

Performance-oriented programming on multicore-based systems

Georg Hager^(a), Jan Treibig^(a), and Gerhard Wellein^(a,b)

^(a)HPC Services, Erlangen Regional Computing Center (RRZE)

^(b)Department for Computer Science

Friedrich-Alexander-University Erlangen-Nuremberg

NEC User Group Meeting

June 12, 2012



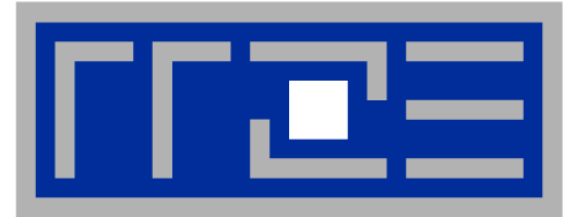
**There is no alternative to knowing what is going on
between your code and the hardware**

**Without performance modeling,
optimizing code is like stumbling in the dark**

Performance x Flexibility = constant
a.k.a. Abstraction is the natural enemy of efficiency



- **A short history of multicore processing**
- **The LIKWID tool suite for multicore programming**
- **Data access on modern processors**
 - Basic performance benchmarks and properties
 - The balance metric: Bandwidth-based performance modeling
 - Optimizing data access by code transformations
- **Performance properties of parallel code on multicore processors and nodes**
 - Exploration by microbenchmarks
 - Sparse matrix-vector multiplication
- **ccNUMA: Properties and efficient programming**
- **Simultaneous Multithreading (SMT)**
 - Principles
 - Expected benefits
 - Mythbusting



Multicore processor and system architecture

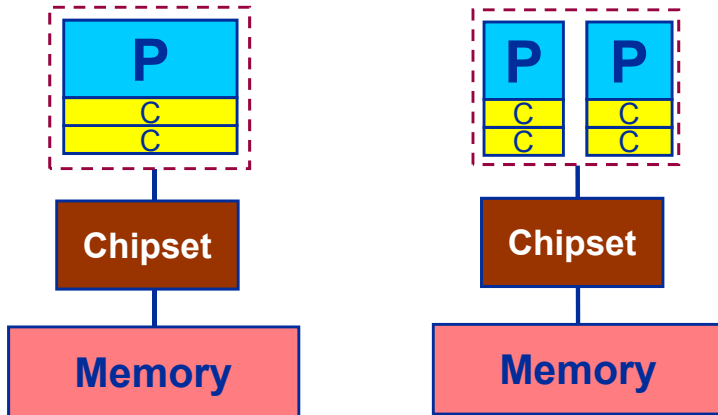
Basics

The x86 multicore evolution so far

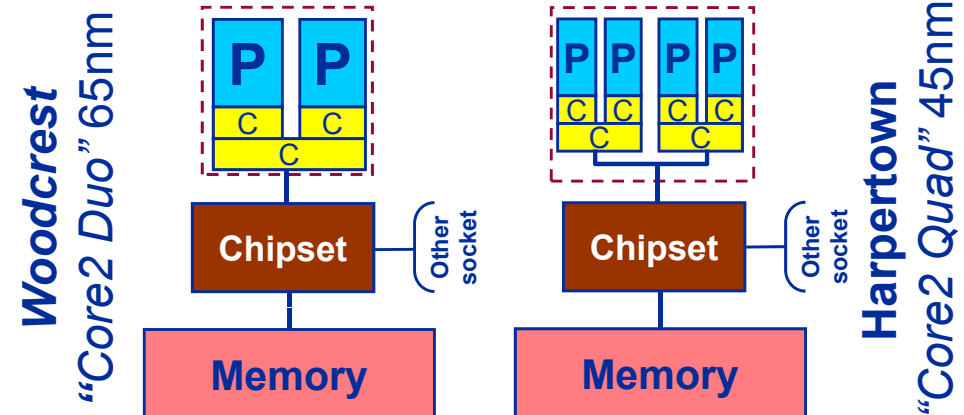
Intel Single-Dual-/Quad-/Hexa-/Cores (one-socket view)



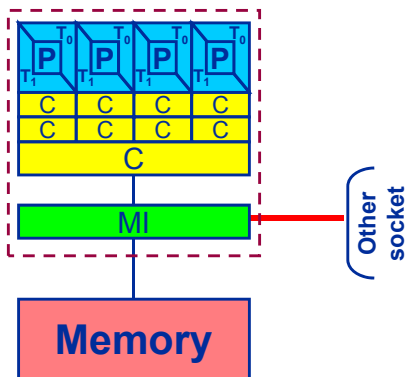
2005: "Fake" dual-core



2006: True dual-core

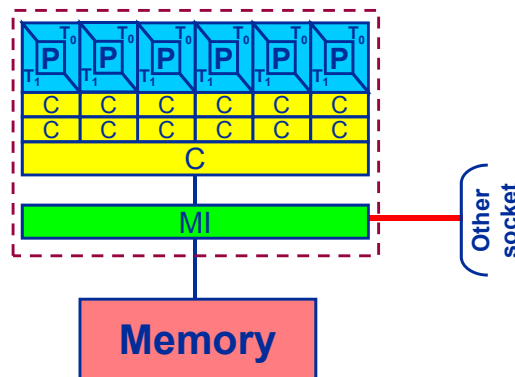


2008: Simultaneous Multi Threading (SMT)



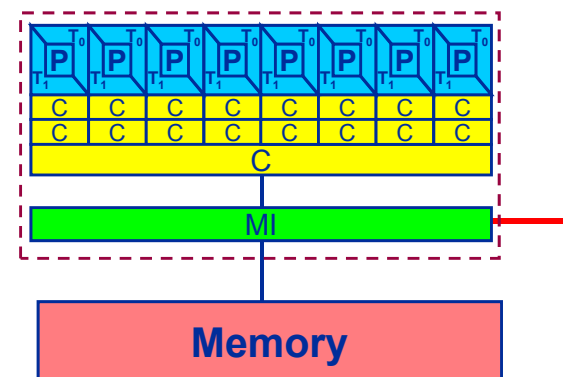
Nehalem EP
"Core i7"
45nm

2010: 6-core chip



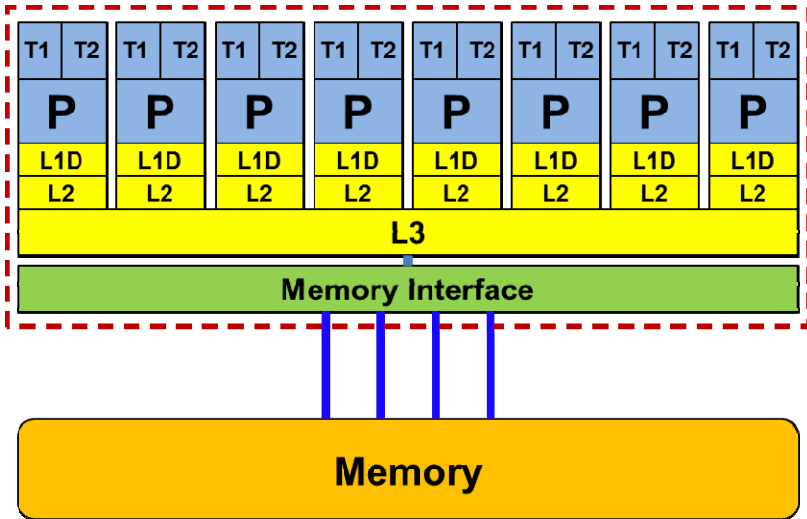
Westmere EP
"Core i7"
32nm

2012: Wider SIMD units
AVX: 256 Bit



Sandy Bridge EP
"Core i7"
32nm

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



Floating Point (FP) Performance:

$$P = n_{\text{core}} * F * S * v$$

n_{core}	number of cores:	8
F	FP instructions per cycle: (1 MULT and 1 ADD)	2
S	FP ops / instruction:	4 (dp) / 8 (sp) (256 Bit SIMD registers – “AVX”)
v	Clock speed :	2.5 GHz

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

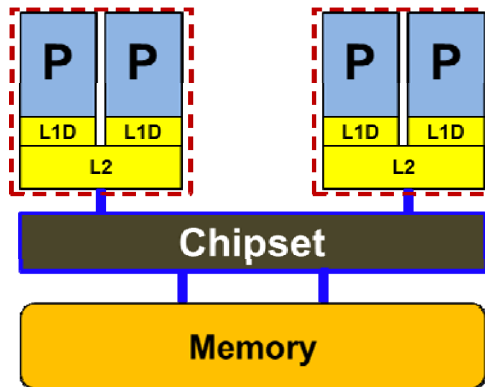
TOP500 rank 1 (1996)

$$P = 160 \text{ GF/s (dp) / 320 GF/s (sp)}$$

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



Yesterday (2006): Dual-socket Intel “Core2” node:

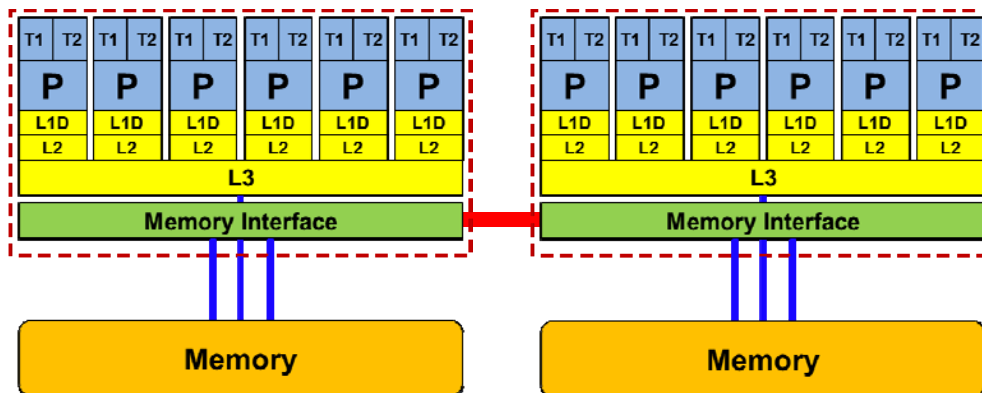


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system “anisotropy”

Today: Dual-socket Intel (Westmere) 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?

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

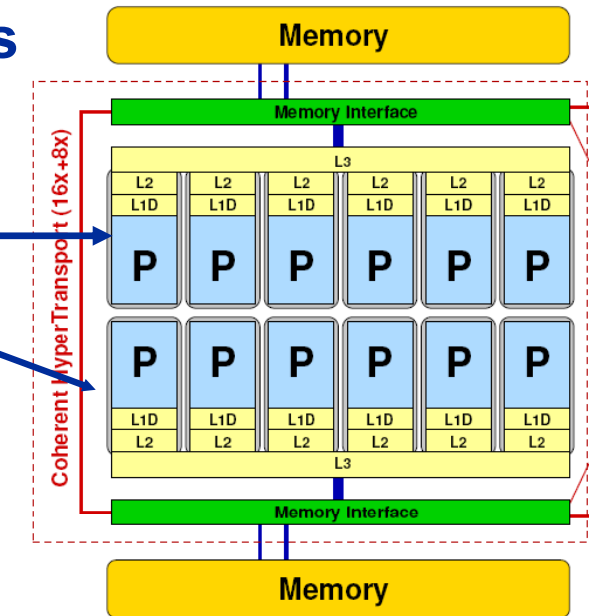
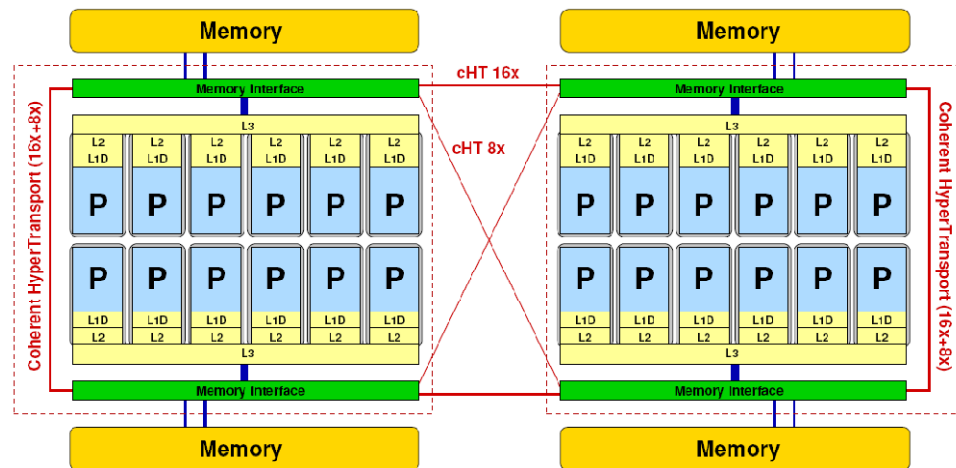
Back to the 2-chip-per-case age

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 to scale:**
2 x 2 memory channels vs. 1 x 4 memory channels per socket

Another flavor of "SMT"

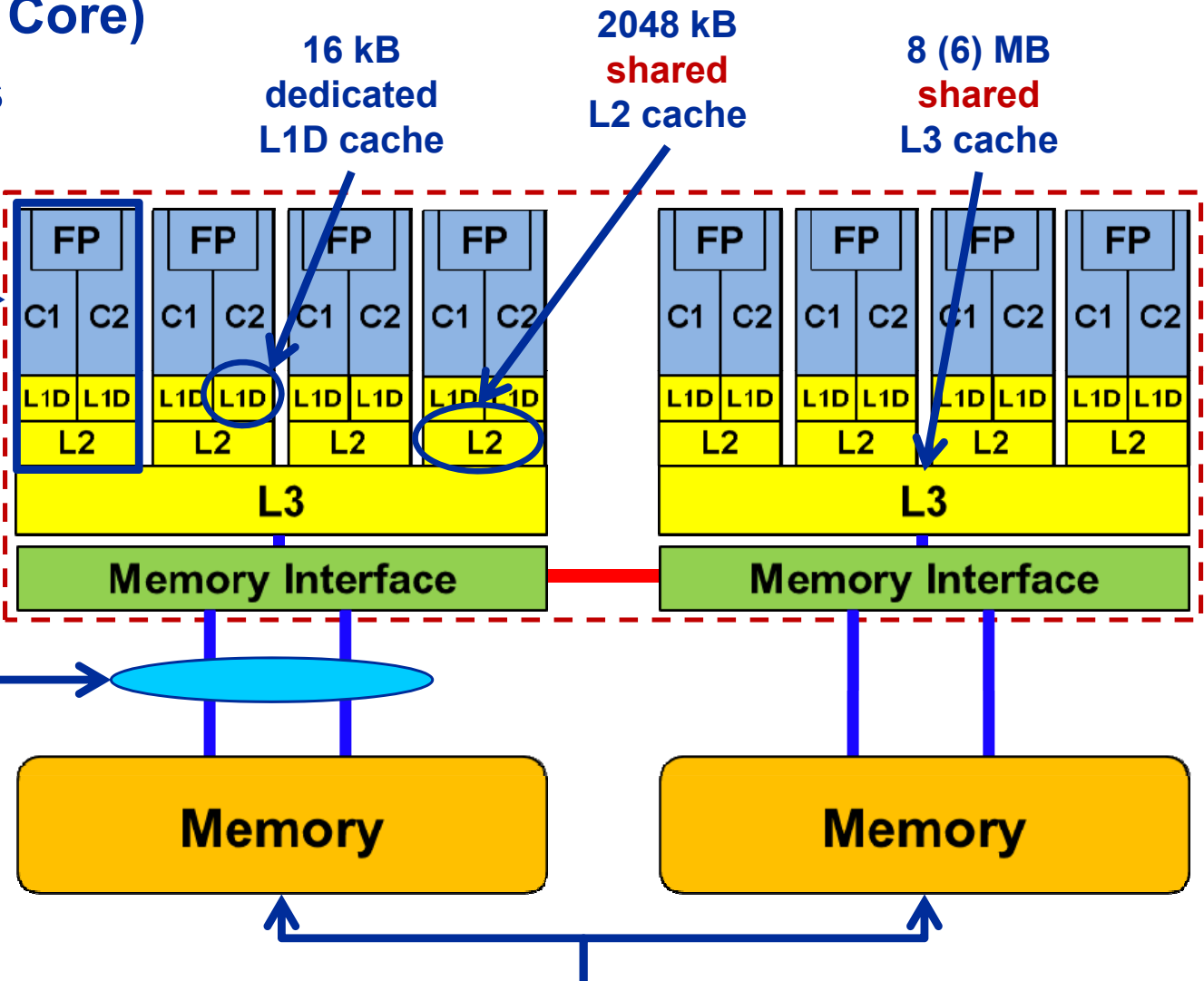
AMD Interlagos / Bulldozer



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

Each Bulldozer module:

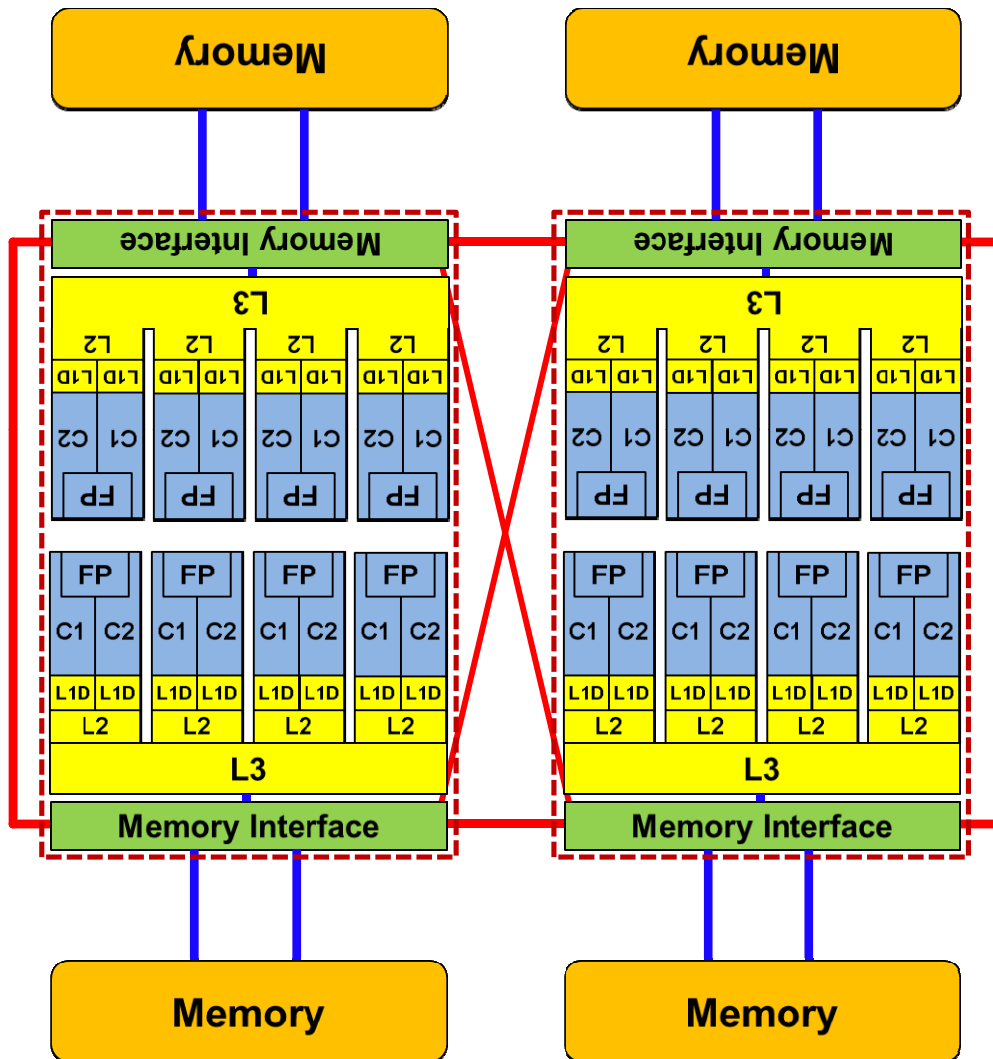
- 2 "lightweight" cores
- 1 FPU: 4 MULT & 4 ADD (double precision) / cycle
- Supports AVX
- Supports FMA4



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

2 NUMA domains per socket

16-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)**



- **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!

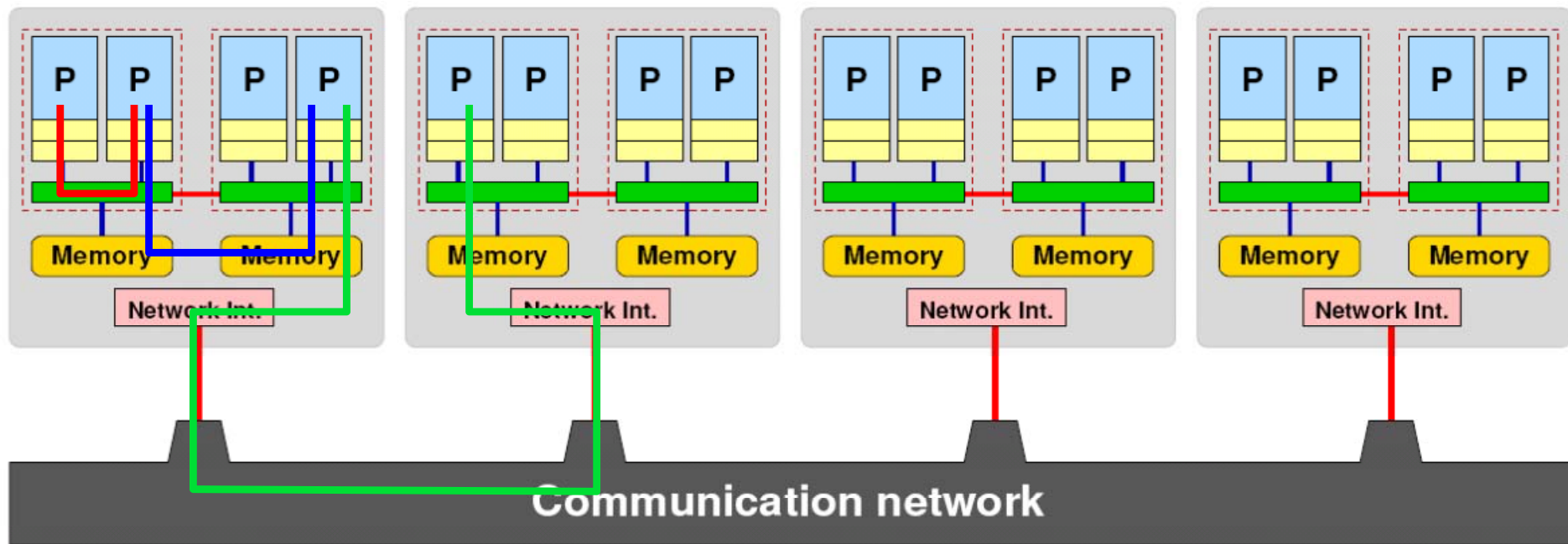
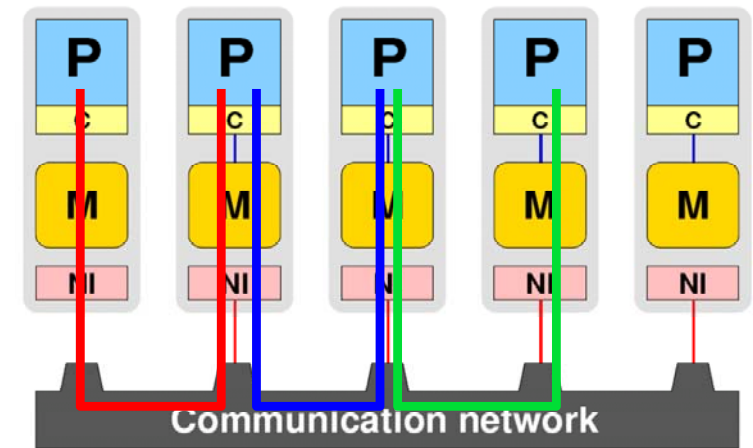
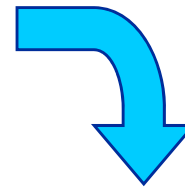


- Machine structure is invisible to user:

- Very simple programming model
- MPI “knows what to do”!?

- Performance issues

- Intranode vs. internode MPI
- Node/system topology



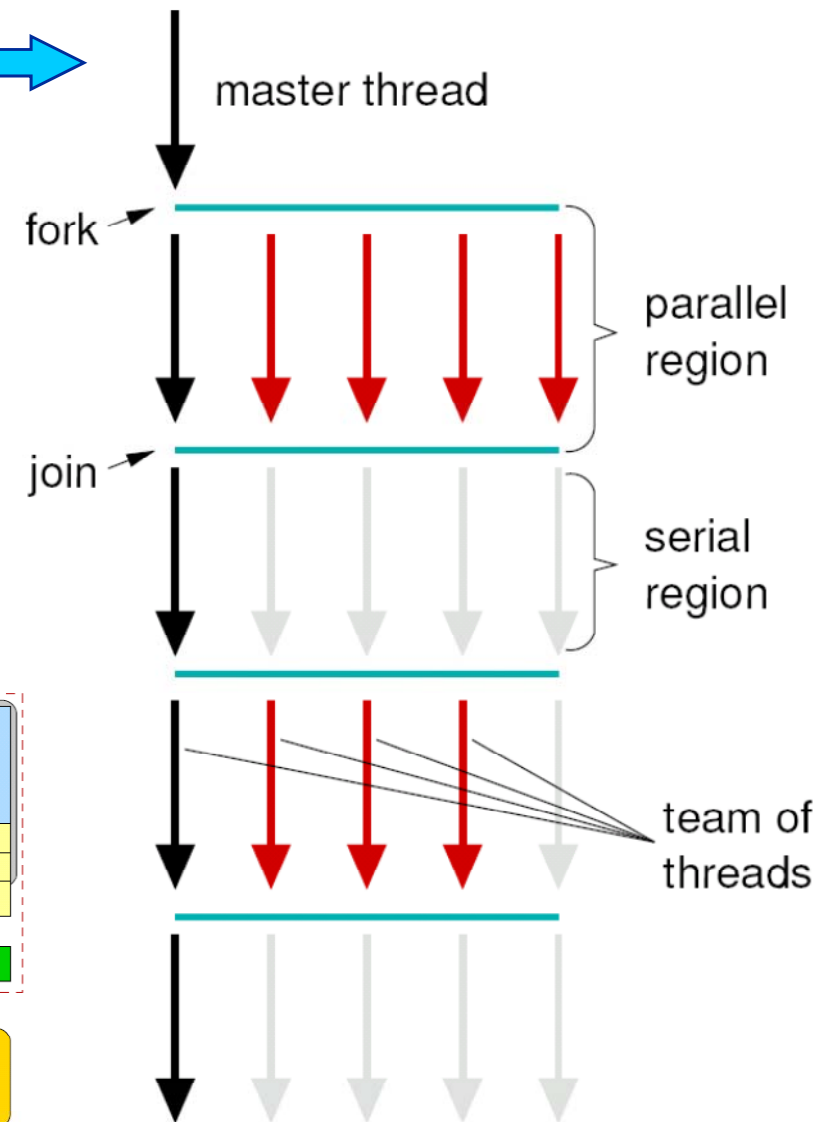
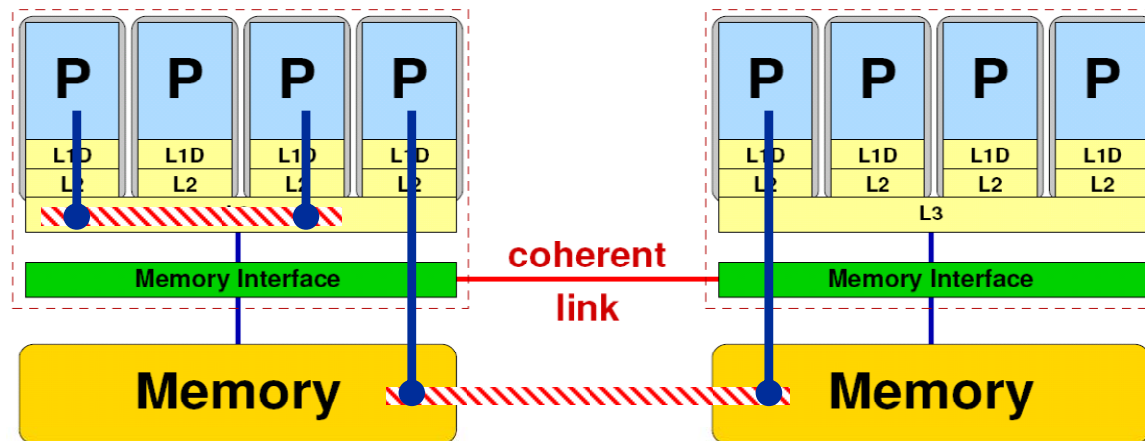


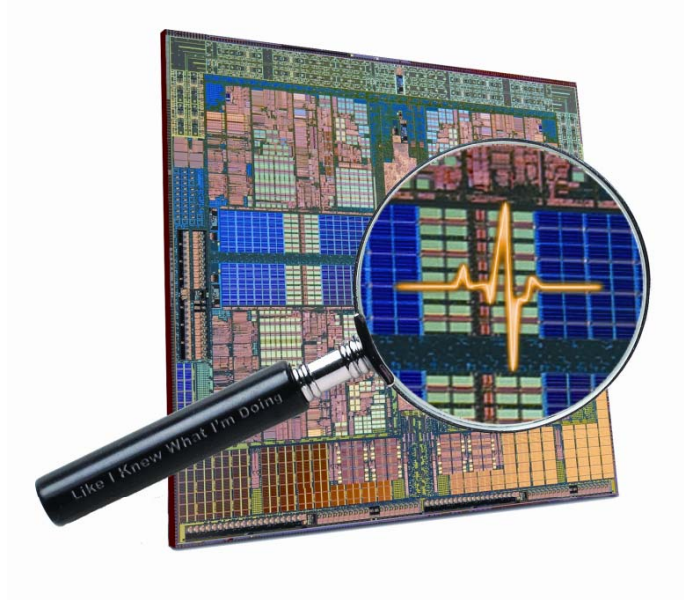
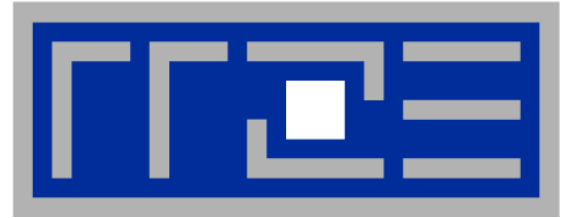
- Machine structure is invisible to user

- Very simple programming model
- Threading SW (OpenMP, pthreads, TBB,...) should know about the details

- Performance issues

- Synchronization overhead
- Memory access
- Node topology





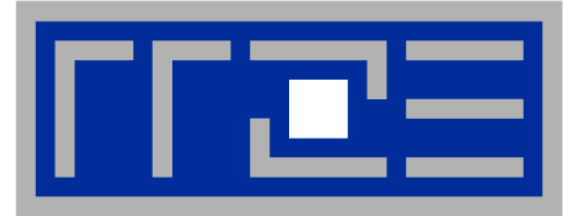
Performance tools for parallel multicore programming

- **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/>



Probing node topology

- **Standard tools**
- **likwid-topology**

How do we figure out the node topology?



- **Topology =**

- Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
- Which cores share which cache levels?
- Which hardware threads (“logical cores”) share a physical core?

- **Linux**

- `cat /proc/cpuinfo` is of limited use
- Core numbers may change across kernels and BIOSes even on identical hardware
- `numactl --hardware` prints ccNUMA node information
- Information on caches is harder to obtain



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



- **Based on `cpuid` information**
- **Functionality:**
 - Measured clock frequency
 - Thread topology
 - Cache topology
 - Cache parameters (-c command line switch)
 - ASCII art output (-g command line switch)
- **Currently supported (more under development):**
 - Intel Core 2 (45nm + 65 nm)
 - Intel Nehalem + Westmere (Sandy Bridge in beta phase)
 - AMD Bulldozer/Interlagos (beta)
 - AMD K10 (Quadcore and Hexacore)
 - AMD K8
 - Linux OS

Output of `likwid-topology -g`

on one node of Cray XE6 "Hermit"



CPU type: AMD Interlagos processor

Hardware Thread Topology

Sockets: 2

Cores per socket: 16

Threads per core: 1

HWThread	Thread	Core	Socket
0	0	0	0
1	0	1	0
2	0	2	0
3	0	3	0
[...]			
16	0	0	1
17	0	1	1
18	0	2	1
19	0	3	1
[...]			

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

Socket 1: (16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

Cache Topology

Level: 1

Size: 16 kB

Cache groups: (0) (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22) (23) (24) (25) (26) (27) (28) (29) (30) (31)

Output of likwid-topology continued



```
-----  
Level: 2  
Size: 2 MB  
Cache groups: ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 ) ( 16 17 ) ( 18  
19 ) ( 20 21 ) ( 22 23 ) ( 24 25 ) ( 26 27 ) ( 28 29 ) ( 30 31 )  
-----
```

```
Level: 3  
Size: 6 MB  
Cache groups: ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 ) ( 16 17 18 19 20 21 22 23 ) ( 24 25 26  
27 28 29 30 31 )  
-----
```

```
*****
```

NUMA Topology

```
*****
```

```
NUMA domains: 4  
-----
```

Domain 0:

```
Processors: 0 1 2 3 4 5 6 7  
Memory: 7837.25 MB free of total 8191.62 MB  
-----
```

Domain 1:

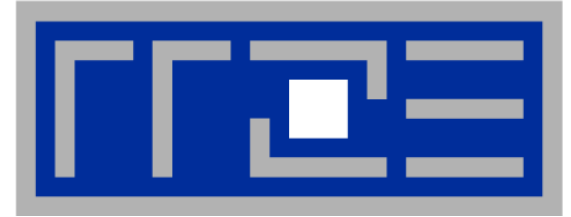
```
Processors: 8 9 10 11 12 13 14 15  
Memory: 7860.02 MB free of total 8192 MB  
-----
```

Domain 2:

```
Processors: 16 17 18 19 20 21 22 23  
Memory: 7847.39 MB free of total 8192 MB  
-----
```

Domain 3:

```
Processors: 24 25 26 27 28 29 30 31  
Memory: 7785.02 MB free of total 8192 MB  
-----
```

Probing performance behavior

likwid-perfctr



- **How do we find out about the performance properties and requirements of a parallel code?**
 - Profiling via advanced tools is often overkill
- **A coarse overview is often sufficient**
 - **likwid-perfctr** (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
 - Simple end-to-end measurement of hardware performance metrics
 - 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



```
$ 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

Always measured

Configured metrics (this group)

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

Derived metrics



Things to look at (in roughly this order)

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

Caveats

- Load imbalance may not show in CPI or # of instructions
 - **Spin loops** in OpenMP barriers/MPI blocking calls
 - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- **Cache miss metrics are overrated**
 - If I really know my code, I can often *calculate* the misses
 - Runtime and resource utilization is much more important



- Instructions retired / CPI may not be a good indication of useful workload – at least for numerical / FP intensive codes....
- Floating Point Operations Executed** is often a better indicator
- Waiting / “Spinning” in barrier generates a high instruction count

Event	core 0	core 1	core 2	core 3	core 4	core 5
INSTR_RETIRED_ANY	2.10045e+10	1.90983e+10	1.729e+10	1.60898e+10	1.67958e+10	1.84689e+10
CPU_CLK_UNHALTED_CORE	1.82569e+10	1.81203e+10	1.81802e+10	1.82084e+10	1.82334e+10	1.82484e+10
CPU_CLK_UNHALTED_REF	1.66053e+10	1.6473e+10	1.65274e+10	1.65531e+10	1.65758e+10	1.65894e+10
FP_COMP_OPS_EXE_SSE_FP_PACKED	2.77016e+08	7.83476e+08	1.39355e+09	1.94365e+09	2.38059e+09	2.85981e+09
FP_COMP_OPS_EXE_SSE_FP_SCALAR	1.70802e+08	2.64065e+08	2.23153e+08	2.60835e+08	2.30434e+08	2.07293e+08
FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION	19	0	0	0	0	0
FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION	4.47818e+08	1.04754e+09	1.61671e+09	2.20448e+09	2.61102e+09	3.0671e+09

```

!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
    
```

Metric	core 0	core 1	core 2	core 3	core 4	core 5
Runtime [s]	6.84594	6.79471	6.81716	6.82773	6.83711	6.84274
Clock [MHz]	2932.07	2933.51	2933.51	2933.51	2933.51	2933.51
CPI	0.869191	0.948789	1.05148	1.13167	1.08559	0.988061
DP MFlops/s	109.192	275.833	453.48	624.893	751.96	892.857



```
env OMP_NUM_THREADS=6 likwid-perfctr -t intel -C S0:0-5 -g FLOPS_DP ./a.out
```

Event	core 0	core 1	core 2	core 3	core 4	core 5
INSTR_RETIRED_ANY	1.83124e+10	1.74784e+10	1.68453e+10	1.66794e+10	1.76685e+10	1.91736e+10
CPU_CLK_UNHALTED_CORE	2.24797e+10	2.23789e+10	2.23802e+10	2.23808e+10	2.23799e+10	2.23805e+10
CPU_CLK_UNHALTED_REF	2.04416e+10	2.03445e+10	2.03456e+10	2.03462e+10	2.03453e+10	2.03459e+10
FP_COMP_OPS_EXE_SSE_FP_PACKED	3.45348e+09	3.43035e+09	3.37573e+09	3.39272e+09	3.26132e+09	3.2377e+09
FP_COMP_OPS_EXE_SSE_FP_SCALAR	2.93108e+07	3.06063e+07	2.9704e+07	2.96507e+07	2.41141e+07	2.37397e+07
FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION	19	0	0	0	0	0
FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION	3.48279e+09	3.46096e+09	3.40543e+09	3.42237e+09	3.28543e+09	3.26144e+09

Higher CPI but
better performance

Metric	core 0	core 1	core 2	core 3	core 4	core 5
Runtime [s]	8.42938	8.39157	8.39206	8.3923	8.39193	8.39218
Clock [MHz]	2932.73	2933.5	2933.51	2933.51	2933.51	2933.51
CPI	1.22757	1.28037	1.32857	1.34182	1.26666	1.16726
DP MFlops/s	850.727	845.212	831.703	835.865	802.952	797.113
Packed MUOPS/s	423.566	420.729	414.03	416.114	399.997	397.101
Scalar MUOPS/s	3.59494	3.75383	3.64317	3.63663	2.95757	2.91165
SP MUOPS/s	2.33033e-06	0	0	0	0	0
DP MUOPS/s	427.161	424.483	417.673	419.751	402.955	400.013

```
!$OMP PARALLEL DO
```

```
DO I = 1, N
```

```
DO J = 1, N
```

```
    x(I) = x(I) + A(J,I) * y(J)
```

```
ENDDO
```

```
ENDDO
```

```
!$OMP END PARALLEL DO
```



- **likwid-perfctr counts events on the specified cores**
- **No connection to running code / binary / process / thread**
- **Listen what currently happens without any overhead:**

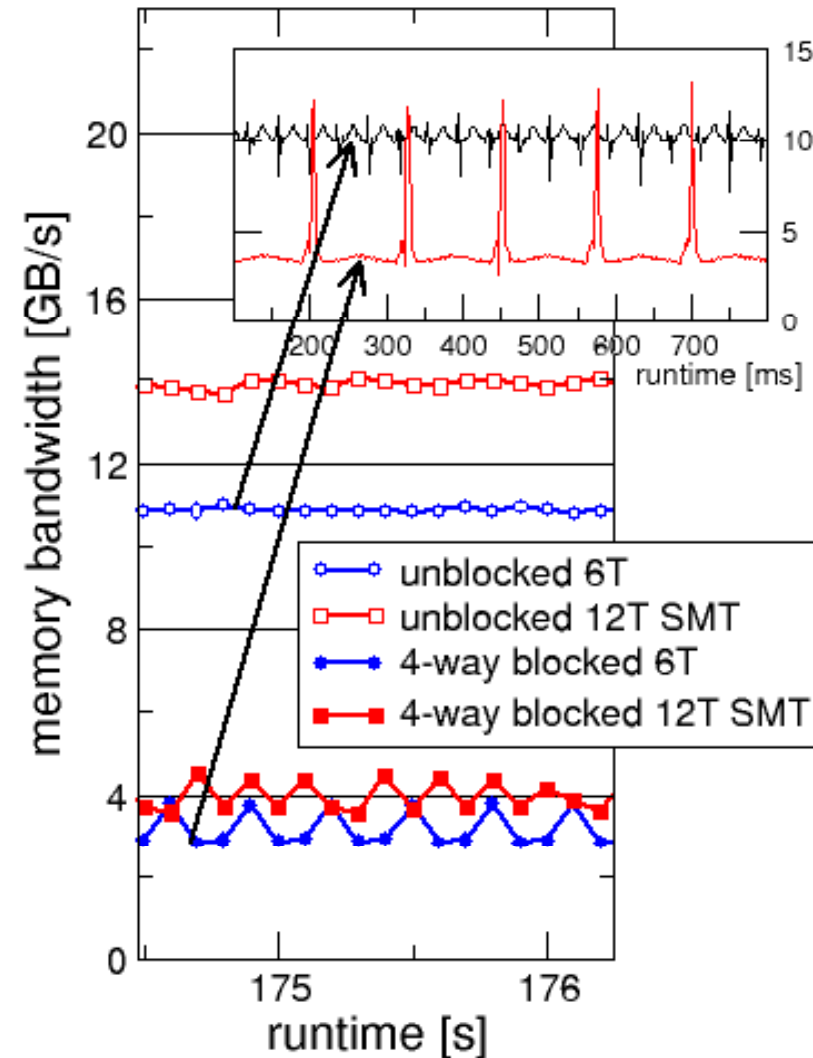
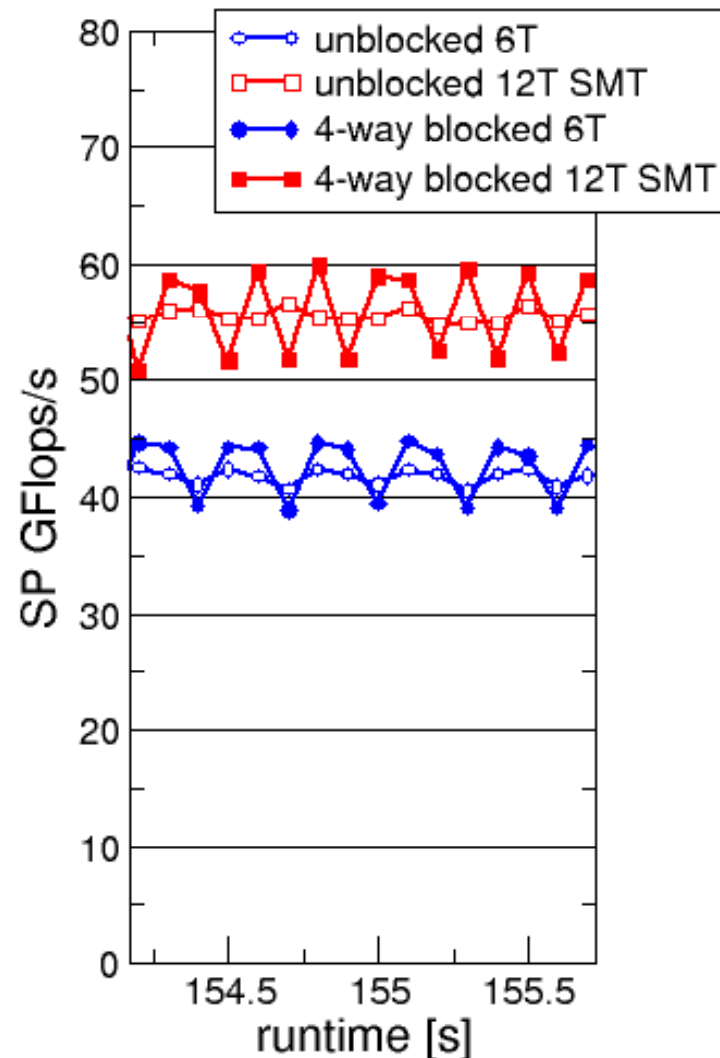
```
likwid-perfctr -c N:0-11 -g FLOPS_DP -s 10
```

- **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
```





- **Marker API: Manually instrument** parts of an application
- **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
```



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
- The code is generated on recompile
- `likwid-perfctr -a` outputs list of groups
- A documentation text is available for every group



- **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



- 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
```

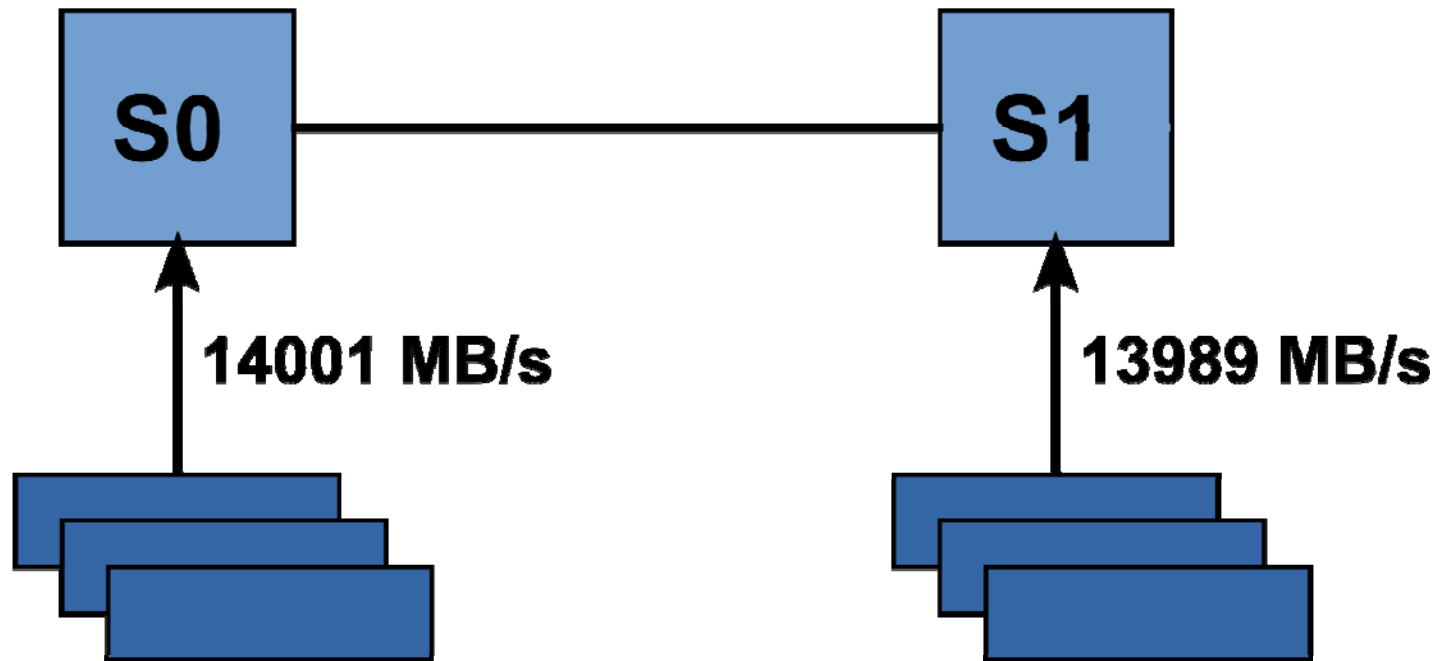
```
STREAMS 2
TYPE DOUBLE
FLOPS 0
BYTES 16
LOOP 32
movaps    FPR1, [STR0 + GPR1 * 8 ]
movaps    FPR2, [STR0 + GPR1 * 8 + 64 ]
movaps    FPR3, [STR0 + GPR1 * 8 + 128 ]
movaps    FPR4, [STR0 + GPR1 * 8 + 192 ]
movaps    [STR1 + GPR1 * 8 ], FPR1
movaps    [STR1 + GPR1 * 8 + 64 ], FPR2
movaps    [STR1 +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



- 2 thread group (2x4 threads) on socket 0/1
- Correct NUMA placement (first touch)

```
likwid-bench -g 2 -i 1000 -t copy -w S0:500MB:4  
-w S1:500MB:4
```

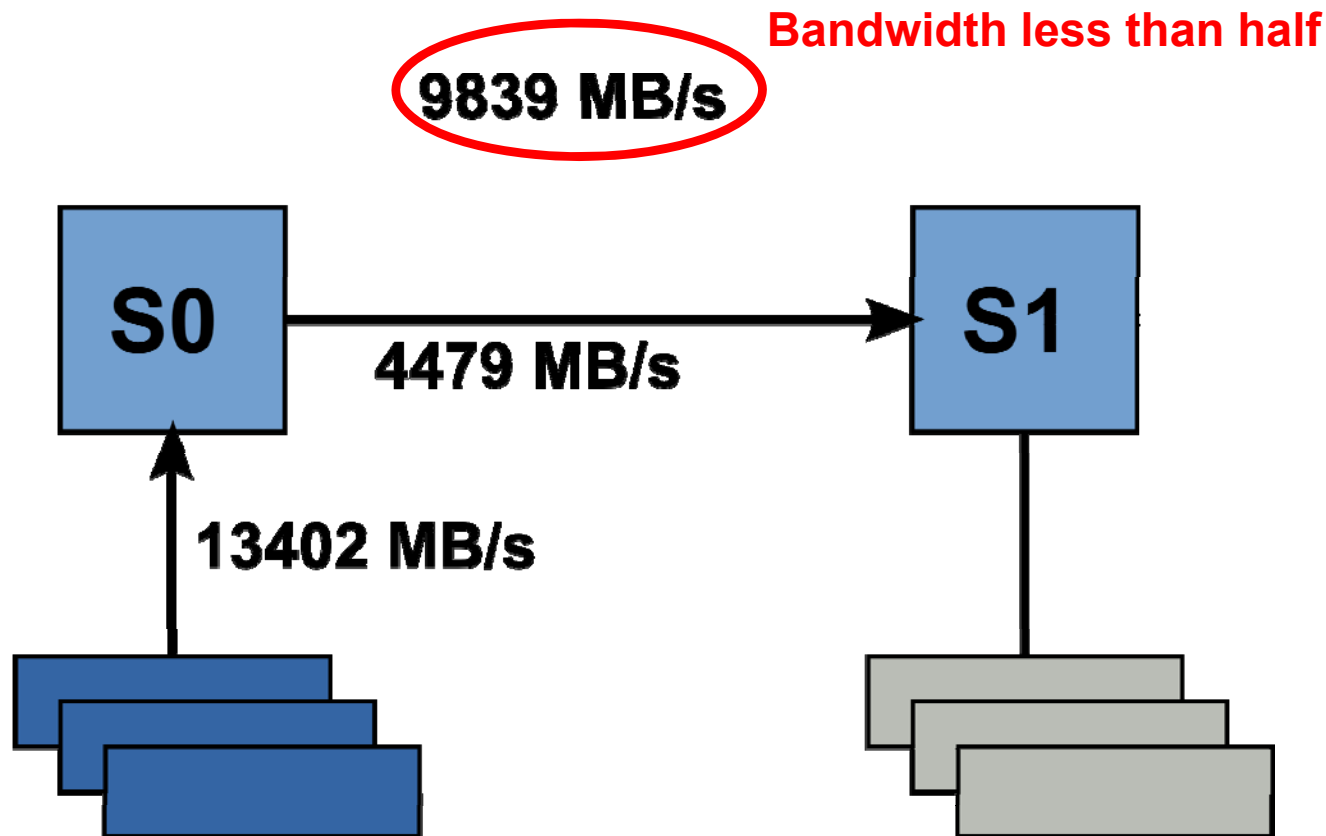
23147 MB/s

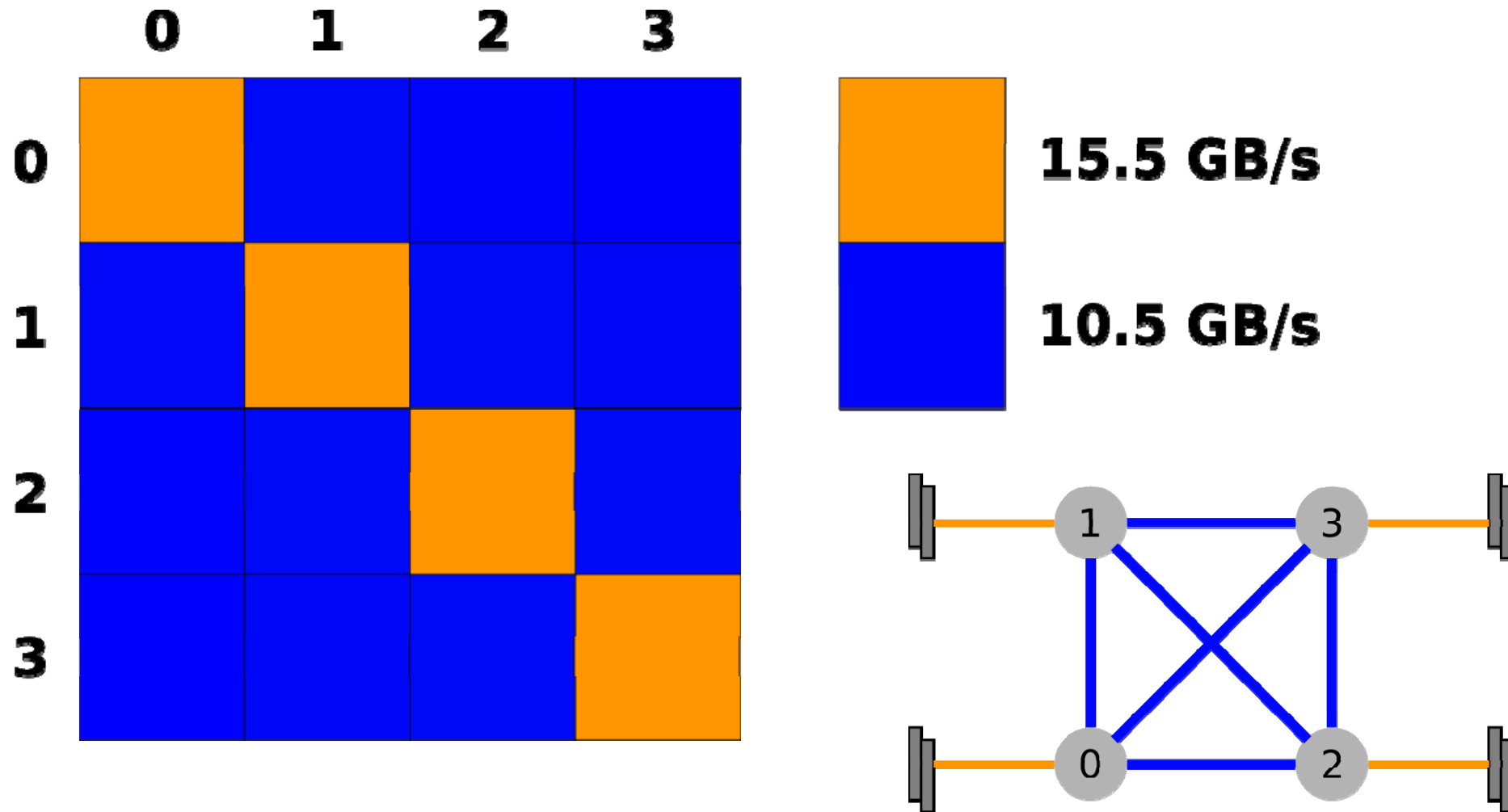




- 2 thread group (2x4 threads) on socket 0/1
- Common problem: memory placed on one socket only

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

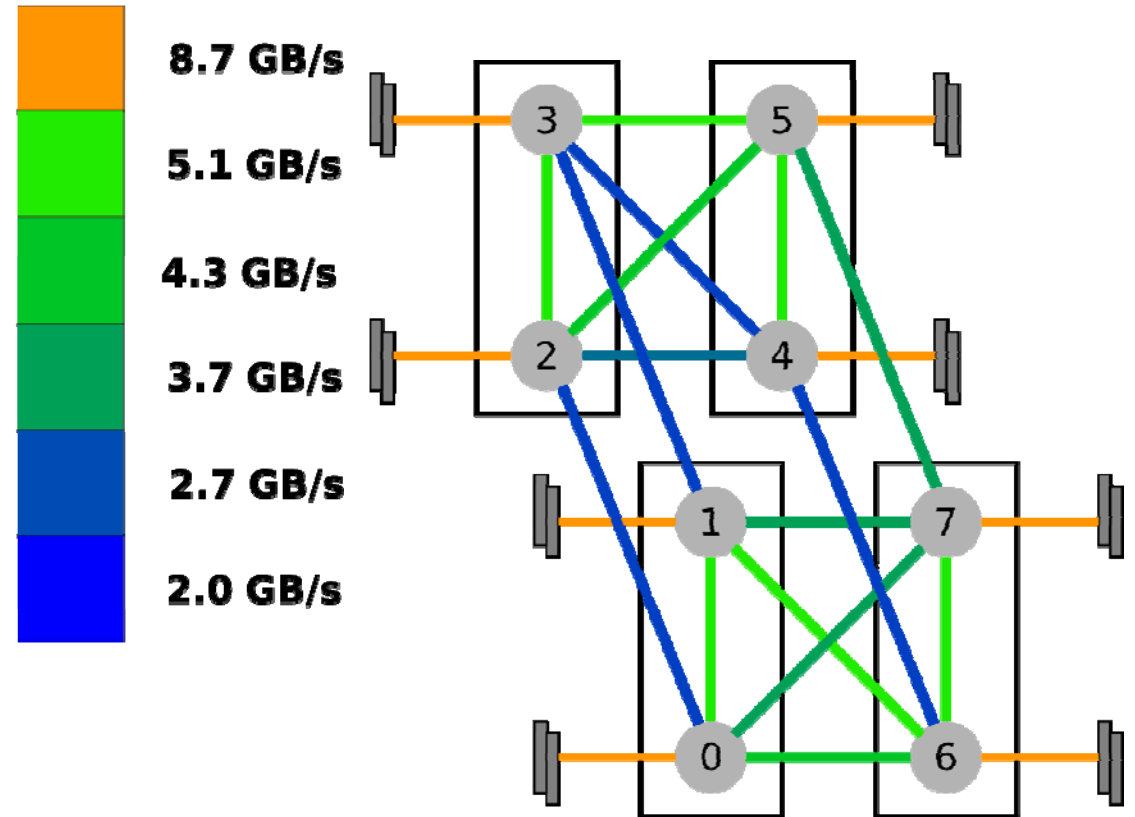
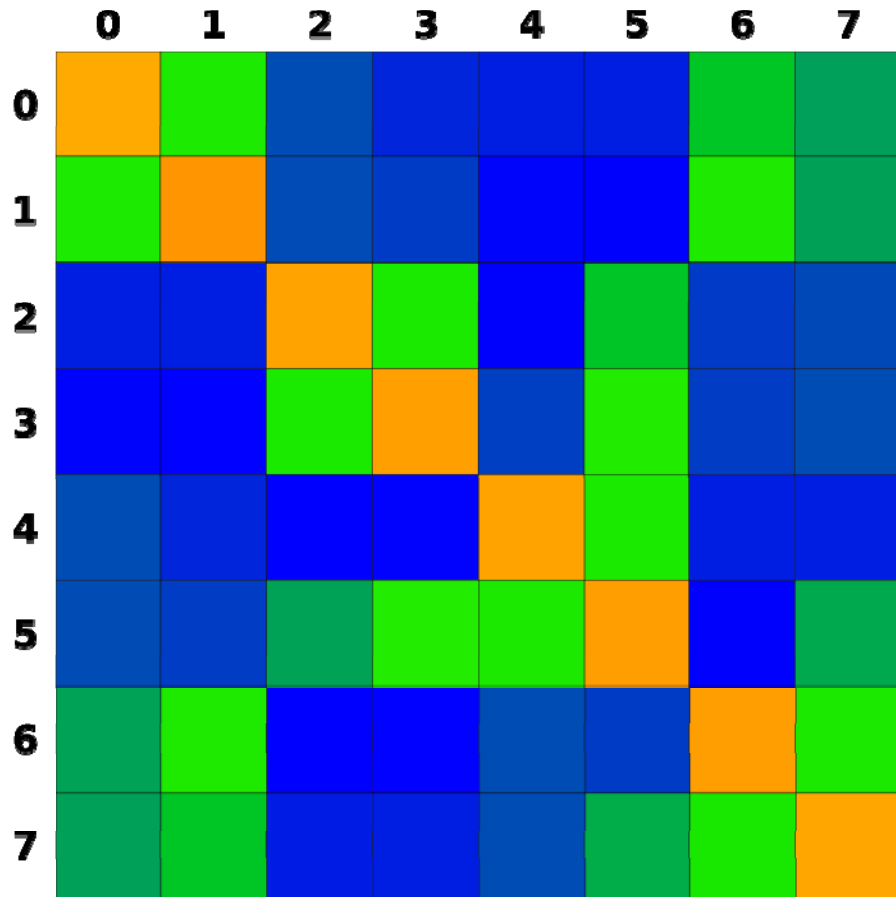




Bandwidth map data measured with likwid-bench. All cores used in one NUMA domain, memory is placed in a different NUMA domain. Test case: simple copy $A(:,) = B(:,)$, large arrays

AMD Magny Cours 4-socket system

Topology at its best?





- **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

Energy consumption: Example

A medical image reconstruction code on Sandy Bridge

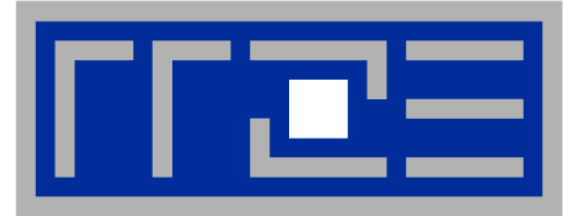


Sandy Bridge EP (8 cores, 2.7 GHz base freq.)

Test case	Runtime	Power	Energy
8 cores, plain C	90.43 s	89.69 Watt	8111 Joule
8 cores, SSE	29.63 s	92.62 Watt	2745 Joule
8 cores (SMT), SSE	22.61 s	102.07 Watt	2308 Joule
8 cores (SMT), AVX	18.42 s	110.80 Watt	2041 Joule

Sandy Bridge UP-Xeon E31280 (4 cores, 3.5 GHz base freq.)

Test case	Runtime	Power	Energy
4 cores, plain C	154.72 s	55.61 Watt	8605 Joule
4 cores, SSE	49.99 s	58.01 Watt	2900 Joule
4 cores (SMT), SSE	-	-	-
4 cores (SMT), AVX	36.73 s	66.43 Watt	2440 Joule



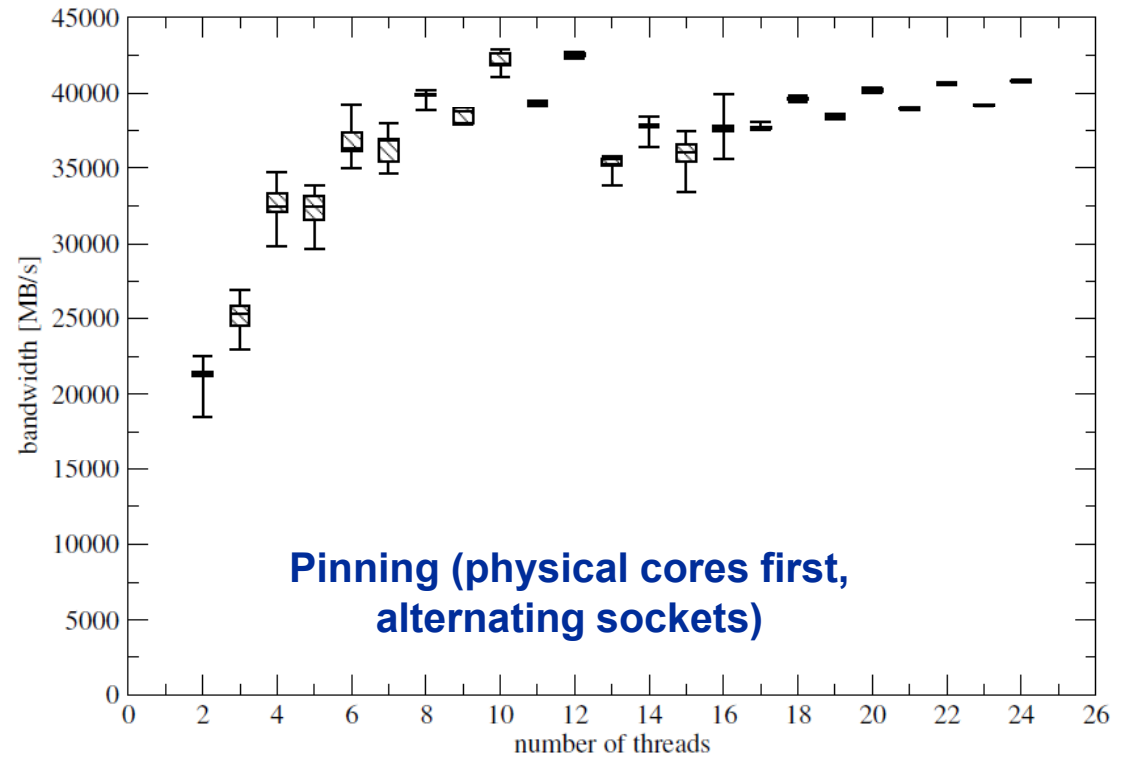
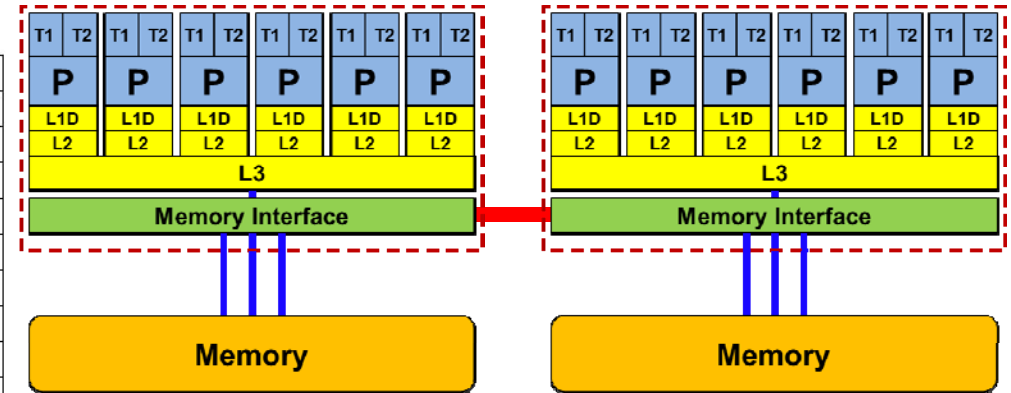
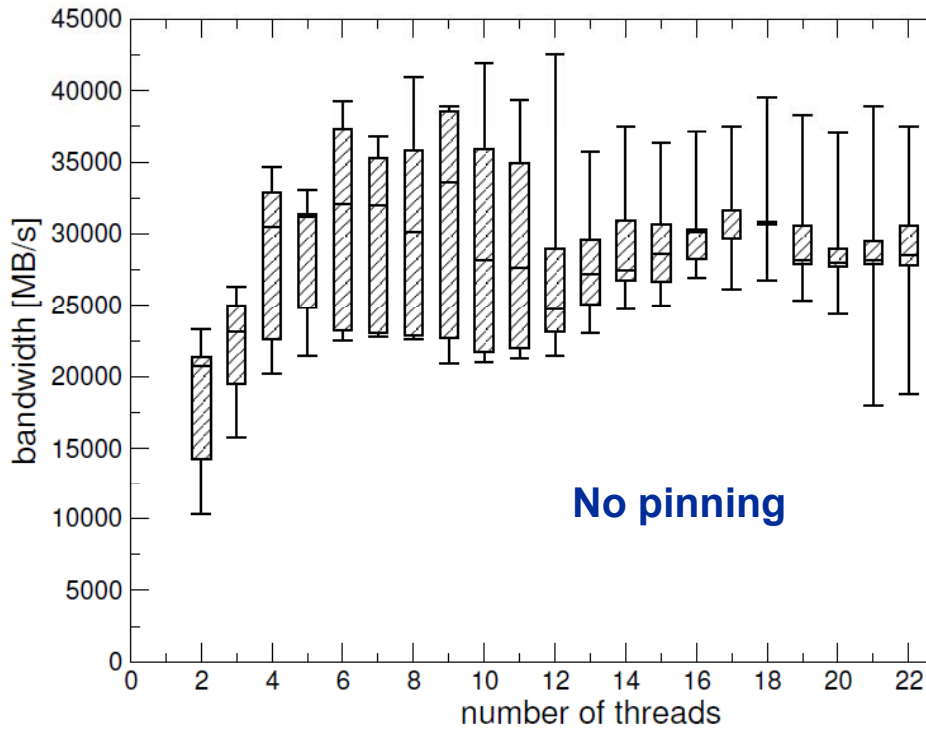
Enforcing thread/process-core affinity under the Linux OS

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

Example: STREAM benchmark on 12-core Intel Westmere:



Anarchy vs. thread pinning



There are several reasons for caring about affinity:

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



Overview

- `taskset [OPTIONS] [MASK | -c LIST] \
[PID | command [args]...]`
- **taskset** binds processes/threads to a set of CPUs. Examples:

```
taskset 0x0006 ./a.out  
taskset -c 4 33187  
mpirun -np 2 taskset -c 0,2 ./a.out # doesn't always work
```
- **Processes/threads can still move within the set!**
- **Alternative: let process/thread bind itself by executing syscall**

```
#include <sched.h>  
int sched_setaffinity(pid_t pid, unsigned int len,  
                     unsigned long *mask);
```
- **Disadvantage: which CPUs should you bind to on a non-exclusive machine?**
- **Still of value on multicore/multisocket cluster nodes, UMA or ccNUMA**



- **Complementary tool: `numactl`**

Example: `numactl --physcpubind=0,1,2,3 command [args]`

Bind process to specified physical core numbers

Example: `numactl --cpunodebind=1 command [args]`

Bind process to specified ccNUMA node(s)

- **Many more options (e.g., interleave memory across nodes)**
 - → see section on ccNUMA optimization
- **Diagnostic command (see earlier):**
`numactl --hardware`
- **Again, this is not suitable for a shared machine**



- **Highly OS-dependent system calls**

- But available on all systems

Linux: `sched_setaffinity()`, PLPA (see below) → `hwloc`

Solaris: `processor_bind()`

Windows: `SetThreadAffinityMask()`

...

- **Support for “semi-automatic” pinning in some compilers/environments**

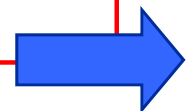
- Intel compilers > V9.1 (`KMP_AFFINITY` environment variable)
- PGI, Pathscale, GNU
- SGI Altix `dp1ace` (works with logical CPU numbers!)
- Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)

- **Affinity awareness in MPI libraries**

- SGI MPT
- OpenMPI
- Intel MPI
- ...

Example for program controlled affinity: Using PLPA under Linux!

SKIPPED





- Pins processes and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library → **binary must be dynamically linked!**
- Can also be used as a superior **replacement for taskset**

- Usage examples:
 - `likwid-pin -t intel -c 0,2,4-6 ./myApp parameters`
 - `likwid-pin -s 3 -c S0:0-3 ./myApp parameters`



- Running the STREAM benchmark with likwid-pin:

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

Main PID always
pinned

Skip shepherd
thread

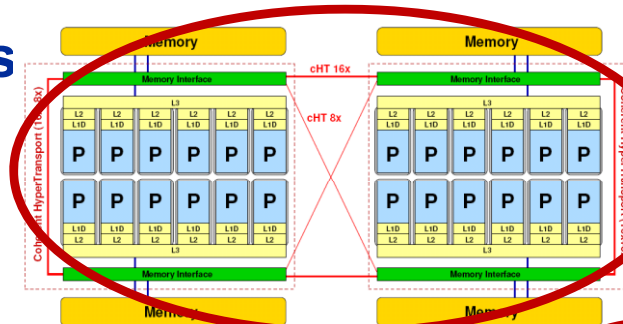
Pin all spawned
threads in turn



- Possible unit prefixes

N

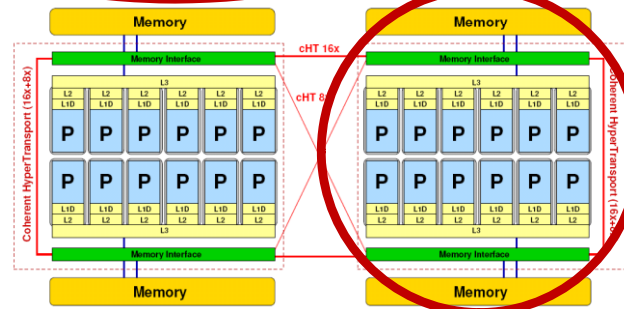
node



Default if -c is not specified!

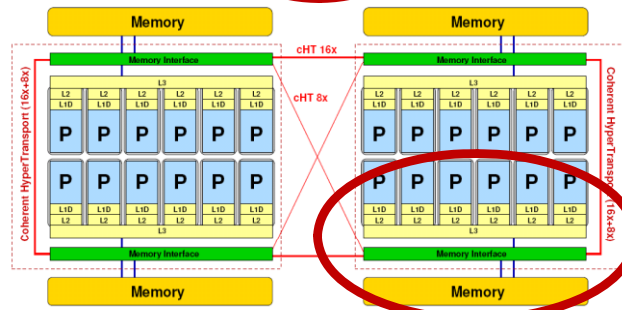
S

socket



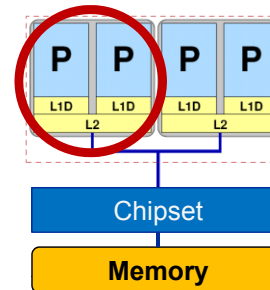
M

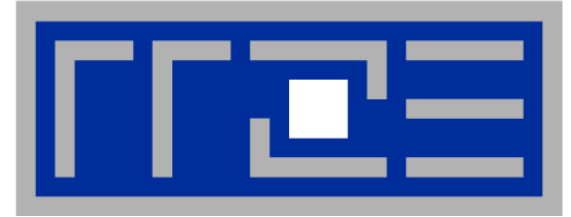
NUMA domain



C

outer level cache group





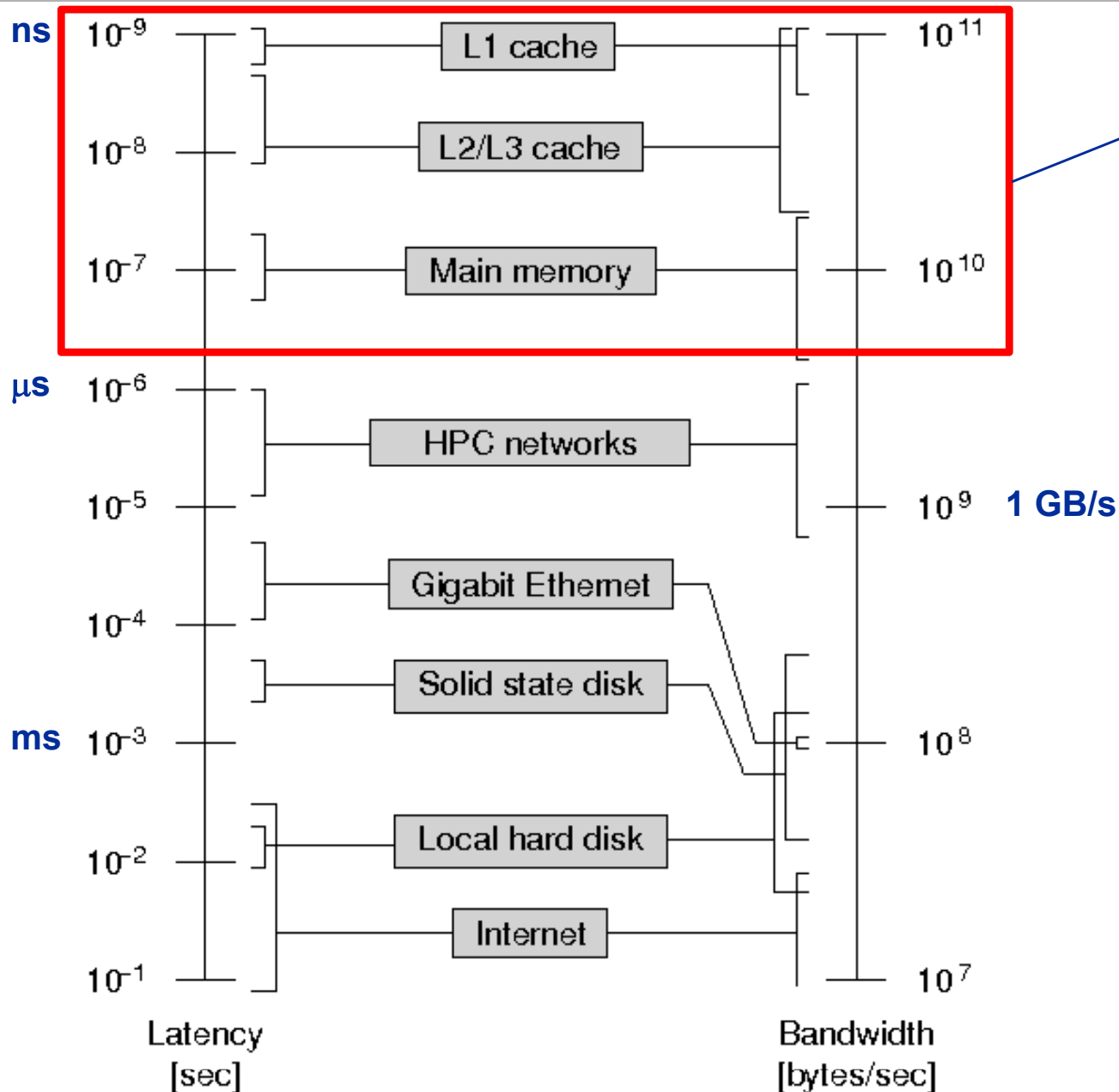
Data access on modern processors

Characterization of memory hierarchies

Balance analysis and light speed estimates

Data access optimization

Latency and bandwidth in modern computer environments



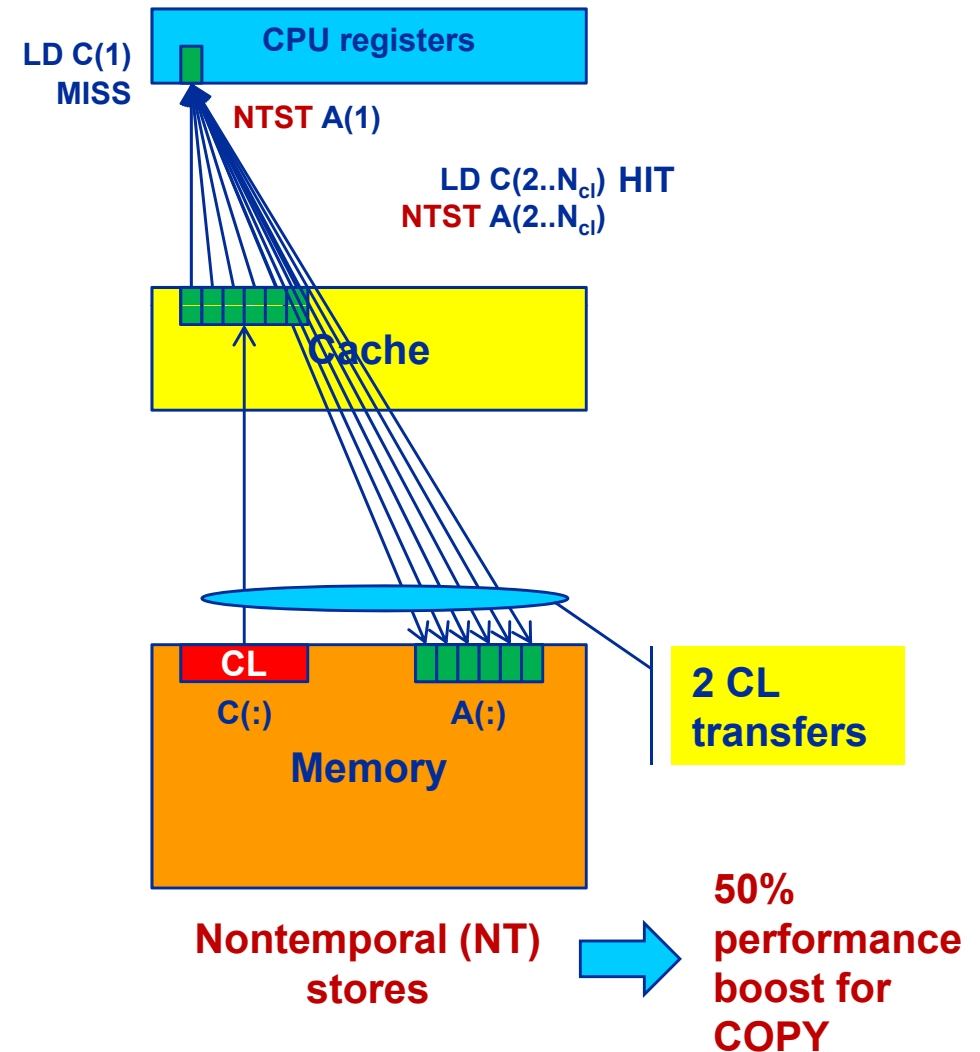
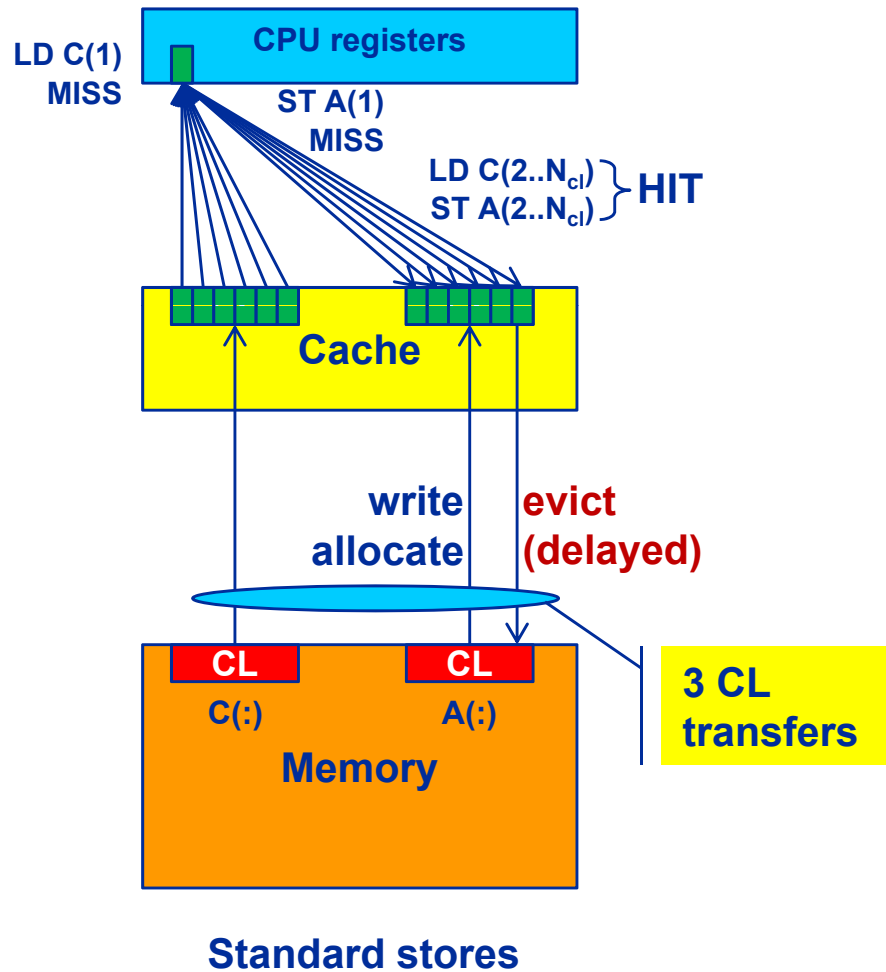
We care about this region today

Avoiding slow data paths is the key to most performance optimizations!

Interlude: Data transfers in a memory hierarchy



- How does data travel from memory to the CPU and back?
- Example: Array copy $A(:) = C(:)$





Simple streaming benchmark:

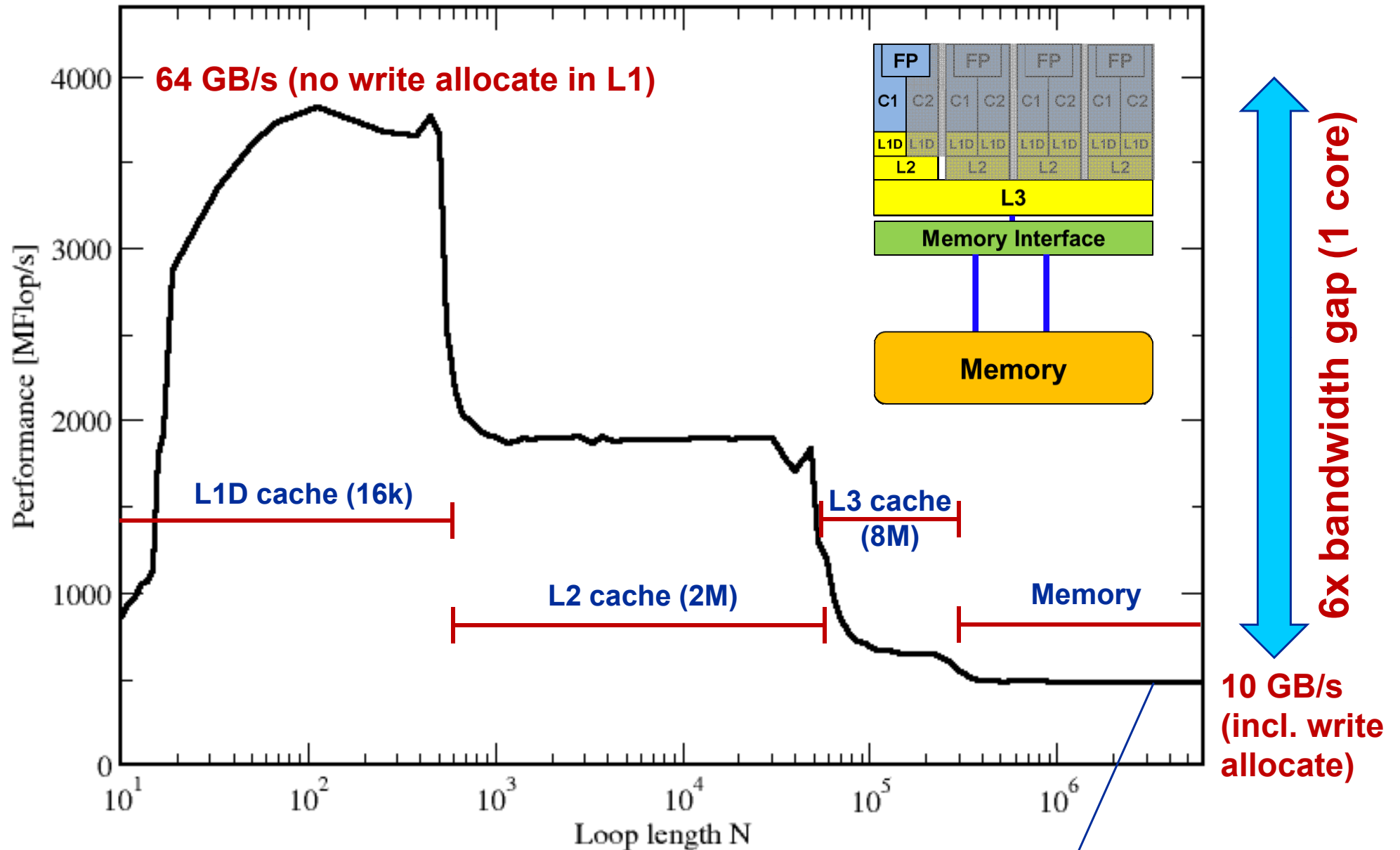
```
double precision, dimension(N) :: A,B,C,D  
A=1.d0; B=A; C=A; D=A
```

```
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!**

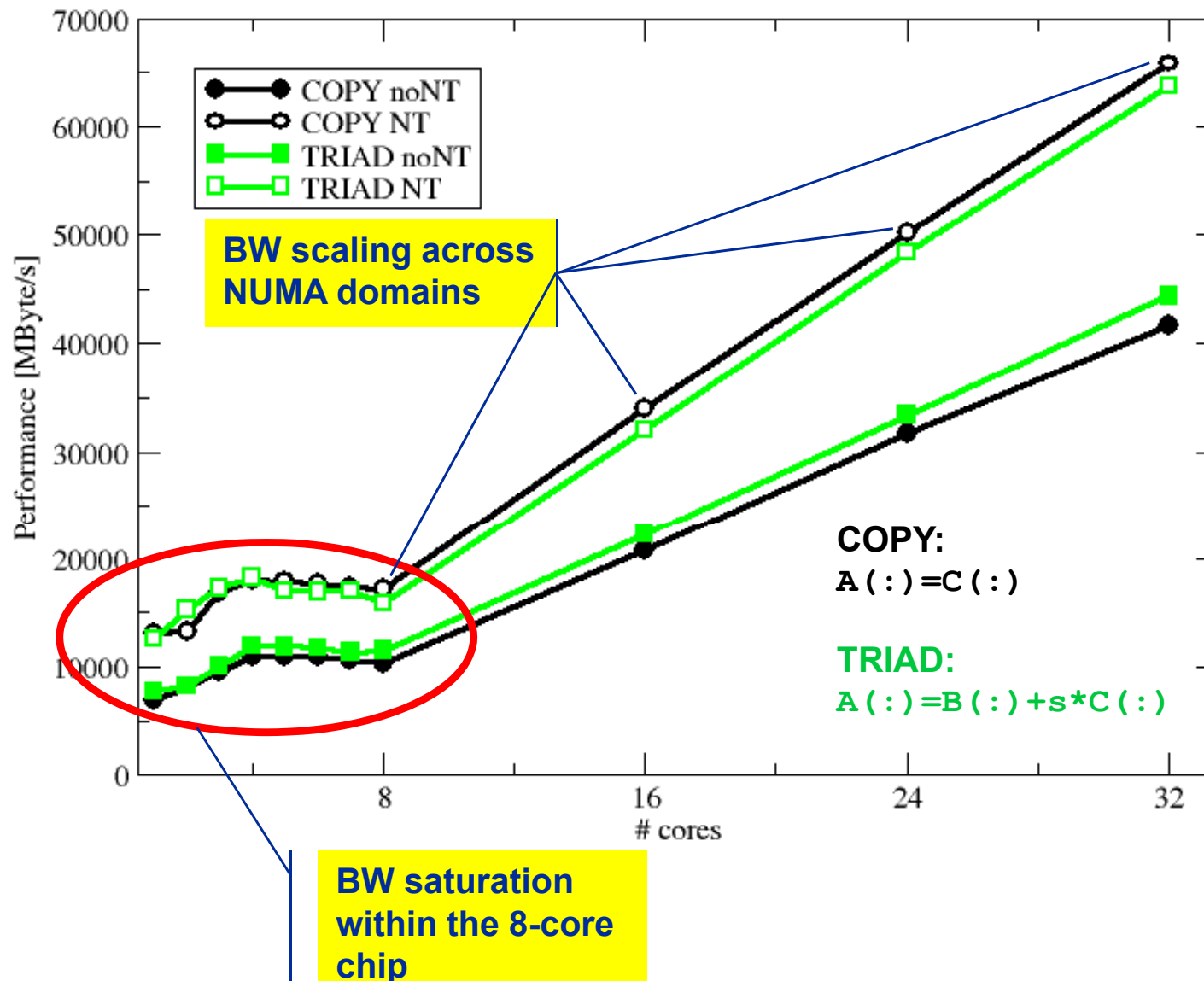
A(:)=B(:)+C(:)*D(:) on one Interlagos core



Is this the limit???

STREAM benchmarks:

Memory bandwidth on AMD Interlagos node



- **STREAM is the “standard” for memory BW comparisons**
- **NT store variants save write allocate on stores → 50% boost for copy, 33% for TRIAD**
- **STREAM BW is practical limit for all codes**



- The **machine balance** for data memory access of a specific computer is given by (architectural limitation)

$$B_m = \frac{b_s \text{ [words/s]}}{P_{\max} \text{ [flops/s]}}$$

- **Bandwidth:**

1 W = 8 bytes = 64 bits

b_s = achievable bandwidth over the slowest data path

Floating point peak:

P_{\max}

- **Machine Balance** = How many input operands can be delivered for each FP operation?

- **Typical values (main memory):**

AMD Interlagos (2.3 GHz): $B_m = \{(17/8) \text{ GW/s}\} / \{4 \times 2.3 \times 8 \text{ GFlop/s}\} \sim \mathbf{0.029 \text{ W/F}}$

Intel Sandy Bridge EP (2.7 GHz): $\sim \mathbf{0.025 \text{ W/F}}$

NEC SX9 (vector): $\sim \mathbf{0.3 \text{ W/F}}$

nVIDIA GTX480 $\sim \mathbf{0.026 \text{ W/F}}$



Data path	Balance B_M [W/F]
Cache	0.5 – 1.0
Machine (main memory)	0.01 – 0.5
Interconnect (Infiniband)	0.001 – 0.002
Interconnect (GBit ethernet)	0.0001 – 0.0007
Disk (or disk subsystem)	0.0001 – 0.001

Double precision: W \leftrightarrow 64-Bit

$1/B_M$ = “Computational Intensity”: How many FP ops can be performed before FP performance becomes a bottleneck?



- B_M tells us what the hardware can deliver at most
- **Code balance** (B_C) quantifies the requirements of the code:

$$B_c = \frac{\text{data transfer (LD/ST) [words]}}{\text{arithmetic operations [flops]}}$$

- **Expected fraction of peak performance** („lightspeed“):
 $l = 1 \rightarrow$ code is not limited by bandwidth

$$l = \min\left(1, \frac{B_m}{B_c}\right)$$

This is what we get

This is what we need

- **Lightspeed** for absolute performance: (P_{\max} : “applicable” peak performance)

$$P = l \cdot P_{\max} = \min\left(P_{\max}, \frac{b_s}{B_C}\right)$$

- **Example: Vector triad** $A(:,) = B(:,) + C(:,) * D(:,)$ on 2.3 GHz Interlagos
 - $B_c = (4+1) \text{ Words} / 2 \text{ Flops} = 2.5 \text{ W/F}$ (including write allocate)

$B_m/B_c = 0.029/2.5 = 0.012$, i.e. **1.2 % of peak performance (~1.7 GF/s)**



- **The balance metric formalism is based on some (crucial) assumptions:**
 - The code makes **balanced use of MULT and ADD** operation. For others (e.g. $A=B+C$) the peak performance input parameter P_{\max} has to be adjusted (e.g. $P_{\max} \rightarrow P_{\max}/2$)
 - **Attainable bandwidth of code = input parameter!** Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications.
 - Definition is based on 64-bit arithmetic but can easily be adjusted, e.g. for 32-bit
 - **Data transfer and arithmetic overlap perfectly!**
 - **Slowest data path is modeled only;** all others are assumed to be infinitely fast
 - Latency effects are ignored, i.e. **perfect streaming mode**



- Diffusion equation in 2D $\frac{\partial \Phi}{\partial t} = \Delta \Phi$

- Stationary solution** with Dirichlet boundary conditions using Jacobi iteration scheme can be obtained with:

```
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
integer :: t0,t1
t0 = 0 ; t1 = 1
do it = 1,itmax      ! choose suitable number of sweeps
  do k = 1,kmax
    do i = 1,imax
      ! four flops, one store, four loads
      phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
                     + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
    enddo
  enddo
  ! swap arrays
  i = t0 ; t0=t1 ; t1=i
enddo
```

Reuse when computing
phi(i+2,k,t1)

Balance (crude estimate incl. write allocate):

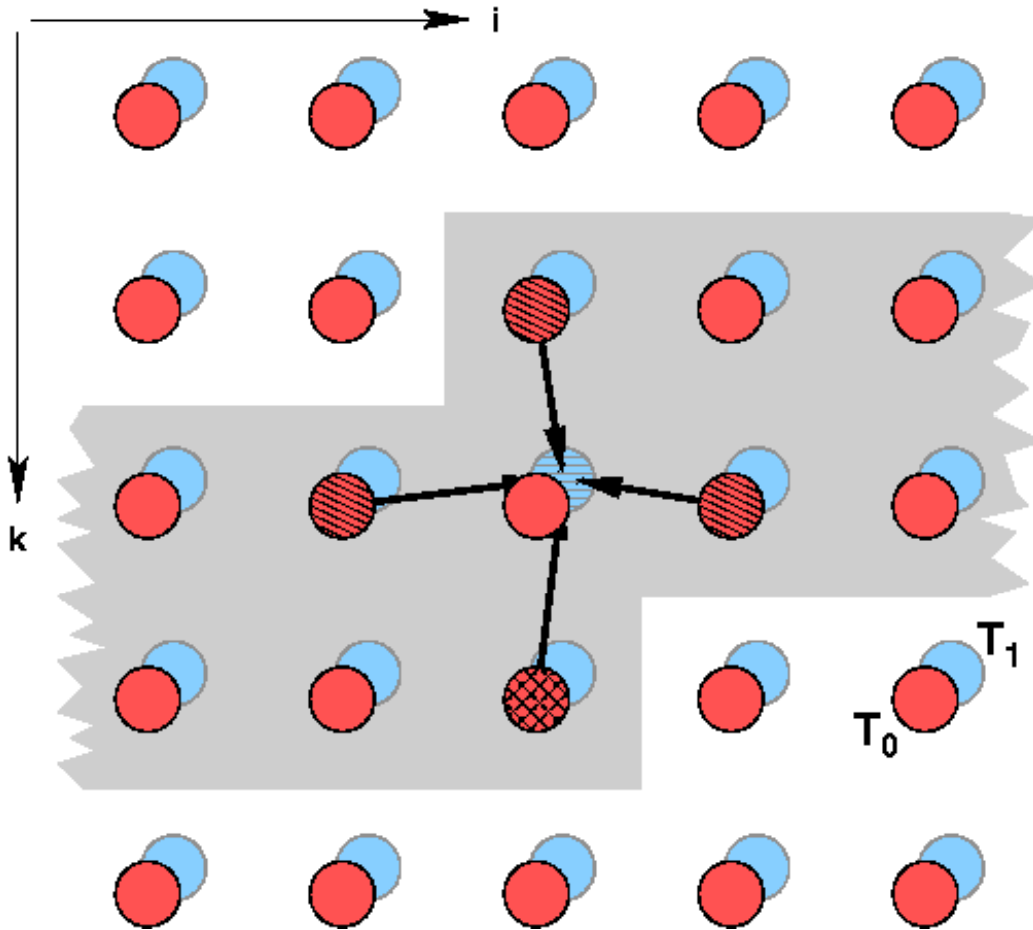
phi(:, :, t0): 3 LD +
phi(:, :, t1): 1 ST+ 1LD

→ $B_c = 5 W / 4 FLOPs = 1.25 W / F$

WRITE ALLOCATE:
LD + ST phi(i,k,t1)



- Modern cache subsystems may further reduce memory traffic



If cache is large enough to hold at least 2 rows (shaded region): Each $\phi(i, k, t_0)$ is loaded once from main memory and reused 3 times from cache:

$$\phi(i, k, t_0): 1 \text{ LD} + \phi(i, k, t_1): 1 \text{ ST} + 1 \text{ LD} \\ \rightarrow B_C = 3W / 4F = 0.75W / F$$

If cache is large enough to hold at least one row $\phi(i, k-1, t_0)$ needs to be reloaded:

$$\phi(i, k, t_0): 2 \text{ LD} + \phi(i, k, t_1): 1 \text{ ST} + 1 \text{ LD} \\ \rightarrow B_C = 4W / 4F = 1.0W / F$$

Beyond that:

$$\phi(i, k, t_0): 2 \text{ LD} + \phi(i, k, t_1): 1 \text{ ST} + 1 \text{ LD} \\ \rightarrow B_C = 5W / 4F = 1.25W / F$$



- **Alternative implementation (“Macho FLOP version”)**

```
do k = 1, kmax
  do i = 1, imax
    phi(i, k, t1) = 0.25 * phi(i+1, k, t0) + 0.25 * phi(i-1, k, t0)
                  + 0.25 * phi(i, k+1, t0) + 0.25 * phi(i, k-1, t0)
  enddo
enddo
```

- **MFlops/sec increases by 7/4 but time to solution remains the same**
- **Better metric (for many iterative stencil schemes):**
Lattice Site Updates per Second (LUPs/sec)

2D Jacobi example: Compute LUPs/sec metric via

$$P[MLUPS / s] = \frac{it_{\max} \cdot i_{\max} \cdot k_{\max}}{T_{\text{wall}}}$$



- **3D sweep:**

```
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      phi(i,j,k,t1) = oos * (phi(i-1,j,k,t0)+phi(i+1,j,k,t0) &
        + phi(i,j-1,k,t0)+phi(i,j+1,k,t0) &
        + phi(i,j,k-1,t0)+phi(i,j,k+1,t0))
    enddo
  enddo
enddo
```

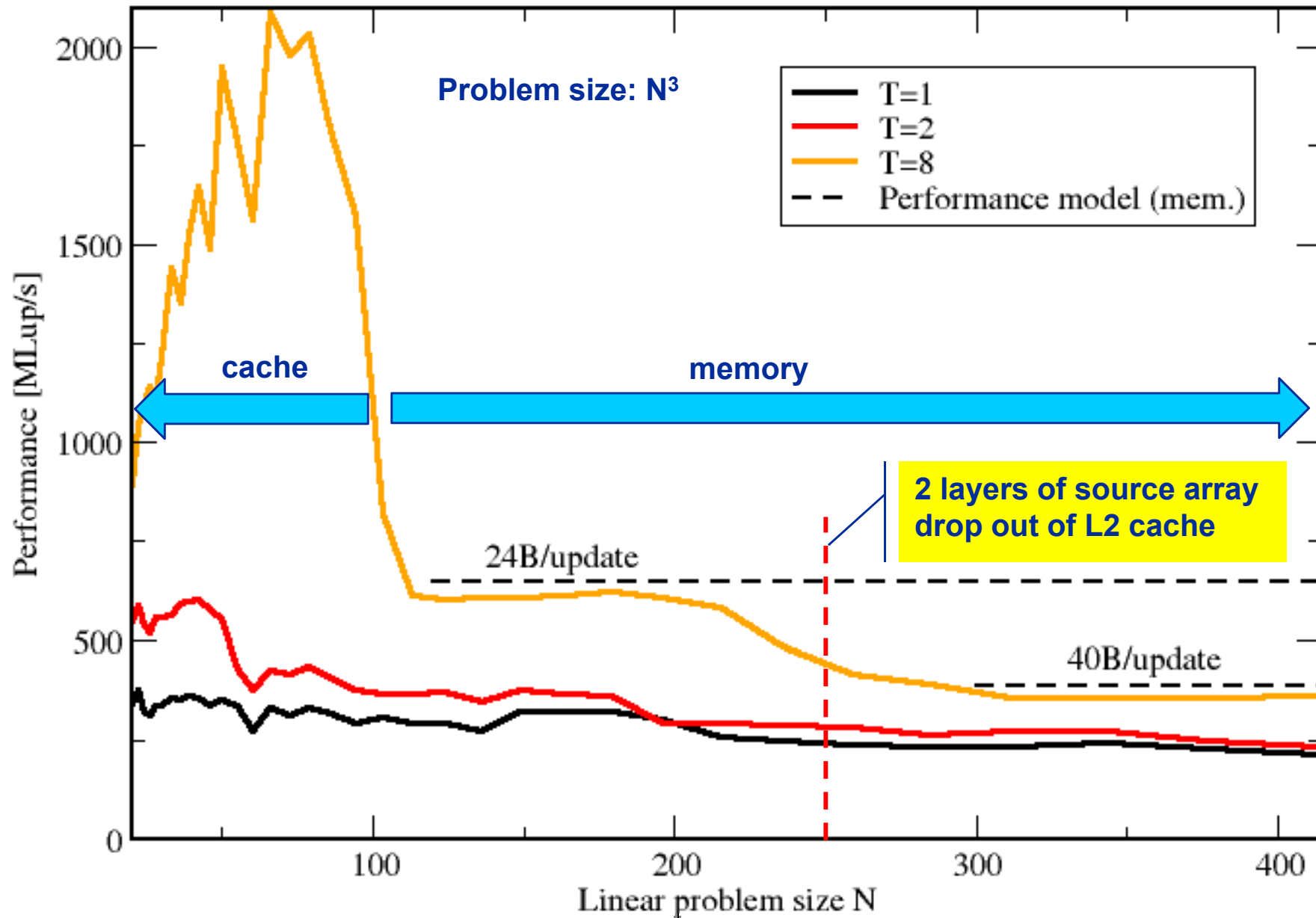
- **Best case balance:** 1 LD $\phi(i,j,k+1,t0)$
1 ST + 1 write allocate $\phi(i,j,k,t1)$
6 flops

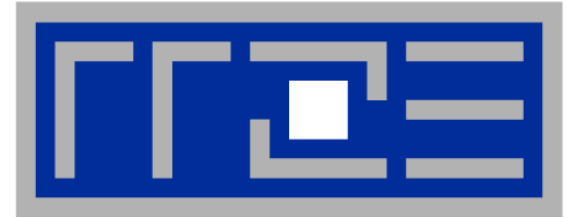
→ $B_c = 0.5 W/F$

- If 2-layer condition does not hold but 2 rows fit: $B_c = 5/6 W/F$
- Worst case (2 rows do not fit): $B_c = 7/6 W/F$

3D Jacobi solver

Performance of vanilla code on one Interlagos chip (8 cores)





Data Access Optimizations

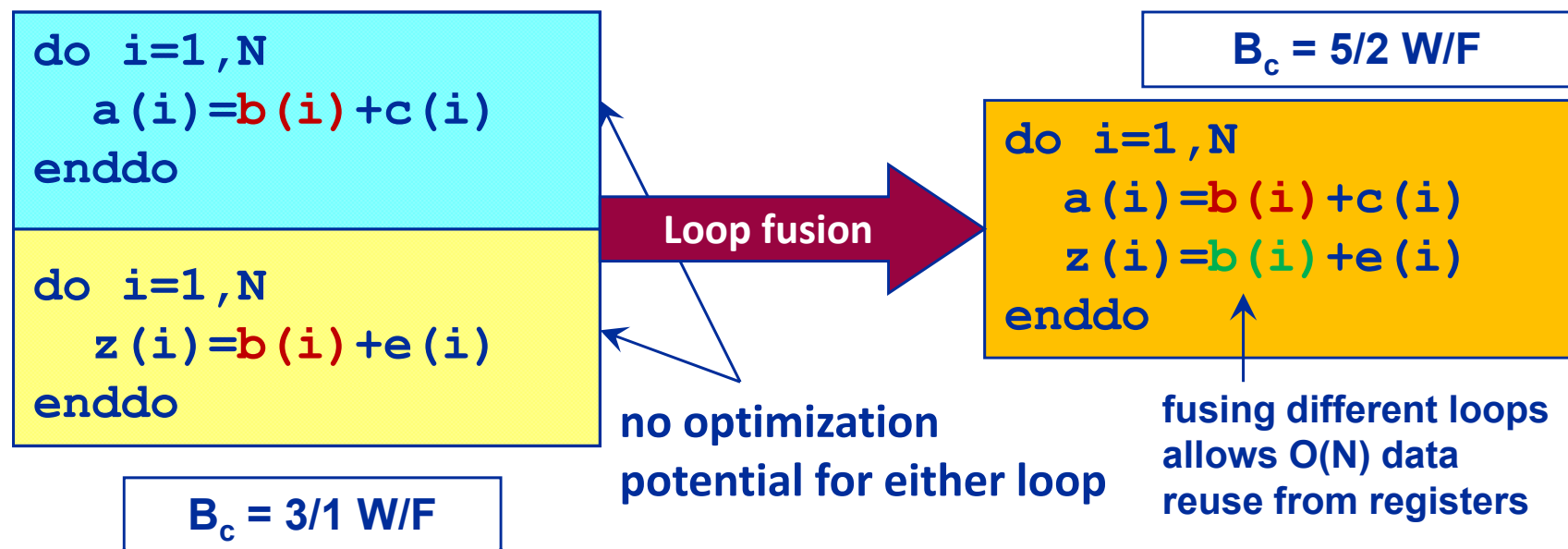
General considerations

Case study: Optimizing a Jacobi solver



Case 1: $O(N)/O(N)$ Algorithms

- $O(N)$ arithmetic operations vs. $O(N)$ data access operations
- Examples: Scalar product, vector addition, sparse MVM etc.
- Performance **limited by memory BW** for large N (“memory bound”)
- Limited optimization potential for single loops
 - ...at most a constant factor for multi-loop operations
- Example: successive vector additions

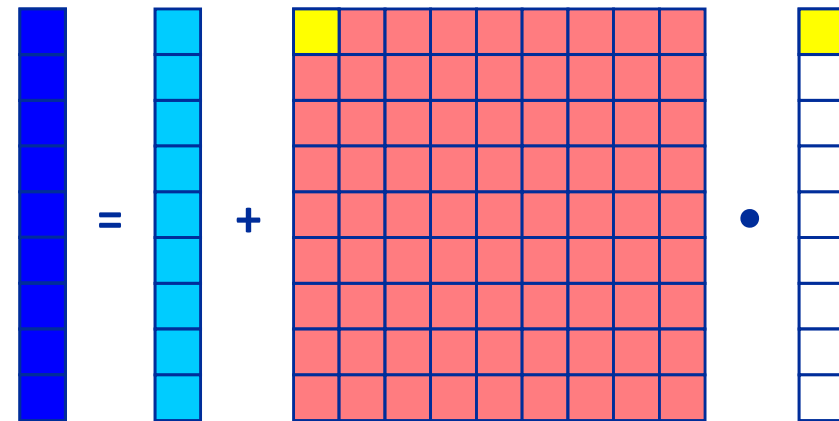




Case 2: $O(N^2)/O(N^2)$ algorithms

- Examples: dense matrix-vector multiply, matrix addition, dense matrix transposition etc.
 - Nested loops
- Memory bound for large N
- Some optimization potential (at most constant factor)
 - Can often enhance code balance by **outer loop unrolling** or **spatial blocking**
- Example: dense matrix-vector multiplication

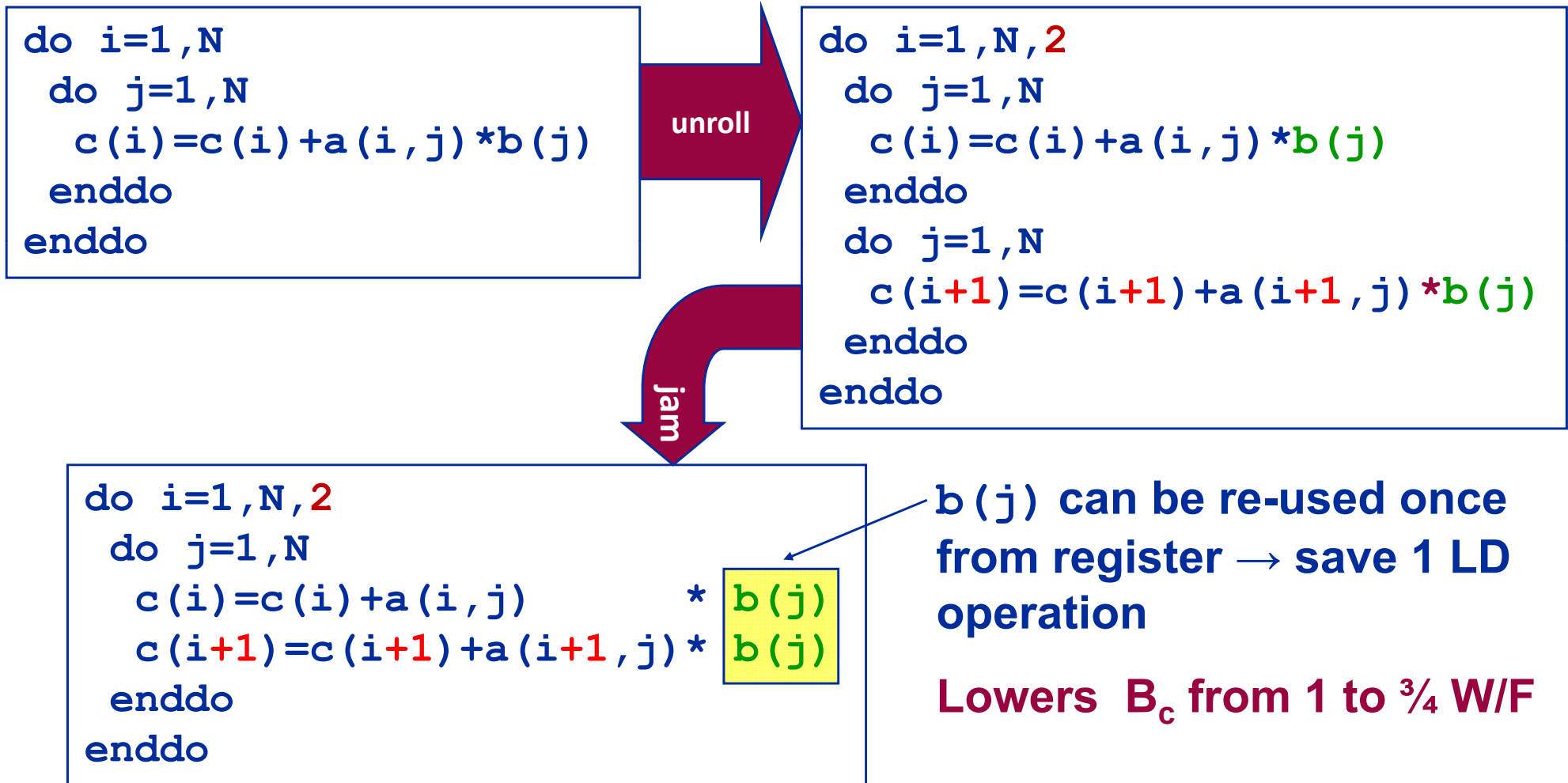
```
do i=1,N
  do j=1,N
    c(i)=c(i)+a(i,j)*b(j)
  enddo
enddo
```



Naïve version loads $b []$ N times!



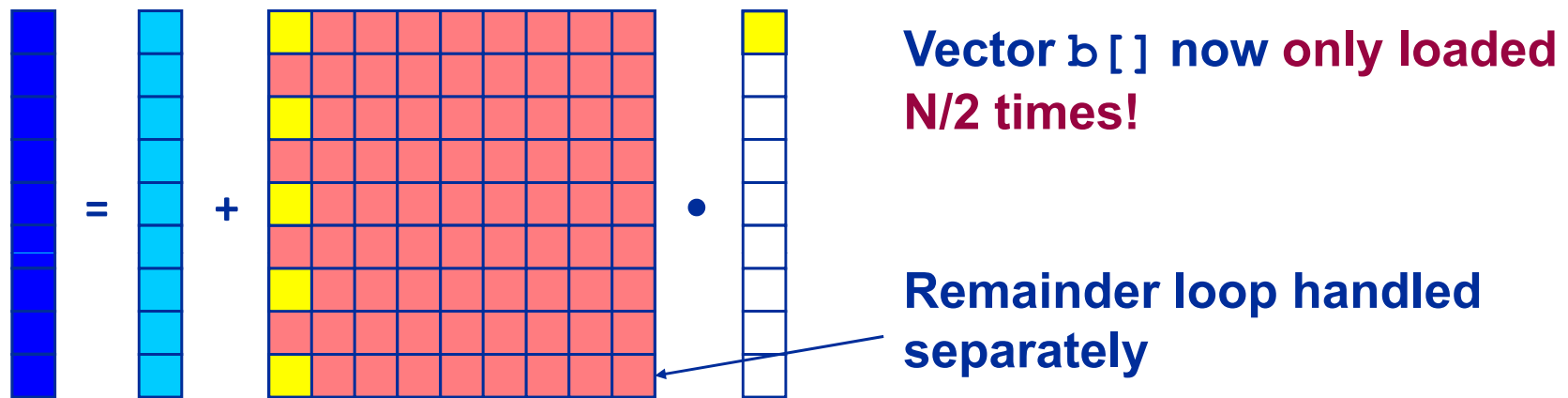
- **$O(N^2)/O(N^2)$ algorithms cont'd**
 - “Unroll & jam” optimization (or “outer loop unrolling”)



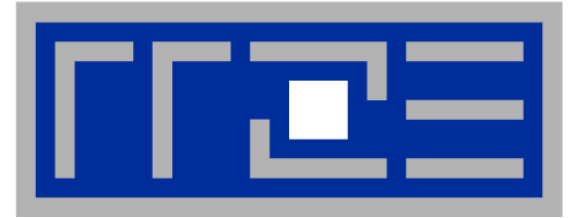


■ $O(N^2)/O(N^2)$ algorithms cont'd

- Data access pattern for 2-way unrolled dense MVM:



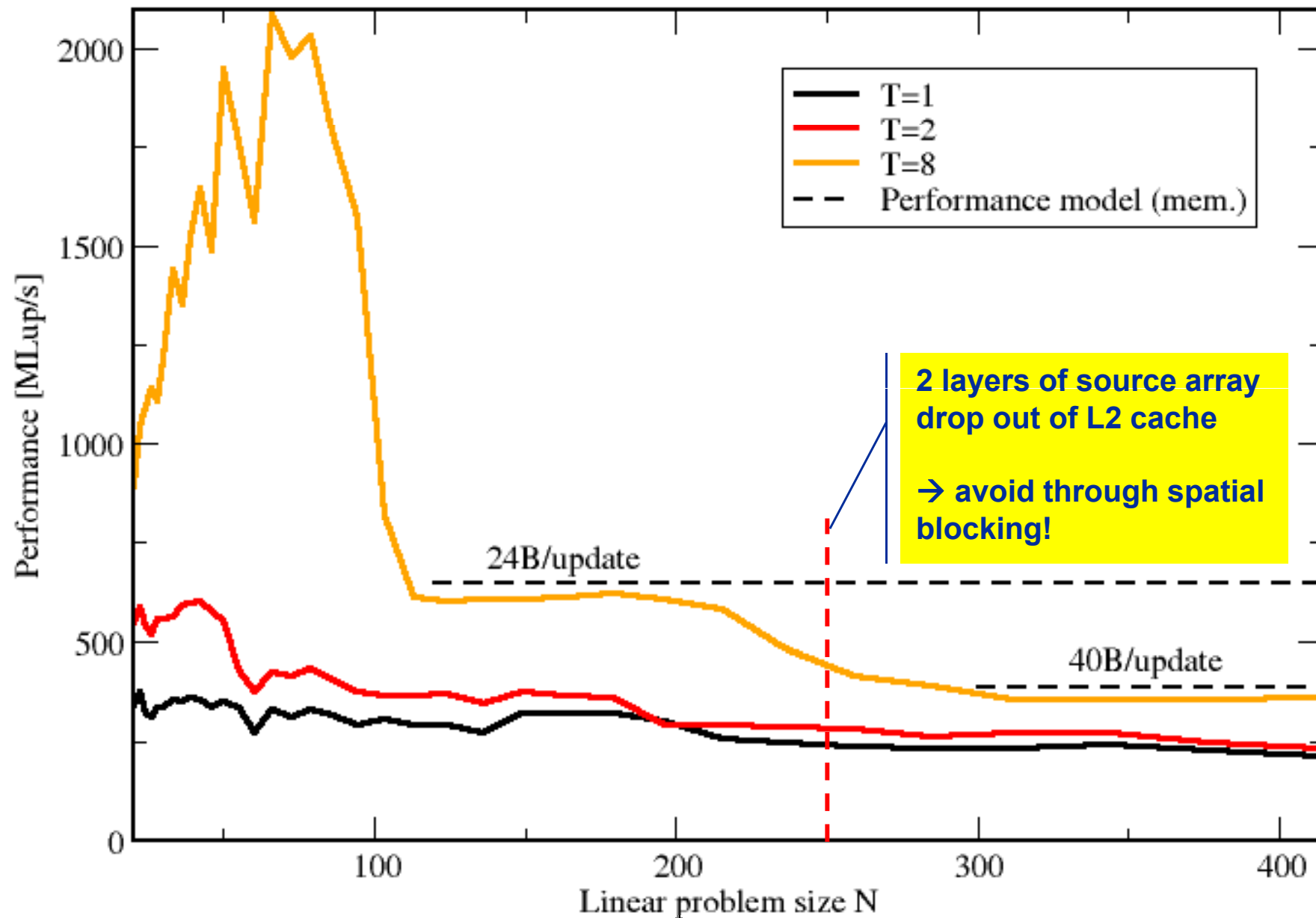
- Data transfers can further be reduced by more aggressive unrolling (i.e., m -way instead of 2-way)
- Significant code bloat (try to use compiler directives if possible)
 - Main memory limit: $b[]$ only be loaded once from memory ($B_c \approx \frac{1}{2} W/F$) (can be achieved by high unrolling OR large outer level caches)
 - **Outer loop unrolling can also be beneficial to reduce traffic within caches!**
 - Beware: CPU registers are a limited resource
 - Excessive unrolling can cause register spills to memory



**Case study:
3D Jacobi solver**

Spatial blocking for improved cache utilization

Remember the 3D Jacobi solver?

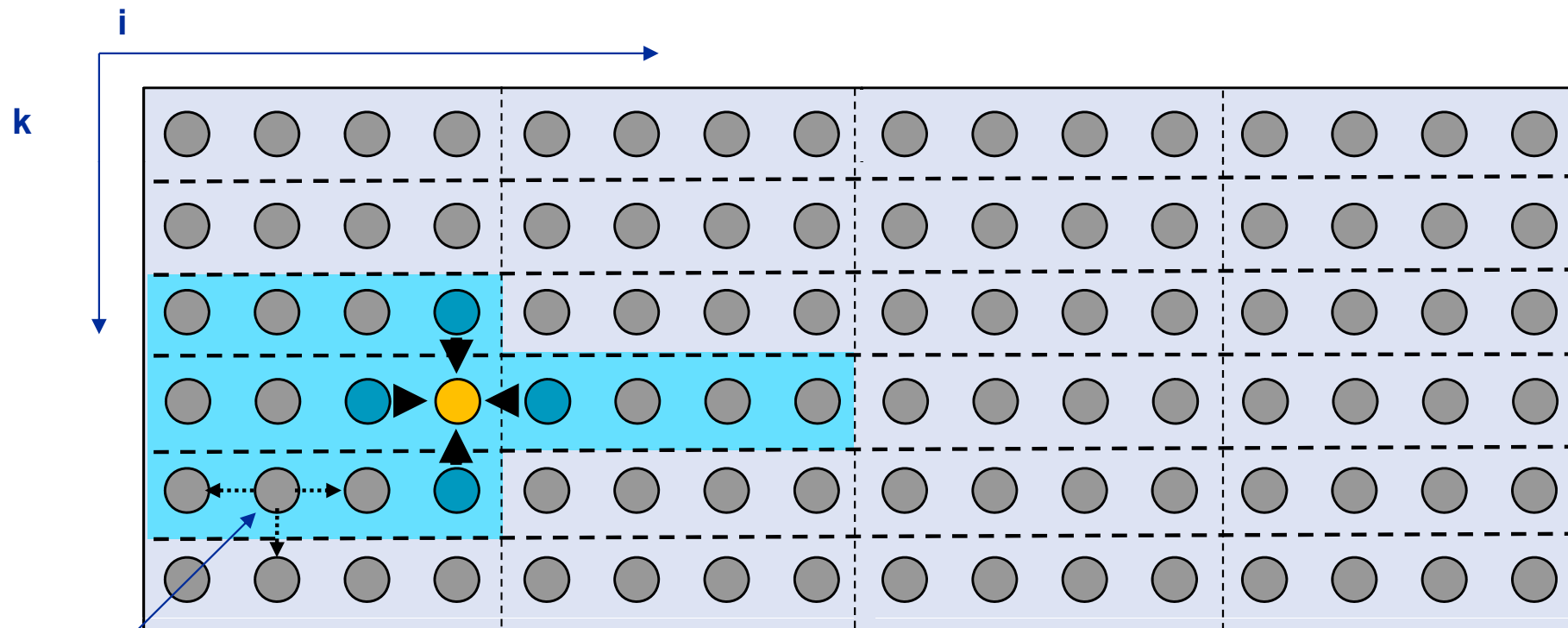


Jacobi iteration (2D): No spatial Blocking



- **Assumptions:**

- cache can hold 32 elements (16 for each array)
- Cache line size is 4 elements
- Perfect eviction strategy for source array



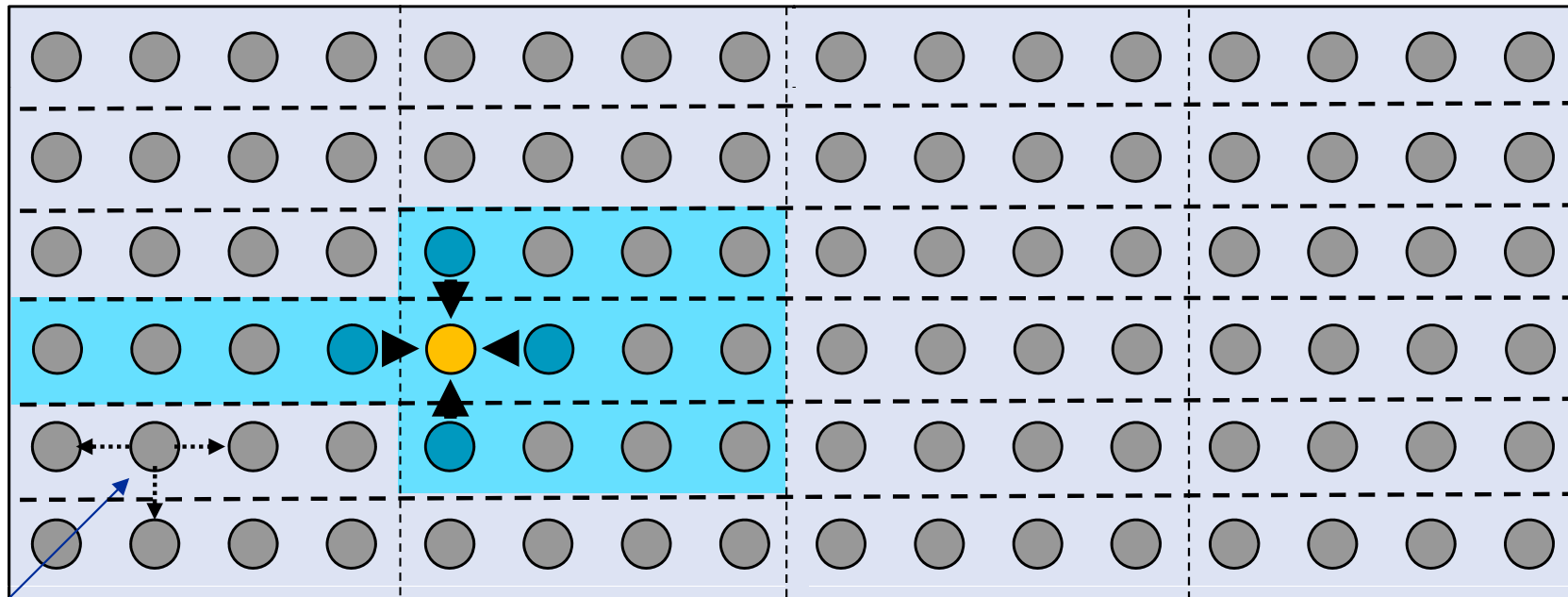
This element is needed for three more updates

Jacobi iteration (2D): No spatial blocking



- **Assumptions:**

- cache can hold 32 elements (16 for each array)
- Cache line size is 4 elements
- Perfect eviction strategy for source array

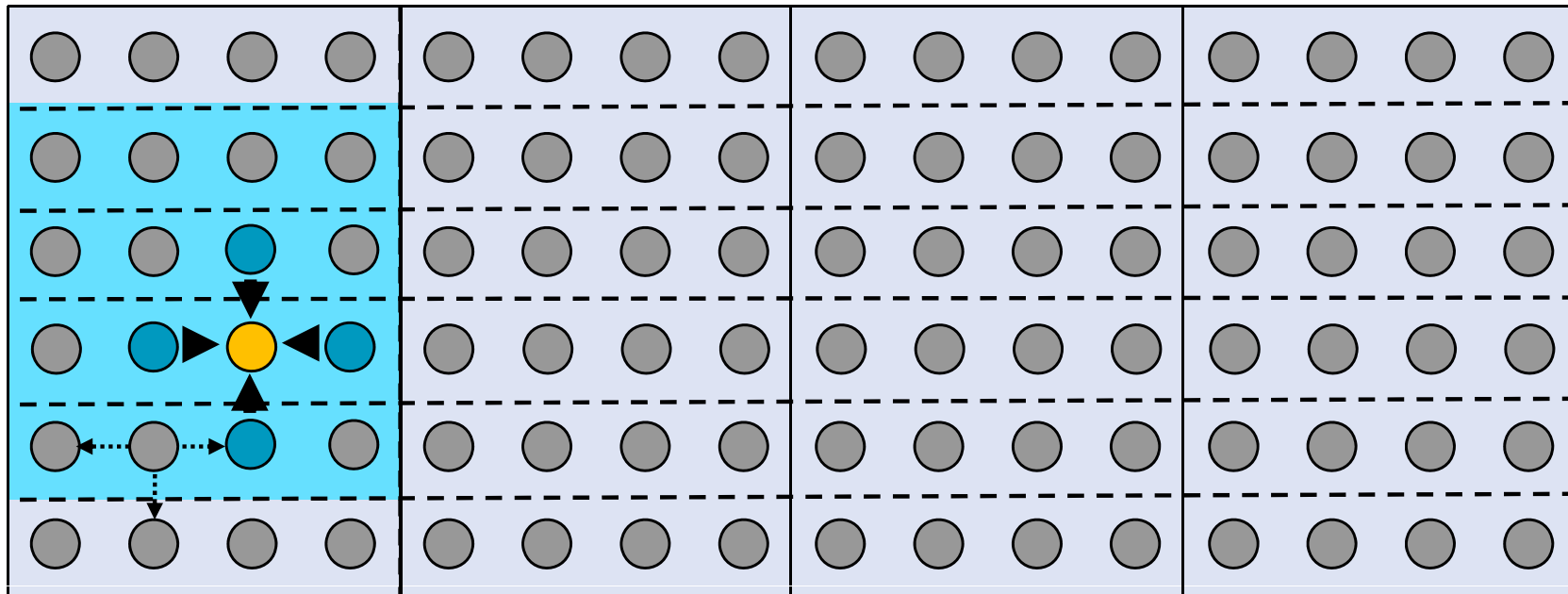


This element is needed for three more updates but has been evicted

Jacobi iteration (2D): Spatial Blocking



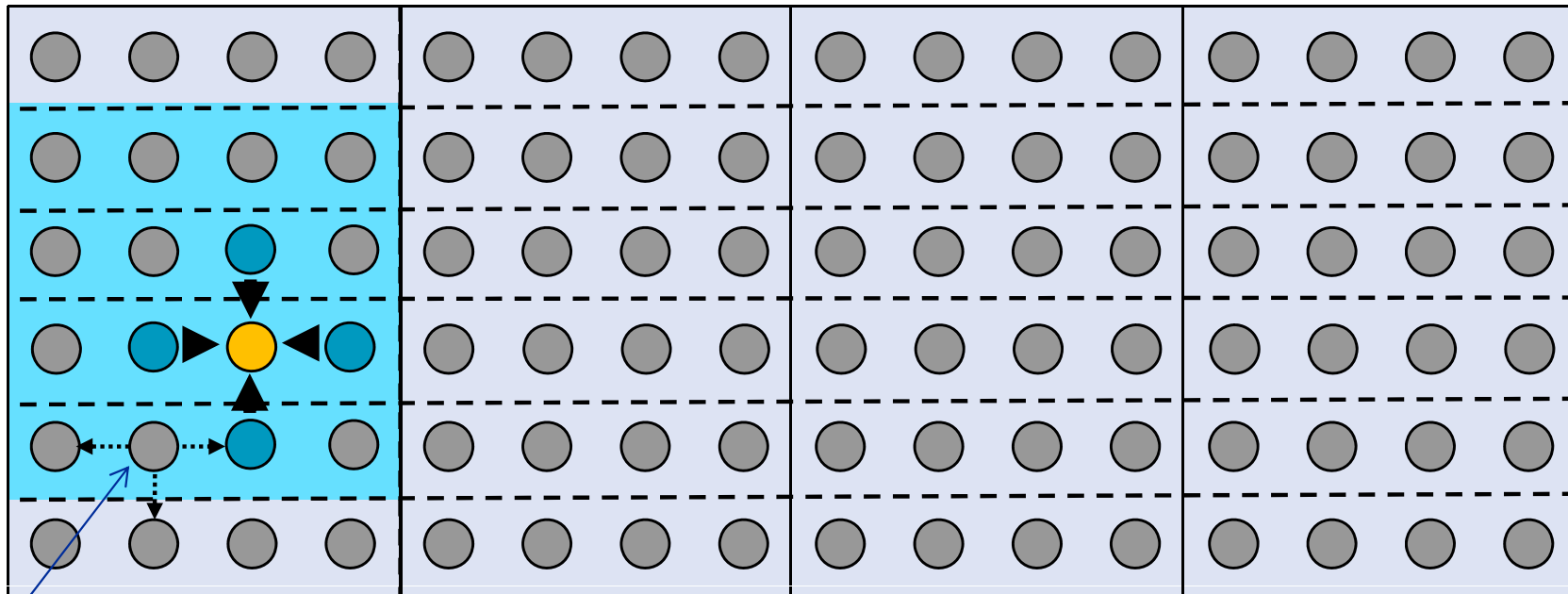
- divide system into blocks
- update block after block
- same performance as if three complete rows of the systems fit into cache



Jacobi iteration (2D): Spatial Blocking



- **Spatial blocking reorders traversal of data to account for the data update rule of the code**
- **Elements stay sufficiently long in cache to be fully reused**
- **Spatial blocking improves temporal locality!**
(Continuous access in inner loop ensures spatial locality)



This element remains in cache until it is fully used



- **Implementation:**

```
do it=1,itmax
  do ioffset=1,imax,iblock
    do k=1,kmax
      do i=ioffset, min(imax,ioffset+iblock-1)
        phi(i, k, t1) = ( phi(i-1, k, t0) + phi(i+1, k, t0)
                        + phi(i, k-1, t0) + phi(i, k+1, t0) ) * 0.25
      enddo
    enddo; enddo; enddo
```

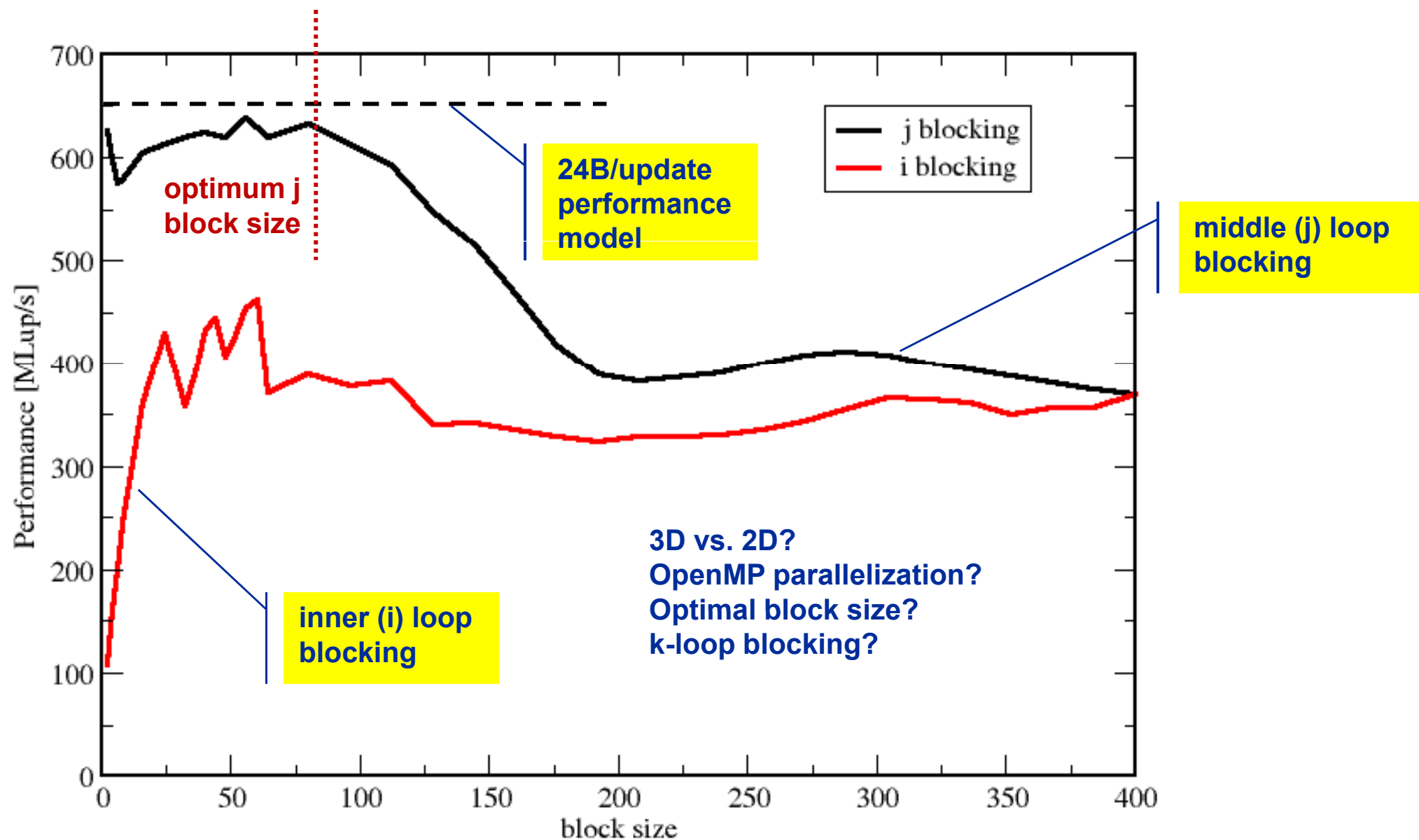
loop over i-blocks

- **Guidelines:**

- Blocking of inner loop levels (traversing continuously through main memory)
- Blocking size **iblock** large enough to keep elements sufficiently long in cache but cache size is a hard limit!
- Blocking loops may have some impact on ccNUMA page placement (see later)

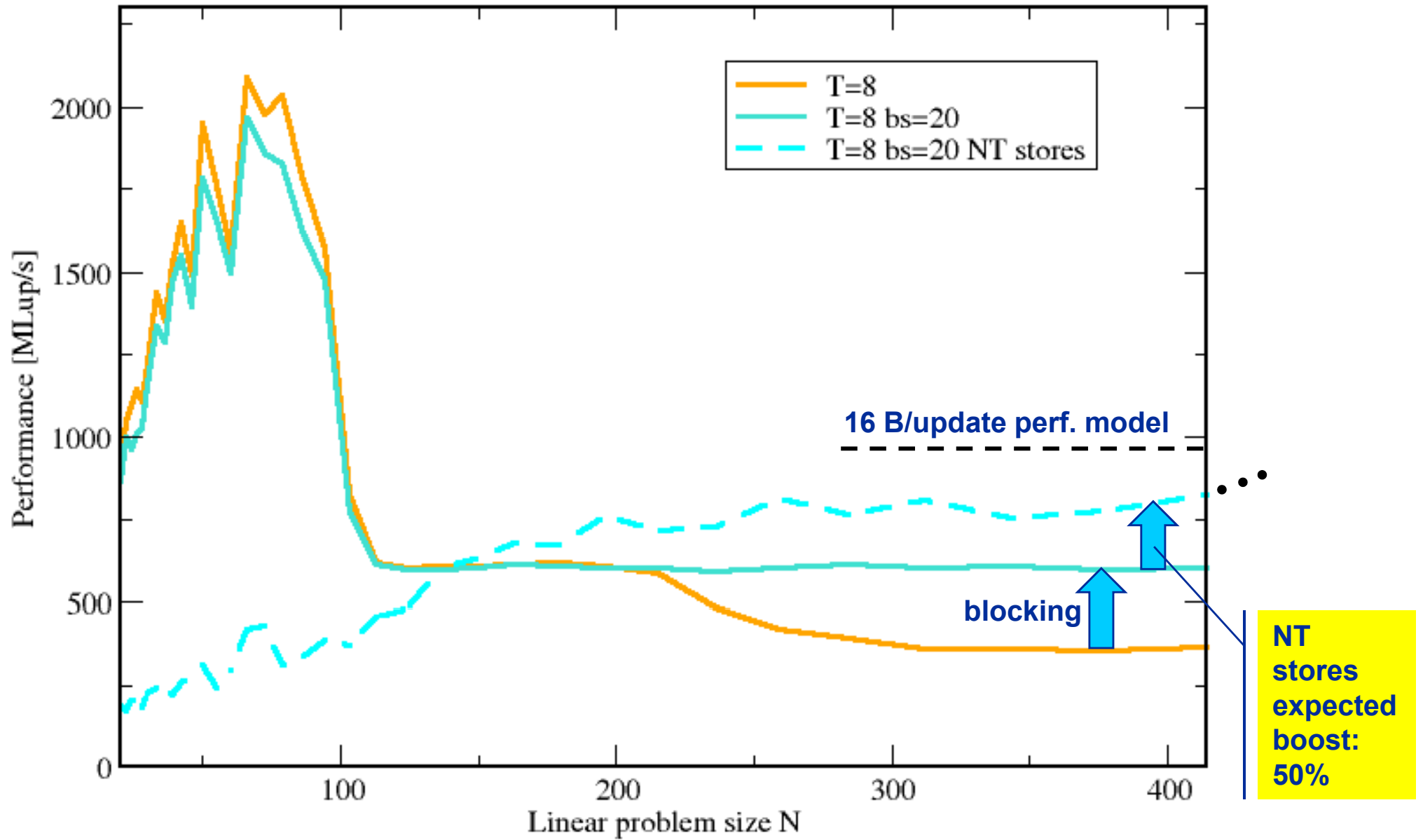
3D Jacobi solver (problem size 400³)

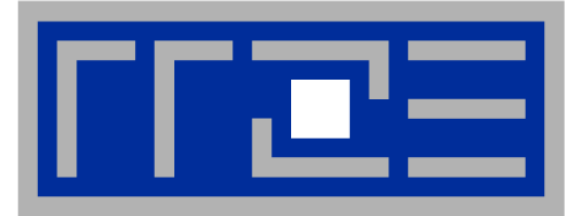
Blocking different loop levels (8 cores Interlagos)



3D Jacobi solver

Spatial blocking + nontemporal stores





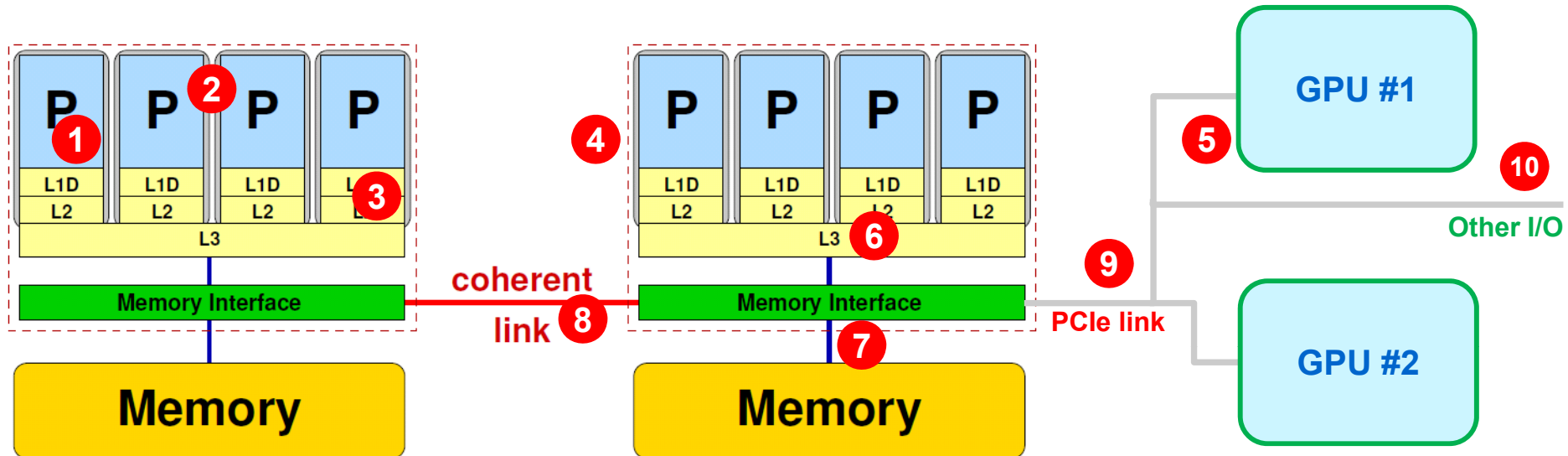
General remarks on the performance properties of multicore multsocket systems

Shared vs. scalable resources

Synchronization overhead



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?

The parallel vector triad benchmark

(Near-)Optimal code on x86 machines



```
call get_walltime(S)
!$OMP parallel private(j)
do j=1,R
  if(N.ge.CACHE_LIMIT) then
!DEC$ VECTOR NONTEMPORAL
!$OMP parallel do
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP end parallel do
  else
!DEC$ VECTOR TEMPORAL
!$OMP parallel do
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP end parallel do
  endif
  ! prevent loop interchange
  if(A(N2).lt.0) call dummy(A,B,C,D)
enddo
!$OMP end parallel

call get_walltime(E)
```

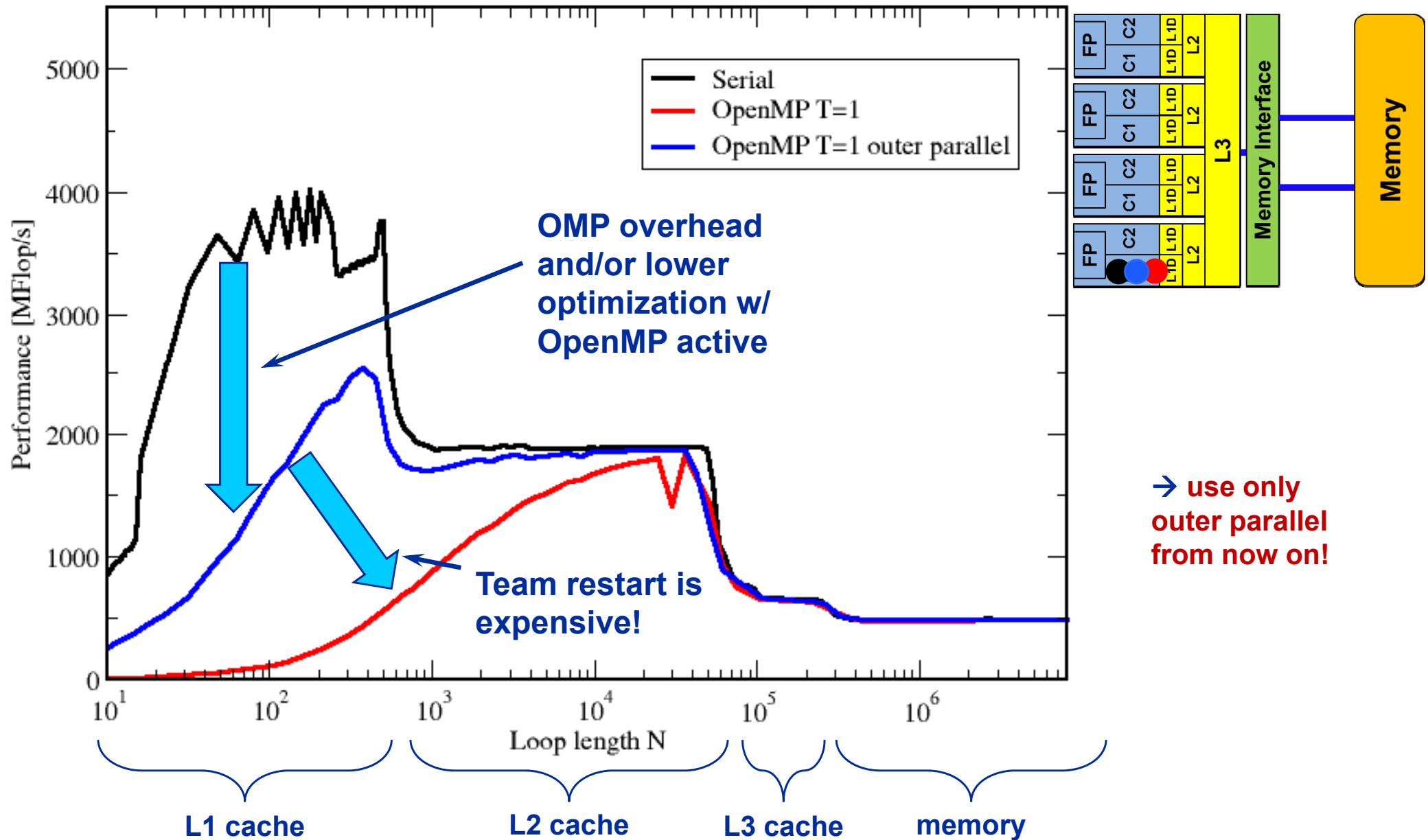
“outer parallel”: Avoid thread team restart at every workshared loop

Large-N version
(nontemporal stores)

Small-N version
(standard stores)

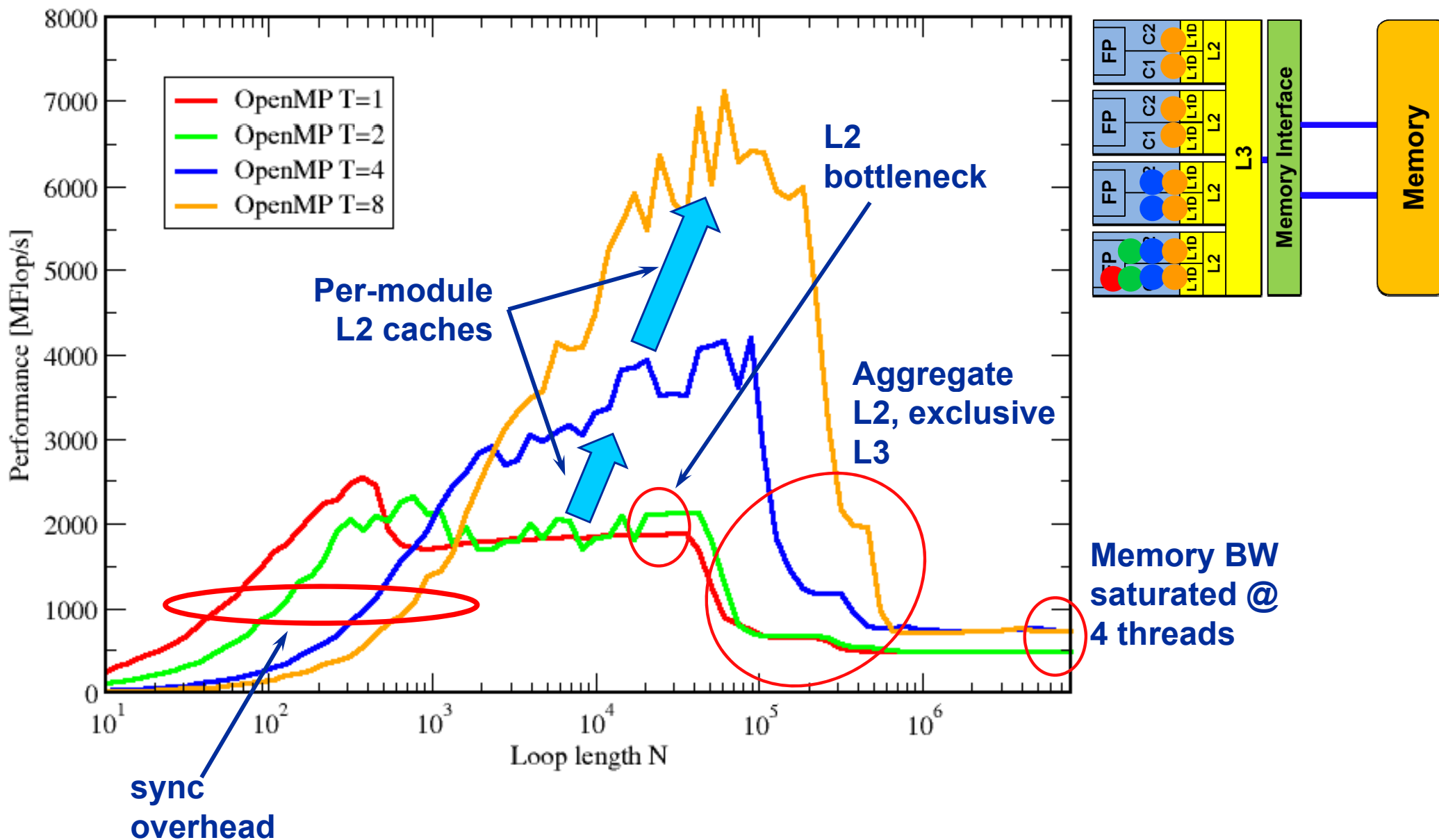
The parallel vector triad benchmark

Single thread on Interlagos node



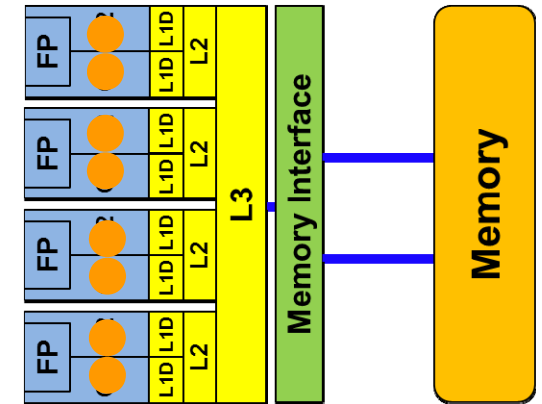
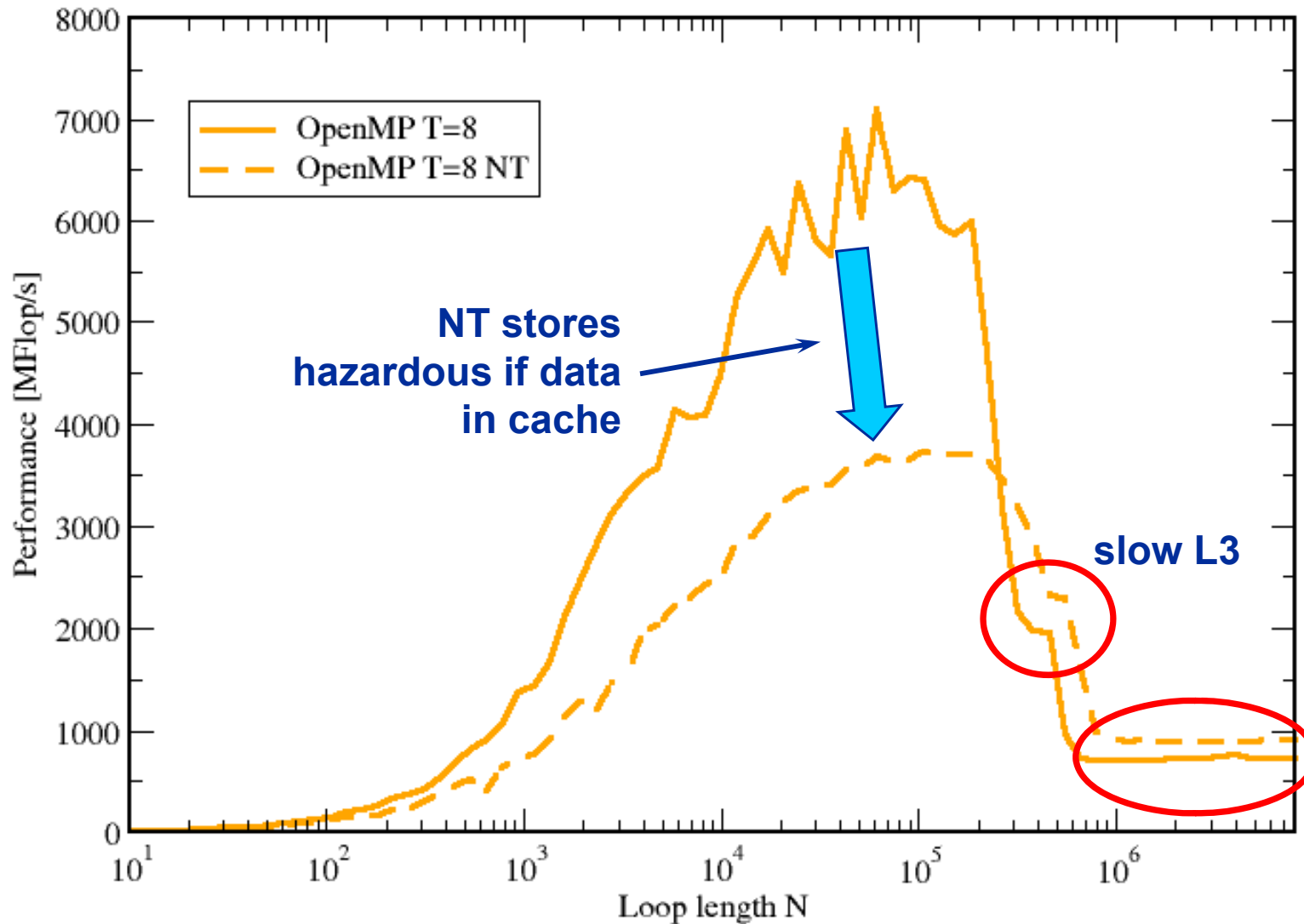
The parallel vector triad benchmark

Intra-chip scaling on Interlagos node



The parallel vector triad benchmark

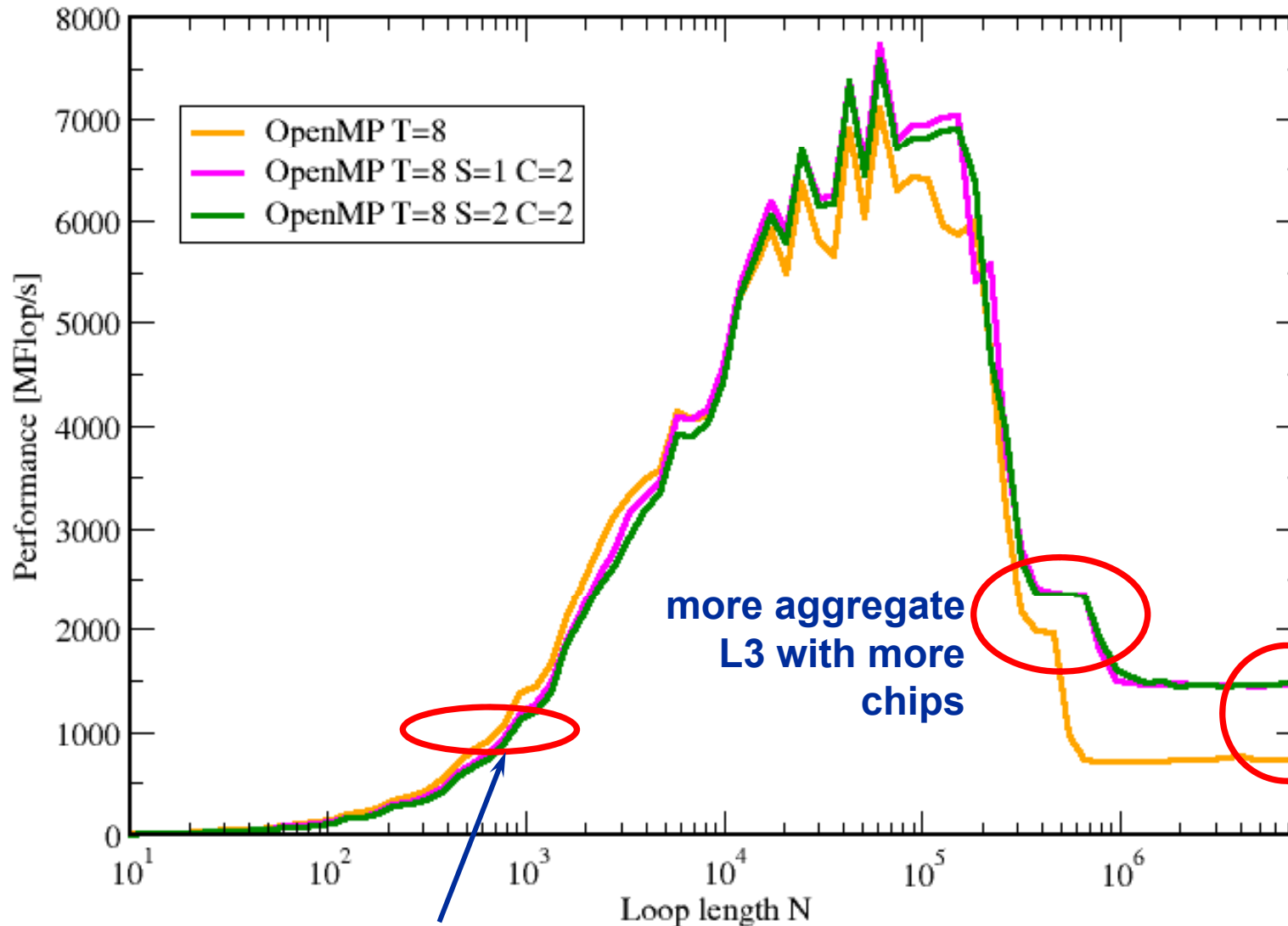
Nontemporal stores on Interlagos node



25% speedup for vector triad in memory via NT stores

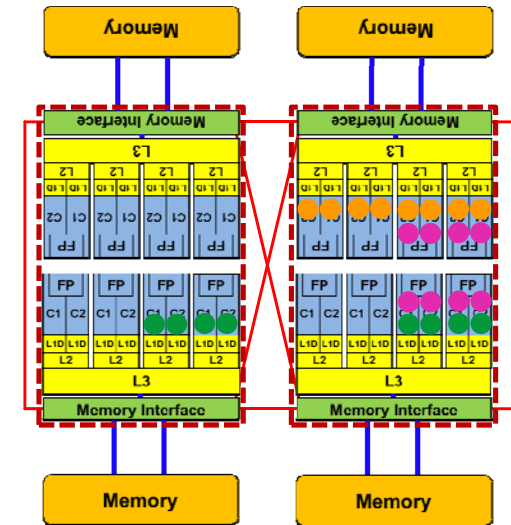
The parallel vector triad benchmark

Topology dependence on Interlagos node



sync overhead nearly topology-independent @ constant thread count

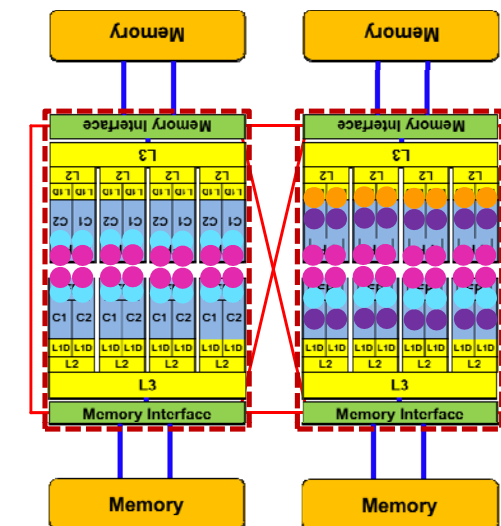
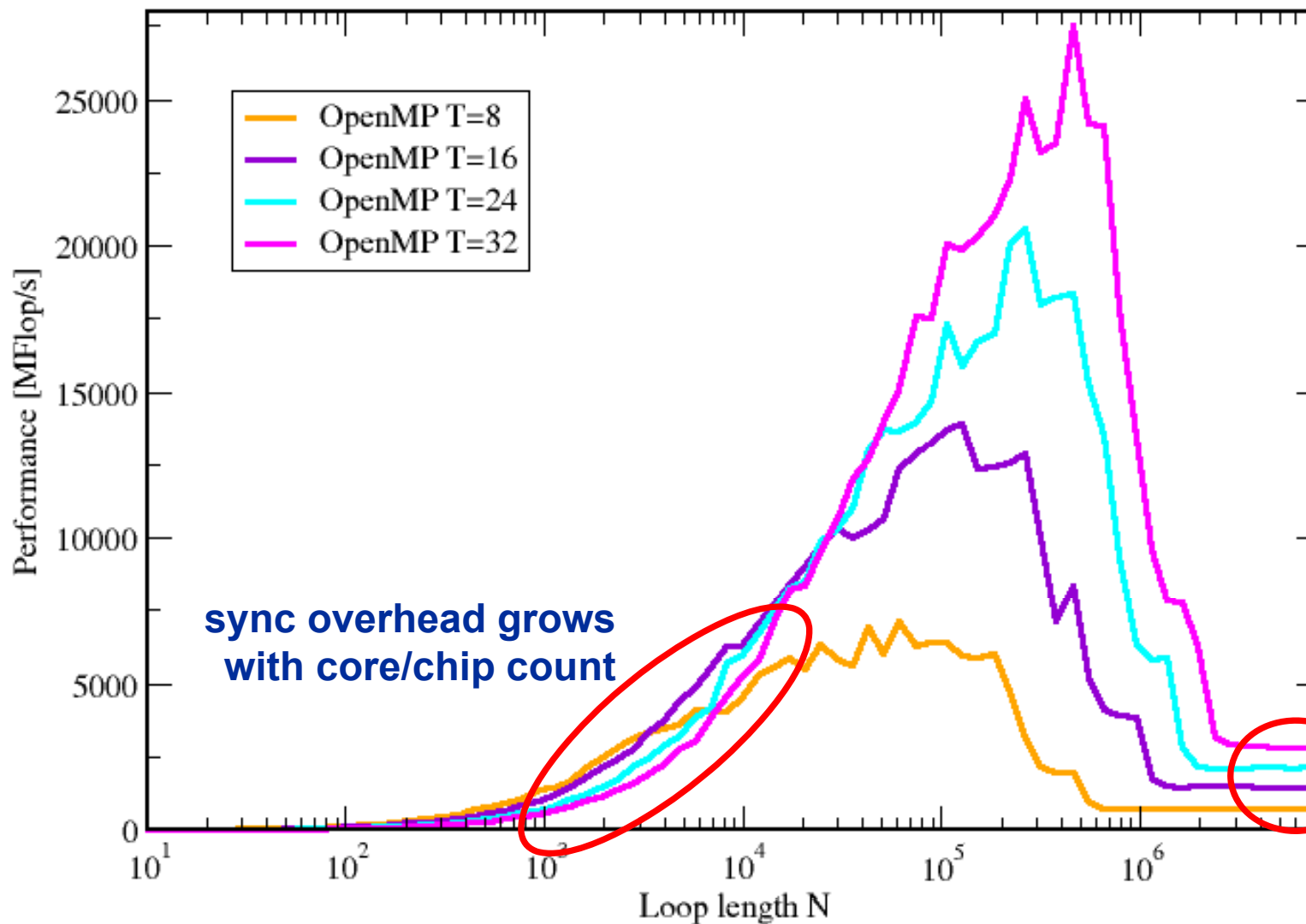
more aggregate L3 with more chips

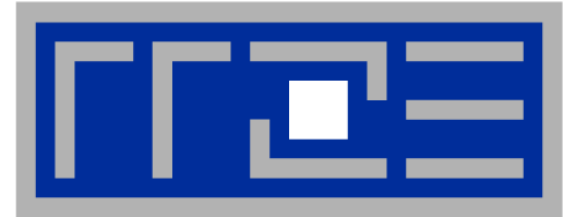


bandwidth scalability across memory interfaces

The parallel vector triad benchmark

Inter-chip scaling on Interlagos node



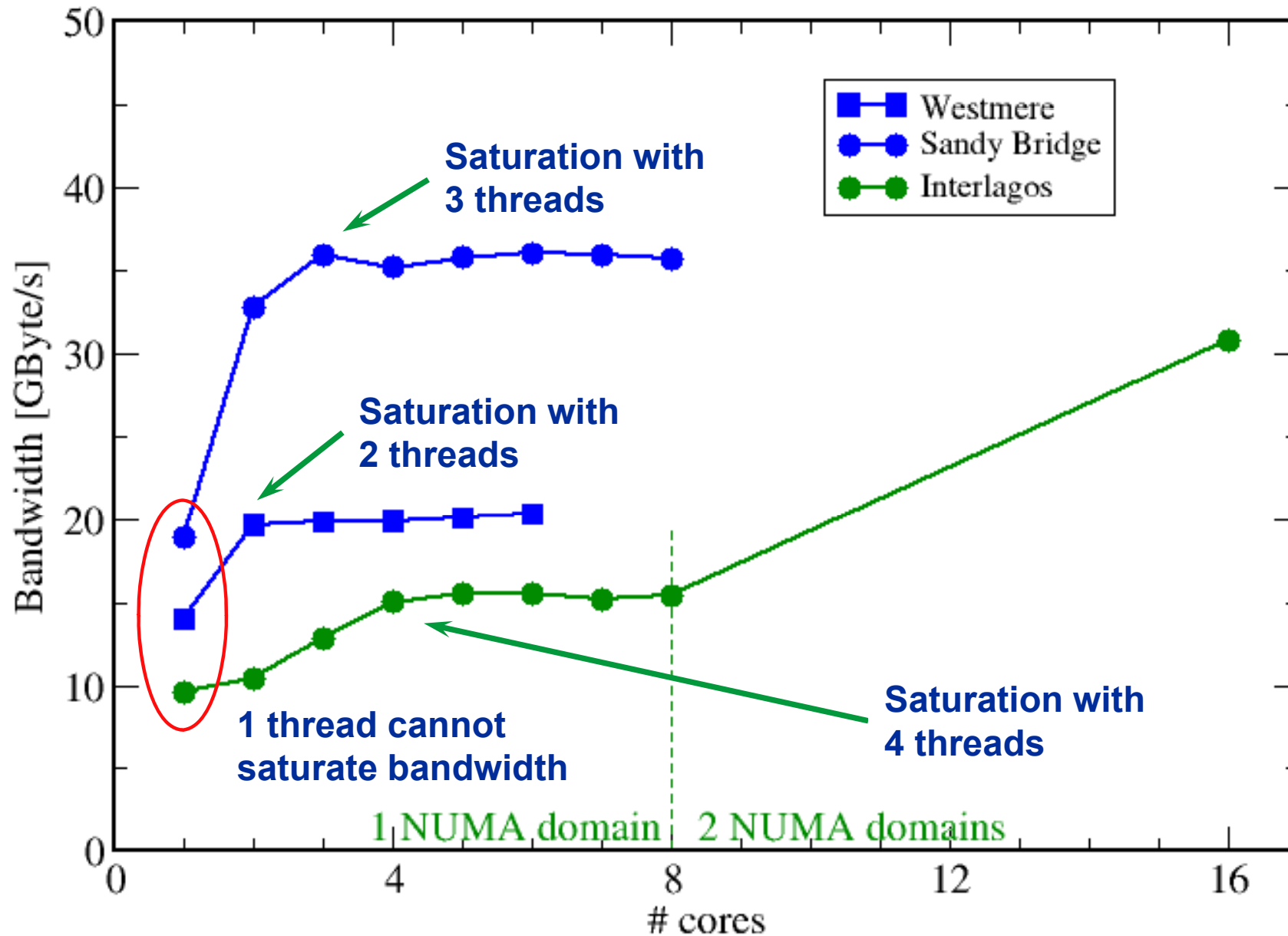


Bandwidth saturation effects in cache and memory

Low-level benchmark results

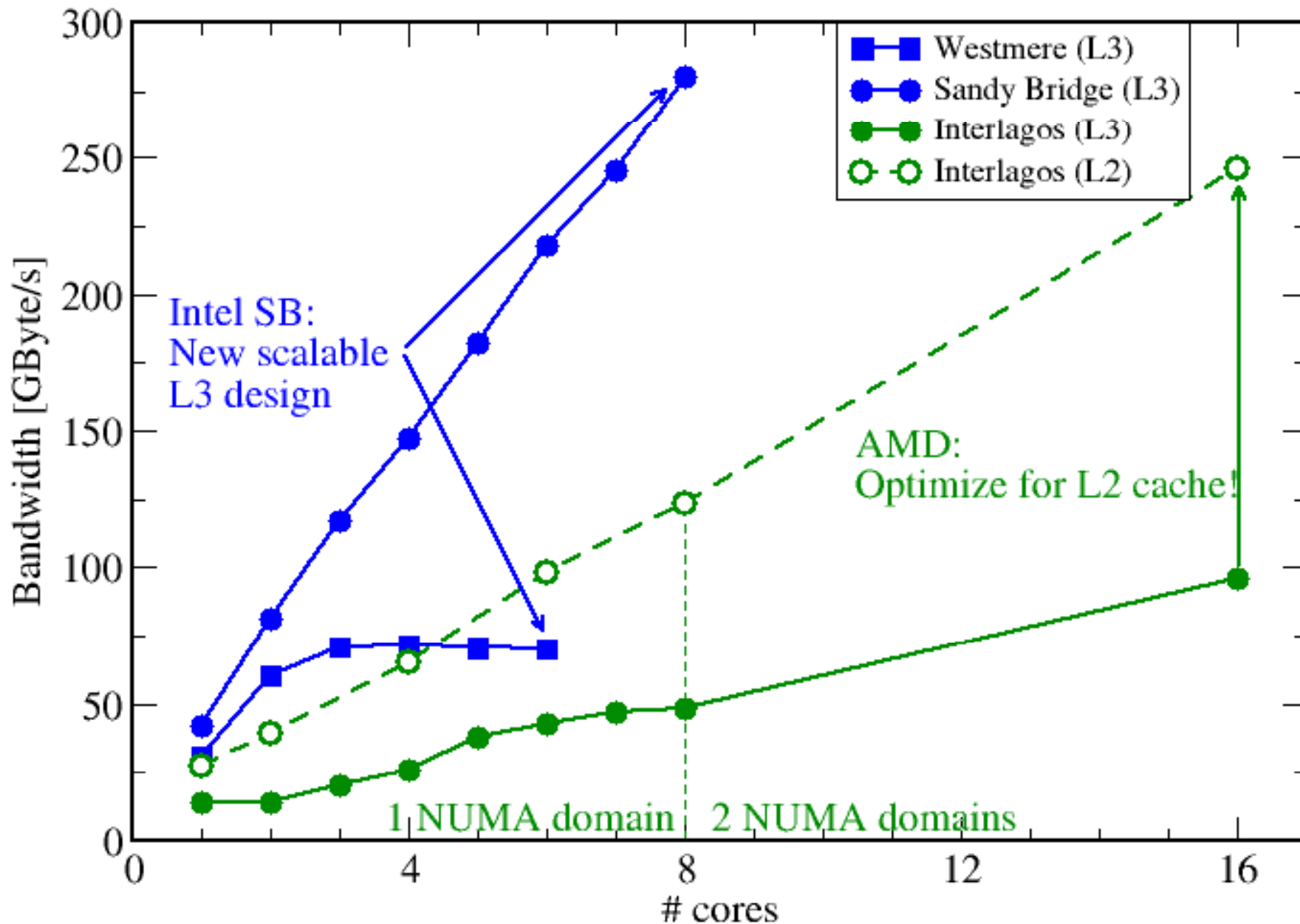
Bandwidth limitations: Main Memory

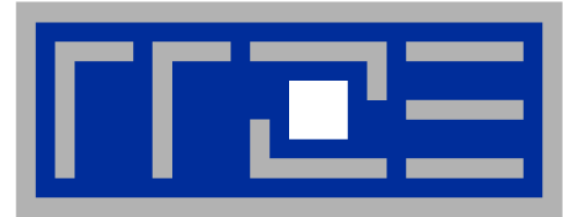
Scalability of shared data paths inside NUMA domain ($A(:) = B(:)$)



Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache





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 programs.

Determine costs via modified OpenMP
Microbenchmarks testcase (epcc)

On x86 systems there is no hardware support for synchronization!

- Next slide: Test **OpenMP** Barrier performance...
- for different compilers
- and different topologies:
 - shared cache
 - shared socket
 - between sockets
- and different thread counts
 - 2 threads
 - full domain (chip, socket, node)

Thread synchronization overhead on Interlagos

Barrier overhead in CPU cycles



2 Threads	Cray 8.03	GCC 4.6.2	PGI 11.8	Intel 12.1.3
Shared L2	258	3995	1503	128623
Shared L3	698	2853	1076	128611
Same socket	879	2785	1297	128695
Other socket	940	2740 / 4222	1284 / 1325	128718



Intel compiler barrier very expensive on Interlagos

OpenMP & Cray compiler 

Full domain	Cray 8.03	GCC 4.6.2	PGI 11.8	Intel 12.1.3
Shared L3	2272	27916	5981	151939
Socket	3783	49947	7479	163561
Node	7663	167646	9526	178892

Thread synchronization overhead on Intel CPUs

pthreads vs. OpenMP vs. Spin loop



2 Threads	Q9550 (shared L2)	i7 920 (shared L3)
pthreads_barrier_wait	23739	6511
omp barrier gcc 4.3.3	22603	7333
omp barrier icc 11.0	399	469
Spin loop	231	270

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

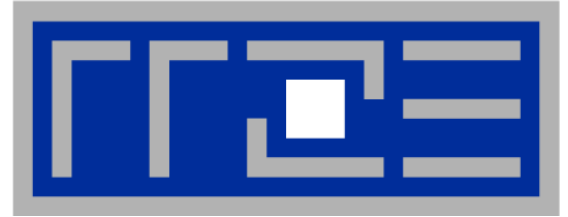
pthreads → OS kernel call



Spin loop does fine for shared cache sync

OpenMP & Intel compiler





**Case study:
OpenMP-parallel sparse matrix-vector
multiplication**

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



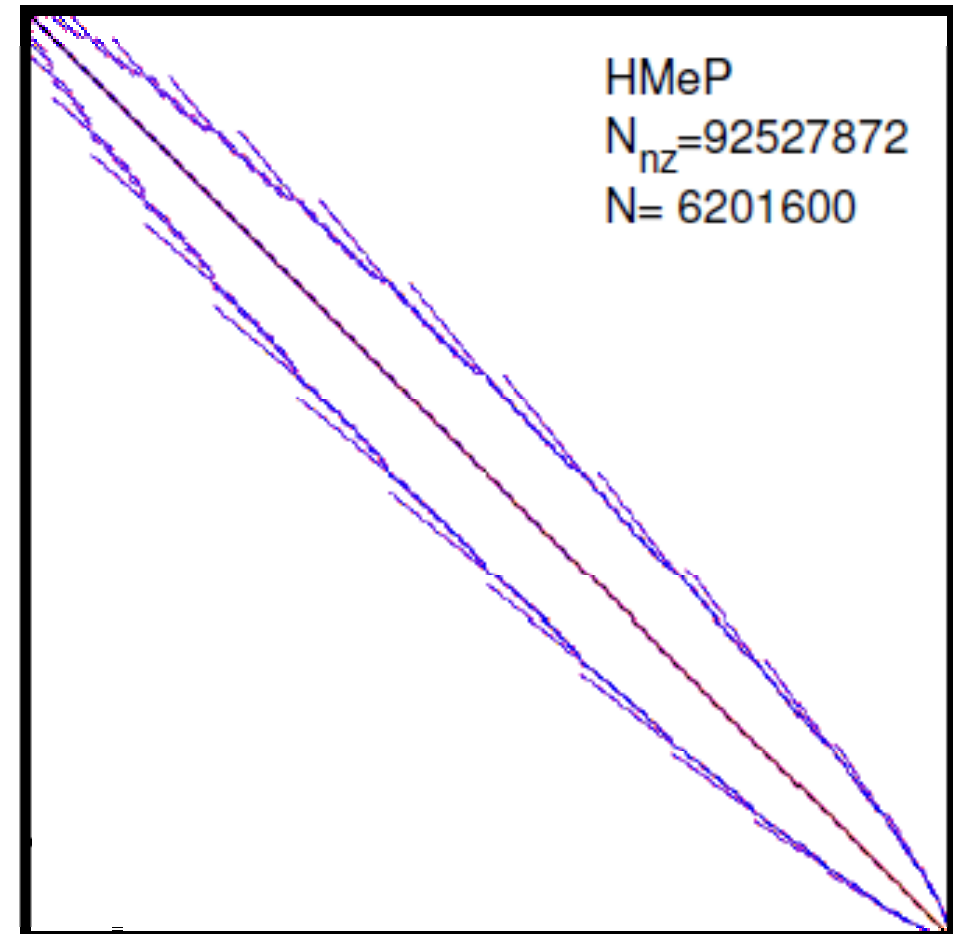
- Important kernel in many applications (matrix diagonalization, solving linear systems)
- **Strongly memory-bound for large data sets**
 - Streaming, with partially indirect access:

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

- Usually many spMVMs required to solve a problem
- **Following slides: Performance data on one 24-core AMD Magny Cours node**



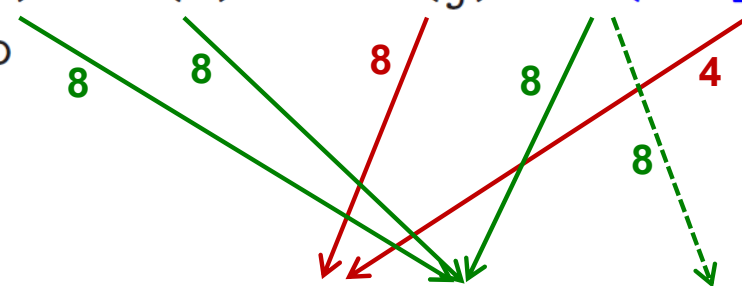
- **Data storage format is crucial for performance properties**
 - Most useful general format: Compressed Row Storage (**CRS**)
 - SpMVM is **easily parallelizable** in shared and distributed memory
- **For large problems, spMVM is inevitably memory-bound**
 - **Intra-LD saturation effect** on modern multicores
- **MPI-parallel spMVM is often communication-bound**
 - See hybrid part for what we can do about this...





- **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 = $\text{streamBW}/B_{\text{CRS}}$

- Determine κ by measuring performance and actual memory BW

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

$$= \left(6 + \frac{12}{N_{\text{nzr}}} + \frac{\kappa}{2} \right) \frac{\text{bytes}}{\text{flop}} .$$

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](https://arxiv.org/abs/1101.0091)

Application: Sparse matrix-vector multiply

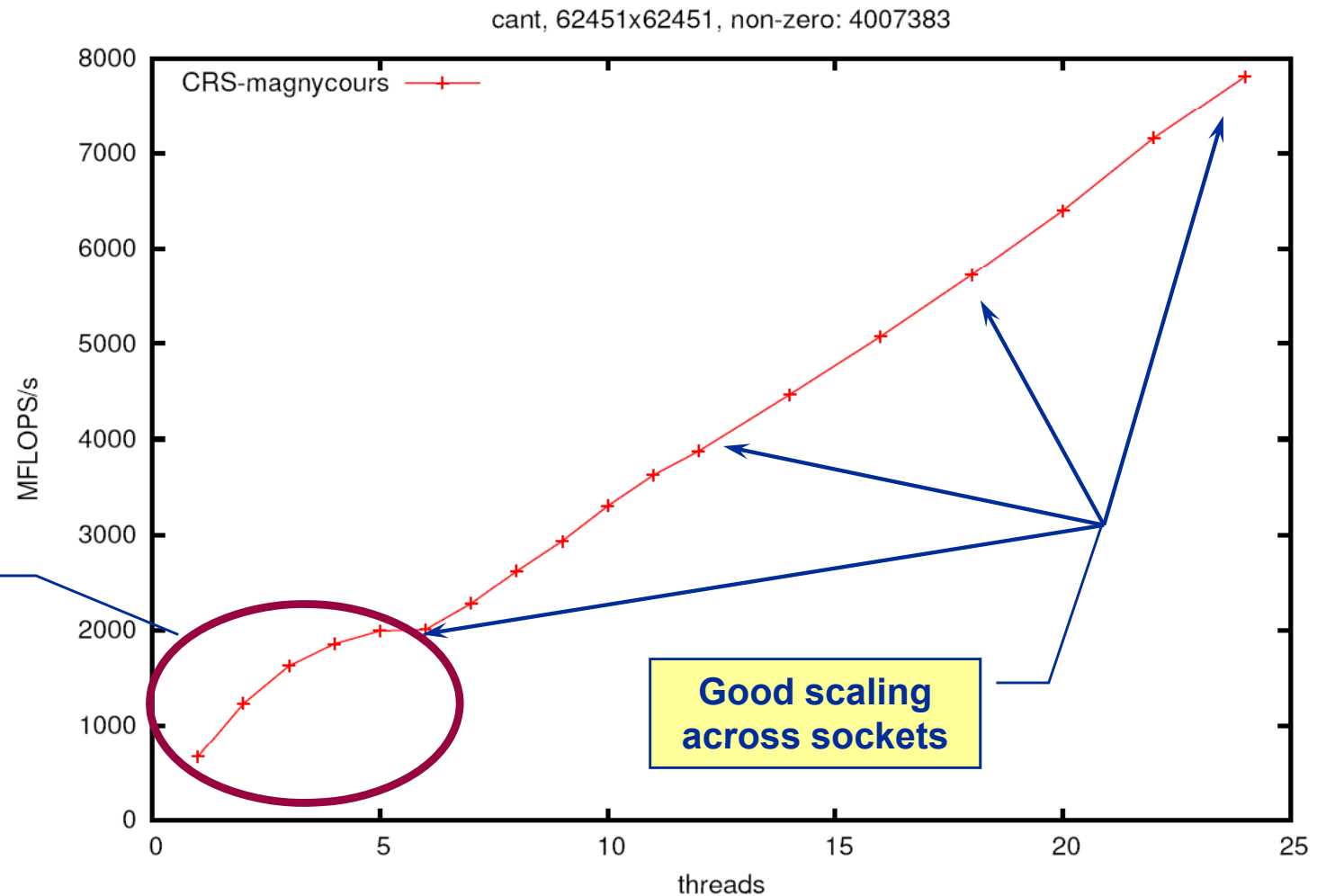
Strong scaling on one XE6 Magny-Cours node



Case 1: Large matrix



Intrasocket bandwidth bottleneck

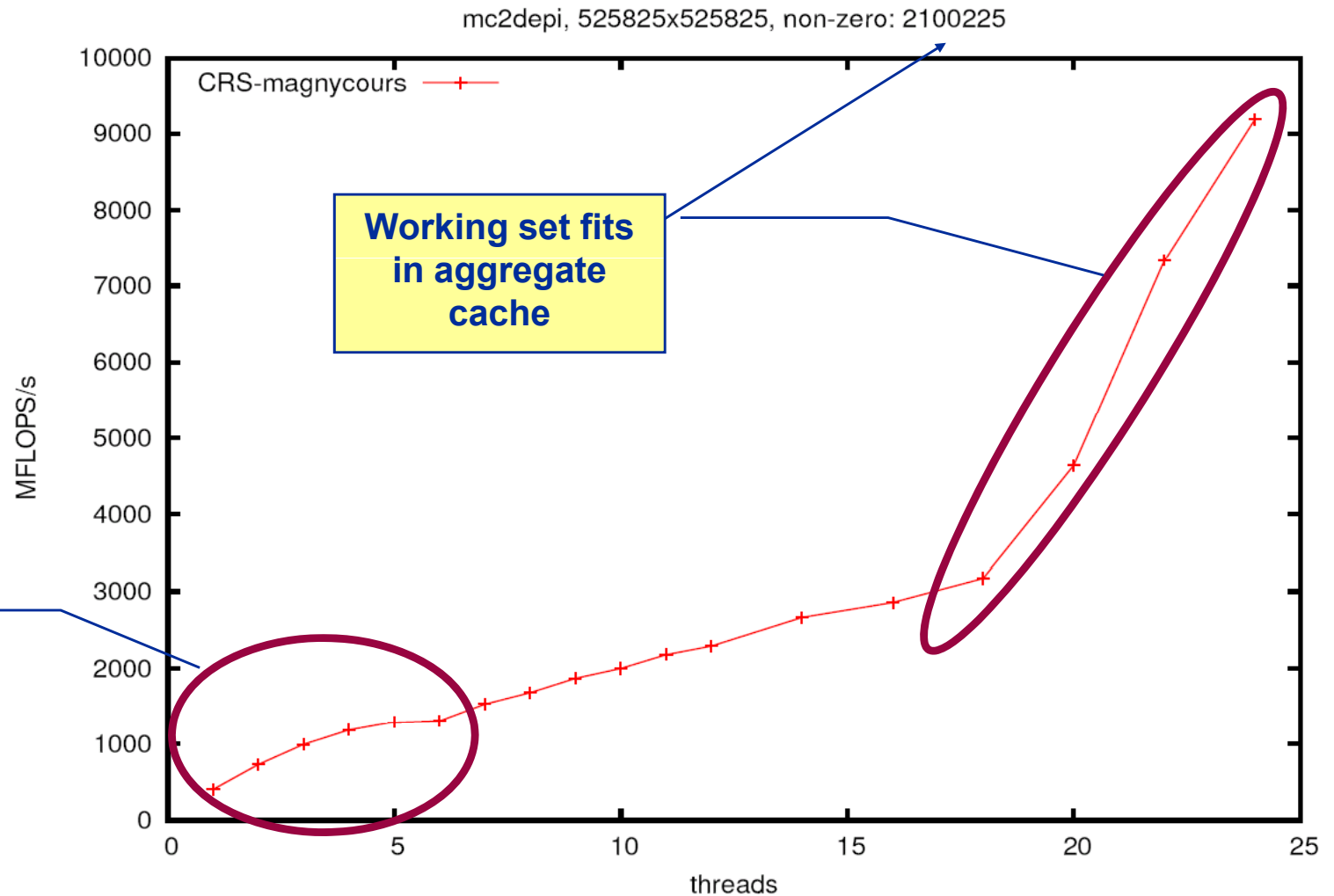




Case 2: Medium size



Intrasocket bandwidth bottleneck

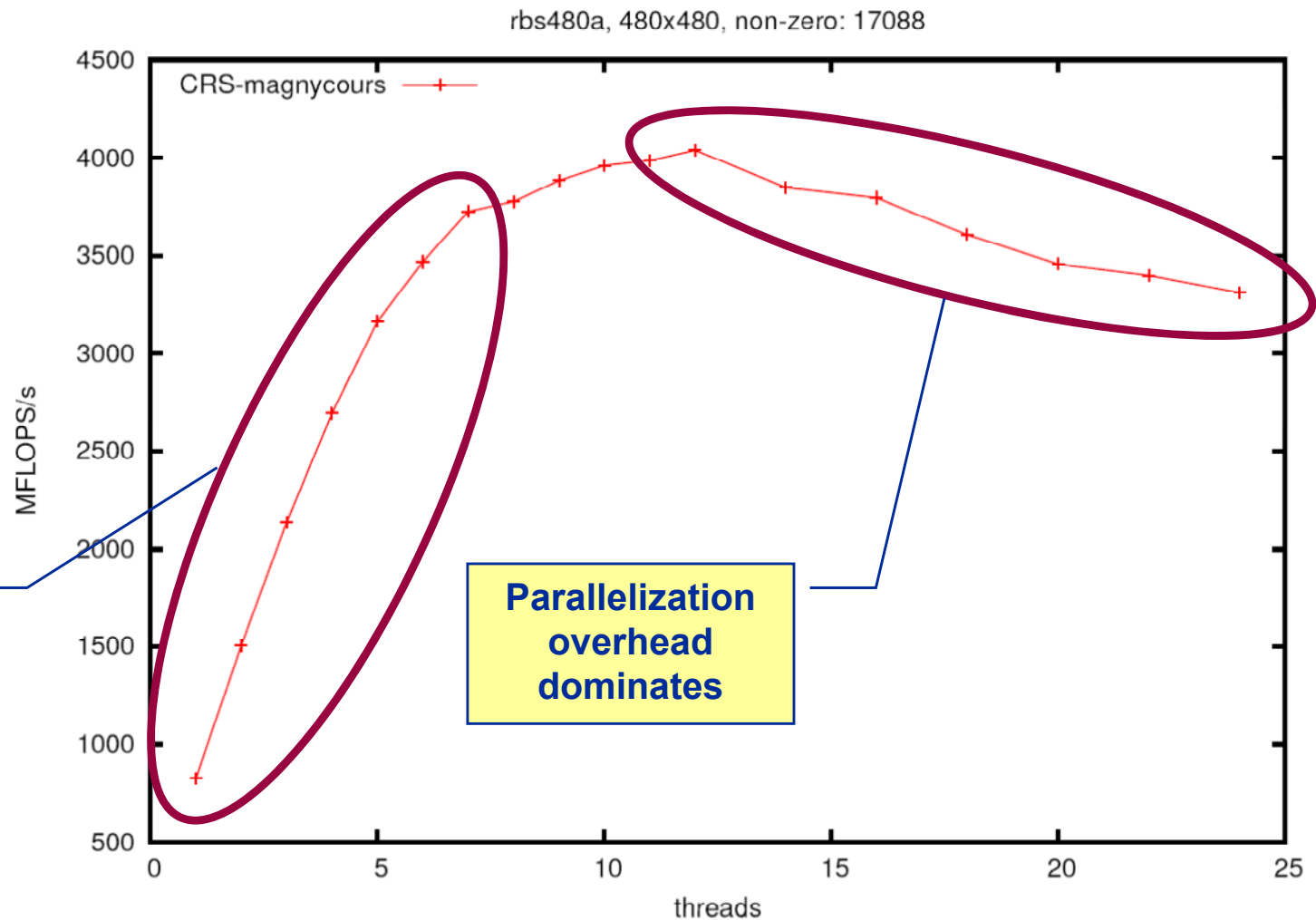




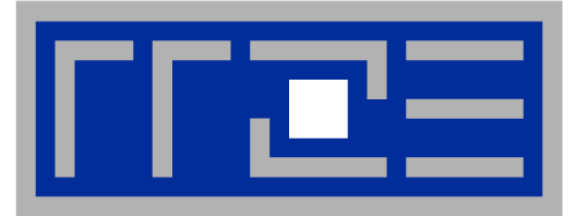
Case 3: Small size



No bandwidth bottleneck



Parallelization overhead dominates



Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

First touch placement policy

C++ issues

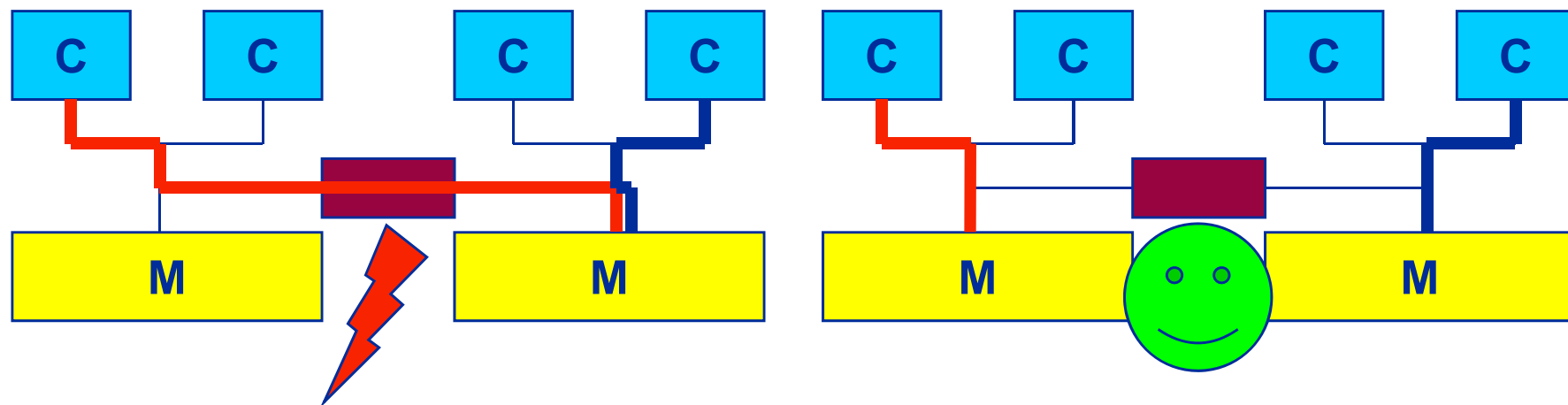
ccNUMA locality and dynamic scheduling

ccNUMA locality beyond first touch



■ ccNUMA:

- 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?**



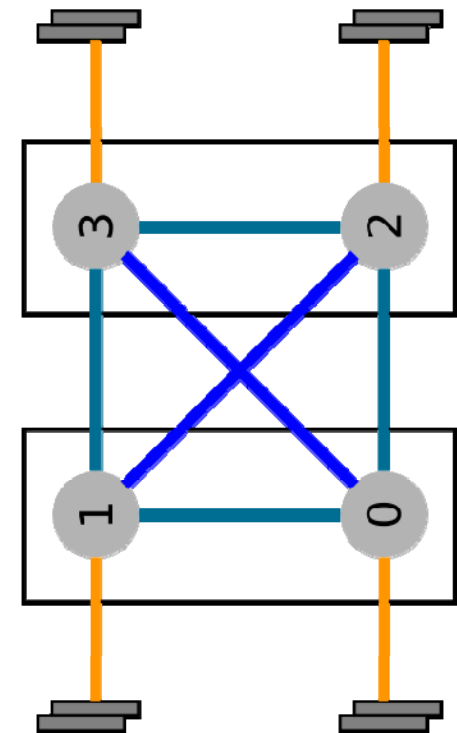
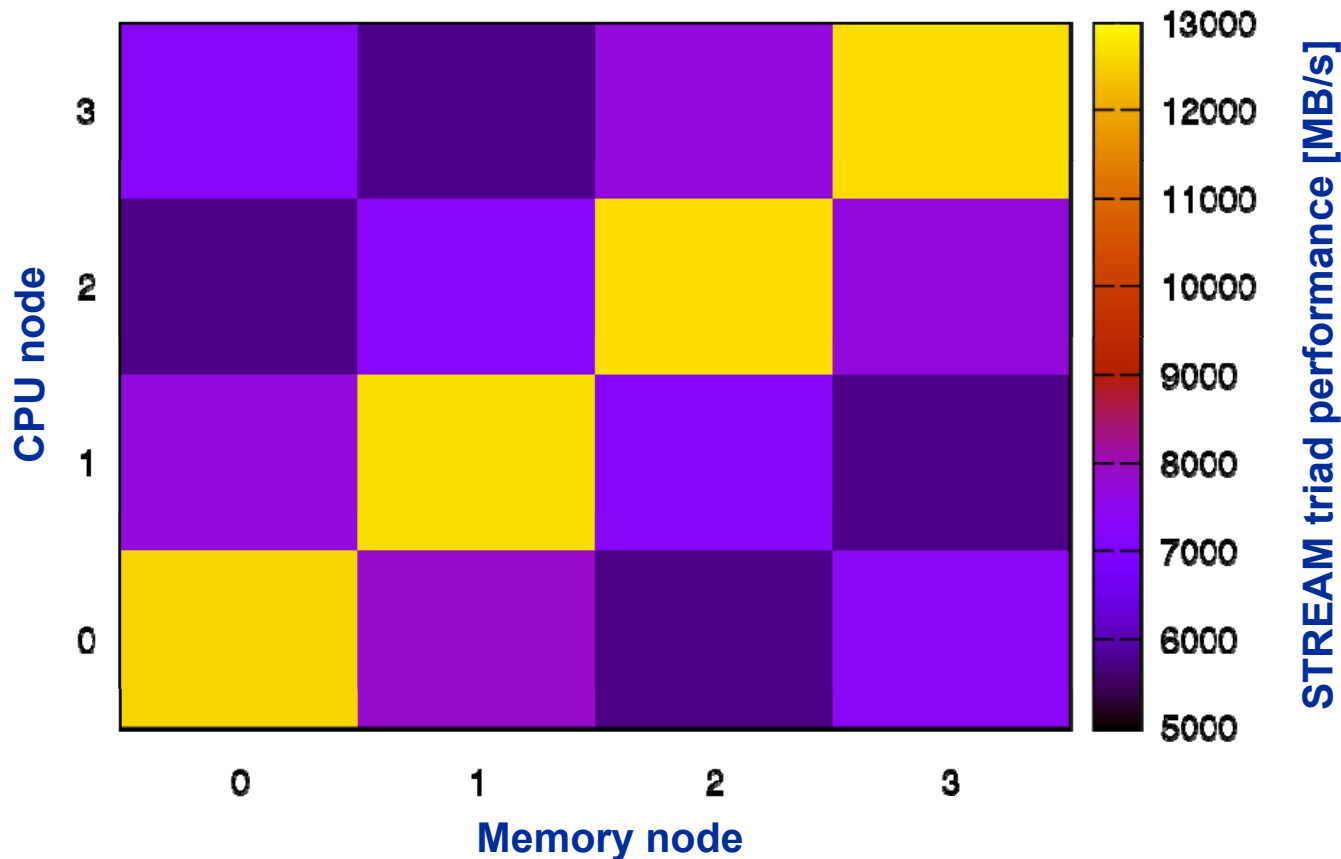
- Page placement is implemented in units of OS pages (often 4kB, possibly more)

Cray XE6 Interlagos node

4 chips, two sockets, 8 threads per ccNUMA domain



- **ccNUMA map: Bandwidth penalties for remote access**
 - Run 8 threads per ccNUMA domain (1 chip)
 - Place memory in different domain → 4x4 combinations
 - STREAM triad benchmark using nontemporal stores





- **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?**



- **"Golden Rule" of ccNUMA:**

A memory page gets mapped into the local memory of the processor that first touches it!

- Except if there is not enough local memory available
- This might be a problem, see later
- **Caveat: "touch" means "write", not "allocate"**
- **Example:**

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

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

Memory not mapped here yet

Mapping takes place here

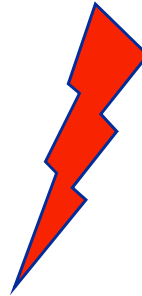
- **It is sufficient to touch a single item to map the entire page**



- 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
```



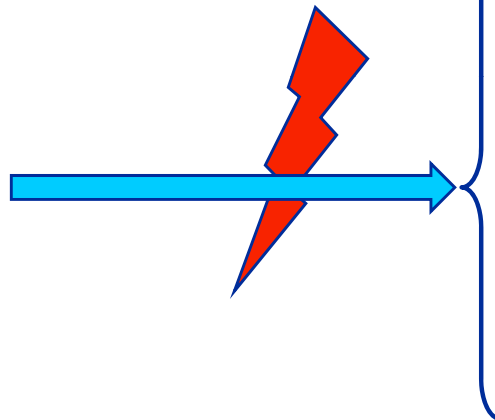


- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

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

```
READ(1000) A
```

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





- **Required condition: OpenMP loop schedule of initialization must be the same as in all computational loops**
 - Only 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 and static schedule
 - **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)



- **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 `D::operator new[]`**

```
void* D::operator new[](size_t n) {
    char *p = new char[n];    // allocate

    size_t i, j;
    #pragma omp parallel for private(j) schedule(...)
    for(i=0; i<n; i += sizeof(D))
        for(j=0; j<sizeof(D); ++j)
            p[i+j] = 0;
    return p;
}
```

parallel first
touch

```
void D::operator delete[](void* p) throw() {
    delete [] static_cast<char*>p;
}
```

- **Placement of objects is then done automatically by the C++ runtime via “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(10000000)
```



- 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 N:0-7 \  
-t intel ./a.out
```

Using performance counters for diagnosing bad ccNUMA access locality



Intel Nehalem EP node:

Uncore events only counted once per socket

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

RDTSC timing: 0.827196 s

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

Half of read BW comes from other socket!

If all fails...



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

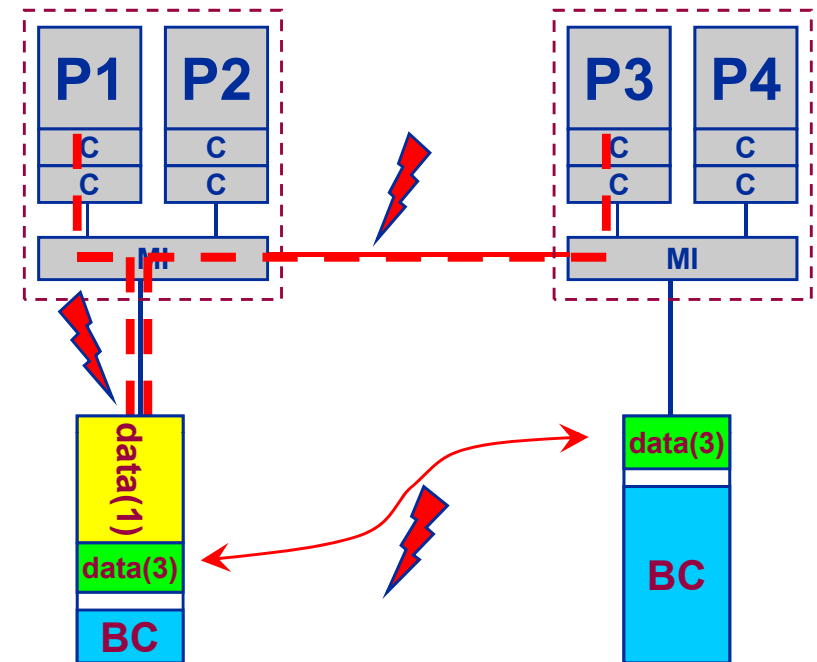
```
# numactl --hardware      # idle node!  
available: 2 nodes (0-1)  
node 0 size: 2047 MB  
node 0 free: 906 MB  
node 1 size: 1935 MB  
node 1 free: 1798 MB
```

```
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
```



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

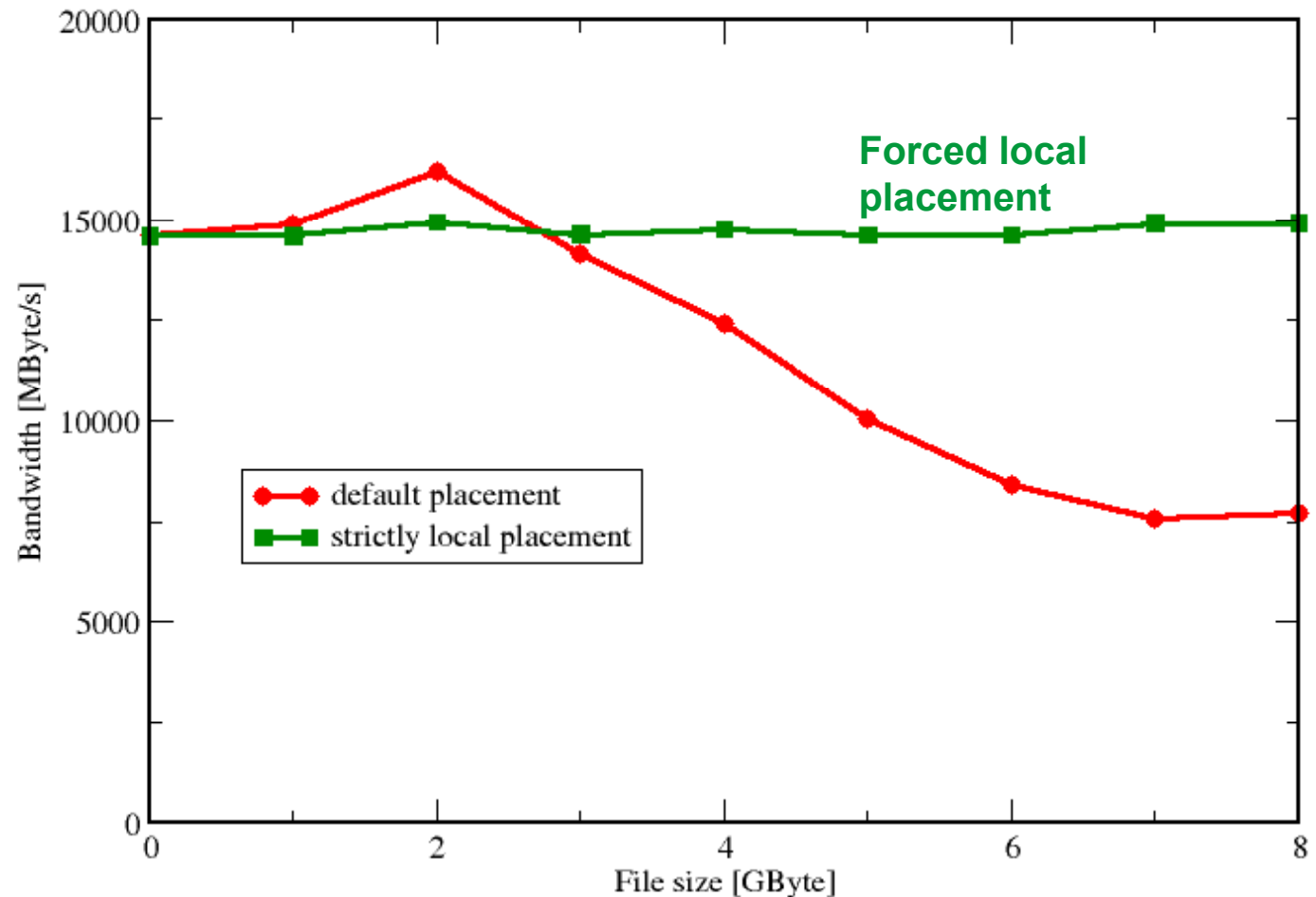


Real-world example: ccNUMA and the Linux buffer cache

Benchmark:

1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory installed in LD0

Result: By default, Buffer cache is given priority over local page placement
→ restrict to local domain if possible!





- 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



- **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`:

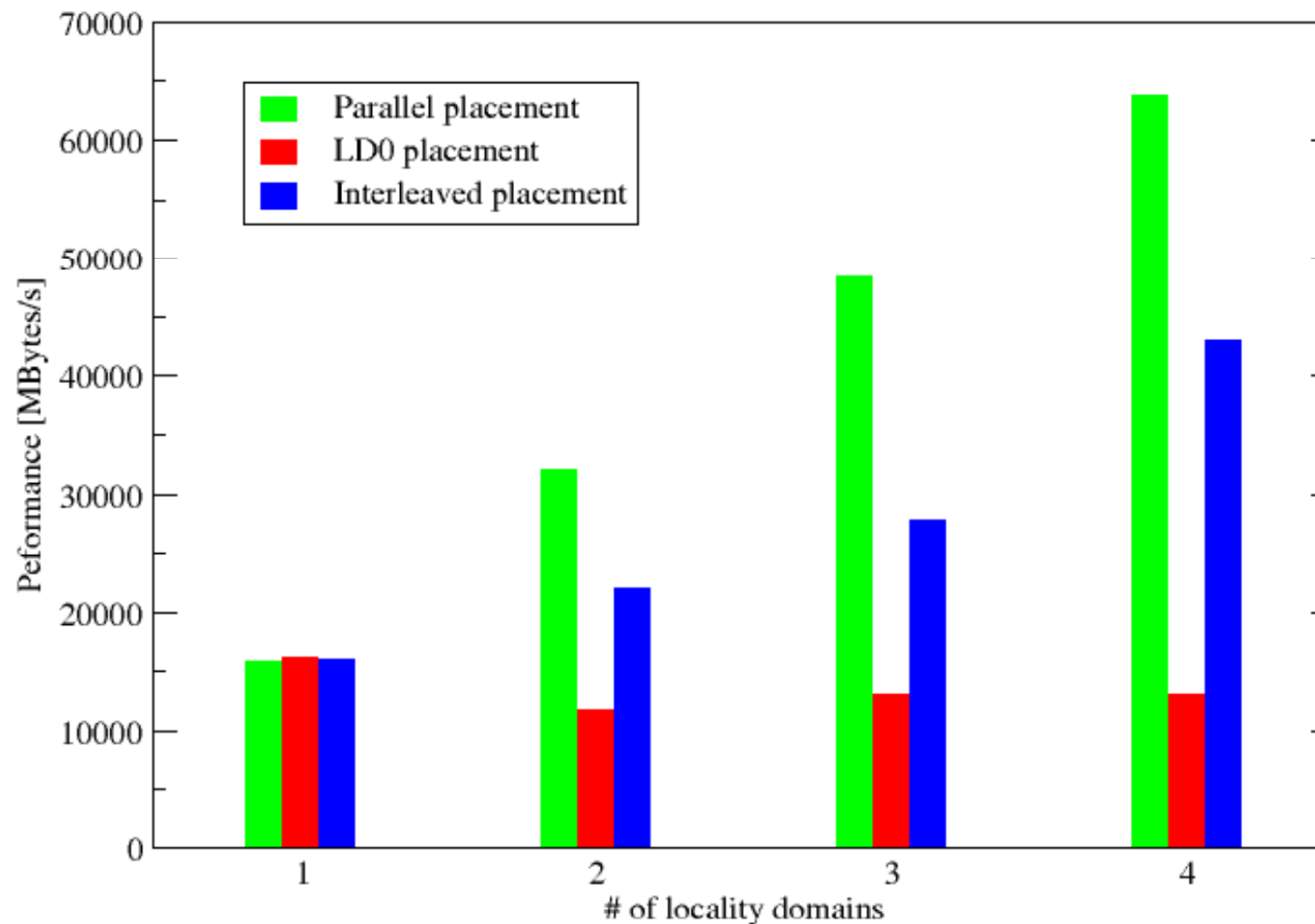
```
numactl --interleave=0-3 ./a.out
```

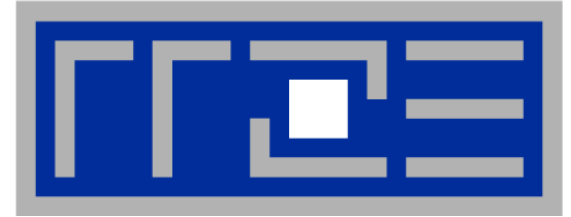
This is for **all** memory, not just the problematic arrays!

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



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





Simultaneous multithreading (SMT)

Principles and performance impact

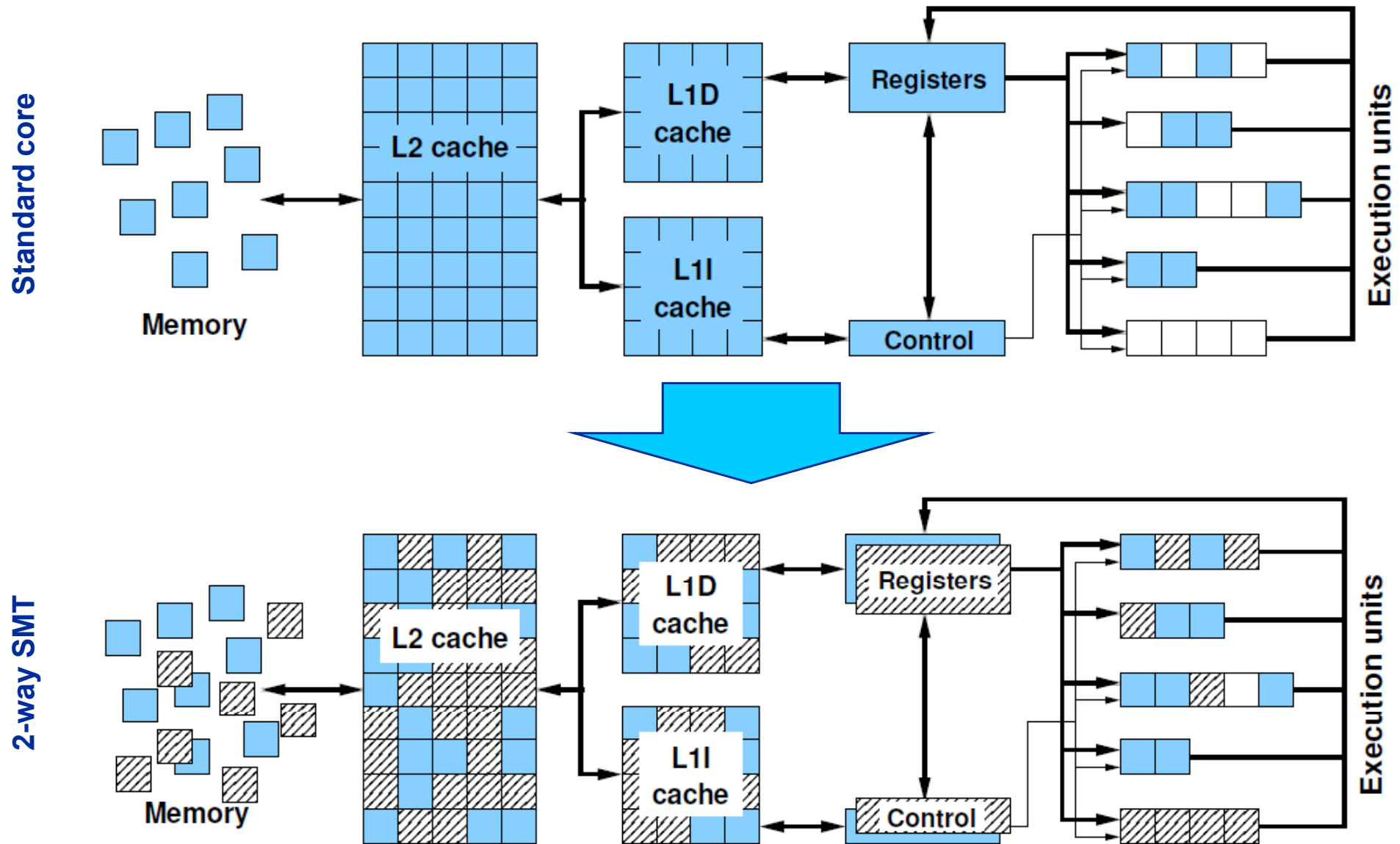
SMT vs. independent instruction streams

Facts and fiction

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

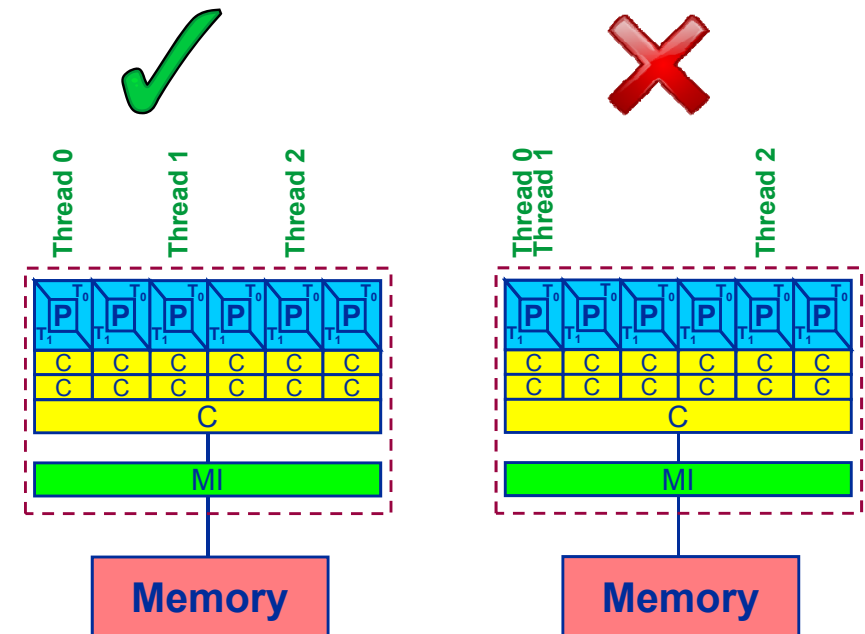


▪ SMT principle (2-way example):

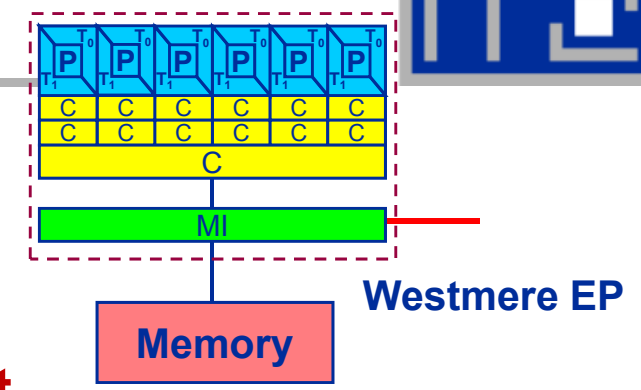




- **SMT is primarily suited for increasing processor throughput**
 - With multiple threads/processes running concurrently
- **Scientific codes tend to utilize chip resources quite well**
 - Standard optimizations (loop fusion, blocking, ...)
 - High data and instruction-level parallelism
 - Exceptions do exist
- **SMT is an important topology issue**
 - SMT threads share almost all core resources
 - Pipelines, caches, data paths
 - **Affinity matters!**
 - If SMT is not needed
 - pin threads to physical cores
 - or switch it off via BIOS etc.



SMT impact



- SMT adds **another layer of topology** (inside the physical core)
- **Caveat: SMT threads share all caches!**
- **Possible benefit: Better pipeline throughput**
 - Filling otherwise unused pipelines
 - Filling pipeline bubbles with other thread's executing instructions:

Thread 0:

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

Thread 1:

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

Dependency → pipeline stalls until previous MULT is over

Unrelated work in other thread can fill the pipeline bubbles

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

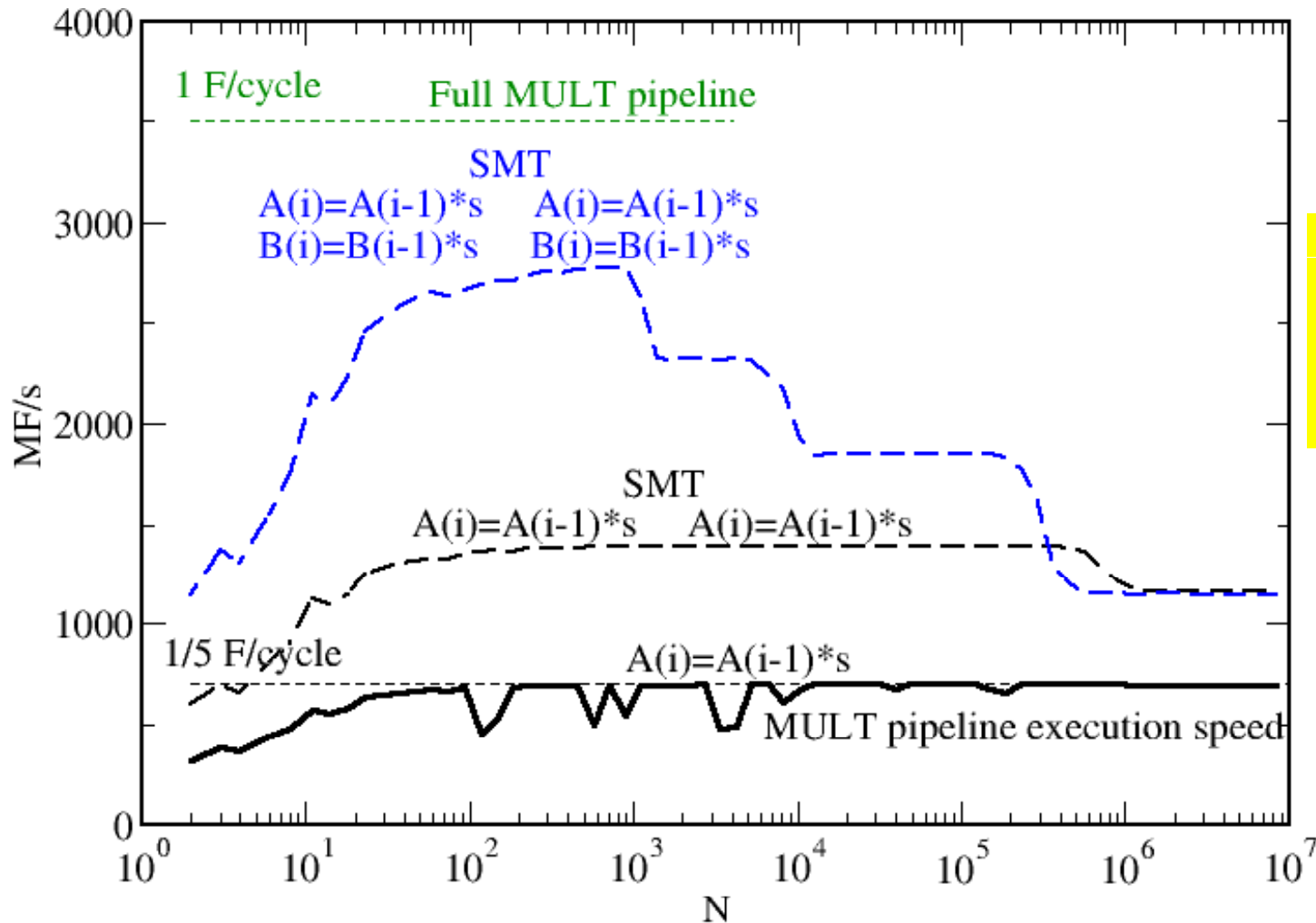
```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = func(i)*d
enddo
```

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:

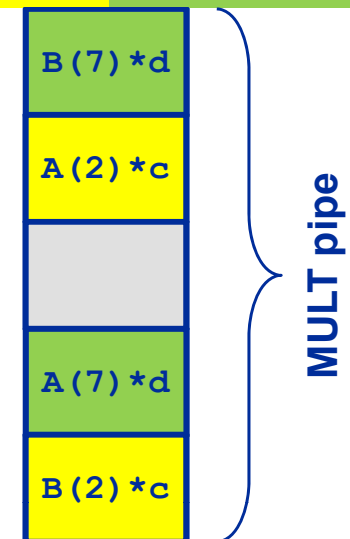
- SMT
- Multiple streams

Thread 0:

```
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
```

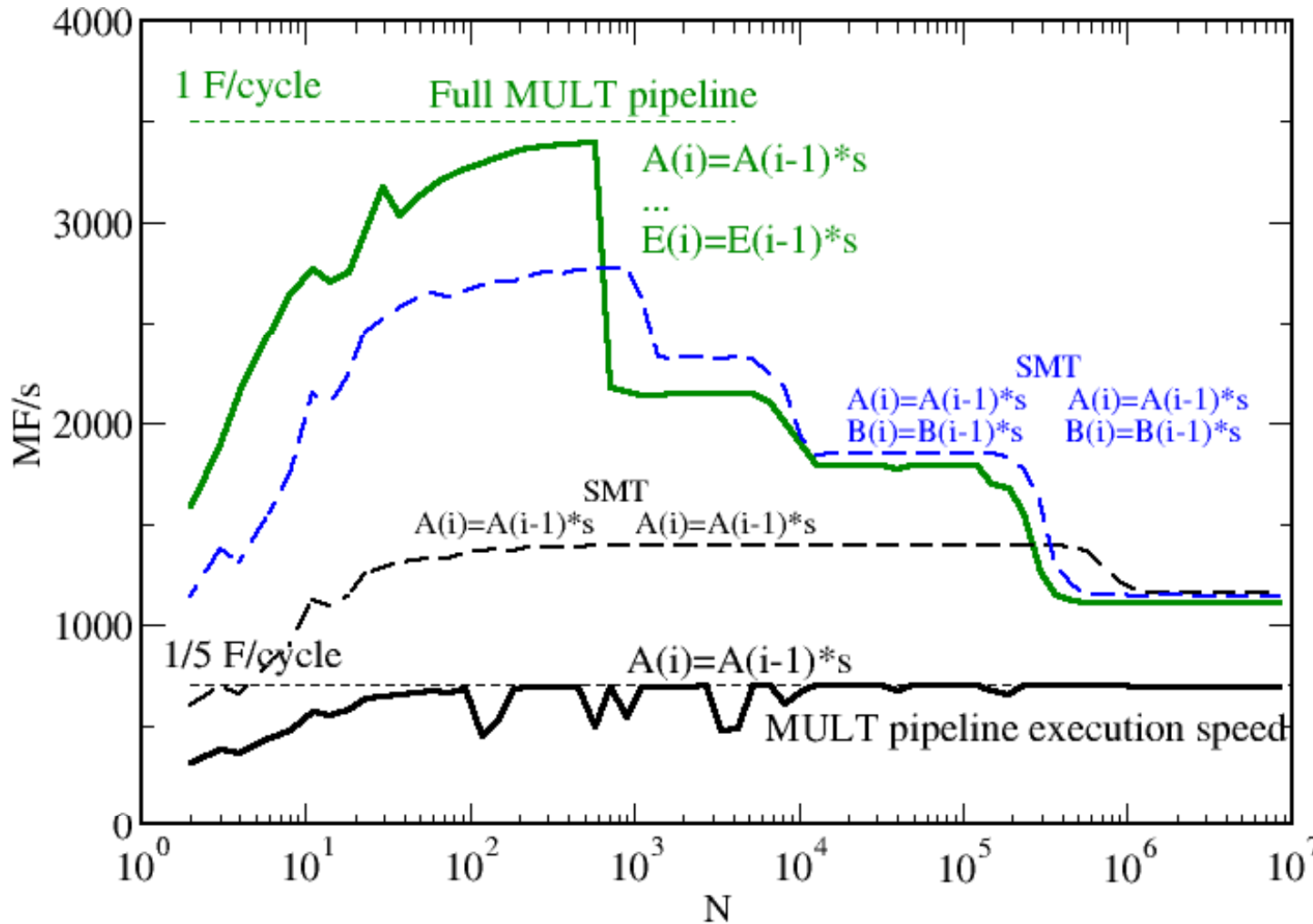


Simultaneous recursive updates with SMT



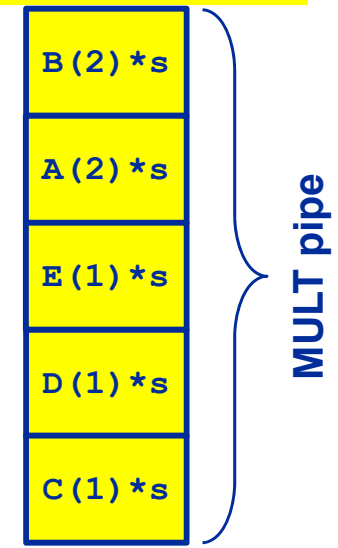
Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

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



```

Thread 0:
do i=1,N
  A(i)=A(i-1)*s
  B(i)=B(i-1)*s
  C(i)=C(i-1)*s
  D(i)=D(i-1)*s
  E(i)=E(i-1)*s
enddo
    
```



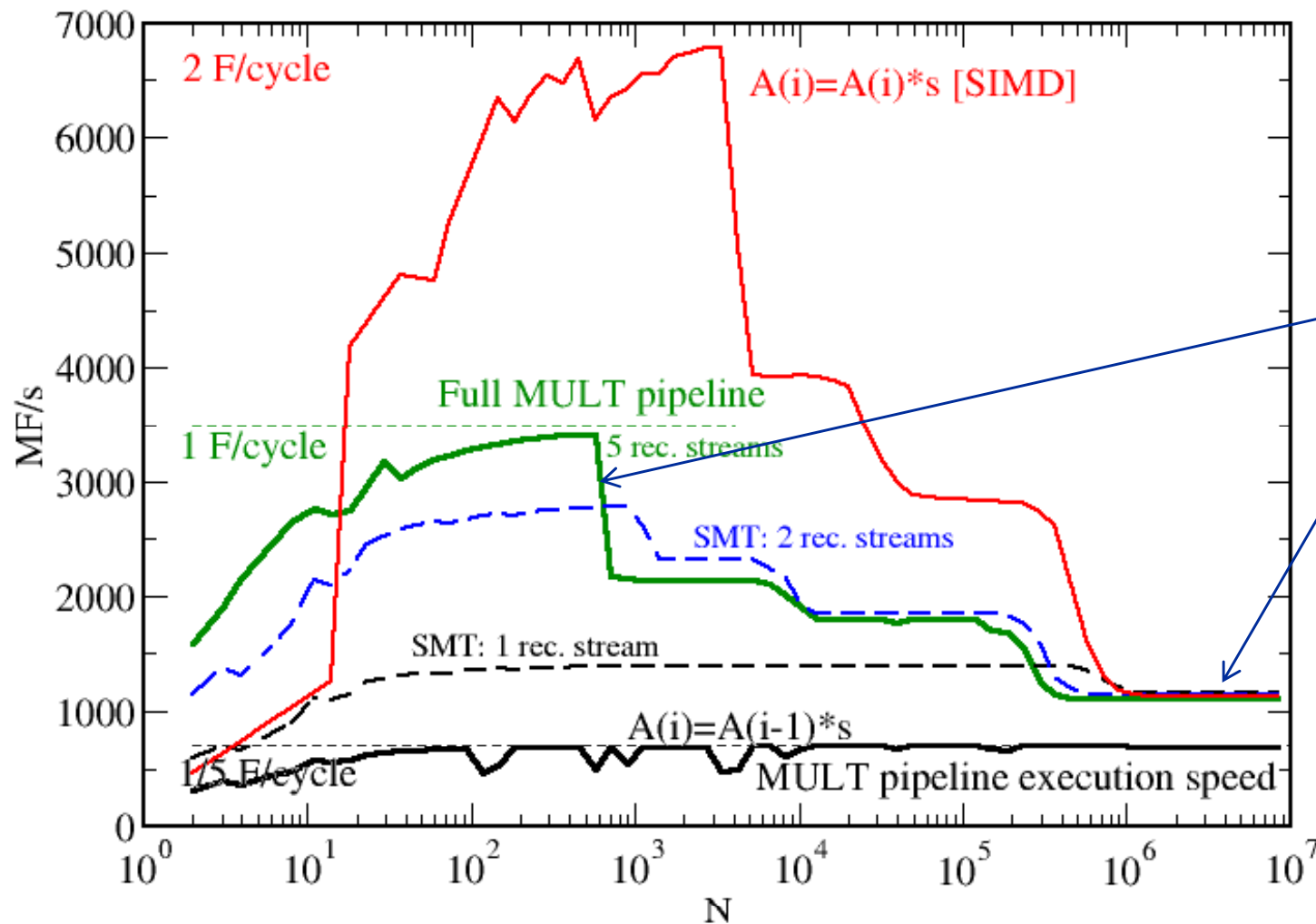
5 independent updates on a single thread do the same job!

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)**



Recursive update:

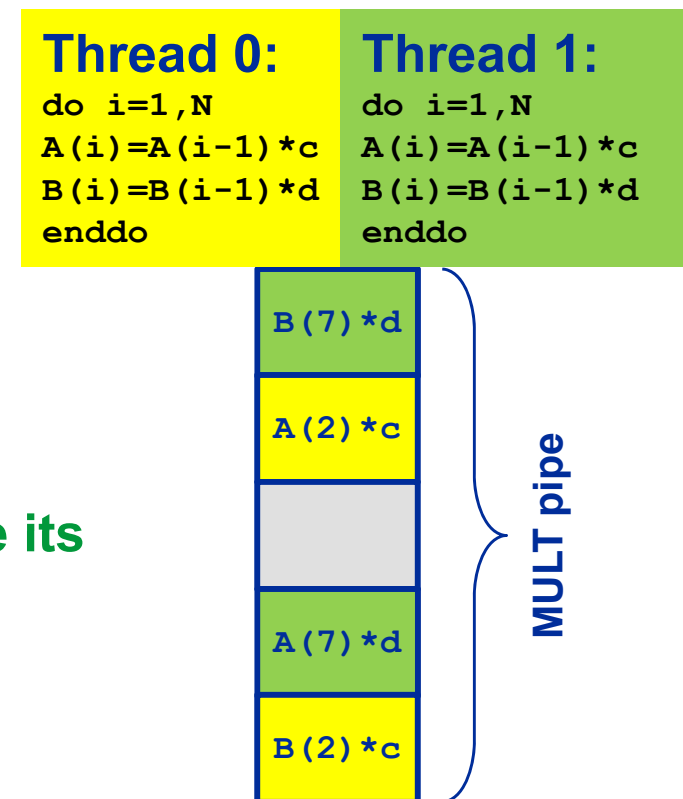
- SMT can fill pipeline bubbles
- A single thread can do so as well
- Bandwidth does not increase through SMT
- **SMT can not replace SIMD!**



- **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**



SMT myths: Facts and fiction (2)

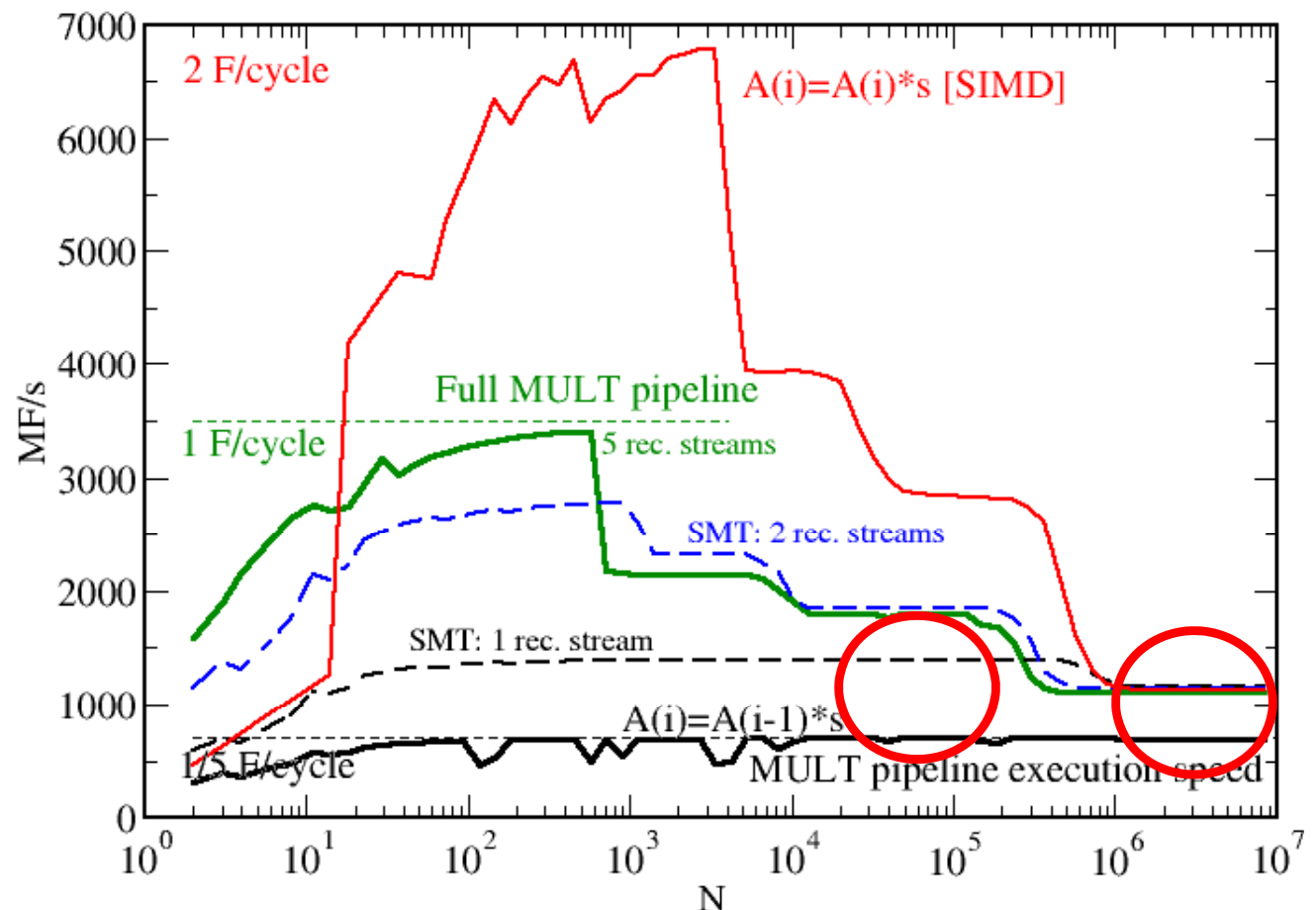


- **Myth:** “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

- **Truth:**

1. If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.

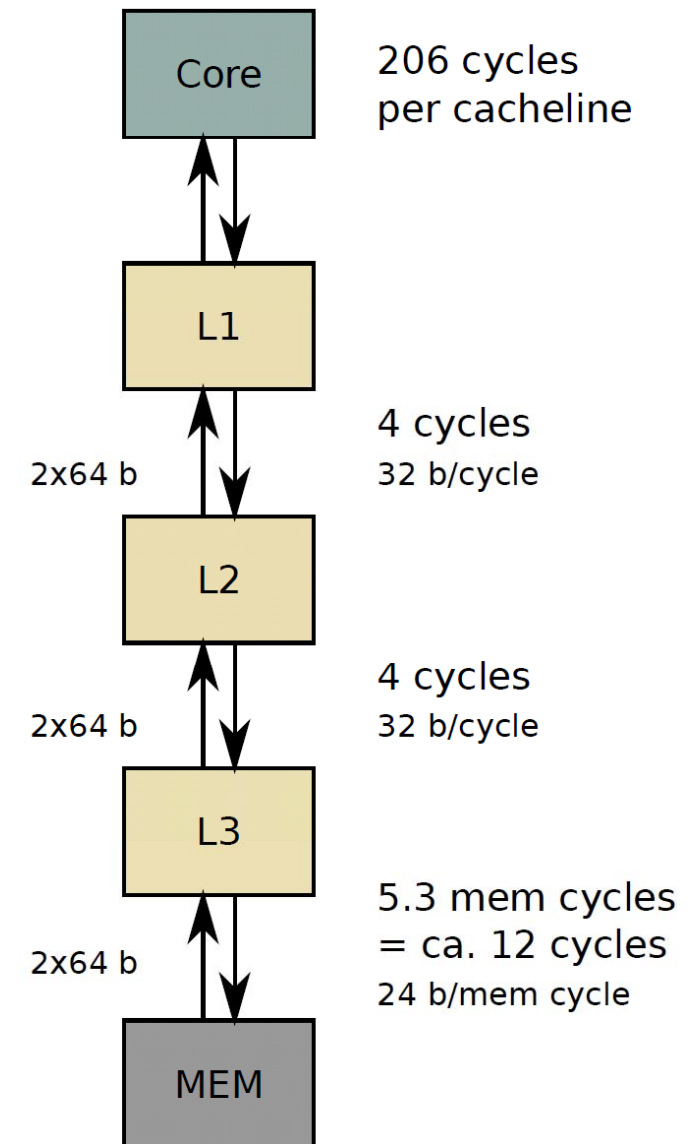
2. If the maximum memory bandwidth is not reached, SMT may help since it can fill bubbles in the LOAD pipeline.



SMT myths: Facts and fiction (3)










- **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	