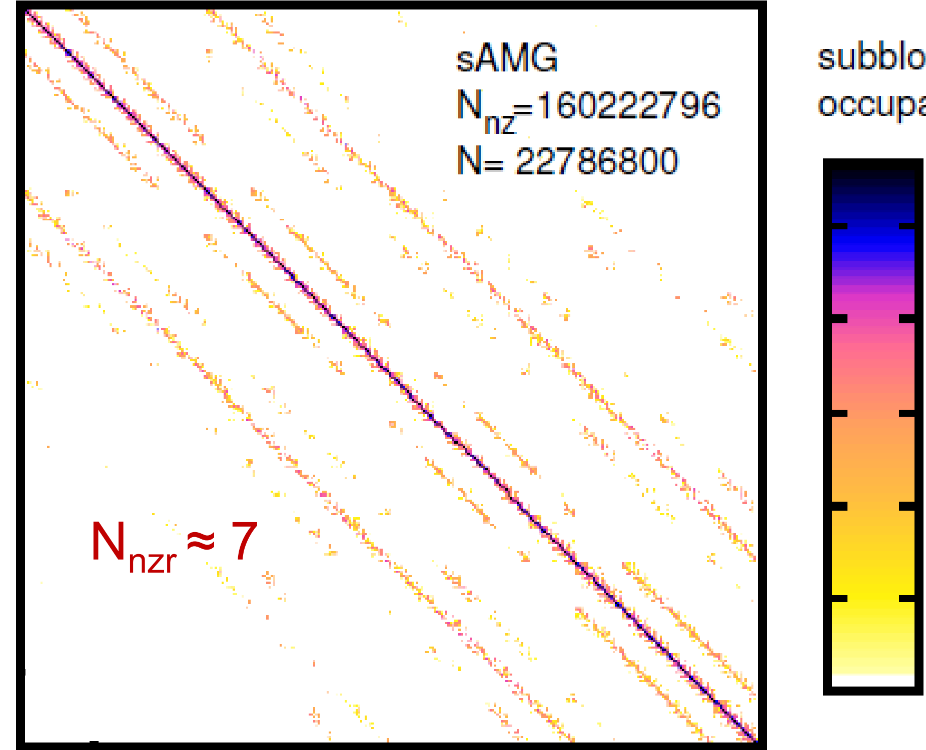
Case study:
MPI/OpenMP hybrid parallel
sparse matrix-vector multiplication

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

Local operation – no communication required

P0

P1

P2

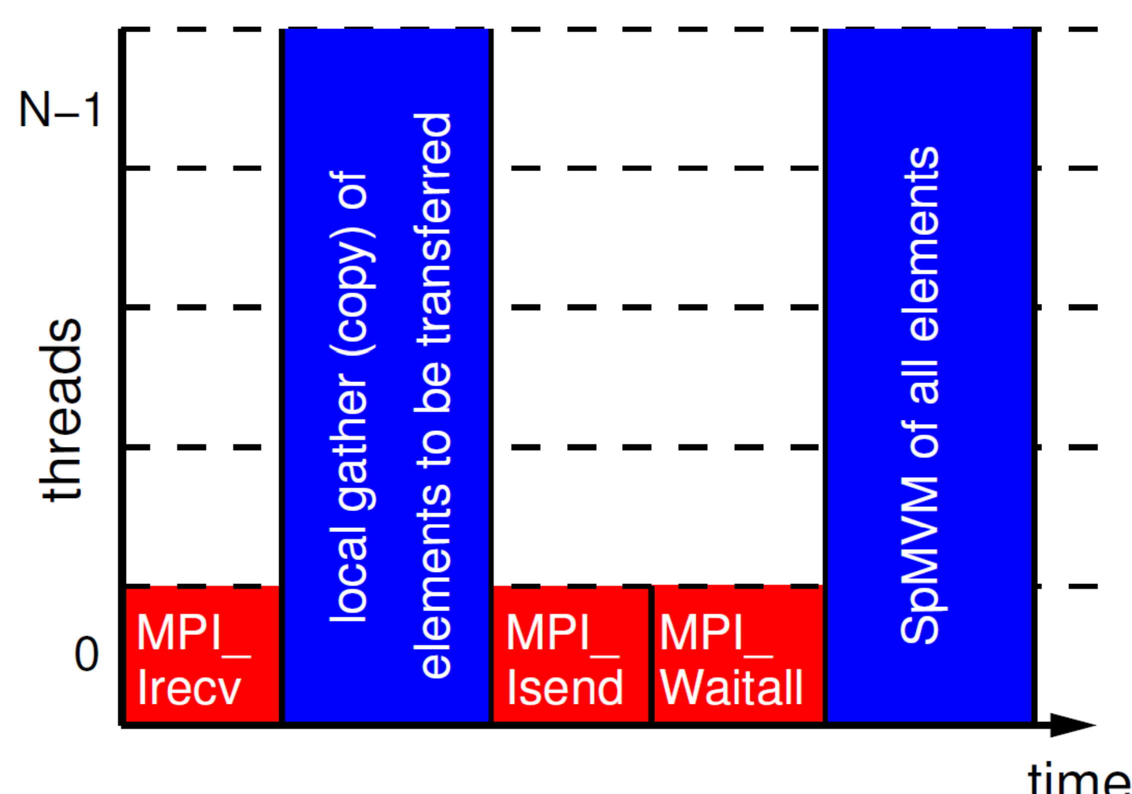
P3

=

Nonlocal RHS elements for P0

Distributed-memory parallelization of spMVM

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

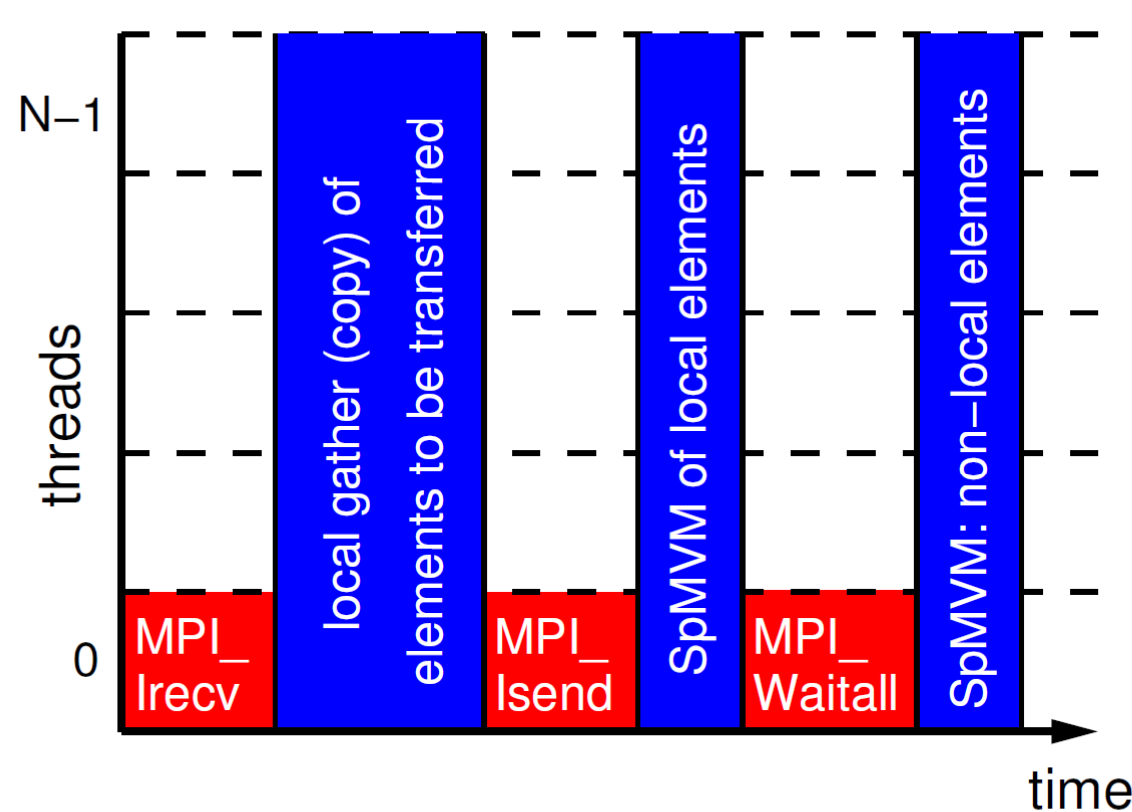


- **Standard concept for "hybrid MPI+OpenMP"**
- **Multithreaded computation (all threads)**
- **Communication only outside of computation**
- **Benefit of threaded MPI process only due to message aggregation and better load balancing**

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

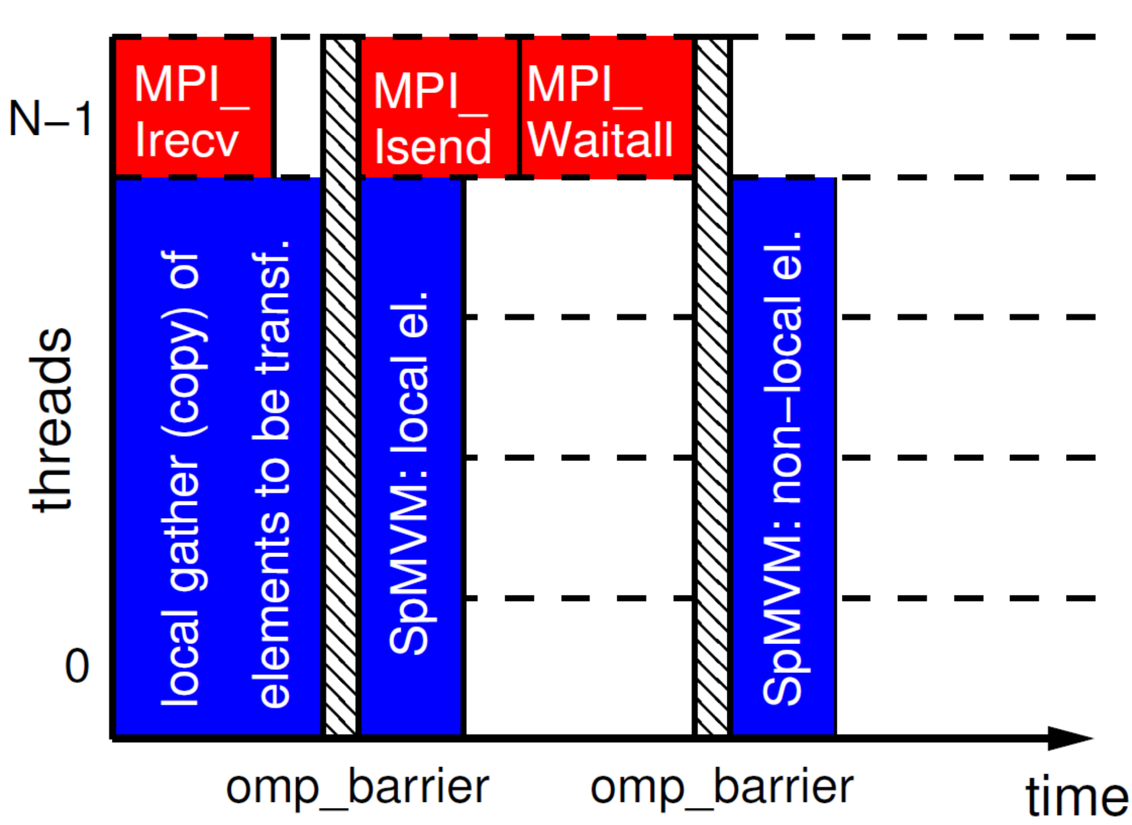
- **Variant 2: "MASTERONLY mode" with naïve overlap ("good faith hybrid")**



- **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: "FUNNELED-HYBRID mode" with dedicated comm. 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)



G. Schubert, H. Fehske, G. Hager, and G. Wellein: *Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems.* Parallel Processing Letters **21**(3), 339-358 (2011). DOI: 10.1142/S0129626411000254

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

performance [GFlop/s]

(a) MASTERONLY without overlap
(b) MASTERONLY naive overlap
(c) FUNNELED-HYB.
best Cray

one MPI process per physical core

FUNNELED-HYB. uses virtual core for communication @ 1 process/core

one MPIprocess per NUMA LD

50% efficiency w/ respect to best 1-node performance

one MPI process per node

#nodes

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

Results sAMG

performance [GFlop/s]

(a) MASTERONLY without overlap
(b) MASTERONLY naive overlap
(c) FUNNELED-HYB.
best Cray

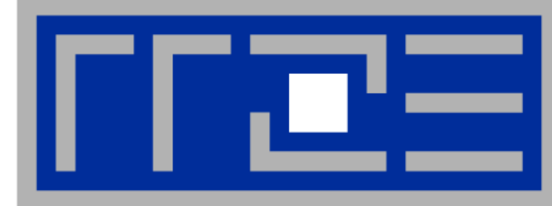
one MPI process per physical core

one MPI process per NUMA LD

one MPI process per node

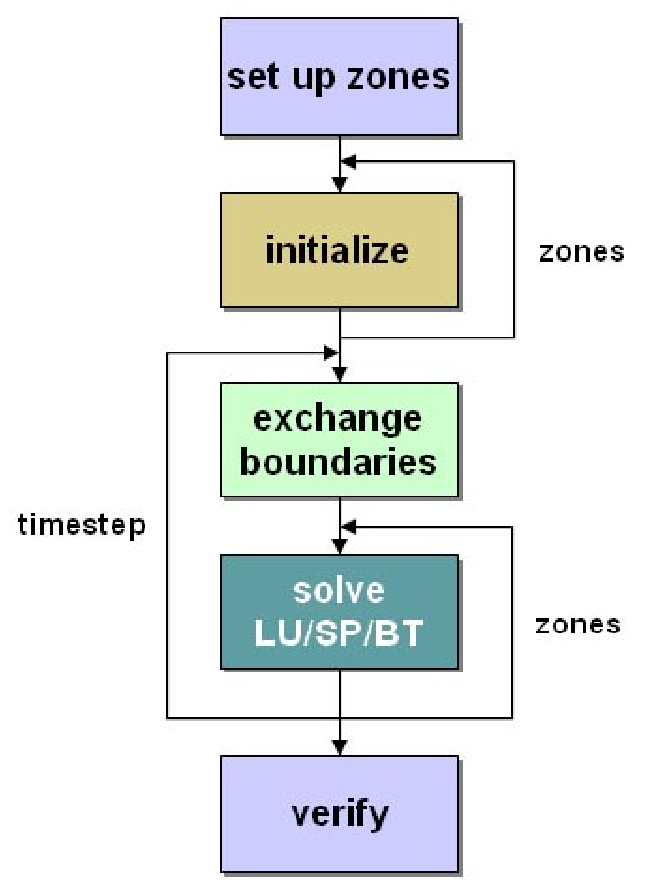
#nodes

- **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"

Golden Rule for ccNUMA: “First touch”

- A memory page gets mapped into the local memory of the processor that first touches it!
- Caveats:
  - possibly not enough local memory
  - "touch" means "write", not "allocate"

```
c-------------------------------------------------------------------
c      do one time step to touch all data
c-------------------------------------------------------------------
       do iz = 1, proc_num_zones
         zone = proc_zone_id(iz)
         call adi(rho_i(start1(iz)), us(start1(iz)),
     $            vs(start1(iz)), ws(start1(iz)),
          …..
     $ end do
       do iz = 1, proc_num_zones
         zone = proc_zone_id(iz)
         call initialize(u(start5(iz)),…
     $ end do
```

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

Hybrid code on cc-NUMA architectures

- **OpenMP:**
  - Support only per MPI process
  - Version 3.0 does not provide support to control to map threads onto CPUs. Support to specify thread placement is still under discussion.
  - Version 3.1 has support for binding of threads via OMP_PROC_BIND environment variable

- **MPI:**
  - Initially not designed for NUMA architectures or mixing of threads and processes, MPI-2 supports threads in MPI
  - API does not provide support for memory/thread placement

- **Vendor specific APIs to control thread and memory placement:**
  - Environment variables
  - likwid-pin (see first part of tutorial)
  - System commands like *numactl,taskset,dplace,omplace etc*

  → *http://www.halobates.de/numaapi3.pdf*

  → *More in "How-to's"*

Dell Linux Cluster Lonestar

- Located at the Texas Advanced Computing Center (TACC), University of Texas at Austin (http://www.tacc.utexas.edu)
- 1888 nodes, 2 Xeon Intel 6-Core 64-bit Westmere processors, 3.33 GHz, 24 GB memory per node, Peak Performance 160 Gflops per node, 3 channels from each processor's memory controller to 3 DDR3 ECC DIMMS, 1333 MHz,
- Processor interconnect, QPI, 6.4GT/s
- Node Interconnect: InfiniBand, fat-free topology, 40Gbit/sec point-to-point bandwidth
- More details: http://www.tacc.utexas.edu/user-services/user-guides/lonestar-user-guide
- Compiling the benchmarks:
  - ifort 11.1, Options: -O3 –ipo –openmp –mcmodel=medium
- Running the benchmarks:
  - MVAPICH 2
  - setenv OMP_NUM_THREADS=
  - ibrun tacc_affinity ./bt-mz.x

Dell Linux Cluster Lonestar 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                             | |
| +-------------------------------------------------------------+ |
+---------------------------------------------------------------+
```

Dell Linux Cluster Lonestar Topology

```
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 )
---------------------------------
```

Careful! Numbering scheme of cores is system-dependent (likwid-pin supports logical numbering, however)

NPB-MZ Class E scalability on Lonestar

GFlops
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
BT-MZ
SP-MZ
No idle cores
512 cores
1024 cores
2048 cores
4096 cores
8192 cores
Unexpected!
Unexpected!
256x1
256x6
512x6
256x8
4096x1
2048x2
1024x4
1024x6
512x8
512x12
MPIxOMP

Cores were allocated in chunks of 12. Therefore there are idle cores for some MPIxOMP combinations.

LU-MZ Class D Scalability on Lonestar

Cray XE6 Hermit

- Located at HLRS Stuttgart, Germany (https://wickie.hlrs.de/platforms/index.php/Cray_XE6)
- 3552 compute nodes 113.664 cores
- Each node contains two AMD 6276 Interlagos processors with 16 cores each, running at 2.3 GHz (TurboCore 3.3GHz)
- Around 1 Pflop theoretical peak performance
- 32 GB of main memory available per node
- 32-way shared memory system
- High-bandwidth interconnect using Cray Gemini communication chips.

Cray XE6 Hermit Node Topology

```
--------------------------------------------------------------------
CPU type:       AMD Interlagos processor
********************************************************************
Hardware Thread Topology
********************************************************************
Sockets:                2
Cores per socket:       16
Threads per core:       1
--------------------------------------------------------------------
Socket 0:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| |   0   | |   1   | |   2   | |   3   | |   4   | |   5   | |   6   | |   7   | |   8   | |   9   | |  10   | |  11   | |  12   | |  13   | |  14   | |  15   | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ |
| |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |
| +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ |
| +-----------------------------------------------------------------------------+ +-----------------------------------------------------------------------------+ |
| |                                      6MB                                    | |                                      6MB                                    | |
| +-----------------------------------------------------------------------------+ +-----------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
Socket 1:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| |  16   | |  17   | |  18   | |  19   | |  20   | |  21   | |  22   | |  23   | |  24   | |  25   | |  26   | |  27   | |  28   | |  29   | |  30   | |  31   | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | | 16kB  | |
| +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ +-------+ |
| +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ |
| |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |       2MB       | |
| +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ +-----------------+ |
| +-----------------------------------------------------------------------------+ +-----------------------------------------------------------------------------+ |
| |                                      6MB                                    | |                                      6MB                                    | |
| +-----------------------------------------------------------------------------+ +-----------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

4 NUMA domains

NPB-MZ Class E on Hermit

GFlops: 0, 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000

bt-mz sp-mz

8K cores

16K cores

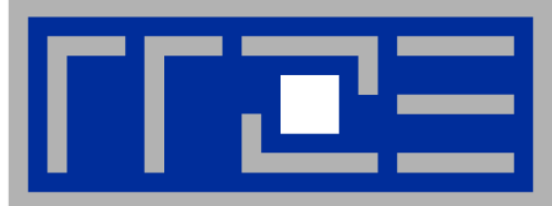
32K cores

128x2, 256x1, 128x4, 256x2, 512x1, 128x8, 256x4, 512x2, 1024x1, 128x16, 256x8, 2048x1, 1024x2, 512x4, 128x32, 256x16, 512x8, 1024x4, 2048x2, 4096x1, 512x16, 1024x8, 2048x4, 4096x2, 1024x16, 2048x8, 4096x4, 512x32, 4096x8, 2048x16, 1024x32

MPI x OpenMP

Conclusions from the NPB Case Studies

- **Hybrid Code Opportunities:**
  - Lower communication overhead
    - Few multi-threaded MPI processes vs. many single-threaded processes
    - Fewer number of calls and smaller chunks of data communicated
    - e.g., SP-MZ depending on interconnect and MPI stack
  - 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
    - eg BT-MZ
  - Increase parallelism
    - Domain decomposition as well as loop level parallelism can be exploited
    - eg SP-MZ, LU-MZ



Hybrid programming with accelerators and compiler directives

- **Under Discussion: OpenMP support for Accelerators in 4.0**
  - To be announced at SC12
  - Multiple devices of the same type (homogeneous)
  - Device type known at compile time
  - Automatic run-time and programmed user-control device selection
  - Structured and unstructured block data placement
    - Data regions and mirror directives
  - Synchronous and asynchronous data movement
  - Accelerator-style parallel launch with multiple 'threads' of execution on the device: e.g., accelerator parallel regions
  - Dispatch-style parallel launch(offload) to a single thread of execution on the device; eg accelerator tasks

Accelerator Memory Model

- **Current memory model:**
  - Relaxed-Consistency Shared-Memory
  - All threads have access to the memory
  - Data-sharing attributes: shared, private

- **Proposed additions to memory model**
  - Separate Host and Accelerator Memory
  - Data Movement Host ↔ Accelerator indicated by compiler directives
  - Updates to different memories indicated by compiler directives

```
#pragma omp acc_data [clause]
```

  - `acc_shared`
  - `acc_copyout, acc_copyin`

Accelerator Execution Model

- **Current OpenMP Execution Model:**
  - Execution starts single threaded
  - Fork-Join Threads at OpenMP parallel regions
  - Work-sharing indicated via compiler directives

- **Proposed additions to the Execution Model:**
  - Explicit accelerator regions or tasks are generated at beginning of accelerator regions

```
#pragma acc_region [clause]
```

  - Purpose: Define code that is to be run on accelerator
  - `acc_copyin (list)`
  - `acc_copyout (list)`

```
#pragma omp acc_loop [clause]
```

Test Case: Hybrid Jacobi using PGI directives

- PGI (http://www.pgroup.com) provides compiler directives for accelerators
  - Website for some documentation
- PGI active member of OpenMP Language committee
  - Use PGI Directives
- OpenMP Language committee follows path set by PGI
- Original Hybrid MPI/OpenMP implementation provided by courtesy of EPCC (Edingburgh Parallel Computing Center) (http://www.epcc.ed.ac.uk)

Example: Jacobi Iteration OpenMP directives

```
!$OMP PARALLEL DO PRIVATE(i,j,k)
     DO k = 1, Z, 1
       DO j = 1, Y, 1
         DO i = 1, X, 1
             data(i,j,k,new) = &
                ( data(i-1,j,k,old) + data(i+1,j,k,old) +&
                  data(i,j-1,k,old) + data(i,j+1,k,old) + &
                  data(i,j,k-1,old) + data(i,j,k+1,old) - &
                  edge(i,j,k) ) / 6.0
          END DO
        END DO
    END DO
```

Version 0: Unoptimized

```
!$omp acc_region
DO k = 1, Z, 1
   DO j = 1, Y, 1
      DO i = 1, X, 1
         data(i,j,k,new) = &
       ( data(i-1,j,k,old) + &
         data(i+1,j,k,old) + &
         data(i,j-1,k,old) + &
         data(i,j+1,k,old) + &
         data(i,j,k-1,old) + &
         data(i,j,k+1,old) - &
         edge(i,j,k) ) / 6.0
     END DO
   END DO
 END DO
!$omp end acc_region
```

```
jacobistep:
    59, Loop carried dependence of 'data' prevents parallelization
        Loop carried backward dependence of 'data' prevents vectorization
    60, Loop carried dependence of 'data' prevents parallelization
        Loop carried backward dependence of 'data' prevents vectorization
    61, Loop carried dependence of 'data' prevents parallelization
        Loop carried backward dependence of 'data' prevents vectorization
        Accelerator kernel generated
        59, !$acc do seq
        60, !$acc do seq
        61, !$acc do seq
            Non-stride-1 accesses for array 'data'
            Non-stride-1 accesses for array 'edge'
```

No performance increase when using accelerator

Example: Jacobi Iteration OpenMP Accelerator

```
!$omp acc_data copyin( edge ) copy( data )
!$omp acc_region_loop PRIVATE(i,j,k)
DO k = 1, Z, 1
   DO j = 1, Y, 1
      DO i = 1, X, 1
         data(i,j,k,new) = &
                 ( data(i-1,j,k,old) + data(i+1,j,k,old) + &
                   data(i,j-1,k,old) + data(i,j+1,k,old) + &
                   data(i,j,k-1,old) + data(i,j,k+1,old) - &
                   edge(i,j,k) ) / 6.0
     END DO
   END DO
 END DO
!$omp end acc_region_loop
!$omp end acc_data
```

Version 1: Optimized for parallelization….

```
!$acc data region local(temp2) &
  updatein(data(0:X+1,0:Y+1,0:Z+1,old))  &
  updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)
!$acc region
 temp2 = data (:,:,:,old)
 DO k = 1, Z, 1
  DO j = 1, Y, 1
   DO i = 1, X, 1
    data(i,j,k,new) = &
 ( temp2(i-1,j,k) + &
   temp2(i+1,j,k) + &
      & ……
   edge(i,j,k) ) / 6.0
  END DO
 END DO
END DO
!$acc end region
!$acc end data region
```

```
244, Loop is parallelizable
245, Loop is parallelizable
246, Loop is parallelizable
 Accelerator kernel generated
244, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
 245, !$acc do parallel, vector(4) ! blockidx%x
threadidx%y
 246, !$acc do vector(16) ! threadidx%x
          Cached references to size [18x6x6] block of
'temp2'
```

Version 1 (cont): ….and data movement

```
module glob
    real (kind(1.0e0)), dimension(:,:,:,:), allocatable,pinned :: data
    real (kind(1.0e0)), dimension(:,:,:), allocatable,pinned :: edge
    logical first
!$acc mirror(data,edge)
 end module glob

!$acc data region local(temp2)
  updatein(data(0:X+1,0:Y+1,0:Z
  updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)
!$acc region
 temp2 = data (:,:,:,old)
 DO k = 1, Z, 1
  DO j = 1, Y, 1
   DO i = 1, X, 1
   data(i,j,k,new) = ( temp2(i-1,j,k) + temp2(i+1,j,k) + … edge (I,j,k))/6.
  END DO
 END DO
END DO
!$acc end region
!$acc end data region
```

```
if (first) then
        macc = MOD(rank,2)+1
        call acc set device num
(macc,acc_device_type)
   endif
```

Use different devices for different MPI processes

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
- **Case studies for hybrid MPI/OpenMP**
  - Overlap of communication and computation for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - Hybrid computing with accelerators and compiler directives
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

Hybrid programming: Opportunities and pitfalls

- **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
  - Exploit accelerators
    - OpenMP-“like” models for accelerators exist
    - Less pain than explicit CUDA/OpenCL/whatever
    - Mustl still be well understood to be used efficiently

Hybrid programming: Opportunities and pitfalls

- **Pitfalls:**
  - Mapping problems
    - Every programming model requires topology awareness and affinity mechanisms
    - Changing the model (i.e., adding another level of parallelism) will not make the problems go away
    - SMT adds to complexity
  - Inherent OpenMP overheads
    - Implicit OpenMP barriers
    - Many OpenMP regions also mean frequent synchronization
    - SMT adds to complexity (again)
    - ccNUMA is more complex to handle with OpenMP
  - Complexity of programming
    - Simple MASTERONLY style is just the beginning and leaves opportunities on the table
    - FUNNELED-HYBRID style promises best performance
    - OpenMP tasking may take away complexity, but must be fully understood

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 to cores and nodes
- **Application Requirements:**
  - Expose multiple levels of parallelism
    - Coarse-grained and fine-grained
    - Enough fine-grained parallelism to allow OpenMP to scale “reasonably well” (up to the inherent limitations of multicore chips)
- **Performance is not portable:**
  - 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,…

Tutorial outline (2)

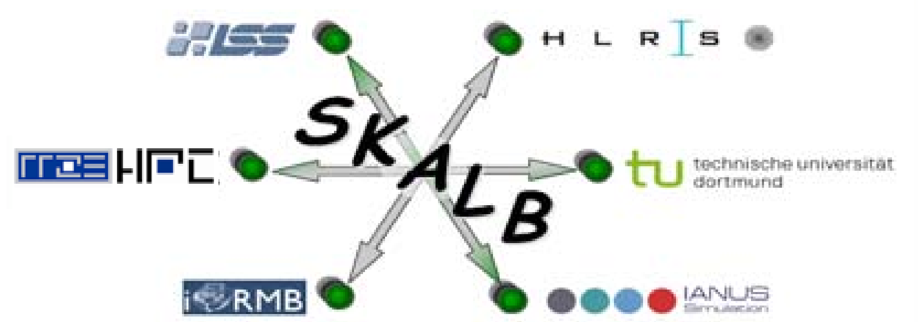
- **Hybrid MPI/OpenMP**
  - MPI vs. OpenMP
  - Thread-safety quality of MPI libraries
  - Strategies for combining MPI with OpenMP
  - Topology and mapping problems
  - Potential opportunities
- **Case studies for hybrid MPI/OpenMP**
  - Overlap of communication and computation for hybrid sparse MVM
  - The NAS parallel benchmarks (NPB-MZ)
  - Hybrid computing with accelerators and compiler directives
- **Summary: Opportunities and Pitfalls of Hybrid Programming**
- **Overall summary and goodbye**

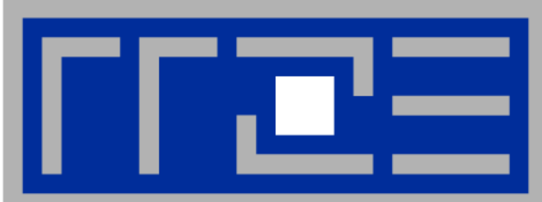
Overall tutorial summary

- **Modern multicore-based hardware, even the most "commodity" type, is hierarchical**
  - SMT, cores, cache groups, NUMA, sockets, nodes, networks
- **Ignoring its specific properties costs performance**
  - Scalable and non-scalable resources
- **Tools (even simple ones!) can help figure out what's going on**
  - Know what your code does to the hardware!

- **The programming model must be able to exploit the hardware up to the relevant bottleneck**
  - All models have their pitfalls and there is no simple answer to "what is best?"
  - Mapping/mismatch problems are the most prevalent ones on hybrid hardware
  - Opportunities for hybrid MPI+OpenMP do exist, even for very simple MASTERONLY style – but you need to dig deep to get it all!

Thank you





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

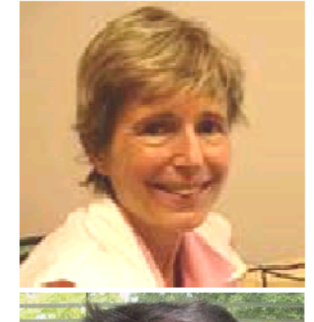
References

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, H. Fehske, G. Hager, and G. Wellein: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters 21(3), 339-358 (2011). DOI: 10.1142/S0129626411000254
- 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.fau.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. His textbook "Introduction to High Performance Computing for Scientists and Engineers" is recommended reading in HPC-related lectures and workshops worldwide. See his blog at http://blogs.fau.de/hager for current activities, publications, talks, and teaching.



Gabriele Jost (gabriele.jost@amd.com) 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. After engagements with Sun/Oracle and the Texas Advanced Computing Center (TACC) Gabriele joined Advanced Micro Devices (AMD) in 2011 as a design engineer in the Systems Performance Optimization group.

Rolf Rabenseifner (rabenseifner@hlrs.de) holds a PhD in Computer Science from the University of Stuttgart. Since 1984, he works at the High-Performance Computing-Center Stuttgart (HLRS). Since 1996, he has been a member of the MPI-2 Forum and since Dec. 2007 he works in the steering committee for MPI-3. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is head of Parallel Computing - Training and Application Services at HLRS. Recent research includes benchmarking, parallel programming models for clusters of SMP nodes, and optimization of MPI collective routines. In workshops and summer schools, he teaches parallel programming models, and in Jan. 2012, he was appointed as GCS' PATC director.

Jan Treibig (jan.treibig@rrze.fau.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. He is the main developer of the LIKWID multicore tool suite. Recently he has founded a spin-off company, "LIKWID High Performance Programming."

Abstract

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

- **Presenters:** Georg Hager, Gabriele Jost, Jan Treibig, Rolf Rabenseifner
- **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.