Parallel Algorithms Reconsidered

Peter Sanders

Karlsruhe Institute of Technology
Karlsruhe, Germany
sanders@kit.edu

Abstract

Parallel algorithms have been a subject of intensive algorithmic research in the 1980s. This research almost died out in the mid 1990s. In this paper we argue that it is high time to reconsider this subject since a lot of things have changed. First and foremost, parallel processing has moved from a niche application to something mandatory for any performance critical computer applications. We will also point out that even very fundamental results can still be obtained. We give examples and also formulate some open problems.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity, F.1.2 Parallelism and Concurrency

Keywords and phrases parallel algorithm, algorithm engineering, communication efficient algorithm, polylogarithmic time algorithm, parallel machine model

Digital Object Identifier 10.4230/LIPIcs.STACS.2015.10

Category Invited Talk

1 Introduction

Parallel algorithms were a hot topic in the 1980 but then the subject almost died. For example, a quick, subjective count of the parallel algorithm papers in STOC 1985, 1990, 1995, 2000, 2005, 2010, 2014 gave 13, 8, 11, 6, 1, 1, 6 papers respectively. The left column of the following table gives a number of interrelated very strong reasons why this happened. However, if you also look at the right column, you see that these reasons are not relevant any more.

Parallel computing was in practice **used rarely** because parallel computers were expensive and hard to program due to exotic hardware and software.	Today, parallel hardware is everywhere (see Figure 1). Even smart phones have quad-core processors. The latest Intel server processors support up to 18 cores. With multiple sockets and hardware multithreading, this already ranges into three digit numbers of threads. Graphics processors increasingly used for general purpose computing (GPGPU) have thousands of cores. For example, the NVidia GTX 980 card has 2048 cores and needs a number of hardware threads at least an order of magnitude larger to achieve full performance.

© Peter Sanders;
licensed under Creative Commons License CC-BY
32nd Symposium on Theoretical Aspects of Computer Science (STACS 2015).
Editors: Ernst W. Mayr and Nicolas Ollinger; pp. 10–18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

For most programmers, it was easier to wait until the microprocessor industry provided new processor designs that translate the additional transistor budget due to Moore's law into higher **clock frequency** and higher instruction parallelism.

This stopped when processor design ran against the "power wall" – it is no longer feasible to significantly increase processor clock speeds since this increases the energy consumption to a point where energy costs are too high and where cooling becomes too expensive [16]. For example, in 2004, Intel presented the Pentium 4 Prescott microarchitecture that increased clock frequency and energy consumption but *not* benchmark performance compared to previous models. A short time later, the Netburst line of microarchitectures used for the Pentium 4 was discontinued and Intel started to design more conservative cores putting more and more of them on the same chip.

The actual applications of parallel computers were mostly **numerical** simulations that needed little of the results developed by the algorithm theory community. Exceptions (that almost prove the rule) can be found for lower level aspects like network topologies, e.g. [21].

Now, we have to look for parallelization opportunities in every performance critical application since this is the only way to exploit the available hardware. Moreover, the Big Data boom has produced many new applications outside numerical simulations where massively parallel processing is crucial. Furthermore, the methodology of algorithm engineering (e.g. [29]) makes it easier to bridge gaps between theory and practice.

The machine models used by theorists, like the **PRAM** model were widely criticized as too remote from practice.

Message passing models (e.g., [41, 33]) or memory hierarchies [4] avoid some of the pitfalls of PRAMs. Moreover, PRAM algorithms are often not that impractical if we avoid the fallacy of speeding up computations at the cost of highly inefficient computing.

Most companies specializing in parallel computers did go bankrupt.

Now, parallel computers are mainstream products of the big players. Big data and cloud is at the core of the business of some of the worlds most profitable companies. Computer games proved to a be a killer application (almost literally), catalyzing the use of architectures (GPUs) that would otherwise have been dismissed as too exotic and cumbersome.

In the late 1990s, the **Internet boom** (aka Dotcom bubble) drew parallel algorithms researchers into startups (e.g., Akamai) and into new research fields related to emerging internet applications (e.g., algorithmic game theory).

Some of these people and a new generation of researchers now look at parallel algorithms from a fresh Big Data perspective. Indeed, in 2014 there were 6 STOC papers on parallel algorithms again.

These observations indicate that parallel algorithms should be an even hotter topic than in the 1980s. It seems that today the theory community is lagging behind an important trend. One way to explain this lack of enthusiasm is the hypothesis that, perhaps, researchers may have done a very thorough job in the past and discovered almost all the really interesting parallel algorithms that are there to discover. The main purpose of the remainder of this paper is to refute this hypothesis.

First it should be noted, that in the last two decades there have been important trends in computer science that have a largely unaddressed parallel processing aspect:

- There has been a lot of work on processing large data sets in the presence of *memory hierarchies* (e.g., [23, 43]). Many of the techniques developed there, e.g., time forward processing (e.g., [10, 11]), do not readily translate to parallel processing and thus pose important open problems.

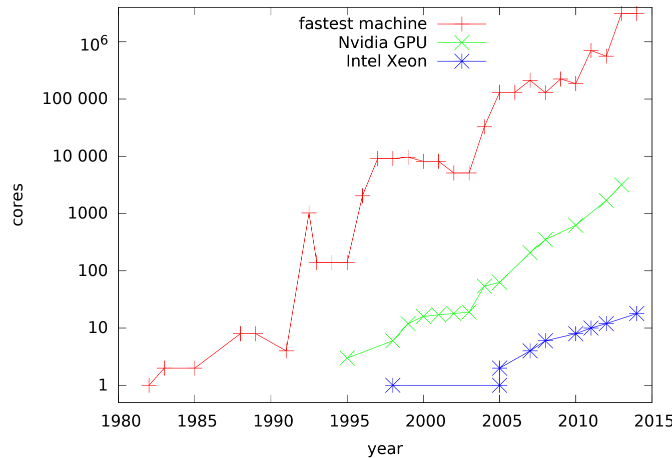

Figure 1 Number of processors (cores) in the worlds fastest supercomputers [40], Nvidia GPUs [44], and Intel Xeon server processors (single chip) [45].

- Data movement in memory hierarchies is vertical data movement between memory units at different levels. Horizontal data movements between processors in a distributed memory machine is an equally important related problem but has been studied much less. An important difference is that horizontal communication volume can be sublinear in the input size if we manage to solve problems by predominantly local computation. The resulting area of *communication efficient algorithms* is full of interesting open problems [33].
- *Streaming* algorithms [18] have explicitly been developed to allow processing large data sets. However, the basic model for streaming algorithms is inherently sequential and needs parallel generalizations. See also [33].
- Many *succinct data structures* (e.g. [17]) have been designed to handle large data sets. However, for many of them it is not clear how to construct them efficiently in parallel.
- *Smoothed analysis* [37] is a sound way to explain why certain algorithms for hard problems are efficient in practice. However, few parallel algorithms have been analyzed in this framework.
- *Fixed-parameter algorithms* (e.g., [25]) study efficient algorithms for “easy” instances of hard problems. Few of these algorithms have been parallelized so far.
- There has been some early work on parallel *approximation algorithms* [22] but very little on parallelizing the vast number of approximation algorithms studied since then. It is particularly surprising that even the intensive work on scheduling parallel processors has seen very few algorithms for doing that in parallel [3, 31].
- Application areas like *bioinformatics* or *computational finance* recently had large impact on algorithmic research. Many of the investigated problems require parallel algorithms to be useful in practice.
- The *big data* boom brought a large number of new applications into focus, in particular, algorithms for *data analysis* and *machine learning* become important.
- The *energy consumption* of computations is becoming more important than running time. this should become important for algorithm design, in particular for *exascale computing* where the computer architects are already basing most of their design decisions on energy consumption (e.g. [20]).

- Applications on exascale computers and Big data also require *fault tolerance*. Building that already into the algorithms is a promising research area. The algorithm theory community has done some work on *resilient algorithms* [14] that can survive certain memory corruptions but is has not embraced fault tolerant parallel algorithms. This is surprising because fault tolerance is actually *easier* to achieve in a parallel system since intact processors may step in for faulty ones.

2 Examples from our Work

In order to illustrate that there is a bonanza of quite fundamental results on parallel algorithms still to be found, we describe a selection of our results on parallel algorithms published since 2013.

2.1 Sorting

Sorting is one of the most intensively studied algorithmic problems. It is of particular interest to parallel computing since sorting is often used to bring data together that has to be processed together. We were able to obtain several quite fundamental new results on sorting.

String sorting. is practically important since many big data applications have variable length keys. The theoretical challenge here is to exploit that only *distinguishing prefixes* need to be inspected – indeed sequential string sorting needs work only linear in the total distinguishing prefix size. We found *no* previous work on parallel string sorting except some PRAM algorithms always inspecting the entire input which are thus work-inefficient. We adapted parallel sorting algorithms for atomic objects so that they only inspect the distinguishing prefixes [8, 7].

Malleable sorting. In practice, parallel programs have to share resources (e.g. processors) with other programs. Therefore, the amount of available resources for a particular program may vary over time in an online fashion. Thus parallel algorithms should be able to dynamically adapt to the amount of available resources. We have studied this phenomenon for the example of sorting and show that this yields advantages over leaving this adaptivity to the operating system [15].

Massively parallel sorting. There are many asymptotically efficient sorting algorithms running in polylogarithmic time. However, all these algorithms require the data to be moved at least a logarithmic number of times. On the other hand, there are algorithms that need to move data only once which makes them much more practical for sorting large inputs on distributed memory machines. However, these algorithms need a linear number of message startups on the critical path which makes them impractical for large machines. We have designed algorithms that interpolate between these to extremes – moving the data k times reduces the critical path length to $kp^{1/k}$ [5]. There were similar algorithms but none with a comparable worst case guarantee.

2.2 Data Structures

There has been a lot of work on concurrent data structures (e.g. [19]). However, much of this is very slow in the worst case since contention of operations competing for the same

place in the data structure can occur. It turns out that these problems can sometimes be avoided by relaxing the data structure semantics or by considering bulk operations.

Relaxed Priority Queues. support concurrent insertions and deletions of elements that have near minimum values. We have designed a very simple such data structure (MultiQueue) based on multiple sequential priority queues. Insertions go to random queues and deletions take the minimum from two randomly sampled queues [28]. This data structure considerably outperforms much more complicated previous data structures.

Approximate Membership. Bloom filters save communication volume by providing a space efficient data structure for approximate membership queries. However, there is surprisingly little work on distributed Bloom filters. For example, a recent survey on Bloom filters in distributed systems [39] mentions no less than 23 variants but none that truly distributes the data structure over multiple processors and thus scales to the largest data sets. We have designed such a structure and apply it for communication efficient duplicate detection and database join [33].

2.3 Graph Algorithms

Multi-objective Shortest Paths. is an intensively studied problem of high practical relevance where parallelization is attractive since it is computationally much more expensive than the standard single-objective case. While the latter problem is difficult to parallelize in the worst case, we have shown that the additional work due to the added objectives is easy to parallelize. Indeed, a very simple generalization of Dijkstra's well known single-objective algorithm turns out to be a scalable parallel algorithm requiring the same number of n iterations [32]. This algorithm also works well in practice [13]. Another interesting aspect of this problem is that it combines graphs and computational geometry.

Maximal matchings. We give a simple linear work polylogarithmic time algorithm for computing maximal matchings in [9]. The algorithm also computes 1/2-approximations of weighted matchings and works well in practice.

Graph partitioning. asks for partitioning the vertex set of a graph into k pieces of about equal size such that the number of cut edges is small. This is a frequently needed (NP-hard) problem that is particularly important for processing graphs in parallel. Our partitioner KaHIP [34] yields the highest quality world wide for a large spectrum of inputs including some of the largest inputs considered so far [1, 24]. The algorithms used are complex heuristics combining many techniques. What is interesting from an algorithm theory point of view is that the practical quality improvements we achieve are in large parts due to integrating solvers for graph theoretical subproblems for which polynomial time algorithms are known. For example, this includes maximum flows, strongly connected components, negative cycle detection, or edge coloring.

2.4 Linear Algebra

One criticism of classical PRAM algorithms is not so much founded in the machine model but in the strive for polylogarithmic execution time even at the cost of inefficient algorithms. One such example are algorithms for matrix inversion and related problems. Theoretical research has found polylogarithmic time inefficient algorithms whereas the algorithms used

in practice perform a near optimal amount of work yet need time $\Omega(n)$ where n is the matrix dimension. We found an asymptotically efficient polylogarithmic time algorithm that also works well in practice [35]. The algorithm combines Strassen's recursive algorithm [38] with an inefficient algorithm based on Newton's method [26]. Overall, this reduces matrix inversion to a polylogarithmic number of matrix multiplications. Since the inefficient algorithm is only applied to small subproblems, the overall algorithm is efficient.

3 Selected Open Problems

To further underline that there is a lot to be done, we give a list of open problems selected for being quite fundamentally interesting and spanning a wide range of topics. The ordering is roughly from rather specific questions to quite general problem areas with a large number of concrete possible projects.

1. *Priority Queues:* Show probabilistic quality guarantees for the MultiQueues from [28] or design a comparably fast data structure with provable guarantees.
2. *Strongly Connected Components:* Is there a polylogarithmic time, work efficient algorithm for finding strongly connected components? (Similar questions can be asked for many graph problems.)
3. *Matchings:* Give a linear work polylogarithmic time parallel algorithm for $(1-\epsilon)$-approximation of weighted matchings – perhaps by parallelizing [12]. Even the unweighted case, or the $2/3-\epsilon$-approximation algorithms like [27] would be an interesting result.
4. *Data Exchange:* There has been intensive work on routing in a wide spectrum of networks. However, even very simple models have wide open problems. For example, consider a half-duplex fully connected model: any one of p processors can move a data packet to any other processor in one step. However, at any time, a processor can only be involved in a single communication. Let h denote the maximum number of packets any processor is involved in. Delivering the data directly is equivalent to edge coloring of multigraphs and thus takes about $\frac{3}{2}h$ steps in the worst case [36]. We show that routing the packets on detours can lower this to about $\frac{6}{5}h$ [30]. In the very special case that one fourth of the processors need not communicate at all, this can be reduced to h steps [2]. An interesting conjecture is that $\approx h$ steps also suffice as long as the total number of packets is at most $\frac{3}{8}hp$.
5. *Solving Systems of Linear Equations:* Is there a polylogarithmic time algorithm with work $O(n^3)$ that solves a system of equations $Ax = b$ in a comparably stable way as Gaussian elimination? The matrix inversion algorithm from [35] is not stable enough for all applications, in particular if A is not symmetric. On the other hand, Gaussian elimination is P-complete [42].
6. *Compressed Text Indexing:* Develop a polylogarithmic time work-efficient algorithm for constructing compressed suffix arrays or related data structures.
7. *Parallel Paging:* A lot of work has been done on the sequential paging problem [6] – given a single sequence of data block accesses, decide, which of them should be kept in cache at what time to minimize the number of block transfers between slow memory and cache. Very little is known on parallel paging. For example, given n such sequences representing tasks, schedule them on p processors such that the bottleneck number of block transfers is minimized. We may want to distinguish between shared and private caches, online and offline strategies, ...
8. *Energy Efficient Computing:* Find a simple model with high predictive value for the actual energy efficiency of (parallel) algorithms.

9. *Communication Efficient Algorithms:* For your favorite algorithmic problem, find out whether there exists a communication efficient parallel algorithm for it.
10. *Parallel Streaming Algorithms:* Assume data arrives in data streams to p processors with limited *local* memory. How can you approximate important information about the data while keeping the communication volume small? The concrete problem considered could be any problem previously considered in sequential streaming algorithms.

References

1 Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. (semi-)external algorithms for graph partitioning and clustering. In *Alenex 2015*, 2015.

2 Eric Anderson, Joseph Hall, Jason Hartline, Mick Hobbes, Anna Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. Algorithms for data migration. *Algorithmica*, 57(2):349–380, 2010.

3 Richard J. Anderson, Ernst W. Mayr, and Manfred K. Warmuth. Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277, 1989.

4 Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *20th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 197–206. ACM, 2008.

5 Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting–basic algorithmic ideas. *arXiv preprint arXiv:1410.6754*, 2014.

6 A. L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

7 Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *arXiv preprint arXiv:1403.2056*, 2014.

8 Timo Bingmann and Peter Sanders. Parallel string sample sort. In *21st European Symposium on Algorithme (ESA)*, volume 8125 of *LNCS*, pages 169–180. Springer, 2013.

9 Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In *Europar*, LNCS. Springer, 2013.

10 Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

11 R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, pages 195–208, Toulouse, 2004.

12 Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *51st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 673–682. IEEE, 2010.

13 Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced B-trees with bulk updates. In *13th Symposium on Experimental Algorithms (SEA)*, 2014.

14 Irene Finocchi, Fabrizio Grandoni, and Giuseppe F Italiano. Designing reliable algorithms in unreliable memories. In *Algorithms–ESA 2005*, volume 3669 of *LNCS*, pages 1–8. Springer, 2005.

15 Patrick Flick, Peter Sanders, and Jochen Speck. Malleable sorting. In *27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 418–426. IEEE, 2013.

16 Michael Flynn and Patrick Hung. Microprocessor design issues: thoughts on the road ahead. *Micro, IEEE*, 25(3):16–31, 2005.

17 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

18 Monika Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. Computing on data streams. *External Memory Algorithms: DIMACS Workshop External Memory and Visualization, May 20-22, 1998*, 50:107, 1999.

19 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

20 Peter Kogge et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report CSE-TR-2008-13, U. of Notre Dame / DARPA IPTO, 2008.

21 Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, 100(10):892–901, 1985.

22 Ernst W. Mayr. Parallel approximation algorithms. Technical Report STm-CS-88-1225, Stanford University, 1988.

23 U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.

24 Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2015. to appear.

25 Rolf Niedermeier. *Invitation to fixed-parameter algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, Oxford, 2006.

26 Victor Pan and John Reif. Fast and efficient parallel solution of dense linear systems. *Computers & Mathematics with Applications*, 17(11):1481 – 1491, 1989.

27 S. Pettie and P. Sanders. A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004.

28 Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *CoRR*, abs/1411.1209, 2014.

29 P. Sanders. Algorithm engineering – an attempt at a definition. In *Efficient Algorithms*, volume 5760 of *LNCS*, pages 321–340. Springer, 2009.

30 P. Sanders and R. Solis-Oba. How helpers hasten h-relations. *Journal of Algorithms*, 41:86–98, 2001.

31 P. Sanders and J. Speck. Exact parallel malleable scheduling for tasks with concave speedup fun ctions. In *25th IEEE International Parallel and Distributed Processing Symposium*, pages 1156–1166, 2011.

32 Peter Sanders and Lawrence Mandow. Parallel label-setting multi-objective shortest path search. In *27th IEEE International Parallel & Distributed Processing Symposium*, pages 215–224, 2013.

33 Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *IEEE Int. Conf. on Big Data*, 2013.

34 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms*, pages 164–175. Springer Berlin Heidelberg, 2013.

35 Peter Sanders, Jochen Speck, and Raoul Steffen. Work-efficient matrix inversion in polylogarithmic time. In *25th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 214–221. ACM, 2013.

36 C. E. Shannon. A theorem on colouring lines of a network. *J. Math. Phys.*, 28(148–151), 1949.

37 D. Spielman and S.-H. Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

38 Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.

39 Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of Bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1):131–155, 2012.

40 TOP500 lists. http://www.top500.org. for data 1993–, Accessed: 2014-12-04.

41 L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1994.

42 Stephen A Vavasis. Gaussian elimination with pivoting is p-complete. *SIAM journal on discrete mathematics*, 2(3):413–423, 1989.

43 Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.

44 Wikipedia – list of Nvidia graphics processing units. en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units. Accessed: 2014-12-04, sum of number of cores for different purposes.

45 Wikipedia – list of Intel Xeon microprocessors. en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors. Accessed: 2014-12-04.