Open Problems

A number of problems related to proper learning in the PAC model and its extensions are open. Almost all hardness of proper learning results are for learning with respect to unrestricted distributions. For most of the problems mentioned in Sect. "Key Results" it is unknown whether the result is true if the distribution is restricted to belong to some natural class of distributions (e. g. product distributions). It is unknown whether decision trees are learnable properly in the PAC model or in the PAC model with membership queries. This question is open even in the PAC model restricted to the uniform distribution only. Note that decision trees are learnable (non-properly) if membership queries are available [5] and are learnable properly in time $O(n^{\log s})$, where s is the number of leaves in the decision tree [1].

An even more interesting direction of research would be to obtain hardness results for learning by richer representations classes, such as AC^0 circuits, classes of neural networks and, ultimately, unrestricted circuits.

Cross References

- ► Cryptographic Hardness of Learning
- ► Graph Coloring
- ► Learning DNF Formulas
- ► PAC Learning

Recommended Reading

1. Alekhnovich, M., Braverman, M., Feldman, V., Klivans, A., Pitassi, T.: Learnability and automizability. In: Proceeding of FOCS, pp. 621–630 (2004)
2. Ben-David, S., Eiron, N., Long, P. M.: On the difficulty of approximately maximizing agreements. In: Proceedings of COLT, pp. 266–274 (2000)
3. Blum, A.L., Rivest, R.L.: Training a 3-node neural network is NP-complete. Neural Netw. **5**(1), 117–127 (1992)
4. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.: Learnability and the Vapnik-Chervonenkis dimension. J. ACM **36**(4), 929–965 (1989)
5. Bshouty, N.: Exact learning via the monotone theory. Inf. Comput. **123**(1), 146–153 (1995)
6. Feldman, V.: Hardness of Approximate Two-level Logic Minimization and PAC Learning with Membership Queries. In: Proceedings of STOC, pp. 363–372 (2006)
7. Feldman, V.: Optimal hardness results for maximizing agreements with monomials. In: Proceedings of Conference on Computational Complexity (CCC), pp. 226–236 (2006)
8. Garey, M., Johnson, D.S.: Computers and Intractability. W. H. Freeman, San Francisco (1979)
9. Guruswami, V., Raghavendra, P.: Hardness of Learning Halfspaces with Noise. In: Proceedings of FOCS, pp. 543–552 (2006)
10. Hancock, T., Jiang, T., Li, M., Tromp, J.: Lower bounds on learning decision lists and trees. In: 12th Annual Symposium on Theoretical Aspects of Computer Science, pp. 527–538 (1995)
11. Haussler, D.: Decision theoretic generalizations of the PAC model for neural net and other learning applications. Inf. Comput. **100**(1), 78–150 (1992)
12. Jackson, J.: An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. J. Comput. Syst. Sci. **55**, 414–440 (1997)
13. Kearns, M., Schapire, R., Sellie, L.: Toward efficient agnostic learning. Mach. Learn. **17**(2–3), 115–141 (1994)
14. Kearns, M., Valiant, L.: Cryptographic limitations on learning boolean formulae and finite automata. J. ACM **41**(1), 67–95 (1994)
15. Kearns, M., Vazirani, U.: An introduction to computational learning theory. MIT Press, Cambridge, MA (1994)
16. Pitt, L., Valiant, L.: Computational limitations on learning from examples. J. ACM **35**(4), 965–984 (1988)
17. Valiant, L.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)

High Performance Algorithm Engineering for Large-scale Problems

2005; Bader

David A. Bader
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Keywords and Synonyms

Experimental algorithmics

Problem Definition

Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a robust, efficient, well tested, and easily usable implementation. Thus it encompasses a number of topics, from modeling cache behavior to the principles of good software engineering; its main focus, however, is experimentation. In that sense, it may be viewed as a recent outgrowth of *Experimental Algorithmics* [14], which is specifically devoted to the development of methods, tools, and practices for assessing and refining algorithms through experimentation. The *ACM Journal of Experimental Algorithmics (JEA)*, at URL `www.jea.acm.org`, is devoted to this area.

High-performance algorithm engineering [2] focuses on one of the many facets of algorithm engineering: speed. The high-performance aspect does not immediately imply parallelism; in fact, in any highly parallel task, most of the impact of high-performance algorithm engineering tends to come from refining the serial part of the code.

The term *algorithm engineering* was first used with specificity in 1997, with the organization of the first *Workshop on Algorithm Engineering (WAE 97)*. Since then, this

workshop has taken place every summer in Europe. The 1998 *Workshop on Algorithms and Experiments (ALEX98)* was held in Italy and provided a discussion forum for researchers and practitioners interested in the design, analyzes and experimental testing of exact and heuristic algorithms. A sibling workshop was started in the Unites States in 1999, the *Workshop on Algorithm Engineering and Experiments (ALENEX99)*, which has taken place every winter, colocated with the *ACM/SIAM Symposium on Discrete Algorithms (SODA)*.

Key Results

Parallel computing has two closely related main uses. First, with more memory and storage resources than available on a single workstation, a parallel computer can solve correspondingly larger instances of the same problems. This increase in size can translate into running higher-fidelity simulations, handling higher volumes of information in data-intensive applications, and answering larger numbers of queries and datamining requests in corporate databases. Secondly, with more processors and larger aggregate memory subsystems than available on a single workstation, a parallel computer can often solve problems faster. This increase in speed can also translate into all of the advantages listed above, but perhaps its crucial advantage is in turnaround time. When the computation is part of a real-time system, such as weather forecasting, financial investment decision-making, or tracking and guidance systems, turnaround time is obviously the critical issue. A less obvious benefit of shortened turnaround time is higher-quality work: when a computational experiment takes less than an hour, the researcher can afford the luxury of exploration—running several different scenarios in order to gain a better understanding of the phenomena being studied.

In algorithm engineering, the aim is to present repeatable results through experiments that apply to a broader class of computers than the specific make of computer system used during the experiment. For sequential computing, empirical results are often fairly machine-independent. While machine characteristics such as word size, cache and main memory sizes, and processor and bus speeds differ, comparisons across different uniprocessor machines show the same trends. In particular, the number of memory accesses and processor operations remains fairly constant (or within a small constant factor). In high-performance algorithm engineering with parallel computers, on the other hand, this portability is usually absent: each machine and environment is its own special case. One obvious reason is major differences in hardware that affect the balance of communication and computation costs—a true shared-memory machine exhibits very different behavior from that of a cluster based on commodity networks.

Another reason is that the communication libraries and parallel programming environments (e. g., MPI [12], OpenMP [16], and High-Performance Fortran [10]), as well as the parallel algorithm packages (e. g., fast Fourier transforms using FFTW [6] or parallelized linear algebra routines in ScaLAPACK [4]), often exhibit differing performance on different types of parallel platforms. When multiple library packages exist for the same task, a user may observe different running times for each library version even on the same platform. Thus a running-time analysis should clearly separate the time spent in the user code from that spent in various library calls. Indeed, if particular library calls contribute significantly to the running time, the number of such calls and running time for each call should be recorded and used in the analysis, thereby helping library developers focus on the most cost-effective improvements. For example, in a simple message-passing program, one can characterize the work done by keeping track of sequential work, communication volume, and number of communications. A more general program using the collective communication routines of MPI could also count the number of calls to these routines. Several packages are available to instrument MPI codes in order to capture such data (e. g., MPICH's nupshot [8], Pablo [17], and Vampir [15]). The SKaMPI benchmark [18] allows running-time predictions based on such measurements even if the target machine is not available for program development. SKaMPI was designed for robustness, accuracy, portability, and efficiency; For example, SKaMPI adaptively controls how often measurements are repeated, adaptively refines message-length and step-width at "interesting" points, recovers from crashes, and automatically generates reports.

Applications

The following are several examples of algorithm engineering studies for high-performance and parallel computing.

1. Bader's prior publications (see [2] and http://www.cc.gatech.edu/~bader) contain many empirical studies of parallel algorithms for combinatorial problems like sorting, selection, graph algorithms, and image processing.
2. In a recent demonstration of the power of high-performance algorithm engineering, a million-fold speedup was achieved through a combination of a 2,000-fold speedup in the serial execution of the code and a 512-

fold speedup due to parallelism (a speed-up, however, that will scale to any number of processors) [13]. (In a further demonstration of algorithm engineering, additional refinements in the search and bounding strategies have added another speedup to the serial part of about 1,000, for an overall speedup in excess of 2 billion)

3. JáJá and Helman conducted empirical studies for prefix computations, sorting, and list-ranking, on symmetric multiprocessors. The sorting research (see [9]) extends Vitter's external Parallel Disk Model to the internal memory hierarchy of SMPs and uses this new computational model to analyze a general-purpose sample sort that operates efficiently in shared-memory. The performance evaluation uses 9 well-defined benchmarks. The benchmarks include input distributions commonly used for sorting benchmarks (such as keys selected uniformly and at random), but also benchmarks designed to challenge the implementation through load imbalance and memory contention and to circumvent algorithmic design choices based on specific input properties (such as data distribution, presence of duplicate keys, pre-sorted inputs, etc.).
4. In [3] Blelloch et al. compare through analysis and implementation three sorting algorithms on the Thinking Machines CM-2. Despite the use of an outdated (and no longer available) platform, this paper is a gem and should be required reading for every parallel algorithm designer. In one of the first studies of its kind, the authors estimate running times of four of the machine's primitives, then analyze the steps of the three sorting algorithms in terms of these parameters. The experimental studies of the performance are normalized to provide clear comparison of how the algorithms scale with input size on a 32*K*-processor CM-2.
5. Vitter et al. provide the canonical theoretic foundation for I/O-intensive experimental algorithmics using external parallel disks (e. g., see [1,19,20]). Examples from sorting, FFT, permuting, and matrix transposition problems are used to demonstrate the parallel disk model.
6. Juurlink and Wijshoff [11] perform one of the first detailed experimental accounts on the preciseness of several parallel computation models on five parallel platforms. The authors discuss the predictive capabilities of the models, compare the models to find out which allows for the design of the most efficient parallel algorithms, and experimentally compare the performance of algorithms designed with the model versus those designed with machine-specific characteristics in mind. The authors derive model parameters for each platform, analyses for a variety of algorithms (matrix multiplication, bitonic sort, sample sort, all-pairs shortest path), and detailed performance comparisons.
7. The LogP model of Culler et al. [5] provides a realistic model for designing parallel algorithms for message-passing platforms. Its use is demonstrated for a number of problems, including sorting.
8. Several research groups have performed extensive algorithm engineering for high-performance numerical computing. One of the most prominent efforts is that led by Dongarra for ScaLAPACK [4], a scalable linear algebra library for parallel computers. ScaLAPACK encapsulates much of the high-performance algorithm engineering with significant impact to its users who require efficient parallel versions of matrix–matrix linear algebra routines. New approaches for automatically tuning the sequential library (e. g., LAPACK) are now available as the ATLAS package [21].

Open Problems

All of the tools and techniques developed over the last several years for algorithm engineering are applicable to high-performance algorithm engineering. However, many of these tools need further refinement. For example, cache-efficient programming is a key to performance but it is not yet well understood, mainly because of complex machine-dependent issues like limited associativity, virtual address translation, and increasingly deep hierarchies of high-performance machines. A key question is whether one can find simple models as a basis for algorithm development. For example, cache-oblivious algorithms [7] are efficient at all levels of the memory hierarchy in theory, but so far only few work well in practice. As another example, profiling a running program offers serious challenges in a serial environment (any profiling tool affects the behavior of what is being observed), but these challenges pale in comparison with those arising in a parallel or distributed environment (for instance, measuring communication bottlenecks may require hardware assistance from the network switches or at least reprogramming them, which is sure to affect their behavior). Designing efficient and portable algorithms for commodity multicore and many-core processors is an open challenge.

Cross References

- ► Analyzing Cache Misses
- ► Cache-Oblivious B-Tree
- ► Cache-Oblivious Model
- ► Cache-Oblivious Sorting
- ► Engineering Algorithms for Computational Biology

- ▶ Engineering Algorithms for Large Network Applications
- ▶ Engineering Geometric Algorithms
- ▶ Experimental Methods for Algorithm Analysis
- ▶ External Sorting and Permuting
- ▶ Implementation Challenge for Shortest Paths
- ▶ Implementation Challenge for TSP Heuristics
- ▶ I/O-model
- ▶ Visualization Techniques for Algorithm Engineering

Recommended Reading

1. Aggarwal, A., Vitter, J.: The input/output complexity of sorting and related problems. Commun. ACM **31**, 1116–1127 (1988)
2. Bader, D.A., Moret, B.M.E., Sanders, P.: Algorithm engineering for parallel computation. In: Fleischer, R., Meineche-Schmidt, E., Moret, B.M.E. (ed) Experimental Algorithmics. Lecture Notes in Computer Science, vol. 2547, pp. 1–23. Springer, Berlin (2002)
3. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: An experimental analysis of parallel sorting algorithms. Theor. Comput. Syst. **31**(2), 135–167 (1998)
4. Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In: The 4th Symp. the Frontiers of Massively Parallel Computations, pp. 120–127, McLean, VA (1992)
5. Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken,T.: LogP: Towards a realistic model of parallel computation. In: 4th Symp. Principles and Practice of Parallel Programming, pp. 1–12. ACM SIGPLAN (1993)
6. Frigo, M., Johnson, S. G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing, vol. 3, pp. 1381–1384, Seattle, WA (1998)
7. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Ann. Symp. Foundations of Computer Science (FOCS-99), pp. 285–297, New York, NY, 1999. IEEE Press
8. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Technical report, Argonne National Laboratory, Argonne, IL, (1996) www.mcs.anl.gov/mpi/mpich/
9. Helman, D.R., JáJá, J.: Sorting on clusters of SMP's. In: Proc. 12th Int'l Parallel Processing Symp., pp. 1–7, Orlando, FL, March/April 1998
10. High Performance Fortran Forum. High Performance Fortran Language Specification, 1.0 edition, May 1993
11. Juurlink, B.H.H., Wijshoff, H.A.G.: A quantitative comparison of parallel computation models. ACM Trans. Comput. Syst. **13**(3), 271–318 (1998)
12. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, June 1995. Version 1.1
13. Moret, B.M.E., Bader, D.A., Warnow, T.: High-performance algorithm engineering for computational phylogenetics. J. Supercomput. **22**, 99–111 (2002) Special issue on the best papers from ICCS'01
14. Moret, B.M.E., Shapiro, H.D.: Algorithms and experiments: The new (and old) methodology. J. Univers. Comput. Sci. **7**(5), 434–446 (2001)
15. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: visualization and analysis of MPI resources. Supercomputer 63. **12**(1), 69–80 (1996)
16. OpenMP Architecture Review Board. OpenMP: A proposed industry standard API for shared memory programming. www.openmp.org, October 1997
17. Reed, D.A., Aydt, R.A., Noe, R.J., Roth, P.C., Shields, K.A., Schwartz, B., Tavera, L.F.: Scalable performance analysis: The Pablo performance analysis environment. In: Skjellum, A., (ed) Proc. Scalable Parallel Libraries Conf., pp. 104–113, Mississippi State University, October 1993. IEEE Computer Society Press
18. Reussner, R., Sanders, P., Träff, J.: SKaMPI: A comprehensive benchmark for public benchmarking of MPI. Scientific Programming, 2001. accepted, conference version with Prechelt, L., Müller, M. In: Proc. EuroPVM/MPI (1998)
19. Vitter, J. S., Shriver, E.A.M.: Algorithms for parallel memory. I: Two-level memories. Algorithmica. **12**(2/3), 110–147 (1994)
20. Vitter, J. S., Shriver, E.A.M.: Algorithms for parallel memory II: Hierarchical multilevel memories. Algorithmica **12**(2/3), 148–169 (1994)
21. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software (ATLAS). In: Proc. Supercomputing 98, Orlando, FL, November 1998. www.netlib.org/utk/people/JackDongarra/PAPERS/atlas-sc98.ps

Hitting Set

- ▶ Greedy Set-Cover Algorithms
- ▶ Set Cover with Almost Consecutive Ones

Hospitals/Residents Problem

1962; Gale, Shapley

David F. Manlove
Department of Computing Science,
University of Glasgow, Glasgow, UK

Keywords and Synonyms

College admissions problem; University admissions problem; Stable admissions problem; Stable assignment problem; Stable b-matching problem

Problem Definition

An instance I of the Hospitals/Residents problem (HR) [5,6,14] involves a set $R = \{r_1, \ldots, r_n\}$ of *residents* and a set $H = \{h_1, \ldots, h_m\}$ of *hospitals*. Each hospital $h_j \in H$ has a positive integral *capacity*, denoted by c_j. Also, each resident $r_i \in R$ has a *preference list* in which he ranks in strict order a subset of H. A pair $(r_i, h_j) \in R \times H$ is said