Algorithm Engineering: It's All About Speed

Bernard M.E. Moret

moret@cs.unm.edu

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

Disclaimer

- Algorithm Engineering is also about
  - correctness
  - robustness
  - flexibility
  - portability
- But speed is fun and easy to measure ;-)

First Lecture: A Survey

- Motivation: Why Speed?
- A Brief History
- How Can We Get Speed?
- Algorithm Engineering Techniques
- Success Stories

Testing and Serial Speedup

- Computational Phylogenetics
- The Problem Of Test Sets
- On the Distinction Between Applications and Algorithms: A Question of Scale
- How Would *You* Like A Billion-Fold Speedup?

Measuring and Parallel Speedup

- Algorithm Engineering for Parallel Algorithms
- Measuring Parallel Execution
- Message-Passing Computations
- Shared-Memory Computations
- Can We Finally Put to Use 30 Years of Research in PRAM Algorithms?

First Lecture: A Survey

- Motivation: Why Speed?
- A Brief History
- How Can We Get Speed?
- Algorithm Engineering Techniques
- Success Stories

Why Speed?

- *Research Tools:* run many experiments to test hypotheses and for discovery
- *Production Tools:* obviously, save on resources
- *Make It Possible:* a million-fold speedup allows one to solve in a week what would otherwise have been infeasible (requiring millennia)

Comparisons Between Abstract Algorithms

- CS pioneers (Knuth, Floyd, etc.) always gave implementations.
- Jones started formal comparisons in early 80s.
- Many studies since on the best algorithms and data structures for basic abstract data types and basic routines (priority queues, search trees, hash tables, minimum spanning trees, shortest paths, convex hulls, Delaunay triangulations, matching and flow).

Going Beyond Abstract Algorithms

- Comparisons of priority queue implementations or coloring algorithms is not what industry and scientists really need.
- Industry has specific hard problems that cannot be solved without sophisticated algorithmic techniques, but that also exhibit enough special structure to enable the design of much better solutions than are possible for the general problem.

Going Beyond Abstract Algorithms

- Researchers push the envelope of what computation can deliver—some computational biologists run their code for a year or more on a cluster of machines to analyze their data.
- Such research requires an interdisciplinary team—a specialist with domain knowledge, an algorithm specialist, perhaps a systems specialist.

What Is Speed?

- To the theorist developing algorithms, it is the asymptotic worst-case running time.
- To the experimentally savvy algorithm developer, it might be the (more or less) exact worst-case running time couched in terms of the most common or expensive operations, such as main memory accesses.
- To a user, it is simply the time needed to run the code on her dataset.

What Is Speed (cont'd)?

- To the algorithm engineer, it is all of:
  - good asymptotic performance
  - low proportionality constants
  - fast running times on important real-world datasets
  - robust performance across a variety of data
  - robust performance across a variety of platforms
  - scalability to faster platforms and larger datasets

Example: Minimum Spanning Trees

- Best algorithms in asymptotic sense are now linear-time (Karger, Pettie and Ramachandran).
- Fastest algorithm remains Prim's algorithm (Moret & Shapiro), which is over 50 years old in its general statement.
- Good priority queues are the key, but note that Prim's algorithm with simple binary heaps is hard to beat (pairing heaps are better, but not by a huge amount).

How Else Do We Get Speed?

- Algorithms with *significantly* improved asymptotic running time
- Faster machines (a $4K machine today runs at least 1,000 times faster than a $500K mainframe did 15 years ago).
- Parallel computing: if the application can be parallelized and scales well, k processors may run almost k times faster.
- Better code generation: modern optimizing compilers are even capable of limited optimization for cache utilization.

How Does Alg. Eng. Gain Speed?

- Optimize low-level data structures (multiple parallel arrays, maintenance vs. recomputation)
- Optimize low-level algorithmic details (e.g., upper and lower bounds)
- Optimize low-level coding (hand-unrolling loops, keeping local variables in registers)
- Reduce memory footprint (the entire code might fit in cache)
- Maximize locality of reference (the memory hierarchy can cause differentials of 100:1)

Working with the Memory Hierarchy

- Performance assessment for fast algorithms requires that caching be taken into account.
- Performance assessment for algorithms that work with large quantity of data requires that paging be taken into account.

Thus the *entire memory hierarchy* should be part of many experimental studies.

Working with the Memory Hierarchy (cont'd)

We need

- credible models of caching and paging
- analysis methods for the effects of caching (see Ladner and LaMarca)
- design methods for algorithms that use caching and paging to optimize their running time (see LaMarca, Mehlhorn, Vitter, Raman, etc.)

A Brief History

- *50s and 60s:* algorithmic work always accompanied by working code, but testing rare
- *70s and early 80s:* paper and pencil years, but Bentley starts his collection of priceless observations about efficient programming practices
- *mid 80s:* studies by Jones (priority queues, *CACM*), Stasko and Vitter (pairing heaps, *CACM*), Moret and Shapiro (min. test sets, *SIAM JSSC*); Bentley's *Programming Pearls*

A Brief History (cont'd)

- *early 90s:* seminal papers by Johnson's group, beginning of LEDA, calls in many venues by Goldberg, Johnson, Mehlhorn, Moret, Orlin, and others for more experimentation and implementation
- *mid 90s:* ACM starts the *ACM JEA*, ACM Symp. on Comput'l Geometry sets up applied tracks, G. Italiano organizes the first Workshop on Algorithm Engineering (Venice 1997)

Present State of Affairs:

- LEDA is a stable commercial product
- WAE and ALENEX run yearly (WAE is now a track at ESA)
- many new studies appear in various conferences
- memory models of special interest
- modest inroads in applications, esp. computational molecular biology

Algorithm Engineering Challenges I

How to measure performance?

- running time
- structural measures (counting pointer dereferences, memory accesses, cache misses, etc.)
- profiling (where and when is the time spent?)

Algorithm Engineering Challenges II

How to choose test sets?

- generated datasets pinpoint specifics of the program and are most helpful during development
- real application data tests performance where it matters, but are often hard to get
- datasets created from real application data through random perturbations or through probability estimates can prove useful in the testing phase

Algorithm Engineering Challenges III

How to assess measurements?

- data presentation (normalization, graphical tools)
- data analysis (statistical tools)
- data enhancement (e.g., variance reduction, McGeoch)

Guidelines for Experimental Setup

- Begin the work with a clear set of objectives: what are the questions to be answered?
- Design the experiments, gathering test data along the way to help improve the design; discard the data once used.
- Once the experimental design is complete, simply gather data.
- Analyze the data to answer only the original objectives. (Later, consider how a new cycle of experiments can improve your understanding.)

Confounding Factors

- Choice of machine, of language, of compiler
- Consistency and sophistication of programmers, effect of library layers
- Instance selection or generation; in applications, choice of model and model parameters
- Method for loading the datasets in memory
- Method of analysis

Common Pitfalls

Uninteresting Work:

- “I might as well get some numbers out that code I wrote”
- “My parallel code only runs on the Exotica-19.5, but porting it to a more common platform is too much trouble”
- “I bet it’d run faster on machine X, if written in D++, . . . ”

Common Pitfalls

Bad Setup:

- "My machine is slow or has little memory, so I'll just test up to some fixed running time (or space) or just test a few large instances"
- "I don't like (or have time for) coding, so I'll compare with whatever code I can find on the net"
- "I don't have time to look for existing testbeds, but I am running a lot of test cases"
- "I used my measurements to refine my heuristic parameters independently for each dataset"

Common Pitfalls

Bad Analysis or Presentation:

- “These data do not fit the pattern—must be experimental error—so I’ll ignore them”
- “Standards of science say I should give all my data, so here are some pages of numbers”
- “Here are comparisons between the running time of my new super-sort and that of standard radix sort”

Common Pitfalls

David Johnson has a well known list of over a hundred “pet peeves” that he has encountered.

At the *First Workshop on Algorithms in Bioinformatics (WABI’01)*, Alberto Caprara remarked that

- a theoretician solves interesting problems with no practical use whatsoever, whereas
- an experimental algorithmist solves problems of significant practical use, but only tests his solutions on randomly generated instances.

What to Measure?

- In studies of the quality of (approximate) solutions:
  - be sure to measure time in some structural way (e.g., number of iterations), to serve as scale for determination of rates
  - measure the rate of convergence to the final solution
  - whenever possible, determine the optimal solution (by brute force if necessary)

What to Measure?

- In comparative studies of algorithms for tractable problems:
  - *always* measure running time!
  - but also look for low-level structural measures (mems, comparisons, data moves)
  - and be sure to establish reference values for later normalization

How to Present and Analyze the Data

- Ensure reproducibility.
- Do not discard anomalies, but seek to explain them if at all possible.
- Use appropriate statistical measures.
- Minimize influence of platform, coding, caching, etc., through cross-checking and normalization.
- Do not present raw data when normalization is possible (ratio to optimal; ratio of running times to a baseline routine).

Recent Directions: Memory Models

The models are crucial: they must be credible, reflecting fundamental properties of hardware rather than pleasing simplifications.

- Models and analysis tools for the effect of caching (Ladner and LaMarca, others)
- Design of cache-efficient algorithms for the models developed (Raman, others)
- Models of secondary memory (Mehlhorn, Vitter, others)
- Design of algorithms to work in secondary memory (Mehlhorn, Vitter, others)

Recent Directions: Applications

Doing interesting research in experimental algorithmics while putting one's skills in the service of a natural science.

- Work with area experts to improve existing models and solutions. (For example, the best work done on memory models combines systems know-how with algorithmic sophistication.)
- Target solution algorithms at real data sets that require sophisticated solution techniques as well as model refinement.

Recent Directions: Man In The Loop

Many objective functions are ill-defined tradeoffs among separate objectives. Interaction with the user (visualization!) is crucial in such problems:

- the user may be able to give hints to the optimization program;
- the user may want to experiment with a variety of constraints;
- the user can be given a choice of various suboptimal solutions—with potential improvements in robustness.

Some Success Stories

- CPLEX, LEDA, and CGAL (although speed is not a primary objective in CGAL)
- Cache-aware experimental studies (Ladner, LaMarca, Raman, Sanders, Zhang, etc.) and cache-oblivious algorithm design (Bender, Frigo, etc.).
- Secondary memory algorithms and experiments (Arge, Mehlhorn, Vitter, etc.)

Some Success Stories

- Large instances of NP-hard problems solved quickly with fast implementations (TSP, Set Cover, others).
- Spectacular speedups over “everyday” code—over *six* orders of magnitude (Moret).
- Algorithm engineering has “registered” on the map of high-performance computing and of computational biology.