Algorithm Engineering for the Basic Toolbox

Peter Sanders
Fakultät für Informatik
Universität Karlsruhe
76128 Karlsruhe

June 4, 2007

Excerpt from proposal for DFG Priority Programme 1307 — Algorithm Engineering
Contents

1 New Proposal for a Research Grant ........................................ 1
  1.1 Topic ...................................................................... 1
  1.2 Field and Area (Fachgebiet und Arbeitsrichtung) ............... 1
  1.3 Summary (Zusammenfassung) ..................................... 1

2 Previous Work ................................................................. 1
  2.1 State of Research (Stand der Forschung) ......................... 1
  2.2 Own Previous Work (Eigene Vorarbeiten) ....................... 6
1 New Proposal for a Research Grant

1.1 Topic
Engineering efficient algorithms for the basic algorithmic toolbox with emphasis on algorithm libraries, memory hierarchies and parallelism.

1.2 Field and Area (Fachgebiet und Arbeitsrichtung)
computer science, algorithmics

1.3 Summary (Zusammenfassung)
This project addresses algorithm engineering for basic algorithms and data structures that are the most important building blocks for many computer applications — sorting, searching, graph traversal, . . . . Although this topic is as old as computers science itself, many interesting new results have appeared in recent years and many gaps between theory and practice remain. In particular, many interesting approaches have not been thoroughly tried experimentally. Ever more complex hardware with memory hierarchies and several types of parallel processing requires refined models, new algorithms, and efficient implementations. We plan to incorporate the most successful implementations into reusable software libraries such as the the C++ STL.

2 Previous Work
Since the basic toolbox is a wide area, it is impossible to give a complete review of previous work. The presented selection aims at demonstrating that there are many interesting open problems. We also give some background for subareas we plan to work on.

2.1 State of Research (Stand der Forschung)
Sometimes, previous work bearing direct relations to our own past or planned research is mentioned in the later sections to simplify reading and to reduce redundancy.
2.1.1 Models

The RAM model remains a simple and useful model for sequential programs with good memory locality. Disk accesses are successfully modelled by transfers of blocks of size $B$ between secondary memory and a fast memory of size $M$ [AV88, VS94a]. This I/O-model is also useful between main memory and one cache level. By hiding the parameters $B$ and $M$ from the algorithm, we arrive at the cache-oblivious model [FLPR99]. cache-oblivious algorithms work equally well on all levels of a memory hierarchy. However, there are only few implementations of cache oblivious algorithms that yield good performance in practice (e.g., [BFV04]). Parallel models are more problematic. The PRAM model is largely regarded as unrealistic and there is no generally accepted model for asynchronous shared memory machines. BSP [Val94] is a realistic model for distributed memory machines but bases all communication on an expensive and inflexible globally synchronized data exchange. No model is in sight that adequately models multiple levels of caches and local memories, that are shared between some but not all processors. There are several attempts but most of them are too abstract or too complicated. For example, [VS94b] is too complicated although it ignores communication costs.

More detailed and more specialized models are needed for runtime prediction. There has been very little work outside the the area of numeric computing (e.g. [FM97]).

2.1.2 Sequence Representation

State of the art for RAM algorithms are (resizable) arrays and linked list. A folklore result is that for cache efficiency, a hybrid between these approaches if often better. But only very few implementations actually implement this (e.g. [FPR06]). In Section 2.1.10 we outline challenges in the context of concurrent access to sequences.

2.1.3 Hashing

Hashing with chaining and linear probing are still the algorithms most widely used in practice. It remains an interesting question to what extent perfect hashing [DKM+94] or other ways to achieve worst case constant access time will turn out to be useful in practice. For example, cuckoo hashing and its relatives [PR01, DW05] look elegant, simple and efficient on the first glance and experiments look good. However, cuckoo insertion is only guaranteed to terminate for hash functions with certain performance guarantees that are likely to be too expensive in practice. More generally, there seems to be a relatively little work yet on hash functions that are both practical and give good performance guarantees [DadH90, Tho00].
2.1.4 Sorting and Selection

Sorting is perhaps the single most intensively studied algorithmic problem because it is at the same time simple, nontrivial, and immensely useful. The sorting literature shows the typical gap between theory and practice. For example, there is an “optimal” deterministic algorithm for parallel disk sorting [NV95] but its constant factors are so big that “suboptimal” implementations in data base papers outclass it for all practical situations. In some advanced models of computation, there is not even an asymptotically optimal algorithm published.

For internal memory sequential sorting, there has been considerable recent work on cache-efficient sorting, e.g., [RKU00, WACV02].

Theoretical algorithms for integer sorting have made impressive progress [YH02] but none of them is practical and even radix sort for small keys needs very careful (cache-efficient) implementation to beat quicksort (e.g. [GNLP01, Rah03]).

2.1.5 Priority Queues

There are a number of very interesting algorithm engineering results on priority queues: one of the first systematic algorithm engineering papers [Jon86]; a DI-MACS implementation challenge; studies of priority queues together with MST [MS94] and shortest path [CGR96, Gol01] calculations. Some theoretical results like pairing heaps [FSST86] are inspired by the idea of simplifying theoretically optimal algorithms like Fibonacci heaps [FT84]. However, this has resulted in a zoo of variants (e.g. [Høy95] without conclusive experimental comparisons. The only experimental paper we are aware of makes simulations with variants of pairing heaps [SV87] using synthetic inputs but gives little insight on performance of real implementations with real inputs.

There is a gap between theoretical [Tho04] and practical integer priority queues [Bro88, ELL94, AT96] analogous to the one in integer sorting.

There are several interesting results on parallel priority queues (e.g. [Ran94, DPS96, BTZ98]) where it is not clear whether they are practical.

2.1.6 Sorted Sequences

Efficient operations on sorted sequences can be implemented using search tree (e.g. [AVL62]) and skip-list [Pug90] data structures. There are many variants. Experimental studies with sequential, internal memory data structures [MN99] indicate that performance differences are more dependent on implementation details than on algorithmic differences.
There is also a lot of work on search trees in memory hierarchies. There is a lot of implementation work for B-Trees [BM72] in data bases [Com79, GL01, Pag03]. There are theoretical results for string B-Trees [FG99] that contain very interesting ideas but do not look practical yet. As for sorting and priority queues, there is comparably little work on algorithm engineering for sorted sequences with integer keys [vEB77, MN90].

2.1.7 Graphs

While textbooks still only teach the difference between adjacency matrices and adjacency lists, it is now well known that adjacency arrays are the most efficient practical representation of graphs for most algorithms [NZ02]. There is also some work on clustering graphs such that graph traversal is accelerated [MZ03]. However, these techniques may not be fully developed yet.

Relatively little work has been done on systematic engineering of the most basic graph traversal algorithms breadth-first-search and depth-first-search. The reason may be that this looks trivial. However, even here nontrivial algorithmic choices are to be made.

There is so much work on algorithm engineering for shortest paths that we abstain from looking at it. We also omit more complex graph algorithms like maximum flow.

2.1.8 Strings

Algorithm engineering for string processing is highly developed but still suffers from gaps between theory and practice. Section 2.2.9 gives an example for the problem of suffix sorting.

2.1.9 Algorithm Libraries

Numerical algorithm libraries are routinely used since the early 1970s. Progress was much slower for nonnumerical algorithms perhaps because without generic typing, algorithm libraries remain awkward. For example, the C standard library always had a quick sort, but it is slow and rarely used because comparison relies on a pointer to a comparison function.

Object oriented programming languages like Smalltalk, C++ or Java now have algorithm libraries as part of their standard. In particular, collection classes for sets and sequences are used extensively.

More advanced algorithms and data structures are available in LEDA [MN99], JDSL [MN99], or Boost [Boo]. These libraries are widely used, yet perhaps not
as widely as one should think. One problem is performance. For example, programs with a handwritten graph data structure are often several times faster than comparable programs using a library.

2.1.10 Parallel Processing

We will discuss parallel algorithms for fundamental problems in the sections for these problems. However, the basic toolbox for writing parallel programs contains additional ingredients that have their own algorithmic challenges:

Communication. Processors of distributed memory machines interact using a set of frequently occurring communication patterns that are therefore part of message passing libraries such as MPI [GLS95]. In particular, the collective communication operations broadcast, reduction, prefix sum, gossipping, all-to-all, etc. are algorithmically interesting.

Shared Data Structures. Threads in a shared memory machine interact by concurrent access to memory. Synchronizing these accesses is a nontrivial requirement for obtaining correct programs. Currently, synchronization is mostly done via low-level primitives such as locks and semaphores that have limited scalability. Efficient synchronization uses even lower level atomic operations provided by the hardware, such as compare-and-swap and fetch-and-increment. Data structures for memory management, queues, hash tables, search trees, ... implemented using these atomic operations promise to become scalable, high level tools for writing shared memory programs. Interesting algorithms for maintaining stacks and FIFOs on shared memory machines have been developed and evaluated experimentally (e.g. [ALS94, HLM03, DHLM04, DHLM04, LH04, LAKL04, TZ03]). However, this is only slowly adapted into widely available algorithm libraries. For example, even the “Intel Threading Building Blocks” library\(^1\) which claims to give efficient implementations of parallel programming primitives, currently has a quite unsatisfactory FIFO.

Scheduling and Load Balancing: Keeping all processors busy with relevant work without incurring too many expensive interactions is a crucial problem of parallelization. A small set of load balancing algorithms can be used for a large spectrum of applications. This problem has been intensively studied both in theory and practice. However, there are several open problems: scheduling for low energy consumption, coping with ever more complex parallel memory hierarchies, and reusable libraries.

\(^1\)http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm
2.2 Own Previous Work (Eigene Vorarbeiten)

2.2.1 Methodology

We have done work on presenting experimental data [San02a], how (or whether) to infer asymptotic behavior from experiments [SF00, MSF+02], and on parallel algorithm engineering [BMS02].

2.2.2 Models

In [San94b] we have given the strongest known result how single instruction multiple data (SIMD) parallel machines can emulate multiple instruction multiple data (MIMD) machines. Interestingly, the SIMD model has recently been “rediscovered” for the Clearspeed coprocessors which have 96 SIMD processors [Tec06]. Another older result that is increasingly becoming relevant is a paper that analyzes how energy consumption bounds the scalability of parallel machines [SVW97]. A particular focus has been on realistic models for storage arrays [SEK00, SEK03, San04b, San02b, CSW07] and external memory [San03a]. The main result here is that using randomization, redundancy, and sophisticated scheduling algorithms, one can hide the complexity of such systems by emulating a much more powerful model that allows concurrent access to arbitrary data blocks. The influence of the limited associativity of hardware caches is considered in [San99a, MS03a]. Surveys on the I/O model are given in [MSS03, San04a].

2.2.3 Sequence Representation

The hybrid representation mentioned in Section 2.1.2 is used in [KSB05] to design a space efficient linear time suffix array construction algorithm. In [DSSS04], this kind of data structure (in internal memory!) is important for efficient external memory minimum spanning trees. We also have an unpublished space efficient implementation of the priority queue data structure [San00b].

2.2.4 Hashing

Several new and practical hash functions with provable performance guarantees are given in [MS07]. A practical and space efficient data structure for dynamic hashing with worst case constant access time is given in [FPSS03, FPSS05]. Closely related space efficient static hash tables are used in [BFM+07] for fast shortest path queries and in [ST07] for representing inverted indices. This is a good example for our claim that there are enough crosscutting issues in algorithm engineering to justify a quite wide project area.
2.2.5 Sorting, Selection, and Permuting

Sorting and related problems has been one of our main areas of research. The first parallel implementation of quicksort that works efficiently on massively parallel computers is given in [SH97]. On grid connected machines it might be the best practical algorithm. We also have one of the fastest practical external memory sorters [DS03, DKS05] that in contrast to other practical implementations also has good theoretical performance guarantees [HSV01, HSV05].

Even for internal memory, sequential sorting we managed to obtain some interesting results. For example, even highly tuned cache-efficient sorting algorithms such as [RKU00, WACV02, BFV04] yield only about 20% speedup compared to standard quicksort [Hoa61, Mus97], although they incur several times less cache faults. The reason is that quicksort is “sufficiently” cache-efficient so that another bottleneck can manifest itself—branch mispredictions. Indeed, we have shown [KS06a] that the seemingly paradoxical measure to choose pivots that are far from the median can slightly accelerate quicksort because a significantly reduced number of branch mispredictions more than outweighs the increased number of executed instructions and memory accesses. On the first glance, there seems to be no way out of the dilemma between too many executed instructions and too many branch mispredictions for comparison based sorting—the information theoretic lower bound of \(\log n!\) comparisons can only be obtained if the outcome of the comparisons is completely unpredictable. However, in [SW04] we could show that comparisons can be completely decoupled from branch instructions altogether using predicated instructions. Since the same measure reduces data dependencies, we can also improve instruction parallelism. Finally, the method is very cache-efficient. Only combining all these architectural effects, it was possible to obtain a factor up to two speedup compared to an efficient implementation of quicksort.

The recursion of quicksort is also a practical way to solve the more general problem of multiple selection where we only ask for a subset of the input elements with specified ranks [Cha71]. Interestingly, despite of intensive work on multiple selection, the exact analysis of this algorithm was open for 35 years. We have shown that it can be made optimal up to a linear additive term in a strong information theoretical sense that takes the actual values of the ranks into account [KMMS05]. There was also no other algorithm with similar performance known until we gave a deterministic (albeit not so practical) algorithm with similar asymptotic performance [KMMS05].
2.2.6 Priority Queues

We have engineered a parallel priority queue that allows fast concurrent access by many processors [San95b, San98a]. This is important for parallel processing because it allows a bottleneck free implementation of a pool of prioritized jobs. The implementation [San98a] indicates that already for more than about 40 processors, the algorithm outperforms a centralized implementation. The algorithm also has equal or better theoretical performance guarantees than a considerable body of previous theoretical research.

Despite all the work described in Section 2.1.5, from the 1960s until 1999 there was no serious competitor for binary heaps [Wil64] for sequential, comparison-based priority queues in internal memory, if only insertion and delete-min are needed. Small improvements due to better cache efficiency are possible by increasing the degree of the heap [LL96]. But I/O-efficient priority queues like [Arg95, FJKT97, BK98, BCMF99] that promised higher asymptotic improvements have constant factors that make them impractical for the hierarchy between cache and main memory. However, in [San99c, San00b] we showed how to remove a factor of $\geq 3$ in I/O complexity from the approaches [FJKT97, BK98, BCMF99], and thus going close to the lower bound. An efficient internal memory implementation is 2–3 times faster than binary heaps for large inputs. This implementation has now also been adapted to the external setting where it is the work horse for several world leading implementations of external memory graph algorithms [DSSS04, DKS05, Dem06].

2.2.7 Sorted Sequences

In 1977, van Emde Boas invented an intriguing idea for search trees with integer keys in the range $0..2^k - 1$ that allow access in time $O(\log k)$. But only in 1990 there were the first, space efficient implementations [MN90, Wen92]. They turned out to be about a factor two slower than comparison-based search trees [Wen92]. In 2004, we reversed this performance ratio for the important case of 32 bit keys using a highly tuned implementation [DKMS04].

2.2.8 Graphs

We have bridged two gaps between theory and practice for computing minimum spanning trees: [DSSS04] describes the first practical external memory algorithm for minimum spanning trees. The theoretical performance guarantees are asymptotically worse than the best theoretical results [ABT04, ABW02], but a factor $\geq 4$ better than all previously known algorithms, if one takes into account that external
memory is at best a few hundred times cheaper than internal memory. Our implementa-
tion is also only a factor 2–5 slower than a fast internal algorithm that is given
enough fast memory. It can process graphs with billions of nodes in a few hours.

The theoretically best internal memory MST algorithms use the cycle property
and sophisticated data structures for least common ancestor queries to eliminate
heavy edges in constant time per edge [Kin97, KKT95]. However, the algorithms
are so complicated and the involved constant factors are so large that a practical
implementation looks prohibitive. In [KST03], we give the first practical MST al-
gorithm that profits from the cycle property at least for rather dense graph. At the
cost of slower preprocessing, it simplifies the data structures and speeds up the cost
per edge by a crucial constant factor.

We have done intensive work on the shortest path problem. Our parallel shortest
path algorithms [CMMS98, MS98, MS00, MS03b] are now used in several im-
plementations [EBGL06, MBBC06]. A current focus are shortest path queries in
large road networks when (fast and space efficient) preprocessing is allowed. In the
last few years, there has been a veritable race for the fastest method. Since 2005
we are leading this race with short interruptions [SS05, SS06, BFM+07]. For ex-
ample, recently we have won the DIMACS implementation challenge on shortest
path queries. In the same context, we have also obtained results on goal directed
speedup techniques [MSM06, DSSW06] and the first algorithm for very fast com-
putation of distance tables for vehicle routing problems [KSS+07].

Further algorithm engineering work on fundamental graph algorithms includes
the first implementations of external memory breadth first search [ADM07], a sim-
ple algorithm for approximate maximum weighted matching [PS04], and the first
implementation of the theoretically best algorithm for maximum flow [HST98].

2.2.9 Strings

In 2003 we developed the first linear time algorithm for constructing suffix ar-
rays [KS03, KSB05]. Since this algorithm is very simple and because suffix ar-
rays have many applications in data compression, full text indexing, and bioinfor-
matics, there have been several implementations. So far, these implementations
cannot beat highly engineered asymptotically suboptimal implementations such as
[MF02] for real world instances. However, the algorithm turned out to be the basis
for the best practical algorithms for parallel [KS06b, KS07] and external memory
[DKMS05, DKMS07] suffix array construction.

http://dimacs.rutgers.edu/Workshops/Challenge9/
2.2.10 Algorithm Libraries

We have experience with algorithm libraries for algebraic data types [San93], parallel search [San97, San98b], external memory (STXXL) [DKS05, Dem06], and multi-cores (MCSTL) [SS07]. The latter two libraries augment and partially reimplement the C++ standard template library STL. Thus, some programs can be externalized or parallelized by simple recompilation. At the very least, we get an interface that is already known to millions of programmers. Furthermore, we can expect that compiler writers invest a lot of work in efficiently compiling programs written in “STL style”. The STXXL is already used at $\geq 17$ sites.

2.2.11 Parallel Processing Primitives

We have worked on collective communication operations for large messages [SS00, SS03, ST06], for hierarchical machines [ST02] and for irregular message lengths [SSO00, SSO01]. Some of these results have been incorporated into the NEC MPI implementation. We have also developed a widely used benchmark for communication in MPI [RSPM98]. The MCSTL [SS07] uses atomic shared memory operations supported by the hardware for providing its own implementations of a shared queue data structure. MCSTL also load-balances many of its algorithms and contains a dynamic load balancer for parallelizable loops. Further work on load balancing includes [San94a, San95a, San96, San99b, San00a, SEK00, San02c, San03b, KSV03, CSW07].

Appendices

- References cited in the text. Papers with coauthors from our group are marked with a bullet.

- CV with list of publications for Peter Sanders

- A selection of three papers by Peter Sanders [SW04, DKS05, BFM+07] (A short version of the [BFM+07], [BFSS07], will appear in Science.)
References


[BMS02] D. Bader, B. Moret, and P. Sanders. Algorithm engineering for parallel computation. In *Experimental Algorithmics — From Algorithms*


