Microbenchmarking for architectural exploration

Probing of the memory hierarchy

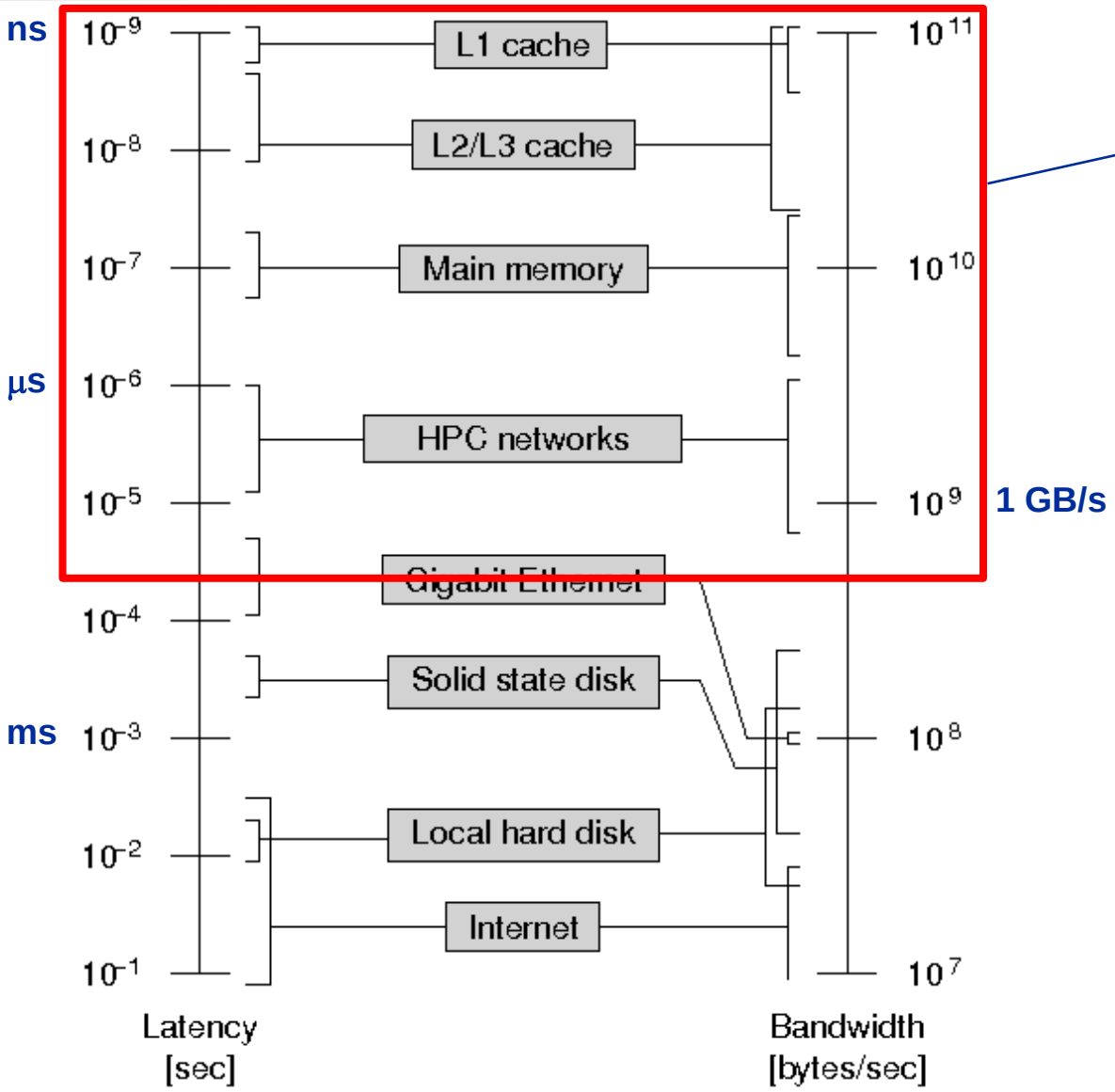
Saturation effects in cache and memory

Typical OpenMP overheads

Motivation for Microbenchmarking as a tool

- **Isolate small kernels to:**
  - Separate influences
  - Determine specific machine capabilities (light speed)
  - Gain experience about software/hardware interaction
  - Determine programming model overhead
  - …
- **Possibilities:**
  - Readymade benchmark collections (epcc OpenMP, IMB)
  - STREAM benchmark for memory bandwidth
  - Implement own benchmarks (difficult and error prone)
  - **`likwid-bench`** tool: Offers collection of benchmarks and framework for rapid development of assembly code kernels

Latency and bandwidth in modern computer environments



HPC plays here

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

Simple streaming benchmark:

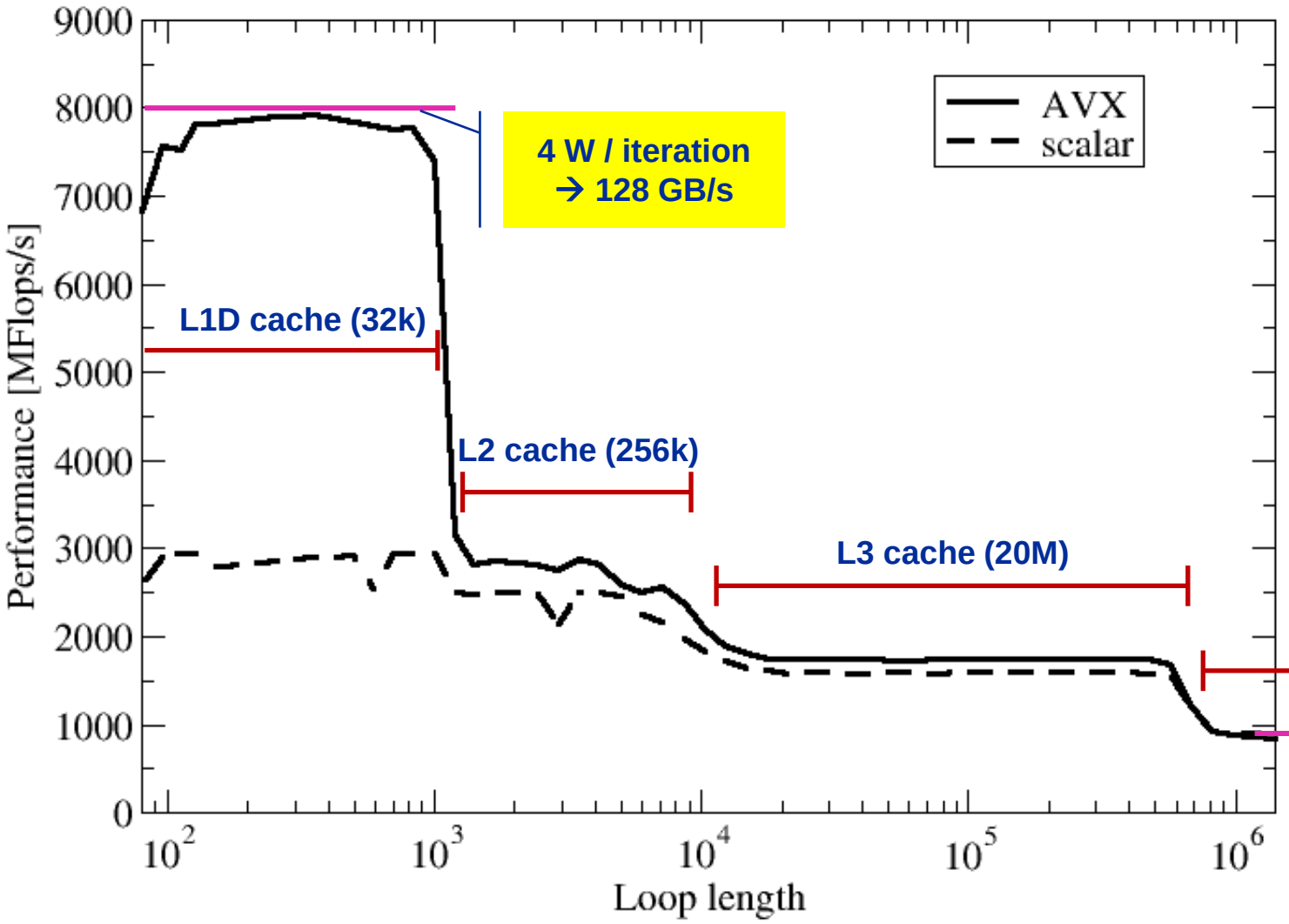
```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

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 Sandy Bridge core (3 GHz)



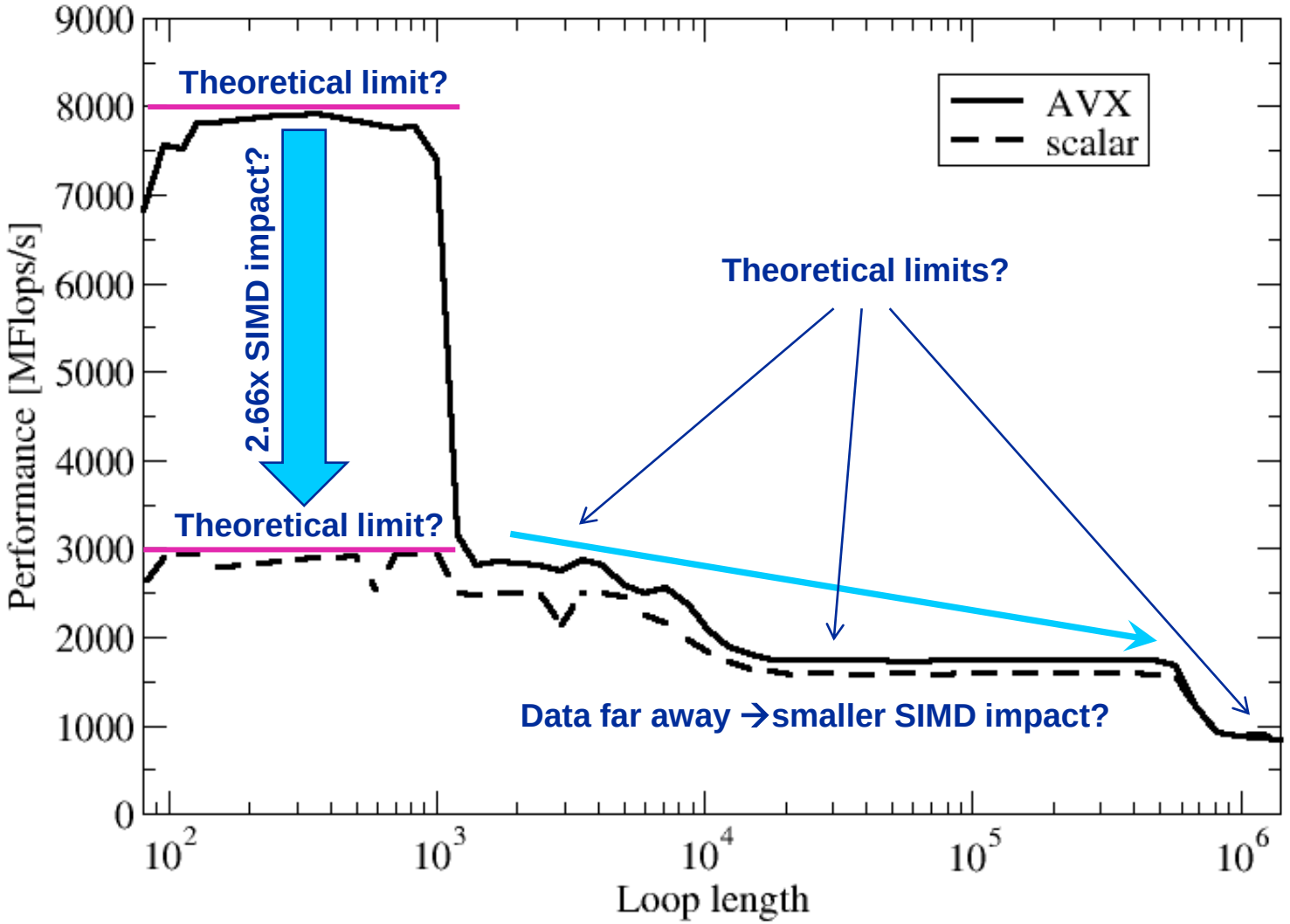
Are the performance levels plausible?

What about multiple cores?

Do the bandwidths scale?

`A(:)=B(:)+C(:)*D(:)` on one Sandy Bridge core (3 GHz):

Observations and further questions



See later for answers!

The throughput-parallel vector triad benchmark

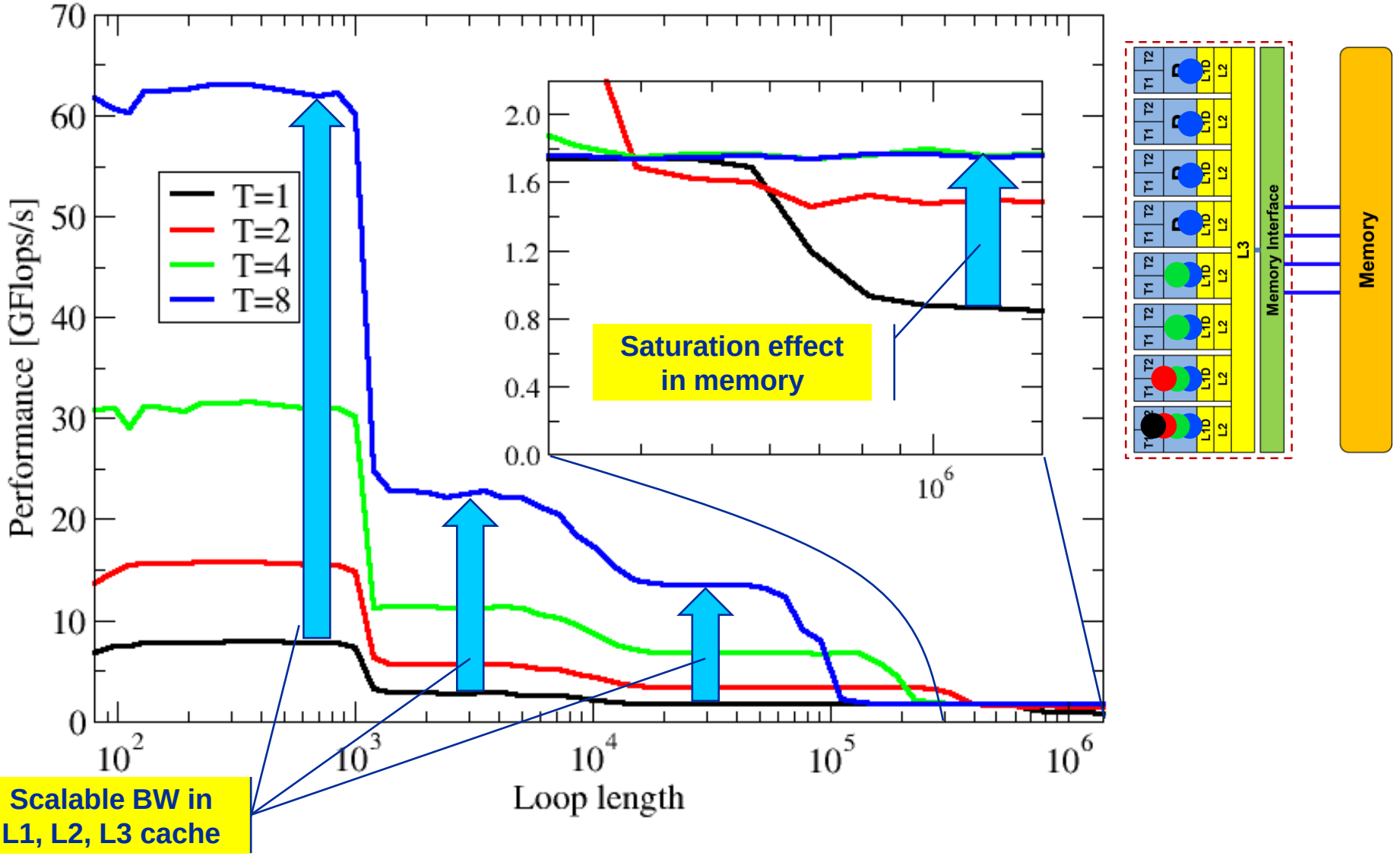
Every core runs its own, independent triad benchmark

```
double precision, dimension(:), allocatable :: A,B,C,D

!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$OMP END PARALLEL
```

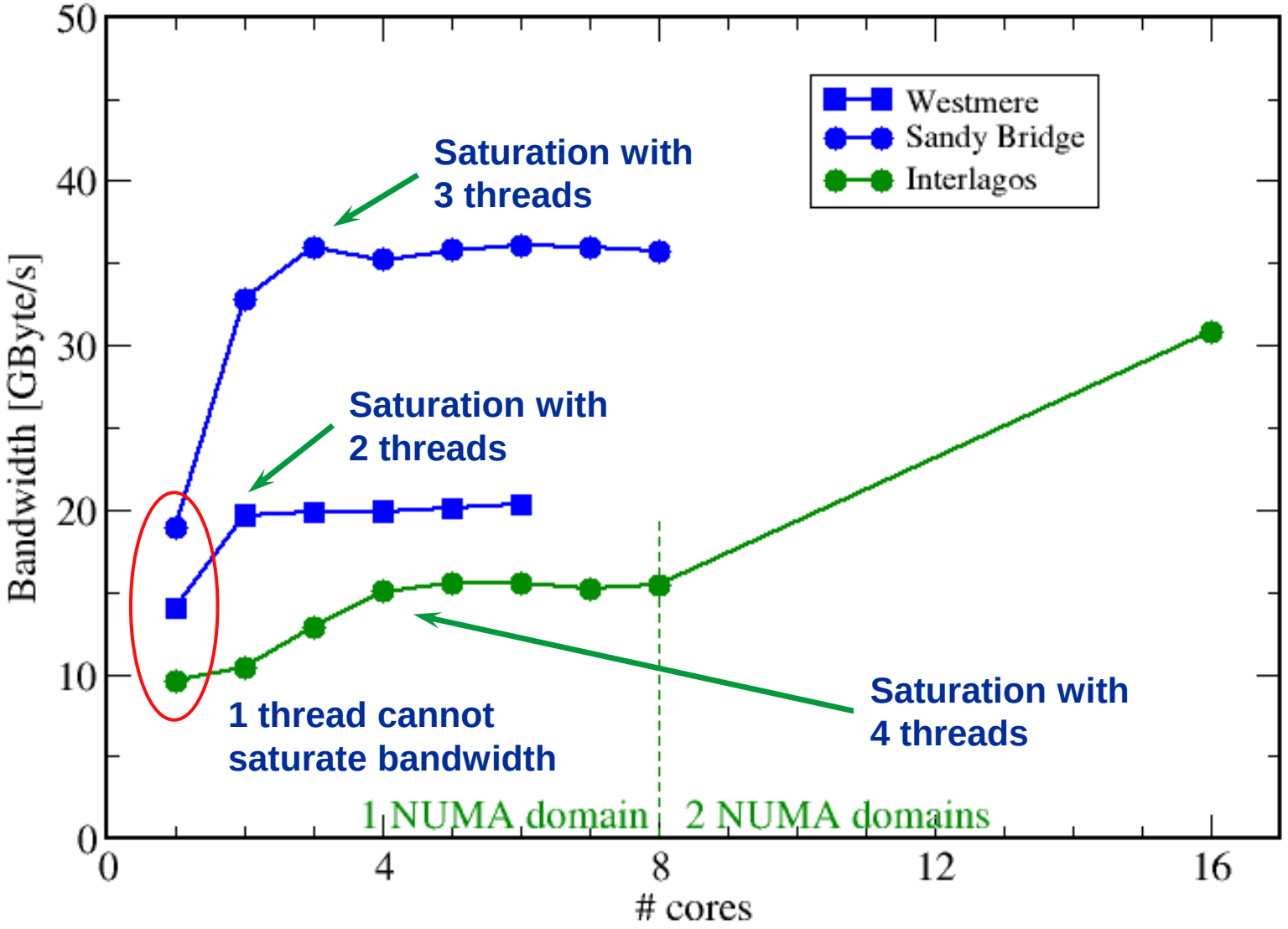
→ pure hardware probing, no impact from OpenMP overhead

Throughput vector triad on Sandy Bridge socket (3 GHz)

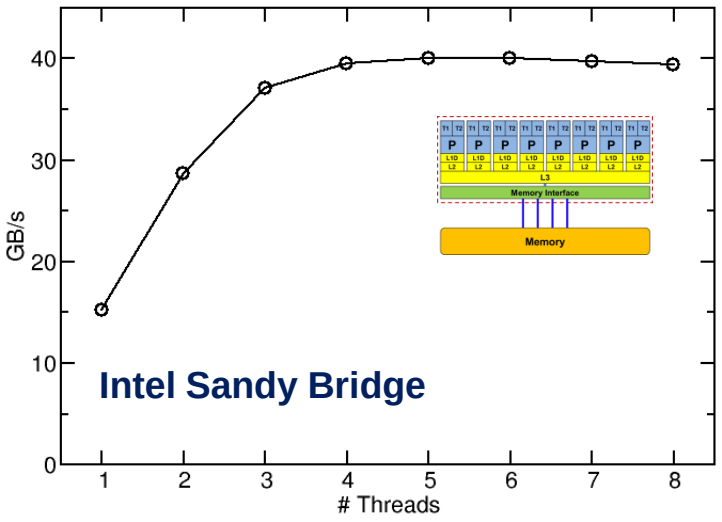


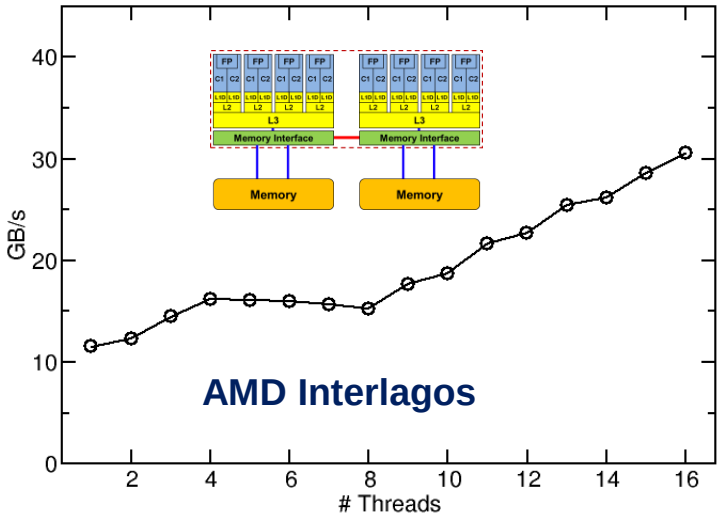
Bandwidth limitations: Main Memory

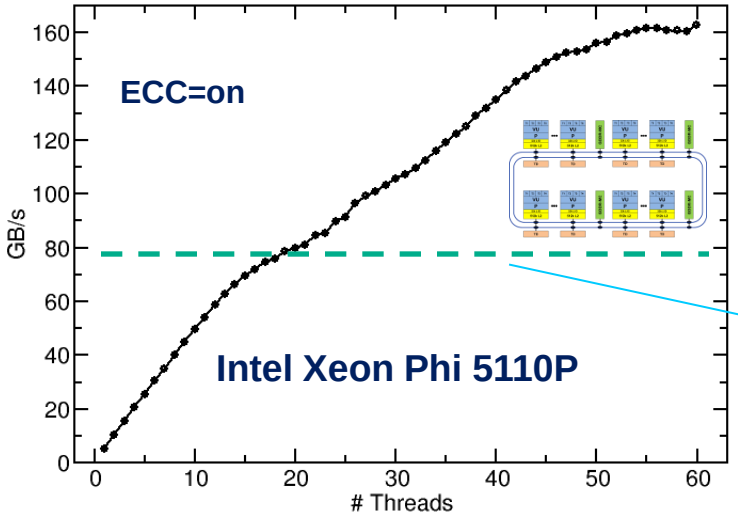
Scalability of shared data paths inside a NUMA domain **(V-Triad)**



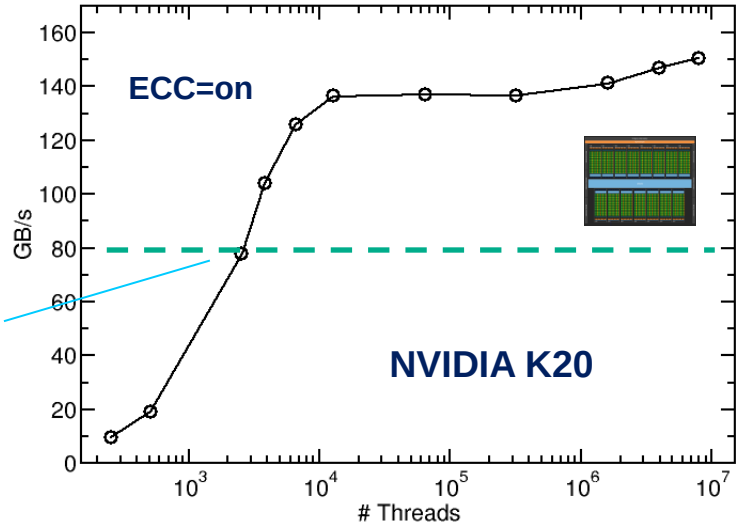
Attainable memory bandwidth: Comparing architectures





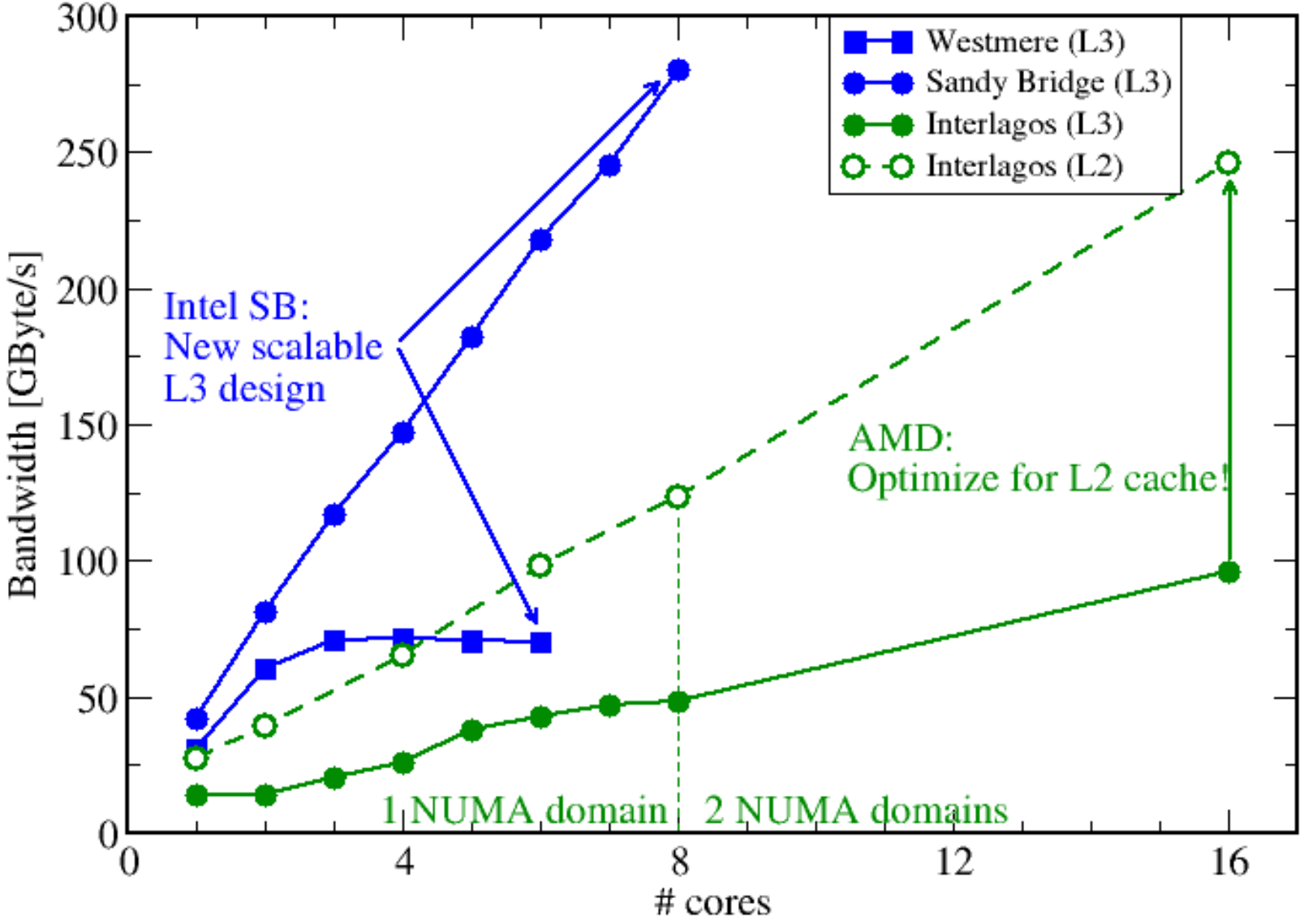


2-socket CPU node



Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache



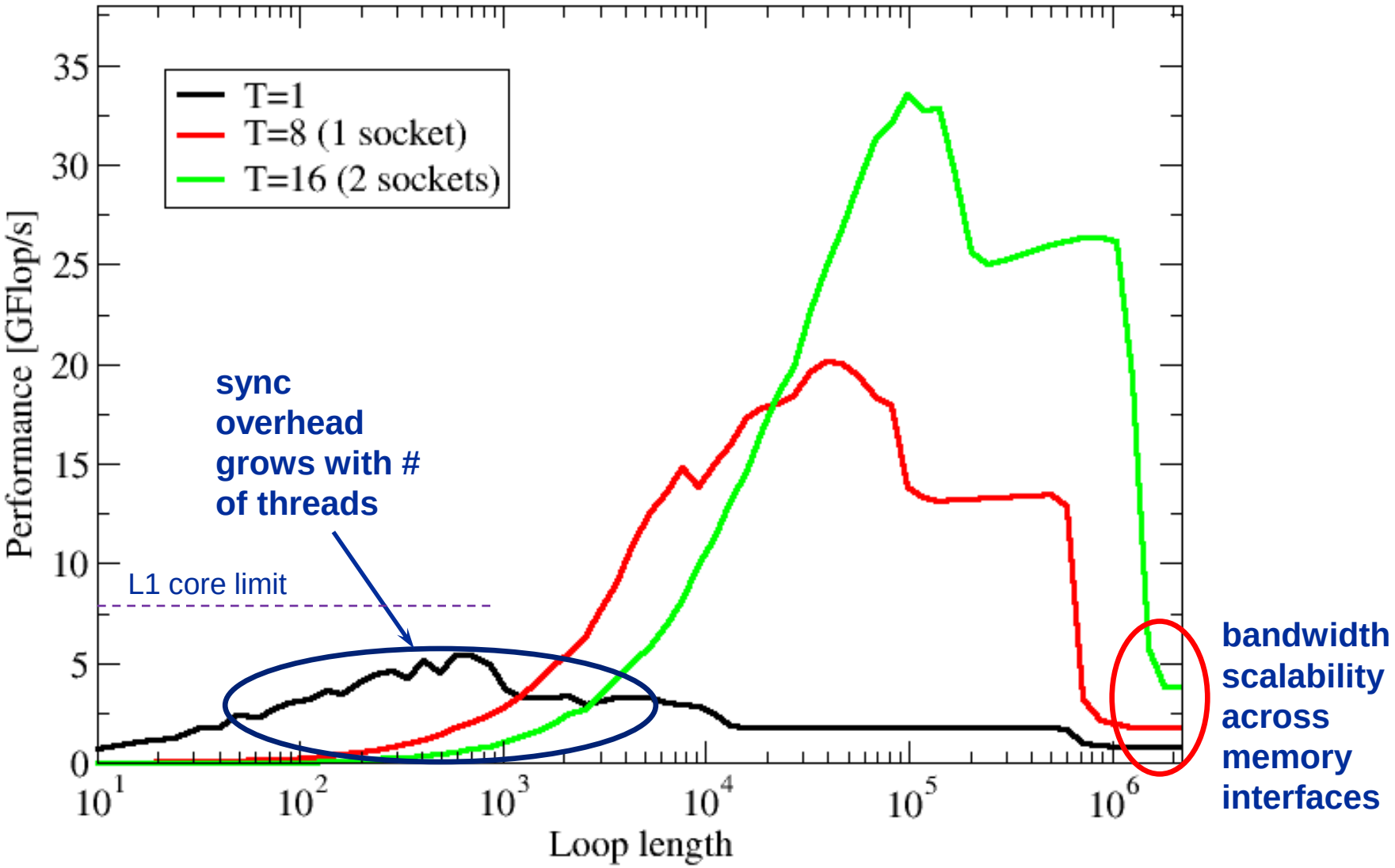
OpenMP work sharing in the benchmark loop

```
double precision, dimension(:), allocatable :: A,B,C,D

allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
!$OMP PARALLEL private(i,j)
do j=1,NITER
!$OMP DO
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP END DO
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$OMP END PARALLEL
```

Implicit barrier

OpenMP vector triad on Sandy Bridge socket (3 GHz)



OpenMP performance issues on multicore

Synchronization (barrier) 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 SandyBridge-EP

Barrier overhead in CPU cycles

2 Threads	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Shared L3	**384**	5242	4616
SMT threads	**2509**	3726	3399
Other socket	1375	5959	4909

Gcc still not very competitive

Intel compiler 

Full domain	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Socket	1497	14546	14418
Node	3401	34667	29788
Node +SMT	6881	59038	58898

Thread synchronization overhead on Intel Xeon Phi

Barrier overhead in CPU cycles

2 threads on distinct cores: 1936

	SMT1	SMT2	SMT3	SMT4
One core	n/a	**1597**	2825	3557
Full chip	10604	12800	15573	**18490**

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Sandy Bridge node

3.75x cores (16 vs 60) on Phi

2x more operations per cycle on Phi

→ 2 · 3.75 = 7.5x more work done on Xeon Phi per cycle

2.7x more barrier penalty (cycles) on Phi

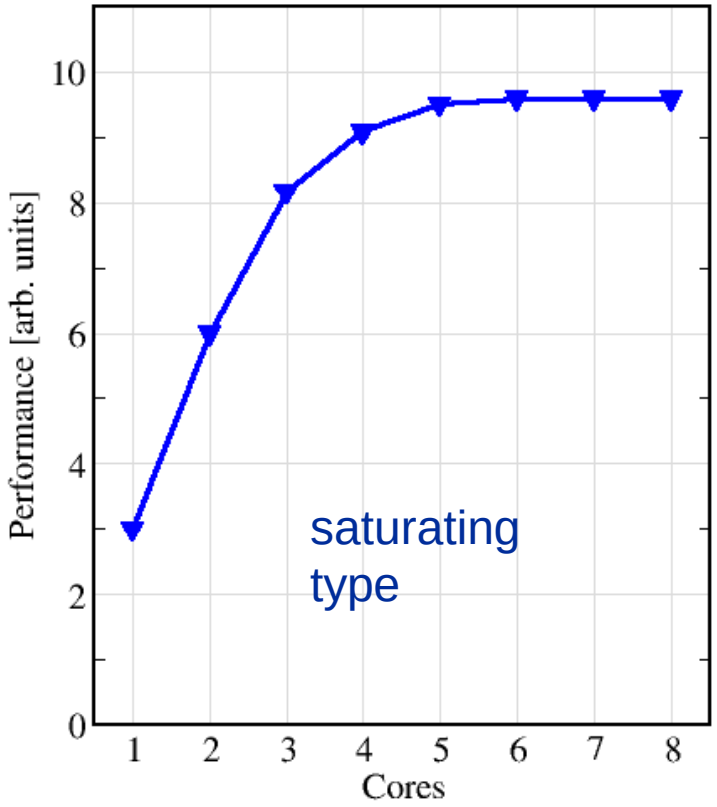
→ One barrier causes 2.7 · 7.5 ≈ 20x more pain ☺.

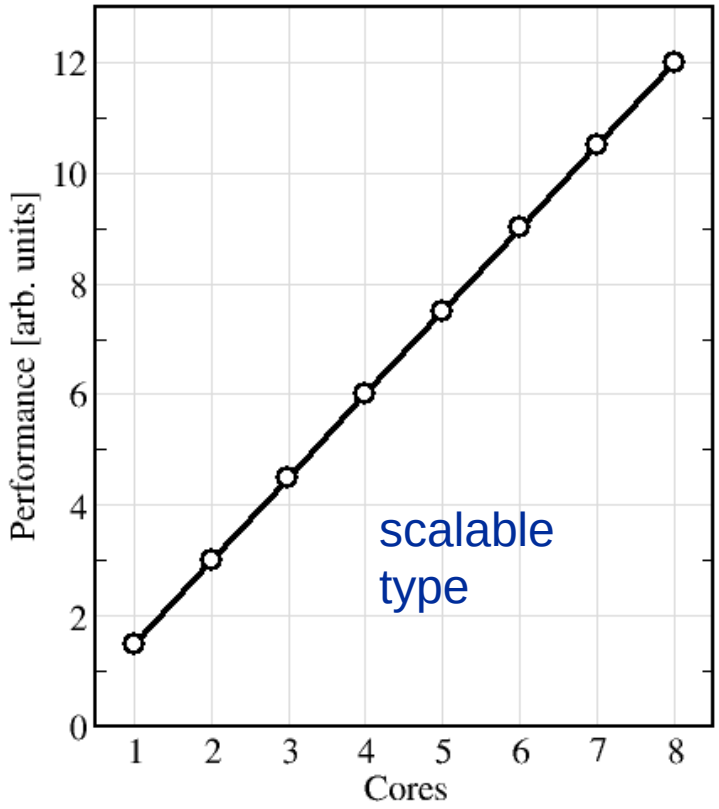
Conclusions from the microbenchmarks

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - Know where your threads are running
    - Know where your data is
- **Bandwidth bottlenecks are ubiquitous**
- **Synchronization overhead may be an issue**
  - … and also depends on affinity!
  - Many-core poses new challenges in terms of synchronization

Epilogue: Consequences from the saturation pattern

- Clearly distinguish between "**saturating**" and "**scalable**" performance on the chip level





Epilogue: Consequences from the saturation pattern

- There is no clear bottleneck for single-core execution
- Code profile for single thread ≠ code profile for multiple threads
  - → Single-threaded profiling may be misleading