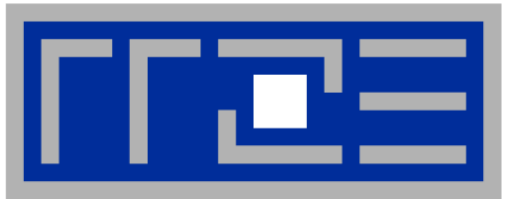


Performance-oriented programming on multicore-based systems, with a focus on the Cray XE6

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

Cray XE6 optimization workshop, November 5-8, 2012, HLRS

The Rules™

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

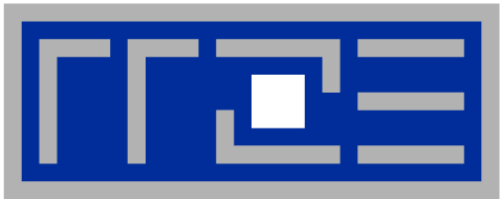
Without performance modeling, optimizing code is like stumbling in the dark

Performance x Flexibility = constant

a.k.a. Abstraction is the natural enemy of efficiency

Agenda

- **Basics of multicore processor and node architecture**
- **Probing node topology with likwid-topology**
- **Data access on modern processors**
  - Basic performance benchmarks and properties
  - The balance metric: Bandwidth-based performance modeling
  - Optimizing data access by code transformations
- **Enforcing affinity in multicore environments**
- **Performance properties of parallel code on multicore processors and nodes**
  - Exploration by microbenchmarks
  - Sparse matrix-vector multiplication
- **Microarchitectural features of modern processors**
  - SIMD parallelism
  - A closer look at the cache hierarchy
  - Performance modeling on the microarchitecture level
- **ccNUMA: Properties and efficient programming**

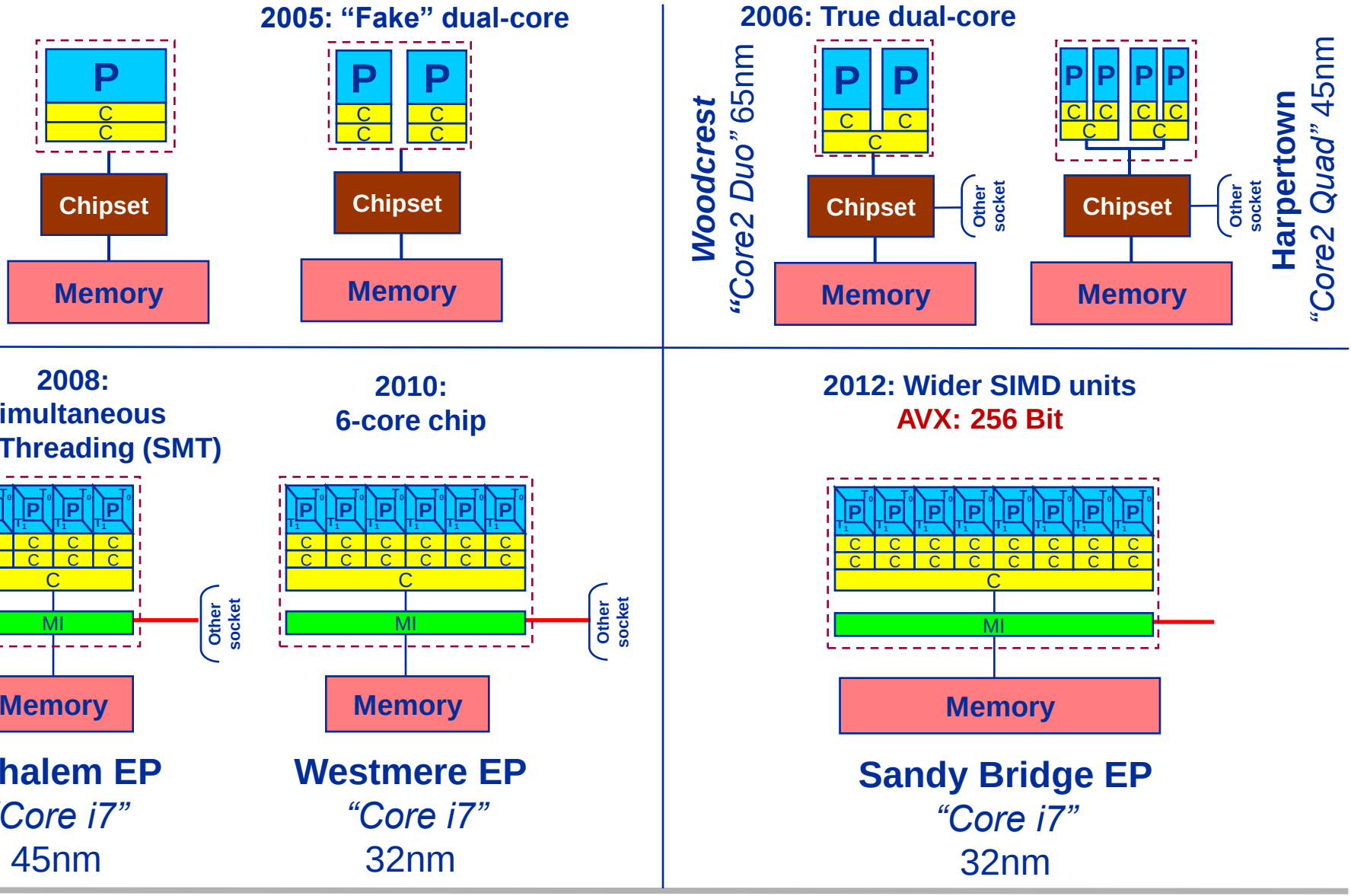


Multicore processor and system architecture

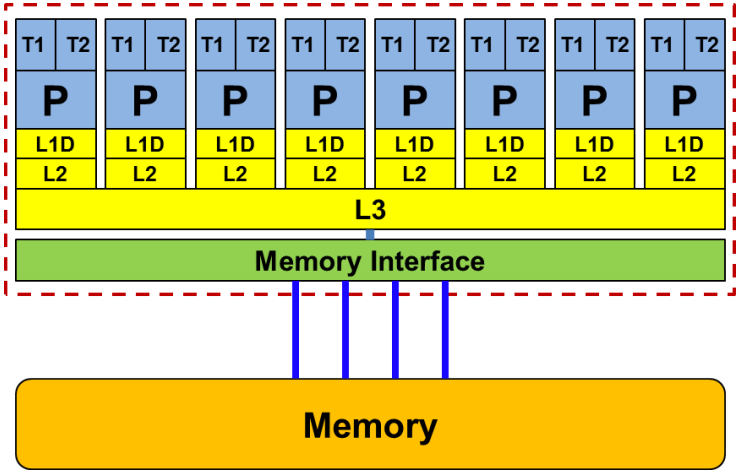
Basics

The x86 multicore evolution so far

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



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: (1 MULT and 1 ADD)	2
S	FP ops / instruction: (256 Bit SIMD registers – "AVX")	4 (dp) / 8 (sp)
ν	Clock speed :	2.5 GHz

TOP500 rank 1 (1996)

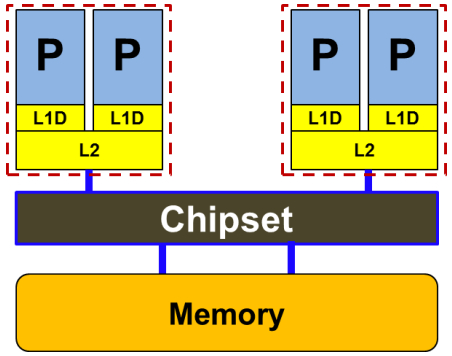
P = 160 GF/s (dp) / 320 GF/s (sp)

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

From UMA to ccNUMA

Basic architecture of commodity compute cluster nodes

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

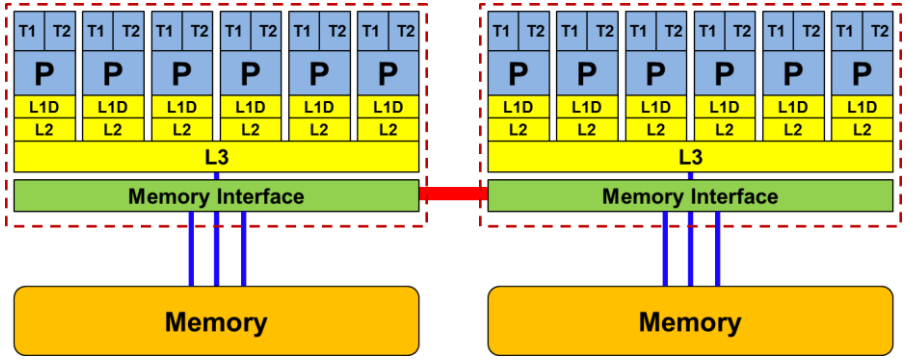


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system "anisotropy"

Today: Dual-socket Intel (Westmere) node:



Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

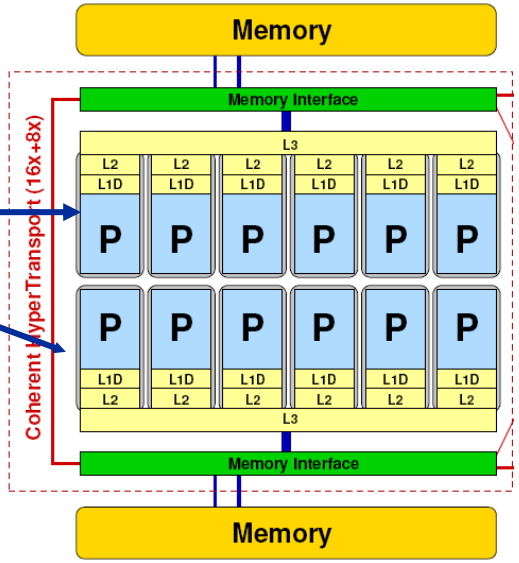
HT / QPI provide scalable bandwidth at the price of ccNUMA architectures: *Where does my data finally end up?*

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

Back to the 2-chip-per-case age

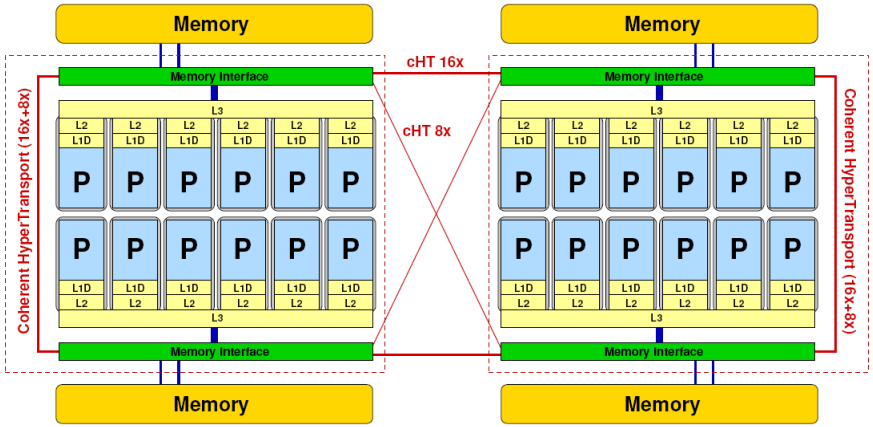
12 core AMD Magny-Cours – a 2x6-core ccNUMA socket

- **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



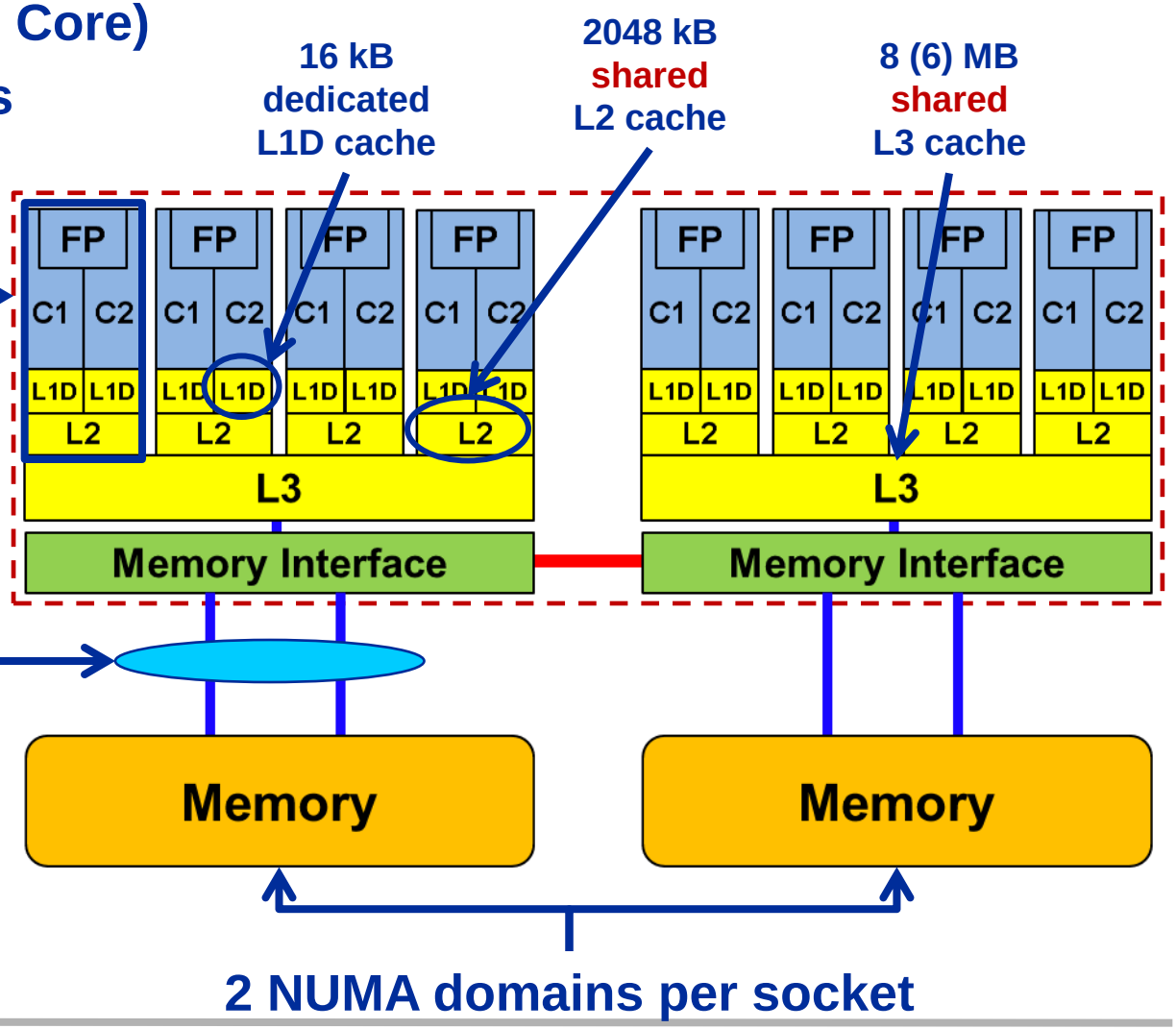
  - 4 socket server: → 8 NUMA domains

- **WHY? → Shared resources are hard 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



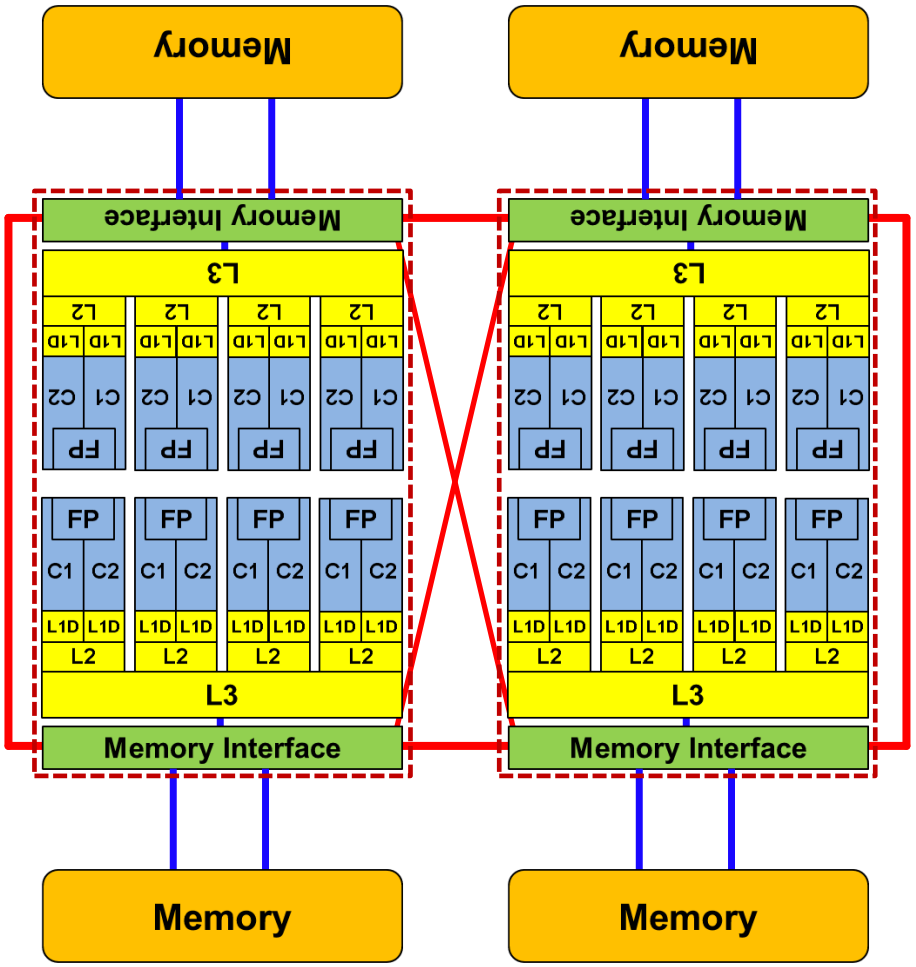
Each Bulldozer module:

- 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

Cray XE6 (Hermit) “Interlagos” 16-core dual socket node



- **Two 8- (integer-) core chips per socket @ 2.3 GHz (3.3 @ turbo)**
- **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)**

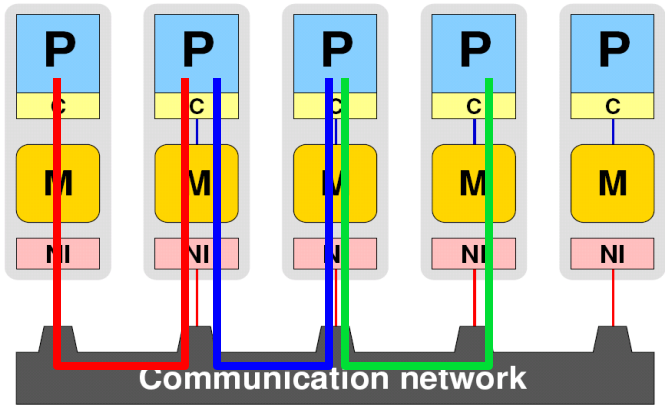
Parallel programming models
on multicore multisocket nodes

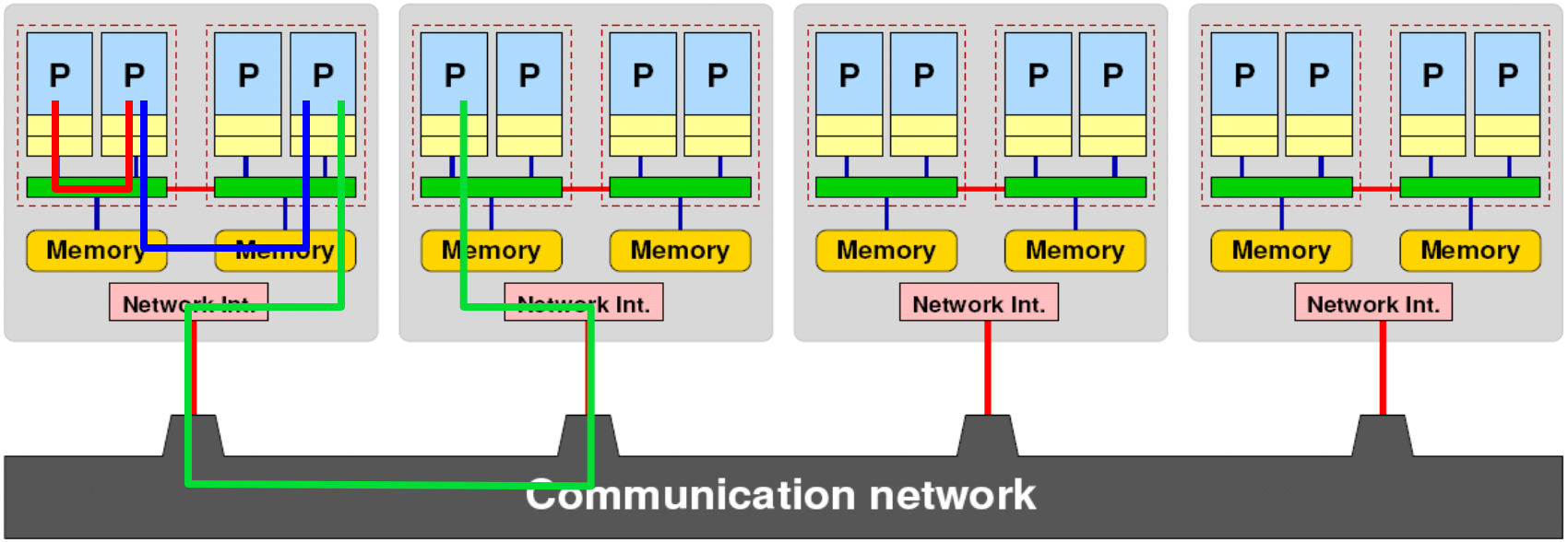
- **Shared-memory (intra-node)**
  - Good old MPI (current standard: 2.2)
  - OpenMP (current standard: 3.0)
  - POSIX threads
  - Intel Threading Building Blocks
  - Cilk++, OpenCL, StarSs,… you name it
- **Distributed-memory (inter-node)**
  - MPI (current standard: 2.2)
  - PVM (gone)
- **Hybrid**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

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

Parallel programming models:
Pure MPI

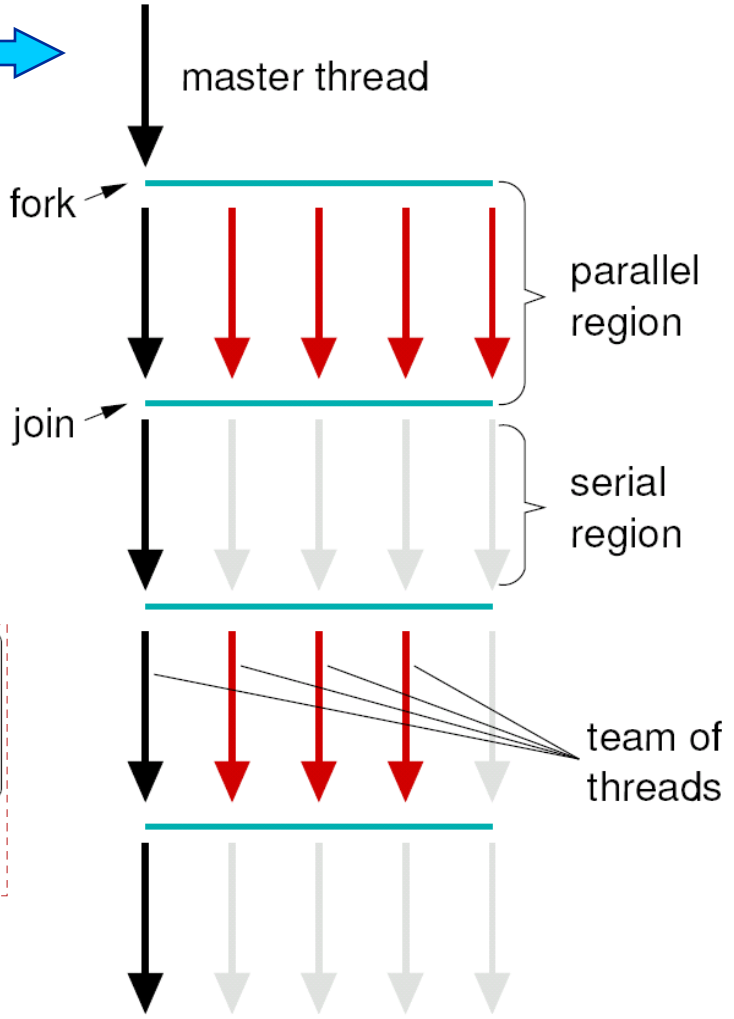
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

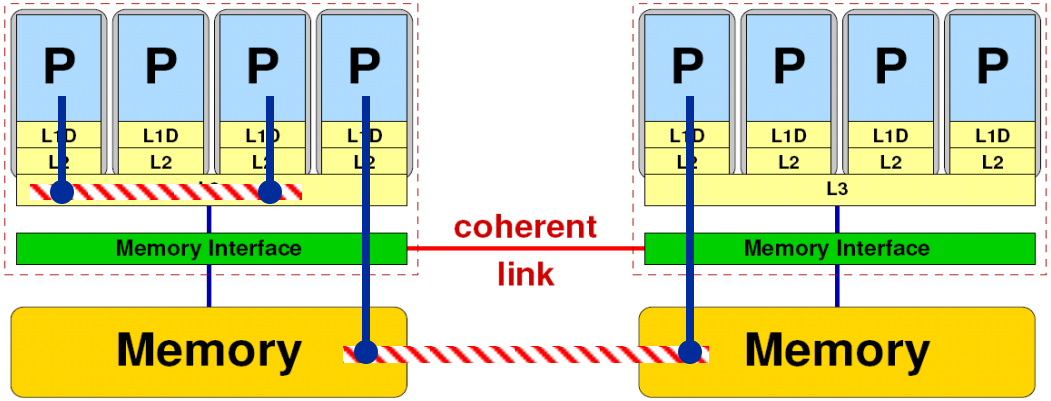




Parallel programming models:
Pure threading on the node

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





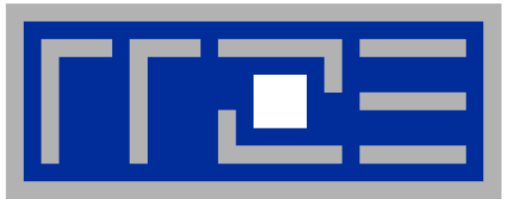
Parallel programming models:
Hybrid MPI+OpenMP on a multicore multisocket cluster

One MPI process / node

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

OpenMP threads pinned **“round robin”** across cores in node

Two MPI processes / socket
OpenMP threads on same socket



Probing node topology

- Standard tools
- likwid-topology

How do we figure out the node topology?

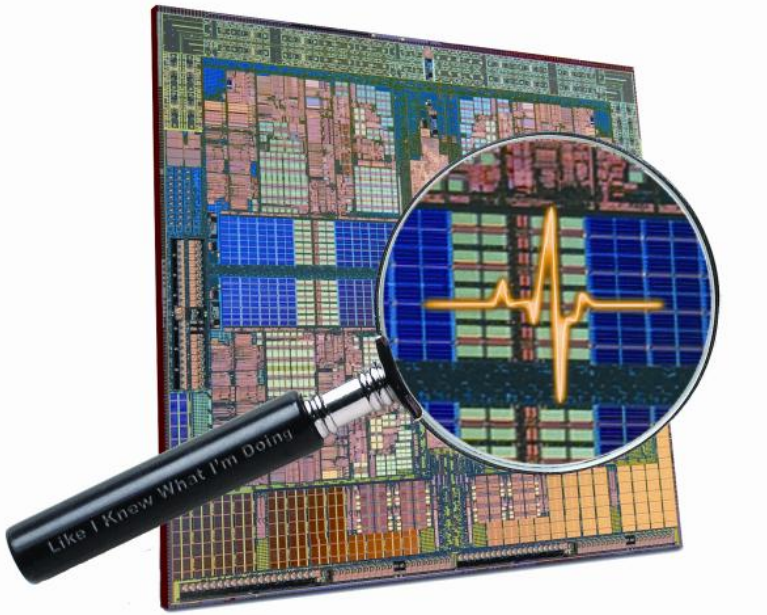
- **Topology =**
  - Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads (“logical cores”) share a physical core?
- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers 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
```

How do we figure out the node topology?

- **LIKWID tool suite:**

  Like
  I
  Knew
  What
  I'm
  Doing



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

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

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

Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- **Current tools:**
  - **likwid-topology**: Print thread and cache topology
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr:** Measure performance counters
  - **likwid-mpirun**: mpirun wrapper script for easy LIKWID integration
  - **likwid-bench**: Low-level bandwidth benchmark generator tool
  - … some more

likwid-topology – Topology information

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

Output of `likwid-topology -g`

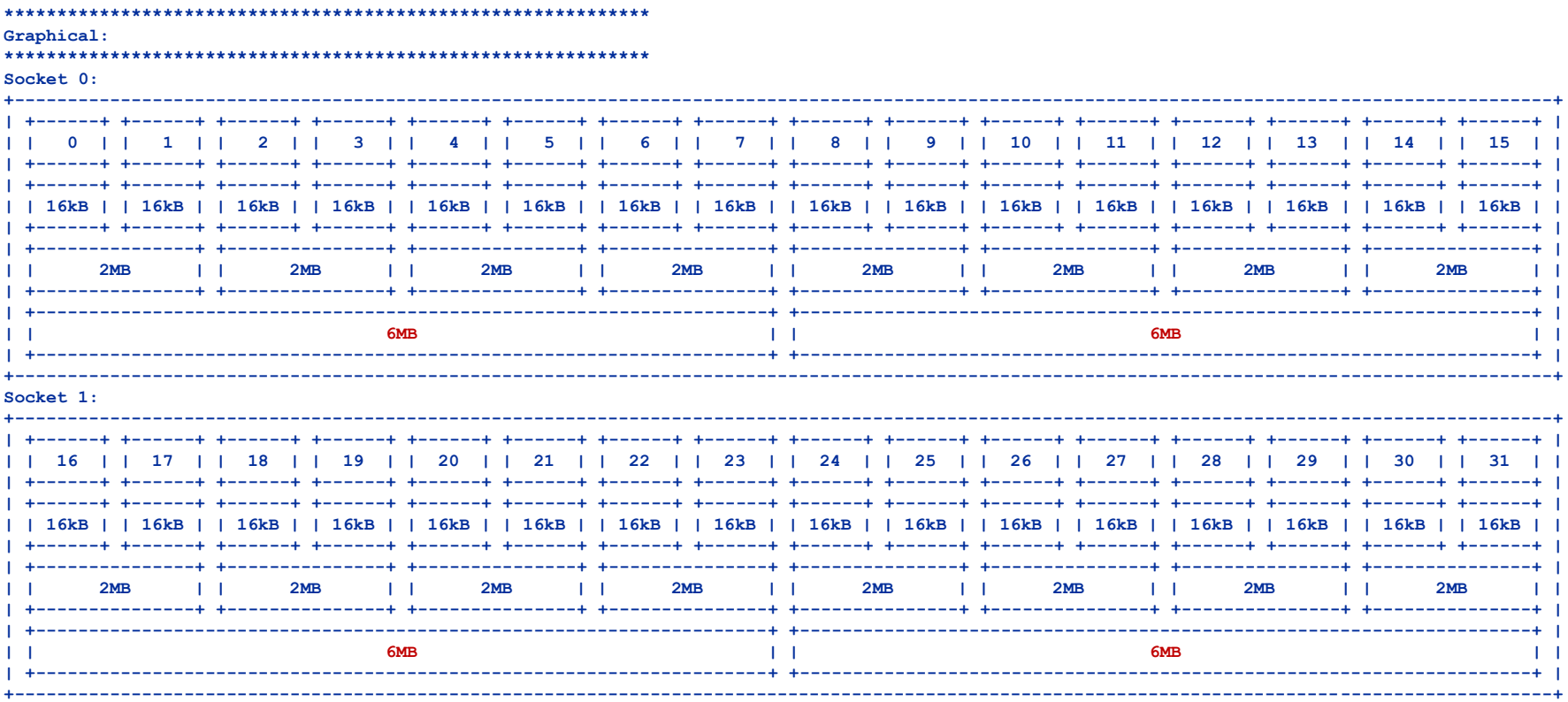
on one node of Cray XE6 "Hermit"

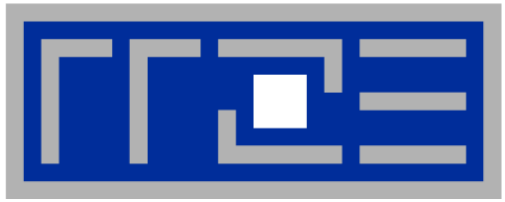
```
--------------------------------------------------------------
CPU type:       AMD Interlagos processor
**************************************************************
Hardware Thread Topology
**************************************************************
Sockets:                2
Cores per socket:       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
--------------------------------------------------------------
```

Output of likwid-topology continued

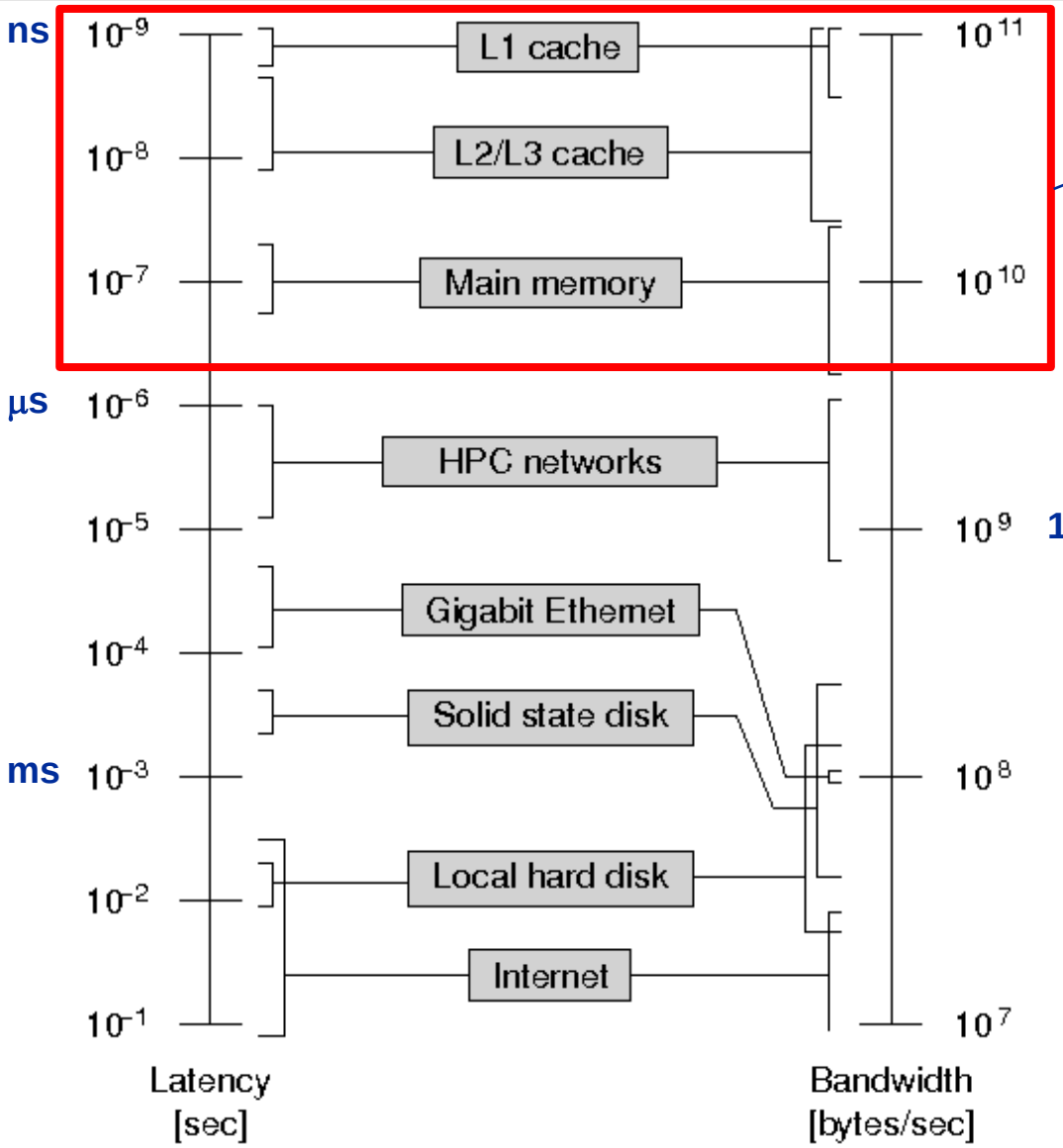




Data access on modern processors

Characterization of memory hierarchies
Balance analysis and light speed estimates
Data access optimization

Latency and bandwidth in modern computer environments



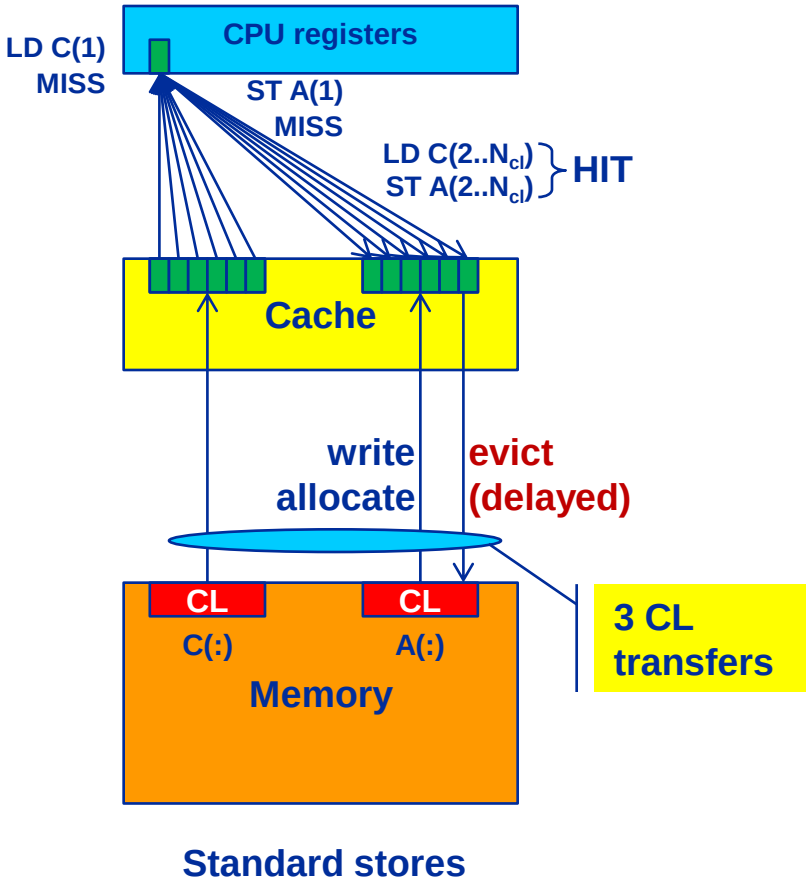
We care about this region today

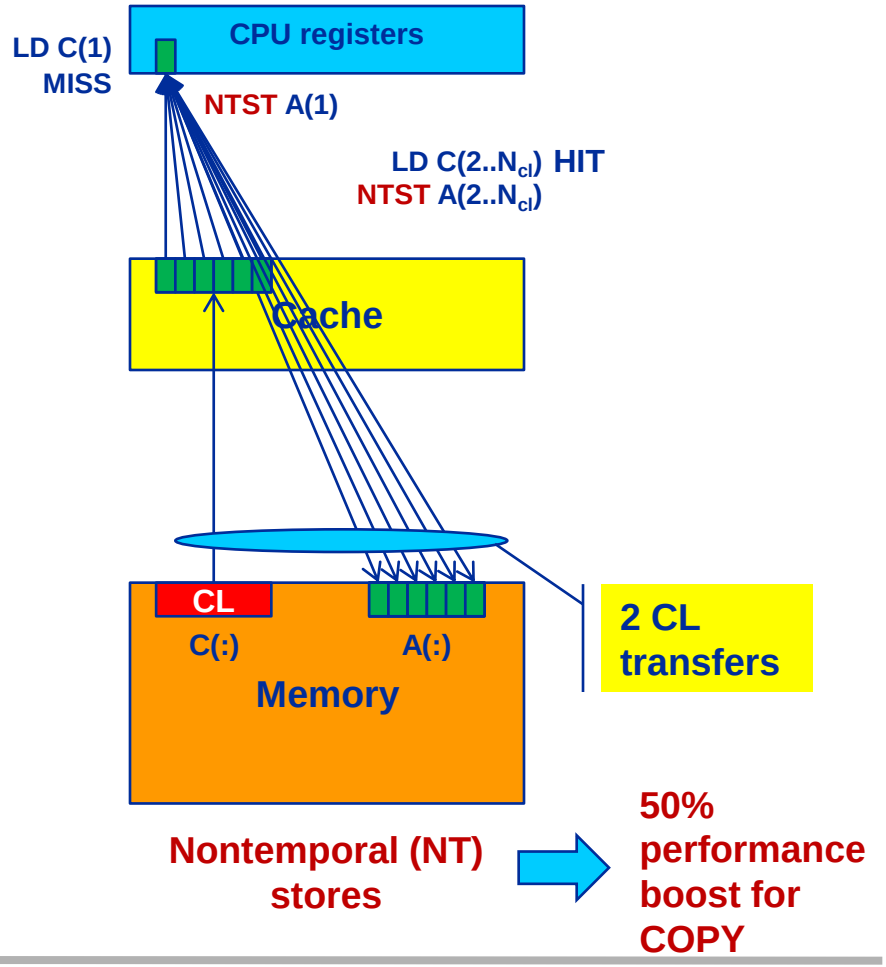
1 GB/s

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(:)`





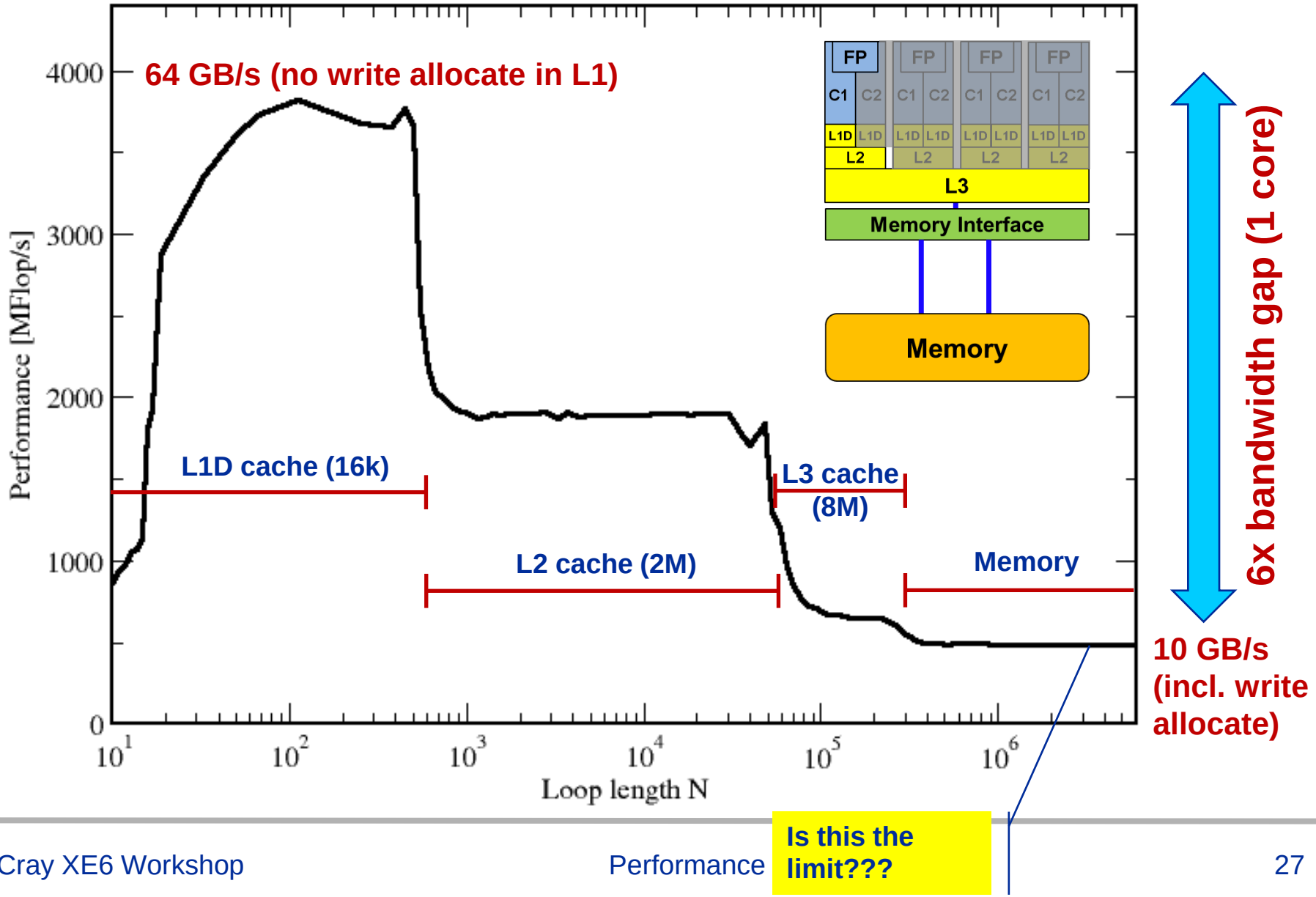
Simple streaming benchmark:

```
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

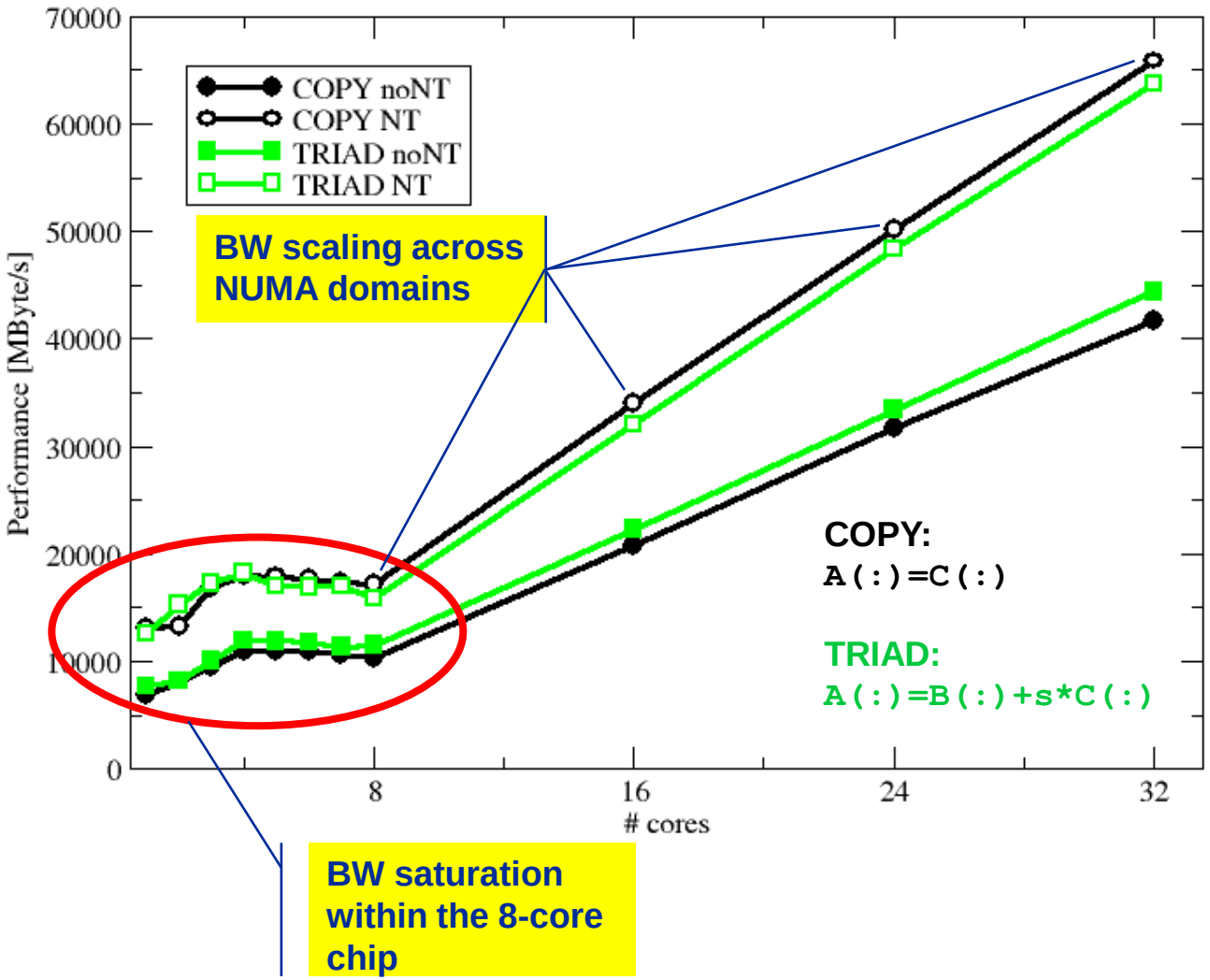
- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!

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



STREAM benchmarks:

Memory bandwidth on Cray XE6 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**

Balance metric: Machine balance

- **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) GW/s} / {4 x 2.3 x 8 GFlop/s} ~**0.029 W/F**

  Intel Sandy Bridge EP (2.7 GHz): ~**0.025 W/F**

  NEC SX9 (vector): ~**0.3 W/F**

  nVIDIA GTX480 ~**0.026 W/F**

Machine Balance: Typical values beyond main memory

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 ←→ 64-Bit

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

Balance metric: Code balance & lightspeed estimates

- **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"):**
  $l = 1$ → 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) Words / 2 Flops = 2.5 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)**

Balance metric (a.k.a. the “roofline model”)

- **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} → $P_{max}/2$)
  - 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

Balance metric: 2D diffusion equation + Jacobi solver

- **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)`

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

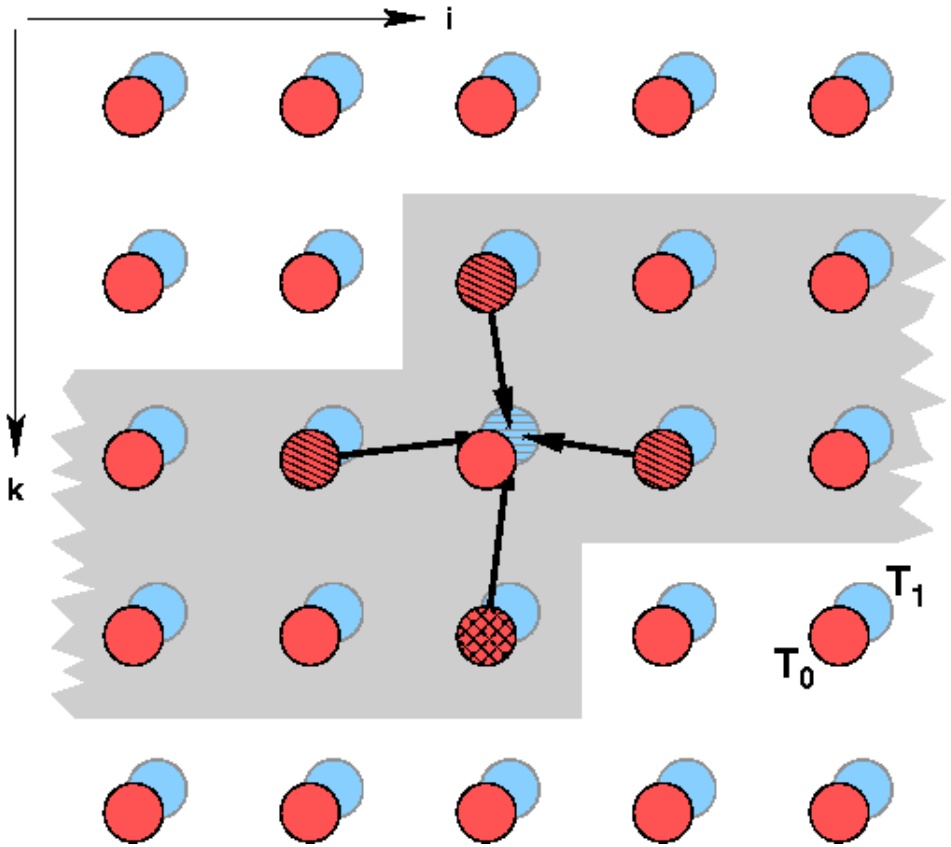
Balance (crude estimate incl. write allocate):

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

→ B_C = 5 W / 4 FLOPs = 1.25 W / F

Balance metric: 2 D Jacobi

- **Modern cache subsystems may further reduce memory traffic**



If cache is large enough to hold at least 2 rows (shaded region): Each `phi(:,:,t0)` is loaded once from main memory and reused 3 times from cache:

`phi(:,:,t0)`: 1 LD + `phi(:,:,t1)`: 1 ST+ 1LD
→B_C = 3 W / 4 F = 0.75 W / F

If cache is large enough to hold at least one row `phi(:,k-1,t0)` needs to be reloaded:

`phi(:,:,t0)`: 2 LD + `phi(:,:,t1)`: 1 ST+ 1LD
→B_C = 4 W / 4 F = 1.0 W / F

Beyond that:
`phi(:,:,t0)`: 2 LD + `phi(:,:,t1)`: 1 ST+ 1LD
→B_C = 5 W / 4 F = 1.25 W / F

Performance metrics: 2D Jacobi

- **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

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

Balance metric for 3D Jacobi

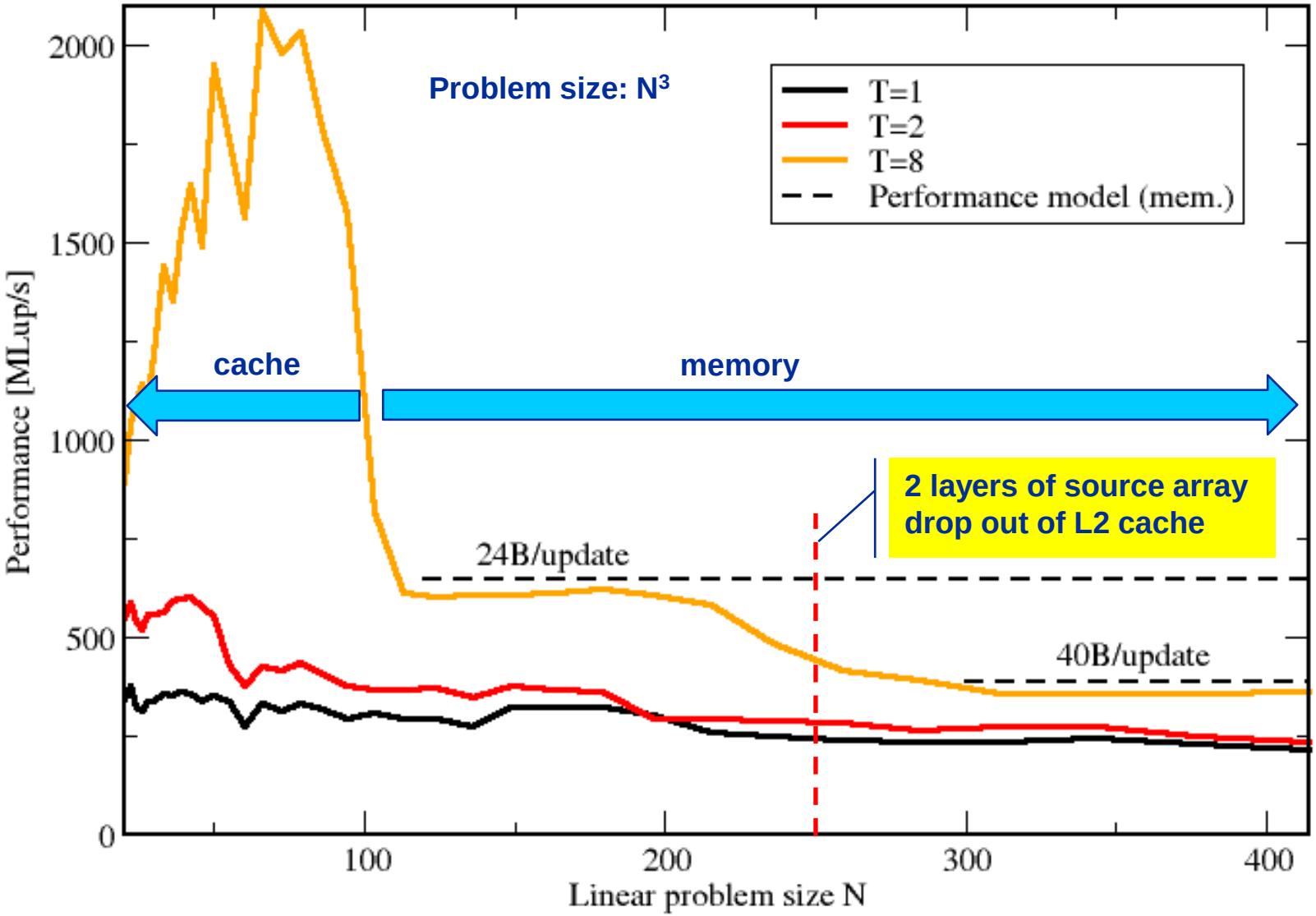
- 3D sweep:

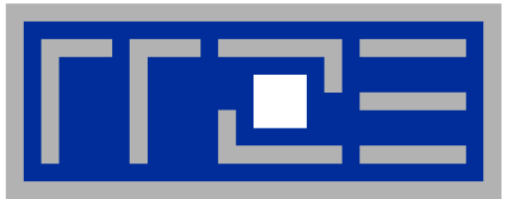
```
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      phi(i,j,k,t1) = oos *(phi(i-1,j,k,t0)+phi(i+1,j,k,t0) &
                          + phi(i,j-1,k,t0)+phi(i,j+1,k,t0) &
                          + phi(i,j,k-1,t0)+phi(i,j,k+1,t0))
    enddo
  enddo
enddo
```

- 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 (24 bytes/update)
- If 2-layer condition does not hold but 2 rows fit:
  → B_C = 5/6 W/F (40 bytes/update)
- Worst case (2 rows do not fit): → B_C = 7/6 W/F (56 bytes/update)

3D Jacobi solver

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



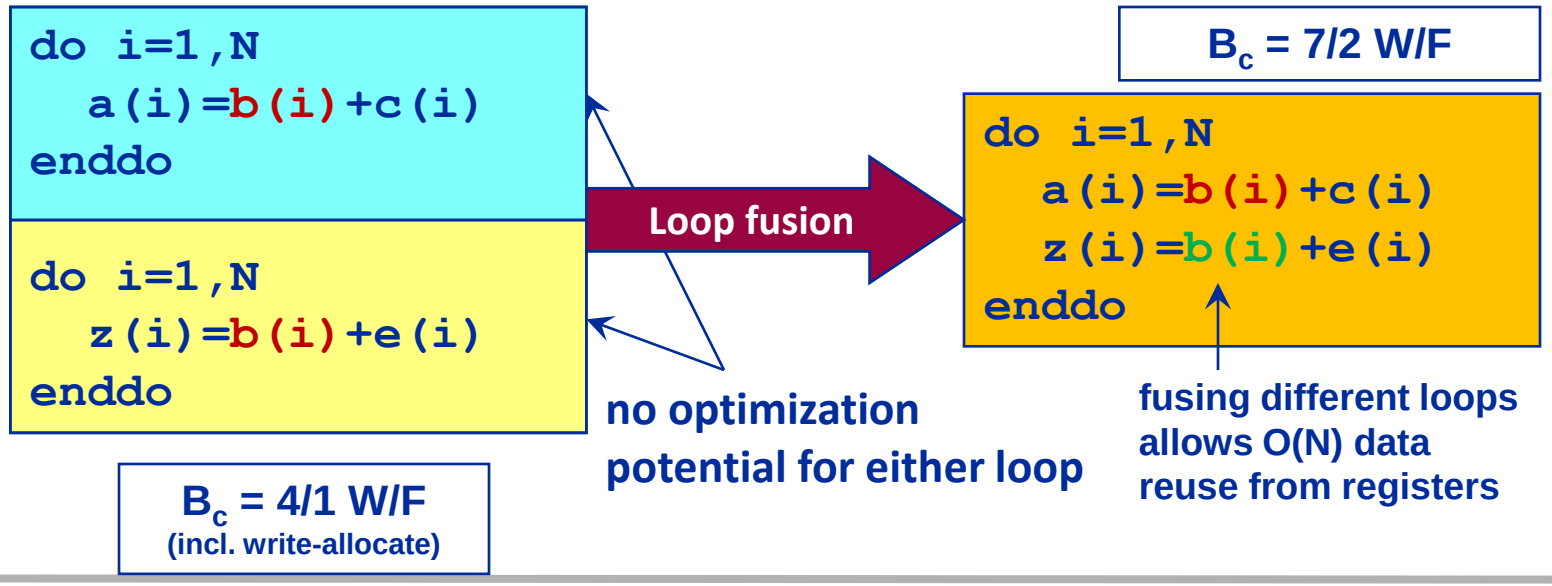


Data Access Optimizations

- General considerations
- Case study: Optimizing a Jacobi solver

Data access – general considerations

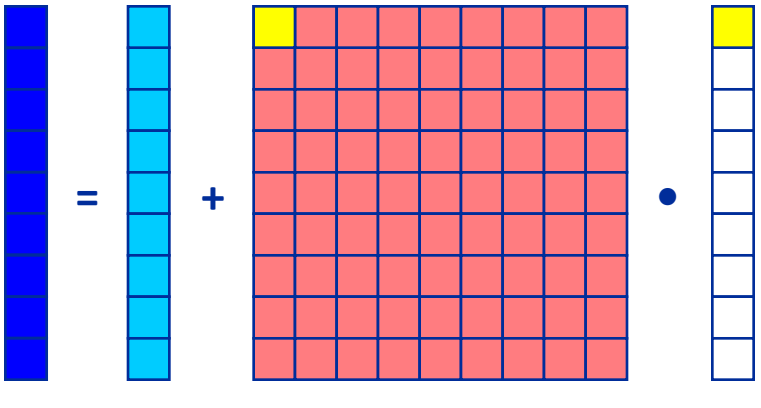
- **Case 1: O(N)/O(N) Algorithms**
  - O(N) arithmetic operations vs. O(N) data access operations
  - Examples: Scalar product, vector addition, sparse MVM etc.
  - Performance limited by memory BW for large N ("memory bound")
  - Limited optimization potential for single loops
    - …at most a constant factor for multi-loop operations
  - Example: successive vector additions



Data access – general guidelines

- **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!

Data access – general guidelines

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

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

unroll

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

jam

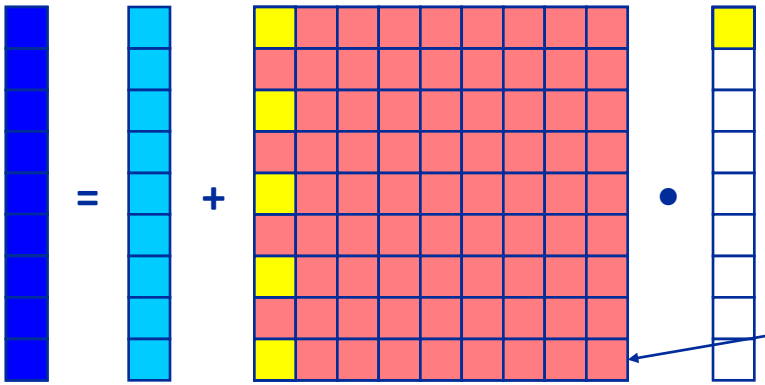
```
do i=1,N,2
 do j=1,N
  c(i)=c(i)+a(i,j)     * b(j)
  c(i+1)=c(i+1)+a(i+1,j)* b(j)
 enddo
enddo
```

b(j) can be re-used once from register → save 1 LD operation

Lowers B_c from 1 to ¾ W/F

Data access – general guidelines

- **O(N²)/O(N²) algorithms cont'd**
  - Data access pattern for 2-way unrolled dense MVM:



Vector `b[]` now only loaded N/2 times!

Remainder loop handled separately

  - 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 ($B_c \approx ½$ 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

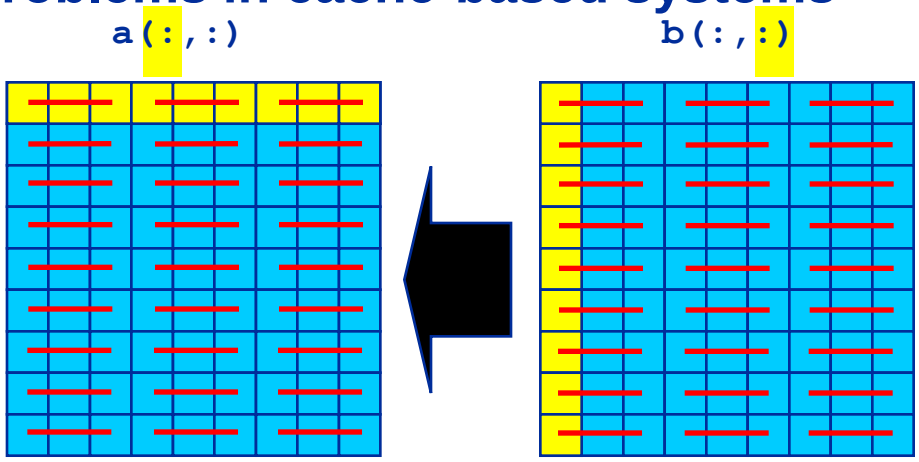
Optimizing data access for dense matrix transpose

Dense matrix transpose

optional

- **Simple example for data access problems in cache-based systems**
- **Naïve code:**

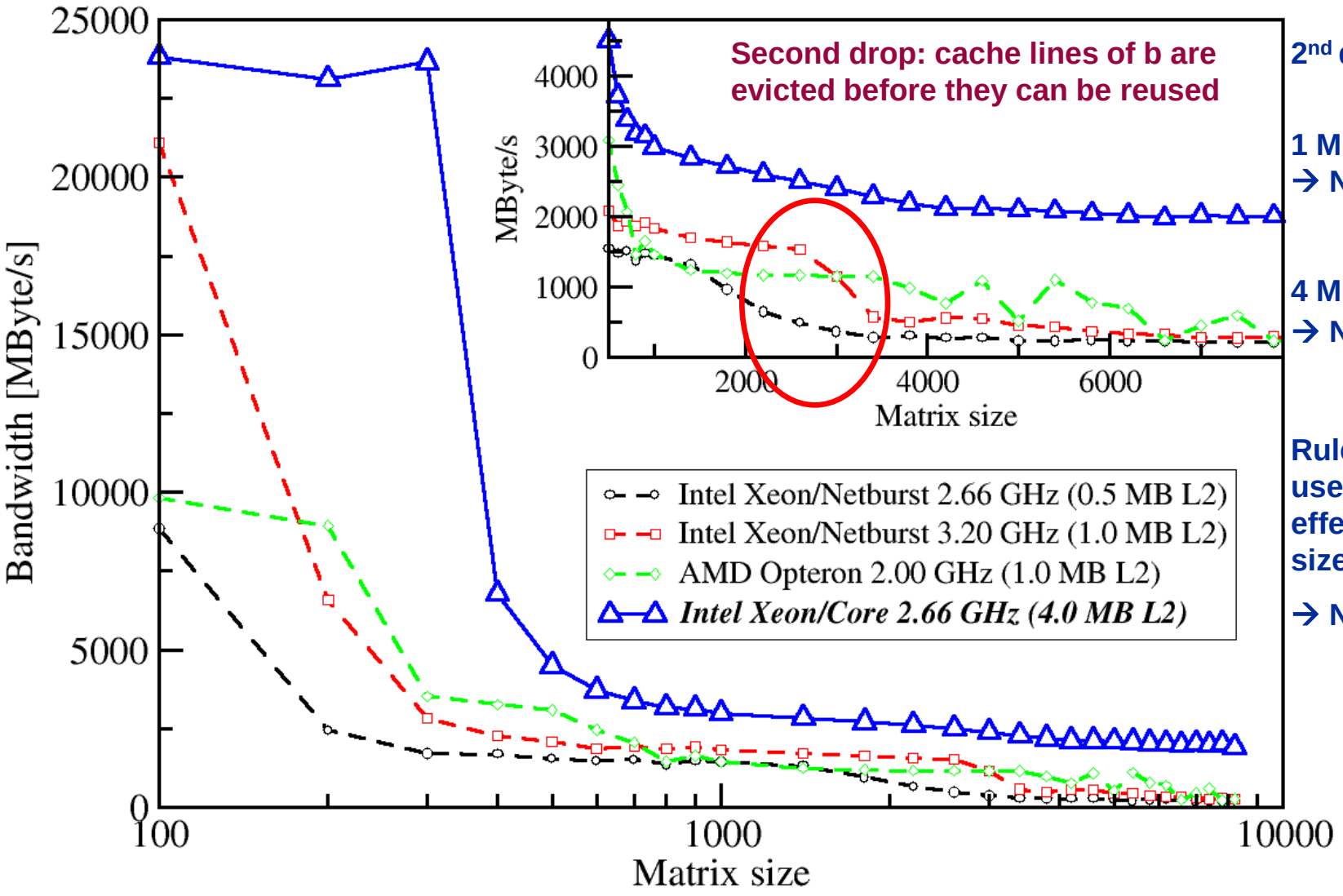
```
do i=1,N
 do j=1,N
  a(j,i) = b(i,j)
 enddo
enddo
```



- **Problem: Stride-1 access for a implies stride-N access for b**
  - Access to a is perpendicular to cache lines (—)
  - Possibly bad cache efficiency (spatial locality)
- **Three performance levels are expected:**
  - C: Cache size; L_C: Cache line length; both are given in double words (8 byte)
  - $2 * N^2 < C$: Both matrices stay in cache
  - $N * L_C + N < C$: N cache lines of b and one row of a stays in cache
  - $N * L_C + N > C$: Matrix b is reloaded from memory L_C times
- **Use outer loop unrolling blocking to reduce / avoid second drop**

Dense matrix transpose: Base version

optional



2nd drop:

1 MB cache
→ N_C ~ 7.530

4 MB cache
→ N_C ~ 30.000

Rule of thumb use C/2 as effective cache size

→ N_C → N_C/2

Dense matrix transpose: Unrolling and blocking

optional

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

unroll/jam

```
do i=1,N,U
 do j=1,N
  a(j,i)     = b(i,j)
  a(j,i+1)   = b(i+1,j)
  ...
  a(j,i+U-1) = b(i+U-1,j)
 enddo
enddo
```

block

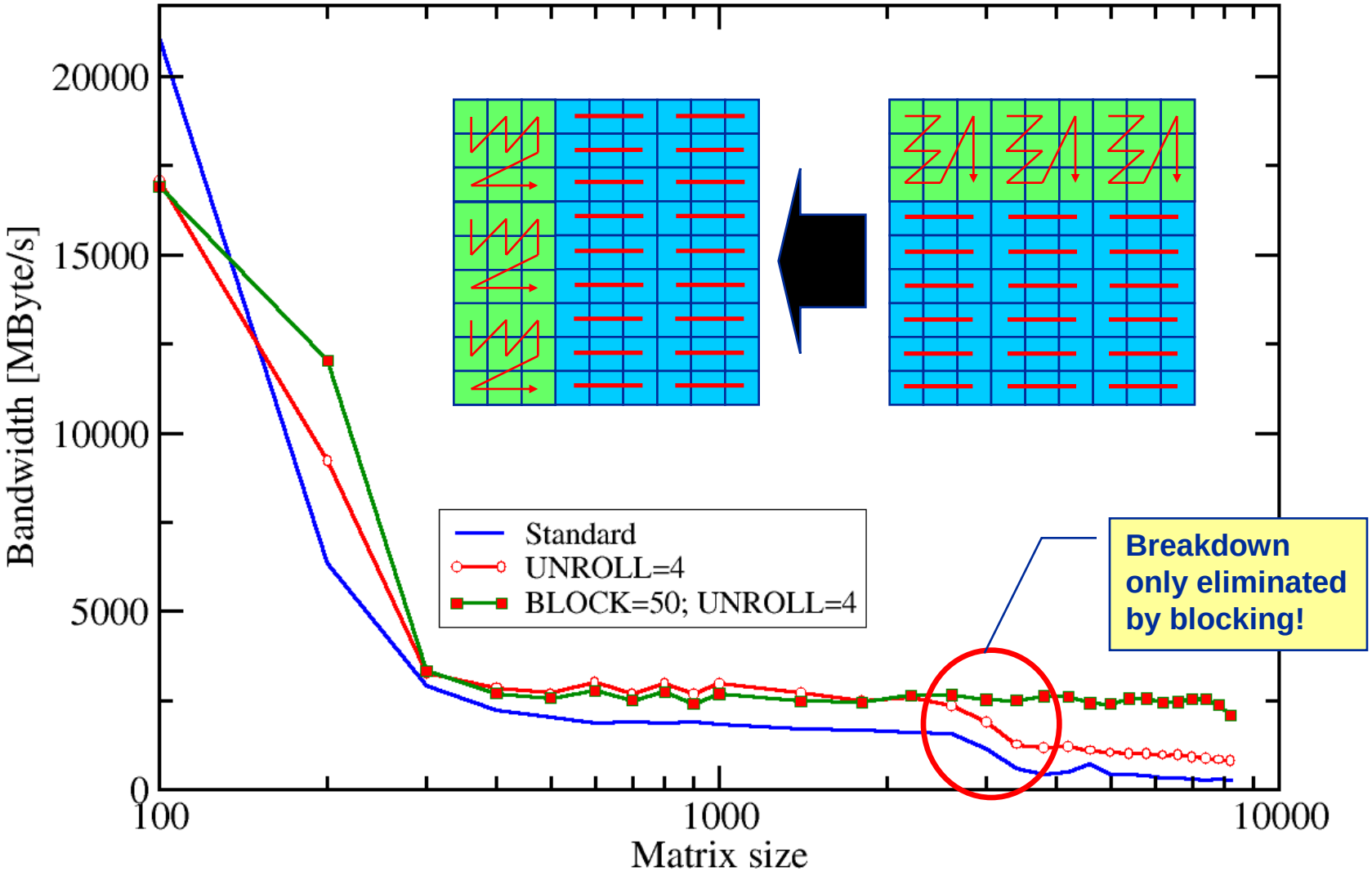
```
do ii=1,N,B
 istart=ii; iend=ii+B-1
 do jj=1,N,B
  jstart=jj; jend=jj+B-1
  do i=istart,iend,U
   do j=jstart,jend
    a(j,i)     = b(i,j)
    a(j,i+1)   = b(i+1,j)
    ...
    a(j,i+U-1) = b(i+U-1,j)
enddo;enddo;enddo;enddo
```

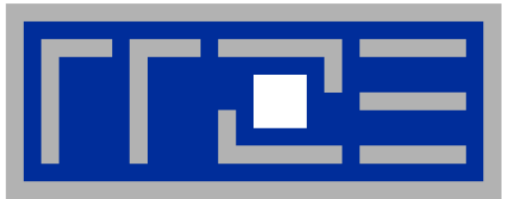
Blocking and unrolling factors (B,U) can be determined experimentally; be guided by cache sizes and line lengths

Dense matrix transpose: Blocked/unrolled versions

optional

- **Intel Xeon/Netburst 3.2 GHz**

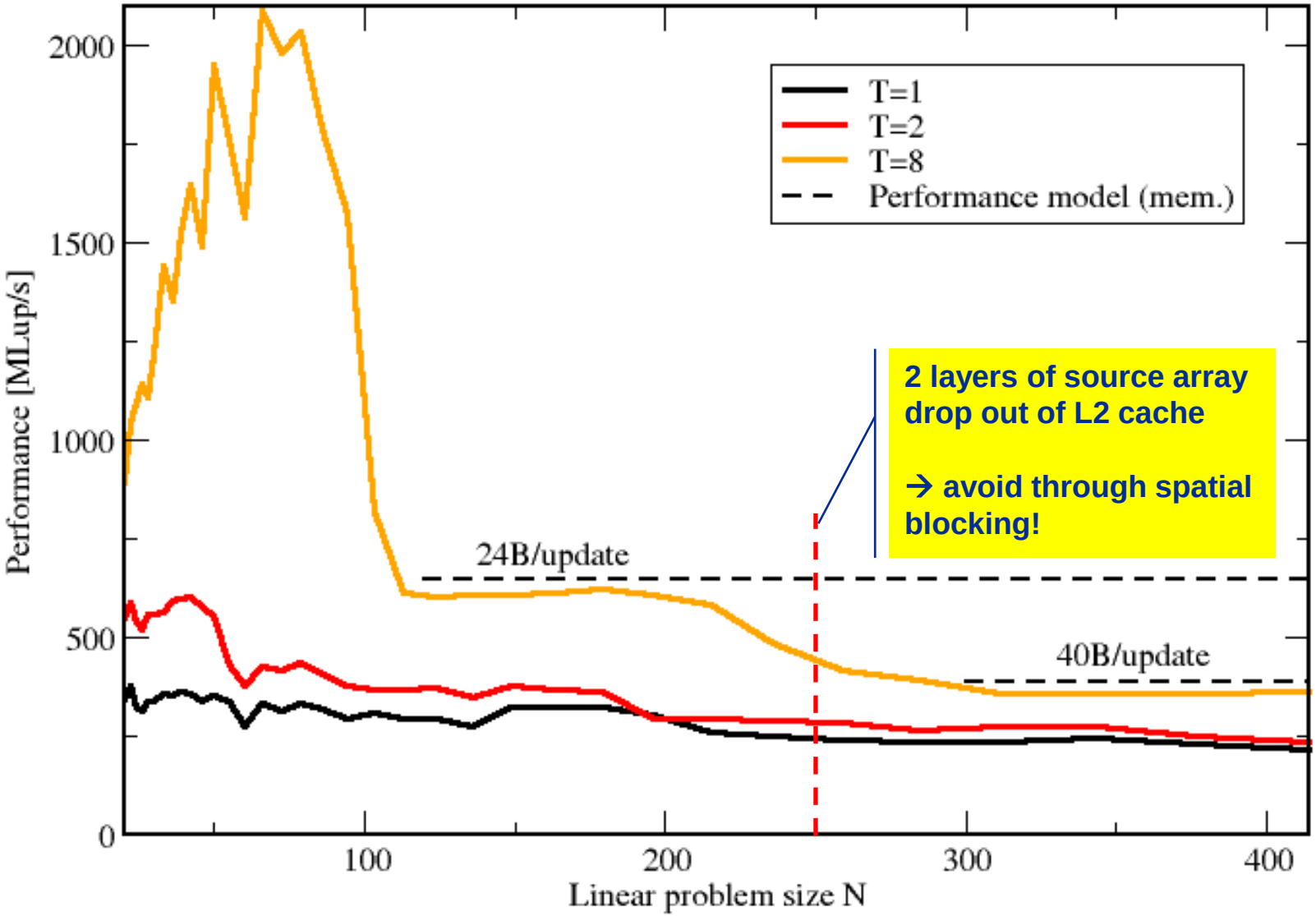




Case study:
3D Jacobi solver

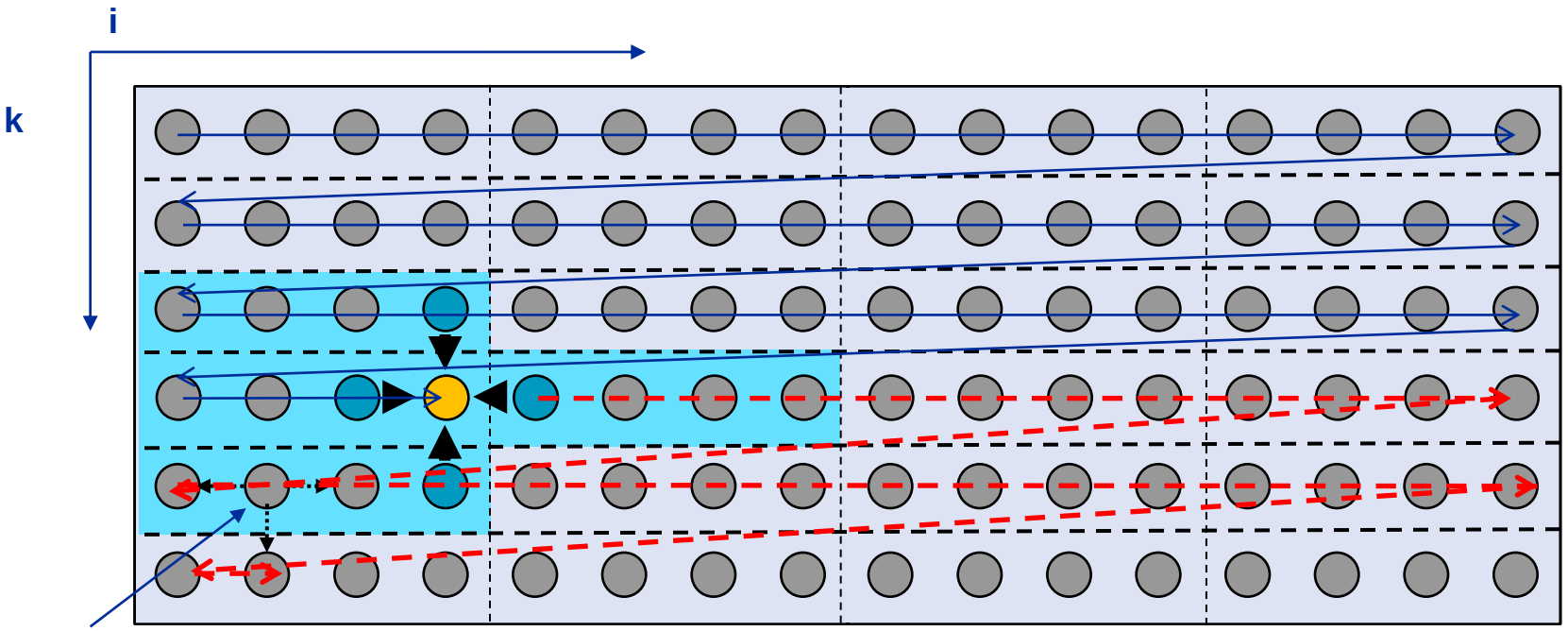
Spatial blocking for improved cache utilization

Remember the 3D Jacobi solver?



Jacobi iteration (2D): No spatial Blocking

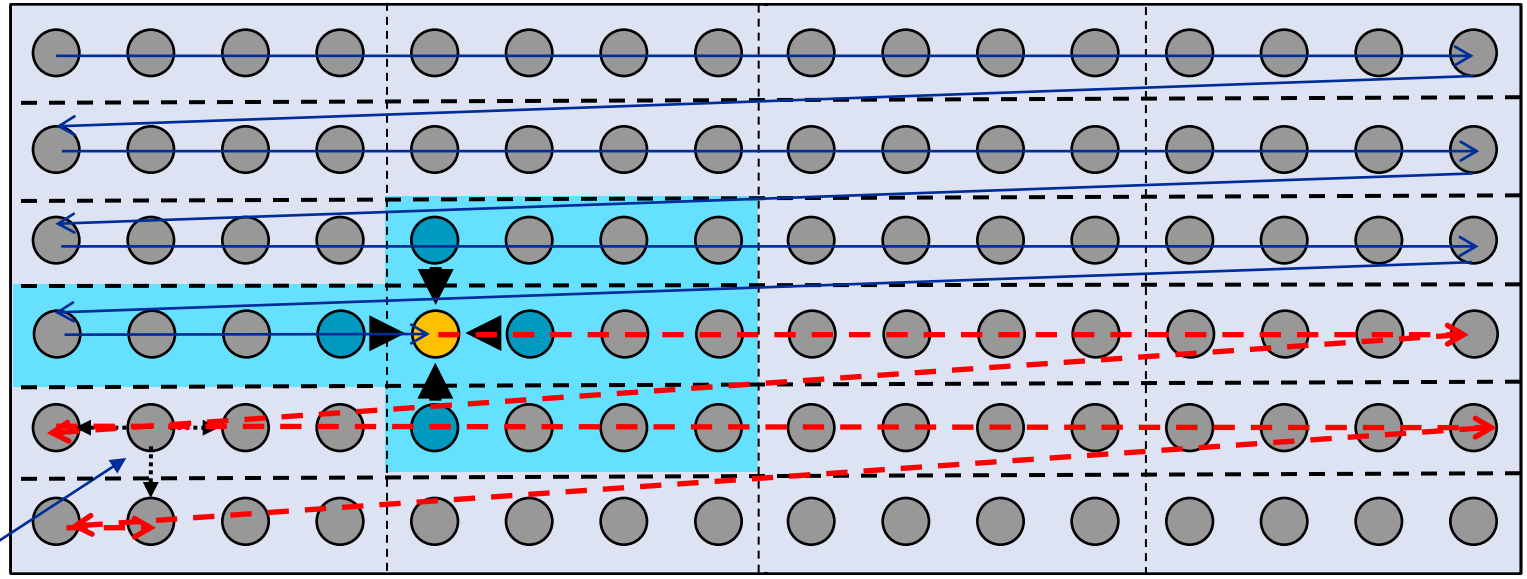
- **Assumptions:**
  - Cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array



This element is needed for three more updates; but 29 updates happen before this element is used for the last time

Jacobi iteration (2D): No spatial blocking

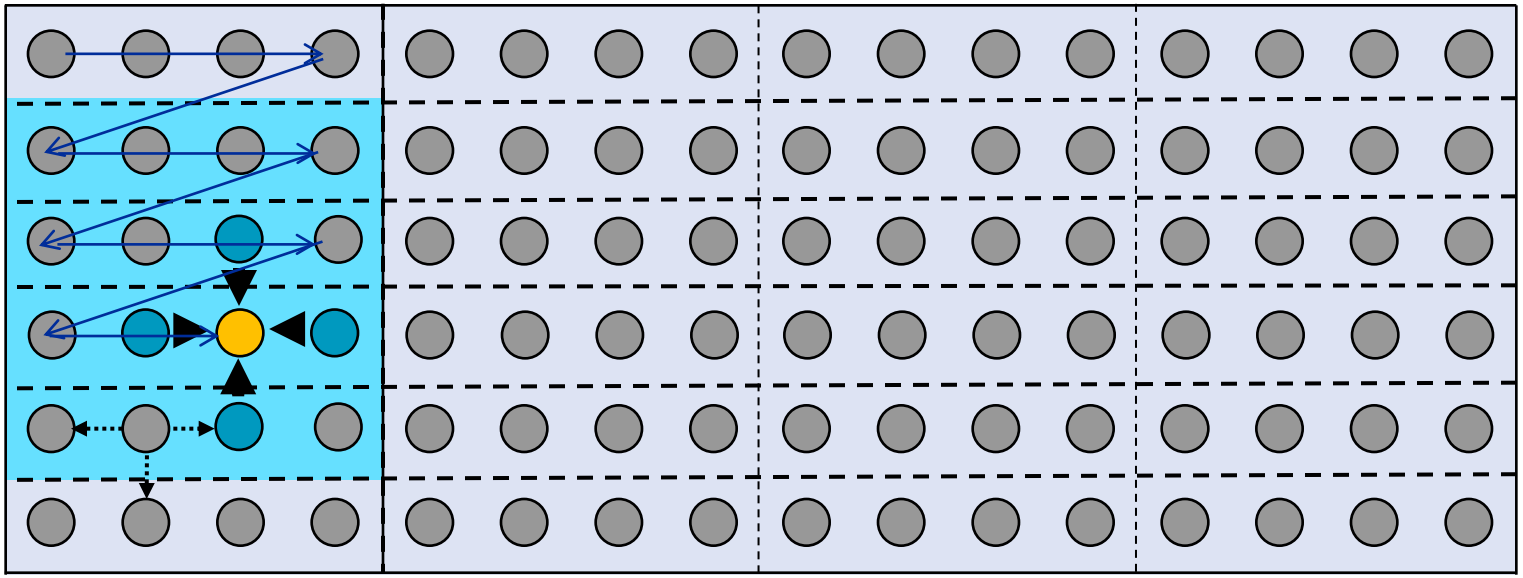
- **Assumptions:**
  - Cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array



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

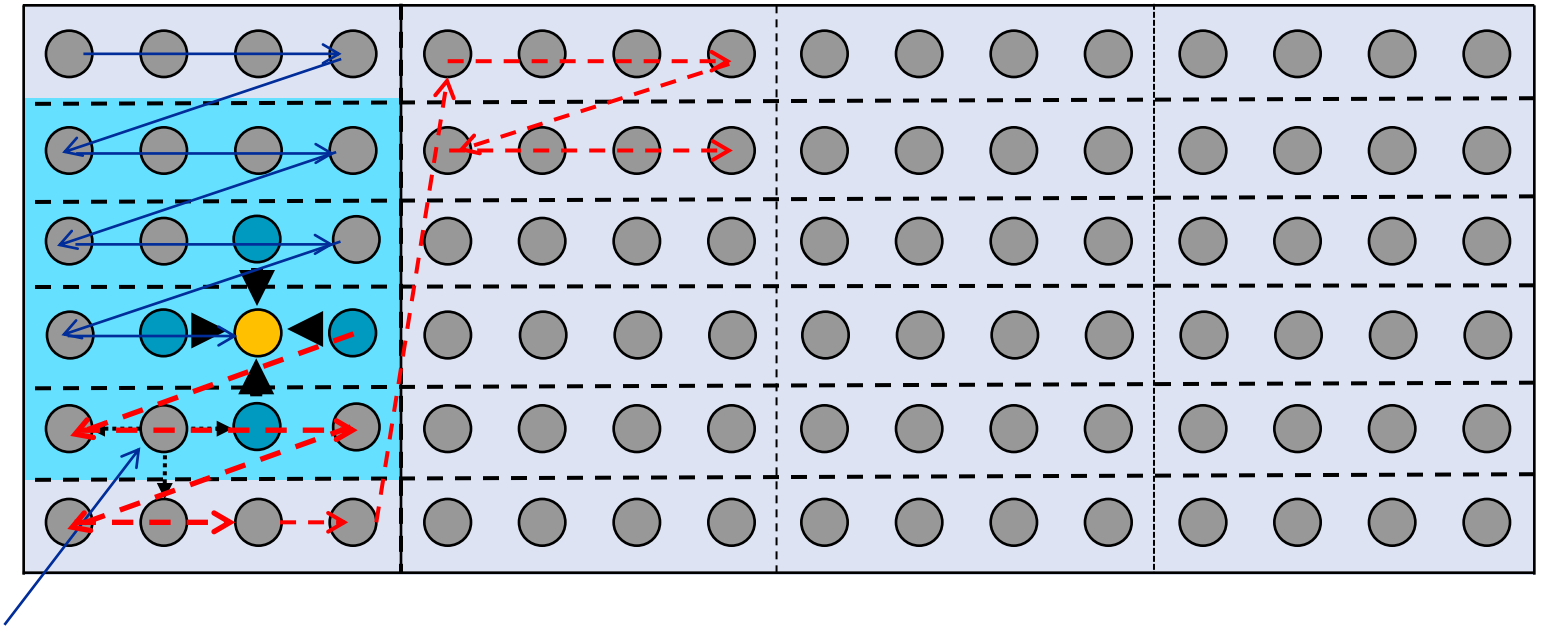


Jacobi iteration (2D): Spatial Blocking

- **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 (only 6 updates happen before last use of this element)

Jacobi iteration (2D): Spatial blocking

- **Implementation:**

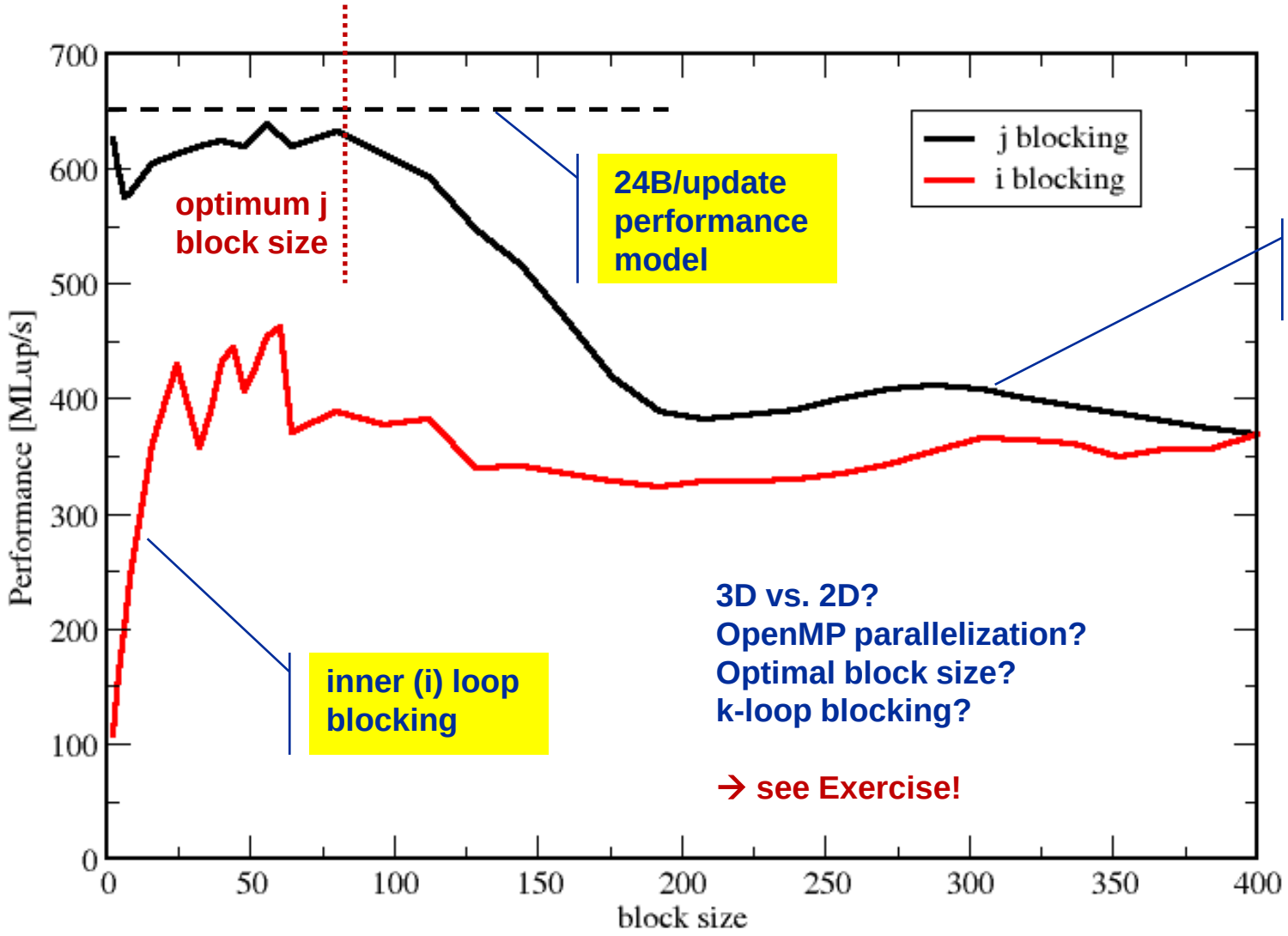
```
do it=1,itmax
  do ioffset=1,imax,iblock
    do k=1,kmax
      do i=ioffset, min(imax,ioffset+iblock-1)
        phi(i, k, t1) = ( phi(i-1, k, t0) + phi(i+1, k, t0)
                        + phi(i, k-1, t0) + phi(i, k+1, t0) )*0.25
enddo; enddo; enddo; enddo
```

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)

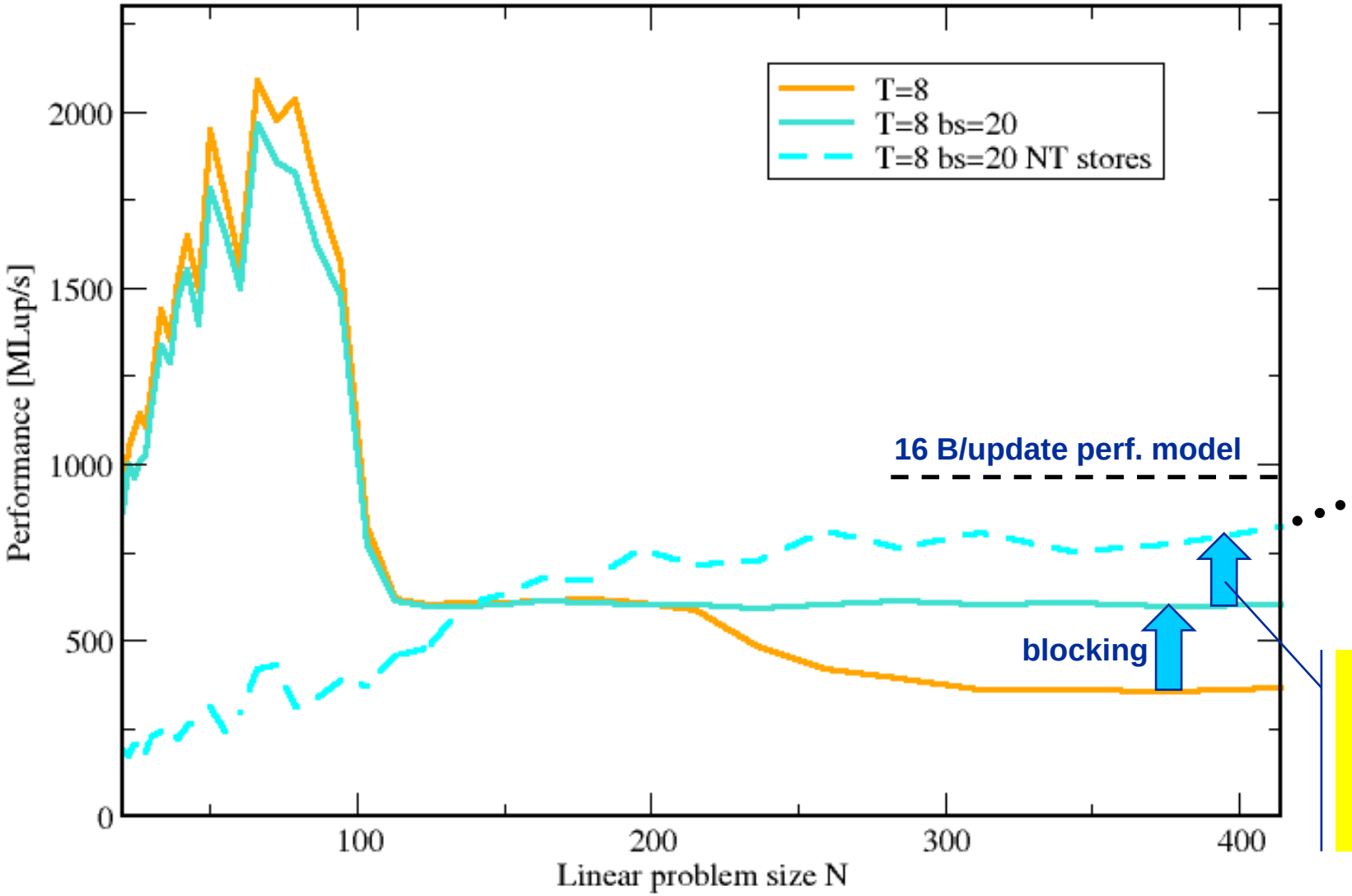
3D Jacobi solver (problem size 400^3)

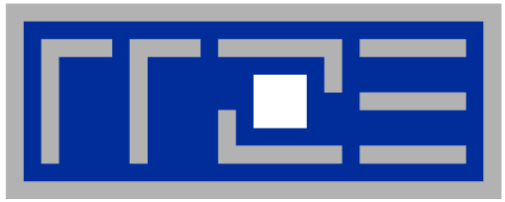
Blocking different loop levels (8 cores Interlagos)



3D Jacobi solver

Spatial blocking + nontemporal stores

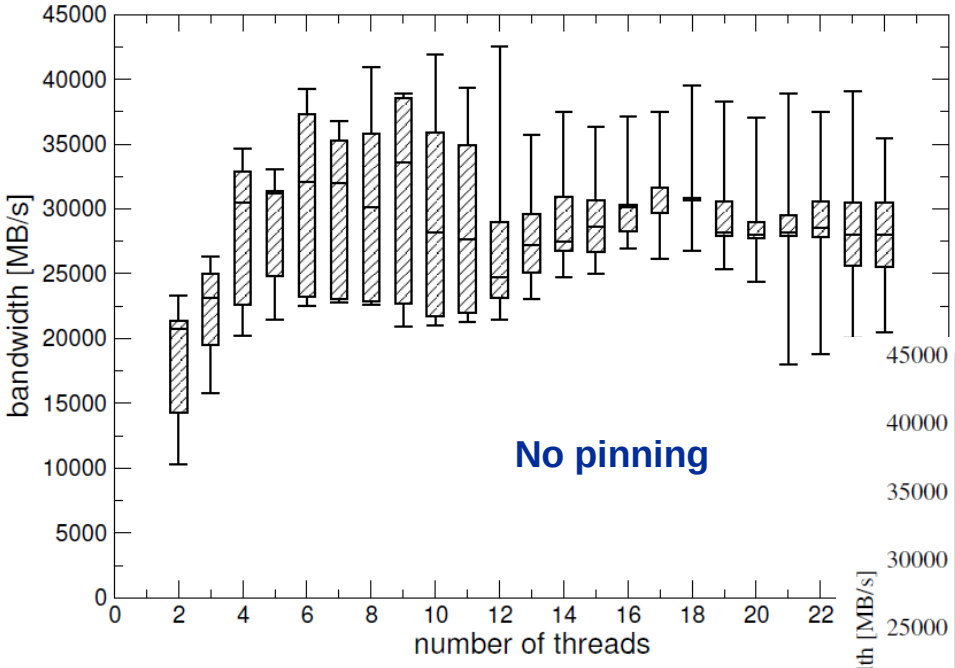


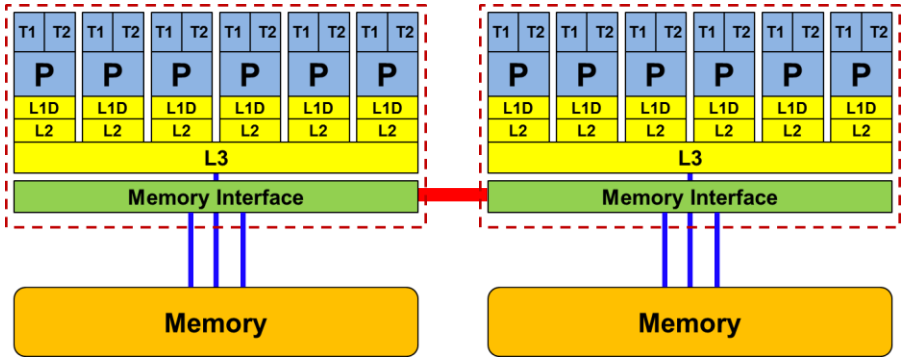


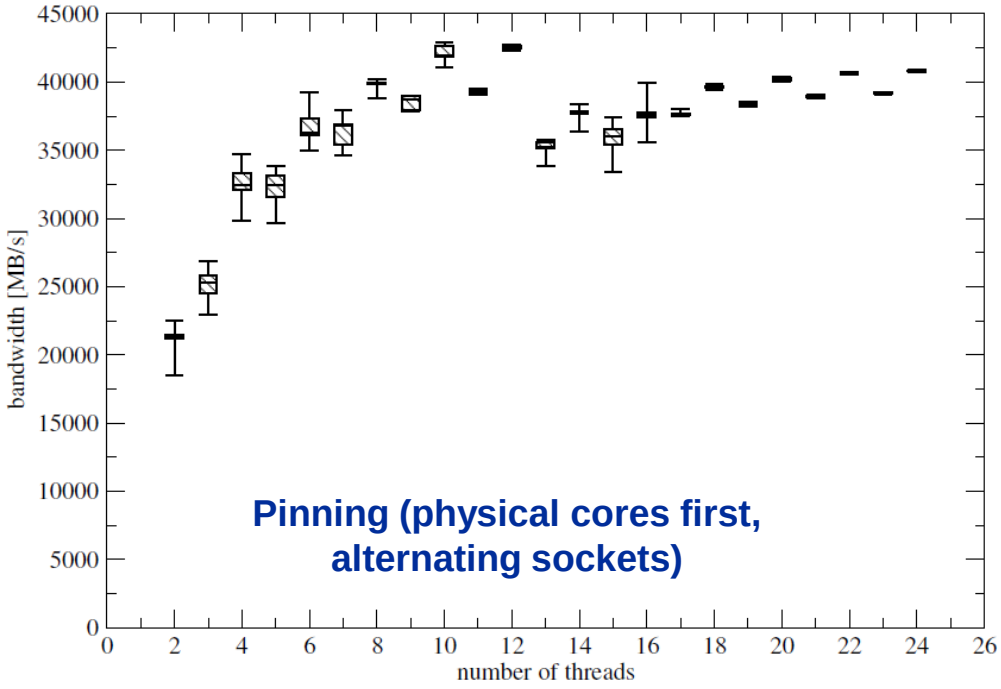
Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
- aprun (Cray)

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







There are several reasons for caring about affinity:

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

Generic thread/process-core affinity under Linux

Overview

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

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

```
taskset 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**

Generic thread/process-core affinity under Linux

- **Complementary tool:** `numactl`

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

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

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

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

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

More thread/Process-core affinity ("pinning") options

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

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

Example for program[illegible]trolled affinity: Using P[illegible]nder Linux!

SKIPPED

Likwid-pin

Overview

- **Inspired by and based on `ptoverride` (Michael Meier, RRZE) and `taskset`**
- **Pins processes and threads to specific cores without touching code**
- **Directly supports pthreads, gcc OpenMP, Intel OpenMP**
- **Allows user to specify skip mask (shepherd threads should not be pinned)**
- **Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!**
- **Can also be used as a superior replacement for taskset**
- **Supports logical core numbering within a node and within an existing CPU set**
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system
- **Configurable colored output**
- **Usage examples:**
  - **`likwid-pin -t intel -c 0,2,4-6  ./myApp parameters`**
  - **`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

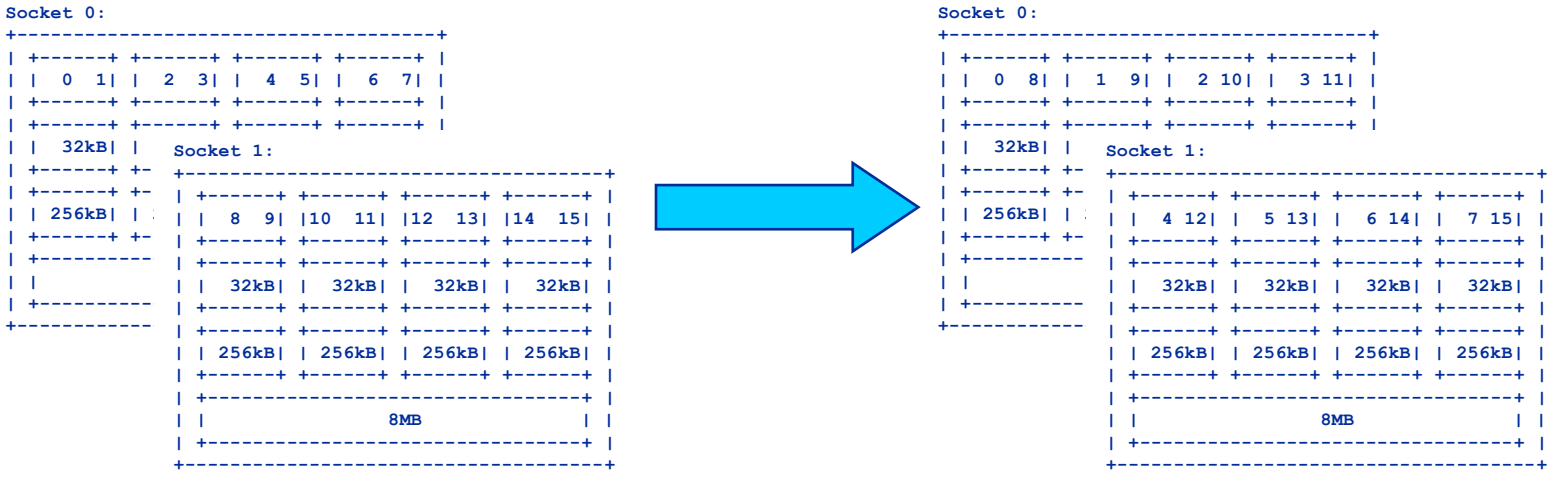
Skip shepherd thread

Pin all spawned threads in turn

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

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

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

Likwid-pin

Using logical core numbering

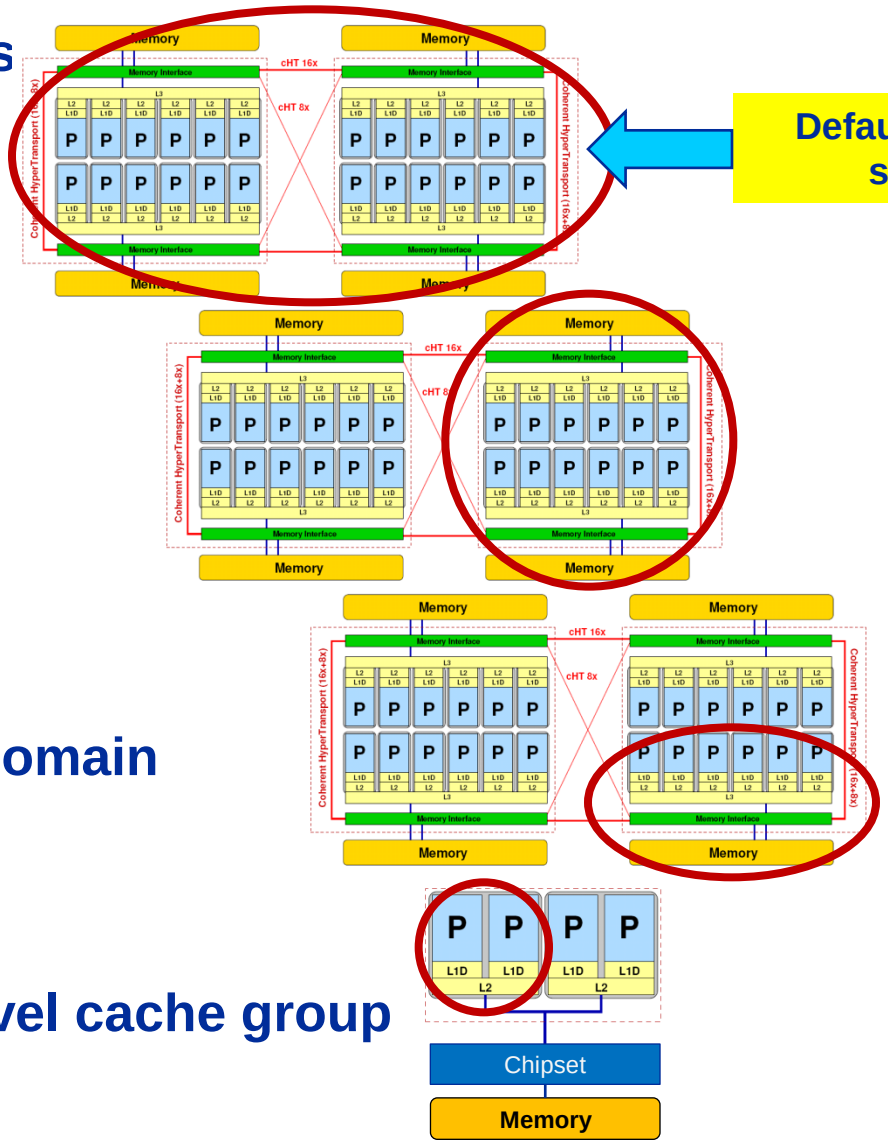
- **Possible unit prefixes**

N **node**

S **socket**

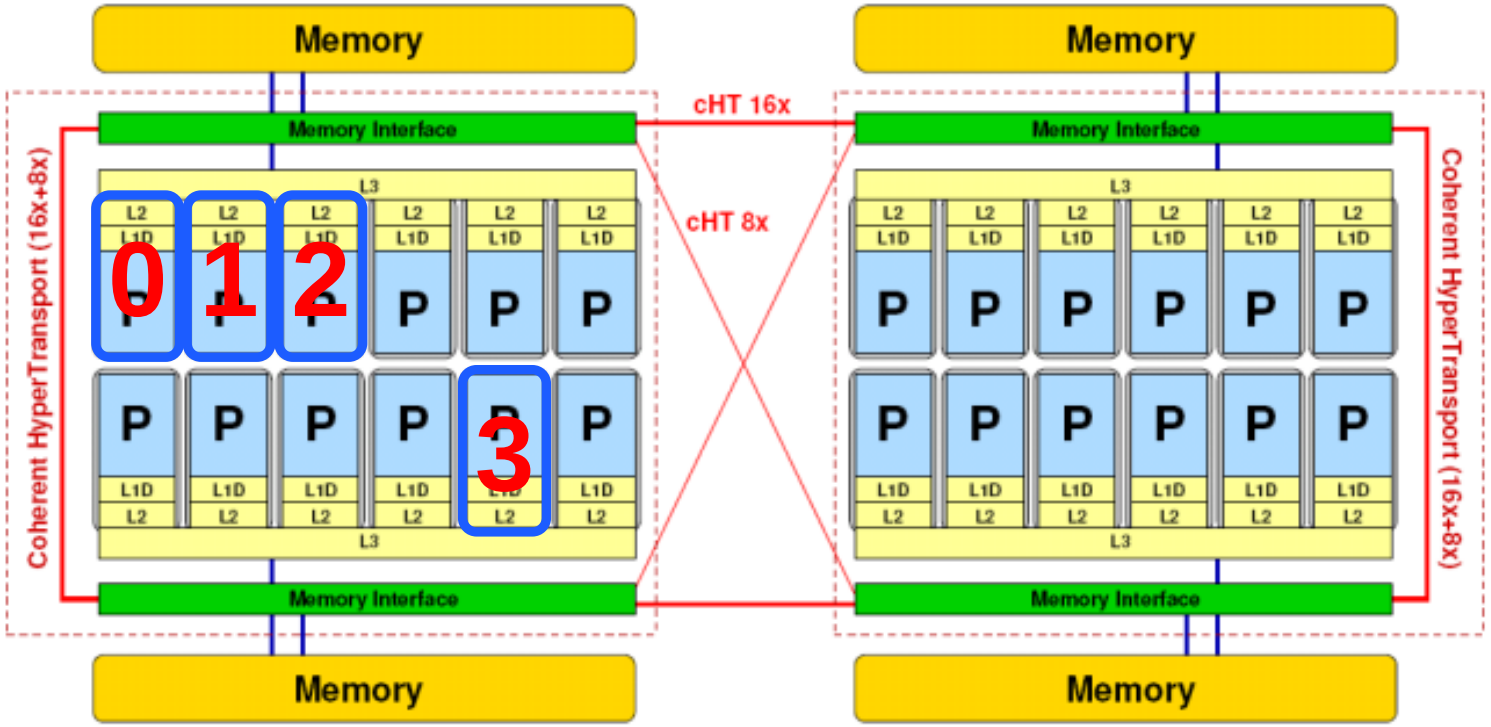
M **NUMA domain**

C **outer level cache group**



Default if –c is not specified!

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



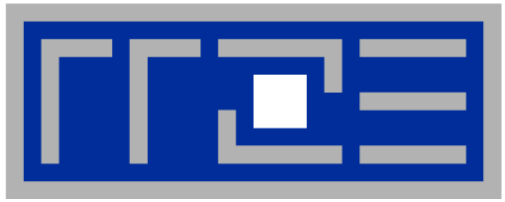
- `OMP_NUM_THREADS=4 likwid-pin -c L:0-3 ./a.out`

aprun on Cray

- **See Cray workshop slides 28*ff***

- **`aprun` supports only physical core numbering**
  - This is OK since the cores are always numbered consecutively on Crays
  - Use **`-ss`** switch to restrict allocation to local NUMA domain (see later for more on ccNUMA)
  - Use **`-d $OMP_NUM_THREADS`** or similar for MPI+OMP hybrid code

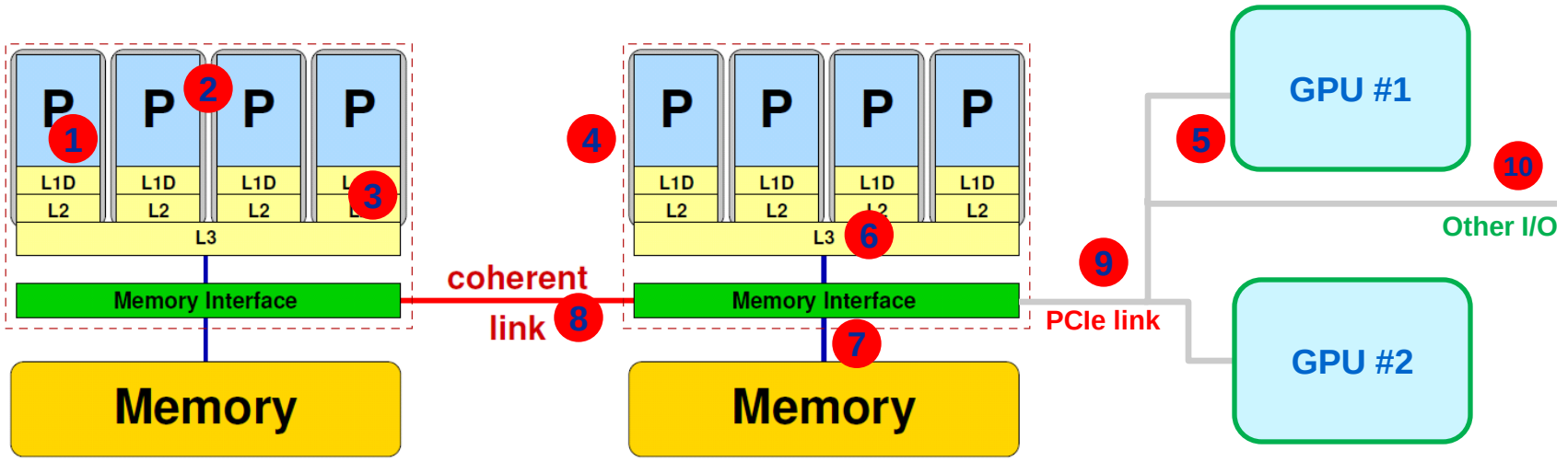
- **See later on how using multiple cores per module/chip/socket affects performance**



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?

The parallel vector triad benchmark

(Near-)Optimal code on Cray x86 machines

```
call get_walltime(S)
!$OMP parallel private(j)
do j=1,R
  if(N.ge.CACHE_LIMIT) then
!DIR$ LOOP_INFO cache_nt(A)
!$OMP parallel do
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
!$OMP end parallel do
  else
!DIR$ LOOP_INFO cache(A)
!$OMP parallel do
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
!$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)
```

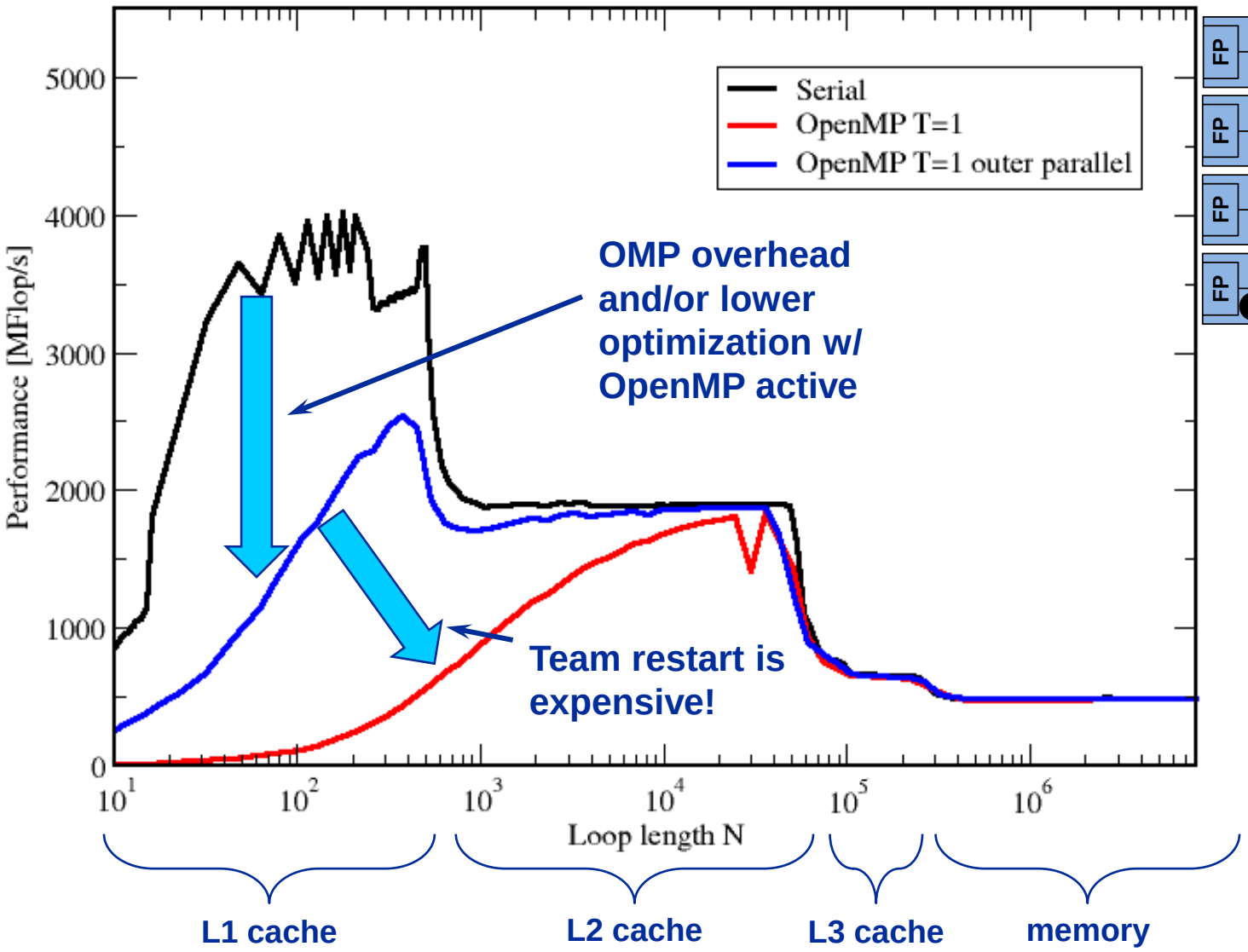
"outer parallel": Avoid thread team restart at every workshared loop

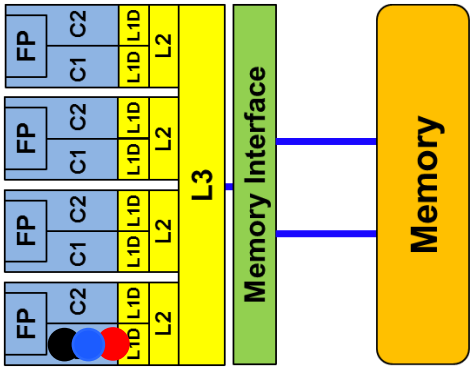
Large-N version (nontemporal stores)

Small-N version (standard stores)

The parallel vector triad benchmark

Single thread on Cray XE6 Interlagos node

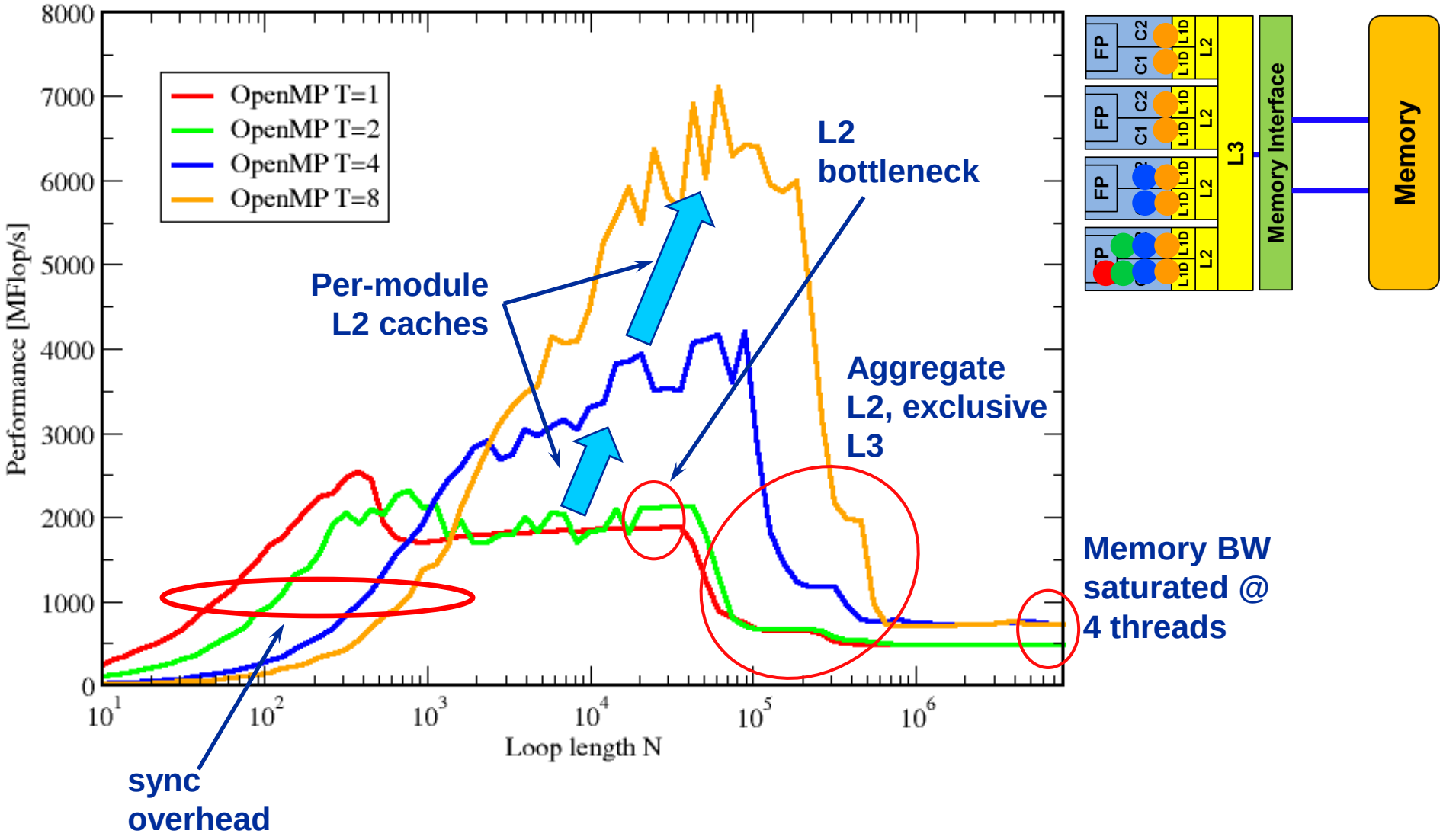




→ use only outer parallel from now on!

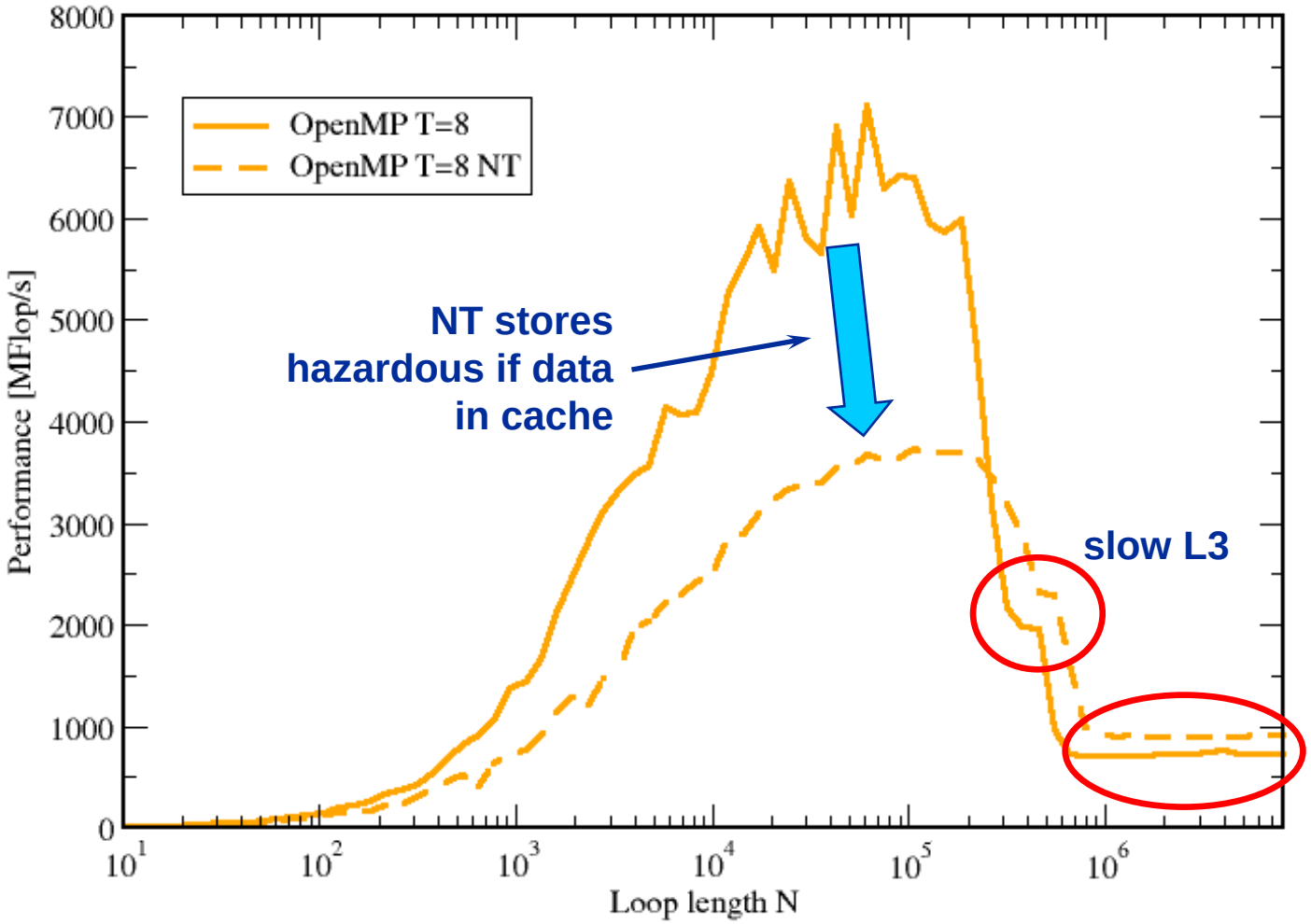
The parallel vector triad benchmark

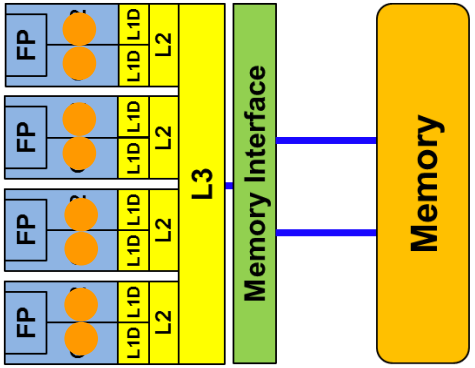
Intra-chip scaling on Cray XE6 Interlagos node



The parallel vector triad benchmark

Nontemporal stores on Cray XE6 Interlagos node

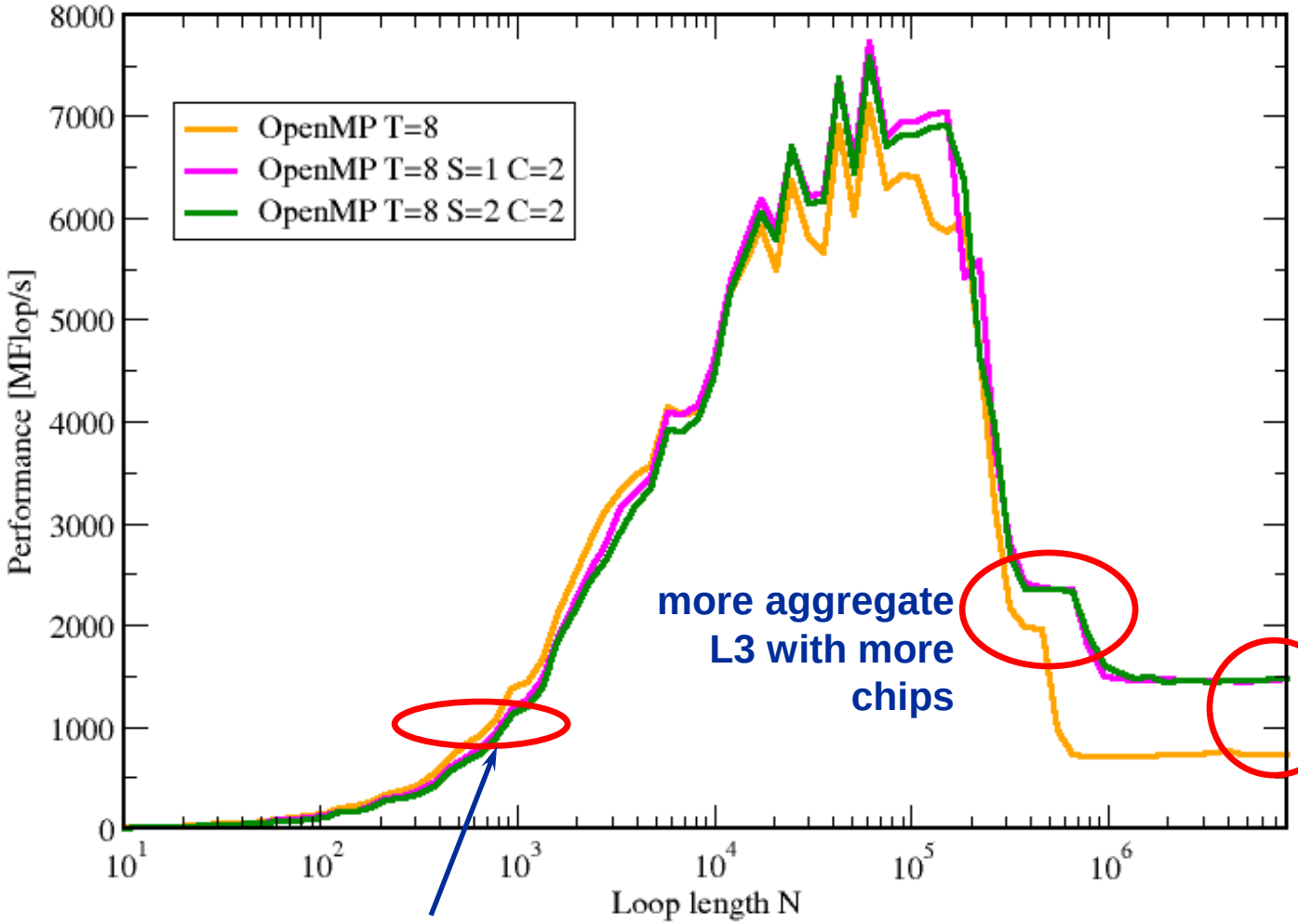


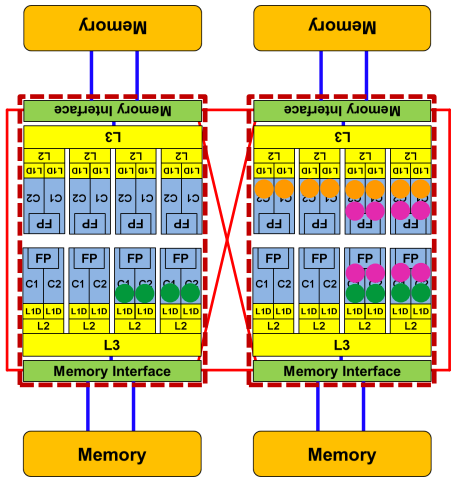


25% speedup for vector triad in memory via NT stores

The parallel vector triad benchmark

Topology dependence on Cray XE6 Interlagos node





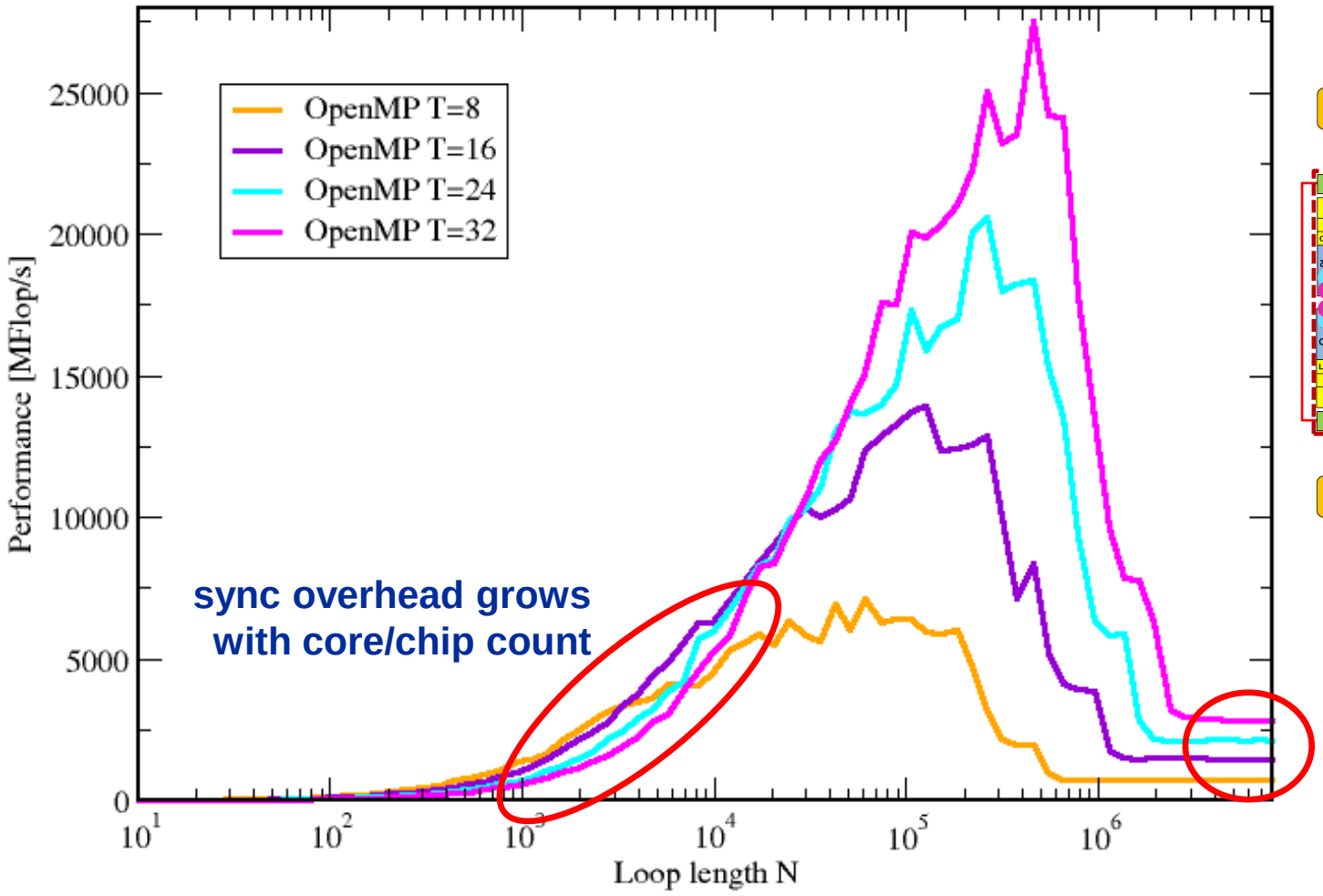
more aggregate L3 with more chips

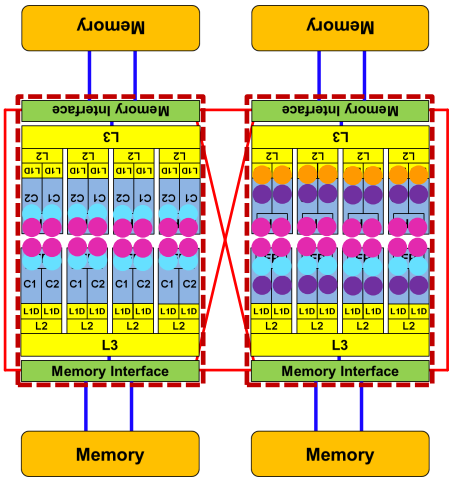
bandwidth scalability across memory interfaces

sync overhead nearly topology-independent @ constant thread count

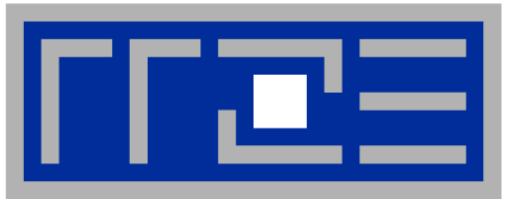
The parallel vector triad benchmark

Inter-chip scaling on Cray XE6 Interlagos node





bandwidth scalability across memory interfaces



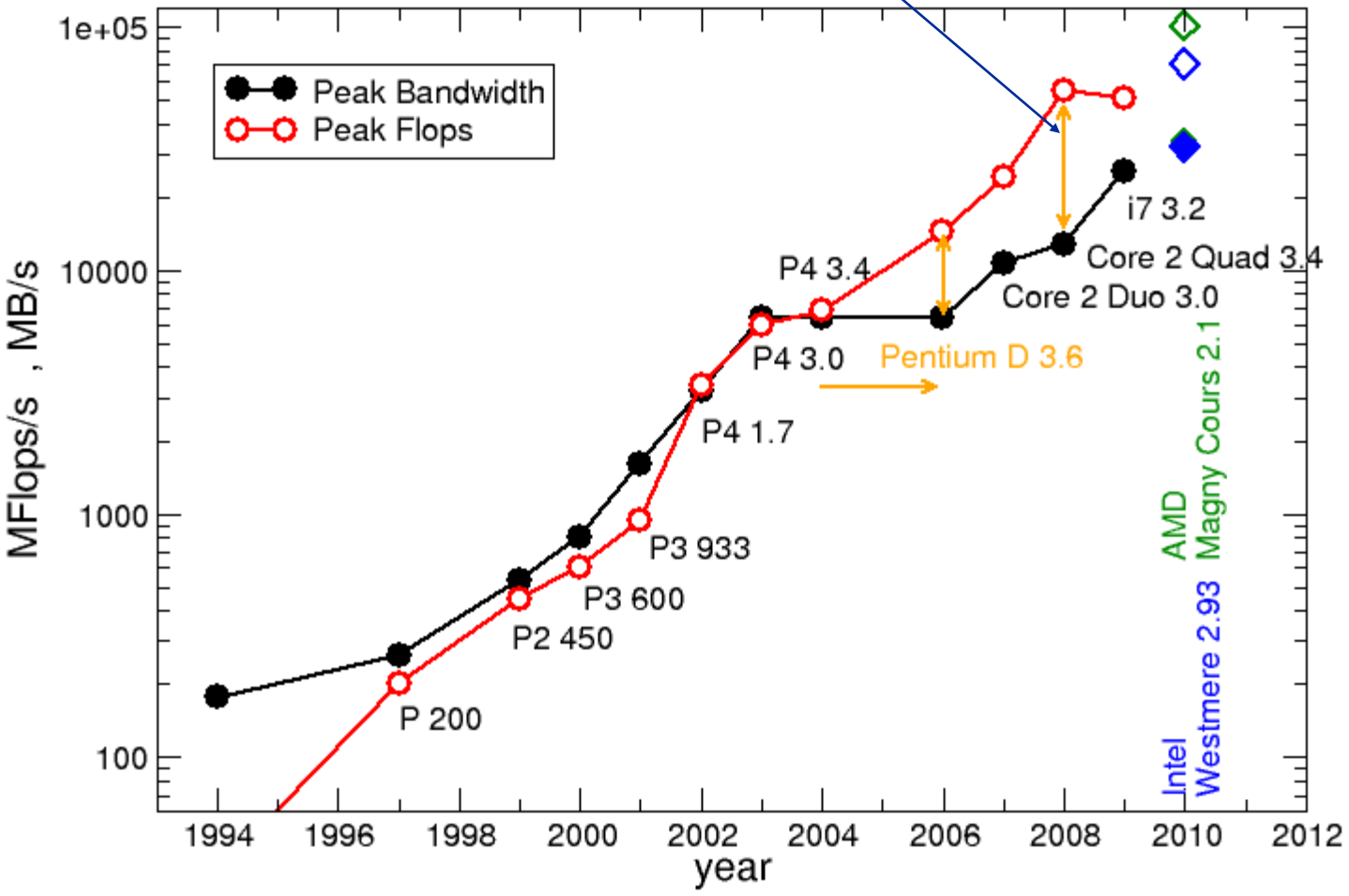
Bandwidth saturation effects in cache and memory

Low-level benchmark results

Bandwidth limitations: Memory

Some problems get even worse….

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



Balance values:

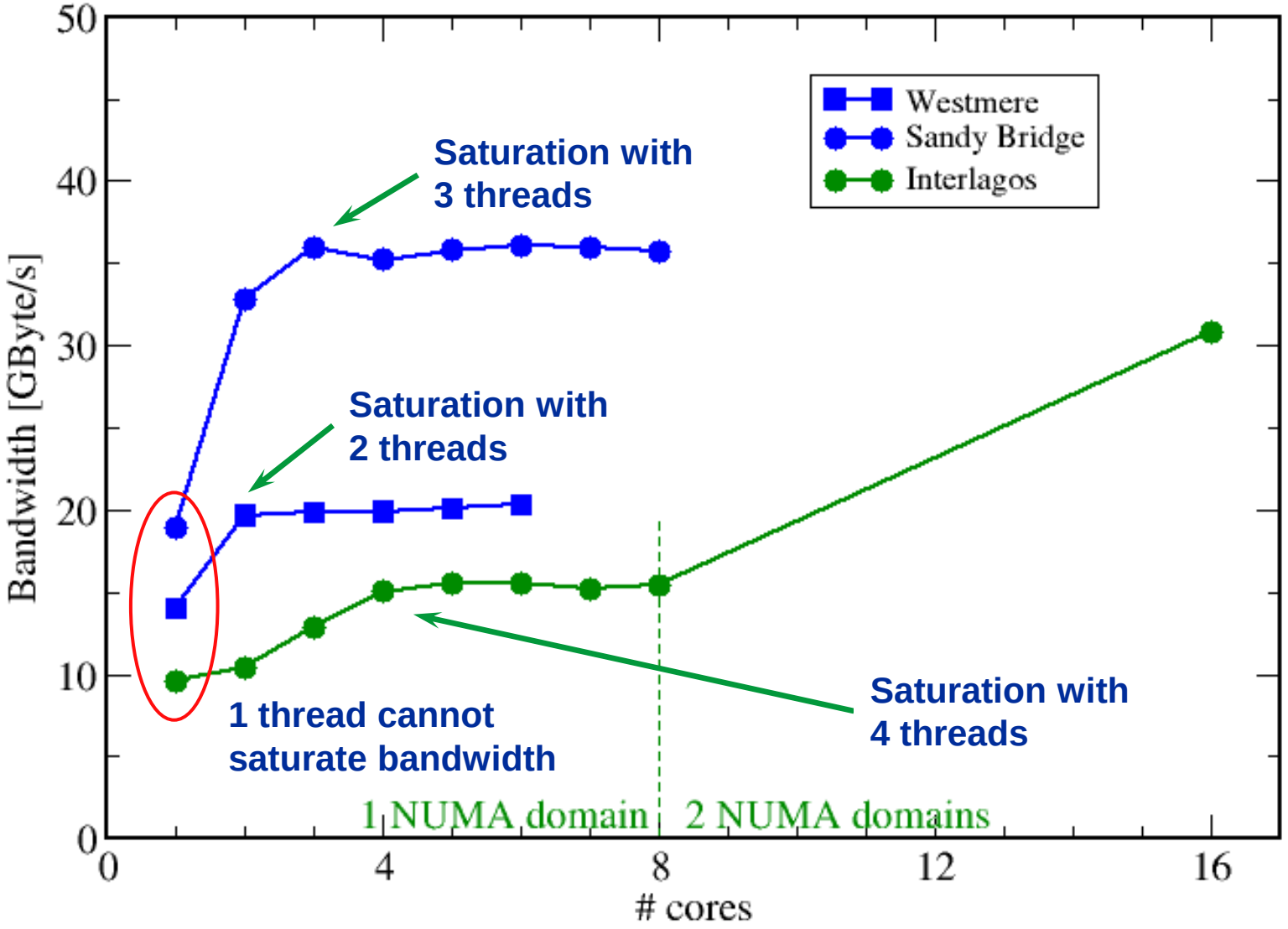
Scalar product:
1 Flop/double
→ 1/32 Peak

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

Large
MatrixMatrix
(BLAS3)

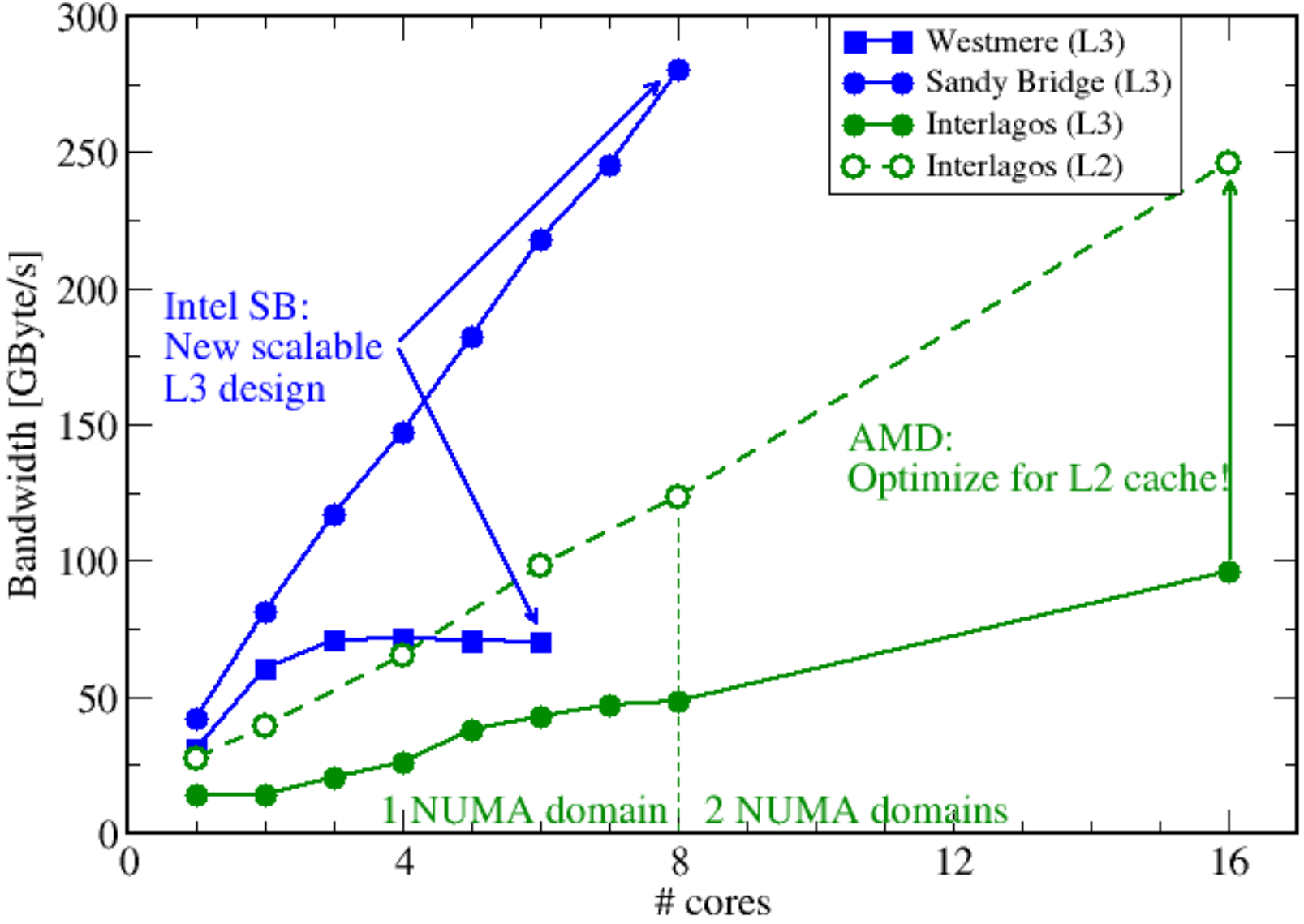
Bandwidth limitations: Main Memory

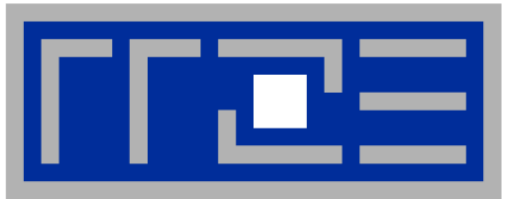
Scalability of shared data paths inside a NUMA domain **(V-Triad)**



Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache (V-Triad)





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!

- **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

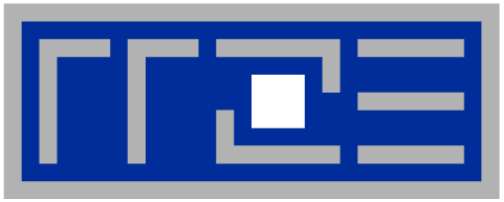
Barrier overhead in CPU cycles

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



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

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:

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

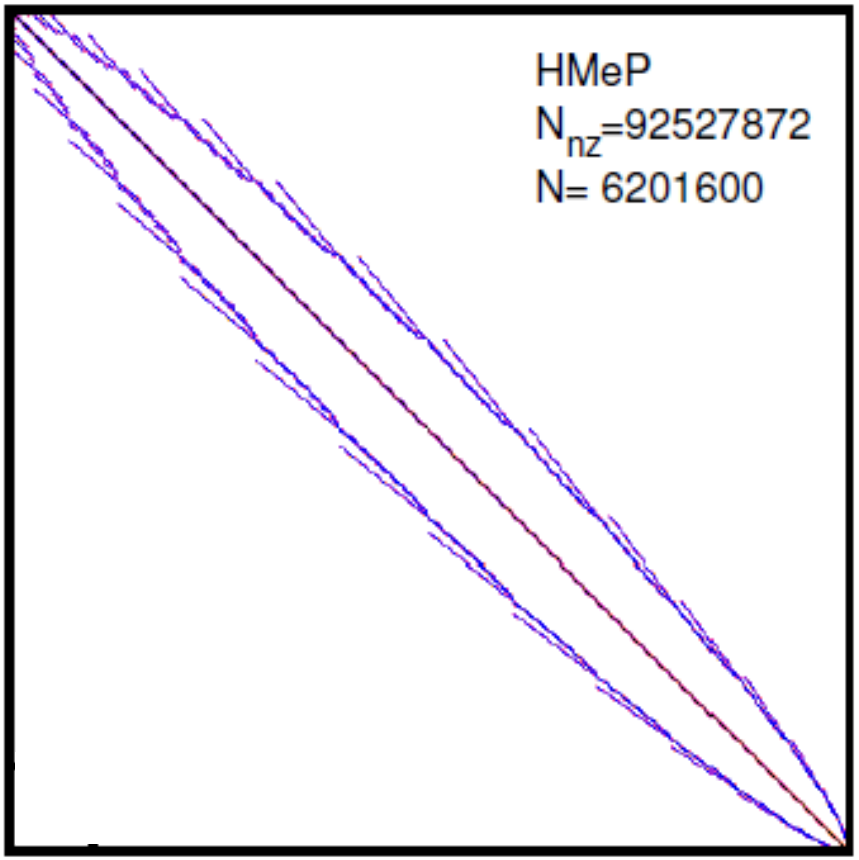
  - Usually many spMVMs required to solve a problem

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

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

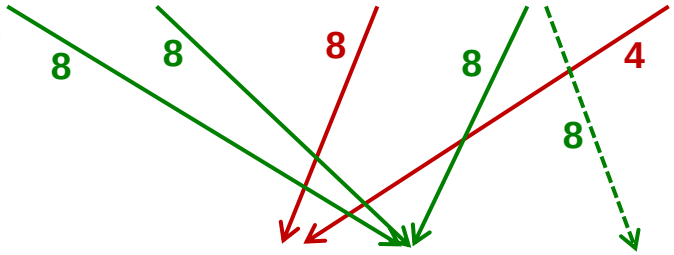


- **MPI-parallel spMVM is often communication-bound**
  - See hybrid part for what we can do about this…

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
  - → Even though the model has a “fudge factor” it is still useful!

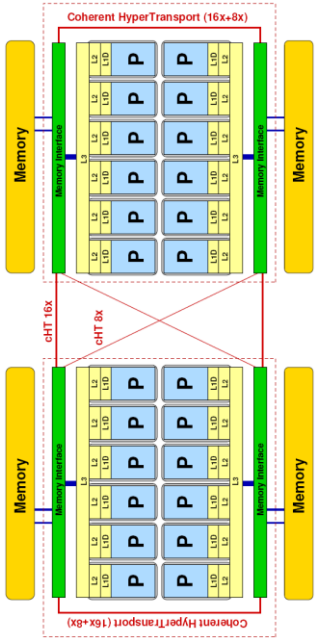
$$B_{\mathrm{CRS}} = \left(\frac{12+24/N_{\mathrm{nzr}}+\kappa}{2}\right)\frac{\mathrm{bytes}}{\mathrm{flop}} = \left(6+\frac{12}{N_{\mathrm{nzr}}}+\frac{\kappa}{2}\right)\frac{\mathrm{bytes}}{\mathrm{flop}}\,.$$

G. Schubert, 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, Preprint: arXiv:1106.5908

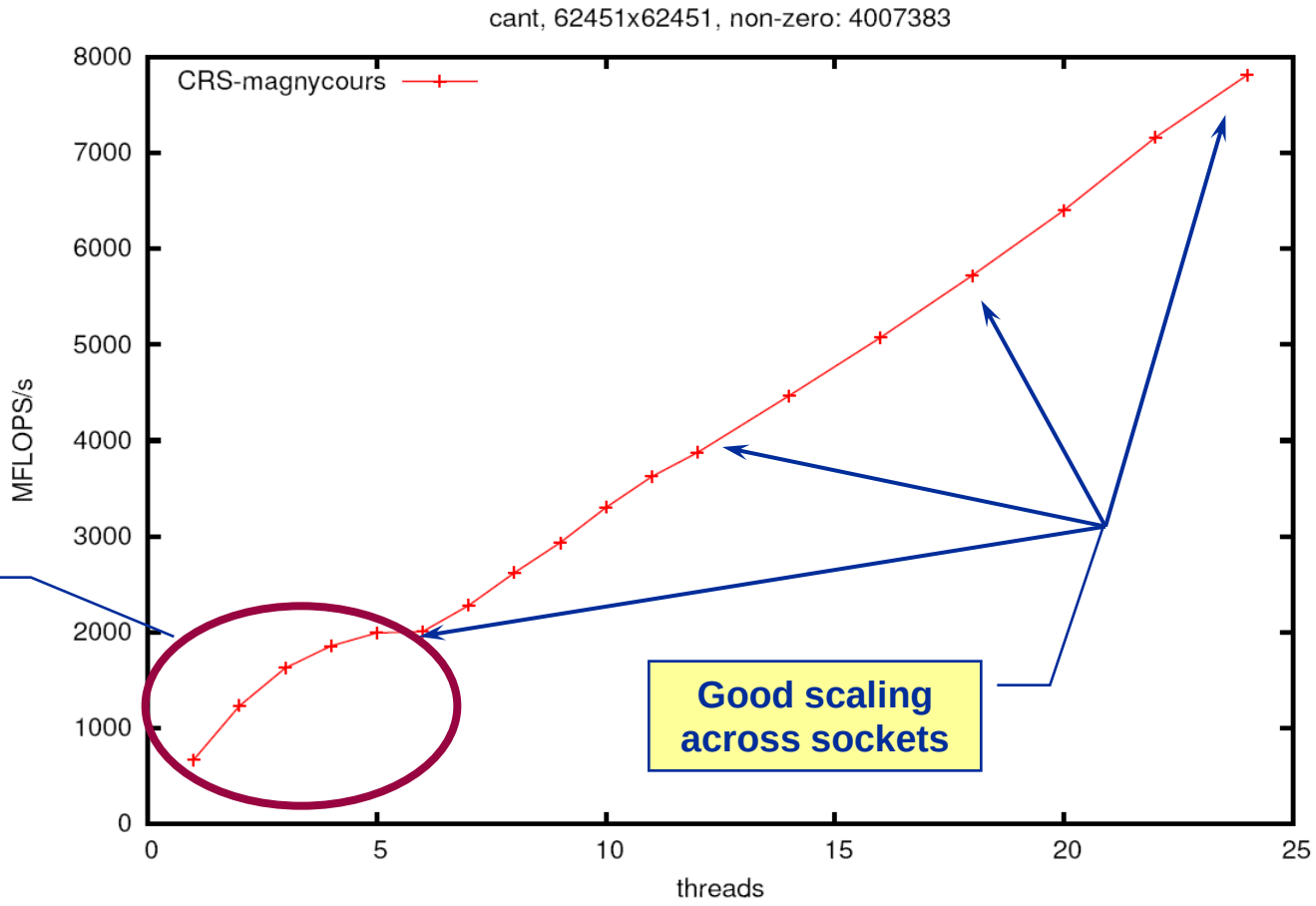
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 1: Large matrix**



Intrasocket bandwidth bottleneck

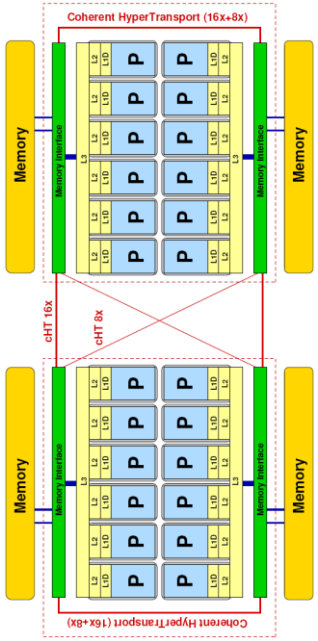


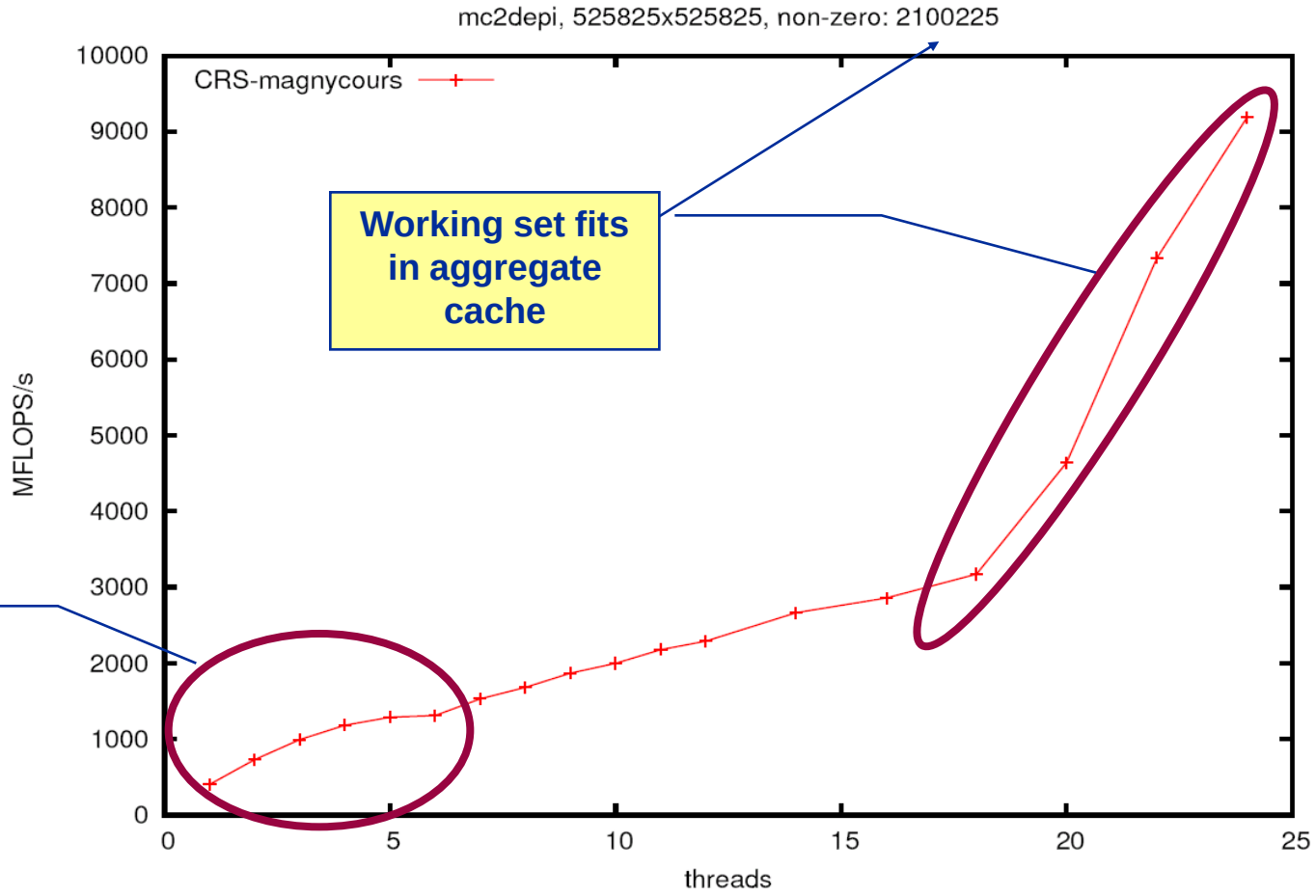
Good scaling across sockets

Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 2: Medium size**





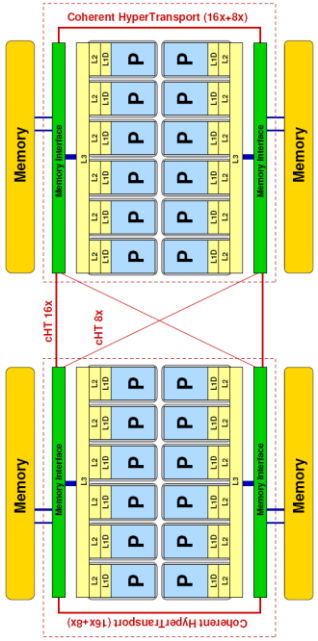
Working set fits in aggregate cache

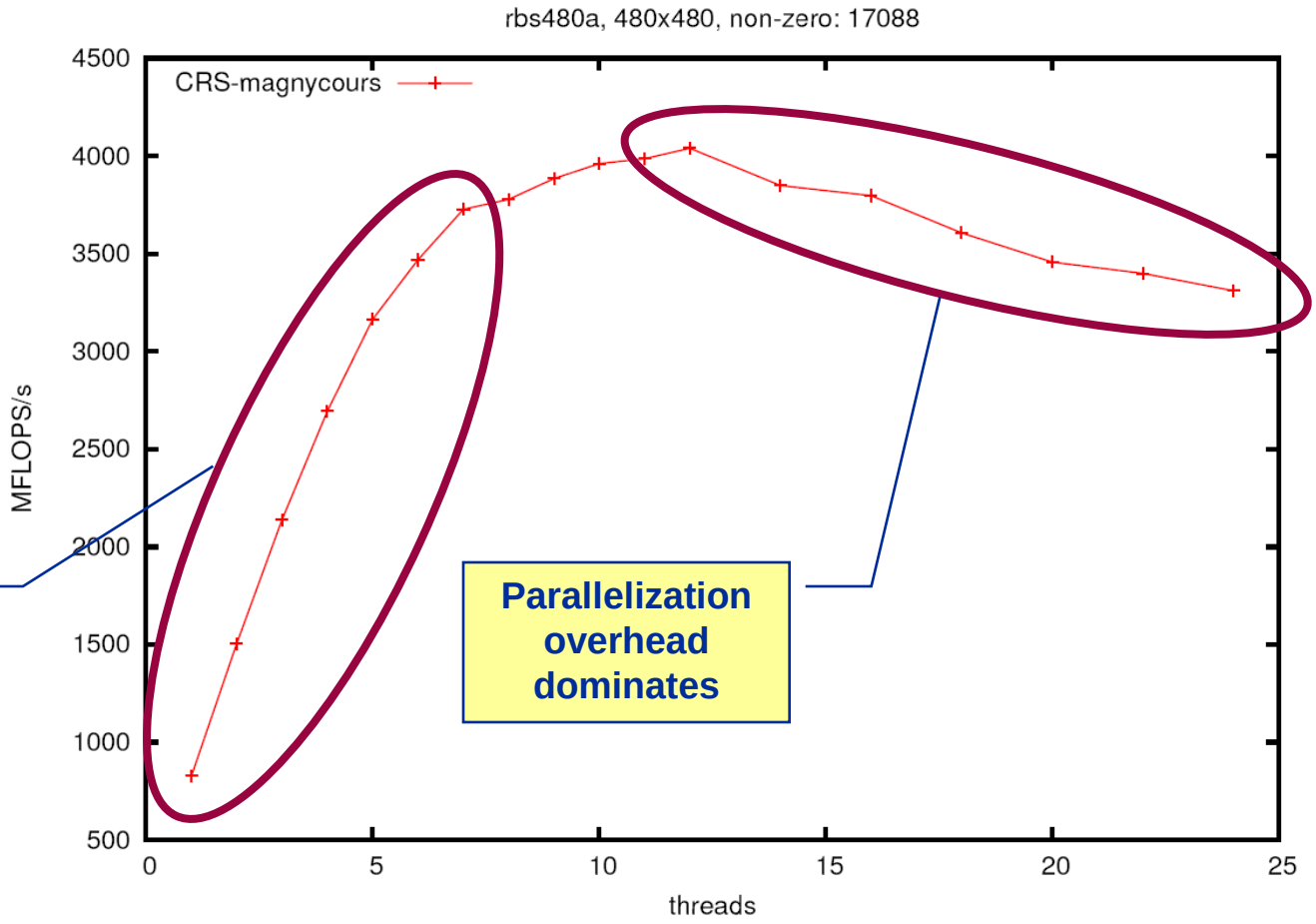
Intrasocket bandwidth bottleneck

Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

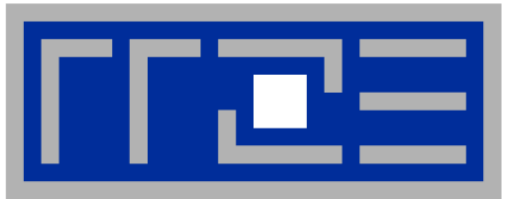
- **Case 3: Small size**





No bandwidth bottleneck

Parallelization overhead dominates



Probing performance behavior

likwid-perfctr

1. **Runtime profile / Call graph (gprof)**
2. **Instrument parts which consume significant part of runtime**
3. **Find performance signatures**

Possible signatures:

- **Bandwidth saturation**
- **Instruction throughput limited (real or language induced)**
- **Latency bound (irregular data access, high branch ratio)**
- **Load inbalance**
- **ccNUMA issues**
- **Pathologic cases (false cacheline sharing, expensive operations)**

Probing performance behavior

- **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
  - “Marker” API for starting/stopping counters
  - Multiple measurement region support
  - Preconfigured and extensible metric groups, list with **`likwid-perfctr -a`**

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

likwid-perfctr

Example usage with preconfigured metric group

```
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -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  |
+-------------------------+------------+----------+----------+----------+
```

Always measured

Configured metrics (this group)

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

likwid-perfctr

Identify load imbalance…

- `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

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

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

likwid-perfctr

… and load-balanced codes

```
env OMP_NUM_THREADS=6 likwid-perfctr -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

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

Higher CPI but better performance

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

Detecting latency-bound codes

… often with graph and tree data structures

Metric	Red-Black tree	Optimized data structure
Instructions retired	1.34268e+11	1.28581e+11
CPI	**9.0176**	**0.71887**
L3-MEM data volume [GB]	301	3.22
TLB misses	3.71447e+09	4077
Branch rate	36%	8.5%
Branch mispredicted ratio	7.8%	0.0000013%
Memory bandwidth [GB/s]	**10.5**	1.1

Useful likwid-perfctr groups: L3, L3CACHE, MEM, TLB, BRANCH

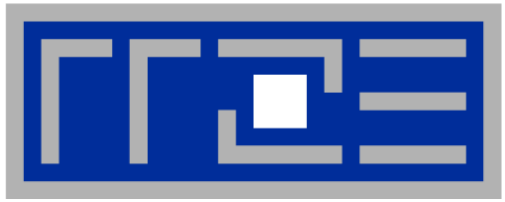
High CPI, near perfect scaling if using SMT threads (Intel).
Note: Latency bound code can still produce significant aggregated bandwidth.

Programming language induced problems

- **The object-oriented programming paradigm implements functionality resulting in many calls to small functions**
- **The ability of the compiler to inline functions (and still generate the best possible machine code) is limited**

- **Symptoms:**
  - Low (“good”) **CPI**
  - Low resource utilization (Flops/s, bandwidth)
  - Orders of magnitude more general purpose than arithmetic floating point instructions
  - High branch rate

- **Solution:**
  - Use basic data types and plain arrays in compute intensive loops
  - Use plain C-like code
  - Keep things simple – do not obstruct the compiler’s view on the code



Microarchitectural features of modern processors

Hardware-software interaction

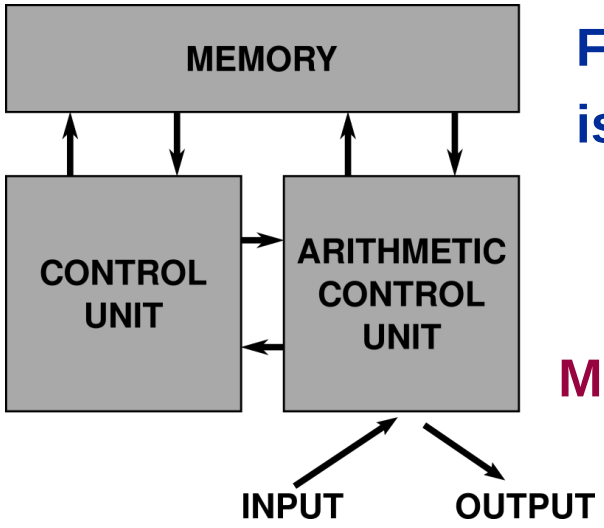
SIMD parallelism

A closer look at the cache hierarchy

Performance modeling on the microarchitecture level

Where do we come from?

Stored program design



Flexible, but optimization is hard!

Architect's view:
Make the common case fast !

EDSAC 1949

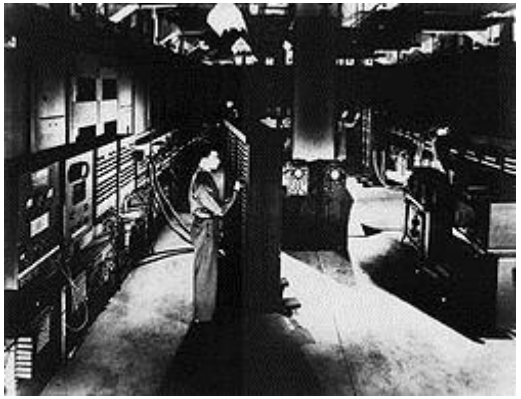
Instruction Level Parallelism
- Pipelining
- Superscalar execution

Data Access Locality
- Memory Hierarchy

Hardware Prefetcher

Data Parallelism
- SIMD execution

MIMD Parallelism
- SMT
- Multicore
- Multisocket
- Cluster



ENIAC 1948

First Assumption: ILP

Assumption: Every sequential instruction stream implies potential parallelism on instruction level (ILP)

Techniques to exploit assumption:

- **Pipelining (Overlap the execution of instructions)**
- **Superscalar design (more than 1 ALU)**
- **Out of order (OoO) execution**

Problems:

- **Makes hardware implementation complex**
- **Benefit is often not worth the effort**
- **Real-world benefit is limited (3-6 ops/cycle, 1 or less on average)**

CPI: A Measure for ILP

CPI: Cycles per Instruction

Ideal CPI for pipelined (non-superscalar) processor: 1

CPI for superscalar processor: < 1

Connection to Runtime:

time = cycles x clock rate

Cycles can be calculated as:

cycles = CPI x number of instructions

Second Assumption: Locality of Data Access

Assumption: If a data item is loaded it is likely that it is loaded again in the near future (temporal locality). If a data item is loaded it is likely that a data item in close distance is also loaded (spatial locality).

Techniques to exploit assumption:

- **Use caches to make repeated data accesses faster**
- **Use cache lines to reduce latency impact**

Problems:

- **Does not work for unstructured data accesses**
- **There are many algorithms with no or weak locality**

Hardware- Software Co-Design?

From algorithm to execution

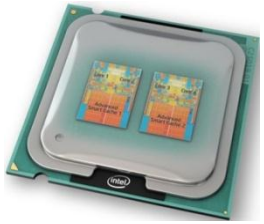
Reality:

Algorithm

The machine view:

ISA (Machine code)

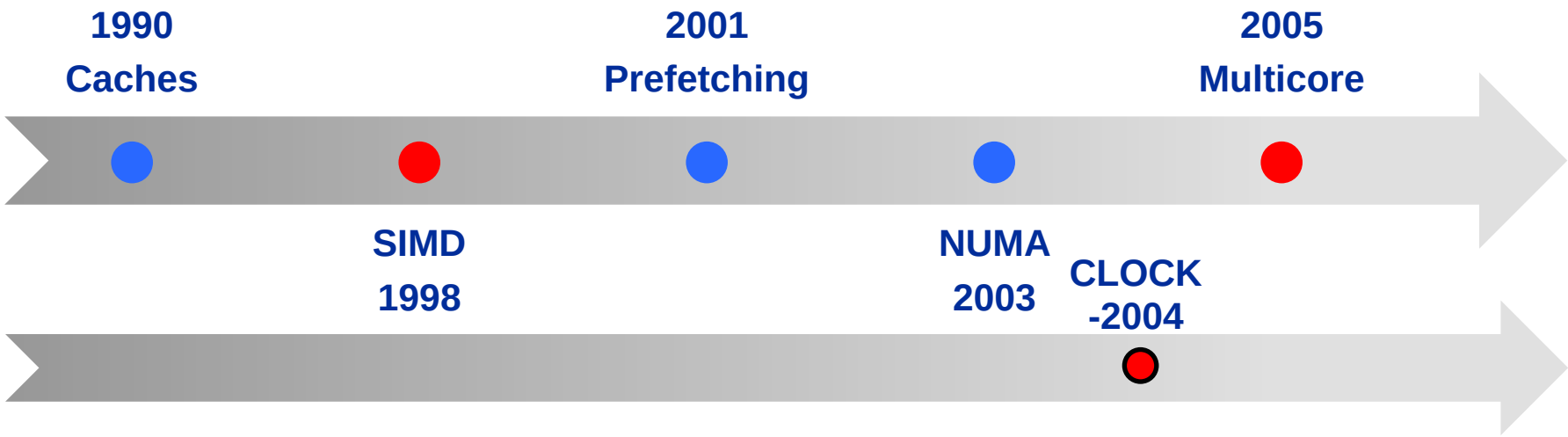
Programming language



Hardware = Black Box

How to achieve Performance

(for data intensive floating point codes on commodity chips)

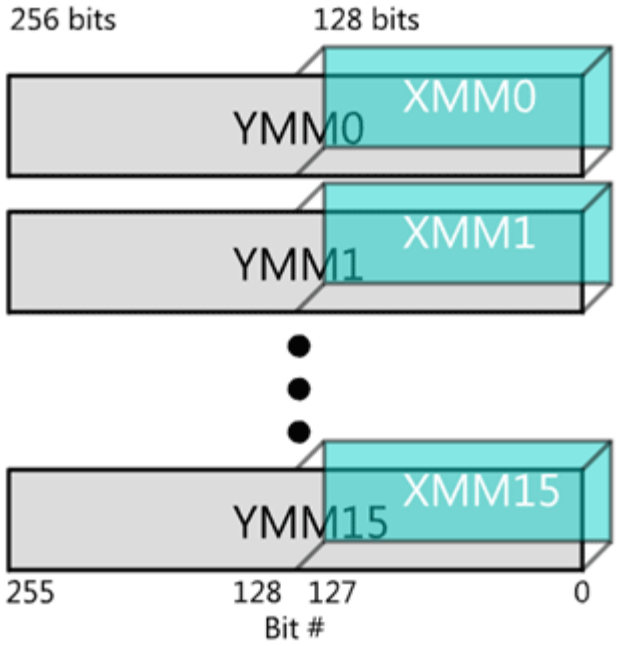


Explicit	Performance factor
Thread level parallelism	4-40x
SIMD	DP 2-4x SP 4-8x
Distributed memory parallelism	unlimited ☺ 1000x

Implicit	Performance factor
Instruction level parallelism	Pipelining 3-4x Superscalar 2x SMT 30%)
Caches	4-6x
NUMA	2-4x

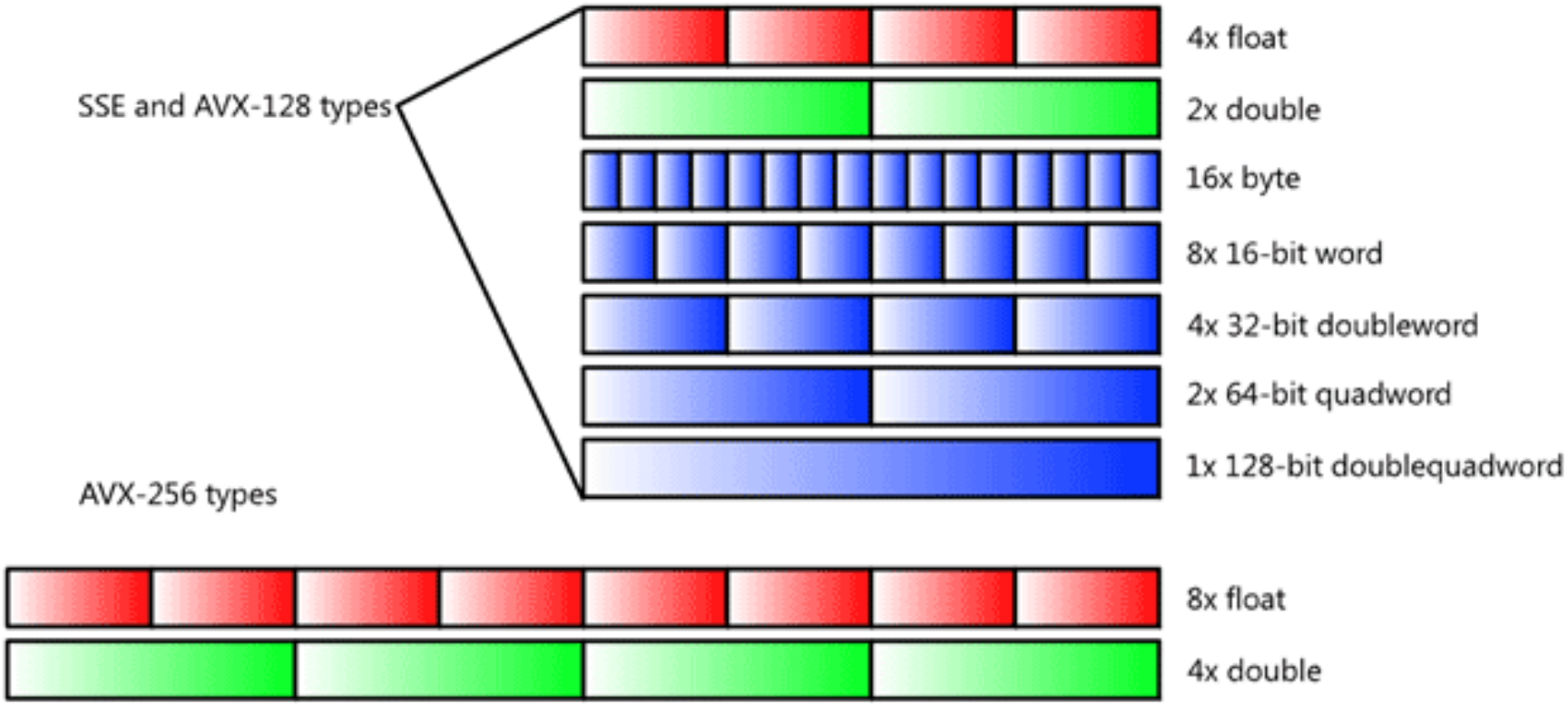
Node Performance: 1TFlops/s, 50-100 GB/s memory bandwidth

- **"Sensible SIMD" came with SSE (Pentium III) and SSE2 (Pentium 4) – Streaming SIMD Extensions**
- **With AVX a new SIMD instruction set with 256 bit register width was introduced**
- **AVX will be the relevant instruction set for the near future**
- **An extension to 512 bit register width is already in planning**

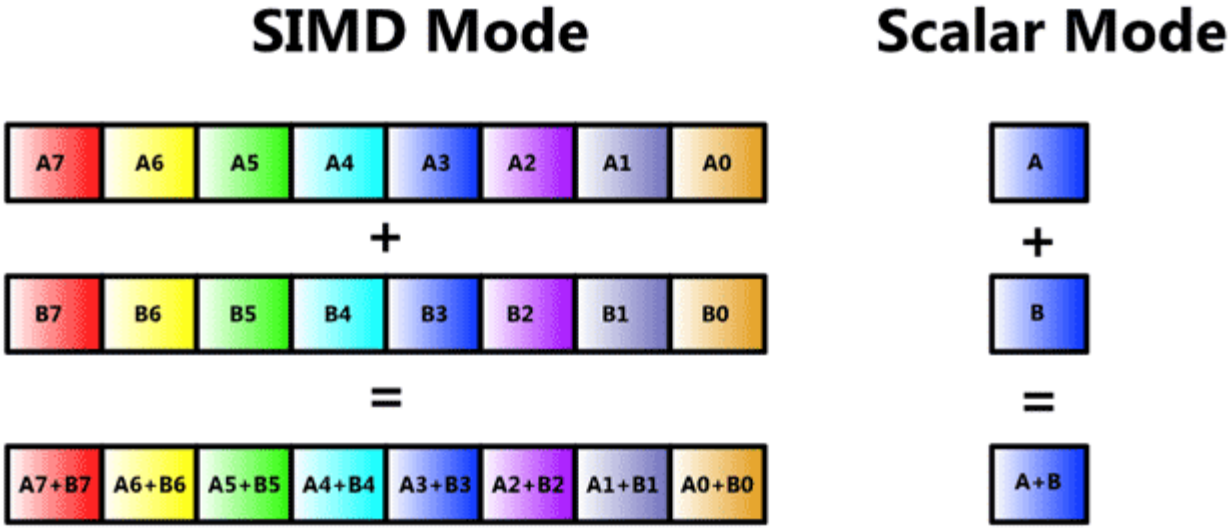


- **Each register can be partitioned into several integer or FP data types**
  - **8 to 128-bit integers**
  - **single or double precision floating point**
- **SIMD instructions can operate on the lowest or all partitions of a register at once**

- **Possible data types in an SIMD register**



- **Example: Single precision FP packed vector addition**



- **Multiple operations are done in one single instruction**
- **Nehalem: 1-cycle throughput for double precision SSE2 MULT & ADD leading to a peak performance of 4 (DP) FLOPs/cycle**
- **Sandy Bridge & Interlagos: Peak performance of 8 (DP) FLOPs/cycle**
  - **Interlagos: Only achievable with FMA instruction**

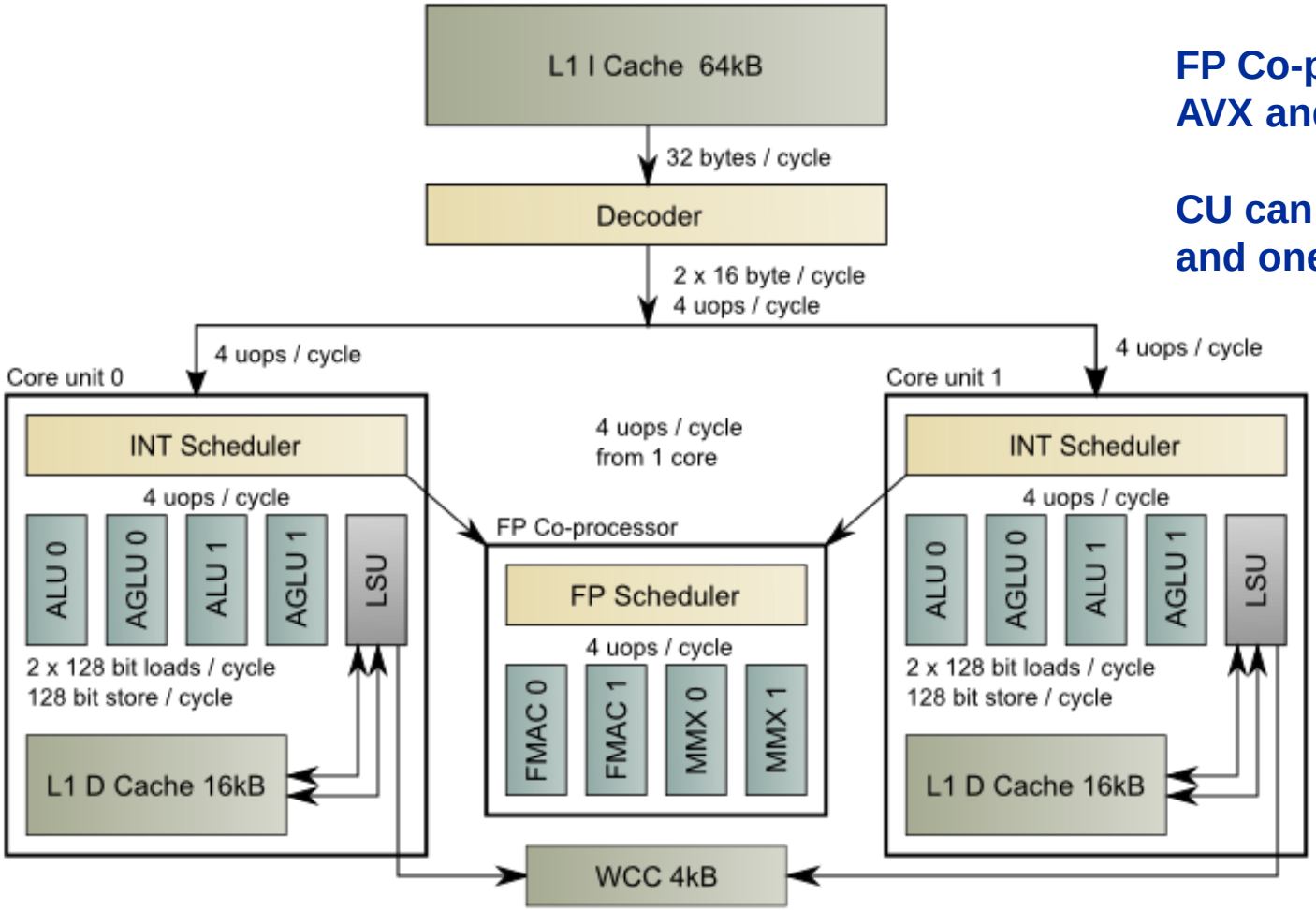
- **Everything on a processor happens in terms of cycles!**

- **All efforts are focused on increasing the average instruction throughput:**

  Metric CPI (cycles per instruction)

- **Important for us:**
- **Arithmetic instruction throughput**
- **Load/Store instruction throughput**
- **Overall instruction throughput**

Runtime Contributions:
1. **Instruction execution**
2. **Data transfers**
   - **Cache transfers**
   - **Memory transfers**

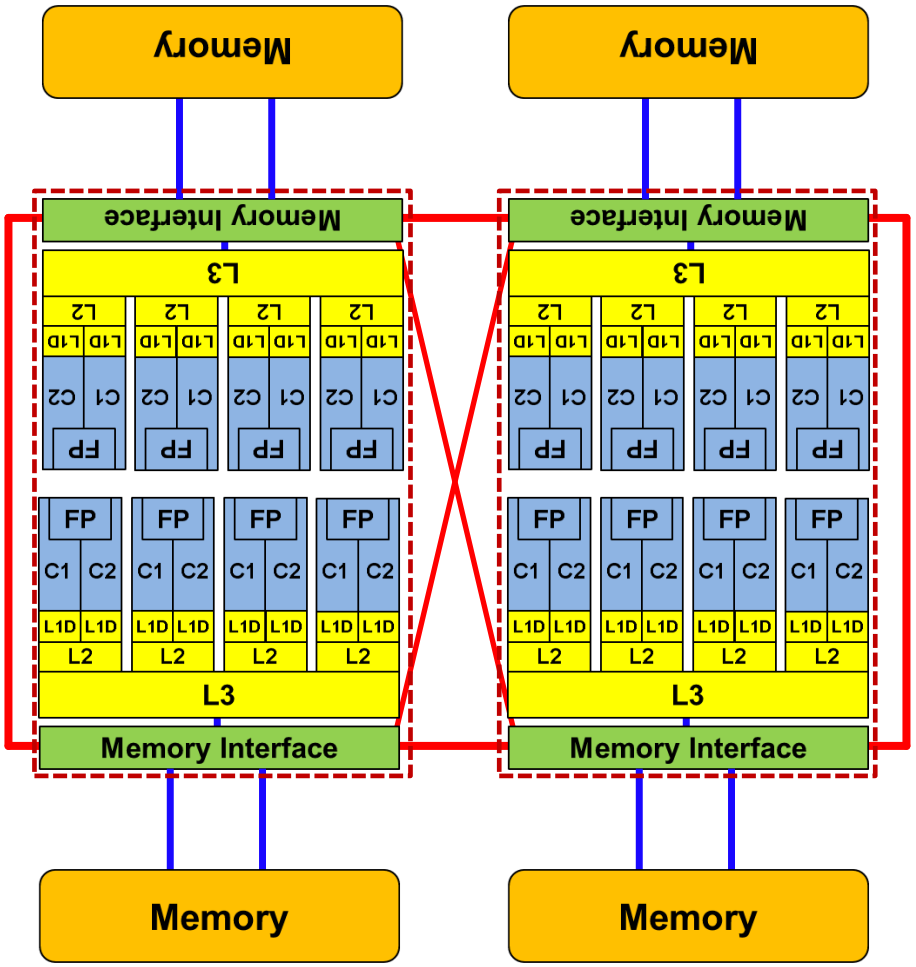


FP units 128bit wide

FP Co-processor supports: AVX and FMA4

CU can sustain two 128bit and one 128 bit store

AMD Interlagos
Node topology



Provide competitive node memory bandwidth for the price of a higher node complexity.

Target cache (i.e., the level that gets filled from memory) is the L2 cache.

Visible L3 cache size is 6 MB per chip (12 MB per socket).

Comparison chart

SIMD instruction throughput (instr/cycle)

Instruction type	SandyBridge	Westmere	MagnyCours	Interlagos
Add SSE	1	1	1	**2**
Mul SSE	1	1	1	**2**
Mul/Add SSE	2	2	2	2
Load SSE	2	1	2	2
Store SSE	1	1	1	1
Load/Store	2	2	2 ?	2
Add AVX	1	-	-	1
Mul AVX	1	-	-	1
Mul/Add AVX	2	-	-	1 (FMA 2)
Load AVX	1	-	-	1
Store AVX	0.5	-	-	0.5
Load/Store AVX	0.5			0.5
Max Overall	6	4	3	4

Comparison chart

Memory Hierarchies

- Intel SandyBridge EP
- 8 cores, 8 FP Units

L1D:
32kB, 8-way, write back
L2:
256kB, 8-way, inclusive
L3:
20MB, 20-way,inclusive, shared 8C
Memory:
4-channel DDR3-1600

Aggregated 40MB node cache size.

- AMD Magny Cours
- 6 cores, 6 FP Units

L1D:
64kB, 2-way, write back
L2:
512kB, 16-way, exclusive
L3:
5 MB, 32-way,exclusive, shared 6C
Memory:
2-channel DDR3-1333

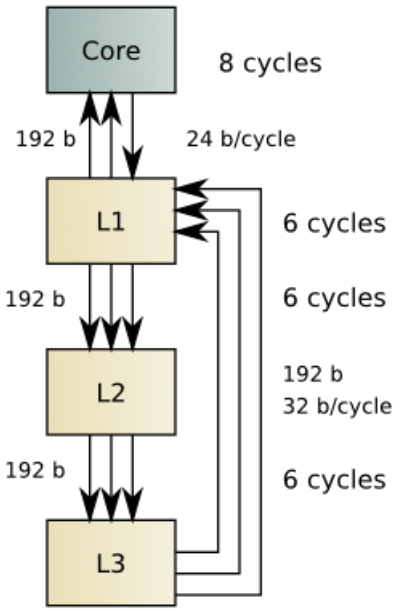
- AMD Interlagos
- 8 cores, 4 FP Units

L1D:
16kB, 4-way, write through
L2:
2MB, 16-way, inclusive. shared 2C
L3:
6 MB, 48-way, exclusive, shared 8C
Memory:
2-channel DDR3-1866

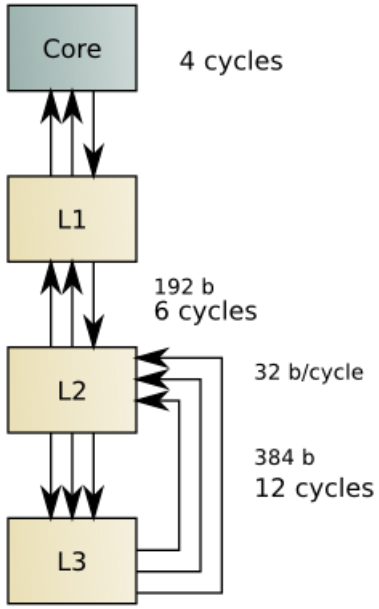
Aggregated 56MB node cache size.

Interlagos design feature
Exclusive caches

- **Exclusive cache means that there is only one copy of a cache line in the cache hierarchy! Often called victim cache**
- **Motivation: Visible cache size for application is larger**
- **BUT: More cache traffic necessary**



Magny Cours



Interlagos

- **The aggregated L3 bandwidth is low**
- **For HPC applications the L3 cache is not attractive**
- **Stream benchmark:**

  L3: IL 40 GB/s, SNB 193 GB/s

  5MB (fits in aggr. L2):
  IL 108 GB/s, SNB 215GB/s

Interlagos design feature

Write through L1 cache

Cycles/CL	load		store		copy		stream triad	
Cores/ CU	1	2	1	2	1	2	1	2
L1	2	4	10	20	10	20	7	14
L2	5.43	5.83	11.21	22.21	13.47	25.21	17.63	30.40
L2 (prefetch)	3.64	5.72	-	-	12.92	25.53	16.22	30.21

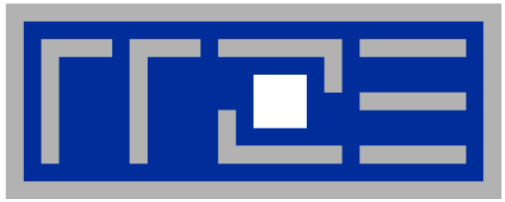
Consequences:

- **Stores involve a large penalty**
- **L2 cache store bandwidth does not scale**
- **Prefetching to L1 only pays off with one core**

Write through motivation:

- **Simpler to implement (cache coherence)**
- **Can save overhead for shared L2 access**
- **No write allocate**
- **But higher cost for stores in L1 cache**

Try to avoid stores as far as possible! ☺



Reading x86 assembly code

To read or write assembly code you have to know about:

- Instruction Set Architecture (ISA)
- Application Binary Interface (ABI)
- Object Code Format (ELF on Linux)
- Assembler specific directives (gas, masm)

Useful tools:

- GNU binutils (objdump, readelf)
- Debugger (gdb)
- Compiler option –S (Intel/GCC)

Basic approach to check the instruction code

- **Get the assembler code (Intel compiler):**

```
icc -S -O3  -xHost  triad.c  -o triad.s
```

- **Disassemble Executable:**

```
objdump -d  ./cacheBench | less
```

- **Things to check for:**
  - Is the code vectorized? Search for pd/ps suffix.

    mulpd, addpd, vaddpd, vmulpd
  - Is the data loaded with 16 byte moves?

    movapd, movaps, vmovupd
  - For memory-bound code: Search for nontemporal stores:

    movntpd, movntps

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B
AMD64 Architecture Programmer's Manual Vol. 1-5

Basics of the x86-64 ISA

- Instructions have 0 to 2 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: `A[i]` equivalent to `*(A+i)` (a pointer has a type: `A+i*8`)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
movaps    %xmm4, 48(%rdi,%rax,8)
addq      $8, %rax
js        ..B1.4
```

```
401b9f: 0f 29 5c c7 30      movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08         add    $0x8,%rax
401ba8: 78 a6               js     401b50 <triad_asm+0x4b>
```

Basics of the x86-64 ISA II

16 general Purpose Registers (64bit):

`rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15`

alias with eight 32 bit register set:

`eax, ebx, ecx, edx, esi, edi, esp, ebp`

Floating Point SIMD Registers:

`xmm0-xmm15` SSE (128bit) alias with 256bit registers

`ymm0-ymm15` AVX (256bit)

SIMD instructions are distinguished by:

AVX (VEX) prefix: `v`

Operation: `mul, add, mov`

Modifier: non temporal (`nt`), unaligned (`u`), aligned (`a`), high (`h`)

Data type: single (`s`), double (`d`)

Basics of x86-64 ABI

- Regulations how functions are called on binary level
- Differs between 32 bit / 64 bit and Operating Systems

x86-64 on Linux:

- Integer or address parameters are passed in the order :
  `rdi, rsi, rdx, rcx, r8, r9`
- Floating Point parameters are passed in the order `xmm0-xmm7`
- Registers which must be preserved across function calls:
  `rbx, rbp, r12-r15`
- Return values are passed in `rax/rdx` and `xmm0/xmm1`

Case Study: summation

```
float sum = 0.0;

for (int j=0; j<size; j++){
    sum += data[j];
}
```

To get code use `objdump -d` on object file or executable.

Instruction code:

```
401d08:    f3 0f 58 04 82      addss   (%rdx,%rax,4),%xmm0
401d0d:    48 83 c0 01         add     $0x1,%rax
401d11:    39 c7               cmp     %eax,%edi
401d13:    77 f3               ja      401d08
```

Instruction address

Opcodes

Assembly code

How to leverage SIMD

- The compiler does it for you (aliasing, alignment, language)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available. To enable instruction set often additional flags are necessary:

- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX)

- `x86intrin.h` (all instruction set extensions)

Case Study: summation using intrinsics

```
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 =  _mm_setzero_ps();
sum1 =  _mm_setzero_ps();
sum2 =  _mm_setzero_ps();
sum3 =  _mm_setzero_ps();

for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

```
sum0 = _mm_add_ps(sum0, sum1);
sum0 = _mm_add_ps(sum0, sum2);
sum0 = _mm_add_ps(sum0, sum3);
sum0 = _mm_hadd_ps(sum0, sum0);
sum0 = _mm_hadd_ps(sum0, sum0);

_mm_store_ss(&scalar_sum, sum0);
```

Case Study: summation, instruction code

```
14:   0f 57 c9                xorps  %xmm1,%xmm1
17:   31 c0                   xor    %eax,%eax
19:   0f 28 d1                movaps %xmm1,%xmm2
1c:   0f 28 c1                movaps %xmm1,%xmm0
1f:   0f 28 d9                movaps %xmm1,%xmm3
22:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
28:   0f 10 3e                movups (%rsi),%xmm7
2b:   0f 10 76 10             movups 0x10(%rsi),%xmm6
2f:   0f 10 6e 20             movups 0x20(%rsi),%xmm5
33:   0f 10 66 30             movups 0x30(%rsi),%xmm4
37:   83 c0 10                add    $0x10,%eax
3a:   48 83 c6 40             add    $0x40,%rsi
3e:   0f 58 df                addps  %xmm7,%xmm3
41:   0f 58 c6                addps  %xmm6,%xmm0
44:   0f 58 d5                addps  %xmm5,%xmm2
47:   0f 58 cc                addps  %xmm4,%xmm1
4a:   39 c7                   cmp    %eax,%edi
4c:   77 da                   ja     28 <compute_sum_SSE+0x18>
4e:   0f 58 c3                addps  %xmm3,%xmm0
51:   0f 58 c2                addps  %xmm2,%xmm0
54:   0f 58 c1                addps  %xmm1,%xmm0
57:   f2 0f 7c c0             haddps %xmm0,%xmm0
5b:   f2 0f 7c c0             haddps %xmm0,%xmm0
5f:   c3                      retq
```

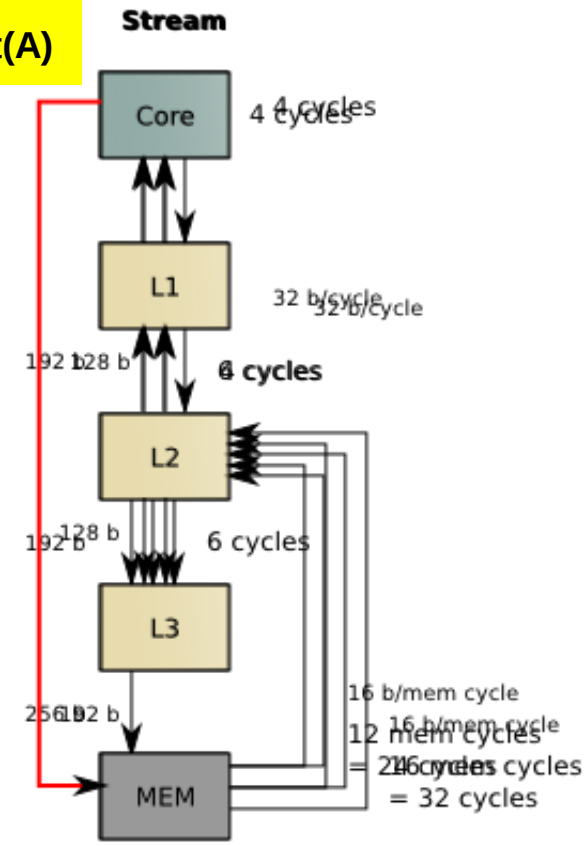
Loop body

Improving Memory Performance

Streaming Stores on Interlagos

```
#pragma vector aligned
#pragma vector always
#pragma vector nontemporal
for (i=0;i< size;i++){
   A[i] = B[i] +alpha* C[i];
}
```

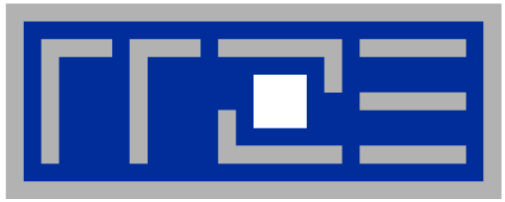
Cray:
LOOP_INFO cache_nt(A)



617 GFlop/s vs. 854 GFlop/s

```
..B1.4:
movaps (%rdx,%rax,8),%xmm1
mulpd  %xmm0, %xmm4
addpd (%rsi,%rax,8),%xmm1
movntpd %xmm1, (%rdi,%rax,8)
addq  1,%rax
cmpq  %rcx, %rax
js  ..B1.4
```

On Interlagos NT stores circumvent both write-through stores and the L3 cache. This makes them even attractive for smaller data sets which could fit into L3 cache. triad (3MB): 783 Gflop/s, NT 1156 Gflop/s



Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

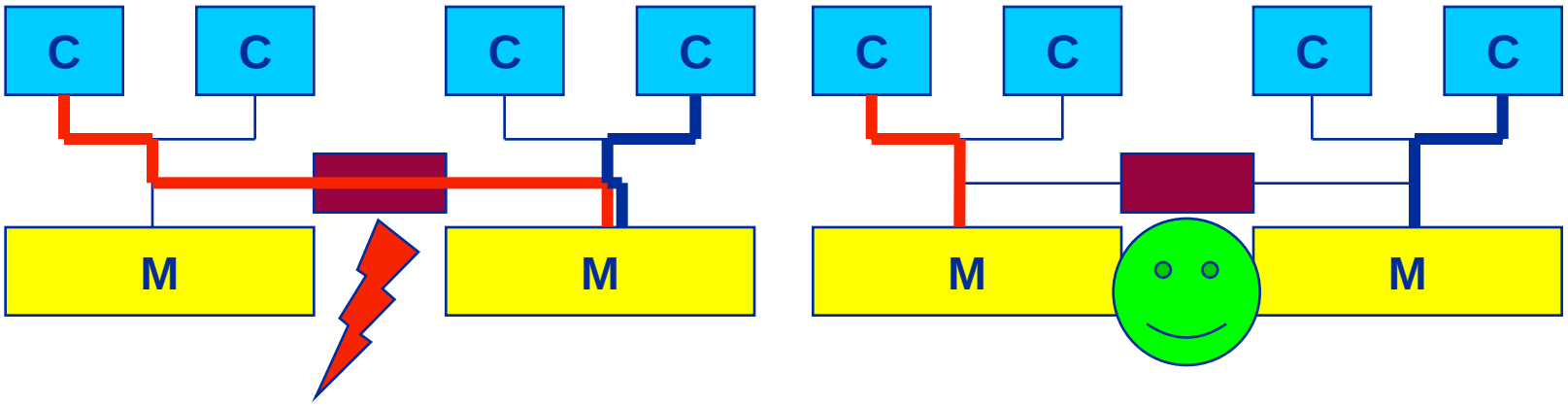
First touch placement policy

C++ issues

ccNUMA locality and dynamic scheduling

ccNUMA locality beyond first touch

- **ccNUMA:**
  - 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?**

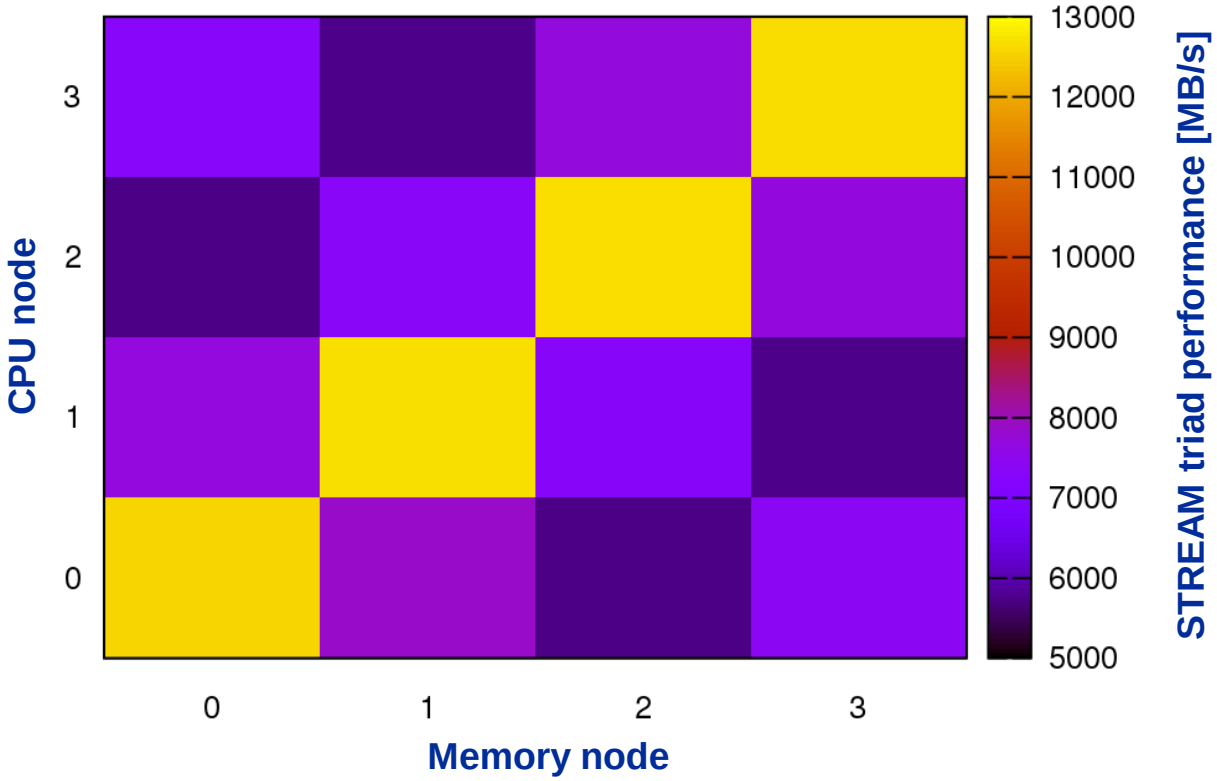


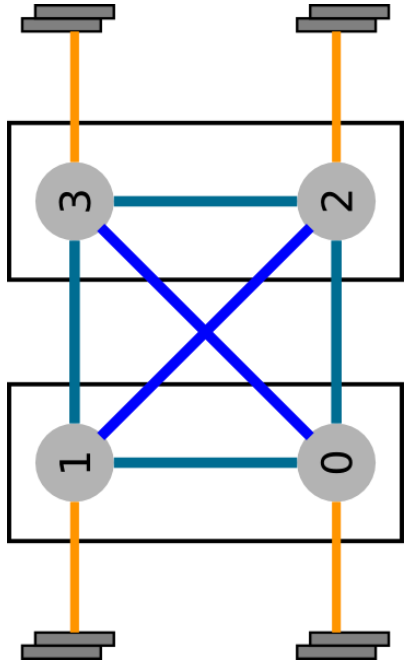
  - Page placement is implemented in units of OS pages (often 4kB, possibly more)

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





ccNUMA locality tool numactl:

How do we enforce some locality of access?

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

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

- **Examples:**

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

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

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

ccNUMA default memory locality

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

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

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

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

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

Memory not mapped here yet

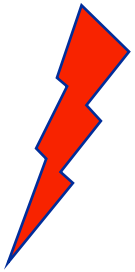
Mapping takes place here

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

Coding for 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
```

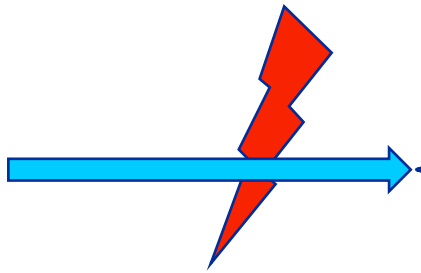


```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

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)

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
```

Coding for Data Locality

- **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)

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

optional

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

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

→ **placement problem with**

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

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

optional

- **Solution: Provide overloaded `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;
}

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

parallel first touch

- **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<>`**

optional

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

Application:

```
vector<double,NUMA_Allocator<double> > x(10000000)
```

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 N:0-7 ./a.out
```

Using performance counters for diagnosing bad ccNUMA access locality

- **Intel Nehalem EP node:**

Uncore events only counted once per socket

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

RDTSC timing: 0.827196 s

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

Half of read BW comes from other socket!

If all fails…

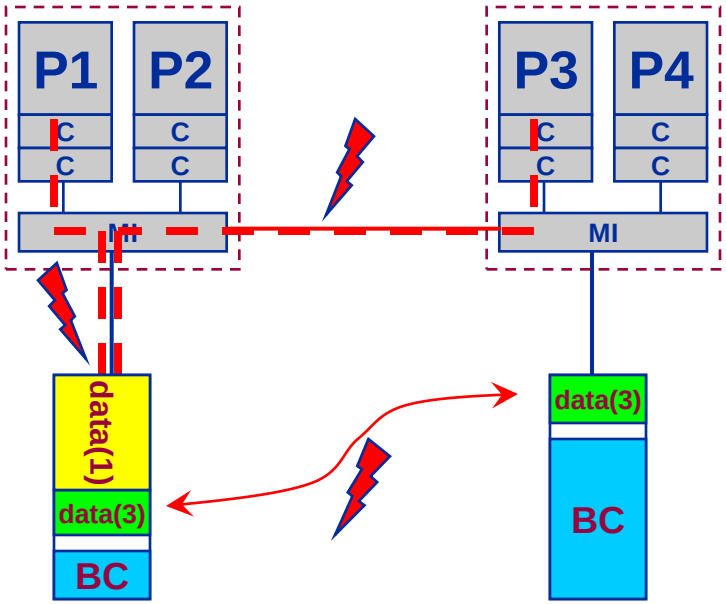
- **Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?**
  - Program has erratic access patters → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

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

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

ccNUMA problems beyond first touch:
Buffer cache

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

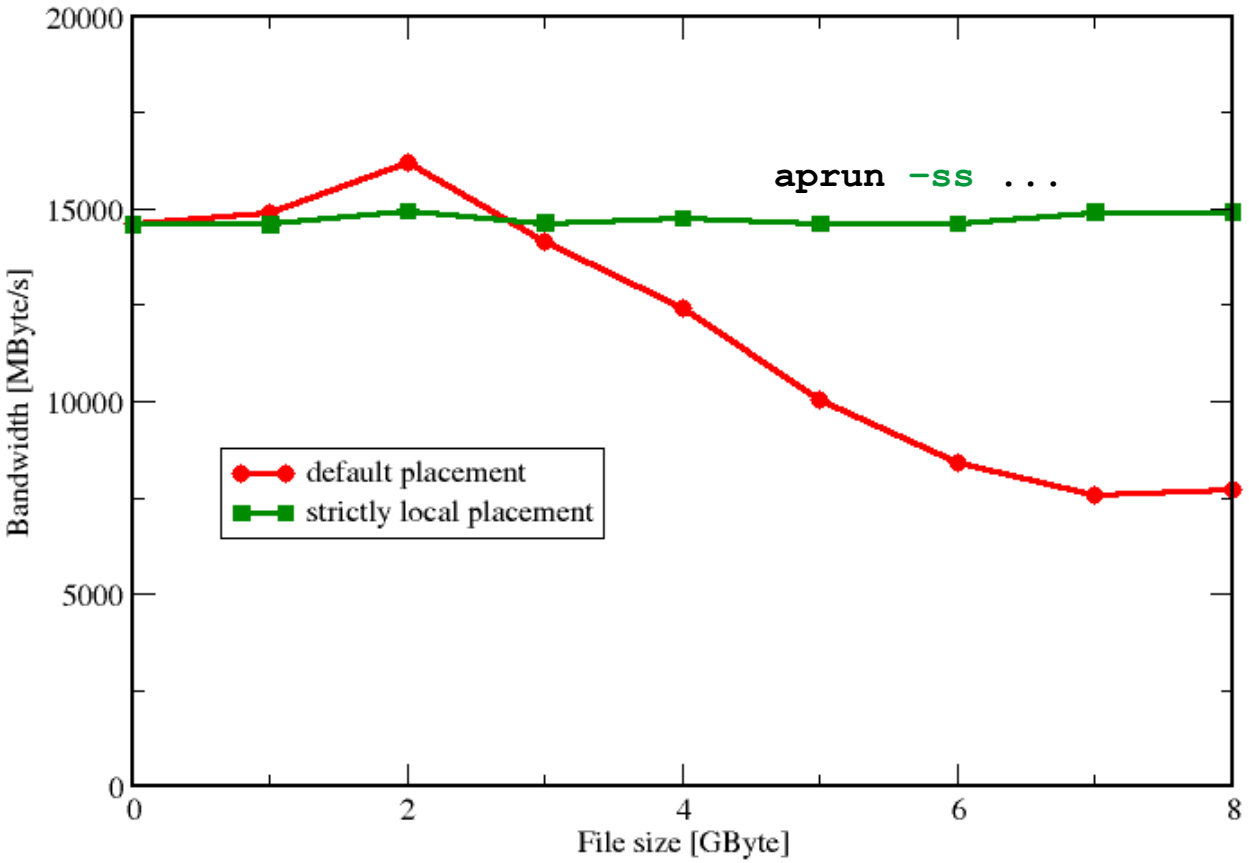


- **Remedies**
  - Drop FS cache pages after user job has run (admin's job)
    - seems to be automatic after aprun has finished on Crays
  - User can run "sweeper" code that allocates and touches all physical memory before starting the real application
  - **`numactl`** tool or **`aprun`** can force 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 available in LD0



Result: By default, Buffer cache is given priority over local page placement

→ restrict to local domain if possible!

ccNUMA placement and erratic access patterns

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

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

- Or you have to use tasking/dynamic scheduling:

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

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

ccNUMA placement and erratic access patterns

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

1. Explicit placement:

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

Observe page alignment of array to get proper placement!

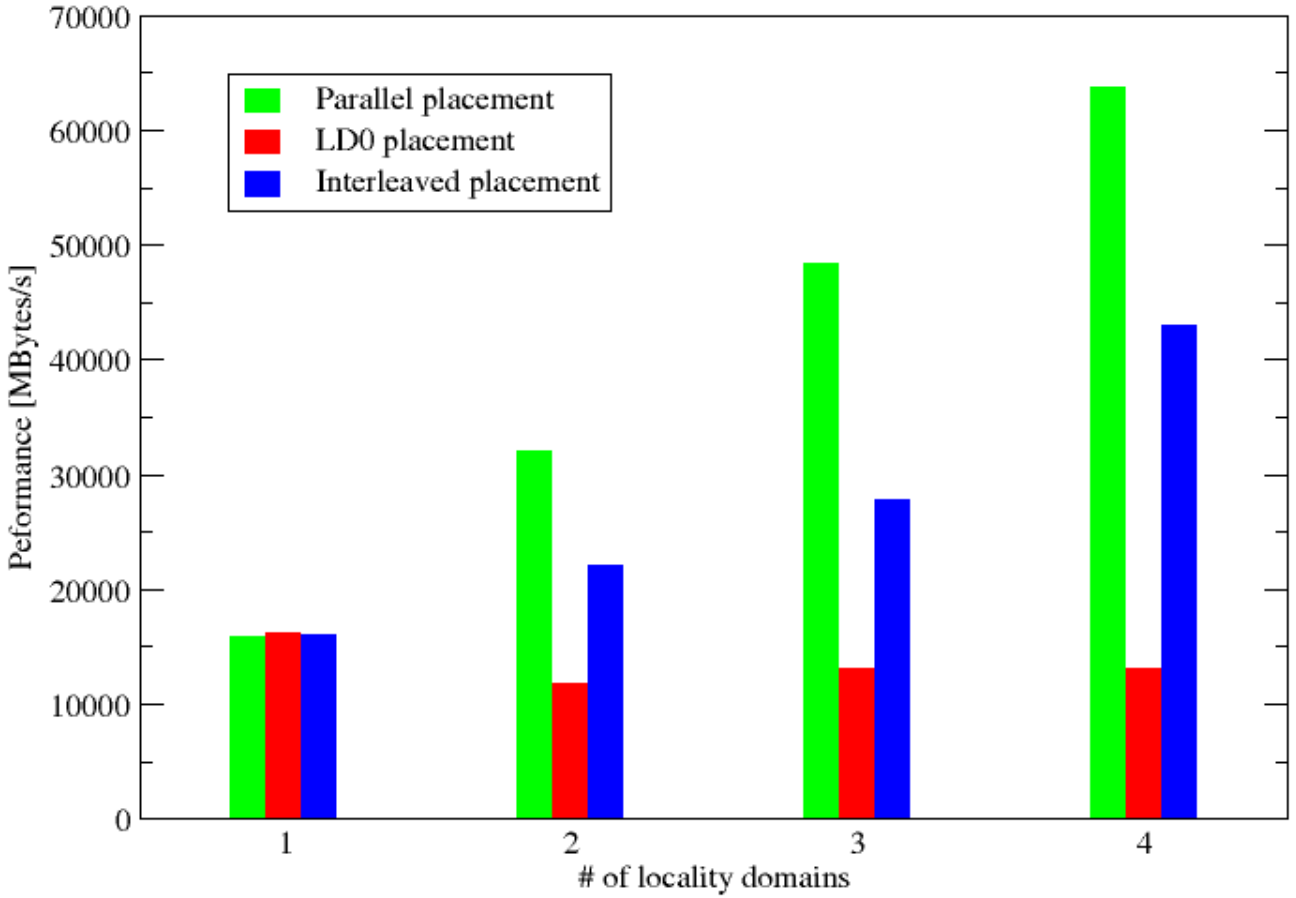
2. Using global control via `numactl`:

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

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

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

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



Conclusions

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 performance