Durham University
Department of Computer Science

Core I—Introduction to High Performance Computing (HPC)

Session I: Machine models and upscaling laws

Dr. Weinzierl

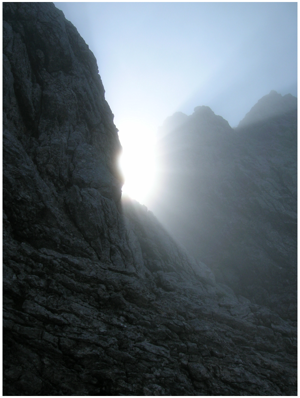
Michaelmas term 2019

Disclaimer

Durham University
Department of Computer Science

You need C.

Outline



Moore's law

Durham University
Department of Computer Science

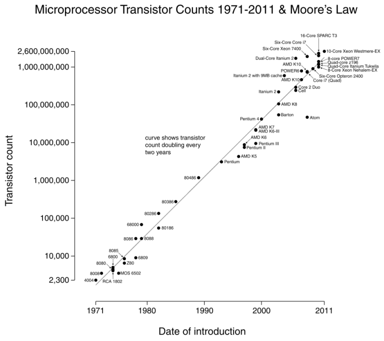
Gordon Moore, 2004

Moore's Law: The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.

- G. Moore: Co-founder of Intel (R&D director of Fairchild Semiconductor)
- G. Moore: *Cramming more components onto integrated circuits*. Electronics Magazine (1965)
- Cf. Intel's tick-tock policy: change micro architecture vs. die shrink
- Carver Mead (CalTech) coined the term *Moore's law* (1975)

⇒ It is about chip complexity, not about speed

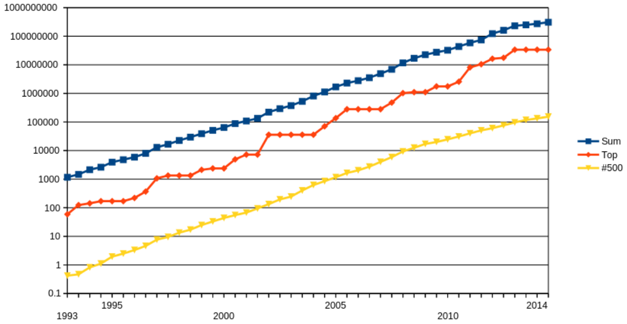
Wikipedia validates Moore's law



Source: Wikipedia

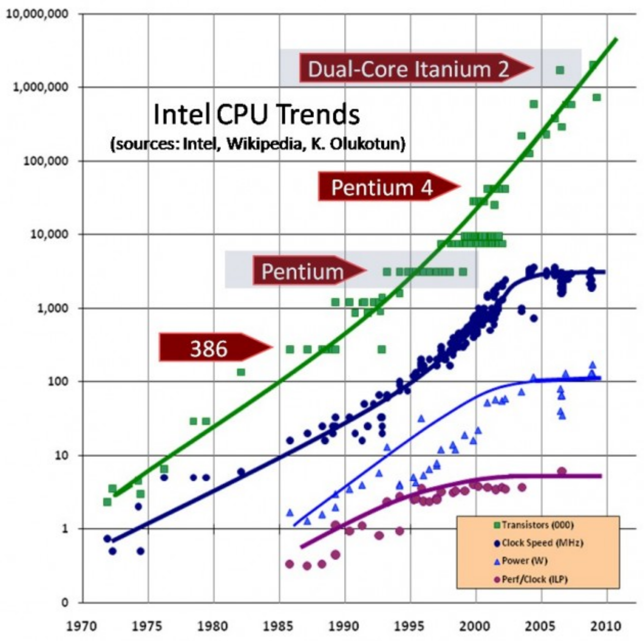
- Moore's law is a statement on transistor count/hardware complexity for fixed price
- Moore's law used to translate directly into performance for decades
- Let's study performance growth for the biggest machines in the world

(Super)Computer speed follows Moore's law



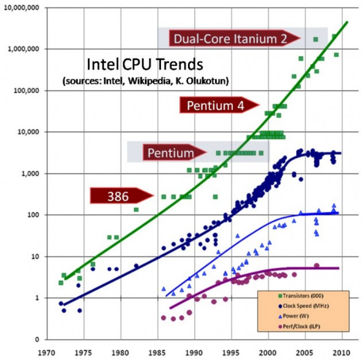
Source: www.top500.org

- Source: Top-500 (`www.top500.org` from SC and ISC)
- Moore's law seems to continue to hold for speed, too
- However . . .


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

The free lunch is over . . .

Durham University
Department of Computer Science



H. Sutter: The Free Lunch Is Over—A Fundamental Turn Toward Concurrency in Software

- These curves quantitatively are the same for all vendors
- Seems that the free performance lunch is over
- Increase of performance stems from **increase in concurrency**

$\Rightarrow$ learn how to exploit concurrency

. . . though we rely on a performance increase

Three classic objectives of HPC/science:

- Throughput & efficiency
  Reduce cost as results are available faster; scientists and engineers can do more experiments in a given time frame or study more parameters (UQ)
- Response time
  Urgent computing as required for tsunami prediction or in medical applications, e.g.; interactive parameter studies (computational steering)
- Problem size
  New insight through never-seen resolution/zoom into scales

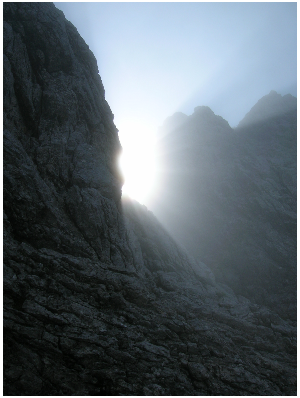
Solution: If chips don't become faster anymore, we have to squeeze more of them into one computer.

Implication: If computers become "more parallel", we have to be able to write "more parallel" codes.

Concept of building block

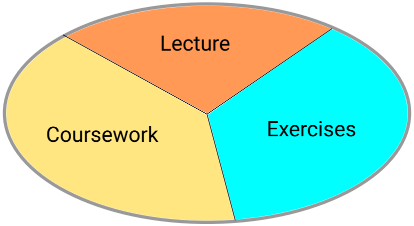
- Content
  - Moore’s law
  - Compare Moore’s law to (super-)computing trends
- Expected Learning Outcomes
  - The student knows Moore’s law
- Material & further reading/homework
  - Read the original paper from Moore
  - Read Herb Sutter: The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software (`http://www.gotw.ca/publications/concurrency-ddj.htm`) (also available on DUO)

Outline



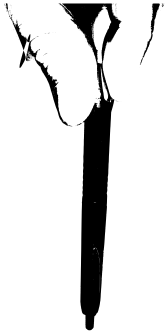
No free lunch
The lecture concept
Flynn’s taxonomy
Speedup laws

The three didactic pillars



- 4 $\times$ 4 sessions (two reduced this year)
- Lectures are split up into theory part plus hands-on
- Third didactic pillar is coursework
- Hands-on: BYOD, group work allowed

Script and notes



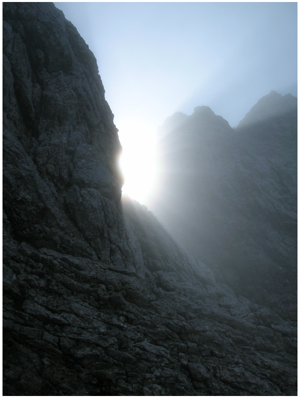
- Slides online yet not sufficient $\Rightarrow$ make notes
- Rudimentary script online $\Rightarrow$ I write as I teach

Tools

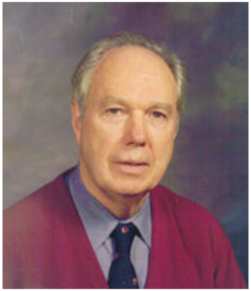
- Experiments/source code online on DUO
- It is all plain C/C++
- Using a Unix environment is strongly recommended
- Access to Hamilton/COSMA can be granted to students

  check registration procedures on DUO
- Tools used:
  - Intel Parallel Studio (free for students)
  - Likwid (optional)
  - Up-to-date GCC (if Intel compiler is not used)

Outline



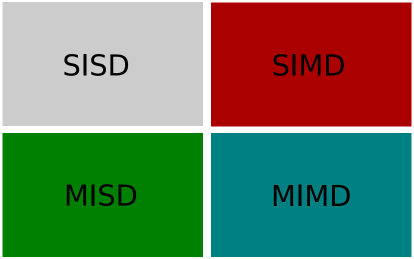
Michael J. Flynn



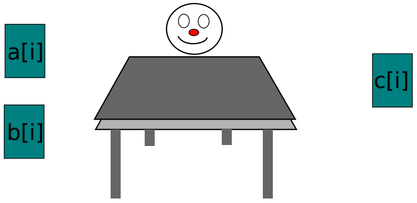
- Michael J. Flynn
- May 20, 1934 in New York City
- Stanford University (emeritus)
- Flynn, M.J.: Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computing C, 1972, pp. 948–960

Flynn's taxonomy

- **Instruction stream**: sequence of commands (top-down classification)
- **Data stream**: sequence of data (left-right classification)



SISD



$c = a + b \qquad a, b, c \in \mathbb{R}^N$

- SISD architecture: One ALU and one activity a time
- Runtime: $(2 \cdot \textit{load} + \textit{add} + \textit{store}) \cdot N$
- Usually *load*, *add*, *store* require multiple cycles (depend both on hardware and environment such as caches; cf. later sessions)

Example: (almost) assembler code in SISD

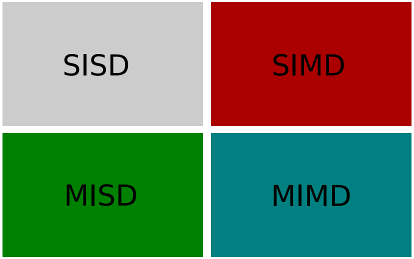
$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N; i++) {
  load a[i]
  load b[i]
  add
  store c[i]
}
```

Flynn’s taxonomy

Durham University
Department of Computer Science

- **Instruction stream**: sequence of commands (top-down classification)
- **Data stream**: sequence of data (left-right classification)



⇒ What is MIMD?

Assembler code running on MIMD hardware

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N/2; i++) { // Processor 1
  load a[i]
  load b[i]
  add
  store c[i]
}
for (int i=N/2; i<N; i++) { // Processor 2
  load a[i]
  load b[i]
  add
  store c[i]
}
```

MIMD remarks

MIMD describes many processors working on their data independently.

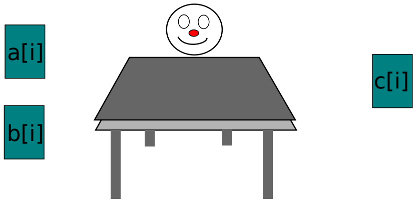
SPMD

- Single Program Multipe Data
- No hardware type but a programming paradigm
- All chips run same instruction stream, but they might process different steps at the same time (no sync)
- But: MIMD doesn't say that we have to run the same program/algorithm everywhere

Classification

- **SISD**
  - von Neumann’s principle
  - classic single processors with one ALU
  - one instruction on “one” piece of data a time
- **MIMD**
  - classic multicore/parallel computer: multiple SISD
  - asynchronous execution of several instruction streams (though it might be copies of the same program)
  - they might work on same data space (shared memory), but the processors do not (automatically) synchronise

SIMD



$c = a + b \qquad a, b, c \in \mathbb{R}^N$

- ALU: One activity a time
- Registers: Hold two pieces of data (think of two logical regs)
- Arithmetics: Update both sets at the same time
- Runtime: $(4 \cdot load + add + 2 \cdot store) \cdot N/2$
- *load*, *add*, *store* slightly more expensive than in SIMD
- *load* and *store* can grab two pieces of data in one rush when they are stored next to each other

Assembler code running on SIMD hardware

Durham University
Department of Computer Science

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N/2; i++) { // Processor 1
  load a[i] into





}
```

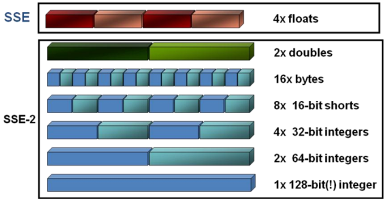
AVX/SSE

Durham University
Department of Computer Science

Vectorisation: Rewrite implementation to make it work with small/tiny vectors.

- Extension to four or eight (single precision) straightforward
- Technique: Vectorisation/vector computing
- Hardware realisation: Large registers holding multiple (logical) registers

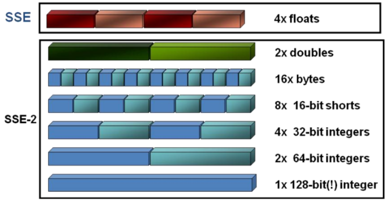
SSE



(C) Intel

- SSE = Streaming SIMD Extension
- SIMD instruction instruction set introduced with Pentium III (1999)
- Answer to AMD's 3DNow (computer games)
- Concept stems from the days when Cray was a big name
- Around 70 single precision instructions
- AMD implements SSE starting from Athlong XP (2001)

AVX



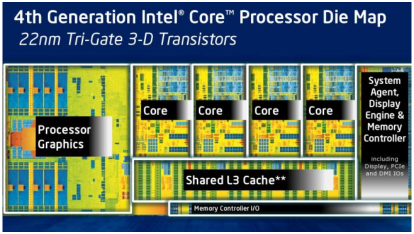
(C) Intel

- Double number of registers
- However, multiple cores might share registers (MIC)
- Operations may store results in third register (original operand not overwritten)
- AVX 2–AVX512: Gather and scatter operations and fused multiply-add

Classification

- **SISD**
  - von Neumann's principle
  - classic single processors with one ALU
  - one instruction on "one" piece of data a time
- **MIMD**
  - classic multicore/parallel computer: multiple SISD
  - asynchronous execution of several instruction streams (though it might be copies of the same program)
  - they might work on same data space (shared memory), but the processors do not (automatically) synchronise
- **SIMD**
  - synchronous execution of one single instruction
  - vector computer: (tiny) vectors
- **MISD**
  - pipelining (we ignore it here)

Flynn’s taxonomy: lessons learned/insights


4th Generation Intel® Core™ Processor Die Map
22nm Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O

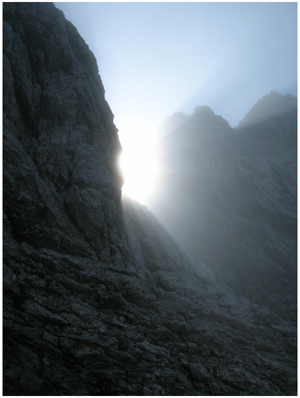
- All three paradigms are often combined (see Haswell above)
- There is more than one flavour of hardware parallelism
- There is more than one parallelisation approach
- MIMD substypes:
  - shared vs. distributed memory
  - races vs. data inconsistency
  - synchronisation vs. explicit data exchange

Concept of building block

Durham University
Department of Computer Science

- Content
  - Some historic background about Flynn's taxonomy
  - Definition of SISD, MISD, SIMD, MIMD
  - Explanation of individual classes
  - Recent trends
- Expected Learning Outcomes
  - The student knows Flynn's taxonomy and can define its acronyms
  - The student can give one example per Flynn hardware class from currrent architectures
  - The student can derive, for a given piece of hardware, into which class this hardware belongs to, i.e. apply the taxonomy
  - The student has an idea what terms out-of-order execution, caches, hyperthreading mean
- Material & further reading/homework
  - Hager/Wellein: Introduction to High Performance Computing for Scientists and Engineers. CRC Press. Chapter 1.1
  - Videos: `https://www.youtube.com/watch?v=hebqwJ3z5Jo`

Outline



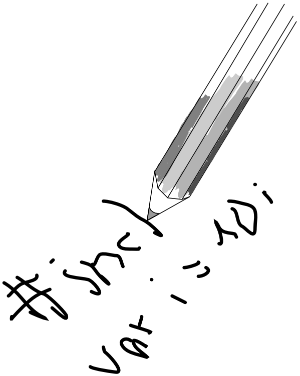
No free lunch
The lecture concept
Flynn’s taxonomy
Speedup laws

Gene Amdahl

Gene Amdahl, 2008

- Gene Amdahl
- November 16, 1922 in South Dakota
- Computer architect (own company and IBM)
- Amdahl, G. M. (1967): *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings (30): 483–485

A gedankenexperiment



- A single-core computer requires 64s to complete a job.
- How much time should a dual core computer require?
- How much time should a quadcore require?
- How would you define the terms speedup and efficiency?
- Write down three reasons why the speedup might actually be lower than expected.

Speedup and efficiency

We are usually interested in two metrics:

Speedup: Let $t(1)$ be the time used by one processor. $t(p)$ is the time required by p processors. The speedup is

$$S(p) = \frac{t(1)}{t(p)}.$$

Efficiency: The efficiency of a parallel application is

$$E(p) = \frac{S(p)}{p}.$$

Amdahl's Law

Durham University
Department of Computer Science

$$t(p) = f \cdot t(1) + (1-f)\frac{t(1)}{p}$$

Ideas:

- Focus on simplest model, i.e. neglect overheads, memory, ...
- Assume that code splits up into two parts: something that has to run serially (f) with a remaining code that scales
- Assume that we know how long it takes to run the problem on one core.
- Assume that the problem remains the same but the number of cores is scaled up

Remarks:

- We do not change anything about the setup when we go from one to multiple nodes. This is called **strong scaling**.
- The speedup then derives directly from the formula.
- In real world, there is some concurrency overhead (often scaling with p) that is neglected here.

Speedup and efficiency—revisited

Durham University
Department of Computer Science

Speedup: Let $t(1)$ be the time used by one processor. $t(p)$ is the time required by p processors. The speedup is

$$S(p) = \frac{t(1)}{t(p)}.$$

- What is a natural upper bound for the speedup?
- Are speedups smaller than 1 possible?
- What is superlinear speedup?

Efficiency: The efficiency of a parallel application is

$$E(p) = \frac{S(p)}{p}.$$

- What is an upper bound for the efficiency?
- What is a lower bound for the efficiency?

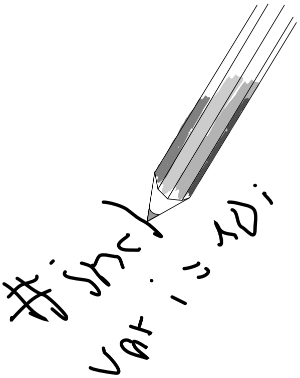
John Gustafson

Durham University
Department of Computer Science

John Gustafson, 2008

- John L. Gustafson
- January 19, 1955
- Computer scientists (Intel, AMD and others)
- John L. Gustafson: *Reevaluating Amdahl’s Law*, Communications of the ACM 31(5), 1988

A gedankenexperiment



- A supercomputer with 64 nodes requires 2s to complete a job.
- How much time would a single node computer require if the program has a sequential fraction of instructions f?
- Write down the speedup.

Gustafson’s Law

$$t(1) = f \cdot t(p) + (1 - f)t(p) \cdot p$$

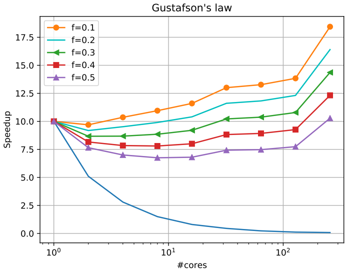
Ideas:

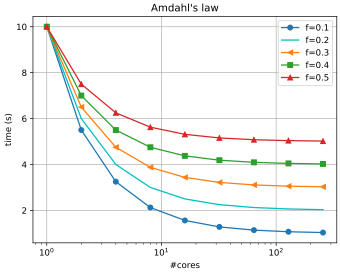
- Focus on simplest model, i.e. neglect overheads, memory, . . .
- Assume that code splits up into two parts (cf. BSP with arbitary cardinality): something that has to run serially (f) and the remaining code that scales.
- Assume that we know how long it takes to run the problem on p ranks.
- Derive the time rerquired if we used only a single rank.

Remarks:

- Single node might not be able to handle problem and we assume that original problem is chosen such that whole machine is exploited, i.e. problem size is scaled This is called **weak scaling**.
- The speedup then derives directly from the formula.
- In real world, there is some concurrency overhead (often scaling with p) that is neglected here.

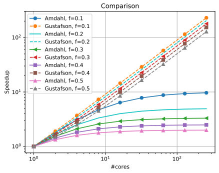
Python exercise





Disclaimer: Only PP session with Python!

Python exercise: Comparison of the speedup laws



$$t(p) = f \cdot t(1) + (1-f)\frac{t(1)}{p} \qquad t(1) = f \cdot t(p) + (1-f)t(p) \cdot p$$

$$S(p) = \frac{t(1)}{t(p)} \qquad S(p) = \frac{t(1)}{t(p)}$$

$$S(p) = \frac{1}{f + (1-f)/p} \qquad S(p) = f + (1-f)p$$

- It depends on whether you fix the problem size.
- It hence depends on your purpose.
- It is crucial to clarify assumptions a priori.
- It is important to be aware of shortcomings.

Concept of building block

- Content
  - Run through assumptions/ideas of Amdahl’s law
  - Derive formulas from these assumptions
  - Run through assumptions/ideas of Gustafson’s law
  - Derive formulas from these assumptions
- Expected Learning Outcomes
  - The student knows Amdahl’s law/strong scaling
  - The student knows Gustafson’s law/weak scaling
  - The student can explain the underlying ideas of the laws and derive the laws from these assumptions/ideas
  - The student can compute the speedup of a given code model
  - The student can present speedup graphs for a code
  - The student can explain when and why Amdahl’s law/Gustafson’s fails
- Material & further reading
  - M. Wolfe: *Compilers and More: Is Amdahl’s Law Still Relevant?* `http://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/`
  - A. Suleman: *Parallel Programming: When Amdahl’s law is inapplicable?* `http://www.futurechips.org/thoughts-for-researchers/parallel-programming-gene-amdahl-said.html`
  - Gustafson’s paper on speedup laws: `http://www.johngustafson.net/pubs/pub13/amdahl.htm`

Summary, outlook & homework

Concepts discussed:

- Motivation of parallel programming through hardware evolution
- Lecture concept
- Machine types (three of them)
- Traditional speedup laws

Next:

- Programming a multicore computer