Association for Computing Machinery
Advancing Computing as a Science & Profession

Ubiquity, an ACM publication
July 2014

Check for updates

Ubiquity Symposium

The Multicore Transformation

Engineering Parallel Algorithms

by Peter Sanders

Editor's Introduction

In the past, parallel processing was a specialized approach to high-performance computing. Today, we have to rethink the computational cores of algorithmic and data structures applications. In this article we discuss how this process of rethinking can be understood using algorithm engineering.

Ubiquity Symposium

The Multicore Transformation

Engineering Parallel Algorithms

by Peter Sanders

In the past, parallel processing was a specialized approach to high-performance computing that was constantly harrowed by ever increasing speed of sequential processors. Since clock frequencies have hit an energy consumption wall and the automatic exploitation of instruction parallelism is largely exhausted, the time of "free lunches" is over. Now, explicit parallelism is the most economic way to exploit the increasing transistor budget available through Moore's law. Hence, all performance critical applications have to use parallel hardware efficiently. Beyond embarrassingly parallel parts of applications, we have to rethink the computational cores of the applications—algorithms and data structures. In this article we discuss how this process of rethinking can be understood using algorithm engineering—a methodology for algorithm development that takes both theoretical and practical considerations into account.

The Method

Algorithmics is the subdiscipline of computer science that studies the systematic development of efficient algorithms. In the past, algorithmics has often been seen as synonymous to algorithm theory, which considers the design and analysis of algorithms with asymptotic worst-case performance guarantees. Unfortunately this view led to big gaps between theory and practice. Therefore, in the last two decades, algorithm engineering (AE) has emerged as a method that unifies theoretical and practical aspects. In AE, design, analysis, implementation, and experimentation form a feedback cycle: Algorithms are designed, then analyzed and implemented. Together with the use realistic inputs, the AE process induces new insights that lead to modified and new algorithms. Realistic models and algorithm libraries further augment the method. All these activities provide much more coupling with applications than pure algorithm theory. Figure 1 gives an overview. In the following, we discuss how these aspects impact the development of parallel algorithms. This article is in many respects a shortened version of a more detailed discussions [1, 2].

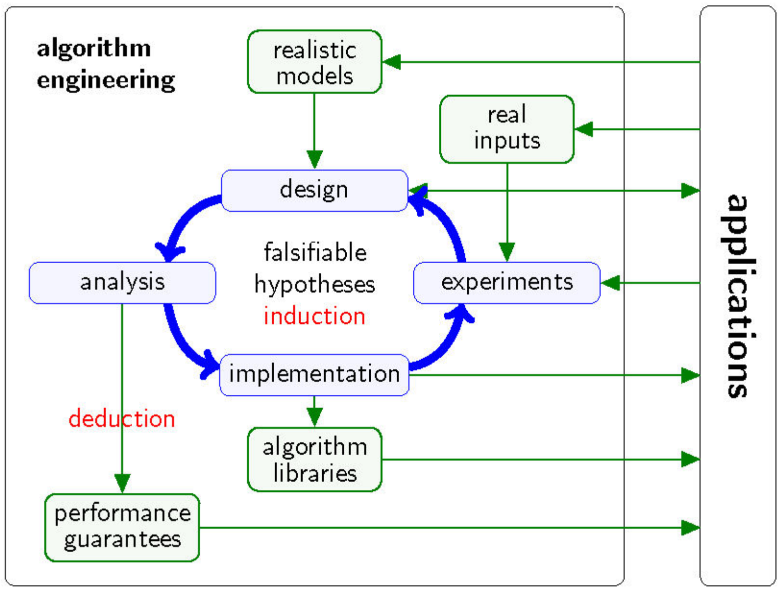

Figure 1. Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses.

We use balanced graph partitioning as an example. Graph partitioning has many applications, not least in parallel computing where each processor should work on a piece of the input minimizing expensive interaction between pieces. Hence, graph partitioning can be important for parallel processing when the parallel program is processing a graph. Our initial goal was to study scalable parallel graph partitioning. This article is also the story of how the AE method led us on quite unexpected paths to interesting results of quite different nature.

Models

Mathematically, clear yet realistic models of the problems and hardware are the basis for AE. Parallel computers are complex systems containing processors, memory modules, and networks connecting them. It would be 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 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. These abstractions have to be concretized by specifying the cost of basic operations. For shared memory systems, PRAM models (parallel random access machines) have been popular for theoretical work since they are simple and powerful extensions of the standard sequential RAM model. Memory can be accessed concurrently in constant time without contention, and synchronization is free since processors proceed in lockstep. PRAM models have been rightfully criticized as grossly unrealistic. However, they are a good starting point for exposing parallelism. When the rules for designing practicable algorithms described are observed, it will often be possible to refine the PRAM algorithms using standard implementation techniques for realistic machines. A striking example is graphics processing units, which require so much parallelism and have so little fast memory per thread that "ancient," and seemingly unrealistic, PRAM algorithms have proved useful where a thread works only on a tiny number of input elements [3].

At the cost of being slightly lower level, message passing models are more directly applicable. By taking both message length and startup overheads into account, we give preference to coarse grained computations and the local memories of the processors already grasp an important aspect of memory hierarchies. Synchronization is naturally connected to message exchange. All these advantages imply that even on shared memory hardware, it often makes sense to design algorithms for a message passing model and then to implement it using shared memory primitives. This approach is particularly successful on modern many-core machines with non-uniform memory access costs (NUMA).

Recent research on machine models tries to take the deep memory hierarchy of modern machines into account (so far with little success). Basically, we have several levels of memory (e.g., registers, L1–L3 cache, socket local memory modules, node local RAM, SSDs, hard disks, etc.), which are partitioned into pieces, each possibly shared by several processing elements. For example, simultaneously executing hardware threads have their own register sets, yet share L1 cache if they run on the same core. Cores on the same chip share L3 cache and the local memory modules, etc. These hierarchies are too complex to take all their aspects into account all the time. One solution is a hierarchical design, e.g., developing a parallel sorting algorithm in a distributed memory model calling a shared memory algorithm for node local sorting and so on. For some applications, abstraction works: The program is formulated in terms of divide-and-conquer recursion and parallel loops. A scheduling algorithm maps the computations to the hardware such that locality is maintained on all levels.

For the equally important aspect of modeling the application let us take a short look at the graph partitioning example: In the standard model, the input consists of a graph and two parameters k and ϵ. The output is a partition of the node set V into k pieces such that no piece contains more than $(1 + \epsilon)|V|/k$ nodes. The objective is to minimize the cut size—the number of edges running between different pieces. This model is an oversimplification. For most parallel processing applications it does not correctly model communication volume or even actual communication time. The model still remains successful since other objective functions are strongly correlated in practice and more difficult to handle algorithmically.

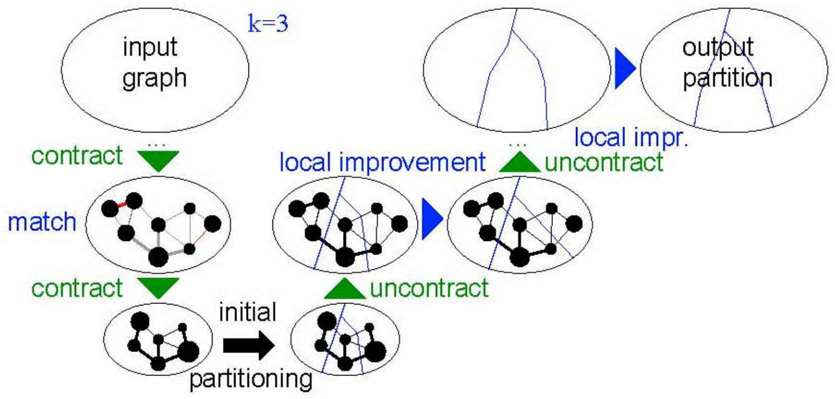

Figure 2. Multilevel graph partitioning.

Design

As in algorithm theory, AE is interested in efficient algorithms. However, in AE, it is equally important to look for simplicity, implementability, testability, and code reuse. Efficiency means not just asymptotic efficiency, but also has to take constant factors into account. For example, operations where several processors interact can be a large constant factor more expensive than local computations. While algorithm theory concentrates on worst-case instances, real-world instances are often much easier.

Of course, the most important (and difficult) issue is where ideas for algorithms come from. What we can always do is to look for known solutions for similar problems or for algorithm design patterns like divide-and-conquer, dynamic programming, greedy, or preprocessing. It is particularly important to be aware of parallel algorithm design patterns and basic tools since those are not yet taught regularly in universities: collective operations like reduction and prefix sums; general purpose load balancing techniques (e.g., using work stealing or graph partitioning); and multilevel algorithms. Additional opportunities stem from interactions with the other activities of AE. A good problem model is sometimes already half of the solution. Experiments and careful analysis can yield insights into things needed in the algorithm.

Multilevel graph partitioners routinely produce high-quality partitions of real-world huge graphs in near linear time although the problem is NP-hard and only expensive approximation algorithms with rather loose guarantees are known. Figure 2 gives the basic scheme: The graph is first shrunk, usually by repeatedly contracting the edges in a matching to single nodes. An initial partition on the coarse graph then already gives quite good starting solutions for the finer levels. These solutions are subsequently improved by moving nodes at every level. We have a developed a system that reuses algorithms and codes for many graph problems like matching, edge coloring, strongly connected components, maximum flows, and negative cycle detection to achieve high-quality solutions.

Analysis

Analyzing parallel algorithms is important since it helps to understand scalability issues. Mathematically speaking, if *n* is the inputs size, *p* the number of processors, and $T_{seq}(n)$ the sequential execution time then the parallel execution time should look something like $O + \left(T_{seq}\left({}^{n}/_{p}\right)\right) + f(n,p)$ where f grows more slowly than T_{seq} for large *n*. This means that for large inputs the algorithm is cost efficient compared to the sequential solution. If *f* grows slowly then the algorithm is scalable for large *p* even for small values of *n/p*. For massively parallel computing, we may adopt the old goal of PRAM algorithms to achieve an f that is polylogarithmic in *n* and *p* (i.e., $O\left(\log^{k}(np)\right)$ for some constant *k*). On smaller machines, less strict scalability requirements may lead to simpler algorithms or better constant factors.

Besides asymptotic worst-case analysis, AE also cares about best cases, average case, or smoothed analysis where we consider randomly perturbed worst-case instances. For example, for our graph partitioner, we made important design decisions based an analytical treatment using (over)simplifying assumptions (e.g., modeling local search as a random walk). Using traditional, accurate theoretical worst-case analysis is useless here since it just tells us the known fact that multilevel graph partitioning does not work for all instances. Doing no analysis at all would lead to an exploding space of poorly understood tuning parameters.

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. Also, 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.

In our graph partitioning example, even a sequential implementation is complex with code size running into a high five-digit number of lines where much of the code is critical when it comes to solution quality over a wide range of inputs. Parallelization introduces additional complexity, in particular since a distributed graph data structure is cumbersome to implement. Since graph partitioning should be part of high-performance, higher-level parallel graph handling tools, we may not be able to use high-level tools ourselves.

Experiments

Meaningful experiments are the key to closing the cycle of the AE process. Similar to the natural sciences the objective is to verify (and subsequently refine) falsifiable hypotheses, e.g., that a graph partitioner computes better solutions than existing systems or that it is faster. Although in AE we can perform many experiments with relatively little effort, planning, evaluation, archiving, postprocessing, and interpretation of results are challenging tasks. Parallel processing makes experimentation even more difficult by introducing additional degrees of freedom (e.g. number of processors), because results can be nondeterministic, and because large parallel machines are an expensive resource.

For our massively parallel graph partitioner we expected from previous work that parallization would lead to decreased solution quality. The big surprise was the contrary effect—parallelization led to better quality. Based on this observation we learned focusing the node movements on one small part of the graph at a time increases the chance of finding nontrivial improvements. Less surprising , but even more important, experiments led to further improvements like better ways to select edges for contraction. Another experimental observation is that handling a large number of partitions ($k \gg 2$) is difficult because getting it right everywhere is unlikely. This observation led the way to a parallel evolutionary algorithm that is able to combine good aspects of several solutions. Similarly, the difficulty of getting almost perfect balance ($\epsilon \approx 0$) led to an algorithm coordinating the movements on many nodes spread over multiple pieces in such a way that the overall balance is maintained or improved.

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 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 can be key to consistent progress. Compared to the alternative—each working group uses its own inputs—benchmark suites offer obvious advantages: there can be a wider range of inputs, results are easier to compare, and bias in instance selection is less likely.

For graph partitioning, Walshaw's benchmark suite [4] has played such a pacemaker role. By listing the software achieving best quality for each instance, it has led to interesting insights and consistent progress. The 10th DIMACS implementation challenge[1] complements Walshaw's benchmark by looking at larger graphs, a wider range of applications, several objective functions, and the tradeoff between running time and quality. Our partitioner not only fared well with respect to the cut size objective, but also with respect to maximum communication volume, which is a better fit for parallel processing applications. This is surprising since it outperformed solvers based on hypergraph partitioning that explicitly take communication volume into account.

Algorithm Libraries

Algorithm libraries bundle implementations of several algorithms using the methods of software engineering. The library functions shield the user from complex algorithm inside and possibly from low-level details of the hardware. The result should be efficient, easy to use, well documented, and portable. Algorithm libraries accelerate the transfer of know-how into

[1] http://www.cc.gatech.edu/dimacs10/

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 implementation is crucial. 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. Also note 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. 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.

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 encapsulate nontrivial algorithms. The Intel Thread Building Blocks 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. Using the functionality provided there (sort, partition, remove if), we were for example able to parallelize the Filter Kruskal algorithm for minimum spanning trees. Perhaps the oldest and most widely used algorithm libraries are for standard numeric problems, in particular linear algebra.

In principle, graph partitioning libraries work well since they encapsulate complex code behind a rather simple interface. However, even if we fix the objective function, the best algorithm configuration highly depends on input size k, ϵ, time budget, and graph structure. For example, road networks, grids from numerical simulations, and social networks can behave fundamentally different. Hence, a robust and efficient graph partitioning library remains an interesting research problem.

Conclusion

We have discussed how a method based on integrating modeling, design, implementation, and experimentation can lead to high-performance parallel algorithms for the computational kernels of those applications where parallelization is nontrivial. A lot of work remains. Even where good parallel algorithms are available, ever more complicated hardware and exploding input sizes imply challenging problems including scalability, fault tolerance, and energy consumption. Calls for better programming abstractions or hardware that is easy to program will remain loud but in the past had limited success. Where they are successful, algorithms will play an important role in translating between user and hardware.

Application programmers often lack experience in algorithmics, math, computer architecture, performance tuning, or experimental methodology to harness the full potential of algorithm engineering. This problem can be mitigated by a three-tiered approach. First, teaching and research in algorithm engineering has to be intensified. Second, applications can employ algorithm libraries and other reusable components. Third, consulting companies and cooperation with research groups can help reimplement the computational kernels—usually without having to touch the bulk of the code around it. Note that the division of labor between applications experts and algorithm engineers implied by the approaches mentioned within requires an exchange of meaningful problem instances between the two groups. This is currently often lacking due to legal uncertainties, mistrust, or simply ignorance of the importance of realistic data. Therefore, a new "culture of collaboration" may be needed involving mutual awareness of the problem, reusable contracts, and procedures for the exchange of data.

References

[1] Sanders, P. Algorithm Engineering: An attempt at a definition. In *Efficient Algorithms*, volume 5760 of LNCS.. Springer, 2009, 321–340.

[2] Sanders, P. Algorithm Engineering. In *Encyclopedia of Parallel Computing*, volume 1. Springer, 2011, 33–38.

[3] Satish, N. , Harris, M., and Garland, M. Designing efficient sorting algorithms for many core GPUs. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, September 2008.

[4] Soperm, A. J., Walshaw, C., and Cross, M. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *J. Global Optimization* 29, 2 (2004), 225–241.

About the Author

Dr. Peter Sanders is a proffers in the Department of Informatics at Karlsruhe Institute of Technology (Germany). His research is focused on algorithm theory and algorithm engineering i.e. the design, the implementation and the analysis of efficient algorithms. In this case, analysis can be both, theoretical and experimental. Some of his important research topics are: parallel processing and communication in networks, solving problems with "irregular" structure, randomized algorithms, memory hierarchies (disks, caches), and realistic models for problems and machines.

DOI: 10.1145/2618399