Synonyms
Experimental Parallel Algorithmics

Definition
Algorithmics is the subdiscipline of computer science that studies the systematic development of efficient algorithms. Algorithm Engineering (AE) is a methodology for algorithmic research that views design, analysis, implementation, and experimental evaluation of algorithms as a cycle driving algorithmic research. Further components are realistic models, algorithm libraries and a multitude of interrelations to applications. Figure 1 gives an overview. A more detailed definition can be found in [6]. This article is concerned with particular issues that arise in engineering parallel algorithms.

Discussion
Introduction
The development of algorithms is one of core areas of computer science. After the early days of the 1940s–1960s, in the 1970s and 1980s, algorithmics was largely viewed as a subdiscipline of computer science that is concerned with “paper-and-pencil” theoretical work – design of algorithms with the goal to prove worst case performance guarantees. However, in the 1990s it became more and more apparent that this purely theoretical approach delays the transfer of algorithmic results into applications. Therefore, in algorithm engineering, implementation and experimentation are viewed as equally important as design and analysis of algorithms. Together these four components form a feedback cycle: Algorithms are designed, then analyzed and implemented. Together with
Figure 1: Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses.

experiments using realistic inputs, this process induces new insights that lead to modified and new algorithms. The methodology of algorithm engineering is augmented by using realistic models that form the basis of algorithm descriptions, analysis, and implementation and by algorithm libraries that give high quality reusable implementations.

The history of parallel algorithms is exemplary for the above development where many clever algorithms were developed in the 1980s that were based on the PRAM model of computation. While this yielded interesting insights into the basic aspects of parallelizable problems, it has proved quite difficult to implement PRAM algorithms on mainstream parallel computers.

The remainder of this article closely follows Figure 1, giving one section for each of the main areas models, design, analysis, implementation, experiments, instances/benchmarks, and algorithm libraries.

Models

Parallel computers are complex systems containing processors, memory modules, and networks connecting them. It would be very complicated to take all these aspects into account at all times when designing, analyzing, and implementing parallel algorithms. Therefore simplified models are needed. Two families of such models have proved very useful: In a shared memory machine,
all processors access the same global memory. In a distributed memory machine, several sequential computers communicate via an interconnection network. While these are useful abstractions, the difficulty is to make these models more concrete by specifying what operations are supported and how long they take. For example, shared memory models have to specify how long it takes to access a memory location. Sequential models usually assume constant access time and this also reflects the best case behavior of many parallel machines. However, in the worst case, most real world machines will exhibit severe contention when many processors access the same memory module. Hence, despite many useful models (e.g., QRW PRAM – Queue Read Queue Write Parallel Random Access Machine [3]), there remains a considerable gap to reality when it comes to large scale shared memory systems.

The situation is better for distributed memory machines, in particular, when the sequential machines are connected by a powerful switch. It can then be assumed that all processors can communicate with an arbitrary partner with predictable performance. The LogP model [2] and the Bulk Synchronous Parallel model [10] put this into a simple mathematical form. Equally useful are folklore models where one simply defines the time needed for exchanging a message as the sum of a startup overhead and a term proportional to the message length. Then the assumption is usually that every processor can only communicate with a single other processor at a time, or perhaps it can receive from one processor and send to another one. Also note that algorithms designed for a distributed memory model often yield efficient implementations on shared memory machines.

**Design**

As in algorithm theory, AE is interested in efficient algorithms. However, in algorithm engineering, it is equally important to look for simplicity, implementability, and possibilities for code reuse. In particular, since it can be very difficult to debug parallel code, the algorithms should be designed for simplicity and testability. Furthermore, efficiency means not just asymptotic worst case efficiency, but also to look at the constant factors involved. For example, operations where several processors interact can be a large constant factor more expensive than local computations. Furthermore, AE is not only interested in worst-case performance but also in 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.

**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, a central problem in parallel processing is partitioning of large graphs into approximately equal sized blocks such that few edges are cut. This problem has many applications, e.g., in scientific computing. Currently available algorithms with performance guarantees are too slow for practical use. Practical methods iteratively contract the graph while preserving its basic structure until only few nodes are left, compute an initial solution on this coarse representation, and then improve by local search on each level. These algorithms are very fast and yield good solutions in many situations yet no performance guarantees are known (see [11] for a recent overview).

Implementation

Despite huge efforts in parallel programming languages and in parallelizing compilers, implementing parallel algorithms is still one of the main challenges in the algorithm engineering cycle. There are several reasons for this. First, there are huge semantic gaps between the abstract algorithm description, the programming tools used, and the actual hardware. In particular, really efficient codes often use fairly low level programming interfaces such as MPI or atomic memory operations in order to keep the overheads for processor interaction manageable. Perhaps more importantly, debugging parallel programs is notoriously difficult.

Since performance is the main reason for using parallel computers in the first place, and because of the complexity of parallel hardware, performance tuning is an important part of the implementation phase. Although the line between implementation and experimentation is blurred here, there are differences. In particular, performance tuning is less systematic. For example, there is no need for reproducibility, detailed studies of variances etc. when one finds out that sequential file I/O is a prohibitive bottleneck or when it turns out that a collective communication routine of a particular MPI implementation has a performance bug.

Experiments

Meaningful experiments are the key to closing the cycle of the AE process. 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.
Experiments with parallel algorithms are challenging because the number of processors (let alone other architectural parameters) provide another degree of freedom for the measurements, because even parallel programs without randomization may show nondeterministic behavior on real machines, and because large parallel machines are an expensive resource.

Experiments on comparing the quality of the computed results are not so much different from the sequential case. The remainder of this section therefore concentrates on performance measurements.

Measuring Running Time

The CPU time is a good way to characterize the time used by a sequential process (without I/Os), even in the presence of some operating system interferences. In contrast, parallel programs have to measure the actual elapsed time (wall-clock time) in order to capture all aspects of the parallel program, in particular communication and synchronization overheads. Of course, the experiments must be performed on an otherwise unloaded machine, by using dedicated job scheduling and by turning off unnecessary components of the operating system on the processing nodes. Usually, also further aspects of the program, like startup, initialization, and shutdown are not interesting for the measurement. Thus timing is usually done as follows: All processors perform a barrier synchronization immediately before the piece of program to be timed; one processor $x$ notes down its local time and all processors execute the program to be measured. After another barrier synchronization, processor $x$ measures the elapsed time. As long as the running time is large compared to the time for a barrier synchronization, this is an easy way to measure wall-clock time. To get reliable results, averaging over many repeated runs is advisable. Nevertheless, the measurements may remain unreliable since rare delays, e.g., due to work done by the operating system can become quite frequent when they can independently happen on any processor.

Speedup and Efficiency

In parallel computing, running time depends on the number of processors used and it is sometimes difficult to see whether a particular execution time is good or bad considering the amount of resources used. Therefore, derived measures are often used that express this more directly. The speedup is the ratio of the running time of the fastest known sequential implementation to that of the parallel running time. The speedup directly expresses the impact of parallelization. The relative speedup is easier to measure because it compares with the parallel algorithm running on a single processor. However, note that the relative speedup can significantly overstate the actual usefulness of the parallel algorithm since the sequential algorithm may be much faster than the parallel algorithm run on a single processor.

For a fixed input and a good parallel algorithm, the speedup will usually start slightly below one for a single processor, and initially goes up linearly with
the number of processors. Eventually, the speedup curve gets more and more flat until parallelization overheads become so large that the speedup goes down again. Clearly, it makes no sense to add processors beyond the maximum of the speedup curve. Usually it is better to stop much earlier in order to keep the parallelization cost effective. Efficiency, the ratio of the speedup to the number of processors, more directly expresses this. Efficiency usually starts somewhere below one and then slowly decreases with the number of processors. One can use a threshold for the minimum required efficiency to decide on the maximum number of efficiently usable processors.

Often, parallel algorithms are not really used to decrease execution time but to increase the size of the instances that can be handled in reasonable time. From the point of view of speedup and efficiency, this is good news because for a scalable parallel algorithm, by sufficiently increasing the input size, one can efficiently use any number of processors. One can check this experimentally by scaling the input size together with the number of processors. An interesting property of an algorithm is how much one has to increase the input size with the number of processors. The isoefficiency function \[4\] expresses this relation analytically, giving the input size needed to achieve some given, constant efficiency. As usual in algorithmics, one uses asymptotic notation to get rid of constant factors and lower order terms.

**Speedup anomalies**

Occasionally, efficiency exceeding one (also called superlinear speedup) causes confusion. By Brent’s principle (a single processor can simulate a \(p\)-processor algorithm with a uniform slowdown factor of \(p\)) this should be impossible. However, genuine superlinear absolute speedup can be observed if the program relies on resources of the parallel machine not available to a simulating sequential machine, e.g., main memory or cache.

A second reason is that the computations done by an algorithm can be done in many different ways, some leading to a solution fast, some more slowly. Hence, the parallel program can be “lucky” to find a solution more than \(p\) times earlier than the sequential program. Interestingly, such effects do not always disappear when averaging over all inputs. For example, Schöning \[7\] gives a randomized algorithm for finding satisfying assignments to formulas in propositional calculus that are in conjunctive normal form. This algorithm becomes exponentially faster when run in parallel on many (possibly simulated) processors. Moreover, its worst case performance is better than any sequential algorithm. Brent’s principle is not violated since the best sequential algorithm turns out to be the emulation of the parallel algorithm.

Finally, there are many cases were superlinear speedup is not genuine, mostly because the sequential algorithms used for comparison is not really the best one for the inputs considered.
Instances and Benchmarks

Benchmarks have a long tradition in parallel computing. Although their most visible use is for comparing different machines, they are also helpful within the AE cycle. During implementation, benchmarks of basic operations help to select the right approach. For example, SKaMPI [5] measures the performance of most MPI calls and thus helps to decide which of several possible calls to use, or whether a manual implementation could help.

Benchmark suites of input instances for an important computational problem can be key to consistent progress on this problem. Compared to the alternative that each working group uses its own inputs, this has obvious advantages: there can be a wider range of inputs, results are easier to compare and bias in instance selection is less likely. For example, Chris Walshaw’s graph partitioning archive [9] has become an important reference point for graph partitioning.

Synthetic instances, though less realistic than real world inputs can also be useful since they can be generated in any size and sometimes are good as stress tests for the algorithms (though it is often the other way round – naively constructed random instances are likely to be unrealistically easy to handle). For example, for the graph partitioning problem, one can generate graphs that almost look like random graphs but have predesigned structure that can be more or less easy to detect according to tunable parameters.

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 supported are unknown at library implementation time and because the separation of interface and (often highly complicated) implementation is very important. Compared to an application-specific reimplementation, using a library should save development time without leading to inferior performance. Compared to simple, easy to implement algorithms, libraries should improve performance. 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. All these difficulties imply that implementing algorithms for use in a library is several times more difficult 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.

In parallel computing, there is a fuzzy boundary between software libraries whose main purpose is to shield the programmer from details of the hardware and genuine algorithm libraries. For example, the basic functionality of MPI (message passing) is of the first kind, whereas its collective communication routines have a distinctively algorithmic flavor. The Intel Thread Building Blocks (http://www.threadingbuildingblocks.org/) offer several algorithmic tools including a load balancer hidden behind a task concept and distributed data structures such as hash tables. The standard libraries of programming languages can also be parallelized. For example, there is a parallel version of the C++ STL in the GNU distribution [8].

The Computational Geometry Algorithms Library (CGAL) www.cgal.org is a very sophisticated example of an algorithms library that is now also getting partially parallelized [1].

Conclusion

This article explains how algorithm engineering (AE) provides a methodology for research in parallel algorithmics that allows to bridge gaps between theory and practice. AE does not abolish theoretical analysis but contains it as an important component that, when applicable, provides particularly strong performance and robustness guarantees. However, adding careful implementation, well designed experiments, realistic inputs, algorithm libraries, and a process coupling all of this together provides a better way to arrive at algorithms useful for real world applications.

References


