



Parallel Algorithm Engineering

Kenneth S. Bøgh
PhD Fellow

Based on slides by
Darius Sidlauskas

Outline

- Background
- Current multicore architectures
- UMA vs NUMA
- The openMP framework
- Examples

Software crisis

“The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

-- E. Dijkstra, 1972 Turing Award Lecture

Before

- The 1st Software Crisis
 - When: around '60 and 70'
 - Problem: large programs written in assembly
 - Solution: abstraction and portability via high-level languages like C and FORTRAN
- The 2nd Software Crisis
 - When: around '80 and '90
 - Problem: building and maintaining large programs written by hundreds of programmers
 - Solution: software as a process (OOP, testing, code reviews, design patterns)
 - Also better tools: IDEs, version control, component libraries, etc.

Recently..

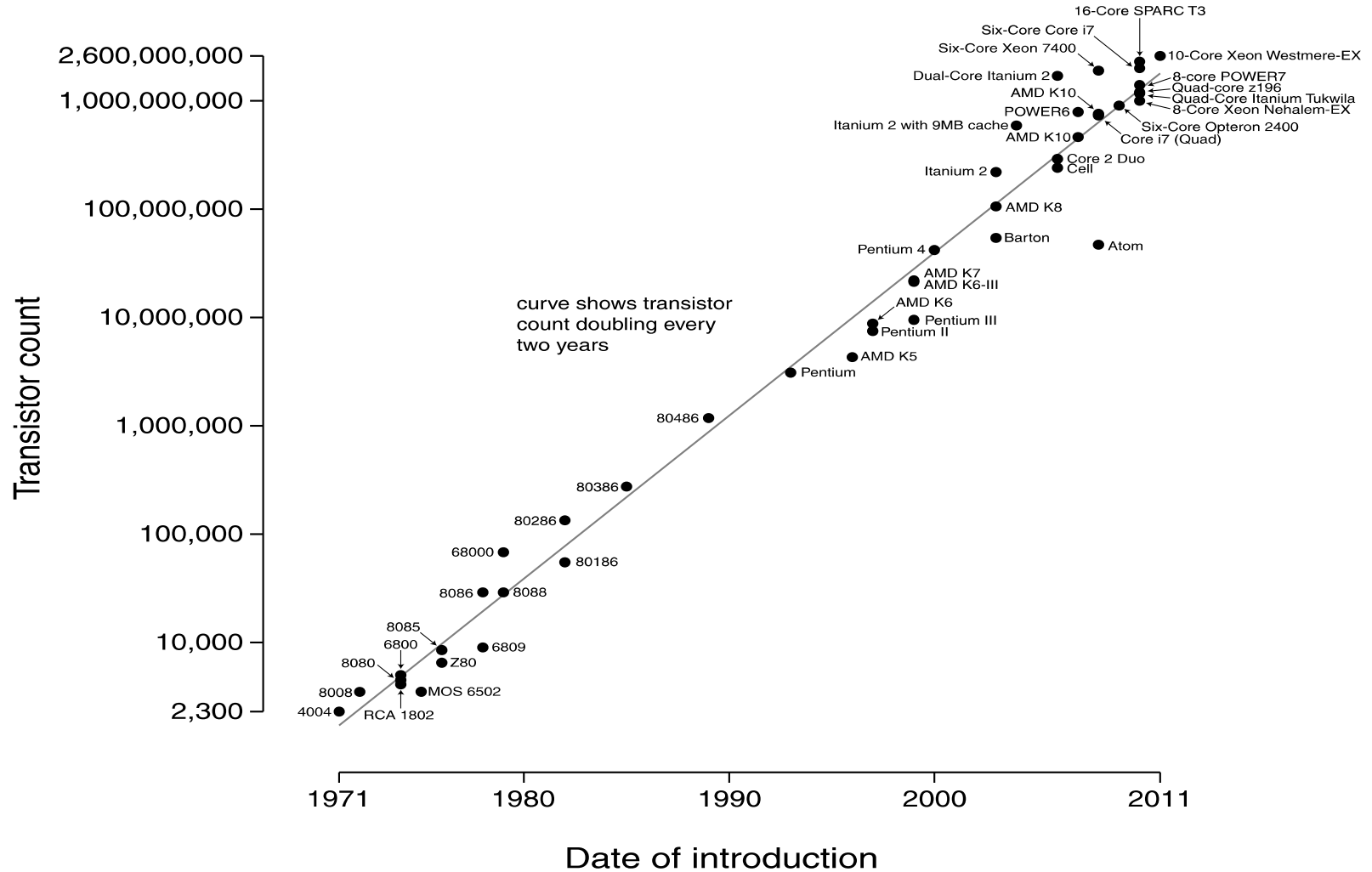
- Processor-oblivious programmers
 - A Java program written on PC works on your phone
 - A C program written in '70 still works today and is faster
 - Moore's law takes care of good speedups

Currently..

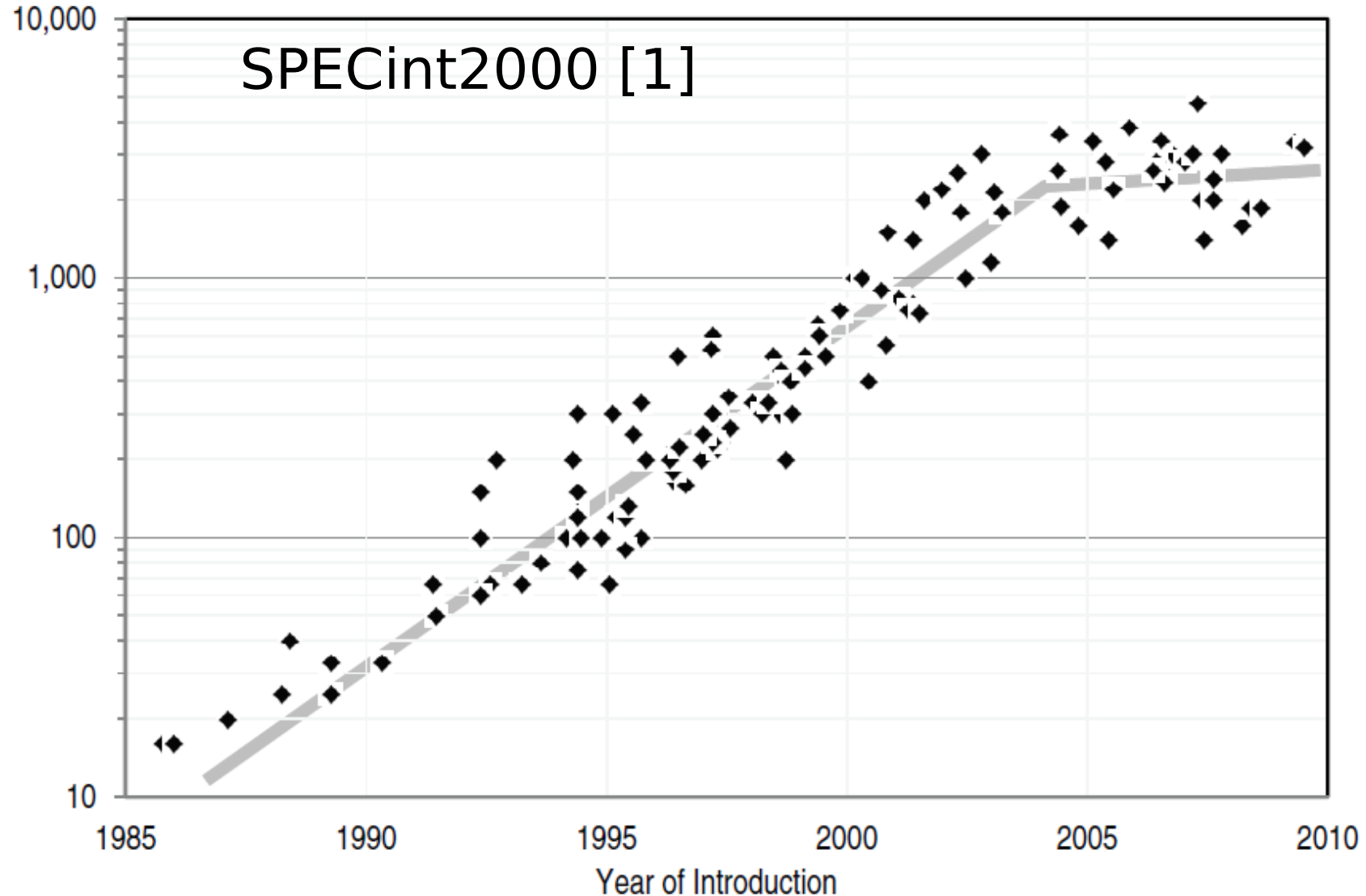
- Software crisis again?
 - When: 2005 and ...
 - Problem: sequential performance is stuck
 - Required solution: continuous and reasonable performance improvements
 - To process large datasets (BIG Data!)
 - To support new features
 - Without loosing portability and maintainability

Moore's law

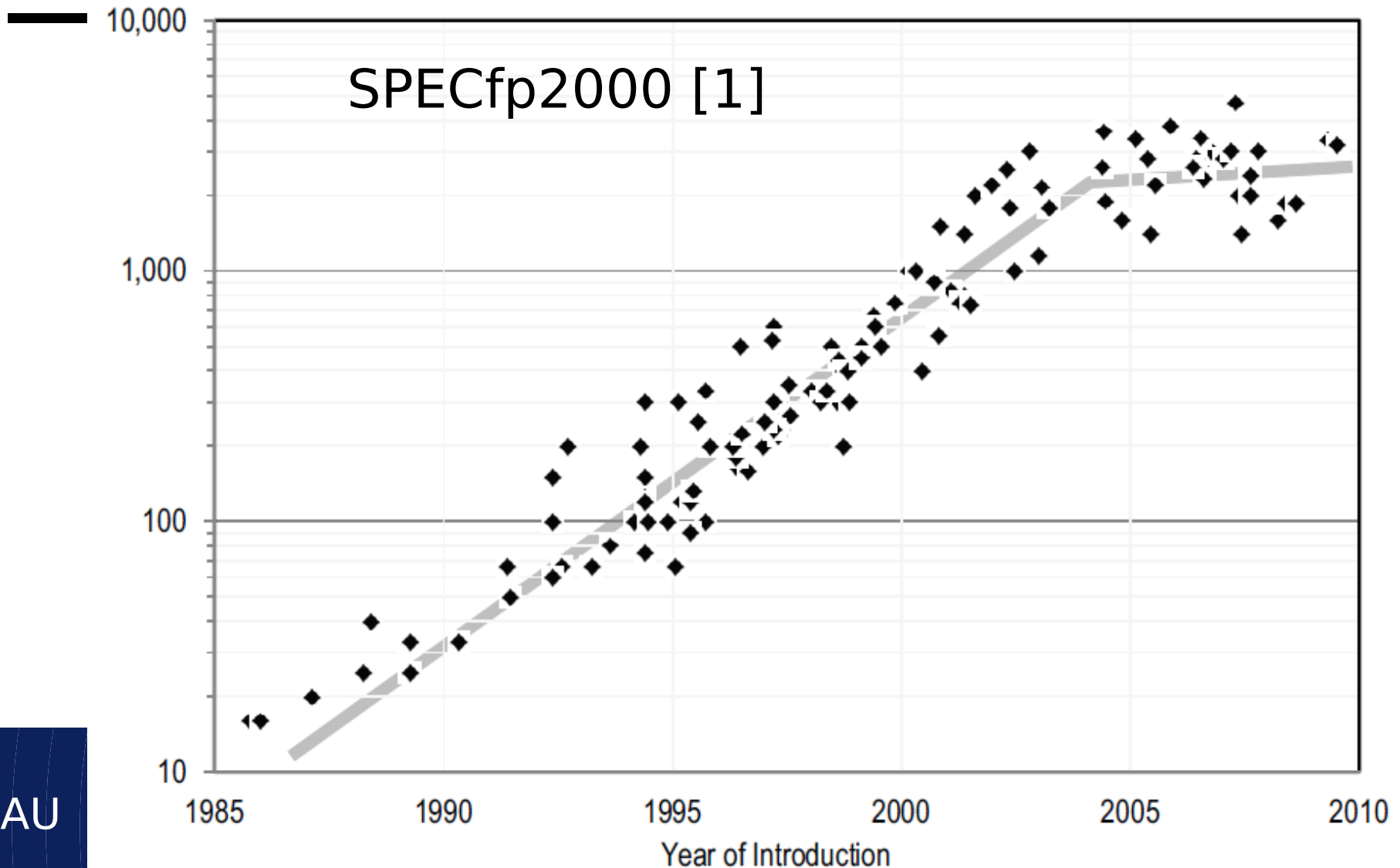
Microprocessor Transistor Counts 1971-2011 & Moore's Law



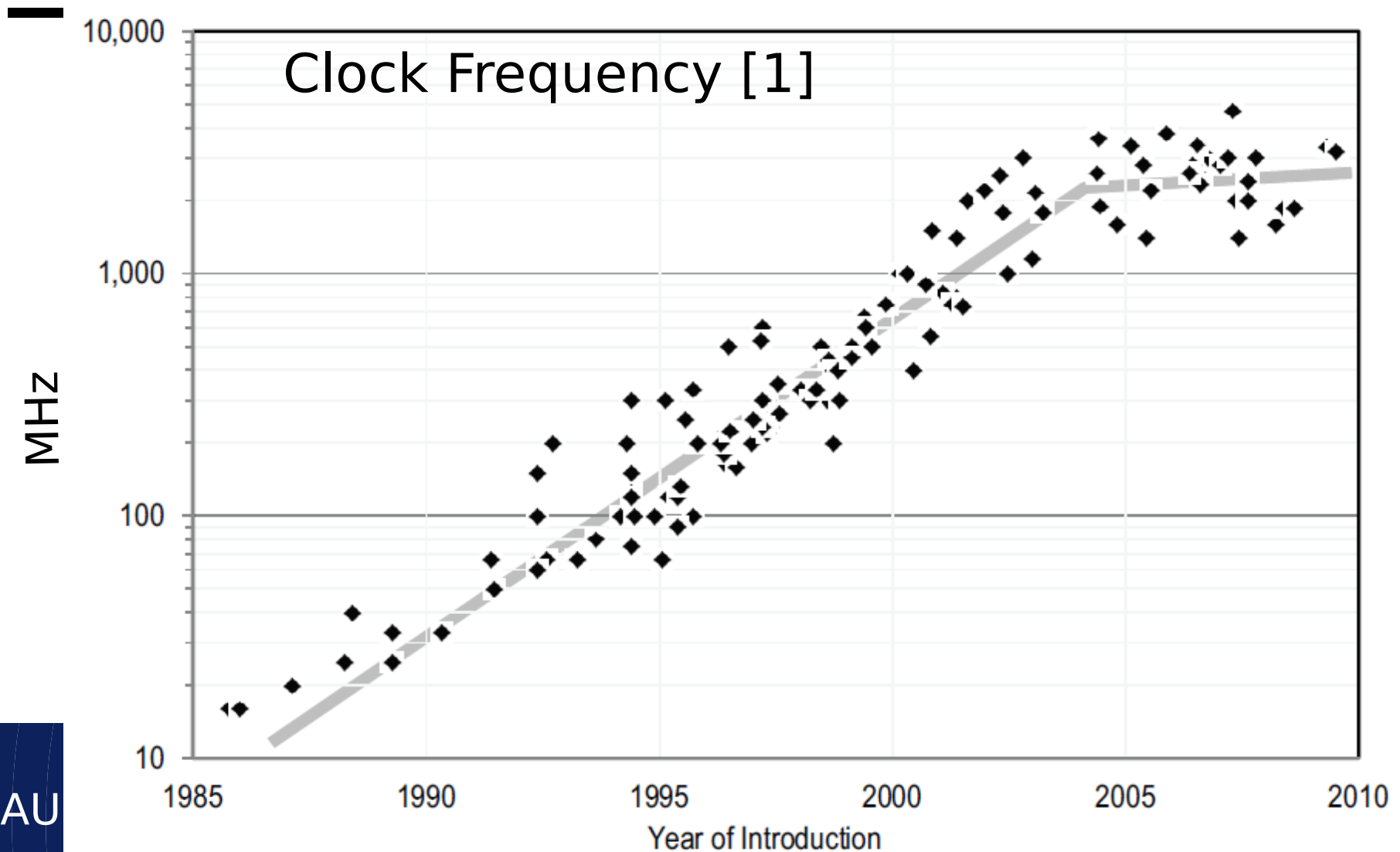
Uniprocessor performance



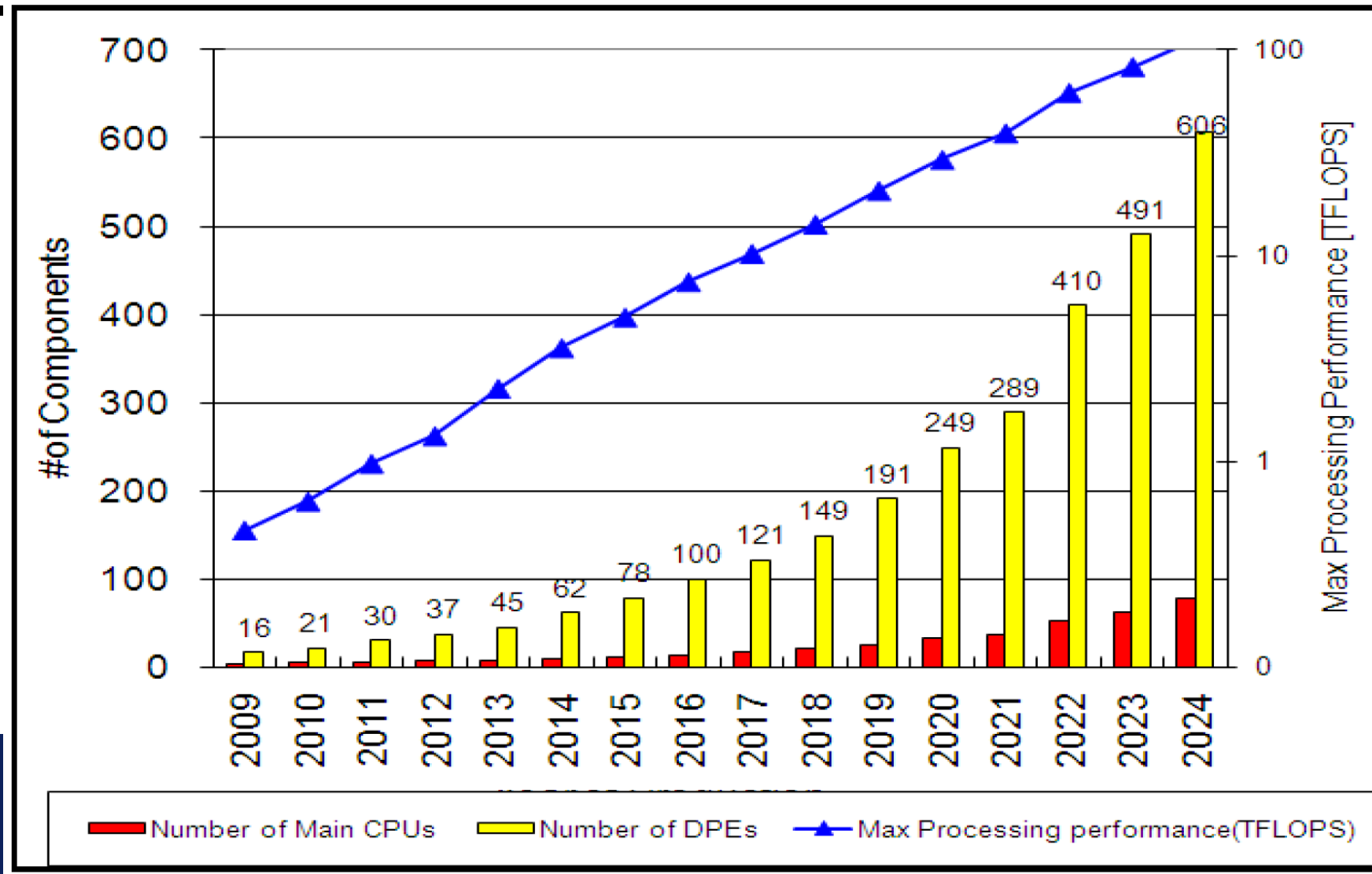
Uniprocessor performance (cont.)



Uniprocessor performance (cont.)



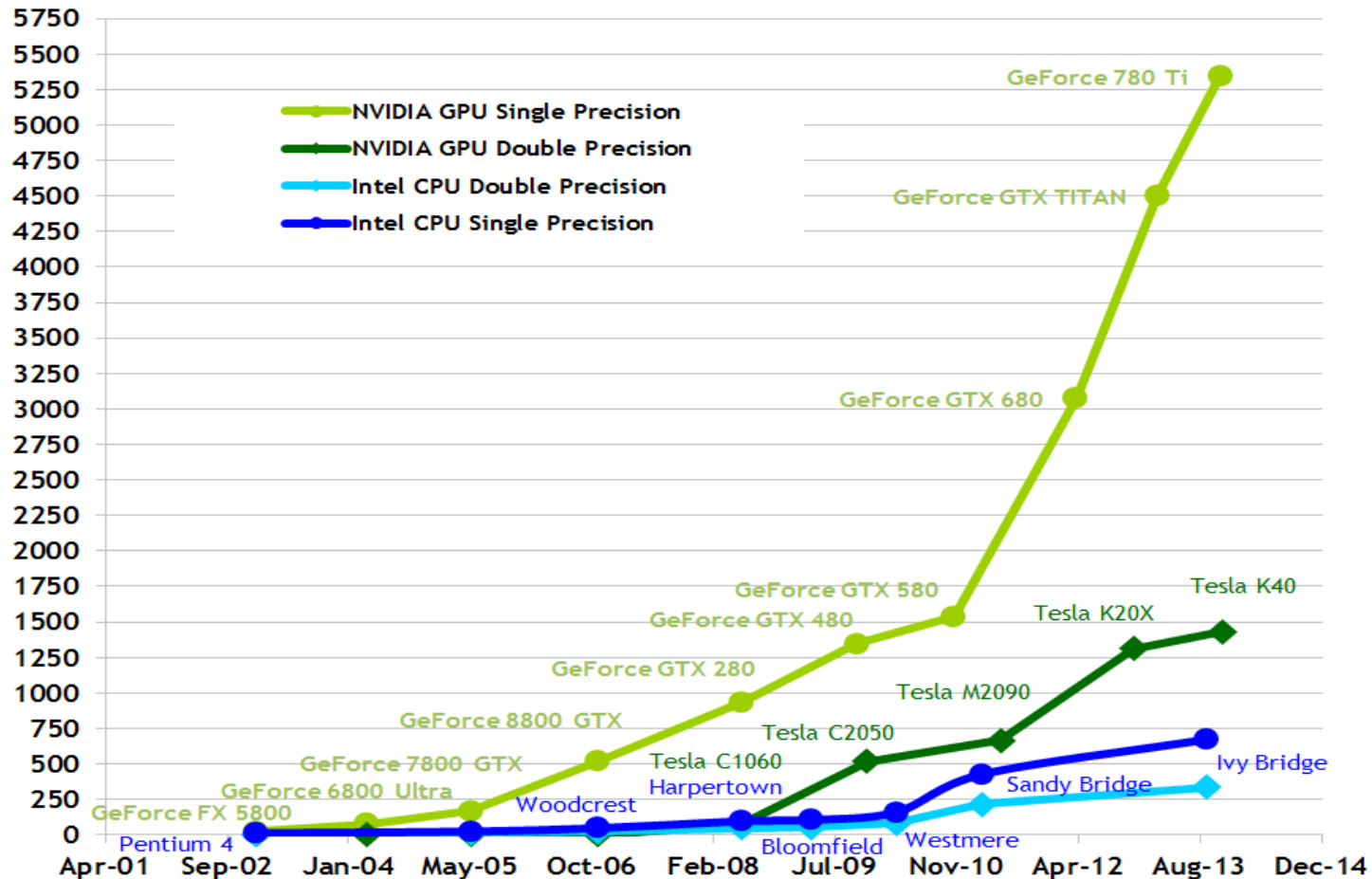
Parallel processing: Predicted # of cores for stationary systems, according to ITRS



Even “worse” for GPUs

- GTX 780 Ti have 2880 cores @ 0.9GHz

Theoretical GFLOP/s



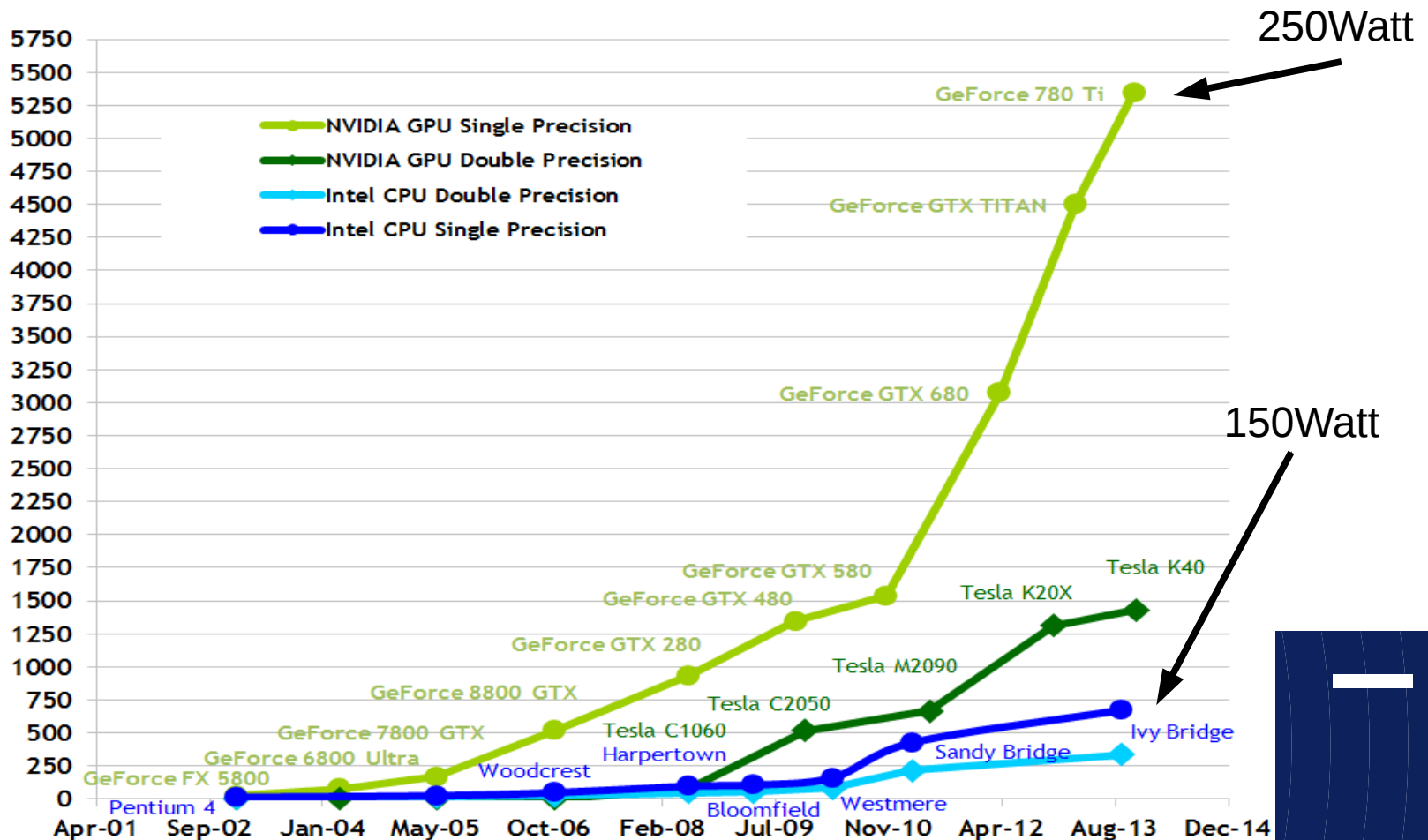
Why

- Power considerations
 - Consumption
 - Cooling
 - Efficiency
- DRAM access latency
 - Memory wall
- Wire delays
 - Range of wire in one clock cycle
- Diminishing returns of more instruction-level parallelism
 - Out-of-order execution, branch prediction, etc.

Power consumptions

- GTX 780 Ti have 2880 cores @ 0.9GHz

Theoretical GFLOP/s

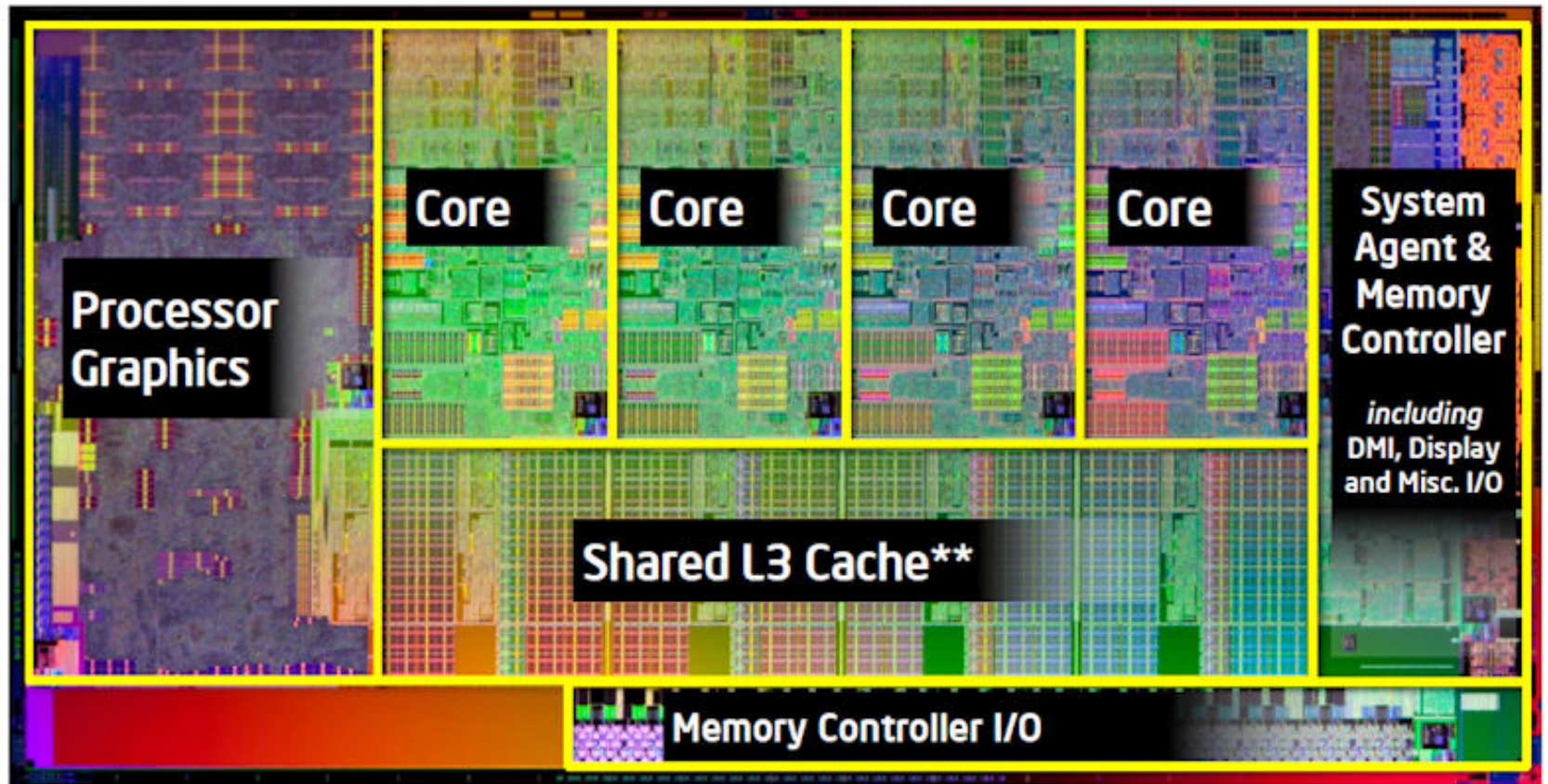


Overclocking

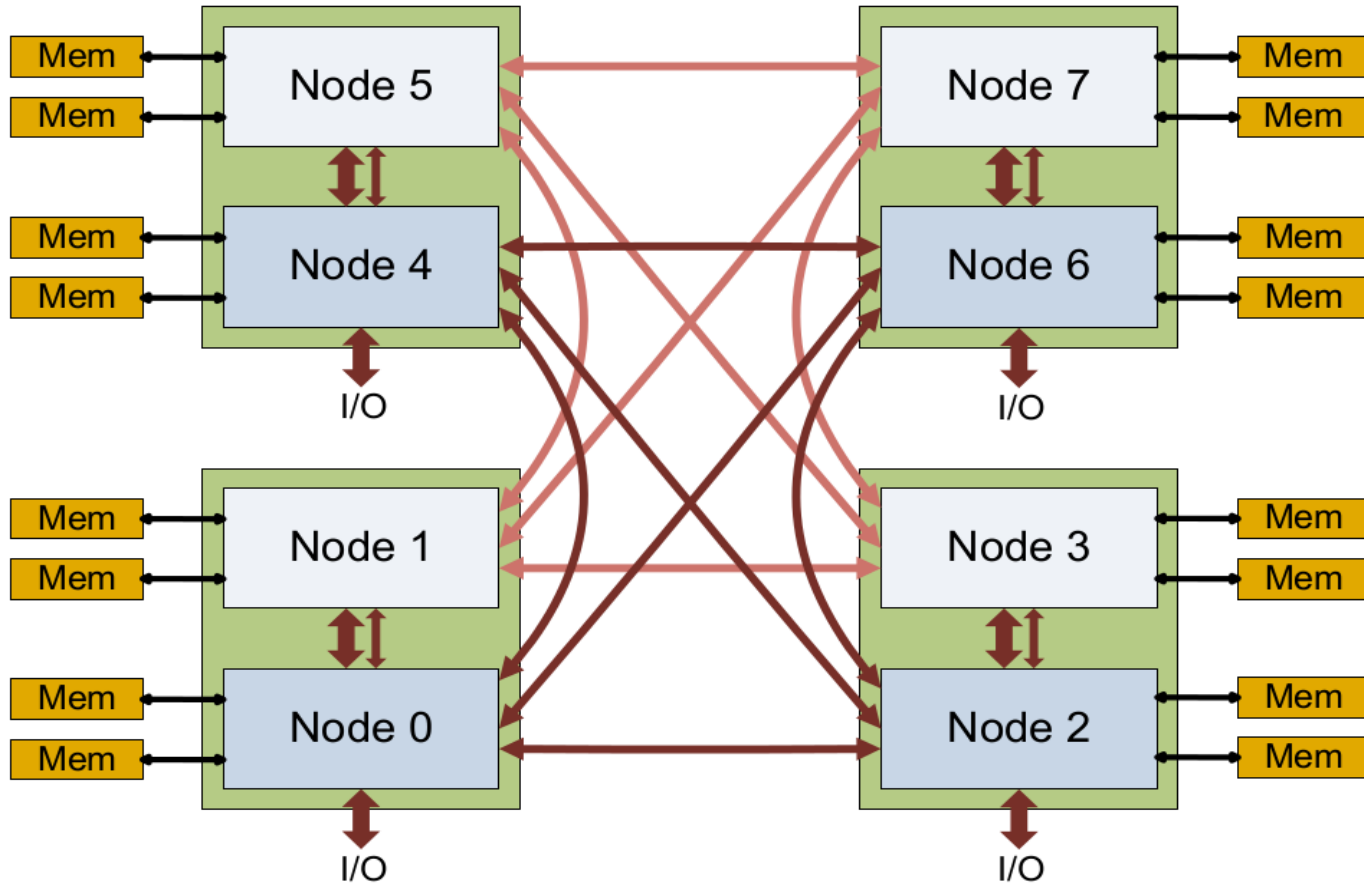
- Air-water: ~5.0 GHz
 - Possible at home
- Phase change: ~6.0 GHz
- Liquid helium: 8.794 GHz
 - Current world record
 - Reached with AMD FX-8350

Towards parallel setups

- Instead of going faster --> go more parallel!
 - Transistors are used now for multiple cores

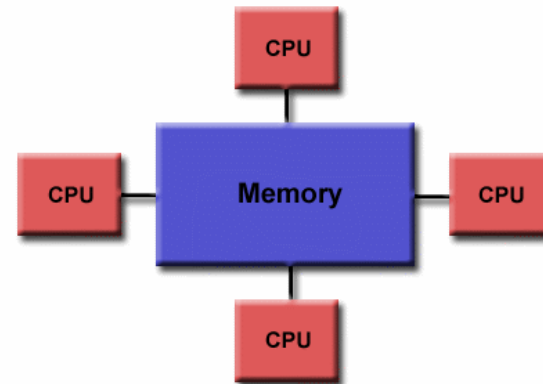
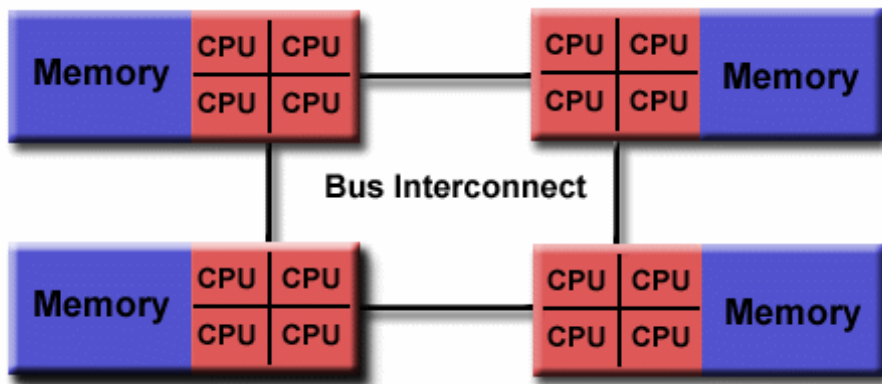


4 sockets – 8 CPU setup



UMA vs NUMA

- All laptops and most desktops are UMA
- Most modern servers are NUMA
- Important to know which you target!



Current commercial MC CPUs

- Intel

- Intel® Core™ i7-5960X: 8-core (16 threads), 20 MB Cache, max 3.5 GHz

- Intel® Xeon® Processor E5-2699 v3: 18-core (36 threads), 45 MB Cache, max 3.6 GHz (x 8-socket configuration)

- Phi 7120P: 61 cores (244 threads), 30.5 MB Cache, max 1.33 GHz, max memory BW 352 GB/s

- AMD

- FX-9590: 8-core, 8 MB Cache, 4.7 GHz

- A10-7850K: 12-core (4 CPU 4 GHz + 8 GPU 0.72 GHz), 4 MB C

- Opteron 6386 SE: 16-core, 16 MB Cache, 3.5 GHz (x 4-socket conf.)

- Oracle

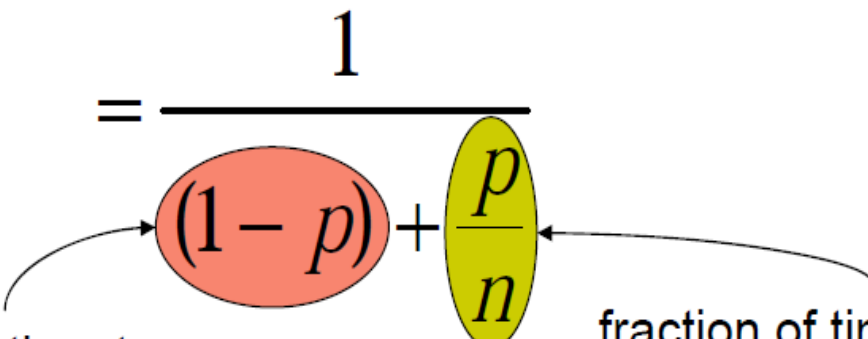
- SPARC M6: 12-core (96 threads), 48 MB Cache, 3.6 GHz (x 32-socket configuration)

Concurrency vs Parallelism

- Parallelism
 - A condition that arises when at least two threads are executing simultaneously
 - A specific case of concurrency
- Concurrency:
 - A condition that exists when at least two threads are making progress.
 - A more generalized form of parallelism
 - E.g., concurrent execution via time-slicing in uniprocessors (virtual parallelism)
- Distribution:
 - As above but running simultaneously on different machines (e.g., cloud computing)

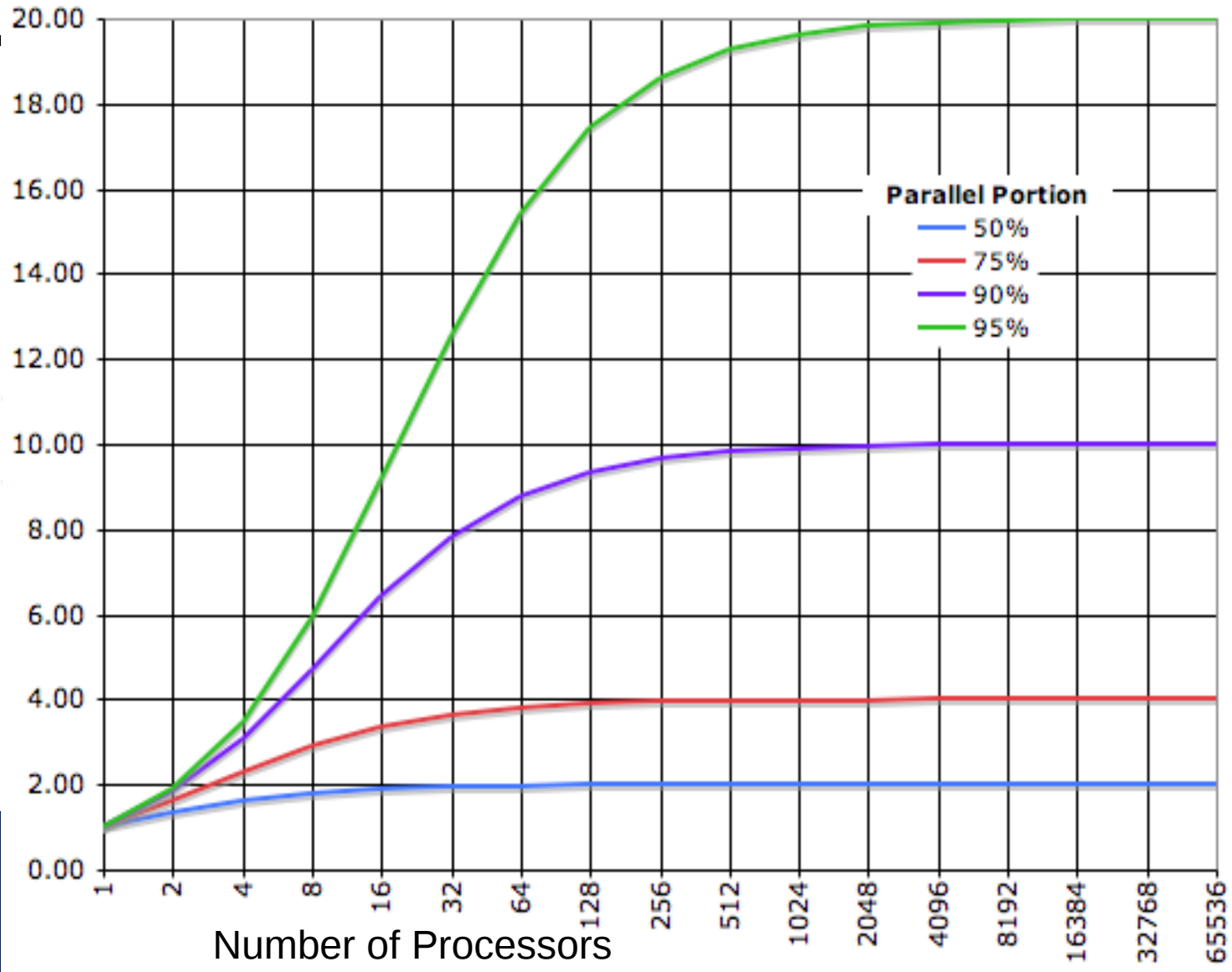
Amdahls law

- Potential program speedup is defined by the fraction of code that can be parallelized
- Serial components rapidly become performance limiters as thread count increases
 - p - fraction of work that can be parallelized
 - n - the number of processors

$$= \frac{1}{(1-p) + \frac{p}{n}}$$


The diagram shows the formula $\frac{1}{(1-p) + \frac{p}{n}}$ with annotations. A red oval contains the term $(1-p)$, with an arrow pointing to the text "fraction of time to complete sequential work". A yellow oval contains the term $\frac{p}{n}$, with an arrow pointing to the text "fraction of time to complete parallel work".

Amdahls law

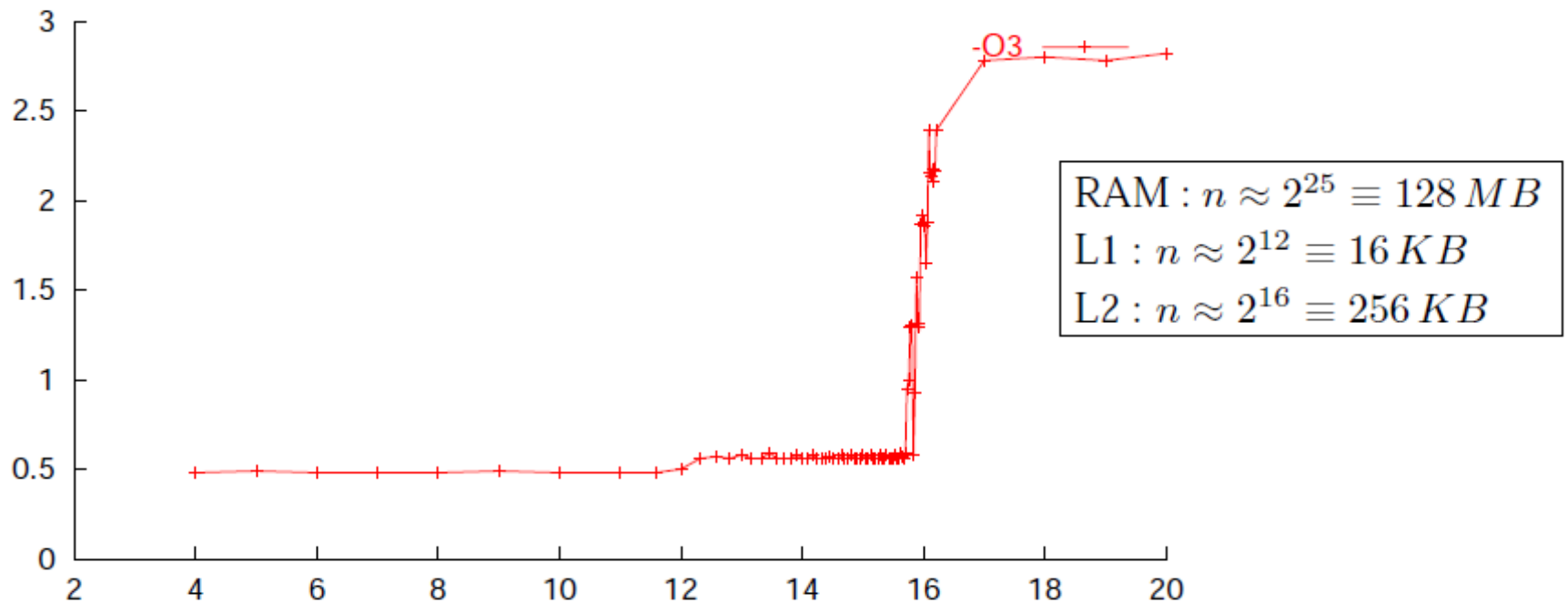


When to parallelize

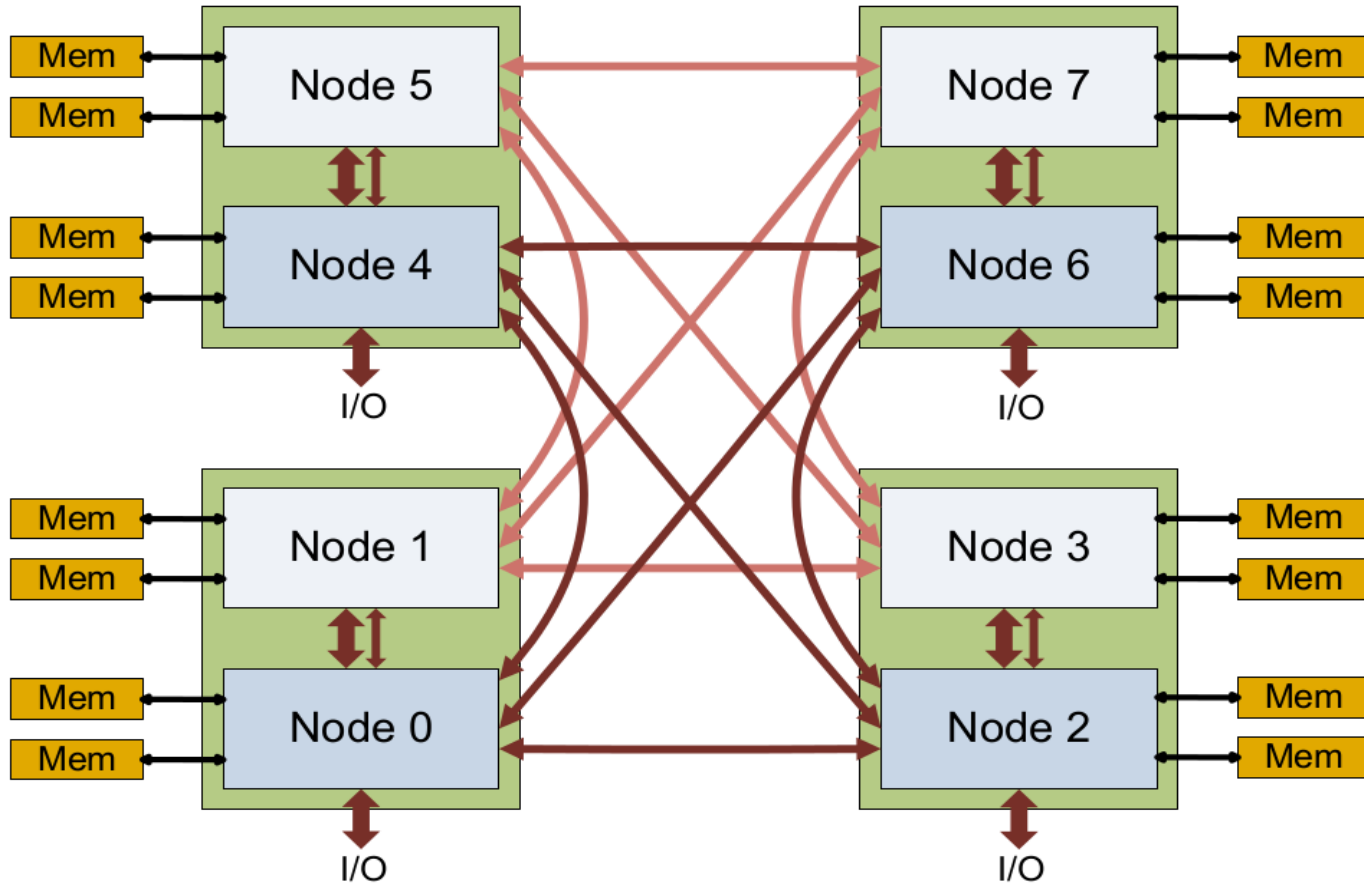
- When you have independent units of work that can be performed
- When your code is compute bound
- Or your code is not utilizing the memory bandwidth
- When you see performance gains in tests :-)

We have seen this previously

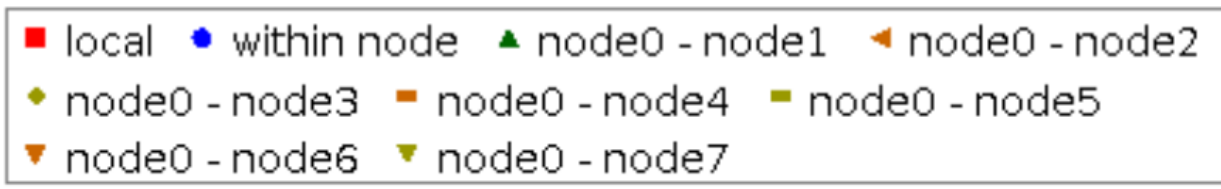
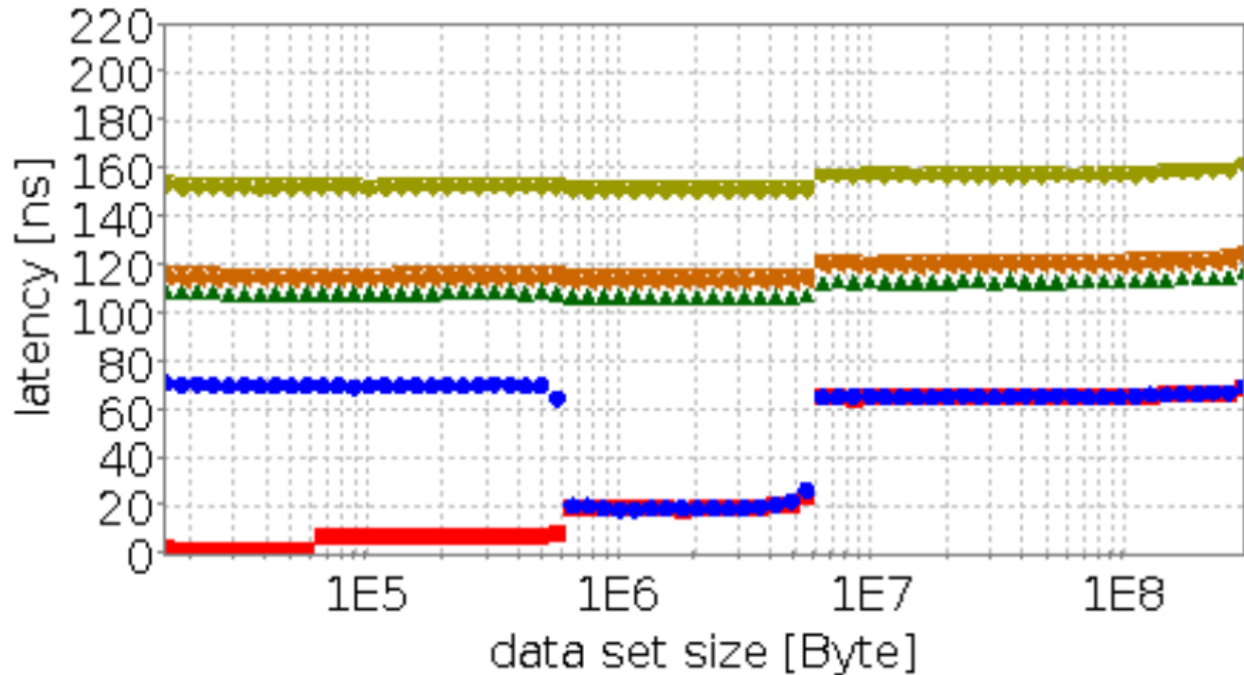
- L1 and L2 cache sizes



Remember from previously

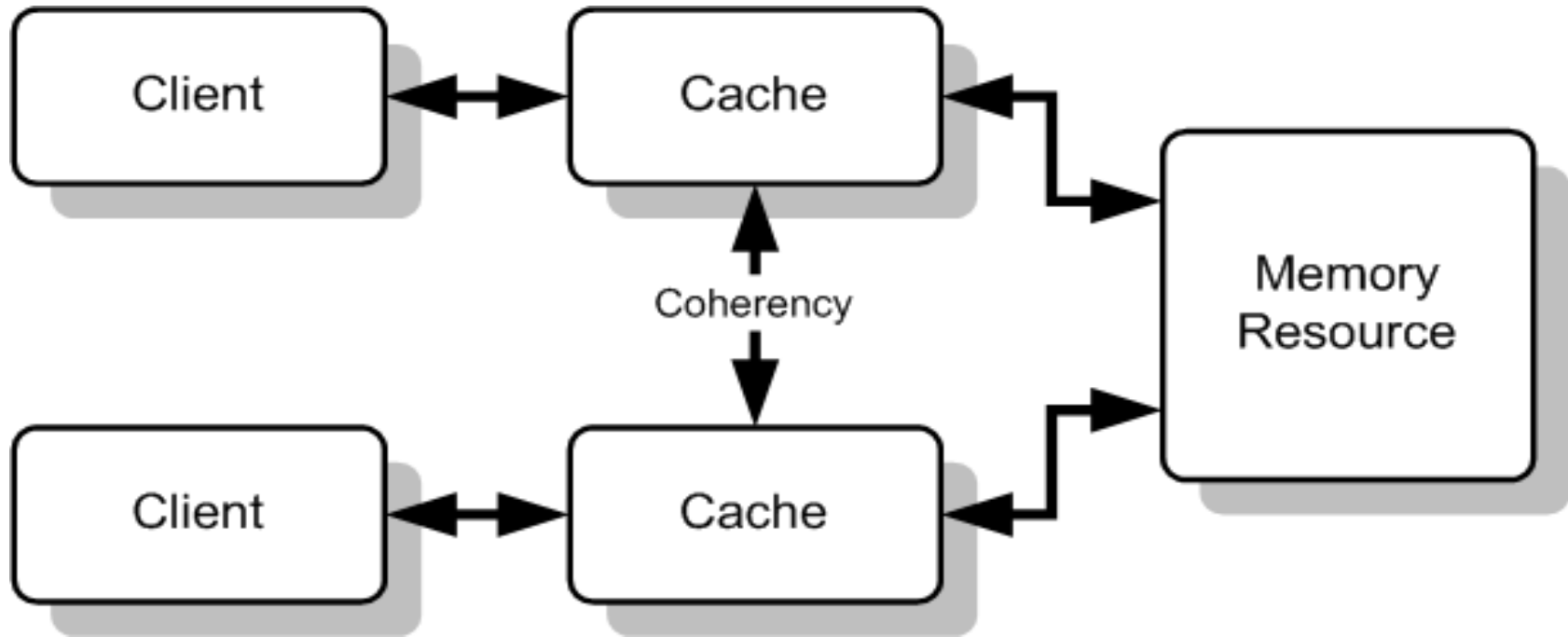


Numa effects



Cache coherence

- Ensures the consistency between all the caches.

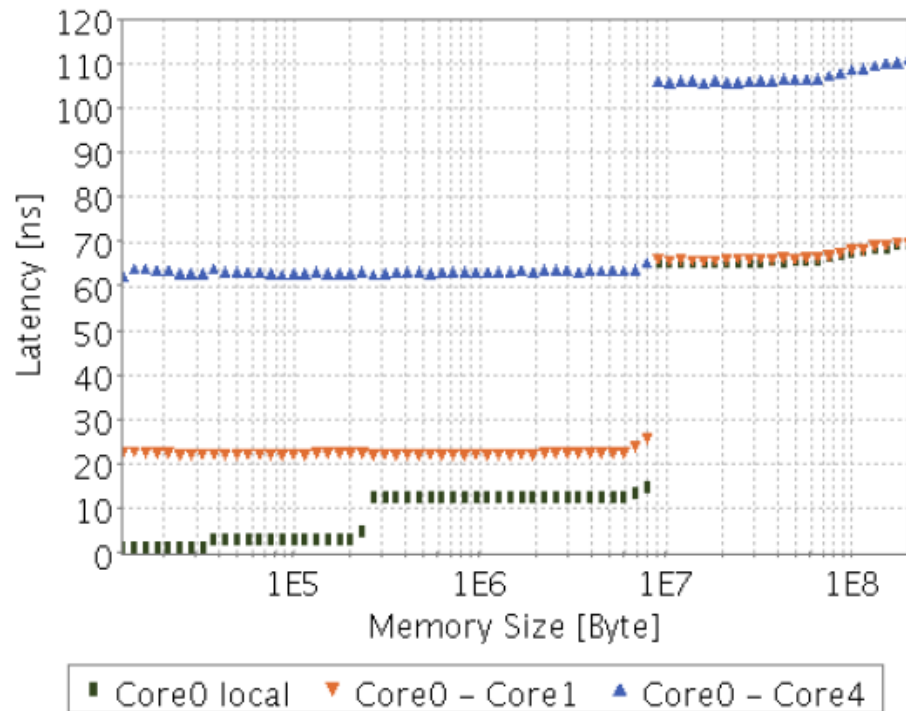


MESIF protocol

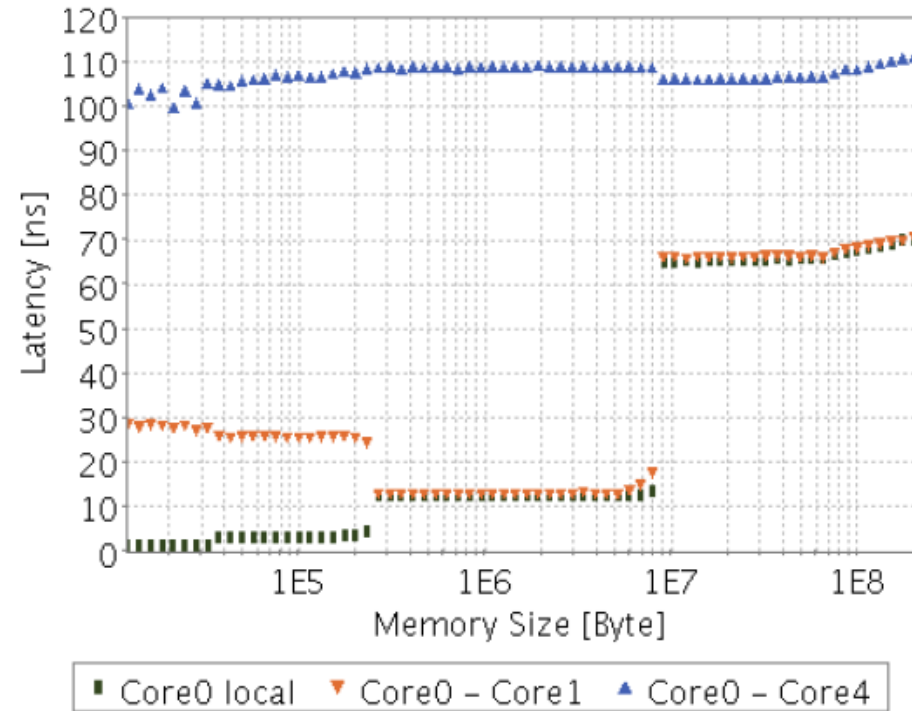
- Modified (M): present only in the current cache and *dirty*. A write-back to main memory will make it (E).
- Exclusive (E): present only in the current cache and *clean*. A read request will make it (S), a write-request will make it (M).
- Shared (S): maybe stored in other caches and *clean*. Maybe changed to (I) at any time.
- Invalid (I): unusable
- Forward (F): a specialized form of the S state

Cache coherency effects

Exclusive cache lines



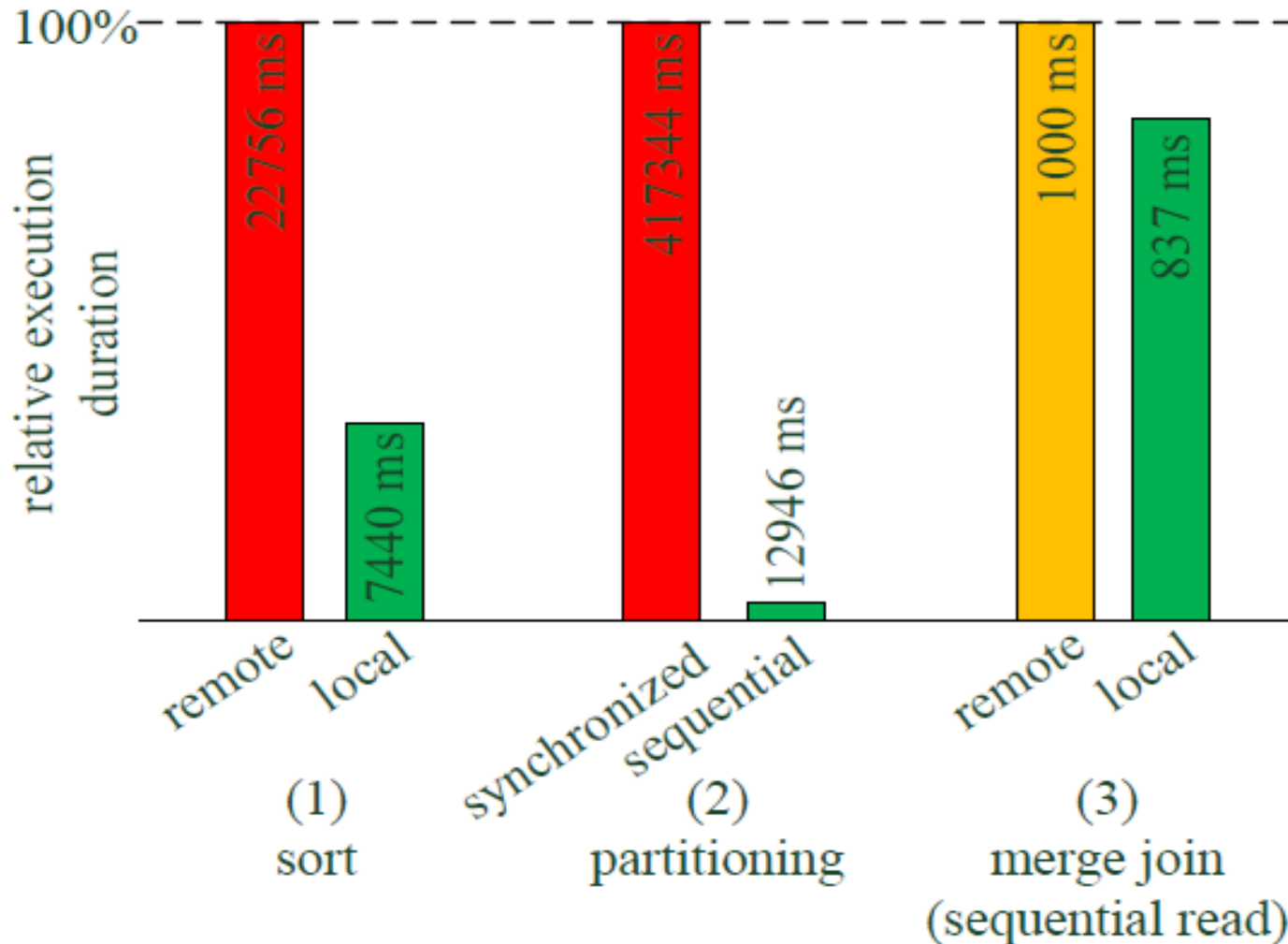
Modified cache lines



Latency in nsec on 2-socket Intel Nehalem

Does it matter?

- Processing 1600M tuples on 32-core machine



Commandments

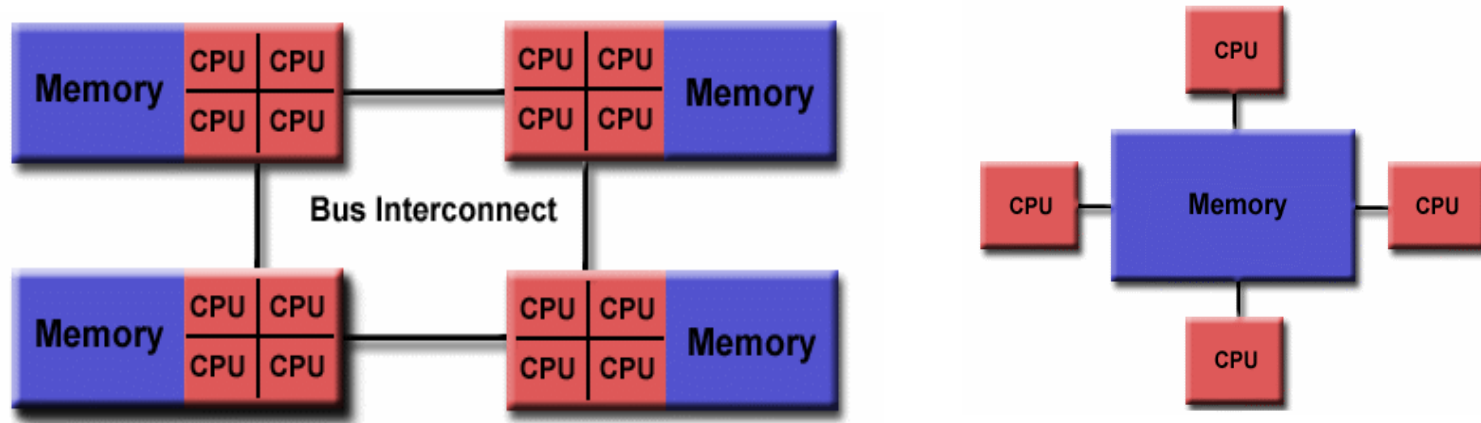
- C1: *Thou shalt not write thy neighbor's memory randomly* – chunk the data, redistribute, and then sort/work on your data locally.
- C2: *Thou shalt read thy neighbor's memory only sequentially* – let the prefetcher hide the remote access latency.
- C3: *Thou shalt not wait for thy neighbors* – don't use fine grained latching or locking and avoid synchronization points of parallel threads.

The openMP framework

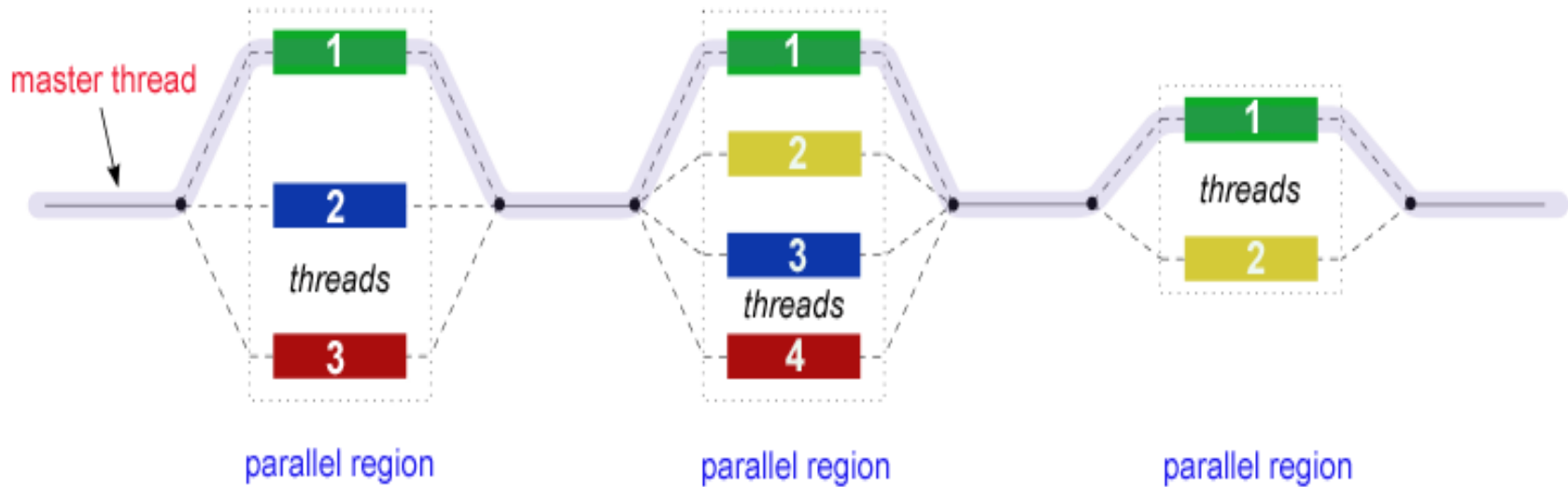
- API for multiprocessing
- Easily applied to parallelize code
- Built for shared memory processors
- Works cross platform
- <http://openmp.org>

Shared memory processors

- Recall the UMA and NUMA architectures
- Both are shared memory processor architectures



General control flow



Compiling openMP

- `#include <omp.h>`
- Compile with the `openmp` flag
 - `Gcc -fopenmp test.cpp`
- Environment variables
 - `setenv OMP_NUM_THREADS 12`
 - `export OMP_NUM_THREADS=12`

Useful functions

- Threadid
 - `omp_get_thread_num();`
- Amount of threads
 - `omp_get_num_threads();`
- Set amount of active threads
 - `omp_set_num_threads(4);`
 - `export OMP_NUM_THREADS=12`

Directives

- Used to communicate with the compiler
- `#pragma` directives used to instruct the compiler to use pragmatic or implementation-dependent features
- One such feature is openMP
- `#pragma omp parallel`

Examples



Problems with NUMA

- We do not know where the data is allocated
- We do not know on which NUMA node the thread is running
- So, no openMP on really parallel machines?

New libraries to the rescue

- We can pin threads to processors
- We can control memory allocations
- Tools
 - Numactl
 - libnuma

libnuma

- Provides c++ header files
- Can be used to create numa awareness in the code
- A bit like openMP but instead provides methods for getting numa node and allocating memory on specific numa nodes

Numactl --hardware

- Shows the current hardware
- Examples

Numactl (continued)

Socket affinity	-N --cpunodebind=	{0,1}	Execute process on cores of these sockets only
Memory policy	-l --localalloc	No argument	Allocate on current socket; fallback to any other if full
Memory policy	-i --interleave=	{0,1}	Allocate round robin (interleave) on these sockets. No fallback
Memory policy	--preferred=	{0,1} select one	Allocate on this socket; fallback to any other if full.
Memory policy	-m --membind=	{0,1}	Allocate only on this (these) socket(s). No fallback.
Core affinity	-C --physcpubind=	{1,2,3,4,5,6 ,7,8,9,10,11,12}	Execute process on this (these) core(s) only



Questions?



AU

AARHUS UNIVERSITET