Algorithm Engineering for Parallel Computation

David A. Bader[1], Bernard M.E. Moret[1], and Peter Sanders[2]

[1] Departments of Electrical and Computer Engineering, and Computer Science, University of New Mexico, Albuquerque, NM 87131 USA. dbader@eece.unm.edu, moret@cs.unm.edu.

[2] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, sanders@mpi-sb.mpg.de.

Abstract. The emerging discipline of algorithm engineering has primarily focussed on transforming pencil-and-paper *sequential* algorithms into robust, efficient, well tested, and easily used implementations. As parallel computing becomes ubiquitous, we need to extend algorithm engineering techniques to parallel computation. Such an extension adds significant complications. After a short review of algorithm engineering achievements for sequential computing, we review the various complications caused by parallel computing, present some examples of successful efforts, and give a personal view of possible future research.

1 Introduction

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, analysis 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)*. 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* [54], 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 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. For instance, in a recent demonstration of the power of high-performance algorithm engineering, a million-fold speed-up 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) [53]. (In a further demonstration of algorithm engineering, further 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 [55].)

All of the tools and techniques developed over the last five 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 [72, 75], virtual address translation [65], and increasingly deep hierarchies of high-performance machines [31]. A key question is whether we can find simple models as a basis for algorithm development. For example, cache-oblivious algorithms [31] 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).

Ten years ago, David Bailey presented a catalog of ironic suggestions in "Twelve ways to fool the masses when giving performance results on parallel computers" [13], which drew from his unique experience managing the NAS Parallel Benchmarks [12], a set of pencil-and-paper benchmarks used to compare parallel computers on numerical kernels and applications. Bailey's "pet peeves," particularly concerning abuses in the reporting of performance results, are quite insightful. (While some items are technologically outdated, they still prove useful for comparisons and reports on parallel performance.) We rephrase several of his observations into guidelines in the framework of the broader issues discussed here, such as accurately measuring and reporting the details of the performed experiments, providing fair and portable comparisons, and presenting the empirical results in a meaningful fashion.

This paper is organized as follows. Section 2 introduces the important issues in high-performance algorithm engineering. Section 3 defines terms and concepts often used to describe and characterize the performance of parallel algorithms in the literature and discusses anomalies related to parallel speedup. Section 4 addresses the problems involved in fairly and reliably measuring the execution time of a parallel program—a difficult task because the processors operate asynchronously and thus communicate nondeterministically (whether through shared-memory or interconnection networks), Section 5 presents our thoughts on the choice of test instances: size, class, and data layout in memory. Section 6 briefly reviews the presentation of results from experiments in parallel computation. Section 7 looks at the possibility of taking truly machine-independent measurements. Finally, Section 8 discusses ongoing work in high-performance algorithm engineering for symmetric multiprocessors that promises to bridge the gap between the theory and practice of parallel computing. In an appendix, we briefly discuss 10 specific examples of published work in algorithm engineering for parallel computation.

2 General Issues

Parallel computer architectures come in a wide range of designs. While any given parallel machine can be classified in a broad taxonomy (for instance, as distributed memory or shared memory), experience has shown that each platform is unique, with its own artifacts, constraints, and enhancements. For example, the Thinking Machines CM-5, a distributed-memory computer, is interconnected by a fat-tree data network [48], but includes a separate network that can be used for fast barrier synchronization. The SGI Origin [47] provides a global address space to its shared memory; however, its non-uniform memory access requires the programmer to handle data placement for efficient performance. Distributed-memory cluster computers today range from low-end Beowulf-class machines that interconnect PC computers using commodity technologies like Ethernet [18, 76] to high-end clusters like the NSF Terascale Computing System at Pittsburgh Supercomputing Center, a system with 750 4-way AlphaServer nodes interconnected by Quadrics switches.

Most modern parallel computers are programmed in single-program, multiple-data (SPMD) style, meaning that the programmer writes one program that runs concurrently on each processor. The execution is specialized for each processor by using its processor identity (id or rank). Timing a parallel application requires capturing the elapsed wall-clock time of a program (instead of measuring CPU time as is the common practice in performance studies for sequential algorithms). Since each processor typically has its own clock, timing suite, or hardware performance counters, each processor can only measure its own view of the elapsed time or performance by starting and stopping its own timers and counters.

High-throughput computing is an alternative use of parallel computers whose objective is to maximize the number of independent jobs processed per unit of time. Condor [49], Portable Batch System (PBS) [56], and Load-Sharing Facility (LSF) [62], are examples of available queuing and scheduling packages that allow a user to easily broker tasks to compute farms and to various extents balance the resource loads, handle heterogeneous systems, restart failed jobs, and provide authentication and security. *High-performance* computing, on the other hand, is primarily concerned with optimizing the speed at which a single task executes on a parallel computer. For the remainder of this paper, we focus entirely on high-performance computing that requires non-trivial communication among the running processors.

Interprocessor communication often contributes significantly to the total running time. In a cluster, communication typically uses data networks that may suffer from congestion, nondeterministic behavior, routing artifacts, etc. In a shared-memory machine, communication through coordinated reads from and writes to shared memory can also suffer from congestion, as well as from memory coherency overheads, caching effects, and memory subsystem policies. Guaranteeing that the repeated execution of a parallel (or even sequential!) program will be identical to the prior execution is impossible in modern machines, because the state of each cache cannot be determined *a priori*—thus affecting relative memory access times—and because of nondeterministic ordering of instructions due to out-of-order execution and run-time processor optimizations.

Parallel programs rely on communication layers and library implementations that often figure prominently in execution time. Interprocessor messaging in scientific and technical computing predominantly uses the Message-Passing Interface (MPI) standard [51], but the performance on a particular platform may depend more on the implementation than on the use of such a library. MPI has several implementations as open source and portable versions such as MPICH [33] and LAM [60], as well as native, vendor implementations from Sun Microsystems and IBM. Shared-memory programming may use POSIX threads [64] from a freely-available implementation (e.g., [57]) or from a commercial vendor's platform. Much attention has been devoted lately to OpenMP [61], a standard for compiler directives and runtime support to reveal algorithmic concurrency and thus take advantage of shared-memory architectures; once again, implementations of OpenMP are available both in open source and from commercial vendors. There are also several higher-level parallel programming abstractions that use MPI, OpenMP, or POSIX threads, such as implementations of the Bulk-Synchronous Parallel (BSP) model [77, 43, 22] and data-parallel languages like High-Performance Fortran [42]. Higher-level application framework such as KeLP [29] and POOMA [27] also abstract away the details of the parallel communication layers. These frameworks enhance the expressiveness of data-parallel languages by providing the user with a high-level programming abstraction for block-structured scientific calculations. Using object-oriented techniques, KeLP and POOMA contain runtime support for non-uniform domain decomposition that takes into consideration the two main levels (intra- and inter-node) of the memory hierarchy.

3 Speedup

3.1 Why speed?

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 (such as long-term global climate change using satellite image processing [83]), 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.

3.2 What is speed?

With sequential codes, the performance indicator is running time, measured by CPU time as a function of input size. With parallel computing we focus not just on running time, but also on how the additional resources (typically processors) affect this running time. Questions such as "does using twice as many processors cut the running time in half?" or "what is the maximum number of processors that this computation can use efficiently?" can be answered by plots of the performance *speedup*. The *absolute speedup* is the ratio of the running time of the fastest known sequential implementation to that of the parallel running time. The fastest parallel algorithm often bears little resemblance to the fastest sequential algorithm and is typically much more complex; thus running the parallel implementation on one processor often takes much longer than running the sequential algorithm—hence the need to compare to the sequential, rather than the parallel, version. Sometimes, the parallel algorithm reverts to a good sequential algorithm if the number of processors is set to one. In this case it is acceptable to report *relative speedup*, i.e., the speedup of the p-processor version relative to the 1-processor version of the same implementation. But even in that case, the 1-processor version must make all of the obvious optimizations, such as eliminating unnecessary data copies between steps, removing self communications, skipping precomputing phases, removing collective communication broadcasts and result collection, and removing all locks and synchronizations. Otherwise, the relative speedup may present an exaggeratedly rosy picture of the situation. *Efficiency*, the ratio of the speedup to the number of processors, measures the effective use of processors in the parallel algorithm and is useful when determining how well an application scales on large numbers of processors. In any study that presents speedup values, the methodology should be clearly and unambiguously explained—which brings us to several common errors in the measurement of speedup.

3.3 Speedup anomalies

Occasionally so-called *superlinear* speedups, that is, speedups greater than the number of processors,[1] cause confusion because such should not be possible by Brent's principle (a single processor can simulate a p-processor algorithm with a uniform slowdown factor of p). Fortunately, the sources of "superlinear" speedup are easy to understand and classify.

Genuine superlinear absolute speedup can be observed without violating Brent's principle if the space required to run the code on the instance exceeds the memory of the single-processor machine, but not that of the parallel machine. In such a case, the sequential code swaps to disk while the parallel code does not, yielding an enormous and entirely artificial slowdown of the sequential code. On a more modest scale, the same problem could occur one level higher in the memory hierarchy, with the sequential code constantly cache-faulting while the parallel code can keep all of the required data in its cache subsystems.

A second reason is that the running time of the algorithm strongly depends on the particular input instance and the number of processors. For example, consider searching for a given element in an unordered array of $n \gg p$ elements. The sequential algorithm simply examines each element of the array in turn until the given element is found. The parallel approach may assume that the array is already partitioned evenly among the processors and has each processor proceed as in the sequential version, but using only its portion of the array, with the first processor to find the element halting the execution. In an experiment in which the item of interest always lies in position $n - n/p + 1$, the sequential algorithm always takes $n - n/p$ steps, while the parallel algorithm takes only *one* step, yielding a relative speedup of $n - n/p \gg p$. Although strange, this speedup does not violate Brent's principle, which only makes claims on the absolute speedup. Furthermore, such strange effects often disappear if one averages over all inputs. In the

[1] Strictly speaking, "efficiency larger than one" would be the better term.

example of array search, the sequential algorithm will take an expected $n/2$ steps and the parallel algorithm $n/(2p)$ steps, resulting in a speedup of p on average.

However, this strange type of speedup does *not* always disappear when looking at all inputs. A striking example is random search for satisfying assignments of a propositional logical formula in 3-CNF (conjunctive normal form with three literals per clause): Start with a random assignment of truth values to variables. In each step pick a random violated clause and make it satisfied by flipping a bit of a random variable appearing in it. Concerning the best upper bounds for its sequential execution time, little good can be said. However, Schöning [74] shows that one gets exponentially better expected execution time bounds if the algorithm is run in parallel for a huge number of (simulated) processors. In fact, the algorithm remains the fastest known algorithm for 3-SAT, exponentially faster than any other known algorithm. Brent's principle is not violated since the best sequential algorithm turns out to be the emulation of the parallel algorithm. The lesson one can learn is that parallel algorithms might be a source of good sequential algorithms too.

Finally, there are many cases were superlinear speedup is not genuine. For example, the sequential and the parallel algorithms may not be applicable to the same range of instances, with the sequential algorithm being the more general one—it may fail to take advantage of certain properties that could dramatically reduce the running time or it may run a lot of unnecessary checking that causes significant overhead. For example, consider sorting an unordered array. A sequential implementation that works on every possible input instance cannot be fairly compared with a parallel implementation that makes certain restrictive assumptions—such as assuming that input elements are drawn from a restricted range of values or from a given probability distribution, etc.

4 Reliable Measurements

The performance of a parallel algorithm is characterized by its running time as a function of the input data and machine size, as well as by derived measures such as speedup. However, measuring running time in a fair way is considerably more difficult to achieve in parallel computation than in serial computation.

In experiments with serial algorithms, the main variable is the choice of input datasets; with parallel algorithms, another variable is the machine size. On a single processor, capturing the execution time is simple and can be done by measuring the time spent by the processor in executing instructions from the user code—that is, by measuring *CPU time*. Since computation includes memory access times, this measure captures the notion of "efficiency" of a serial program—and is a much better measure than *elapsed wall-clock time* (using a system clock like a stopwatch), since the latter is affected by all other processes running on the system (user programs, but also system routines, interrupt handlers, daemons, etc.) While various structural measures help in assessing the behavior of an implementation, the CPU time is the definitive measure in a serial context [54].

In parallel computing, on the other hand, we want to measure how long the entire parallel computer is kept busy with a task. A parallel execution is characterized by the time elapsed from the time the first processor started working to the time the last processor completed, so we cannot measure the time spent by just one of the processors—such a measure would be unjustifiably optimistic! In any case, because data communication between processors is not captured by CPU time and yet is often a significant component of the parallel running time, we need to measure not just the time spent executing user instructions, but also waiting for barrier synchronizations, completing message transfers, and any time spent in the operating system for message handling and other ancillary support tasks. For these reasons, the use of elapsed wall-clock time is mandatory when testing a parallel implementation. One way to measure this time is to synchronize all processors after the program has been started. Then one processor starts a timer. When the processors have finished, they synchronize again and the processor with the timer reads its content.

Of course, because we are using elapsed wall-clock time, other running programs on the parallel machine will inflate our timing measurements. Hence, the experiments must be performed on an otherwise unloaded machine, by using dedicated job scheduling (a standard feature on parallel machines in any case) and by turning off unnecessary daemons on the processing nodes. Often, a parallel system has "lazy loading" of operating system facilities or one-time initializations the first time a specific function is called; in order not to add the cost of these operations to the running time of the program, several warm-up runs of the program should be made (usually internally within the executable rather than from an external script) before making the timing runs.

In spite of these precautions, the average running time might remain irreproducible. The problem is that, with a large number of processors, one processor is often delayed by some operating system event and, in a typical tightly synchronized parallel algorithm, the entire system will have to wait. Thus, even rare events can dominate the execution time, since their frequency is multiplied by the number of processors. Such problems can sometimes be uncovered by producing many fine-grained timings in many repetitions of the program run and then inspecting the histogram of execution times. A standard technique to get more robust estimates for running times than the average is to take the median. If the algorithm is randomized, one must first make sure that the execution time deviations one is suppressing are really caused by external reasons. Furthermore, if individual running times are not at least two to three orders of magnitude larger than the clock resolution, one should not use the median but the average of a filtered set of execution times where the largest and smallest measurements have been thrown out.

When reporting running times on parallel computers, all relevant information on the platform, compilation, input generation, and testing methodology, must be provided to ensure repeatability (in a statistical sense) of experiments and accuracy of results.

5 Test Instances

The most fundamental characteristic of a scientific experiment is reproducibility. Thus the instances used in a study must be made available to the community. For this reason, a common format is crucial. Formats have been more or less standardized in many areas of Operations Research and Numerical Computing. The DIMACS Challenges have resulted in standardized formats for many types of graphs and networks, while the library of Traveling Salesperson instances, TSPLIB, has also resulted in the spread of a common format for TSP instances. The CATS project [32] aims at establishing a collection of benchmark datasets for combinatorial problems and, incidentally, standard formats for such problems.

A good collection of datasets must consist of a mix of real and generated (artificial) instances. The former are of course the "gold standard," but the latter help the algorithm engineer in assessing the weak points of the implementation with a view to improving it. In order to provide a real test of the implementation, it is essential that the test suite include sufficiently large instances. This is particularly important in parallel computing, since parallel machines often have very large memories and are almost always aimed at the solution of large problems; indeed, so as to demonstrate the efficiency of the implementation for a large number of processors, one sometimes has to use instances of a size that exceeds the memory size of a uniprocessor. On the other hand, abstract asymptotic demonstrations are not useful: there is no reason to run artificially large instances that clearly exceed what might arise in practice over the next several years. (Asymptotic analysis can give us fairly accurate predictions for very large instances.) Hybrid problems, derived from real datasets through carefully designed random permutations, can make up for the dearth of real instances (a common drawback in many areas, where commercial companies will not divulge the data they have painstakingly gathered).

Scaling the datasets is more complex in parallel computing than in serial computing, since the running time also depends on the number of processors. A common approach is to scale up instances linearly with the number of processors; a more elegant and instructive approach is to scale the instances so as to keep the efficiency constant, with a view to obtain isoefficiency curves.

A vexing question in experimental algorithmics is the use of worst-case instances. While the design of such instances may attract the theoretician (many are highly nontrivial and often elegant constructs), their usefulness in characterizing the practical behavior of an implementation is dubious. Nevertheless, they do have a place in the arsenal of test sets, as they can test the robustness of the implementation or the entire system—for instance, an MPI implementation can succumb to network congestion if the number of messages grows too rapidly, a behavior that can often be triggered by a suitably crafted instance.

6 Presenting Results

Presenting experimental results for high-performance algorithm engineering should follow the principles used in presenting results for sequential computing. But there are additional difficulties. One gets an additional parameter with the number of processors used and parallel execution times are more platform dependent. McGeoch and Moret discuss the presentation of experimental results in the article "How to Present a Paper on Experimental Work with Algorithms" [50]. The key entries include

- describe and motivate the specifics of the experiments
- mention enough details of the experiments (but do not mention too many details)
- draw conclusions and support them (but make sure that the support is real)
- use graphs, not tables—a graph is worth a thousand table entries
- use suitably normalized scatter plots to show trends (and how well those trends are followed)
- explain what the reader is supposed to see

This advice applies unchanged to the presentation of high-performance experimental results. A summary of more detailed rules for preparing graphs and tables can also be found in this volume.

Since the main question in parallel computing is one of scaling (with the size of the problem or with the size of the machine), a good presentation needs to use suitable preprocessing of the data to demonstrate the key characteristics of scaling in the problem at hand. Thus, while it is always advisable to give some absolute running times, the more useful measure will be speedup and, better, efficiency. As discussed under testing, providing an *ad hoc* scaling of the instance size may reveal new properties: scaling the instance with the number of processors is a simple approach, while scaling the instance to maintain constant efficiency (which is best done after the fact through sampling of the data space) is a more subtle approach.

If the application scales very well, efficiency is clearly preferable to speedup, as it will magnify any deviation from the ideal linear speedup: one can use a logarithmic scale on the horizontal scale without affecting the legibility of the graph—the ideal curve remains a horizontal at ordinate 1.0, whereas log-log plots tend to make everything appear linear and thus will obscure any deviation. Similarly, an application that scales well will give very monotonous results for very large input instances—the asymptotic behavior was reached early and there is no need to demonstrate it over most of the graph; what does remain of interest is how well the application scales with larger numbers of processors, hence the interest in efficiency. The focus should be on characterizing efficiency and pinpointing any remaining areas of possible improvement.

If the application scales only fairly, a scatter plot of speedup values as a function of the sequential execution time can be very revealing, as poor speedup is often data-dependent. Reaching asymptotic behavior may be difficult in such a case, so this is the right time to run larger and larger instances; in contrast, isoefficiency curves are not very useful, as very little data is available to define curves at high efficiency levels. The focus should be on understanding the reasons why certain datasets yield poor speedup and others good speedup, with the goal of designing a better algorithm or implementation based on these findings.

7 Machine-Independent Measurements?

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 [51], OpenMP [61], and High-Performance Fortran [42]), as well as the parallel algorithm packages (e.g., fast Fourier transforms using FFTW [30] or parallelized linear algebra routines in ScaLAPACK [24]), 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 [33], Pablo [66], and Vampir [58]). The SKaMPI benchmark [69] allows running-time predictions based on such measurements even if the target machine is not available for program development. For example, one can check the page of results[2] or ask a customer to run the benchmark on the target platform. 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.

8 High-performance algorithm engineering for shared-memory processors

Symmetric multiprocessor (SMP) architectures, in which several (typically 2 to 8) processors operate in a true (hardware-based) shared-memory environment and are packaged as a single machine, are becoming commonplace. Most high-end workstations are available with dual processors and some with four processors, while many of the new high-performance computers are clusters of SMP nodes, with from 2 to 64 processors per node. The ability to provide uniform shared-memory access to a significant number of processors in a single SMP node brings us much closer to the ideal parallel computer envisioned over 20 years ago by theoreticians, the *Parallel Random Access Machine (PRAM)* (see, e.g., [44, 67]) and thus might enable us at long last to take advantage of 20 years of research in PRAM algorithms for various irregular computations. Moreover, as more and more supercomputers use the SMP cluster architecture, SMP computations will play a significant role in supercomputing as well.

8.1 Algorithms for SMPs

While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work. The number of processors remains quite low compared to the polynomial number of processors assumed by the PRAM model. This difference by itself would not pose a great problem: we can easily initiate far more processes or threads than

[2] `http://liinwww.ira.uka.de/˜skampi/cgi-bin/run_list.cgi.pl`

we have processors. But we need algorithms with efficiency close to one and parallelism needs to be sufficiently coarse grained that thread scheduling overheads do not dominate the execution time. Another big difference is in synchronization and memory access: an SMP cannot support concurrent read to the same location by a thousand threads without significant slowdown and cannot support concurrent write at all (not even in the arbitrary CRCW model) because the unsynchronized writes could take place far too late to be used in the computation. In spite of these problems, SMPs provide much faster access to their shared-memory than an equivalent message-based architecture: even the largest SMP to date, the 106-processor "Starcat" Sun Fire E15000, has a memory access time of less than 300*ns* to its entire physical memory of 576GB, whereas the latency for access to the memory of another processor in a message-based architecture is measured in tens of microseconds—in other words, message-based architectures are 20–100 times slower than the largest SMPs in terms of their worst-case memory access times.

The Sun SMPs (the older "Starfire" [23] and the newer "Starcat") use a combination of large (16×16) data crossbar switches, multiple snooping buses, and sophisticated handling of local caches to achieve uniform memory access across the entire physical memory. However, there remains a large difference between the access time for an element in the local processor cache (below 5*ns* in a Starcat) and that for an element that must be obtained from memory (around 300*ns*)—and that difference increases as the number of processors increases.

8.2 Leveraging PRAM Algorithms for SMPs

Since current SMP architectures differ significantly from the PRAM model, we need a methodology for mapping PRAM algorithms onto SMPs. In order to accomplish this mapping we face four main issues: (i) change of programming environment; (ii) move from synchronous to asynchronous execution mode; (iii) sharp reduction in the number of processors; and (iv) need for cache awareness. We now describe how each of these issues can be handled; using these approaches, we have obtained linear speedups for a collection of nontrivial combinatorial algorithms, demonstrating nearly perfect scaling with the problem size and with the number of processors (from 2 to 32) [11].

Programming Environment: A PRAM algorithm is described by pseudocode parameterized by the index of the processor. An SMP program must add to this explicit synchronization steps—software barriers must replace the implicit lockstep execution of PRAM programs. A friendly environment, however, should also provide primitives for memory management for shared-buffer allocation and release, as well as for contextualization (executing a statement on only a subset of processors) and for scheduling n independent work statements implicitly to $p < n$ processors as evenly as possible.

Synchronization: The mismatch between the lockstep execution of the PRAM and the asynchronous nature of parallel architecture mandates the use of software barriers. In the extreme, a barrier can be inserted after each PRAM step to guarantee a lockstep synchronization—at a high level, this is what the BSP model does. However, many of these barriers are not necessary: concurrent read operations can proceed asynchronously, as can expression evaluation on local variables. What needs to be synchronized is the writing to memory—so that the next read from memory will be consistent among the processors. Moreover, a concurrent write must be serialized (simulated);

standard techniques have been developed for this purpose in the PRAM model and the same can be applied to the shared-memory environment, with the same $\log p$ slowdown.

Number of Processors: Since a PRAM algorithm may assume as many as $n^{O(1)}$ processors for an input of size n—or an arbitrary number of processors for each parallel step, we need to *schedule* the work on an SMP, which will always fall short of that resource goal. We can use the lower-level scheduling principle of the work-time framework [44] to schedule the $W(n)$ operations of the PRAM algorithm onto the fixed number p of processors of the SMP. In this way, for each parallel step k, $1 \leq k \leq T(n)$, the $W_k(n)$ operations are simulated in at most $W_k(n)/p+1$ steps using p processors. If the PRAM algorithm has $T(n)$ parallel steps, our new schedule has complexity of $O(W(n)/p+T(n))$ for any number p of processors. The work-time framework leaves much freedom as to the details of the scheduling, freedom that should be used by the programmer to maximize cache locality.

Cache-Awareness: SMP architectures typically have a deep memory hierarchy with multiple on-chip and off-chip caches, resulting currently in two orders of magnitude of difference between the best-case (pipelined preloaded cache read) and worst-case (non-cached shared-memory read) memory read times. A cache-aware algorithm must efficiently use both spatial and temporal locality in algorithms to optimize memory access time. While research into cache-aware sequential algorithms has seen early successes (see [54] for a review), the design for *multiple* processor SMPs has barely begun. In an SMP, the issues are magnified in that not only does the algorithm need to provide the best spatial and temporal locality to each processor, but the algorithm must also handle the system of processors and cache protocols. While some performance issues such as false sharing and granularity are well-known, no complete methodology exists for practical SMP algorithmic design. Optimistic preliminary results have been reported (e.g., [59, 63]) using OpenMP on an SGI Origin2000, cache-coherent non-uniform memory access (ccNUMA) architecture, that good performance can be achieved for several benchmark codes from NAS and SPEC through automatic data distribution.

9 Conclusions

Parallel computing is slowly emerging from its niche of specialized, expensive hardware and restricted applications to become part of everyday computing. As we build support libraries for desktop parallel computing or for newer environments such as large-scale shared-memory computing, we need tools to ensure that our library modules (or application programs built upon them) are as efficient as possible. Producing efficient implementations is the goal of algorithm engineering, which has demonstrated early successes in sequential computing. In this article, we have reviewed the new challenges to algorithm engineering posed by a parallel environment and indicated some of the approaches that may lead to solutions.

Acknowledgments

This work was supported in part by National Science Foundation grants CAREER ACI 00-93039 (Bader), ACI 00-81404 (Moret/Bader), DEB 99-10123 (Bader), EIA 01-21377 (Moret/Bader), EIA 01-13095 (Moret), and DEB 01-20709 (Moret/Bader), and by the Future and Emerging Technologies program of the EU under contract number IST-1999-14186 (Sanders).

References

[1] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31:1116–1127, 1988.

[2] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation. In *Proc. 7th Ann. Symp. Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, 1995. ACM.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Cros, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 2nd edition, 1995.

[4] D. A. Bader. An Improved Randomized Selection Algorithm With an Experimental Study. In *Proc. The 2nd Workshop on Algorithm Engineering and Experiments (ALENEX00)*, pages 115–129, San Francisco, CA, 2000. `www.cs.unm.edu/Conferences/ALENEX00/`.

[5] D. A. Bader, D. R. Helman, and J. JáJá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM J. Experimental Algorithmics*, 1(3):1–42, 1996. `www.jea.acm.org/1996/BaderPersonalized/`.

[6] D. A. Bader and J. JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. *J. Parallel & Distributed Comput.*, 35(2):173–190, 1996.

[7] D. A. Bader and J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection. In *Proc. 10th Int'l Parallel Processing Symp.*, pages 292–301, Honolulu, HI, 1996.

[8] D. A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). *J. Parallel & Distributed Comput.*, 58(1):92–108, 1999.

[9] D. A. Bader, J. JáJá, and R. Chellappa. Scalable Data Parallel Algorithms for Texture Synthesis Using Gibbs Random Fields. *IEEE Trans. Image Processing*, 4(10):1456–1460, 1995.

[10] D. A. Bader, J. JáJá, D. Harwood, and L. S. Davis. Parallel Algorithms for Image Enhancement and Segmentation by Region Growing with an Experimental Study. *J. Supercomputing*, 10(2):141–168, 1996.

[11] D.A. Bader, A.K. Illendula, B. M.E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[12] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Moffett Field, CA, 1994.

[13] D. H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputer Review*, 4(8):54–55, 1991.

[14] R.D. Barve and J.S. Vitter. A simple and efficient parallel disk mergesort. In *Proc. 11th Ann. Symp. Parallel Algorithms and Architectures*, pages 232–241, Saint Malo, France, 1999. ACM.

[15] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *Theoretical Computer Science*, 203(2):175–203, 1998.

[16] A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Priority Queue Operations and Selection for the BSP* Model. In *Proc. 2nd Int'l Euro-Par Conf.*, volume 1124 of *Lecture Notes in Computer Science*, pages 369–376, Lyon, France, 1996. Springer-Verlag.

[17] A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic Parallel Algorithms: Priority Queue Operations and Selection for the BSP* Model. In *Proc. 2nd Int'l Euro-Par Conf.*, pages 27–29, Lyon, France, 1996. LIP, Ecole Normale Supérier de Lyon.

[18] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *Proc. Int'l Conf. Parallel Processing*, volume 1, pages 11–14, 1995.

[19] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[20] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proc. Symp. Parallel Algorithms and Architectures*, pages 3–16. ACM, 1991.

[21] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[22] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library - design, implementation and performance. In *Proc. 13th Int'l Parallel Processing Symp. and the 10th Symp. Parallel and Distributed Processing (IPPS/SPDP)*, San Juan, Puerto Rico, 1999. `www.uni-paderborn.de/~pub/`.

[23] A. Charlesworth. Starfire: extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998.

[24] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *The 4th Symp. the Frontiers of Massively Parallel Computations*, pages 120–127, McLean, VA, 1992.

[25] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. Fast Parallel Sorting Under LogP: From Theory to Practice. In *Portability and Performance for Parallel Processing*, chapter 4, pages 71–98. John Wiley & Sons, 1993.

[26] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *4th Symp. Principles and Practice of Parallel Programming*, pages 1–12. ACM SIGPLAN, 1993.

[27] J. C. Cummings, J. A. Crotinger, S. W. Haney, W. F. Humphrey, S. R. Karmesin, J. V.W. Reynders, S. A. Smith, and T. J. Williams. Rapid application development and enhanced code interoperabily using the POOMA framework. In M. E. Henderson, C. R. Anderson, and S. L. Lyons, editors, *Proc. 1998 Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, chapter 29. SIAM, Yorktown Heights, NY, 1999.

[28] P. de la Torre and C.P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proc. 2nd Int'l Euro-Par Conf.*, pages 352–358, Lyon, France, 1996. LIP, Ecole Normale Supérier de Lyon.

[29] S. J. Fink and S. B. Baden. Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters. In Y. Ishikawa et al., editor, *Proc. 1997 Int'l Scientific Computing in Object-Oriented Parallel Environments Conf. (ISCOPE '97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 1–8, Marina del Ray, California, 1997. Springer-Verlag.

[30] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, 1998.

[31] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Ann. Symp. Foundations of Computer Science (FOCS-99)*, pages 285–297, New York, NY, 1999. IEEE Press.

[32] A.V. Goldberg and B.M.E. Moret. Combinatorial Algorithms Test Sets (CATS): the ACM/EATCS platform for experimental research. In *Proc. 10th Ann. Symp. Discrete Algorithms (SODA-99)*, pages 913–914, Baltimore, MD, 1999. ACM-SIAM.

[33] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. 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/`.

[34] S. E. Hambrusch and A. A. Khokhar. C^3: A Parallel Model for Coarse-grained Machines. *J. Parallel & Distributed Comput.*, 32:139–154, 1996.

[35] D. R. Helman, D. A. Bader, and J. JáJá. A Parallel Sorting Algorithm With an Experimental Study. Technical Report CS-TR-3549 and UMIACS-TR-95-102, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, 1995.

[36] D. R. Helman, D. A. Bader, and J. JáJá. Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study. In *Proc. 8th Ann. Symp. Parallel Algorithms and Architectures*, pages 211–220, Padua, Italy, 1996. ACM.

[37] D. R. Helman, D. A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm With an Experimental Study. *J. Parallel & Distributed Comput.*, 52(1):1–23, 1998.

[38] D. R. Helman and J. JáJá. Sorting on clusters of SMP's. In *Proc. 12th Int'l Parallel Processing Symp.*, pages 1–7, Orlando, FL, 1998.

[39] D. R. Helman and J. JáJá. Designing Practical Efficient Algorithms for Symmetric Multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, 1999. Springer-Verlag.

[40] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *J. Parallel & Distributed Comput.*, 61(2):265–278, 2001.

[41] D. R. Helman, J. JáJá, and D. A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. *ACM J. Experimental Algorithmics*, 3(4), 1997. `www.jea.acm.org/1998/HelmanSorting/`.

[42] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1.0 edition, 1993.

[43] J.M.D. Hill, B. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. Technical Report PRG-TR-29-97, Oxford University Computing Laboratory, 1997. `www.BSP-Worldwide.org/implmnts/oxtool/`.

[44] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[45] B. H.H. Juurlink and H. A.G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Trans. Computer Systems*, 13(3):271–318, 1998.

[46] S. N.V. Kalluri, J. JáJá, D. A. Bader, Z. Zhang, J. R.G. Townshend, and H. Fallah-Adl. High Performance Computing Algorithms for Land Cover Dynamics Using Remote Sensing Data. *Int'l J. Remote Sensing*, 21(6):1513–1536, 2000.

[47] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA'97)*, pages 241–251, Denver, CO, 1997.

[48] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *J. Parallel & Distributed Comput.*, 33(2):145–158, 199.

[49] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, San Jose, CA, 1998.

[50] C.C. McGeoch and B.M.E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.

[51] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, 1995. Version 1.1.

[52] F. Meyer auf der Heide and R. Wanka. Parallel bridging models and their impact on algorithm design. In *Proc. Int'l Conf. on Computational Science, Part II*, volume 2074 of *Lecture Notes in Computer Science*, pages 628–637, San Francisco, CA, 2001. Springer-Verlag.

[53] B.M.E. Moret, D.A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. *J. Supercomputing*, 22:99–111, 2002. Special issue on the best papers from ICCS'01.

[54] B.M.E. Moret and H.D. Shapiro. Algorithms and experiments: The new (and old) methodology. *J. Universal Computer Sci.*, 7(5):434–446, 2001.

[55] B.M.E. Moret, A.C. Siepel, J. Tang, and T. Liu. Inversion medians outperform breakpoint medians in phylogeny reconstruction from gene-order data. In *Proc. 2nd Workshop Algs. in Bioinformatics (WABI'02)*, volume 2542 of *Lecture Notes in Computer Science*, Rome, Italy, 2002. Springer-Verlag.

[56] MRJ Inc. The Portable Batch System (PBS). `pbs.mrj.com`.

[57] F. Müller. A Library Implementation of POSIX Threads under UNIX. In *Proc. 1993 Winter USENIX Conf.*, pages 29–41, San Diego, CA, 1993. `www.informatik.hu-berlin.de/~mueller/projects.html`.

[58] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: visualization and analysis of MPI resources. *Supercomputer 63*, 12(1):69–80, 1996.

[59] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is data distribution necessary in OpenMP. In *Proc. Supercomputing*, Dallas, TX, 2000. IEEE Press.

[60] Ohio Supercomputer Center. *LAM / MPI Parallel Computing*. The Ohio State University, Columbus, OH, 1995. `www.lam-mpi.org`.

[61] OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. `www.openmp.org`, 1997.

[62] Platform Computing Inc. The Load Sharing Facility (LSF). `www.platform.com`.

[63] E. D. Polychronopoulos, D. S. Nikolopoulos, T. S. Papatheodorou, X. Martorell, J. Labarta, and N. Navarro. An efficient kernel-level scheduling methodology for multiprogrammed shared memory multiprocessors. In *12th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS)*, Ft. Lauderdale, FL, 1999. ISCA.

[64] POSIX. *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*. Portable Applications Standards Committee of the IEEE, 1996-07-12 edition, 1996. ISO/IEC 9945-1, ANSI/IEEE Std. 1003.1.

[65] N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. In *Proc. The 2nd Workshop on Algorithm Engineering and Experiments (ALENEX00)*, pages 131–146, San Francisco, CA, 2000. `www.cs.unm.edu/Conferences/ALENEX00/`.

[66] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In A. Skjellum, editor, *Proc. Scalable Parallel Libraries Conf.*, pages 104–113, Mississippi State University, 1993. IEEE Computer Society Press.

[67] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[68] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In *EuroPVM/MPI see also `liinwww.ira.uka.de/˜skampi/`*, number 1497 in Lecture Notes in Computer Science, pages 52–59, 1998.

[69] R. Reussner, P. Sanders, and J. Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 2001. accepted, conference version with L. Prechelt and M. Müller in Proc. EuroPVM/MPI 1998.

[70] P. Sanders. *Load Balancing Algorithms for Parallel Depth First Search (In German: Lastverteilungsalgorithmen für parallele Tiefensuche)*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, Berlin, 1997.

[71] P. Sanders. Randomized priority queues for fast parallel access. *J. Parallel & Distributed Comput.*, 49(1):86–97, 1998. Special Issue on Parallel and Distributed Data Structures.

[72] P. Sanders. Accessing multiple sequences through set associative caches. In *Proc. 26th Int'l Colloquium on Automata, Languages and Programming (ICALP'99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 655–664, Prague, Czech Republic, 1999. Springer-Verlag.

[73] P. Sanders and T. Hansch. On the efficient implementation of massively parallel quicksort. In *Proc. 4th Int'l Workshop On Solving Irregularly Structured Problem In Parallel (IRREGULAR 1997)*, volume 1253 of *Lecture Notes in Computer Science*, pages 13–24, Paderborn, Germany, 1997. Springer-Verlag.

[74] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th IEEE Symp. Foundations of Computer Science*, pages 410–414, 1999.

[75] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proc. 11th Ann. Symp. Discrete Algorithms (SODA-00)*, pages 829–838, San Francisco, CA, 2000. ACM-SIAM.

[76] T.L. Sterling, J. Salmon, and D.J. Becker. *How to build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, Inc., Cambridge, MA, 1999.

[77] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[78] J. S. Vitter and E. A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[79] J. S. Vitter and E. A.M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994.

[80] R. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS). In *Proc. Supercomputing 98*, Orlando, FL, 1998. `www.netlib.org/utk/people/JackDongarra/PAPERS/atlas-sc98.ps`.

[81] H. A.G. Wijshoff and B. H.H. Juurlink. A quantitative comparison of parallel computation models. In *Proc. 8th Ann. Symp. Parallel Algorithms and Architectures*, pages 13–24, Padua, Italy, 1996. ACM.

[82] Y. Yan and X. Zhang. Lock bypassing: An efficient algorithm for concurrently accessing priority heaps. *ACM J. Experimental Algorithmics*, 3(3), 1998. `www.jea.acm.org/1998/YanLock/`.

[83] Z. Zhang, J. JáJá, D. A. Bader, S. Kalluri, H. Song, N. El Saleous, E. Vermote, and J. Townshend. Kronos: A Software System for the Processing and Retrieval of Large-Scale AVHRR Data Sets. *Photogrammetric Engineering & Remote Sensing*, 66(9):1073–1082, 2000.

A Examples of Algorithm Engineering for Parallel Computation

Within the scope of this paper, it would be difficult to provide meaningful and self-contained examples for each of the various points we made. In lieu of such target examples, we offer here several references[3] that exemplify the best aspects of algorithm engineering studies for high-performance and parallel computing. For each paper or collection of papers, we describe those aspects of the work that led to its inclusion in this section.

1. The authors' prior publications [53, 11, 4, 46, 8, 71, 68, 37, 41, 73, 36, 5, 10, 7, 6, 9] contain many empirical studies of parallel algorithms for combinatorial problems like sorting [5, 35, 41, 73, 36], selection [4, 71, 7], and priority queues [71], graph algorithms [53], backtrack search [70], and image processing [46, 10, 6, 9].
2. JáJá and Helman conducted empirical studies for prefix computations [40], sorting [38] and list-ranking [39] on symmetric multiprocessors. The sorting paper [38] extends Vitter's external Parallel Disk Model [1, 78, 79] 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.)
3. In [20, 21] 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 $32K$-processor CM-2.
4. Vitter *et al.* provide the canonical theoretic foundation for I/O-intensive experimental algorithmics using external parallel disks (e.g., see [1, 78, 79, 14]). Examples from sorting, FFT, permuting, and matrix transposition problems are used to demonstrate the parallel disk model. For instance, using this model in [14], empirical results are given for external sorting on a fixed number of disks with from 1 to 10 million items, and two algorithms are compared with overall time, number of merge passes, I/O streaming rates, using computers with different internal memory sizes.
5. Hambrusch and Khokhar present a model (C^3) for parallel computation that, for a given algorithm and target architecture, provides the complexity of computation, communication patterns, and potential communication congestion [34]. This paper is one of the first efforts to model collective communication both theoretically and through experiments, and then validate the model with coarse-grained computational applications on an Intel supercomputer. Collective operations are thoroughly

[3] We do not attempt to include all of the best work in the area: our selection is perforce idiosyncratic.

characterized by message size and higher-level patterns are then analyzed for communication and computation complexities in terms of these primitives.

6. While not itself an experimental paper, Meyer auf der Heide and Wanka demonstrate in [52] the impact of features of parallel computation models on the design of efficient parallel algorithms. The authors begin with an optimal multisearch algorithm for the Bulk Synchronous Parallel (BSP) model that is no longer optimal in realistic extensions of BSP that take critical blocksize into account such as BSP* (e.g., [17, 16, 15]). When blocksize is taken into account, the modified algorithm is optimal in BSP*. The authors present a similar example with a broadcast algorithm using a BSP model extension that measures locality of communication, called D-BSP [28].
7. Juurlink and Wijshoff [81, 45] 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.
8. The LogP model of Culler *et al.* [26] (and its extensions such as logGP [2] for long messages) provides a realistic model for designing parallel algorithms for message-passing platforms. Its use is demonstrated for a number of problems, including sorting [25]. Four parallel sorting algorithms are analyzed for LogP and their performance on parallel platforms with from 32 to 512 processors is predicted by LogP using parameter values for the machine. The authors analyze both regular and irregular communication and provide normalized predicted and measured running times for the steps of each algorithm.
9. Yun and Zhang [82] describe an extensive performance evaluation of lock bypassing for concurrent access to priority heaps. The empirical study compares three algorithms by reporting the average number of locks waited for in heaps of 255 and 512 nodes. The average hold operation times are given for the three algorithms for uniform, exponential, and geometric, distributions, with inter-hold operation delays of 0, 160, and 640μs.
10. 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 [24, 19], 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. In [24], for instance, experimental results are given for parallel LU factorization plotted in performance achieved (gigaflops per second) for various matrix sizes, with a different series for each machine configuration. Because ScaLAPACK relies on fast sequential linear algebra routines (e.g., LAPACK [3]), new approaches for automatically tuning the sequential library (e.g., LAPACK) are now available as the ATLAS package [80].