Algorithm Engineering

Jens K. Mueller

jkm@informatik.uni-jena.de

Department of Mathematics and Computer Science
Friedrich Schiller University Jena

Monday 20th October, 2014

Organization & Introduction

Administrative

- 6 LP course (1 LP is about 30 h)
  180 h/14 week $\approx$ 13 h/week (on average)
- Exercises sheets are to be handed in weekly
- **Monday 2.15pm** in **EAP2 3325** and
  Tuesday 10.15am in **EAP2 3325**

Course web page at http://theinf2.informatik.uni-jena.de/Lectures/Algorithm+Engineering+(lecture).html

Lecture and Tutorial

- Exemplify algorithm engineering
- Develop/Show tools
- Toy problems for illustration
- Follow best practices
- Rubber ducking

Your Part

- Do the weekly exercises
- Special last exercise combines all previous exercises (little project)
- 50% of all points for admission

Rules

- Use available resources but give credit where credit is due
- Make sure you understand/know what you are doing
- Cheating will be reported and results in failing this course

Examination

- Oral exam (very near the course's end)
- About your last exercise and the lecture
- To evaluate your knowledge and skills

Algorithm Engineering

Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a robust, efficient, well tested, and easily usable implementation. Thus it encompasses a number of topics, from modeling cache behavior to the principles of good software engineering; its main focus, however, is experimentation.

— Bader, Moret, and Sanders in [BMS02]

Why Algorithm Engineering

Gap between theoretical analysis and empirical performance

- Simplifying assumption needed for theoretical analysis
  Constants in big O notation, memory hierarchy, NUMA architectures, advanced CPU instructions
- Asymptotic analysis
  Asymptotic optimal algorithms can be impracticable
- Worst case analysis
  Worst cases may not occur in actual input data
- Problem specific properties not exploited
- NP-hard problems may be solvable for practical applications

Bridge the gap by addressing the (simplifying/impractical) assumptions (simplified machine model, big O notation, ...) by means of experimentation.

Experiment with Algorithms

You may not want to improve your implemented algorithm but you want to verify and know about its properties in real applications.

Key properties

- Usability
- Correctness
- Efficiency

Summary

Engineering an algorithm

1. Implement easy to understand, usable, and tested algorithm
2. Wring (desired) efficiency

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Summary

Engineering an algorithm

1. Implement easy to understand, usable, and tested algorithm
2. Wring (desired) efficiency

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

— Donald E. Knuth in [Knu74]

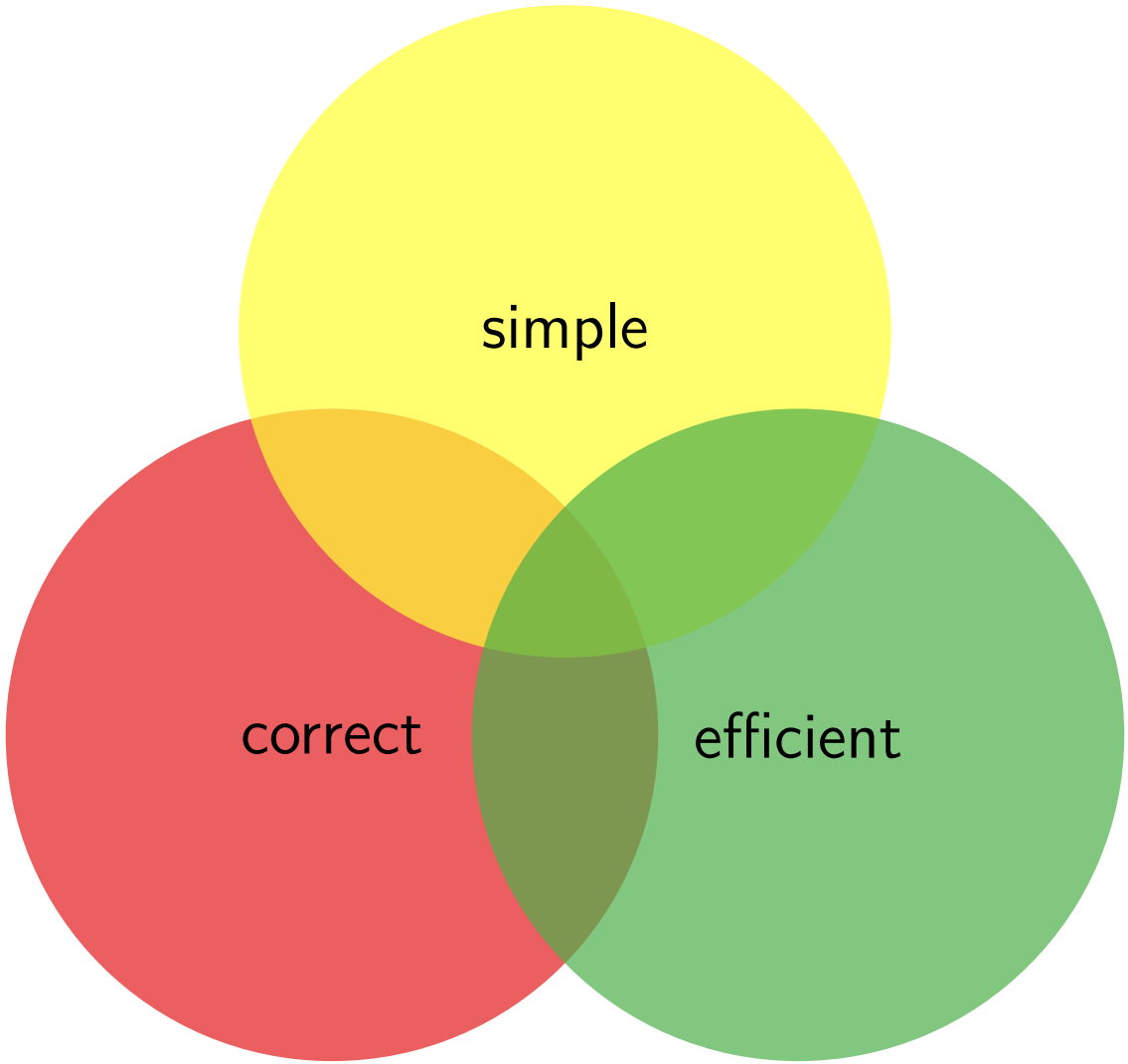
What I want you to take away

1. A strategy to engineer algorithms
2. What and how to exploit hardware for algorithm engineering

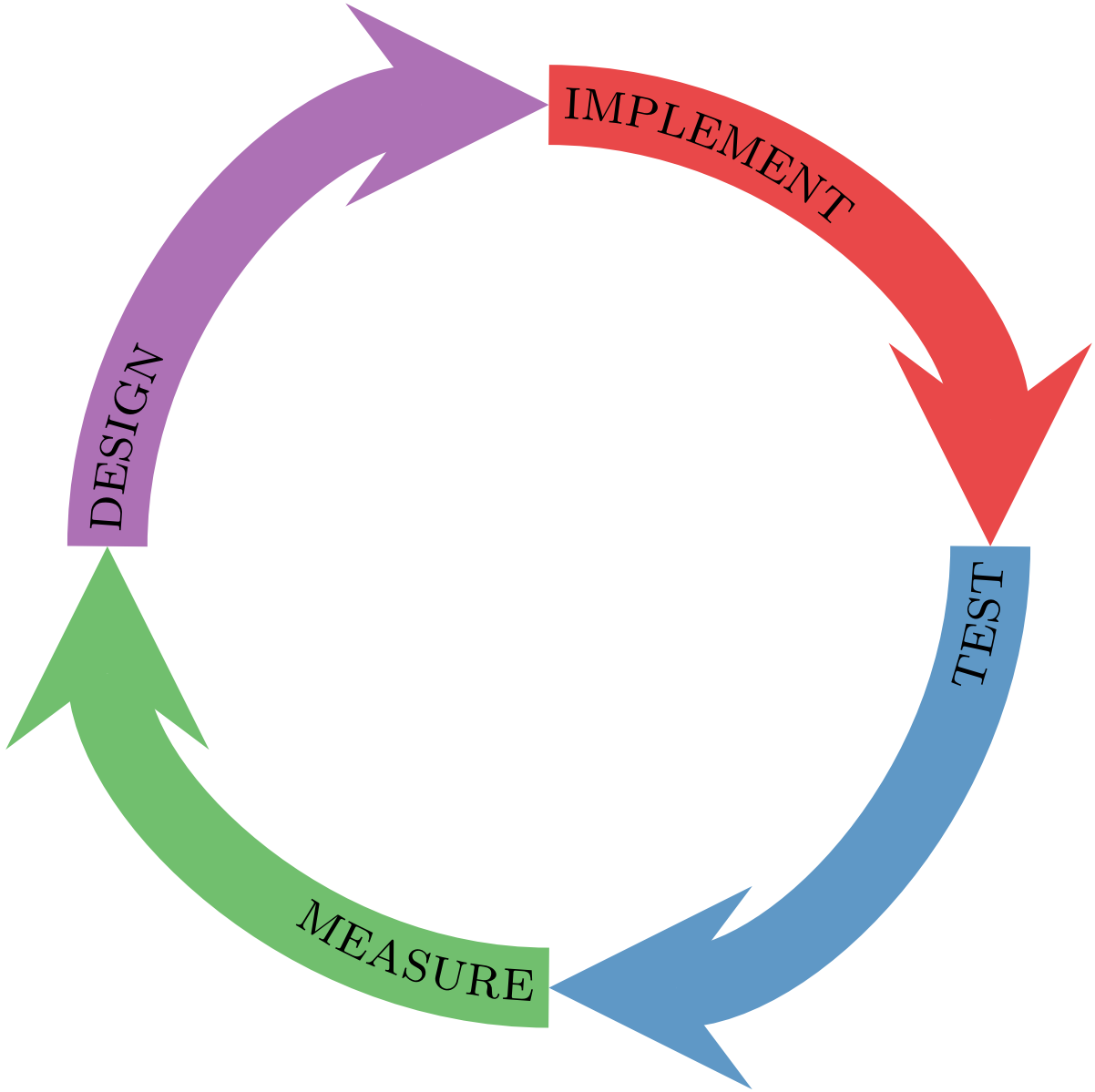
What to expect

- Programming intense
- Nitty-gritty details
- Assembly code
- CPU architecture

Paths of Glory



Each Step is a Cycle



Engineering Aspects

- Readability/Documentation (usability)
- Testing (correctness)
- Debugging (correctness)
- Profiling & Measuring (efficiency)
- Fail early (robustness)

Your Environment

- Choose a language (C, C++, or D with ≥ 2 compilers)
- Pick whatever editor/IDE you're comfortable with
- Track your source code
- Setup to build, compile and reproduce your results

Example D tools

- Install D latest compilers (dmd, gdc, ldc)
  http://wiki.dlang.org/Compilers
- Editors
  http://wiki.dlang.org/Editors
- IDEs
  http://wiki.dlang.org/IDEs
  - DDT (Eclipse-based) installation
  - Visual D (Visual Studio) installation
  - Mono-D installation
  - Code::Blocks

Literature

- Brian W. Kernighan and Rob Pike. The Practice of Programming. Addison-Wesley Longman, 1999. ISBN: 0-201-61586-X
- Randal E. Bryant and David R. O'Hallaron. Computer Systems: A Programmer's Perspective. 2nd. USA: Addison-Wesley, 2010. ISBN: 0136108040, 9780136108047

Rob Pike Rules

Rule 1 You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2 Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

Rule 3 Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.) For example, binary trees are always faster than splay trees for workaday problems.

Rob Pike Rules (cont.)

Rule 4 Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.
The following data structures are a complete list for almost all practical programs: array, linked list, hash table, binary tree.
Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters.

Rule 5 Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be selfevident. Data structures, not algorithms, are central to programming.

Rob Pike Rules (cont.)

Rule 6 There is no Rule 6.

Unix Philosophy

- Rule of Modularity
  Write simple parts connected by clean interfaces.
- Rule of Clarity
  Clarity is better than cleverness.
- Rule of Composition
  Design programs to be connected to other programs.
- Rule of Separation
  Separate policy from mechanism; separate interfaces from engines.
- Rule of Simplicity
  Design for simplicity; add complexity only where you must.

Unix Philosophy (cont.)

- Rule of Parsimony
  Write a big program only when it is clear by demonstration that nothing else will do.Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
- Rule of Transparency
  Design for visibility to make inspection and debugging easier.
- Rule of Robustness
  Robustness is the child of transparency and simplicity.
- Rule of Representation
  Fold knowledge into data so program logic can be stupid and robust.

Unix Philosophy (cont.)

- Rule of Least Surprise
  In interface design, always do the least surprising thing.
- Rule of Silence
  When a program has nothing surprising to say, it should say nothing.
- Rule of Repair
  When you must fail, fail noisily and as soon as possible.
- Rule of Economy
  Programmer time is expensive; conserve it in preference to machine time.

Unix Philosophy (cont.)

- Rule of Generation
  Avoid hand-hacking; write programs to write programs when you can.
- Rule of Optimization
  Prototype before polishing. Get it working before you optimize it.
- Rule of Diversity
  Distrust all claims for one true way.
- Rule of Extensibility
  Design for the future, because it will be here sooner than you think.

Summary

- Semantically equivalent programs may not have equal performance
- Performance matters in practical applications beyond theoretical analysis
- Adopt good engineering habits
- Measure, measure, measure, ...
- Squeezing the hardware out

References

[BMS02] David A. Bader, Bernard M. E. Moret, and Peter Sanders. "Algorithm Engineering for Parallel Computation". In: *Experimental Algorithmics*. Ed. by Rudolf Fleischer, Bernard Moret, and Erik Meineche Schmidt. 2002, pp. 1–23. ISBN: 978-3-540-00346-5. DOI: 10.1007/3-540-36383-1_1 (cit. on p. 8).

[BO10] Randal E. Bryant and David R. O'Hallaron. Computer Systems: A Programmer's Perspective. 2nd. USA: Addison-Wesley, 2010. ISBN: 0136108040, 9780136108047 (cit. on p. 20).

[KP99] Brian W. Kernighan and Rob Pike. The Practice of Programming. Addison-Wesley Longman, 1999. ISBN: 0-201-61586-X (cit. on p. 20).

References (cont.)

[Knu74] Donald E. Knuth. Structured Programming with Go to Statements. In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640 (cit. on pp. 11, 12).

[Pik89] Rob Pike. Notes on Programming in C. 1989. URL: http://www.lysator.liu.se/c/pikestyle.html.