

Algorithm Engineering – An Attempt at a Definition Using Sorting as an Example*

Peter Sanders[†]

Abstract

The talk describes algorithm engineering (AE) as a methodology for algorithmic research where design, analysis, implementation and experimental evaluation of form a feedback cycle driving the development of efficient algorithm. Additional important components of the methodology include realistic models, algorithm libraries, and collections of realistic benchmark instances. Examples are given for the fundamental problem of sorting with particular emphasis on huge data sets, advanced hardware, and energy efficiency.

1 Introduction

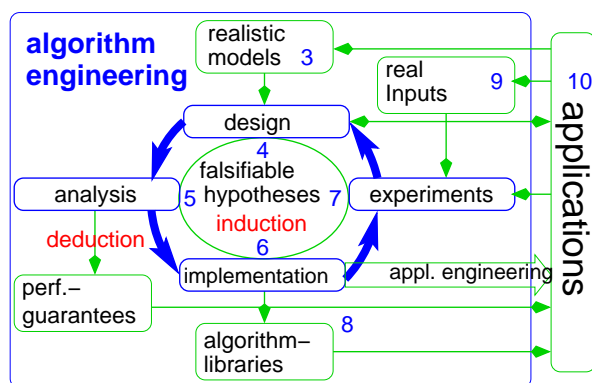


Figure 1: Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses. The numbers refer to sections.

Algorithms and data structures are at the heart of every computer application and thus of decisive importance for permanently growing areas of engineering, economy, science, and daily life. The subject of *Algorithmics* is the systematic development of efficient algorithms and therefore has pivotal influence on the effective development of reliable and resource-conserving

technology. We only mention search engines, bioinformatics, computer graphics, image processing, geographic information systems, cryptography, planning in production, logistics and transportation as example areas where algorithms play a key role.

How is algorithmic innovation transferred to applications? Traditionally, algorithmics used the methodology of *algorithm theory* which stems from mathematics: algorithms are designed using simple models of problem and machine. Main results are provable performance guarantees for all possible inputs. This approach often leads to elegant, timeless solutions that can be adapted to many applications. The hard performance guarantees lead to reliably high efficiency even for types of inputs that were unknown at implementation time. From the point of view of algorithm theory, taking up and implementing an algorithmic idea is part of application development. Unfortunately, it can be universally observed that this mode of transferring results is a slow process. With growing requirements for innovative algorithms, this causes widening gaps between theory and practice: Realistic hardware with its parallelism, memory hierarchies etc. is diverging from traditional machine models. Applications become more and more complex. At the same time, algorithm theory develops increasingly elaborate algorithms that may contain important ideas but are usually not directly implementable. Furthermore, real-world inputs are often far away from the worst case scenarios of the theoretical analysis. In extreme cases, promising algorithmic approaches are neglected because a mathematical analysis would be difficult.

Since the early 1990s it therefore became more and more apparent that algorithmics cannot restrict itself to theory. So, what else should algorithmicists do? *Experiments* play a pivotal here. Algorithm engineering (AE) is therefore sometimes equated with *experimental algorithmics*. However, in this paper we argue that this view is too limited. First of all, to do experiments, you also have to *implement* algorithms. This is often equally interesting and revealing as the experiments themselves, needs its own set of techniques, and is an important interface to software engineering. Furthermore, it makes

*Partially supported by DFG grant SA 933/3-2.

[†]Karlsruher Institut für Technologie, 76128 Karlsruhe, Germany sanders@kit.edu

little sense to view design and analysis on the one hand and implementation and experimentation on the other hand as separate activities. Rather, a feedback loop of design, analysis, implementation, and experimentation that leads to new design ideas materializes as the central process of algorithmics.

This cycle is quite similar to the cycle of theory building and experimental validation in Popper's scientific method [23]. We can learn several things from this comparison. First, this cycle is driven by *falsifiable hypotheses* validated by experiments – an experiment cannot prove a hypothesis but it can support it. However, such support is only meaningful if there are conceivable outcomes of experiments that prove the hypothesis wrong. Hypotheses can come from creative ideas or result from *inductive reasoning* stemming from previous experiments. Thus we see a fundamental difference to the *deductive reasoning* predominant in algorithm theory. Experiments have to be *reproducible*, i.e., other researchers have to be able to repeat an experiment to the extent that they draw the same conclusions or uncover mistakes in the previous experimental setup.

There are further aspects of AE as a methodology for algorithmics, outside the main cycle. Design, analysis and evaluation of algorithms are based on some *model* of the problem and the underlying machine. Since gaps between theory and practice often relate to these models, they are an important aspect of AE. Since we aim at practicality, *applications* are an important aspect. However we choose to view applications as being outside the methodology of AE since it would otherwise become too open ended and because often one algorithm can be used for quite diverse applications. Also, every new application will have its own requirements and techniques some of which may be abstracted away for algorithmic treatment. Still, in order to reduce gaps between theory and practice, as many interactions as possible between the application and the activities of AE should be taken into account: Applications are the basis for *realistic* models, they influence the kind of analysis we do, they put constraints on useful implementations, and they supply *realistic inputs* and other design parameters for experiments. On the other hand, the results of analysis and experiments influence the way an algorithm is used (fast enough for real time or interactive use?, . . .) and implementations may be the basis for software used in applications. Indeed, we may view *application engineering* as a separate process living in both AE and a concrete application domain where methods from both areas are used to adapt an algorithm to a particular application. Application engineering bridges remaining unavoidable gaps between experimental implementations and production quality code. Note that

there are important differences between these two kinds of code: fast development, efficiency, and instrumentation for experiments are very important for AE, while thorough testing, maintainability, simplicity, and tuning for particular classes of inputs are more important for the applications. Furthermore, the algorithm engineers may not even know all the applications for which their algorithms will be used. Hence, *algorithm libraries* of highly tested codes with clear simple user interfaces are an important link between AE and applications.

Figure 1 summarizes the resulting schema for AE as a methodology for algorithmics. The following sections will describe the activities in more detail. We give examples of challenges and results that are a more or less random sample biased to results we know well. In addition, we will use our work on sorting as a more concrete example. This example was chosen because it is a simple, fundamental algorithmic problem and already illuminates a lot of issues.

This paper is a shortened version of [26] where minimum spanning trees were used as an example. Please refer to this paper and the citations therein for more details. We will also keep the sorting example brief and refer to further papers for more details.

2 A Brief “History” of Algorithm Engineering

The methodology described here is not intended as a revolution but as a description of observed practices in algorithmic research being compiled into a consistent methodology. Basically, all the activities in algorithm development described here have probably been used as long as there are computers. However, in the 1970s and 1980s algorithm theory had become a subdiscipline of computer science that was almost exclusively devoted to “paper and pencil” work. Except for a few papers around D. Johnson, the other activities were mostly visible in application papers, in operations research, or J. Bentley's programming pearls column in *Communications of the ACM*. In the late 1980s, people within algorithm theory began to notice increasing gaps between theory and practice leading to important activities such as the Library of Efficient Data Types and Algorithms (LEDA, since 1988) by K. Mehlhorn and S. Näher and the DIMACS implementation challenges (<http://dimacs.rutgers.edu/Challenges/>). It was not before the end of the 1990s that several workshops series on experimental algorithmics and algorithm engineering were started.¹ There was a Dagstuhl workshop

¹The Workshop on Algorithm Engineering (WAE) is not the engineering track of ESA. The Alex workshop first held in Italy in 1998 is now the ALENEX workshop held in conjunction with SODA. WEA, now SEA was first organized in 2002.

in 2000 [10], and several overview papers on the subject were published [1, 21, 17, 18, 13].

The term “algorithm engineering” already appears 1986 in the Foreword of [3] and 1989 in the title of [30]. No discussion of the term is given. At the same time T. Beth started an initiative to move the CS department of the University of Karlsruhe more into the direction of an engineering discipline. For example, a new compulsory graduate-level course on algorithms was called “Algorithmentechnik” which can be translated as “algorithm engineering”. Note that the term “engineering” like in “mechanical engineering” means the *application* oriented use of science whereas our current interpretation of algorithm engineering has applications not as its sole objective but equally strives for general scientific insight as in the natural sciences. However, in daily work the difference will not matter much.

P. Italiano organized the “Workshop on Algorithm Engineering” in 1997 and also uses “algorithm engineering” as the title for the algorithms column of EATCS in 2003 [9] with the following short abstract: “Algorithm Engineering is concerned with the design, analysis, implementation, tuning, debugging and experimental evaluation of computer programs for solving algorithmic problems. It provides methodologies and tools for developing and engineering efficient algorithmic codes and aims at integrating and reinforcing traditional theoretical approaches for the design and analysis of algorithms and data structures.” Independently but with the same basic meaning, the term was used in the influential policy paper [1]. The present paper basically follows the same line of argumentation attempting to work out the methodology in more detail and providing a number of hopefully interesting examples.

3 Models

A big difficulty for defining models for problems and machines is that (apparently) only complex models are adequate images of reality whereas only simple models lead to simple, widely usable, portable, and analyzable algorithms. Therefore, AE must simultaneously and carefully abstract from reality and refine theoretical models.

Even such simple problem as sorting has a few interesting modelling issues. In particular, for comparison based algorithms there is an $\Omega(n \log n)$ lower bound whereas elements with short integer keys can be sorted in linear time using radix sort.

In our own work we have used sorting as a guinea pig for learning about algorithmic aspects of machine models for realistic hardware. Such aspects include cache efficiency [22], translation lookaside buffers [24],

limited associativity of caches [20], cache obliviousness [11, 4], (parallel disk) external memory [8, 12], massively parallel external [25], shared memory [31], graphics processors [16], branch mispredictions [28, 14], and energy efficiency. Since these aspects can have a considerable impact on performance, we will use these sorting algorithms for these models as our main pool of examples in the following.

4 Design

As in algorithm theory, we are interested in efficient algorithms. However, in AE, it is equally important to look for simplicity, implementability, and possibilities for code reuse. Furthermore, efficiency means not just asymptotic worst case efficiency, but we also have to look at the constant factors involved and at the performance for real-world inputs. In particular, some theoretically efficient algorithms have similar best case and worse case behavior whereas the algorithms used in practice perform much better on all but contrived examples.

There are many examples of gaps between theory and practice for sorting. For example, Cole’s ingenious PRAM algorithm for sorting [5] has large constant factors not only in the algorithm itself, but more importantly in the translation from the theoretical machine model to realistic hardware. We have found that surprisingly often, long known algorithms like quicksort and mergesort can be adapted to advanced models. In particular, their multiway variants samplesort and multiway mergesort often are the basis for the best practical methods [22, 4, 8, 12, 25, 28]. However, there remain interesting algorithmic questions in subroutines like disk scheduling [12], load balancing [25] or partitioning the input [32].

5 Analysis

Even simple and proven practical algorithms are often difficult to analyze and this is one of the main reasons for gaps between theory and practice. Thus, the analysis of such algorithms is an important aspect of AE. For example, randomized algorithms are often simpler and faster than their best deterministic competitors but even simple randomized algorithms are often difficult to analyze.

Randomized sorting algorithms yield many interesting examples. For example, our parallel disk and parallel external sorting algorithms [12, 25] rely heavily on randomization and their analysis is not easy because of dependencies between random variables. For example, the result in [12] gives a partial answer to an exercise in Knuth’s famous text book [15] that is given difficulty 48 – one of the most difficult questions in the book. The

original question still remains open.

Sorting is also a good example that algorithm analysis is often neglected in papers from practical computer science. In particular, algorithms proposed for sorting huge data sets in parallel or using external memory can have very bad performance for worst case inputs. Our results in [8, 12, 25] show that giving worst case guarantees needs some care but need not inhibit practical performance.

Many complex optimization problems are attacked using *meta heuristics* like (randomized) local search or evolutionary algorithms. Algorithms of this type are simple and easily adaptable to the problem at hand. However, only very few such algorithms have been successfully analyzed (e.g. [33]) although performance guarantees would be of great theoretical and practical value.

6 Implementation

Implementation only appears to be the most clearly prescribed and boring activity in the cycle of AE. One reason is that there are huge semantic gaps between abstractly formulated algorithms, imperative programming languages, and real hardware. An extreme example for the semantic gap are geometric algorithms which are often designed assuming exact arithmetics with real numbers and without considering degenerate cases.

Even the implementation of relatively simple basic algorithms can be challenging. You often have to compare several candidates based on small constant factors in their execution time. Since even small implementation details can make a big difference, the only reliable way is to highly tune all competitors and run them on several architectures. It can even be advisable to compare the generated machine code sorting is a very good example here (e.g., [28], [4]).

7 Experiments

Meaningful experiments are the key to closing the cycle of the AE process. They can validate or falsify existing hypotheses and their results can motivate new hypotheses that lead to new algorithms, improved analysis or better implementations.

Compared to the natural sciences, AE is in the privileged situation that it can perform many experiments with relatively little effort. However, the other side of the coin is highly nontrivial planning, evaluation, archiving, postprocessing, and interpretation of results. The starting point should always be falsifiable hypotheses on the behavior of the investigated algorithms which stem from the design, analysis, implementation, or from previous experiments. The result is a confirmation, falsification, or refinement of the hypothesis. The results

complement the analytic performance guarantees, lead to a better understanding of the algorithms, and provide ideas for improved algorithms, more accurate analysis, or more efficient implementation.

Successful experimentation involves a lot of software engineering. Modular implementations allow flexible experiments. Clever use of tools simplifies the evaluation. Careful documentation and version management help with reproducibility – a central requirement of scientific experiments, that is challenging due to the frequent new versions of software and hardware.

Experiments on sorting algorithms can vary the machine architecture, the number of processors, the memory devices (e.g., disks, SSDs) used on various levels of the memory hierarchy, the element size, the key size, the comparison function, and perhaps most importantly, the input permutation. In a high level article like this one, or for a reviewer, it is easy to say that one should vary *all* parameters systematically. But in the real world, even in its most academic enclaves, this is clearly impossible. For example, in [8, 25] the sheer running times of the experiments (summing to days and weeks) were a major limiting factor for keeping submission deadlines.

So how can the big space of possible experiments be pruned? One powerful tool is theory. For example, sample sort (e.g. [28]) provably works similarly for all possible inputs. Hence, using random input permutations is sufficient.² In contrast, when such guarantees are not proven, working with a large spectrum of possible inputs is very important (e.g. [16]). Rather than blindly measuring execution time on many different machine configurations, one can also evaluate performance parameters like cache faults, branch mispredictions, I/O cost, etc. that allow predictions about the impact of various architectural parameters and about the influence of element size and comparison function.

8 Algorithm Libraries

Algorithm libraries are made by assembling implementations of a number of algorithms using the methods of software engineering. The result should be efficient, easy to use, well documented, and portable. Algorithm libraries accelerate the transfer of know-how into applications. Within algorithmics, libraries simplify comparisons of algorithms and the construction of software that builds on them. The software engineering involved is particularly challenging, since the applications to be

²Actually, the implementation used in [28] can be fooled with many identical keys. However, it can be argued that a slightly more sophisticated implementation would fix that problem without incurring performance penalties.

supported are *unknown* at library implementation time and because the separation of interface and (often highly complicated) implementation is very important. Compared to applications-specific reimplementations, using a library should save development time without leading to inferior performance. Compared to simple, easy to implement algorithms, libraries should improve performance. In particular for basic data structures with their fine-grained coupling between applications and library this can be very difficult. To summarize, the triangle between generality, efficiency, and ease of use leads to challenging tradeoffs because often optimizing one of these aspects will deteriorate the others. It is also worth mentioning that *correctness* of algorithm libraries is even more important than for other software because it is extremely difficult for a user to debug library code that has not been written by his team. Sometimes it is not even sufficient for a library to be correct as long as the user does not *trust* it sufficiently to first look for bugs outside the library. This is one reason why result checking, certifying algorithms, or even formal verification are an important aspect of algorithm libraries. All these difficulties imply that implementing algorithms for use in a library is several times more difficult / expensive / time consuming / frustrating / ... than implementations for experimental evaluation. On the other hand, a good library implementation might be *used* orders of magnitude more frequently. Thus, in AE there is a natural mechanism leading to many exploratory implementations and a few selected library codes that build on previous experimental experience.

Let us now look at a few successful examples of algorithm libraries. The Library of Efficient Data Types and Algorithms LEDA [19] has played an important part in the development of AE. LEDA has an easy to use object-oriented C++ interfaces. Besides basic algorithms and data structures, LEDA offers a variety of graph algorithms and geometric algorithms.

Programming languages come with a run-time library that usually offers a few algorithmic ingredients like sorting and various collection data structures (lists, queues, sets, ...). For example, the C++ standard template library (STL) has a very flexible interface based on templates. Since so many things are resolved at compile time, programs that use the STL are often equally efficient as hand-written C-style code even with the very fine-grained interfaces of collection classes.

This is one of the reasons why our group is looking at implementations of the STL for advanced models of computation like external computing (STXXL [7]), multicore parallelism (MCSTL, GNU C++ standard library [29]), or a combination of both [2]. The support of fast sorting routines is perhaps the main reason why

our STL implementations are used – they offer fast implementations for a frequently used and performance critical function that is also easy to use.

9 Instances and Benchmarks

Collections of realistic problem instances for benchmarking have proven crucial for improving algorithms. This was very successful for some NP-hard problems like travelling salesman, Steiner trees, satisfiability, set covering, or graph partitioning. It is a bit odd that similar benchmarks for problems that are polynomially solvable are sometimes more difficult to obtain. For route planning in road networks, realistic inputs have become available in 2005 [27] enabling a revolution with speedups of up to six orders of magnitude over Dijkstra's algorithm and a perspective for many applications [6]. In string algorithms and data compression, real-world data is also no problem. But for many typical graph problems like flows, random inputs are still common practice. We suspect that this often leads to unrealistic results in experimental studies. Naively generated random instances are likely to be either much easier or more difficult than realistic inputs. With more care and competition, such as for the DIMACS implementation challenges, generators emerge that drive naive algorithms into bad performance. While this process can lead to robust solutions, it may overemphasize difficult inputs.

For sorting, there is a well established benchmark from the database community (<http://sortbenchmark.org/>) that is an interesting focus point for research on sorting large data sets stored in a file system. We have now participated in all its main categories³: Penny sort asks for the amount of data that can be sorted for one US cent assuming the cost of the machines if depreciated linearly over three years. STXXL [7] provides a code [8, 2] that comes close to the fastest specialized codes. Our parallel external sorter [25] currently leads the MinuteSort benchmark which asks for the largest amount of data sorted in one minute and the GraySort benchmark that asks for sorting 100 Terabyte of data. Interestingly, the jury decided to give the prize to us although a competing system based on Hadoop is slightly faster than our system – but we need more than an order of magnitude less hardware. Based on both above programs, we have developed a sorter for the youngest category of the SortingBenchmark – JouleSort, which asks for sorting certain input sizes using as little energy as possible. We

³All our entries are in the fixed width *Indy* subcategories whereas *Daytona* asks for more flexible programs that can handle variable length elements and key.

have now submitted results that improve the current records by a factor of 3–4. The main improvement here is due to choosing the right hardware – an Intel Atom processors and solid state disks. All categories of SortingBenchmark ask for sorting uniformly distributed 100 byte elements with 10 byte keys. Unfortunately, this allows entries that will perform catastrophically bad on nonrandom inputs. Many academic papers are more careful with the inputs considered and use a variety of synthetic and real world inputs.

10 Applications

We could discuss many important applications where algorithms play a major role and a lot of interesting work remains to be done. Since this would go beyond the scope of this paper, we only want to mention a few: Bioinformatics (e.g. sequencing, folding, docking, phylogenetic trees, DNA chip evaluations, reaction networks); information retrieval (indexing, ranking); algorithmic game theory; traffic information, simulation and planning for cars, buses, trains, and air traffic; geographic information systems; communication networks; machine learning; real time scheduling.

The effort for implementing algorithms for a particular application usually lies somewhere between the effort for experimental evaluation and for algorithm libraries depending on the context.

An important goal for AE should be to help shaping the applications rather than act as an ancillary science for other disciplines like physics, biology, mechanical engineering,...

11 Conclusions

We hope to have demonstrated that AE is a “round” methodology for the development of efficient algorithms which simplifies their practical use. We want to stress, however, that it is not our intention to abolish algorithm theory. The saying that “there is nothing as practical as good theory” remains true for algorithmics because an algorithm with proven performance guarantees has a degree of generality, reliability, and predictability that cannot be obtained with any number of experiments. However, this does not contradict the proposed methodology since it views algorithm theory as a subset of AE, making it even more rich by asking additional interesting kinds of questions (e.g. simplicity of algorithms, care for constant factors, smoothed analysis,...). We also have no intention of criticizing some highly interesting research in algorithm theory that is less motivated from applications than by fundamental questions of theoretical computer science such as computability or complexity theory. However, we do want to criticize those papers that begin with a vague claim of relevance

to some fashionable application area before diving deep into theoretical constructions that look completely irrelevant for the claimed application. Often this is not intentionally misleading but more like a game of “Chinese whispers” where a research area starts as a sensible abstraction of an application area but then develops a life of itself, mutating into a mathematical game with its own rules. Even this can be interesting but researchers should constantly ask themselves why they are working on an area, whether there are perhaps other areas where they can have larger impact on the world, and how false claims for practicality can damage the reputation of algorithmics in practical computer science.

Acknowledgements

I would like to thank the coiniciators of the DFG SPP 1307, Kurt Mehlhorn, Rolf Möhring, Burkhard Monien, and Petra Mutzel for their advice and fruitful discussions that led to the definition presented here. Discussions with many other colleagues, in particular with Rudolf Fleischer have helped to shape my view of algorithm engineering. The sorting results outlined here have been obtained in cooperation with Andreas Beckmann, Roman Dementiev, David Hutchinson, Kanella Kaligosi, Nikolaj Leischner, Kurt Mehlhorn, Ulrich Meyer, Vitaly Osipov, Mirko Rahn, Johannes Singler, Jeff Vitter, and Sebastian Winkel.

References

- [1] A. V. Aho, D. S. Johnson, R. M. Karp, S. Rao Kosaraju, C. C. McGeoch, C. H. Papadimitriou, and P. Pevzner. Emerging opportunities for theoretical computer science. *SIGACT News*, 28(3):65–74, 1997.
- [2] A. Beckmann, R. Dementiev, and J. Singler. Building a parallel pipelined external memory algorithm library. In *23rd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.
- [3] T. Beth and M. Clausen, editors. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 4th International Conference, AAEECC-4, Karlsruhe, FRG, September 23-26, 1986, Proceedings*, volume 307 of *Lecture Notes in Computer Science*. Springer, 1988.
- [4] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.
- [5] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.
- [6] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS State-of-the-Art Survey*, pages 117–139. Springer, 2009.
- [7] R. Dementiev, L. Kettner, and P. Sanders. STXXL:

- Standard Template Library for XXL data sets. *Software Practice & Experience*, 38(6):589–637, 2008.
- [8] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.
- [9] C. Demetrescu, I. Finocchi, G. F., and Italiano. Algorithm engineering. *Bulletin of the EATCS*, 79:48–63, 2003.
- [10] R. Fleischer, B. Moret, and E. Meineche Schmidt, editors. *Experimental Algorithmics From Algorithm Design to Robust and Efficient Software*, volume 2547 of *LNCS*. Springer, 2002.
- [11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.
- [12] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. *SIAM Journal on Computing*, 34(6):1443–1463, 2005.
- [13] D. S. Johnson. A theoretician’s guide to the experimental analysis of algorithms. In M. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, 2002.
- [14] K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *14th European Symposium on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 780–791, 2006.
- [15] D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
- [16] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *24th IEEE International Parallel and Distributed Processing Symposium*, 2010. accepted, also arXiv:0909.5649.
- [17] C. C. McGeoch, D. Precup, and P. R. Cohen. How to find big-oh in your data set (and how not to). In *Advances in Intelligent Data Analysis*, number 1280 in *LNCS*, pages 41–52, 1997.
- [18] C.C. McGeoch and B. M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.
- [19] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [20] K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.
- [21] B. M. E. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge*, DIMACS Monograph Series, 2000. to appear.
- [22] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD*, pages 233–242, 1994.
- [23] K. R. Popper. *Logik der Forschung*. Springer, 1934. English Translation: *The Logic of Scientific Discovery*, Hutchinson, 1959.
- [24] N. Rahman. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Algorithms for Hardware Caches and TLB, pages 171–192. Springer, 2003.
- [25] M. Rahn, P. Sanders, and J. Singler. Scalable distributed-memory external sorting. In *26th IEEE International Conference on Data Engineering*, 2010. to appear, also at <http://arxiv.org/abs/0910.2582v1>.
- [26] P. Sanders. Algorithm engineering - an attempt at a definition. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009.
- [27] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 568–597. Springer, 2005.
- [28] P. Sanders and S. Winkel. Super scalar sample sort. In *12th European Symposium on Algorithms*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004.
- [29] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007.
- [30] D. Gollman T. Beth. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, 1989.
- [31] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *PDP*, pages 372–381. IEEE Computer Society, 2003.
- [32] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *J. Par. & Dist. Comp.*, 12(2):171–177, 1991.
- [33] I. Wegener. Simulated annealing beats metropolis in combinatorial optimization. In *32nd International Colloquium on Automata, Languages and Programming*, pages 589–601, 2005.