Chapter 2

Machine Models

with Darren Strash

Draft

In Section 1.1, we introduced machine models as a necessity to abstractly design and analyse algorithms without reference to a particular hardware used to exe-

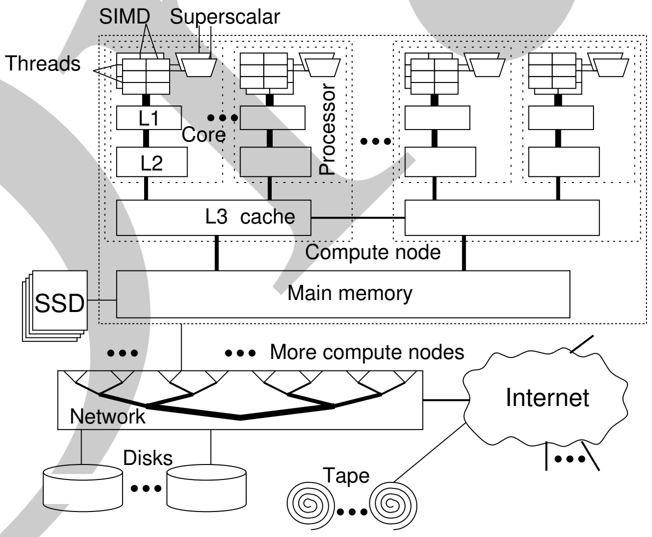


Figure 2.1: Some aspects of computer architecture for which we would like machine models.

cute a program. Generally speaking, we are facing a chicken-egg problem here. We can use machine models to abstract from existing hardware. But we can also strive to develop hardware that matches a certain model in order to simplify programming. The first view is usually justified since computer architecture often follows a quantitative approach [229] changing architectures to improve performance for existing benchmark programs. When this process comes up with new architectural features like caches or pipelines, models have to catch up to make it possible to develop new programs with reasonably predictable performance. This also helps in tuning existing programs.

But also the second view makes sense and can be observed in practice. For example, shared-memory multiprocessors with approximately symmetric memory access costs (SMPs) have been very successful and kept reappearing although complex memory hierarchies promise more peak performance; see also Section 2.5.5.

The "daily work" of an algorithm engineer faces a different question. Out of several or many available machine models, which one makes sense for the problem at hand? A straight-forward answer is to look at the machine one plans to use and pick the "standard" model used for it. A closer inspection shows that the Algorithm Engineering (AE) cycle of Chapter 1 is at work again. During design, analysis, implementation, or experimental evaluation, we may learn that we have to change the machine used or that we have to choose a different (perhaps more detailed) model to explain the peculiarities of the studied software. Switching the model may then also have a profound effect on the way we design and implement our software.

As already discussed in Section 1.1, modeling machines faces a difficult tradeoff between simplicity and fidelity. Figure 2.1 summarizes some aspects of computer architecture that one would like to model. In reality, things are even more complicated – aspects like instruction pipelining, branch prediction, virtual memory, transactional memory, memory access contention, or further complexities of different microarchitectures are not shown at all. To program a modern microprocessor, one has access to thousands of pages of documentation but there is no specification of how expensive a machine instruction is – indeed this varies with the concrete processor model and a lot of context (cache content, state of the execution pipeline, activities of other threads, temperature, etc.).[1]

[1]It is worth noting though that this complexity is a relatively new development. In his pioneering algorithmics book [276] Knuth specifies his MIX machine language where each machine instruction takes the same amount of time. In this model, the running time of a deterministic algorithm was a function that could in principle be analyzed analytically. One author's first computer

Draft

One way out is to work with one or several simple models that each consider one important aspect. The result can then be used to compare how well different algorithms take these aspects into account.

Open Problem 1 (Model-based auto tuning) The case for simplicity in machine models is closely connected to the readability of the result of an algorithm analysis. However, more complicated models can be used to produce complicated but accurate descriptions of the complexity of an algorithm. These can then help to derive the value of tuning parameters. While we know of several failed attempts at that, there are also success stories (e.g. [117]). Nevertheless, the approach seems to warrant additional research. The basic idea is to perform a detailed analysis of one or several algorithms/implementations that estimates their execution times as a function of parameters that describe the machine, input, and the configuration of the algorithm (tuning parameters). If the input and machine parameters are known (derived by measuring them or by requiring them as part of the input), one can use these formulae to select the best algorithm together with optimal tuning parameters. This is potentially much more powerful than traditional auto-tuning (e.g., [21, 42]) that blindly tries many combinations of tuning parameters for a fixed set of inputs in the hope that this finds good values that also work well for future inputs.

In the remainder of this chapter, we describe a wide spectrum of (more or less) simple models. In some cases, we add new variants that address limitations of the basic models. These variants all have a '$^+$'-superscript added to their basic name. Often, we also explain complications of actual hardware going beyond these models. This can be helpful in performance tuning beyond theoretical analysis or in understanding deviations between analysis and experiments. However, the level of detail varies significantly. In part due to space constraint, the expertise (or lack thereof) of the authors, or perceived relevance to AE. For example, Section 2.3 gives a lot of detail on memory hierarchies where the author (and the AE community in general) have a lot of experience. On the other hand, our account of analog computing (Section 2.13.3) remains on a much more cursory level since it

(1983) used an INMOS 6510 processor which understood less than 100 simple machine instructions. Each instruction was documented in detail including the number of clock cycles needed to execute it. Hence, Knuth's approach still worked. Even in the late 1980s, processors like the Motorola 68000 were simple enough for this approach. Then, things like caching and pipelined instruction execution made processors faster but also less predictable. Research in real-time systems since then has struggled to be able to at least guarantee some upper bounds [479].

Draft

can currently be considered an exotic topic with more importance in the past but considerable potential for the future.

There is no crystal clear delineation between abstract machine models as discussed in this chapter and performance tuning for particular architectures. However, issues discussed in this modeling section have the property that one can describe the aspect to be optimized using a simple abstraction. For example, branch-prediction mechanisms are quite complicated but simply striving to avoid mispredictions is a good abstraction; see Section 2.2.5.

Chapter overview. In the following sections, we introduce concrete machine models. Sections 2.1–2.3 discuss increasingly refined models of *sequential* computing. Then Sections 2.4–2.7 present a large number of models for *parallel* computing, where Section 2.4 motivates its importance and where Section 2.7 attempts to bring some order into the "zoo" of possibilities. Section 2.8 on *circuits* takes a lower-level view on computing that allows us to reason about hardware. Sections 2.9–2.11 look at aspects of computing orthogonal to the descriptions above: processing data *streams* (Section 2.9) that do not fit into memory, *fault tolerance* (Section 2.10), and *privacy* (Section 2.11)

Section 2.12 deviates from classical models of computation by outlining some models for *quantum computing* which may revolutionize some areas of computing in the (near?) future. Indeed, there is no scarcity of further *unconventional* models, some of which are briefly discussed in Section 2.13 (e.g., DNA, analog, or neural computing).

While running time is the driving motivation between most of the models above, other resources like space, I/O volume or communication cost are also important. In particular, *energy consumption* is at least equally fundamental. Section 2.14 discusses how to take these resources into account. Section 2.15 summarizes the chapter with a brief look into conceivable futures.

2.1 Turing Machines

Perhaps the first abstract machine model was introduced by Alan Turing in 1936 [457] in order to characterize computability. A finite state machine operates on a tape by reading and writing symbols from a finite alphabet and by moving the tape. Allowing multiple tapes already gives surprisingly high flexibility in programming. See Figure 2.2 for an illustration. Turing machines are used in complexity theory and computability theory because of their great simplicity and flexibility. They are also useful to nail down the complexity of an algorithm in terms of bit

Draft

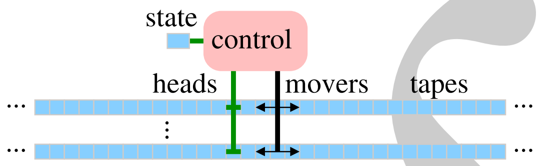


Figure 2.2: Multitape Turing machines.

operations, e.g., for integer multiplication [195, 225]. Turing machines seem less useful for designing algorithms that work well on real-world machines because they differ from them in important aspects (no random memory access, finite alphabet, etc.). However, there are notable exceptions; in their book [424], Schönhage et al. demonstrate how multitape Turing machines can be a good model for engineering numerical algorithms, such as dividing complex numbers or taking square roots.

Exercise 1 *Describe a Turing machine model operating on a two-dimensional tape.*

2.2 The von Neumann Model and its Variants

Most algorithm development, in particular in algorithm theory, is still done using the *von Neumann model* or one of its variants as described below. It goes back to one of the first designs for a universal digital computer [352] and it is a successful way to formalize the basic computational cost of an algorithm. However further aspects, such as memory hierarchies and parallelism, have to be considered later in order to arrive at really high performance in practice. In our experience, this is often more important than using advanced techniques in a simple model, such as complicated bit parallel operations.

2.2.1 Random Access Machines (RAMs)

The *random access machine (RAM)* [429], in its modern form (the *word RAM*), has a computing unit, a register file, and a freely-addressable memory consisting of machine words.[2] See Figure 2.3. All operations execute in constant time.

[2]There are several quite different definitions of the RAM model around. Often, RAMs are also confused with register machines; see Section 2.2.2. In particular, note that RAMs that are only

Draft

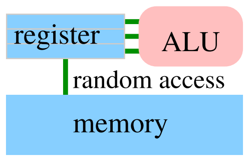


Figure 2.3: The RAM / von Neumann model. ALU stands for arithmetical logical unit.

Operations consist of load and store (also using indirect addressing), arithmetic operations on register operands, and (conditional) branches.[3] The textbook [410, Section 2.2] introduces a variant of this model in more detail – with all operations and an explanation of how pseudocode can be compiled into RAM operations so that we can directly do asymptotic algorithm analysis on pseudocode. The main point here is that by ignoring constant factors throughout, we can sweep many aspects of an actual microarchitecture under the rug. We will also assume an operation for generating machine words consisting of random bits. In reality, randomness is often "simulated" using pseudo-random number generators.

An important technical detail is the machine word size w. By default, it is $\Theta(\log n)$, where n is the input size. In particular, this allows a considerable amount of *word-parallelism*. It may seem odd that we can do a nonconstant amount of work in constant time. However, the alternatives are even odder. Constant-size machine words would not even allow us to address the input. This is the reason why Turing machine models are often used to discuss algorithms where bit-parallelism is not allowed. Sections 2.2.2 and 2.2.3 discuss the effect of unlimited word sizes.

Exercise 2 *Outline how to implement the set operations* $\cup$, $\cap$, $\neg$, *and* $|\cdot|$ *for subsets of* $0..n-1$ *in time* $\mathrm{O}(n/w)$.

allowed to increment or decrement, despite being universal machines, are not useful for analyzing the complexity of algorithms.

[3]From a mathematical point of view, we would get a slightly simpler variant of the RAM model by eliminating the register file. We choose to stick to it for several reasons: First, load-store architectures represent an important development in computer architecture (also part of the RISC versus CISC debate). In particular, access to registers is much cheaper than memory access in practice so we already get a first hint at memory hierarchies. On the theoretical side, dropping the memory gives us register machines as a special case.

Draft

2.2.2 Register Machines

On a RAM with unlimited word size, we can escalate word parallelism to such extremes that nondeterminism can be simulated with only polynomial-time overhead, giving **P** = **PSPACE** [224]. Nevertheless, this variant is useful to discuss computability in general. As a result, we no longer need memory; removing the memory from the RAM model gives the *register machine model*. Registers with "infinite" capacity are also implicit in the real RAM model discussed below.

2.2.3 The Real RAM

In computational geometry [378, 131] one also considers the *real RAM* model where memory cells and registers can store real-valued numbers. Some operations are forbidden in this model (specifically the floor operation), otherwise the model is too powerful, similar to the register machine model. The real RAM is highly unrealistic yet allows us to abstract away issues related to numerical precision in order to develop the algorithmic basis of geometric algorithms. These can then be made realistic by using software libraries that support exact predicates on symbolic representations of real numbers [163]. For example, the real numbers most frequently needed in computational geometry can be represented as roots of a polynomial with rational coefficients (known as *algebraic numbers*). Other than exact computation, robust geometric computation can also be done with fixed precision, or by transformations that preserve a chosen topological property (such as planarity) [428].

2.2.4 Pointer Machines and Other Restricted RAMs

The full generality of the RAM model makes it difficult to prove lower bounds. Therefore, one also considers restricted variants. For example, *pointer machines* [423, 218] are models where arithmetic is not possible. Instead, a finite state machine operates on a dynamic graph. Other, more ad hoc restrictions are used for particular families of problems. For example, for sorting and related problems, we can consider elements that can only be moved, copied, and compared with a $\leq$ operation but not otherwise manipulated or inspected.

2.2.5 Instruction Parallelism

As a first small step to modeling parallel processing, one can take into account that modern microprocessors can execute several machine instructions in each clock

Draft

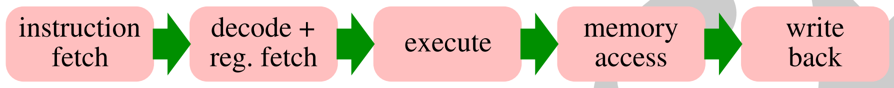


Figure 2.4: A simple 5-stage pipeline [229] that splits an instruction execution into fetching the instruction, decoding it and fetching input data that is stored in registers, performing actual calculations, memory access, and writing back results to registers.

cycle. This is achieved by *pipelining* and *instruction parallelism*. In pipelining, instruction execution is split into suboperations. Figure 2.4 shows a simple 5-stage pipeline. Current pipelines for high-performance processors are considerably longer – sometimes using 20 or more stages. There is one pipeline stage for each suboperation. Each pipeline stage handles one operation in each clock cycle. Several such pipelines are running in parallel – often specialized to particular types of operations (integer, floating point, load, store, etc.). Thus, overall, several dozen machine instructions are being executed in parallel at any point in time. This works fine for straight-line sequences of instructions without too many data dependencies between the operations. In this situation, conditional branch instructions are a problem because a branch can interrupt the stream of instructions. When this happens, many partially executed instructions have to be abandoned, their effect has to be rolled back, and it will take several clock cycles until the next instruction is completed.

Processors therefore invest considerable resources into *predicting* the outcome of a branch instruction. The instruction stream can then be continued in the predicted way without emptying the pipelines. In the most simple case, the compiler can predict a branch as taken or not taken. For example, the branch at the end of a repeat–until loop can be predicted as taken. This will fail only at the last iteration of that loop. More sophisticated techniques discover patterns in the most recent executions of a branch instruction using simple state machines [229]. Branch prediction works surprisingly well in practice. Computer architecture textbooks report typical rates of at least 90% correct branch predictions [229].

However, in some algorithms, branch mispredictions are hard to avoid. For example, efficient comparison-based sorting algorithms need about $n \log n$ element comparisons. Traditional implementations of these algorithms associate one conditional branch with each of the comparisons. For fundamental information-theoretic reasons, these branches cannot be predicted at all – they are taken 50% of the time in a completely unpredictable way regardless of how much prediction

Draft

hardware is used. In such algorithms, the pipeline interruptions due to branch mispredictions can completely ruin performance. Algorithm analysis therefore sometimes also analyzes how many branch mispredictions can happen [351, 478]. More importantly, there are techniques to avoid conditional branches [263, 420, 159, 37]. This can greatly improve performance. In order to perform comparison-based algorithms without conditional branches, one has to dissociate comparisons from branches. The idea is that only very simple operations should depend on the comparison. For example, superscalar sample sort [420, 37] only performs an increment operation if a comparison operation computes the value **true**. Such simple operations can be done using *predicated instructions*. These are only executed if a special condition flag is set that is the outcome of a comparison. Otherwise, predicated instructions do nothing. Note that skipping one machine instruction costs only a fraction of a clock cycle while a branch misprediction costs several clock cycles.

***Exercise 3** *Develop a routine for binary search of an element x in a sorted array a of size n that uses conditional branches only for testing loop exit and for a conditional move of the form* **if** c **then** $a := b$. *You can assume that n is known at compile time. Discuss how to generalize the code so that it can make a batch of several searches in an instruction parallel way. Why is that still likely to be slower than searches based on implicit search trees as used in super scalar sample sort[420, 38]? Implement your solution and benchmark it compared to a more classical formulation of binary search, e.g., [410, Section 2.7].*

Open Problem 2 (Priority queues without branch mispredictions) Design a priority queue that avoids conditional branches like the super scalar sample sort [420, 37] and, at the same time, is as cache efficient as sequence heaps [396] . Can this be implemented in such a way that significant performance improvements are possible?

Open Problem 3 (Where do branch mispredictions matter?) Most algorithms where branch mispredictions are known to matter are comparison-based algorithms for sorting and related problems (e.g., merging and partitioning). Can you find further algorithmic problems? Flow computations? Monte Carlo simulations? How should these algorithms be modified to eliminate hard-to-predict branches?

SIMD Instructions. Another dimension of instruction parallelism is exploited

Draft

by *SIMD instructions* (aka *vector instructions*) that work on extra-wide registers (currently up to 512 bits) containing short vectors of numbers (e.g., 32 entries of 16 bits each). For example, an addition of two SIMD registers would add the two stored vectors component-wise.

2.3 External Memory

In computer architecture, there is a large spectrum of technologies for storing data. In particular, we face a tradeoff between price per bit on the one hand and speed on the other hand. Furthermore, there are fundamental physical reasons (like the limited speed of light) why a large memory must have large access latencies. Thus, there are good reasons why real-world (sequential[4]) computers with good performance must have both large cheap memory and small fast memory. This runs counter to the uniform memory in RAMs and a main principle in von Neumann's original idea of a universal computer [352]. We now discuss simple abstract models that grasp the resulting *memory hierarchy*.

The basic *external* or *secondary memory* model (EM) [467], also called the I/O model, is very simple; see Figure 2.5. We have a random access machine with (fast) memory limited to M machine words. In addition, there is a large secondary memory. Access to secondary memory is in blocks of size B. In algorithm theory, analyzing external memory algorithms amounts to counting the number c of block accesses *(I/Os)*. Sometimes we also use the *I/O volume* is then cB. In AE, we additionally analyze the *internal work*, e.g., by counting executed machine instructions as in the RAM model. The simplicity of the EM model makes it very

[4]For parallel architectures, there is an option to partition a large cheap memory into many small pieces, each equipped with its own processor core that then has small uniform access latency to its *local* piece of memory (aka *processing in memory (PIM)* [264]); see also Sections 2.4.6 and 2.5.5.

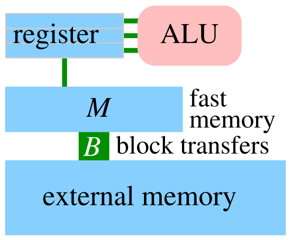


Figure 2.5: The external memory model

Draft

flexible. Slow and fast memory can be any two levels of the physical memory hierarchy. The two levels hard disk and main memory were originally most important. Today, main memory and some unspecified level of the cache hierarchy are at least equally important. The EM model is highly successful because of its simplicity and because algorithms that perform well in this model often also perform well in practice; see [330, 466] for overviews of results.

In the subsequent sections, we present further details and variants of the EM model. This is perhaps the most detailed elaboration of a family of models in this book. We view this as a convenient example of how a simple successful model can relate to reality and refined models. The reader should assume that many of the remaining models in this book deserve a similar amount of further detail. Section 2.3.1 explains how many EM algorithms or their complexity can be expressed based on sorting and scanning primitives. Sections 2.3.2 and 2.3.3 discuss how the EM model does (or does not) grasp details of a real-world machine. Then Section 2.3.4 generalizes the EM model to handle parallel disks. These are the basis for modeling solid state disks in Section 2.3.5 and other nonvolatile memory in Section 2.3.6. Peculiarities of hardware caches are discussed in Section 2.3.7. Section 2.3.8 then explains how a simple twist of the EM model can be used to model algorithms that perform well in multilevel memory hierarchies. Finally, Section 2.3.9 discusses how virtual memory affects algorithm analysis and design. Another variant of the EM model yields a useful model for parallel memory hierarchies; see Section 2.4.6.

2.3.1 Sorting and Scanning

Two simple algorithmic techniques permeate the design of external memory algorithms: *Scanning* n elements (machine words) is possible with n/B I/Os. The shorthand $\text{scan}(n)$ is used for this expression. *Sorting* n elements takes $\text{sort}(n) = \Theta(n/B\lceil 1+\log_{M/B} n/M\rceil)$ I/Os [6]. Using multiway mergesort (Section 11.1.3) we can see that an upper bound for the constant factor in $\text{sort}(n)$ is 2 for sorting machine-word-sized elements.

Open Problem 4 (Exact lower bound for external sorting) The constant factor in the lower bound is still open. Aggarwal and Vitter [6] basically show a factor of 1 and that the factor becomes 2 when one assumes that the number of inputs is the same as the number of outputs. However, this leaves open the existence of more efficient algorithms that do an asymmetric amount of reading and writing.

Draft

Exercise 4 *Describe an external sorting algorithm that needs just $\lceil n/B \rceil$ output operations. You can expend a large number of input operations. But try to limit them to $\mathrm{O}(n^2/BM)$. Also, limit internal work to $\mathrm{O}(n^2 \log n/BM)$.*

The sorting bound is also a lower and/or upper bound for many other computational problems even if the actual algorithms used to solve them are quite different.

From an AE perspective, sorting and scanning are convenient ways to distinguish more or less complicated algorithms in a qualitative sense. However, in practice, the term $\lceil \log_{M/B} n/M \rceil$ is exactly one[5] in many situations. Let us consider the case of hard disk versus main memory. In the last few decades, the cost ratio between mechanical hard disk memory and RAM has remained at around 200. This ratio is not likely to increase dramatically as long as RAM capacities improve at least as fast as hard disk capacities. Hence, in a *balanced* system with similar investments for both levels of memory, the ratio between input size and internal memory size is not huge. In particular, M/B is likely to be much larger than than the cost ratio. But as long as $M/B > n/M$, we have $\lceil \log_{M/B} n/M \rceil = 1$. The cost ratio for nonvolatile memory[6] (SSDs) versus main memory is even smaller. We can have $\lceil \log_{M/B} n/M \rceil > 1$ when straddling several layers of the memory hierarchy, e.g., when fast memory is L1 cache size (kilobytes) and slow memory is the main memory of a large server (terabytes).

2.3.2 What is the Block Size?

On the first glance, the block size B is a parameter defined by the hardware, e.g., the cache line size of a certain hardware cache level. However, a closer look reveals that often there is not one clear block size imposed by the hardware and that B should actually be considered a tuning parameter of the implementation.

This is particularly clear for mechanical hard disks. There are hardware-imposed block sizes used for error detection and correction. However, these values are much too small to be useful for external memory algorithms. A reasonable *linear model* for the time to access ℓ consecutive bytes of data on a hard disk is $\alpha + \beta\ell$ where α is a startup overhead accounting for mechanical and software delays and where β is the achievable data rate once access has started. Indeed, it might be attributed to a historical accident that this model is unusual for hard

[5]Of course, the value is zero when processing can be done within internal memory.

[6]*Nonvolatile* memory does not lose its state when the power is switched off.

Draft

disks but standard for message passing; see Section 2.5. The above linear model suggests that $B \approx \alpha/\beta$ is a reasonable value for the block size. If we make random accesses that actually need less data per access, we are at most twice as slow as using a small block size. If we make a large consecutive access we are at most twice as fast as accessing the same amount of data with random access of block size α/β. For current hard disks, this means that block sizes should be a couple of megabytes. However, we can also turn this argument around. We can achieve acceleration up to a factor of two by choosing a block size more appropriate for the application at hand. For example, for sorting we can choose large block sizes as long as this leaves $\lceil \log_{M/B} n/M \rceil$ at a value of 1. For index data structures with access time $\mathrm{O}(\log_B n)$, we may want to choose smaller blocks; see also Section 13.2.

We have moved this discussion outside the section on hard disks because it emphasizes the fact that block sizes are tuning parameters. At the other extreme, even hardware cache lines may not be the right choice for the tuning parameter B. For example, many Intel processors access two consecutive cache lines considerably faster than two arbitrary cache lines. Hence, setting B to two (or more) cache lines may be a reasonable choice.

2.3.3 Modeling Mechanical Hard Disks

In Section 2.3.2 we used the formula $\alpha + \beta\ell$ for modeling the time needed to access ℓ bytes of data on a disk.[7] This is a gross oversimplification for mechanical hard disks. There are three main additional issues illustrated in Figure 2.6.

A. The disk *rotates* and the time needed to rotate to the beginning of the intended data block depends on the current rotation angle. Thus, for a particular data block, the *rotational delay* will oscillate over time corresponding to a sawtooth-shaped periodic function.

B. The access head has to *seek* to the right track before accessing a block.

C. The disk rotates at fixed angular velocity[8] and the data is stored with an (approximately) fixed number of bits per millimeter of disk surface. This

[7] Hard disks are getting less and less important as this book is written. We still believe that this section remains interesting as an example of how more detailed modeling can have an appreciable performance impact yet is also impractical in many cases.

[8] There are exceptions, e.g., audio CDs.

Draft

implies that data stored on the outer zones of the disk is transferred faster than data on inside zones. Further complications arise for example due to caching within the disk and because faulty tracks on the disk are replaced by reserve tracks (usually on the slow inside zones of the disk).

These peculiarities can in principle be exploited for algorithm tuning. Batches of blocks to be read may be scheduled so that overall rotational delays are minimized [191, 443]. Data accessed together can be stored close together. Important data could be stored on outer zones.

Such tuning measures are good examples of optimizations one should often *avoid* since they can be fragile and nonportable. These optimizations should only be done on the right level of the software stack and using the right abstractions. The processor controlling the disk can try to perform optimizations for issues A and B when it is given batches or queues of outstanding I/Os. A disk usually presents itself to the operating system as an array of blocks. Hence, the operating system or a runtime system of a database could perform optimization for issue C if it is understood that blocks with the smallest index are the fastest ones.

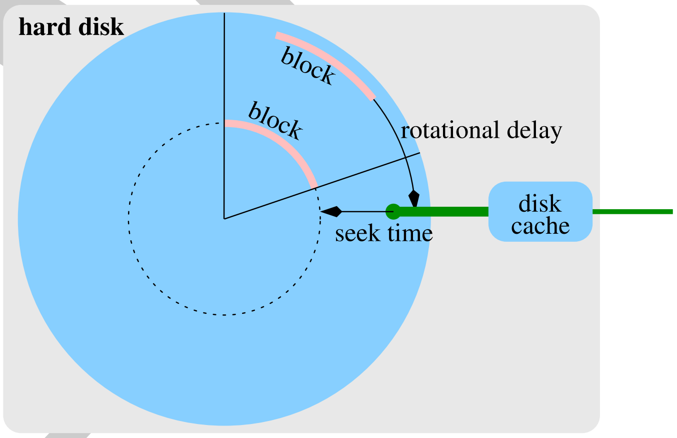


Figure 2.6: Schematic drawing of a mechanical hard disk – illustrating seek time, rotational delay, and different data densities.

Draft

2.3.4 Parallel Disks

The external memory model has been generalized to consider D identical disks – in one I/O step, D blocks can be transferred [6, 467]. Aggarwal and Vitter [6] propose a simple variant where *any* D blocks can be accessed (referred to as *access-any* variant). Vitter and Shriver [467] allow only *one* block to be accessed from *each* disk (access *one-each*). Figure 2.7 depicts these two model variants. In both variants, we can hope to reduce the number of I/O steps by a factor up to D. To achieve this speedup, we need enough parallelism in the application. The one-each variant in addition requires clever data allocation and disk scheduling (e.g., see [141, 250] for sorting).

The one-each variant is more realistic than the access-any variant. The access-any variant is still useful because it allows to discuss parallelism independently of memory allocation and scheduling – see also Section 2.3.5 on SSDs. Moreover, the access-any variant can emulate the one-each variant with surprisingly small overhead [407] (a small constant factor using randomization) and there are generalizations to asynchronous access, heterogeneous disks, etc. [397, 398].

Exercise 5 *Give an example access pattern, where the one-each variant of the parallel disk model takes D times more input steps than the access-any variant. What changes with this pattern if each data block is independently allocated to a random disk?*

A simple way to exploit parallel disks is *striping* – we concatenate D physical blocks of size B to one logical block of size DB. We can then apply any single-disk algorithm using block size DB. This automatically exploits disk parallelism

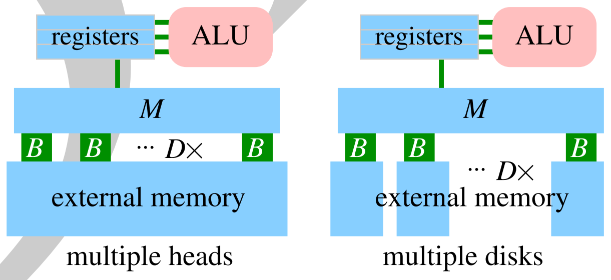


Figure 2.7: The external memory model with parallel heads (access any) [6] and parallel disks (access one-each) [467].

Draft

in a perfectly load-balanced fashion. The problem with this approach is that not all applications can make very good use of such large block sizes or that there may not be enough fast memory to store enough blocks in internal memory.

2.3.5 Solid-State Disks

Solid-state disks (SSDs) are built using semiconductor technology that does not lose its memory when power is lost [331, 7]. Access is done in blocks to hide access latencies and to simplify error correction. SSDs are cheaper than RAM because their cells are slightly smaller, because they store several bits per cell, and because one can stack memory cells in hundreds of layers. On the other hand, SSD access is slower than RAM access. At the time of writing, SSDs are rapidly replacing mechanical hard disks in more and more applications – they are faster, more compact, and need less power. In particular, the nonvolatile memory of a smartphone is essentially an SSD.

SSDs have smaller blocks than hard disks, higher overall bandwidth, and much lower access latency. There are also two important qualitative differences compared with hard disks. First, there is a marked asymmetry between reading and writing. Writing is usually slower, consumes more energy, and often uses larger block sizes than reading. More precisely, what is most expensive and uses larger blocks is *erasing* data blocks which is required to overwrite them with different data later. Also, erasing the same physical block multiple times wears it out and eventually destroys it. The disk controller therefore employs *wear-leveling*

Draft

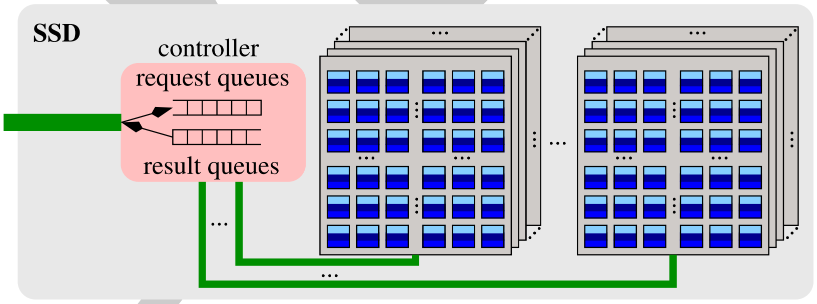


Figure 2.8: Schematic drawing of a solid state disk (SSD). Each cell stores several bits. Many layers of cells and chips side by side give significant parallelism in request processing.

algorithms that change the mapping from logical to physical blocks to spread out write accesses over the entire disk. For the user, it is important that background processes for wear-leveling and erasing invalidated blocks can cause performance anomalies in particular for applications with complicated writing patterns. Also, a nearly full SSD can be much slower for writing than an empty one.

The second important difference is that to achieve maximum throughput of random disk accesses, one needs to make many accesses in parallel. The currently dominating NVM Express protocol (NVMe) therefore offers 2^{16} command FIFO queues, each of which can buffer up to 2^{16} commands. Figure 2.10 illustrates this view of an SSD. For example, in an external hash table benchmark [281] we use an asynchronous parallel approach where up to 128 requests can be in flight at a time. In a sense, each SSD device behaves like an array of many small parallel disks. However, we have no control over the allocation of logical blocks to these "sub-disks".

Overall, two simple models for SSD seem reasonable. We can use the single-disk EM model with a block size much larger than the physical block size of the SSD. This will exploit the parallelism within the SSD because the controller internally stripes large blocks over the different memory modules. For applications with many random accesses to small data objects, it is better to use the access-any variant of the parallel disk model with a sufficiently large value of D to grasp the parallelism within the SSD. The more complex one-each model is not helpful here since we have no control over the allocation of blocks to disks.

Exercise 6 *Produce a table comparing mechanical hard disks, SSDs and DRAM memory at the current technology and market prices. Possibly include both mainstream and high-end variants of the hardware. Compare cost per bit, bandwidth, latency, and energy consumption. Document your methodology – from where do you get the prices? How do you define latency? For energy consumption, can you differentiate between reading, writing, and idling? For hardware that you actually have at hand, try to compare actual measurements with data sheets or benchmarks done elsewhere. Discuss differences.*

2.3.6 Other Nonvolatile Memory

SSDs may be only an intermediate step to nonvolatile memory that looks more like main memory than like a disk. This means that block sizes are cache line sizes and that reads are as fast as for main memory. Writes may still be more expensive in terms of time and energy consumption and the memory may wear out faster

Draft

so that the number of write operations to a cache line has to be limited. This situation can be analyzed in the context of *write-efficient algorithms* [74] where one considers write operations to main memory to be a factor ω more expensive than read operations. Another interesting aspect of this is how to make algorithms *persistent* when using such memory, i.e., to allow restarts after failures that erase the local state of the processors and the content of the fast memory [76, 477].

2.3.7 Hardware Caches

One assumption of the EM model is that the algorithm has full control over the content of the fast memory. In contrast, the content of a processor cache is controlled by the hardware. A typical strategy are *a-way set associative* caches [229], where a is a small constant: Suppose the cache contains ak cache lines. It is then divided into k cache sets of size a. Cache line i is mapped to cache set $i \bmod k$. Each cache set is managed separately using (an approximation of) the least recently used eviction strategy (LRU), i.e., when a new cache line enters a cache set s then the least recently used block in s is evicted; (see also Section 2.3.9). Sometimes the LRU strategy is only approximated. The case $a = 1$ is called *direct-mapped cache*. Figure 2.9 illustrates this variant of the EM model.

In practice, set-associative caches work quite well, even for relatively small values of a [229], e.g., $a = 4$. However, bad situations may arise. Even direct-mapped caches are efficient if one maintains full control over memory allocation and simple deterministic access patterns of the algorithm. For more complex access patterns there is less previous work. In [326] we consider the situation where the program scans k arrays of total length n. This covers many external memory

Draft

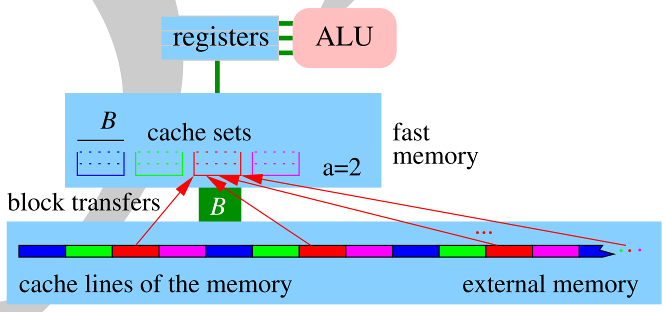


Figure 2.9: External memory with set-associative fast memory.

algorithms akin to sorting. If $k > a$, every element access may cause a cache fault in the worst case. However, if the starting addresses of the arrays are randomized, $\mathrm{O}(n/B)$ cache faults suffice provided that $k = \mathrm{O}\left(M/B^{1+1/a}\right)$. One can also bypass the hardware cache replacement [189, 427] – although at the cost of considerable overhead.

Exercise 7 *For the computer you are mainly using, try to find as much information as you can on its memory hierarchy: how many cache levels? block sizes? associativity of each level? speed? Make a table and document your sources.*

2.3.8 Cache-Oblivious Algorithms / Multilevel Hierarchies

Of course, we can design algorithms that explicitly handle multiple levels of memory hierarchy and then analyze the number of I/Os for each level. However, this is rarely done since it leads to complicated algorithms and complicated results of the analysis. Anyway, in practice usually two levels of the hierarchy will be the performance bottleneck so that setting the parameters B and M of the EM model to these parameters will yield an efficient program. However, which two levels are relevant may depend on the actual machine and on the size of the input. For example, when the input fits in the L3 cache, L2 cache misses may be the performance bottleneck whereas, for large inputs, L3 cache misses may dominate. Hence, in general we need an (auto)tuning mechanism to tune the parameters B and M.

An elegant alternative to tuning is to design I/O-efficient algorithms that do not rely on the values of B and M. Such *cache-oblivious algorithms* are automatically efficient for all levels of the memory hierarchy [189, 279, 25]. For example, an algorithm that just scans an array of size n will need $\mathrm{scan}(n)$ I/Os regardless of the concrete values of B and M. Section 9.3 describes cache-oblivious unbounded arrays and queue-like data structures. Hashing with linear probing (Section 10.4.1) is another simple example for a cache-oblivious algorithm.

Open Problem 5 (AE for cache-oblivious algorithms) Few cache-oblivious algorithms have been evaluated experimentally [89, 227, 91]. Even fewer can actually compete with the best cache-aware algorithms. Even defining “compete” is an interesting question here. How much overhead is worth the robustness of avoiding tuning? Does the cache-oblivious algorithm ever outperform a cache-aware algorithm that uses fixed values for M and B across all inputs and machines? Hence, AE for cache-oblivious algorithms is a wide-open topic.

Draft

Exercise 8 *Perform a case study on engineering algorithms for matrix transposition. Previous algorithms and proof-of-concept implementations [279, 189] can serve as a starting point. To simplify the situation, assume that we transpose $2^k \times 2^k$-matrices out-of-place, i.e., the result is in a separate piece of memory. Compare*

a) A straight-forward nested loop implementation

b) A tuned cache-aware algorithm that cuts the matrix into tiles of size $B \times B$. Subroutines for moving or swapping and transposing tiles should be carefully tuned.

c) A simple recursive cache-oblivious algorithm

d) A tuned cache-oblivious algorithm that uses a tuned base case and a tuned copy operation that may be similar to the ones developed for 2.

e) A carefully tuned code from a numerical library.[9]

Compare the versions for different input sizes and on different architectures.

2.3.9 Virtual Memory

Virtual memory is a combined hardware/operating system mechanism that allows all processes to use the same logical address space starting from 0. Ideally, this mechanism should be invisible. However, performance penalties due to the translation of virtual to physical addresses do show up and can also be modeled. One important effect is due to the *translation lookaside buffer (TLB)* – a cache for m page addresses that allows us to quickly translate accesses to these pages. TLB misses can be modeled like cache faults for a cache where page size B is the virtual memory page size and where the cache size is $M = mB$. One can reduce TLB misses by configuring the operating system to use larger page sizes. Applications that have many TLB misses (e.g. when they frequently access large hash tables) experience memory access delays that are not constant but grow with the input size. The reason is that tree-like data structures are used to resolve TLB misses [261].

[9] The documentation of operation *cmatrixtranspose* in the Alglib library (`www.alglib.net/translator/man/manual.cpp.html`, accessed Oct. 17, 2023) mentions that it is actually using a cache-oblivious implementation. Often matrix transposition is a special case of a more flexible operation, e.g., with the suffix *matcopy*.

Draft

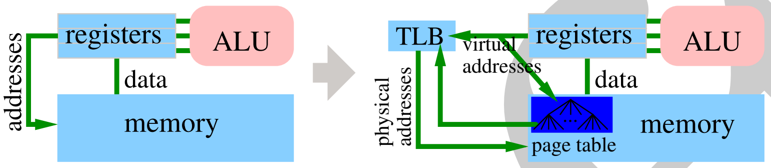


Figure 2.10: Left: schematic of the RAM model with the address data path exposed. Right: refined model of the address data path taking translation lookaside buffer (TLB) and the page table data structure into account.

Virtual memory also supports programming techniques that would otherwise not be available. For example, the Templated Portable I/O Environment (TPIE) library for external memory computing [27] uses operating system mechanisms to emulate a logical memory that is larger than the physical memory. However, most efficient implementations of external memory algorithms use more efficient and more flexible mechanisms. A related (efficient) technique is *memory overcommitting* where the logical address space used is larger than the physical one but the program never accesses more than the physically available memory. This allows the allocation of several large arrays whose total size is known but whose individual size is unknown. Section 10.8 gives an application for space-efficient hash tables.

When a large external memory is used to extend a small physical memory, efficient caching mechanisms become essential. The LRU strategy mentioned in Section 2.3.7 is considered reasonably effective here. However, it is expensive to implement it precisely. Therefore, various approximations are considered whose precise implementation depends on the capabilities of the required hardware. For example, one can maintain a (possibly approximate) priority queue of cached pages whose priority is a time stamp. Since maintaining precise time stamps is too expensive, one can maintain a lower bound for the last access. When a page has the lowest-known bound, it is not immediately evicted but goes into a (possibly approximate) FIFO of eviction candidates. In addition, the page is marked to throw an exception when it is accessed next. If this happens before the page is actually evicted, a fresh timestamp is noted and used to reinsert it into the priority queue.

A disadvantage of LRU is that it caches data that is merely scanned and never accessed again. Therefore various refinements have been designed that evict pages that are "not in active use".

Draft

2.4 Shared-Memory Parallel Models

Perhaps the main weakness of the RAM model is that it considers only sequential algorithms. However, it is difficult to make sequential computers faster. Increasing the clock frequency reduces energy efficiency. Increasing instruction parallelism only helps for a limited set of computations. Increasing the size of the memory increases access latencies. Overall, there is a limited return on investment regarding performance when one invests more transistors or more energy to get faster sequential computers. These limitations can be overcome by looking at parallel computing that allows near-linear scaling of computing power with the invested resources. Not surprisingly, the most successful models of parallel computing are slight generalizations of the RAM model. In this section, we begin with the approach to keep a global *shared* memory and to replicate the processing cores. Section 2.5 replicates RAMs and connects them by a network. A more abstract view taken in Section 2.6 is to look at a small set of parallelizable operations applied to sets or sequences

We begin with the simple PRAM in Section 2.4.1 that can already serve as a basis for devising parallel algorithms in a high-level fashion, concentrating on exposing the parallelism in the problem. There, we also introduce several important conventions for analyzing parallel algorithms. On the other hand, real shared-memory machines are ubiquitous now from smartphones to servers with hundreds of cores. Further subsections cover important aspects of real-world shared-memory machines.

2.4.1 Classical Parallel Random Access Machine (PRAM)

A PRAM consists of p execution units (PEs) of random access machines (RAMs, Section 2.2.1) attached to a single shared memory. The PEs are numbered from 1 to p (or from 0 to $p-1$, or whatever is most convenient for the algorithm description), which are their *IDs*. As in the RAM model, instructions are assumed to need constant time. The PEs work in a lockstep fashion, i.e., those PEs that load a memory cell do so based on the value at the beginning of a time step. Those that write a value to a memory cell store this value during the step and the value is visible there in the next time step.

PRAMs come in several variants depending on the rules for concurrent access to the same memory cell. In the acronyms for these variants, an “E” stands for “exclusive”, i.e., concurrent access to the same memory cell in the same time step is forbidden and a “C” stands for “concurrent”, i.e., concurrent access is allowed.

Draft

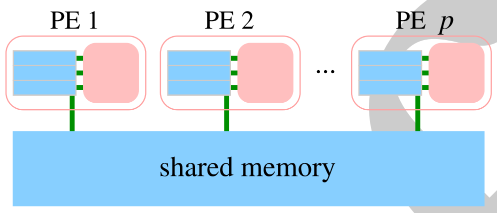


Figure 2.11: The parallel random access machine (PRAM).

By distinguishing between read access (R) and write access (W), we get three main model variants – EREW, CREW, and CRCW.

Exercise 9 *What would an ERCW PRAM be? Discuss why this variant is not important.*

For CRCW PRAMs there are several submodels depending on the semantics of concurrent writes:

Common: Concurrent accesses are only allowed if all PEs that attempt to write to the cell concurrently write the same value. This is the weakest model variant but is already surprisingly powerful.

Arbitrary: If several different values are written concurrently to a cell, one of these values will be stored. The algorithm has to remain correct regardless of which of the values is chosen.

Priority: Among the PEs that try to write, the one with the smallest PE ID writes. This should not be confused with a prioritization based on the written value which falls into the Combine category below.

Combine: A commutative and associative operation like sum, min, max, or xor is applied to the written values. This is a fairly expensive operation.

Table 2.1 gives examples.

An abstract representation of a PRAM computation is a directed acyclic graph (DAG) of elementary operations where one measures the size of the DAG (*work*) and the number of operations on its longest path (*span*) which is a measure for the *latency* of a computation. The goal is often to achieve work similar to the best-known sequential algorithm and low span, preferably *polylogarithmic* in the

Draft

Table 2.1: Results of a concurrent write operation to memory cell 42 using different variants of the CRCW PRAM model.

input			result			
PE 1	PE 2	PE 3	Common	Arbitrary	Priority	Combine-Add
3	3	3	3	3	3	9
1	2	3	Error	$x \in \{1,2,3\}$	1	6
2	1	3	Error	$x \in \{1,2,3\}$	2	6

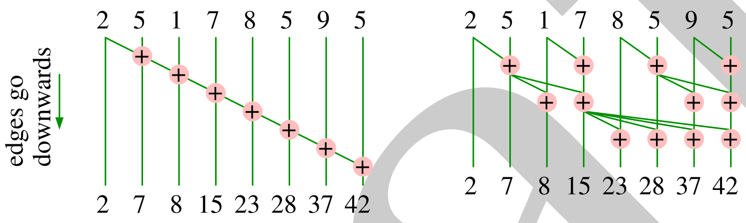


Figure 2.12: DAGs for two ways to compute 8-element prefix sums.
Left: with span and work 7 (additions). Right: with span 3 and work 12.

input size n, i.e., $\log^{O(1)} n$. See Figure 2.12 for an example. Note the relation to the size and depth of a circuit explained in Section 2.8.1.

PRAMs have been criticized as unrealistic because of their lockstep operation principle and because they do not account well for communication costs. However, our impression is that this criticism is only partially warranted. PRAMs are an easy way to express parallelism and are thus a logical first step to a parallel algorithm. Many PRAM algorithms have later been implemented on realistic machines. In our opinion, what should be questioned is the large number of *inefficient* PRAM algorithms that invest a polynomial factor of additional work in order to achieve a polylogarithmic span. However, this issue is not so much due to problems with the PRAM model but with an unrealistic framework for algorithm analysis.

Open Problem 6 (Slow but efficient parallel algorithms) Polylogarithmic span has been a primary objective of parallel algorithm development since the 1980s for complexity-theoretic reasons [214] – problems that do not allow polylogarithmic span were deemed to be hard to parallelize. However, from an AE perspective, *efficient* algorithms with larger span, such as n^α for $\alpha < 1$, are perfectly fine. Can we find such algorithms for problems that are otherwise hard, e.g., BFS, strongly

Draft

connected components (see also [179, 144])? Recently, several promising theoretical results in this direction have been achieved for problems related to graph reachability [256, 176].

Exercise 10 *Describe fast algorithms that perform the logical and of n Boolean values using n PEs of a PRAM for a) CREW PRAM; and b) common CRCW PRAM. Compare the achieved running times.*

2.4.2 Asynchronous PRAM

To refine PRAM algorithms for real-world shared-memory machines, one can use an asynchronous model with queued writing [202, 203, 410]. This is a clean model of the memory access *contention* that is a major performance problem in practice. Let us single out the aCRQW PRAM model – the asynchronous concurrent read queued write PRAM [410, Section 2.4.1]. We assume concurrent reads without delays since the local caches of the PEs enable contention-free concurrent read accesses in many practical situations. However, concurrent writing to the same memory cell involves *queuing*.[10] More concretely, a FIFO queue is associated with every memory cell. During any clock cycle and for any cell C, first, all read operations to C return its old value. Then write operations to C are appended to the queue of C – possibly several ones. Finally, the first write operation in the queue of C is executed and the corresponding operation finishes. The remaining write operations remain in the queue, delaying the corresponding PEs. Table 2.2

[10]In practice, contention may also happen when different cells are accessed, e.g., when they are located in the same cache line (*false sharing*). Careful implementation can often avoid such effects.

Draft

Table 2.2: Example of queued writing where PEs 1–3 concurrently access memory cell $S[42]$. Each row corresponds to one clock cycle on an aCRQW PRAM (which is still an abstraction of a real-world machine).

$S[42]$				
value	queue	PE 1	PE 2	PE 3
0	$\langle\rangle$	$S[42] := 11$	$S[42] := 22$	$S[42] := 33$
11	$\langle 2,3\rangle$	$S[42] := 111$	queued	queued
22	$\langle 3,1\rangle$	queued	instr. x	queued
33	$\langle 1\rangle$	queued	instr. y	instr. v
111	$\langle\rangle$	instr. u	instr. z	instr. w

gives an example. Other operations may also vary in their execution time – we drop the lockstep synchronization assumed for PRAMs. Further variants can be considered.

2.4.3 Atomic Operations

To achieve consistent behavior in asynchronous shared-memory machines, we need *atomic operations* that perform a set of memory operations uninterrupted by other PEs; see also [410, Section 2.4.3]. The most widely considered atomic operation is compare-and-swap (CAS). A call CAS(i,e,d) specifies a value e expected to be present in memory cell i and the value d that it wants to write. If the expectation is true, d is written to memory cell i, the operation succeeds by returning 1. If the value is different, some other PE has modified cell i in the meantime. CAS writes the new value of cell i into e and fails by returning 0. See Table 2.3 for an example. CAS can be used to implement locks and other synchronization primitives. It can also be used to update the content of a cell. For example, a loop of CAS operations can be used to atomically add an offset to the content of a cell. This fetch-and-add operation as well as similar update operations are also directly supported by many architectures (see Table 2.4 for an example). Particularly special are *priority updates*, where [432] $S[i] := \max(S[i], x)$. If one assumes that updates arrive in an order that is not correlated to their value, the cell value changes only a logarithmic number of times and thus the updates lead to little contention.

Table 2.3: Example of two concurrent CAS instructions executed on memory cell $S[42]$ in the aCRQW PRAM model. The columns labelled R_1 and R_2 give the current value of these registers. PE 1 succeeds in writing the value 1. PE 2 is first queued and then its CAS instruction fails because the actual value of $S[42]$ is now 1 while value 0 was expected. The actual value is returned in register R_1.

$S[42]$		PE 1			PE 2		
val	queue	instruction	R_1	R_2	instruction	R_1	R_2
0	$\langle\rangle$	$R_2 := CAS(42, R_1, 1)$	0	-1	$R_2 := CAS(42, R_1, 2)$	0	-1
1	$\langle 2\rangle$	instr. x	0	1	queued	0	-1
1	$\langle\rangle$	instr. y	0	1	instr. z	1	0

Some architectures support *transactional memory*. Here, a computer program can label a (small) subsequence of instructions as a *transaction*. The hardware

Draft

Table 2.4: Example of two concurrent fetch-and-add operations incrementing memory cell $S[42]$ in the aCRQW PRAM model.

$S[42]$	queue	PE 1	PE 2
0	$\langle\rangle$	$\mathit{fetchAdd}(42,1)$	$\mathit{fetchAdd}(42,1)$
1	$\langle 2\rangle$	instr. x	queued
2	$\langle\rangle$	instr. y	instr. z

guarantees that a transaction is either executed atomically or fails without changing the memory. Transactions can immensely simplify the design of concurrent programs and they often improve performance. A downside is that there are no hard guarantees that transactions will eventually succeed (perhaps after some retries). Hence, a program also needs a fallback implementation using traditional techniques such as locks or simpler atomic operations.

Exercise 11 *Repeat Exercise 10 (logical and) for the aCRQW PRAM using CAS instructions. How can you achieve logarithmic worst-case execution time?*

2.4.4 Lock-Free and Wait-Free Algorithms

Multiple threads can stand in each other's way in highly complicated ways. For example, three threads might wait for locks held by other threads in a cyclical fashion – a deadlock situation that can bring the system to a standstill. A particularly complex situation is when a thread t holds a lock ℓ and then is itself blocked; for example, because it currently does not have a PE assigned to it by the operating system or because it waits for the completion of an I/O operation. Thread t can then delay many other threads waiting to acquire lock ℓ. Therefore, there has been intensive work on algorithms that avoid such situations. A *non-blocking algorithm* avoids any kind of locking, i.e., no blocked thread can block another thread. A *lock-free algorithm* moreover guarantees that some thread in the system can always make progress towards reaching its overall goal. Finally, a *wait-free algorithm* guarantees progress by each thread. For more details and examples refer to a widely used textbook [233].

Open Problem 7 (Scalable concurrent algorithms with or without locks)
Many lock-free algorithms seem to be so complicated that papers on them only discuss their correctness. However, we also would like to know their scalability when running on p PEs (i.e., hardware threads). There are very few results in

Draft

this direction and solving such problems might be an important step to better concurrent algorithms. In particular, many current lock-free algorithms have severe scalability bottlenecks. An example on priority queues can be found in Section 12.5.2.[11] If we use an asynchronous PRAM model, we can also exploit assumptions that do not hold for general lock-free algorithms. In particular, each thread t gets permanently assigned a PE. Thus, we can often *prove* that t will only hold a lock for a constant amount of time. Therefore, locking does not necessarily stand in the way of progress guarantees.[12]

2.4.5 The Work-Span Model

A more abstract way to look at shared-memory computations is the *work-span model* (aka *work-depth model*) where we abstract from the actual number of processors [4, 75]. We only look (1): at the total *work* W performed by a computation, i.e., the time needed for machine operations that are part of the actual computation, and (2): at its *span* T_∞(aka *depth*), i.e., the longest sequence of dependent operations within the computation. Roughly, the span is the time needed when an infinite number of processors is available. The computation forms a graph of dependencies that is unfolded using *fork operations* that spawn additional threads and atomic shared-memory operations. The model comes in different variants depending on what exactly these operations can do. In this book, the default will be binary forking [75] together with the aCRQW model from Section 2.4.2 and the atomic operations from Section 2.4.3. In particular, this allows parallel recursion. Figure 2.13 gives an example. Circuit models (see Section 2.8) take a similar, more low-level and hardware-oriented view of computations.

Exercise 12 *Show that function sumArray in Figure 2.13 has work* $\mathrm{O}(n)$ *and span* $\mathrm{O}(\log n)$. *Implement it using a system supporting task creation. Now tune it, e.g. by using a larger base case.*

An obvious lower bound for the time needed to execute a computation in the work-span model on a PRAM with p PEs is

$$T(p) = \frac{W}{p} + T_\infty.$$

[11]The underlying paper [480] also gives a curious example of an algorithm that uses locks but is nevertheless wait-free because it never waits for a lock.

[12]In practice, we may have to ensure that the operating system indeed never takes away the assigned hardware thread, e.g., by reserving one or several cores for the operating system. We also have to be careful about threads that perform I/Os.

Draft

```
**Function** $sumArray(a : Array, i, j)$                    // compute $\sum_{k=i}^{j} a[k]$
    **if** $i = j$ **then return** $a[i]$
    **else return** $sumArray(a, i, \lfloor \frac{i+j}{2} \rfloor) \parallel sumArray(a, \lfloor \frac{i+j}{2} \rfloor + 1, j)$
```

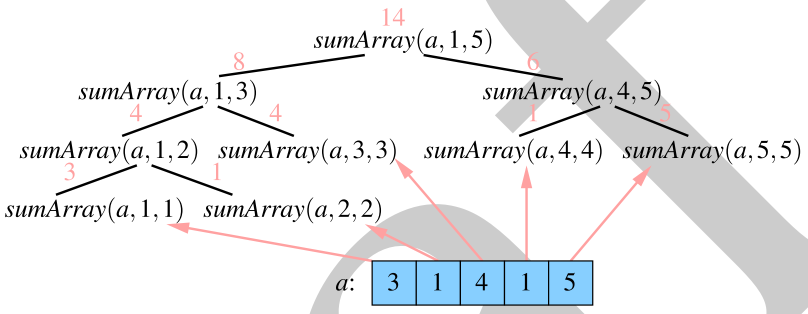


Figure 2.13: A function using parallel recursion to compute the sum of the elements of an array. The picture gives an example for the array $[3, 1, 4, 1, 5]$.

A crucial result is that using appropriate load balancers, this is also an upper bound at least in a probabilistic sense using randomization in a work-stealing load balancer [78] (see also [410, Section 14.5, 14.6]).

2.4.6 Parallel Memory Hierarchies

The Parallel External Memory (PEM) model [26] is a natural extension of PRAMs. RAMs with fast memory of size M each are attached to a large shared memory. As

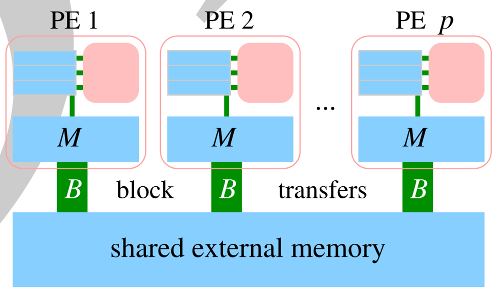


Figure 2.14: The parallel external memory model (PEM).

Draft

in the I/O model, access to that memory is in blocks of size B. As with PRAMs, we can consider several variants with respect to what kind of concurrent accesses are supported. PEM is an elegant way to model the differences between the cost of local and global memory accesses.

Exercise 13 *Describe an algorithm that transposes an $n \times n$ matrix on a PEM using $\mathrm{O}\left(n^2/Bp\right)$ I/O steps assuming $M > B^2$.*

However, as with basic PRAMs, the PEM model is unrealistic with respect to the cost of synchronization and contention. To handle this issue, let us introduce an asynchronous variant with queued writing, *PEM*$^+$, that adds block access to the aCRQW PRAM model from Section 2.4.2. One can now also consider atomic operations on the block level. Unfortunately, current processor architectures only support atomic operations on machine words (and sometimes on double words). On the other hand, hardware transactional memory (see also Section 2.4.3) operates on cache lines. Hence, let us assume that PEM$^+$ also supports transactions.

One issue with the PEM model that has led to confusion is that Vitter and Shriver [467] had previously introduced a parallel external memory model. But for them, M is the *overall* size of the fast memory. This implies that the bounds based on the original model [467] (e.g., [387]) are harder to obtain than bounds in the PEM model and require more sophisticated algorithms that treat communication and I/Os separately.

The PEM model is a good abstraction of a shared-memory machine where the PEs have a private cache and symmetric access to the main memory. However, many practical machines have a more complicated hierarchical structure that can be approximated by a tree of PEs. Level i of the memory is partitioned into k_i pieces and a subtree of p/k_i PEs share access to that piece. Consider a fictitious but realistic example: Four hardware threads of a core may share an L1 cache. Six cores on a *chiplet* may share an L2 cache. Three chiplets may share an L3 cache on a processor *socket*. Two sockets may have shared access to the main memory. Figure 2.15 illustrates this example. Some of the aspects shown in Figure 2.1 can also be mapped to this model. In that example, $k_1 = k_2$, i.e., each core has its own L1 and L2 cache. This can make sense in practice since these two cache levels can reflect different tradeoffs between latency, size, and cost per bit. Similarly, SSDs and main memory may both be attached to sockets although they differ in size, cost, latency, and volatility.

Section 2.5.5 further generalizes the hierarchical model to allow horizontal communication on each level. In practice, this is also relevant in a shared-memory

Draft

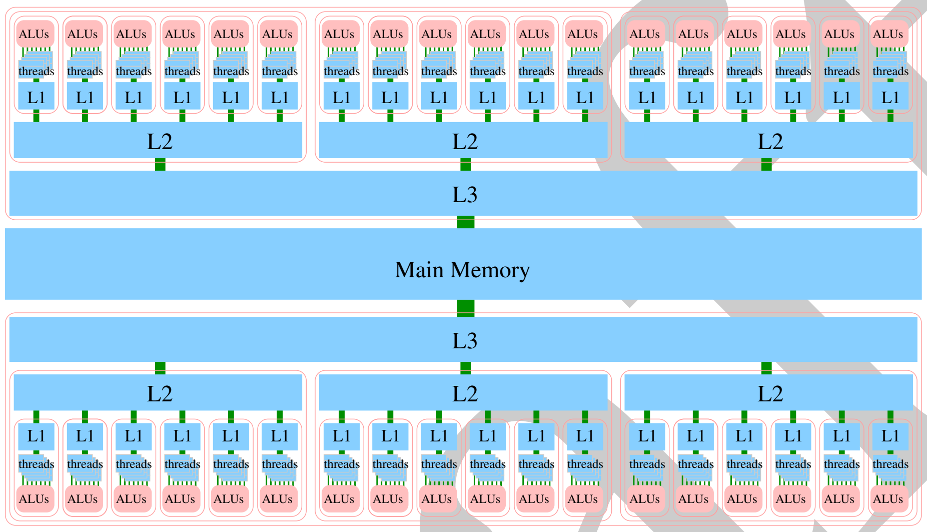


Figure 2.15: Example of a shared-memory machine with $p = 144$ threads, $k_1 = 36$ cores, $k_2 = 6$ chiplets, and $k_3 = 2$ sockets.

setting. In particular, each socket may be directly connected to several main memory modules. Every core can still access every memory module also on remote sockets. However, the access costs will be larger the "farther away" the memory module is. This issue is known as *non-uniform memory access (NUMA)*. Sockets[13] are therefore also called *NUMA nodes*.

Of course, completely analyzing algorithms in such a complex model is even more forbidding than multilevel external memory. However, we can adopt the approach to analyze a selected aspect like the number of I/Os on a particular level that experiments may indicate as a bottleneck. We can also generalize the approach of cache-oblivious algorithms to parallel memory hierarchies [118].

2.4.7 Graphics Processing Units (GPUs), Accelerators, etc.

The transition of traditional general-purpose sequential processors to multi-core processors has been a fairly conservative process leading to processors supporting a moderate number of parallel threads, favoring fast individual cores with

[13] Or any part of a machine that has uniform access to a set of memory modules, e.g., a chiplet within a multi-chip module.

Draft

large cache memories and large main memories. It turned out that this is not the ideal path to maximum peak arithmetic performance or maximum memory bandwidth. A more radical approach is to support massive parallelism even on a single chip dedicating a larger fraction of chip area to arithmetic units and to connect it to memory chips with a different tradeoff between bandwidth, latency, and cost. Figure 2.16 illustrates this tradeoff. Originally fueled by the large market for hardware accelerated computer graphics (e.g., for games), so-called *graphics processing units (GPUs)* have been particularly successful. Since the beginning of the millennium, they developed from special purpose processors to general purpose processors (*GPGPUs*) that are less and less different from traditional CPUs. GPU applications are now coming from a much wider range, with machine learning and mining crypto-currencies as major driving forces. In high-performance computing, the biggest machines currently invest heavily in GPU hardware and thus exert major pressure on application developers to use GPUs for more and more applications.

Exercise 14 *Compare current high-end CPUs and GPUs of leading manufacturers in a table. Possible rows of the table could be price, number of transistors, thermal design power, cache sizes, maximal attached memory, memory bandwidth, peak arithmetic performance for various number systems, and performance for some basic tasks. It would be particularly interesting to also include performance for basic operations that are not the core domain of GPUs, e.g., sorting, hash table access,.... Interpreting this data might involve normalizing performance in relation to price or power consumption.*

This has not been the only development that makes the landscape of computer architecture more heterogeneous. Processors now routinely contain cores that have the same instruction set architecture but different microarchitectures. Typically a mix of high-performance cores with slower but more energy-efficient cores. There are also special units for generating random numbers, for performing cryptographic operations, for data (de)compression, etc. Highly interesting is also reconfigurable hardware (field-programmable gate arrays, FPGAs).

Heterogeneous computing opens up interesting algorithmic questions concerning the scheduling of computations on heterogeneous resources. However, from the point of view of abstract models of computation, we have to be careful to keep the model simple. Therefore, it seems a good approximation to decompose a computation into parts where each part may have a different kind of processing unit that can handle it best. We can then analyze each part in a model appropriate

Draft

for this processing unit. We also identify bottlenecks, i.e., parts of the computation that may dominate cost for the available hardware. In particular, parts that are not bottlenecks can then also be executed on hardware that is not ideally suited for it. For example, a numerical computation might require some preprocessing that works best on the CPU followed by an expensive computation that works best on the GPU. Then we might decide to perform the preprocessing on the GPU anyway in order to keep the program structure simple and to save data transfer costs between CPU and GPU. In this case, the simplicity of the GPU preprocessing may be more important than an efficient implementation. With this process, we will often arrive at an implementation that works on simple homogeneous hardware or that has a small number of stages working on different hardware. Further tuning by offloading parts of the computation to inappropriate yet idle hardware is a possibility but often not worth the effort.[14] It certainly does not warrant a full-fledged heterogenous model of computation. In summary, heterogeneous computing is a reality that we have to face. On the other hand, overtuning to use all parts of such hardware can be considered an anti-pattern of software engineering.

We still need models for the different components of a heterogeneous system. Here simplicity is again an important guiding principle. For example, high-performance cores and efficiency cores in a modern processor still fit into the general models for shared-memory computing. We only have to be careful not to assume that all cores have the same speed. Note that this is not even guaranteed on homogeneous cores. For example, a recent high-intensity computation might have forced one core to reduce its clock frequency due to overheating. Often basic load balancing techniques will be able to handle this situation. Let us have a closer look at GPUs to consider a more interesting example:

Modeling GPUs. Modeling GPUs is an open problem because different vendors have different architectures that also change from generation to generation. Even a single architecture generation contains different compute units, e.g., for general-purpose processing, tensor processing, ray tracing, texture mapping, or video encoding. Let us make the case for maximal simplicity here. Our general discussion of heterogeneity implies here that we may focus on one stage of an application that has one type of computing unit that is most useful. Often, these will be the general compute units. The main difference between these units to classical CPU cores is quantitative. For example, an NVidia RTX 5000 Ada GPU contains

[14]For example, suppose we use a GPU with 1PetaFlop peak performance to perform a matrix multiplication. Offloading part of this computation to CPU that can do 200 GFlops, i.e., less than a per mille of that work, will not be very helpful and may actually harm energy consumption.

Draft

12800 CUDA cores, 100 times as many as a contemporary high-end server CPU. On the other hand, it has only 32 GByte of memory while a server CPU can address several terabytes of memory. Some server CPUs have up to 1152MByte of cache memory while the RTX 5000 GPU has only 72MByte (already up by an order of magnitude from the previous generation). These differences in numbers already by themselves imply a significantly different programming style and spectrum of applications without the need for different models of computation.

There are further architectural differences that may warrant a GPU model of computation. However, there also seems to be some convergence in the development of CPU and GPU architecture. For example, a major difference between GPUs to CPUs is that there are groups of GPU threads that share the same stream of machine instructions (called a *warp* in NVidia CUDA). However, threads in a warp acquired increasing autonomy in subsequent generations of architecture. On the other hand, *SIMD* instructions are gaining increasing importance in CPU architectures. These work on registers that contain multiple machine words (currently up to 512 bits), applying the same operation to multiple pieces of these registers (see also Section 2.2.5). The effect is quite similar and one might imagine that a CPU investing heavily in SIMD units with otherwise lightweight cores and small caches might have similar tradeoffs as a GPU.

Another important feature of the CUDA programming model is that there is a hierarchy of threads – one or multiple kernels executing a grid of thread blocks, which are themselves partitioned into warps. Optionally, there is an additional level of thread-block clusters. Threads within the same block can interact via on-chip shared memory. Note that CPUs also have some hierarchy reflecting the hardware; see Section 2.4.6. Hence, overall we are undecided. GPU and CPU programs are sufficiently different to make it likely that different models may help to understand this more abstractly. On the other hand, architectural convergence and the strife for simplicity suggest that we can use the same shared memory models and keep in mind that different key parameters like the number of cores and cache sizes can have big impacts even within the same general model.

Exercise 15 *Compare existing programming models for GPUs like CUDA, OpenCL, or Vulkan. Identify commonalities and differences between the models. Where do these models deviate from models that are compatible with CPU programming?*

Open Problem 8 (Modeling GPUs) Develop a simple and useful model for GPU processing that works for the most important vendors and is stable over multiple architecture generations.

Draft

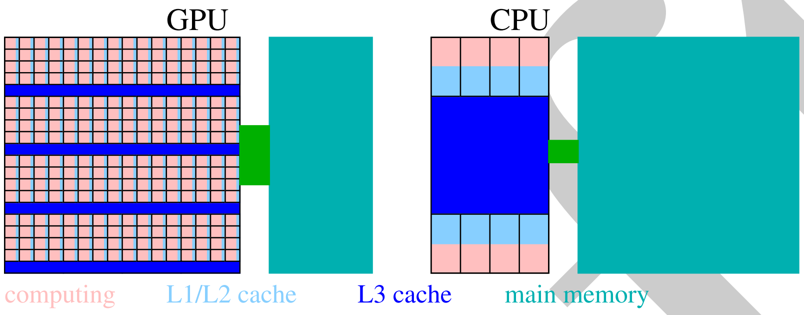


Figure 2.16: Illustration of the relative areas invested into computing and memory resources in GPUs (left) and CPUs (right).

2.5 Distributed Models

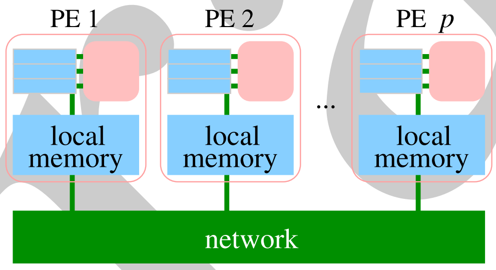


Figure 2.17: A distributed-memory parallel machine.

A simple and highly scalable way to build parallel computers is to connect multiple smaller computers (*nodes* – perhaps themselves shared-memory parallel machines) by a communication network. Algorithms working on such machines then perform local computations and communication. While there is a large variety of concrete communication operations, these are usually implemented using point-to-point communication of messages, where one node sends a message to a receiver node. Alternatively, the system may support access to blocks of the memory at remote nodes. However, from the algorithmic point of view, this makes little difference, so we stick to the message view here.[15]

[15]Some systems also have hardware support for some of the collective operations from Chapter 19. Radio networks and Ethernet support broadcast to a set of neighboring nodes.

Draft

How can we model communication cost? Let us consider a simple situation first, where two nodes exchange messages of length ℓ and the network allows direct communication between these two nodes. To eliminate synchronization delays, also assume a situation where messages go back and forth repeatedly (the *ping-pong benchmark* [238]). The average time per message transfer will then have a constant offset – the startup overhead – plus a term that depends on ℓ. The startup overhead is mostly due to the underlying software and thus highly dependent on which message-passing software is actually used. The latter term might be assumed to be approximately linear in ℓ – see also Section 2.5.1 below. The function can also have steps in it because long messages are likely to be chopped into packets of fixed size. Moreover, different protocols might be used for different ranges of message sizes. So, macroscopically, we might expect a piecewise linear function where each sub-function has small steps.

Exercise 16 *Implement the ping-pong benchmark on your system, e.g., using MPI. Running two processes on different machines, make detailed measurements of the round-trip time as a function of message length. Make multiple repetitions and measure average time as well as variance and outliers. Discuss possible explanations for deviations from the simple linear model ($\alpha + \ell\beta$).*

In more general situations, things get much more complicated. Communication cost can depend on where the nodes are in the network, how the communication is delayed by traffic between other nodes (*congestion*), and what the receiver is currently doing (for example there might be long queues of messages currently waiting to be processed and complicated rules for matching messages that the receiver is willing to process). The startup latency may be very high if the nodes communicate for the first time. This is because communication often requires a data structure for a communication channel between pairs of nodes. This data structure needs to be initialized when a node pair communicates for the first time. On large systems, there may also be insufficient space for communication channels between all pairs of nodes so that channels can also be deallocated. Even if the network and the communication software behave in a predictable way, running time analysis of an application based on message passing can be complicated by synchronization delays. For example, an operation that receives a message has to wait for the matching message to be sent.

***Exercise 17** *In continuation of Exercise 16, now consider a situation where p processes are matched up in pairs. Concretely, for even p, consider a pseudo-*

Draft

random permutation (see Section 10.1.8) π and let PE $\pi(2i)$ communicate with PE $\pi(2i+1)$. How does the speed of communication change with p? Are there increased fluctuations? Discuss. How does the speed change again if you change π after every iteration (or after a small number of repetitions) with a barrier synchronization (see, e.g., [410, Section 13.4.2]) in between?

Machine models for distributed-memory parallel computing cannot grasp all these complex details. Hence, simplifications are necessary and naturally lead to a wide spectrum of possible models. We describe some of the most popular ones below. Also, it should be noted that modern nodes of distributed-memory machines are shared-memory parallel machines themselves. Section 2.5.5 discusses how to handle such hierarchical systems.

2.5.1 Point-to-Point Fully-Connected Communication

The standard model for distributed-memory parallel machines in this book (the *point-to-point model*) is very simple. All p PEs are connected by a network. A PE can send a message of size ℓ to any other PE in time $\alpha+\beta\ell$ [185, 112, 410]. Here, α is the *startup overhead* and β controls the bandwidth of communication. For simpler asymptotic algorithm analysis, the parameters α and β are sometimes treated as constants. In that case, we often get very similar complexities as in PRAM algorithms. In particular, via PRAM emulation [389], one can see that the slowdown compared to PRAM is at most a logarithmic factor.[16]

However, $\alpha \gg \beta$ in many situations and because also the communication bandwidth can be much lower than the local rate of computation,[17] we also sometimes treat the parameters as variables in asymptotic analysis.

Open Problem 9 (Scalable middleware)
As pointed out above, current systems software is sometimes far away from the simple point-to-point model discussed in this section and implicitly assumed in countless programs. Better approximating it in large systems would greatly simplify parallel software development. Concrete problems involve, for example, avoiding costly and unpredictable connection setup overhead and reducing message matching overhead when many messages arrive.

[16] Roughly, we can resolve contention in accessing a memory cell by a tree combining/replicating the queried or accessed data.

[17] More precisely, $\beta \gg \gamma$ where γ is time needed to execute a (typical) machine instruction.

Draft

Open Problem 10 (Low latency communication hardware and middleware) There is no physical reason why the startup latency α should be orders of magnitude larger than the time per machine word β . Indeed, there are many efforts in that direction but it seems that more can be achieved. What might help is a model where threads do not inject single messages into the system but batches of messages.

Communication in our variant of the model is assumed to be *single-ported*; that is, a PE can only participate in a single communication at a time. More precisely, we assume the *full-duplex* model where a PE can at the same time send one message and receive another one – potentially involving two different communication partners. See Section 2.5.1.2 for variants of this aspect. A PE has full control over when it sends messages. However, the asynchronous nature of distributed-memory computing implies that it cannot directly control when messages arrive. Hence, our cost model has to take into account contention on the receiver side. We use a queuing model similar to the aCRQW PRAM model discussed in Section 2.4.2. When a message is sent to PE i, it is appended to a queue of messages already waiting for delivery to PE i. Thus it will experience a delay corresponding to the sum of the communication costs of the previously queued messages. The sender may or may not experience this delay or part of it.

2.5.1.1 Synchronization by Communication

PEs can coordinate by communication operations not only due to the information contained in the messages but also because they imply synchronization. In particular, receiving a message m implies that m has been sent by some PE i, which in turn implies something about the state of the computation at PE i. Practical communication libraries such as MPI also offer several variants of synchronization by communication. The variants we describe above use the basic semantics of the MPI send operation, where the call returns when the memory areas containing the message can be overwritten again. There is also a synchronous send operation where additionally the matching receive operations must have started executing. Furthermore, there are asynchronous send and receive operations and ways to check whether messages have arrived or whether asynchronous operations have finished.

Exercise 18 *Implement the operation barrier that synchronizes all processors using point-to-point communication. The operation should take time* $\mathrm{O}(\alpha \log p)$.

Draft

Hint: if you are out of ideas, check [410, Section 13.4.2].

2.5.1.2 What Communications Can Be Done in Parallel?

We can distinguish between several variants of the point-to-point model where it may or may not be allowed to receive a message at the same time as sending a message:

half-duplex: At any point in time, each PE can either send or receive a message.

full-duplex: Each PE can send *and* receive one message concurrently.

postal model: A restricted form of full-duplex where concurrent send and receive is only allowed if the communications partners are the same; that is, if PE i sends to PE j, it may only concurrently receive from j.

Multiported variants where $K > 1$ concurrent communication partners are allowed have also been considered [46, 456] and are sometimes supported by hardware.

In this book, we mostly assume the full-duplex model because coordinating the PEs to obey the constraints of the half-duplex model is more complicated than it may appear. In particular, while it is clear that the full-duplex model can be up to two times faster than the half-duplex model, there are cases where the difference is larger. For example, consider three PEs A, B, and C that want to communicate a message of size ℓ in a circular fashion, i.e., A sends to B, B sends to C, and C sends to A. This can be done in time $\alpha + \beta\ell$ in the full-duplex model. In the half-duplex model, it takes *three* times as long; see Figure 2.18.

Things get even more complex when we consider the more general problem of scheduling the exchange of a set of messages with uniform length (see also [410, Section 13.6]). It turns out that in the full-duplex case, this problem can be

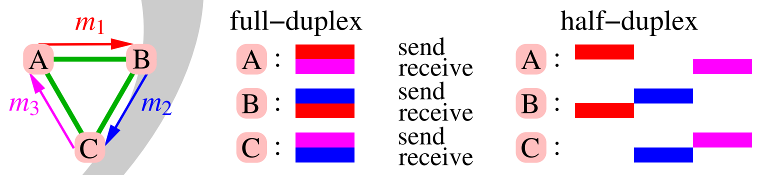


Figure 2.18: Communication of three messages in a cyclic pattern needs only one step with full-duplex communication yet three steps using half-duplex communication.

reduced to the problem of edge-coloring a bipartite multigraph. This problem can be solved in polynomial time and the number of steps (colors) needed is always the obvious lower bound given by the maximum number of messages sent or received (the maximum degree in this graph). There are also fast parallel algorithms for edge-coloring bipartite multigraphs. In contrast, the corresponding problem for the half-duplex model is edge-coloring a general multigraph. This problem is NP-hard and the number of steps may be up to $3/2$ times the number of messages sent or received. Indeed, it may be advantageous to send messages on detours to improve scheduling flexibility [415].

2.5.1.3 The LogP Model

A variant of the point-to-point model is the LogP model [128]. This model assumes constant size messages and has parameters L (*latency*), o (*overhead*), g (*gap*), and P (*processors*). Compared to the point-to-point model, we have $L = \alpha$ and $P = p$. The overhead $o \le L$ is the local processing time needed to send or receive a message. The gap g ($o \le g \le L$) is the time between consecutive message transmissions. Differentiating between latency, overhead, and gap allows a detailed analysis of the behavior of asynchronous algorithms. We prefer the point-to-point model here because it has one parameter less but nevertheless better grasps the important difference between short and long messages. Consequently, there is also a generalization of LogP that has an additional parameter G for modeling message lengths – LogGP [15].

2.5.2 Communication-Efficient Algorithms

An implicit assumption of the point-to-point model, LogP (and the BSP model in Section 2.5.3) is that the network can be arbitrarily scaled without affecting latency and communication bandwidth for point-to-point messages. Unfortunately, this assumption is overly optimistic in practice. For example, it can be shown that a 2D layout of a network supporting the required bandwidth on a chip requires area $\Omega(p^2)$ [459]. Thus, communication bandwidth is the limiting factor for large machines [19, 83]. In Section 2.5.4 we address this by explicitly modeling the network. However, this ties algorithm design to the specific network architecture at hand and thus makes algorithms more complicated and less portable. A more attractive approach is to stick to the point-to-point model but to have a closer look at the relationship between local work and communication costs. We say that an algorithm is *communication efficient* if its communication cost is *sublinear* in

Draft

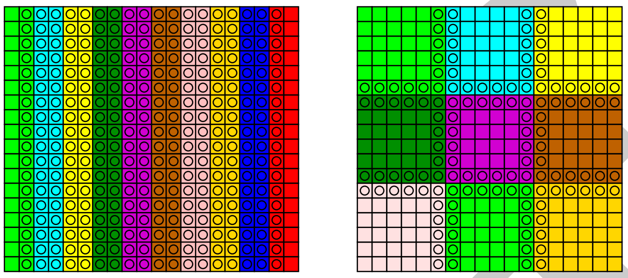


Figure 2.19: Domain decomposition for an 18×18 array of elements on 9 PEs (color coded). Left: using a one-dimensional logical topology. Right: using a 3×3 two-dimensional topology. The circles indicate boundary nodes that need to be exchanged in computations where an array entry depends on its neighbors. One can see that the two-dimensional approach is more scalable and more communication efficient.

the local work [412]. In particular, the *bottleneck communication volume* is important, i.e., the maximum amount of data that some PE has to communicate. A simple but typical example of a communication-efficient algorithm is illustrated in Figure 2.19 where averaging adjacent entries of an array only requires few array elements to be communicated. Nevertheless, there has been surprisingly little work on communication-efficient algorithms, and therefore they have become a major theme in our research [244, 414, 409, 245, 192, 247, 248, 72]. An important exception is the work on *communication avoiding algorithms*, e.g., [439, 44]. These results optimize nested loops with a predictable and regular access pattern. This approach can be traced to a classical parallel algorithm for $n \times n$ matrix multiplication [136]. The idea is to partition the space of n^3 multiply-add operations into p cubes. This leads to an algorithm that requires bottleneck communication volume $\mathrm{O}\left(n^2/p^{2/3}\right)$ in contrast to volume $\mathrm{O}\left(n^2/p^{1/2}\right)$ required by methods that only partition the matrices. Note that this approach leads to algorithms that require space superlinear in the input size.

Communication efficiency can be analyzed in the context of any of the distributed-memory models discussed in this section.

2.5.3 Bulk Synchronous Parallel (BSP, BSP*, BSP$^+$, CGM) Models

So far we have discussed asynchronous models based on point-to-point communication. These are highly flexible and can be used to analyze a wide range of

Draft

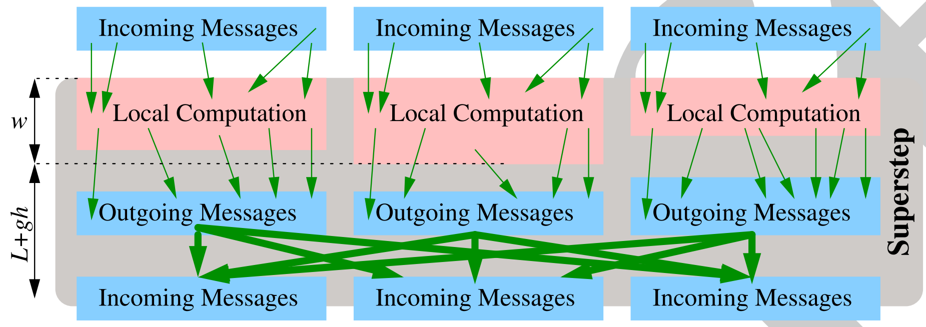


Figure 2.20: A superstep in the BSP model for $p = 3$ PEs. Parameter h is the maximum input or output buffer size produced by the superstep. For basic BSP this is measured in machine words; for BSP* and BSP$^+$ the unit is a packet of size B.

parallel algorithms. However, they are rather low-level. Everything has to be broken down into individual message transfers. In particular, analyzing synchronization and queuing delays can be quite complicated. The *bulk synchronous parallel (BSP)* model [463] addresses this issue by grouping many message exchanges into batches that are synchronized together. During a *superstep*, PEs can do independent local work and they can add arbitrary point-to-point messages to the message batch for that superstep. After the superstep, the messages in the batch are delivered and the PEs are globally synchronized. Let w denote the maximum amount of work done by any PE during a superstep, let h denote the maximum number of machine words sent or received by a PE in that batch. Then the time for the superstep in the BSP model is $w + L + hg$. The parameter L is called the *latency* for the superstep and g is called the *gap*. The problem of executing such a communication is also called an *h-relation*.

Comparing this to our point-to-point model, one is tempted to assume $L \approx \alpha$ and $g \approx \beta$. However, this is far too optimistic when messages are short. Rather, several interpretations are possible depending on the actual message exchange pattern or how the data delivery for a superstep is implemented. One could set $L \approx \alpha \log p$ to account for the fact that a barrier synchronization is performed (see also Section 19.1) and $g \approx \alpha$ conservatively assuming that each machine word is delivered as an individual message. This will be far away from the truth when

Draft

messages are actually long.[18] One could also set $L \approx \alpha p$ and $g \approx \beta$. This is a good approximation if $h \gg p$ or if each PE communicates with all (or most) other PEs. Otherwise, it is far too pessimistic. One could also implement the data exchange using the indirect all-to-all algorithm from Section 19.7.1. Then one gets $L = \mathrm{O}(\alpha \log p)$ and $g = \mathrm{O}(\beta \log p)$. A compromise could be to use a constant number of indirections as it is already used in high-performance sorting algorithms [39]. From this discussion we conclude that the BSP model can be rather inaccurate and that it is unclear how to set the parameters to get a reasonable approximation of reality.

Exercise 19 *Implement several ways to support a BSP superstep using a point-to-point system like MPI. Compare their performance for simple benchmarks. This is a good project for a course where each student contributes one superstep implementation or one benchmark.*

Bäumker and Meyer auf der Heide [52] made a similar observation and propose to slightly extend the BSP model. The *BSP* model* replaces machine words in the definition of h by *blocks* (or packets) of size B. The parameter B is chosen in such a way that sending messages of this size in the ping-pong benchmark achieves near-maximum bandwidth.[19] Using the terminology of the point-to-point model, this means $B = \Theta(\alpha/\beta)$. Let M_i denote the set of messages sent or received by PE i in a BSP* superstep. Then redefine h to be $\max_i \sum_{m \in M_i} \lceil |m|/B \rceil$. The gap g now defines the gap between sending *packets* of size B. The cost for a superstep in the BSP* model is then again $w + L + hg$.[20] BSP* is realistic insofar as an h-relation can be delivered in time $\mathrm{O}(\alpha \log p + \beta B h)$ in the point-to-point model and on some concrete interconnection networks, e.g., hypercubes. .

***Exercise 20** *Prove the proposition by outlining an algorithm for achieving this. Hint: use existing work on routing in hypercubes [288].*

Another problem of BSP (or BSP*) is that it needs time $\Omega(L \log p)$ for simple collective operations like broadcast and reduction. Since $L = \Omega(\alpha \log p)$ in or-

[18]This would be the right interpretation of BSP on systems where Open Problem 10 is solved.

[19]Bäumker and Meyer auf der Heide [52] formulate this slightly differently "The parameter B is [...] the size the messages must have in order to fully exploit the bandwidth of the router." However, taking this literally would lead to huge values of B in particular when the system software uses throughput optimizing protocols for very large messages. Indeed, in their experiments, they seem to use B closer to our interpretation.

[20]Bäumker et al. use $w + \max(L, gh)$. Disregarding constant factors, our definition is equivalent and sticks closer to the original BSP model.

der to support global barrier synchronization, we get time $\Omega(\alpha \log^2 p)$ for these operations. A possible further extension of BSP* is therefore to allow not only point-to-point messages but also broadcast, (all-)reduction, and prefix sums of size h without changing the definition of the cost of a superstep. This can be done without further increasing the time for a superstep. This has already been proposed in an early paper [236] and was (partially) implemented in the PUB library [81] from Paderborn University and in Apache Hama[21]. We will call this model *BSP$^+$*.

Open Problem 11 (Validating and benchmarking the BSP model) Our arguments above are based on the assumption that the point-to-point model is close to reality. However, it would be much better to thoroughly compare the models experimentally. Is BSP*/BSP$^+$ indeed closer to reality? One should try different applications and benchmark kernels. What are the right values for the parameters for different implementations of BSP and for different machines? Which implementations of message exchange work best for which applications? Can those be selected adaptively?

Open Problem 12 (Exploring the BSP* and BSP$^+$ models) So far, few algorithms have been designed or analyzed for the BSP* or BSP$^+$ model. Assuming these models are more realistic than plain BSP, how does this affect the landscape of algorithms for BSP-like models? Are there new algorithms that now work better? Does that transfer into practice? Are tradeoffs between local work, communication volume, and latency changing? It could happen that a spectrum of BSP algorithms with different such tradeoffs collapses into a smaller set of (possibly simpler) BSP* or BSP$^+$ algorithms.

We can also consider *specializations* of the BSP model. The *coarse-grained multicomputer (CGM)* [135] model assumes that an input of size n is evenly distributed over all PEs and that $h = n/p$ in every superstep, i.e., all the data is always communicated. This has the advantage that for sufficiently large n, the latency term is dwarfed by the term $gh = gn/p$ and that, essentially, we only need to count supersteps to determine the communication cost. Note that this goes in the opposite direction of the communication-efficient algorithms discussed in Section 2.5.1 and thus may not lead to scalable algorithms in practice$^{\mathrm{DS}}$. Still, the simplicity of the CGM model has attracted a significant number of publications using it [133, 134].

[21] hama.apache.org/hama_bsp_tutorial.html, accessed Mar 6, 2023.

Draft

2.5.4 Networks with Known Structure

When you build a distributed-memory computer, you are more or less free to choose the structure of its interconnection network. Of course, there is a cost-performance tradeoff. Consequently, there has been a lot of work on networks with good tradeoffs and on algorithms that exploit these structures. Moreover, in the early days of parallel computers, there was little support for routing: messages could only be exchanged between neighbors in the network. Even with routing support, fitting your algorithm to the network structure can improve communication performance. In particular, this can avoid *contention*, i.e., the situation that several messages compete for the same resource (e.g., a router node or network link). Tuning an algorithm for a particular network is nowadays done rarely. However, some of the network structures are so elegant that algorithms for them can be useful, even on different networks. This is also a reason for knowing some of the networks presented below even if you are unlikely to face them on a real machine – often the best known algorithms make use of a logical network that is then embedded into the physical network (in the case of the fully-connected model from Section 2.5.1, any mapping will do).

The cost for a message exchange can adapt the point-to-point model from Section 2.5.1 – A PE can send a message of size ℓ to any *neighboring* PE in time $\alpha + \beta\ell$. When the latency of a hardware router dwarfs the software setup latency, we can generalize this to allow arbitrary communication partners *provided that the routing algorithm can find contention-free paths*. Note that the latter constraint can be difficult to check since one needs detailed knowledge about the routing algorithm.

Open Problem 13 (Networks reconsidered) Most work on network-specific algorithms and routing is more than 25 years old (at least from the algorithms community). Are all these problems really solved? Are there new opportunities or new problems that might warrant reconsidering some of these questions? Our observation is that at least routing algorithms should still be interesting since we can now consider much more complex computations in the network and because networks used in practice show massive performance problems (e.g., [36, Appendix B.1]).

Several properties are important for good interconnection networks of parallel machines. Figure 2.21 illustrates these parameters.

Simplicity. A simple mathematical structure helps to design algorithm for a network and often also simplifies building the network.

Draft

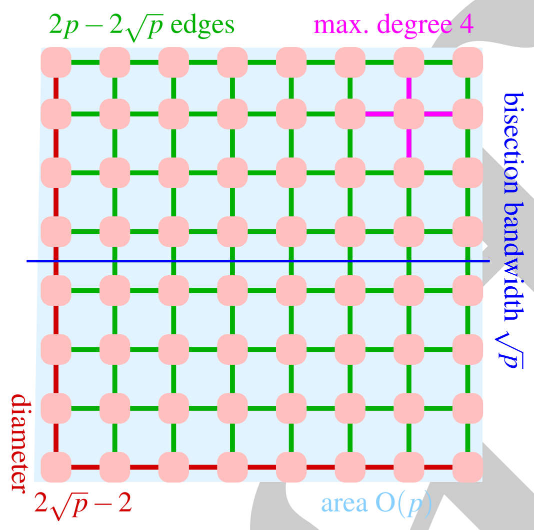


Figure 2.21: Illustration of the parameters maximum degree, number of edges, diameter, bisection bandwidth, and 2D layout cost (area) for a mesh network.

Node degrees. To keep costs low, one would like to have maximum node degrees bounded by a (small) constant. For simplicity, it is also desirable to have uniform node degrees.

Number of edges. Another cost criterion is to have a low number of connections, ideally, $m = \mathrm{O}(n)$.

Diameter. The network graph should have a low diameter in order to allow fast coordination of the PEs. Typically one would like to achieve $\mathrm{O}(\log p)$. Lower diameter would not help in presence of single-ported communication and it would increase node degrees and costs.

Bisection bandwidth. This is the minimum number of links one has to cut in order to partition the network into two halves with at least $\lfloor p/2 \rfloor$ nodes each. High bisection bandwidth means that we can support high bandwidth communication for arbitrary communication patterns. Bandwidth $\Omega(p)$ (i.e., *linear*) is needed in order to fully support the point-to-point model from Section 2.5.1. Many networks actually do not achieve this bandwidth. Computing the bisection bandwidth is NP-hard in general [198]. An upper bound is easy to establish by evaluating the cut size for a particular bisection. A lower bound can be given by embedding the

Draft

complete (clique) graph into the network [288, Theorem 1.21].

Layout costs. One would like to be able to put the components of the network into 3D space (or even better on a planar chip) such that connections are short and have fixed width. Obviously, there is a fundamental physical tradeoff here. We discuss this tradeoff in more detail in Section 2.8.2.

We now describe a few networks that are interesting for parallel computers, starting with simple cheap ones progressing to more and more powerful ones. Table 2.5 summarizes their properties.

****Exercise 21** *Show the values for area, volume, and longest edge for d-D meshes and tori in Table 2.5. Hint: use induction on d.*

2.5.4.1 Paths and Rings

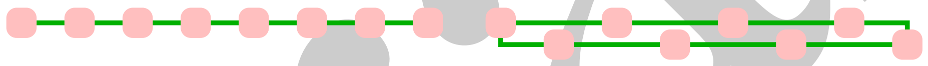


Figure 2.22: A path and a ring network with 8 PEs.

Paths and rings are unbeatable in cost and simplicity. A path has a maximum degree of two and one achieves constant-length wires by just putting PEs next to each other. Figure 2.22 shows that one can also lay out rings with constant wire length. Compared to paths, the one additional edge pays off by about halving the diameter and improving simplicity – we now have uniform degree two. It now also suffices to have *unidirectional links*. Ring and path networks are therefore popular for designing small networks. For larger configurations, their small bisection bandwidth makes them less attractive. Many algorithms naturally work on a path or ring. For example, by assigning slices of an array to each node, computations where only neighboring cells of the array interact can be implemented using communications between neighbors only.

2.5.4.2 Trees

Compared to paths, (balanced binary) trees are attractive because, by increasing the maximum degree by one, we can reduce the diameter from linear to logarithmic. Trees naturally lend themselves to implementing divide-and-conquer algorithms. Since bisection bandwidth does not go up, trees are not a good general-

Draft

Table 2.5: Key properties of networks commonly used in parallel computing for connecting p PEs. Abbreviations: “degree” is the maximum degree, “bisection” is the bisection bandwidth, “length” is the maximum wire length in a 3D layout, “area/volume” is the layout complexity in 2D/3D. The columns “bisection” to ”volume” have an implicit $\mathrm{O}(\cdot)$. The columns “diameter” and “#edges” have an implicit $\cdot(1+o(1))$.

name	diameter	degree	#edges	bisection	length	area	volume	section
path	p	2	p	1	1	p	p	2.5.4.1
ring	$p/2$	2	p	1	1	p	p	2.5.4.1
binary tree	$\log p$	3	p	1	$p^{1/3}$	p	p	2.5.4.2
r-ary tree	$\log_r p$	$r+1$	p	1	$p^{1/3}$	p	p	2.5.4.2
star	1	$p-1$	p	p	$p^{1/3}$	p	p	2.5.4.3
mesh	$2p^{1/2}$	4	$2p$	$p^{1/2}$	1	p	p	2.5.4.4
torus	$p^{1/2}$	4	$2p$	$p^{1/2}$	1	p	p	2.5.4.4
3D mesh	$3p^{1/3}$	6	$3p$	$p^{2/3}$	1	$p^{4/3}$	p	2.5.4.5
3D torus	$3p^{1/3}/2$	6	$3p$	$p^{2/3}$	1	$p^{4/3}$	p	2.5.4.5
d-D mesh	$dp^{1/d}$	$2d$	dp	$p^{1-1/d}$	$p^{(1-3/d)/2}$	$p^{2-2/d}$	$p^{(3-3/d)/2}$	2.5.4.6
d-D torus	$dp^{1/d}/2$	$2d$	dp	$p^{1-1/d}$	$p^{(1-3/d)/2}$	$p^{2-2/d}$	$p^{(3-3/d)/2}$	2.5.4.6
hypercube	$\log p$	$\log p$	$p\log(p)/2$	p	$p^{1/2}$	p^2	$p^{3/2}$	2.5.4.7
CCC	$2\log p$	3	$3p/2$	$p/\log p$	$p^{1/2}$	p^2	$p^{3/2}$	2.5.4.8
butterfly	$\log p$	4	$2p$	$p/\log p$	$p^{1/2}$	p^2	$p^{3/2}$	2.5.4.8
multistage butterfly	$\log p$	4	$2p\log p$	p	$p^{1/2}$	p^2	$p^{3/2}$	2.5.4.8
fat tree	$\mathrm{O}(\log p)$	flexible	$\mathrm{O}(p\log p)$	p	$p^{1/2}$	p^2	$p^{3/2}$	2.5.4.10
expander graph	$\mathrm{O}(\log p)$	$\mathrm{O}(1)$	$\mathrm{O}(p)$	p	$p^{1/2}$	p^2	$p^{3/2}$	2.5.4.11

Draft

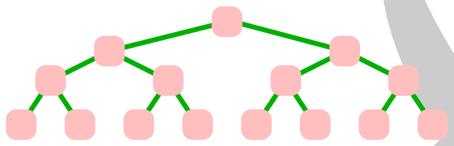


Figure 2.23: A 15-node complete binary tree network with depth 3.

purpose network for large parallel computers (but stay tuned for *fat trees* introduced in Section 2.5.4.10). However, tree networks have been combined with other networks to give hardware support for very low-latency collective communication operations. This is one example of why it can make sense when parallel computers have several different networks.

2.5.4.3 Stars

An extreme tree is a single central node with $p-1$ other nodes directly connected to it. We mention this as a kind of "anti-pattern" for parallel computing. Many simple parallel programs use this as a logical structure, for example, the master-worker load balancing schema (see [410, Section 14.3]). Such programs are simple but with obvious limitations for scalability. More surprisingly, the star structure is hardwired into several theoretical models for distributed computing (see also Open Problem 21 and Section 2.5.8).

2.5.4.4 Meshes and Tori

Arranging p nodes in a rectangular grid with side lengths $\Theta(\sqrt{p})$ leads to a network with diameter and bisection bandwidth $\Theta(\sqrt{p})$ that can be laid out with linear area in the plane, with constant wire lengths and $\approx 2n$ edges; see Figure 2.24. Using an analogous trick as with the ring network, this also transfers to *torus networks* where nodes on one border of the rectangle are connected with the node on the opposite side. Usually one uses links in four orthogonal directions, leading to a maximum degree of 4. Sometimes also diagonal connections are present, resulting in a maximum degree of 8. Unidirectional links make sense with torus networks. Many algorithms have been designed specifically for mesh and torus networks. For example, two-dimensional arrays can easily be mapped to a mesh by cutting it into rectangular pieces (*domain decomposition*). Figure 2.19 gives a simple example. An interesting nontrivial example is Cannon's algorithm for par-

Draft

allel matrix multiplication [103] where square tiles of a matrix are rotated through a torus network.

Exercise 22 *Design triangular and hexagonal variants of mesh networks. Analyze their properties and discuss advantages and disadvantages compared to the more common meshes we consider here.*[22]

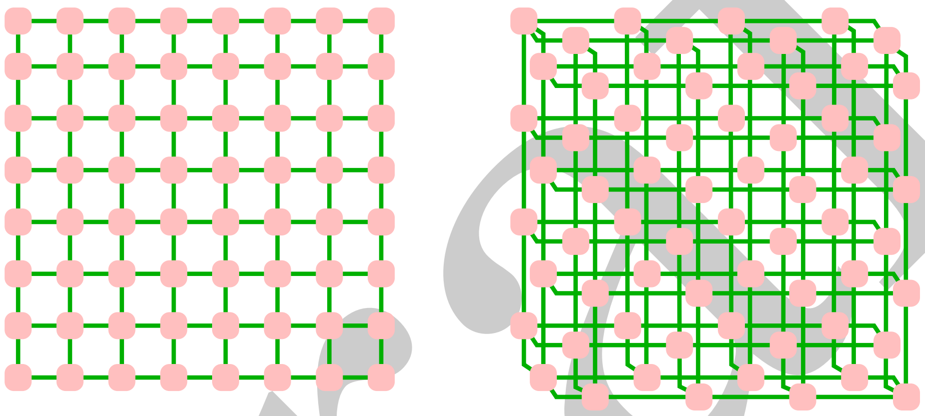


Figure 2.24: An 8×8 mesh (left) and torus (right) network.

2.5.4.5 3D Meshes and Tori

The step from lines or rings to 2D meshes or tori can be extended to 3 dimensions. By going to maximum degree 6 and $\approx 3n$ edges, we can reduce the diameter to $O(p^{1/3})$ and increase bisection bandwidth to $p^{2/3}$ while keeping constant wire length if we can arrange our PEs in 3 dimensions. Thus, this architecture looks very natural for obtaining a massively scalable network for large parallel computers.[23] With routing done by hardware, the latency might not be worse than for the logarithmic diameter networks discussed below. The same applies to the actual bisection bandwidth – for a 3D mesh, we can afford much higher-bandwidth physical connections than for the long connections in the networks below. Once more, the application to domain decomposition is important. In particular, 3D meshes

[22] An interesting application of triangular meshes is in some state-of-the-art climate and weather models. [205].

[23] With a caveat with respect to cooling – see Section 2.8.2.

Draft

are promising for simulating physical objects where 3 dimensions are what we are facing most of the time. It should be noted though, that the graph used for the computations may have a different shape than the interconnection network so that some overheads for embedding the computational structure into the physical network might ensue.

2.5.4.6 Higher Dimensional Meshes and Tori

The transition 1D→2D→3D can be logically continued. For constant dimension d, we obtain diameter $\mathrm{O}(p^{1/d})$ and bisection bandwidth $p^{d/(d-1)}$. However, since we are running out of (usable) physical dimensions, wire lengths and layout costs go up. Nevertheless, such networks have been used for supercomputers because with their fixed maximum degree, they can be scaled to different machine sizes without fundamentally changing the network topology. For example, IBM Blue-Gene/Q used a 5D torus.

2.5.4.7 Hypercubes

When $p = 2^d$ is a power of two, we can lead the above process to its logical conclusion by considering a mesh with side length two and d dimensions – a *d-dimensional hypercube*. We interpret PE numbers from $0..p-1$ for $p = 2^d$ as d-digit binary vectors. Then, PEs i and j are connected if and only if $i \oplus j = 2^k$ for some k in $0..d-1$. We say that i and j are connected along dimension k of the hypercube. See Figure 2.25 for an example. We obtain logarithmic diameter and linear bisection bandwidth at the considerable cost of logarithmic node degrees and $n \log n$ edges. Hypercubes are sometimes used for building actual computers. An influential historical example was the first Connection Machines [237] which

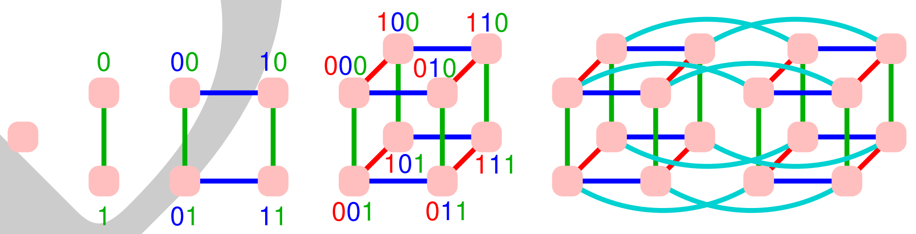


Figure 2.25: 0- to 4-dimensional hypercube networks.

Draft

used a 16-dimensional hypercube.[24]

Many divide-and-conquer algorithms can be mapped very naturally to a hypercube. For example, several of the collective-communication problems have very simple, optimal solutions on hypercubes (see [410, Chapter 13]).

2.5.4.8 Constant Degree Networks with Logarithmic Diameter and Bisection Bandwidth $\Theta(p/\log p)$

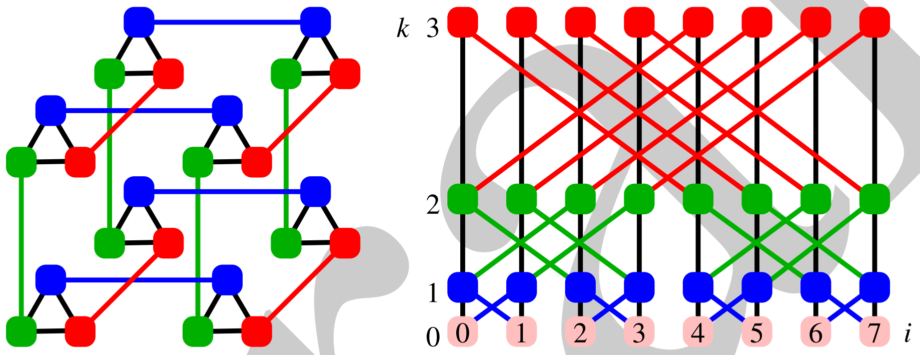


Figure 2.26: 3-dimensional cube-connected cycle (left) and butterfly (right).

A hypercube can be transformed into a *cube-connected cycle network (CCC)* with node degree 3 and logarithmic diameter by replacing the nodes of the hypercube by rings of size d. Node (i,k) – the k-th node on ring i – is connected with the k-th node in ring j if $i \oplus j = 2^k$. Figure 2.26 (left) gives an example. The resulting bisection bandwidth is $\mathrm{O}(p/\log p)$. The *butterfly* network has similar properties. Here PE (i,k) is connected with PEs $(i \oplus 2^k, k+1)$ and $(i, k+1)$ for $0 \le k < d$ and $0 \le i < 2^d$. Figure 2.26 (right) gives an example.

Many other constructions lead to the same asymptotic results for degree, diameter, and bisection bandwidth, e.g., shuffle-exchange, de Bruijn, and several other so-called *Cayley graphs*, which establish an interesting connection between networks and group theory [235].

[24]The HAWK supercomputer at HLRS in Stuttgart uses a 9-dimensional hypercube network. Each hypercube node is a switch. 16 compute nodes are connected to each of these switches. Hence, we have a hybrid between a hypercube and a multistage network; see Section 2.5.4.9 and `hlrs.de/en/systems/hpe-apollo-9000-hawk/`

Draft

2.5.4.9 Multistage Networks

Above, we make the oversimplification to identify nodes of the interconnection network and processors that run the parallel application. However, usually (multicore-)processors are just attached to nodes of the interconnection network (*switches*). These interface with the processors and have the task of routing messages through the network to other switches. In particular, some switches may not have processors attached to them. A typical such architecture is a *multistage network* where switches are arranged in layers. Processors form the bottom layer 0 and are connected to switches on layer 1. Switches at layer i connect to nodes on layers $i-1$ and $i+1$ only. A common construction principle is that the switches have k network links for some parameter k and that a switch can pass data packets between any pair of links. When $p \leq k$, all PEs are simply attached to a single switch. Otherwise, k' PEs are attached to each switch and its remaining $k-k'$ links are used to connect to the next layer 2. This is done in such a way that there is (at least) one path between each pair of PEs. Moreover, the resulting bisection bandwidth should be large while using a "reasonable" number of switches. There are many concrete construction principles for multistage networks. Usually, the programmer does not want to care about them but the assumption is that a good routing algorithm is implemented that can handle arbitrary communication patterns efficiently. The butterfly network shown in Figure 2.26 shows a multistage network with $k=4$ and $k'=2$. The fat-tree network in Figure 2.27 has $k=6$.

2.5.4.10 Fat Trees

A frequently used family of multistage networks has the following basic structure: For a k-layer network, consider only layers $0..i$ of the network for $0<i<k$. Then the network breaks into several connected components. View all layer-i switches in the same component as a single *virtual switch* at level i. In a fat tree [289] the virtual switches form a tree. (See Figure 2.27 for an example.) A message from PE i to PE j is routed first up to the lowest common ancestor of PEs i and j in the fat tree and then down to PE j. Note that when routing in the actual physical network, there may be many concrete paths from i to j but all of them map to the same path in the fat tree. Typically, the higher up we are in the fat tree, the more physical links correspond to a logical link in the fat tree. Thus, a fat tree can be seen as an abstraction of a multistage network where link bandwidth increases as we go up the layers. If the bandwidth of the fat tree's (virtual) links grows linearly with the number of PEs in the subtrees they connect, then we will have linear

Draft

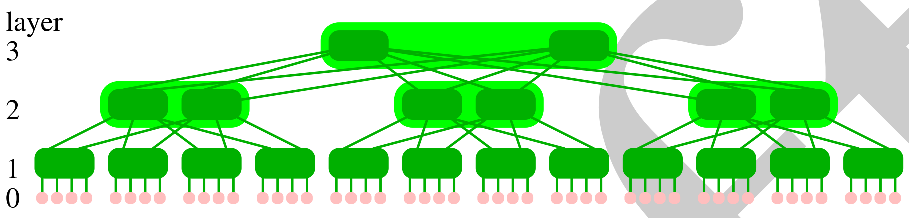


Figure 2.27: A 3-layer fat tree network with 48 leaves based on 6-way switches. The switches at layers 1 and 2 use 4 downward edges and 2 upward ones. Light green areas encompassing switches indicate virtual switches.

bisection bandwidth. In large computer systems, linear bisection bandwidth may be too expensive, however. This is why we view the concept of communication-efficient algorithms introduced in Section 2.5.2 as so important. Fat trees often have higher per-node communication bandwidth for local communication within subtrees and are hence well-modeled by the multilevel models discussed in Section 2.5.5.

2.5.4.11 Expander Graphs

So far, our examples for networks with linear bisection bandwidth (hypercube, multistage networks) needed $\Omega(n \log n)$ edges. Interestingly, there are also networks where a linear number of edges suffices. (Edge) *expander graphs* have the property that for any cut $C \subseteq V$ with $|C| \leq |V|/2$,

$$|\{(u,v) \in E : u \in C, v \in V \setminus C\}| \geq h \cdot |C|$$

for some constant h. Since h is fairly small for known deterministic constructions of expanders [239, 17, 278], in practice one would use randomized constructions. For example, consider the following $2d$-regular *random cycle graph*: $G_d(n) := (1..n, E_1 \cup \cdots \cup E_d)$ where E_i builds the cycle $(\pi_i(1), \ldots, \pi_i(n), \pi_i(1))$ for a random permutation π_i (π_1 can also be the identity mapping, see Figure 2.28 for an example). For $d \geq 2$, such a graph is an expander graph with high probability.[25]

Open Problem 14 (Expander graphs in practice) Although expander graphs are convincingly proposed as practical interconnection networks for real-world applications (e.g., [11, 223]), we are not aware of actual implementations. One

[25]If one wants a $2d+1$-regular graph one can use d cycles and one random matching.

Draft

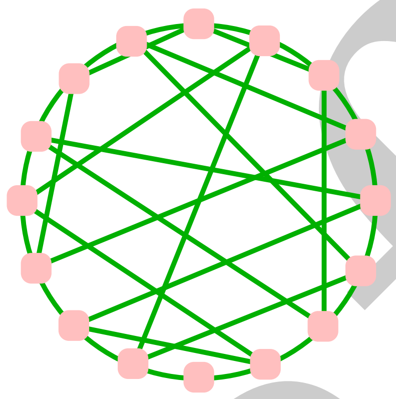


Figure 2.28: An instance of the graph family $G_2(16)$.

reason may be that the number of edges is not a good predictor of the actual physical cost of building a network. In particular, fat trees and related networks can be built from standardized components that are connected by regular patterns with mostly short connections. Hence, more work on *practical* expander graphs seems interesting. One could begin by empirically studying the expansion properties of different variants of random expander graphs. For example, the above random cycle graphs might have better expansion for small subsets than the graphs one finds in theory where edge endpoints are chosen randomly from $2 \cdot n \cdot d$ possible endpoints in a d-regular graph [482].[26] Subsequently, one could consider actual construction costs and compare them with other approaches like fat trees. Eventually, one could evaluate the impact on applications.

2.5.5 Hierarchical Networks and Combination with Shared Memory

We are facing the dilemma that models assuming a fully-connected network are too optimistic with respect to the bisection bandwidth but that tuning for one particular network topology is complicated and not portable. A compromise is to exploit that in all the above topologies it makes sense to distinguish between local and global communication on two or more levels of a hierarchy. This is perhaps most visible in a fat tree topology (see Section 2.5.4.10). PEs in the same subtree can communicate with higher bandwidth per PE and also with slightly smaller la-

[26]The more well known Erdős–Rényi random graphs are not expanders for linear m since they contain some small connected components.

Draft

tency. On mesh and torus networks (Sections 2.5.4.1–2.5.4.6) we can artificially impose such a tree-hierarchy by recursively bisecting the network (this works best when p is a power of two). This also works for hypercubes (Section 2.5.4.7) where a subcube is such a natural local set of PEs. Similarly, CCCs or butterflies can be viewed as variants of hypercubes where groups of PEs represent one node of a hypercube (Section 2.5.4.8). Only expander graphs (Section 2.5.4.11) – on purpose – have very little structure. Still, our permutation-based construction provides a known ring structure with some locality between PEs with nearby PE numbers.

A hierarchical network can also model the important fact that modern distributed-memory machines consist of nodes that are shared-memory machines. Thus, the lower levels of our hierarchy correspond to PEs on the same network node that can communicate very fast using their shared memory. Alternatively, we can follow a hybrid approach where we use a shared-memory model on the nodes and a distributed-memory model for the overall machine.

Hierarchical models of distributed memory are quite similar to the parallel memory hierarchies discussed in Section 2.4.6. The main difference is that PEs on level i do not interact by accessing memory at level $i+1$ but rather communicate at level i. Bilardi et al. [70] give a generalization of the BSP model to multiple levels of memory hierarchy. The *processing in memory model* (PIM) [264] combines a conventional shared-memory machine with additional processing cores attached to a local piece of memory, highlighting an interesting option for integrating memory and processing.

2.5.6 Arbitrary Networks

The networks discussed above are highly structured and explicitly designed to allow simple and efficient parallel programs. However, computer networks might have a very different structure perhaps based on the physical locations of their nodes and the available connections between them. In that case, one can consider algorithms that organize these networks, e.g., to establish spanning trees in them, to elect a leader, etc. [370, 32, 390]. These algorithms are often called *distributed graph algorithms* or more generally, *distributed algorithms*. However, these should not be confused with parallel (graph) algorithms running on distributed-memory machines. In the latter case, the nodes and edges of the graph are distributed over the network. Usually, the graph is much larger than the number of nodes in the network. In the former case, the graph to be processed and the network are the same thing. Initially, each node only knows its neighbors. Nodes

Draft

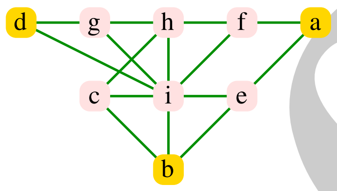


Figure 2.29: Illustration of a step in a distributed graph algorithm in the CONGEST model. The nodes are labeled with random letters. In a single step, these labels are sent to neighboring nodes. With this information, nodes can decide locally whether they have a smaller label than all their neighbors. These are colored in orange. The orange nodes form an *independent set* of the graph. By removing these nodes and their neighbors from the graph and iterating these two steps, one can compute a *maximal independent set* of the input graph in a logarithmic number of rounds with high probability [297].

have a unique ID but they are not numbered consecutively $1..n$. Also, the cost models can be different. In the theory of distributed algorithms, the following two models called LOCAL and CONGEST are frequently used. Both of them count globally synchronized rounds of communication and assume that each node can communicate with all of its neighbors within a round. In the LOCAL model, the message length is unbounded whereas in the CONGEST model, only $O(\log n)$ bits can be exchanged. Figure 2.29 gives an example.

Open Problem 15 (Theory versus practice for distributed algorithms)
Distributed algorithms in the LOCAL and CONGEST models have been intensively studied in the algorithm theory community. They also make a lot of sense for proving lower bounds. However, many distributed algorithms have never been implemented. How do these algorithms perform in practice? Can they be adapted to more realistic cost models, i.e., where only a constant number of messages can be exchanged in constant time and where nodes work asynchronously? Can the techniques used for distributed graph algorithms also be adapted to yield efficient parallel algorithms?

2.5.7 Cellular Automata

A cellular automaton (CA) consists of identical finite automata (cells) arranged in a regular d-dimensional grid; see Figure 2.30. All cells operate synchronously.

Draft

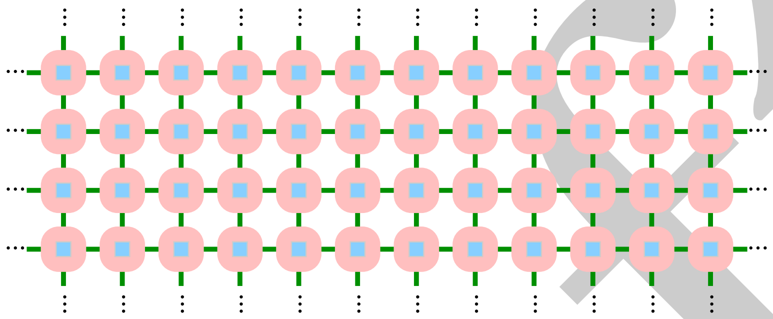


Figure 2.30: A 2-dimensional cellular automaton with von Neumann neighborhood, i.e., the next state of each finite automaton depends on its previous state (blue square) and the previous states of its four immediate neighbors.

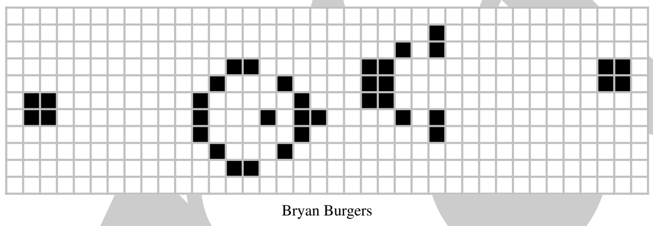


Figure 2.31: Gosper's Glider Gun that produces a stream of gliders in the game of life cellular automaton. A cell in the game of life is dead or alive. We consider 3×3 "Morton" neighborhoods. A living cell stays alive if and only if it has 2 or 3 living neighbors. A dead cell becomes alive if it has exactly 3 living neighbors.

Their state in the next step is a function of their current state and the state of a fixed set of neighboring cells. An input of size n is presented as the initial state of a set of n contiguous cells. All other cells are initially in a ground state. The grid itself is potentially infinite. Cellular automata are popular as a simple model for complex systems. For example, Conway's Game of Life is a staple of recreational mathematics [64]; see also Figure 2.31. Indeed, CAs represent the arguably most simple universal model of computation. Not only the game of life but even some very simple one-dimensional automata with one bit of state are already Turing complete [125]. This is surprising as there are only 256 ways to specify the transition function of such an automaton. All of these have been investigated intensively [481]. Cellular automata have been used as a basis for distributed algorithm devel-

Draft

opment [469] to study fundamental problems such as self-replication [99], leader election [53, 354], or synchronization [395, 460]. Another advantage is that CAs are physically realizable in the sense of Section 2.8.2, i.e., with respect to layout area/volume and signal propagation delays. The somewhat unrealistic assumption of synchronous operation can be lifted by emulating a CA on a machine that only needs local synchronization [347]

Nevertheless, CAs are rarely used for actual software development because they operate on a very low level – the PEs (cells) do not know about their index and even implementing a counter requires a joint effort of multiple cells.

Exercise 23 *Design a 1D cellular automaton that counts the number of cells whose initial state is* 1. *At the end of the computation, the leftmost cells of the automaton shall contain a unary representation of the desired output. For example* [0010000100110001] $\rightarrow$ [1111100000000000]. *Repeat the exercise with a binary output represented in the leftmost cells.*

One could also consider a model CA$^+$ where cells are RAMs restricted to a polylogarithmic amount of memory. This is very close to distributed-memory machines with mesh networks, however with an additional restriction to severely limited local memory. One would probably also drop the implicit assumption that $\alpha \gg \beta$ because such a fine-grained model only makes sense if communication has very low latency.

2.5.8 Communication Complexity

Research in communication complexity is about establishing lower and upper bounds on the number of bits that need to be communicated in order to solve certain problems [282]. However, in contrast to the communication-efficient algorithms discussed in Section 2.5.2, communication complexity research has so far concentrated on networks with very simple structure. Even just two nodes are already a challenging setting. Multiparty communication has looked essentially at star networks, where there is a centralized coordinator [153] or blackboard [282] used for information exchange.

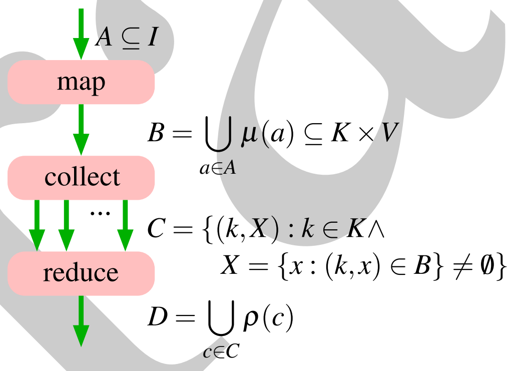
Open Problem 16 (Multiparty communication complexity)
Study lower bounds of fundamental problems in communication complexity for the general case that we have a fully connected network of p nodes and that we are interested in the bottleneck communication complexity as in Section 2.5.2.

Draft

For example, in [412] we show an upper bound of about $\log p$ bits per element to solve the classical duplicate detection problem. We conjecture that this is tight. However, no formal proof is known yet.

2.6 MapReduce and Other High-Level Models

The above models for parallel processing require the explicit coordination of p PEs. In nontrivial cases, complex load-balancing strategies are required and in large practical configurations, the computations have to be done in a fault-tolerant manner. Also, we have already seen that realistic modeling of the cost of these computations is complicated. Hence, we might want to look at much more abstract models that leave parallelization, load balancing, and fault tolerance to a framework for big data processing.



$A \subseteq I$
map
$B = \bigcup_{a \in A} \mu(a) \subseteq K \times V$
collect
...
$C = \{(k,X) : k \in K \wedge X = \{x : (k,x) \in B\} \neq \emptyset\}$
reduce
$D = \bigcup_{c \in C} \rho(c)$

Figure 2.32: Computations specified by a MapReduce step.

A simple and well-known approach is MapReduce [132]. MapReduce steps get a multiset $A \subseteq I$ of elements from an input data type A and *map* A to a multiset of *key-value pairs* $B = \bigcup_{a \in A} \mu(a) \subseteq K \times V$ for a user-defined mapping function μ. Next, values with the same key are collected together, i.e., the system computes the set $C = \{(k,X) : k \in K \wedge X = \{x : (k,x) \in B\} \neq \emptyset\}$. Finally, a user-defined *reduction* function ρ is applied to the elements of C to obtain an output multiset D.[27] Figure 2.32 summarizes the resulting logical data flow. The user only needs to specify μ and ρ; the system is taking care of the rest. Chaining

[27]In most papers, the sets A and D are also defined to be key-value pairs. However, there seems to be no mathematical necessity for this specialization so we use a more general definition here.

Draft

several MapReduce steps with different mapping and reduction operations yields a wide spectrum of useful applications.

Exercise 24 *Define mapping and reduction operations for a MapReduce step that solves the* word-count problem. *The input is a multiset of lines of text, each containing a sequence of words separated by spaces. The output is a set of key-value pairs (w,c) where w is a word and c is the number of time w occurs in the input.*

The MapReduce concept has been developed into a theoretical model of "big data" computations (MRC) [267] that is popular in the algorithm theory community. Problems are in MRC if they can be solved using a polylogarithmic number of MapReduce steps and if a set of (rather loose) additional constraints is fulfilled: Let n denote the input size and $\varepsilon > 0$ some constant. The time for one invocation of μ or ρ must be polynomial in n using "substantially" sublinear space, i.e., $\mathrm{O}(n^{1-\varepsilon})$. The overall space used for B must be "substantially" subquadratic, i.e., $\mathrm{O}(n^{2-2\varepsilon})$. While MRC has given new impulses to parallel complexity theory, it opens a new gap between theory and practice. MRC based algorithms that use the full leeway of the model are unlikely to be efficient in practice. They are not required to achieve any speedup over the best practical sequential algorithm. They are also allowed to use near quadratic space so that they may not be able to solve large instances at all.

However, a slightly more precise analysis can yield a model MRC$^+$ that is more predictive for efficiency and scalability yet maintains the high level of abstraction of MRC. The main change is to not only count MapReduce steps but also to analyze the communication volume and local work.[28] In other words, rather loose space and time constraint of MRC are changed into obligations for analysis in MRC$^+$.

More precisely, let w denote the total time needed to evaluate the functions μ and ρ on all their inputs. Let $\hat{w}$ denote the maximum time for a single call to these functions. Let m denote the total number of machine words contained in the sets A–D. Let $\hat{m}$ denote the maximum number of machine words produced or consumed by one call of the functions μ and or ρ. It turns out that, in some sense, these four parameters fully characterize the difficulty of a MapReduce step:

Theorem 1 ([404]) Assume that the input set A of a MapReduce step is distributed over the PEs such that each PE stores $\mathrm{O}(m/p+\hat{m})$ words of it. Then

[28]Note that there is an analogy here to the CGM and BSP models where the former also only counts steps.

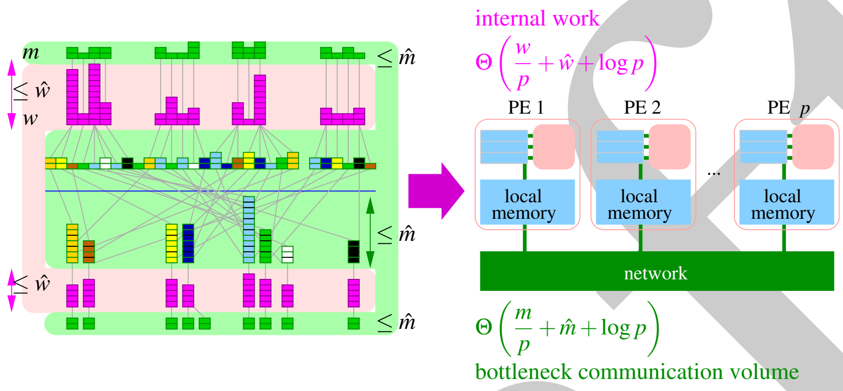


Figure 2.33: Illustration of Theorem 1.

it can be implemented to run on a distributed-memory parallel computer with expected local work and bottleneck communication volume

$$\Theta\left(\frac{w}{p}+\hat{w}+\log p\right) \text{ and } \Theta\left(\frac{m}{p}+\hat{m}+\log p\right), \tag{2.1}$$

respectively. These bounds are tight, i.e., there exist inputs where no better bounds are possible. Moreover, no PE produces more than

$$\mathrm{O}\left(\sum_{d\in D}\frac{|d|}{p}+\max_{d\in D}|d|\right)=\mathrm{O}\left(\frac{m}{p}+\hat{m}\right)$$

words of output data. Figure 2.33 illustrates this result.

The pre and postconditions are formulated in such a way that several MapReduce steps can be chained. Theorem 1 "almost" means that a MapReduce step can be simulated by a constant number of supersteps in the BSP model (see also Section 2.5.3). Indeed, the paper [404] shows a slightly weaker upper bound that can be implemented using two BSP supersteps. To achieve the tight bound, an asynchronous work-stealing algorithm is needed (see also Section 18.3). Therefore Theorem 1 does not use the BSP model. It also does not use the point-to-point model from Section 2.5.1 because the implementation employs a general BSP-like data exchange step which entails several implementation tradeoffs as already discussed in Section 2.5.3.

Draft

Open Problem 17 (Rooting MapReduce models in reality.) The results from Theorem 1 are only the beginning. One should also consider more detailed models of realistic parallel machines. One would like to have a more explicit treatment of startup overheads, memory hierarchies (disk versus RAM), and fault tolerance (see also Section 2.10). One should also analyze the performance of the algorithms actually used in current MapReduce implementations to see what bounds one gets and whether the algorithms can be improved for better scalability or robustness.

Exercise 25 *Analyze your solution to the word-count problem (Exercise 24) using the terminology and results of theorem 1. Why does this result have very limited scalability for real-world inputs? Hint: what about very frequent words? Adapt the model to enable a specialized (highly local) treatment of associative reduction operations. How does this change the analysis of the word-count problem?*

At the same time the MRC model has gained popularity as a theoretical model, practitioners have increasingly realized that plain MapReduce alone is not enough to implement a sufficiently wide range of applications efficiently. Breaking down an application into MapReduce steps often requires a large number of steps and thus complicates algorithm design. This is exacerbated by the requirement to communicate (and possibly move to/from external memory) basically all the involved data in every step. Even the original MapReduce publication [132] already introduces a more communication-efficient variant with reducers that allow local reduction of data with the same key, e.g., using a commutative, associative operator like $+$ or min. More recent big data tools such as Spark [487], Flink [106], or Thrill [71] adopt the highly abstract basic approach of MapReduce but offer additional operations and/or data types. For example, the Thrill framework [71] is based on arrays and offers operations, for mapping, reducing, union, sorting, merging, concatenation, prefix sums, windows, etc. The MRC$^+$ model introduced above can be adapted to this approach. For each operation, we analyze its complexity in a realistic model of parallel computation. Similar to the BSP$^+$ model from Section 2.5.3, we are likely to end up with a small number (one for BSP$^+$) of categories of operations that are assigned the same cost in the model based on a small number of involved parameters.

Open Problem 18 (Generalized MapReduce-like models.) The above approach is just an outline of how to deduce such a model. One should spell it out for the most popular big data frameworks arriving at a small number of

Draft

Table 2.6: Summary of some main characteristics of parallel machine models discussed in this chapter. See the text for more details.

name	parameters	local	granularity	sync.?	Section
			shared memory		
*R*W-PRAM	p	–	**low**	yes	2.4.1
a*RQW-PRAM	p	–	**low**	no	2.4.2
PEM/PEM$^+$	p, B, M	$\star\star$	medium	yes	2.4.6
par-obliv.	many	$\star\star\star\star$	medium	yes	2.4.6
			distributed memory		
point2point	p, α, β	$\star\star\star$	medium	no	2.5.1
networks	p, α, β, graph	$\star\star\star\star\star$	low	no	2.5.4
LogP	p, L, o, g	$\star\star$	low	no	2.5.1
LogGP	p, L, o, g, G	$\star\star\star$	medium	no	2.5.1
BSP	p, L, g	$\star\star$	high	yes	2.5.3
BSP*/BSP$^+$	p, L, g, B	$\star\star\star$	high	yes	2.5.3
CGM	p	$\star$	**high**	yes	2.5.3
MRC	–	$(\star)$	**high**	yes	2.6
MRC$^+$	–	$(\star)$	high	yes	2.6

proposals for an abstract model. At the same time, the set of operations supported by the existing tools might not be the last words. Perhaps the abstract models help to conceive gaps in the spectrum of available operations. In particular, one would like to have data access operations that only require communication for performed queries but not for the accessed data structure. For example, we might want to implement a distributed full-text index that supports fast and communication-efficient batched queries [178].

2.7 Taming the Zoo of Parallel Models

In the previous sections, we have seen a large number of models for parallel computing. This is confusing and perhaps one reason for the slow progress of parallel algorithmics. So let us compare them in order to understand their respective advantages, to help select the right one for a particular situation, and perhaps to come to a synthesis. Table 2.6 summarizes some basic properties. Column "parameters" lists the parameters used for describing the machine.

Column "local" gives a score of how well the model takes locality of com-

Draft

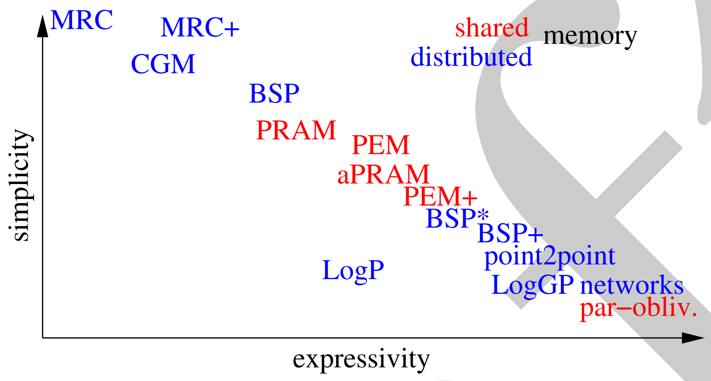


Figure 2.34: Simplicity versus expressivity of some models of parallel computing.

putation into account. Basic PRAMs do not model locality at all. MapReduce also scores low since it basically moves all the data except that there may be some locality within the mapping operation. CGM is little better but at least allows arbitrary operations on all the local data. The other models handle locality fairly well with a minus point for PEM compared to the distributed-memory models as it requires loading all input data into the cache first. LogP and BSP get a minus point because they do not handle the spatial locality of messages well. Parallel memory hierarchies and concrete interconnection networks get plus points for modeling not just the two categories local and global.

Column "granularity" stands for the granularity of the computations done in the respective model. This ranges from very fine-grained PRAM computations to CGM and MRC, where each parallel step implies the communication of all the data using complex communication patterns. Column "sync.?" indicates whether the computations proceed in a globally synchronized manner.

It is difficult to derive interesting qualifications such as simplicity, or realism from these basic properties. For example, while the number of machine parameters is certainly correlated with simplicity, we abstain from also displaying parameters related to the input instance or the analyzed algorithm that are needed to perform an algorithm analysis in the model. Hence, Table 2.6 is mostly useful as a way to select relevant features for the application at hand. How relevant is communication efficiency? How coarse-grained can the computation be, given constraints on the tolerable latency? Do we want to consider asynchronous algorithms?

Draft

Figure 2.34 gives a more aggregated and more subjective classification of the models along the dimensions of simplicity and expressivity, i.e., how many aspects relevant to realistic machines and algorithms are grasped by the model. We see that most of the models are on a Pareto curve, i.e., we pay with a more complicated model in order to achieve higher expressivity. Thus, all these models have their justification. On the other hand, perhaps we do not need such a fine-grained tradeoff between simplicity and expressivity.

One can turn this around and observe that a particular abstract algorithm may be formulated and analyzed in many models in sufficient detail to understand its main properties. For example, consider parallel sample sort (see [410, Section 5.14]) – a generalization of quicksort doing multiway partitioning based on a sample of the input. Sample sort can be expressed in all the above models except MRC. Most of these models (except (a)PRAM and LogP) correctly express the communication volume which is the bottleneck for large inputs. Further restrictions on the model only become relevant when we want to grasp exactly when the algorithm becomes efficient and how this depends on the algorithm used for sorting the sample. Even then, the models PEM, BSP$^+$, and point-to-point are similarly useful for investigating this. Concrete network models or multilevel hierarchies seem overly complicated to investigate such a simple two-level algorithm.

Exercise 26 *Pick two models from Table 2.6 and compare them. Point out advantages and disadvantages of each model. Try to name applications where each model would be (in)adequate. Some interesting pairs: PEM versus point-to-point or BSP versus MRC.*

As a conclusion and partial synthesis, we propose to take concrete models into account as late as possible in the design and analysis process. We could call this the *model-agnostic* approach. Rather, we should try to describe algorithms in terms of basic primitives that have concrete implementations in many models. If desired, we can then plug in the analysis of the primitives in a particular model to obtain a concrete analysis. For example, a basic sample sort can be expressed as 1: local *sampling*, 2: a *gather* of the sample, 3: local sorting of the sample, 4: *broadcasting* of splitter elements, 5: local partitioning, 6: *all-to-all* data exchange, and 7: local sorting. A more scalable variant replaces steps 2–4 with a fast parallel sorting algorithm for small inputs.

Draft

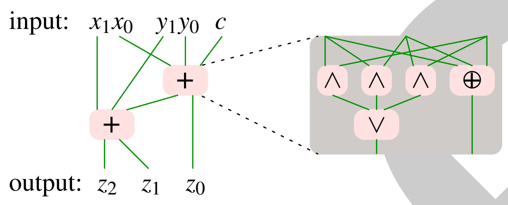


Figure 2.35: Circuit for a 2-bit adder that computes $z = x + y + c$ where c is an input carry bit, x and y are 2-bit numbers and z is a 3-bit number. The circuit is built out of full adders that take three input bits resulting in two output bits.

2.8 Circuits

The models considered so far describe the step-by-step manipulation of the state of a computation by a program. Now, we take a lower-level, memoryless view where we consider how a single function is composed from a set of primitive operations.

2.8.1 Logical Circuits

The most simple circuits [437, Section 9.3] are described by a directed acyclic graph (DAG) that describes a Boolean function $f : \{0,1\}^n \to \{0,1\}^k$. Nodes of this graph are called *gates*. There are n *input gates* with in-degree zero. The i-th input gate outputs the i-th input bit x_i. All other gates compute the logical "and", "or", "xor" or "not" of their input bits and output this to a set of other gates. There are k designated *output gates* that output $f(x_1,\ldots,x_n)$. Figure 2.35 gives an example of a circuit with five input bits and three output bits. Since a single circuit can only handle inputs of a fixed size, one also considers *circuit families* with one circuit for each possible input size. The most important complexity measures of a circuit are its size (number of gates) and depth (the length of a longest path in the circuit DAG). These are related to the concept of work and span introduced in Section 2.4.1 for PRAMs.

Despite their simplicity, circuits are a universal model of computation. In particular, any computation that is done by a Turing machine can also be done by a circuit family with a size that depends quadratically on the time required by the Turing machine.

We can also generalize circuits to allow further gates (with a bounded number of inputs) and wires that carry a bounded number of bits. This changes the com-

Draft

plexity of a circuit at most by a constant factor (but note that then the graph structure does not contain all the required information because, for non-commutative operations, it matters to what *port* of a gate a wire is connected). For example, such a generalization makes sense when we are designing circuits that perform arithmetic operations. The textbook [410, Section 1.1.1] gives an example for performing addition. Here, a full-adder gate with three input bits and two output bits is a natural building block (see also Figure 2.35). Indeed, as long as we keep in mind what we are doing, we can also model more abstract computations as a kind of circuit. For example, a *sorting network* consists of "gates" which compute $s(x,y) = (\min(x,y), \max(x,y))$ where x and y are abstract objects that allow no further introspection [51, 10]. In Section 2.4.1, we used these kinds of generalized gates to model computations on entire machine words.

Exercise 27 *Look for sorting networks for 5 elements with depth 5. Can you achieve the optimal size of 9?*

When implementing algorithms in hardware, we also need a way to represent state. *Registers* can store a vector of signals. In order to describe such systems, we need a hardware description language such as VHDL [294] or Verilog [452]. To precisely define how such a logical hardware description behaves, these languages need to define the temporal behavior of the registers. Describing this precisely is beyond the scope of this book. A frequent approach is that a clock signal plus further logic defines when a register takes over signals from its inputs. Note that *field programmable gate arrays (FPGAs)* blur the distinction between software and hardware, i.e., on a machine with FPGAs, we can compile algorithms described by a hardware description language such that it can be executed using the FPGAs.

Open Problem 19 (Scalable hardware) Much of the algorithms community is deterred by the complexity of describing hardware algorithms. On the other hand, computer architecture is dominated by a quantitative approach that concentrates on detailed experiments [229]. However, using experiments exclusively makes it difficult to ascertain scalability to large systems. We believe that an asymptotic analysis of the scalability behavior of hardware might help to arrive at more scalable hardware architectures, e.g., for shared-memory management in large parallel computers. A successful example (given the limited budget) is the SBPRAM project [3, 369]. One interesting question could be a generalization to exploiting modern technology parameters, caches, asynchrony, etc.

Draft

2.8.2 Physically Realistic Circuits

All the machine models discussed so far are abstractions that make simplifying assumptions that decouple the notion of costs in the model with the costs in a physical realization of the machine. Even circuits are unrealistic in that sense because they assume constant time communication along wires of arbitrary length. Also, the volume or weight needed for a physical realization of a circuit grows more than proportionally to their size measured in the number of gates if we need long wires. Thus, more realistic models could take such physical constraints into account.

In the 1980s, a model for two-dimensional VLSI circuits was intensively investigated [453]. Here, signal propagation delay along wires is still considered a constant but the actual area of the circuit in a two-dimensional layout is taken into account. In particular, an interesting tradeoff between area and time has been found. For many problems, there are lower bounds on the product AT^2 where A is the VLSI layout area and where T is the required time. For example, all the communication networks with logarithmic delay and linear bisection bound discussed in Section 2.5.4 need quadratic area $\Omega\left(p^2\right)$. An informal way to understand that is to look at the layout of butterfly networks in Figure 2.26-right. In order to have a constant distance between wires, the distance between layers has to be doubled from layer to layer. With $\log p$ layers and linear area of the first layer, we arrive at quadratic overall area. These considerations can in part be generalized for three-dimensional layouts, e.g., [377]. Three dimensions help by allowing the volume to grow with only $O(p^{3/2})$.

A next step is to take into account that the speed of light limits the speed of computation. Hence, at a fixed feature size of a chip technology, the running time of most nontrivial computations on n bits will grow at least with $\Omega\left(n^{1/3}\right)$ (or even $\Omega\left(n^{1/2}\right)$ for two-dimensional layout). Note that this already applies to a single memory access to a data structure storing n bits. From that point of view, the fastest realistic machine consists of a regular three-dimensional grid of small processors. The cellular automata discussed in Section 2.5.7 are such a model. However, further limitations may play a role. Most importantly, assuming that computations are connected with producing heat, we must be able to remove that heat through the surface of the machine; see also Section 2.14. Since the surface grows only as $p^{2/3}$, this limits the amount of computation possible per unit of time. This implies different tradeoffs for how to arrange the processors [417]. Perhaps the most practical arrangement is to have the computation take place close to the

Draft

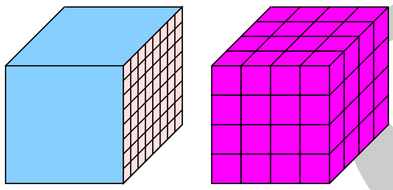


Figure 2.36: Two extreme configurations of memory (blue) and heat-producing processing (pink) that are physically feasible with respect to cooling. Left: computing is only on the boundary of the computer. Right: A more coarse-grained 3D arrangement of units that combine computing, local memory, communication, and cooling infrastructure.

surface of the machine and to use the third dimension for storage and communication; see also [377]. In particular, moving heat from the inside has additional issues. Doing this by diffusion is inherently inefficient. Also in more directed cooling approaches, friction of a gas or liquid used for cooling may asymptotically become larger than the amount of energy that can be removed. Indeed, this observation seems to be reflected in recent technological developments. For example, SSDs now routinely use many layers of memory per chip and they also densely stack memory chips. In contrast, processor chips stick to two dimensions (except for memory modules that may be stacked on top of them) and are nevertheless highly constrained by their heat dissipation.

Open Problem 20 (Scalability and physics) In continuation of Open Problem 19, one can go beyond standard simplifying assumptions of hardware design and can take physical constraints such as cooling and wire delays into account. For example, one could consider how the situation for the very simple cellular automaton model [417] changes if one uses full-blown RAMs plus a memory hierarchy: Assume a hardware budget for n machine words of memory and $p = \tilde{O}(n^{2/3})$ PEs.[29] One approach is a flat array of $\tilde{O}(n^{1/3} \times n^{1/3})$ PEs with a (rather deep) memory hierarchy arranged in the third dimension and latency $\tilde{O}(n^{1/3})$. How should this memory hierarchy be arranged? Perhaps it makes sense to allow global memory access to a large part of the memory by all PEs? Another approach would be near cubic PEs of side length $\tilde{\Omega}(n^{1/9})$ arranged into a cube of $\tilde{O}(n^{2/9} \times n^{2/9} \times n^{2/9})$ PEs. Each PE could have a (rather shallow) local memory

[29]Here we use $\tilde{O}(\cdot)$ as a variant of asymptotic notation that ignores polylogarithmic factors.

Draft

hierarchy with latency $\tilde{\Omega}(n^{1/9})$.[30] Figure 2.36 illustrates these two configurations. How can such a system be cooled without friction asymptotically eating up all of the cooling effects? Would a fractal structure of cooling tubes and channels help? How does this compare with the fractal structure of our blood vessels [196]? Does this fractal structure influence the arrangement of the PEs?

****Exercise 28** *Consider the latter approach described in Open Problem 20 with PEs that are cubes of side length $\tilde{O}(n^{1/9})$. A simple approach to cooling such a system would be $\tilde{O}(n^{2/9} \times n^{2/9})$ water pipes of diameter $\tilde{O}(n^{1/9})$ going through the entire cube. With this system, constant flow velocity suffices to transport the $\tilde{\Omega}(n^{2/3})$ W of heat generated by the PEs (assuming each PE dissipates constant power). Calculate the dissipated heat due to friction within the pipes. Do that first assuming laminar flow. Now take into account, that with growing pipe diameter, the Reynolds number of this system grows, eventually leading to turbulent flow. How does this affect the asymptotic friction loss of the cooling system? (see also* `en.wikipedia.org/wiki/Friction_loss`*, accessed Dec. 17, 2023).*

2.9 Streaming Algorithms

The concept of a streaming algorithm originates from processors analyzing data streams in a network. Their memory is often much smaller than the total analyzed data volume. Hence, we need algorithms that take very little memory. However, there are also many other sources of data streams. They may originate from sensor hardware (e.g., video cameras) or from within a long-running application. The latter data source might for example be for performance profiling or for analyzing the outcome of a scientific simulation over time.

More formally, a streaming algorithm gets a sequence of N (small) objects $o_1, o_2, \ldots, o_N$ as input that it cannot store. It is required to take total space $o(N)$ and processing time $o(N)$ per object [230, 40, 344, 126]. See also Figure 2.37. Often the requirements are even more strict, with a target of polylogarithmic space and near-linear total execution time. Note that the input may arrive in real-time so that the analysis can also have a real-time aspect (or, sufficient space for buffering unprocessed objects is needed).

[30] This can be viewed as a physical model explaining why the *processor in memory* approach is useful [265].

Draft

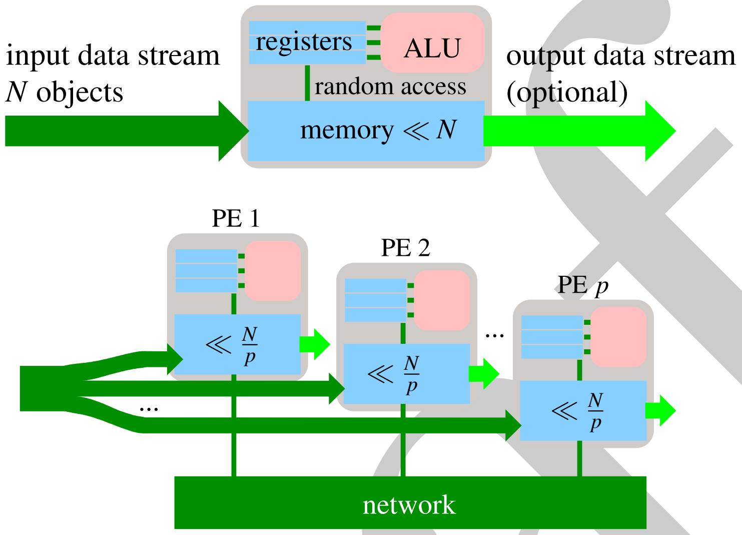


Figure 2.37: Streaming algorithms look at a random access machine (see also Figure 2.3) with limited memory processing a huge data stream (top). In distributed streaming (bottom), the data stream is split over p PEs of a distributed memory machine (see also Figure 2.17). Some theoretical models restrict the network to a star network centered at a coordinating PE.

Exercise 29 *Give a constant-space streaming algorithm for calculating the sample mean and variance of each prefix of a stream of numbers. Perform a web/literature search to discuss tradeoffs with respect to efficiency and numerical stability of various concrete algorithms.*

More specific constraints are used for particular classes of streaming problems. For example, in *graph streaming* algorithms [169, 321], one often observes a sequence of edges of a graph. Then we may be allowed to store a constant (or polylogarithmic) number of machine words per node of the graph.

Particularly large data sets are involved when the data stream arrives in a distributed fashion at many places. Hence, we need *distributed streaming algorithms* based on any of the models discussed in Section 2.5, see Figure 2.37. This makes particular sense when the analyzed data stems from a parallel application such as a massively parallel simulation. *Discretized streams* [488] are an abstraction that considers batched, stateless computations on small batches of data and thus al-

Draft

lows scalable fault tolerance. Allowing state but restricting it to a small amount of internal memory in each PE seems like a useful model for distributed streaming. We used this model for describing highly scalable reservoir sampling [248].

Open Problem 21 (Scalable distributed streaming algorithms) Surprisingly, using the terminology from Section 2.5, most work on distributed streaming so far concentrates on the notoriously nonscalable star network from Section 2.5.4, e.g. [127]. Hence, there is great potential for more scalable distributed streaming algorithms.

A logical extension of the single-pass streaming algorithms considered here are multi-pass algorithms. It is difficult however to match this approach with the original motivation for streaming problems. The concept seems interesting anyway, for example as a special kind of external memory algorithm that is only allowed to scan the input. In such a case, one should however compare multipass streaming with other algorithms that are allowed more general access patterns, e.g., the random block accesses from external memory.

2.10 Fault Tolerance

A crucial assumption made in most abstract models of computation is that all components work 100% correctly. However, this becomes increasingly unrealistic. Computer systems contain more and more components (transistors, wires, lines of code, etc.) each of which can fail. Moreover, in order to get more energy efficient, the physical events defining a logical event in a digital computer get smaller and smaller; see also Section 2.14. For example, the number of electrons used to switch a transistor is decreasing. This makes the system more susceptible to random events like thermal noise, cosmic radiation, or tunneling of electrons through a potential barrier. Models like quantum computers (Section 2.12) or analog computers (Section 2.13.3) inherently have to handle such random events. Already now, hardware and system software are offering several fault tolerance mechanisms. Error-correcting codes protect memory and communication channels. Detected errors are hidden by retrying an operation and by reconfiguring the system to leave out permanently faulty components. This is already now an interesting area of algorithmic research but still specialized to a small number of tasks such as writing checkpoints and rolling back to them.

However, with increasingly frequent faults, it will become necessary to move fault tolerance into more and more algorithms. Basic toolbox algorithms and the

Draft

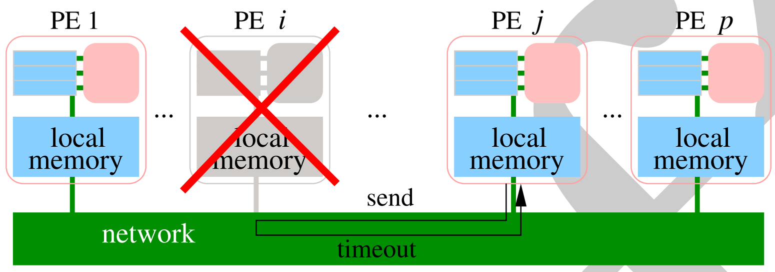


Figure 2.38: A distributed-memory parallel machine where PE i has failed. PE j learns about this when it attempts to send a message to PE i.

basic operations of the big data tools discussed in Section 2.6 may be the first targets. As usual in models of computations, we need abstractions from the many things that can happen in practice. In principle, any component can fail at any time and we may not be able to cover all possible scenarios. On the other hand, it may suffice to handle the most frequent types of faults or those that cannot be covered by hardware or systems software. Here we mention a few such models without attempting to give a complete overview.

The algorithm theory community has investigated *resilient* algorithms that work correctly in the presence of faulty memory [88, 177]. The model used is a random access machine with $O(1)$ "safe" memory words that are guaranteed to work correctly. An adversary can corrupt up to δ other memory words. This model is simple and addresses faults in the most numerous components of a computer system – its memory cells. On the other hand, the overhead of resilient algorithms is considerable [175] and seems to exceed the cost of hardware measures like error-correcting codes.

Lower overhead can be achieved in distributed computers. A useful partial model for fault tolerance is to request a computation to work correctly when a constant number of PEs stop their work [422] – the *fail-stop* model; see also Figure 2.38. Let us work out what this means for the point-to-point model from Section 2.5.1.[31] At any point in time t some PE i may stop working. Messages it has started to send before t, which are not yet delivered at time t, may or may not arrive. Messages sent to PE i that have not been delivered at time t will never be delivered. A PE j communicating with PE i learns about this by getting an error

[31] A similar model has been proposed for version 4.0 of the message passing interface (MPI) standard [451].

Draft

code that signals that a communication has failed due to a stopped PE. To analyze the running time of fault-tolerant algorithms, we need a bound on the time delay of such a notification. For simplicity, let us optimistically assume it to be $O(\alpha)$. But we should keep in mind that the delays may be much larger in practice.[32] PE j can then inform other PEs about the failure of PE i. For example, it could initiate a process for excluding PE i from some communication context and to produce an updated PE numbering that is once again consecutive. Or they could employ a spare processor to fill the role previously played by PE i.

The fail-stop model makes the implicit assumption that this kind of fault is the most frequent type of error, or, more precisely, that other errors can be hidden or converted into PE failures. This is the point of view we want to take here as a starting point. Message exchange between functioning PEs can be protected by error-correcting codes and by re-transmitting messages when errors are detected. Network errors that cut off a small number of PEs can be handled by assuming that these PEs have failed. Only large-scale network errors have to be avoided, e.g., by having enough redundancy in the network. PEs that only fail intermittently will be treated like failed nodes.

Open Problem 22 (Scalable fault tolerance) Fault tolerance mechanisms that scale to the largest machines (where they are most needed) seem to be a wide-open field. We observe shrinking times between failures and growing times for known fault tolerance mechanisms like check-pointing [161, 394, 486, 438, 105, 231]. *Algorithm-based fault tolerance* [243, 84, 231] makes the algorithms themselves fault-tolerant and thus promises a more scalable approach. However, so far this has only been done for select numerical computations [161, 394, 486, 231]. For basic toolbox operations like sorting or the operations of big data tools like MapReduce this is still an open problem. Even simple basic mechanisms like the heartbeat protocol for identifying failed machines [241] or leader election protocols for replacing failed coordinators [447, 113] currently need time $\Omega(p)$ where we would like protocols with (poly)logarithmic delay. On the theory side, such delays have been proven for techniques like randomized rumor spreading [152]. Although these methods were developed for use in fault-tolerant algorithms, it is not quite clear how they would be used in practice.

[32] Underneath, the communication system will use some timeout mechanisms. These are a tricky business. Too long periods slow down the program. Too short periods will produce false alarms. It will not be possible in general to simply set the timeout to $O(\alpha)$ – even for a large constant factor – since even a working PE may incur large delays if it is otherwise busy.

Draft

Even with the simplifying assumptions of the fail-stop model, we get very different problem characteristics depending on how many PEs will fail at the same time or how frequent failures are. The easiest case is the traditional view that failures are so rare that even redoing a complete job is feasible. In the largest current systems and taking some inspiration from the scalability of traditional parallel processing, one could consider subproblems that take polylogarithmic time as fast enough to be repeated when an error is detected. Longer tasks, in particular those that take time at least linear in p (e.g., operations involving all-to-all communication with direct data exchange), should perhaps be made fault-tolerant with low overheads. If we take asymptotics as seriously as in Section 2.8.2, we must assume that a (small) *constant fraction* of all components will fail as the system is scaled. An extreme version of fault-tolerance is considered in *peer-to-peer networks* (e.g. [442]) that operate correctly even when a large fraction of the nodes continuously enter or leave the network.

Exercise 30 *Perform an internet search to estimate the hardware failure rates of the CPU and main memory used in your computer. How big could a distributed-memory parallel computer become using these components in each compute node if we want to ensure that the average number of failing nodes within a week is at most one? Name additional likely causes of failures in such a machine.*

The fail-stop model does not directly handle resources that are much slower than expected. For example, when a processor chip is insufficiently cooled, it uses a very small clock frequency to protect itself from overheating. The PEs on this chip will work correctly but may considerably slow down applications. Such a situation can be handled using appropriate dynamic load balancing or by monitoring the speed of all PEs and treating overly slow ones as faulty.

Open Problem 23 (Integrating fault tolerance and load balancing) The above observation suggests that load balancing and fault tolerance should be treated in an integrated way. Fault tolerance will not catch slow PEs if it does not integrate a load-balancing aspect. Load balancing can treat failed PEs "almost" as a very slow PE if it can handle the non-cooperative behavior of failed PEs. Doing at least both in a scalable way as discussed in Open Problem 22 and in the textbook [410, Section 14.1] seems to be an important open problem.

Difficult to handle are *Byzantine failures* [304] where faulty components may be taken over by an adversary that actively attacks the system.

Draft

Another aspect are *transient errors*, also known as *soft errors* that occur only temporarily and may disappear when an operation is retried. These errors are currently rare and stem mostly from cosmic radiation but may also stem from thermal noise or quantum effects in future systems that are more tuned for energy efficiency. In analog computers, at least small deviations from accurate computations have to be expected all the time. Soft errors are routinely detected and corrected in storage devices and communication channels. They also (more rarely) occur in logical circuits. Protecting against them requires checking the results of computations using the techniques described in Section 6.1.

2.11 Private Computations

The cryptography community has done intensive research on computations where two or more parties collectively perform a computation on private data. Each party provides part of the input. The other parties are not supposed to obtain anything additional about the input except the result of the computation [210]. For example, suppose that some big companies in the computer industry would like to collectively compute the average income of their engineers but no company wants to divulge the income level it pays. There are quite practical protocols for computing such linear functions [345]. Figure 2.39-left illustrates this situation. There are also general methods for privately evaluating a function defined by a

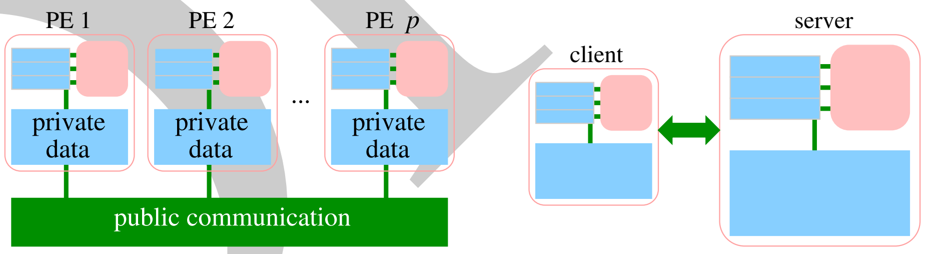


Figure 2.39: Left: a variant of the distributed-memory model for privately evaluating a function f that depends on private data. The PEs do not want to reveal any information on their private data except for the value of the computed function. This should work, even if an attacker can read all the exchanged messages. Right: A client-server model for private computation where the client has limited resources and thus needs the service of a server with larger resources. However, the server is not trusted.

Draft

circuit doing work proportional to the circuit size. However, the involved constant factors are large and converting an arbitrary computation that takes time t into a circuit results in a circuit of size t^2; see Section 2.8.1, [437].

An interesting variant is the client-server scenario: A client C, e.g., a small company, wants to use computer resources, e.g., from a provider P. C does not want P to learn anything (useful) about the actual computations done on its premises. This is easy if P just provides storage. C can encrypt its data. P will only learn the update pattern and the total volume of the data. Even this information can be reduced by adding fake data and fake updates. Indeed, with a logarithmic overhead, the access patterns can be hidden to the extent that the server provides a random access memory without learning more than the number of memory accesses [210]; see also Figure 2.39-right.

Open Problem 24 (Engineering private algorithms) It is currently believed that in order to make private computing practical on data sets of significant size, we need protocols specialized for a particular problem, or at least for some basic toolbox components. The hope is that by exploiting the special properties of the problem, we can circumvent the large overheads of general approaches. This seems like a task where the methodology of AE is needed. For example, there is a result on oblivious priority queues [255] that discusses the role of external memory algorithms, sorting networks, and RAM algorithms in developing oblivious algorithms.

2.12 Quantum Computing

Quantum computing [217, 355] is a fascinating emerging field that holds a lot of promise but can also be confusing. In the last few decades, it was mostly advanced by physicists. Since most computer scientists use different terminology and lack some of the physics and mathematics background, it is sometimes difficult for them to understand the results. This section tries to remedy some of these difficulties by presenting quantum computing using computer science terminology and by abstracting away some of the physics details. In contrast to "popular science" articles however, the idea is to preserve enough precision to make it possible to describe and analyze quantum algorithms presented at this level of abstraction. In the following sections, we present two important variants of quantum computing. Section 2.12.1 talks about how quantum operations are incorporated into familiar models like circuits and RAM machines. Section 2.12.2 explains

Draft

how quantum computing can directly solve combinatorial optimization problems. Both approaches may yield machines that are exponentially faster than classical computers but there are also a lot of question marks.

Open Problem 25 (From quantum "supremacy" to usefulness and viability) The term "quantum supremacy" is being used for an important first demonstration of the power of quantum computing. It marks the point when, for the first time, a quantum computer can solve *some instance* of *some problem* much more efficiently than any classical computer.[33] Note that the definition of this point allows massive cherry picking on the quantum side. The first problems being investigated are highly adapted to the available quantum hardware and even the problem instances under consideration are carefully chosen to be easy for the quantum computer and difficult for classical algorithms. Algorithm engineering will play an important role in marking this point. In particular, it is the main asset on the classical side of this benchmark – allowing new algorithms to be designed and tuned for the very special problems and instances selected for the quantum machine. Beyond "quantum supremacy" things will get even more interesting. When will quantum computers actually solve *useful* instances of a *useful* problem? When does such a niche grow into an economically *viable* area of computing? Will quantum computing eventually become the dominating paradigm of computing? We see algorithm engineering as central at least in the first stages of this evolution – we need many new algorithms both on the quantum and classical side and we need careful experimental evaluation to evade rash conclusions.

2.12.1 From Quantum Circuits to QRAMs

Quantum computers process data objects, that can be in a superposition of several classical states. The most common case is a *qubit* that can be in a superposition of 0 and 1. Usually, such a superposition is written as a complex number of absolute value 1. The power of quantum computing stems from the fact that n qubits can be *entangled* and then represent a superposition of up to 2^n possible states, i.e., a linear combination of states defined in an appropriate vector space. Operations performed on a set of entangled qubits can thus exhibit exponential *quantum parallelism*. This is exciting because quantum computers can do certain

[33]Many people are reluctant to drop the quotes around "quantum supremacy" since more modest terms like "eligibility" or "first qualification" might better describe this point and would avoid a lot of confusion.

Draft

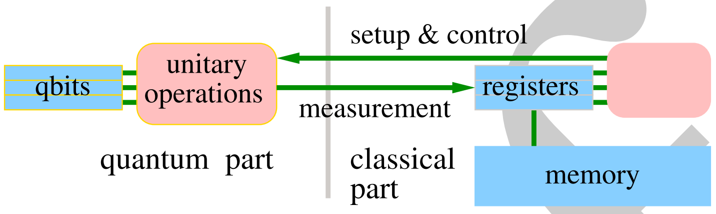


Figure 2.40: A quantum random access machine (QRAM).

things that classical computers can only do with exponentially more hardware. The caveat is that the operations allowed on an entangled set of qubits are quite limited – they have to be *unitary*. Unitary operations U can be described as square linear matrices U over an appropriate complex vector space[34] with the property that $UU^* = U^*U = I$ where I is the identity matrix and where U^* denotes the conjugate transpose operation.[35] In particular, unitary operations are *reversible*, i.e., they are not allowed to destroy any information and can thus be undone.

Quantum algorithms are often described using circuits whose gates perform unitary operations; see also Section 2.8.1. Deutsch [143] describes a quantum Turing machine where the state of the finite state machine consists of m quantum bits and each cell of the tape consists of one quantum bit. More close to currently planned machines is the QRAM [275] which is a classical RAM (see Section 2.2.1) that also has some quantum registers[36] on which it can perform unitary operations. In particular, the control flow is entirely classical. Interfacing between the classical and quantum parts of the machine is via preparing the initial state of quantum bits and via measuring the state of qubits. The probability of measuring a particular state is proportional to its amplitude in the linear combination stored in the qubits. The measurement operation collapses the superposition of states stored in the entangled set of qubits and forces their state to be the measured state. Thus measurement should usually only happen when the desired result has been

[34]The scalars in this vector space are complex numbers rather than real numbers as in the more familiar Euclidean space.

[35]The matrix is first transposed and then each matrix entry $a+ib$ is conjugated, i.e., changed to $a-ib$.

[36]This paper [275] does not specify how many bits these registers contain. Following our convention of allowing $\mathrm{O}(\log n)$ bits would restrict the quantum parallelism to be polynomial in the input size which would make the QRAM much less interesting from a complexity-theoretic perspective. Hence, it makes sense to specify the size of quantum registers separately to be potentially larger than the size of classical registers (but perhaps polynomial in the input size).

Draft

computed with sufficient probability. The probabilistic nature of measurement also implies that quantum algorithms are inherently randomized in nature.

Preserving the entanglement of qubits requires them to be perfectly isolated from their environment. Since this cannot be done perfectly, quantum computations can reliably perform only a rather short sequence of operations. Furthermore, fault tolerance, error correction, etc. will be an important ingredient of quantum computers and quantum algorithms; see also Section 2.10.

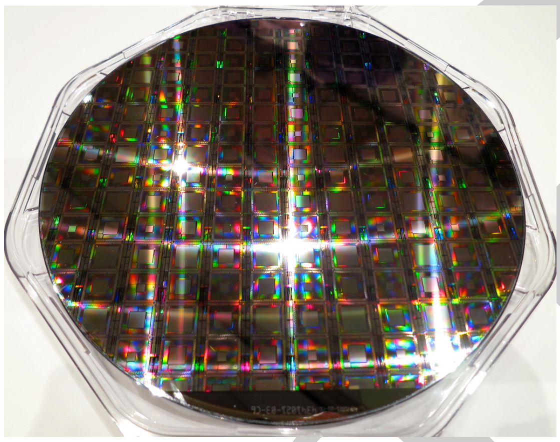
Open Problem 26 (Theory versus practice for quantum machine models)
From what we have said so far, it seems likely that algorithms using a QRAM with careful consideration of fault tolerance and the number of required qubits could be a useful abstraction for developing and analyzing quantum algorithms. How close this model is to reality remains to be seen. At least the first actual quantum computers have a lot of restrictions with respect to how quantum states can be prepared, what unitary operations are possible, which qubits can be entangled, etc. This may just be details we can abstract from as the RAM model abstracts from complications of classical processors but we do not know yet. Thus co-exploring the design space of quantum hardware, quantum machine models, and quantum algorithms is an interesting task for the future.

This will have an impact on which applications will actually emerge. There seems to be a consensus that the simulation of quantum mechanical processes will be an interesting application with an impact on physics, chemistry, and materials science. Later, quantum cryptoanalysis might have a great impact on which cryptographic mechanisms remain secure. Applications like combinatorial optimization and machine learning are also being discussed but sometimes wishful thinking or marketing considerations seem to be involved. ■

2.12.2 Quantum Annealing

We now present a more restricted but simple and powerful abstraction of quantum computing. In very general terms, the idea is to have a specialized quantum computer that can “solve” instances (of bounded size) of some NP-hard optimization problem. We use quotation marks since the machine will only “in principle” obtain optimal solutions if it can use enough time and if it is not subject to any noise. In practice, the result will often be suboptimal. An interesting implication from the perspective of theoretical computer science is that, suddenly, the seemingly academic endeavor of designing reductions between NP-hard problems becomes a constructive way of “programming” a quantum computer. Indeed,

Draft



Steve Jurvetson creativecommons.org/licenses/by/2.0/

Figure 2.41: A 512-qubit chip for quantum annealing by DWAVE.

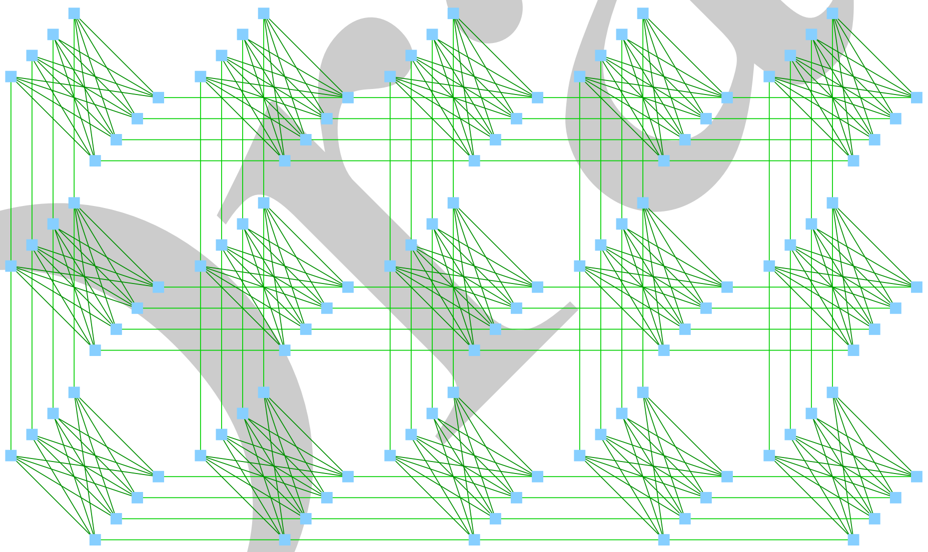


Figure 2.42: A 3×5 *chimera graph*, i.e., a 3×5 grid of bipartite 4-cliques (known as $K_{4,4}$ in graph theory). The left nodes in the cliques have vertical connections to the corresponding nodes in the neighboring cliques. The right nodes have horizontal connections.

Draft

many practical optimization problems have already been treated in this way, including graph partitioning, graph isomorphism, clustering, and machine learning [319, 490, 461, 30, 29, 206].

Open Problem 27 (Engineering reductions for quantum annealing) The methodology of algorithm engineering can help to engineer reductions. Suddenly constant factors matter. It might also be that different reductions have different properties with respect to how well a particular quantum (or classical analog) computer can solve them.

The term “annealing” stems from an analogy to the classical metaheuristic of *simulated annealing* [2] that explores the search space in a probabilistic way, see also [410, Section 12.5.2]. Indeed, also classical analog computers (see also Section 2.13.3) can be used similarly since – given enough time[37] – simulated annealing will find an optimal solution. The *quantum annealing* process [262] can more efficiently converge to an optimal solution since, besides probabilistically climbing over “mountains” of the objective function, it can also *tunnel* through them.

The NP-hard problem used in current quantum computers is the Ising spin-glass model. This problem is defined by a weighted graph $G = (V = 1..n, E)$ and the objective function

$$H(x_1,\ldots,x_n) := -\sum_{e=\{u,v\}\in E} w(e)x_u x_v - \sum_{v\in V} c(v)x_v$$

where $x_i \in \{-1, 1\}$. For a given quantum computer, the graph is fixed but the node weights and edge weights can be specified to define the concrete input instance to be solved. The most well-known quantum computers at the time of writing this section are produced by the company DWAVE and support a graph of size $n = 2048$. The supported graph is the *chimera graph* depicted in Figure 10.8. Refer to Section 2.13.3 for classical (hybrid) analog computers for solving the Ising spin-glass model.

Exercise 31 *Explain how to choose the node weights and edge weights of a quantum annealer using a graph $G = (V, E)$ such that it solves the* vertex cover *problem on a subgraph of G. The vertex cover problem asks for a minimum cardinality set*

[37] The bad news is that the best known worst-case bound for simulated annealing is no better than simply repeatedly trying solutions at random.

Draft

V' of nodes such that all edges are incident to a node in V'. Hint: as the Ising spin-glass model does not support constraints like "each edge must be incident to a selected node", convert these constraints to penalty terms *in the objective function, i.e., terms that make the solution worse if the constraint is violated. By scaling these penalties to sufficiently large values, they can enforce the constraint in an optimal solution.*

2.13 Further Unconventional Models

2.13.1 The Zoo of Turing-Complete Models

Many models of universal computation were not intended as an abstraction of a computer. Some of the most surprising ones stem from undecidability proofs for certain problems. For example, Hilbert's tenth problem asks for an algorithm that decides whether a multivariate polynomial with integer coefficients (a Diophantine equation) can take the value zero by assigning integer values to the variables. The undecidability proof [313] shows that any recursively enumerable set can be encoded as the solution set of a Diophantine equation. Similarly, any computational problem can be encoded as a Post correspondence problem – given two lists of strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$, is there a sequence of indices $i_1, \ldots, i_k$ such that $\alpha_{i_1} \cdots \alpha_{i_k} = \beta_{i_1} \cdots \beta_{i_k}$ [376].

There are also models that were defined in order to make complexity theoretic arguments. For example, *nondeterministic list processing PRAMs (NLPRAMs)* only need time $\mathrm{O}(\log t)$ to solve any problem that can be solved by a nondeterministic Turing machine in time t. Note that this implies that NLPRAMs can solve PSPACE-complete problems in logarithmic time [421] (however, by using an exponential number of processors that can work on objects of exponential size in constant time). Several such models are discussed by Vollmar and Worsch [470]. Since it is unlikely that highly theoretical models become relevant for algorithm engineering, we do not go into more detail.

The situation is different for models that were invented to formalize certain kinds of abstract computations. For example, Markov algorithms and semi-Thue systems [82, 359] describe simple rules for string rewriting that lead to a universal model of computing. Logical programming is based on resolution in Horn clauses [464, 296]. Such models are relevant for algorithm engineering if one wants to execute such computations on a real-world machine. Nevertheless, we do not go into more detail in order to limit the range of models discussed in this book.

2.13.2 DNA Computing and Molecular Computing

Molecules that can exhibit complex behavior are much smaller than transistors. For example, a nucleotide that is the basic building block of DNA and RNA has a length of about 0.3nm compared to the feature-length of VLSI chips that has currently reached 5nm. Moreover, molecular interactions take very little energy and thus can also take place in three dimensions without overheating; see also Sections 2.14 and 2.8.2. In combination, this means that computations that are directly based on molecules can support a vast amount of parallelism. For example, 10^{23} carbon atoms have a mass of only about 2g.

The power of molecular computing has first been demonstrated for *DNA computing* [5, 295]. DNA computing works with a huge number of short DNA and RNA strands (i.e., sequences of the four basic nucleotides adenine (A), cytosine (C), guanine (G), and thymine (T)) in water. By performing simple operations like copying, selection, extraction, etc. this set of strands can be manipulated in a massively parallel way. It has been shown that NP-hard problems can be solved using a small number of these steps. However, this approach has several disadvantages. Asymptotically speaking, we are only winning a (large) constant factor of *parallelism* over a more conventional machine. On the other hand, we lose a large factor concerning the *speed* of a single operation (nanoseconds versus minutes). One might even argue that this factor grows exponentially with the size of the involved strands since a DNA computing step involves an annealing process for energy minimization. Furthermore, DNA computing is prone to errors that have to be equalized by introducing a second large factor of redundancy. Finally, and more interesting for algorithm engineering, DNA computing algorithms have to use simple brute-force approaches because only a small number of simple steps can affordably be done. In contrast, conventional computing can use very sophisticated algorithms. For example, DNA computing has been demonstrated for traveling salesman problems (see also [410, Section 11.7.2]) with around 100 cities while state-of-the-art conventional solvers can handle thousands of cities [23].

However, several other approaches to molecular computing can be tried. So far, none of them succeeded [270]. For algorithm engineering, it is interesting to note that several of these failures were connected to nonrepeatable results (and in one case outright fraud). However, with Moore's law slowing down, alternatives to 2D silicon electronics will become increasingly urgent. For example, cellular automata that self-assemble (i.e., crystallize) from simple building blocks seem interesting; see also [107] and Section 2.5.7. More specifically, already growing

Draft

a crystal that implements an array of shift registers would constitute a very high-capacity memory. The algorithm theory community therefore shows continued interest in this problem with some focus on self-assembling programmable matter that has considerably more complex building blocks compared to the molecular cellular automata discussed above [65]. It is also interesting to note that pioneering papers on self-replication [353] and self-organization [458] go back to the pioneers of the conventional computer (John von Neumann, and Alan Turing, respectively).

Open Problem 28 (Practical molecular computing) Will the vision of molecular computing eventually become a practical technology? This is perhaps mainly a question of hardware design – in a very fundamental meaning involving physics, chemistry, and materials science. However, algorithm engineering may also play an important role in the task of explaining how to perform useful computations with very simple unconventional components, with appropriate abstract models, and with careful consideration of experimental methodology.

2.13.3 Analog Computing

The term "digital age" is used as a synonym for the information age. This comes from the observation that several revolutionary changes of the last decades have digital electronics as their basis. Moreover, important analog technologies like photography, movies, phonographs, telephone, radio, and television were replaced

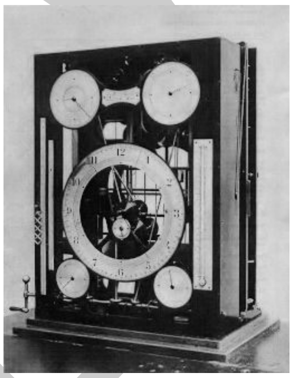
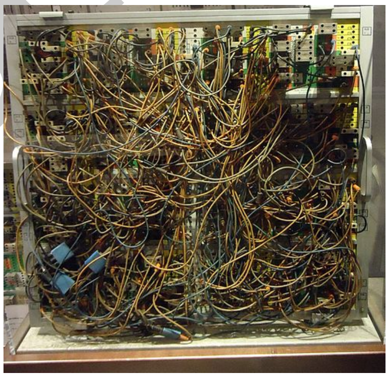



Figure 2.43: Analog computers. Left: Ferrelt's tide predictor from 1882. Right: an EAI 580 from around 1968.

Draft

by more efficient and flexible digital technologies.[38] Reducing the state of the basic components to just representing 0 and 1 makes it possible to keep these components simple and highly reliable. All higher-level functionality can then be built up from this digital basis. However, this success story is increasingly questioned since it implies a rather wasteful use of resources, e.g., switching a single bit means moving a large number of electrons; see also Section 2.14. Also, arithmetical operations like multiplication need a large number of transistors in a digital computer whereas they can be performed with surprisingly simple analog hardware . Therefore *analog computing* that directly works with continuous signals may experience a renaissance. In the past, different kinds of special-purpose analog computers were built based on mechanical, electrical, and electronic components. For example, complex electro-mechanical analog computers were used for controlling the guns of battleships beginning in the early 20th century until as late as the 1980s. Electronic analog computers were even *developed* for flight simulators until the 1970s.

An abstract model for these classical analog computers are networks of components that perform operations such as addition, multiplication, division, exponentiation, logarithms, differentiation, and integration on analog signals. Besides input signals, there are also generators for basic signals, e.g., sine or sawtooth. Analog computers can be made more flexible by integrating them with digital computers – *hybrid* computers. Interfacing between the analog and digital world uses analog/digital and digital/analog converters [226]. Figure 2.43 gives examples.

Since analog computers lack the flexibility of digital computers, it seems likely that many future uses of analog computing will be restricted to specialized functions within a hybrid computer in order to improve energy efficiency, cost, or speed. In this restricted form, the main impact of analog components on the programming model is that the precision of their computations may fluctuate. Hence, the techniques for fault tolerance discussed in Section 2.10 will become more relevant. Indeed, this is already widely used for storage and communication channels. In solid-state memory, a capacitor stores information encoded as its amount of charge. Communication channels transmit an analog signal where the information is encoded as a combination of amplitude and phase. This analog data is translated into several bits of information using discretization. To achieve high efficiency, occasional errors can be tolerated through the use of error-correcting

[38] Arguably, *telegraphy* using Morse codes was a digital technology. It used binary amplitudes and analog timing but only two delays – “long” and “short” were used.

Draft

codes and other fault-tolerance techniques.

Intensive research has been done on implementing neural networks using analog technology [425]. This is a good match because even digital implementations of neural networks successfully work with low-precision arithmetics. Also, it is natural to replicate biological neural networks using analog hardware. However, this technology is not yet used in practice since we are lacking scalable technologies for reprogramming and training analog neural networks. How exactly abstract models for analog neural information processing will look may have to wait until it is clear which kind of technologies will actually work in practice.

Open Problem 29 (Biology meets machine models) However, already now, we could consider computer-science style abstract models for biological neural networks with an eye on artificial neural networks. This closes the loop to the pioneers since Turing machines were intended as an abstraction of the operations performed by a human mathematician.

One can consider analog computers for solving NP-hard problems as in Section 2.12.2. This also makes sense with machines using classical computing. Indeed, several designs for the Ising spin-glass model have been considered. For example, Inagaki et al. [251] propose a hybrid machine that evaluates the objective function using a digital computer. This machine can handle quite large, fully connected instances. It is very much an open question whether quantum annealers, classical analog machines, or traditional digital computers will win this race [222]. With respect to quality, digital computing seems to be ahead [251]. Quantum machines can currently only score for instances that are easy to embed into the underlying network, e.g., a chimera graph.

2.13.4 Neural Networks

Neural networks are a wide field that is not directly related to machine models. Thus we do not intend to give a comprehensive survey here and rather refer to textbooks on machine learning e.g., [212]. Here, we only want to mention some relations to abstract machine models.

In its most simple (feed-forward) form, a neural network can be abstractly modeled as a circuit that implements a vector-valued function $f(\mathbf{x}, \theta)$ where $\mathbf{x}$ is an input vector, and where θ is a parameter vector. See Figure 2.44 for a visualization. A numerical optimization algorithm is used to adjust θ in such a way that inputs are mapped to "desired" outputs. In *supervised learning*, this optimization

Draft

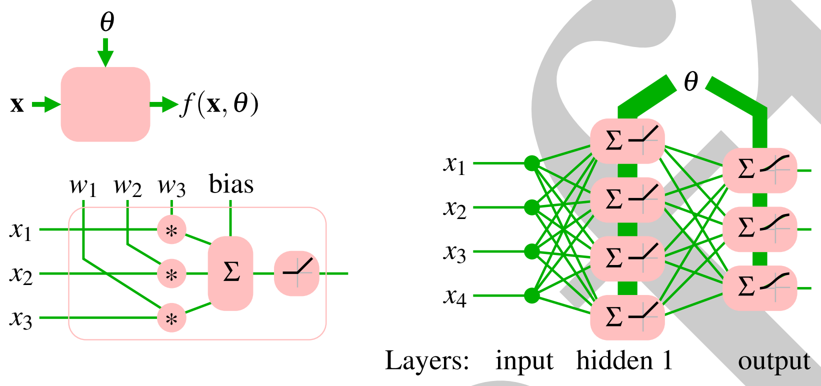


Figure 2.44: A feed-forward neural network in its most abstract form (top left), a 3-input artificial neuron with ReLU activation function (bottom left), and a 3-layer network with 4 inputs, 5 neurons in its single hidden layer, and 3 output neurons. The neurons in the hidden layer have a ReLU activation function while the output neurons use a logistic activation function.

is based on training examples that provide inputs and desired outputs. The optimization algorithm takes into account how well the output matches the desired output and how well the network is expected to generalize to unseen inputs. The values propagated along edges in the circuit are conceptually real numbers – using low-precision representations in practice. Typical computational nodes multiply values with components from θ (weights), add values, or transform a value in a nonlinear way (activation function). For example, a simple and successful activation function is $a(x) = \max(0,x)$ – the *rectified linear unit (ReLU)*. Another example is the logistic function $a(x) = 1/(1+e^{-2\beta x})$ where the additional parameter β controls the steepness of the function. Often, the network consists of multiple layers. Each node v in a layer computes a weighted sum s of a subset of output values provided by the previous layer. The weights are components of θ. Node v then provides the output value $a(s)$ to the next layer using a nonlinear activation function a. Neural networks can approximate more or less arbitrary continuous functions even with only two layers (one hidden layer). However, this is not a constructive result and the hidden layer may have to get very large.

Various generalizations are possible. For example, there can be feedback of output values into inputs used in subsequent steps. With this generalization, neural networks become a universal model of computation [435], i.e., they can simulate

Draft

Turing machines.

For a fixed value of θ, a neural network can be viewed as a circuit in the sense of Section 2.8. Circuits describing computer hardware typically use other data types, basic operations, and architectures but the general concept is the same. The more important difference is that the designer of a neural network specifies only the basic architecture and then an optimization process on θ determines the actual behavior of the network.[39]

Exercise 32 *Consider a neural network with a inputs, o outputs and k layers of size n. Analyze the asymptotic number of arithmetical operations needed to evaluate it. What is the size of its parameter vector θ? Discuss the possible cost savings of the following tuning measures using asymptotic considerations (and always assuming that the task performed by the network is still done in a satisfactory way): Reducing the precision of the calculations; pruning some layers to have fewer neurons; making the network more* sparse *by allowing only* $\mathrm{O}(\sqrt{n})$ *outputs of each hidden neuron.*

2.14 Energy Consumption and Other Resources

Most of what we say in this chapter focuses on modeling execution time. Several other resources are important. *Space consumption*, *I/O volume*, or *communication volume* are often considered. The number of *random bits* needed is sometimes used in theoretical research because it also allows us to infer interesting tradeoffs.

However, *energy consumption* is arguably even more important than execution time [382] since it is the limiting factor in mobile applications, because it has a growing part in the overall cost of computations, and because it is more directly related to the environmental impact of computing. Energy consumption is also a fundamental link between physics and computing. In this section, we first consider pragmatic aspects of energy consumption in current architectures – Section 2.14.1. Then Section 2.14.2 has a more fundamental look at energy consumption in possible future architectures.

[39]Actually, also *hyper parameters* determining the architecture, e.g., number of layers, size of layers, etc., can be found using an optimization process.

Draft

2.14.1 Energy Consumption in Current Architectures

One reason why energy consumption is often not directly considered is that it is highly correlated to execution time in sequential computations. If you reduce execution time by a factor x you also reduce the energy consumption of the computer by factor x if you assume that the machine runs with constant power (i.e., energy consumption per unit of time). It is known that there are deviations from this assumption but they are quite complicated. For example, heavy use of vector instructions costs energy. Memory accesses/communication (in particular off-chip) are more expensive than other operations. On the other hand, a cache fault often makes the processor wait for the result which reduces power – time remains a good measure here. Hence, what really reduces the correlation between time and energy is when the processor successfully overlaps many independent memory accesses and internal computations.

When we tune a program to reduce execution time by keeping more pieces of the hardware busy, we may reduce total energy consumption because the idle power of unused components is saved. Albeit, the savings will be less than what we would expect with a constant power assumption. On the other hand, accelerating a program by doing more overall work may waste energy.

An interesting issue is the influence of clock frequency on energy consumption. Increasing clock frequency makes the computation faster but costs more energy. For example, for tasks with a deadline, one can ask what clock frequencies one should choose to meet all deadlines while minimizing energy consumption. The increase in energy consumption is superlinear in the clock frequency since increasing clock frequency also requires increased voltage. This is further complicated by bounded maximum frequency, discrete steps, and parts of the system (e.g., memory subsystem) that have a fixed frequency. All of this is much more complicated than the simple models used in theoretical scheduling papers [45, 12, 416, 382].

This becomes even more complicated with parallelism. Processor cores can run at different clock speeds. In principle, one can save energy by using many cores with small clock frequency rather than few fast cores. This is particularly true if cores consume energy even if they are not used. On the other hand, parallelization overheads easily eat up possible savings. Similarly, communication channels can run at different bandwidths and thus further broaden the spectrum of possible strategies. Thus, specifying concrete models for energy consumption is complicated. However, let us now try to formulate concrete models. A component running at speed s will require power $\alpha + f(s)$ where f is a monotonously

Draft

growing function. Moreover, the maximum speed is bounded. Concrete forms for f could be a piece-wise constant function or a function proportional to s^γ for some constant $\gamma > 1$. We can also count different operations separately that are known to cost significant energy, e.g., floating point multiplications, or main memory accesses.

Exercise 33 *Find a way to perform power measurements on one of your computers. There are both software solutions (e.g. powertop on Linux) and cheap hardware solutions. Now measure power consumption in different modes: idle, performing light interactive work. Run a sequential algorithm (e.g., sorting). Run a parallel algorithm performing the same task using a variable number of cores. Which configuration is most efficient? Explore further situations, e.g., with and without GPU use, memory-bound tasks versus compute-bound tasks,... Summarize your findings.*

Open Problem 30 (Energy-efficient AE) Overall, AE for energy-efficient algorithms is a wide-open field. For example: What are appropriate models for energy consumption? Where do energy-efficient algorithms make a real difference compared to fast algorithms? What are realistic models for energy-efficient scheduling? When/how does parallelization help? Pruhs [382] gives several interesting open theoretical problems with important links to practice. However, we believe that AE engineering can address a much wider range of problems with an even larger impact on practical solutions – constant factors matter, make experiments, etc.

2.14.2 Energy Consumption in Future Architectures

The increasing importance of energy consumption is one of the main driving factors for considering some of the unconventional models of computation considered above. The reason is that the prevalent model of accurate, highly reliable digital computers comes at the cost of a large amount of redundancy – switching a transistor means moving a large number of electrons. Molecular computing (Section 2.13.2) or analog computing (Section 2.13.3) lifts some of these assumptions and thus could be much more energy efficient. This comes at the price of lower precision and higher failure rates. Hence, the fault tolerance techniques discussed in Section 2.10 are likely to become more relevant. The same happens when traditional semiconductor technology is operated in a regime with maximum energy efficiency [157].

Draft

In designing future, energy-efficient computer architectures, the fundamental physical connections between computing and energy consumption will become more important; see also Section 2.8.2. In particular, there is a fundamental limit of dissipating energy of at least $kT\ln(2)$ Joules of energy for erasing one bit of information at a temperature of T Kelvin where $k = 1.380649 \cdot 10^{-23} J/K$ is the Boltzmann constant [283] (the *von Neumann–Landauer limit*). Hence, any irreversible computation is associated with this energy consumption. This is several orders of magnitude away from the current state of the art but approaching this limit increases the likelihood that *thermal noise* leads to computation errors. Very interesting is the "loophole" that we can, in principle, do any computation in a reversible way [61].[40] Then, computations can, in principle, be arbitrarily energy efficient. There is also evidence that reversible computing can already help save energy long before the von Neumann–Landauer limit is reached [186]. Also note that the quantum part of a quantum computer requires reversible computations; see Section 2.12. However, energy-efficient reversible computing is also interesting for future classical machines.

Open Problem 31 (Engineering reversible algorithms) Reversible algorithms have not been intensively investigated by the algorithms community so far (but see [137, 462]) and the perspective of AE may give additional opportunities to arrive at interesting approaches that may prove useful for future energy efficient computer architectures.

2.15 Summary and Outlook

We have seen a large number of abstract models of computation. One might be tempted to say that there are too many of them. However, the requirement to have simple models also implies that one needs a significant number of models in order to illuminate different aspects of a problem. Still, some model variants explained above may be interchangeable. Also, some of the models might serve a particular purpose well but some results developed for them might overexploit simplifying assumptions of the model so that they are far from being useful in practice.

It should also be noted that even for developing and analyzing a particular algorithm, several models can be useful. For example, consider *super scalar sample sort* [420, 38] – one of the best comparison based sorting algorithms in practice;

[40] The other loophole – reducing temperature – is problematic because the costs for cooling may eat up the savings within a cooled computer.

Draft

see [410, Section 5.13]. One might analyze the impact of the splitter selection strategy by just counting comparisons. Its main feature is a way to avoid branch mispredictions; see also Section 2.2.5. The algorithm is also cache efficient, which can be analyzed in the external memory model of Section 2.3. One can then go on to analyze additional aspects like associativity misses [326], TLB misses, or memory traffic. The shared-memory version [38] can be analyzed in a PRAM model, perhaps using a realistic one like aCRQW PRAM (see Section 2.4.2) that illuminates access contention. Main-memory traffic of that algorithm can be discussed using the PEM model of Section 2.4.6. One can then go on to consider NUMA effects using a hierarchical shared-memory model or a distributed-memory model.

An interesting question is how models of computing will develop in the future. The one safe bet is that we will see many further models. While some models may fall into disuse because they are awkward or too removed from reality, new models will be needed to cover the more diverse landscape of actual computers. The push for more (sustainable) performance makes *parallel* processing and *energy efficiency* a prevalent issue. This strengthens more *physically realistic* models and moves *fault tolerance* and *analog computing* into the mainstream.

Whether *quantum computing* will become widely useful depends on future technological breakthroughs, in particular with respect to error correction and large-scale entanglement. If this eventually works, a likely outcome is that it is used for solving relatively small instances of selected hard problems, while solving simpler problems on large data sets is still cheaper with classical computing[41]. However, an interesting scenario would be a disruptive technology that allows large-scale quantum computing and makes it *more* energy efficient than classical computing. We would then have to reinvestigate many algorithmic problems in a reversible, fault-tolerant, and quantum-parallel manner.

Even if quantum, analog, fault-tolerant, or physically realistic computing takes more and more room, we believe that the abstraction of a reliable, classical, digital computer is too attractive to drop it entirely. For more and more problems, the bottleneck is not computing performance but the ability of (possibly AI-assisted) humans to design and implement the required software. The sweet spot is likely that most application engineers use the above conservative models but that they will design their algorithms such that they work in parallel and such that they can use a basic toolbox that encapsulates aggressively tuned algorithms based on more low-level models. Of course, the scenario of super-human artifi-

[41] It is also hard to imagine a smartphone whose processor is cooled to near absolute zero using battery power.

Draft

cial intelligences that design software and hardware themselves would remove the main prerequisite of the above argument. Then it is philosophically interesting to speculate whether even those machines would find our current abstractions useful but it might no longer be practically relevant.

Draft