Introduction to Parallel Computing

Gianfranco Bilardi

Department of Information Engineering, University of Padova

Spring 2020

1 Parallel Computing: motivation and context

The central goal of computer science and engineering is the *automatic solution* of *computational problems*. At the logical level, the automatism is provided by an *algorithm*, a finite description of a computational process which, if fed with any (of the generally infinitely many) inputs of the problem, produces a corresponding output. At the physical level, the automatism is achieved by a *computing machine*, a system capable of executing the computational process specified by the algorithm, on any given input.

In general, an algorithm specifies the computational process in terms of "elementary" operations which act on data and produce other data. Naturally, it is of great interest to execute the process as fast as possible. The basic, intuitive idea behind *parallel computing* is very simple: reduce the time of a computation by executing concurrently as many operations as possible. In spite of its simplicity, the full implementation of this idea has proven rather challenging and has required the development of a sophisticated conceptual framework, within several areas of computer science and engineering. This development is still in progress, although spectacular results have been achieved. This course will introduce the main ideas in the framework of parallel computing, which have enabled such results. In the reminder of this section, we provide a little context on the degree of parallelism of current computer systems and on the applications of computing that motivate further progress in the field of parallel computing. In the next section, we will take a closer look at the reasons why the exploitation of parallelism has required significant conceptual and engineering advances and will provide a first overview of the course topics.

PFlop/s = petaflop per second = 10^{15} floating point ops per second

R_{max} = maximum number of PFlops reached

R_{peak} = theoretical limit of PFlops

1.1 Today's supercomputers

To get an overview of the most powerful computers in the world, one can consult the list of the top machines at `https://www.top500.org`, which is updated twice a year, in June and in November. The top 500 supercomputers are all parallel systems with a number of processing elements (cores) ranging from tens of thousands to several millions. As an indication of the state of the art, in the November 2019 list, the top entry is *Summit*, by IBM, built also on NVIDIA and Mellanox technology. This machine (i) has 2,414,592 cores; (ii) executes about 148 petaFLOPS (1 petaFLOF=10^{15} floating point operations per second) on the LINPACK benchmark (a linear algebra library); (iii) has 200 petaFLOPS of theoretical peak performance; (iv) consumes energy at the rate of about 10MW; (v) is the fifth most efficient machine in the GREEN500 list, with about 15.7×10^{9} FLOP per Joule (or, FLOPS per Watt); (vi) covers an area of about 1,000 m^2; (vii) is connected by 136 miles of cabling; (viii) cost around \$200 million; and (ix) is intended for applications in the areas of cosmology, medicine, and climatology.

The race to build more powerful machines is still very active, with the goal of reaching and surpassing the exaFLOPS barrier (10^{18} floating point operations per second). ↳ (reached)

1.2 Who needs more computational power?

Today, information processing plays a crucial role in science, engineering, business, government, etc... The recent progress in areas such as Big Data Analitics and Machine Lerning would not have been possible without the availability of substantial computing power. However, in several areas, the computational requirements of certain problems exceeds the power of current supercomputers. Some examples are given below, just as an illustration.

1. **Fundamental physics.** Current knowledge about fundamental particles and interactions is captured by three theories: quantum electrodynamics (QED) for the electro-weak interaction, quantum chromodynamics (QCD) for the strong interaction, and General Relativity (GR) for the the gravitational interaction. In QED, there are quantities for which the value computed from the theory agrees with the one provided by the experiment better than a part in a billion. In contrast, an important quantity such as the mass of the proton is currently computable from QCD with an accuracy of less than 1%, while it is known experimentally with a relative standard precision of 3.1×10^{10}. Without more powerful machines and/or more efficient algorithms, science

becomes limited in the very basic capability of comparing theory and experiment.

2. **Cosmology.** To study various hypotheses on the origin and structure of the universe, including the fundamental theories mentioned in the previous point, it would be of great interest to simulate its evolution. Condidering that the known universe is estimated to contain of the order of 10^{11} galaxies, it is not surprising that only extremely coarse models are computationally tractable today. It is hoped that more computing power will help our understanding of the universe.

3. **Biology.** Spectacular improvements in DNA sequencing technology are making a wealth of data available, but also require substantial amounts of computing to extract useful knowledge from the data. A gene in the DNA determines the chemical structure of a corresponding protein, as a sequence of amino acids. In many applications, such as drug design, one would like to know the three-dimensional structure of the protein. Computing this structure from the chemical formula is in principle possible by simulating the protein as a dynamical system, but the simulation requires substantial time even on the top supercomputers, especially for proteins with many atoms and long folding time. In this context, an interesting approach is represented by the project Folding@home, promoted by researchers at Stanford University, which has assembled a distributed supecomputer accessing personal computing resources made available by many people on their own platforms. The computing power today is nearly 100 petaFLOPS, and is deployed to study protein folding and other biological, medical, and farmaceutical problems.

4. **Neuroscience.** The study of the nervous system and, in particular, of the brain poses great computational challanges. One very ambitious objective is the simulation of an entire human brain. Considering that the brain contains about 10^{11} neurons, connected via 10^{14} synapses, commuting thousands of times per second, even assuming a minimal binary model for the neuron state, quickly leads to exascale simulation requirements, which may become heavier depending on the level of detail to which individual neurons may need to be simulated. It ought to be highlighted that the detailed stucture of the human brain is far from being known today, hence the staggering computational requirements are just one of the obstacles to be overcome (unlike cases like QCD, where the equations to be solved are precisely known). However,

neuroscience is arguably the most important frontier of human knowledge, and does provide strong motivation to strive for more powerful computing systems.

5. **Artificial intelligence and machine learning.** In 1951, Alan Turing - perhaps the most influential figure in computer science - predicted that in 50 years intelligent robots would be an integral part of human society. (Arthur Clark has explicitly acknowledged that the choice of 2001 as the year for his Space Odissey is linked to this prediction: 1951+50=...). Today, computers are better than humans on carring out many specific tasks; however, general-purpose artificial intelligence is generally regarded as (considerably) inferior to human intelligence. Whether better models, algorithms, and machines will eventually lead to artificial systems with general intelligence superior to that of biological beings is a widely debated questions. Whether computation can generate conscious experience is an even more difficult question. But there is little question that increased computational power will play a key role in the realization of increasingly sophisticated forms of machine intelligence and perhaps machine consciousness.

6. **Cryptography.** Several contemporary cryptographic codes are based on a private key and a public key. Mathematically, the private key is uniquely determined by the public one. Therefore, the security of the code crucially depends on the circumstance that (at the present and at the forseeable state of the art) the most efficient known algorithms run on the most powerful available computers cannot compute the private key from the public key in a useful time. Since the developement of new algorithms and/or new computers could dramatically change the state of the art, high performance parallel computing may represent at the same time a threat and an opportunity in the field of cryptography. This scenario is not just hypothetical: efficient quantum algorithms are already known for certain computational problems that are at the heart of some cryptografic methods widely used today and we are currently witnessing advances in the realization of quantum computers. For this reasons, considerable attention is being currently devoted to alternate codes, which seem less prone to a quantum "attack"; this area of research goes by the name of post-quantum cryptography.

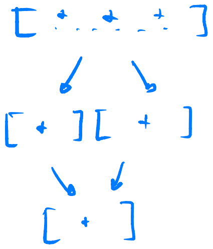
2 Challenges of Parallel Computing

Systematically pursuing the idea of parallel computing requires a rethinking of various parts of computer science and engineering. In this section, we provide an overview of the main issued to be confronted.

2.1 Parallel Algorithms

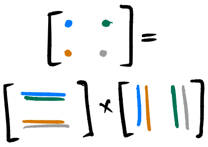
To effectively utilize many computing resources, an algorithm must enable the execution of many operations at once. Not all algorithms enjoy this property.

The simple algorithm which adds N numbers $a[1], \ldots, a[N]$ by first setting $S := a[1]$ and then executing $S := S + a[i]$, for $i := 2, \ldots, N$, has basically no opportunity for parallelism. In fact, each iteration of the loop can only be executed after all the preceeding ones have completed. However, it is not difficult to develop a highly parallel algorithm for the same problem (see Problem 2.1.1).



There are problems for which designing a parallel algorithm does require some ingenuity. For example, consider the "standard" algorithm that merges two sorted sequences by repeatedly extracting and placing in the output the minimum of the minima of each sequence (until one of the two sequences becomes empty). Clearly, the extractions must occur one after the other, with one comparison at the time being executed. How we will see in a later chapter, there are efficient algorithms for merging, but they require new ideas for design and analysis, with respect to the standard, sequential algorithm. (See Problem 2.1.2 for a modest parallelization of standard merging.)

On the other end of the spectrum, there are algorithms which, although initially adopted for sequential computing, naturally present an abundance of parallelism. Take, for example, the standard algorithm to multiply two matrices A and B to obtain $C = AB$. Since each element of C is the inner product of a row of A by a column of B, the computation of all elements of C can be carried out concurrently, with a degree of parallelism equal to the size (number of elements) of C (see Problem 2.1.3). The computation of each inner product could be further parallelized, although there is seldom the need to do so in practice.



In the early stages of investigation of parallel computing, there was some skepticism about its general applicability, based on the possibility that many computational problems of interest may not admit (significantly) parallel algorithms. This skepticism has turned out to be unjustified; in fact, efficient parallel algorithms have been discovered for virtually all computational problems of practical interest. This historical development naturally raises the question whether all computational problems are actually solvable by highly parallel algorithms. Of course, before searching for a mathematical answer, this question has to be formulated more specifically and more rigorously. We briefly mention one such formulation, known as the **P** versus **NC** question, which has received considerable attention form a theoretical perspective.

P is the class of computational problems that can be solved by a sequential algorithm with computation time polynomial in the input size N, that is, $T = O(N^k)$, for some positive integer k. **NC** is the class of computational problems that can be solved by a parallel algorithm with computation time polylogarithmic in the input size N, that is, $T = O((\log_2 N)^h)$, for some positive integer h, using a polynomial number of processors, that is, $P = O(N^g)$, for some positive integer g[1] . It is quite straightforward to see that **NC** $\subseteq$ **P**. However, at the state of the art, it is not known whether **NC** $\subset$ **P** or **NC** $=$ **P**. If the latter were the case, then the entire class **P** would be highly parallelizable. Like **P** versus **NP**, one of the seven Millennium Prize Problems of mathematics, **P** vs. **NC** is a very hard question. There is a class of problems, called **P**-complete, such that if any one of them is in **NC** then **NC** $=$ **P**. An example is the *Circuit Value Problem (CVP)*: Given a Boolean circuit, the inputs to the circuit, and one gate in the circuit, calculate the output of that gate. (See Problem 2.1.4.) (See also the Wikipedia entry "P-complete" for a list of other interesting **P**-complete problems.)[2] It is worth observing that achieving polylogarithmic time (with polynomially many processors) is a very ambitious goal. Even if this goal has not been achieved (and might not be achievalble) for **P**-complete problems, algorithms with non trivial amount of parallelism are nevertheless known for several of them. Unfortunately, in this course, we will not have the time to develop this interesting chapter of parallel computational complexity. However, the moral of the **P** vs. **NC** question is that, thus far, establishing non parallelizability has proven much harder than showing parallelizability.

matrix mult
with as many
processor as
the size of the
matrix requires
linear time and
not log time...

Summary. Known algorithms have to be reconsidered to evaluate their degree of parallelism. If the degree of parallelism is not satisfactory, new

[1] Curiosity: while the familiar notations **P** and **NP** have a "technical" motivation, respectively denoting deterministic and nondeterministic polynomial time, the name **NC** was proposed by Stephen Cook (winner of the 1982 Turing Award for his contributions to the theory of NP-completeness) as a mnemonic for "Nick's class," in recognition of Nicholas Pippenger, who introduced this class in 1979.

[2] The CVP is different from, but related to Boolean Circuit Satisfiability: Given a Boolean circuit and one gate in the circuit, determine whether there is an assigment of values to the inputs of the circuit such that the chosen gate outputs the value 1. Boolean Circuit Satisfiability is NP-complete, as proven by Stephen Cook in 1971.

algorithms have to be designed, with the parallelism as an explicit goal. Several decades of research have shown that most computational problems of interest admit significantly parallel algorithms.

2.2 Analysis of Algorithmic Parallelism

In the previous section, we have appealed to the intuitive notion of degree of parallelism. In a later chapter, we will provide a formal counterpart to this intuitive notion, which will give a quantitative measure of the degree of parallelism of a given algorithm.

Divide and conquer is the main paradigm even in the design of parallel algorithms. The running time of these algorithms, assuming the availability of a certain number of processors, can be typically obtained by formulating and solving a recurrence relation, much as for sequential algorithms. In formulating the recurrence, we just need to properly take into account which subproblems can be managed concurrently. For example, to add $N > 2$ numbers (for convenience, a power of two), we can divide them into two subsets of size $N/2$, separateley and concurently add each subset, and finally add the sums of the two subsets. If $N = 2$, simply add the two numbers. The corresponding recurrence relations is

$$T(N) = T(N/2) + 1, \quad (1)$$

$$T(2) = 1. \quad (2)$$

By standard techniques, we obtain $T(N) = \log_2 N$. Sequentially, we would have written $T(N) = 2T(N/2) + 1$, with solution $T(N) = N - 1$, because there are two subproblems. But when these two subproblems are solved in parallel, the time for two is the same as the time for one (given enough processors, which we are assuming to be the case.)

One fear that was often reported in the early stages of parallelism was that, even when parallel algorithms are available in principle, it might be difficult for people to "think parallel". This fear has also proven to be unwarranted. When we think about sequential algorithms, we do have to think about multitudes of operations corresponding to multiple iterations of a loop or to multiple instances of a recursive call. In a parallel context, we have to learn to recognize conditions under which loop iterations or procedure calls do not produce data for each other, hence they can proceed concurrently. Although this aspect has to be confronted, a modicum of reflection and practice is typically sufficient to manage it.

Summary. Analysis of parallelism in algorithms does require some adaptation of the standard divide-and-conquer and recurrence-relation machinery.

In general, this adaptation is relatively simple and can be mastered with a bit of practice.

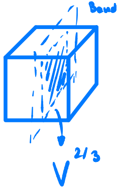
2.3 Communication Requirements

A significant complication that does arise in parallel computing is instead connected with the cost of data movement. Larger computing systems imply larger data movement costs. To gain some preliminary intuition about this issue, consider a physical machine which is laid out in three-dimensional space. In a fixed technology, if laid out most compactly, a machine with P processors will occupy a region with a volume $V = \theta(P)$. Then, the average distance between two processors will be $\theta(P^{1/3})$ and the bandwidth between two suitably chosen halves of the machine will be $\theta(P^{2/3})$. Both the distance and the bandwidth constrains imply that a parallel step where P operations concurrently access their operands from random locations in the machine will take $\Omega(P^{1/3})$ time to execute[3].

Information

$$T \geq \frac{I}{B} = \frac{V}{V^{2/3}}$$

Bandwidth



If every step of a P-processor machine were to actually take time $\theta(P^{1/3})$, the resulting performance would be of $\theta(P^{2/3})$ operations per step. Execution would still be faster than with $P = 1$, but the utilization of processing units would asymptotically vanish, proportionally to $1/P^{1/3}$. Fortunately, most computations of interest do not access data in a completely random way. In general, with a judicious choice of where and when each operation is executed and its result is stored, a much better performance can be achieved.[4] (See Problem 2.2.2).

The judicious choice we have just mentioned is the root of the complication. In fact, to optimize this choice, the algorithm designer has to consider not only which operations provide operands to a given operation, to maximize the parallelism of the schedule, but also the time-space aspects that influence the distance and the bandwidth requirements, to efficiently move data from the producing operations to the consuming ones. Designing a communication-efficient schedule for an algorithm adds a significant burden

[3] Rigorous arguments will be developed in later chapters. Intuitively, time must be at least proportional to the $\theta(P^{1/3})$ distance, due to the physical constraint that information cannot travel faster than light. (This limit holds in all fundamental theories of current physics, that is, QED, QCD, and GR.) Furthermore, in a statistically random access step, half of the operands in one half of the machine will come from the other half of the machine, whence $\theta(P)$ messages must traverse the $\theta(P^{2/3})$ bandwidth between the two halves; time is at least information over bandwidth, hence $\Omega(P^{1/3})$.

[4] We can guess that this must be the case for the LINPACK benchmark on a machine like Summit, with millions of processing elements, otherwise 75% efficiency would not be achievable.

to the development of efficient parallel algorithms, when compared with sequential algorithms[5].

Designing communication efficient algorithms becomes even more difficult for the following reason. The best operation schedule is significantly influenced by the structure of the machine on which the algorithm will be executed. Since machines, both those that have been proposed in the literature and those commercially available, exhibit different structures, it becomes very hard to achieve portable performance, that is, to design software that is simultaneously optimized for different machines.

Summary. As we build larger machines, the running time of algorithms is increasingly dominated by data movement, due to increasing distances between different parts of the machine and to a bandwidth that grows sublinearly with the number of processors. To maximize performance, the communication requirements must be optimized as part of the algorithm design and implementation. This adds a level of significant complication, since the time-space mapping of the computation has to be explicitly dealt with. This is a major theme in parallel computing.

2.4 Models for Parallel Computation

Models of computation play an important role in informatics [6]. For example, *Turing Machines* (TMs, 1936) have provided the basis for a rigorous definition of the concept of algorithm, for the discovery that there are mathematically well-defined problems that are not solvable by any algorithm, for the formulation of the notion of *Universal Computer*, programmable to execute any conceivable algorithm. TMs also support the analysis of time complexity (number of elementary machine steps or state transitions) and space complexity (number of tape cells reached by the machine head) [7]. However, by

[5] This statement is not completely accurate. Even in machines with just one processor, the cost of data movement across the different levels of the memory hierarchy (registers, caches, main memory, solid state disks, and magnetic disks) has a significant impact on performance. For example, on many types of processor, for (dense) matrix multiplication, a block-recursive implementation can easily give two orders of magnitude performance improvements, with respect to a straightforward triple-loop implementation, by substantially reducing data transfers between the memory levels. The two implementations do execute the same operations, but in different order.

[6] For on overview of the subject, one may consult the book by John Savage, *Models of computation. Exploring the power of computing.* Addison-Wesley (1998).

[7] Interestingly enough, even the parallelism of a computation of a (sequential, one tape, one head) TM can be estimated considering the number of times when the head changes its direction of movement. This number is called the *reversal complexity.* The degree of

and large, the analysis of sequential algorithms has been developed pon the model known as *Random Access Machine* (RAM), which more closely captures the nature of the one-step instrucions of a typical processor, as well as the addressable nature of its memory system. Several popular programming languages, like FORTRAN (1952-57), C (1972), C++(1985), Java (1995), Pyhton (1991), etc..., implicitly assume a virtual machine in the spirit of the RAM.

Defining a model of parallel computation that can serve as a general basis for complexity theory, algorithm design and analysis, and programming languages has proven a formidable challenge. As of 2020, a standard model has not yet emerged. Rather, several different models are in use, each useful for some purposes, but not adequate for others.

One way to understand the plurality of possible models is to consider that computation includes two dimensions. First, a *logical dimension*, concerned with how the input values are combined to obtain new values, and so no, until the desired output values are obtained. Second, a *physical dimension*, concerned with where and when the values are produced, stored, and moved. Clearly, a real computing platform has to deal with all the physical aspects of the computation. In principle, it is possible to build systems capable of taking a purely logical specification of the computation (for example, provided in a functional language) and automatically manage the physical resources (processing elements, storage locations, communication channels) to ensure the correct execution of the logically specified computation.

Significant automatic management of resources does indeed take place in current systems. For example, in nearly all programming languages, storage locations form a virtual space. During program execution, the machine hardware and the operating system map virtual locations into physical locations, in a dynamic way (the map evolves in time) that is completely outside the program control[8].

parallelism can be estimated as time complexity divided by reversal complexity, that is, by the average number of steps between two consecutive changes of direction. As it turns out, the actions of a TM in a period of time during which the head moves always in the same direction can be simulated very efficiently in parallel. The ideas behind this simulation are far from trivial; they will be explored in a later chapter, as an application of *prefix computation*, a simple and general type of problem that can be defined for any associative operation.

[8] For example, policies implemented in hardware decide which locations are mapped to the first level cache, whereas other policies implemented within the operating system decide which pages (sets of consecutive locations in virtual space) are mapped to main memory. In general, programming languages do not include instructions to tell caches or main memory what to do. Of course, this does not rule out the possibility that a programmer who knows the policies of a system can optimize the code accordingly.

Unfortunately, the complexity of resource management increases with the size of the machine, since larger machines have more resources. Furthermore, efficient resource management for a given algorithm is often enabled by an understanding of the structure of the executions of the algorithm, a structure that can be characterized mathematically, but that cannot be automatically extracted from the program, at least not at the state of the art[9].

In the outlined context, a number of different models of parallel computation have been proposed, with different tradeoffs between ease of describing algorithms, efficiency of the execution, and portability of performance across different platforms. In general, adding explicit aspects of resource control make the algorithm more difficult to describe, but also leads to more efficient algorithms. There are models, such as functional languages, which only deal with the logical level of the computation, in terms of values and operations. Other models explicitly introduce the concept of memory as a set of shared locations that can hold values; operands of operations are specified in terms of the corresponding memory locations; control structures allow to specify which operations can proceed concurrently. The Parallel Random Access Machine (PRAM) assumes a shared memory and a set of processors; a PRAM program does specify which processor executes which operation and at which step. Other models explicitly account for the fact that memory is partitioned into modules and that only locations in different modules can be accessed concurrently. These models can also be refined by introducing a network with a specified topology that connects processors and memory modules; a memory access must then be routed through the network, incurring suitable delays. Finally, there are models that explicitly capture how computing system is embedded in two-dimensional or three-dimensional space, thus enabling a proper evaluation of the area or volume taken by the wires (which impact the cost of the machine) as well as the length of the wires (which impact transmission time).

Summary. A spectrum of models for parallel computation arises from different decisions about which resource management aspects to expose at the level of algorithmic description. These models tend to realize different tradeoffs bewtees ease of programming, performance, and portability. No single model has achieved universal acceptance, which makes the field both more

[9] Substantial research effort has gone into the field of *optimizing compilers*, to try to automatically restructure programs so that they run more efficiently. Several good results have come out of this effort, but the general optimization problem is far from being satisfactorily solved by compilers, in part because program execution is influenced by properties of the input, typically not known at compile time, and in part because even the properties that hold for all inputs are quite difficult to determine.

complicated and more interesting. It is important to master at least a few different models.

2.5 Parallel Architectures

It is natural to view the degree of parallelism of a machine as roughly captured by its number of functional units, which can concurrently execute different operations. However, the machine will also need memory modules and a network that interconnects the various processing and the storage components. But what is the "best" overall organization for a parallel machine?

We have written the word "best" in quotes, because the very metrics by which the machines are to be evaluated requires careful reflection. A way to approach the issue could be to say that we interested in the machine with maximum *performance* (in the sense of minimum computation time) among all machines of a given *cost*.[10]

But how do we measure the cost in a reasonably abstract way, to develop a theory that can provide guidelines robust with respect to the technological and economical factors, which tend to evolve quite rapidly? Two basic approaches have been pursued in this respect: (i) measuring the cost by the number of processing elements, and (ii) measuring the cost by the amount of space occupied by the machine (area or volume)[11]. Both metrics will be used in subsequent chapters.

Another question is: "best" at what task? A priori, it is possible that different machines are optimal for different computational tasks? Especially in view of the fact that different algorithms generate different patterns of data communication, it is quite likely that they are best served by different

[10] In a dual way, one can be interested in the smaller cost for a given performance. (See Problem 2.5.1.)

[11] In principle, it is desirable to make full use of all three dimensions of physical space, both to reduce distances and to increase bandwidth. However, the full use of the third dimension runs into obtsacles of various nature. First, both integrated-circuit technology and printed-board technology are currently essentially planar. Second, energy dissipation poses problems in a three-dimensional machine, say with the shape of a cube or a sphere, since energy is consumed at a rate proportional to the volume V, whereas the resulting heat can only be extracted at a rate proportional to the area A of the bounding surface, which grows more slowly, i.e., $A = \theta(V^{2/3})$. Third, in asymptotic terms (not very relevant to engineering for the forseeable future, nevertheless intellectually interesting), one could not scalably amass processors and memories at a constant density in three dimensions, because eventually the machine would collapse due to its own gravitational field and become a black hole. The curious reader is referred to the *holographic principle*, a conjecture proposed by Gerard t'Hoft (in 1993), according to which (roughly speaking) the information stored by a physical system cannot exceed the quantity $A/4$ bit, where A is the area of a surface enclosing the system, expressed in Planck units ($3.3\ 10^{-69} m^2$). See also Problem 2.5.2

network topologies. Historically, a number of topologies have been proposed, typically showing that they support well certain classes of algorithms.

The idea of an efficient general-purpose parallel computer has also been pursued. An interesting approach is based on the notion of *cost-performance universal architecture*. In this approach, one considers a certain class of machines, for example all networks with P processors, with fixed degree (the degree is the maximum number of direct neighbours of any given node; fixed means independent of P). Informally, a universal architecture for this class is one that can simulate efficiently any machine in the class. More quantitatively, one can define a (β, σ)-universal architecture for the given class as one with β times the cost of the machines in the class and a time performance that is never more than σ times the performance of any machine in the class. The quantity β is called the *cost blowup*, while the quantity σ is called the *slowdown*. The closer β and σ are to 1, the better the quality of the universal machine.

One of the successes of the theory of parallel machines has been the development of provably good performance-universal architectures for important classes of machines. For example, as will be shown in later chapters, there are universal architectures for the class of P-processor networks with fixed degree, with cost blowup $\beta = 1$ and slowdown $\sigma = O(\log P)$. A key role in achieving this type of results is played by the methodologies for simulating a guest machine on a host machine: universality of a given architecture is established by showing that it can simulate well of machines of the target class.

The structure of the universal architecture will generally depend upon the class of target machines considered of interest, upon how we measure their cost (e.g., processors or volume), on how we model their performance, and on what tradeoffs we consider reasonalbe between blowup and slowdown. For these reasons, the field of parallel architectures is still open. Nevertheless, the study of universality has provided very valuable insights into this field.

Several other aspects of machine organization require novel thinking when going from sequential to parallel. For example, how will the memory be (logically) organized? Just one space visible to all processing elements or a different private space for each processing element? Especially in the first case, designing a good caching system is quite challenging.

Another question is: how will the machine be controlled? One possibility is to have a central controller producing, at each step, one instruction that all processing elements have to execute. Of course, different processors will apply the instruction to different data. This type of machines are referred to as Single Instruction stream Multiple Data (SIMD). Another possibility if to let each processing element execute the (same) program, but independently of the others; due to different ways of resolving branch conditions, at a given time, different processing elements may be executing different instrutions, on different data. This type of machines are referred to as Multiple Instruction stream Multiple Data (MIMD). Today, most machines are of the MIMD type, but the SIMD paradigm has been implemented on some machines, and it is still often used to control small regions of a machine.

Summary. There are exponentially many different ways of connecting a certain number of processing elements and memory modules to form a parallel computing platform. The "best" way depends on the restrictions posed by technology to the possible interconnections, by how we measure cost, by which computational tasks are chosen as benchmarks, etc... All these factors are still evolving, although for some we can predict the asymptotic behavior, based on fundamental physical limits. Memory organization and control structure are other areas where identifying the best solution is not always immediate. Decades of research have provided us with a wealth of useful and insightful results about parallel machine architecture; the field is a rich one and is still evolving.

3 Problems

Problem 1.1.1. - The energy cost of computing. In a year, here are 8760 hours. Considering an estimated cost of energy of 0.20 dollars per kwh, we obtain a cost of 1,752,000 dollars for 1 MW per year. How much does it cost to run a 10MW supercomputer for three years? (The real cost is even higher, due to the power spent for cooling.)

Problem 2.1.1. - Adding N numbers. Develop an algorithm to add N numbers with $P \leq N/2$ processors (each processor has one adder) in $T(N, P) = O(N/P + \log_2(P))$ steps, in each of which at most P additions can be executed. Which property of addition have you exploited in your algorithm? What happens if more than $N/2$ processors are available?

Problem 2.1.2. - Merging with $P = 2$ comparators. Develop a way to speed up the implementation of the standard merging algorithm if you have $P = 2$ processors (each equipped with one comparator). What if $P = 3$?

Problem 2.1.3. - Multiplying square matrices in parallel. Outline a parallel algorithm to multiply two $n \times n$ matrices with $P \leq n^2$ processors, in parallel time $T(n, P) = n^3/P$. Hint: allocate n^2/P inner products per processor. Assume each processor can perform a multiply&add instruction ($c := c + ab$) per step. How large an n could be handled by a teraFLOPS, a petaFLOPS, and an exaFLOPS machine, respectively, in 10^3 seconds? What would the corresponding energy consumption (in Joule) be, assuming an efficiency of 10^{10} Joule per FLOP?

Problem 2.1.4. - Circuit Value Problem (CVP). Prove that CVP can be solved in linear (hence, polynomial) time in the size of the circuit, given by the number of gates (nodes of a circuit-graph) plus the number of wires (edges of the circuit graph). Hint: (a) topologically sort the gates of the circuit, in linear time, by known algorithms; (b) compute the value output by each gate, by scanning the gates in topological order (which ensures that the inputs to a gate have already been computed by the time that gate output's has to be computed). **Remark 1.** An efficient algorithm to solve the CVP can be very useful, for example to test whether a circuit that we have designed does compute the intended function. **Remark 2.** The approach suggested for the CVP does not really rely on the Boolean nature of the gates, hence it immediately adapts to the case where the gates compute arbitrary operations, for example arithmetic ones or comparisons.

Problem 2.2.1. - Distance and bandwidth in two-dimensional space. Adapt the distance and the bandwidth arguments of Section 2.3 to machines laid out in two dimensions to show that the time needed to execute one parallel step where P operation concurrently access their operands from random locations is $\Omega(P^{1/2})$.

Problem 2.2.2. - Communication requirements of the "Binary String Equality" problem. This problem is meant to illustrate, in a simple case, how data and operation placement can significantly affect the communication requirements of a problem. In "Binary String Equality," we are given two (input) binary strings $x_1 \dots x_N$ and $y_1, \dots, y_N$ and we have to determine whether they are equal (that is, whether $x_j = y_j$ for each $j = 1, \dots, N$). Two processors, P_0 and P_1, connected by a communication channel, will cooperate on the problem to parallelize the task.
Scenario 1. The elements x_j, y_j are initially available to P_0, for $j = 1, \dots, N/2$ and to P_1 for $j = N/2 + 1, \dots, N$. P_1 has to produce the answer (one bit).
Scenario 2. All the x_j's are initially available to P_0 and all the y_j's are initially available to P_1, which has to produce the answer.
Question: What is the minimum amount of information that P_0 and P_1 have to exchange (in the worst case, with respect to all possible values of the intup), in Scenario 1 and Scenario 2, respectively?
Remark 1. Intuitively, the answer is simple to guess in both cases. A rigorous proof of minimality is reasonably simple for Scenario 1. In contrast, fully analyzing Scenario 2 is at the level of a small research problem, as it requires a sophisticated set up, to cover all possible communication strategies that different algorithms may adopt. While it is not expected that a student will solve the problem, she/he can at least formulate the right conjecture, based on the obvious strategy and on failure to conceive anything better.
Remark 2. The requirement that the amount of information be evaluated in the *worst case* is crucial for Scenario 2. If all inputs of size N are equally likely (or, equivalently, if the x_j's and the y_j's are mutually independent, unbiased random bits) then it is not difficult to see that, in the *average case*, only a constant amount of information needs to be exchanged between P_0 and P_1. Intuitively, the worst case occurs when the two sequences are equal, which has probability 2^{-N}.

Problem 2.5.1. - Cost-performance tradeoff. Assume cost is modeled by the number P of processing elements and that performance is modeled by computation time T. Assume that the minimum cost of a machine working in time T satisfies the relationship $P = \pi(T)$, for some function π.

1. Argue, informally, that π is monotonically nonincreasing with T.

2. Let $\tau(P)$ the minimum time achievable by any machine of cost P. What relationship should hold between function τ and function π?

Problem 2.5.2. - Holographic bound on information. According to the holographic principle, the number of bits stored by a physical system enclosed by a surface of area A is at most $A/4$, with A in Planck units (Planck unit of area = $3.3\ 10^{-69} m^2$). Therefore, the information is finite and scales with the area (not with the volume).

- Compute the holographic upper bound to the information of a sphere of radius $r = 0.067m$, the approximate radius of an average human brain. You will find a huge number, compared, e.g., to the 10^{11} neurons or the 10^{14} synapses. Why? Beacause the description of the brain as a network of neurons is a very succint summary of the brain as a physical system. In fact, the holographic bound is huge even for one atom ($r = 10^{-10}m$); check it out.

Problem 2.5.3. - Performance universality of Turing Machines. A Universal Turing Machine (UTM) M_U can simulate any specific TM, M, when a suitable description of M and the input to M are provided as inputs to M_U. (The description of M can be viewed as the (software) program of M_U.) The existence of UTMs has been established by Turing (1936); it represents one of the most important milestones of human thought, in the 20th century. To be specific, consider TMs with just one (infinite) tape and one head. To solve this problem, one needs to know the definitions of TM, UTM, and how a UTM can simulate a TM.

1. What can be said of the slowdown σ of a UTM, with respect to the class of all TMs?
   Suggestions. (a) Reflect on how the simulation of one step of M is carried out by M_U to argue that T steps of M can be simulated in $T_U \leq \sigma_M T$ steps of M_U, where σ_M is constant with respect to the input to M, but will generally depend upon M. (b) Consider how σ_M is affected by the size of M, in terms of the number of states and of the number of symbols in the tape alphabet. (c) Use the fact that there are TMs of arbitrarily large size.

2. What can be said of the slowdown σ of a UTM, with respect to the class of all TMs with a fixed maximum size?

.