Chapter 5. Realistic Computer Models

Deepak Ajwani[⋆] and Henning Meyerhenke[⋆⋆]

5.1 Introduction

Many real-world applications involve storing and processing large amounts of data. These data sets need to be either stored over the memory hierarchy of one computer or distributed and processed over many parallel computing devices or both. In fact, in many such applications, choosing a realistic computation model proves to be a critical factor in obtaining practically acceptable solutions. In this chapter, we focus on realistic computation models that capture the running time of algorithms involving large data sets on modern computers better than the traditional RAM (and its parallel counterpart PRAM) model.

5.1.1 Large Data Sets

Large data sets arise naturally in many applications. We consider a few examples here.

– GIS terrain data: Remote sensing [435] has made massive amounts of high resolution terrain data readily available. NASA already measures the data volumes from satellite images in petabytes (10^{15} bytes). With the emergence of new terrain mapping technologies such as laser altimetry, this data is likely to grow much further. Terrain analysis is central to a range of important geographic information systems (GIS) applications concerned with the effects of topography.
– Data warehouses of companies that keep track of every single transaction on spatial/temporal databases. Typical examples include the financial sector companies, telecommunication companies and online businesses. Many data warehouse appliances already scale to one petabyte and beyond [428].
– The World Wide Web (WWW) can be looked upon as a massive graph where each web-page is a node and the hyperlink from one page to another is a directed edge between the nodes corresponding to those pages. As of August 2008, it is estimated that the indexed web contains at least 27 billion webpages [208].
  Typical problems in the analysis (e.g., [129, 509]) of WWW graphs include computing the diameter of the graph, computing the diameter of the core

⋆ Supported by German Science Foundation (DFG) grant ME 3250/1-1, DFG grant ME 2088/1-3, and by MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation

⋆⋆ Partially supported by German Science Foundation (DFG) Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo) and by DFG Priority Programme 1307 *Algorithm Engineering*.

of the graph, computing connected and strongly connected components and other structural properties such as computing the correct parameters for the power law modeling of WWW graphs. There has also been a lot of work on understanding the evolution of such graphs.

Internet search giants and portals work on very large datasets. For example, Yahoo!, a major Internet portal, maintains (as of 2008) a database of more than a petabyte [426].

- Social networks: Social networks provide yet another example of naturally evolving massive graphs [55]. One application area is citation graphs, in which nodes represent the papers and an edge from one paper to another shows the citation. Other examples include networks of friends, where nodes denote individuals and edges show the acquaintance, and telephone graphs, where nodes represent phone numbers and edges represent phone call in the last few days. Typical problems in social networks include finding local communities, e. g., people working on similar problems in citation graphs.
- Artificial Intelligence and Robotics: In applications like single-agent search, game playing and action planning, even if the input data is small, intermediate data can be huge. For instance, the state descriptors of explicit state model checking softwares are often so large that main memory is not sufficient for the lossless storage of reachable states during the exploration [267].
- Scientific modeling and simulation (e. g., particle physics, molecular dynamics), engineering (e. g., CAD), medical computing, astronomy and numerical computing.
- Network logs such as fault alarms, CPU usage at routers and flow logs. Typical problems on network logs include finding the number of distinct IP addresses using a given link to send their traffic or how much traffic in two routers is common.
- Ad hoc network of sensors monitoring continuous physical observations – temperature, pressure, EMG/ECG/EEG signals from humans, humidity etc.
- Weather prediction centers collect a massive amount of weather, hydrological, radar, satellite and weather balloon data and integrate it into a variety of computer models for improving the accuracy of weather forecasts.
- Genomics, where the sequence data can be as large as a few terabytes [111].
- Graphics and animations [281].

Note that the term “large” as used in this chapter is in comparison with the memory capacity and it depends not only on the level of memory hierarchy but also the computational device in use. For instance, road network of a small city may fit in the main memory of modern computers, but still be considered “large” for route planning applications involving a flash memory card on a small mobile device like Pocket PC [342, 699] or in the context of cache misses.

Next, we consider the traditional RAM model of computation and the reasons for its inadequacy for applications involving large data sets.

5.1.2 RAM Model

The running time of an algorithm is traditionally analyzed by counting the number of executed primitive operations or "instructions" as a function of the input size n (cf. Chapter 4). The implicit underlying model of computation is the one-processor, *random-access machine (RAM)* model. The RAM model or the "von Neumann model of computation" consists of a computing device attached to a storage device (or "memory"). The following are the key assumptions of this model:

– Instructions are executed one after another, with no concurrent operations.
– Every instruction takes the same amount of time, at least up to small constant factors.
– Unbounded amount of available memory.
– Memory stores words of size $O(\log n)$ bits where n is the input size.
– Any desired memory location can be accessed in unit time.
– For numerical and geometric algorithms, it is sometimes also assumed that words can represent real numbers accurately.
– Exact arithmetic on arbitrary real numbers can be done in constant time.

The above assumptions greatly simplify the analysis of algorithms and allow for expressive asymptotic analysis.

5.1.3 Real Architecture

Unfortunately, modern computer architecture is not as simple. Rather than having an unbounded amount of unit-cost access memory, we have a hierarchy of storage devices (Figure 5.1) with very different access times and storage capacities. Modern computers have a microprocessor attached to a file of *registers*. The *first level (L1) cache* is usually only a few kilobytes large and incurs a delay of a few clock cycles. Often there are separate L1 caches for instructions and data. Nowadays, typical *second level (L2) cache* has a size of about 32-512 KB and access latencies around ten clock cycles. Some processors also have a rather expensive *third level (L3) cache* of up to 256 MB made of fast static random access memory cells. A cache consists of *cache lines* that each store a number of memory words. If an accessed item is not in the cache, it and its neighbor entries are fetched from the main memory and put into a cache line. These caches usually have limited associativity, i. e., an element brought from the main memory can be placed only in a restricted set of cache lines. In a *direct-mapped* cache the target cache line is fixed and only based on the memory address, whereas in a *full-associative* cache the item can be placed anywhere. Since the former is too restrictive and the latter is expensive to build and manage, a compromise often used is a *set-associative* cache. There, the item's memory address determines a fixed set of cache lines into which the data can be mapped, though within each set, any cache line can be used. The typical size of such a set of cache lines is a power of 2 in the range from 2 to 16. For more details about the structure of caches the interested reader is referred to [631] (in particular its Chapter 7).

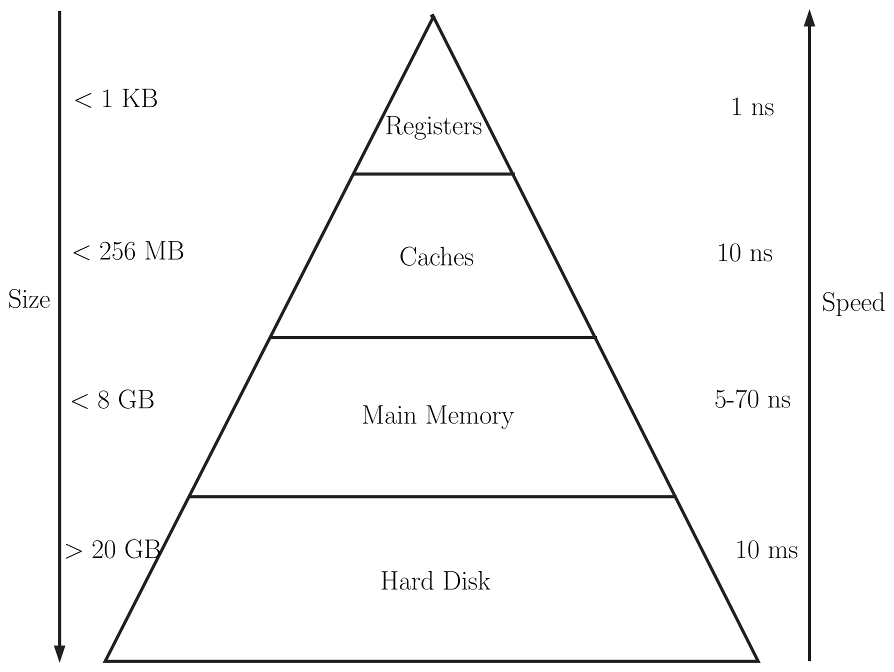

Figure 5.1. Memory hierarchy in modern computer architecture.

The *main memory* is made of dynamic random access memory cells. These cells store a bit of data as a charge in a capacitor rather than storing it as the state of a flip-flop which is the case for most static random access memory cells. It requires practically the same amount of time to access any piece of data stored in the main memory, irrespective of its location, as there is no physical movement (e. g., of a reading head) involved in the process of retrieving data. Main memory is usually volatile, which means that it loses all data when the computer is powered down. At the time of the writing, the main memory size of a PC is usually between 512 MB and 32 GB and a typical RAM memory has an access time of 5 to 70 nanoseconds.

Magnetic *hard disks* offer cheap non-volatile memory with an access time of 10 ms, which is 10^6 times slower than a register access. This is because it takes very long to move the access head to a particular track of the disk and wait until the disk rotates into the seeked position. However, once the head starts reading or writing, data can be transferred at the rate of 35-125 MB/s. Hence, reading or writing a contiguous block of hundreds of KB takes only about twice as long as accessing a single byte, thereby making it imperative to process data in large chunks.

Apart from the above mentioned levels of a memory hierarchy, there are instruction pipelines, an instruction cache, logical/physical pages, the translation lookaside buffer (TLB), magnetic tapes, optical disks and the network, which further complicate the architecture.

The reasons for such a memory hierarchy are mainly economical. The faster memory technologies are costlier and, as a result, fast memories with large capacities are economically prohibitive. The memory hierarchy emerges as a reasonable compromise between the performance and the cost of a machine.

Microprocessors like Intel Xeon have multiple register sets and are able to execute a corresponding number of threads of activity in parallel, even as they share the same execution pipeline. The accumulated performance is higher, as a thread can use the processor while another thread is waiting for a memory access to finish.

Explicit parallel processing takes the computer architecture further away from the RAM model. On parallel machines, some levels of the memory hierarchy may be shared whereas others are distributed between the processors. The communication cost between different machines is often the bottleneck for algorithms on parallel architectures.

5.1.4 Disadvantages of the RAM Model

The beauty of the RAM model lies in the fact that it hides all the 'messy' details of computer architecture from the algorithm designer. At the same time, it encapsulates the comparative performance of algorithms remarkably well. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. The performance guarantees in the RAM model are not architecture-specific and therefore robust. However, this is also the limiting factor for the success of this model. In particular, it fails significantly when the input data or the intermediate data structure is too large to reside completely within the internal memory. This failure can be observed between any two levels of the memory hierarchy.

For most problems on large data sets, the dominant part of the running time of algorithms is not the number of "instructions", but the time these algorithms spend waiting for the data to be brought from the hard disk to internal memory. The I/Os or the movement of data between the memory hierarchies (and in particular between the main memory and the disk) are not captured by the RAM model and hence, as shown in Figure 5.2, the predicted performance on the RAM model increasingly deviates from the actual performance. As we will see in Section 5.5.2, the running times of even elementary graph problems like breadth-first search become I/O-dominant as the input graph is just twice as large as the available internal memory. While the RAM model predicts running time in *minutes*, it takes *hours* in practice.

Since the time required by algorithms for large data sets in the sequential setting can be impractical, a larger number of processors are sometimes used to compute the solution in parallel. On parallel architectures, one is often interested in the parallel time, work, communication costs etc. of an algorithm. These performance parameters are simply beyond the scope of the traditional one-processor RAM model. Even the parallel extension of the RAM model, the

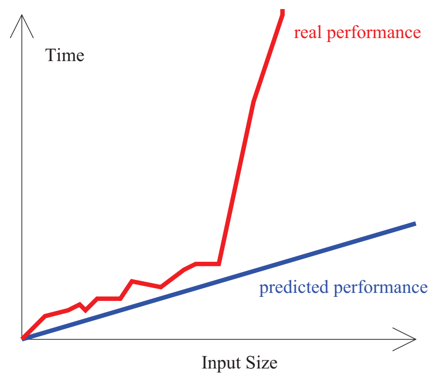

Figure 5.2. Predicted performance of RAM model versus its real performance.

PRAM model, fails to capture the running time of algorithms on real parallel architectures as it ignores the communication cost between the processors.

5.1.5 Future Trends

The problem is likely to aggravate in the future. According to Moore's law, the number of transistors double every 18 months. As a result, the CPU speed continued to improve at nearly the same pace until recently, i.e., an average performance improvement of 1% per week. Meanwhile, due to heat problems caused by even higher clock speeds, processor architects have passed into increasing the number of computing entities (*cores*) per processor instead. The usage of parallel processors and multi-cores makes the computations even faster. On the other hand, random access memory speeds and hard drive seek times improve at best a few percentages per year. Although the capacity of the random access memory doubles about every two years, users double their data storage every 5 months. Multimedia (pictures, music and movies) usage in digital form is growing and the same holds true for the content in WWW. For example, the number of articles in the online encyclopedia Wikipedia has been doubling every 339 days [830] and the online photo sharing network Flickr that started in 2004 had more than three billion pictures as of November 2008 [289] and claims that three to five million photos are updated daily on its network. Consequently, the problem sizes are increasing and the I/O-bottleneck is worsening.

5.1.6 Realistic Computer Models

Since the RAM model fails to capture the running time of algorithms for problems involving large data sets and the I/O bottleneck is likely to worsen in future, there is clearly a need for realistic computer models – models taking explicit care of memory hierarchy, parallelism or other aspects of modern architectures. These models should be simple enough for algorithm design and analysis, yet they should be able to capture the intricacies of the underlying architecture. Their performance metric can be very different from the traditional "counting the instructions" approach of the RAM model and algorithm design on these models may need fundamentally different techniques. This chapter introduces some of the popular realistic computation models – external memory model, parallel disk model, cache-oblivious model, and parallel bridging models like BSP, LogP, CGM, QSM etc. – and provides the basic techniques for designing algorithms on most of these models.

In Section 5.2, many techniques for exploiting the memory hierarchy are introduced. This includes different memory hierarchy models, algorithm design techniques and data structures as well as several optimization techniques specific to caches. After the introduction of various parallel computing models in Section 5.3, Section 5.4 shows the relationship between the algorithms designed in memory hierarchy and parallel models. In Section 5.5, we discuss success stories of Algorithm Engineering on large data sets using the introduced computer models from various domains of computer science.

5.2 Exploiting the Memory Hierarchy

5.2.1 Memory Hierarchy Models

In this section, we introduce some of the memory hierarchy models that have led to successful Algorithm Engineering on large data sets.

External Memory Model. The I/O model or the external memory (*EM*) model (depicted in Figure 5.3) as introduced by Aggarwal and Vitter [11] assumes a single central processing unit and two levels of memory hierarchy. The internal memory is fast, but has a limited size of M words. In addition, we have an external memory which can only be accessed using I/Os that move B contiguous words between internal and external memory. For some problems, the notation is slightly abused and we assume that the internal memory can have up to M *data items of a constant size* (e. g., vertices/edges/characters/segments etc.) and in one I/O operation, B contiguous data items move between the two memories. At any particular timestamp, the computation can only use the data already present in the internal memory. The measure of performance of an algorithm is the number of I/Os it performs. An algorithm A has lower I/O-complexity than another algorithm A' if A requires less I/Os than A'.

Although we mostly use the sequential variant of the external memory model, it also has an option to express disk parallelism. There can be D parallel disks and in one I/O, D arbitrary blocks can be accessed in parallel from the disks. The usage of parallel disks helps us alleviate the I/O bottleneck.

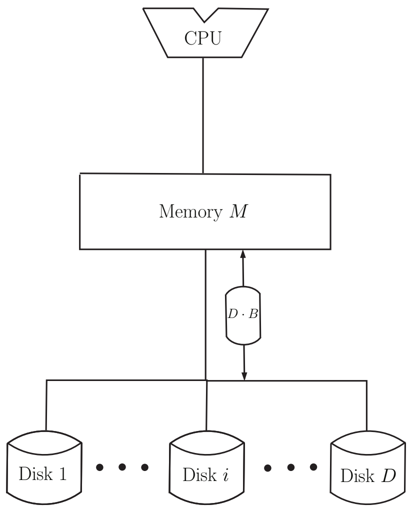

Figure 5.3. The external memory model.

Parallel Disk Model. The parallel disk model (depicted in Figure 5.4) by Vitter and Shriver [810] is similar to the external memory model, except that it adds a realistic restriction that only one block can be accessed per disk during an I/O, rather than allowing D *arbitrary* blocks to be accessed in parallel. The parallel disk model can also be extended to allow parallel processing by allowing P parallel identical processors each with M/P internal memory and equipped with D/P disks.

Sanders et al. [696] gave efficient randomized algorithms for emulating the external memory model of Aggarwal and Vitter [11] on the parallel disk model.

Ideal Cache Model. In the external memory model we are free to choose any two levels of the memory hierarchy as internal and external memory. For this reason, external memory algorithms are sometimes also referred to as *cache-aware* algorithms ("aware" as opposed to "oblivious"). There are two main problems with extending this model to caches: limited associativity and automated replacement. As shown by Sen and Chatterjee [724], the problem of limited associativity in caches can be circumvented at the cost of constant factors. Frigo et al. [308] showed that a regular algorithm causes asymptotically the same number of cache misses with LRU (least recently used) or FIFO (first-in first-out) replacement policy as with optimal off-line replacement strategy. Intuitively, an algorithm is called *regular* if the number of incurred cache misses (with an optimal off-line replacement) increase by a constant factor when the cache size is reduced to half.

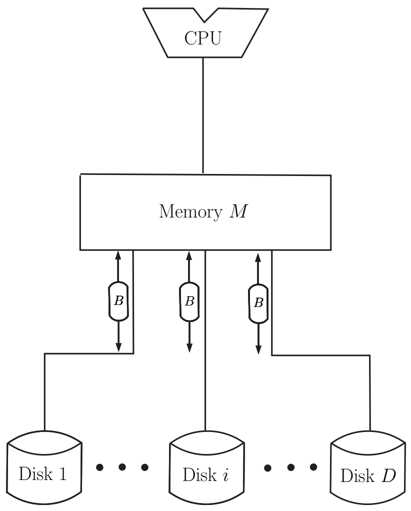

Figure 5.4. The parallel disk model.

Similar to the external memory model, the ideal cache model [308] assumes a two level memory hierarchy, with the faster level having a capacity of storing at most M elements and data transfers in chunks of B elements. In addition, it also assumes that the memory is managed automatically by an optimal offline cache-replacement strategy, and that the cache is fully associative.

Cache-Oblivious Model. In practice, the model parameters B and M need to be finely tuned for an optimal performance. For different architectures and memory hierarchies, these values can be very different. This fine-tuning can be at times quite cumbersome. Besides, we can optimize only one memory hierarchy level at a time. Ideally, we would like a model that would capture the essence of the memory hierarchy without knowing its specifics, i. e., values of B and M, and at the same time is efficient on all hierarchy levels simultaneously. Yet, it should be simple enough for a feasible algorithm analysis. The cache-oblivious model introduced by Frigo et al. [308] promises all of the above. In fact, the immense popularity of this model lies in its innate simplicity and its ability to abstract away the hardware parameters.

The cache-oblivious model also assumes a two level memory hierarchy with an internal memory of size M and block transfers of B elements in one I/O. The performance measure is the number of I/Os incurred by the algorithm. However, the algorithm does not have any knowledge of the values of M and B. Consequently, the guarantees on I/O-efficient algorithms in the cache-oblivious model hold not only on any machine with multi-level memory hierarchy but also on all levels of the memory hierarchy at the same time. In principle, these algorithms are expected to perform well on different architectures without the need of any machine-specific optimization.

The cache-oblivious model assumes full associativity and optimal replacement policy. However, as we argued for the ideal cache model, these assumptions do not affect the asymptotics on realistic caches.

However, note that cache-oblivious algorithms are usually more complicated than their cache-aware I/O-efficient counterparts. As a result, the constant factors hidden in the complexity of cache-oblivious algorithms are usually higher and on large external memory inputs, they are slower in practice.

Various Streaming Models. In the data stream model [603], input data can only be accessed sequentially in the form of a data stream, and needs to be processed using a working memory that is small compared to the length of the stream. The main parameters of the model are the number p of sequential passes over the data and the size s of the working memory (in bits). Since the classical data stream model is too restrictive for graph algorithms and even the undirected connectivity problem requires $s \times p = \Omega(n)$ [387] (where n is the number of nodes in a graph), less restrictive variants of streaming models have also been studied. These include the stream-sort model [12] where sorting is also allowed, the W-stream model [232] where one can use intermediate temporary streams, and the semi-streaming model [284], where the available memory is $O(n \cdot polylog(n))$ bits.

There are still a number of issues not addressed by these models that can be critical for performance in practical settings, e. g., branch mispredictions [451], TLB misses etc. For other models on memory hierarchies, we refer to [53, 658, 505, 569].

5.2.2 Fundamental Techniques

The key principles in designing I/O-efficient algorithms are the exploitation of locality and the batching of operations. In a general context, *spatial locality* denotes that data close in address space to the currently accessed item is likely to be accessed soon whereas *temporal locality* refers to the fact that an instruction issued or a data item accessed during the current clock cycle is likely to be issued/accessed in the near future as well. The third concept is *batching*, which basically means to wait before issuing an operation until enough data needs to be processed such that the operation's cost is worthwhile. Let us see in more detail what this means for the design of I/O-efficient algorithms.

– **Exploiting spatial locality**: Since the data transfer in the external memory model (as well as the cache-oblivious model) happens in terms of block of elements rather than a single element at a time, the entire block when accessed should contain as much useful information as possible. This concept is referred to as "exploiting spatial locality". The fan-out of B in a B-tree exploiting the entire information accessible in one I/O to reduce the height of the tree (and therefore the worst-case complexity of various operations) is a typical example of "exploiting spatial locality".

Spatial locality is sometimes also used to represent the fact that the likelihood of referencing a resource is higher if a resource near it (with an appropriate measure of “nearness”) has just been referenced. Graph clustering and partitioning techniques are examples for exploiting “nearness”.

- **Exploiting temporal locality**: The concept of using the data in the internal memory for as much useful work as possible before it is written back to the external memory is called “exploiting temporal locality”. The divide and conquer paradigm in the external memory can be considered as an example of this principle. The data is divided into chunks small enough to fit into the internal memory and then the subproblem fitting internally is solved completely before reverting back to the original problem.
- **Batching the operations**: In many applications, performing one operation is nearly as costly as performing multiple operations of the same kind. In such scenarios, we can do lazy processing of operations, i. e., we first batch a large number of operations to be done and then perform them “in parallel” (altogether as one meta operation). A typical example of this approach is the buffer tree data structure described in more detail in Section 5.2.3. Many variants of external priority queue also do lazy processing of decrease-key operations after collecting them in a batch.

The following tools using the above principles have been used extensively in designing external memory algorithms:

Sorting and Scanning. Many external memory and cache-oblivious algorithms can be assembled using two fundamental ingredients: scanning and sorting. Fortunately, there are matching upper and lower bounds for the I/O complexity of these operations [11]. The number of I/Os required for scanning n data items is denoted by $\text{scan}(n) = \Theta(n/B)$ and the I/O complexity of sorting n elements is $\text{sort}(n) = \Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os. For all practical values of B, M and n on large data sets, $\text{scan}(n) < \text{sort}(n) \ll n$. Intuitively, this means that reading and writing data in sequential order or sorting the data to obtain a requisite layout on the disk is less expensive than accessing data at random.

The $O(n/B)$ upper bound for scanning can easily be obtained by the following simple modification: Instead of accessing one element at a time (incurring one I/O for the access), bring B contiguous elements in internal memory using a single I/O. Thus for the remaining $B-1$ elements, one can do a simple memory access, rather than an expensive disk I/O.

Although a large number of I/O-efficient sorting algorithms have been proposed, we discuss two categories of existing algorithms - merge sort and distribution sort. Algorithms based on the *merging paradigm* proceed in two phases: In the *run formation phase*, the input data is partitioned into sorted sequences, called “runs”. In the second phase, the *merging phase*, these runs are merged until only one sorted run remains, where merging k runs $S_1, \ldots, S_k$ means that a single sorted run S' is produced that contains all elements of runs $S_1, \ldots, S_k$. In the external memory sorting algorithm of Aggarwal and Vitter [11], the first phase produces sorted runs of M elements and the second phase does a $\frac{M}{B}$-way merge, leading to $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os. In the cache-oblivious setting, funnelsort [308] and lazy funnelsort [131], also based on the merging framework, lead to sorting algorithms with a similar I/O complexity. Algorithms based on the *distribution paradigm* compute a set of splitters $x_1 \leq x_2 \leq \ldots \leq x_k$ from the given data set S in order to partition it into subsets $S_0, S_1, \ldots, S_k$ so that for all $0 \leq i \leq k$ and $x \in S_i$, $x_i \leq x \leq x_{i+1}$, where $x_0 = -\infty$ and $x_{k+1} = \infty$. Given this partition, a sorted sequence of elements in S is produced by recursively sorting the sets $S_0, \ldots, S_k$ and concatenating the resulting sorted sequences. Examples of this approach include BalanceSort [616], sorting using the buffer tree [35], randomized online splitters [810], and algorithms obtained by simulating bulk-synchronous parallel sorting algorithms [215].

Simulation of Parallel Algorithms. A large number of algorithms for parallel computing models can be simulated to give I/O-efficient algorithms and sometimes even I/O-optimal algorithms. The relationship between the algorithms designed in the two paradigms of parallel and external computing is discussed in detail in Section 5.4.

Graph Decomposition and Clustering. A large number of external memory graph algorithms involve decomposing the graphs into smaller subgraphs. Planar graph separator [528] and its external memory algorithm [535] are a basis for almost all I/O-efficient planar graph algorithms [45, 40, 46]. Similarly, the tree-decomposition of a graph leads to external algorithms for bounded treewidth graphs [534]. For general graphs, the I/O-efficient undirected BFS algorithm of Mehlhorn and Meyer [555] relies on clustering of the input graph as an important subroutine. These separators, decompositions and clusterings can be used to divide the problem into smaller subproblems that fit into the internal memory [46] or to improve the layout of the graph on the disk [555].

Time Forward Processing. Time forward processing [35] is an elegant technique for solving problems that can be expressed as a traversal of a directed acyclic graph (DAG) from its sources to its sinks. Given the vertices of a DAG G in topologically sorted order and a labelling ϕ on the nodes of G, the problem is to compute another labelling ψ on the nodes such that label $\psi(v)$ for a node v can be computed from labels $\phi(v)$ and the labels $\psi(u_1), \ldots, \psi(u_k)$ of v's in-neighbors $u_1, \ldots, u_k$ in $O(sort(k))$ I/Os. This problem can be solved in $O(sort(m))$ I/Os, where m is the number of edges in the DAG. The idea [35] is to process the nodes in G by increasing topological number and use an external priority queue (Section 5.2.3) to realize the "sending" of information along the edges of G. When a node u_i wants to send its output $\psi(u_i)$ to another node v, it inserts $\psi(u_i)$ into priority queue Q and gives it priority v. When the node v is being evaluated, it removes all entries with priority v from Q. As every in-neighbor of v sends its output to v by queuing it with priority v, this provides v with the required labels and it can then compute its new label $\psi(v)$ in $O(sort(k))$ I/Os.

Many problems on undirected graphs can be expressed as evaluation problems of DAGs derived from these graphs. Applications of this technique for the construction of I/O-efficient data structures are also known.

Distribution Sweeping. Goodrich et al. [349] introduced distribution sweeping as a general approach for developing external memory algorithms for problems which in internal memory can be solved by a divide-and-conquer algorithm based on a plane sweep. This method has been successfully used in developing I/O-efficient algorithms for orthogonal line segment intersection reporting, all nearest neighbors problem, the 3D maxima problem, computing the measure (area) of a set of axis-parallel rectangles, computing the visibility of a set of line segments from a point, batched orthogonal range queries, and reporting pairwise intersections of axis-parallel rectangles. Brodal et al. [131] generalized the technique for the cache-oblivious model.

Full-Text Indexes. A full-text index is a data structure storing a text (a string or a set of strings) and supporting string matching queries: Given a pattern string P, find all occurrences of P in the text. Due to their fast construction and the wealth of combinatorial information they reveal, full-text indexes are often used in databases and genomics applications. The external memory suffix tree and suffix array can serve as full-text indexes. For a text T, they can be constructed in $O(sort(n))$ I/Os [280], where n is the number of characters in T. Other external full text indexing schemes use a hierarchy of indexes [58], compact Pat trees [176] and string B-trees [285].

There are many other tools for designing external memory algorithms. For instance, list ranking [733, 168], batch filtering [349], Euler tour computation [168], graph blocking techniques [10, 615] etc. Together with external memory data structures, these tools and algorithms alleviate the I/O bottleneck of many problems significantly.

5.2.3 External Memory Data Structures

In this section, we consider basic data structures used to design worst-case efficient algorithms in the external memory model. Most of these data structures are simple enough to be of practical interest.

An I/O-efficient storage of a set of elements under updates and query operations is possible under the following circumstances:

- Updates and queries are localized. For instance, querying for the most recently inserted element in case of a stack and least recently inserted element in case of a queue.
- We can afford to wait for an answer of a query to arrive, i. e., we can batch the queries (as in the case of a buffer tree).

– We can wait for the updates to take place, even if we want an online answer for the query. Many priority queue applications in graph algorithms are examples of this.

For online updates and queries on arbitrary locations, the B-tree is the most popular data structure supporting insertion, deletion and query operations in $O(\log_B n)$ I/Os.

Stacks and Queues. Stacks and queues are two of the most basic data structures used in RAM model algorithms to represent dynamic sets of elements and support deletion of elements in (last-in-first-out) LIFO and (first-in-first-out) FIFO order, respectively. While in internal memory, we can implement these data structures using an array of length n and a few pointers, it can lead to one I/O per insert and delete in the worst case. For the case of a stack, we can avoid this by keeping a buffer of $2B$ elements in the internal memory that at any time contains k most recently added set elements, where $k \leq 2B$. Removing an element needs no I/Os, except for the case when the buffer is empty. In this case, a single I/O is used to retrieve the block of B elements most recently written to external memory. Similarly, inserting an element uses no I/Os, except when the buffer runs full. In this case, a single I/O is used to write the B least recent elements to a block in external memory. It is not difficult to see that for any sequence of B insert or delete operations, we will need at most one I/O. Since at most B elements can be read or written in one I/O, the amortized cost of $1/B$ I/Os is the best one can hope for storing or retrieving a sequence of data items much larger than internal memory.

Analogously, we keep two buffers for queues: a read buffer and a write buffer of size B consisting of least and most recently inserted elements, respectively. Remove operations work on the read buffer and delete the least recent element without any I/O until the buffer is empty, in which case the appropriate external memory block is read into it. Insertions are done to the write buffer which when full is written to external memory. Similar to the case of stacks, we get an amortized complexity of $1/B$ I/Os per operation.

Linked Lists. Linked lists provide an efficient implementation of ordered lists of elements, supporting sequential search, deletion and insertion in arbitrary locations of the list. Traversing a pointer based linked list implementation used commonly in an internal memory algorithm may need to perform one I/O every time a pointer is followed. For an I/O-efficient implementation of linked lists, we keep the elements in blocks and maintain the invariant that there are more than $\frac{2}{3}B$ elements in every pair of consecutive blocks. Inserting an element can be done in a single I/O if the appropriate block is not full. If it is full but any of its two neighbors has spare capacity, we can push an element to that block. Otherwise, we split the block into two equally sized blocks. Similarly for deletion, we check if the delete operation results in violating the invariant and if so, we merge the two violating blocks. Split and merge can also be supported in $O(1)$ I/Os similarly.

To summarize, such an implementation of linked lists in external memory supports $O(1)$ I/O insert, delete, merge and split operations while supporting $O(i/B)$ I/O access to the i^{th} element in the list.

B-tree. The B-tree [77, 182, 416] is a generalization of balanced binary search trees to a balanced tree of degree $\Theta(B)$. Increasing the degree of the nodes helps us exploit the information provided by one I/O block to guide the search better and thereby reducing the height of the tree to $O(\log_B n)$. This in turn allows $O(\log_B n)$ I/O insert, delete and search operations. In external memory, a search tree like the B-tree or its variants can be used as the basis for a wide range of efficient queries on sets.

The degree of a node in a B-tree is $\Theta(B)$ with the root possibly having smaller degree. Normally, the n data items are stored in the $\Theta(n/B)$ leaves (in sorted order) of a B-tree, with each leaf storing $\Theta(B)$ elements. All leaves are on the same level and the tree has height $O(\log_B n)$. Searching an element in a B-tree can be done by traversing down the tree from the root to the appropriate leaf in $O(\log_B n)$ I/Os. One dimensional range queries can similarly be answered in $O(\log_B n + T/B)$ I/Os, where T is the output size. Insertion can be performed by first searching the relevant leaf l and if it is not full, inserting the new element there. If not, we split l into two leaves l' and l'' of approximately the same size and insert the new element in the relevant leaf. The split of l results in the insertion of a new routing element in the parent of l, and thus the need for a split may propagate up the tree. A new root (of degree 2) is produced when the root splits and the height of the tree grows by one. The total complexity of inserting a new element is thus $O(\log_B n)$ I/Os. Deletion is performed similarly in $O(\log_B n)$ I/Os by searching the appropriate leaf and removing the element to be deleted. If this results in too few elements in the leaf, we can fuse it with one of its siblings. Similar to the case of splits in insertion, fuse operations may propagate up the tree and eventually result in the height of the tree decreasing by one. The following are some of the important variants of a B-tree:

- Weight balanced B-tree [47]: Instead of a degree constraint (that the degree of a node v should be $\Theta(B)$ in a normal B-tree), in this variant, we require the weight of a node v to be $\Theta(B^h)$ if v is the root of a subtree of height h. The weight of v is defined as the number of elements in the leaves of the subtree rooted in v.
- Level balanced B-tree: Apart from the insert, delete and search operations, we sometimes need to be able to perform divide and merge operations on a B-tree. A divide operation at element x constructs two trees containing all elements less than and greater than x, respectively. A merge operation performs the inverse operation. This variant of B-tree supports both these operations in $O(\log_B n)$ I/Os.
- Partially persistent B-tree: This variant of the B-tree supports querying not only on the current version, but also on the earlier versions of the data structure. All elements are stored in a slightly modified B-tree where we also

associate a node existence interval with each node. Apart from the normal B-tree constraint on the number of elements in a node, we also maintain that a node contains $\Theta(B)$ alive elements in its existence interval. This means that for a given time t, the nodes with existence intervals containing t make up a B-tree on the elements alive at that time.
– String B-tree: Strings of characters can often be arbitrarily long and different strings can be of different length. The string B-tree of Ferragina and Grossi [285] uses a blind trie data structure to route a query string q. A blind trie is a variant of the compacted trie [482, 588], which fits in one disk block. A query can thus be answered in $O(\log_B n + |q|/B)$ I/Os.

Cache-oblivious variants of B-trees will be discussed later in Section 5.2.6.

Buffer Tree. A buffer tree [35] is a data structure that supports an arbitrary sequence of n operations (inserts, delete, query) in $O(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B})$ I/Os. It is similar to a B-tree, but has degree $\Theta(M/B)$ and each internal node has an associated buffer which is a queue that contains a sequence of up to M updates and queries to be performed in the subtree where the node is root. New update and query operations are "lazily" written to the root buffer (whose write buffer is kept in the internal memory), while non-root buffers reside entirely in external memory. When the buffer gets full, these operations are flushed down to the subtree where they need to be performed. When an operation reaches the appropriate node, it is executed.

Priority Queue. The priority queue is an abstract data structure of fundamental importance in graph algorithms. It supports insert, delete-min and decrease-key operations in $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{n}{B})$ I/Os amortized, while keeping the minimum element in the internal memory. The key technique behind the priority queue is again the buffering of operations. The following invariants added to the buffer tree provide an implementation of the priority queue:

– The buffer of the root node is always kept in the internal memory.
– The $O(M/B)$ leftmost leaves, i. e., the leaves of the leftmost internal node, are also always kept in the internal memory.
– All buffers on the path from the root to the leftmost leaf are empty.

The decrease-key operation in external memory is usually implemented by inserting the element with the new key and "lazily" deleting the old key.

There are many other external memory data structures, like interval tree [47], priority search tree, range tree, Bkd-tree [649], O-tree [453], PR-tree [42] etc. For a survey on I/O-efficient data structures, refer to [808, 37, 36, 809].

5.2.4 Cache-aware Optimization

In this section we present some important techniques for an efficient use of caches. Recall that caches are part of the memory hierarchy between processor registers and the main memory. They can make up several levels themselves and exploit the common observation that computations are local. If the code does not respect the locality properties (temporal and spatial), a required data item is likely to be not in the cache. Then, a *cache miss* occurs and several contiguous data words have to be loaded from memory into the cache.

Some techniques to avoid these expensive cache misses are presented in this section. Although these concepts are mainly designed for caches in the original sense, some of them might also give insights for the optimization of any level of the memory hierarchy. We consider two computationally intense areas, namely numerical linear algebra and computer graphics. In particular for numerical applications it is well-known that on many machine types the theoretical peak performance is rarely reached due to memory hierarchy related issues (e. g., [335]). Typically, the codes in both fields perform most work in small computational kernels based on loop nests. Therefore, while instruction cache misses are no problem, the exploitation of locality for efficient reuse of already cached data must be of concern in order to obtain satisfactory performance results.

Detecting Poor Cache Performance. The typical way in practice to analyze the performance of a program, and in particular its performance bottlenecks, is to use profiling tools. One freely available set of tools for profiling Linux or Unix programs comprises gprof [351] and the Valgrind tool suite [613], which includes the cache simulator cachegrind. While gprof determines how much CPU time is spent in which program function, cachegrind performs simulations of the L1 and L2 cache in order to determine the origins of cache misses in the profiled code. These results can also be displayed graphically with kprof [498] and kcachegrind [825], respectively.

Some tools provide access to certain registers of modern microprocessors called *performance counters*. These accesses provide information about certain performance-related events such as cache misses without affecting the program's execution time. Note that a variety of free and commercial profiling and performance tuning tools exists. An extensive list of tools and techniques is outside the scope of this work. The interested reader is referred to Kowarschik and Weiß [497] and Goedecker and Hoisie [335] for more details and references.

Fundamental Cache-Aware Techniques. In general, it is only worthwhile to optimize code portions that contribute significantly to the runtime because improvements on small contributors have only a small speedup effect on the whole program (cf. Amdahl's law in Chapter 6, Section 6.3).

In cases where the profiling information shows that severe bottlenecks are caused by frequent cache misses, one should analyze the reasons for this behavior and try to identify the particular class of cache-miss responsible for the problem. A cache miss can be categorized as *cold miss* (or *compulsory miss*), *capacity miss*, or *conflict miss* [395]. While a cold miss occurs when an item is accessed for the first time, a capacity miss happens when an item has been in the cache before the current access, but has already been evicted due to the cache's limited size. Conflict misses arise when an accessed item has been replaced because another one is mapped to its cache line. The following selection of basic and simple-to-implement techniques can often help to reduce the number of these misses and thus improve the program performance. They fall into the categories data access and data layout optimizations. The former consists mostly of loop transformations, the latter mainly of modifications in array layouts.

Loop Interchange and Array Transpose. Since data is fetched blockwise into the cache, it is essential to access contiguous data consecutively, for example multidimensional arrays. These arrays must be mapped onto a one-dimensional memory index space, which is done in a *row-major* fashion in C, C++, and Java and in a *column-major* fashion in Fortran. In the former the rightmost index increases the fastest as one moves through consecutive memory locations, where in the latter this holds for the leftmost index.

The access of data stored in a multidimensional array often occurs in a loop nest with a fixed distance of indices (*stride*) between consecutive iterations. If this data access does not respect the data layout, memory references are not performed on contiguous data (those with stride 1), which usually leads to cache misses. Therefore, whenever possible, the order in which the array is laid out in memory should be the same as in the program execution, i. e., if i is the index of the outer loop and j of the inner one, then the access $A[i][j]$ is accordant to row-major and $A[j][i]$ to column-major layout. The correct access can be accomplished by either exchanging the loop order (*loop interchange*) or the array dimensions in the declaration (*array transpose*).

Loop Fusion and Array Merging. The *loop fusion* technique combines two loops that are executed directly after another with the same iteration space into one single loop. Roughly speaking, this transformation is legal unless there are dependencies from the first loop to the second one (cf. [497] for more details). It results in a higher instruction level parallelism, reduces the loop overhead, and may also improve data locality. This locality improvement can be highlighted by another technique, the *array merging*. Instead of declaring two arrays with the same dimension and type (e. g., `double a[n], b[n]`), these arrays are combined to one multidimensional array (`double ab[n][2]`) or as an array of a structure comprised of `a` and `b` and length `n`. If the elements of `a` and `b` are typically accessed together, this ensures the access of contiguous memory locations.

Array Padding. In direct-mapped caches or caches with small associativity the entries at some index i of two different arrays might be mapped to the same cache line. Alternating accesses to these elements therefore cause a large number of conflict misses. This can be avoided by inserting a *pad*, i. e., an allocated, but unused array of suitable size to change the offset of the second array, between the two conflicting arrays (*inter-array padding*). The same idea applies to multidimensional arrays, where the leading dimension (the one with stride-1 access) is padded with unused memory locations (*intra-array padding*) if two elements of the same column are referenced shortly after another.

For additional cache-aware optimization techniques the interested reader is again referred to Kowarschik and Weiß [497] and Goedecker and Hoisie [335].

Cache-Aware Numerical Linear Algebra. The need for computational kernels in linear algebra that achieve a high cache performance is addressed for instance by the freely available implementations of the library interfaces *Basic Linear Algebra Subprograms* (BLAS) [105] and *Linear Algebra Package* (LAPACK) [30]. While BLAS provides basic vector and matrix operations of three different categories (level 1: vector-vector, level 2: matrix-vector, level 3: matrix-matrix), LAPACK uses these subroutines to provide algorithms such as solvers for linear equations, linear least-square and eigenvalue problems, to name a few. There are also vendor-specific implementations of these libraries, which are tuned to specific hardware, and the freely available *Automatically Tuned Linear Algebra Software* (ATLAS) library [829]. The latter determines the hardware parameters during its installation and adapts its parameters accordingly to achieve a high cache efficiency on a variety of platforms. In general it is advantageous to use one of these highly-tuned implementations instead of implementing the provided algorithms oneself, unless one is willing to carry out involved low-level optimizations for a specific machine [829].

One very important technique that is used to improve the cache efficiency of numerical algorithms is *loop blocking*, which is also known as *loop tiling*. The way it can be applied to such algorithms is illustrated by an example after giving a very brief background on sparse iterative linear equation solvers. In many numerical simulation problems in science and engineering one has to solve large systems of linear equations $\mathbf{A}x = b$ for x, where x and b are vectors of length n and the matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is sparse, i. e., it contains only $O(n)$ non-zero entries. These systems may stem from the discretization of a partial differential equation. As these linear systems cannot be solved by direct methods due to the large runtime and space consumption this would cause, iterative algorithms that approximate the linear system solution are applied. They may range from the basic splitting methods of Jacobi and Gauß-Seidel over their successive overrelaxation counterparts to Krylov subspace and multigrid methods [686]. The latter two are hard to optimize for cache data reuse [781] due to global operations in the first case and the traversal of a hierarchical data structure in the second one.

Since Krylov subspace and multigrid methods are much more efficient in the RAM model than the basic splitting algorithms, some work to address these issues has been done. Three general concepts can be identified to overcome most of the problems. The first aims at reducing the number of iterations by performing more work per iteration to speed up convergence, the second concept performs algebraic transformations to improve data reuse, and the third one removes data dependencies, e. g., by avoiding global sums and inner products. See Toledo's survey [781] for more details and references.

For multigrid methods in particular, one can optimize the part responsible for eliminating the high error frequencies. This *smoothing* is typically performed by a small number of Jacobi or Gauß-Seidel iterations. If the variables of the matrix

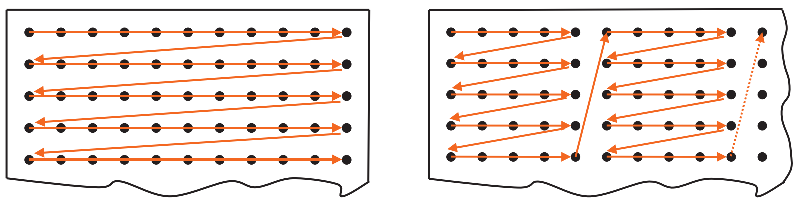

Figure 5.5. Rather than iterating over one complete matrix row (left), the loop blocking techniques iterates over small submatrices that fit completely into the cache (right).

correspond to graph nodes and the non-zero off-diagonal entries to graph edges, one can say that these algorithms update a node's approximated solution value by a certain edge-weighted combination of the approximated solution values at neighboring nodes. More precisely, the iteration formula of Gauß-Seidel iterations for computing a new approximation $x^{(k+1)}$ given an initial guess $x^{(0)}$ is

$$x_i^{(k+1)} = a_{i,i}^{-1} \left(b_i - \sum_{j<i} a_{i,j} x_j^{(k+1)} - \sum_{j>i} a_{i,j} x_j^{(k)} \right), 1 \leq i \leq n.$$

Some of the previously presented data layout and access optimizations can be applied to enhance the cache performance of the Gauß-Seidel algorithm [497]. Data layout optimizations include array padding to reduce possible conflict misses and array merging to improve the spatial locality of the entries in row i of $\mathbf{A}$ and b_i. As indicated above, a very effective and widely used technique for the improvement of data access and therefore temporal locality in loop nests is loop blocking. This technique changes the way in which the elements of objects, in our case this would be $\mathbf{A}$ and also the corresponding vector elements, are accessed. Rather than iterating over one row after the other, the matrix is divided into small block matrices that fit into the cache. New inner loops that iterate within the blocks are introduced into the original loop nest. The bounds of the outer loops are then changed to access each such block after the other. An example of this process assuming the traversal of a dense matrix is shown in Figure 5.5.

For simple problems such as matrix transposition or multiplication this is rather straightforward (a more advanced cache-oblivious blocking scheme for matrix multiplication is described in Section 5.2.5). However, loop blocking and performing several Gauß-Seidel steps one after another on the same block appears to be a little more complicated due to the data dependencies involved. When iterating over blocks tailored to the cache, this results in the computation of parts of $x^{(k')}, k' > k+1$, before $x^{(k+1)}$ has been calculated completely. However, if these blocks have an overlap of size $k' - (k+1)$ and this number is small (as is the case for multigrid smoothers), the overhead for ensuring that each block has to be brought into the cache only once is small [723]. This blocking scheme eliminates conflict misses and does not change the order of calculations (and thus the numerical result of the calculation). Hence, it is used in other iterative algorithms, too, where it is also called *covering* [781].

The case of unstructured grids, which is much more difficult in terms of cache analysis and optimization, has also been addressed in the literature [254]. The issues mainly arise here due to different local structures of the nodes (e. g., varying node degrees), which make indirect addressing necessary. In general, indirect addressing deteriorates cache performance because the addresses stored in two adjacent memory locations may be far away from each other. In order to increase the cache performance of the smoother in this setting, one can use graph partitioning methods to divide the grid into small blocks of nodes that fit into the cache. Thus, after a reordering of the matrix and the operators, the smoother can perform as much work as possible on such a small block, which requires the simultaneous use of one cache block only.

The speedups achievable by codes using the presented optimization techniques depend on the problem and on the actual machine characteristics. Kowarschik and Weiß [497] summarize experimental results in the area of multigrid methods by stating that an optimized code can run up to five times faster than an unoptimized one.

5.2.5 Cache-Oblivious Algorithms

As indicated above, cache-aware optimization methods can improve the runtime of a program significantly. Yet, the portability of this performance speedup from one machine to another is often difficult. That is why one is interested in algorithms that do not require specific hardware parameters.

One algorithmic technique to derive such *cache-oblivious* algorithms is the use of space-filling curves [687]. These bijective mappings from a line to a higher-dimensional space date back to the end of the 19th century [635,390]. They have been successfully applied in a variety of computer science fields, e. g., management of multimedia databases and image processing as well as load balancing of parallel computations (see Mokbel et al. [583]). When applied to objects with a regular structure, for instance structured or semi-structured grids, space-filling curves often produce high-quality solutions, e. g., partitionings of these graphs with high locality [862]. Here we present how this concept can be used to derive a cache-oblivious matrix multiplication algorithm. However, in case of unstructured grids or meshes that contain holes, space-filling curves usually work not as well as other approaches. The way to deal with these issues is shown afterwards by means of the cache-oblivious reordering of unstructured geometric meshes.

Matrix Multiplication. Multiplying two matrices is part of many numerical applications. Since we use it as a reference algorithm throughout this chapter, we define it formally.

Problem 1. Let $\mathbf{A}$ and $\mathbf{B}$ be two $n \times n$ matrices stored in the memory mainly intended for the computational model. Compute the matrix product $\mathbf{C} := \mathbf{AB}$

Algorithm 3 Naive matrix multiplication

```
1: for $i = 1$ to $n$ do
2:     for $j = 1$ to $n$ do
3:         $C[i, j] = 0.0$;
4:         for $k = 1$ to $n$ do
5:             $C[i, j] = C[i, j] + A[i, k] \cdot B[k, j]$;
```

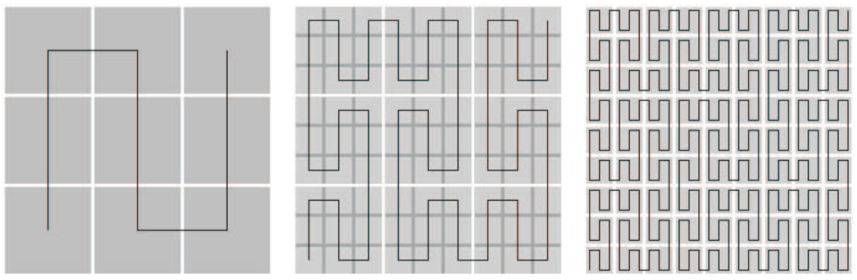

Figure 5.6. Recursive construction of the Peano curve.

and store it in the same type of memory using an algorithm resembling the naive one (cf. Algorithm 3).

Algorithm 3 is called standard or naive[1] and requires $O(n^3)$ operations. It contains a loop nest where two arrays of length n are accessed at the same time, one with stride 1, the other one with stride n. A loop interchange would not change the stride-n issue, but by applying the loop blocking technique, cached entries of all matrices can be reused. An automatic and therefore cache-oblivious blocking of the main loop in matrix multiplication can be achieved by recursive block building [369]. Several techniques have been suggested how to guide this recursion by space-filling curves. A method based on the Peano curve [635] (see Figure 5.6, courtesy of Wikipedia [634]) seems to be very promising, because it increases both spatial and temporal locality. We therefore illustrate its main ideas, the complete presentation can be found in Bader and Zenger [57].

Again, the key idea for a cache-efficient computation of $\mathbf{C} := \mathbf{AB}$ is the processing of matrix blocks. Each matrix is subdivided recursively into $n_x \times n_y$ block matrices until all of them are small, e. g., some fraction of the cache size. To simplify the presentation, we use nine recursive blocks (as in Figure 5.6) and the recursion stops with submatrices that have three rows and three columns. Note that, according to its authors [57], the algorithm works with any block size $n_x \times n_y$ if n_x and n_y are odd. Each submatrix of size 3×3 is stored in a

[1] *Naive* refers to the fact that asymptotically faster, but more complicated algorithms exist [758, 186].

Peano-like ordering, as indicated by the indices:

$$\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix} \cdot \begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix} = \begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}$$

The multiplication of each block is done in the standard way, for example, $c_7 := a_1 b_6 + a_4 b_7 + a_7 b_8$. In general, an element c_r can be written as the sum of three products $c_r = \sum_{(p,q)\in I_r} a_p b_q$, where I_r contains the three respective index pairs. Hence, after initializing all c_r to 0, one has to execute for all triples (r, p, q) the instruction $c_r \leftarrow c_r + a_p b_q$ in an arbitrary order. To do this cache-efficiently, jumps in the indices $r, p,$ and q have to be avoided. It is in fact possible to find such an operation order where two consecutive triples differ by no more than 1 in each element, so that optimal spatial and very good temporal locality is obtained. The same holds for the outer iteration, because the blocks are also accessed in the Peano order due to the recursive construction.

The analysis of this scheme for the 3×3 example in the ideal cache model with cache size M shows that the spatial locality of the elements is at most a factor of 3 away from the theoretical optimum. Moreover, the number of cache line transfers $T(n)$ for the whole algorithm with n a power of 3 is given by the recursion $T(n) = 27T(n/3)$. For blocks of size $k \times k$ each block admits $T(k) = 2 \cdot \lceil k^2/B \rceil$, where B is the size of a cache line. Altogether this leads to the transfer of $O(n^3/\sqrt{M})$ data items (or $O(n^3/B\sqrt{M})$ cache lines) into the cache, which is asymptotically optimal [781] and improves the naive algorithm by a factor of $\sqrt{M}$. The Peano curve ordering plays also a major role in a cache-oblivious self-adaptive full multigrid method [553].

Mesh Layout. Large geometric meshes may contain hundreds of millions of objects. Their efficient processing for interactive visualization and geometric applications requires an optimized usage of the CPU, the GPU (*graphics processing unit*), and their memory hierarchies. Considering the vast amount of different hardware combinations possible, a cache-oblivious scheme seems most promising. Yoon and Lindstrom [853] have developed metrics to predict the number of cache misses during the processing of a given mesh layout, i. e., the order in which the mesh objects are laid out on disk or in memory. On this basis a heuristic is described which computes a layout attempting to minimize the number of cache misses of typical applications. Note that similar algorithmic approaches have been used previously for unstructured multigrid (see Section 5.2.4) and for computing a linear ordering in implicit graph partitioning called graph-filling curves [702].

For the heuristic one needs to specify a directed graph $G = (V, E)$ that represents an anticipated runtime access pattern [853]. Each node $v_i \in V$ corresponds to a mesh object (e. g., a vertex or a triangle) and a directed arc (v_i, v_j) is inserted into E if it is likely that the object corresponding to v_j is accessed directly after the object represented by v_i at runtime. Given this graph and some probability measures derived from random walk theory, the task is to find a one-to-one mapping of nodes to layout indices, $\varphi : V \to \{1, \ldots, |V|\}$, that reduces the expected number of cache misses. Assuming that the cache holds only a single block whose size is a power of two, a cache-oblivious metric based on the *arc length* $l_{ij} = |\varphi(v_i) - \varphi(v_j)|$ is derived, which is proportional to the expected number of cache misses:

$$COM_g(\varphi) = \frac{1}{|E|} \sum_{(v_i,v_j) \in E} \log(l_{ij}) = \log \left(\left(\prod_{(v_i,v_j) \in E} l_{ij} \right)^{\frac{1}{|E|}} \right),$$

where the rightmost expression is the logarithm of the geometric mean of the arc lengths. The proposed minimization algorithm for this metric is related to multilevel graph partitioning [386], but the new algorithm's refinement steps proceed top-down rather than bottom-up. First, the original mesh is partitioned into k (e. g., $k = 4$) sets using a graph partitioning tool like METIS [468], which produces a low number of edges between nodes of different partitions. Then, among the $k!$ orders of these sets the one is chosen that minimizes $COM_g(\varphi)$. This partitioning and ordering process is recursively continued on each set until all sets contain only one vertex. Experiments show that the layout computed that way (which can be further improved by cache-awareness) accelerates several geometric applications significantly compared to other common layouts.

Other Cache-Oblivious Algorithms. Efficient cache-oblivious algorithms are also known for many fundamental problems such as sorting [308], distribution sweeping [131], BFS and shortest-paths [134], and 3D convex hulls [158]. For more details on cache-oblivious algorithms, the reader is referred to the survey paper by Brodal [130].

5.2.6 Cache-Oblivious Data Structures

Many cache-oblivious data structures like static [650] and dynamic B-trees [90, 88, 133], priority queue [132, 38], kd-tree [9], with I/O complexity similar to their I/O-efficient counterparts have been developed in recent years. A basic building block of most cache-oblivious data structures (e. g., [9, 90, 88, 133, 657, 89]) is a recursively defined layout called the *van Emde Boas layout* closely related to the definition of a van Emde Boas tree [794]. For the sake of simplicity, we only describe here the van Emde Boas layout of a complete binary tree T. If T has only one node, it is simply laid out as a single node in memory. Otherwise, let h be the height of T. We define the top tree T_0 to be the subtree consisting of the nodes in the topmost $\lfloor h/2 \rfloor$ levels of T, and the bottom trees $T_1, \ldots, T_k$ to be the $2^{\lfloor h/2-1 \rfloor}$ subtrees of size $2^{\lceil h/2 \rceil} - 1$ each, rooted in the nodes on level $\lceil h/2 \rceil$ of T. The van Emde Boas layout of T consists of the van Emde Boas layout of T_0 followed by the van Emde Boas layouts of $T_1, \ldots, T_k$.

A binary tree with a van Emde Boas layout can be directly used as a static cache-oblivious B-tree [650]. The number of I/Os needed to perform a search in

T, i. e., traversing a root-to-leaf path, can be analyzed by considering the first recursive level of the van Emde Boas layout when the subtrees are smaller than B. The size of such a *base tree* is between $\Theta(\sqrt{B})$ and $\Theta(B)$ and therefore, the height of a base tree is $\Omega(\log B)$. By the definition of the layout, each base tree is stored in $O(B)$ contiguous memory locations and can thus be accessed in $O(1)$ I/Os. As the search path traverses $O(\log n/\log B) = O(\log_B n)$ different base trees (where n is the number of elements in the B-tree), the I/O complexity of a search operation is $O(\log_B n)$ I/Os.

For more details on cache-oblivious data structures, the reader is referred to a book chapter by Arge et al. [39].

5.3 Parallel Computing Models

So far, we have seen how the speed of computations can be optimized on a serial computer by considering the presence of a memory hierarchy. In many fields, however, typical problems are highly complex and may require the processing of very large amounts of intermediate data in main memory. These problems often arise in scientific modeling and simulation, engineering, geosciences, computational biology, and medical computing [108, 147, 388, 494, 660] for more applications). Usually, their solutions must be available within a given timeframe to be of any value. Take for instance the weather forecast for the next three days: If a sequential processor requires weeks for a sufficiently accurate computation, its solution will obviously be worthless. A natural solution to this issue is the division of the problem into several smaller subproblems that are solved concurrently. This concurrent solution process is performed by a larger number of processors which can communicate with each other to share intermediate results where necessary. That way the two most important computing resources, computational power and memory size, are increased so that larger problems can be solved in shorter time.

Yet, a runtime reduction occurs only if the system software and the application program are implemented for the efficient use of the given parallel computing architecture, often measured by their *speed-up* and *efficiency* [503]. The *absolute speedup*, i. e., the running time of the best sequential algorithm divided by the running time of the parallel algorithm, measures how much faster the problem can be solved by parallel processing. Efficiency is then defined as the absolute speedup divided by the number of processors used.[2] In contrast to its absolute counterpart, *relative speedup* measures the inherent parallelism of the considered algorithm. It is defined as the ratio of the parallel algorithm's running times on one processor and on p processors [767].

To obtain a high efficiency, the application programmer might not want to concentrate on the specifics of one architecture, because it distracts from the actual problem and also limits portability of both the code and its execution

[2] On a more technical level efficiency can also be defined as the ratio of real program performance and theoretical peak performance.

speed. Therefore, it is essential to devise an algorithm design model that abstracts away unnecessary details, but simultaneously retains the characteristics of the underlying hardware in order to predict algorithm performance realistically [379]. For sequential computing the random access machine (RAM) has served as the widely accepted model of computation (if EM issues can be neglected), promoting "consistency and coordination among algorithm developers, computer architects and language experts" [533, p. 1]. Unfortunately, there has been no equivalent with similar success in the area of parallel computing.

One reason for this issue is the diversity of parallel architectures. To name only a few distinctions, which can also be found in Kumar et al. [503, Chapter 2], parallel machines differ in the control mechanism (SIMD vs. MIMD), address-space organization (message passing vs. shared memory), the interconnection networks (dynamic vs. static with different topologies), and processor granularity (computation-communication speed ratio). This granularity is referred to as *fine-grained* for machines with a low computation-communication speed ratio and as *coarse-grained* for machines with a high ratio. As a consequence of this diversity, it is considered rather natural that a number of different parallel computing models have emerged over time (cf. [379, 533, 539, 743]).

While shared-memory and network models, presented in Sections 5.3.1 and 5.3.2, dominated the design of parallel algorithms in the 1980's [798, Chapters 17 and 18], their shortcomings regarding performance prediction or portability have led to new developments. Valiant's seminal work on bulk-synchronous parallel processing [789], introduced in 1990, spawned a large number of works on parallel models trying to bridge the gap between simplicity and realism. These bridging models are explained in Section 5.3.3.

In Section 5.3.5 we present an algorithmic example and comparisons for the most relevant models and argue why some of them are favored over others today. Yet, considering recent works on different models, it is not totally clear even today which model is the best one. In particular because the field of parallel computing experiences a dramatic change: Besides traditional dedicated supercomputers with hundreds or thousands of processors, standard desktop processors with multiple cores and specialized multicore accelerators play an ever increasing role.

Note that this chapter focuses on parallel *models* rather than the complete process of parallel Algorithm Engineering; for many important aspects of the latter, the reader is referred to Bader et al. [56].

5.3.1 PRAM

The parallel random access machine (PRAM) was introduced in the late 1970s and is a straightforward extension of the sequential RAM model [300]. It consists of p processors that operate synchronously under the control of a common clock. They have each a private memory unit, but also access to a single global (or shared) memory for interprocessor communication (see [432, p. 9ff.]). Two measures determine the quality of a PRAM algorithm, the *time* and the *work*. Time denotes the number of parallel time steps an algorithm requires, work the product of time and the number of processors employed. Alternatively, work can be seen as the total number of operations executed by all processors. Three basic models are usually distinguished based on the shared memory access, more precisely if a cell may be read or written by more than one processor within the same timestep. Since there exist efficient simulations between these models, concurrent access does not increase the algorithmic power of the corresponding models dramatically [432, p. 496ff.].

The PRAM model enables the algorithm designer to identify the inherent parallelism in a problem and therefore allows the development of architecture-independent parallel algorithms [379]. However, it does not take the cost of interprocessor communication into account. Since the model assumes that global memory accesses are not more expensive than local ones, which is far from reality, its speedup prediction is typically inconsistent with the speedups observed on real parallel machines. This limitation has been addressed by tailor-made hardware [632, 806] and a number of extensions (cf. [23, 533] and the references therein). It can also be overcome by using models that reflect the underlying hardware more accurately, which leads us to the so-called network models.

5.3.2 Network Models

In a network model the processors are represented by nodes of an undirected graph whose edges stand for communication links between the processors. Since each processor has its own local memory and no global shared memory is present, these links are used to send communication messages between processors. During each algorithm step every node can perform local computations and communication with its neighbor nodes. If the algorithm designer uses a network model with the same topology as the actual machine architecture that is supposed to run the algorithm, the performance inconsistencies of the PRAM can be removed. However, porting an algorithm from one platform to another without a severe performance loss is often not easy. This portability issue is the reason why the use of network models is discouraged today for the development of parallel algorithms (see, e. g., [198]). For more results on these models we refer the interested reader to the textbooks of Akl [22] and Leighton [514], who present extensive discussions and many algorithms for various representatives of networks, e. g., arrays, meshes, hypercubes, and butterflies.

5.3.3 Bridging Models

The issues mentioned before and the convergence in parallel computer architectures towards commodity processors with large memory have led to the development of bridging models [198, 199]. They attempt to span the range between algorithm design and parallel computer architecture [332] by addressing the issues experienced with previous models, in particular by accounting for interprocessor communication costs and by making only very general assumptions about

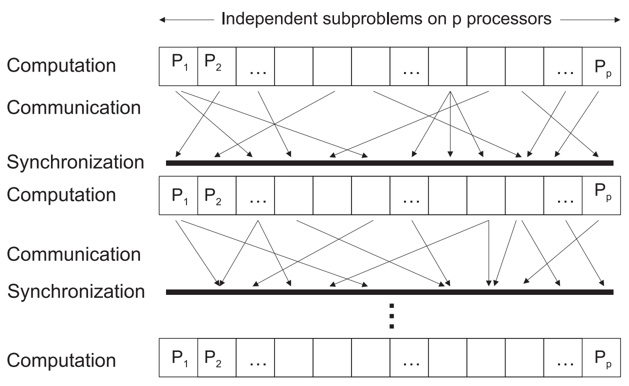

Figure 5.7. Schematic view of a sequence of supersteps in a BSP computation.

the underlying hardware. The presentation in this section is mainly in historical order, mentioning only the most relevant bridging models and important variations.

Bulk-Synchronous Parallel Model and its Variants. The bulk-synchronous parallel (BSP) model [789] consists of a number of sequential processors with local memory, a network router that delivers messages directly between any pair of processors for interprocessor communication, and a mechanism for global synchronization at regular intervals. A BSP algorithm is divided into so-called *supersteps*, each of which consists of local computations on already present data, message transmissions and receptions. Between each superstep a synchronization takes place, as illustrated in Figure 5.7. This decoupling of computation and communication simplifies the algorithm design to reduce the likelihood of errors.

For the analysis of such an algorithm three parameters besides the input size n are used: the number of processors p, the minimum superstep duration l arising from communication latency and synchronization (compare [329]), and finally the *gap* g, which denotes the ratio between computation and communication speed of the whole system. The model assumes that delivering messages of maximum size h (so-called *h-relations*) within one superstep requires $gh+l$ machine cycles. This accounts for the cost of communication by integrating memory speed and bandwidth into the model. Hence, the cost of a superstep is $w+gh+l$, where w denotes the maximum number of machine cycles over all processors required for local computation in this superstep. The cost of the complete algorithm is the sum of all supersteps' costs. Another measure sometimes used is called *slackness* or *slack*. It refers to the lower bound of n/p from which on the algorithm's runtime achieves an asymptotically optimal, i. e., linear, speedup.

On some parallel machines very small messages exhibit significant overhead due to message startup costs and/or latency. This can lead to a severe misestimation of an algorithm's performance [444]. Therefore, one variation of Valiant's original model called BSP* [76] addresses the granularity of messages by introducing a parameter B, the "optimum" message size to fully exploit the bandwidth of the router. Messages smaller than B generate the same costs as messages of size B, thus enforcing their algorithmic grouping to achieve higher communication granularity.

Many parallel machines can be partitioned into smaller subsets of processors where communication within each subset is faster than between different ones (consider, e. g., the BlueGene/L supercomputer architecture [778], a cluster of symmetric multiprocessors, or grid computing with parallel machines at different sites). This fact is incorporated in the decomposable BSP model [209], abbreviated D-BSP. Here the set of processors can be recursively decomposed into independent subsets. For each level i of this decomposition hierarchy, the p processors are partitioned into 2^i fixed and disjoint groups called i-clusters ($p = 2^k$, $k \in \mathbb{N}$, $0 \leq i \leq \log p$). A D-BSP program proceeds then as a sequence of labeled supersteps, where in an i-superstep, $0 \leq i < \log p$, communication and synchronization takes place only within the current i-clusters. Messages are of constant size and each level i of the decomposition hierarchy has its own gap g_i, where it is natural to assume that the gap increases when one moves towards level 0 of the hierarchy, thereby rewarding locality of computation. According to Bilardi et al. [99], D-BSP models real parallel architectures more effectively than BSP. As usual, this comes along with a more complicated model.

Coarse-Grained Multicomputer. Observed speedups of BSP algorithms may be significantly lower than expected if the parameter g and the communication overhead are high, which is true for many loosely-coupled systems like clusters. This is mainly due to the impact of small messages and has led to the coarse-grained multicomputer (CGM) model [216]. CGM enforces coarse-grained communication by message grouping, a similar idea as in the BSP* model, but without using an additional model parameter. It consists of p processors with $O(\frac{n}{p})$ local memory each, which are connected by an arbitrary connection network (even shared memory is allowed).

Analogous to BSP, an algorithm consists of supersteps that decouple computation and communication. The main difference is that during each communication round every processor groups all the messages for one target into a single message and sends and receives in total $O(\frac{n}{p})$ data items with high probability. Furthermore, communication calls can be seen as variations of global sorting operations on the input data, which facilitates a simple estimation of communication costs. Typically, the total running time is given as the sum of computation and communication costs, where the number of communication rounds (and therefore supersteps) is desired to be constant. Coarse-grained parallel algorithms based on the CGM model have become quite popular, e. g., see two special issues of Algorithmica on coarse-grained parallel computing [212, 213].

QSM. The authors of the Queuing Shared Memory (QSM) model advocate a shared-memory model enriched by some important architectural characteristics such as bandwidth constraints [332]. Their main argument is that a shared-memory model allows for a smooth transition from sequential algorithm design to symmetric multiprocessors and, ultimately, massively parallel systems. Consequently, the QSM model consists of a number of homogeneous processors with local memory that communicate by reading from and writing to shared memory. Like BSP this model assumes program execution in phases between which synchronization is performed. Within each phase one is free to interleave the possible operations shared-memory read, shared-memory write, and local computation arbitrarily. The only parameters used are the number of processors p and the computation-communication gap g.

Shared-memory accesses during a phase may access the same location either reading or writing (but not both) and complete by the end of that phase. For the cost analysis one determines the cost of a single phase, which is the maximum of the costs for the three following operations: maximum number of local operations, gap g times the maximum number of shared-memory reads or writes, and the maximum shared-memory contention. The cost of the complete algorithm is again the sum of all phase costs.

5.3.4 Recent Work

Bridging Models. To cover follow-up research, we first turn our attention to heterogeneous parallel computing, where one uses a heterogeneous multicomputer by combining different types of machines over different types of network. This can be viewed as a precursor to grid computing. Hence, the two extensions of CGM and BSP that incorporate heterogeneity, HCGM [587] and HBSP [836], might be of interest there. Both models account for differing processor speeds, but possible network differences are not distinguished. This issue and limited success of heterogeneous high performance computing may prevent a wide applicability of these models without modifications.

A more recent bridging model is PRO [322], a restriction of BSP and CGM whose main characteristic is the comparison of all metrics to a specific sequential algorithm A_{seq} with time and space complexity $T(n)$ and $S(n)$, respectively. Similar to CGM, the underlying machine consists of p processors having $M = O(S(n)/p)$ local memory each, where a coarseness of $M \geq p$ is assumed. The execution proceeds in supersteps of separated computation and communication. The latter is performed with grouped messages and costs one time unit per word sent or received. Interestingly, the quality measure of PRO is not the time (which is enforced to be in $O(T(n)/p)$), but the range of values for p that facilitate a linear speedup w.r.t. A_{seq}. This measure is called Grain(n) and shown to be in $O(\sqrt{S(n)})$ due to the coarseness assumed in the model. The better of two PRO algorithms solving the same problem with the same underlying sequential algorithm is therefore the one with higher grain.

As noted before, there are a large number of other parallel computing models, mostly modifications of the presented ones, dealing with some of their issues. Yet, since they have not gained considerable importance and an exhaustive presentation of this vast topic is outside the scope of this work, we refer the interested reader to the books [22, 192, 193, 503, 514, 660], the surveys [190, 379, 465, 533, 539, 743], and [332, 353, 790].

Multicore Computing: Algorithmic Models and Programming Frameworks. Most models that have been successful in the 1990s do not assume shared memory but incorporate some form of explicit inter-processor communication. This is due to the widespread emergence of cluster computers and other machines with distributed memory and message passing communication during that time. Meanwhile nearly all standard CPUs built today are already parallel processors because they contain multiple computing cores. The idiosyncracies of this architectural change need to be reflected in the computational model if algorithms are to be transformed into efficient programs for multicore processors or parallel machines of a large number of multicore CPUs.

One particular issue, which combines the topics hierarchical memory and parallel computing, is the *sharing* of caches. In modern multicore processors it is common that the smallest cache levels are private to a core. However, usually the larger the cache level is, the more cores share the same cache. Savage and Zubair [701] address cache sharing with the universal multicore model (UMM). They introduce the Multicore Memory Hierarchy Game (MMHG), a pebbling game on a DAG that models the computations. By means of the MMHG Savage and Zubair derive general lower bounds on the communication complexity between different hierarchy levels and apply these bounds to scientific and financial applications.

With the prevalence of multicore chips with shared memory the PRAM model seems to experience a renaissance. While it is still regarded as hardly realistic, it recently serves as a basis for more practical approaches. Dorrigiv et al. [253] suggest the LoPRAM (low degree parallelism PRAM) model. Besides having two different thread types, the model assumes that an algorithm with input size n is executed on at most $\mathcal{O}(\log n)$ processors – instead of $\mathcal{O}(n)$ as in the PRAM model. Dorrigiv et al. show that for a wide range of divide-and-conquer algorithms optimal speedup can be obtained. Vishkin et al. [806] propose a methodology for converting PRAM algorithms into explicit multi-threading (XMT) programs. The XMT framework includes a programming model that resembles the PRAM, but relaxes the synchronous processing of individual steps. Moreover, the framework includes a compiler of XMTC (an extension of the C language) to a PRAM-on-chip hardware architecture. Recent studies suggest that XMT allows for an easier implementation of parallel programs than MPI [399] and that important parallel algorithms perform faster on the XMT PRAM-on-chip processor than on a standard dual-core CPU [150].

Valiant extends his BSP model to hierarchical multicore machines [791]. This extension is done by assuming d hierarchy levels with four BSP parameters each, i.e., level i has parameters (p_i, g_i, L_i, m_i), where p_i denotes the number of subcomponents in level i, g_i their bandwidth, L_i the cost of synchronizing them, and m_i the memory/cache size of level i. For the problems of associative composition, matrix multiplication, fast Fourier transform, and sorting, lower bounds on the communication and synchronization complexity are given. Also, for the problems stated above, algorithms are described that are optimal w. r. t. to communication and synchronization up to constant factors.

A more practical approach to map BSP algorithms to modern multicore hardware is undertaken by Hou et al. [413]. They extend C by a few parallel constructs to obtain the new programming language BSGP. Programs written in BSGP are compiled into GPU kernel programs that are executable by a wide range of modern graphics processors.

The trend to general purpose computations on GPUs can be explained by the much higher peak performance of these highly parallel systems compared to standard CPUs. Govindaraju et al. [350] try to capture the most important properties of GPU architectures in a cache-aware model. They then develop cache-efficient scientific algorithms for the GPU. In experiments these new algorithms clearly outperform their optimized CPU counterparts.

The technological change to multicore processors requires not only algorithmic models for the design of theoretically efficient algorithms, but also suitable programming frameworks that allow for an efficient implementation. Among these frameworks are:

- OpenMP [161], Cilk++ [174], and Threading Building Blocks [667] are APIs or runtime environments for which the programmer identifies independent tasks. When the compiled application program is executed, the runtime environment takes care of technical details such as thread creation and deletion and thus relieves the programmer from this burden.
- Chapel [155], Fortress [24], Unified Parallel C (UPC) [95], Sequoia [282], and X10 [162] are parallel programming languages, whose breakthrough for commercial purposes has yet to come.
- CUDA [617], Stream [8], and OpenCL [473] are intended for a simplified programming of heterogeneous systems with CPUs and GPUs, in case of OpenCL also with other accelerators instead of GPUs.

A further explanation of these works is outside the scope of this chapter since their main objective is implementation rather than algorithm design.

5.3.5 Application and Comparison

In this section, we indicate how to develop and analyze parallel algorithms in some of the models presented above. The naive matrix multiplication algorithm serves here again as an example. Note that we do not intend to teach the development of parallel algorithms in detail, for this we refer to the textbooks stated in the previous section. Instead, we wish to use the insights gained from the example problem as well as from other results to compare these models and argue why some are more relevant than others for today's parallel algorithm engineering.

Algorithm 4 PRAM algorithm for standard matrix multiplication

The processors are labelled as $P(i,j,k), 0 \leq i,j,k < p^{1/3}$.

```
1: $P(i,j,k)$ computes $C'(i,j,k) = A(i,k) \cdot B(k,j)$
2: for $h := 1$ to $\log n$ do
3:     if $(k \leq \frac{n}{2^h})$ then
4:         $P(i,j,k)$ sets $C'(i,j,k) := C'(i,j,2k-1) + C'(i,j,2k)$
5: if $(k = 1)$ then
6:     $P(i,j,k)$ sets $C(i,j) := C'(i,j,1)$
```

Algorithm 5 BSP algorithm for standard matrix multiplication

Let $\mathbf{A}$ and $\mathbf{B}$ be distributed uniformly, but arbitrarily, across the p processors denoted by $P(i,j,k)$, $0 \leq i,j,k < p^{1/3}$. Moreover, let $\mathbf{A}[i,j]$ denote the $s \times s$ submatrix of $\mathbf{A}$ with $s := n/p^{1/3}$. Define $\mathbf{B}[i,j]$ and $\mathbf{C}[i,j]$ analogously.

```
1: $P(i,j,k)$ acquires the elements of $\mathbf{A}[i,j]$ and $\mathbf{B}[j,k]$.
2: $P(i,j,k)$ computes $\mathbf{A}[i,j] \cdot \mathbf{B}[j,k]$ and sends each resulting value to the processor
   responsible for computing the corresponding entry in $\mathbf{C}$.
3: $P(i,j,k)$ computes each of its final $n^2/p$ elements of $\mathbf{C}$ by adding the values received
   for these elements.
```

Algorithm Design Example. Algorithm 4 [432, p. 15f.] performs matrix multiplication on a PRAM with concurrent read access to the shared memory. Here and in the following two examples we assume that the algorithm (or program) is run by all processors in parallel, which are distinguished by their unique label. The algorithm's idea is to perform all necessary multiplications in $\log n$ parallel steps with $n^3/\log n$ processors (Step 1) and to compute the sums of these products in $\log n$ parallel steps (Steps 4 and 6). The latter can be done by means of a binary tree-like algorithm which sums n numbers in the following way: Sum the index pair $2i-1$ and $2i, 1 \leq i \leq n/2$ in parallel to obtain $n/2$ numbers and proceed recursively. Hence, for the second step $O(n^3)$ processors require $O(\log n)$ steps. This would lead to a time complexity of $O(\log n)$ and a suboptimal work complexity, because the processor-time product would be $O(n^3 \log n)$. However, it is not difficult to see that Step 4 can be scheduled such that $O(n^3/\log n)$ processors suffice to finish the computation in $O(\log n)$ timesteps, resulting in the optimal work complexity for this algorithm of $O(n^3)$.

This algorithm illustrates both the strength and the weakness of the PRAM model. While it makes the inherent parallelism in the problem visible, the assumption to have $p = n^3/\log n$ processors to solve a problem of size $n \times n$ is totally unrealistic today. On the other hand we can use the idea of emulating the algorithm with only $p' < p$ processors. If each of the p' processors operates on a block of the matrix instead of a single element, we already have an idea how a coarse-grained algorithm might work.

Indeed, Algorithm 5, due to McColl and Valiant [543], performs matrix multiplication in the BSP model by working on matrix blocks. Its cost analysis

Algorithm 6 CGM and PRO algorithm for standard matrix multiplication

Let the matrices **A** and **B** be distributed onto the processors blockwise such that processor $P(i,j)$ stores $\mathbf{A}[i,j]$, the $s \times s$ ($s = n/p^{1/2}$) submatrix of **A**, and $\mathbf{B}[i,j]$, $0 \le i,j < p^{1/2}$.

```
1: $P(i,j)$ computes $\mathbf{C}[i,j] := \mathbf{A}[i,j] \cdot \mathbf{B}[i,j]$.
2: for superstep $i := 1$ to $p^{1/2}$ do
3:     $P(i,j)$ sends the block of $\mathbf{A}$ processed in the previous step to $P(i,(j+1) \bmod p^{1/2})$ and receives the new block from $P(i,(j-1) \bmod p^{1/2})$.
4:     $P(i,j)$ sends the block of $\mathbf{B}$ processed in the previous step to $P((i+1) \bmod p^{1/2}, j)$ and receives the new block from $P((i-1) \bmod p^{1/2}, j)$.
5:     $P(i,j)$ determines the product of the current submatrices of $\mathbf{A}$ and $\mathbf{B}$ and adds the result to $\mathbf{C}[i,j]$.
```

proceeds as follows: the first superstep requires the communication of $n^2/p^{2/3}$ values, resulting in $O(g \cdot n^2/p^{2/3} + l)$ time steps. Computation and communication of Superstep 2 account together for $O(n^3/p + g \cdot n^2/p^{2/3} + l)$ time steps and the final superstep requires costs of $O(n^2/p^{2/3} + l)$. This yields a total runtime of $O(n^3/p + g \cdot n^2/p^{2/3} + l)$, which is optimal in terms of communication costs for any BSP implementation of standard matrix multiplication [543]. Algorithm 5 is therefore best possible in the sense that it achieves all lower bounds for computation, communication, and synchronization costs. Note that the memory consumption can be reduced at the expense of increased communication costs [544], a basic variant of which is presented in the following paragraph.

Recall that the CGM model requires that communication is grouped and may not to exceed $O(n^2/p)$ values per round (note that the input size of the considered problem is n^2 instead of n). Hence, the blocking and communication scheme of the algorithm above has to be adapted. First, this is done by setting $s := n/p^{1/2}$. Then, using the definitions from Algorithm 5 and assuming for simplicity that s and $p^{1/2}$ are integers, we obtain Algorithm 6, which is briefly mentioned by McColl [543].

It is easy to verify that the computation costs account for $O(n^3/p)$ and the communication costs for $O(n^2/p^{1/2})$ cycles. Thus, it becomes a valid CGM algorithm with $O(p^{1/2})$ communication rounds and can also be used in the PRO model with the desired speedup property. To compute the quality measure Grain(n), observe that the communication within the loop must not be more expensive than the computation. This is fulfilled whenever $n^3/p^{3/2} \ge n^2/p \Leftrightarrow p \le n^2$ and we obtain with the coarseness assumption the optimal grain of $O(n)$.

The examples for the more realistic bridging models show that blocking and grouping of data is not only essential in the external memory setting but also for parallel algorithms. It is sometimes even better to perform more internal work than necessary if thereby the communication volume can be reduced. Note that this connection between the two computational models is no coincidence since both aim at the minimization of communication. For the I/O model communication means data transfers to/from the external disk, for parallel models it refers to inter-processor communication. Before we investigate this connection in more detail in Section 5.4, the bridging models discussed above are compared.

Further Model Comparison. The reasons for discouraging the sole use of PRAM and network models for parallel algorithm development have already been discussed before. In this brief comparison we therefore focus on the major bridging models.

The main aim of another bridging model, called LogP [198], is to capture machine characteristics for appropriate performance prediction. This burdens the algorithm designer with the issue of stalling due to network contention and nondeterminism within the communication. Since it has been shown that stall-free LogP programs can be efficiently emulated on a BSP machine (and vice versa) [100], this has led to the conclusion that BSP offers basically the same opportunities as LogP while being easier to deal with. Consequently, apart from a number of basic algorithms for LogP, there seems to be little interest in further results on design and analysis of LogP algorithms (compare [661] and [187]).

A similar argument applies to QSM, because it can also be emulated efficiently on a BSP machine (and vice versa) [332, 661]. Although QSM can be used to estimate the practical performance of PRAM algorithms and it requires only two parameters, it seems that it has had only limited success compared to BSP related models based on point-to-point messages. This might be due to the fact that it does not reward large messages and that more focus was put on massively parallel systems rather than shared-memory machines. It remains to be seen if some QSM ideas might experience a revival with the ubiquity of multicore CPUs.

One restriction of the coarse-grained models BSP, CGM (and also PRO, which has yet to prove its broad applicability) is their disregard of actual communication patterns. Although some patterns are more expensive than others, this is not incorporated into the models and can show large differences between estimated and actual performance [353, 444]. Nevertheless, for many algorithms and applications these models and their extensions provide a reasonably accurate performance and efficiency estimate. Their design capabilities capture the most important aspects of parallel computers. Moreover, the analysis can be performed with a small set of parameters for many parallel architectures that are in use today and in the near future. Another reason for the wide acceptance of BSP and CGM might be their support of message passing. This type of interprocessor communication has been standardized by the Message Passing Interface Forum[3] as the MPI library [747], whose implementations are now probably the most widely used communication tools in distributed-memory parallel computers.

All this has led to the fact that BSP and CGM have been used more extensively than other models to design parallel algorithms in recent years [187]. Even libraries that allow for an easy implementation of BSP and CGM algorithms have been developed. Their implementations are topics of a success story on parallel computing models in Section 5.6.

[3] See http://www.mpi-forum.org/ .

Given the convergence of parallel machines and networking hardware to commodity computing and the prevalence of multicore CPUs with shared memory and deep memory hierarchies, a model that combines these features in a both realistic and simple way would certainly be valuable, as Cormen and Goodrich already expressed in 1996 [190]. Recently, Arge et al. [43] have proposed the Parallel External-Memory model as a natural parallel extension of the external-memory model of Aggarwal and Vitter [11], to private-cache chip multiprocessors.

On the other hand, the connection between parallel and external memory algorithms has been investigated by stating efficient simulations of parallel algorithms in external memory. These results are presented in the upcoming section.

5.4 Simulating Parallel Algorithms for I/O-Efficiency

Previously in this chapter we have presented several models and various techniques for I/O-efficiency, cache optimization, and parallel computing. Generally speaking, I/O-efficient algorithms are employed to deal with massive data sets in the presence of a memory hierarchy, while parallel computing is more concerned with the acceleration of the actual on-chip computations by dividing the work between several processors. It might not be a surprise that there are some similarities between the models and techniques. In cases where one needs to process extremely large data sets with high computational power, methods from both fields need to be combined. Unfortunately, there is no model that incorporates all the necessary characteristics.

In this section we show the connection of the concepts presented previously and indicate how to derive sequential and parallel external memory algorithms by simulation. Generally speaking, simulations transform known parallel algorithms for a given problem P into an external memory algorithm solving P. The key idea is to model inter-processor communication as external memory accesses. Since efficient parallel algorithms aim at the minimization of communication, one can often derive I/O-efficient algorithms this way. Note, however, that the simulation concept should be thought of as a guide for designing algorithms, rather than for implementing them.

First, we explain a simulation of PRAM algorithms in Section 5.4.1. Since there exists an obvious similarity between bulkwise inter-processor communication and blockwise access to external memory, one would also expect I/O-efficient simulation results of coarse-grained parallel algorithms. Indeed, a number of such simulations have been proposed; they are discussed in Section 5.4.2.

5.4.1 PRAM Simulation

The first simulation we describe obtains I/O-efficient algorithms from simulating PRAM algorithms [168]. Its value stems from the fact that it enables the efficient transfer of the vast amount of PRAM algorithms into the external memory setting. The key idea is to show that a single step of a PRAM algorithm processing n data items can be simulated in $O(sort(n))$ I/Os. For this consider a PRAM algorithm A that utilizes n processors and $O(n)$ space and runs in time $O(T(n))$. Let each processor perform w. l. o. g. within a single PRAM step $O(1)$ shared-memory (SM) reads, followed by $O(1)$ steps for local computation and $O(1)$ shared-memory writes. We now simulate A on an external memory machine with one processor. For this assume that the state information of the PRAM processors and the SM content are stored on disk in a suitable format.

The desired transformation of an arbitrary single step of A starts by simulating the SM read accesses that provide the operands for the computation. This requires a scan of the processor contexts to store the read accesses and their memory locations. These values are then sorted according to the indices of the SM locations. Then, this sorted list of read requests is scanned and the contents of the corresponding SM locations are retrieved and stored with their requests. These combined values are again sorted, this time according to the ID of the processor performing the request. By scanning this sorted copy, the operands can be transferred to the respective processor. After that, we perform the computations on each simulated processor and write the results to disk. These results are sorted according to the memory address to which the processors would store them. The sorted list and a reserved copy of memory are finally scanned and merged to obtain the previous order with the updated entries. This can all be done with $O(1)$ scans and $O(1)$ sorts for n entries, so that simulating all steps of A requires $O(T(n) \cdot sort(n))$ I/Os in total.

This simulation has a noteworthy property in case of PRAM algorithms where the number of active processors decreases geometrically with the number of steps. By this, we mean that after a constant number of steps, the number of active processors (those that actually perform operations instead of being idle) and the number of memory cells used afterwards has decreased by a constant factor. Typically, the work performed by these algorithms, i. e., their processor-time product, is not optimal due to the high number of inactive processors. These inactive processors, however, do not need to be simulated in the external memory setting. One can therefore show that such a non-optimal PRAM algorithm leads to the same simulation time of $O(T(n) \cdot sort(n))$ I/Os as above, which means that the non-optimal work property of the simulated algorithm does not transfer to the algorithm obtained by simulation.

5.4.2 Coarse-grained Parallel Simulation Results

The simulations of coarse-grained parallel algorithms shown in this section resemble the PRAM simulation. They also assume that the state information of the simulated processors are stored on disk, and they simulate one superstep after the other. This means that one reads the processor *context* (memory image and message buffers) from disk first and then simulates incoming communication, computation, and outgoing communication, before the updated context is written back to disk. However, the actual implementations need to consider the idiosyncrasies of the different coarse-grained parallel models.

Note that the virtual processors of the parallel algorithm are simulated by a possibly smaller number p of processors in the external memory model. Then, the simulation starts with processors $0, \ldots, p-1$, proceeds with the next p processors, and so on. This serialization of the parallel program is valid due to the independence of processors within the same superstep. Recall that M denotes the size of the internal memory and B the block size in the EM model.

Single-processor Simulations. Since it is based on a simple framework, we proceed our explanation with the sequential simulation of *BSP-like* algorithms [734]. A BSP-like algorithm assumes the memory space to be partitioned into p blocks of suitable size. It proceeds in discrete supersteps, is executed on a virtual machine with p processors, and satisfies the following conditions (cmp. [734, Definition 1]):

– In superstep $s, s \geq 1$, processor $p_i, 0 \leq i < p$, operates only on the data in block $\mathcal{B}_i$ and on the messages $Mes(j, i, s), 0 \leq j < p$.
– In superstep $s, s \geq 1$, processor $p_i, 0 \leq i < p$, generates messages $Mes(i, j, s+1)$ to be 'sent' to $p_j, 0 \leq j < p$. The size of each message is at most $M/3p$. The initial messages of timestep 1 are void.

Then, the simulation can proceed for each superstep as described at the beginning of this section. In each superstep processor $p_i, 0 \leq i < p$, fetches $\mathcal{B}_i$ and its respective message buffers $Mes(j, i, s), 0 \leq j < p$, from disk, simulates the computations of the superstep, and stores the updated block $\mathcal{B}_i$ as well as new message buffers to disk in suitable locations.

For these BSP-like algorithms new parameters $P = \lceil 3 \cdot n/M \rceil, G$, and L are introduced to relate coarse-grained models to the EM model. The I/O transfer gap G denotes the ratio of the number of local computation operations and the number of words that can be transferred between memory and disks per unit time, while L denotes the synchronization time of the simulation. They measure the quality of their simulation by the notion of c-optimality [329], which is transferred to the I/O setting. An EM algorithm is called *c-optimal* if its execution time is at most c times larger than that of a sequential computer with infinite memory. The main result states that if the BSP parameters (p, g, l) coincide with the new parameters (P, G, L) and there is a c-optimal BSP algorithm for the same problem, then the corresponding BSP-like algorithm in external memory is also c-optimal [734, Theorem 3].

If one accepts that the external memory size is bounded from above by M^2 (which is a reasonable assumption), the simulation of PRO algorithms in external memory is another option [370]. It introduces the notion of *RAM-awareness*, which provides a measure for the number of random memory accesses that might correspond to page faults. If this measure of a PRO algorithm A on $p = \text{Grain}(n)$ processors does not exceed the sequential runtime of the underlying algorithm and A requires $T(n)$ time and $S(n)$ space over all processors, A can be simulated in $O(T(n))$ computation time with $O(S(n)/\operatorname{Grain}(n) + \operatorname{Grain}(n))$ internal and $O(S(n))$ external memory.

Multiple-processor Simulations. Dehne et al. [215, 214] show how to simulate algorithms for the models BSP, BSP*, and CGM on sequential and parallel machines with parallel disks. These combined models are then called EM-BSP, EM-BSP*, and EM-CGM, respectively, and extend the parameter set of their underlying parallel models by M (local memory size for each processor), D (number of parallel disks connected to each processor), B (transfer block size), and G (I/O transfer gap in terms of memory block transfer). More precisely, the simulation costs are the same as for the simulated program plus the costs induced by I/O, which is taken as the maximum over all processors.

As above, the simulation of the v virtual processors is performed in supersteps. During each such superstep every simulating processor loads the context of the virtual processors for which it is responsible from the disk. Whenever virtual communication is replaced by parallel disk I/O, care is taken that irregular routing schemes are mapped to disks in a balanced way to obtain optimal I/O costs. Amongst others, this is done by setting the total communication amount of each processor to $\Theta(n/v)$ and by fixing the message size to $c \cdot B$ for some $c \geq 1$, which resembles the idea of BSP*.

The c-optimality notion [329] is extended from local computation to cover also communication and I/O. Using this, one can show that a work-optimal, communication-efficient, and I/O-efficient algorithm can be simulated with a small overhead by an algorithm that is also work-optimal, communication-efficient, and I/O-efficient for a wide range of parameters by using the techniques of Dehne et al. [215]. There, it is also shown that these methods have led to improved parallel EM algorithms.

Cache-Oblivious Simulation of D-BSP. For the final topic of this section, our simulation target is one level higher in the memory hierarchy. More precisely, we simulate D-BSP programs to achieve sequential cache-oblivious algorithms [636]. (Related simulation results are also presented by Bilardi et al. [99].) The technique exploits that the D-BSP model assumes a hierarchical decomposition of a BSP computer in processor groups to capture submachine locality. Recall that the cache in the Ideal Cache Model (ICM) contains M words organized into lines of B words each. It is fully associative and assumes the optimal offline strategy for cache-line replacement. To simulate a D-BSP program in the ICM in a cache-oblivious manner, the simulation algorithm for improving locality in a multilevel memory hierarchy [279] is adapted. First of all, the slower memory of the ICM hierarchy is divided into p blocks of size $\Theta(\mu)$, where μ is the size of one D-BSP processor context. Each block contains one processor context and some extra space for bookkeeping purposes.

Recall that each processor group on level i of the D-BSP hierarchy is called an i-cluster. Its processors collaborate with each other in an i-superstep. Therefore, the simulation proceeds in rounds, where each round simulates one i-superstep for a certain i-cluster in two phases (local computation and communication) and determines the cluster for the next round. Message distribution for intra-cluster communication is simulated by sorting the contexts of the processors involved, similar to the method proposed by Fantozzi et al. [279]. In particular by simulating the same cluster in consecutive supersteps, this simulation strategy is able to improve the locality of reference, because the necessary processor contexts are already cached. If sorting the processors' contexts for simulating communication is done in a cache-oblivious manner, the whole algorithm is cache-oblivious since it does not make use of the parameters M and B.

5.5 Success Stories of Algorithms for Memory Hierarchies

In this section we describe some implementations of algorithms for memory hierarchies that have improved the running time on very large inputs considerably in practice.

5.5.1 Cache-Oblivious Sorting

Brodal et al. [135] show that a careful implementation of a cache-oblivious lazy funnelsort algorithm [131] outperforms several widely used library implementations of quicksort on uniformly distributed data. For the largest instances in the RAM, this implementation outperforms its nearest rival std::sort from the STL library included in GCC 3.2 by 10-40% on many different architectures like Pentium III, Athlon and Itanium 2. Compared to cache-aware sorting implementations exploiting L1 and L2 caches, TLBs and registers [41, 504, 843, 782], the cache-oblivious implementation is not only more robust – it exploits several levels of memory hierarchy simultaneously – but also faster. Overall, the results of Brodal et al. [135] show that for sorting, the overhead involved in being cache-oblivious can be small enough in order to allow nice theoretical properties to actually transfer into practical advantages.

5.5.2 External Memory BFS

The implementation of the external memory BFS algorithms [600,555] exploiting disk parallelism on a low cost machine makes BFS viable for massive graphs [19, 20]. On many different classes of graphs, this implementation computes BFS level decomposition of around billion-edge graphs in few *hours* which would have taken the traditional RAM model BFS algorithm [191] several *months*. In fact, the difference between the RAM model algorithm and the external memory algorithms is clearly visible even when more than half of the graph fits in the internal memory. As shown in Figure 5.8, the running time of the traditional BFS algorithm significantly deviates from the predicted RAM performance taking *hours*, rather than *minutes* for random graphs less than double the size of the internal memory. On the other hand, the external BFS implementations referred to as MR_BFS and MM_BFS in the plot, compute the BFS level decomposition in a few *minutes*.

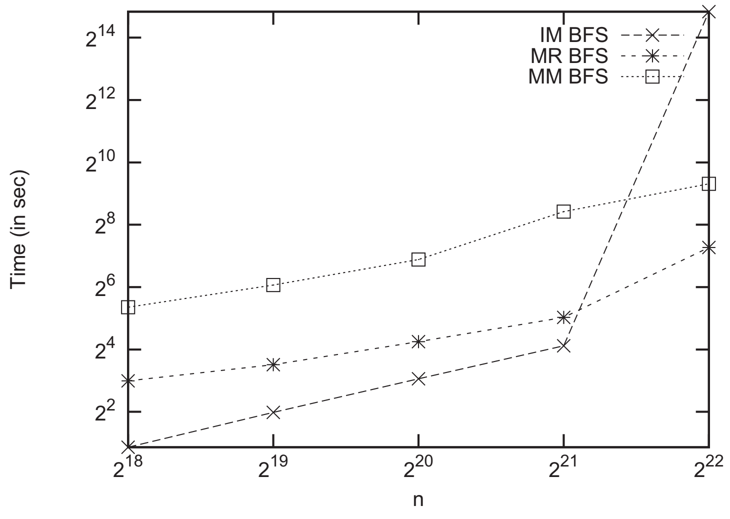

Figure 5.8. Running time of the RAM model BFS algorithm IM_BFS [191] and the external memory BFS algorithms MR_BFS [600] and MM_BFS [555] with respect to the number of nodes (n) of a random graph. The number of edges is always kept at $4n$.

5.5.3 External Suffix Array Construction

The suffix array, a lexicographically sorted array of the suffixes of a string, has received considerable attention lately because of its applications in string matching, genome analysis and text compression. However, most known implementations of suffix array construction could not handle inputs larger than 2 GB. Dementiev et al. [229] show that external memory computation of suffix arrays is feasible. They provide a EM implementation that can process much larger character strings in hours on low cost hardware. In fact, the running time of their implementation is significantly faster than previous external memory implementations.

5.5.4 External A*-Search

In many application domains like model checking and route planning, the state space often grows beyond the available internal memory. Edelkamp et al. [267] propose and implement an external version of A* to search in such state spaces. Embedding their approach in the model checking software SPIN, they can detect deadlocks in an optical telegraph protocol for 20 stations, with an intermediate data requirement of 1.1 Terabytes on hard disk (with only 2.5 GB of available main memory).

5.6 Parallel Bridging Model Libraries

The number of publications on parallel algorithms developed for one of the major bridging models, in particular BSP and CGM, shows their success in the academic world. Moreover, following the Algorithm Engineering paradigm and for an easier use of these models in practice, library standards have been developed. The older one is the BSPlib standard [393], whose corresponding library implementations shall provide methods for the direct transformation of BSP algorithms into parallel applications. According to Bisseling [102], two efficient implementations exist, the Oxford BSP toolset [625] and the Paderborn University BSP library (PUB) [119]. A more recent implementation [766] has been developed, which facilitates the use of BSPlib on all platforms with the message-passing interface MPI. Its objective is to provide BSPlib on top of MPI, making the library portable to most parallel computers. CGMlib is a library following the same ideas for the coarse-grained multicomputer model. So far, there exists only one implementation known to the authors [157]. Although a widespread use of these libraries outside the academic world is not apparent, their influence should not be underestimated. They can, for instance, be used for a gentle introduction to parallel programming [102] and as a basis for distributed web/grid computing [344, 118].

Note that there exist many more languages, libraries, and tools for parallel programming, as well as applications, of course. Even an approximate description of these works would be outside the scope of this chapter. Since they are also not as close to the original models, we instead refer the interested reader to Fox et al. [305] and various handbooks on parallel computing [108, 147, 388, 494, 660]. They cover many aspects of parallel computing from the late 1980s until today.

5.7 Conclusion

The simple models RAM and PRAM have been of great use to designers of both sequential and parallel algorithms. However, they show severe deficiencies as well. The RAM model fails to capture the idiosyncrasies of large data sets that do not fit into main memory, the PRAM does not model the costs arising by inter-processor communication. Since both, parallel computation and the processing of very large data sets, have become more and more important in practice, this has led to the development of more realistic models of computation. The external memory (EM) model has proved to be quite successful in algorithm engineering on problems involving large data sets that do not fit in the main memory and thus, reside on the hard disk. In the parallel setting the bulk-synchronous approach (BSP) is very important, which models inter-processor communication explicitly. Several variants of both have been developed, e. g., to include the specifics of caches (ICM) or of coarse-grained communication (CGM). Although developed for different purposes, all these models have several strategies in common on how to avoid I/O transfer and communication, respectively, in particular the exploitation of locality and the grouping of data before their transmission.

Fundamental techniques for an efficient use of the memory hierarchy or of parallel computers have been illustrated by means of different external memory data structures, cache-aware, cache-oblivious, and parallel algorithms. This has been supplemented by a description of successful implementations of external memory algorithms that facilitate the efficient processing of very large data sets. Also, libraries for an easy implementation of parallel algorithms developed in one of the models mentioned above have been presented. These examples show the impact of realistic computational models on the design and practical implementation of algorithms for these purposes. Moreover, one can say that for very large data sets and complex parallel computations it is hardly possible nowadays to obtain efficient programs without using the techniques and ideas of the models presented in this chapter.

Despite these successes it should be noted that models necessarily have their disadvantages because they are only abstractions and simplifications of the real world. While the interest in new parallel models seemed to be decreasing until the mid 2000s, the general breakthrough of multicore processors has produced a number of new models and in particular practical programming frameworks (parallel languages, runtime environments, etc.). A rather simple model combining parallelism and memory hierarchy issues, in particular with automated optimizations in a hardware-oblivious way, would certainly be a step forward towards even more realistic performance prediction. The very recent proposals on multicore models have yet to prove their suitability in this regard. From a practical perspective it will be very interesting to see which developments in languages and runtime environments will experience widespread adoption both in academia *and* in industry. We believe that a mostly seamless transition from a realistic model to the actual implementation – as previously in the sequential case – will be the key to success.

Bibliography

1. Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck, *Highway dimension, shortest paths, and provably efficient algorithms*, Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 782–793.
2. Jean-Raymond Abrial, *The B-book: Assigning programs to meanings*, Cambridge University Press, August 1996.
3. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, *Specification language*, On the Construction of Programs: An Advanced Course (R. M. McKeag and A. M. Macnaghten, eds.), Cambridge University Press, 1980, pp. 343–410.
4. Dimitris Achlioptas, Marek Chrobak, and John Noga, *Competitive analysis of randomized paging algorithms*, Theoretical Computer Science **234** (2000), no. 1–2, 203–218.
5. Tobias Achterberg, Timo Berthold, Alexander Martin, and Kati Wolter, *SCIP – solving constraint integer programs*, `http://scip.zib.de/`, 2007.
6. Tobias Achterberg, Martin Grötschel, and Thorsten Koch, *Software for teaching modeling of integer programming problems*, ZIB Report 06-23, Zuse Institute Berlin, 2006.
7. Michael J. Ackerman, *The visible human project - getting the data*, `http://www.nlm.nih.gov/research/visible/getting_data.html` (last update: 11 January 2010), 2004.
8. Advanced Micro Devices, Inc., *AMD developer central - ATI stream software development kit (SDK)*, `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`, 2009.
9. Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley, *Cache-oblivious data structures for orthogonal range searching*, Proceedings of the 19th Annual ACM Symposium on Computational Geometry, ACM Press, 2003, pp. 237–245.
10. Pankaj K. Agarwal, Lars A. Arge, T. M. Murali, Kasturi Varadarajan, and Jeffrey Vitter, *I/O efficient algorithms for contour-line extraction and planar graph blocking*, Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1998, pp. 117–126.
11. Alok Aggarwal and Jeffrey S. Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31(9) (1988), 1116–1127.
12. Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl, *On the streaming model augmented with a sorting primitive*, Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS) (2004), 540–549.
13. Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, *PRIMES is in P*, Annals of Mathematics **160** (2004), no. 2, 781–793.
14. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley Publishing Company, 1974.
15. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, *Network flows: Theory, algorithms, and applications*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

16. Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan, *Faster algorithms for the shortest path problem*, Journal of the ACM **37** (1990), no. 2, 213–223.
17. Ravindra K. Ahuja and James B. Orlin, *Use of representative operation counts in computational testing of algorithms*, INFORMS Journal on Computing **8** (1992), 318–330.
18. *AIX versions 3.2 and 4 performance tuning guide*, `http://www.unet.univie.ac.at/aix/aixbman/prftungd/toc.htm`, 1996.
19. Deepak Ajwani, Roman Dementiev, and Ulrich Meyer, *A computational study of external-memory BFS algorithms*, Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 601–610.
20. Deepak Ajwani, Roman Dementiev, Ulrich Meyer, and Vitaly Osipov, *The shortest path problem: The ninth DIMACS implementation challenge*, DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 74, ch. Breadth first search on massive graphs, pp. 291–308, American Mathematical Society, 2009.
21. Deepak Ajwani, Itay Malinger, Ulrich Meyer, and Silvan Toledo, *Characterizing the performance of flash memory storage devices and its impact on algorithm design*, Experimental Algorithms (WEA 2008), LNCS, vol. 5038, Springer, Heidelberg, 2008, pp. 208–219.
22. Selim G. Akl, *Parallel computation: models and methods*, Prentice-Hall, Inc., 1997.
23. Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman, *LogGP: Incorporating long messages into the LogP model for parallel computation*, Journal of Parallel and Distributed Computing **44** (1997), no. 1, 71–79.
24. Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr., *Project Fortress: A multicore language for multicore processors*, Linux Magazine (2007), 38–43.
25. Ernst Althaus, Tobias Polzin, and Siavash V. Daneshmand, *Improving linear programming approaches for the Steiner tree problem*, Research Report MPI-I-2003-1-004, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2003.
26. Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Readings in computer architecture, Morgan Kaufmann Publishers Inc., 1999, pp. 79–81.
27. Nina Amenta and Günter M. Ziegler, *Deformed products and maximal shadows of polytopes*, Contemporary Mathematics, vol. 223, pp. 57–90, American Mathematical Society, 1999.
28. Mohammad M. Amini and Richard S. Barr, *Network reoptimization algorithms: A statistically designed comparison*, ORSA Journal on Computing **5** (1993), no. 4, 395–409.
29. Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger, *STAPL: An adaptive, generic parallel C++ library*, LCPC, 2001, pp. 193–208.
30. Ed Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and Danny Sorensen, *LAPACK users' guide*, 3rd ed., Society for Industrial and Applied Mathematics, 1999.
31. Stephanos Androutsellis-Theotokis and Diomidis Spinellis, *A survey of peer-to-peer content distribution technologies*, ACM Computing Surveys **36** (2004), no. 4, 335–371.
32. Yash P. Aneja, *An integer linear programming approach to the Steiner problem in graphs*, Networks **10** (1980), 167–178.

33. David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook, *The traveling salesman problem: A computational study*, Princeton University Press, 2006.
34. Krzysztof Apt, *Principles of constraint programming*, Cambridge University Press, 2003.
35. Lars Arge, *The buffer tree: A new technique for optimal I/O-algorithms*, Algorithms and Data Structures, 4th International Workshop, WADS '95 (Selim G. Akl, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, eds.), LNCS, vol. 955, Springer, Heidelberg, 1995, pp. 334–345.
36. ———, *External memory data structures*, ESA 2001 (Friedhelm Meyer auf der Heide, ed.), LNCS, vol. 2161, Springer, Heidelberg, 2001, pp. 1–29.
37. ———, *External memory data structures*, Handbook of Massive Data Sets (James Abello, Panos M. Pardalos, and Mauricio G. C. Resende, eds.), Kluwer Academic Publishers, 2002, pp. 313–357.
38. Lars Arge, Michael Bender, Erik Demaine, Bryan Holland-Minkley, and J. Ian Munro, *Cache-oblivious priority-queue and graph algorithms*, Proceedings of the 34th ACM Symposium on Theory of Computing (STOC), ACM Press, 2002, pp. 268–276.
39. Lars Arge, Gerth S. Brodal, and Rolf Fagerberg, *Cache-oblivious data structures*, Handbook on Data Structures and Applications (D. P. Mehta and S. Sahni, eds.), CRC Press, 2004.
40. Lars Arge, Gerth S. Brodal, and Laura Toma, *On external-memory MST, SSSP and multi-way planar graph separation*, Journal of Algorithms **53(2)** (2004), 186–206.
41. Lars Arge, Jeffrey Chase, Jeffrey Vitter, and Rajiv Wickremesinghe, *Efficient sorting using registers and caches*, ACM Journal of Experimental Algorithmics **7** (2002), no. 9, 1–17.
42. Lars Arge, Mark de Berg, Herman Haverkort, and Ke Yi, *The priority R-tree: A practically efficient and worst-case optimal R-tree*, SIGMOD International Conference on Management of Data (2004), 347–358.
43. Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava, *Fundamental parallel algorithms for private-cache chip multiprocessors*, Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2008, pp. 197–206.
44. Lars Arge, Octavian Procopiuc, and Jeffrey S. Vitter, *Implementing I/O-efficient data structures using TPIE*, 10th European Symposium on Algorithms (ESA) (Rolf H. Möhring and Rajeev Raman, eds.), LNCS, vol. 2461, Springer, Heidelberg, 2002, pp. 88–100.
45. Lars Arge and Laura Toma, *Simplified external memory algorithms for planar DAGs*, Algorithm Theory - SWAT 2004, (Torben Hagerup and Jyrki Katajainen, eds.), LNCS, vol. 2461, Springer, Heidelberg, 2004, pp. 493–503.
46. Lars Arge, Laura Toma, and Norbert Zeh, *I/O-efficient topological sorting of planar DAGs*, Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM, 2003, pp. 85–93.
47. Lars Arge and Jeffrey S. Vitter, *Optimal dynamic interval management in external memory*, Proceedings of the 37th Annual IEEE Symposium on Foundations Of Computer Science (FOCS), 1996, pp. 560–569.
48. Sanjeev Arora, *Polynomial time approximation schemes for the Euclidean traveling salesman and other geometric problems*, Journal of the ACM **45** (1998), 753–782.

49. Sunil Arya and David M. Mount, *Approximate nearest neighbor queries in fixed dimensions*, Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1993, pp. 271–280.
50. Anthony C. Atkinson, *Plots, transformations and regression: an introduction to graphical methods of diagnostic regression analysis*, Oxford University Press, U.K., 1987.
51. Franz Aurenhammer and Rolf Klein, *Voronoi diagrams, ch. 5*, Handbook of Computational Geometry (Jörg-Rüdiger Sack and Jorge Urrutia, eds.), North-Holland, 1999, pp. 201–290.
52. Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco Preparata, and Mariette Yvinec, *Evaluating signs of determinants using single precision arithmetic*, Algorithmica **17** (1997), no. 2, 111–132.
53. Brian Babcock, Shirnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom, *Models and issues in data stream systems*, ACM PODS (2002), 1–16.
54. David A. Bader, Varun Kanade, and Kamesh Madduri, *SWARM: A parallel programming framework for multi-core processors*, 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), 2007, pp. 1–8.
55. David A. Bader and Kamesh Madduri, *SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks*, 22nd International Parallel and Distributed Processing Symposium (IPDPS 2008), 2008, pp. 1–12.
56. David A. Bader, Bernard M. E. Moret, and Peter Sanders, *Algorithm engineering for parallel computation*, Experimental Algorithmics. From Algorithm Design to Robust and Efficient Software (Rudolf Fleischer, Bernard Moret, and Erik Meineche Schmidt, eds.), LNCS, vol. 2547, Springer, Heidelberg, 2002, pp. 1–23.
57. Michael Bader and Christoph Zenger, *Cache oblivious matrix multiplication using an element ordering based on a Peano curve*, Linear Algebra and its Applications (Special Issue in honor of Friedrich Ludwig Bauer) **417** (2006), no. 2-3, 301–313.
58. Ricardo Baeza-Yates, Eduardo F. Barbosa, and Nivio Ziviani, *Hierarchies of indices for text searching*, Journal of Information Systems **21** (1996), 497–514.
59. Helmut Balzert, *Lehrbuch der Software-Technik*, Spektrum Akademischer Verlag, Heidelberg, 1996.
60. Richard S. Barr, Bruce L. Golden, James P. Kelly, Mauricio G. C. Resende, and William R. Stewart Jr., *Designing and reporting on computational experiments with heuristic methods*, Journal of Heuristics **1** (1995), no. 1, 9–32.
61. Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner, *Engineering label-constrained shortest-path algorithms*, Shortest Path Computations: Ninth DIMACS Challenge (Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, eds.), DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 309–319.
62. Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe, *Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router*, Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02) (Rolf H. Möhring and Rajeev Raman, eds.), LNCS, vol. 2461, Springer, Heidelberg, 2002, pp. 126–138.
63. Chris Barrett, Riko Jacob, and Madhav V. Marathe, *Formal-language-constrained path problems*, SIAM Journal on Computing **30** (2000), no. 3, 809–837.

64. Roman Barták, *Constraint programming: In pursuit of the holy grail*, Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic, MatFyzPress, 1999.
65. Victor R. Basili, Barry Boehm, Al Davis, Watts S. Humphrey, Nancy Leveson, Nancy R. Mead, John D. Musa, David L. Parnas, Shari L. Pfleeger, and Elaine Weyuker, *New year's resolutions for software quality*, IEEE Softw. **21** (2004), no. 1, 12–13.
66. Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes, *In transit to constant shortest-path queries in road networks*, Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07), SIAM, 2007, pp. 46–59.
67. Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes, *Fast routing in road networks with transit nodes*, Science **316** (2007), no. 5824, 566.
68. Holger Bast and Ingmar Weber, *Don't compare averages*, 4th International Workshop on Experimental and Efficient Algorithms (WEA) (Sotiris E. Nikoletseas, ed.), LNCS, no. 3503, Springer, Heidelberg, 2005, pp. 67–76.
69. Vicente H. F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler, *Parallel geometric algorithms for multi-core computers*, Proceedings of the 25th Annual ACM Symposium on Computational Geometry, ACM, 2009, pp. 217–226.
70. Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter, *Time-dependent contraction hierarchies*, Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09), SIAM, 2009, pp. 97–105.
71. Reinhard Bauer and Daniel Delling, *SHARC: Fast and robust unidirectional routing*, Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08) (Ian Munro and Dorothea Wagner, eds.), SIAM, April 2008, pp. 13–26.
72. ———, *SHARC: Fast and robust unidirectional routing*, ACM Journal of Experimental Algorithmics **14** (2009), 2.4–2.29, Special Section on Selected Papers from ALENEX 2008.
73. Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner, *Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm*, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08) (Catherine C. McGeoch, ed.), LNCS, vol. 5038, Springer, Heidelberg, 2008, pp. 303–318.
74. ———, *Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm*, ACM Journal of Experimental Algorithmics **15** (2010), no. 3.
75. Reinhard Bauer, Daniel Delling, and Dorothea Wagner, *Shortest-path indices: Establishing a methodology for shortest-path problems*, Tech. Report 2007-14, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2007.
76. Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide, *Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model.*, Theoretical Computer Science **203** (1998), no. 2, 175–203.
77. Rudolf Bayer and Edward M. McCreight, *Organization and maintenance of large ordered indexes*, Acta Informatica (1972), 173–189.
78. John E. Beasley, *An algorithm for the Steiner tree problem in graphs*, Networks **14** (1984), 147–159.
79. ———, *OR-Library: Distributing test problems by electronic mail*, Journal of the Operation Research Society **41** (1990), 1069–1072.
80. John E. Beasley and Abilio Lucena, *A branch and cut algorithm for the Steiner problem in graphs*, Networks **31** (1998), 39–59.

81. Andreas Beckmann, Roman Dementiev, and Johannes Singler, *Building a parallel pipelined external memory algorithm library*, 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2009.
82. Nelson H. F. Beebe, *GNU scientific library*, http://www.math.utah.edu/software/gsl.html, 2001.
83. René Beier and Berthold Vöcking, *Probabilistic analysis of knapsack core algorithms*, Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2004, pp. 468–477.
84. ______, *Random knapsack in expected polynomial time*, Journal of Computer and System Sciences **69** (2004), no. 3, 306–329.
85. ______, *An experimental study of random knapsack problems*, Algorithmica **45** (2006), no. 1, 121–136.
86. ______, *Typical properties of winners and losers in discrete optimization*, SIAM Journal on Computing **35** (2006), no. 4, 855–881.
87. Shai Ben-David and Allan Borodin, *A new measure for the study of on-line algorithms*, Algorithmica **11** (1994), no. 1, 73–91.
88. Michael A. Bender, Richard Cole, and Rajeev Raman, *Exponential structures for cache-oblivious algorithms*, Proceedings of 29th International Colloquium on Automata, Languages, and Programming (ICALP) (Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, eds.), LNCS, vol. 2380, Springer, Heidelberg, 2002, pp. 195–207.
89. Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton, *Cache-oblivious B-trees*, Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, 2000, pp. 399–409.
90. Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu, *A locality-preserving cache-oblivious dynamic dictionary*, Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 29–38.
91. Jon L. Bentley, *Multidimensional binary search trees used for associative searching*, Communications of the ACM **18** (1975), no. 9, 509–517.
92. ______, *Experiments on traveling salesman heuristics*, Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 91–99.
93. ______, *Programming perls*, Addison Wesley Professional, 2000.
94. Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, and Nicola Wolpert, *EXACUS: Efficient and exact algorithms for curves and surfaces*, Algorithms - ESA 2005 (Gerth Stølting Brodal and Stefano Leonardi, eds.), LNCS, vol. 3669, Springer, Heidelberg, 2005, pp. 155–166.
95. Berkeley Unified Parallel C (UPC) Project, *Berkeley Unified Parallel C (UPC) project*, http://upc.lbl.gov/, 2009.
96. Piotr Berman and Viswanathan Ramaiyer, *Improved approximations for the Steiner tree problem*, Journal of Algorithms **17** (1994), 381–408.
97. Emanuele Berretini, Gianlorenzo D'Angelo, and Daniel Delling, *Arc-flags in dynamic graphs*, ATMOS'09 - Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.
98. Thomas Beth and Dieter Gollmann, *Algorithm engineering for public key algorithms*, IEEE Journal on Selected Areas in Communications **7** (1989), 458–466.

99. Gianfranco Bilardi, Andrea Pietracaprina, and Geppino Pucci, *Handbook of parallel computing: Models, algorithms and applications*, ch. Decomposable BSP: A Bandwidth-Latency Model for Parallel and Hierarchical Computation, CRC Press, 2007.
100. Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, Kieran T. Herley, and Paul G. Spirakis, *BSP versus LogP*, Algorithmica **24** (1999), no. 3-4, 405–422.
101. Robert V. Binder, *Testing object-oriented systems: Models, patterns, and tools*, Addison-Wesley Professional, October 1999.
102. Rob H. Bisseling, *Parallel scientific computation. A structured approach using BSP and MPI*, Oxford University Press, 2004.
103. Robert E. Bixby, *Solving real-world linear programs: A decade and more of progress*, Operations Research **50** (2002), 3–15.
104. Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto, *Fourier meets Möbius: fast subset convolution*, Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), ACM, 2007, pp. 67–74.
105. L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and Richard C. Whaley, *An updated set of basic linear algebra subprograms (BLAS)*, ACM Trans. Math. Software **28** (2002), no. 2, 135–151.
106. Paul Blaer and Peter K. Allen, *Topbot: automated network topology detection with a mobile robot*, Proceedings of the 2003 IEEE International Conference on Robotics and Automation, 2003, pp. 1582–1587.
107. Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash, *Compact representations of separable graphs*, Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 679–688.
108. Jacek Blazewicz, Denis Trystram, Klaus Ecker, and Brigitte Plateau (eds.), *Handbook on parallel and distributed processing*, Springer-Verlag New York, Inc., 2000.
109. *Blitz++: Object-oriented scientific computing*, http://www.oonumerics.org/blitz/, 2005, Version 0.9.
110. Joshua Bloch, *Effective Java: Programming language guide*, Java series, Addison-Wesley, Boston, 2001.
111. Toby Bloom and Ted Sharpe, *Managing data from high-throughput genomic processing: A case study*, Very Large Data Bases (VLDB) (2004), 1198–1201.
112. Manuel Blum and Sampath Kannan, *Designing programs that check their work*, Journal of the ACM **42** (1995), no. 1, 269–291.
113. Hans L. Bodlaender, *A linear-time algorithm for finding tree-decompositions of small treewidth*, SIAM J. Computing **25** (1996), no. 6, 1305–1317.
114. Hans L. Bodlaender and Jan A. Telle, *Space-efficient construction variants of dynamic programming*, Nordic Journal of Computing **11** (2004), 374–385.
115. Andrej Bogdanov and Luca Trevisan, *Average-case complexity*, Found. Trends Theor. Comput. Sci. **2** (2006), no. 1, 1–106.
116. Béla Bollobás, *Modern graph theory*, Springer-Verlag, New York, 2002.
117. Andre B. Bondi, *Characteristics of scalability and their impact on performance*, WOSP '00: Proceedings of the 2nd international workshop on Software and performance (New York, NY, USA), ACM Press, 2000, pp. 195–203.
118. Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide, *A web computing environment for parallel algorithms in Java*, Journal on Scalable Computing: Practice and Experience **7** (2006), no. 2, 1–14.
119. Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping, *The Paderborn University BSP (PUB) library*, Parallel Computing **29** (2003), no. 2, 187–207.

120. *Boost C++ Libraries*, http://www.boost.org, 2010, version 1.42.
121. Manjit Borah, Robert M. Owens, and Mary J. Irwin, *A fast and simple Steiner routing heuristic*, Discrete Applied Mathematics **90** (1999), 51–67.
122. Christian Borgelt and Rudolf Kruse, *Unsicherheit und Vagheit: Begriffe, Methoden, Forschungsthemen*, KI, Künstliche Intelligenz **3/01** (2001), 18–24.
123. Egon Börger and Robert Stärk, *Abstract state machines: A method for high-level system design and analysis*, Springer-Verlag, 2003.
124. Karl H. Borgwardt, *The simplex method – a probabilistic analysis*, Springer, 1987.
125. Allan Borodin and Ran El-Yaniv, *Online computation and competitive analysis*, Cambridge University Press, 1998.
126. Glencora Borradaile, Claire Kenyon-Mathieu, and Philip N. Klein, *A polynomial-time approximation scheme for Steiner tree in planar graphs*, Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007, pp. 1285–1294.
127. ———, *Steiner tree in planar graphs: An $O(n \log n)$ approximation scheme with singly-exponential dependence on epsilon*, Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings (Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, eds.), LNCS, vol. 4619, Springer, Heidelberg, 2007, pp. 275–286.
128. Nirmal K. Bose and A.K. Garga, *Neural network design using Voronoi diagrams*, IEEE Transactions on Neural Networks **4** (1993), no. 5, 778–787.
129. Ulrich Brandes and Thomas Erlebach (eds.), *Network analysis*, LNCS, vol. 3418, Springer, Heidelberg, 2005.
130. Gerth S. Brodal, *Cache-oblivious algorithms and data structures*, Algorithm Theory - SWAT 2004 (Torben Hagerup and Jyrki Katajainen, eds.), LNCS, vol. 3111, Springer, Heidelberg, 2004, pp. 3–13.
131. Gerth S. Brodal and Rolf Fagerberg, *Cache oblivious distribution sweeping*, Proceedings of the 29th International Colloquium Automata, Languages and Programming (ICALP) (Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, eds.), LNCS, vol. 2380, Springer, Heidelberg, 2002, pp. 426–438.
132. ———, *Funnel heap - a cache oblivious priority queue*, Proceedings of the 13th International Symposium on Algorithms and Computation (Prosenjit Bose and Pat Morin, eds.), LNCS, vol. 2518, Springer, Heidelberg, 2002, pp. 219–228.
133. Gerth S. Brodal, Rolf Fagerberg, and Riko Jacob, *Cache oblivious search trees via binary trees of small height*, Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 39–48.
134. Gerth S. Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh, *Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths*, Algorithm Theory - SWAT 2004 (Torben Hagerup and Jyrki Katajainen, eds.), LNCS, vol. 3111, Springer, Heidelberg, 2004, pp. 480–492.
135. Gerth S. Brodal, Rolf Fagerberg, and Kristoffer Vinther, *Engineering a cache-oblivious sorting algorithm*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2004, pp. 4–17.
136. Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion, *Interval arithmetic yields efficient dynamic filters for computational geometry*, Proceedings of the 14th Annual ACM Symposium on Computational Geometry, 1998, pp. 165–174.
137. Hervé Brönnimann, Ioannis Z. Emiris, Victor Y. Pan, and Sylvain Pion, *Computing exact geometric predicates using modular arithmetic with single precision*, Proceedings of the 13th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1997, pp. 174–182.

138. Hervé Brönnimann and Mariette Yvinec, *Efficient exact evaluation of signs of determinants*, Proceedings of the 13th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1997, pp. 166–173.
139. Anthony Brooke, David Kendrick, Alexander Meeraus, and Richard E. Rosenthal, *GAMS - A user's guide*, 2006.
140. Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner, *Space-efficient SHARC-routing*, SEA 2010 (Paola Festa, ed.), LNCS, vol. 6049, Springer, Heidelberg, 2010, pp. 47–58.
141. Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn, and Stefan Schirra, *Efficient exact geometric computation made easy*, Proceedings of the 15th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1999, pp. 341–350.
142. ———, *A strong and easily computable separation bound for arithmetic expressions involving radicals*, Algorithmica **27** (2000), 87–99.
143. Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt, *A separation bound for real algebraic expressions*, Proceedings of the 9th Annual European Symposium on algorithms (ESA 2001) (F. Meyer auf der Heide, ed.), LNCS, vol. 2161, Springer, Heidelberg, 2001, pp. 254–265.
144. Christoph Burnikel, Stefan Funke, and Michael Seel, *Exact geometric computation using cascading*, International Journal of Computational Geometry and Applications **11** (2001), no. 3, 245–266.
145. Michael Bussieck, *Optimal lines in public rail transport*, Ph.D. thesis, Technische Universität Braunschweig, 1998.
146. David R. Butenhof, *Programming with POSIX threads*, Addison-Wesley, 1997.
147. Rajkumar Buyya (ed.), *High performance cluster computing: Programming and applications*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.
148. *C++ applications*, http://public.research.att.com/~bs/applications.html, 2009.
149. Scott Camazine, Nigel R. Franks, James Sneyd, Eric Bonabeau, Jean-Louis Deneubourg, and Guy Theraula, *Self-organization in biological systems*, Princeton University Press, Princeton, NJ, USA, 2001.
150. George C. Caragea, A. Beliz Saybasili, Xingzhi Wen, and Uzi Vishkin, *Brief announcement: Performance potential of an easy-to-program PRAM-on-chip prototype versus state-of-the-art processor*, SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009, ACM, 2009, pp. 163–165.
151. *CGAL: Computational Geometry Algorithms Library*, http://www.cgal.org/, 2009, Version 3.4.
152. *CGAL user and reference manual*, 2009, http://www.cgal.org/Manual/index.html.
153. *cgmLIB: A library for coarse-grained parallel computing*, http://lib.cgmlab.org/, 2003, version 0.9.5 Beta.
154. Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria, *The organic grid: Self-organizing computation on a peer-to-peer network*, IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans **35** (2005), no. 3, 373–384.
155. Bradford L. Chamberlain, David Callahan, and Hans P. Zima, *Parallel programmability and the Chapel language*, Int. J. High Perform. Comput. Appl. **21** (2007), no. 3, 291–312.

156. Albert Chan and Frank Dehne, *CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters*, PVM/MPI (Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, eds.), LNCS, vol. 2840, Springer, Heidelberg, 2003, pp. 117–125.

157. Albert Chan, Frank Dehne, and Ryan Taylor, *CGMGRAPH/CGMLIB: Implementing and testing CGM graph algorithms on PC clusters and shared memory machines*, International Journal of High Performance Computing Applications **19** (2005), no. 1, 81–97.

158. Timothy M. Chan and Eric Y. Chen, *Optimal in-place algorithms for 3-D convex hulls and 2-D segment intersection*, Proceedings of the 25th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 2009, pp. 80–87.

159. Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald, *Parallel programming in openMP*, Morgan Kaufmann, San Francisco, 2000.

160. Ee-Chien Chang, Sung W. Choi, DoYong Kwon, Hyungja Park, and Chee-Keng Yap, *Shortest path amidst disc obstacles is computable*, Proceedings of the 21st Annual ACM Symposium on Computational Geometry, ACM Press, 2005, pp. 116–125.

161. Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable shared memory parallel programming*, MIT Press, 2007.

162. Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar, *X10: an object-oriented approach to non-uniform cluster computing*, OOPSLA, 2005, pp. 519–538.

163. Bernard Chazelle, *Triangulating a simple polygon in linear time*, Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 29–38.

164. ______, *Triangulating a simple polygon in linear time*, Discrete Computational Geometry **6** (1991), 485–524.

165. ______, *Cuttings*, Handbook of Data Structures and Applications, CRC Press, 2005.

166. Jianer Chen, Iyad A. Kanj, and Ge Xia, *Improved parameterized upper bounds for vertex cover*, Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science (MFCS '06) (Rastislav Kralovic and Pawel Urzyczyn, eds.), LNCS, vol. 4162, Springer, Heidelberg, 2006, pp. 238–249.

167. Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik, *Shortest paths algorithms: Theory and experimental evaluation*, Mathematical Programming **73** (1996), 129–174.

168. Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter, *External-memory graph algorithms*, Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1995, pp. 139–149.

169. Samir Chopra, E.R. Gorres, and M.R. Rao, *Solving the Steiner tree problem on a graph using branch and cut*, ORSA Journal on Computing **4** (1992), 320–335.

170. Samir Chopra and M.R. Rao, *The Steiner tree problem I: Formulations, compositions and extension of facets*, Mathematical Programming **64** (1994), 209–229.

171. El-Arbi Choukhmane, *Une heuristique pour le probleme de l'arbre de Steiner*, RAIRO Rech. Opér. **12** (1978), 207–212.

172. Nicos Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Tech. Report 388, GSIA, Carnegie-Mellon University, Pittsburgh, 1976.
173. Chris Chu and Yiu-Chung Wong, *Fast and accurate rectilinear Steiner minimal tree algorithm for VLSI design*, ISPD '05: Proceedings of the 2005 international symposium on Physical design (New York, NY, USA), ACM Press, 2005, pp. 28–35.
174. Cilk Arts, *Multicore programming software*, `http://www.cilk.com/`, 2009.
175. *CLAPACK: f2c'ed version of LAPACK*, `http://www.netlib.org/clapack/`, 2008, Version 3.1.1.1.
176. David R. Clark and J. Ian Munro, *Efficient suffix trees on secondary storage*, Proceedings of the 7th Annual ACM-SIAM Symposium On Discrete Algorithms (SODA) (1996), 383–391.
177. Kenneth L. Clarkson, *Safe and effective determinant evaluation*, Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS) (Pittsburgh, PA), October 1992, pp. 387–395.
178. Alan Cobham, *The intrinsic computational difficulty of functions*, Proc. 1964 International Congress for Logic, Methodology, and Philosophy of Science (Y. Bar-Hillel, ed.), North-Holland, Amsterdam, 1964, pp. 24–30.
179. Ernest J. Cockayne and Denton E. Hewgill, *Exact computation of Steiner minimal trees in the plane*, Information Processing Letters **22** (1986), 151–156.
180. ———, *Improved computation of plane Steiner minimal trees*, Algorithmica **7** (1992), no. 2/3, 219–229.
181. Marie Coffin and Matthew J. Saltzmann, *Statistical analysis of computational tests of algorithms and heuristics*, INFORMS Journal on Computing **12** (2000), no. 1, 24–44.
182. Douglas Comer, *The ubiquitous B-tree*, ACM Computing Surveys (1979), 121–137.
183. William J. Conover, *Practical nonparametric statistics*, John Wiley & Sons, 1980.
184. Stephen A. Cook, *The complexity of theorem-proving procedures*, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC), 1971, pp. 151–158.
185. William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver, *Combinatorial optimization*, Wiley, New York, 1998.
186. Don Coppersmith and Shmuel Winograd, *Matrix multiplication via arithmetic progressions*, J. Symb. Comput. **9** (1990), no. 3, 251–280.
187. Massimo Coppola and Martin Schmollinger, *Hierarchical models and software tools for parallel programming*, Algorithms for Memory Hierarchies (Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, eds.), LNCS, vol. 2625, Springer, Heidelberg, 2003, pp. 320–354.
188. Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento, *An improved algorithm for matching large graphs*, 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, May 2001.
189. *The Core library*, `http://cs.nyu.edu/exact/core_pages/index.html`, 2004, Version 1.7.
190. Thomas H. Cormen and Michael T. Goodrich, *A bridging model for parallel computation, communication, and I/O*, ACM Computing Surveys **28** (1996), Article No. 208.
191. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, 3rd ed., MIT Press, 2009.

192. Ricardo Corrêa, Inês Dutra, Mario Fiallos, and Fernando Gomes (eds.), *Models for parallel and distributed computation. Theory, algorithmic techniques and applications*, Kluwer, 2002.
193. Michael Cosnard and Denis Trystram, *Parallel algorithms and architectures*, PWS Publishing Co., 1995.
194. Richard Courant and Herbert Robbins, *What is mathematics?*, Oxford University Press, 1941.
195. Andreas Crauser and Kurt Mehlhorn, *LEDA-SM, extending LEDA to secondary memory*, 3rd International Workshop on Algorithmic Engineering (WAE) (Jeffrey Scott Vitter and Christos D. Zaroliagis, eds.), LNCS, vol. 1668, Springer, Heidelberg, 1999, pp. 228–242.
196. Harlan P. Crowder, Ron S. Dembo, and John M. Mulvey, *Reporting computational experiments in mathematical programming*, Mathematical Programming **15** (1978), 316–329.
197. ———, *On reporting computational experiments with mathematical software*, ACM Transactions on Mathematical Software **5** (1979), no. 2, 193–203.
198. David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus E. Schauser, Ramesh Subramonian, and Thorsten von Eicken, *LogP: a practical model of parallel computation*, Commun. ACM **39** (1996), no. 11, 78–85.
199. David E. Culler, Jaswinder P. Singh, and Anoop Gupta, *Parallel computer architecture - a hardware/software approach*, Morgan Kaufmann, 1999.
200. Ole-Johan Dahl, Edsger W. Dijkstra, and Charles A. R. Hoare, *Structured programming*, Academic Press, New York, 1972.
201. Aldo Dall'Osso, *Computer algebra systems as mathematical optimizing compilers*, Science of Computer Programming **59** (2006), no. 3, 250–273.
202. George B. Dantzig, *Linear programming and extensions*, Princeton University Press, Princeton, NJ, 1963.
203. *Dash optimization – Leading optimization software*, http://www.dashoptimization.com/home/products/products_optimizer.html, 2007.
204. M. Poggi de Aragão and Renato F. Werneck, *On the implementation of MST-based heuristics for the Steiner problem in graphs*, Proceedings of the Fourth International Workshop on Algorithm Engineering and Experiments (ALENEX'02) (David M. Mount and Clifford Stein, eds.), LNCS, vol. 2409, Springer, Heidelberg, 2002, pp. 1–15.
205. Mark de Berg, *Linear size binary space partitions for fat objects*, Proceedings of the 3rd Annual European Symposium on Algorithms (ESA 1995) (Paul G. Spirakis, ed.), LNCS, vol. 979, Springer, Heidelberg, 1995, pp. 252–263.
206. Mark de Berg, Otfried Cheong, Marc van Krefeld, and Mark Overmars, *Computational geometry: Algorithms and applications*, 3rd rev. ed., Springer-Verlag, 2008.
207. Mark de Berg, A. Frank van der Stappen, Jules Vleugels, and Matthew J. Katz, *Realistic input models for geometric algorithms*, Algorithmica **34** (2002), no. 1, 81–97.
208. Maurice de Kunder, *Geschatte grootte van het geïndexeerde world wide web*, Master's thesis, Universiteit van Tilburg, 2008.
209. Pilar de la Torre and Clyde P. Kruskal, *Submachine locality in the bulk synchronous setting (extended abstract)*, Proc. 2nd Intl. Euro-Par Conference (Euro-Par'96) - Volume II, Springer-Verlag, 1996, pp. 352–358.
210. Angela Dean and Daniel Voss, *Design and analysis of experiments*, Springer Texts in Statistics, Springer, 1999.

211. Rina Dechter and Judea Pearl, *Tree clustering for constraint networks*, Artificial Intelligence **38** (1989), no. 3, 353–366.
212. Frank Dehne, *Guest editor's introduction*, Algorithmica **24** (1999), no. 3-4, 173–176.
213. ______, *Guest editor's introduction*, Algorithmica **45** (2006), no. 3, 263–267.
214. Frank Dehne, Wolfgang Dittrich, and David Hutchinson, *Efficient external memory algorithms by simulating coarse-grained parallel algorithms*, Algorithmica **36** (2003), 97–122.
215. Frank Dehne, Wolfgang Dittrich, David Hutchinson, and Anil Maheshwari, *Bulk synchronous parallel algorithms for the external memory model*, Theory Comput. Systems **35** (2002), 567–597.
216. Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin, *Scalable parallel computational geometry for coarse grained multicomputers.*, Int. J. Comput. Geometry Appl. **6** (1996), no. 3, 379–400.
217. Theodorus J. Dekker, *A floating-point technique for extending the available precision*, Numerische Mathematik **18** (1971), no. 3, 224–242.
218. Daniel Delling, *Time-dependent SHARC-routing*, Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08) (Dan Halperin and Kurt Mehlhorn, eds.), LNCS, vol. 5193, Springer, Heidelberg, 2008, Best Student Paper Award - ESA Track B, pp. 332–343.
219. ______, *Engineering and augmenting route planning algorithms*, Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.
220. ______, *Time-dependent SHARC-routing*, Algorithmica (2009), Special Issue: European Symposium on Algorithms 2008.
221. Daniel Delling, Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter, *Exact routing in large road networks using contraction hierarchies*, submitted to Transportation Science, 2009.
222. Daniel Delling and Giacomo Nannicini, *Bidirectional core-based routing in dynamic time-dependent road networks*, Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08) (Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, eds.), LNCS, vol. 5369, Springer, Heidelberg, 2008, pp. 813–824.
223. Daniel Delling, Thomas Pajor, and Dorothea Wagner, *Accelerating multi-modal route planning by access-nodes*, Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09) (Amos Fiat and Peter Sanders, eds.), LNCS, vol. 5757, Springer, Heidelberg, September 2009, pp. 587–598.
224. Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner, *Engineering route planning algorithms*, Algorithmics (Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, eds.), LNCS, vol. 5515, Springer, Heidelberg, 2009, pp. 117–139.
225. ______, *Highway hierarchies star*, Shortest Path Computations: Ninth DIMACS Challenge (Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, eds.), DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 141–174.
226. Daniel Delling and Dorothea Wagner, *Landmark-based routing in dynamic graphs*, Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07) (Camil Demetrescu, ed.), LNCS, vol. 4525, Springer, Heidelberg, 2007, pp. 52–65.
227. ______, *Pareto paths with SHARC*, Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09) (Jan Vahrenhold, ed.), LNCS, vol. 5526, Springer, Heidelberg, June 2009, pp. 125–136.

228. ———, *Time-dependent route planning*, Robust and Online Large-Scale Optimization (Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, eds.), LNCS, vol. 5868, Springer, Heidelberg, 2009, pp. 207–230.
229. Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders, *Better external memory suffix array construction*, ACM Journal of Experimental Algorithms **12** (2008), no. 3.4, 1–24.
230. Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders, *Engineering a sorted list data structure for 32 bit keys*, ALENEX04: Algorithm Engineering and Experiments, SIAM, 2004, pp. 142–151.
231. Roman Dementiev, Lutz Kettner, and Peter Sanders, *STXXL: Standard template library for XXL data sets*, Software: Practice and Experience **38** (2008), no. 6, 589–637.
232. Camil Demetrescu, Irene Finocchi, and Andrea Ribichini, *Trading off space for passes in graph streaming problems*, Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 714–723.
233. Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (eds.), *Shortest path computations: Ninth DIMACS challenge*, DIMACS Book, vol. 74, American Mathematical Society, 2009.
234. Camil Demetrescu and Giuseppe F. Italiano, *What do we learn from experimental algorithmics?*, MFCS (Mogens Nielsen and Branislav Rovan, eds.), LNCS, vol. 1893, Springer, Heidelberg, 2000, pp. 36–51.
235. Camil Demetrescu and Giuseppe F. Italiano, *Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures*, Journal of Discrete Algorithms **4** (2006), no. 3.
236. Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume, *Software testing and evaluation*, Benjamin-Cummings Publishing, Redwood City, 1987.
237. James Demmel and Yozo Hida, *Fast and accurate floating point summation with application to computational geometry*, Numerical Algorithms **37** (2005), 101–112.
238. National Institute of Standards Department of Commerce and Technology, *Announcing request for candidate algorithm nominations for the advanced encryption standard (AES)*, Federal Register **62** (1997), no. 177, 48051–48058.
239. René Descartes, *Principia philosophiae*, Ludovicus Elzevirius, 1644.
240. Amit Deshpande and Daniel A. Spielman, *Improved smoothed analysis of the shadow vertex simplex method*, Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2005, pp. 349–356.
241. Robert B. Dial, *Algorithm 360: shortest-path forest with topological ordering [H]*, Communications of the ACM **12** (1969), no. 11, 632–633.
242. Reinhard Diestel, *Graph theory*, Graduate Texts in Mathematics, vol. 173, Springer, 2005.
243. Martin Dietzfelbinger, *Primality testing in polynomial time*, Springer, 2004.
244. Edsger W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271.
245. ———, *Notes on structured programming*, circulated privately, April 1970.
246. Paul DiLascia, *What makes good code good?*, MSDN Magazine **19** (2004), no. 7, 144.
247. John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl, *Globally distributed content delivery*, IEEE Internet Computing **6** (2002), no. 5, 50–58.
248. *DIMACS implementation challenges*, `http://dimacs.rutgers.edu/Challenges/`, 2006.

249. *DIMACS TSP challenge*, http://www.research.att.com/~dsj/chtsp/, 2006.
250. *Website of Dinkumware's STL implementation*, http://www.dinkumware.com/cpp.aspx, 2006.
251. Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee, *Multi-criteria shortest paths in time-dependent train networks*, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08) (Catherine C. McGeoch, ed.), LNCS, vol. 5038, Springer, Heidelberg, June 2008, pp. 347–361.
252. *DOC++*, http://docpp.sourceforge.net/, 2003.
253. Reza Dorrigiv, Alejandro López-Ortiz, and Alejandro Salinger, *Optimal speedup on a low-degree multi-core parallel architecture (LoPRAM)*, SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (New York, NY, USA), ACM, 2008, pp. 185–187.
254. Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss, *Cache optimization for structured and unstructured grid multigrid*, Elect. Trans. Numer. Anal. **10** (2000), 21–40.
255. Peter Drayton, Ben Albahari, and Ted Neward, *C# in a nutshell: A desktop quick reference*, second ed., In a nutshell, O'Reilly & Associates, Inc., 2003.
256. Stuart E. Dreyfus and Robert A. Wagner, *The Steiner problems in graphs*, Networks **1** (1971), 195–207.
257. Ding-Zhu Du and Xinzhen Cheng (eds.), *Steiner trees in industries*, Kluwer Academic Publishers, 2001.
258. Ding-Zhu Du and Frank K. Hwang, *A proof of the Gilbert-Pollak conjecture on the Steiner ratio*, Algorithmica **7** (1992), 121–135.
259. Zilin Du, Maria Eleftheriou, José E. Moreira, and Chee-Keng Yap, *Hypergeometric functions in exact geometric computation*, Electronic Notes in Theoretical Computer Science **66** (2002), no. 1, 53–64.
260. Leticia Duboc, David S. Rosenblum, and Tony Wicks, *A framework for modelling and analysis of software systems scalability*, ICSE '06: Proceeding of the 28th International Conference on Software Engineering (New York, NY, USA), ACM Press, 2006, pp. 949–952.
261. Cees W. Duin, *Steiner's problem in graphs: reduction, approximation, variation*, Ph.D. thesis, Universiteit van Amsterdam, 1994.
262. Cees W. Duin and Anton Volgenant, *Reduction tests for the Steiner problem in graphs*, Networks **19** (1989), 549–567.
263. Cees W. Duin and Stefan Voss, *Efficient path and vertex exchange in Steiner tree algorithms*, Networks **29** (1997), 89–105.
264. Joe W. Duran and John J. Wiorkowski, *Quantifying software validity by sampling*, IEEE Transactions on Reliability **R-29** (1980), 141–144.
265. *The* ECL^{i}PS^e *constraint programming system*, http://eclipse.crosscoreop.com/, 2007.
266. *ECMA-334 C# language specification*, www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf, 2006.
267. Stefan Edelkamp, Shahid Jabbar, and Stefan Schrödl, *External A**, Proc. KI 2004 (Susanne Biundo, Thom W. Frühwirth, and Günther Palm, eds.), LNCS, vol. 3238, Springer, Heidelberg, 2004, pp. 226–240.
268. Herbert Edelsbrunner and Ernst P. Mücke, *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms*, Proceedings of the 4th Annual ACM Symposium on Computational Geometry, 1988, pp. 118–133.
269. Jack Edmonds, *Paths, trees, and flowers*, Canadian J. Math. **17** (1965), 449–467.

270. Niklas Eén and Niklas Sörensson, *An extensible SAT-solver*, Proc. 6th Theory and Applications of Satisfiability Testing (SAT '03) (Enrico Giunchiglia and Armando Tacchella, eds.), LNCS, vol. 2919, Springer, Heidelberg, 2003, pp. 502–518.
271. Wolfgang A. Eiden, *Präzise Unschärfe – Informationsmodellierung durch Fuzzy-Mengen*, ibidem-Verlag, 2002.
272. ———, *Scheduling with fuzzy methods*, Operations Research Proceedings 2004 (H. Fleuren, D. den Hertog, and P. Kort, eds.), Operations Research Proceedings, vol. 2004, Springer-Verlag, 2005, pp. 377–384.
273. Ioannis Z. Emiris and John F. Canny, *A general approach to removing degeneracies*, SIAM J. Comput. **24** (1995), no. 3, 650–664.
274. David Eppstein, *Quasiconvex analysis of backtracking algorithms*, Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2004, pp. 788–797.
275. Carl Erikson, *Hierarchical levels of detail to accelerate the rendering of large static and dynamic polygonal environments*, Ph.D. thesis, University of North Carolina, 2000.
276. *EXACUS: Efficient and exact algorithms for curves and surfaces*, `http://www.mpi-inf.mpg.de/projects/EXACUS/`, 2006, Version 1.0.
277. *Exploratory data analysis*, `http://www.itl.nist.gov/div898/handbook/eda/eda.htm`, 2006.
278. Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr, *On the design of CGAL, a computational geometry algorithms library*, Software Practice and Experience **30** (2000), no. 11, 1167–1202.
279. Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci, *Translating submachine locality into locality of reference*, Proc. 18th Intl. Parallel and Distributed Processing Symp. (IPDPS'04), CD-ROM, IEEE Computer Society, 2004.
280. Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan, *On the sorting complexity of suffix tree construction*, Journal of the ACM **47** (2000), 987–1011.
281. Ricardo Farias and Claudio T. Silva, *Out-of-core rendering of large, unstructured grids*, IEEE Computer Graphics and Applications **21** (2001), no. 4, 42–50.
282. Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan, *Sequoia: Programming the memory hierarchy*, Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006.
283. Panagiota Fatourou, Paul Spirakis, Panagiotis Zarafidis, and Anna Zoura, *Implementation and experimental evaluation of graph connectivity algorithms using LEDA*, WAE: International Workshop on Algorithm Engineering (Jeffrey Scott Vitter and Christos D. Zaroliagis, eds.), LNCS, vol. 1668, Springer, Heidelberg, 1999, pp. 124–138.
284. Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang, *On graph problems in a semi-streaming model*, Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP) (Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, eds.), LNCS, vol. 3142, Springer, Heidelberg, 2004, pp. 531–543.
285. Paolo Ferragina and Roberto Grossi, *The string B-tree: A new data structure for string search in external memory and its applications*, Journal of the ACM **46** (1999), 236–280.
286. Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young, *Competitive paging algorithms*, J. Algorithms **12** (1991), no. 4, 685–699.

287. Amos Fiat and Gerhard J. Woeginger (eds.), *Online algorithms: The state of the art*, Springer, 1998.
288. Rudolf Fleischer, Bernard M. E. Moret, and Erik Meineche Schmidt (eds.), *Experimental algorithmics: From algorithm design to robust and efficient software*, LNCS, no. 2547, Springer, Heidelberg, 2002.
289. http://blog.flickr.net/en/2008/11/03/3-billion/, 2008.
290. Christodoulos A. Floudas and Panos M. Pardalos, *A collection of test problems for constrained global optimization problems*, LNCS, vol. 455, Springer, Heidelberg, 1990.
291. Pasquale Foggia, *The VFLib graph matching library, version 2.0*, http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib.html, March 2001.
292. Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch, *Measure and conquer: Domination – a case study*, Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP '05) (Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, eds.), LNCS, vol. 3580, Springer, Heidelberg, 2005, pp. 191–203.
293. Lester Randolph Ford and Delbert Ray Fulkerson, *Flows in networks*, Princeton University Press, Princeton, NJ, 1963.
294. Steven Fortune, *A sweepline algorithm for Voronoi diagrams*, Proceedings of the 2nd Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1986, pp. 313–322.
295. ———, *Stable maintenance of point set triangulations in two dimensions*, Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 1989, pp. 494–499.
296. ———, *Polyhedral modelling with exact arithmetic*, SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications (New York, NY, USA), ACM, 1995, pp. 225–234.
297. ———, *Introduction*, Algorithmica **27** (2000), no. 1, 1–4.
298. Steven Fortune and Christopher J. van Wyk, *Efficient exact arithmetic for computational geometry*, Proceedings of the 9th Annual ACM Symposium on Computational Geometry, 1993, pp. 163–172.
299. ———, *Static analysis yields efficient exact integer arithmetic for computational geometry*, ACM Transactions on Graphics **15** (1996), no. 3, 223–248.
300. Steven Fortune and James Wyllie, *Parallelism in random access machines*, Proceedings of the 10th ACM Symposium on Theory of Computing (STOC), 1978, pp. 114–118.
301. Ulrich Fößmeier and Michael Kaufmann, *On exact solutions for the rectilinear Steiner tree problem*, Tech. Report WSI-96-09, Universität Tübingen, 1996.
302. ———, *On exact solutions for the rectilinear Steiner tree problem Part I: Theoretical results*, Algorithmica **26** (2000), 68–99.
303. Ian T. Foster and Adriana Iamnitchi, *On death, taxes, and the convergence of peer-to-peer and grid computing*, Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03) (M. Frans Kaashoek and Ion Stoica, eds.), LNCS, vol. 2735, Springer, Heidelberg, 2003, pp. 118–128.
304. Robert Fourer, David M. Gay, and Brian W. Kernighan, *AMPL: A modeling language for mathematical programming*, Brooks/Cole Publishing Company, 2002.
305. Geoffrey Fox, Roy Williams, and Paul Messina, *Parallel computing works!*, Morgan Kaufmann, 1994.
306. Leonor Frias, Jordi Petit, and Salvador Roura, *Lists revisited: Cache-conscious STL lists*, WEA (Carme Àlvarez and Maria J. Serna, eds.), LNCS, vol. 4007, Springer, Heidelberg, 2006, pp. 121–133.

307. Jerome H. Friedman, Jon L. Bentley, and Raphael A. Finkel, *An algorithm for finding best matches in logarithmic expected time*, ACM Transactions on Mathematical Software **3** (1977), no. 3, 209–226.
308. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran, *Cache-oblivious algorithms*, Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 1999, pp. 285–298.
309. Bernhard Fuchs, Walter Kern, Daniel Mölle, Stefan Richter, Peter Rossmanith, and Xinhui Wang, *Dynamic programming for minimum Steiner trees*, Theory Comput. Syst. **41** (2007), no. 3, 493–500.
310. Bernhard Fuchs, Walter Kern, and Xinhui Wang, *The number of tree stars is* $O^*(1.357^n)$, Algorithmica **49** (2007), 232–244.
311. Stefan Funke, Christian Klein, Kurt Mehlhorn, and Susanne Schmitt, *Controlled perturbation for Delaunay triangulations*, Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2005, pp. 1047–1056.
312. Zvi Galil, Silvio Micali, and Harold N. Gabow, *An* $O(EV \log V)$ *algorithm for finding a maximal weighted matching in general graphs*, SIAM J. Comput. **15** (1986), no. 1, 120–130.
313. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.
314. Joseph L. Ganley and James P. Cohoon, *Optimal rectilinear Steiner minimal trees in* $O(n^2 2.62^n)$ *time*, Proc. 6th Canad. Conf. on Computational Geometry, 1994, pp. 308–313.
315. Emden R. Gansner and Stephen C. North, *An open graph visualization system and its applications to software engineering*, Software — Practice and Experience **30** (2000), no. 11, 1203–1233.
316. Michael R. Garey, Ronald L. Graham, and David S. Johnson, *The complexity of computing Steiner minimal trees*, SIAM Journal on Applied Mathematics **32** (1977), 835–859.
317. Michael R. Garey and Donald S. Johnson, *The rectilinear Steiner tree problem is NP-complete*, SIAM Journal on Applied Mathematics **32** (1977), 826–834.
318. Bernd Gärtner, Martin Henk, and Günter M. Ziegler, *Randomized simplex algorithms on Klee-Minty cubes*, Combinatorica **18** (1998), no. 3, 349–372.
319. Saul I. Gass and Thomas L. Saaty, *The computational algorithm for the parametric objective function*, Naval Research Logistics Quarterly **2** (1955), 39.
320. Marina Gavrilova, *Weighted Voronoi diagrams in biology*, http://pages.cpsc.ucalgary.ca/~marina/vpplants/, 2007.
321. *Website of the GNU GCC project*, http://gcc.gnu.org/, 2006.
322. Assefaw H. Gebremedhin, Isabelle Guerin Lassous, Jens Gustedt, and Jan A. Telle, *PRO: A model for parallel resource-optimal computation*, Proc. 16th Int. Symp. High Performance Computing Systems and Applications (HPCS), IEEE Computer Society, 2002, pp. 106–113.
323. Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling, *Contraction hierarchies: Faster and simpler hierarchical routing in road networks*, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08) (Catherine C. McGeoch, ed.), LNCS, vol. 5038, Springer, Heidelberg, June 2008, pp. 319–333.
324. Jutta Geldermann and Heinrich Rommelfanger, *Fuzzy Sets, Neuronale Netze und Künstliche Intelligenz in der industriellen Produktion*, VDI-Verlag, Düsseldorf, 2003.

325. Ian P. Gent, Stuart A. Grant, Ewen MacIntyre, Patrick Prosser, Paul Shaw, Barbara M. Smith, and Toby Walsh, *How not to do it*, Tech. Report 97.27, School of Computer Studies, University of Leeds, May 1997.
326. Ian P. Gent, Christopher Jefferson, and Ian Miguel, *MINION: A fast, scalable, constraint solver*, Proc. 17th European Conference on Artificial Intelligence (ECAI '06) (Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, eds.), Frontiers in Artificial Intelligence and Applications, vol. 141, IOS Press, 2006, pp. 98–102.
327. Ian P. Gent and Toby Walsh, *CSPLIB: A benchmark library for constraints*, Tech. Report APES-09-1999, Department of Computer Science, University of Strathclyde, Glasgow, 1999.
328. ______, *CSPLIB: A benchmark library for constraints*, Principles and Practice of Constraint Programming - CP'99 (Joxan Jaffar, ed.), LNCS, vol. 1713, Springer, Heidelberg, 1999, pp. 480–481.
329. Alexandros V. Gerbessiotis and Leslie G. Valiant, *Direct bulk-synchronous parallel algorithms*, J. Parallel Distrib. Comput. **22** (1994), no. 2, 251–267.
330. A. J. Geurts, *A contribution to the theory of condition*, Numerische Mathematik **39** (1982), 85–96.
331. Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, *Fundamentals of software engineering*, Prentice Hall, New Jersey, 1991.
332. Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran, *Can a shared-memory model serve as a bridging model for parallel computation?*, Theory Comput. Syst. **32** (1999), no. 3, 327–359.
333. Robert Giegerich and Stefan Kurtz, *From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction*, Algorithmica **19** (1997), no. 3, 331–353.
334. *GMP: GNU Multiple Precision Arithmetic Library*, http://www.swox.com/gmp/, 2006, Version 4.2.1.
335. Stefan Goedecker and Adolfy Hoisie, *Performance optimization of numerically intensive codes*, Society for Industrial and Applied Mathematics, 2001.
336. Michael X. Goemans and David P. Williamson, *A general approximation technique for constrained forest problems*, SIAM Journal on Computing **24** (1995), no. 2, 296–317.
337. Simon Gog, *Broadword computing and Fibonacci code speed up compressed suffix arrays*, SEA '09: Proceedings of the 8th International Symposium on Experimental Algorithms (Jan Vahrenhold, ed.), LNCS, vol. 5526, Springer, Heidelberg, 2009, pp. 161–172.
338. Andrew V. Goldberg and Chris Harrelson, *Computing the shortest path: A* search meets graph theory*, Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2005, pp. 156–165.
339. Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck, *Reach for A*: Efficient point-to-point shortest path algorithms*, Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06), SIAM, 2006, pp. 129–143.
340. ______, *Better landmarks within reach*, Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07) (Camil Demetrescu, ed.), LNCS, vol. 4525, Springer, Heidelberg, June 2007, pp. 38–51.
341. Andrew V. Goldberg and Bernard M. E. Moret, *Combinatorial algorithms test sets [CATS]: The ACM/EATCS platform for experimental research*, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, 1999, pp. 913–914.

342. Andrew V. Goldberg and Renato F. Werneck, *Computing point-to-point shortest paths from external memory*, Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05), SIAM, 2005, pp. 26–40.
343. David Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys **23** (1991), no. 1, 5–48.
344. Andrei Goldchleger, Alfredo Goldman, Ulisses Hayashida, and Fabio Kon, *The implementation of the BSP parallel computing model on the integrade grid middleware*, MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing (New York, NY, USA), ACM, 2005, pp. 1–6.
345. Bruce L. Golden and William R. Stewart, *The traveling salesman problem – a guided tour of combinatorial optimization*, ch. Empirical analysis of heuristics, pp. 207–249, John Wiley & Sons, 1985.
346. Herman H. Goldstine and John von Neumann, *Numerical inverting of matrices of high order II*, Proc. Amer. Math. Soc. **2** (1951), 188–202, Reprinted in [774, pp. 558–572].
347. Michael T. Goodrich, Mark Handy, Benoît Hudson, and Roberto Tamassia, *Accessing the internal organization of data structures in the JDSL library*, Algorithm Engineering and Experimentation, International Workshop ALENEX '99 (Michael T. Goodrich and Catherine C. McGeoch, eds.), LNCS, vol. 1619, Springer, Heidelberg, 1999, pp. 124–139.
348. Michael T. Goodrich and Roberto Tamassia, *Algorithm design: Foundations, analysis, and internet examples*, Wiley, September 2001.
349. Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey S. Vitter, *External-memory computational geometry*, Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 1993, pp. 714–723.
350. Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha, *A memory model for scientific algorithms on graphics processors*, Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA, 2006, p. 89.
351. Susan Graham, Peter Kessler, and Marshall McKusick, *An execution profiler for modular programs*, Software - Practice and Experience **13** (1993), 671–685.
352. Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to parallel computing*, Pearson Education, 2003.
353. Ananth Grama, Vipin Kumar, Sanjay Ranka, and Vineet Singh, *Architecture independent analysis of parallel programs*, Proc. Intl. Conf. on Computational Science (ICCS'01) - Part II (London, UK), Springer-Verlag, 2001, pp. 599–608.
354. Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier, *Automated generation of search tree algorithms for hard graph modification problems*, Algorithmica **39** (2004), no. 4, 321–347.
355. Torbjörn Granlund, *GMP: The GNU multiple precision arithmetic library*, Free Software Foundation, Boston, MA, 2006.
356. *Graphviz: Graph visualization software*, http://www.graphviz.org/, 2007, version 2.16.
357. Harvey J. Greenberg, *Computational testing: Why, how and how much*, ORSA Journal on Computing **2** (1990), no. 1, 94–97.
358. Daniel H. Greene, *Integer line segment intersection*, unpublished manuscript.
359. Daniel H. Greene and Frances F. Yao, *Finite-resolution computational geometry*, Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 1986, pp. 143–152.

360. Douglas Gregor and Andrew Lumsdaine, *The parallel BGL: A generic library for distributed graph computations*, Tech. report, Open Systems Laboratory, Indiana University, 2005.
361. Martin Grötschel, Alexander Martin, and Robert Weismantel, *The Steiner tree packing problem in VLSI design*, Mathematical Programming **78** (1997), no. 2, 265–281.
362. Penny Grubb and Armstrong A. Takang, *Software maintenance: concepts and practice*, 2. ed., World Scientific, 2003.
363. *GSL: GNU scientific library*, http://www.gnu.org/software/gsl/, 2006, Version 1.8.
364. Leonidas J. Guibas, David Salesin, and Jorge Stolfi, *Constructing strongly convex approximate hulls with inaccurate primitives*, SIGAL '90: Proceedings of the International Symposium on Algorithms (London, UK), Springer-Verlag, 1990, pp. 261–270.
365. Thorsten Gunkel, Matthias Müller–Hannemann, and Mathias Schnee, *Improved search for night train connections*, Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07) (Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, eds.), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007, pp. 243–258.
366. Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele (eds.), *Abstract state machines, theory and applications, international workshop, ASM 2000*, LNCS, vol. 1912, Springer, Heidelberg, 2000.
367. Dan Gusfield, *Algorithms on strings, trees, and sequences*, University of Cambridge Press, 1997.
368. David B. Gustavson, *The many dimensions of scalability*, COMPCON, 1994, pp. 60–63.
369. Fred G. Gustavson, *Recursion leads to automatic variable blocking for dense linear-algebra algorithms*, IBM J. of Research and Development **41** (1999), no. 6, 737–756.
370. Jens Gustedt, *External memory algorithms using a coarse grained paradigm*, Tech. Report 5142, INRIA Lorraine / LORIA, France, March 2004.
371. Jens Gustedt, Stéphane Vialle, and Amelia De Vivo, *The parXXL environment: Scalable fine grained development for large coarse grained platforms*, Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA 2006 (Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, eds.), LNCS, vol. 4699, Springer, Heidelberg, 2006, pp. 1094–1104.
372. Ronald J. Gutman, *Reach-based routing: A new approach to shortest path algorithms optimized for road networks*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), SIAM, 2004, pp. 100–111.
373. Carsten Gutwenger and Petra Mutzel, *A linear time implementation of SPQR-trees*, GD '00: Proceedings of the 8th International Symposium on Graph Drawing (Joe Marks, ed.), LNCS, vol. 1984, Springer, Heidelberg, 2001, pp. 77–90.
374. Thomas Haigh, *Oral history: An interview with Joseph F. Traub*, http://history.siam.org/oralhistories/traub.htm, March 2004.
375. Nicholas G. Hall and Marc E. Posner, *Generating experimental data for computational testing with machine scheduling applications*, Operations Research **49** (2001), no. 7, 854–865.
376. Dan Halperin and Eran Leiserowitz, *Controlled perturbation for arrangements of circles*, Proceedings of the 19th Annual ACM Symposium on Computational Geometry, 2003, pp. 264–273.

377. Dan Halperin and Eli Packer, *Iterated snap rounding*, Comput. Geom. Theory Appl. **23** (2002), 209–225.
378. Dan Halperin and Christian R. Shelton, *A perturbation scheme for spherical arrangements with application to molecular modeling*, Proceedings of the 13th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1997, pp. 183–192.
379. Susanne E. Hambrusch, *Models for parallel computation*, ICPP Workshop, 1996, pp. 92–95.
380. Maurice Hanan, *On Steiner's problem with rectilinear distance*, SIAM Journal on Applied Mathematics **14** (1966), 255–265.
381. P. Hansen, *Bricriteria path problems*, Multiple Criteria Decision Making – Theory and Application – (Günter Fandel and T. Gal, eds.), Springer, 1979, pp. 109–127.
382. Peter E. Hart, Nils Nilsson, and Bertram Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics **4** (1968), 100–107.
383. Refael Hassin, *Approximation schemes for the restricted shortest path problem*, Mathematics of Operations Research **17** (1992), no. 1, 36–42.
384. Christian Heitmann, *Beurteilung der Bestandsfestigkeit von Unternehmen mit Neuro-Fuzzy*, Peter Lang, Frankfurt am Main, 2002.
385. Martin Held, *VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments*, Comput. Geom. Theory Appl. **18** (2001), no. 2, 95–123.
386. Bruce Hendrickson and Robert Leland, *A multilevel algorithm for partitioning graphs*, Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) (New York, NY, USA), ACM Press, 1995, p. 28.
387. Monika R. Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan, *Computing on data streams*, External Memory algorithms, DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 50, 1999, pp. 107–118.
388. Michael A. Heroux, Padma Raghavan, and Horst D. Simon, *Parallel processing for scientific computing (software, environments and tools)*, SIAM, Philadelphia, PA, USA, 2006.
389. Susan Hert, Lutz Kettner, Tobias Polzin, and Guido Schäfer, ExpLab - *a tool set for computational experiments*, `http://explab.sourceforge.net`, 2003.
390. David Hilbert, *Über die stetige Abbildung einer Linie auf ein Flächenstück*, Math. Annalen **38** (1891), 459–460.
391. Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao, *Distributed object location in a dynamic network*, SPAA '02: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (New York, NY, USA), ACM Press, 2002, pp. 41–52.
392. Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling, *Fast point-to-point shortest path computations with arc-flags*, Shortest Path Computations: Ninth DIMACS Challenge (Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, eds.), DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 41–72.
393. Jonathan Hill, William McColl, Dan Stefanescu, Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling, *BSPlib: the BSP programming library*, Parallel Computing **24** (1998), 1947–1980.
394. Mark D. Hill, *What is scalability?*, SIGARCH Computer Architecture News **18** (1990), no. 4, 18–21.
395. Mark D. Hill and Alan J. Smith, *Evaluating associativity in CPU caches*, IEEE Trans. Comput. **38** (1989), no. 12, 1612–1630.

396. Benjamin Hiller, Sven Oliver Krumke, and Jörg Rambau, *Reoptimization gaps versus model errors in online-dispatching of service units for ADAC*, Discrete Appl. Math. **154** (2006), no. 13, 1897–1907.
397. Charles A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580.
398. John D. Hobby, *Practical segment intersection with finite precision output*, Comput. Geom. Theory Appl. **13** (1999), no. 4, 199–214.
399. Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert, *A pilot study to compare programming effort for two parallel programming models*, Journal of Systems and Software **81** (2008), no. 11, 1920–1930.
400. Karla L. Hoffman and Richard H. F. Jackson, *In pursuit of a methodology for testing mathematical programming software*, Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981 (John M. Mulvey, ed.), Lecture Notes in Economics and Mathematical Systems, vol. 199, Springer, 1982, pp. 177–199.
401. Christoph M. Hoffmann, *Robustness in geometric computations*, Journal of Computing and Information Science in Engineering **2** (2001), 143 – 155.
402. Christoph M. Hoffmann, John E. Hopcroft, and Michael S. Karasick, *Towards implementing robust geometric computations*, Proceedings of the 4th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1988, pp. 106–117.
403. Robert C. Holte, *Very simple classification rules perform well on most commonly used datasets*, Machine Learning **11** (1993), 63–91.
404. Martin Holzer, Frank Schulz, and Dorothea Wagner, *Engineering multi-level overlay graphs for shortest-path queries*, Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06), SIAM, 2006.
405. ———, *Engineering multi-level overlay graphs for shortest-path queries*, ACM Journal of Experimental Algorithmics **13** (2008), 2.5:1–2.5:26.
406. Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm, *Combining speed-up techniques for shortest-path computations*, ACM Journal of Experimental Algorithmics **10** (2005), 2.5.
407. Martin Holzer, Frank Schulz, and Thomas Willhalm, *Combining speed-up techniques for shortest-path computations*, Proceedings of the 3rd Workshop on Experimental Algorithms (WEA'04) (Celso C. Ribeiro and Simone L. Martins, eds.), LNCS, vol. 3059, Springer, Heidelberg, 2004, pp. 269–284.
408. John N. Hooker, *Needed: An empirical science of algorithms*, Operations Research **42** (1994), no. 2, 201–212.
409. ———, *Testing heuristics: We have it all wrong*, Journal of Heuristics **1** (1995), no. 1, 33–42.
410. Holger H. Hoos and Thomas Stützle, *SATLIB: An online resource for research on SAT*, SAT 2000, Highlights of Satisfiability Research in the Year 2000 (Ian Gent, Hans van Maaren, and Toby Walsh, eds.), Frontiers in Artificial Intelligence and Applications, vol. 63, IOS Press, 2000, pp. 283–292.
411. John E. Hopcroft and Peter J. Kahn, *A paradigm for robust geometric algorithms*, Algorithmica **7** (1992), no. 4, 339–380.
412. John E. Hopcroft and Robert E. Tarjan, *Efficient planarity testing*, Journal of the ACM **21** (1974), 549–568.
413. Qiming Hou, Kun Zhou, and Baining Guo, *BSGP: Bulk-synchronous GPU programming*, ACM Trans. Graph. **27** (2008), no. 3, 1–12.

414. Stefan Hougardy and Hans-Jürgen Prömel, *A 1.598 approximation algorithm for the Steiner problem in graphs*, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 448–453.
415. Jiang Hu, Charles J. Alpert, Stephen T. Quay, and Gopal Gandham, *Buffer insertion with adaptive blockage avoidance*, ISPD '02: Proceedings of the 2002 international symposium on Physical design (New York, NY, USA), ACM Press, 2002, pp. 92–97.
416. Scott Huddleston and Kurt Mehlhorn, *A new data structure for representing sorted lists*, Acta Informatica (1982), 157–184.
417. Falk Hüffner, *Algorithm engineering for optimal graph bipartization*, Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA '05) (Sotiris E. Nikoletseas, ed.), LNCS, vol. 3503, Springer, Heidelberg, 2005, pp. 240–252.
418. Frank K. Hwang, *On Steiner minimal trees with rectilinear distance*, SIAM Journal on Applied Mathematics **30** (1976), 104–114.
419. Oscar H. Ibarra and Chul E. Kim, *Fast approximation algorithms for the knapsack and sum of subset problems*, Journal of the ACM **22** (1975), no. 4, 463–468.
420. *IEEE standard for binary floating-point arithmetic, ANSI/IEEE standard 754-1985*, Institute of Electrical and Electronics Engineers, New York, 1985, Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.
421. Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, and Kunihiko Mitoh, *A fast algorithm for finding better routes by AI search techniques*, Proceedings of the Vehicle Navigation and Information Systems Conference (VNSI'94), ACM Press, 1994, pp. 291–296.
422. *ILOG CPLEX: High-performance software for mathematical programming and optimization*, `http://www.ilog.com/products/cplex/`, 2009.
423. *ILOG CPLEX 11.2 reference manuals*, 2009, Information available at `www.cplex.com`.
424. *ILOG solver*, `http://www.ilog.com/products/solver/`, 2009.
425. Toshiyuki Imai, *A topology oriented algorithm for the Voronoi diagram of polygons*, Proceedings of the 8th Canadian Conference on Computational Geometry, Carleton University Press, 1996, pp. 107–112.
426. *Yahoo claims record with petabyte database*, `http://www.informationweek.com/news/software/database/showArticle.jhtml?articleID=207801436`, 2008.
427. *Intel threading building blocks website*, `http://osstbb.intel.com/`.
428. *Netezza promises petabyte-scale data warehouse appliances*, `http://www.intelligententerprise.com/showArticle.jhtml?articleID=205600559`, 2008.
429. *ISO/IEC 14882:2003 programming languages – C++*, 2003.
430. Richard H. F. Jackson, Paul T. Boggs, Stephen G. Nash, and Susan Powell, *Guidelines for reporting results of computational experiments. Report of the ad hoc committee*, Mathematical Programming **49** (1991), 413–425.
431. Lars Jacobsen and Kim S. Larsen, *Complexity of layered binary search trees with relaxed balance*, Proceedings of the 7th Italian Conference on Theoretical Computer Science (ICTCS), 2001, pp. 269–284.
432. Joseph JaJa, *An introduction to parallel algorithms*, Addison-Wesley, 1992.
433. Vojtěch Jarník and Miloš Kössler, *O minimálních grafech osahujících n daných bodu*, Ĉas. Pêstování Mat. **63** (1934), 223–235.
434. Jean-Marc Jazequel and Bertrand Meyer, *Design by contract: The lessons of Ariane*, Computer **30** (1997), no. 1, 129–130.
435. John R. Jensen, *Remote sensing of the environment: An earth resource perspective*, Prentice Hall, 2007.

436. John E. Beasley, *An SST-based algorithm for the Steiner problem in graphs*, Networks **19** (1989), 1–16.
437. David S. Johnson, *A theoretician's guide to the experimental analysis of algorithms*, Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges (M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, eds.), DIMACS Monographs, vol. 59, 2002, pp. 215–250.
438. David S. Johnson and Catherine C. McGeoch (eds.), *Network flows and matching: First DIMACS implementation challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 12, AMS, Providence, RI, 1993.
439. David S. Johnson and Lyle McGeoch, *Experimental analysis of heuristics for the STSP*, The Traveling Salesman Problem and its Variations (Gutin and Punnen, eds.), Kluwer Academic Publishing, Dordrecht, 2002, pp. 369–443.
440. David S. Johnson and Lyle A. McGeoch, *The traveling salesman problem: A case study in local optimization*, Local Search in Combinatorial Optimization (Emile H.L. Aarts and Jan Karel Lenstra, eds.), John Wiley and Sons, 1997.
441. Stephen Johnson, *Lint, a C program checker*, Unix Programmer's Manual, AT&T Bell Laboratories, 1978.
442. James A. Jones, Mary Jean Harrold, and John Stakso, *Visualization of test information to assist fault localization*, ICSE 2002 International Conference on Software Engineering, 2002, pp. 467–477.
443. Mihaljo Jovanovich, Fred Annexstein, and Kenneth Berman, *Scalability issues in large peer-to-peer networks - a case study of Gnutella*, Tech. report, ECECS Department, University of Cincinnati, 2001.
444. Ben H. H. Juurlink and Harry A. G. Wijshoff, *A quantitative comparison of parallel computation models*, ACM Trans. Comput. Syst. **16** (1998), no. 3, 271–318.
445. M. Frans Kaashoek and David R. Karger, *Koorde: A simple degree-optimal distributed hash table*, Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03) (M. Frans Kaashoek and Ion Stoica, eds.), LNCS, vol. 2735, Springer, Heidelberg, 2003.
446. Andrew B. Kahng, Ion I. Măndoiu, and Alexander Z. Zelikovsky, *Highly scalable algorithms for rectilinear and octilinear Steiner trees*, Proceedings 2003 Asia and South Pacific Design Automation Conference (ASP-DAC), 2003, pp. 827–833.
447. ———, *Approximation algorithms and metaheuristics*, ch. Practical Approximations of Steiner Trees in Uniform Orientation Metrics, Chapman & Hall/CRC, 2007.
448. Andrew B. Kahng and Gabriel Robins, *A new class of iterative Steiner tree heuristics with good performances*, IEEE Trans. Computer-Aided Design **11** (1992), 1462–1465.
449. Gil Kalai, *A subexponential randomized simplex algorithm*, Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC), 1992, pp. 475–482.
450. Gil Kalai and Daniel J. Kleitman, *A quasi-polynomial bound for the diameter of graphs of polyhedra*, Bulletin Amer. Math. Soc. **26** (1992), 315.
451. Kanela Kaligosi and Peter Sanders, *How branch mispredictions affect quicksort*, 14th Annual European Symposium on Algorithms (ESA) (Yossi Azar and Thomas Erlebach, eds.), LNCS, vol. 4186, Springer, Heidelberg, 2006, pp. 780–791.
452. John J. Kanet, Sanjay L. Ahire, and Michael F. Gorman, *Handbook of scheduling: Algorithms, models, and performance analysis*, ch. Constraint Programming for Scheduling, pp. 47–1–47–21, Chapman & Hall / CRC, 2004.

453. Kothuri V.R. Kanth and Ambuj Singh, *Optimal dynamic range searching in non-replicating index structures*, International Conference on Database Theory ICDT (1999), 257–276.
454. Craig S. Kaplan, *Voronoi diagrams and ornamental design*, Proceedings of the First Annual Symposium of the International Society for the Arts, Mathematics, and Architecture (ISAMA 1999, San Sebastián, Spain, 71111 June 1999), 1999, pp. 277–283.
455. Haim Kaplan and Nira Shafrir, *The greedy algorithm for shortest superstrings*, Information Processing Letters **93** (2005), no. 1, 13–17.
456. George Karakostas, *A better approximation ratio for the vertex cover problem*, Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP '05) (Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, eds.), LNCS, vol. 3580, Springer, Heidelberg, 2005, pp. 1043–1050.
457. Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee-Keng Yap, *A core library for robust numeric and geometric computation*, Proceedings of the 15th Annual ACM Symposium on Computational Geometry, 1999, pp. 351–359.
458. Michael Karasick, Derek Lieber, and Lee R. Nackman, *Efficient Delaunay triangulation using rational arithmetic*, ACM Trans. Graph. **10** (1991), no. 1, 71–91.
459. Menelaos I. Karavelas, *A robust and efficient implementation for the segment Voronoi diagram*, International Symposium on Voronoi Diagrams in Science and Engineering (VD2004), 2004, pp. 51–62.
460. David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*, ACM Symposium on Theory of Computing, May 1997, pp. 654–663.
461. Anna R. Karlin, Steven J. Phillips, and Prabhakar Raghavan, *Markov paging*, Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS), 1992, pp. 208–217.
462. Björn Karlsson, *Beyond the C++ standard library: An introduction to Boost*, Addison-Wesley, 2005.
463. Narendra Karmarkar, *A new polynomial-time algorithm for linear programming*, Combinatorica **4** (1984), no. 4, 373–396.
464. Richard M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (R.E. Miller and J.W. Thatcher, eds.), Plenum Press, New York, 1972, pp. 85–104.
465. Richard M. Karp and Vijaya Ramachandran, *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, Elsevier, 1990, pp. 869–942.
466. Marek Karpinski and Alexander Zelikovsky, *New approximation algorithms for the Steiner tree problem*, Journal of Combinatorial Optimization **1** (1997), 47–65.
467. George Karypis, *METIS - family of multilevel partitioning algorithms*, 2007.
468. George Karypis and Vipin Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput. **20** (1998), no. 1, 359–392.
469. Jonathan A. Kelner and Daniel A. Spielman, *A randomized polynomial-time simplex algorithm for linear programming*, Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC), 2006, pp. 51–60.
470. Lutz Kettner, *Reference counting in library design — optionally and with union-find optimization*, Library-Centric Software Design (LCSD'05) (San Diego, CA, USA) (Andrew Lumsdaine and Sibylle Schupp, eds.), Department of Computer Science, Texas A&M University, October 2005, pp. 1–10.

471. Lutz Kettner, Kurt Mehlhorn, Silvain Pion, Stefan Schirra, and Chee-Keng Yap, *Classroom examples of robustness problems in geometric computations*, Proceedings of the 12th Annual European Symposium on Algorithms (ESA 2004) (Susanne Albers and Tomasz Radzik, eds.), LNCS, vol. 2321, Springer, Heidelberg, 2004, pp. 702–713.
472. Leonid G. Khachiyan, *A polynomial algorithm in linear programming*, Dokl. Akad. Nauk SSSR **244** (1979), 1093–1096.
473. Khronos Group, *OpenCL*, `http://www.khronos.org/opencl/`, 2009.
474. Christian Klein, *Controlled perturbation for Voronoi diagrams*, Master's thesis, Universität des Saarlandes, April 2004.
475. Jon Kleinberg and Eva Tardos, *Algorithm design*, Pearson Education, 2006.
476. Darwin Klingman, H. Albert Napier, and Joel Stutz, *NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems*, Management Science **20** (1974), no. 5, 814–821.
477. Donald E. Knuth, *The art of computer programming, Volume 2: Seminumerical algorithms*, 1st ed., Addison-Wesley Professional, 1969.
478. ———, *Structured programming with go to statements*, ACM Computing Surveys **6** (1974), 261–301.
479. ———, *The WEB system of structured documentation*, Stanford Computer Science Report CS980, September 1983.
480. ———, *Literate programming*, The Computer Journal **27** (1984), no. 2, 97–111.
481. ———, *The Stanford graphbase: A platform for combinatorial computing*, ACM Press, 1993.
482. ———, *The art of computer programming, Volume 3: Sorting and searching*, 2nd ed., Addison-Wesley Professional, 1998.
483. ———, *The art of computer programming, Volume 4, Fascile 1: Bitwise tricks and techniques; binary decision diagrams*, Addison-Wesley Professional, 2009.
484. Donald E. Knuth and Silvio Levy, *The CWEB system of structured documentation, version 3.0*, Addison-Wesley, Reading, MA, USA, 1993.
485. Johannes Köbler, Uwe Schöning, and Jacobo Toran, *The graph isomorphism problem: Its structural complexity*, Birkhäuser, 1993.
486. Thorsten Koch, *ZIMPL user guide*, ZIB Report 00-20, Zuse Institute Berlin, 2001, Current version available at `http://zimpl.zib.de/download/zimpl.pdf`.
487. ———, *Rapid mathematical programming*, Ph.D. thesis, Technische Universität Berlin, 2004, ZIB-Report 04-58.
488. ———, *ZIMPL*, `http://zimpl.zib.de/`, 2008.
489. Thorsten Koch and Alexander Martin, *Steinlib*, ftp://ftp.zib.de/pub/Packages/mp-testdata/steinlib/index.html, 1997.
490. ———, *Solving Steiner tree problems in graphs to optimality*, Networks **33** (1998), 207–232.
491. Thorsten Koch, Alexander Martin, and Stefan Voß, *SteinLib: An updated library on Steiner tree problems in graphs*, Tech. Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin, 2000.
492. Werner Koch et al., *The GNU privacy guard, version 1.4.5*, Source code available at `http://gnupg.org/`, 2006.
493. Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling, *Acceleration of shortest path and constrained shortest path computation*, Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05) (Sotiris E. Nikoletseas, ed.), LNCS, Springer, Heidelberg, 2005, pp. 126–138.
494. Erricos John Kontoghiorghes (ed.), *Handbook of parallel computing and statistics*, Chapman & Hall/CRC, 2005.

495. Jeffrey Kotula, *Source code documentation: An engineering deliverable*, TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00) (Washington, DC, USA), IEEE Computer Society, 2000, p. 505.
496. L. Kou, George Markowsky, and Leonard Berman, *A fast algorithm for Steiner trees*, Acta Inform. **15** (1981), 141–145.
497. Markus Kowarschik and Christian Weiß, *An overview of cache optimization techniques and cache-aware numerical algorithms*, Algorithms for Memory Hierarchies (Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, eds.), LNCS, vol. 2625, Springer, Heidelberg, 2002, pp. 213–232.
498. *KProf – profiling made easy*, http://kprof.sourceforge.net/, 2002.
499. Balakrishnan Krishnamurthy, *Constructing test cases for partitioning heuristics*, IEEE Transactions on Computers **36** (1987), no. 9, 1112–1114.
500. Sven O. Krumke and Hartmut Noltemeier, *Graphentheorische Konzepte und Algorithmen*, B. G. Teubner, 2005.
501. Dietmar Kühl, Marco Nissen, and Karsten Weihe, *Efficient, adaptable implementations of graph algorithms*, Proceedings of the 1st Workshop on Algorithm Engineering (WAE '97), 1997, http://www.dsi.unive.it/~wae97/proceedings/.
502. Thomas S. Kuhn, *The structure of scientific revolutions*, The University of Chicago Press, 1970.
503. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to parallel computing: Design and analysis of algorithms*, Benjamin-Cummings Publishing, 1994.
504. Anthony LaMarca and Richard E. Ladner, *The influence of caches on the performance of sorting*, Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1997, pp. 370–379.
505. ______, *The influence of caching on the performance of sorting*, Journal of Algorithms **31** (1999), 66–104.
506. David Lane, Joan Lu, Camille Peres, and Emily Zitek, *Online statistics: An interactive multimedia course of study*, http://onlinestatbook.com/index.html, 2006.
507. *LAPACK: Linear Algebra PACKage*, http://www.netlib.org/lapack/, 2007, Version 3.1.1.
508. Bruno Latour, *Science in action*, Havard University Press, 1987.
509. Luigi Laura, Stefano Leonardi, Stefano Millozzi, Ulrich Meyer, and Jop F. Sibeyn, *Algorithms and experiments for the webgraph*, European Symposium on Algorithms (ESA) (Giuseppe Di Battista and Uri Zwick, eds.), LNCS, vol. 2832, Springer, Heidelberg, 2003, pp. 703–714.
510. Ulrich Lauther, *An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background*, Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung, vol. 22, IfGI prints, 2004, pp. 219–230.
511. Pierre L'Ecuyer, *Simulation of algorithms for performance analysis*, INFORMS Journal on Computing **8** (1996), no. 1, 16–20.
512. *The LEDA user manual*, http://www.algorithmic-solutions.info/leda_manual/, 2009.
513. LEDA, *Library for efficient data types and algorithms*, http://www.algorithmic-solutions.com/, 2009, Version 6.2.1.
514. Frank T. Leighton, *Introduction to parallel algorithms and architectures: Arrays - trees - hypercubes*, Morgan Kaufmann, 1992.
515. Ernst L. Leiss, *A programmer's companion to algorithm analysis*, Chapman & Hall/CRC, 2006.

516. Thomas Lengauer, *Combinatorial algorithms for integrated circuit layout*, Wiley, Chichester, 1990.
517. Christian Lennerz and Sven Thiel, *Handling of parameterized data types in LEDA*, Tech. report, Algorithmic Solutions GmbH, 1997.
518. David Lester and Paul Gowland, *Using PVS to validate the algorithms of an exact arithmetic*, Theoretical Computer Science **291** (2003), 203–218.
519. Leonid A. Levin, *Universal sequential search problems*, Problems of Information Transmission **9** (1973), no. 3, 265–266.
520. Anany Levitin, *Introduction to the design and analysis of algorithms*, Pearson Education, 2003.
521. Bil Lewis, *Debugging backward in time*, Proceedings of the 5. International Workshop on Automated and Algorithmic Debugging AADEBUG (http://www.lambdacs.com/debugger/debugger.html), 2003.
522. Harry R. Lewis and Christos H. Papadimitriou, *Elements of the theory of computation*, Prentice-Hall, 1981.
523. Chen Li and Chee-Keng Yap, *A new constructive root bound for algebraic expressions*, Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001, pp. 496–505.
524. ———, *Recent progress in exact geometric computation*, in S. Basu and L. Gonzalez-Vega, editors, Proc. DIMACS Workshop on Algorithmic and Quantitative Aspects of Real Algebraic Geometry in Mathematics and Computer Science, March 12 - 16, 2001., 2001.
525. Jian Liang, Rakesh Kumar, and Keith W. Ross, *Understanding KaZaA*, http://citeseer.ist.psu.edu/liang04understanding.html, 2004.
526. Gideon Lidor, *Construction of nonlinear programming test problems with known solution characteristics*, Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981 (John M. Mulvey, ed.), Lecture Notes in Economics and Mathematical Systems, vol. 199, Springer, 1982, pp. 35–43.
527. Thomas Lindner, *Train schedule optimization in public rail transport*, Ph.D. thesis, Technische Universität Braunschweig, Germany, 2000.
528. Richard J. Lipton and Robert E. Tarjan, *A separator theorem for planar graphs*, SIAM journal applied mathematics **36** (1979), 177–189.
529. Barbara Liskov and John Guttag, *Abstraction and specification in program development*, MIT Press, Cambridge, MA, USA, 1986.
530. *Literate programming*, http://www.literateprogramming.com, 2009.
531. Marco E. Lübbecke and Jacques Desrosiers, *Selected topics in column generation*, Operations Research **53** (2005), no. 6, 1007–1023.
532. Bin Ma, *Why greed works for shortest common superstring problem*, Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM '08) (Paolo Ferragina and Gad M. Landau, eds.), LNCS, vol. 5029, Springer, Heidelberg, 2008, pp. 244–254.
533. Bruce M. Maggs, Lesley R. Matheson, and Robert E. Tarjan, *Models of parallel computation: A survey and synthesis*, Proceedings of the 28th Hawaii International Conference on System Sciences, January 1995, pp. 61–70.
534. Anil Maheshwari and Norbert Zeh, *I/O-efficient algorithms for graphs of bounded treewidth*, Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM-SIAM, 2001, pp. 89–90.
535. ———, *I/O-optimal algorithms for planar graphs using separators*, Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM-SIAM, 2002, pp. 372–381.

536. Andrew Makhorin, *GNU linear programming kit reference manual version 4.11*, Dept. Applied Informatics, Moscow Aviation Institute, 2006.
537. Dahlia Malkhi, Moni Naor, and Davod Ratajczak, *Viceroy: A scalable and dynamic emulation of the butterfly*, Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 2002, pp. 183–192.
538. Ernesto Queiros Martins, *On a multicriteria shortest path problem*, European Journal of Operational Research **26** (1984), no. 3, 236–245.
539. Yossi Matias, *Parallel algorithms column: On the search for suitable models*, ACM SIGACT News **28** (1997), no. 3, 21–29.
540. Jirí Matoušěk, János Pach, Micha Sharir, Shmuel Sifrony, and Emo Welzl, *Fat triangles determine linearly many holes*, SIAM Journal on Computing **23** (1994), no. 1, 154–169.
541. Jirí Matoušěk, Micha Sharir, and Emo Welzl, *A subexponential bound for linear programming*, Algorithmica **16** (1996), no. 4/5, 498–516.
542. Bruce McCarl, *McCarl GAMS user guide*, http://www.gams.com/dd/docs/bigdocs/gams2002/mccarlgamsuserguide.pdf, 2008.
543. Wiliam F. McColl, *Scalable computing*, Computer Science Today: Recent Trends and Developments (J. van Leeuwen, ed.), vol. 1000, Springer-Verlag, 1995, pp. 46–61.
544. William F. McColl and Alexandre Tiskin, *Memory-efficient matrix multiplication in the BSP model*, Algorithmica **24** (1999), no. 3-4, 287–297.
545. Catherine C. McGeoch, *Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups*, ACM Computing Surveys **24** (1992), no. 2, 195–212.
546. ______, *Challenges in algorithm simulation*, INFORMS Journal on Computing **8** (1996), no. 1, 27–28.
547. ______, *Toward an experimental method for algorithm simulation*, INFORMS Journal on Computing **8** (1996), no. 1, 1–15.
548. ______, *Experimental analysis of algorithms*, Notices of the AMS **48** (2001), no. 3, 304–311.
549. ______, *Experimental algorithmics*, Communications of the ACM **50** (2007), no. 11, 27–31.
550. Catherine C. McGeoch and Bernard M. E. Moret, *How to present a paper on experimental work with algorithms*, SIGACT News **30** (1999), no. 4, 85–90.
551. Catherine C. McGeoch, Peter Sanders, Rudolf Fleischer, Paul R. Cohen, and Doina Precup, *Using finite experiments to study asymptotic performance*, in Fleischer et al. [288], pp. 93–126.
552. Nimrod Megiddo, *Improved asymptotic analysis of the average number of steps performed by the self-dual simplex algorithm*, Mathematical Programming **35** (1986), no. 2, 140–172.
553. Miriam Mehl, Tobias Weinzierl, and Christoph Zenger, *A cache-oblivious self-adaptive full multigrid method*, Numer. Linear Algebra Appl. **13** (2006), no. 2–3, 275–291.
554. Kurt Mehlhorn, *A faster approximation algorithm for the Steiner problem in graphs*, Information Processing Letters **27** (1988), 125–128.
555. Kurt Mehlhorn and Ulrich Meyer, *External-memory breadth-first search with sublinear I/O*, Proc. 10th Ann. European Symposium on Algorithms (ESA) (Rolf H. Möhring and Rajeev Raman, eds.), LNCS, vol. 2461, Springer, Heidelberg, 2002, pp. 723–735.

556. Kurt Mehlhorn, Rolf H. Möhring, Burkhard Monien, Petra Mutzel, Peter Sanders, and Dorothea Wagner, *Antrag auf ein Schwerpunktprogramm zum Thema Algorithm Engineering*, http://www.algorithm-engineering.de/beschreibung.pdf, 2006.

557. Kurt Mehlhorn and Petra Mutzel, *On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm*, Algorithmica **16** (1996), no. 2, 233–242.

558. Kurt Mehlhorn and Stefan Näher, *Algorithm design and software libraries: Recent developments in the LEDA project*, Algorithms, Software, Architectures, Information Processing — Proc. IFIP Congress, vol. 1, Elsevier Science, 1992, pp. 493–505.

559. ______, *LEDA: A platform for combinatorial and geometric computing*, CACM: Communications of the ACM **38** (1995), 96–102.

560. ______, *From algorithms to working programs: On the use of program checking in LEDA*, Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98) (Lubos Brim, Jozef Gruska, and Jirí Zlatuska, eds.), LNCS, vol. 1450, Springer, Heidelberg, 1998, pp. 84–93.

561. ______, *LEDA: A platform for combinatorial and geometric computing*, Cambridge University Press, Cambridge, November 1999.

562. Kurt Mehlhorn and Peter Sanders, *Algorithms and data structures - the basic toolbox*, Springer, 2008.

563. Kurt Mehlhorn and Guido Schäfer, *Implementation of $O(nm \log n)$ weighted matchings in general graphs: The power of data structures*, ACM Journal of Experimental Algorithms **7** (2002), no. 4, 1–19.

564. Zdzislaw Alexander Melzak, *On the problem of Steiner*, Canad. Math. Bull. **4** (1961), 143–148.

565. Alberto O. Mendelzon and Peter T. Wood, *Finding regular simple paths in graph databases*, SIAM Journal on Computing **24** (1995), no. 6, 1235–1258.

566. Bertrand Meyer, *Design by contract*, Tech. Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

567. ______, *Applying 'design by contract'*, Computer **25** (1992), no. 10, 40–51.

568. ______, *Object-oriented software construction*, second ed., Prentice Hall PTR, March 2000.

569. Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn (eds.), *Algorithms for memory hierarchies*, LNCS, vol. 2625, Springer, Heidelberg, 2003.

570. Scott Meyers, *More effective C++*, Addison-Wesley, 1996.

571. ______, *Effective C++*, 3rd ed., Addison-Wesley, 2005.

572. Victor J. Milenkovic, *Verifiable implementation of geometric algorithms using finite precision arithmetic*, Artif. Intell. **37** (1988), no. 1-3, 377–401.

573. ______, *Shortest path geometric rounding*, Algorithmica **27** (2000), no. 1, 57–86.

574. Gary L. Miller, *Riemann's hypothesis and tests for primality*, Journal of Computer and System Sciences **13** (1976), 300–317.

575. Tsuyoshi Minakawa and Kokichi Sugihara, *Topology oriented vs. exact arithmetic - experience in implementing the three-dimensional convex hull algorithm*, ISAAC (Hon Wai Leong, Hiroshi Imai, and Sanjay Jain, eds.), LNCS, vol. 1350, Springer, Heidelberg, 1997, pp. 273–282.

576. Bhubaneswar Mishra, *Algorithmic algebra*, Texts and Monographs in Computer Science, Springer, 1993.

577. Joseph S. B. Mitchell, *Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems*, SIAM Journal on Computing **28** (1999), no. 4, 1298–1309.

578. ______, *A PTAS for TSP with neighborhoods among fat regions in the plane*, Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007, pp. 11–18.
579. Rolf H. Möhring, *Verteilte Verbindungssuche im öffentlichen Personenverkehr – Graphentheoretische Modelle und Algorithmen*, Angewandte Mathematik, insbesondere Informatik – Beispiele erfolgreicher Wege zwischen Mathematik und Informatik (Patrick Horster, ed.), Vieweg, 1999, pp. 192–220.
580. Rolf H. Möhring and Matthias Müller-Hannemann, *Complexity and modeling aspects of mesh refinement into quadrilaterals*, Algorithmica **26** (2000), 148–171.
581. Rolf H. Möhring, Matthias Müller-Hannemann, and Karsten Weihe, *Mesh refinement via bidirected flows: Modeling, complexity, and computational results*, Journal of the ACM **44** (1997), 395–426.
582. Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm, *Partitioning graphs to speed up Dijkstra's algorithm*, Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05) (Sotiris E. Nikoletseas, ed.), LNCS, Springer, Heidelberg, 2005, pp. 189–202.
583. Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel, *Analysis of multi-dimensional space-filling curves*, Geoinformatica **7** (2003), no. 3, 179–209.
584. Bernard M. E. Moret, *Towards a discipline of experimental algorithmics*, Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges (Michael H. Goldwasser, David S. Johnson, and Catherine C. McGeoch, eds.), DIMACS Monographs, vol. 59, American Mathematical Society, 2002, pp. 197–213.
585. Bernard M. E. Moret and Henry D. Shapiro, *Algorithms and experiments: The new (and old) methodology*, Journal of Universal Computer Science **7** (2001), no. 5, 434–446.
586. Bernhard M.E. Moret and Henry D. Shapiro, *An empirical assessment of algorithms for constructing a minimal spanning tree*, Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 15, 1994, pp. 99–117.
587. Pat Morin, *Coarse grained parallel computing on heterogeneous systems*, Proc. 1998 ACM Symp. on Applied Computing (SAC'98), ACM Press, 1998, pp. 628–634.
588. Donald R. Morrison, *PATRICIA: Practical algorithm to retrieve information coded in alphanumeric*, Journal of the ACM **15** (1968), 514–534.
589. Rajeev Motwani and Pabhakar Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.
590. David M. Mount, *ANN programming manual*, `http://www.cs.umd.edu/~mount/ANN`, 2006.
591. MPFI, *Multiple precision floating-point interval library*, `http://gforge.inria.fr/projects/mpfi/`, 2006, Version 1.3.4-RC3.
592. MPFR, *A multiple precision floating-point library*, `http://www.mpfr.org/`, 2005, Version 2.2.0.
593. *MTL: The matrix template library*, `http://www.osl.iu.edu/research/mtl/`, 2005, Version 2.1.2-22.
594. Matthias Müller–Hannemann and Mathias Schnee, *Finding all attractive train connections by multi-criteria Pareto search*, Algorithmic Methods for Railway Optimization (Frank Geraets, Leo G. Kroon, Anita Schöbel, Dorothea Wagner, and Christos D. Zaroliagis, eds.), LNCS, vol. 4359, Springer, Heidelberg, 2007, pp. 246–263.

595. Matthias Müller–Hannemann and Karsten Weihe, *Pareto shortest paths is often feasible in practice*, Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01) (Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, eds.), LNCS, vol. 2141, Springer, Heidelberg, 2001, pp. 185–197.
596. Matthias Müller-Hannemann, *High quality quadrilateral surface meshing without template restrictions: A new approach based on network flow techniques*, International Journal of Computational Geometry and Applications **10** (2000), 285–307.
597. Matthias Müller-Hannemann and Sven Peyer, *Approximation of rectilinear Steiner trees with length restrictions on obstacles*, 8th Workshop on Algorithms and Data Structures (WADS 2003) (Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Michiel H. M. Smid, eds.), LNCS, vol. 2748, Springer, Heidelberg, 2003, pp. 207–218.
598. Matthias Müller-Hannemann and Alexander Schwartz, *Implementing weighted b-matching algorithms: Insights from a computational study*, ACM Journal of Experimental Algorithms **5** (2000), 8.
599. Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica **7** (1987), no. 1, 105–113.
600. Kameshwar Munagala and Abhiram Ranade, *I/O-complexity of graph algorithms*, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1999, pp. 687–694.
601. Bruce A. Murtagh, *Advanced linear programming*, McGraw-Hill, 1981.
602. David R. Musser, *Introspective sorting and selection algorithms*, Software: Practice and Experience **27** (1997), no. 8, 983–993.
603. S. Muthu Muthukrishnan, *Data streams: Algorithms and applications*, Foundations and Trends in Theoretical Computer Science, vol. 1 (2), NOW, 2005.
604. Ion I. Măndoiu, Vijay V. Vazirani, and Joseph L. Ganley, *A new heuristic for rectilinear Steiner trees*, ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design (Piscataway, NJ, USA), IEEE Press, 1999, pp. 157–162.
605. Nathan C. Myers, *Traits: A new and useful template technique*, C++ Report **7** (1995), no. 5, 32–35.
606. Stefan Näher, *Delaunay triangulation and other computational geometry experiments*, http://www.informatik.uni-trier.de/~naeher/Professur/research/index.html, 2003.
607. Stefan Näher and Oliver Zlotowski, *Design and implementation of efficient data types for static graphs*, ESA 2002 (Rolf H. Möhring and Rajeev Raman, eds.), LNCS, vol. 2461, Springer, Heidelberg, 2002, pp. 748–759.
608. Aleksandar Nanevski, Guy Blelloch, and Robert Harper, *Automatic generation of staged geometric predicates*, Higher-Order and Symbolic Computation **16** (2003), 379–400.
609. Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes, *Bidirectional A* search for time-dependent fast paths*, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08) (Catherine C. McGeoch, ed.), LNCS, vol. 5038, Springer, Heidelberg, 2008, pp. 334–346.
610. Moni Naor and Udi Wieder, *Novel architectures for P2P applications: The continuous-discrete approach*, SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms, 2003, pp. 50–59.
611. George L. Nemhauser and Laurence A. Wolsey, *Integer and combinatorial optimization*, John Wiley & Sons, New York, NY, USA, 1988.

612. ______, *Integer programming*, Optimization (George L. Nemhauser et al., ed.), Elsevier North-Holland, Inc., New York, NY, USA, 1989, pp. 447–527.
613. Nicholas Nethercote and Julian Seward, *Valgrind: A program supervision framework*, Electronic Notes in Theoretical Computer Science **89** (2003), no. 2, 44–66.
614. Benne K. Nielsen, Pawel Winter, and Martin Zachariasen, *An exact algorithm for the uniformly-oriented Steiner tree problem*, ESA 2002 (Rolf H. Möhring and Rajeev Raman, eds.), LNCS, vol. 2461, Springer, 2002, pp. 760–772.
615. Mark H. Nodine, Michael T. Goodrich, and Jeffrey S. Vitter, *Blocking for external graph searching*, Algorithmica **16(2)** (1996), 181–214.
616. Mark H. Nodine and Jeffrey S. Vitter, *Deterministic distribution sort in shared and distributed memory multiprocessors*, Proceedings of the 5th annual ACM Symposium on Parallel Algorithms and Architectures (1993), 120–129.
617. NVIDIA Corporation, *CUDA zone – the resource for CUDA developers*, http://www.nvidia.com/cuda/, 2009.
618. *OGDF — open graph drawing framework*, http://www.ogdf.net, 2008.
619. Yasuaki Oishi and Kokichi Sugihara, *Topology-oriented divide-and-conquer algorithm for Voronoi diagrams*, CVGIP: Graphical Model and Image Processing **57** (1995), no. 4, 303–314.
620. Chris Okasaki, *Red-black trees in a functional setting*, Journal of Functional Programming **9** (1999), no. 4, 471–477.
621. Richard P. O'Neill, *A comparison of real-world linear programs and their randomly generated analogs*, Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981 (John M. Mulvey, ed.), Lecture Notes in Economics and Mathematical Systems, vol. 199, Springer, 1982, pp. 44–59.
622. James B. Orlin, *On experimental methods for algorithm simulation*, INFORMS Journal on Computing **8** (1996), no. 1, 21–23.
623. Mark H. Overmars and A. Frank van der Stappen, *Range searching and point location among fat objects*, Journal of Algorithms **21** (1996), no. 3, 629–656.
624. Sam Owre, Natarajan Shankar, and John M. Rushby, *PVS: A prototype verification system*, Proceedings of the 11th Conference on Automated Deduction (Deepak Kapur, ed.), LNCS, vol. 607, Springer, Heidelberg, 1992, pp. 748–752.
625. *The Oxford BSP toolset*, http://www.bsp-worldwide.org/implmnts/oxtool/, 1998, Version 1.4.
626. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, *The PageRank citation ranking: Bringing order to the web*, http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf, 1999.
627. Victor Y. Pan, Yanqiang Yu, and C. Stewart, *Algebraic and numerical techniques for the computation of matrix determinants*, Computers and Mathematics with Applications **34** (1997), no. 1, 43–70.
628. Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial optimization*, Dover Publications, Inc., 1998.
629. David Lorge Parnas and Paul C. Clements, *A rational design process: How and why to fake it*, IEEE Trans. Softw. Eng. **12** (1986), no. 2, 251–257.
630. Mike Paterson and F. Frances Yao, *Efficient binary space partitions for hidden-surface removal and solid modeling*, Discrete & Computational Geometry **5** (1990), 485–503.
631. David A. Patterson and John L. Hennessy, *Computer organization and design. The hardware/software interface*, 3rd ed., Morgan Kaufmann Publishers Inc., 2005.

632. Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig, *Real PRAM programming*, Euro-Par '02: Proc. 8th International Euro-Par Conference on Parallel Processing, Springer-Verlag, 2002, pp. 522–531.
633. *PBGL: The parallel boost graph library*, http://www.osl.iu.edu/research/pbgl/, 2009, version 0.7.0.
634. *Space-filling curve*, http://en.wikipedia.org/wiki/Space-filling_curve (last visited 15.2.2009).
635. Giuseppe Peano, *Sur une courbe qui remplit toute une aire plane*, Math. Annalen **36** (1890), 157–160.
636. Andrea Pietracaprina, Geppino Pucci, and Francesco Silvestri, *Cache-oblivious simulation of parallel programs*, Proc. 8th Workshop on Advances in Parallel and Distributed Computational Models (CD), IEEE Computer Society, 2006.
637. Sylvain Pion and Chee-Keng Yap, *Constructive root bound for k-ary rational input numbers*, Proceedings of the 19th ACM Symposium on Computational Geometry, ACM Press, 2003, San Diego, California, pp. 256–263.
638. David Pisinger, *Algorithms for knapsack problems*, Ph.D. thesis, University of Copenhagen, Dept. of Computer Science, 1995.
639. Maurizio Pizzonia and Giuseppe Di Battista, *Object-oriented design of graph oriented data structures*, ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation (Michael T. Goodrich and Catherine C. McGeoch, eds.), LNCS, vol. 1619, Springer, Heidelberg, 1999, pp. 140–155.
640. P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser, *The C++ Standard Template Library*, Prentice-Hall, 2000.
641. Ira Pohl, *Bi-directional search*, Proceedings of the Sixth Annual Machine Intelligence Workshop (Bernard Meltzer and Donald Michie, eds.), vol. 6, Edinburgh University Press, 1971, pp. 124–140.
642. Tobias Polzin and Siavash V. Daneshmand, *Primal-dual approaches to the Steiner problem*, Approximation Algorithms for Combinatorial Optimization (APPROX 2000) (K. Jansen and S. Khuller, eds.), LNCS, vol. 1913, Springer, Heidelberg, 2000, pp. 214–225.
643. ______, *A comparison of Steiner relaxations*, Discrete Applied Mathematics **112** (2001), 241–261.
644. ______, *Improved algorithms for the Steiner problem in networks*, Discrete Applied Mathematics **112** (2001), 263–300.
645. ______, *Practical partitioning-based methods for the Steiner problem*, Experimental Algorithms: 5th International Workshop (WEA 2006) (Carme Àlvarez and Maria J. Serna, eds.), LNCS, vol. 4007, Springer, Heidelberg, 2006, pp. 241–252.
646. Vaughan R. Pratt, *Every prime has a succinct certificate*, SIAM Journal of Computing **4** (1975), 214–220.
647. Lutz Prechelt, *An empirical comparison of seven programming languages*, Computer **33** (2000), no. 10, 23–29.
648. Douglas M. Priest, *On properties of floating point arithmetics: Numerical stability and the cost of accurate computations*, Ph.D. thesis, University of California at Berkeley, 1992.
649. Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey S. Vitter, *Bkd-tree: A dynamic scalable kd-tree*, 8th International Symposium on advances in Spatial and Temporal Databases, SSTD (2003), 46–65.
650. Harald Prokop, *Cache-oblivious algorithms*, Master's thesis, Massachusetts Institute of Technology, 1999.

651. Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Experimental comparison of shortest path approaches for timetable information*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), SIAM, 2004, pp. 88–99.
652. ———, *Towards realistic modeling of time-table information through the time-dependent approach*, Proceedings of ATMOS Workshop 2003, 2004, pp. 85–103.
653. ———, *Efficient models for timetable information in public transportation systems*, ACM Journal of Experimental Algorithmics **12** (2007), Article 2.4.
654. Sigal Raab and Dan Halperin, *Controlled perturbation for arrangements of polyhedral surfaces*, `http://acg.cs.tau.ac.il/danhalperin/publications/dan-halperins-publications`, 2002.
655. Michael O. Rabin, *Mathematical theory of automata*, Proceedings of the 19th ACM Symposium in Applied Mathematics, 1966, pp. 153–175.
656. ———, *Probabilistic algorithm for testing primality*, Journal of Number Theory **12** (1980), 128–138.
657. Naila Rahman, Richard Cole, and Rajeev Raman, *Optimized predecessor data structures for internal memory*, Proc. 3rd Workshop on Algorithm Engineering (Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, eds.), LNCS, vol. 2141, Springer, Heidelberg, 2001, pp. 67–78.
658. Naila Rahman and Rajeev Raman, *Analysing the cache behaviour of non-uniform distribution sorting algorithm*, Algorithms - ESA 2000 (Mike Paterson, ed.), LNCS, vol. 1879, Springer, Heidelberg, 2000, pp. 380–391.
659. Sridhar Rajagopalan and Vijay V. Vazirani, *On the bidirected cut relaxation for the metric Steiner tree problem*, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1999, pp. 742–751.
660. Sanguthevar Rajasekaran and John Reif (eds.), *Handbook of parallel computing: Models, algorithms and applications*, Chapman & Hall CRC Computer & Information Science, CRC Press, 2007.
661. Vijaya Ramachandran, *Parallel algorithm design with coarse-grained synchronization*, International Conference on Computational Science (2), 2001, pp. 619–627.
662. Norman Ramsey, *Literate programming simplified*, IEEE Softw. **11** (1994), no. 5, 97–105.
663. Ronald L. Rardin and Benjamin W. Lin, *Test problems for computational experiments – issues and techniques*, Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981 (John M. Mulvey, ed.), Lecture Notes in Economics and Mathematical Systems, vol. 199, Springer, 1982, pp. 8–15.
664. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker, *A scalable content-addressable network*, Proceedings of the ACM SIGCOMM, ACM Press, August 2001, pp. 161–172.
665. Helmut Ratschek and Jon Rokne, *Exact computation of the sign of a finite sum*, Applied Mathematics and Computation **99** (1999), 99–127.
666. Eric S. Raymond, *The art of UNIX programming*, Pearson Education, 2003.
667. James Reinders, *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*, O'Reilly Media, Inc., 2007.
668. Gerhard Reinelt, *TSPLIB—a traveling salesman problem library*, ORSA Journal on Computing **3** (1991), no. 4, 376–384.
669. Gabriel Dos Reis, Bernard Mourrain, Fabrice Rouillier, and Philippe Trébuchet, *An environment for symbolic and numeric computation*, International Congress of Mathematical Software ICMS'2002, April 2002.

670. Craig W. Reynolds, *Flocks, herds, and schools: A distributed behavioral model*, Computer Graphics **21** (1987), no. 4, 25–34.
671. John R. Rice, *A theory of condition*, SIAM J. Num. Anal. **3** (1966), 287–310.
672. Daniel Richardson, *How to recognize zero*, Journal of Symbolic Computation **24** (1997), no. 6, 627–645.
673. Anthony P. Roberts and Ed J. Garboczi, *Elastic moduli of model random three-dimensional closed-cell cellular solids*, Acta Materialia **49** (2001), no. 2, 189–197.
674. Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas, *Efficiently four-coloring planar graphs*, Proceedings of the 28th ACM Symposium on Theory of Computing (STOC), ACM Press, 1996, pp. 571–575.
675. Neil Robertson and Paul D. Seymour, *Graph minors. XIII: The disjoint paths problem*, J. Comb. Theory Ser. B **63** (1995), no. 1, 65–110.
676. ———, *Graph minors. XX. Wagner's conjecture*, J. Comb. Theory Ser. B **92** (2004), no. 2, 325–357.
677. Gabriel Robins and Alexander Zelikovsky, *Improved Steiner tree approximation in graphs*, Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000, pp. 770–779.
678. ———, *Tighter bounds for graph Steiner tree approximation*, SIAM Journal on Discrete Mathematics **19** (2005), no. 1, 122–134.
679. Heiko Röglin and Berthold Vöcking, *Smoothed analysis of integer programming*, Proceedings of the 11th International Conference on Integer Programming and Combinatorial Optimization (IPCO) (Michael Jünger and Volker Kaibel, eds.), LNCS, vol. 3509, Springer, Heidelberg, 2005, pp. 276–290.
680. Francesca Rossi, *Constraint (logic) programming: A survey on research and applications*, Selected papers from the Joint ERCIM/Compulog Net Workshop on New Trends in Constraints (London, UK), Springer-Verlag, 2000, pp. 40–74.
681. Francesca Rossi, Charles Petrie, and Vasant Dhar, *On the equivalence of constraint satisfaction problems*, ECAI'90: Proceedings of the 9th European Conference on Artificial Intelligence (Stockholm) (Luigia Carlucci Aiello, ed.), Pitman, 1990, pp. 550–556.
682. Francesca Rossi, Peter van Beek, and Toby Walsh (eds.), *Handbook of constraint programming*, Elsevier, 2006.
683. Antony Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, Middleware 2001 (Rachid Guerraoui, ed.), LNCS, vol. 2218, Springer, Heidelberg, 2001, pp. 329–350.
684. Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi, *Accurate floating-point summation part I: Faithful rounding*, SIAM Journal on Scientific Computing **31** (2008), no. 1, 189–224.
685. ———, *Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest*, SIAM Journal on Scientific Computing **31** (2008), no. 2, 1269–1302.
686. Yousef Saad, *Iterative methods for sparse linear systems*, 2nd ed., Society for Industrial and Applied Mathematics, April 2003.
687. Hans Sagan, *Space-filling curves*, Springer-Verlag, 1994.
688. David Salesin, Jorge Stolfi, and Leonidas J. Guibas, *Epsilon geometry: building robust algorithms from imprecise computations*, Proceedings of the 5th Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1989, pp. 208–217.
689. Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto, *Sensitivity analysis in practice: A guide to assessing scientific models*, John Wiley & Sons, 2004.

690. Hanan Samet, *The quadtree and related hierarchical data structures*, ACM Computing Surveys **16** (1984), no. 2, 187–260.
691. Johannes Sametinger, *Software engineering with reusable components*, Springer, 1997.
692. Laura A. Sanchis, *On the complexity of test case generation for NP-hard problems*, Information Processing Letters **36** (1990), no. 3, 135–140.
693. Peter Sanders, *Presenting data from experiments in algorithmics*, in Fleischer et al. [288], pp. 181–196.
694. ______, *Algorithm engineering - an attempt at a definition*, Efficient Algorithms (Susanne Albers, Helmut Alt, and Stefan Näher, eds.), LNCS, vol. 5760, Springer, Heidelberg, 2009, pp. 321–340.
695. ______, *Algorithm engineering — an attempt at a definition using sorting as an example*, ALENEX10 (Philadelphia) (Guy Blelloch and Dan Halperin, eds.), SIAM, 2010, pp. 55–61.
696. Peter Sanders, Sebastian Egner, and Jan H. M. Korst, *Fast concurrent access to parallel disks*, Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000, pp. 849–858.
697. Peter Sanders and Dominik Schultes, *Highway hierarchies hasten exact shortest path queries*, Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05) (Gerth Stølting Brodal and Stefano Leonardi, eds.), LNCS, vol. 3669, Springer, Heidelberg, 2005, pp. 568–579.
698. ______, *Engineering highway hierarchies*, Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06) (Yossi Azar and Thomas Erlebach, eds.), LNCS, vol. 4168, Springer, Heidelberg, 2006, pp. 804–816.
699. Peter Sanders, Dominik Schultes, and Christian Vetter, *Mobile route planning*, Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08) (Dan Halperin and Kurt Mehlhorn, eds.), LNCS, vol. 5193, Springer, Heidelberg, September 2008, pp. 732–743.
700. John E. Savage, *Models of computation, exploring the power of computing*, Addison Wesley, 1998.
701. John E. Savage and Mohammad Zubair, *A unified model for multicore architectures*, Proceedings of the 1st International Forum on Next-Generation Multicore/-Manycore Technologies, IFMT 2008, Cairo, Egypt, 2008, p. 9.
702. Stefan Schamberger and Jens M. Wierum, *A locality preserving graph ordering approach for implicit partitioning: Graph-filling curves*, Proc. 17th Intl. Conf. on Parallel and Distributed Computing Systems, (PDCS'04), ISCA, 2004, pp. 51–57.
703. Stefan Schirra, *A case study on the cost of geometric computing*, Selected Papers from the International Workshop on Algorithm Engineering and Experimentation (ALENEX'99) (Michael T. Goodrich and Catherine C. McGeoch, eds.), LNCS, vol. 1619, Springer, Heidelberg, 1999, pp. 156–176.
704. ______, *Robustness and precision issues in geometric computation*, Handbook of Computational Geometry (Jörg Rüdiger Sack and Jorge Urrutia, eds.), Elsevier, Amsterdam, The Netherlands, January 2000, pp. 597–632.
705. ______, *Real numbers and robustness in computational geometry*, 6th Conference on Real Numbers and Computers, Schloss Dagstuhl, Germany, November 2004, Invited Lecture.
706. Walter Schmitting, *Das Traveling-Salesman-Problem - Anwendungen und heuristische Nutzung von Voronoi-/Delaunay-Strukturen zur Lösung euklidischer, zweidimensionaler Traveling-Salesman-Probleme*, Ph.D. thesis, Heinrich-Heine-Universität Düsseldorf, 1999.

707. Steve Schneider, *The B-method: An introduction*, Palgrave, 2002.
708. Peter Schorn, *Robust algorithms in a program library for geometric computation*, Ph.D. thesis, ETH: Swiss Federal Institute of Technology Zürich, 1991, Diss. ETH No. 9519.
709. ———, *An axiomatic approach to robust geometric programs*, J. Symb. Comput. **16** (1993), no. 2, 155–165.
710. Alexander Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, Inc., 1998.
711. Dominik Schultes, *Route planning in road networks*, Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008.
712. Dominik Schultes and Peter Sanders, *Dynamic highway-node routing*, Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07) (Camil Demetrescu, ed.), LNCS, vol. 4525, Springer, Heidelberg, 2007, pp. 66–79.
713. Frank Schulz, *Timetable information and shortest paths*, Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.
714. Frank Schulz, Dorothea Wagner, and Karsten Weihe, *Dijkstra's algorithm on-line: An empirical case study from public railroad transport*, Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99) (Jeffrey Scott Vitter and Christos D. Zaroliagis, eds.), LNCS, vol. 1668, Springer, Heidelberg, 1999, pp. 110–123.
715. ———, *Dijkstra's algorithm on-line: An empirical case study from public railroad transport*, ACM Journal of Experimental Algorithmics **5** (2000).
716. Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Using multi-level graphs for timetable information in railway systems*, Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02) (David M. Mount and Clifford Stein, eds.), LNCS, vol. 2409, Springer, Heidelberg, 2002, pp. 43–59.
717. Robert Sedgewick, *Implementing quicksort programs*, Communications of the ACM **21** (1978), no. 10, 847 – 857.
718. Michael Seel, *Eine Implementierung abstrakter Voronoidiagramme*, Master's thesis, Universität des Saarlandes, 1994.
719. Mark Segal, *Using tolerances to guarantee valid polyhedral modeling results*, SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM Press, 1990, pp. 105–114.
720. Mark Segal and Carlo H. Séquin, *Consistent calculations for solids modeling*, Proceedings of the 1st Annual ACM Symposium on Computational Geometry (New York, NY, USA), ACM Press, 1985, pp. 29–38.
721. Raimund Seidel, *Constrained Delaunay triangulations and Voronoi diagrams*, Report 260 IIG-TU Graz (1988), 178–191.
722. ———, *The nature and meaning of perturbations in geometric computing*, STACS '94: Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, eds.), LNCS, vol. 775, Springer, Heidelberg, 1994, pp. 3–17.
723. Sriram Sellappa and Siddhartha Chatterjee, *Cache-efficient multigrid algorithms*, Int. J. High Perform. Comput. Appl. **18** (2004), no. 1, 115–133.
724. Sandeep Sen and Siddhartha Chatterjee, *Towards a theory of cache-efficient algorithms*, Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000, pp. 829–838.
725. Saurabh Sethia, Martin Held, and Joseph S. B. Mitchell, *PVD: A stable implementation for computing Voronoi diagrams of polygonal pockets*, ALENEX '01:

Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation (Adam L. Buchsbaum and Jack Snoeyink, eds.), LNCS, vol. 2153, Springer, Heidelberg, 2001, pp. 105–116.

726. *Seti@home*, `http://setiathome.berkeley.edu`, 2006.

727. *Website of SGI's STL implementation*, `http://www.sgi.com/tech/stl/`, 2006.

728. Michael I. Shamos and Dan Hoey, *Closest-point problems*, Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 1975, pp. 151–162.

729. David J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*, CRC Press, 2007.

730. Jonathan Richard Shewchuk, *Companion web page to [731]*, `http://www.cs.cmu.edu/~quake/robust.html`.

731. ———, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete and Computational Geometry **18** (1997), 305–363.

732. Douglas R. Shier, *On algorithm analysis*, INFORMS Journal on Computing **8** (1996), no. 1, 24–26.

733. Jop F. Sibeyn, *From parallel to external list ranking*, 1997, Technical Report MPI-I-97-1-021, Max-Planck Institut für Informatik.

734. Jop F. Sibeyn and Michael Kaufmann, *BSP-like external memory computation*, Proc. 3rd Italian Conf. Algorithms and Complexity (Gian Carlo Bongiovanni, Daniel P. Bovet, and Giuseppe Di Battista, eds.), LNCS, vol. 1203, Springer, Heidelberg, 1997, pp. 229–240.

735. Sidney Siegel, *Nonparametric statistics for the behavioral sciences*, McGraw-Hill, 1956.

736. Jeremy G. Siek, L. Lee, and Andrew Lumsdaine, *The Boost graph library*, Addison-Wesley, 2002.

737. Jeremy G. Siek and Andrew Lumsdaine, *The matrix template library: A generic programming approach to high performance numerical linear algebra*, Proceedings of Computing in Object-Oriented Parallel Environments: Second International Symposium, ISCOPE 98 (Denis Caromel, R. R. Oldehoeft, and Marydell Tholburn, eds.), LNCS, vol. 1505, Springer, Heidelberg, 1998, pp. 59–70.

738. Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates, *Fast and flexible word searching on compressed text*, ACM Transactions on Information Systems **18** (2000), no. 2, 113–139.

739. Johannes Singler, *Graph isomorphism implementation in LEDA 5.1*, `http://www.algorithmic-solutions.de/bilder/graph_iso.pdf`, 2006, Version 2.0.

740. Johannes Singler, Peter Sanders, and Felix Putze, *MCSTL: The Multi-Core Standard Template Library*, Euro-Par 2007: Parallel Processing (Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, eds.), LNCS, vol. 4641, Springer, Heidelberg, 2007, pp. 682–694.

741. Steven Skiena, *Who is interested in algorithms and why? Lessons from the Stony Brook algorithms repository*, Algorithm Engineering (Kurt Mehlhorn, ed.), Max-Planck-Institut für Informatik, 1998, pp. 204–212.

742. Steven S. Skiena, *The algorithm design manual*, second ed., Springer Verlag, New York, 2008.

743. David B. Skillicorn and Domenico Talia, *Models and languages for parallel computation*, ACM Computing Surveys **30** (1998), no. 2, 123–169.

744. Daniel D. Sleator and Robert E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28** (1985), no. 2, 202–208.

745. Steve Smale, *On the average number of steps of the simplex method of linear programming*, Mathematical Programming **27** (1983), 241–262.

746. Warren D. Smith, *How to find Steiner minimal trees in Euclidean d-space*, Algorithmica **7** (1992), 137–177.
747. Marc Snir and Steve Otto, *MPI – the complete reference: The MPI core*, 2nd ed., MIT Press, 1998.
748. Ian Sommerville, *Software engineering*, 8th ed., International Computer Science Series, vol. -, Addison-Wesley, New York - Amsterdam - Bonn, 2006.
749. Daniel A. Spielman and Shang-Hua Teng, *Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time*, Journal of the ACM **51** (2004), no. 3, 385–463.
750. Paul G. Spirakis and Christos D. Zaroliagis, *Distributed algorithm engineering*, Experimental Algorithmics. From Algorithm Design to Robust and Efficient Software (Rudolf Fleischer, Bernard M. E. Moret, and Erik Meineche Schmidt, eds.), LNCS, vol. 2547, Springer, Heidelberg, 2002, pp. 197–228.
751. *Splint – secure programming lint*, http://splint.org/, 2007, Version 3.1.2.
752. Joel Spolsky, *User interface design for programmers*, Apress, Berkeley, CA, USA, 2001.
753. Peter Sprent and N. C. Smeeton, *Applied nonparametric statistical methods*, Chapman & Hall/CRC, 2001.
754. Richard M. Stallman et al., *GCC, the GNU compiler collection, version 4.3.3*, Source code available at http://gcc.gnu.org/, 2009.
755. Alexander A. Stepanov and Meng Lee, *The standard template library*, Tech. Report HPL-95-11, Hewlett Packard, November 1995.
756. *Website of STLport*, http://www.stlport.org/, 2006.
757. Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, Proceedings of the 2001 ACM SIGCOMM Conference, 2001, pp. 149–160.
758. Volker Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik **13** (1969), 354–356.
759. Bjarne Stroustrup, *The C++ programming language, special edition*, Addison-Wesley, 2000.
760. ———, *A brief look at C++0x*, http://www.artima.com/cppsource/cpp0x.html, January 2006.
761. *STXXL: Standard template library for extra large data sets*, http://stxxl.sourceforge.net/, 2008, version 1.2.1.
762. Kokichi Sugihara, *A robust and consistent algorithm for intersecting convex polyhedra*, Comput. Graph. Forum **13** (1994), no. 3, 45–54.
763. Kokichi Sugihara and Masao Iri, *A solid modelling system free from topological inconsistency*, J. Inf. Process. **12** (1989), no. 4, 380–393.
764. ———, *A robust topology-oriented incremental algorithm for Voronoi diagrams*, Int. J. Comput. Geometry Appl. **4** (1994), no. 2, 179–228.
765. Kokichi Sugihara, Masao Iri, Hiroshi Inagaki, and Toshiyuki Imai, *Topology-oriented implementation - an approach to robust geometric algorithms*, Algorithmica **27** (2000), no. 1, 5–20.
766. Wijnand J. Suijlen, *BSPonMPI*, http://bsponmpi.sourceforge.net/, 2006.
767. Xian-He Sun and Lionel M. Ni, *Another view on parallel speedup*, Supercomputing '90: Proc. ACM/IEEE Conf. on Supercomputing, IEEE Computer Society, 1990, pp. 324–333.
768. Geoff Sutcliffe and Christian B. Suttner, *The TPTP problem library - CNF release v1.2.1*, Journal of Automated Reasoning **21** (1998), no. 2, 177–203.
769. Z. Sweedyk, *A $2\frac{1}{2}$-approximation algorithm for shortest superstring*, SIAM Journal on Computing **29** (1999), no. 3, 954–986.

770. Roberto Tamassia and Luca Vismara, *A case study in algorithm engineering for geometric computing*, International Journal of Computational Geometry Applications **11** (2001), no. 1, 15–70.
771. Robert E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. ACM **22** (1975), no. 2, 215–225.
772. ______, *Updating a balanced search tree in $O(1)$ rotations*, Information Processing Letters **16** (1983), no. 5, 253–257.
773. ______, *Amortized computational complexity*, SIAM Journal on Algebraic and Discrete Methods **6** (1985), no. 2, 306–318.
774. A. H. Taub (ed.), *John von Neumann collected works*, vol. V, Design of Computers, Theory of Automata and Numerical Analysis, Pergamon, Oxford, 1963.
775. Siamak Tazari, Matthias Müller-Hannemann, and Karsten Weihe, *Workload balancing in multi-stage production processes*, Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings (Carme Àlvarez and Maria J. Serna, eds.), LNCS, vol. 4007, Springer, Heidelberg, 2006, pp. 49–60.
776. Siamak Tazari and Matthias Müller-Hannemann, *Dealing with large hidden constants: Engineering a planar Steiner tree PTAS*, ALENEX 2009, SIAM, Philadelphia, 2009, pp. 120–131.
777. Robert D. Tennent, *Specifying software*, Cambridge University Press, 2002.
778. The BlueGene/L Team, *An overview of the BlueGene/L supercomputer*, Proc. ACM/IEEE Conf. on Supercomputing, 2002, pp. 1–22.
779. Sven Thiel, *The LEDA memory manager*, Tech. report, Algorithmic Solutions GmbH, August 1998, http://www.algorithmic-solutions.info/leda_docs/leda_memmgr.ps.gz.
780. Mikkel Thorup, *Integer priority queues with decrease key in constant time and the single source shortest paths problem*, Journal of Computer and System Sciences **69** (2004), no. 3, 330–353.
781. Sivan Toledo, *A survey of out-of-core algorithms in numerical linear algebra*, External memory algorithms, American Mathematical Society, 1999, pp. 161–179.
782. *TPIE: A transparent parallel I/O environment*, http://www.cs.duke.edu/TPIE/, 2005, version from September 19, 2005.
783. Lloyd N. Trefethen and David Bau, III (eds.), *Numerical linear algebra*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. MR MR1444820 (98k:65002)
784. Edward R. Tufte, *The visual display of quantitative information*, Graphics Press, 1983.
785. John W. Tukey, *Exploratory data analysis*, Reading, MA, Addison-Wesley, 1977.
786. Alan M. Turing, *Rounding-off errors in matrix processes*, Quarterly Journal of Mechanics and Applied Mathematics **1** (1948), 287–308, reprinted in [787] with summary and notes (including corrections).
787. ______, *Pure mathematics*, Collected Works of A. M. Turing, North-Holland, Amsterdam, The Netherlands, 1992, Edited and with an introduction and postscript by J. L. Britton and Irvine John Good. With a preface by P. N. Furbank.
788. *The universal protein resource (UniProt)*, http://www.uniprot.org/, 2007.
789. Leslie G. Valiant, *A bridging model for parallel computation*, Commun. ACM **33** (1990), no. 8, 103–111.
790. ______, *General purpose parallel architectures*, Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), Elsevier, 1990, pp. 943–972.

791. ———, *A bridging model for multi-core computing*, Proceedings of 16th Annual European Symposium on Algorithms (ESA) (Dan Halperin and Kurt Mehlhorn, eds.), vol. 5193, Springer, 2008, pp. 13–28.
792. Gabriel Valiente, *Algorithms on trees and graphs*, Springer, 2002.
793. A. Frank van der Stappen, *Motion planning amidst fat obstacles*, Ph.D. thesis, Department of Computer Science, Utrecht University, March 1994.
794. Peter van Emde Boas, *Preserving order in a forest in less than logarithmic time and linear space*, Information Processing Letters **6** (1977), 80–82.
795. Peter van Emde Boas, R. Kaas, and E. Zijlstra, *Design and implementation of an efficient priority queue*, Mathematical Systems Theory **10** (1977), 99–127.
796. Dimitri van Heesch, *The Doxygen website*, http://www.stack.nl/~dimitri/doxygen/, 2009.
797. J. A. van Hulzen, Ben J. A. Hulshof, Barbara L. Gates, and M. C. van Heerwaarden, *A code optimization package for REDUCE*, Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, 1989, pp. 163–170.
798. Jan van Leeuwen (ed.), *Handbook of theoretical computer science, volume A: Algorithms and complexity*, Elsevier and MIT Press, 1990.
799. Marc A. van Leeuwen, *Literate programming in C: CWEBx manual*, Report AM-R9510, Centrum voor Wiskunde en Informatica, Department of Analysis, Algebra and Geometry, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1995.
800. David Vandervoorde and Nicolai M. Josuttis, *C++ templates: the complete guide*, Addison-Wesley, 2003.
801. Todd L. Veldhuizen, *Expression templates*, C++ Report **7** (1995), no. 5, 26–31.
802. ———, *Arrays in Blitz++*, Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98) (Denis Caromel, R. R. Oldehoeft, and Marydell Tholburn, eds.), LNCS, vol. 1505, Springer, Heidelberg, 1998, pp. 223–230.
803. Bill Venners, *Joshua Bloch: A conversation about design (An interview with effective Java author, Josh Bloch by Bill Venners)*, First Published in JavaWorld, http://www.javaworld.com/javaworld/jw-01-2002/jw-0104-bloch.html, January 2002.
804. Roman Vershynin, *Beyond Hirsch conjecture: walks on random polytopes and smoothed complexity of the simplex method*, Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2006, pp. 133–142.
805. Sebastiano Vigna, *Broadword implementation of rank/select queries*, WEA (Catherine C. McGeoch, ed.), LNCS, vol. 5038, Springer, Heidelberg, 2008, pp. 154–168.
806. Uzi Vishkin, George C. Caragea, and Bryant C. Lee, *Handbook of parallel computing: Models, algorithms and applications*, ch. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform, CRC Press, 2007.
807. *Visone: Analysis and visualization of social networks*, http://visone.info/, 2008, Version 2.3.5.
808. Jeffrey S. Vitter, *External memory algorithms and data structures: Dealing with massive data*, ACM Computing Surveys **33(2)** (2001), 209–271.
809. Jeffrey S. Vitter, *Algorithms and data structures for external memory*, Foundations and Trends in Theoretical Computer Science, NOW Publishers, 2008.
810. Jeffrey S. Vitter and Elizabeth A. M. Shriver, *Algorithms for parallel memory I: Two level memories*, Algorithmica **12** (1994), no. 2–3, 110–147.

811. ______, *Algorithms for parallel memory, I/II*, Algorithmica **12** (1994), no. 2/3, 110–169.
812. Jules Vleugels, *On fatness and fitness – realistic input models for geometric algorithms*, Ph.D. thesis, Department of Computer Science, Utrecht University, March 1997.
813. John von Neumann and Herman H. Goldstine, *Numerical inverting of matrices of high order*, Bull. Amer. Math. Soc. **53** (1947), 1021–1099, Reprinted in [774, pp. 479–557].
814. Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, 2nd ed., Cambridge University Press, 2003.
815. Georgy Voronoi, *Nouvelle applications des paramètres continus à la theorie des formes quadratiques*, J. Reine Angew. Math. **134** (1908), 198–287.
816. Dorothea Wagner and Thomas Willhalm, *Geometric speed-up techniques for finding shortest paths in large sparse graphs*, Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03) (Giuseppe Di Battista and Uri Zwick, eds.), LNCS, vol. 2832, Springer, Heidelberg, 2003, pp. 776–787.
817. Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis, *Dynamic shortest path containers*, Proceedings of ATMOS Workshop 2003, 2004, pp. 65–84.
818. ______, *Geometric containers for efficient shortest-path computation*, ACM Journal of Experimental Algorithmics **10** (2005), 1.3.
819. Peter J. L. Wallis (ed.), *Improving floating-point programming*, Wiley, London, 1990.
820. Jie Wang, *Average-case computational complexity theory*, Complexity Theory Retrospective, in Honor of Juris Hartmanis on the Occasion of His Sixtieth Birthday, July 5, 1988 (Alan L. Selman, ed.), vol. 2, Springer, 1997.
821. David M. Warme, *A new exact algorithm for rectilinear Steiner minimal trees*, Tech. report, System Simulation Solutions, Inc., Alexandria, VA 22314, USA, 1997.
822. ______, *Spanning trees in hypergraphs with applications to Steiner trees*, Ph.D. thesis, Computer Science Dept., The University of Virginia, 1998.
823. David M. Warme, Pawel Winter, and Martin Zachariasen, *Exact algorithms for plane Steiner tree problems: A computational study*, Tech. Report TR-98/11, DIKU, Department of Computer Science, Copenhagen, Denmark, 1998.
824. ______, *GeoSteiner 3.1*, DIKU, Department of Computer Science, Copenhagen, Denmark, http://www.diku.dk/geosteiner/, 2003.
825. Josef Weidendorfer, *Performance analysis of GUI applications on Linux*, KDE Contributor Conference, 2003.
826. Karsten Weihe, *A software engineering perspective on algorithmics*, ACM Computing Surveys **33** (2001), no. 1, 89–134.
827. Karsten Weihe, Ulrik Brandes, Annegret Liebers, Matthias Müller-Hannemann, Dorothea Wagner, and Thomas Willhalm, *Empirical design of geometric algorithms*, Proceedings of the 15th Annual ACM Symposium on Computational Geometry, 1999, pp. 86–94.
828. Maik Weinard and Georg Schnitger, *On the greedy superstring conjecture*, SIAM Journal on Discrete Mathematics **20** (2006), no. 2, 502–522.
829. Richard Clint Whaley, Antoine Petitet, and Jack J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing **27** (2001), no. 1–2, 3–35.
830. *Wikipedia*, `http://en.wikipedia.org/wiki/Wikipedia:Modelling_Wikipedia's_growth`, 2010.

831. James H. Wilkinson, *Rounding errors in algebraic processes*, IFIP Congress, 1959, pp. 44–53.
832. ______, *Error analysis of floating-point computation*, Numer. Math. **2** (1960), 319–340.
833. ______, *Rounding errors in algebraic processes*, Notes on Applied Science, No. 32, Notes on Applied Science No. 32, Her Majesty's Stationery Office, London, 1963, Also published by Prentice-Hall, Englewood Cliffs, NJ, USA, 1964, translated into Polish as *Bledy Zaokragleń w Procesach Algebraicznych* by PWW, Warsaw, Poland, 1967 and translated into German as *Rundungsfehler* by Springer-Verlag, Berlin, Germany, 1969. Reprinted by Dover Publications, New York, 1994.
834. James H. Wilkinson and Christian H. Reinsch (eds.), *Handbook for automatic computation, Vol. 2, Linear Algebra*, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1971.
835. Ross Williams, *FunnelWeb user's manual*, `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Adelaide, South Australia, Australia, 1992.
836. Tiffani L. Williams and Rebecca J. Parsons, *The heterogeneous bulk synchronous parallel model*, Proc. 15th Intl. Parallel and Distributed Processing Symp. (IPDPS'00), Workshops on Parallel and Distr. Processing, Springer-Verlag, 2000, pp. 102–108.
837. Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, *Dynamic storage allocation: A survey and critical review*, IWMM '95: Proceedings of the International Workshop on Memory Management (Henry G. Baker, ed.), LNCS, vol. 986, Springer, Heidelberg, 1995, pp. 1–116.
838. Pawel Winter, *An algorithm for the Steiner problem in the Euclidean plane*, Networks **15** (1985), 323–345.
839. Pawel Winter and Martin Zachariasen, *Euclidean Steiner minimum trees: An improved exact algorithm*, Networks **30** (1997), 149–166.
840. Laurence A. Wolsey, *Integer programming*, John Wiley & Sons, 1998.
841. Richard T. Wong, *A dual ascent approach for Steiner tree problems on a directed graph*, Mathematical Programming **28** (1984), 271–287.
842. Jim C. P. Woodcock and Jim Davies, *Using Z: Specification, proof and refinement*, Prentice Hall International Series in Computer Science, 1996.
843. Li Xiao, Xiaodong Zhang, and Stefan A. Kubricht, *Improving memory performance of sorting algorithms*, ACM Journal of Experimental Algorithmics **5(3)** (2000).
844. Chee-Keng Yap, *A geometric consistency theorem for a symbolic perturbation scheme*, J. Comput. Syst. Sci. **40** (1990), no. 1, 2–18.
845. ______, *Symbolic treatment of geometric degeneracies*, J. Symb. Comput. **10** (1990), no. 3-4, 349–370.
846. ______, *Towards exact geometric computation*, Comput. Geom. Theory Appl. **7** (1997), no. 1-2, 3–23.
847. ______, *Fundamental problems of algorithmic algebra*, Oxford University Press, 2000.
848. ______, *Robust geometric computation*, Handbook of Discrete and Computational Geometry (Jacob E. Goodman and Joseph O'Rourke, eds.), Chapmen & Hall/CRC, Boca Raton, FL, 2nd ed., 2004, pp. 927–952.
849. Chee-Keng Yap and Kurt Mehlhorn, *Towards robust geometric computation*, Fundamentals of Computer Science Study Conference (Washington DC), July 25-27 2001.

850. Tzuoo-Hawn Yeh, Cheng-Ming Kuo, Chin-Laung Lei, and Hsu-Chun Yen, *Competitive analysis of on-line disk scheduling*, Theory of Computing Systems **31** (1998), 491–506.
851. Jay Yellen and Jonathan L. Gross, *Graph theory and its applications*, CRC Press, 1998.
852. Mehmet C. Yildiz and Patrick H. Madden, *Preferred direction Steiner trees*, GLSVLSI '01: Proceedings of the 11th Great Lakes symposium on VLSI (New York, NY, USA), ACM Press, 2001, pp. 56–61.
853. Sung-Eui Yoon and Peter Lindstrom, *Mesh layouts for block-based caches*, IEEE Trans. Visualization and Computer Graphics **12** (2006), no. 5, 1213–1220.
854. Edward Yourdon, *Flashes on maintenance from techniques of program structure and design*, Techniques of Program Structure and System Maintenance, QED Information Science, 1988.
855. Martin Zachariasen, *Rectilinear full Steiner tree generation*, Tech. Report TR-97/29, DIKU, Department of Computer Science, Copenhagen, Denmark, 1997.
856. Martin Zachariasen and Andre Rohe, *Rectilinear group Steiner trees and applications in VLSI design*, Mathematical Programming **94** (2003), 407–433.
857. Alexander Z. Zelikovsky, *An 11/6-approximation algorithm for the network Steiner problem*, Algorithmica **9** (1993), 463–470.
858. Andreas Zeller, *Why programs fail – a guide to systematic debugging*, second ed., Dpunkt, 2009.
859. *ZIB optimization suite*, http://zibopt.zib.de/, 2009.
860. Joachim Ziegler, *The LEDA tutorial*, http://www.leda-tutorial.org/, 2006.
861. David Zokaities, *Writing understandable code*, Software Development (2001), 48–49.
862. Gerhard Zumbusch, *Parallel multilevel methods. Adaptive mesh refinement and loadbalancing*, Teubner, 2003.