<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Realistic Computer Models</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors(s)</strong></td>
<td>Ajwani, Deepak; Meyerhenke, Henning</td>
</tr>
<tr>
<td><strong>Publication date</strong></td>
<td>2010-01-01</td>
</tr>
<tr>
<td><strong>Publication information</strong></td>
<td>Muller-Hannemann, M., Schirra, S. (eds.). Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice</td>
</tr>
<tr>
<td><strong>Series</strong></td>
<td>Lecture Notes in Computer Science (LNCS, volume 5971)</td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
<td>Springer</td>
</tr>
<tr>
<td><strong>Item record/more information</strong></td>
<td><a href="http://hdl.handle.net/10197/9901">http://hdl.handle.net/10197/9901</a></td>
</tr>
<tr>
<td><strong>Publisher's statement</strong></td>
<td>The final publication is available at <a href="http://www.springerlink.com">www.springerlink.com</a>.</td>
</tr>
<tr>
<td><strong>Publisher's version (DOI)</strong></td>
<td>10.1007/978-3-642-14866-8_5</td>
</tr>
</tbody>
</table>
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 sought 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
5. Realistic Computer Models

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.
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.


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.
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 $O(n \cdot \text{polylog}(n))$ 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 \( \text{sort}(n) = \Theta((\frac{n}{B} \log_2 \frac{M}{B} \frac{B}{n})) \) 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 \( \Theta(n/B) \) upper bound for scanning can easily be obtained by the following simple modification: Instead of accessing one element at a time (incuring 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 \( \frac{M}{M'} \)-way
merge, leading to $O\left(\frac{n}{\log M/B} \log n \right)$ 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 $\phi$ on the nodes of $G$, the problem is to compute another labelling $\psi$ 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 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 $\frac{3}{2}B$ 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 $O(n \log_B n)$ 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 $O(\frac{1}{B} \log \frac{n}{B})$ 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 Weiβ [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 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 Weiβ [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 $Ax = b$ for $x$, where $x$ and $b$ are vectors of length $n$ and the matrix $A \in \mathbb{R}^{n \times n}$ 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
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

\[
x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \quad 1 \leq i \leq n.
\]

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 \( A \) 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 \( A \) 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 \( A \) and \( B \) be two \( n \times n \) matrices stored in the memory mainly intended for the computational model. Compute the matrix product \( C := AB \).
Algorithm 3 Naive matrix multiplication
1: \textbf{for} $i = 1$ to $n$ \textbf{do}
2: \hspace{1em} \textbf{for} $j = 1$ to $n$ \textbf{do}
3: \hspace{2em} $C[i,j] = 0.0$;
4: \hspace{1em} \textbf{for} $k = 1$ to $n$ \textbf{do}
5: \hspace{2em} $C[i,j] = C[i,j] + A[i,k] \cdot B[k,j]$;

---

![Recursive construction of the Peano curve.](image)

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 $C := AB$ 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 $3 \times 3$ 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:

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

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 \(3 \times 3\) 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 \(T(n) = 27 T(n/3)\). 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
5. Realistic Computer Models

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:

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

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 \( 2^{\lfloor h/2 \rfloor - 1} \) subtrees of size \( 2^{\lfloor h/2 \rfloor} - 1 \) each, rooted in the nodes on level \( \lfloor h/2 \rfloor \) 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
5. Realistic Computer Models

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 $O\left(\frac{n}{p}\right)$ 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 $O\left(\frac{n}{p}\right)$ 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 \cite{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 \cite{587} and HBSP \cite{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 \cite{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
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 idiosyncrasies 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 \( O(\log n) \) processors – instead of \( O(n) \) 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 algo-
rithms clearly outperform their optimized CPU counterparts.

The technological change to multicore processors requires not only algorithm-
ic 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 en-
vironment 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 de-
velopment 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 $(k \leq n/2^h)$ 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 $A$ and $B$ 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 $A[i, j]$ denote the $s \times s$ submatrix of $A$ with $s := n/p^{1/3}$. Define $B[i, j]$ and $C[i, j]$ analogously.
1: $P(i, j, k)$ acquires the elements of $A[i, j]$ and $B[j, k]$.
2: $P(i, j, k)$ computes $A[i, j] \cdot B[j, k]$ and sends each resulting value to the processor responsible for computing the corresponding entry in $C$.
3: $P(i, j, k)$ computes each of its final $n^2/p$ elements of $C$ 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 $O(n^3)$ 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 \( A[i,j] \), the \( s \times s \) \((s = n/p^{1/2})\) submatrix of \( A \), and \( B[i,j] \), \( 0 \leq i,j < p^{1/2} \).

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

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 McCall [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 \( n^3/p^{1/2} \geq n^2/p \iff p \leq n^2 \) 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 effi-
ciently 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 com-
munication 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 accu-
rate performance and efficiency estimate. Their design capabilities capture the
most important aspects of parallel computers. Moreover, the analysis can be per-
formed 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 Forum3
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 ex-
tensively than other models to design parallel algorithms in recent years [187].
Even libraries that allow for an easy implementation of BSP and CGM algo-

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

rithms have been developed. Their implementations are topics of a success story
on parallel computing models in Section 5.6.
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 \( B_i \) 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 \( B_i \) 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 \( B_i \) 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 $O(S(n)/\text{Grain}(n) + \text{Grain}(n))$ 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 $\mu$ 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.
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


74. ______, *Combining hierarchical and goal-directed speed-up techniques for Dijkstra’s algorithm*, ACM Journal of Experimental Algorithmics 15 (2010), no. 3.


Bibliography 473


Bibliography


484 Bibliography


490 Bibliography


693. Peter Sanders, *Presenting data from experiments in algorithmics*, in Fleischer et al. [288], pp. 181–196.


725. Saurabh Sethia, Martin Held, and Joseph S. B. Mitchell, *PVD: A stable implementation for computing Voronoi diagrams of polygonal pockets*, ALENEX ’01:


746. Warren D. Smith, How to find Steiner minimal trees in Euclidean $d$-space, Algorithmica 7 (1992), 137–177.


Bibliography


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.


