Parallel Algorithm Engineering

Manuel R. Ciosici
PhD Fellow

based on slides by Kenneth S. Bøgh and Darius Sidlauskaš

Outline

- Background
- Current multicore architectures
- The OpenMP framework
- UMA vs. NUMA and NUMA control
- Examples

Software crisis

“the major cause 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 had become an equally gigantic problem.”

–Edsger W. Dijkstra, ACM Turing Lecture 1972

https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

A long time ago…

A long time ago…

The 1st Software Crisis

- **When**: around 60s and 70s
- **Problem**: large programs written in assembly
- **Solution**: abstraction and portability via high-level languages like C and FORTRAN

A long time ago…

The 1st Software Crisis

- **When**: around 60s and 70s
- **Problem**: large programs written in assembly
- **Solution**: abstraction and portability via high-level languages like C and FORTRAN

The 2nd Software Crisis

- **When**: around 80s and 90s
- **Problem**: building and maintaining large programs written by hundreds of programmers
- **Solution**: software as a process (OOP, testing, code reviews, design patterns), better tools (IDEs, version control, component libraries, etc.)

Recently…

Processor-oblivious programmers:

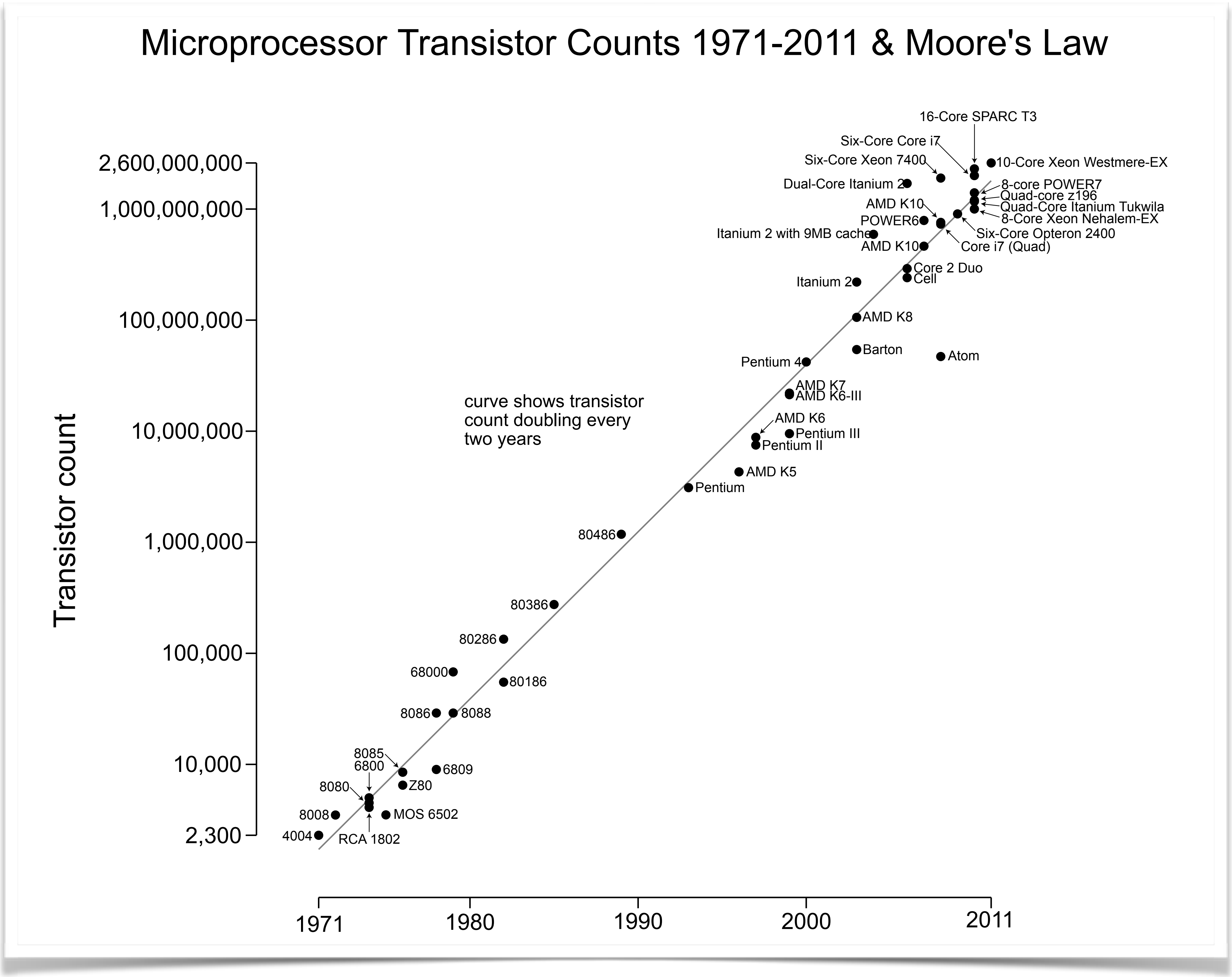
- A Java program written on PC works on your phone
- A C program written in 70s still works today and is faster
- Moore’s law takes care of good speedups

Currently…

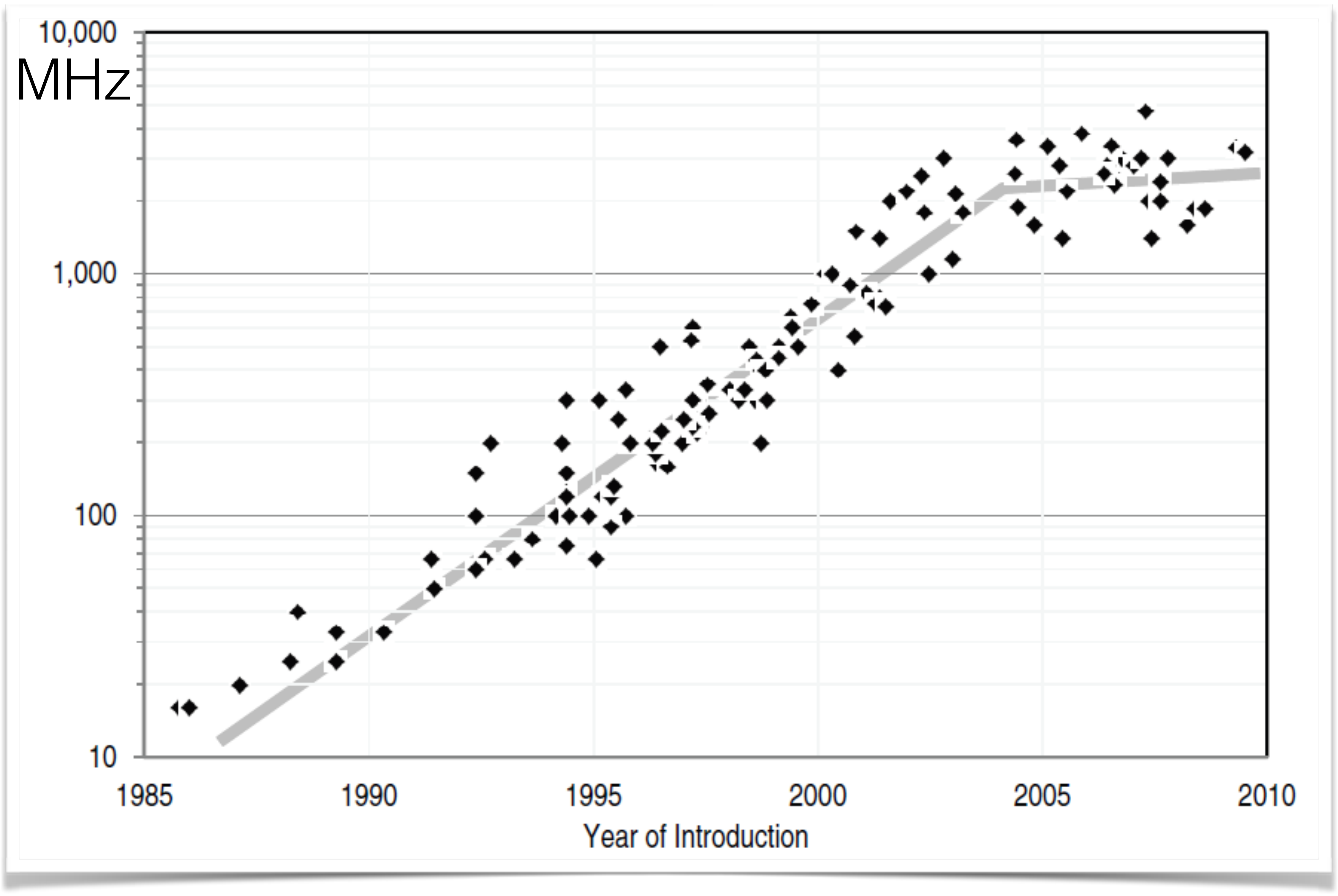
Software crisis (again?)

- **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
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
curve shows transistor count doubling every two years
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
AMD K10
Quad-Core Itanium Tukwila
POWER6
8-Core Xeon Nehalem-EX
Itanium 2 with 9MB cache
Six-Core Opteron 2400
AMD K10
Core i7 (Quad)
Core 2 Duo
Cell
Itanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
Z80
8080
8008
MOS 6502
4004
RCA 1802

Uniprocessor performance



SPECint2000

Overclocking

is not a solution

Overclocking
is not a solution

- Air-water: ~5.0 GHz (possible at home)

Overclocking

is not a solution

- Air-water: ~5.0 GHz (possible at home)
- Phase change: ~6.0 GHz

Overclocking
is not a solution

- 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

Let’s parallelise!

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 general 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)

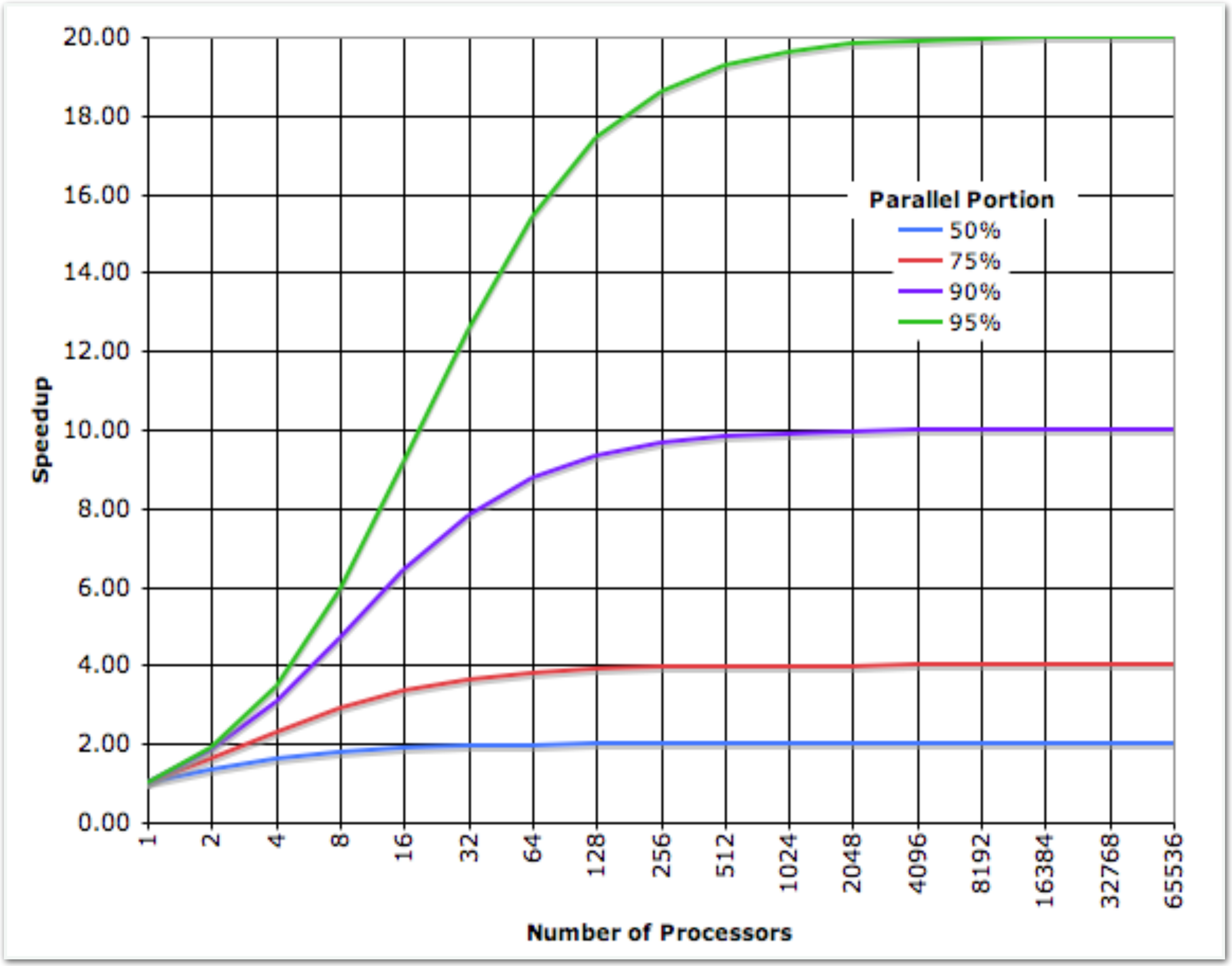
Amdahl’s Law

- Potential program speedup is defined by the fraction of code that can be parallelised
- Serial components rapidly become performance limiters as thread count increases

$$=\frac{1}{\underbrace{(1-p)}_{\text{fraction of time to complete sequential work}}+\underbrace{\frac{p}{n}}_{\text{fraction of time to complete parallel work}}}$$

  - p – fraction of work that can parallelised
  - n – number of processors

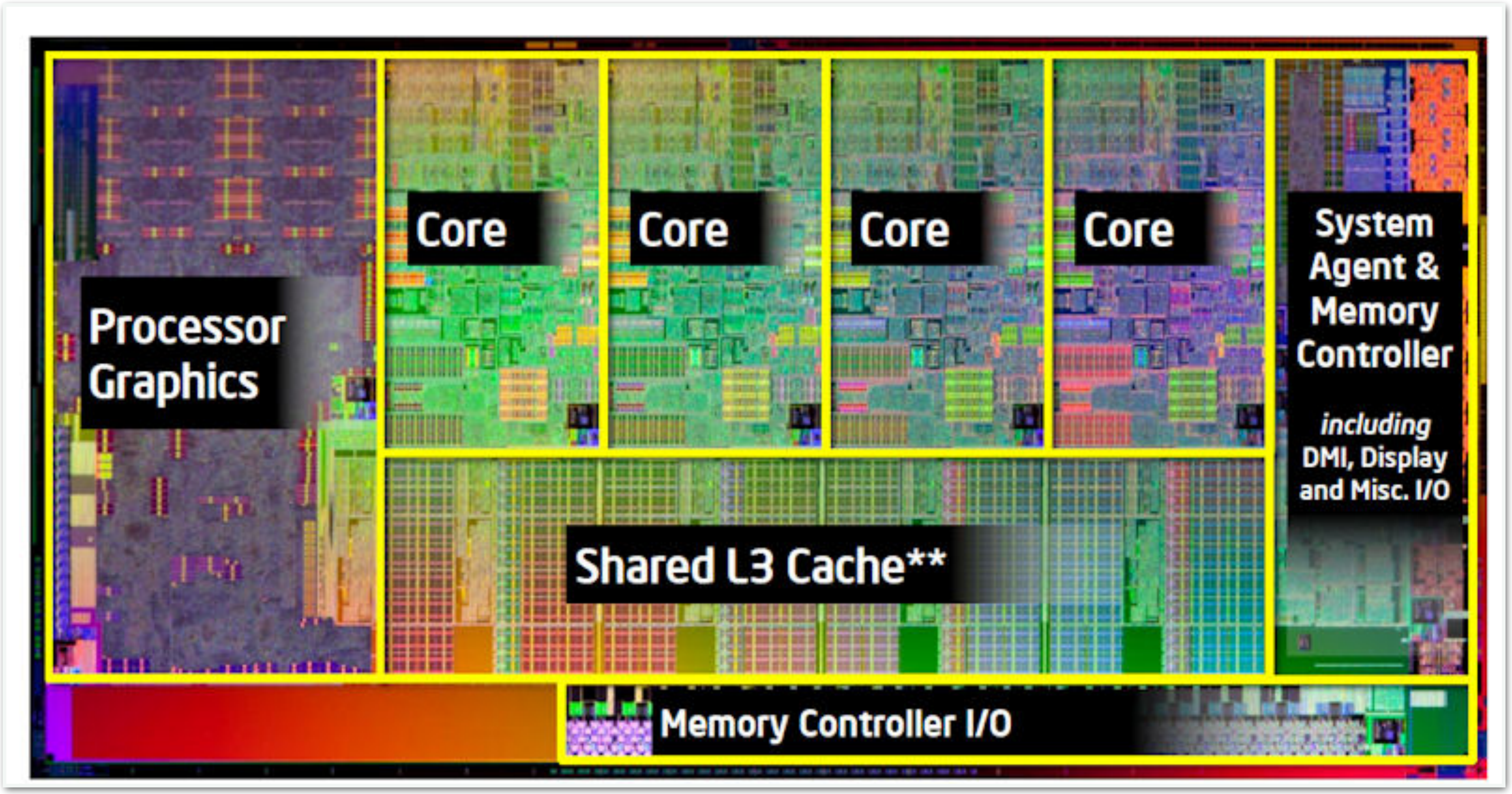
Amdahl's Law



Parallel Portion
50%
75%
90%
95%
Speedup
Number of Processors
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Towards parallel setups

Let's use transistors for multiple cores



Intel® Core™ i7-2600K Processor

Current commercial multi-core CPUs

Intel

- Intel® Core™ i7-6950X Processor Extreme Edition 10 cores (20 hw threads), 25 MB cache, max 3.5 GHz
- Intel® Xeon® Processor E7-8894 v4 24 cores (48 hw threads), 60 MB cache, max 3.4 GHz
- Intel® Xeon Phi™ Processor 7210 64 cores (256 hw threads), 32 MB Cache, max 1.5 GHz

AMD (may be out of date)

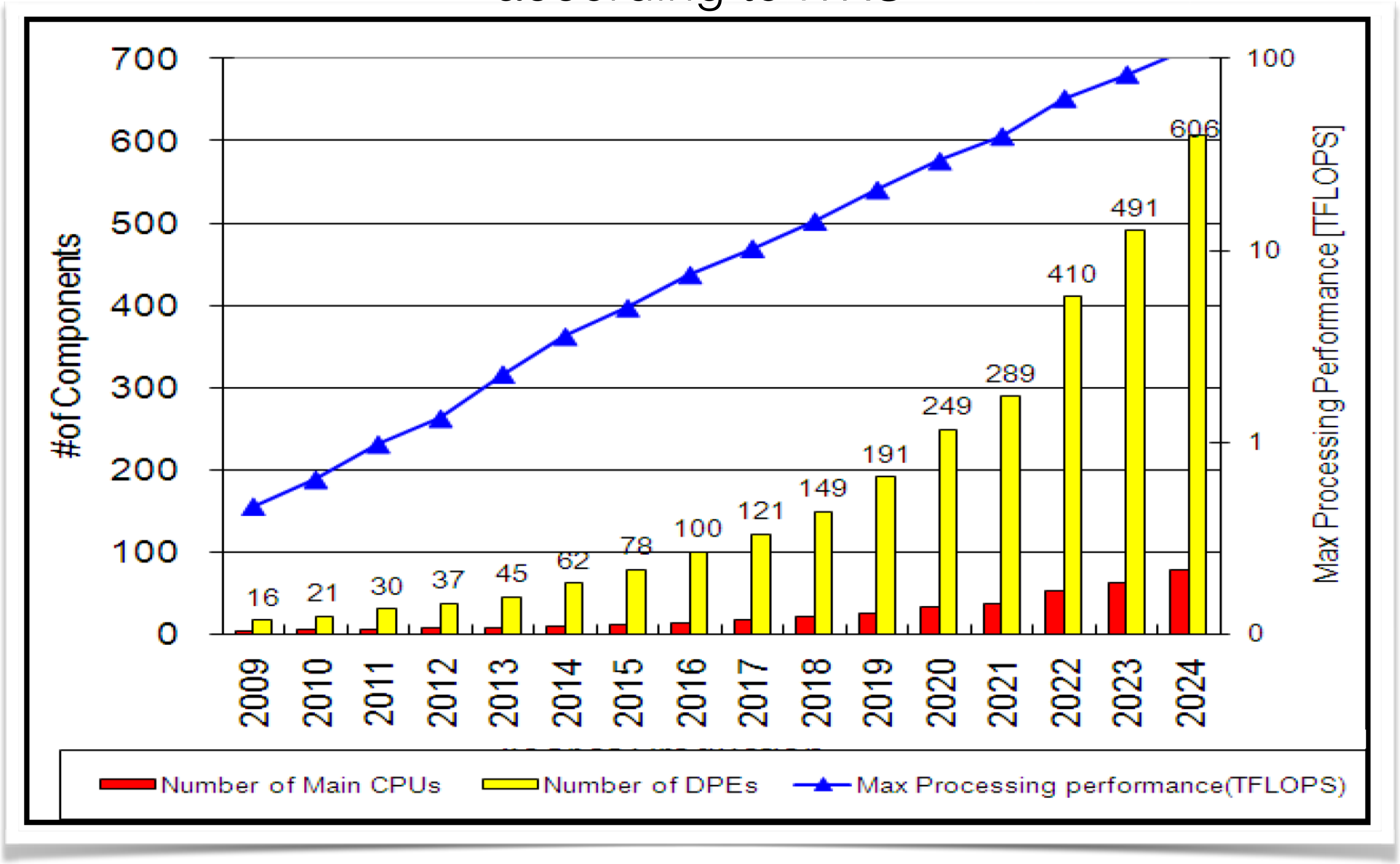
- FX-9590: 8 cores, 8 MB Cache, 4.7 GHz
- A10-7850K: 12 cores (4 CPU 4 GHz + 8 GPU 0.72 GHz), 4 MB Cache
- Opteron 6386 SE: 16 cores, 16 MB Cache, 3.5 GHz (x 4-socket conf.)

Oracle

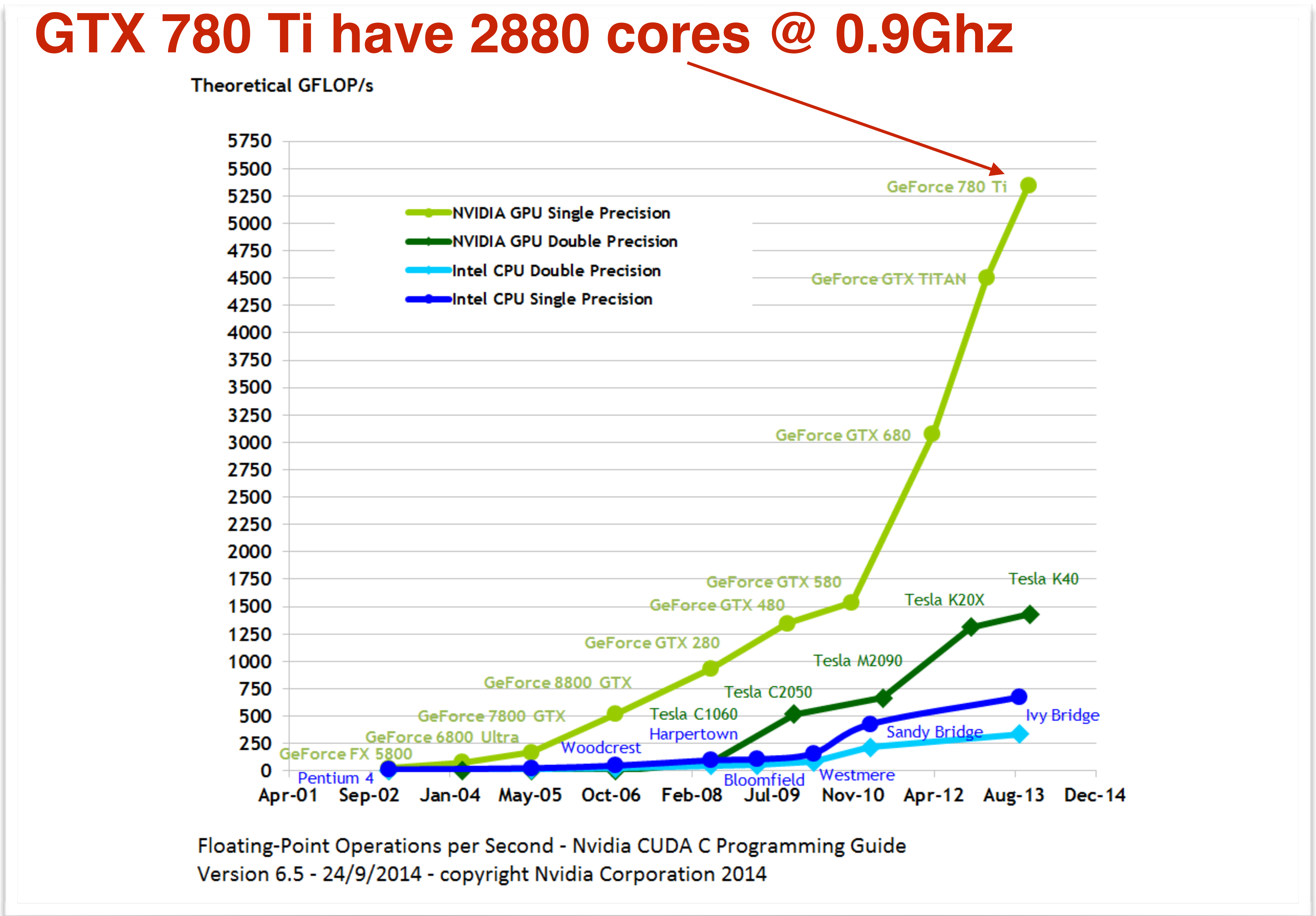
- SPARC M7: 32 cores (hw 256 threads), 64 MB Cache, 4.13 GHz

Parallel processing

Predicted # of cores for stationary systems, according to ITRS

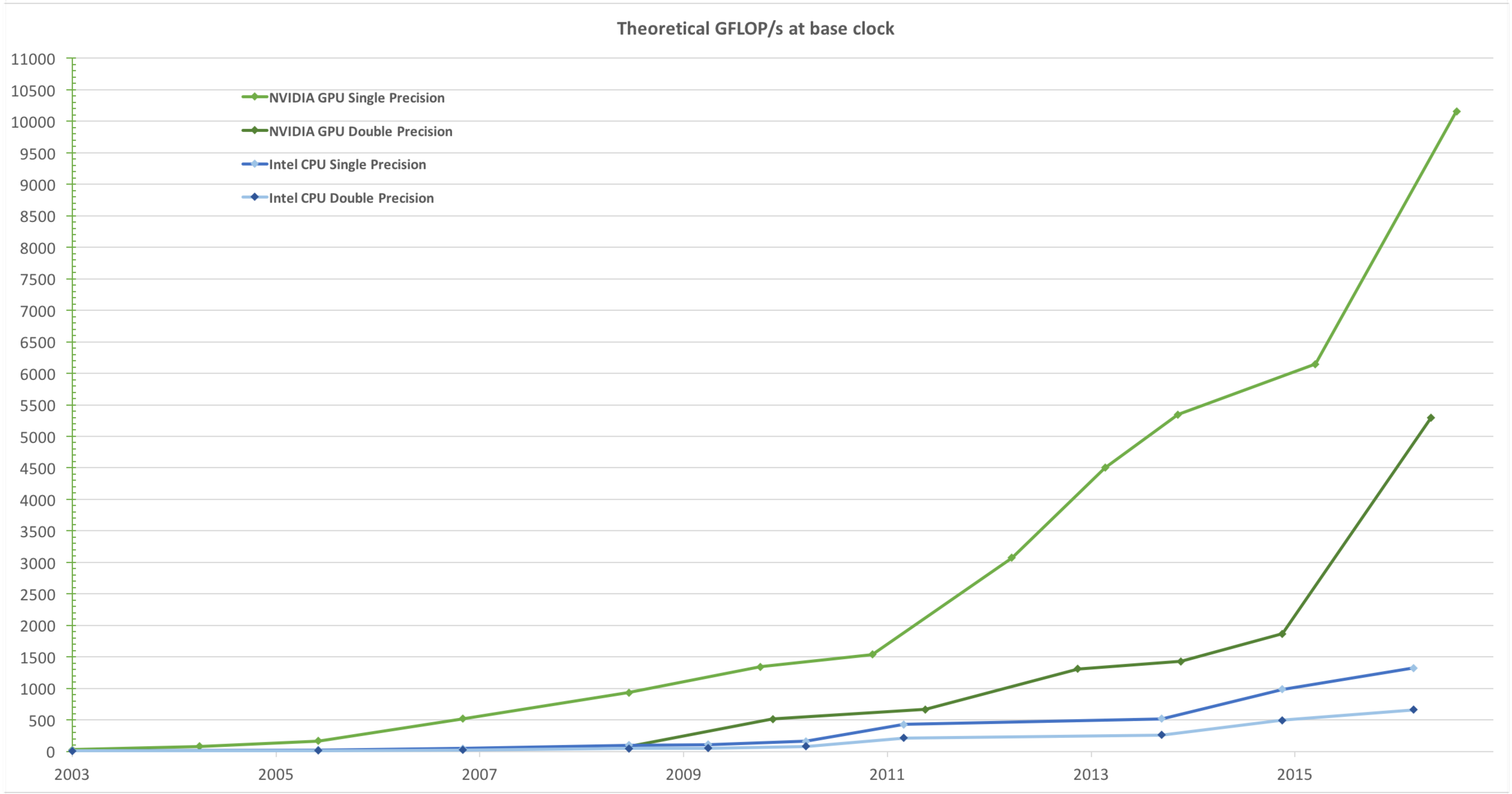


Even "worse" for GPUs



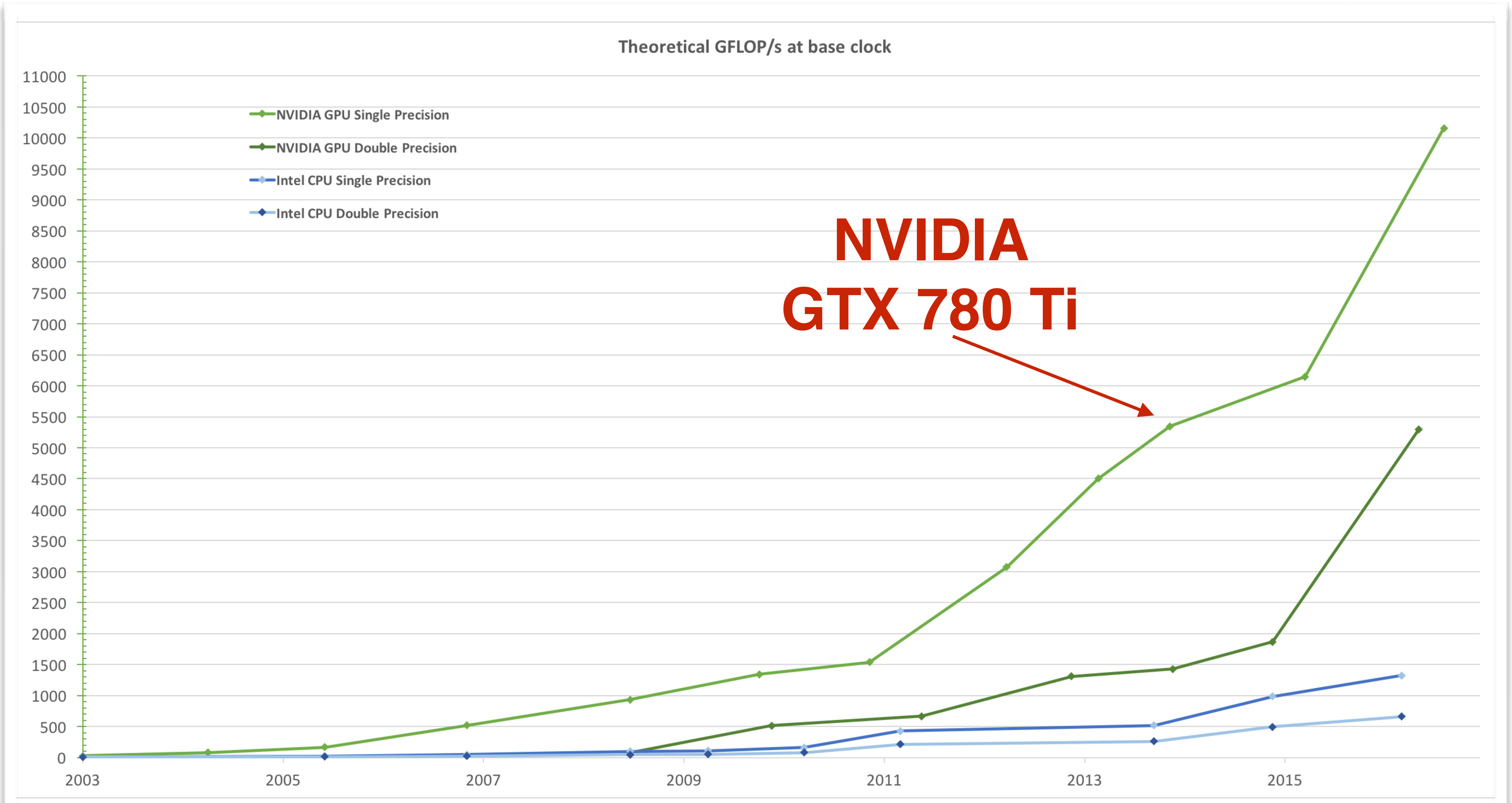
Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

Even "worse" for GPUs



Floating point operations per second – NVIDIA C Programming Guide version 8 – 27 Feb 2017
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-computing

Even “worse” for GPUs



Floating point operations per second – NVIDIA C Programming Guide version 8 – 27 Feb 2017
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-computing

Why

Power considerations

- Consumption, Cooling, Efficiency

DRAM access latency

- Memory wall

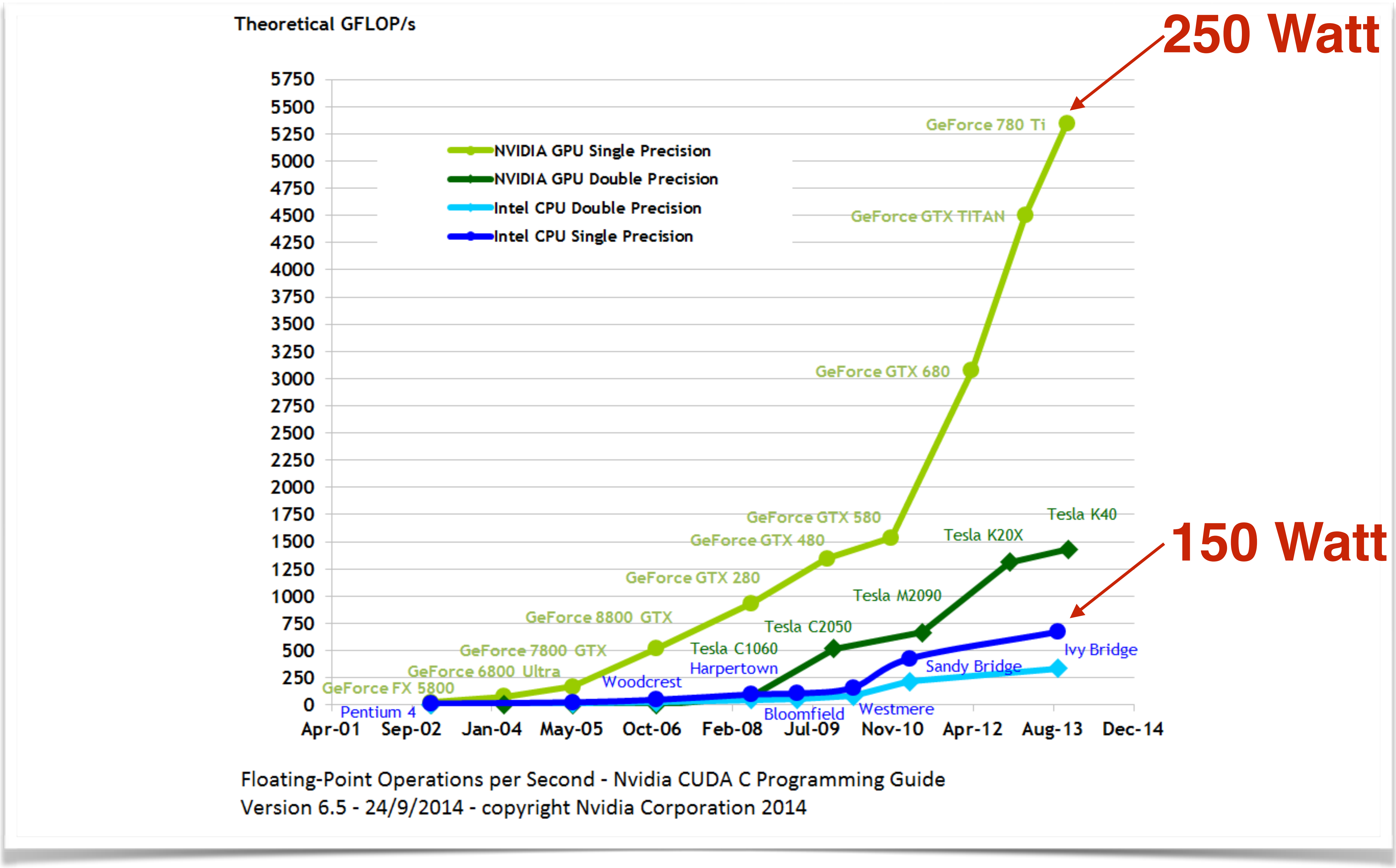
Wire delays

- Range of wire in one clock cycle

Diminishing returns of more instruction-level parallelism

- Out-of-order execution, branch prediction, etc.

Power consumption



Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

Single Instruction Multiple Data (SIMD)

- One SIMD processing unit per core
- Modern compilers automatically use SIMD in simpler cases (remember the -mavx compiler parameter)
- How to use:
  - compiler intrinsics – see https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html
  - libraries – see one example at www.agner.org/optimize/#vectorclass

Kinds of parallelism

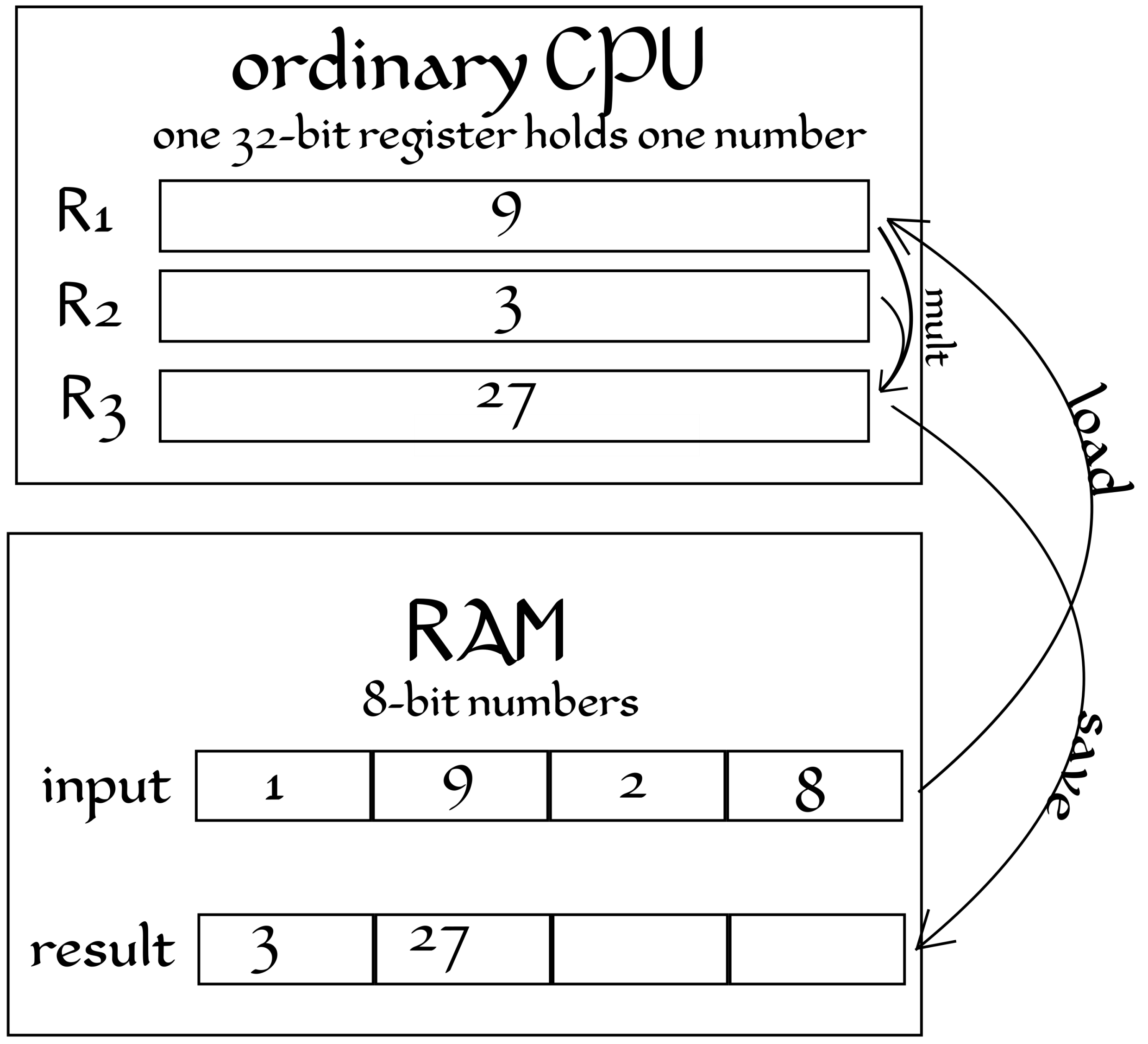
- Single Instruction Multiple Data (SIMD) – briefly
- Single Instruction Multiple Threads (SIMT) – GPUs - not covered
- Task Parallelism

Single Instruction Multiple Data (SIMD)

- Exploits data level parallelism
- Initially introduced in desktop CPUs in order to speed up media applications
- Available in most desktop CPUs since early late 90s: MMX (64-bit), SSE (128-bit), AVX (256-bit and 512-bit)
- Available in mobile SoCs for a few years now: NEON instructions (128-bit)

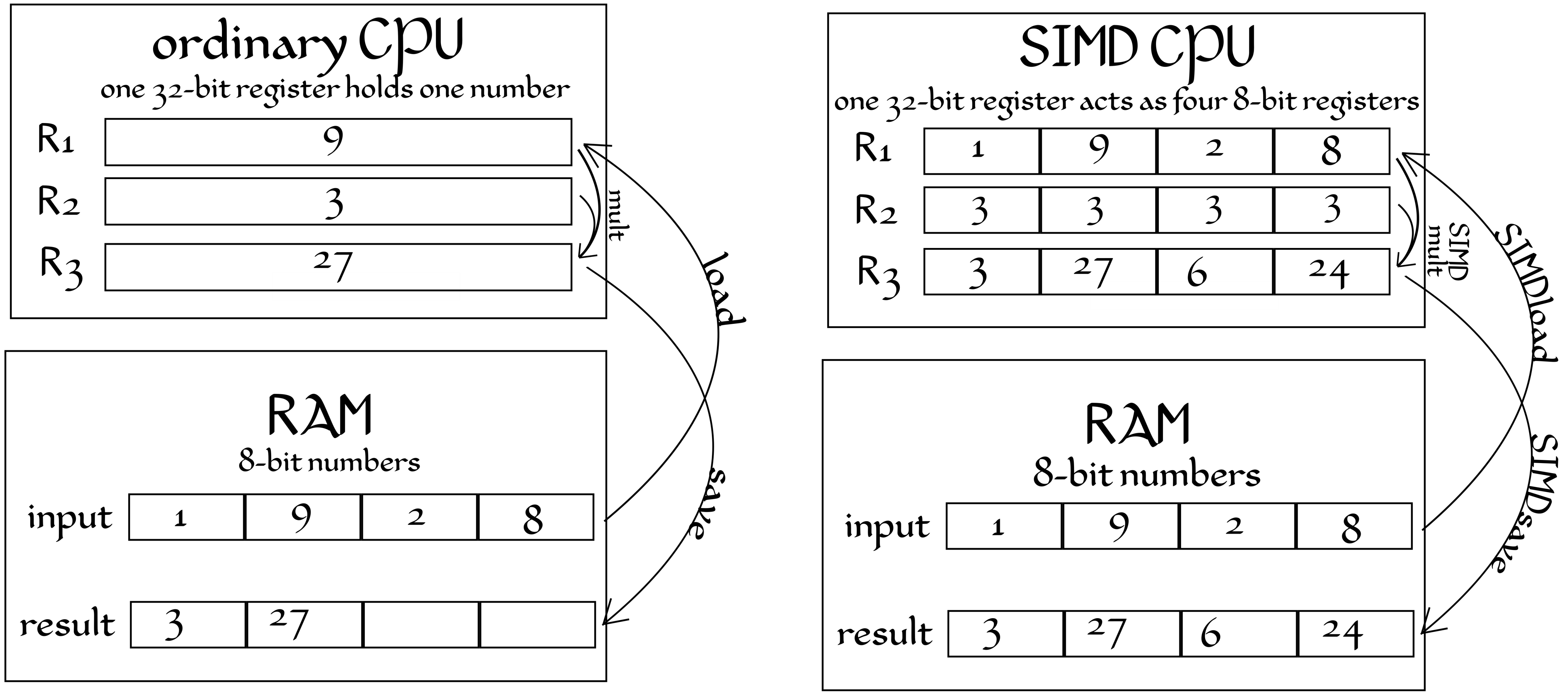
SIMD

SIMD



Operation Count:
4 loads, 4 multiplies, and 4 saves

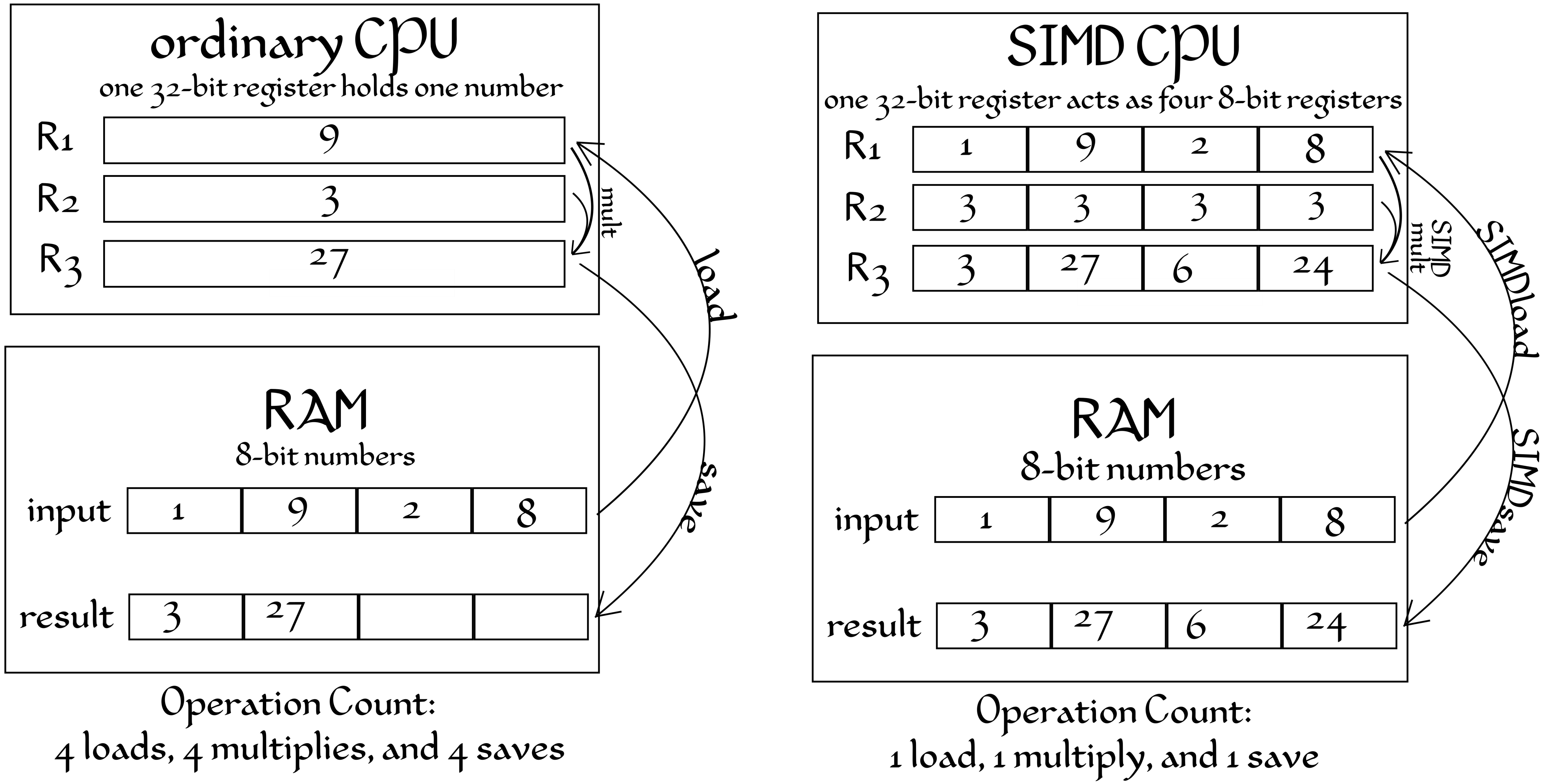
SIMD



Operation Count:
4 loads, 4 multiplies, and 4 saves

Operation Count:
1 load, 1 multiply, and 1 save

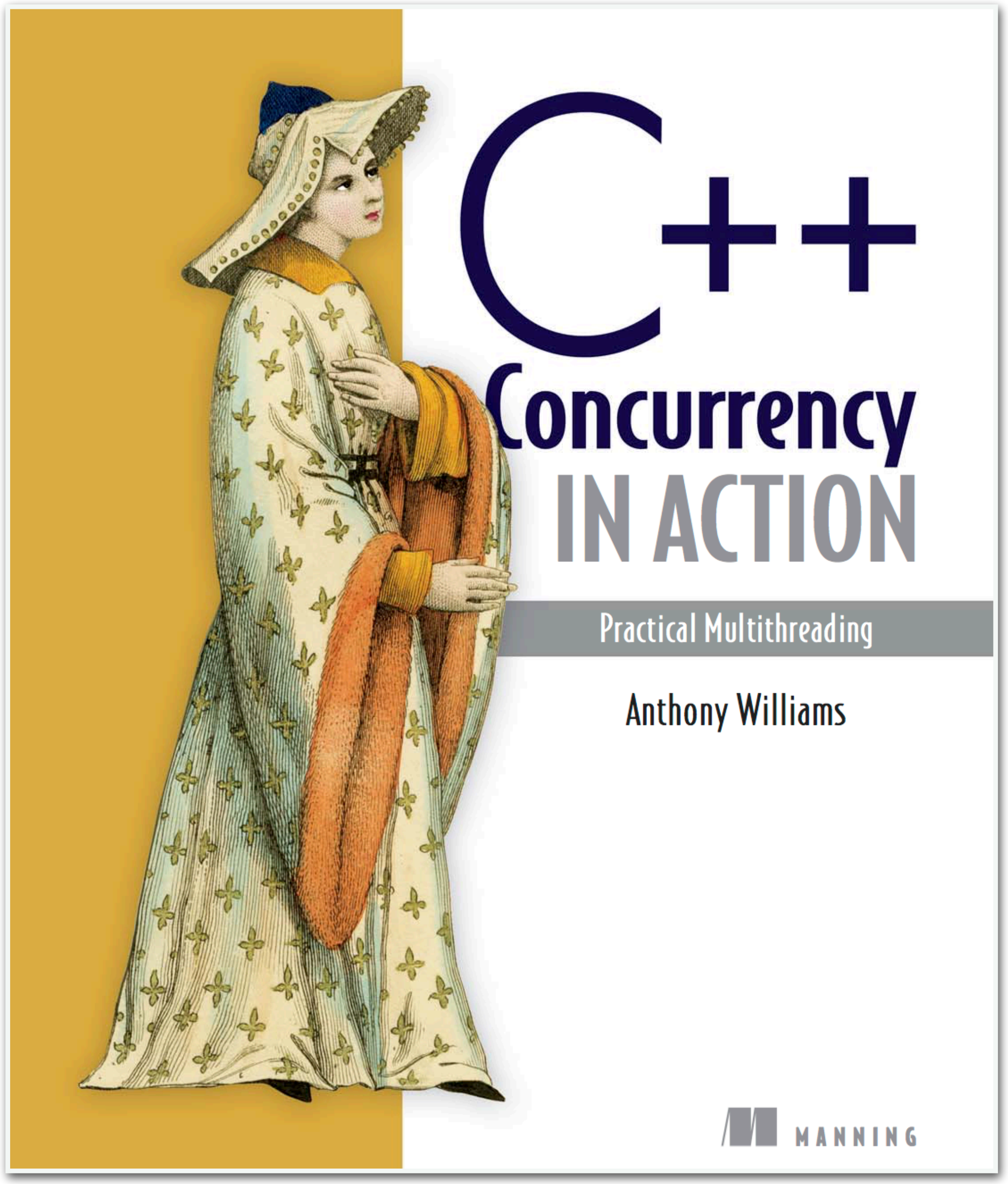
SIMD



Speedup: 4x

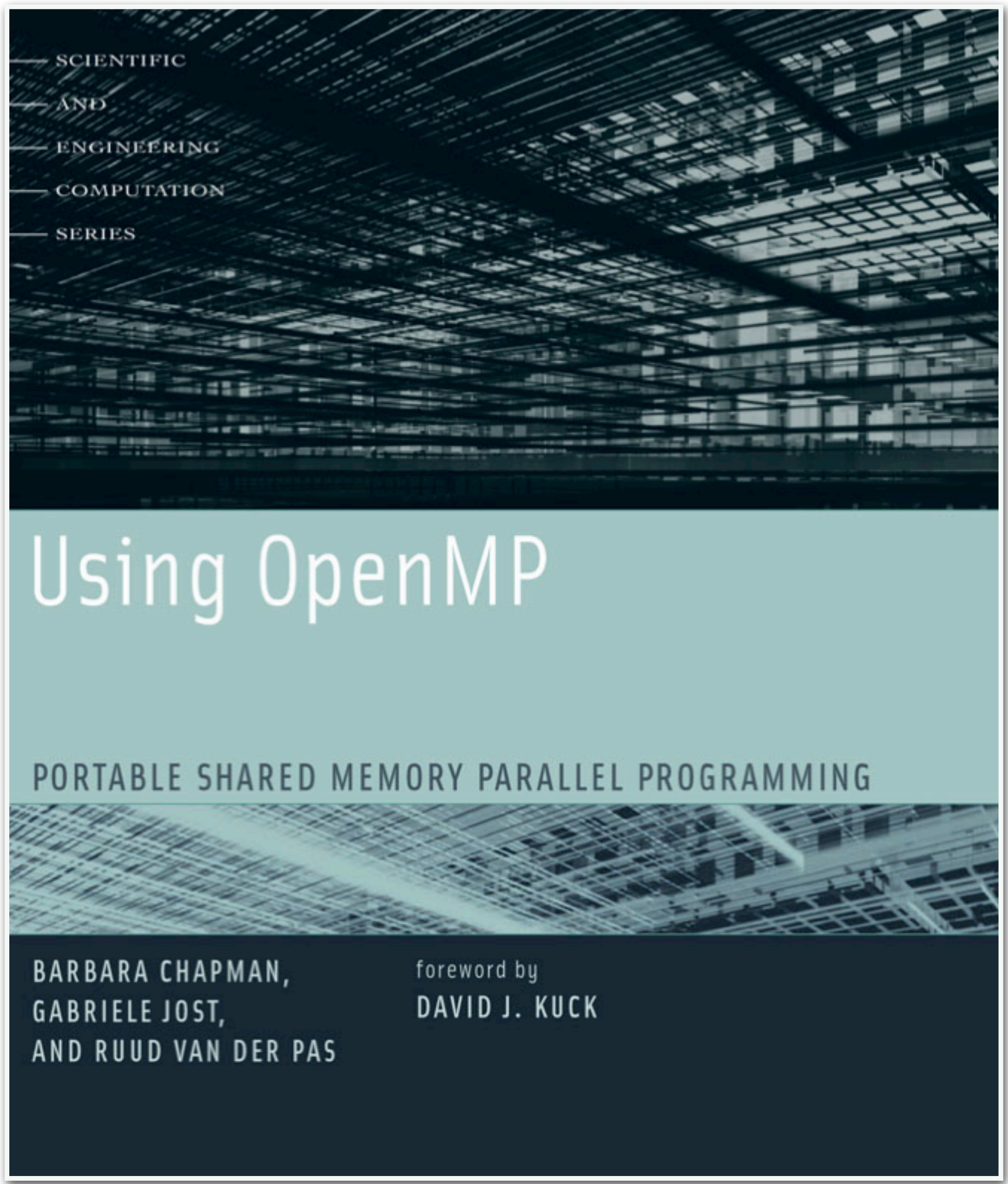
Task parallelism

- Multiple threads are executed in parallel, performing multiple tasks
- C++11 brings a unified memory model and native thread support (read cross platform)
- See C++ Concurrency in Action

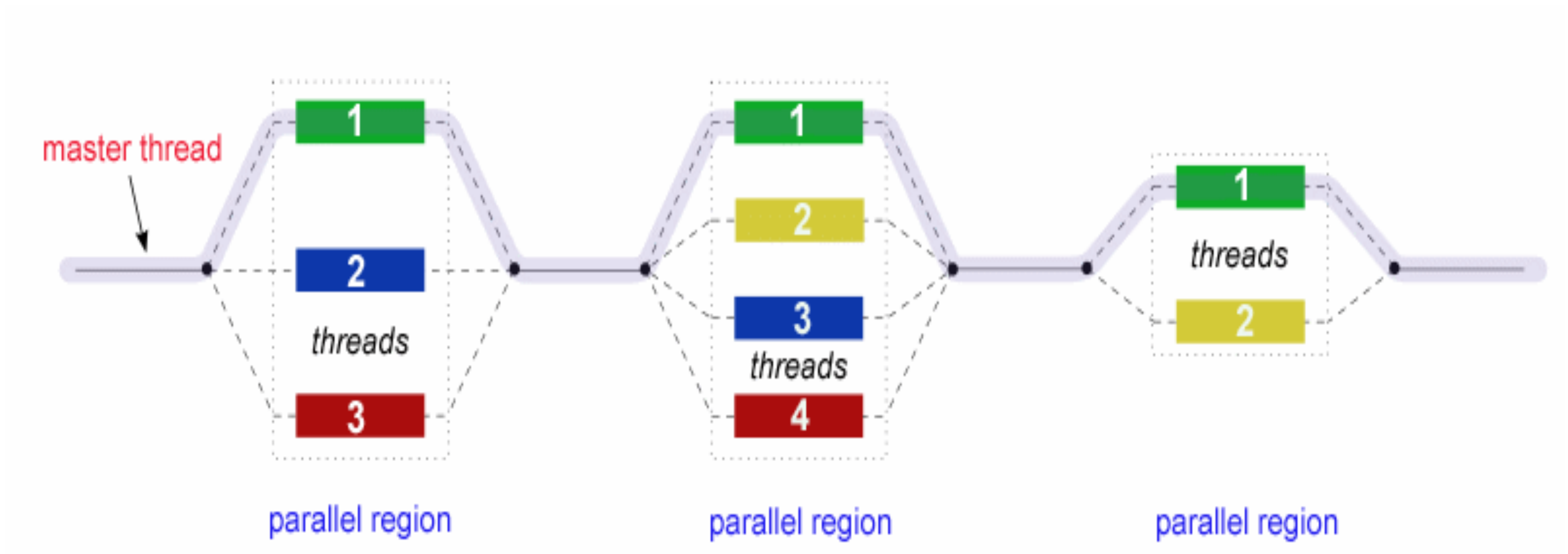


The OpenMP Framework

- API for multiprocessing
- Easily applied to parallelise code
- Built for shared memory processors
- Works cross platform
- See the specifications and official examples at www.openmp.org/specifications/
- Using OpenMP – older book, but great learning resource



General flow control



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

Useful functions

- Thread-ID: 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

Compiling OpenMP

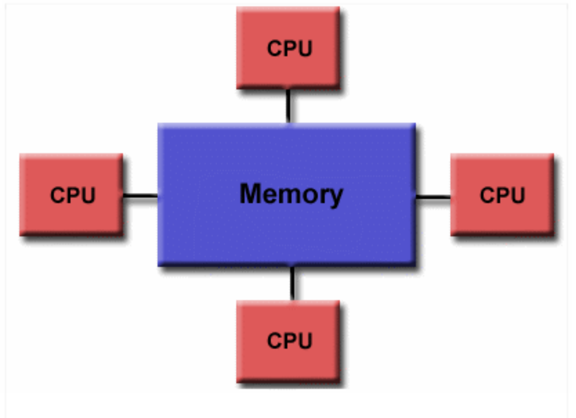
- #include <omp.h>
- Compile with the OpenMP flag
    - g++ -fopenmp test.cpp
- Environment variables
    - setenv OMP_NUM_THREADS 12
    - export OMP_NUM_THREADS=12

When to parallelise

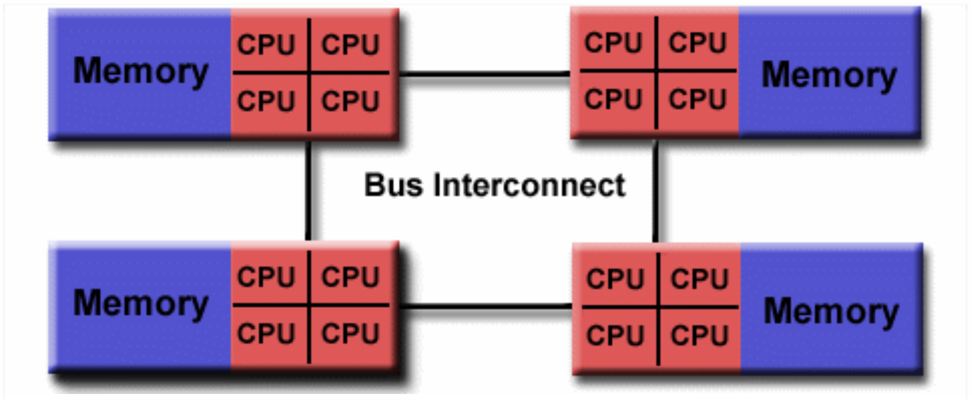
- When you have independent units of work
- When your code is compute bound
- Or your code is not utilising the memory bandwidth
- When you see performance gains in tests :-)

UMA vs NUMA

- All laptops and most desktops are UMA (Uniform Memory Access) – single CPU
- Most modern servers are NUMA (Non Uniform Memory Access) – multiple CPUs
- Important to know which you target!

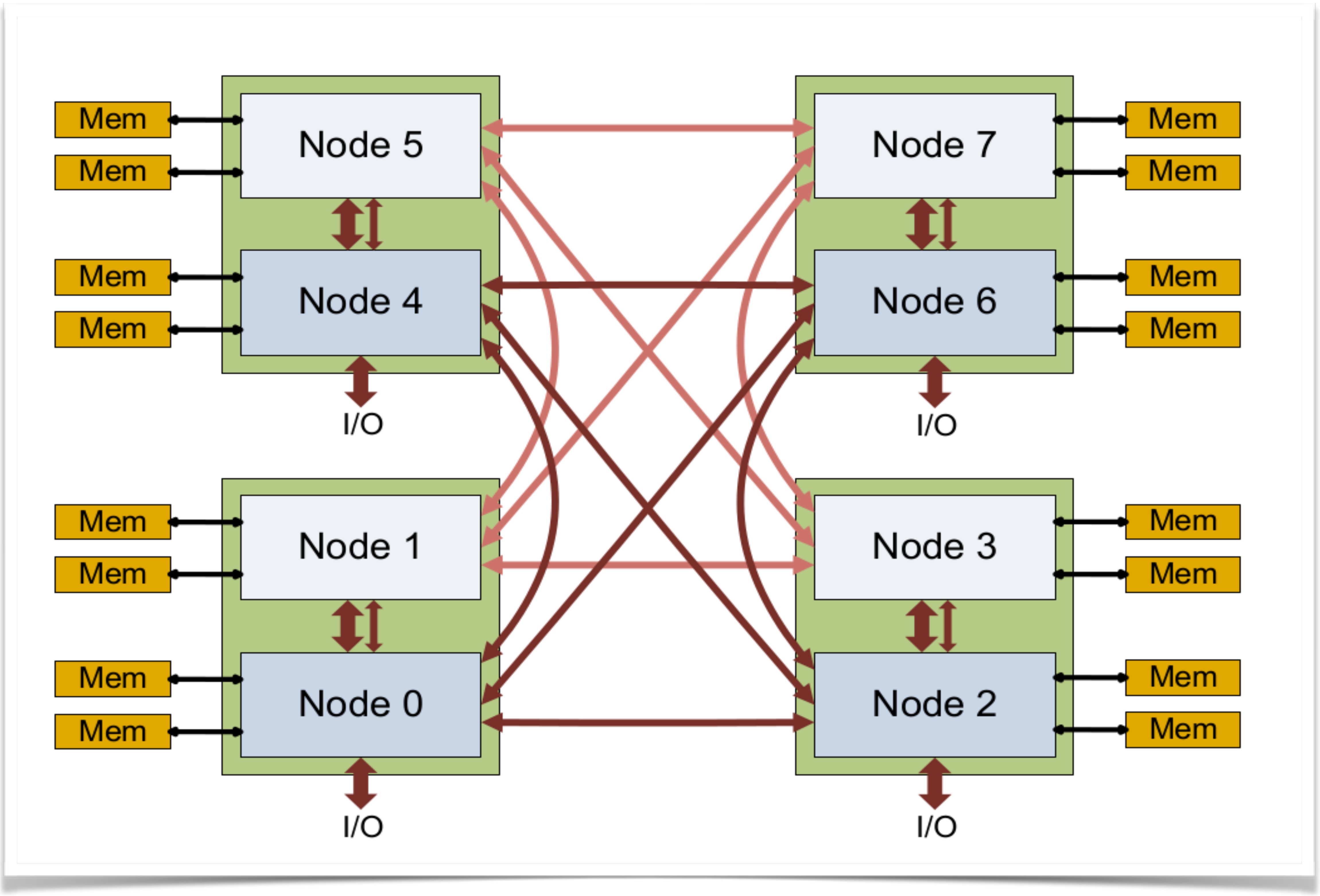


UMA

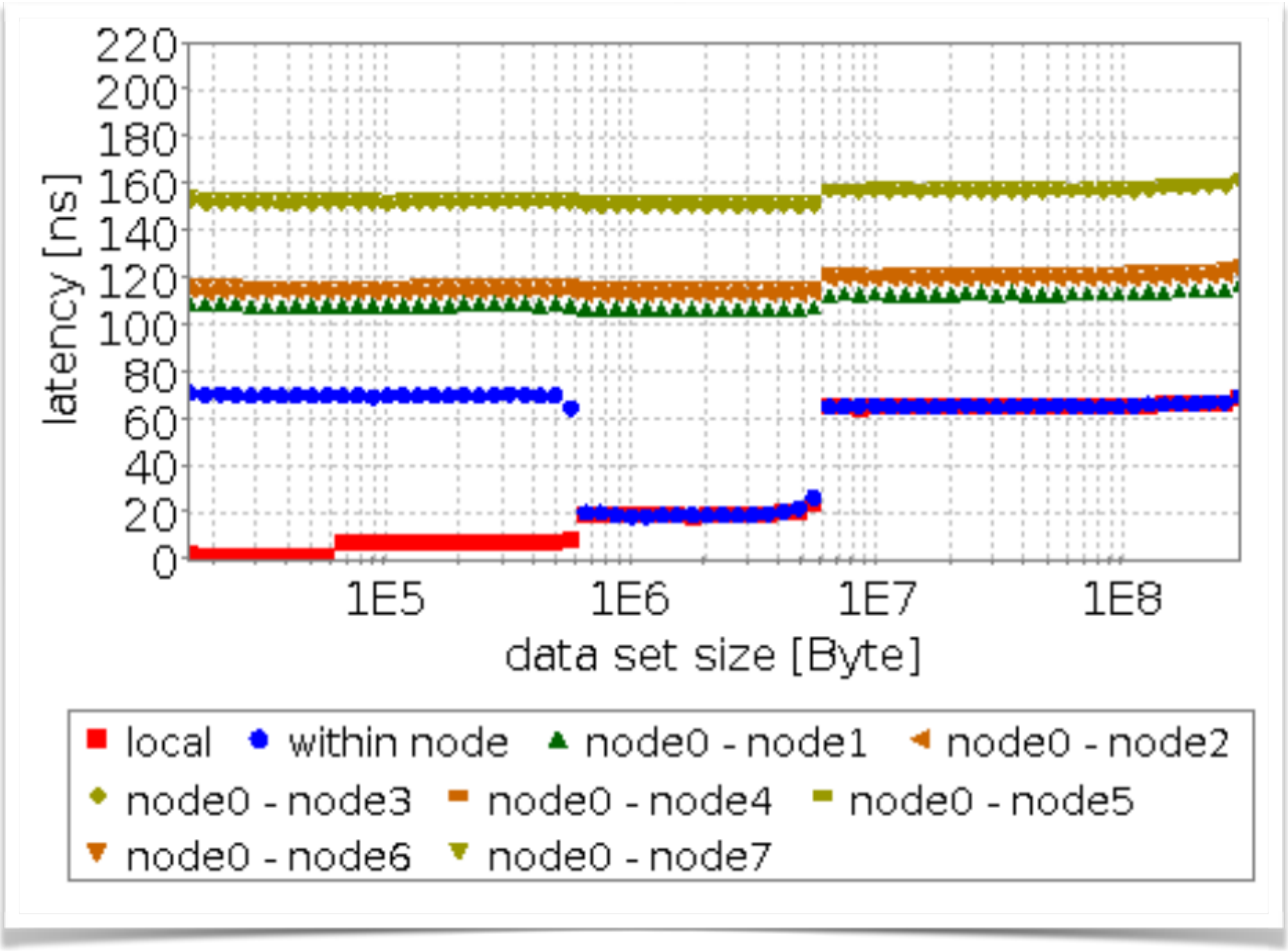


NUMA

4 sockets – 8 CPU setup

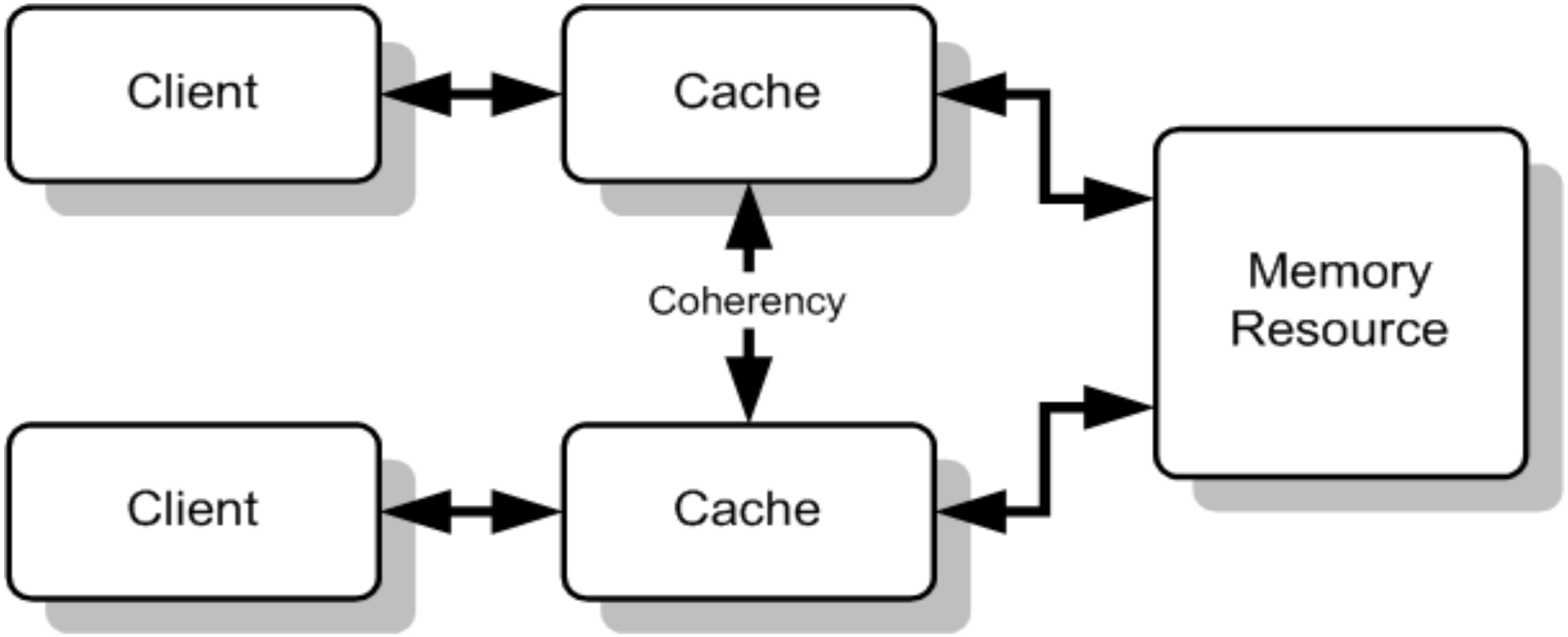


NUMA effects



Cache coherence

Ensures consistency between all the caches.

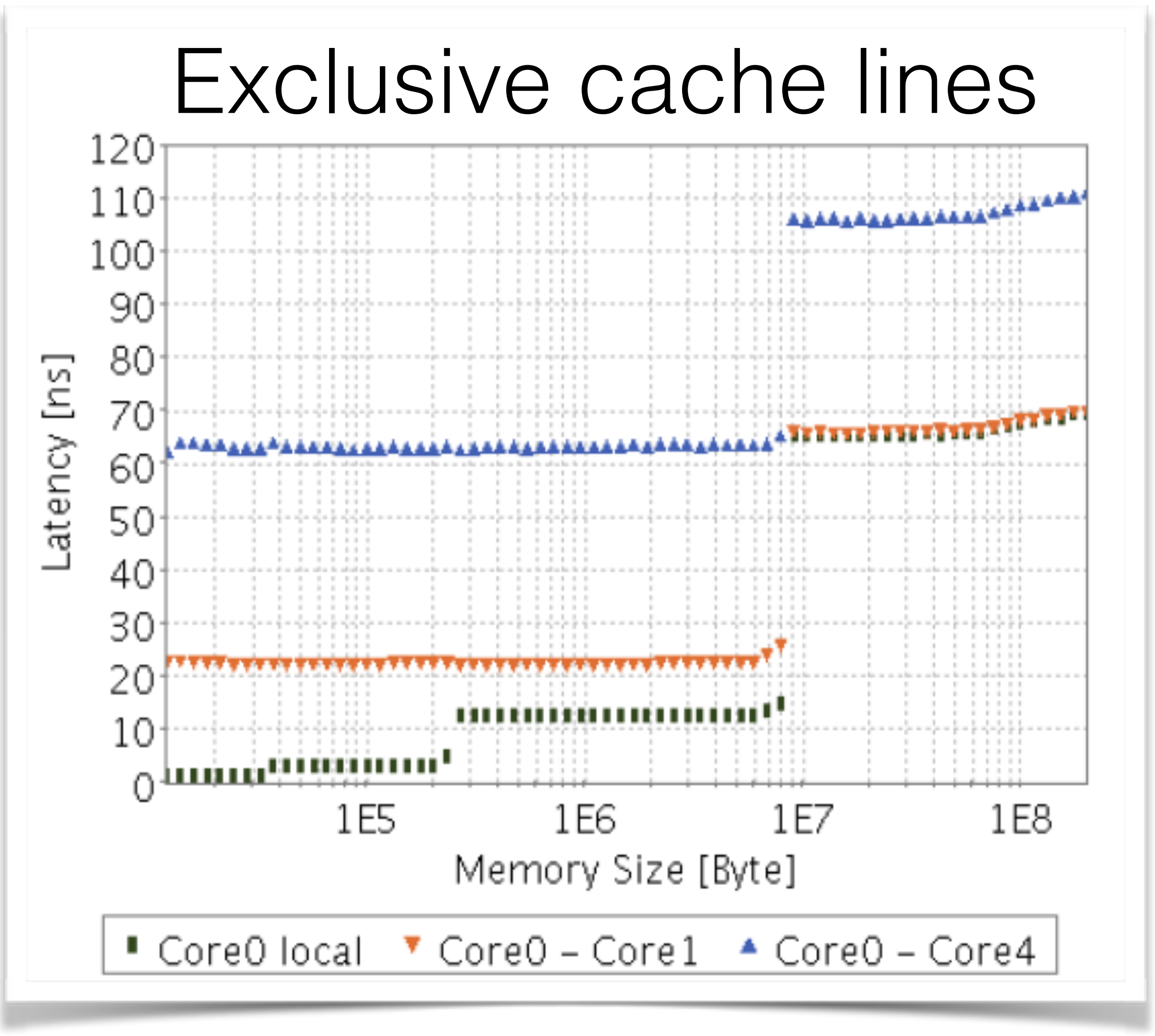


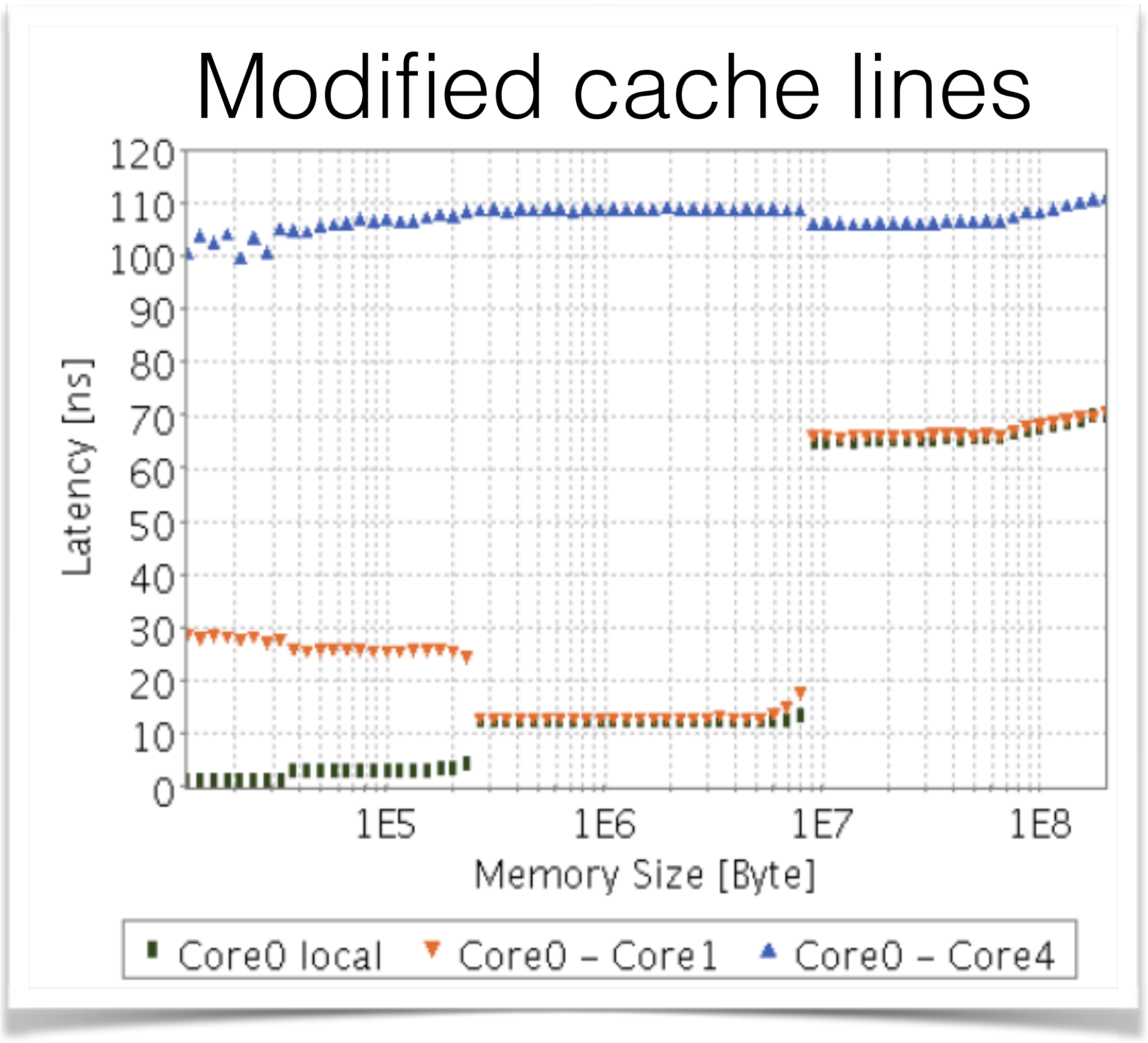
MESIF protocol

- 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): may be stored in other caches and clean. May be changed to (I) at any time.
- Invalid (I): unusable
- Forward (F): a specialised form of the S state

For more on MESI and MESIF see https://www.youtube.com/watch?v=S3kg_zCz_PA
and http://www.realworldtech.com/common-system-interface/5/

Cache coherence effects





Latency in nsec on 2-socket Intel Nehalem (4 cores)

Commandments

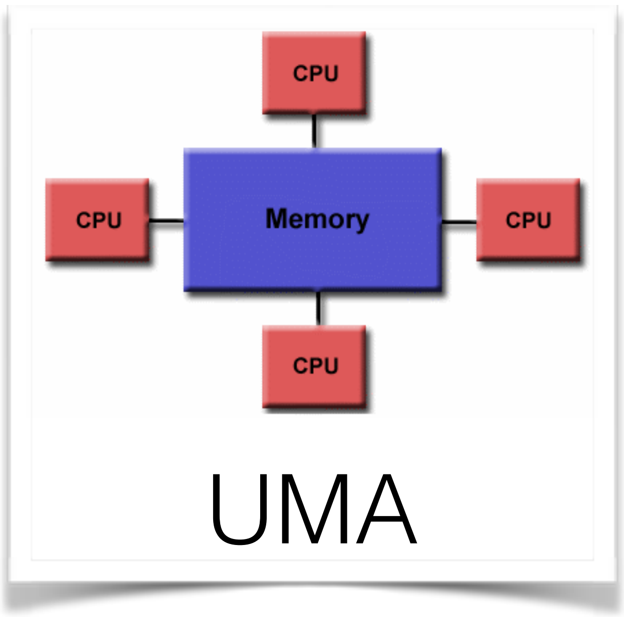
1. Thou shalt not write thy neighbour’s memory randomly – chunk the data, redistribute, and then sort/work on your data locally.

2. Thou shalt read thy neighbour’s memory only sequentially – let the prefetcher hide the remote access latency.

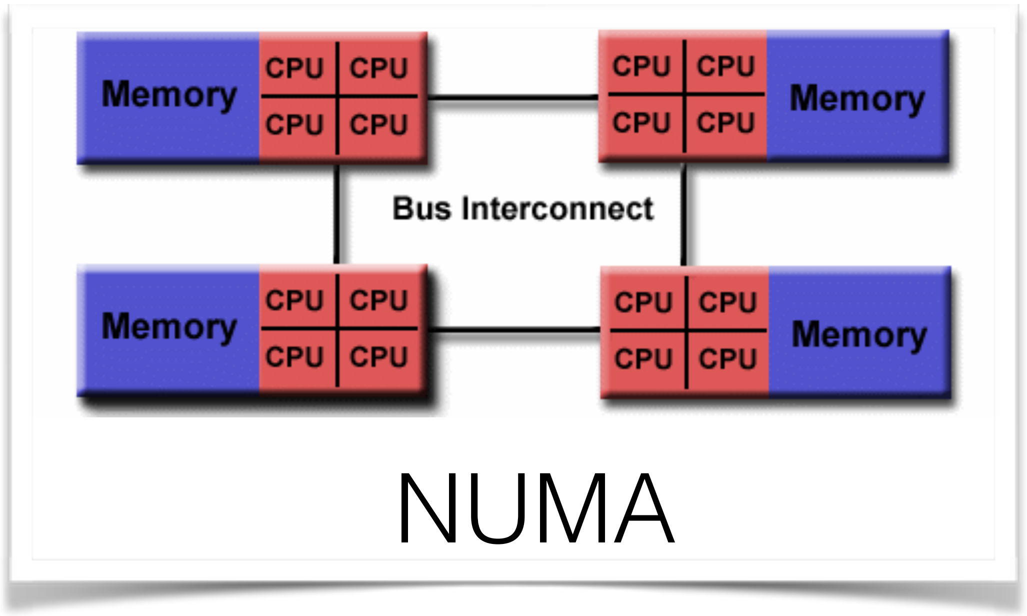
3. Thou shalt not wait for thy neighbours – don’t use fine grained latching or locking and avoid synchronisation points of parallel threads.

Shared memory processors

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



UMA



NUMA

Problems with NUMA

Problems with NUMA

- We do not know where the data is allocated

Problems with NUMA

- We do not know where the data is allocated
- We do not know on which NUMA node the thread is running

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

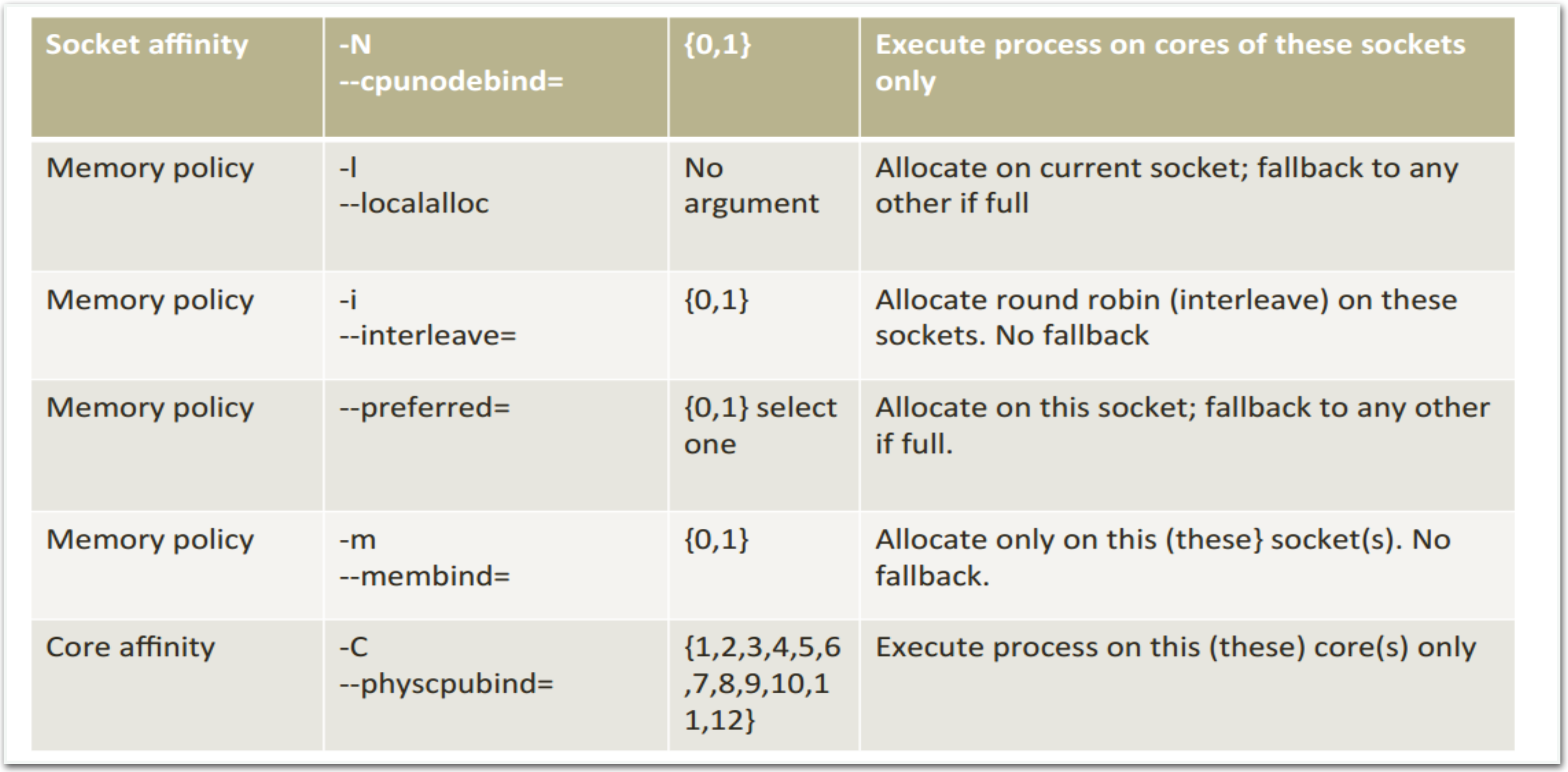
libnuma

- 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

- Like libnuma, but controlled from the shell
- Can be used to control existing software without changing the code
- Very useful when running experiments

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

Extra

- Sometimes getting PAPI to work is difficult
- You can find a nice PAPI wrapper at https://github.com/sean-chester/papi-wrapper

Examples

Questions?