Algorithm Engineering for Scalable Parallel External Sorting

Peter Sanders

Karlsruher Institut für Technologie
Karlsruhe, Germany
Email: sanders@kit.edu

***Abstract*—The talk describes algorithm engineering (AE) as a methodology for algorithmic research where design, analysis, implementation and experimental evaluation of algorithms 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. We use one main example throughout this paper: sorting huge data sets using many multi-core processors and disks. The described system is the current record holder for the GraySort and MinuteSort sorting benchmarks.**

***Keywords*-algorithm engineering, sorting, parallel external memory algorithms, massive data sets**

Algorithms and data structures are at the heart of every computer application and thus of critical 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, or planning in production, logistics and transportation as example areas where algorithms play a key role.

How is algorithmic innovation transferred to applications?

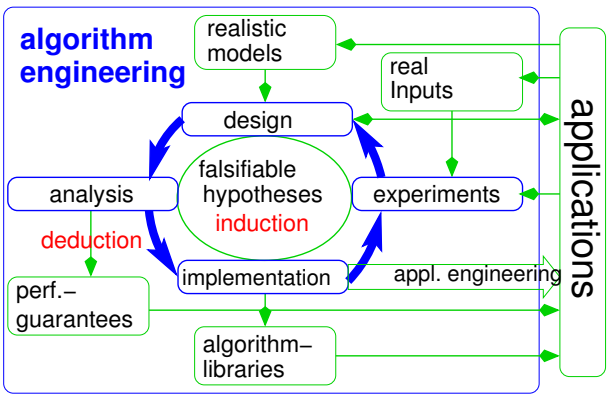

Figure 1. Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses.

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. diverges 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 too 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, 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 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

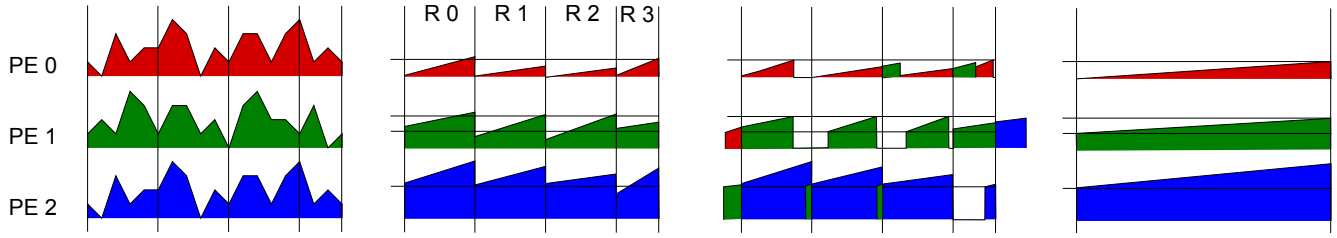

Figure 2. Parallel distributed memory external multiway mergesort.

and experimental evaluation used in the natural sciences. 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 at least 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 applications and algorithm libraries. 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. This paper is a shortened version of [3] where minimum spanning trees were used as an example.

Parallel External Sorting

The sorting example is described in more detail in [2]. Here we highlight some aspects of this work for each of the main activities of algorithm engineering:

Model: We have aspects of parallel disk external memory, distributed memory, and (per-node) shared memory. Instead of attempting to fit all this into one rather specialized model, we design and analyze the algorithm one aspect at a time.

Design: The algorithms are basically parallelizations of external memory multiway mergesort. Besides a theoretically motivated algorithm that minimizes I/Os, we focus on a practical algorithm that minimizes both communication and I/Os in most practical cases. Figure 2 gives an overview. We start with a run formation phase that fills the aggregate memory of the parallel machine with data, uses internal memory parallel sorting, and writes the sorted runs back to disk. Let p denote the number of nodes. The subsequent multiway merging is parallelized by partitioning each run into p pieces such that the i-th piece contains exactly the data destined for processor i. A key observation is that for huge inputs, this can be done using an amount of work that is negligible compared to the overall work. Each processor is then merging the data from its pieces. Parallel multiway mergesort is also used further down in the hierarchy – in (distributed) internal memory and within a node with multiple cores.

Analysis: In the best case, data for processor i will already be written to the disk of processor i after run

formation. The same will very nearly true for the random inputs used in the SortBenchmark[1] (which of course has to move almost all data during run formation). Moreover, we show that by reading random blocks during run formation we can achieve a similar effect.

Implementation: The GraySort benchmark asks for sorting 100 TB of data which is more than half the space available on our machine. Hence we had do implement inplace external sorting and inplace all-to-all communication.

Experiments: We considered difficult instances where active randomiztion is important and the input conventions of the SortBenchmark. The latter considers 100-byte elements with a 10-byte key. The testing machine was a 200-node Linux cluster. Each node consists of two Quad-Core Intel Xeon X5355 processors clocked at 2 667 MHz with 16 GiB main memory. The nodes are connected by a 288-port InfiniBand 4xDDR switch. Each node has four disks with a capacity of 250 GB each. The results using 195 nodes show that we solve the Terabyte benchmark in less than 64 seconds, which is about a third of the time needed by the 2007 winner. This is despite the fact the we use the same number of cores, but only a third of the hard disks. We also slightly improve on a recent result for the Terabyte category published informally by Google [2], where 12 000 disks were used instead of 780 as in our case. In the MinuteSort category, a time limit of one minute is given, the processed amount of the data is the metric. We have beaten the 2007 record by a factor of 3.6, processing 955 GB of data. Yahoo achieved a result half as high using the Hadoop framework, but with a machine 7 times as large. However, for the SortBenchmark results mentioned so far, $N < M$, so the sort is merely internal and only 2 I/Os per block of elements are needed. In the newly established GraySort category, we sort 10^{14} bytes (close to 100 TiB) in about three hours, resulting in about 564 GB/min. The Google program in this case takes only twice the time for ten times the amount of data, but they use an even larger machine than before, featuring 48 000 disks, which is a factor of 61 larger. The better performance of a factor of 5 is thus reduced to less than 0.1 in terms of relative efficiency. Yahoo's result of 578 GB/min is only 2.5% faster than us, but its efficiency is much worse, since they used 17 times the number of nodes. Those nodes were very similar to the ones used by us, except having only half the memory. They also had a worse communication bandwidth. However, this would not have been a limiting factor for our algorithm.

Algorithm Libraries: We use MCSTL [4] – a parallel implementation of the C++ STL for multi-core parallel sorting and multiway merging. STXXL [1] – an external memory implementation of STL is used to provide asynchronous I/O with little data copying and overlapping of I/O and computation. Communication is done with MPI. However we need a layer on top of MPI supporting inplace operation and 64 bit message lengths.

Instances and Benchmarks: The SortBenchmark is a good example how a generally accepted benchmark can promote and focus a research area. However, our experiences with worst case inputs also show that one should not solely rely on uniformly distributed random data.

[1] http:\\www.sortbenchmark.org

[2] http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html

ACKNOWLEDGEMENTS

I would like to thank the coinitiators of the DFG SPP 1307 Algorithm Engineering[3], Kurt Mehlhorn, Rolf Möhring, Burkhard Monien, and Petra Mutzel for their advice and fruitful discussions that led to the definition presented here. The sorting results have been obtained in cooperation with Roman Dementiev, Mirko Rahn, and Johannes Singler. Partially supported by DFG grant SA 933/3-2.

REFERENCES

[1] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL data sets. *Software Practice & Experience*, 38(6):589–637, 2008.

[2] M. Rahn, P. Sanders, and J. Singler. Scalable distributed-memory external sorting. In *26th IEEE International Conference on Data Engineering*, 2010. to appear, extended version at http://arxiv.org/abs/0910.2582v1.

[3] 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.

[4] 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.

[3] http://www.algorithm-engineering.de/