Mestrado em Engenharia Informática

2024/25

João Luís Sobral

Course foundations

- **Lecturing/teaching team**
  - Lectures: João Sobral
  - Lab classes: João Sobral, Rui Silva, João Barbosa, Miguel Braga
- **Documentation & oral lectures/classes**
  - Documentation (slides, lab guides, books, reports…): English, on BB
  - Oral presentations and work defence: optional, but usually Portuguese
- **Expected background**
  - Imperative Programming and Computer Architecture
  - Lab skills in Unix, C programming and debugging
- **Assessment**
  - 1x written test/exam (test 20th Dec, exam 24th Jan, weight: 30%)
  - 2x intermediate works w/ reports (submission 21th Oct & 19th Nov, weight: 2x15%)
  - 1x final work w/ report & oral defence (end of 2024 & 6-10 Jan, weight: 40%)

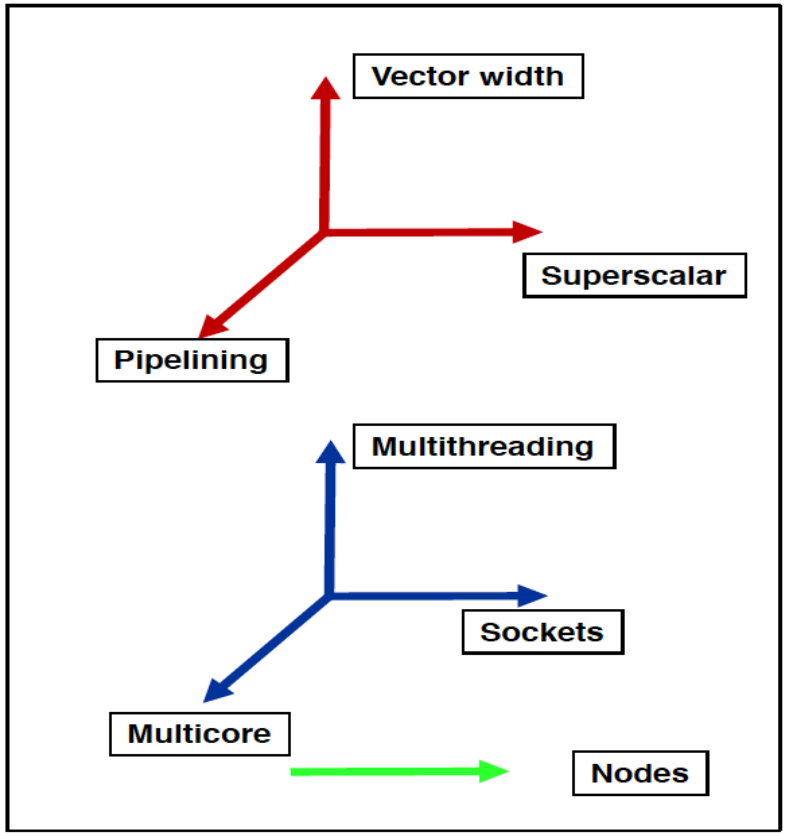
Focus on this course: performance engineering

How:

- plan and code efficient algorithms & data structures
- profiling & evaluating (through measurements) the execution efficiency
- understanding the organization of a computer system (its architecture)

Where in the hardware:

- in sequential code with ILP
  - pipelining
  - superscalar w/ out-of-order exec
  - vector processing
- in a memory hierarchy
  - multi-level caches (+ distributed memory)
- w/ support for thread parallelism
  - multithreading in-core (SMT)
  - multithreading in multicore
- w/ computing accelerators (GPU, …)
- w/ support for process-based parallelism
  - multiprocessing in a computer cluster



Key textbook (1)

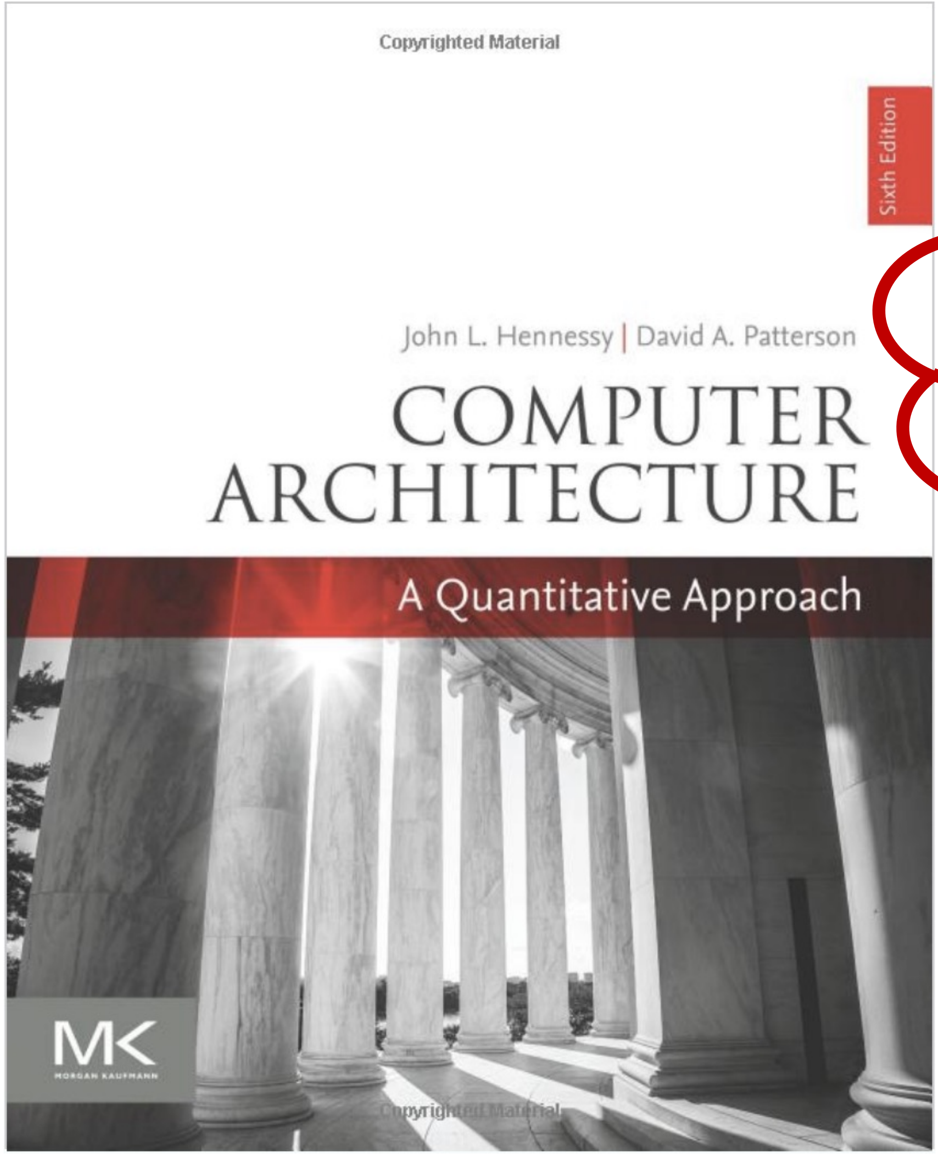
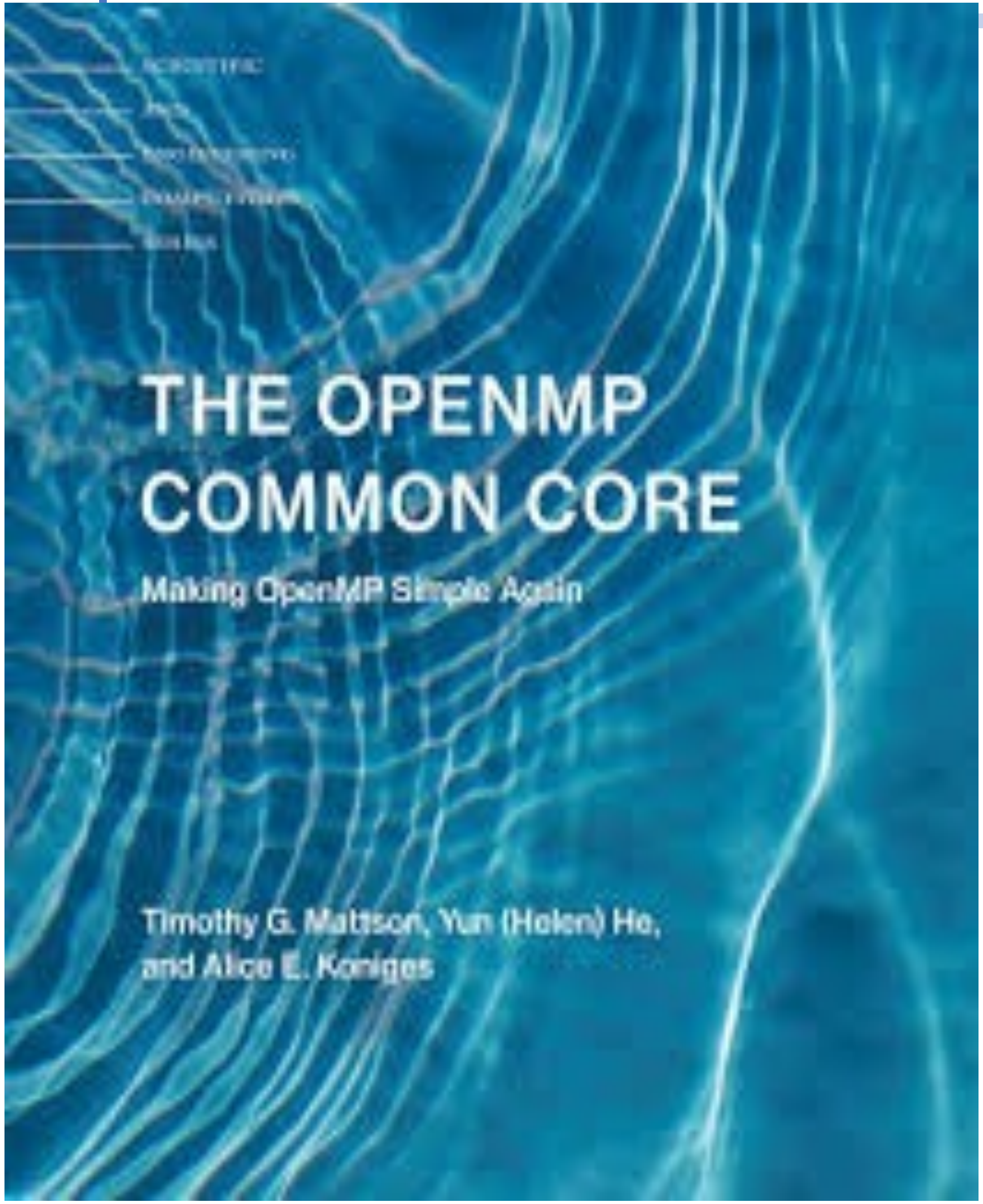


Table of Contents

Key textbook *(2)*


THE OPENMP COMMON CORE
Making OpenMP Simple Again
Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges

Taylor & Francis Group
an informa business

Key textbook *(3)*

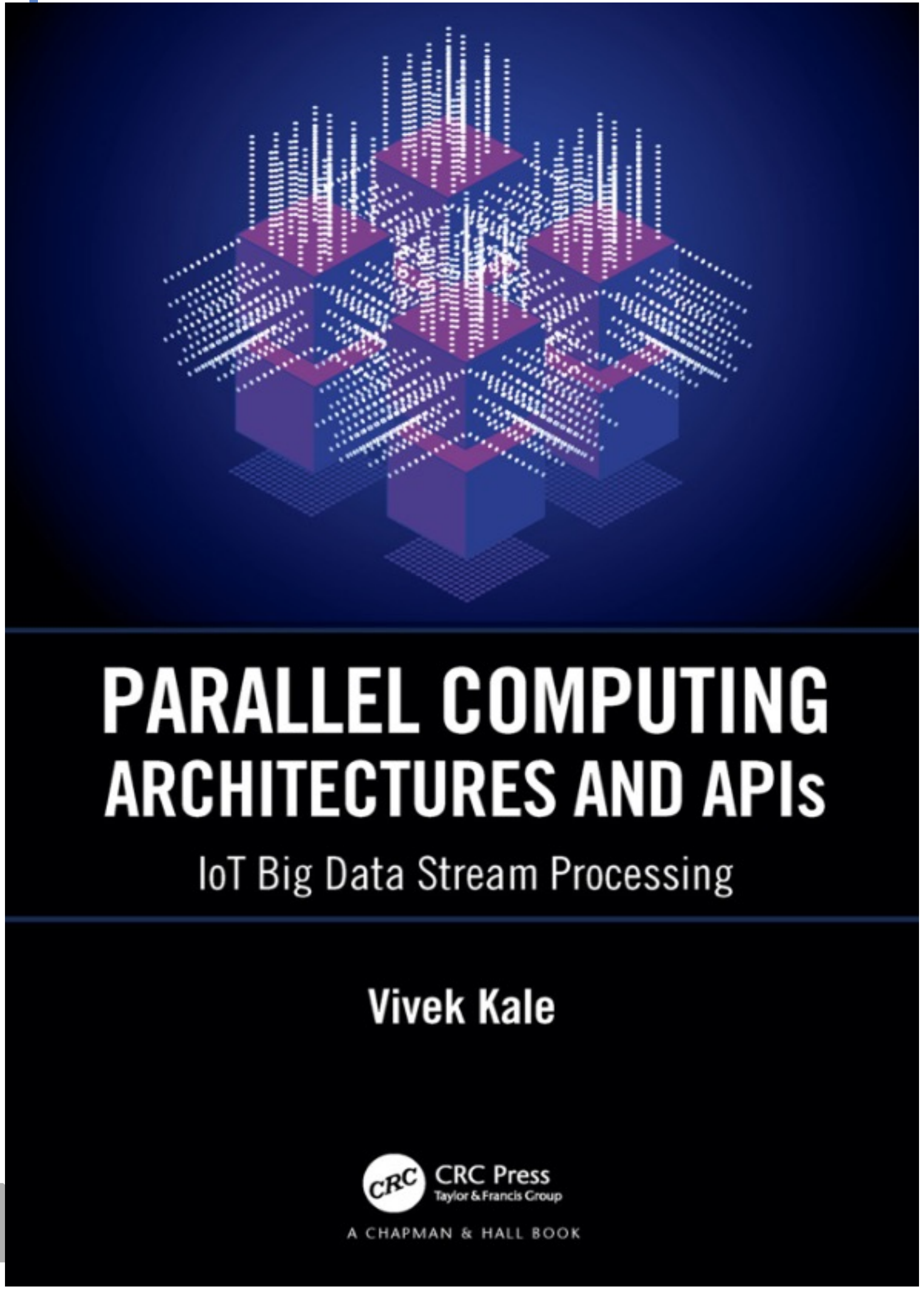

PARALLEL COMPUTING
ARCHITECTURES AND APIs
IoT Big Data Stream Processing
Vivek Kale
CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

TABLE OF CONTENTS

Key textbook (4)

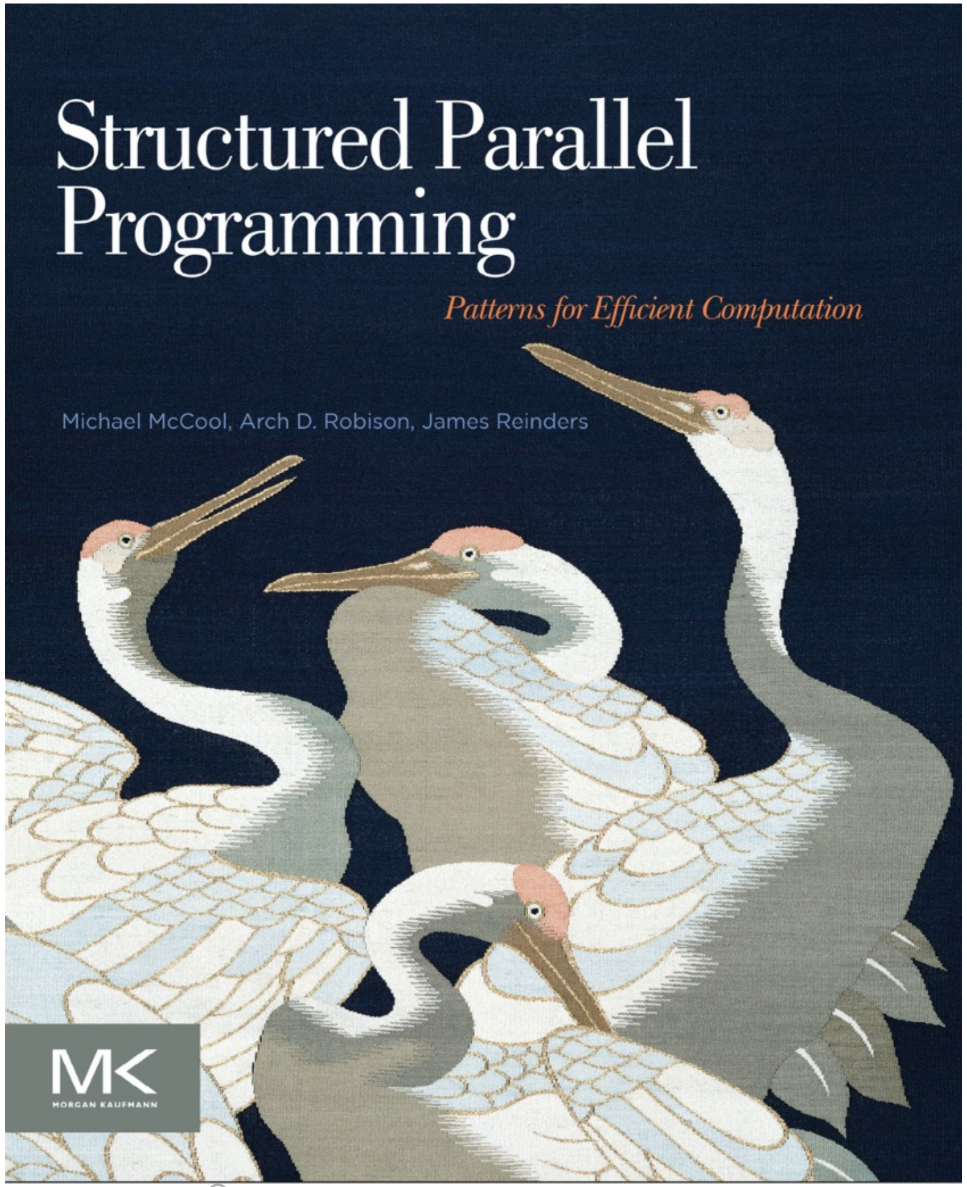
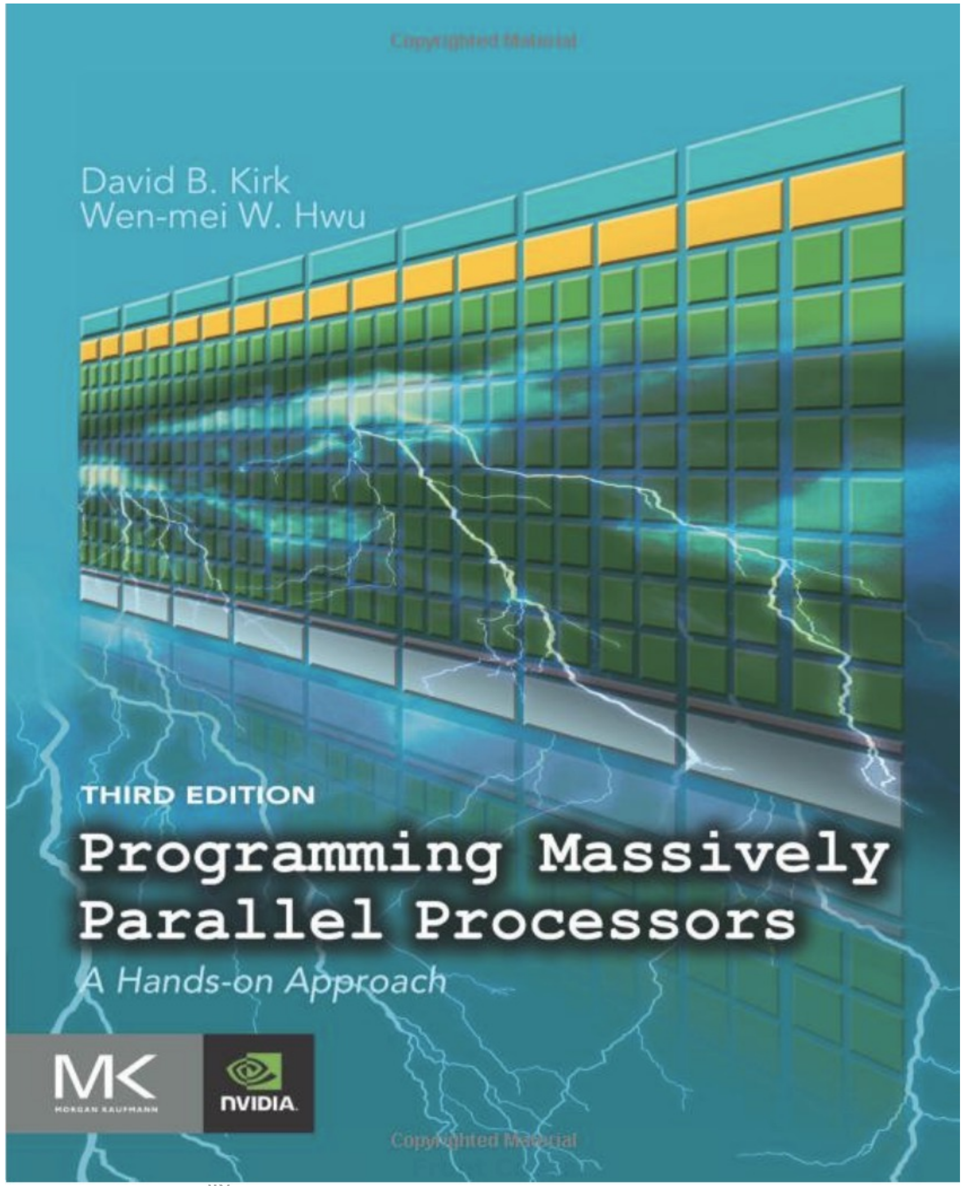

Structured Parallel Programming
Patterns for Efficient Computation
Michael McCool, Arch D. Robison, James Reinders
MK
MORGAN KAUFMANN

Table of contents

Recommended textbook *(1)*



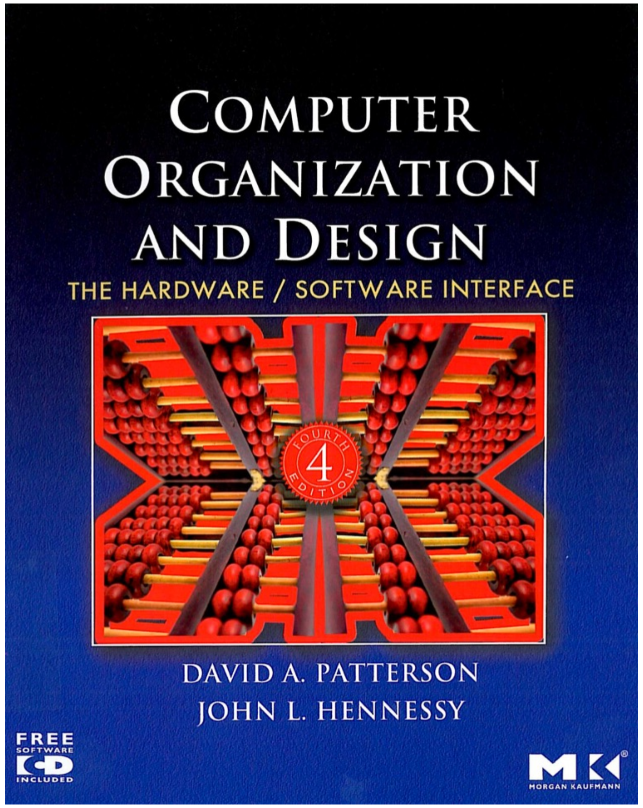
Contents

GPU & CUDA

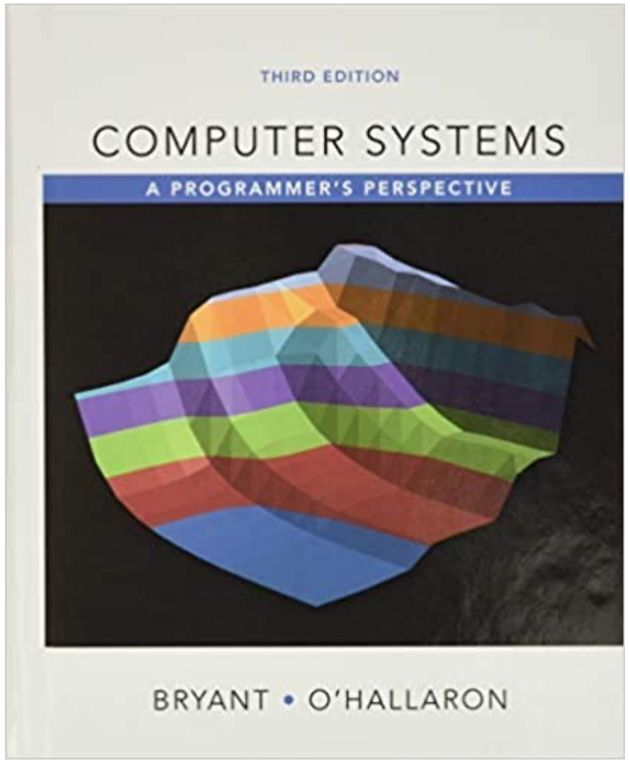
Background concepts from a basic Computer Systems course

Some notes/comments for this course:

– ***some slides are borrowed from***


COMPUTER ORGANIZATION AND DESIGN
THE HARDWARE / SOFTWARE INTERFACE
FOURTH 4 EDITION
DAVID A. PATTERSON
JOHN L. HENNESSY
FREE SOFTWARE CD INCLUDED
MK MORGAN KAUFMANN

and some from



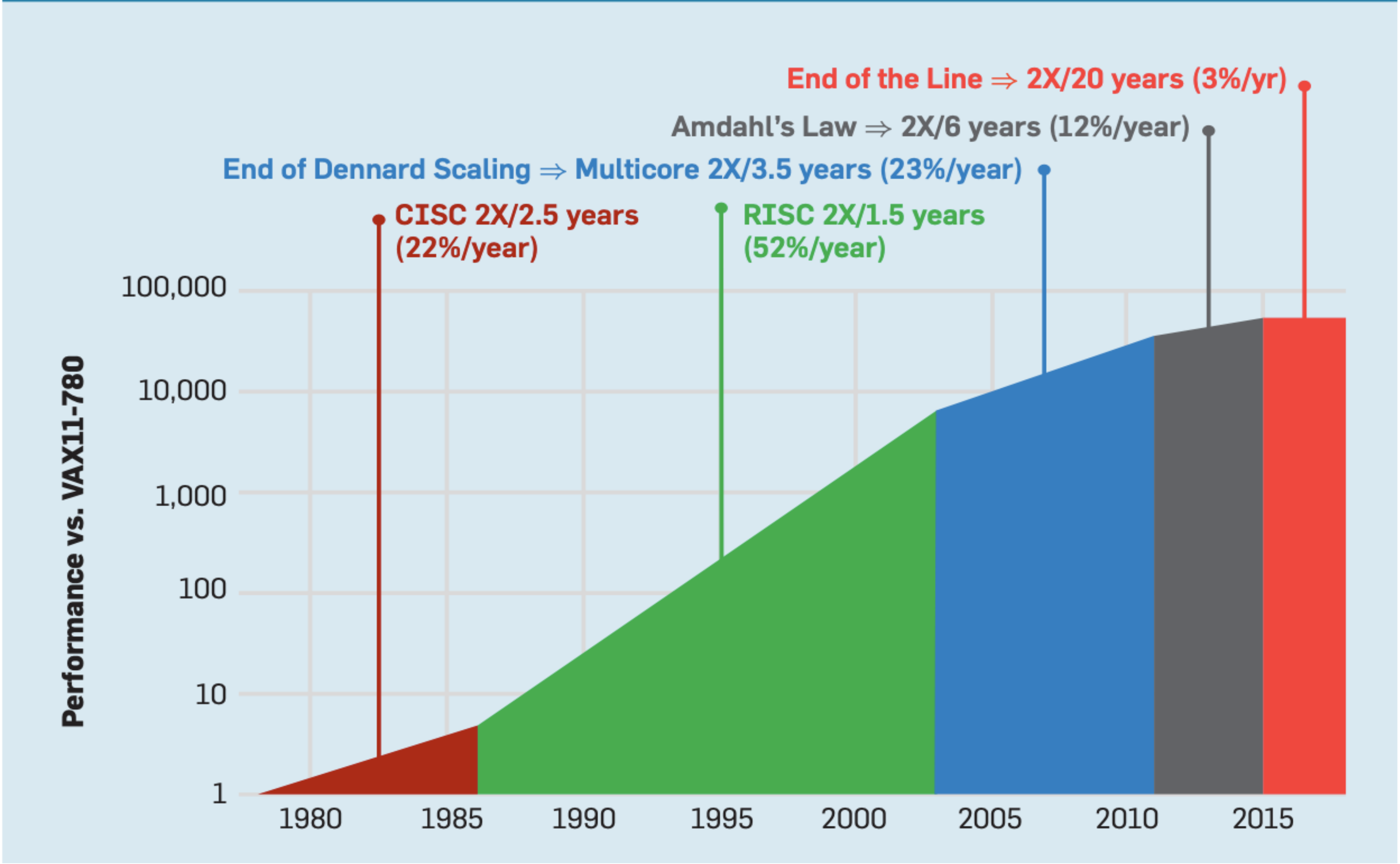
Growth of computer performance *(single-core)*

A New Golden Age for Computer Architecture

BY JOHN L. HENNESSY AND DAVID A. PATTERSON

COMMUNICATIONS OF THE ACM | FEBRUARY 2019 | VOL. 62 | NO. 2

Figure 6. Growth of computer performance using integer programs (SPECintCPU).

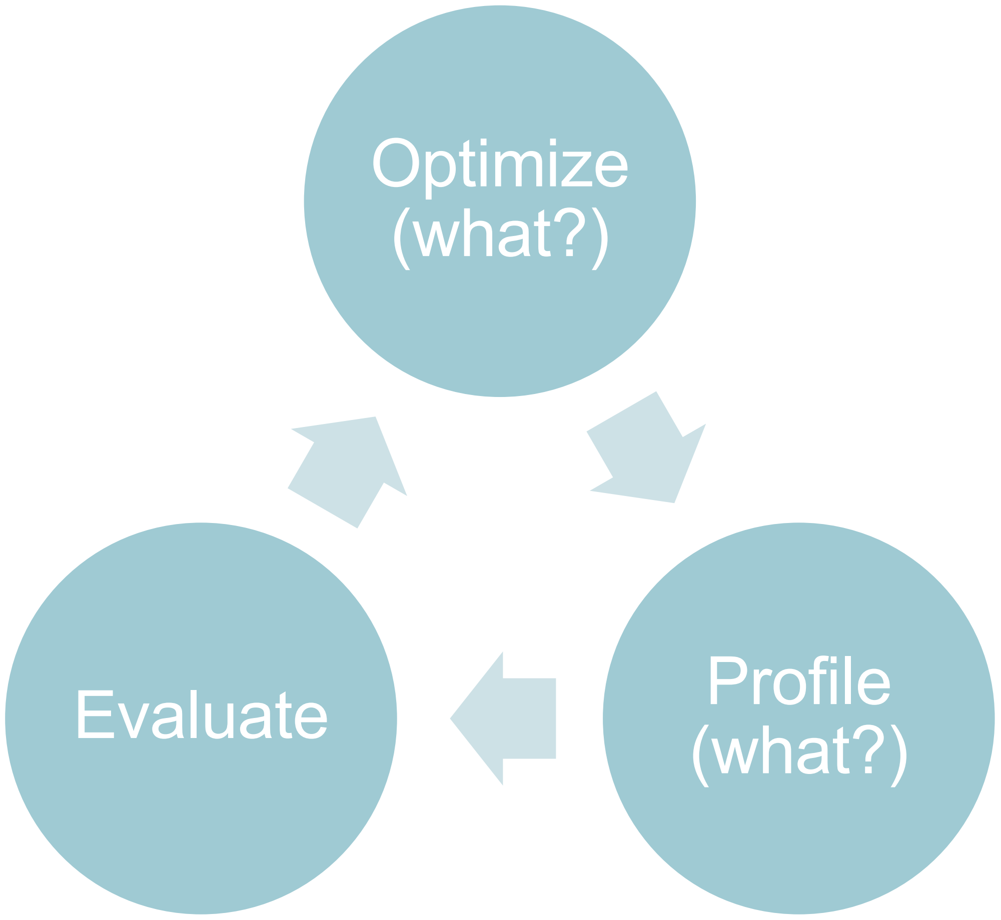


Current Trends in Architecture

- Cannot continue to leverage Instruction-Level Parallelism (ILP)
  - Single processor performance improvement ended in 2003
- New models for performance:
  - Data-level **parallelism** (DLP)
  - Thread-level **parallelism** (TLP)
  - Process-level **parallelism**
  - **Parallelism** with accelerators
- These require explicit restructuring of the application

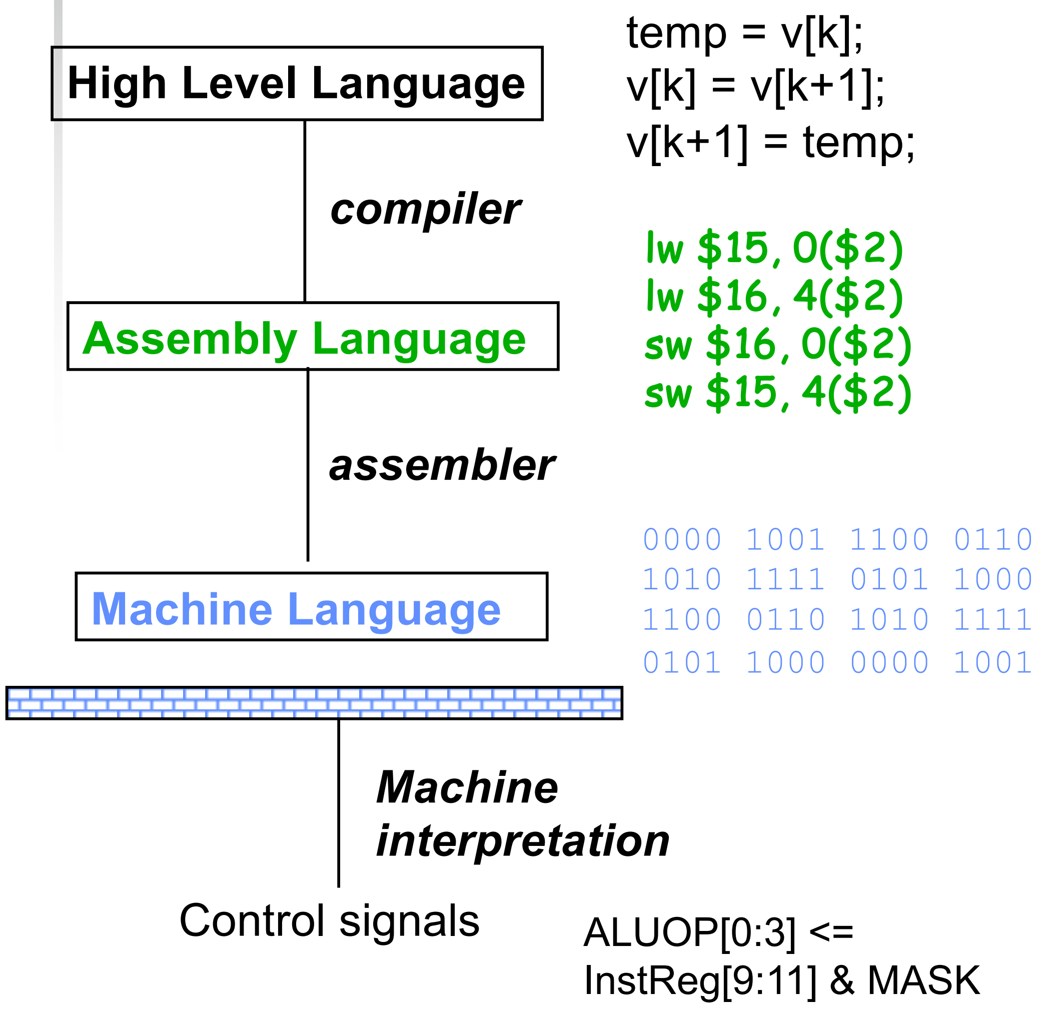
Performance optimization

- **Typical optimisation cycle**



Performance evaluation

- **Can be performed at different levels**

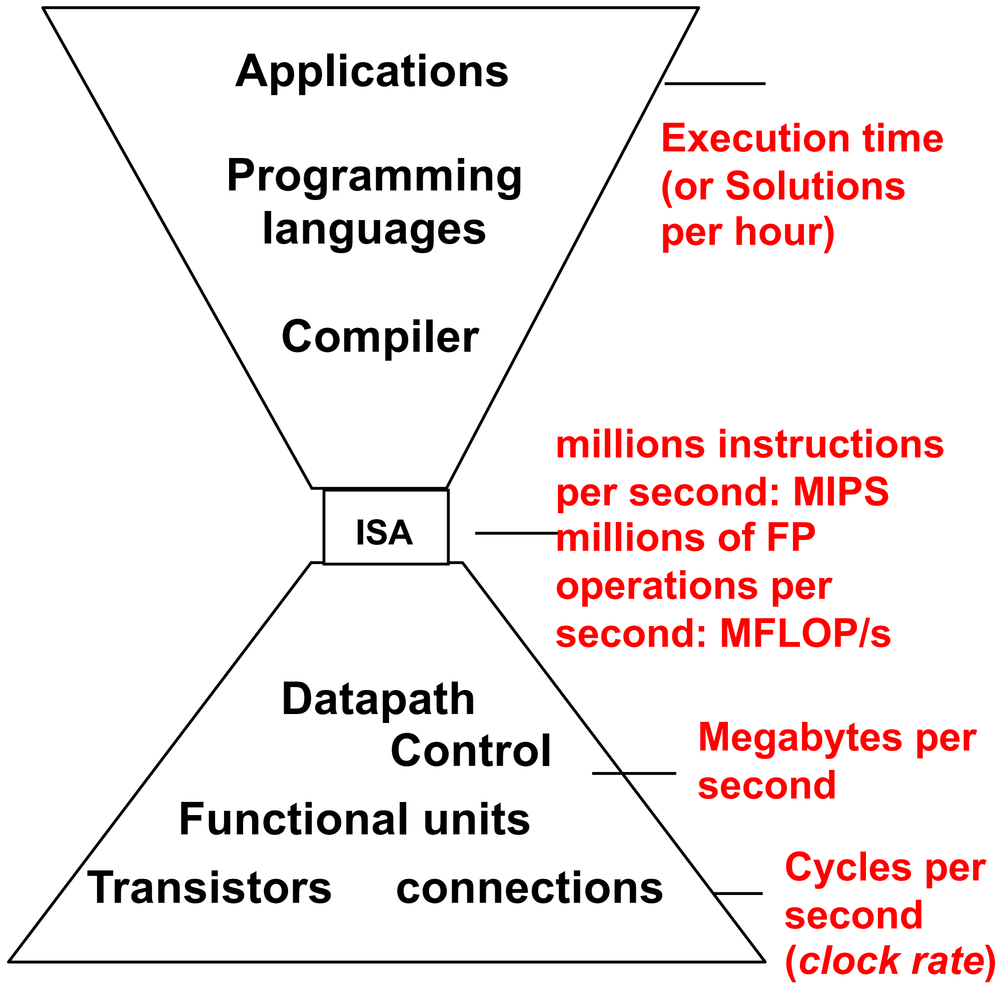


```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

```
0000 1001 1100 0110
1010 1111 0101 1000
1100 0110 1010 1111
0101 1000 0000 1001
```

```
ALUOP[0:3] <=
InstReg[9:11] & MASK
```


Applications
Programming languages
Compiler
ISA
Datapath
Control
Functional units
Transistors
connections
Execution time (or Solutions per hour)
millions instructions per second: MIPS
millions of FP operations per second: MFLOP/s
Megabytes per second
Cycles per second (clock rate)

Bandwidth and Latency

- Bandwidth or throughput
  - Total work done in a given time
  - 32,000-40,000x improvement for processors
  - 300-120x improvement for memory and disks
- Latency or response time
  - Time between start and completion of an event
  - 50-90x improvement for processors
  - 6-8x improvement for memory and disks

Measuring Performance

- Typical performance metrics:
  - Response time
  - Throughput
- Speedup of X relative to Y
  - Execution time$_Y$ / Execution time$_X$
- Execution time
  - Wall clock time: includes all system overheads
  - CPU time: only computation time
- Benchmarks
  - Kernels (e.g. matrix multiply)
  - Toy programs (e.g. sorting)
  - Synthetic benchmarks (e.g. Dhrystone)
  - Benchmark suites (e.g. SPEC06fp, TPC-C)

Principles of Computer Design

- The Processor Performance Equation

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

Measuring the CPU Time (single-core): an example

Consider the following piece of executable x86 code:

```
  movl 10, %eax
  movl 0, %ecx
loop:
  addl %eax, %ecx
  decl %eax
  jnz loop
```

Estimate its execution time in a computer system with a 2 GHz clock frequency and a CPI=1.5.

CPU_{time} = #Instr * CPI * Clk_{cycle}

PU_{time} = 32 * 1.5 * 1/(2*10^{-9}) sec = 24* 10^{-9} s = **24 ns**

Lei de Amdahl

O ganho no desempenho – *speedup* –
obtido com a melhoria do tempo de execução de uma parte do sistema, está limitado pela fração de tempo dessa parte.

$$\textit{Speedup}_{overall} = \frac{\text{Tempo\_exec}_{antigo}}{\text{Tempo\_exec}_{novo}} = \frac{1}{f / s}$$

f -	fração com melhoria **s**
s -	*speedup* da fração

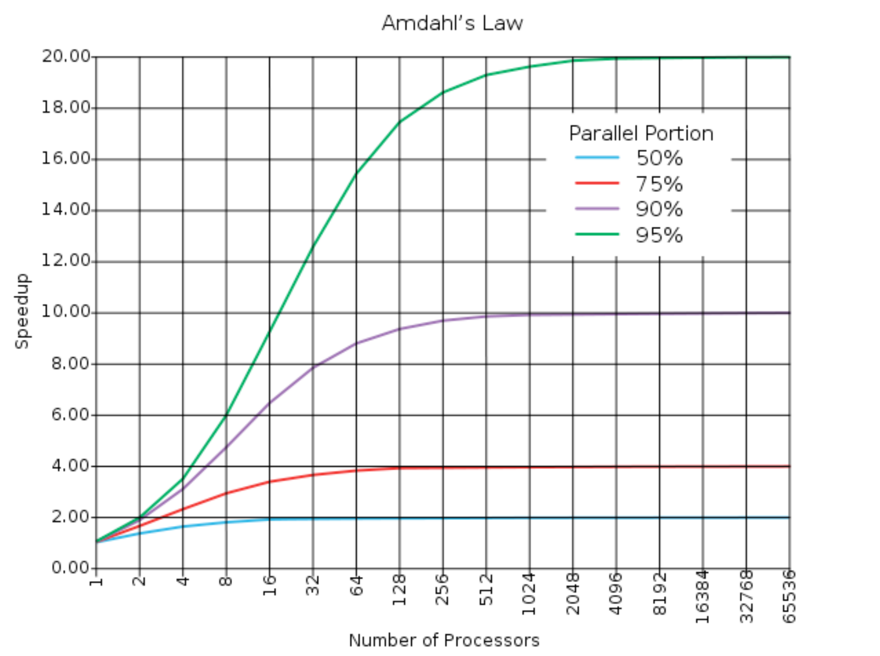
Ex.1: Se **<u>10%</u>** de um programa executa **<u>90x</u>** mais rápido

***Overall speedup* = 1.11**

Ex.2: Se **<u>90%</u>** de um prog executa **<u>90x</u>** mais rápido

***Overall speedup* = 9.09**

Paralelismo:
se $\mathbf{N_{proc}} \equiv$ ***speedup***, trocar **s** por $\mathbf{N_{proc}}$



Amdahl's Law

$$\text{Overall Speedup} = \frac{\text{Old execution time}}{\text{New execution time}}$$

$$= \frac{1}{\left((1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)}$$

$$Speedup = \frac{1}{(1-p) + p/N}$$

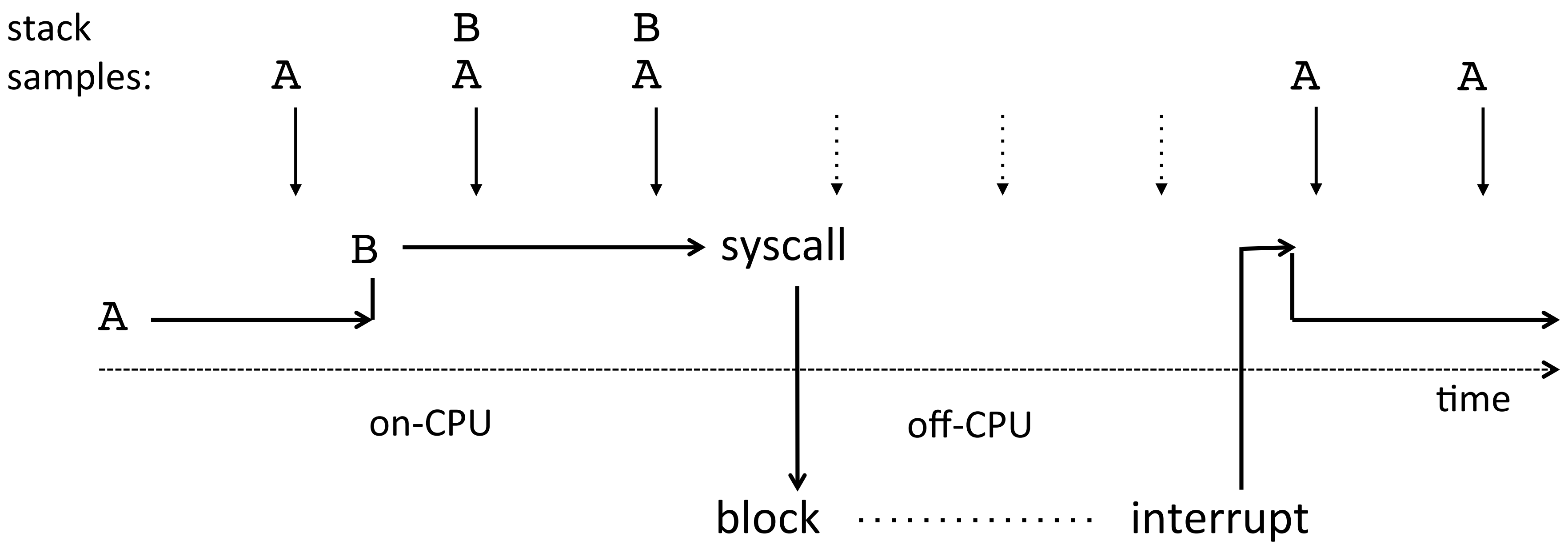
$$\text{Speedup}(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Serial part of job =
1 (100%) - Parallel part

Parallel part is divided
up by N workers

CPU profiling using sampling

- Record stacks at a timed interval: simple and effective
  - Pros: Low (deterministic) overhead
  - Cons: Coarse accuracy, but usually sufficient



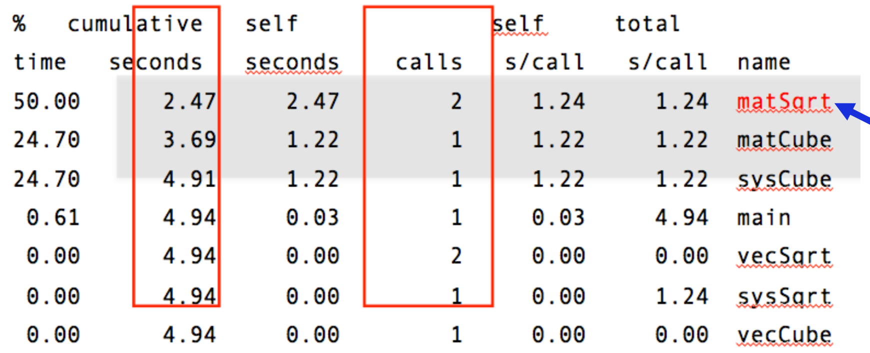
Profiling code execution

Techniques to measure the application time-profile (profiling)

Polling (sampling)

- the application is periodically interrupted to collect performance data
- Example: `gprof`

Flat view



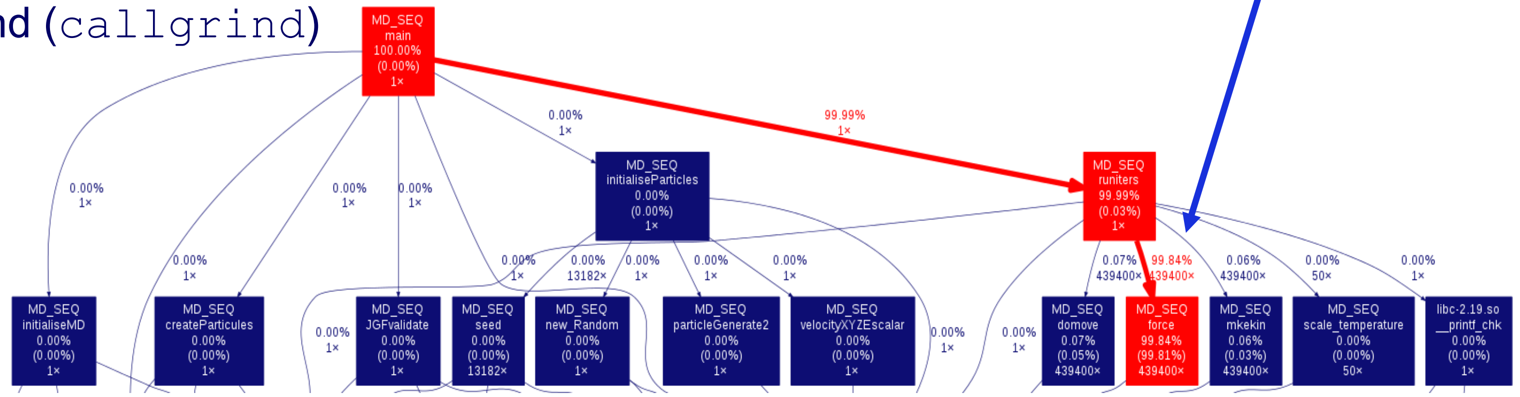
```
 %    cumulative   self              self     total
time    seconds   seconds    calls  s/call   s/call  name
50.00     2.47     2.47        2     1.24     1.24  matSqrt
24.70     3.69     1.22        1     1.22     1.22  matCube
24.70     4.91     1.22        1     1.22     1.22  sysCube
 0.61     4.94     0.03        1     0.03     4.94  main
 0.00     4.94     0.00        2     0.00     0.00  vecSqrt
 0.00     4.94     0.00        1     0.00     1.24  sysSqrt
 0.00     4.94     0.00        1     0.00     0.00  vecCube
```

Hot-spot

Instrumentation

- programmer (or tools) introduce code to collect performance data on useful events
  - may produce better results but also produces more interference (e.g., overhead)
- Example: Valgrind (`callgrind`)

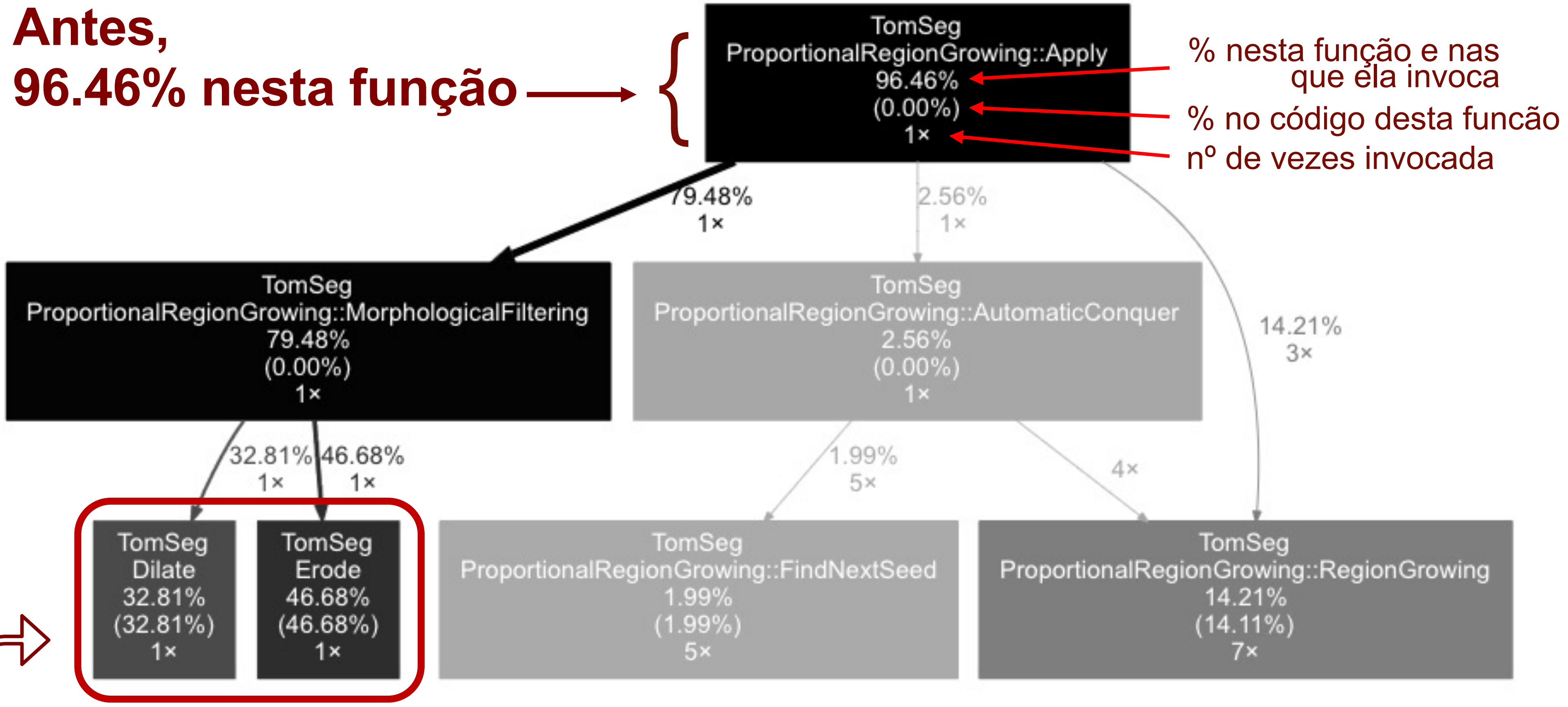
Tree view



MD_SEQ main 100.00% (0.00%) 1×
MD_SEQ initialiseParticles 0.00% (0.00%) 1×
MD_SEQ runiters 99.99% (0.03%) 1×
99.99% 1×
MD_SEQ initialiseMD 0.00% (0.00%) 1×
MD_SEQ createParticules 0.00% (0.00%) 1×
MD_SEQ JGFvalidate 0.00% (0.00%) 1×
MD_SEQ seed 0.00% (0.00%) 13182×
MD_SEQ new_Random 0.00% (0.00%) 1×
MD_SEQ particleGenerate2 0.00% (0.00%) 1×
MD_SEQ velocityXYZEscalar 0.00% (0.00%) 1×
MD_SEQ domove 0.07% (0.05%) 439400×
MD_SEQ force 99.84% (99.81%) 439400×
MD_SEQ mkekin 0.06% (0.03%) 439400×
MD_SEQ scale_temperature 0.00% (0.00%) 50×
libc-2.19.so __printf_chk 0.00% (0.00%) 1×
0.07% 439400×
99.84% 439400×
0.06% 439400×
0.00% 50×
0.00% 13182×
0.00% 1×

Code profiling: análise visual da melhoria de código duma função (antes)

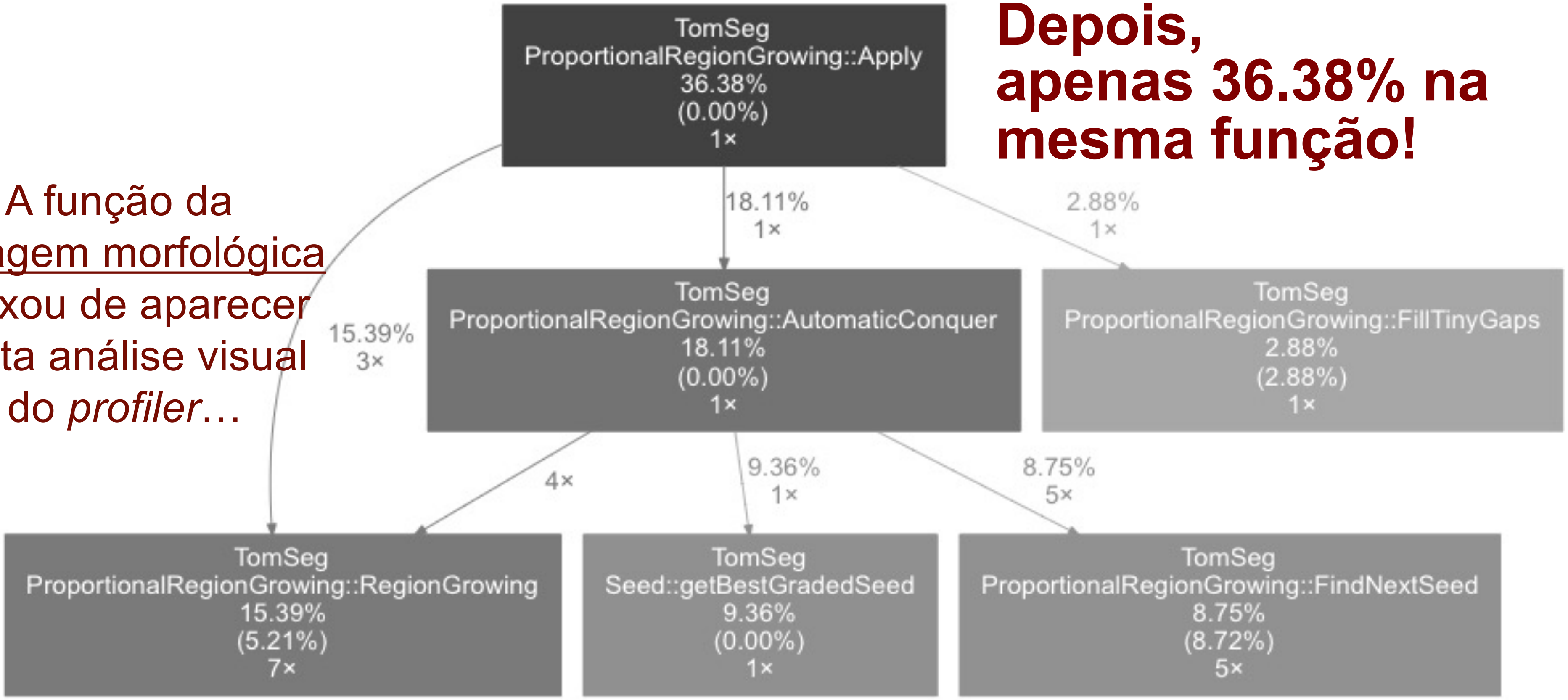
Antes,
96.46% nesta função



Quase 80% do tempo total é gasto nestas 2 funções da filtragem morfológica!
<u>Conclusão</u>: é aqui que se deve investir para melhorar a *performance* global

Figure 5.7.: Call-graph of the first version of *Propor. Region Growing* (DS3)

Code profiling: análise visual da melhoria de código duma função (depois)


TomSeg
ProportionalRegionGrowing::Apply
36.38%
(0.00%)
1×
18.11%
1×
2.88%
1×
15.39%
3×
TomSeg
ProportionalRegionGrowing::AutomaticConquer
18.11%
(0.00%)
1×
TomSeg
ProportionalRegionGrowing::FillTinyGaps
2.88%
(2.88%)
1×
4×
9.36%
1×
8.75%
5×
TomSeg
ProportionalRegionGrowing::RegionGrowing
15.39%
(5.21%)
7×
TomSeg
Seed::getBestGradedSeed
9.36%
(0.00%)
1×
TomSeg
ProportionalRegionGrowing::FindNextSeed
8.75%
(8.72%)
5×

Depois, apenas 36.38% na mesma função!

A função da <u>filtragem morfológica</u> deixou de aparecer nesta análise visual do *profiler*…

Figure 5.9.: Call-graph from the last version of *Propor. Region Growing* (DS3)