Models of Parallel Computation and Parallel Complexity

George Lentaris

$\mu\prod\lambda\forall$

A thesis submitted in partial fulfilment of the
requirements for the degree of Master of Science in
Logic, Algorithms and Computation.

Department of Mathematics,
National & Kapodistrian University of Athens

supervised by:
Asst. Prof. Dionysios Reisis, NKUA
Prof. Stathis Zachos, NTUA

Athens, July 2010

Abstract

This thesis reviews selected topics from the theory of parallel computation. The research begins with a survey of the proposed models of parallel computation. It examines the characteristics of each model and it discusses its use either for theoretical studies, or for practical applications. Subsequently, it employs common simulation techniques to evaluate the computational power of these models. The simulations establish certain model relations before advancing to a detailed study of the parallel complexity theory, which is the subject of the second part of this thesis. The second part examines classes of feasible highly parallel problems and it investigates the limits of parallelization. It is concerned with the benefits of the parallel solutions and the extent to which they can be applied to all problems. It analyzes the parallel complexity of various well-known tractable problems and it discusses the automatic parallelization of the efficient sequential algorithms. Moreover, it compares the models with respect to the cost of realizing parallel solutions. Overall, the thesis presents various class inclusions, problem classifications and open questions of the field.

George Lentaris

Committee: Asst. Prof. Dionysios Reisis, Prof. Stathis Zachos,
Prof. Vassilis Zissimopoulos, Lect. Aris Pagourtzis.

Graduate Program in Logic, Algorithms and Computation

Departments of Mathematics, Informatics, M.I.TH.E.
–National & Kapodistrian University of Athens–
General Sciences, Electrical and Computer Engineering
–National Technical University of Athens–
Computer Engineering and Information
–University of Patras–

$\mu\prod\lambda\forall$

Contents

Chapter 1

Introduction

Digital parallel computers date back to the late 1950s. At that time the industry had already developed a number of sequential machines and the engineers became interested in the use of parallelism in numerical calculations. Consequently, during the 1960s and the 1970s, several shared-memory 'multiprocessor' systems were designed for academic and commercial purposes. Such systems consisted of a small number of processors (1 to 4) connected to a few memory modules (1 to 16) via a crossbar switch. The processors were operating side-by-side on shared data. The advances in the technology of integrated circuits and the ideas of famous architects, such as G. Amdahl and S. Cray, contributed in the evolution of supercomputers. By the early 1980s, the *Cray X-MP* supercomputer could perform more than 200 Million FLoating Operations Per Second (MFLOPS). In the mid-1980s, massively parallel processors (MPPs) were developed by connecting mass market, off-the-self, microprocessors. The *ASCI Red* MPP was the first machine to rate above 1 Tera-FLOPS in 1996 by utilizing more than 4,000 computing nodes arranged in a grid. The 1990s also saw the evolution of computer clusters, which are similar to the MPPs without, however, being as tightly coupled. A cluster consists of mass market computers, which are connected by an off-the-shelf network (e.g., Ethernet). This architecture has gained ground over the years characterizing most of the latest top supercomputers. The *Cray-Jaguar* cluster is the fastest parallel machine today, performing almost 2 Peta-FLOPS and utilizing more than 200,000 cores.

The purely theoretical study of parallel computation begun in the 1970s [1]. The first results concerned circuit simulations relating the model to the Turing machine. In the second half of the decade various models of

parallel computation were formulated, including the Parallel Random Access Machine. Also, Pippenger introduced the class of feasible highly parallel problems (NC), the theory of P-completeness was born, and the researchers started identifying the inherently sequential problems. During the 1980s and the 1990s the area expanded dramatically, flooding the literature with numerous important papers and books. Until today, parallel computation constitutes an active research area, which targets efficient algorithms and architectures, as well as novel models of computation and complexity results.

Basic terms and notions

Through all these years of theory and practice certain terms and notions have been developed. To begin with, most researchers would describe parallel computing as the combined use of two, or more, processing elements to compute a function. That is, the parts of the computation are performed by distinct processing elements, which communicate during the process to exchange information and coordinate their operations. The number and the size of the parts of the computation, as well as the volume of the exchanged information, are qualitatively characterized by terms such as *granularity* and *level* of parallelism. Roughly, *granularity* indicates the ratio of computation to communication steps: a *fine* grained parallel process consists of numerous successive communication events with little local operations in between them, while a *coarse* grained computation is partitioned into big tasks requiring limited coordination (synchronization). Extremely coarse grained computations with almost no task interdependencies can be used for the so called 'embarrassingly parallel problems', which are the easiest to solve in parallel. Evidently, the granularity is related to the chosen *level* of parallelism: *task*, *instruction*, *data*, and *bit* level. *Task* level parallelism focuses on assigning entire subroutines of the algorithm (possibly very different in structure and scope) to distinct processors of the machine. At *instruction* level, we focus on parallelizing consecutive program instructions by preserving their interdependencies (as, e.g., in superscalar processors). *Data* level parallelism distributes the data to be processed across different computing nodes, which work side-by-side possibly executing the same instructions. At *bit* level, the machine processes concurrently several bits of the involved datum (a technique explored mostly in circuit design). Generally, the parallelism level and the granularity of the computation determine the use of either *tightly* or *loosely* coupled systems. A coarse grained, task level, parallel computa-

tion can be performed by a machine with *loosely* coupled components, e.g., off-the-self computers located at distant places (clusters). On the contrary, a fine grained, data level, computation should be implemented on a *tightly* coupled system with, e.g., shared memory multiprocessors.

Clearly, parallelization is used to obtain faster solutions than those offered by the sequential computation. In this direction, we define the *speedup* of a parallel algorithm utilizing p processors as the ratio of the time required by a sequential algorithm over the time required by the parallel algorithm, i.e., $speedup = T(1)/T(p)$. More precisely, when the sequential time corresponds to the fastest known algorithm for that specific problem, we get the *absolute* speedup. Else, when the sequential time is measured with the parallel algorithm itself running on a single processor, we get the *relative* speedup. Note that a parallel solution cannot reduce the sequential time by a factor greater than the number of the utilized processors, i.e., $speedup \leq p$. Consequently, from a complexity theoretic viewpoint, in order to significantly reduce the polynomial time of a tractable problem to, say, sublinear parallel time, we have to invest at least a polynomial number of processors.

The aforementioned *linear* speedup is ideal and in most cases it cannot be achieved by real world machines. Among applications, the speedup usually ranges from p to $\log p$ (with a middle ground at $p/\log p$) [2]. Based on empirical data, the researchers propose various practical upper bounds for the speedup, the most notable being Amdahl's law. According to Amdahl, each computation includes an inherently sequential part, which cannot be parallelized. Assuming that this part is a fraction f of the entire computation, we get $speedup \leq 1/(f + (1-f)/p)$. This 'sequential overhead' f varies with respect to the problems and the algorithms. Fortunately, in some cases f is very small, and in other cases f decreases as the size of the problem increases. For instance, Gustafson's law states that increasing sufficiently the size of a problem and distributing the workload in several processors will probably result in almost linear speedup (experimentally confirmed), i.e., $speedup \leq p - f_n \cdot (p-1)$. Notice the difference between the two laws: Amdahl examines the performance of the algorithm with respect to the number of processors by assuming fixed-size inputs and constant f, while Gustafson observes that f is size-dependent and probably diminishes for large datasets. Overall, in the case of fixed size problems, we have that $speedup \to 1/f$ as $p \to \infty$. In other words, beyond a certain number of processors, adding more hardware to the parallel machine will have no actual effect to the execution time. In fact, the ratio $T(1)/T(\infty)$ is called *parallelism* of the computation

and it can be used either as a theoretical maximum of the speedup or as a practical limit on the number of the useful processors [3].

Besides speedup, we are interested in the effectiveness of the parallel computation. To be more precise, we define the *parallel efficiency* as the ratio *speedup*/p (again, we have *absolute* and *relative* metrics). The efficiency value –typically between 0 and 1– reflects the overhead of the parallelism, such as the time spent for communication and synchronization steps (uniprocessors and optimal speedup algorithms have parallel efficiency equal to 1). Alternatively, we can express the efficiency as the ratio $T(1)/pT(p)$, which compares the sequential cost $T(1)$ to the parallel cost $p \cdot T(p)$. Note here that the product $processors \times time$ is a common cost function, which takes into account both time and hardware requirements of the parallel solution. Additionally to the efficiency, we define ratios such as the $redundancy = W(p)/W(1)$, and the $utilization = W(p)/pT(p)$, where $W(p)$ denotes the total number of operations performed by the p processors and is called *work* (or energy) of the computation [2].

Thesis scope and organization

The current thesis reviews parallel computation from a theoretical viewpoint. First, it is concerned with the formal description of the computation and the understanding of abstract notions such as concurrency and coordination. Hence, *chapter 2* surveys the models of parallel computation that have been proposed by various researchers in the literature. All these modeling approaches show the line of thought developed by experts studying parallelism and, at the same time, summarize the design trends used in real world applications.

The diversity of the models in the literature gives rise to certain questions. Most of them regard the computational power of these models and the ability to migrate solutions between them in a mechanical fashion. That is, given the formal description of a computation for a specific model (a program or a circuit) it is important to study generalized techniques, called simulations, which allow the given computation to be performed by another model. *Chapter 3* uses simulations to investigate the abilities of certain categories of models and to compare model variations with respect to the cost of realizing parallel solutions.

Having been acquainted with the basics of parallel computation, the thesis continues with a second part devoted to the study of problems. Specifically, it

is concerned with fundamental questions such as: are all problems amenable to parallelization? Can we use parallelization to solve intractable problems? How can we classify problems respectfully to the parallelization difficulty? How are these classes related? What is the relation between parallel and sequential computation cost? Is there a way to transform algorithmically an efficient sequential solution to an efficient parallel solution? Such questions are tackled, among others, by the parallel complexity theory, which is explored in *chapter 4*. Finally, *chapter 5* concludes this thesis.

Chapter 2

Models of Parallel Computation

Prior to designing parallel solutions, analyzing algorithms, or studying the parallel complexity of problems, we must define a suitable model to describe the parallel computation. A model of parallel computation is a parameterized description of a class of machines [1]. Each machine in the class is capable of computing a specific function and is obtained from the model by determining values for its parameters (e.g., a Turing machine is obtained by determining the work tapes, the symbol sets, the states and the transition function).

Over the years, several models of parallel computation have been proposed in the literature. Although many of them are widely used until today, none was ever universally accepted as dominant. The reason is that each model serves a slightly different goal than the others, rendering its own use indisputable. To explain the diversity in modeling parallel computation, one should examine the modeling trends summarized in the following four axes: computation, concurrency, theory, and practice. That is, a parallel model should capture the notion of computation and, moreover, the notion of concurrency. Regarding the first, it is clear that we can adopt the outcome of the efforts made in the domain of the sequential computation (which is already vast). Regarding the second, the model must reflect the fact that different parts of the computation are performed in parallel and, more essentially, it must include a means of communication for the coordination of the entire computation. By combining computation and communication ideas, we can derive a plethora of parallel models. In another direction, the model designer

must make choices between theory and practice. That is, the designer must balance factors such as generality and technicality, ease of performance analysis and plausibility of real-world implementation, novelty and historical use, etc. Depending on the intended use and the available technology, the choices add a flexibility, which inevitably leads to a proliferation of different models.

Considering the above, the current chapter describes the majority of the parallel models of the literature. The presentation of the models is organized as follows. We begin with the processor-based machines, i.e., the models capturing computation with the use of powerful components able to compute complicated functions on their own. These models are further divided in two major categories according to the employed communication method: common memory or message exchanges. The third section introduces the circuit models, which capture computation based on very low complexity components and communication lines. Within each one of the three sections, the presentation begins with a high level of abstraction (mostly theoretic models) and continues with the incorporation of real-world details to discuss practical models. Finally, the fourth section describes models beyond these categories, obsolete and novel modeling approaches, as well as architectural taxonomies of the modern parallel computers.

2.1 Shared Memory Models

The shared memory models were among the first to be proposed for the study of parallel computation (1970s). The most characteristic shared memory model is the PRAM. In fact, every other model of the category can be viewed as an extension of the original PRAM of Fortune and Wyllie. In a natural generalization of the single processor, the PRAM is a collection of independent processors with ordinary instruction sets and local memories, plus one central, shared, memory to which every processor is connected in a random access fashion (figure 2.1). Note that the processors cannot communicate directly with each other; their communication is performed through the central memory (via designated variables).

The original PRAM is an ideal model, which abstracts away many of the real world implementation details. As a result, it allows for a straightforward performance analysis of the parallel algorithm. However, in some cases, the disregard of real world parameters might lead to false estimations of practical situations. As a remedy, certain extensions have been added to

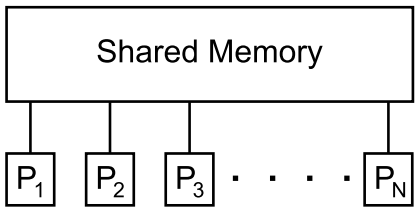


Figure 2.1: The shared memory model: processors and common memory

the PRAM over the years in order to keep track of the current technology aspects. The following subsections describe the PRAM with its variations and its extensions.

2.1.1 PRAM

The *Parallel Random Access Machine (PRAM)* was introduced in 1978 [4]. It bases on a set of *Random Access Machines (RAMs)* sharing a common memory and operating in parallel. Formally,

Definition 2.1.1. *A PRAM consists of an unbounded set of processors* $P = \{P_0, P_1, \ldots\}$*, an unbounded global memory, a set of input registers, and a finite program. Each processor has an unbounded local memory, an accumulator, a program counter, and a flag indicating whether the processor is running or not.*

All memory locations (cells) and accumulators are capable of holding arbitrary long integers. Following the RAM notion in [5], the PRAM program is a finite sequence of instructions $\Pi = \langle \pi_0, \pi_1, \ldots, \pi_m \rangle$, where each labeled instruction π_i is one of the types *LOAD*, *STORE*, *ADD*, *SUB*, *JUMP*, *JZERO*, *READ*, *FORK*, *HALT*. At each time instant, P_i can access either a cell of the global memory or a cell of its local memory (it cannot access the local memory of another processor).

Initially, the input to the PRAM, $x \in \{0,1\}^n$, is placed in the input registers (n is placed in the accumulator of P_0), all memory is cleared and P_0 is started. During the computation, a processor P_i can activate any other processor P_j by using a *FORK* instruction to determine the contents of P_j's program counter. The processors operate in lockstep, i.e. they are synchronized to execute their corresponding instructions simultaneously (in one unit of time). Afterwards, they immediately advance to the execution of their next

instruction indicated by their corresponding program counter. The program execution terminates when P_0 halts and the input is accepted only when the accumulator of P_0 contains "1". The execution time is defined as the total number of instructions (steps) executed by P_0 during the computation.

Definition 2.1.2. *Let M be a PRAM. M computes in parallel time $t(n)$ and processors $p(n)$ if for every input $x \in \{0,1\}^n$, machine M halts within at most $t(n)$ time steps and activates at most $p(n)$ processors.*

M computes in sequential time $t(n)$ if it computes in parallel time $t(n)$ using 1 processor.

Definitions 2.1.1 and 2.1.2 form the basis of the PRAM. By elaborating on the specifics of the model, we derive a number of well-known PRAM variations. Naturally, the first question that follows the above description concerns the memory policy of the model: what happens when more than one processors try to access the same cell of the shared memory? As shown below, simultaneous access to shared memory can be arbitrated in various ways.

Memory Policies of PRAMs

Generally, we consider that the instruction cycle separates shared memory reads from writes [1]. Each PRAM instruction is executed in a cycle with three phases: 1) a read phase –if any– from the shared memory is performed, 2) a computation –if any– associated with the instruction is done, and 3) a write –if any– to shared memory is performed. This convention eliminates read/write conflicts to the shared memory, but it does not eliminate read/read and write/write conflicts. These conflicts are resolved based on concepts such as exclusiveness, concurrency, priority, etc. More specifically, we have the following model variations [2]:

- *EREW-PRAM*: exclusive-read, exclusive-write PRAM. Only one processor can read the contents of a shared cell at any time instant. Similarily, only one processor can write to a shared cell at any time instant (conflicts result in execution halting and rejection).

- *CREW-PRAM*: concurrent-read, exclusive-write PRAM. This model corresponds to the first definition of the PRAM and it is considered as the default variation of the model. Any number of processors can read

from the same memory location, but only one can write to a shared cell at any time instant.

- *ERCW-PRAM*: exclusive-read, concurrent-write PRAM. Only one processor can read from a cell, but many processors can write to a cell at any time instant (not a very rational convention, considering that CW technology should render CR even simpler).
- *CRCW-PRAM*: concurrent-read, concurrent-write PRAM. Any number of porcessors can read/write to any cell at any time instant. It is the most powerfull of the model variations.

Besides EW, more restricted models exist, which are based on the concept of ownership [6]. These are called the *owner write (EROW/CROW)* models and they avoid conflicts by having each processor own one cell to which all his write operations are performed.

In the case of concurrent writes, we must further define a policy determining the exact data to be written in the requested cell [2]:

undefined CRCW-PRAM:	an undefined value is written in the cell.
detecting CRCW-PRAM:	a special code "detected collision" is written.
random CRCW-PRAM:	an offered datum in random is chosen.
common CRCW-PRAM:	write when all the offered data are equal.
max/min CRCW-PRAM:	the largest/smallest datum is written.
reduction CRCW-PRAM:	a logical/arithmetic operation (and/or/sum) is performed to the multiple data and the resulting value is written in the cell.
priority CRCW-PRAM:	based on a predetermined ordering (e.g. that of the PIDs), the processor with the highest priority writes its datum in the cell.

Computing Functions with PRAM

Besides accepting languages, PRAMs can be used to compute functions. For such computations we must slightly modify the output of the model: we equip the PRAM with a set of output registers (similar to the input registers). We say that a machine M computes a function $f(x) = y$ with $x, y \in \{0,1\}^*$ if whenever it is started with x in its input registers, it will eventualy halt holding y in its output registers. Note that the use of I/O registers allows the study of sublinear time [4].

Another input/output convention [1], is to present an input $x \in \{0,1\}^n$ to the machine M by placing the integer n in the first shared memory cell C_0 and the bits $x_1, x_2, \ldots, x_n$ in cells $C_1, C_2, \ldots, C_n$. Similarly, M will present its output $y \in \{0,1\}^m$ by placing m in C_0 and the bits $y_1, y_2, \ldots, y_n$ in cells $C_1, C_2, \ldots, C_n$.

Definition 2.1.3. *Let f be a function from $\{0,1\}^*$ to $\{0,1\}^*$. The function f is computable in parallel time $t(n)$ and processors $p(n)$ if there is a PRAM M that on input x outputs $f(x)$ in time $t(n)$ and processors $p(n)$.*

Nondeterminism in PRAM

Another important aspect of PRAM is nondeterminism. A nondeterministic PRAM can be defined as a collection of nondeterministic RAMs operating in parallel. To be more precise, consider that each instruction π_i of the PRAM program Π is labeled. If some label appears more than once in Π, then we have a nondeterministic PRAM M [4]. Alternatively, we can envisage the nondeterministic PRAM as a program Π with uniquely labeled instructions π_i, where each *JUMP* instruction can branch to more than one labels (e.g. *JUMP* {*L1,L5,L7*}). This definition is analogous to the definition of the nondeterministic Turing Machine (TM), where the transition function of the DTM becomes a relation allowing the NTM to branch from a single configuration to many distinct configurations at any step of the computation. Similarly to the NTM, the input to the NPRAM is accepted only if there exists a computation path in which processor P_0 halts with a "1" in its accumulator. Time and processor complexity for the NPRAM is defined as in definition 2.1.2.

SIMDAG

The *Single Instruction stream Multiple DAta stream Global memory machine (SIMDAG)* was introduced independently in 1978 [7]. However, it can be considered as the special case of the PRAM where the program counter is common for all processors, i.e., all active processors execute the same instruction π_i at each step of the computation. Note that each processor may operate on different data values than the other processors (multiple data stream). The original SIMDAG corresponds to a CREW machine with a control unit common for all processors.

2.1.2 PRAM Extensions

The ordinary PRAM is a good model for expressing the logical structure of parallel algorithms and is very useful for their gross classification. However, the abstraction level of this model hides important performance bottlenecks that appear in real world applications. The PRAM assumes unit cost access to the shared memory (independent of the number of processors), infinite bandwidth for the memory-to-processor communication (and thus for the processor-to-processor communication), and makes many more unrealistic implications. Such conventions might lead to the development of algorithms with a very degraded performance when tested in practice. To reduce the theory-to-practice disparity, numerous modifications and extensions have been added to the PRAM model during the past three decades. The following paragraphs present the most prominent of these models.

PRAM with Memory Latency

The most common criticism of the simple PRAM model is the unit cost access to the shared memory. Under this assumption, the PRAM does not discourage the design of algorithms which access the shared memory continuously (nor the superfluous interprocessor communication via the shared memory). It charges the same cost to any instruction independently of the number of the active processors. However, the implementation of algorithms that read/write excessively to the shared cells would lead any real world system to memory contention. Therefore, the cost of a memory access would increase and the actual performance of the algorithm would diverge from its theoretical estimation[1].

A shared memory access in real multiprocessor systems consumes tens to thousands of instruction cycles. Whichever the underlying interconnection network is, the delay increases with the number of the active processors (either because of the message contention, or because of the expansion of the hardware to support larger communication paths). Such a phenomenon should be taken into account by any theoretical model trying to capture the behavior of a parallel algorithm and lead to efficient solutions.

A first approach is to equip the PRAM with a parameter δ defining the delay of the processor-to-memory communication, measured in elementary

[1] notice that the memory policies discussed in the previous subsection deal with contention for a single shared cell, not with contention for the entire shared memory.

steps of the processors (instruction cycles) [2]. Since δ is a function of the number of processors $|P|$ and the underlying interconnection, it should be determined prior to the design or the analysis of a parallel algorithm for the PRAM. In the resulting model the cost of a read/write operation to the shared memory is δ and not unit (unless $\delta = 1$), i.e., we have a "non-uniform memory access" model. Consequently, the algorithm designer should attemp to minimize the global communication by judicious use of the corresponding read/write instructions in the porgram Π.

A second approach –a generalization of the first– is given in [8]. The key idea bases on an observation regarding the memory service in real world computers. When a processor accesses a block of shared cells, typically, it takes a substantial period of time to access the first cell, but after that, subsequent cells can be accessed quite rapidly. This might occur because of the data caching techniques or because of the policies employed for congestion control (in message passing systems).

The *Block Parallel Random Access Machine (BPRAM)* is in many ways similar to the pure PRAM. It includes a shared memory of unbounded size and a number of processors $|P|$ with their own private memories (also of unbounded size). The instructions and the I/O conventions are the same for both models. The basic differences lie in the number of processors, which is fixed for the BPRAM, and in the accessing of the shared memory: the *LOAD-STORE* instructions of the BPRAM refer to a whole block of the shared memory (a number of contiguous cells) of length b ($b = 1$ for a single cell). The cost of accessing such a block is $l + b$, where l is the latency of the memory service discussed above. The latency l and the number of processors $|P|$ are the parameters of the BPRAM. To conclude with the definition, any number of processors may access global memory simultaneously, but in an exclusive-read-exclusive-write (EREW) fashion. In other words, blocks that are being accessed simultaneously cannot overlap.

Asynchronous PRAM

Synchronization is an important consideration in massively parallel machines, of great concern to the hardware and software communities, but largely ignored by the theory community. The simple PRAM assumes that the processors execute in lockstep. However, the construction of a real machine with this specification becomes impractical for large number of processors. This is because of the various problems that arise with the synchronization

of the processors, which depends on the execution time of their instructions. To determine the length of the processors' lockstep, one must study worst case scenarios taking into account phenomena such as the clock skew, the interconnection network congestion, the delay of a costly instruction (e.g. a floating point multiplication), etc. All these delay parameters add up to a large overhead increasing the execution time of a single computation step and decreasing the utilization of the real machine. The synchronous approach seems even more inefficient considering the potential functionality gained by a generalization of the PRAM: the *Asynchronous PRAM (APRAM)* model [9].

The APRAM is defined as a PRAM without a global clock, i.e., each processor can operate in its own speed. The memory-processors convention and the set of instructions remain the same for the APRAM, with the exception of an new type of instructions called the "synchronization steps". A synchronization step among a set P of active processors is a logical point during the computation where each processor P_i waits for all the other processors in P to arrive before continuing with its own local program. The use of these instructions divides the execution of a program Π in phases, within which each processor runs independently from the others (and thus it cannot know the progress of the others).

The execution time of an APRM program is determined by the execution time of its phases. Each phase is completed after all processors arrive at the synchronization step. Consequently, each phase consumes time equal to that of the last processor to arrive at the step plus the execution time of the step itself, which depends to the total number of the active processors.[2]

Similar to the PRAM, the APRAM is a family of models that differ in the types of the synchronization steps, in the memory policy, and in the cost of accessing the memory. Regarding the memory policies, the APRAM supports both exclusive and concurrent read/writes (EREW, CREW, CECW). Its major difference from the PRAM is the inability of a processor to read a shared cell while another processor writes to it. Since APRAM misses the lockstep, it is impossible to use the three cycles (read-compute-write) described in the previous section to resolve the read/write conflict. Instead, we can only use a synchronization step involving both processors between the two accesses. With this convention, APRAM eliminates the possibility of

[2]the cost of a synchronization instruction is not unit; it is $B(|P|)$ where B is a non-decreasing function. This function is a parameter of the model (not fixed) allowing the analysis to adapt to the characteristics of the underlying real world machine.

race conditions among processors. Regarding the access cost, APRAM can account for a communication delay to the shared memory or not (unit cost access). A commonly used assumption is that a read operation consumes $2d$ time while a write operation consumes d time, where d is a paramemter of the model. Regarding the types of synchronization, APRAM can be modified to support *subset synchronization* where multiple disjoint sets of processors can synchronize independently and in parallel. The cost for a synchronization step among the processors in a set P' is charged only to those processors in P'. The case $P' = P$ is called *Phase PRAM* denoting the *all-processor synchronization*.

Shared Memory Divided in Modules

There are even more specialized criticisms about the shared memory assumptions of the PRAM. Consider for example the ability to access an unlimited number of memory cells simultaneously (potentially all of them). Even though it is not technologically impossible, it is quite costly to support such functionality and it is not the trend of today's industry. The most common memory designs lead to a memory module consisting of a set of individual cells, which however are grouped under a common interface. As a result, only one cell of the module can be accessed at a time. A more practical model should take into account this restriction and assume a shared memory organized in modules.

In the *Module Parallel Computer (MPC)* [10] the number of memory modules m is bounded[3], constituting a parameter of the model along with the number of RAM processors $|P|$. Each module m_i can be accessed by the processors in a EREW fashion. Therefore, the challenge is to distribute the logical addresses of the entire shared memory so as to minimize the potential memory conflicts between processors without stalling the execution of the algorithm. Of course, it is expected that less modules lead to more access delays (because of the exclusiveness restriction) even though the interconnection network is idealized in MPC.

The mapping of a parallel algorithm to the MPC is called the *granularity of parallel memories* problem. The authors in [10] propose two generic solutions: (i) randomization, where the logical addresses are randomly distributed among the memory modules by selecting a hash function which can

[3]assuming an unbounded set of memory modules brings us back to the original definition of the PRAM

be efficiently computed by any processor, (ii) copies, where several copies of each logical address is stored in distinct memory modules (storage redundancy). The randomization solution is shown to keep memory contention low in the average, while the copies solution decreases memory contention in the worst case.

In a more radical approach, [11] describes a parallel machine with a shared memory divided in modules and arranged in a tree. The leaves of the tree correspond to the processors, while the levels of the tree capture the caching techniques applied to most real world machines. The processors communicate directly with their parent memory modules and similarly, memory modules communicate with their children and parent modules by exchanging blocks of values. The length of a block depends on the level of the tree that the exchange takes place (typically, the length doubles at each level towards the root). The model can support several memory policies, namely EREW, CREW and CRCW. The parameters of the so called *Uniform Parallel Memory Hierarchy (UPMH)* define the number of the modules, the depth of the memory tree, the communicated blocksize, and even the transfer costs. In the same approach, non-uniform communication costs of various interconnection topologies can be modelled by combining several levels of a PMH. The idea to use such a model for parallel computation was inspired by the observation that the techniques employed to optimize sequential algorithms destined for memory hierarchies are similar to those for developing parallel algorithms.

2.2 Distributed Memory Models

The distributed memory model of computation assumes a set of autonomous processors with ordinary instruction sets and local memories. Additionally, each processor can send and receive finite length messages with the use of special purpose instructions. As opposed to the shared memory models, the processors here are not connected to a common memory. Rather, they are connected to a –common– communication medium (or, more precisely, interconnected via a specific network). Hence, the processors can communicate direclty with each other by exchanging messages. The computation advances in a sequence of steps involving local operations, communication, and possible synchronization. Notice that the memory of the parallel machine is distributed over the local memories of its processors. Figure 2.2 illustrates

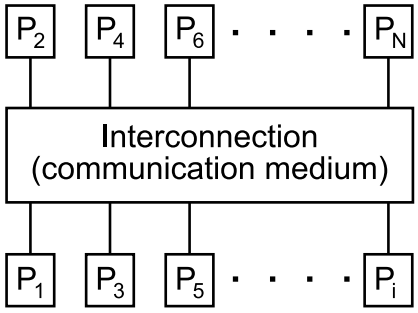


Figure 2.2: Distributed memory model: a network of processors

such a setting.

The distributed memory model is widely used today for the development of supercomputers. The design and study of algorithms for this model strongly depend on the characteristics of the underlying interconnection network of the machine, i.e., to various machine-specific details. On one hand, these details define a large number of architectures allowing the engineer to develop efficient solutions tailored to the requirements of the application. On the other hand, the existence of a large number of network architectures renders the unified theoretical study of algorithms in this model quite troublesome. Naturally, there have been proposed "bridging" models which abstract away the details of the network and allow the analysis of the algorithm to base only on a limited number of parameters. In this direction, the following subsection presents the well-known BSP model.

2.2.1 BSP

The apparent need for a unifying model for parallel computation led Valiant in defining the "Bulk Synchronous Parallel" model (BSP) [12]. The major purpose of such a model is to act as a standard on which people can agree. It is intended neither as a hardware nor as a programming model, but something in between, analogously to the von Neumann model in sequential computation. The von Neumann model abstracts the underlying technology of a computer. Even with rapidly changing technology and architectural ideas, hardware designers can still share the common goal of realizing efficient von Neumann machines. They are not too concerned about the software that is going to be executed. Similarly, the software industry in all its diversity can

aim to write programs that can be executed efficiently on this model, without explicit consideration of the hardware. Thus, the von Neumann model is the connecting bridge that enables programs from the diverse and chaotic world of software to run efficientby on machines from the diverse and chaotic world of hardware.

The BSP is a bridging model in the realm of parallel computation. It consists of a number of *components* performing processing and/or memory functions and a *router* delivering messages point-to-point between pairs of components. The router abstracts the underlying interconnection network of the components by introducing certain parameters in the model, described below. Moreover, the BSP incorporates a synchronization mechanism similar to the one used in the shared memory APRAM (see section 2.1.2): the processing components are synchronized at regular time intervals of L time units, where L is the *periodicity* of the computation. Synchronization is performed by the *Barrier Synchronizer* of the model (a separate entity).

The structure of the BSP computation reflects the way that a parallel programmer thinks: perform some local computation, exchange information required for the next local computation, and synchronize the processes to assure the correctness of the algorithm. To be more precise, a BSP computation consists of a sequence of *supersteps*. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and message arrivals from other components. Conceptually [13], the superstep is divided in three phases: the local computation phase, the communication phase, and the barrier synchronization. Each processor can be thought of as being equipped with an output pool, into which outgoing messages are inserted, and an input pool, from which incoming messages are extracted. During the local computation phase, a processor may insert messages into its output pool, extract messages from its input pool, and perform operations involving data held locally. During the communication phase, every message held in the output pool of a processor is transferred to the input pool of its destination processor. The previous contents of the input pools, if any, are erased. The superstep is concluded by barrier synchronization. Every L time units, a global check is made to determine whether the superstep has been completed by all the components (i.e., all local computations are completed and every message has reached its destination). If it has, the machine proceeds to the next superstep. Otherwise, one more time period L is allocated to the unfinished superstep. Note that [14], although the model emphasizes global barrier style synchronization,

pairs of processors are always free to synchronize pairwise by, for example, sending messages to and from an agreed memory location. These message transmissions, however, would have to respect the superstep rules.

BSP parameters and characteristics

In the BSP model we assume that each local operation costs one unit of time. The task of the router is to realize an arbitrary *h-relation*, i.e. a superstep in which each component sends and receives at most h messages. We properly define a BSP computer by first determining three basic parameters[4] [12] [15]:

p: the number of components

s: the startup cost of the h-relation realization (in time units)

g: the multiplicative inverse of the router's throughput[5]

The parameters s and g describe the performance of the router. Their values depend on the underlying interconnection network and are nondecreasing functions of p. For example, the g parameter depends on the bisection bandwidth of the network, the communication protocols, the routing, the buffer management, etc. Similarly, the s parameter depends on software issues of each processor, the wait cycles for each synchronization step, and other characteristics of the network. Both s and g can be bounded in theory, but in practise they are empirically determined by running suitable benchmarks. Most of the factors affecting s and g become even more apparent as the number of processors increases. As a result, the time costs incurred by s and g increase with p. However, there is a way around this problem: the g parameter can be controled, within limits, by adjusting the router design. It can be kept low by using more pipelining or by having wider communication channels. Keeping g low or fixed as the machine size p increases incurs, of course, other type of costs. In particular, as the machine scales up, the hardware investment for communication needs to grow faster than that for computation [12].

[4] Alternatively, [14] uses the parameters p-proseccors, g-bandwidth and L-periodicity

[5] More precisely, g must be regarded as the ratio of the number of local computational operations performed per second by all the processors, to the total number of data words delivered per second by the router (alternatively, $1/g$ is the available bandwidth per processor). It is used to measure communication costs, i.e. we consider that an h-relation is realized at the cost of $g \cdot h$ time units for h larger than some h_0

The time complexity of a BSP algorithm is measured by summing the costs of its supersteps. The cost of a superstep can be defined in various ways, depending on specific assumptions made by the model. In the most common approach, we measure g when the router is in continuous use (i.e. the throughput of a pipelined architecture) and thus, the cost of realizing an h-relation becomes[6] $g \cdot h + s$. Consequently, when the communication operations do not overlap with the local computation steps at each processor, the cost of a superstep i becomes

$$Cost_i = g \cdot h_i + s + m_i$$

where g, s are the model parameters, h_i is the number of the exchanged messages per processor (the h-relation), and m_i is the number of the computation steps per processor during the superstep i. Note that h_i and m_i correspond to the maximum values over all processors.

Before using the aforementioned cost function[7], one should specify the values of its parameters according to the characteristics of the underlying –physical– machine. Intuitively [13], for sufficiently large sets of messages ($h >> s/g$), the communication medium must deliver p messages every g units of time. Parameter s must be an upper bound for the time required for global barrier synchronization ($m_i = 0$, $h_i = 0$). Moreover, $g + s$ must be an upper bound to the time needed to route any partial permutation and therefore to the latency of a message in the absence of other messages ($m_i = 0$, $h_i = 1$). Practically, g and s have been measured for various real world machines and their values are given in [16].

BSP example

A complete set of programming tools and specific C libraries has been developed for compiling and running BSP programs [17]. Bellow we give a simple BSP example [16]; the program *bsp_sum()* computes the sum of p integers on p processors (initially held in distinct processors). The computation is performed using the following logarithmic technique. At each algorithmic step, processor p_i adds the value sent by $p_{i/2}$ to its local partial sum, and

[6] the time to sent h messages through the router is $g \cdot h$ and the time to initiate such a process is s

[7] another worth mentioning approach is to charge each superstep as $Cost_i = \max(h_i + s,\ m_i)$. This function is suitable for models where the communication operations overlap with the local computations

forwards the result to p_{2i}. At the end of the computation, the rightmost processor, $p-1$, holds the result. Each algorithmic step is implemented as a BSP superstep. A total of $\lceil \log p \rceil$ supersteps is required.

```
int bsp_sum(int x) {
  int i, left, right;
  bsp_pushregister(&left,sizeof(int));
  bsp_sync();

  right=x;
  for (i=1; i<bsp_numofprocs(); i=i*2) {
    if ( bsp_id()+i < bsp_numofprocs() )
      bsp_put(bsp_pid()+i, &right, &left, 0, sizeof(int));
    bsp_sync();
    if ( bsp_pid() >= i ) right=right+left;
  }
  bsp_popregister(&left);
  return right;
}
```

The above code-sample is copied and executed at each BSP processor concurrently. The function *bsp_pid()* gives a unique ID to the processor which called it. The *bsp_sync()* implements the barrier synchronization of the BSP. The *bsp_pushregister()* declares the variable *left* as a storage space which will be used by other processors for writing (for sending data to the current processor). The variable *right* of the processor p_i is initially given the input value of p_i and is used during the execution to accumulate the partial sums. The *bsp_put()* is used for communication: it will copy the *right* value of processor *bsp_pid()* to the value *left* of processor *bsp_pid()*+i. Finally, *bsp_popregister()* removes the global visibility of the variable *left*.

The cost of this algorithm is $\lceil \log p \rceil (1 + g + s) + s$, because there are $\lceil \log p \rceil$ supersteps for computations and one for registration. Notice that at each superstep we have one local addition ($m = 1$) and only one word is communicated between any pair of processors ($h = 1$).

2.2.2 LogP

The BSP was an inspiration for the authors in [18] to introduce a similar model named LogP (the name is derived from the four parameters of the model, which will be described next). Targeting similar goals, the LogP was designed taking into account the technology trends underlying parallel computers [8]. It is intended to serve as a basis for developing fast portable

[8] in fact, the LogP authors in [18] discus technological aspects and make accurate predictions of how Massively Parallel Processors will be designed throughout the 90's

parallel algorithms, as well as to offer guidelines to hardware designers. LogP, like BSP, attempts to strike a balance between detail and simplicity by using certain parameters to abstract the communication between the processors.

LogP is a distributed memory multiprocessor in which processors communicate by point-to-point messages. Contrary to the BSP, LogP is asynchronous, i.e. it features no barrier synchronization. Two processors can synchronize only by exchanging messages between them. LogP uses parameters for modeling the performance of the interconnection network, but it does not describe the structure of the network. It extends BSP by using one more parameter and by imposing a network capacity constraint, i.e. up to a maximum number of messages can be in transit concurrently.

Conceptually [13], for each processor there is an output register where the processor puts any message to be submitted to the communication medium. The "preparation" of a message requires certain time and, once submitted, the message is "accepted" by the communication medium. It will be delivered to its destination with a certain delay. When a submitted message is accepted, the submitting processor reverts to the operational state, where it continues with its local operations. Upon arrival, a message is promptly removed from the communication medium. This message can be immediately processed by the receiving processor or it can be buffered. As with the preparation of an outgoing message, the acquisition of an incoming message requires certain time, which is modeled by the new parameter of LogP, named "overhead".

LogP parameters and characteristics

Once again, in the LogP model we assume that each local operation costs one unit of time (one "cycle"). We formally define the model by defining its four parameters:

L: the latency, the delay of a message to reach its destination

o: the overhead, the time a processor engages in a transmission or reception

g: the gap, minimum time between message transmissions (per processor)

P: the number of components (i.e., processor/memory modules)

The parameters L, o and g are measured as multiples of the processor cycle. More precisely, the "latency" is an upper bound of the time required for a

single word message to travel from its source to its destination. It depends mostly on the underlying interconnection network and it can be calculated as $L = Hr + \lceil M/w \rceil$, for an M-bit message traveling through w-wide channels, across H hops with r delay each. The "overhead" is a period during which the processor is engaged in sending/receiving messages and cannot perform any other operations. Notice that o does not include L and that it is mostly related to the underlying technology of the processor. It is regarded as the mean overhead $(T_{snd} + T_{rcv})/2$. The "gap" is a parameter used to model the available per processor bandwidth. Since the maximum speed at which a processor can send messages is one every g time units, then the reciprocal of g corresponds to the bandwidth. It can be calculated [15] as $g = {}^{PM}/_{wW}$, where W is the bisection width of the network. Overall, the total time to communicate a message between two processors can be calculated as $T = 2o+L = T_{snd}+Hr+\lceil M/w \rceil+T_{rcv}$. In practice, the values of these parameters have been measured for several real world parallel machines, e.g. in [18].

From the above definition, it is clear that any processor can have no more than $\lceil L/g \rceil$ of its messages traveling in the communication medium concurrently. In fact, the model assumes finite network capacity, such that no more than $\lceil L/g \rceil$ messages can be in transit from any processor, or to any processor, at any time. If a processor attempts to transmit a message that would exceed this limit it stalls until the message can be sent without exceeding the capacity limit. Notice here that the model does not ensure that the messages will arrive at the same order that they were sent.

The above parameters are not considered equally important in all situations [18]. For an easier algorithm analysis, it is possible to ignore one or more parameters and work with a simpler model. For example, in algorithms that communicate data infrequently, it is reasonable to ignore the bandwidth and capacity limits. In some algorithms messages are sent in long streams which are pipelined through the network, so that message transmission time is dominated by the inter-message gaps, and the latency may be disregarded. Also, notice that g and o can be merged in one parameter without altering much the results of the analysis [18] [9].

[9] approximation by a factor of at most 2 (if we consider $g = o$). Arguably, o is unnecessarily inserted in the model [15] [16]. Actually, the LogP authors hope that technology (off-the-shelf processors) will eventually eliminate o [18].

LogP example

Arguably [13] [16], LogP provides a less convenient programming abstraction compared to BSP. The analysis of a LogP algorithm is somewhat less straightforward, primarily due to the lack of the synchronization barriers. As an example, we discuss here the simple problem presented in the previous section (for the BSP case): the optimal summation of P integers.

If we were to write a code sample for *logp_sum()*, we would certainly omit the *bsp_sync()* function used in the BSP case. Instead, we would take for granted that any message arrives at most after L time units.

As with BSP, the LogP processors will gradually construct a tree for communicating their values. The idea [18] is that each processor will sum a set of the input elements and then (except for the root processor) it will transmit the result to its parent as quickly as possible (we must ensure that no processor receives more than one message). The elements to be summed by a processor consist of original inputs stored in its memory, together with partial results received from its children in the communication tree. The main difference from BSP is that the LogP tree will not be binary [10]. It will be an unbalanced tree, with the fan-out of each node determined by the values L, o, g and the following analysis.

To specify an optimal algorithm, we must determine (off-line) the optimal schedule of communication events and then determine the distribution of the initial inputs over the P processors. We start by considering how to sum as many values as possible within a fixed amount of time T. If $T < L + 2o$, the optimal solution is to sum $T + 1$ values on a single processor, since there is not sufficient time to receive data from another processor. Otherwise, the last step performed by the root processor (at time $T - 1$) is to add a value it has computed locally to a value it just received from another processor. The remote processor must have sent the value at time $T - 1 - L - 2o$ and we assume recursively that it forms the root of an optimal summation tree with this time bound. The local value must have been produced at time $T - 1 - o$. Since the root can receive a message every g cycles, its children in the communication tree should complete their summations at times $T-(2o+L+l), T-(2o+L+l+g), T-(2o+L+l+2g), \ldots$. The root performs $g-o-1$ additions of local input values between messages, as well as

[10] the root of the tree will have more children than other nodes nested deeper in the tree. The reason is that as a message travels, the source processor has time to send new messages.

the local additions before it receives its first message. This communication schedule must be modified by the following consideration: since a processor invests o cycles in receiving a partial sum from a child, all transmitted partial sums must represent at least o additions.

A good LogP algorithm should coordinate work assignment with data placement, provide a balanced communication schedule, and overlap communication with processing.

2.2.3 Fixed-Connection Networks

The BSP and LogP models presented above abstract the structure of the interconnection network of the processors. They introduce certain parameters to measure the communication cost during the computation, without taking into account the relative location of the processors that exchange messages. Such a uniform approach simplifies the performance analysis and, moreover, is accurate in various situations (e.g., ethernet, or fully connected networks). However, in many applications, a parallel machine includes only a limited number of internal connections between predetermined pairs of processors. Communication is allowed only for the directly connected processors, while the remote destination messages are explicitly forwarded through intermediate nodes. Here, studying the exact structure of the interconnection network becomes very important for the design of an efficient parallel machine.

In the fixed-connection network model the parallel machine is represented as a graph G, where the vertices correspond to processors and the edges correspond to direct communication links [2]. The computational power of each processor may vary, although it is common to assume that they involve low complexity control and limited local storage [19]. Depending on the architecture, the computation can be either synchronous or asynchronous. In the first case, we assume that a global clock signal traverses the entire graph determining the steps of the computation. At each step, each processors can receive data from its neighbors, read its local storage, perform local computations, update its local memories, or generate output data/messages. In the case of an asynchronous computation, there is no global clock. Instead, a processor can send or receive messages from its neighbors at any time instant. The communication might be *blocking* or *non-blocking* (i.e., the sender suspends its operations until the receiver sends back a response, or the sender continues independently of the receiver) and the activity of the machine is coordinated via designated messages (coarse-grained synchronization).

Apparently, an important aspect in the development of an algorithm for the fixed-connection network model is the design of a scheme for the interprocessor communication within the parallel machine. In the general case, the messages have to be transferred over a number of intermediate edges before reaching their destination. The message transfer is handled by the processors in between the source and the destination of the message, based on a predetermined procedure. Such a procedure is called *routing* and is of great importance for the performance of the machine. Usually, several routing problems have to be solved just to implement a single parallel algorithm. The routing problem is defined as a set of messages with specific destinations, which are either stored initially in distinct nodes of the network, or they are generated dynamically during the computation. Targeting solutions for distinct problems and network topologies, several routing algorithms have been proposed in the literature: online or offline, deterministic or randomized, with and without queues, greedy, flooding, etc [19].

In the network model, the parallel machines differentiate primarily with respect to their interconnection topology. Among others, the topology of the network determines the implementation cost, the communication delays, and hence, the efficiency of the parallel algorithm. The design of the network depends on the application and, naturally, a plethora of these has been presented in the literature over the years. In fact, it can be shown that the machines benefit greatly from carefully tailored networks with judicious processor-to-node mappings. Overall, considering the performance analysis of each network, the available routing algorithms, and the various techniques for cross-simulating networks, one can be led to an immense area of study (beyond the scope of this thesis) [19]. In the context of this review chapter, the following subsection presents a few of the most commonly used fixed-connection networks.

Common network topologies

We begin with the definition of three parameters characterizing the topology of the network, i.e., of the graph G: the *diameter*, the *maximum degree* and the *bisection width* (or *edge connectivity*) [20] [2]. The *diameter* of G is the longest of the shortest paths between any pair of nodes of G (maxmin path). The *maximum degree* corresponds to the maximum number of edges incident to any node of G. The *bisection width* measures the minimum number of edges that need to be removed for the partition of G in two disjoint subgraphs

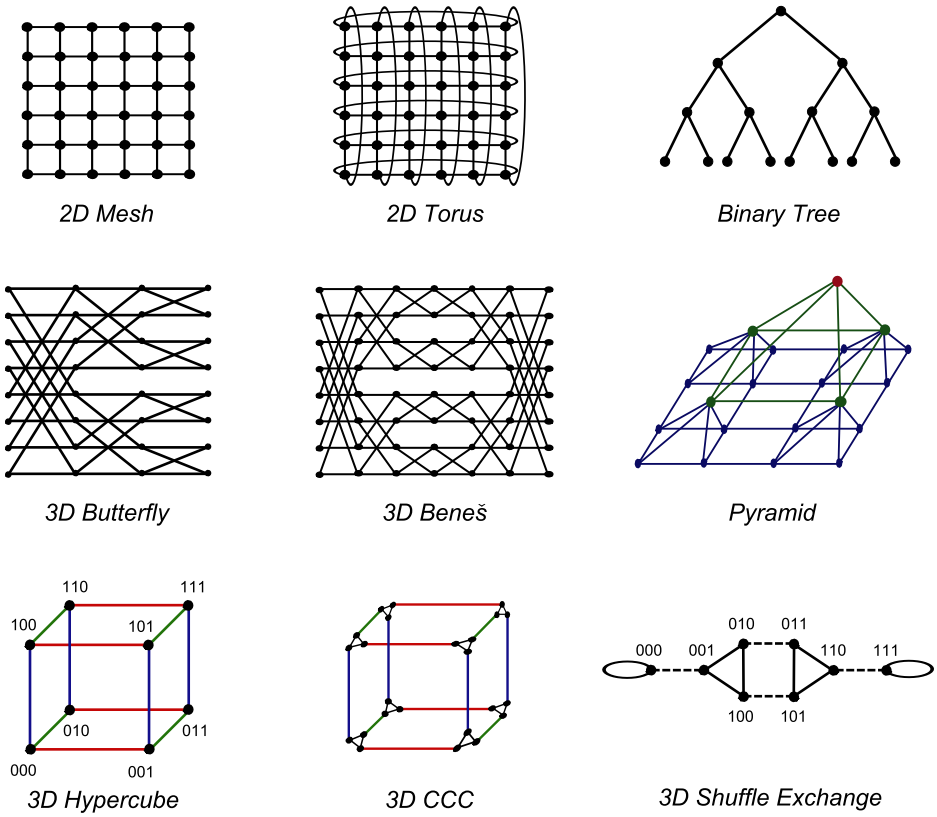
of equal size (i.e., $||G_1| - |G_2|| \leq 1$)[11]. Regarding the practical effect of these parameters, the diameter of the network affects the communication delay, especially when the message routing is performed in a store-and-forward fashion (each node buffers the entire message before retransmitting it, instead of continuously forwarding its parts as smaller packets). The bisection width is related to the bandwidth of the network (depends on the bandwidth of each link separately) and is important for communication intensive algorithms where the message destination follows a uniform distribution over the nodes. The degree of G has an impact on the implementation cost of the nodes. Examples for these three parameters are given bellow [19] [2].

Undoubtedly, the most simple fixed connection network is the *1D Mesh*, or *linear processor array* [19]. It consists of p processors $P_1, P_2, \ldots, P_p$ connected in a linear array, i.e., P_i is connected to P_{i-1} and P_{i+1}. In the case of processors P_1 and P_p being connected directly, with wraparound wires, we get the definition of a *Ring*, or *1D Torus*. Note that the diameter of the 1D array is $p-1$, while that of the ring is $p/2$. The bisection width is equal to 1 for the array and equal to 2 for the ring. Both networks have maximum degree equal to 2. The construction of these networks can be easily generalized to two, three, or more dimensions (see figure 2.3).

The r-dimensional *Butterfly* has $N = 2^r(r+1)$ nodes and $r2^{r+1}$ edges (undirected graph) [19]. Each node is labeled, virtually, with a unique $\langle w, i\rangle$ bit-string, where w is an r-bit number denoting the row of the node and i, $0 \leq i \leq r$, denotes the *stage* (the column) of the node. Two nodes $\langle w, i\rangle$ and $\langle w', i'\rangle$ are connected if and only if $i' = i+1$ and, (i) $w = w'$ or, (ii) w differs from w' in exactly the i'th bit. Notice the recursive structure of the butterfly network (its butterfly subgraphs) and the uniqueness of the paths from any "input" to any "output" node of the graph (see figure 2.3).

The butterfly is a widely-used network and it features certain variations. The *Wrapped Butterfly* is constructed, essentially, by merging the first and the last stages of an ordinary butterfly network. That is, we merge two nodes (one input and one output node) when they belong to the same row of the initial butterfly. Hence, every node of the resulting network has degree equal to 4. Note that the two networks have similar computational power, i.e., they can simulate each other with a slowdown factor ≤ 2 (the same holds for many networks and their wraparounds: the linear array and the ring,

[11] the 'bisection problem' is NP-hard, as opposed to the 'mincut', which can be solved efficiently by using flow techniques (the mincut places not constraint on the partitions).



Network	*Nodes*	*Degree*	*Diameter*	*Bisection*
1D Mesh	k	2	k-1	1
1D Torus (ring)	k	2	k/2	2
2D Mesh	k^2	4	2k-2	k
2D Torus	k^2	4	k	2k
3D Mesh	k^3	6	3k-3	k^2
Binary Tree	2^r-1	3	2r-2	1
Pyramid	$(4^{r+1}-1)/3$	9	2r	2^{r+1}
Butterfly	$2^r(r+1)$	4	2r	2^r
Beneš	$2^r(2r+1)$	4	2r	2^{r+1}
Hypercube	2^r	r	r	2^{r-1}
CCC	$r2^r$	3	2r	2^{r-1}
Shufle Exch.	2^r	4	2r-1	$2^{r-1}/r$

Figure 2.3: Outline and parameters of common fixed-connection networks

the mesh and the torus in 2D, etc). In another variation, the *Beneš* network is, essentially, two butterfly networks connected back-to-back (showing a reflection symmetry, fig. 2.3).

The r-dimensional *Hypercube* has $N = 2^r$ nodes and $r2^{r-1}$ edges (undirected graph) [19]. Each node is labeled, virtually, with a unique r-bit binary string. Two nodes are connected if and only if their labels differ in exactly one bit (such a connection is called a dimension-k edge, where k is the position of the nonidentical bit within the label). Therefore, each node has degree r. Conversely, an r-dimensional hypercube can be constructed from two (r–1)-dimensional hypercubes by connecting the nodes having the same label via a dimension-r edge (the labels in the resulting hypercube will have one extra bit, '1' for denoting the nodes of the first hypercube and '0' for denoting those of the second). Note that the hypercube can be viewed as a folded up butterfly: each butterfly row corresponds to a hypercube node (consider merging each row of the butterfly to a single node and removing the resulting edge copies).

The r-dimensional *Cube Connected Cycles* (CCC) network is constructed from the r-dimensional hypercube by replacing each node with a cycle of r nodes. In such a cycle, each node is connected to a distinct edge of the initial hypercube (from those incident to the initial node, fig. 2.3). Overall, the r-dimensional CCC has $r2^r$ nodes, each with degree 3. Note that, from a computational point of view, the CCC, the Butterfly, and the Wrapped Butterfly are identical networks (compared to the hypercube, the CCC introduces a logarithmic slowdown [19]).

The r-dimensional *Shuffle Exchange* network has $N = 2^r$ nodes and $3{\cdot}2^{r-1}$ edges (undirected graph). Each node is labeled, virtually, with a unique binary string of r-bits and two nodes are connected if and only if: (i) their labels differ only at their last bit or, (ii) their labels are left or right cyclic shifts of each other. Edges of the former kind are called *exchange*, while those of the latter kind are called *shuffle*. The *Shuffle Exchange* is closely related to the *de Bruijn* network, which can be obtained by contracting out all the *exchange* edges from the first (we start with a *shuffle exchange* of dimension $r+1$ to derive a *de Bruijn* of dimension r). Both graphs share very interesting properties, which can be used even to formulate card tricks [19].

2.3 Circuit Models

The study of *switching circuits* dates back to the late 1930s and specifically to the papers of Shannon and Lupanov. Shannon used Boolean algebra to design and analyze circuits, while Lupanov worked on the question of how many gates a circuit must have to perform certain tasks [21]. Thereafter, the switching circuits and the logical design developed rapidly both in theory and in practice, with notable contributions by Pippenger, Borodin, Ruzzo, Savage, Cook, Allender, and many others.

Like the fixed-connection networks of processors, the circuits capture parallelism due to the ability of their network nodes to operate concurrently. The major difference of the two models lies in the reduced computational power of each node-gate; instead of processors, the circuits use gates implementing single operations. The following subsections present various models and variations. We start from the most simple abstraction, namely the Boolean circuits, and, by successively introducing more possibilities and details, we conclude with the VLSI circuits, which are used to model modern technology.

2.3.1 Boolean Circuits

The PRAM is a very attractive model of parallel computation due to its simplicity and its natural parallel extension of the RAM model. However, designing in such high level raises certain feasibility concerns. For example, does the PRAM model correspond to a physically implementable device? Is it fair to allow unbounded numbers of processors and memory cells? How reasonable is it to have unbounded size integers in memory cells? Is it sufficient to simply have a unit charge for the basic operations? To expose issues like these, it is useful to have a more primitive model that is closely related to the realities of physical implementation. A perfect candidate for this purpose is the *Boolean Circuit* model [22] [1] [5].

The boolean circuit is an idealization of real electronic computing devices. It abstracts their basic principles while, at the same time, it makes a compromise between simplicity and realism by ignoring many of the implementation details. Overall, a circuit consists of gates performing elementary logical functions and wires carrying information among the gates (figure 2.4). Formally, we let $B_k = \{f | f : \{0,1\}^k \rightarrow \{0,1\}\}$ denote the set of all k-ary Boolean functions, and we define

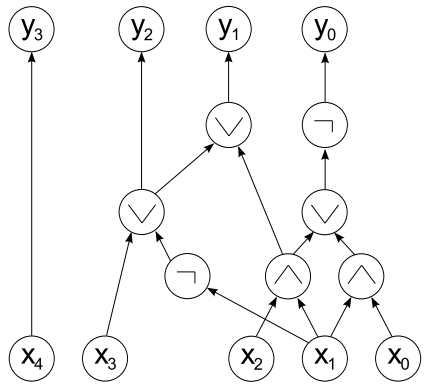


Figure 2.4: A Boolean Circuit (size=16, depth=4, width=3, n=5)

Definition 2.3.1. *A Boolean Circuit C_n is a labeled, finite, directed acyclic graph. Each vertex v has a type $\tau(v) \in I \cup B_0 \cup B_1 \cup B_2$. A vertex of type I is called an input and has zero indegree. The inputs of C_n are given as a tuple $\langle x_1, x_2, \ldots, x_n \rangle$ corresponding to n distinct vertices. A vertex with zero outdegree is called an output. The outputs of C_n are given as a tuple $\langle y_1, y_2, \ldots, y_m \rangle$ corresponding to m distinct vertices. Finally, any other vertex with $\tau(v) \in B_i$ has indegree i and is called a gate.*

Notice here that a Boolean circuit is memoryless. Most often, the gates used are the AND, OR, and NOT, and thus, the order of the inputs of each gate is irrelevant. We consider that, by inputing $\langle x_1, x_2, \ldots, x_n \rangle$ and outputting $\langle y_1, y_2, \ldots, y_m \rangle$, C_n realizes a certain function $f : \{0,1\}^n \to \{0,1\}^m$.

The resources of interest are the *size* of the circuit C_n, i.e., the total number of vertices in C_n, and the *depth*, i.e., the length of the longest path in C_n from an input to an output node. Additionally, we can measure the *width* of the circuit, which corresponds to the maximum number of gate values needing to be preserved, excluding inputs, when evaluating the circuit level by level (we consider as level i of C_n the set of vertices, which are located exactly i edges away from the input nodes of C_n). A circuit is said to be *synchronous* if all inputs to level i are outputs of level $i-1$.

Similar to the description of Turing Machines via strings of predefined format, each circuit C_n is described by a string denoted $\bar{C_n}$. Such a "blueprint" can be expressed in various formats, but most naturally, as a graph and a list of vertex types (labels)[12]. To be precise, we will describe here the *standard*

[12]another common format is the *straight line program*, a sequence of assignments to

encoding. Assume that we use the binary alphabet augmented with parentheses and commas. The descriptor $\bar{C_n}$ is a sequence of 4-tuples of the form (v, g, l, r), followed by two strings, $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$. For each vertex v of C_n, the $\bar{C_n}$ contains a distinct 4-tuple (v, g, l, r) at some arbitrary position within the encoding. The numbers l and r correspond to the vertices connected to the left and right inputs of v, respectively (all vertices are numbered uniquely and arbitrarily in the range $1, \ldots, size(C_n)^{O(1)}$). The number g denotes the type of the vertex v. In the two concluding strings, the number x_i is the vertex number of the i^{th} input, while y_j is that of the j^{th} output of C_n. Overall, $\bar{C_n}$ is an easy to generate and manipulate string of length $O\left(size(C_n) \cdot \log\left(size(C_n)\right)\right)$.

At this point, we can make a general observation regarding the circuit models of computation (not only Boolean). The circuits have a characteristic, which significantly differentiates them from any of the aforementioned processor-based models. A circuit consists of a fixed number of fundamental elements, namely the gates. Since a gate processes a fixed number of information bits, a circuit can only process inputs of fixed length. Hence, in order to solve a problem we must assemble an infinite number of well-defined circuits, one for each input length (contrast this to a TM, which can input a string of arbitrary length). To be precise, consider that the output length, m, is a function of the input length, n, and let us define

Definition 2.3.2. *A Family of Boolean Circuits $\{C_n\}$ is a collection of circuits, with each member C_n computing a function $f^n : \{0,1\}^n \to \{0,1\}^{m(n)}$. The function computed by $\{C_n\}$ is the function $f_C : \{0,1\}^* \to \{0,1\}^*$, defined by $f_C(x) \equiv f^{|x|}(x)$.*

One step farther, we can use circuit families to decide languages. We say that $L \subseteq \{0,1\}^*$ is decided by the boolean circuit family $\{C_n\}$ computing $f_C : \{0,1\}^* \to \{0,1\}$, when we have that $x \in L$ iff $f_C(x) = 1$.

Notice that the algorithmic solution (TM, or PRAM, or BSP, etc) of a problem is a finite object. On the contrary, the circuit family of a problem is an infinite collection of finite objects. Inevitably, this peculiarity of the model renders the portability and the study of circuit solutions questionable. The most common remedy is to impose a restriction on the construction of the circuit families by requiring a computationally simple rule for generating the

variables (wires) using instructions based on binary operators (as in Hardware Description Languages). In fact, a circuit is a DAG of a SLP [23].

various circuit-members of $\{C_n\}$. Intuitively, we require that all the members of the family are similar in structure, but differ in input sizes (consider, e.g., a XOR-tree to decide PARITY). In general, a family is said to be *uniform* if there exists an efficient algorithm to output $\bar{C}_n$ upon request.

Definition 2.3.3. *A family $\{C_n\}$ of Boolean circuits is "logspace uniform" if the transformation $1^n \rightarrow \bar{C}_n$ can be computed in $O(log(size(C_n)))$ space on a deterministic Turing machine.*

The *logspace* uniformity (Borodin-Cook uniformity) is the most widely used and, for polynomial size circuits, it can be equally defined by DTMs of logarithmic space in –unary– n (hereafter, the two versions of the definition are used interchangeably). Even though *logspace* DTMs solve a small class of problems, they can describe a wide class of useful circuit families. Furthermore, we can define several types of uniformity based on the resources and the capabilities of the machine generating the circuit descriptions [24]. Our choice on the exact type of uniformity depends mostly on the intended use of the circuit and on the complexity of the class under examination (see chapter 4). Generally, we avoid allowing more computational power to the constructor than to the circuit itself. Notice that the circuit constructor serves as a single object representing the entire family.

Not to overlook the *non-uniform* families, we mention here that they are also used in the study of circuits. First, they are famous for deciding non-recursive languages (i.e., undecidable by TMs). Consider for example a non-recursive unary language $L_u \subseteq \{1\}^*$ (at least one exists, because any non-recursive L can be reduced to some L_u, e.g., via binary-to-unary expansions). Strikingly, there exists a $\{C_n\}$ to decide L_u: for each $n \in \mathbb{N}$ we define C_n to be a tree of AND gates iff $1^n \in L_u$ [5]. Unfortunately, even though $\{C_n\}$ exists, there is no way to construct $\{C_n\}$ algorithmically, because no TM can decide whether $1^n \in L_u$ when generating $\bar{C}_n$. Second, non-uniform families can be used to prove lower bounds; if a function cannot be computed by such a family, then it cannot be computed by a weaker, uniform, family of the same size.

The aforementioned definitions give us the basis of the Boolean circuit model. Several variations have been proposed. In many cases we allow the use of gates with fan-in greater than 2, or even the use of *unbounded* fan-in gates. Such possibilities can "speed up" the computation by up to a $O(\log n)$ factor without increasing its size. Other variations limit the gate types to be used. For example, we can use only AND and OR gates to study *monotone* circuits.

The term monotone connotes here that the change of any input variable from '0' to '1' cannot result in an opposite change of the output (from '1' to '0'). Note that, since {AND,OR} is not a functionally complete set of logic operators, monotone circuits cannot compute all Boolean functions. Other circuits might even introduce new types of gates as, e.g., the *threshold* gate (outputs '1' iff the number of its input values add up to some threshold) for modeling neural networks. In another direction, we can impose restrictions on the structure of the circuit graph. For example, we can confine ourselves to the study of *planar* circuits, i.e., circuits that can be drawn in 2-D with no edge crossovers. Besides the above, many criteria and resource bounds can be used, or combined, allowing the power analysis and/or the development of circuit solutions for specialized applications.

The Boolean circuits will be examined in detail in the following chapters. In fact, we will consider them as the basic model of our parallel computation study. However, not all circuits proposed in the literature are built upon Boolean operations. We mention here the *arithmetic* circuits [22]. An arithmetic circuit is like a Boolean circuit except that the inputs are indeterminate $\langle x_1, x_2, \ldots, x_n \rangle$, possibly constants $c_i \in F$, F a field (e.g., the rationals Q), the internal gates are arithmetic operations $+, -, \times, \div$, and the outputs are elements of $F(x_1, x_2, \ldots, x_n)$. Arithmetic circuits are used to study the complexity of multivariate polynomials or rational functions. As with their Boolean counterparts, we are interested in the depth and size of uniform arithmetic circuits, such that we can bound the cost of computing certain polynomials. Overall, arithmetic and combinatorial (boolean) complexity are related and questions of one domain can be translated to the other; we even encounter the algebraic analog of "P vs NP", i.e., the "VP vs VNP" conundrum (VP is the class of polynomials computed by polynomial size arithmetic circuits).

2.3.2 FSM, Conglomerates and Aggregates

The Boolean circuits were defined as acyclic graphs. Such a definition prohibits the resource reuse during the computation, as well as the construction of memory cells. Arguably, an extension of the model allowing feedback paths within the circuit would enhance its ability of estimating hardware costs in the modern technology. Before presenting the *Aggregate* model for this purpose, we briefly describe the state machines, which precede the aggregates without necessarily basing on circuits.

Finite State Machines

Let us mention first the Finite State Acceptor (FSA), which is a weak relative of the Turing machine. This automaton is defined similarly to the TM, including a set of states Q, initial and final states, input alphabet A, and a transition function δ. However, the FSA has the ability to read its input tape only once starting from the leftmost symbol and moving to the right. At each step, the state q_i of the FSA changes according to a transition function of the form $\delta : (q_i, \alpha) \rightarrow (q_j)$, $\alpha \in A$. If we allow the FSA to output a symbol at each step, i.e., if we define an output alphabet B and a transition function $\delta : (q_i, \alpha) \rightarrow (q_j, \beta)$, $\beta \in B$, we get a Finite State Machine (FSM) [25].

The FSM is a notion used beyond the context of Turing machines. For instance, we can envisage an FSM as a graph consisting of a memory node and a processing node. Synchronously, the memory stores the current state, while the processing node reads the input symbol and the current state to generate a new state and an output symbol. Note that, if we restrict the input values to some fixed length, then we can implement the above nodes by using circuits. In fact, a common application of the FSM is in the design of digital systems.

To conclude the deterministic FSM presentation, we define two FSM types based on the form of the output function. The first is the *Mealy Machine* (introduced by Mealy in 1955) where the output value is determined at each step by both the current state and the current input symbol, i.e., $f_{out} : Q \times A \rightarrow B$. The second is the *Moore Machine* (introduced by E.F. Moore in 1956) where the output value is determined at each step by the current state alone, i.e., $f_{out} : Q \rightarrow B$. It can be shown that the two machines are computationally equivalent. Moreover, the FSMs decide exactly the class of regular languages (languages described by regular expressions) [23].

Conglomerates

In an attempt to study parallel computation with the use of state machines, Goldschlager introduced in 1977 the *Conglomerates*. A conglomerate is, essentially, a collection of interconnected, synchronous, deterministic, FSMs [7].

To be precise, a conglomerate $C = \langle I, F, f \rangle$ is an infinite set F of isomorphic copies of a single FSM, $M_i = \langle \Sigma, S, \delta, r \rangle$, where I is the input alphabet, Σ is the communication alphabet, S is the set of states, r is the number of inputs to each M_i, and δ is the transition function. The function f defines the

interconnection network within the conglomerate by generating the indices of the predecessors of each M_i. All sets, except F, are finite.

To explain the definition, a conglomerate consists of identical FSMs. Each M_i has a predefined number of r inputs and one output, which is connected to a number of distinct M_j's. That is, the fan-in of each M_i is bounded, the fan-out is unbounded, and the interconnection network of the M_i's is fixed. By convention, the input w of the conglomerate is distributed among the first $|w|$ FSMs (one bit per M_i). Hence, given that the number of FSMs used during the computation reflects its *hardware* complexity, all conglomerates have $\Omega(n)$ cost. The computation advances in steps: each M_i starts at its initial state and it successively generates internal states and output values. The total number of steps is considered as the *time* cost of the model. A conglomerate accepts w if at any time M_o enters a designated state q_{acc}.

The restrictions imposed on the singularity and the output of the above FSMs might seem restrictive for the computational power of the model. However, it is shown that the above conglomerate can simulate with no time loss the variant of the model, which allows many outputs and distinct types of FSMs (finitely many). Moreover, it is shown that the SIMDAG (see p. 13) can be implemented with a simple conglomerate, the connection function of which can be computed with only logarithmic-space RAMs. Analogously to the boolean circuit model, when studying the computational power of the conglomerate we impose a uniformity constraint on the construction of its interconnection. Arguably, a conglomerate is implementable if its connection function can be computed by polynomial-space RAMs. It is shown that the class of these conglomerates is equal to the class of polynomially space bounded computers. More specifically, the –parallel– time of such a conglomerate is polynomially related to the –sequential– space of a Turing Machine [7].

Aggregates

By combining the boolean circuits and the conglomerates, Dymond and Cook introduced the *Aggregate* model in 1980 [26]. Like bounded fan-in circuits, each aggregate consists of 2-pin boolean gates and works for inputs of fixed length. Like the conglomerates, the gates of an aggregate work synchronously. Unlike either circuits or conglomerates, the input convention for aggregates is such that either the hardware used or the time taken in a computation can be sublinear in the input size.

Formally, an aggregate β_n works on n input bits $x_0, x_1, \ldots, x_{n-1}$. It is a directed graph (not necessarily acyclic) of boolean gates and input nodes. A configuration C of β_n is an assignment of 0 or 1 to each node u. The assignment of u at C_{t+1} is determined by the operation of u and by its input pins at C_t. A computation of β_n is a sequence $C_o, C_1, \ldots, C_T$ of configurations. The maximum T over all inputs x, $|x| = n$, is the *time* complexity $T(\beta_n)$, while the total number of nodes is the *hardware* complexity $H(\beta_n)$. Given a distinguished pair $\langle u_0, u_1 \rangle$, the output of β_n is equal to the value of u_0 when u_1 turns to 1 for the first time (all nodes are initialized to 0). A family of aggregates $\{\beta_n\}$ recognizes a set $A \subseteq \{0,1\}^*$ if β_n recognizes A^n for all n. An aggregate family $\{\beta_n\}$ is uniform if each member β_n can be described by a DTM in space $\log(T(\beta_n) \cdot H(\beta_n))$ on input 1^n.

The input convention of the aggregate model is analogous to the TM, which ignores the space cost of its input tape. Here, each input node v is associated with a unique register R_v specifying the input bit x_i to be assigned to v. The R_v is considered as a sequence of $\log n$ special purpose gates. We can envisage v as a binary multiplexer controlled by the contents of R_v, which can change dynamically to specify any index of x. Overall, the hardware cost of node v is 1 and its assignment is delayed by $\log n$ steps. Note that if we assume that an aggregate β_n examines all the information of its input, then we have $T(\beta_n) \cdot H(\beta_n) = \Omega(n)$.

Regarding the computational power of the aggregates, it is shown that AG-TIME(T)=BC-DEPTH(T) and that UAG-TIME(T)=UBC-DEPTH(T). In other words, aggregates define exactly the same parallel time classes as boolean circuits, with and without the uniformity constraints (models of equal power). Moreover, the time of the uniform aggregate is polynomially related to the space of the Turing Machine. In terms of hardware, we get DTM-SPACE(S)=UAG-HARDWARE(S)=UBC-WIDTH(S). Finally, uniform aggregate hardware and time are themselves polynomially related [26].

2.3.3 VLSI

The aforementioned circuit models ignore the cost of placing wires within the circuit to connect its gates. Nonetheless, such cost is far from negligible in today's technology. Given the tremendous reduction of the size of a boolean gate, the wiring accounts for more than half the cost of the entire modern microchip [23]. The VLSI model, along with its several variations, was introduced in an attempt to include the details of the microchip technology to

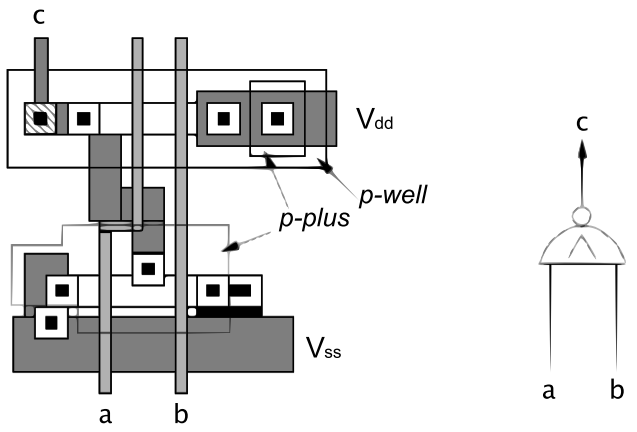


Figure 2.5: VLSI layout of a NAND gate (substrate, wells and wires shown)

the theoretical study of computation.

Design at the Physical Layer

Very Large Scale Integration (VLSI) is the process of creating integrated circuits by combining thousands of transistor-based boolean gates and memory cells into a single chip. As figure 2.5 shows, each gate is itself the combination of a number of semiconducting elements (forming a few interconnected transistors) occupying little square micrometers on the wafer. Starting from a thousand transistors in the early 1970s, microelectronics allow today the integration of a billion transistors in a single chip.

In the VLSI setting, the surface of a wire within the circuit is comparable to that of a single gate. Moreover, the fabrication process won't allow more than a constant number of wires to cross each other at any point of the wafer (typically, less than a dozen metal layers are used). As a consequence, the interconnection of an architecture has a definitive impact on its area cost. Besides, it affects the timing of the chip, as the signals propagate through the chip following the laws of physics and not instantaneously. Clearly, the wiring is as important as any other algorithmic aspect of the design and requires extra study (compared to the boolean circuit model).

The crossover constraint is even more strict for the gates, which cannot overlap at all. Even in the emerging technology, namely the 3-D Integrated Circuit, this constraint is still present, preventing the design of arbitrarily "thick" circuits. Overall, a VLSI circuit is quite similar to the planar circuit of

the boolean model. To underline their similarity we use the term *quasiplanar* [23], which denotes the fact that the VLSI circuit allows only a constant number of crossovers (planar implies zero crossovers).

Additionally to boolean gates and wires, the VLSI circuits include certain memory cells. Such a characteristic element is the *flip-flop*. We can envisage the flip-flop as a special purpose gate, which can store the value of its input bit for an arbitrary amount of time. The amount of storage time is controlled by designated signals, amongst which is the *clock* of the design. The clock signal traverses the entire circuit defining the time instants at which the computation advances to the next step. That is, in general, the VLSI circuit operates *synchronously*. The time cost of the computation is equal to the required number of steps (clock cycles) multiplied by the clock period. The clock period (operating frequency) is defined prior to the computation and is bounded by the propagation time of the largest asynchronous path in the circuit (path between two successive flip-flops, with transistors and wires).

The inputs and outputs (I/O) of a VLSI circuit are communicated via designated *pins*. The I/O pins are wires of much greater surface than the internal wires of the circuit, and thus, they are studied separately. Analogously to the I/O conventions adopted in the other models, the VLSI features its own I/O specifications. A port of a VLSI circuit can receive several input values, each one at a distinct time instant, e.g., at specified clock cycles. Also, an input can be supplied at several ports simultaneously. Moreover, a VLSI circuit has the possibility of reading a certain value more than once from its input ports. The VLSI computation is designed as *semellective* (read once) or *multielective* (read/request data multiple times). Usually, the I/O operations are *where-* and *when-oblivious* [23], i.e., they occur at specified pins and cycles. Similar conventions are adopted for the output ports of a circuit. In fact, any pin can be used as an input, output, or input/output of the chip. Given the above I/O possibilities, it is clear that a VLSI circuit can perform several, distinct, computations concurrently. In a *pipeline* fashion, it can input an new problem instance (a set of new values), while still processing some older values at a deeper stage within its architecture. Hence, to measure the performance of a chip we use (besides the computation *time* and *area*) the *rate* at which the chip can receive new instances, or the *throughput*: the number of instances solved, or information processed, over the total time consumed.

To complete the description of the VLSI technology we mention the parameter λ. In photolithography, a technique employed in microfabrication,

the geometrical patterns comprising the integrated circuit (figure 2.5) are printed on the wafer using light projection equipment. The projection features certain resolution, determined by physics and state-of-the-art technology. This resolution defines the minimum separation length, λ, between the elements of the chip (wires, transistors, etc). Typically, the width of an internal wire is a small multiple of λ and the area of a transistor is a multiple of λ^2. That is, the *minimum feature size* λ specifies the transistor density of a VLSI wafer, and to some extent, the computational power of a chip. Naturally, λ is significantly improved over the years. Starting from 10 μm in the 1970s, today we use 0.045 μm and we target 0.032 μm manufacturing process[13].

VLSI Technology Abstraction and Study

In all theoretical models of VLSI, circuits are made of two basic components, namely the *wires* and the *gates* (boolean operations, or flip-flops) [27]. The wires are embedded in the plane (quasiplanar) and have unit width and arbitrary length. They usually have unit bandwidth (one bit per transmission). Their delay is determined as follows: the *synchronous model* assumes unit delay independently of the length, the *transmission-line model* assumes that the delay is proportional to the length of the wire, whereas the *diffusion model* assumes that the delay is quadratic in the length [23]. Regarding the gates and the I/O pins, they are considered as nodes, which are connected via the above wires (hyperedges). Each node is characterized by a specific transmission function and it has constant area, constant delay and constant fan-in/fan-out. The space units are usually related to λ (e.g., width_unit=λ, area_unit=λ^2, and each gate, memory cell, port, or pair of crossing wires, occupies λ^2 area). Similar assumptions are made for the time unit in an attempt to abstract the fabrication parameters and measure the performance of the architecture independently of the technology, e.g., in terms of clock cycles and area units (the implementation values of which tend to reduce year after year). In practice, of course, an architecture is evaluated based on its implementation results including the operating frequency and the actual layout size of the chip.

Each VLSI model further defines the I/O, the degree of concurrency, the energy consumption, the performance, etc [23] [27]. These definitions, more

[13]according to Moore, Intel co-founder, the transistor density doubles approximately every two years. Of course, his "law" cannot hold for ever (due to atomic limitations).

or less, choose among the aforementioned VLSI technology possibilities. We can additionally mention here that, some authors assume an off-chip memory to store the I/O bits, such that multiple multilocal accesses are possible (the memory cost is usually excluded from the analysis, allowing a reduction of the chip's cost). The *concurrency*, c, of an architecture is equal to the number of problem instances solved simultaneously. Its area performance is equal to the total area units of the chip (smallest enclosing rectangular), divided by c. The time performance is equal to the time between the first and last I/O memory access, divided by c. For a detailed analysis, we even define the data format according to which the problem instance will be presented to the chip.

Having defined the model, one can continue with studying the computational power of the VLSI circuits. First, note that every boolean function can be implemented on VLSI because it has a planar boolean circuit (every function has a boolean circuit, which can be transformed into planar with at most a quadratic increase of its size). Second, note that the area of a VLSI circuit is proportional to the number of, say, the PRAM processors that can fit in the chip[14]. Given the similarity of the *time* resource of the two models (PRAM and VLSI), we are driven to the study of the product area$\times$time for the VLSI circuits (the AT measure). Clearly, this is analogous to the product processors$\times$time used in the PRAM case. This product constitutes a PRAM performance measure, which is lower bounded by the minimum sequential time, T_s, of the problem. Similarly, we have that $AT = \Omega(T_s)$. Note that, for low complexity problems we can devise circuits, which are optimal in terms of AT (performing at its lower bound). However, the AT bounds are not the strongest studied in VLSI. Stronger complexity bounds are achieved for the AT^2, which is the most common VLSI complexity measure used in the literature (A^2T is also used) [23].

Various methods can be employed in order to prove an AT^2 lower bound for a given problem. The most common method is to use a "fooling argument", i.e., show that a minimum amount of information must be transfered between two parts of the chip, or else one part can be fooled into thinking that the other inputs data different than the actual. Transferring such information requires either some large amount of time or a large number of

[14]likewise, the area is analogous to the amount of information memorized by the VLSI circuit at any time instant. Recall that this also holds for the space of a Turing machine. Intuitively, the area complexity of an algorithm is loosely coupled to its space complexity.

wires (area), and thus, a lower bound on the AT^2 of the chip can be proved. Another method to prove VLSI lower bounds for a specific problem is by studying the complexity of the planar circuit of that problem (recall that the VLSI is already quasiplanar). It can be shown that any VLSI computation can be simulated by planar circuits of size $O(AT^2)$ and $O(A^2T)$ [23].

To report some examples, the "n-tuple prefix sum" has VLSI complexity $AT = \Omega(n)$, while the "n×n matrix-vector multiplication" has $AT = \Omega(n^2)$. Both problems feature AT–optimal circuits, which base on H-trees[15]. The "integer multiplication" and the "n-tuple shifting" have VLSI complexity $AT^2 = \Omega(n^2)$. Moreover, they both feature AT^2–optimal semellective circuits based on 2-D systolic arrays. The "n-tuple sorting" has VLSI complexity $AT^2 = \Omega(n^2 \log n)$, while the "n-point FFT" has $AT^2 = \Omega(n^2 \log^2 n)$ [27]. The "n×n matrix-matrix multiplication" has complexity $A^2T, AT^2 = \Omega(n^4)$. Separate bounds for A, T are also computed for many problems, as for example the $A = \Omega(n)$ of the "n-vector shifting" (shifting in $T = O(\sqrt{n/\log n})$ is a characteristic case where most of the VLSI area is consumed by wires) [23].

We conclude the VLSI model by noting its area-time tradeoffs (Thompson, 1980). As mentioned above, problems feature certain lower bounds on their AT, AT^2, and A^2T complexities. Such bounds imply that, when designing optimal VLSI circuits, in order to improve A we have to degrade T by some certain factor, and reversely. Of course, analogous quantitative or qualitative estimations hold for parallel computation in general.

2.4 More Models and Taxonomies

Besides the aforementioned, the literature includes a variety of models for studying and developing parallel algorithms. A number of these models are based on extensions of the ideas described so far, while others incorporate entirely novel techniques and technology trends. To complete the survey of the current chapter, the following subsections refer to a representative set of the remaining parallel models of the literature. Moreover, we report the proposed architectural/programming taxonomies, which are used to classify the parallel computers today. We begin with the details of a classical model, which is widely used in complexity theory.

[15]similar to the binary tree, except that each node has 4 children. In the binary tree, one can see a repetition of the shape Λ in a recursive fashion. In the H-tree, the repeated shape is H, rendering the mapping of the tree on the plane more efficient in terms of area.

2.4.1 Alternating Turing Machine

Alternating Turing Machines (ATM) were introduced in 1981 by Chandra et al. [28]. They are generalizations of Non-Deterministic Turing Machines (NDTM). Like the NDTM, the ATM is not a realistic model of computation. However, it is very useful in studying certain languages with no obvious short certificate for membership. Moreover, ATMs capture the parallel complexity of various classes of problems.

Recall that a NDTM is essentially characterized by a computation tree, the nodes of which represent the configurations and the edges represent the potential transitions of the machine [5]. In the case of an NP language, the NDTM accepts if *there exists* an accepting leaf in the tree. In contrast, in the case of a co-NP language, the NDTM accepts if *for all* leaves we get a negative response. Loosely speaking, one can see only OR ($\exists$) nodes at the first tree and only AND ($\forall$) nodes at the second. The ATM generalizes by allowing both types of nodes in the same computation tree.

In an ATM, each state $q_i \in Q$ is labeled with its own logical connective $l \in \{\vee, \wedge\}$. Consequently, each configuration of the ATM is also labeled by l. Let us define recursively a configuration to be *accepting* if: (i) the state of the configuration is a 'yes' state, (ii) it is labeled $\vee$ and at least one of its children is an accepting configuration, or (iii) it is labeled $\wedge$ and all of its children are accepting configurations. The ATM computation *accepts* if the initial configuration is an accepting configuration (notice the resemblance to the evaluation of a circuit).

The remaining of the ATM is defined exactly as the NDTM, i.e., it is a seven-tuple $\langle k, Q, \Sigma, \Gamma, \delta, q_0, g\rangle$ where k is the number of tapes, Q is the finite set of states, Σ is the input alphabet, Γ is the work tape alphabet, δ is the transition relation, q_0 the initial state and $g : Q \rightarrow \{\vee, \wedge, yes, no\}$ is a function labeling the states of the ATM. The input tape is read only. An ATM decides a language $L \in \Sigma^*$ when it accepts x iff $x \in L$. Let ATIME($f(n)$) be the class of languages decided by ATMs, all computations of which on input x terminate after at most $f(|x|)$ steps. Similarly, ASPACE($f(n)$) includes the languages decided in alternating space $f(|x|)$. Note that, besides time and space, we use the *alternations* to study the complexity of an ATM algorithm; this measure counts the maximum number of changes between $\vee$ and $\wedge$ states over all paths of the computation tree.

Regarding the computational power of the model, the ATMs accept exactly the same set of languages as the DTMs (the recursively enumerable

languages) [28]. To be more precise, it was proved that alternating space is equal to deterministic time, only one exponential higher. Also, alternating time is polynomially related to deterministic space. We mention here that the idea of alternation can be applied in other models too. In some cases, alternation enhances the computational ability of the entire model: alternating pushdown automata are strictly more powerful than nondeterministic pushdown automata (the former accept languages accepted by DTM in time c^n, while the latter do not).

Regarding the interests of this thesis, the ATM can be viewed as a machine spawning processes with unbounded parallelism. That is, when a node v of the tree branches, it spawns new processes, which will run to completion and report acceptance or rejection back to the node v. Afterwards, the node v combines the answers in a simple way (AND, OR) and forwards the result to its own parent, and so on. Note that the process communication is confined only between parents and children. Overall, the ATM is closely related to the models of parallel computation. A characteristic example is the correspondence between the number of alternations and the depth of the unbounded fan-in circuits. The ATM is used to study parallel complexity theory and we will come back to it in chapter 4.

2.4.2 Other Models

The **k-PRAM** model was introduced in 1979 by Savitch and Stimson [29]. Unlike the ordinary PRAM, this model has no shared memory. Instead, a processor communicates directly with another processor by making a call and passing parameters via some channel registers. The processor cannot communicate with an unbounded number of distinct processors. The parameter k denotes the branching factor out of each processor. The above idea is similar to that of the **recursive-TM**. Such a machine has two input tapes, one containing the input of the problem and one containing the output of the calling machine. That is, the TM can spawn an identical TM by passing a parameter via its output tape (to the input tape of the new TM). In this way, the TM performs a subroutine call and waits until its termination (accept/reject state) before continuing with the remaining of the computation. One step farther, the **parallel-TM** is defined by allowing two output tapes per machine and two simultaneous subroutine calls, which will spawn two new machines working in parallel.

The **Hardware Modification** and the **Parallel Pointer Machines**

are essentially two versions of the same model, which is characterized by its reconfigurability at run-time [30]. As with the Conglomerate model, HMM and PPM consist of a finite collection of FSMs pairwise connected in a direct fashion. However, as opposed to the Conglomerate where the connections are fixed, the communication channels between the FSMs of the HMM and PPM can change during the computation. To be precise for the PPM case, each FSM has k input lines (links, or pointers) and, at each step, it can redirect one of its links to any other FSM, which is located no more than two hops away from it. These models were introduced (1980s) to study the computational power of machines falling in between the two major categories of models: those with bounded and fixed inter-process communication (e.g., circuits), and those with unlimited communication links (e.g., shared memory). Interestingly, the results of the literature show that the PPM falls very close to the second and more powerful category, i.e., it can simulate sequential-space $S(n)$ in only $O(S(n))$ parallel-time (and not $O(S^2(n))$).

The **Reconfigurable Mesh** is a model capturing features from the fixed-connection networks and the VLSI, characterized by broadcast buses and reconfigurability at run-time [31]. Specifically, it consists of a $\sqrt{N} \times \sqrt{N}$ array of processors connected via a grid-shaped reconfigurable broadcast bus. Each processor has four local switches, which allow the bus to be divided in sub-buses to connect various sets of processors, instantly. The broadcast on the bus can be performed in an exclusive-write or common-write fashion. Similar to the VLSI, we assume $\Theta(N)$ total area cost, unit area for a link between adjacent switches, and either unit-time delay (any broadcast operation consumes $\Theta(1)$ time), or logtime delay (broadcast in $\Theta(\log s)$ time, where s counts the switches traversed by the signal). Note that under the unit-time delay assumption, the reconfigurable mesh owns greater power than the ordinary circuits and the PRAM, e.g., it computes the PARITY in $O(1)$ time.

The **Vector Machine** is, roughly, a RAM including bit-vector registers and bitwise instructions: AND/OR/NOT operations, vector shifting, etc [32]. These instructions are executed in one cycle and involve all bits of the register (of infinite length), i.e., we get parallel processing at bit level. The ability of the machine to shift integers by multiple bit-positions to the left (equal to the length of the integer) in a single cycle, gives extraordinary power to the model. In fact, the Vector Machine can generate exponential information in only polynomial number of steps. This is also a characteristic of the **M-RAM** model, which is a RAM equipped with a unit-cost instruction for multiplying variables. It is proved that both machines can solve NP-complete

problems in polynomial time [32] [5].

The **CLUMPS** model is a generalization of the LogP (Campbell, 1994) [15]. A set of autonomous processors communicate directly via a partitionable interconnection network. Similar parameters to that of the LogP are used for measuring the performance of the parallel algorithms. Also, a network capacity is determined. However, instead of the global parameters L and g of the LogP, the CLUMPS uses latency and bandwidth parameters, which are defined for certain regions of the machine. Each region determines a collection of processes allocated to a specific partition of the interconnection network. Inter-region communication is costed by merging the parameters of the corresponding regions. The distributed memory of the model can be viewed as a multi-level hierarchy with each level consisting of the collective memory of a region (the local memories of each processor are located at the lowest level, while at the highest level we get the memory of the entire machine). Overall, introducing numerous regions and parameters is liable to increasing the difficulty of the analysis, but it also allows algorithms to run faster if we carefully map the processes to appropriate partitions.

The **Cellular Automata** came into existence with the work of von Neumann, among others, during the 1940s. Today, the model is used in biology, chemistry, physics, as well as in computation theory [33]. Similar to the Conglomerates, a CA is a –possibly infinite– collection of finite state automata. These FSAs, however, are not defined here with explicit I/O or interconnections; we are interested only on their states. Their communication is performed indirectly by relating their states. Specifically, each FSA is called here a *cell*. The cells are placed on a discrete regular grid, i.e., on a lattice (most often, the CA is a 2D array of cells). A *neighborhood* formula associates each cell with a finite number of specific cells in its vicinity. The neighborhood determines the state of the cell during the computation. The computation is a synchronous evolution of *configurations*, i.e., of vectors describing the states of all cells at each time instant. Starting from an initial configuration, the state of each cell changes at each step according to a predefined *rule* (a transition function), which takes as input the previous states of its neighborhood cells. This rule is the dominant characteristic of each CA. Interestingly, one can see certain *patterns* evolving within the configurations during the computation (shapes often appearing in the study of CAs, or even in the nature itself). Well-known problems expressed in terms of CAs are the "configuration reachability" (can we obtain picture X starting from Y) and the "predecessor existence" (deciding whether X has a certain predecessor

Y). The former is undecidable for infinite 1D CAs (analogously to the halting of a TM), while the latter is NL-complete for 1D CAs and NP-complete for 2D CAs. We know that CA can simulate TM and vice versa.

Natural Computing is a field of research swarming with peculiar models of parallel computation [34]. It is concerned with the computing paradigms encountered in nature, i.e., it studies the principles/mechanisms underlying natural systems, which can inspire humans to design new algorithms and computers. Since nature works most often in parallel, most of the resulting models perform parallel computations. The cellular automata mentioned above is a first, characteristic, example of natural computing inspired by the self-reproductive mechanisms in biological organisms. Other examples include neural networks, membrane computing, molecular (DNA) computing, evolutionary (genetic) algorithms, ant colony optimization, artificial immune systems, etc. Some of these attempts lead to heuristic algorithms (e.g., for search problems), while others can lead to the design of computational models with much greater power than the ordinary TM (e.g., the "transition P system" can solve PSPACE-complete problems in polynomial time [35]). The literature on natural computing is vast, varying from pure theoretical to experimental laboratory research.

Dynamic Multithreading is a more of a programming model, which can also be used to study parallel algorithms [3]. The model assumes an ideal multiprocessor and abstracts the need for communication protocols and load balancing (such issues are handled in practice by a *concurrency platform* placed between the programmer and the actual multiprocessor). A parallel program consists of ordinary instructions (add, jump, etc) plus three special purpose instructions: *spawn*, *parallel*, and *sync*. The spawn command allows for nested parallelism, as the programmer uses this to start a new subroutine (a thread) to work in parallel with the caller routine (via a call with parameters). The parallel command is used to execute certain loops in parallel without explicitly calling subroutines. The sync command determines specific time barriers within a routine, forcing the routine execution to wait until all of its subroutines have terminated and returned their results. The above three commands are *concurrency keywords* and are inserted in a –otherwise serial– pseudocode to express the logical parallelism of the computation, indicating which parts of the computation may proceed in parallel. Alternatively, one can construct the directed acyclic graph of the computation with nodes-instructions and edges-dependencies (fanout>1 denotes a spawn and fanin>1 denotes a sync). We are interested in measures such as

Model	*Complexity Measures*		
Turing Machine	time(steps),	space(cells),	reversals
Alternating TM	alternations,	space(cells),	time(steps)
PRAM	time(instr),	processors,	processors$\times$time
BSP	supersteps,	processors,	$g \cdot h_i + s + m_i$
Boolean Circuit	depth(gates),	size(gates),	width(gates)
VLSI Circuit	time(sec),	area(μm^2),	area$\times$time^2

Table 2.1: Most common complexity measures (models' resources)

the *work* of the computation (its sequential time) and the *span* (its parallel time, i.e, the longest path in the dag).

Papadimitriou and Yannakakis described in 1990 a method for evaluating the performance of a parallel algorithm with the solution of a scheduling problem [36]. PY view the parallel algorithm as a directed acyclic graph indicating the elementary computations of the algorithm and their interdependence. They do not simply measure the depth of this graph to find the time complexity of the algorithm, because such a method disregards the communication delay of the processors. Rather, they introduce a parameter τ capturing the delays of the underlying machine and they define a scheduling problem on an unbounded number of processors. As a schedule constraint, each task T_i must be completed by time t–1, where t denotes the start of any child T_j of T_i, or by time t–1–τ when T_i and T_j are not scheduled on the same processor. The optimum makespan of this scheduling problem is regarded as the complexity of the algorithm under examination (given as a function of τ). Since scheduling is a NP-complete problem, PY devise a scheduling method to approximate the optimum makespan to within a factor of 2.

We conclude the model presentation in this thesis by summarizing the resources of interest for the most widely used models in table 2.1. These resources are used to measure the complexity of the implemented algorithm or circuit. Note that the first column, provably, captures the time complexity of the computation. Interestingly, the measures of the second column are also related: they reflect the amount of information carried during the course of the computation.

2.4.3 Taxonomy of Parallel Computers

The models mentioned so far, along with their variations, lead to the development of a plethora of processor-based parallel machines. Considerable effort has been spent in identifying the commonalities of these machines and classifying them based on practical criteria (programming, architectural, etc) [2] [37].

The first well-cited attempt was made by Flynn in the 1960s. Flynn based his taxonomy primarily on the programmer's viewpoint. Specifically, Flynn was interested in whether the processors of a system will execute the same instruction at any given cycle (central control), or not. Moreover, Flynn was interested in whether each processor will operate on the same data with the other processors at each cycle (common input), or not. In other words, Flynn's criteria are the *instruction stream* and the *data stream* of the system. In a "Single Instruction - Multiple Data stream" computer (SIMD) all processors execute the same command per cycle, even though each one of them processes distinct data (recall the SIMDAG). The SIMDs are also called array or systolic processors. In a "Multiple Instruction - Multiple Data stream" (MIMD) every processor has its own program and processes a distinct data set (recall the general PRAM). Of course, there exist uniprocessor systems (SISD), as well as specialized MISD systems. The SIMD multiprocessors were widely used in the past decades. Today, the top supercomputers are based on MIMD architectures. During the 1980s, the MIMD class was further divided into the SPMD and the MPMD subclasses by considering the diversity of the programs stored in the processors of the parallel machine. The "Single Program - Multiple Data stream" systems compose a wide and important class, which can be viewed as a compromise between SIMD and MIMD: each processor is given its own copy of the same program. As a result, each processor can follow a separate branch during the program execution, e.g., with an if–then–else command. This allows for more flexibility and less idle time per processor. However, SPMDs require synchronization points within the program (in contrast to the SIMD where, by definition, every processor knows the state of the others). Note that SPMD is nowadays the most common style of parallel programming and forms the basis of the Message Passing Interface (MPI)[16].

[16]MPI is a language-independent communication protocol used to program a distributed memory system. It emphasizes in message passing among the processors. It is widely used today in computer clusters and supercomputers.

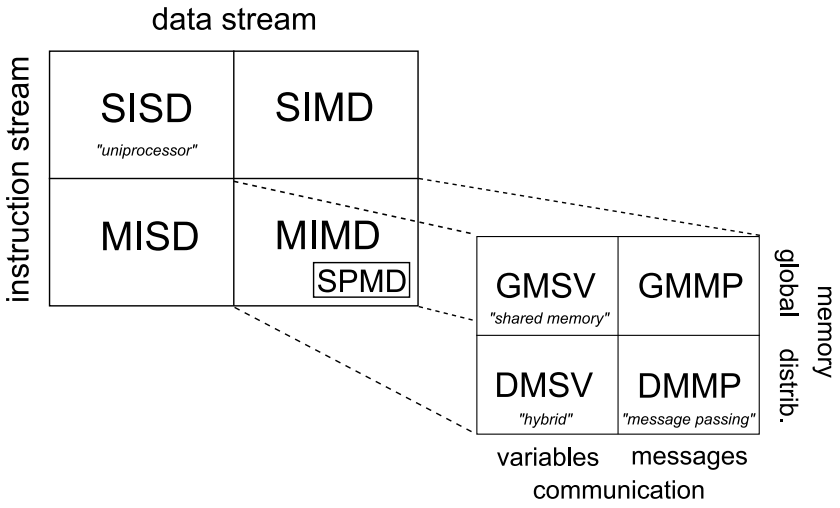


Figure 2.6: The Flynn-Johnson classification of computer systems

Following Flynn, Johnson proposed his own taxonomy in 1988 [37]. Johnson was interested solely in the MIMD class and extended his view to include architectural criteria. That is, he distinguished between shared and distributed memory systems. He also distinguished between systems, which use messages for the communication of the processors and those, which use shared variables. The characteristic member of the "Global Memory - Shared Variables" (GMSV) class is the PRAM, while BSP characterizes the "Distributed Memory - Message Passing" (DMMP). The GMMP systems have isolated process address spaces (usually virtual, within the same memory). In the DMSV, or "hybrid" systems, the memory is distributed among processors, but communication and synchronization take place through shared variables (e.g., via a Butterfly network). Figure 2.6 illustrates the Flynn-Johnson taxonomy.

We continue with an alternative view of the MIMD class, which highlights the structure of the interconnection network and not the programming model underlying the processor communication. That is, we examine the physical connections of the processors. We distinguish between a *bus* architecture and a *switched* network. The former is a single medium connecting all processors, while the latter uses fixed connections between certain pairs of processors. In a bus architecture the bandwidth is shared, whereas in a switched network every pair has its own channel. The second criterion used

here is the standard distinction between shared and distributed memory. However, the shared memory architectures are now called *multiprocessors*, while the distributed memory architectures are called *multicomputers* to denote the autonomy usually encountered in each component of the system. Multiprocessors are considered as *tightly coupled* systems, in the sense that their components tend to be in close proximity with short communication delay and high bandwidth. Multicomputers on the other hand are considered as *loosely coupled* systems, (e.g., a number of workstations on an Ethernet LAN). Roughly, bus–multiprocessors correspond to GMSV systems, while bus–multicomputers correspond to DMMP systems.

Besides the above classifications, one can point out the UMA and NUMA distinction. In the "Uniform Memory Access" architecture all processors share the common memory uniformly, i.e., the penalty of accessing a non-local memory does not depend on the relative position between the processor and the memory module. If this is not the case, we have a "Non-Uniform Memory Access". The first UMA example that comes in mind is a simple PRAM, while the first NUMA is a fixed-connection network of memory/processor components. In general, multiprocessors are considered UMA and multicomputers are considered NUMA, but the precise classification depends on the specifics of the implemented machine. Let us finally mention an architectural distinction between "Paracomputers" and "Ultracomputers" introduced by Schwartz in 1980; the former use an idealized central memory like the PRAM, while the latter use a number of memory modules and processors connected via an interconnection network (message passing).

To conclude this subsection, we report an attempt to classify entire models of computation instead of mere processor-based systems. Cook proposed in 1981 a "structural" classification [29], according to which we have the *fixed* and the *modifiable structure* models. The first is a class representing parallel models whose interconnection pattern is fixed during the computation, while the second class represents models whose communication links are allowed to vary. Examples of fixed structure models are: the ATM, the bounded fan-in uniform Boolean Circuit families, the uniform aggregates, and the k-PRAM. Examples of modifiable structure models are: the Hardware Modification Machines, the PRAM, the SIMDAG, the Vector Machine, and the unbounded fan-in uniform Boolean Circuit families. Note how the above distinction would reflect to the sequential computation models: we would get the fixed versus the modifiable *storage* structure models, e.g., the Turing Machine versus the RAM. Interestingly, we know that the fixed struc-

ture models are slightly less powerful than the modifiable: for the former we usually get DSPACE($S(n)$) $\subseteq$ F-TIME($S^2(n)$) $\subseteq$ DSPACE($S^2(n)$), while for the latter DSPACE($S(n)$) $\subseteq$ M-TIME($S(n)$) $\subseteq$ DSPACE($S^2(n)$).

Chapter 3

Model Simulations and Equivalences

The literature includes a plethora of computational models and model variations, either sequential or parallel. The extent at which these models differ from each other can be explained, at some level, by the needs of the scientific community. We can describe here two opposite modeling trends. The first mandates the definition of parameters and structures, which comply with the cutting-edge technology. The second abstracts as many details as possible to create "simple" models. Clearly, the first is a trend promoted by the fabrication industry, while the second is promoted by the theorists. In between, one can find the numerous model variations described in the previous chapter.

Such a plethora of models might lead to a confusion regarding the scope and the computational abilities of each machine. From a theoretical viewpoint, we must settle whether these models are able to compute the same functions. From a practical viewpoint, we must find ways to "translate" programs automatically from one model to the other. Both of these goals can be achieved with the development of certain *simulation* techniques. Loosely speaking, we say that a machine simulates another machine when it executes a program mimicking the functionality and following the computation steps of the second machine, in order to generate the same output on any given input. If we develop a generic technique to simulate one model with another, we can run any program of the first to the second, at the expense of some extra resources (time, space, etc). The amount of the extra resources is used to compare the computational power of the models. Trivially, if two models can simulate each other, then they have the same computational ability.

3.1 Shared Memory Models

The existence of numerous PRAM variations gives rise to a common theoretical question regarding the computational power of each machine. On one hand, we expect that they are all computationally equivalent, i.e., they can compute the same functions regardless of the required time/space/processor resources. On the other hand, we expect that certain PRAM variations require less steps than others when computing specific functions.

3.1.1 Overview

The PRAM variations share common instruction sets and operating principles (see section 2.1.1). Consequently, it is possible to "translate" a program, which was coded for a specific PRAM model, to a program running on a different machine (to compute the same output). Most often, only the read/write operations to the shared memory need to be restructured. Several techniques have been published in the literature for this purpose, allowing the *simulation* of one machine by another. These simulations prove the equivalence of the various PRAM models with respect to their computational ability. However, they also show that not all models are equally powerful in terms of time, space, or processors. The "weak" models usually require more resources in order to complete their calculations.

We say that two PRAM models are equally powerful if they can simulate each other with a constant factor slowdown [2]. A PRAM model M_1 is strictly less powerful than another model M_2 (denoted as $M_1 < M_2$) if there exists any problem for which the former, M_1, requires significantly more computational steps than the latter, M_2. For instance, the detecting-CRCW is strictly less powerful than the max-CRCW PRAM (writing the maximum value offered): the latter can find the largest number of a size-p vector V in a single step (assuming that p is equal to the number of processors, consider a parallel execution where each processor P_i reads $V[i]$ and writes it to the shared memory cell $p+1$), whereas the detecting-CRCW needs at least $\Omega(\log p)$ steps.

The PRAM models can be ordered according to their power simply by examining their cross-simulations. More specifically, given two models M_1 and M_2, if M_2 can simulate M_1 with only a constant factor slowdown, whereas the converse cannot hold, clearly, we have that $M_1 < M_2$. Such model comparisons can also involve the number of processors and/or space required

Original Model	*#Proc*	**Simulating Model**	*#Proc*	**Simulation Time**
priority-CRCW random-CRCW common-CRCW	p	EREW CREW	$\geq p$	$\Theta(\log p)$
priority-CRCW random-CRCW	p	common-CRCW	kp	$\Theta\left(\frac{\log p}{k(\log\log p - \log k)}\right)$
priority-CRCW	p	random-CRCW	kp	$O\left(\frac{\log\log p}{\log(k+1)}\right)$
CREW	p	EREW	≥ 1	$\Omega\left(\frac{\sqrt{\log p}}{\log\log p}\right)$
priority-CRCW random-CRCW common-CRCW	p	EREW CREW	$\geq p + m^2$	$\Theta\left(\frac{p}{m}\right)$
priority-CRCW random-CRCW	p	common-CRCW	p	$\Omega\left(\log\frac{p}{m}\right)$
priority-CRCW	p	random-CRCW	kp	$\Omega\left(\log\frac{p}{m}\right)$

Table 3.1: PRAM simulations (p: processors, m: memory cells) [6]

by each machine. A chain of some well known relations among models using the same number of processors is [2]:

$$\text{EREW} < \text{CREW} < \text{detecting-CRCW} < \text{common-CRCW} < \text{random-CRCW} < \text{priority-CRCW}$$

Note that, even though all CRCW models are strictly more powerful than the EREW (the weakest model), the EREW can simulate the most powerful CRCW listed above with at most logarithmic slowdown (assuming that extra memory cells can be used during the simulation).

Towards a more accurate view of the PRAM model relations, Table 3.1 summarizes the results of various simulations [6]. The last column of the table reports the slowdown factor for each case (the time of the simulation over the time of the original execution). The upper part of the table assumes no memory constraints, while the last three rows refer to simulations where the models (both the original and the simulating) use only m shared memory cells and the input is initially distributed among the processors' local memories. The comparisons highlight the speed of each model. The slowdown suffered in each case depends on the number of the processors and not –directly– on the input size. Notice that the differences between the CW models are sub-logarithmic, i.e., they are much smaller than the logarithmic slowdown of the EW model. Also, the memory constraint can lead to significant time losses (in the EREW case it results in a potentially linear slowdown).

3.1.2 Simulations between PRAMs

To better understand the aforementioned simulation techniques, we will study the most prominent of them in the following theorems. We begin by making clear that each of the described techniques allows the simulation of the entire functionality of a PRAM model, and not some specific algorithm. In other words, given two models M_1 and M_2, we are interested in a generic technique to support the execution of any M_2 algorithm to the M_1 machine. Most often, each technique employs certain parallel algorithms to simulate the read/write operations of the original model. Naturally, we examine the amount of resources required to simulate a model with a weaker model (the converse simulation is straightforward).

SIMULATION: *priority-CRCW on EREW and CREW*

The next theorem [20] establishes that any priority-CRCW algorithm with running time $T(n)$ in p processors can be simulated by an EREW algorithm with running time $\Theta(T(n) \cdot \log p)$ in p processors. Since the only difference of the two models lies in the accessing of the shared memory, we are interested only in simulating a concurrent read –or write– operation with the EREW model. The remaining operations will be executed by the simulating algorithm in their original form (instruction copy).

Theorem 3.1.1. *A concurrent read (or write) operation of a p-processor priority-CRCW PRAM can be implemented on a p-processor EREW PRAM to run in $O(\log p)$ time.*

Proof. Let $Q_1, \ldots, Q_p$ and $P_1, \ldots, P_p$ denote the processors of the priority-CRCW and the EREW PRAMs respectively. Each P_i will simulate the operations of each Q_i. Moreover, we use the idea that the $P_1, \ldots, P_p$ processors can cooperate to find out which Q_j has the priority to write (read) on a memory cell by sorting a list of the $Q_1, \ldots, Q_p$ concurrent requests in $\Theta(\log p)$ time. For simplicity, we assume here a SIMD algorithm where the conflict resolution procedure is started only when the program counter reaches a read/write operation. Note that for MIMD algorithms the slowdown remains $\Theta(\log p)$ even with a naive application of the conflict resolution procedure: we execute the procedure at each step of the algorithm, regardless of which processors actually access the shared memory, and then continue with the next non read/write instruction.

In a priority-CRCW PRAM each processor Q_i has a unique ID_i ($\log p$ bits). In a EREW we can also assign IDs and we can reserve memory locations M_1 to M_{2p} to be used only during the conflict resolution procedure. Let us assume that during the CRCW computation Q_i requests memory location $M_{f(i)}$. To start the conflict resolution procedure, each EREW processor P_i will append the ID_i number to its $M_{f(i)}$ number and will store the resulting value $< M_{f(i)}, ID_i >$ in M_i. Next, all the EREW processors will execute a parallel mergesort algorithm in $\Theta(\log p)$ time [20] to sort the list formed in memory locations M_1 to M_p. In this way, the requests will group according to their addresses (namely $M_{f(i)}$) and the rightmost member of each group will correspond to the lowest ID_i of the processor that requested $M_{f(i)}$ (i.e. to the one with the highest priority for accessing $M_{f(i)}$). Afterwards, each EREW processor P_i will read the locations M_{i-1} and M_i in two steps, in order

to determine whether there is an access conflict (same address request). If not, P_i can perform the access described in M_i. Clearly, the accesses chosen by this procedure will correspond to the accesses performed by the CRCW processors. Memory locations M_{p+1} to M_{2p} can serve as a buffer for communicating the request data between the requesting processor P_i and the processor P_j which actually executes the request. For example, in a write operation P_i will temporarily store its data in M_{p+i} and afterwards, if P_i gets the highest priority, the data will be forwarded to $M_{f(i)}$ by P_j (which, besides $M_{f(i)}$ knows the ID_i and thus can locate M_{p+i}).

The running time of the above simulation is determined mainly by the mergesort execution time $\Theta(\log p)$. The remaining operations can be completed in $O(1)$ time as the arithmetic computations and the consequent accesses to the memory locations M_1 to M_{2p} require constant time (by construction of the simulation, no conflicts delay these accesses). □

Using the same idea, any CREW algorithm can be simulated by a EREW within the same logarithmic bound. Moreover, since a EREW can be simulated by a CREW PRAM using the exact same operations (no slowdown), we deduce that the CREW can simulate the priority-CRCW with at most a logarithmic slowdown [20].

SIMULATION: *priority-CRCW on random-CRCW*

The next theorem [20] [6] shows that the computational power of the random-CRCW is quite similar to the power of the priority-CRCW. Their only difference is in the writing of the shared memory, where

Theorem 3.1.2. *A concurrent write operation of a p-processor priority-CRCW PRAM can be simulated on a p-processor random-CRCW PRAM in* $O(\log \log p)$ *time.*

Proof. We prove that resolving the conflict of an arbitrary number of random-CRCW processors, which try to access the cell M_C requires $O(\log \log p)$ steps.

Assume that the p processors are partitioned in $\sqrt{p}$ groups of $\sqrt{p}$ processors each. In the first step, the processors of group G_j that request an access to the cell M_C will write their IDs into cell M_j (as usual, we reserve a number of shared cells for use only in the conflict resolution procedure). A random –or none– processor P_i^j in each group G_j will succeed in this operation. In the second step, all the involved processors read their group cell

M_j to identify P_i^j. This concludes the first phase of the simulation. For the remaining phases of this simulation, P_i^j will be the representative of G_j.

In the second phase, we have a group G_R of $\sqrt{p}$ representative processors along with the $\sqrt{p}$ groups of the first phase (missing their representatives). In a recursive fashion, the processors of each of these groups are partitioned in $\sqrt{\sqrt{p}}$ sub-groups and, by following the same steps with phase 1, new representatives are selected. Again, all groups operate in parallel.

The recursion continues until the processors contract into groups of two. At this point, it is easy to decide which one of the two processors has the lowest ID (thus, the priority) and "shut down" the other. Following the recursion backwards, when a representative is "shut down", so are all of the members of its group. On the other hand, if a representative is prioritized, at each backward step we compare its ID to the single other ID of its group having survived the procedure up to that point (by working in parallel with the representative). Such a comparison requires $O(1)$ time. Fewer and fewer representatives will stay "alive" until only two processors will reach their starting group G_j and decide the final winner of the procedure, i.e. the lowest ID which tried to access M_C.

The problem described above is widely referred to as the LEFTMOST-PRISONER (find the leftmost processor participating in a specific conflict). The running time of the proposed solution clearly satisfies the recurrence relation $T(p) = T(\sqrt{p}) + O(1)$. Therefore, $T(p) = O(\log \log p)$.

Note: it is possible that we have more than one conflicts at each priority-CRCW step. Therefore, the above procedure should be executed more than once (once for each conflicting cell). However, all of these conflict resolutions can be executed in parallel, not increasing the simulation time computed above. Since each processor is interested only for its own target cell, he will participate only in one conflict resolution as follows. The reserved shared memory will be divided into m parts, one for each original memory cell, each one sufficiently large to accommodate the execution of the above described procedure. Each processor will request a shared cell M_x by writing its ID in the reserved part which corresponds to the specific M_x conflict resolution procedure; it will continue working within this specific part and at the end of the procedure it will erase its ID (allowing the correct simulation of the next step of the priority-CRCW). □

SIMULATION: *priority-CRCW on common-CRCW with more processors*

As expected, the simulation of a model can consume much less time if we use more than p processors (i.e. more than the original number of processors). In some cases, the weaker model can simulate the stronger model with only a constant factor slowdown. The next theorem [20] [6] gives proof for such a case:

Theorem 3.1.3. *A concurrent write operation of a p-processor priority-CRCW PRAM can be simulated on a $(p \log p)$-processor common-CRCW PRAM in $O(1)$ time.*

Proof. We prove that the simulating model can solve the LEFTMOST- PRISONER problem (i.e. find the leftmost processor participating in the conflict) for the p original processors in $O(1)$ time. With the use of the last note of the proof of theorem 3.1.2, all conflicts arising in the simulation can be resolved in parallel by allowing each processor to join the LEFTMOST solution procedure of its interest.

Assume that a number of priority-CRCW processors issue a request for the same cell M_C. The common-CRCW processors resolve the conflict as follows. First, let us visualize a binary tree T whose leaves correspond to the p original processors. Each node of this tree T is initially assigned "0". However, if an original processor p_i has requested M_C, then we assign "1" to the corresponding leaf l_i. Moreover, we assign "1" to each ancestor $a_{j,i}$ of each leaf l_i if l_i is assigned "1" and is located to the left subtree of $a_{j,i}$. Clearly, this coloring procedure leads to the safe conclusion that l_i is the leftmost leaf marked "1" if and only if every ancestor of l_i which contains l_i to its right subtree is marked "0".

Fortunately, all the above assignments can be done in parallel. Therefore, we associate $\log p$ simulating processors with each original processor and program them to place the corresponding marks on the tree T concurrently (each node of the tree T corresponds to a reserved shared cell of the common-CRCW). Specifically, each "slave" processor will be associated with a specific node of a specific level of the tree T and a "master" original processor (i.e. a leaf). In the first phase of the conflict resolution, each slave will read its master's request and mark the corresponding node on the tree T according to the aforementioned rule in $O(1)$ time. Note here that the only possible write attempts on a shared cell regard 1's and therefore, the common-CRCW policy will permit the write operation independently of the number of processors

involved in it. In the second phase, each slave s_i associated to a node n_i containing the master p_i to its right subtree will read n_i. If n_i is marked "0" then the slave will attempt to write "OK" at a specific cell M_i which will afterwards be read by processor p_i. On the contrary, if n_i is marked "1" the slave will attempt to write "notOK". Since we use a common-CRCW, the "OK" value will be written only if all slaves agree, i.e. if the under examination master is the leftmost (prioritized) processor. Clearly, both phases require $O(1)$ time. □

The above simulation is not considered "efficient" since the number of processors has to increase by $\log p$ (also, the shared memory has to increase from m to $\Theta(pm)$). There exist a more efficient simulation technique [20] which requires only p processors and $O(\log p / \log\log p)$ time. Note that this improved technique –which leads to the result reported in table 3.1 [6]– is characterized by a smaller product *processors* $\times$ *time* compared to the technique described in theorem 3.1.3.

SIMULATION: *priority-CRCW on random-CRCW with limited memory*

In general, when we limit the shared memory of the simulating model, the simulation requires significantly more time. The next theorem [6] studies such a case where the simulation time increases exponentially –when compared to the unrestricted simulation.

Theorem 3.1.4. *The simulation of a concurrent write operation of a p-processor priority-CRCW PRAM with m shared memory cells requires* $\Omega(\log(p/m))$ *steps on a p-processor random-CRCW PRAM with m shared memory cells.*

Proof. We give a lower bound for the solution of a fundamental problem on the examined model. Assume that the p original processors are divided into m equal sized groups and that each group G_i tries to access cell M_i. The simulating model must identify the highest priority processor of each group. This problem is a restricted version of the LEFTMOST WRITER problem (a generalization of the LEFTMOST PRISONER problem where more than one conflicts occur).

Initially, there are $\Omega((p/m)^m)$ solutions: within each group, any member can be the leftmost processor of the group. We can envisage the random-CRCW solution as a step by step procedure where a number of candidate

answers is eliminated until the final solution is reached. We will show that at each step, the total number of possible answers is decreased by a factor of at most 2^{2m}. Therefore, any algorithm cannot perform less than $\Omega(\log(p/m))$ steps to determine the final solution of the problem.

We will use the idea of an "adversary" for the following worst case analysis. Each processor in each G_i must have value either 0 or i (by problem definition). Our adversary will fix the values of the processors and the contents of the shared cells at each step by choosing the processors who will succeed in their write operations to them (we examine a random model, thus any processor could be chosen in practice, unpredictably). More specifically, the goal of the adversary is to fix the values of certain processors after each step in a way that allows any *free* processor (not yet fixed) to be the leftmost writer in its group. To accomplish this, the adversary never fixes a processor –except with value 0– as long as there is a free processor of lower index in the same group. The algorithm cannot terminate unless every free processor is fixed.

Initially, the rightmost processor of each G_i is fixed to i and all other processors are left free. In the remaining steps, as the processors execute their instructions, the adversary monitors their requests and allows write operations to the shared memory as follows. It will allow, if possible, only fixed processors to complete their requests. Otherwise, it will allow –and fix– a processor by choosing only from the right half of the free processors of a group. Accordingly, the number of possible answers is decreased by at most a factor of 2. Moreover, the adversary fixes the values of the left half free processors of a group at "0" to ensure that they will not write at the remaining unfixed cells. This move decreases the number of possible answers by at most a factor of 2^m. By combining these two factors, we conclude to the alleged 2^{2m} reduction factor. □

3.2 Distributed Memory Models

The same features that make the PRAM an attractive model for the programmer, unfortunately, also make the PRAM unattractive for the fabrication engineer. Its global memory is difficult to implement in hardware. Instead, the distributed memory models are widely used in the industry (especially the fixed-connection network models). Consequently, it is rational to develop simulations techniques that allow the execution of PRAM algorithms on the

distributed memory machines. Moreover, similar to the PRAM comparisons of the previous section, we are interested in comparing the distributed memory models in terms of computational power.

3.2.1 The simulation of a shared memory

Various techniques have been developed for simulating the shared memory of the PRAM with a distributed memory model [19]. Naturally, one can think of ways to distribute the shared data among the local memories of the processors, which will communicate during the computation to exchange the required data via messages. The data distribution might involve randomized and/or deterministic hash functions. Moreover, it might involve data replication (multiple copies stored in distinct processors). In any case, the major problem encountered in such simulations is the contention of the network.

In the remaining of this subsection we describe two distinct simulations of the PRAM. First, we consider a p-node Butterfly network using the same number of processors with the PRAM. We show that the Butterfly slowdown is at most logarithmic in p. Second, we examine the case of a BSP machine with sufficiently less processors than the PRAM. The *slackness* of the BSP results in an optimal simulation with respect to the total number of operations. In both examples, we will use probabilistic analysis, and thus, the results are given in terms of expected values (which hold with high probability).

SIMULATION : *PRAM on a Butterfly Network*

Consider a p-node Butterfly Network (BN) and a p-processor PRAM, which includes a shared memory of m cells [19]. We distribute, randomly and evenly, the m cells to the p processors of the BN by using some predetermined, pseudorandom, hash function $h : [1, m] \rightarrow [1, p]$. Each node N_i of the BN will simulate the operations of a distinct processor P_i of the PRAM.

Assume that P_i will access, during step t of the PRAM execution, the shared cell $f(i, t)$. Hence, during the simulation, the N_i will send a message to the node $h(f(i, t))$. The problem reduces now to the routing of such packets with the smallest possible delay. We will use the straightforward approach of sending the packet to the first column of the BN and, afterwards, to its destination (in case of a read operation, a response packet will be sent back). Assuming an EREW PRAM, the above greedy routing algorithm results with high probability in an average of $O(\log p)$ slowdown of each PRAM step.

Moreover, by carefully distributing the queues along the nodes of each BN row, the routing algorithm requires only constant size queues at the expense of some extra constant factor slowdown [19].

The above technique works equally well for CRCW PRAM simulations, provided that we are allowed to combine (e.g., concatenate) the data destined for the same shared cell. Notice that, in any case, we use no data replication. Finally, we can speed-up the simulation by introducing more nodes to the BN [19]. For instance, if we use a $\log p$ dimensional BN, which has $\log p$ more connections than the ordinary BN (more bandwidth), then the simulation results in only $\Theta(1)$ slowdown.

SIMULATION : *PRAM on BSP*

The CRCW PRAM can be simulated optimally (up to constant factors) by the BSP model if we assume constant bandwidth g and sufficient *parallel slackness*, i.e., sufficiently large ratio of PRAM processors per BSP processor. In particular, [12] reports that if $v = p^{1+\epsilon}$ for any $\epsilon > 0$, a v-CRCW-PRAM can be simulated on a p-BSP, with $L \geq \log p$, in time $O(v/p)$, where v and p denote the number of processors of each machine. However, the constant multiplier which is hidden in the $O(v/p)$ time bound grows as $\epsilon \to 0$. Such constants can be avoided, where possible, by using simpler solutions. The following simulation provides such a solution by using randomization and sufficient slackness.

Let us begin with some notation and assumptions. We shall say that a BSP algorithm is *one-optimal in computation* if it performs the same number of operations, to a multiplicative factor of $1 + o(1)$ as $n \to \infty$, as a specified corresponding sequential or PRAM algorithm. Also, a PRAM simulation is *one-optimal in communication* when $T(n)$ read/write operations of a PRAM can be simulated in $(2gT(n)/p)(1 + o(1))$ time units on the BSP [1]. Let us assume that, during the simulation, the PRAM processors can be evenly distributed among the BSP processors ($v > p$). Moreover, since we simulate a shared memory model with a distributed memory model, we will assume that there is a –computable– hash function h that will randomize and distribute the memory requests evenly among the p memory modules of the p-BSP (the contents of a memory address m will be mapped to $h(m)$, which we

[1]The factor of $2g$ is necessary since each read operation is implemented as a message to a distant memory block followed by a return message, and time g is charged per message.

can assume random). The next theorem [14] gives the required slackness for ensuring the optimality of the PRAM simulation on the BSP.

Theorem 3.2.1. *Let w_p be any function of p such that $w_p \to \infty$ as $p \to \infty$. Then the following amounts of slack are sufficient for simulating any one step of the EREW-PRAM or CRCW-PRAM on the BSP in one-optimal expected time for communication, and optimal time in local operations (if $g = O(1)$).*

(i) $w_p \cdot p \cdot \log_2 p$ *EREW-PRAM processors*
(ii) $w_p \cdot p^2 \cdot \log_2 p$ *CRCW-PRAM processors*

Proof. Separately:

(i) $w_p \cdot p \cdot \log_2 p$ EREW-PRAM processors

After the distribution of $w_p \log p$ tasks to each processor of the BSP, these tasks are completed in one superstep. The duration of the superstep will be chosen large enough to accommodate the routing of an $(1+\epsilon)w_p \log p$ -relation, thus allowing the choice of an L as large as $(1+\epsilon)w_p \log p$, for any appropriate $\epsilon > 0$ that depends on the choice of w_p.

Under the perfect hash function assumption, all references to memory are mapped to memory modules independently of each other, and thus the probability that a memory location maps to some module (say that of processor i), is $1/p$. We then use the bound for the right tail of the binomial distribution. The probability that the number of successes X in n independent Bernoulli trials, each with probability of success P, is greater than $(1+\epsilon)nP$, for some ϵ $(0 < \epsilon < 1)$ is at most $Prob(X > (1+\epsilon)nP) \leq e^{-\epsilon^2 nP/3}$. Thus, the probability that among $w_p p \log p$ requests to memory, we have more than $(1+\epsilon)w_p \log p$ requests to a specific memory module is at most $p^{-{w_p}^{1/3}}$, for say, $\epsilon = \sqrt{3}{w_p}^{-1/3}$. Since there are p such modules, the probability that more than $(1+\epsilon)w_p \log p$ requests are directed to any of these, is at most $p^{1-{w_p}^{1/3}} = p^{-\Omega({w_p}^{1/3})}$. It follows that reads or writes can be implemented as desired, in two or one supersteps respectively.

(ii) $w_p \cdot p^2 \cdot \log_2 p$ CRCW-PRAM processors

Each processor of the BSP will simulate $w_p p \log p$ processors of the CRCW-PRAM. Assume that each simulated processor requests one datum from some other simulated processor. We will show how this communication can be realized with the BSP in two phases, each of $(1+o(1))w_p p \log p$ steps. Write requests (with various conventions for concurrent writes) can be simulated in a similar fashion, in just one phase.

Suppose we number the BSP processors $0, 1, \ldots, p-1$. The simulation consists of the following four parts.
A) Each BSP processor will sort the up to $w_p p \log p$ read requests it has according to the BSP processors numbers to which they are destined. This can be done in time linear in $w_p p \log p$ using bucket sort.
B) Each BSP processor hashes the actual addresses in the destination module of each request into a hash table to identify all such address (i.e. not module) collisions. This takes linear expected time. For each destination address one request is considered "unmarked" and the rest are "marked".
C) A round-robin algorithm is used to implement all the "unmarked" requests. In the k-th of the $p-1$ stages processor j will transmit all its messages destined for memory module $(j+k) \bmod p$, and receive back the values read.
D) The "marked" requests –which did not participate in (C)– are now given the values acquired by their "unmarked" representative in phase (C). This takes time linear in the number of requests per processor.

We can implement each of the $p-1$ stages of (C) in two supersteps, where $L = g(1+\epsilon)w_p \log p,\ \epsilon = w_p^{-1/3}$. This is because the probability that the up to $w_p p \log p$ read requests from a fixed processor will have more than $(1+\epsilon)w_p \log p$ addressed to any fixed module is then at most $exp(-\Omega(w_p^{1/3} \log p))$ by a similar estimation to part (i). Since there are p^2 choices of the fixed processor and module, a bound of $p^2 exp(-\Omega(w_p^{1/3} \log p)) = exp(-\Omega(w_p^{1/3} \log p))$ follows. Finally we note that instead of doing the round robin in $2(p-1)$ supersteps we could do it in just two and allow L as large as $L = g(1+\epsilon)w_p p \log p$. □

3.2.2 Simulations between BSP and LogP

This subsection compares the computational power of the two most commonly used distributed memory models. BSP and LogP are similar in construction and can be viewed as closely related variants within the bandwidth-latency framework for modeling parallel computation. Consequently, it turns out that they can compute the same functions and that their power is almost equal in theory (BSP is slightly more powerful). Moreover, it is shown that both models can be implemented with similar performance on most point-to-point networks [13].

Before describing the cross-simulations of the two models, we give some technical clarifications. First, the presented schemes assume *well-behaved* LogP programs. That is, (i) every LogP processor sends/receives at most one message every g_{logp} time steps, (ii) every message is received at most L time units after departure, and (iii) capacity constraints are fully complied with. In other words, no processor stalls. Second, we set the maximum number of messages that are simultaneously in transit, to be equal to $L/2g_{logp}$ (instead of using the original L/g_{logp}). The factor $1/2$ is used to avoid unrealistic situations in the underlying machine [13]. Third, to better facilitate the comparison between BSP and LogP, we omit the overhead parameter o of the LogP (in this way, we have only three parameters[2] to take into consideration for both models). This is a convenient approximation technique (setting $g_{logp} = o$) leading to a discrepancy of at most 2 [18] (practically, the contribution of o to the overall cost of an algorithm is embodied in the value of the parameter g_{logp}).

SIMULATION : *LogP on BSP*

The BSP can efficiently simulate the LogP with only a constant slowdown when the router parameters of the two models have similar values [13].

Theorem 3.2.2. *LogP can be simulated by BSP with slowdown* $O(1 + g_{bsp}/g_{logp} + s/L)$. *When* $s = \Theta(L)$ *and* $g_{bsp} = \Theta(g_{logp})$, *the slowdown becomes constant.*

Proof. The i-th BSP processor mimics the activities of the i-th LogP processor and uses its own local memory to represent the contents of the local memory of its LogP counterpart. Each superstep of the BSP simulates the effects of $L/2$ consecutive LogP steps. Such a superstep consists of two periods. In the first period, each BSP processor interleaves the reception of the messages sent to it during the preceding superstep with the local computation and message generation prescribed by the LogP program for the current superstep. Specifically, the processor receives incoming messages every g_{logp} time units, until its input pool is exhausted, and executes local computation between consecutive receptions. In the second period, all messages generated

[2]note that both models use p to denote the number of their processors, they use g to somehow describe the communication bandwidth and they also use the parameters s and L, respectively, which are related to the latency of the network (loosely and indirectly for s, which can only serve as a time bound for the barrier synchronization in BSP).

in the first period are delivered to the intended destinations. Notice that since the LogP program is well-behaved, in $L/2$ consecutive steps, no more than $L/2g_{logp}$ messages are generated by any processor, and no processor can be the destination for more than $L/2g_{logp}$ such messages. This implies that in the first period of a simulation cycle each BSP processor has at most $L/2g_{logp}$ incoming messages to read, and that the second period involves the routing of an h-relation, where $h < L/2g_{logp}$. Hence, the overall running time of a cycle is at most $L/2 + g_{bsp} \cdot L/2g_{logp} + s$. Considering that a superstep corresponds to a segment of the LogP computation of duration $L/2$, the slowdown stated in this theorem is established by simple division. □

SIMULATION : *BSP on LogP*

The LogP is slightly less powerful than the BSP: in theory, we get a logarithmic slowdown when we run a BSP algorithm on the LogP (again, the comparison is made when the router parameters of the two models have similar values). However, there is a notable range of values for which the slowdown is constant, i.e., the LogP is equally powerful to the BSP. Specifically, we know the following [13]:

Theorem 3.2.3. *Any BSP superstep involving at most m local operations per processor and the routing of an h-relation can be simulated with LogP in time $O\left(m + (g_{logp}h + L) \cdot S(L, g_{logp}, p, h)\right)$, where $S(L, g_{logp}, p, h)$ is at most $O(\log p)$. When $g_{logp} = \Theta(g_{bsp})$ and $L = \Theta(s)$, then $S(L, g_{logp}, p, h)$ is the slowdown of the simulation. Moreover, if $h = \Omega(p^{\epsilon} + L \log p)$ then $S(L, g_{logp}, p, h) = O(1)$.*

Proof. Each LogP processor simulates the local computation and message transmissions of the corresponding BSP processor. The execution advances in stages, with each one of them corresponding to a superstep of the BSP. The two important points that have to be examined here are the simulation of the barrier synchronization and the routing of the h-relations. These are the only true requirements for the correct execution of the BSP algorithm –given of course that the LogP processors will execute faithfully the local operations of the BSP processors.

We will make use of the Combine-and-Broadcast (CB) and Prefix algorithms of LogP. Assume that we are given an associative operator, *op*, and a number of input values $(x_0, x_1, \ldots, x_{p-1})$, with each one initially held by a

distinct processor. CB returns $op(x_0, x_1, \ldots, x_{p-1})$ to all p processors, while Prefix returns $op(x_0, x_1, \ldots, x_i)$ to the i-th processor, $0 \leq i \leq p-1$. The use of the Prefix operation is instrumental to the routing of the BSP h-relation, while the CB is used for barrier synchronization purposes.

Barrier Synchronization

A simple algorithm for CB is obtained by viewing the p processors of LogP as the nodes of a tree of degree $\lceil L/(2g_{_{logp}}) \rceil$. At the beginning of the algorithm, a leaf processor just sends its local input to its parent. An internal node waits until it receives a value from each of its children, then combines their values and forwards the result to its parent. Eventually, the root computes the final result and starts a descending broadcast phase[3]. Note that the algorithm complies with the LogP capacity constraint, since no more than $\lceil L/(2g_{_{logp}}) \rceil$ messages can be in transit to the same processor at any time.

Upon completion of its own activity for the stage, a processor enters a boolean "1" as input to a CB computation with boolean AND as the associative operator *op*. A stage terminates when CB returns "1" to all processors. More specifically, the barrier can be implemented by interleaving the routing algorithm with executions of CB which count the number of messages sent and received by all processors (the barrier establishes that all the messages of the h-relation have reached their destination). Given the depth of the CB tree, the running time of this algorithm is $T_{sync} = O\left(L \frac{\log p}{\log 2\lceil L/(2g_{_{logp}}) \rceil}\right)$.

Routing of h-relations

The difficulty here is to avoid violating the capacity constraint of the LogP, which is not imposed by the BSP. We will devise a mechanism to decompose the h-relation into sub-relations of degree at most $\lceil L/(2g_{_{logp}}) \rceil$. Before continuing, we note that we can find the maximum r over all processors in time T_{CB}, where r_i denotes the number of messages sent by processor p_i during the current superstep: the processors will execute the CB described previously with $op = \max$ and $x_i = r_i$ (assuming p_i knows r_i).

During the simulation on the LogP, the routing of an h-relation consists of the following two phases:

[3]Prefix admits a very similar implementation with CB, the only difference being that an internal processor has to store the values received during the ascending of the messages. These will be used by the processor in the descending period, in combination with the value received from its parent, to determine the values to be sent to its children. Prefix and CB have the same running time.

(A) Sorting. Each processor creates a number of dummy messages (with nominal destination p) to make the number of messages held by each processor equal to r. Then, the $p \cdot r$ messages are sorted in order of increasing destination, so that at the end each processor holds h consecutive messages in the sorted sequence. Such a procedure requires $T_{sort} = O((g_{logp} r + L) \log p)$ time when executed in LogP by using, for instance, the AKS network [13]. This time bound can be reduced to $O((g_{logp} r + L)$ for large values of h (e.g. when $r = p^{\epsilon}$) by using, for instance, Cubesort [13].
(B) Message Delivery. After the sorting phase, messages destined to the same processor are adjacent in the sorted sequence and, therefore, form a subsequence held by processors indexed by consecutive numbers. At this point, we will make use of the aforementioned Prefix computation[4] to assign consecutive ranks to messages destined to the same processor. Then, each processor partitions its messages into two sets. The first set contains all messages belonging to subsequences starting within the processor (i.e., the processor stores the message in the subsequence with rank one). The second set contains all remaining messages. Note that, in each processor, the second set contains only messages for a single destination. These messages are part of long subsequences spanning more than one processor. The actual routing of the messages is done in two stages (dummy messages are discarded). In the first stage, each processor sends the messages belonging to the first set in an arbitrary order, one message every g_{logp} steps. The capacity constraint is never violated since no two processors send messages for the same destination. In the second stage, the processors send the messages belonging to the second set according to their rank. If a processor has one message of rank i, it sends such message at time $g_{logp} \cdot i$. Both stages require time $T_{delivery} = O\left(g_{logp} h + \left(L \frac{\log p}{\log \lceil L/(2g_{logp}) \rceil}\right)\right)$.
To sum up [13], the overall time T to simulate a BSP superstep in LogP is $T = O(m + T_{sync} + T_{delivery} + T_{sort})$. Replacing the previously computed running times leads to the analytic form given in the hypothesis of the theorem. Moreover, we get an explicit expression for $S(L, g_{logp}, p, h)$, which in all cases is $O(\log p)$, and for sufficiently large h is constant. □

[4] to be precise, we use the "segmented prefix" [19]

3.3 Boolean Circuits Simulations

The model simulation survey takes us, inevitably, to the Boolean circuits. In fact, the circuits will be at the basis of our study for the remaining of this thesis. At first sight, the Boolean Circuit model might seem quite different in nature from the rest of the models mentioned so far. The PRAM (as well as, the BSP, the LogP, etc) consists of processors and memory cells, i.e., it uses structural units of much greater complexity than the simple AND/OR/NOT gates. Moreover, a PRAM machine is a single, finite, object (a program) solving some specific problem. That is, when given a PRAM algorithm, we can modify it offline (e.g., by paper and pencil) to generate an equivalent program for any another model. Every simulation presented so far fulfills this purpose. Instead, an "algorithm" designed for the circuit model is not a single object. Rather, it is an infinite family of objects-circuits (recall that a processor can operate on inputs of variable length, while a circuit C_n can process only fixed length inputs). In general, it is impossible to modify offline an infinite number of circuits in order to develop a program for another machine. So, how can we compare a model with such distinct characteristics to a processor based model like the PRAM?

Fortunately, we can deal with both of the above peculiarities of the circuit model. Let us explain intuitively before studying the corresponding simulations. First, consider that the processors and the memories that we use in our everyday life are all made of ordinary circuits. Hence, it should be clear that no processor possesses greater power than some predefined number of –carefully interconnected– boolean gates. Second, and most important, recall the uniformity constraint that we can impose on each circuit family. A uniform circuit is defined by a single, finite, object: the Turing Machine describing its structure (a constructor). Consequently, any simulation involving circuits turns, in practice, to the modification of the circuit constructor, or to the incorporation of the constructor in a PRAM program, or to the design of a constructor "following" the instructions of a PRAM program. Of course, a comparison among models (or algorithm complexities) requires further analysis regarding the characteristics of the constructed circuit (depth, size, etc). Notice that the PRAM was used in the above discussion as an example and that the circuit simulations can involve any other model of computation. Indeed, we begin with a comparison to the most common model, the Deterministic Turing Machine.

3.3.1 Equivalence to the Turing Machine

We will establish here a relation of the boolean circuit model to the deterministic Turing machine (DTM). More specifically, we will show that by imposing certain constraints on both models we get two equal models in terms of computational power. That is, the set of functions computed by DTMs in polynomial time is equal to the set of functions computed by *logspace* uniform circuits of polynomial size. The equivalence follows the two theorems [5] [23]:

Theorem 3.3.1. *Every total function $f : B^* \to B^*$ computed by a polynomial-time DTM can be computed by a logspace uniform circuit family C of polynomial size.*

Proof. Let M^f be the DTM which computes f. It suffices to show that there exists a logarithmic-space DTM M^c, which on input 1^n will output the circuit C_n, which on input x will output the value $M^f(x)$, $|x| = n$.

We construct M^c as follows. The description of M^f is incorporated in M^c, together with the running time of M^f (a priori known). Upon input 1^n, M^c will construct C_n based on the computation table [5] of M^f. On this table, the value of each cell $T_{r,j}$ depends only on the previous cells $T_{r-1,j-1}$, $T_{r,j-1}$, $T_{r+1,j-1}$ (including the state). Since all of these values can be represented with a constant length identifier, M^c can construct constant size circuits to compute the value of $T_{r,j}$ (by constant we mean independent of the input x). The construction requires only the description of M^f. These small circuits will be connected in a cascade fashion to cover the entire computation table of M^f (we copy them cell after cell, row after row). The depth of the entire, cascaded, circuit is that of the running time of M^f, and it depends only on the length of the input 1^n (can be calculated).

The cascade described above is actually the requested circuit C_n. The input of the cascade will be the input $|x|$ of M^f and the output will be exactly $M^f(x)$, because the circuits will perform level after level the same operations as M^f. The DTM space cost for constructing $\bar{C_n}$ is logarithmic

[5] a table representing the entire computation of the DTM [5]. Each row r corresponds to the configuration of the DTM at step r of the computation. Each cell of the row depicts the contents of a corresponding cell of the DTM tape. Moreover, each row r includes a cell depicting the state of the DTM at step r; the position of this cell in the row corresponds to the position of the DTM head on the tape. Notice that we use $\sqcup$ padding to fix the length of the configurations (so that we can combine them to form the table of the computation).

in 1^n; the required variables are used for positioning within the computation table which, by definition, is of polynomial size in n. □

Theorem 3.3.2. *Every total function $f : B^* \to B^*$ computed by a logspace uniform circuit family C of polynomial size can be computed by a polynomial-time DTM.*

Proof. Let M^c be the *logspace* DTM that computes the circuit family C. We design the DTM M^f to compute the function f in polynomial-time .

The description of M^c is incorporated in M^f. On input x, M^f computes a unary representation $1^{|x|}$, which uses to simulate $M^c(1^{|x|})$ and to obtain a description of the circuit $C_{|x|}$. It then simulates $C_{|x|}$ on input x to compute its output (it simply computes the value of each gate by looking up the description of $C_{|x|}$).

The unary representation is computed in time polynomial in $|x|$. Also, since the size of the circuit is polynomial in $|x|$ the evaluation of $C_{|x|}$ is polynomial in the length of the input x. □

Theorems 3.3.1 and 3.3.2 prove an equivalence which, in the way stated above, affects only those problems within the complexity class P. However, we know that the result can be extended as follows: DTM time is polynomially related to uniform circuit size [24]. By taking a closer look at the proofs of both theorems, we see that relaxing the polynomial size and the *logspace* uniformity constraints on our circuits, we can derive

$$DTIME(T) \subseteq UniformSize(T \log T) \subseteq DTIME\left(T \log^3 T\right)$$

An important implication here is that every language in P has a polynomial size circuit. A similar result establishes that TM space is polynomially related to uniform circuit depth [24], i.e.

$$NSPACE(S) \subseteq UniformDepth\left(S^2\right) \subseteq DSPACE\left(S^2\right)$$

In the general case, we cannot assert that the two models –circuits and TM– are equally powerful[6]. For instance, completely removing the uniformity constraint allows even some polynomial-size circuits to decide non-recursive

[6]any language $L \subseteq \{0,1\}^*$ can be decided by a (non-uniform) family of circuits of size $O(n2^n)$: it suffices to implement the characteristic boolean function of L (e.g. via its disjunctive normal form). This implies that the set of circuit families is uncountable infinite (compare this to the set of TM, which is countable infinite).

languages [38]. In this direction, note that although (non-uniform) polynomial circuits have such exquisite power, they cannot decide all recursive languages. In fact, we know that most languages do not have polynomial circuits (by a counting argument) and moreover, we can show that there exists some language decided by an *exponential-space* DTM, which has no polynomial size circuit [5]. Overall, it is conjectured that NP-complete problems do not have polynomial circuits, uniform or not [5]. More on circuit complexity will be covered in the next chapter.

3.3.2 Equivalence to the PRAM

This subsection compares the computational power of the *logspace* uniform circuits to that of the CREW PRAM. We begin by giving a lemma, which highlights the way we simulate a specific bounded fan-in circuit (not *family*) with a CREW PRAM [20].

Lemma 3.3.3. *A bounded fan-in circuit of size S_n and depth D_n, where n denotes the size of its input, can be simulated by a CREW PRAM with S_n processors and S_n common memory cells in $O(D_n)$ time.*

Proof. Let us assume that any logic gate has at most k inputs (bounded fan-in). We will associate each of the S_n PRAM processors to a distinct logic gate of the circuit ("1-1" correspondence). We will do the same for the S_n memory cells of the PRAM (again, "1-1" correspondence with the gates). That is, we will enumerate the gates of the circuit –starting from the first level of the DAG– so that processor P_i will simulate the behavior of gate g_i and will write the output of g_i to the PRAM cell M_i.

Specifically, since the description of the circuit is known to us, we can program processor P_i to read successively the contents M_j from all cells which correspond to the gates g_j forwarding data to g_i. P_i will then execute the operation described by g_i (e.g. AND) and will write the outcome to the memory location M_i. Note that P_i is the only processor that will write to M_i (by definition of boolean circuits) and thus, the exclusive-write property of our PRAM suffices. Also, P_i can read from any common cell without conflicts due to the common-read property of our PRAM.

A processor can be invoked either by using a *FORK* operation (executed from one of his predecessors in the DAG) or by using a private counter (with a value depending the time required by its predecessors to finish). Notice that the *FORK* approach increases slightly the execution time, because the

first group of processors (which will read the input of the PRAM) will have to be invoked explicitly (e.g. in $\log n$ steps). The last group of processors will write their results at the output of the PRAM.

The time required for any processor to run is $O(k) = O(1)$. Since the depth of the circuit is D_n, at most D_n processors will be invoked in a serial fashion, resulting in $O(D_n)$ execution time. The number of processors and the number of the common memory cells is clearly S_n. □

As already explained, we are interested in more generic results, which involve entire families of circuits. We showed the equivalence of the *logspace* uniform circuits to a well studied, sequential, model of computation (the polynomial-time DTM). We will prove here their equivalence to a parallel model consuming subpolynomial time, which captures efficient parallel computation. As it turns out, in order to prove their equivalence to such fast machines, we have to impose upper bounds on the depth of the circuits. Specifically, the following two theorems show that the *logspace* uniform circuits of polylogarithmic depth are equivalent to the CREW PRAM with a polynomial number of processors running in polylogarithmic time [1] [5].

Theorem 3.3.4. *Every total function $f : B^* \to B^*$ computed by a logspace uniform family C of circuits with polylogarithmic depth and polynomial size, can be computed by a CREW PRAM in polylogarithmic parallel time with a polynomial number of processors.*

Proof. The simulation consists of two phases. During the first phase, the PRAM uses n (the size of the input) to construct the description of the circuit C_n. During the second phase, the PRAM simulates C_n on input x.
Phase 1
Assume that M^c is the *logspace* DTM that describes the members of the family C. The description of M^c can be incorporated in our PRAM. Since n is given as input to the PRAM (by definition), it suffices to simulate $M^c(n)$ in time $O(\log n)$.

Recall that M^c uses only $O(\log n)$ space. Therefore, we have only $O(n^k)$ possible configurations of M^c, for some arbitrary k. We can assign each processor of the PRAM to each distinct configuration of the M^c. Upon input, these processors will cooperate to construct the sequence of configurations, i.e. the computation path of M_c. They can do so efficiently, by using the parallel prefix algorithm of Ladner and Fischer [1]. The description of C_n can be extracted from this sequence by locating every configuration (i.e.,

every processor) that writes a bit to the output tape of M_c (virtually), and then, by doing parallel list contraction to produce a single string composed of these bits. All of the above operations can be performed in time $O(\log n)$. Moreover, the number of utilized processors is $O(n^k)$.
Phase 2
With the description of C_n available, we use Lemma 3.3.3 to simulate $C_n(x)$ on the PRAM. According to the lemma, since the depth of the circuit is polylogarithmic, the time required is also polylogarithmic. Also, since the circuit is *logspace* uniform, the size of the circuit is at most polynomial and thus, the number of the processors is at most polynomial. □

Theorem 3.3.5. *Every total function $f : B^* \to B^*$ computed by a CREW PRAM in polylogarithmic parallel time with a polynomial number of processors, can be computed by a logspace uniform family C of circuits with polylogarithmic depth and polynomial size.*

Proof. Let M^{pram} be the PRAM which computes $f(x)$ on some arbitrary input x. Also, let $p(n)$ be the number of processors utilized by M^{pram} and $t(n)$ the time of the M^{pram} computation, $n = |x|$. We will use a similar technique to that used in the proof of theorem 3.3.1 (where a circuit simulates a DTM). That is, we will combine circuits designed to simulate distinct operations of a PRAM, in order to simulate one potential step of M^{pram}. We will then connect these small circuits in a cascade fashion, layer after layer, so that the output of one layer will become the input of the next layer. With each layer of this cascade corresponding to a single step of M^{pram}, the entire circuit will have $t(n)$ layers to simulate step after step the M^{pram} computation of $f(x)$.

Similar to the *configuration* of the DTM, the *configuration* of the PRAM is a tuple which contains every information required to describe a single step of the PRAM computation: the program counters of all processors, the contents of their local memories (registers), the shared memory cells [5]. As in theorem 3.3.1, we must fix the length of such a configuration at some worst case value depending on n. Again, this is done because the simulating circuit will have an input of fixed size. Notice that the binary numbers (variables) used during the M^{pram} computation cannot exceed the length $l(n) = n + t(n) + const$, $const > 0$: at each step, M^{pram} can only increase the length of its variables by using add/subtract instructions (such an instruction –and any other possibly included in the machine– can only increase the operand length by 1). Also, we know that at most $t(n)$ local cells will be used per processor, and at most $t(n) \cdot p(n)$ shared cells will be used by

all $p(n)$ processors [7]. The M^{pram} program is finite (a total of $|\Pi|$ instructions) and thus, we only need a constant number of bits for each program counter. The above statements bound the size of every element of the PRAM configuration tuple. Consequently, the configuration can be encoded using $Cb(n) = O\left(2 \cdot t(n)p(n) \cdot (n + t(n)) + p(n) \log |\Pi|\right)$ bits, which in any case is polynomial in n. Since $p(n)$ and $t(n)$ can be efficiently computed by any DTM [8], the configuration tuple can be fixed at size $Cb(n)$ for further processing by the DTM (it will be padded wherever necessary). Note here that, any element within the configuration can be found simply by using counters of logarithmic size.

At this point, we must construct circuits to compute one M^{pram} configuration from another, i.e. we must construct one layer of our aforementioned cascaded circuit. This construction is somewhat more complex than the circuits developed in the proof of theorem 3.3.1: first, because the operations of a RAM are more complex than those of a DTM, and second, because the value of one element of the configuration does not depend only on three elements of the previous configuration. However, we can estimate the size and depth of the layer as follows. Recall that we have already bounded the PRAM variables to some polynomial in n (both their size and their number). The under construction layer will be composed of 'small' circuits simulating distinct operations of a PRAM (ADD, SUB, etc). Such circuits can be designed with polynomial size and logarithmic depth [23] [1]. Specifically, for each one of our polynomially many M^{pram} processors, we will use a constant number of 'small' circuits able to simulate any potential step of the processor. Such a 'small' circuit will depend, at most, on a polynomial number of local memory cells –configuration elements– and thus, we will not exceed our polynomial size limit. This holds even for those circuits simulating the operations on the shared memory cells, which themselves depend on a polynomial number of processors. All these 'small' circuits (polynomially many) will be copied side-by-side to form a circuit layer. The exact output of this layer will be controlled (e.g. with multiplexors) by the program counter –configuration element– and its corresponding instruction of the PRAM (known correspon-

[7] for sake of simplicity we will assume that the available address space of M^{pram} is $t(n) \cdot p(n)$. However, this is not a prerequisite of this proof (there is a way around this problem by using feasible highly parallel algorithms during the simulation to dynamically allocate memory cells from a pool –each cell tagged with its own address) [1]

[8] to be precise, this is a condition of the theorem, i.e. $p(n)$ and $t(n)$ must be computed in logarithmic space upon input 1^n [5]

dence, independent of the input size). Overall, the layer will have polynomial size and logarithmic depth.

Finally, we return to our original goal: given M^{pram}, we must construct a *logspace* DTM M^c which on input 1^n will output the circuit C_n, which on input x will output the value $M^{pram}(x)$, $|x| = n$. Since M^{pram} is given, we can incorporate its description (the program Π) in M^c. Moreover, we have shown above how M^c can bound the configurations of M^{pram} and construct a circuit layer. On input 1^n, M^c will start with the initial layer and by copying it in a cascade fashion, it will build the –description of– the entire circuit C_n. The total number of layers is equal to the $t(n)$ steps of M^{pram} (fixed at the worst case value). As a result, the size of C_n will be polynomial in size and polylogarithmic in depth (considering all of the above analysis). Note here that, M^c can complete its computation in logarithmic space, because the required variables are used only for counting and indexing through the polynomial sized configurations and the constant sized description of M^{pram}. □

Theorems 3.3.4 and 3.3.5 imply that the two models –*logspace* uniform circuits and PRAM– are equally powerful under the constraint of polynomial work in polylogarithmic time. Moreover, their simulations reveal a close relation of the circuit depth to the PRAM time. Especially in the first simulation we see that depth is related to time by some constant factor (the converse is not exactly true, as the structural unit of a circuit is less powerful than that of a PRAM).

The current section has shown equivalences between certain model variations. Such equivalences allow us to study parallel computation with a specific model and, afterwards, to draw generic conclusions. Furthermore, the current section introduced the use of resource bounds and constraints on our machines/circuits. Such restrictions facilitate the detailed study of a model by creating model variations with respect to its computational ability. This is a common technique used, among others, in complexity theory, which is presented in the following chapter.

Chapter 4

Parallel Complexity Theory

Parallel complexity theory is the study of resource-bounded parallel computation. It is the branch of computational complexity theory, which focuses on the classification of problems according to their parallelization difficulty.

The issues raised here concern primarily the speedup gained from our parallel machines. Common questions driving the research towards this direction are the following: –Are all problems amenable to fast solutions? –What is the minimum amount of resources required for solving an important problem in parallel? –Is it possible to achieve a dramatic speedup of a sequential solution while maintaining reasonable hardware cost? –Is there a way of transforming algorithmically a sequential solution to a highly parallel one?

Such questions offer an insight to the nature of parallel computation. The answers lead to certain problem classification with respect to the difficulty and/or the impossibility of designing efficient parallel solutions. Moreover, they assist the researchers in redefining 'good' models for parallel computation and identifying the resources of interest. It is worth mentioning that, analogously to the classical "P versus NP" conundrum, some of these questions remain open till today. The theory itself establishes relations between parallel and sequential classes, contributing in this way to the evolution of the entire computational complexity theory.

This chapter is a survey of the most important results presented over the past four decades regarding parallel complexity. It presents common complexity classes, inclusion proofs, classification of problems and equivalences of models (or classes). Further, it discusses some of the inherent limits of parallelization together with the conjectures of the research community. Hereafter, our study bases mainly on the boolean circuits and the Turing

machines; due to certain model equivalences, the circuits and the TM can cover most of the study in the field of parallel complexity (we also report the PRAM, or any other model, wherever it is related to our results).

4.1 Complexity Classes

One of the first issues raised in classical complexity theory is the distinction between feasible and infeasible solutions. It is important to identify those problems which have a practical solution (one that can be obtained within the available resources, even when the size of the input increases significantly) and those which haven't. The community adopted the polynomial time bound to mark the territory of the feasible computation. That is, a solution (algorithm) is considered *feasible* (efficient) if it requires at most polynomial time in the length of the input. A problem is called tractable if it has such a solution. The well-known class P was defined accordingly, to capture the notion of feasible sequential computation.

Similar to the classical approach, the parallel complexity theory also makes its first distinction between feasible and infeasible solutions. In the context of parallel computation though, besides time cost, the feasibility of a solution depends also on its hardware cost (the required number of processors, gates, etc). It is natural to readopt here the polynomial bound for characterizing feasible parallel computation. That is, a parallel solution is considered feasible if its implementation requires at most polynomially many hardware resources, which operate for at most a polynomial amount of time. To defend the above choice intuitively, one could point out the following real-world analogy. Exponential (superpolynomial) functions grow fast enough to numbers exceeding both the lifetime of our universe and the particles that constitute it. In other words, it is impractical not only to wait for the termination, but even to start manufacturing an exponential-cost parallel machine in our universe. Note that the class of feasible parallel problems is equivalent to the class P. Imposing restrictions on the parallel time and on the hardware resources has led to the definition of certain parallel complexity classes, which are studied below.

When it comes to parallel time, the aforementioned polynomial bound seems rather modest for a theoretical study of feasible problems (their sequential time is already polynomial). In fact, the most interesting applications of parallelization are those resulting in dramatic time savings: as shown

for certain problems, the sequential and the parallel solution are separated by an exponential time gap (e.g. polynomial to logarithmic). In general, we consider as *highly parallel* those algorithms, which require at most polylogarithmic parallel time in the size of their input. Significant effort has been spent in identifying those members of P, which can be solved highly parallel by consuming only a polynomial amount of hardware resources. We refer to such problems as *feasible highly parallel* [1] and we have defined several classes to capture and study them. The class definitions base on certain parallel time or circuit depth bounds (e.g. specific logarithms), on hardware resource bounds (e.g. polynomial amount, uniformity, gate fan-in), on alternating TM space or time, on the number of alternations, etc. Probably the most notable of the parallel classes is the NC^2, which is characterized by its $O(\log^2 n)$ circuit depth bound. As it turns out, NC^2 contains a lot of the feasible highly parallel problems/functions, especially those used by the engineers in a daily basis. Consequently, NC^2 is considered as a perfectly good candidate for capturing the –more conservative– notion of *efficient parallel computation* [5].

In the following subsections we study several parallel complexity classes; we give definitions, inclusion proofs, implications and we show some of the known relations between parallel and sequential computation. Moreover, throughout this section, certain parallel algorithms are described, common simulation techniques are employed and a representative set of parallel problems is classified. Subsection 4.1.1 is concerned with feasibility, while subsection 4.1.2 examines classes of greater complexity.

4.1.1 Feasible Highly Parallel Computation

We begin by defining the complexity classes, which, by convention, contain the feasible highly parallel problems. In general, the complexity of a problem is reflected on the amount of the resources consumed by its solution (time, space, etc). Therefore, to define a class of problems, we define an available amount of resources over some predefined model of computation: a problem belongs to the class if it can be solved by the model within the available resources (we always examine the worst case consumption).

The following class definitions base on a specific model of parallel computation, i.e., the boolean circuits. Note that, provably, the same classes can be defined by using other models, e.g., the PRAM, the ATM, etc (such class equivalences are presented later in this subsection). The definitions

place bounds on two kinds of resources, simultaneously. First, to meet the feasibility requirement, we bound polynomially the hardware resources. We also impose a *logspace* uniformity restriction on the circuit families (consider any of the two definitions of section 2.3.1). Second, to meet the requirement for fast solutions, we bound the depth of the circuit (which captures parallel time). We do this by imposing certain logarithmic bounds. Other parameters of the model, such as the type of the gates, are handled separately in each of the definitions. Specifically [1]:

Definition 4.1.1.

NC^k *For $k \geq 1$, let NC^k be the class of languages decided by logspace uniform families of boolean circuits, $\{C_n\}$, of depth $O(\log^k n)$ and size $n^{O(1)}$. The circuits must consist only of bounded fan-in AND, OR, NOT, gates.*

NC *Let NC be the union of all NC^k, i.e.,* $NC = \bigcup_{k\geq 1} NC^k$.

AC^k *For $k \geq 0$, let AC^k be that generalization of NC^k, where the circuits consist of unbounded fan-in AND, OR, NOT, gates (logspace uniform, $O(\log^k n)$ depth, $n^{O(1)}$ size circuits).*

AC *Let AC be the union of all AC^k, i.e.,* $AC = \bigcup_{k\geq 0} AC^k$.

TC^k *Let TC^k be defined as the AC^k, except that, the circuits must consist of NOT and MAJ gates (a boolean 'majority' gate with fan-in $= r$ outputs 1 iff at least $r/2$ of its inputs are 1).*

TC *Let TC be the union of all TC^k, i.e.,* $TC = \bigcup_{k\geq 0} TC^k$.

Before continuing, we must mention that the above definition is not unique throughout the literature. Even when it bases on the same model (boolean circuits), it might differ with respect to the underlying uniformity. For example, we can avoid imposing a direct polynomial bound on the number of the circuit gates, by defining *logspace* uniformity to measure space with respect to the input 1^n of the DTM (hence, we bound the output of the DTM by a polynomial) [39], [40]. Other authors [41], [25], use a slightly different kind of uniformity; they say that the circuit family, $\{C_n\}$, must be $DLOGTIME$-uniform. That is, they can use a logarithmic-time DTM with

random access to its input tape (via an "index" tape, on which the DTM writes a number j and then enters a state to receive the j-th bit of the input) for answering questions of the form "is there a path from node u to node v in C_n?", or "what type of gate is node u in C_n?". This definition presupposes that the size complexity of the family is polynomial. Other definitions use the notion of P-uniformity, i.e., the circuits must be constructed by polynomial-time DTMs. Notice that in any case, the circuit family is required to be uniform, as this is a prerequisite of feasibility. Essentially, all these definitions, although not necessarily equivalent, give the same meaning to the classes NC, AC and TC.

To explain the nomenclature [41], Nicholas Pippenger was one of the first to study polynomial-size, polylogarithmic-depth circuits in the late 1970s, and NC was dubbed "Nick′s Class"[1]. The "A" in AC connotes alternation, because of the close relation between the depth of the AC circuit and the number of alternations of the ATM (see page 96). Finally, the "T" in TC denotes the use of threshold gates. Threshold gates model the technology of neural networks. The general threshold gate inputs boolean arguments, it associates numerical weights $w_1, \ldots, w_r$ with each of the r inputs, it uses internally a threshold t, and it outputs a boolean value. Specifically, the gate will output '1' iff the inputs, multiplied by their corresponding weights, add up to t. The definition 4.1.1 uses one kind of threshold gates, the MAJ, which is the special case with $w_1 = \ldots w_r = 1$ and $t = r/2$. Note that a depth-2 circuit of MAJ gates can simulate the general threshold gate [41].

The little details in the definitions of NC, AC, and TC, make the classes seem quite different from each other. Questions that naturally come in mind here include: –Does the unbounded interconnection offer significantly greater power? –Is the MAJ gate making any difference at all? –How are these classes related to each other? –Which are the real-world problems contained in these classes? We proceed with answering such questions by studying, first, the inclusions between parallel complexity classes.

[1]Pippenger repaid this favor by giving the name "Steve's Class" to the class of languages decided by DTMs in polynomial-time and polylogarithmic-space, simultaneously [5]. Specifically, SC^k=DTM[time($n^{O(1)}$),space($O(\log^k n)$)] and $\mathrm{SC}=\bigcup_{k\geq 1}\mathrm{SC}^k$.
SC captures the efficient algorithms consuming little space, and is equal to the class of languages decided by *logspace* uniform circuits of polylogarithmic width. Intuitively, NC involves "shallow" circuits, while SC involves "narrow" circuits.

Relations between parallel complexity classes

Trivially, we have $\mathrm{NC}^k \subseteq \mathrm{NC}^{k+1}$, $\mathrm{AC}^k \subseteq \mathrm{AC}^{k+1}$, and $\mathrm{TC}^k \subseteq \mathrm{TC}^{k+1}$. Hence, these classes form potential hierarchies of infinitely many distinct levels. Notably, it is not yet proved that the above inclusions are proper. For all that we know, it may be the case that, e.g., NC^1=NC. However, this is not expected to be true and it is conjectured that the NC hierarchy is proper [5]. What we know for certain about the NC hierarchy, is that it collapses at the k-th level if $\mathrm{NC}^k = \mathrm{NC}^{k+1}$ (the same holds for the AC and TC hierarchies) [5]. Moreover, it is proved that the *monotone*-NC, i.e., the analog of NC consisting only of AND/OR gate circuits, indeed forms a proper hierarchy [42].

Then, we show how the three hierarchies interleave with each other. We start by examining the relations between NC^k and AC^k [5], [41].

Theorem 4.1.1. *For $k \geq 1$, $NC^k \subseteq AC^k \subseteq NC^{k+1}$.*

Proof. The first inclusion is trivial: every circuit with bounded fan-in gates, can be viewed as a specific instantiation of a circuit with unbounded fan-in gates.

The second inclusion is based on a minor modification of the AC^k circuit, namely the ${}^uC_n$, where $n = |x|$ is the length of the input. Recall that AC^k includes only polynomial size circuits. Therefore, any gate of ${}^uC_n$ must have a number of inputs which is at most polynomial in n. Notice that we can transform any of these gates (AND/OR) to a functionally equivalent tree of 2-input gates (AND/OR). Clearly, the size of this tree is polynomial, while its depth is logarithmic. By transforming every gate of ${}^uC_n$ in the above way, we derive a circuit with bounded fan-in gates, namely ${}^bC_n$, which has size polynomial in n. Since each gate of ${}^uC_n$ increases the depth of ${}^bC_n$ by at most $O(\log n)$, the entire depth of ${}^bC_n$ will be $O(\log^k n \cdot \log n)$. Finally, note that bC_n can be described by a *logspace* DTM; the above transformation can be executed within a modified version of the DTM describing ${}^uC_n$ (it requires only extra counters of size logarithmic to the size of ${}^uC_n$, which is polynomial). □

According to theorem 4.1.1, whichever language can be decided (or, function computed) by *unbounded* fan-in circuits, it can also be decided by *bounded* fan-in circuits, feasibly. What's more interesting, though, is that the penalty of bounding our communication is only a logarithmic slowdown.

This result contradicts one's –probable– overestimation that the unbounded fan-in provides tremendous computational power. The key point –probably– missed here, is that the gate fan-in is not totally "unbounded". Rather, it is bounded by the polynomial size of the circuit. To better understand the differences between gate types, the following theorem compares the NC^k and AC^k classes to the TC^k [41]:

Theorem 4.1.2. *For* $k \geq 0$, $\quad AC^k \subseteq TC^k \subseteq NC^{k+1}$

Proof. To show the first inclusion, we describe the simulation of an arbitrary AND/OR gate using MAJ and NOT gates. We can simulate one k-input OR gate with exactly one MAJ gate. Specifically, we use a MAJ gate with $2k-1$ inputs: the k inputs of the simulated OR gate plus $k-1$ inputs of constant value '1' (*dummy signal*). Clearly, when at least one of the k variable inputs is '1', the output of the MAJ gate is '1'. Similarly, we can simulate one k-input AND gate with one MAJ gate and $k+1$ NOT gates. Again, our MAJ gate has $2k-1$ inputs: the inverted k inputs of the simulated AND gate (inverted with NOT) plus $k-1$ inputs of constant value '1'. Also, we use one extra NOT gate at the output of the MAJ gate. Clearly, the above circuit outputs '1' if and only if all of the k variable inputs are '1'. Notice that a *dummy signal* can be easily acquired by driving any wire of the circuit to a 2-input MAJ gate, the first of which is inverted (with NOT). Both of the above simulating circuits increase the depth of the original NC circuit by at most a constant factor. Moreover, they increase the size of the original circuit by at most a polynomial factor; for each k-input gate, at most $k+4$ gates are required, where k is polynomially bounded in the size of the circuit's input (the size of the original circuit is itself polynomially bounded). Finally note that, such a circuit can be easily described by a DTM in *logspace* (the DTM incorporates the description of the original circuit family and uses only logarithmic counters), therefore $AC^k \subseteq TC^k$.

To show the second inclusion, we must simulate an arbitrary MAJ gate with AND/OR/NOT gates of 2-inputs. The idea is simple. We add the k 1-bit inputs of the MAJ gate and compare the result to $k/2$. Both operations can be done with circuits of depth $O(\log k)$ and size $O(k)$. As mentioned above, k is polynomially bounded in the size of the input. Therefore, substituting every MAJ gate as described above will increase the depth of the original circuit to $O(\log n) \cdot O(\log n) = O(\log^2 n)$, where n denotes the length of the circuit's input. The size of the new circuit will remain polynomial in n. The new circuit can be easily described by a DTM in *logspace* (the DTM

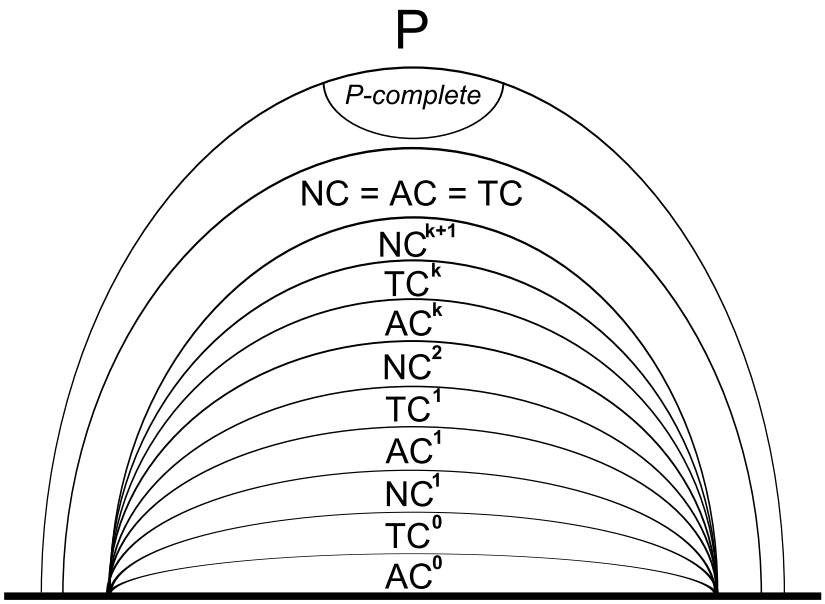


Figure 4.1: Parallel classes and hierarchies

incorporates the description of the original circuit family and uses only logarithmic counters) and thus, $TC^k \subseteq NC^{k+1}$. □

Theorem 4.1.2 proves that the use of threshold gates results in, at most, a logarithmic speedup. Once again, the ordinary bounded fan-in AND/OR gates are shown to have adequate power for studying parallel computation. Theorem 4.1.2 also shows, for all k, how the classes NC^k, AC^k, and TC^k, alternate in terms of computational power. Consequently, we now see the big picture (figure 4.1, given the widely accepted conjectures) where the parent classes NC, AC, and TC, are equal.

Corollary 4.1.3. $NC = AC = TC$

Proof. Directly from theorems 4.1.1 and 4.1.2. □

Corollary 4.1.4. $NC \subseteq P$

Proof. Directly from theorem 3.3.2. □

The above corollaries place feasible highly parallel computation within P. Indeed, we anticipated this result since the introduction of this section.

However, corollary 4.1.4 leaves an open question: does "NC equal P?", i.e., do all problems with feasible solutions also have feasible highly parallel solutions? This is, arguably, the most important open question in the realm of parallel computation. In fact, it is concerned with the inherent limitations of parallelization. We examine this topic in section 4.2.2.

In complexity theory, proving that a class inclusion is proper, is a rare and striking result. As usual, the aforementioned inclusions are not known to be proper for arbitrary values of k (mere conjectures). However, we do have proofs of proper inclusions for the first level, $k = 0$, of the classes given in theorem 4.1.2. Specifically, we know that

$$\mathrm{AC}^0 \subset \mathrm{TC}^0 \subseteq \mathrm{NC}^1.$$

We do not know whether TC^0 is strictly contained in NC^1 (we conjecture that it is). Below we describe a proof for the $\mathrm{AC}^0 \subset \mathrm{NC}^1$ inclusion. Note here that when studying the first level of the NC, AC, or TC hierarchy, our choice on the uniformity type of the circuit plays an important role. The rigorous approach is to use some "conservative" uniformity type, e.g. the $DLOGTIME$, and not the *logspace*. In the following proof we use the *logspace*, as this result holds for any of the commonly used uniformities [21]:

Theorem 4.1.5. *$AC^0 \subset NC^1$.*

Proof. To show that $AC^0 \subseteq NC^1$ we can use the same argument with that used in th proof of theorem 4.1.1. To show proper inclusion, it suffices to find a problem which belongs in NC^1 but does belong in AC^0. We point out the $PARITY$ problem: "does the parity of x equal 1 ? ". Recall the parity function $f_p(x) = (x_1 + x_2 + \ldots x_n) \bmod (2)$, where $n = |x|$.

On one hand, the authors in [43] prove that the parity function $f_p(x)$ cannot be computed by AC circuits of constant-depth and polynomial-size [2]. Therefore $PARITY \notin AC^0$.

On the other hand, it is trivial for any DTM to describe a XOR gate using AND, OR, NOT gates: recall that $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. On input 1^n, the DTM can output the description of a XOR tree of n leaves. The resulting circuit will have depth $O(\log n)$ and size $O(n)$. Such a DTM is clearly *logspace*, because it only uses indexing variables for positioning within an object of size $O(n)$. The XOR tree computes the parity of its input. Therefore, $PARITY \in NC^1$. □

[2] specifically, they show that polynomial-size parity circuits must have depth $\Omega(log^* n)$.

Relations between parallel and sequential complexity classes

Much effort has been devoted in parallel complexity theory towards relating parallel to sequential computation. Arguably, the best way to establish such a connection is by studying the inclusions between parallel and sequential classes.

Naturally, we begin by comparing to the ordinary Turing Machine. The weakest of the widely used TM classes are the logarithmic space classes L and NL. Proving that these two classes are not contained in NC would severely damage the image of the parallelization power. Fortunately, not only both L and NL fall within NC, but they are contained in the lower levels of the NC hierarchy. Let us first consider the deterministic class [5]:

Theorem 4.1.6. $NC^1 \subseteq L$

Proof. Assume that M^c is a *logspace* DTM describing a circuit family C of NC^1. We will show that there is a *logspace* DTM M, which can compute the output of $C_n(x)$ for any input x with $|x| = n$.

For starters, let us assume that a binary tree-like circuit C_n is given as input to a DTM M_a, together with an input x. M_a can compute $C_n(x)$ using space $O(\log n)$. The idea is simple. First, notice that we can mark any path within a binary tree by using a unique identifier of $\log n$ bits: all the way from the root to any leaf, '0' represents the left edge of a node and '1' the right edge. With this in mind, we can start from the root and perform an ordered walk of the entire tree by using $O(\log n)$ space as follows. We keep track of our path by using a specific variable, which we update step after step (starting with '0'). At any step, this variable corresponds to one of the unique identifiers described above. We update this tracking variable (append one bit, change last bit, remove last bit) by checking our location on the tree: we find the current node on the graph at the input tape of M_a (only logarithmic counters are required for this) and, if possible, we move left (depth first search). Second, notice that the nodes here correspond to logic AND/OR operations (and NOT). This has the following impact on the aforementioned ordered walk of the circuit. When we encounter an AND node, we will continue our walk back to the upper levels of the tree (i.e. we will change/remove the last bit of our tracking variable) only if both children of the node were previously evaluated 'true'. Practically, when we encounter a 'false' node with an AND parent, we immediately evaluate the parent 'false' and update our tracking variable. Similarly, when the parent is an OR gate

and the current node is 'true', we mark the parent as 'true'. Note that we proceed to the evaluation of the right child of a node only in certain cases: when the node is AND and we have just evaluated its left child as 'true', or when it is OR and its left child is 'false'. Consequently, in any case, we can deduce the truth of a parent node simply by knowing the truth of the current node and whether it is a left/right child (we do not need to store the values of any previous nodes). By following these rules, the tree walk will finally lead to the evaluation of the root. We can implement the above recursive algorithm by reusing the space of the tracking variable and, therefore by using only $O(\log n)$ space.

Unfortunately, not all circuits are binary tree-like circuits. Fortunately, we can transform any NC^1 circuit to an equivalent binary tree (which computes the same function) by using only logarithmic amounts of space. First, if $k > 2$, we can replace each k-input gate with a small equivalent binary tree (of depth $\log k$ and size $2k-1$). Second, starting from the output of our "binary" circuit, we name the paths to each input gate as described in the above paragraph. However, in this technique we use the names of these paths ('0', '01', etc) to mark the intermediate gates of the original circuit. Each of these names, together with the type of its corresponding gate, represents a new gate of the transformed circuit. We proceed by reusing space to output in this way every gate of the transformed circuit. Note that the new gates have out-degree '1' (because each original gate reachable by several paths will be represented many times), and thus, the resulting circuit is a binary tree. The two operations result in a circuit which, like our original circuit, is of logarithmic depth and polynomial size. Moreover, they can be performed in $O(\log n)$ space from a DTM M_b, because the only variables required are for positioning/counting within an object of polynomial size (also, the gate names are $O(\log n)$ wide because the original circuit has $O(\log n)$ depth).

We now return to the description of our *logspace* DTM M. M will use the two algorithms described above and the descritpion of M^c (incorporated). On input $x, |x| = n$, our M will simulate $M^c(1^n)$ to acquire C_n and then it will evaluate $C_n(x)$ as described above. However, there is one last obstacle before completing the proof: the outputs of the machines M^c and M_b are polynomial in size. If we store their outputs as intermediate results in a tape of M, the space of M will not remain logarithmic. We overcome this obstacle as follows [5]. When seen as functional blocks, the input of M_a is the output of M_b and the input of M_b is the output of M^c. We must solve the general problem, where two machines A and B want to exchange information

without writing down all of it at once (in a single string). Notice that, either way, machine B reads one bit at a time from the "tape" in-between A and B. Moreover, this bit is located in a specific cell beneath the head of B. The idea is that, before executing each step of B we know the location of the head on the input tape of B and, therefore, we can simulate A until it writes that specific cell. Then, we can continue with the step of B. By interleaving the simulation of the two machines in the above way, the composite computation is perfomed with only one "communication" variable (the location of the B's input head). In our case, the length of this variable is logarithmic, because the intermediate "tape" is polynomial (for both M^c to M_b and M_b to M_a "communication"). To conclude, all of the above algorithms require logarithmic space and the resulting DTM correctly evaluates the output of $C_n(x)$. □

Although not yet proved, many researchers believe that NC^1 is properly included in L [41]. For this reason, using the *logspace* uniformity to define NC^1 is controversial (probably, it allows more computing power to the "preprocessing stage" than to the NC^1 circuits themselves). To be precise, it is still unknown whether the *logspace*-uniform NC^1 is equal to the $DLOGTIME$-uniform NC^1, or, if the *logspace*-uniform AC^0 is equal to the $DLOGTIME$-uniform AC^0 (other types of uniformity are also involved in such disputes, e.g., the *logspace* and the P uniformity: is L equal to P?). However, we know that the class NC^k for $k \geq 2$ is identical under the two definitions (respectively, AC^k for $k \geq 1$) [41]. In fact, the most commonly used types of uniformities all lead to equivalent definitions of the class NC^k for $k \geq 2$ [24].

We continue by examining the converse of theorem 4.1.6. Specifically, we make use of the nondeterministic version of L to prove that [5], [23]:

Theorem 4.1.7. $NL \subseteq NC^2$

Proof. It suffices to show that a NL-complete problem belongs to NC^2. We point out the REACHABILITY problem: "given a graph G and two nodes s, t, is there a path from s to t?"

We start by showing that REACHABILITY is NL-complete. First, we show that REACHABILITY$\in$NL. Nondeterministically, we guess the requested path (if any). That is, starting from s, step after step we temporarily store the current edge (i, j) and we find the next by looking at the description of G. We terminate when we discover t or when we perform more

than n steps (discover a cycle), where n denotes the size of the input. The above search requires logarithmic space (only counters). Second, we give a *logspace* reduction from any language A∈NL to REACHABILITY. Assume that the *logspace* NDTM M^A solves A. On input x, a *logspace* DTM (which incorporates the description of M^A) outputs the entire configuration graph of $M^A(x)$. In this graph, each node corresponds to a configuration of the $M^A(x)$ computation and each edge corresponds to a probable computation step of $M^A(x)$. The DTM proceeds as follows: it writes down every possible pair of strings of length $c \log |x|$ (i.e. the space bound of $M^A(x)$) and it check each one of them separately. Specifically, it checks whether the first string is a valid configuration of $M^A(x)$ and if the pair constitutes a valid computation step of $M^A(x)$ (e.g. by simulating all possible transitions –logarithmically bounded– from the first configuration string). Notice that the DTM can write the under examination pair of strings one after the other (e.g. lexicographically) by reusing its space. In this way the DTM can output a descritpion of the entire configuration graph of $M^A(x)$ by using only logarithmic space ($2c \log |x|$, plus counters). Moreover, during this transformation, we can ensure that there is only one accepting node (e.g. by connecting all accepting nodes to a new one). Clearly, M^A accepts x if and only if there is a path in the above constructed graph from the initial node to the single accepting node (solved by REACHABILITY).

We continue by showing that REACHABILITY∈NC^2. Assume that we are given the $n \times n$ adjacency matrix V of a graph G with n nodes. In this graph, the *max-min* path has length at most $n-1$. Therefore, to compute the transitive closure G^*, it suffices to take the adjacency matrix to the power of n by using boolean matrix multiplication [23]. After the calculation of V^n, we can answer the REACHABILITY question simply by looking at the $[s,t]$ position in the resulting table. Notice that [44], instead of calculating all the powers of V up to n, we can square the $(V+I)$ matrix: we add the identity matrix I to V and successively square the result, i.e. $(V+I)^2$, $(V+I)^4$, ..., untill we obtain $(V+I)^m$ for some $m \geq n$. The resulting matrix $(V+I)^m$ suffices for answering the REACHABILITY question. Therefore, the computation of the transitive closure of V requires only $\lceil \log n \rceil$ steps. A boolean matrix multiplication can be calculated by circuits of logarithmic depth and polynomial size [23]. Specifically, since the boolean product $F = R \times T$ of the $n \times n$ matrices R, T is defined by $F[i,j] = \bigvee_{l=0}^{n-1} (R[i,l] \wedge T[l,j])$, we will compute it by using one distinct circuit per cell $[i,j]$: a tree whith n leaves as AND gates and $n-1$ internal nodes as OR gates. To compute $(V+I)^m$

we will use $\lceil \log n \rceil$ layers of such multiplication circuits (each layer performs a matrix squaring and forwards its data to the next layer). This construction results in a cascaded circuit with $O(\log^2 n)$ depth and polynomial size. Finally, we will add one more layer at the top of the cascaded circuit, which will serve as a multiplexer to output the final value of the cell $[s, t]$, where s and t are also given as inputs (recall that the content of this cell is the answer to the REACHABILITY question). Multiplexers can be constructed with circuits of logarithmic depth and polynomial size and thus, the final layer will not affect the aforementioned bounds. Clearly, the construction of the entire circuitry described above is straightforward and can be perfomed by any DTM in logarithmic space (only counters are required for positioning within objects of polynomial size). To conlcude, REACHABILITY$\in$NC^2. □

It is trivial to show that L $\subseteq$ NL. By further combining theorems 4.1.6 and 4.1.7 we get corollary 4.1.8. Note that whether L $\subset$ NL is still an open question, and thus, we cannot use this chain yet to infer that NC^1 is strictly contained in NC^2.

Corollary 4.1.8. $NC^1 \subseteq L \subseteq NL \subseteq NC^2$

The above result establishes that any logarithmic space algorithm (deterministic or not) can be efficiently parallelized[3]. The converse is not known, i.e. we cannot tell whether all NC circuits can be simulated in logarithmic space. However, we can establish an analogous result for the generalization of non-determinism, the Alternating Turing Machine (ATM). Specifically, we show that NC^k languages can be decided by ATMs which use logarithmic space and perform at most $\log^k n$ alternations [44]:

Theorem 4.1.9. *For $k \geq 1$, $NC^k \subseteq ATM[space(O(\log n)), altn(O(\log^k n))]$.*

Proof. Without loss of generality, we assume that the circuit C_n uses only 2-input AND/OR gates and also that, NOT gates can be applied only at the n inputs of C_n (every NC circuit has an equivalent circuit of this form, which can be constructed in *logspace* [44]).

We will contsruct an ATM N to simulate $C_n(x)$ on input x, $n = |x|$. N will incorporate the description of the circuit family C. Consequently,

[3] a RAM algorithm (besides the multiple or single tape TM) can also be used to measure space and check the aforementioned logarithmic space "criterion". As shown in [45], TM and RAM can simulate one another within only a constant factor of extra space.

N can find any gate g_i within C_n, together with its two predecessors, in logarithmic space (without alternating). We program N to do the following operations. N will start by locating the output gate g_o of C_n. If g_o is an AND (OR) gate, then N will jump to a special purpose state labeled 'AND' ('OR'). This state will have two outgoing transitions, one corresponding to the evaluation of the left child of g_o and one to the right. The choice will be taken by N nondeterministically. The computation will continue in this fashion, recursively, until N reaches a NOT gate; at that point, N enters the 'yes' or 'not' state depending on the value of x at the under evaluation position.

Note that the only nondeterministic moves included in N are those taken in the special purpose state (lebeled 'AND','OR'). Everything else (e.g., gate or wire searching) is performed deterministically with no alternations between 'AND' and 'OR' labels. Also note that the computation tree characterising N has one branch per gate, for every gate of the circuit C_n. In fact, we can view each branching after a node/configuration of the computation as the creation of two new "subprocesses" (each subprocess searches the predecessors of a gate without alternating, and then spawns two children). Clearly, the computation tree of N reflects the structure of C_n and thus, it will output the same result with C_n on input x.

The machine N requires $O(\log n)$ space for the above operations: each "subprocess" uses only enumeration variables to locate a specific gate g_i –and its incomming wires– during the construction of C_n (which takes place several times during the computation by reusing space). Since C is *logspace* uniform, $O(\log n)$ space suffices for the enumeration. Moreover, N will alternate at most $O(\log^k n)$ times, because the number of alternations is bounded by the length of the maximum path within C_n (by construction, no more AND/OR transitions can take place in N). □

Theorem 4.1.9 proves only one direction of a well-known class equivalence. It is also shown [44] that $\mathrm{NC}^{k+1} \supseteq \mathrm{ATM}[\mathrm{space}(O(\log n)), \mathrm{altn}(O(\log^k n))]$. Consequently, the languages decided by ATM with logarithmic space and polylogarithmic alternations are exactly those languages in NC, i.e. the feasible highly parallel languages

$$\mathrm{NC} = \mathrm{AC} = \mathrm{ATM}[\mathrm{space}(O(\log n)), \mathrm{altn}(\log^{O(1)} n)]$$

Compared to NC, the AC equivalence noted above is even more closely related to the ATM alternations. When we use unbounded fan-in gates in the

aforementioned simulation[4], the depth of the simulating circuit decreases by a logarithmic factor. Hence, it turns out that the languages in AC^k are exactly those languages decided by a *logspace* ATM, which preforms $O(\log^k n)$ alternations

$$\mathrm{AC}^k = \mathrm{ATM}[\mathrm{space}(O(\log n)), \mathrm{altn}(O(\log^k n)]$$

Such a close relation to the ATM can be established also for the NC^k circuits [24]. In this case, we have a direct correspondence between the depth of the circuit and the running time of the (random access input tape) ATM. Specifically, the languages in NC^k are exactly those languages decided by a *logspace* ATM which preforms $O(\log^k n)$ steps

$$\mathrm{NC}^k = \mathrm{ATM}[\mathrm{space}(O(\log n)), \mathrm{time}(O(\log^k n)]$$

These results are analogous to those in section 3.3, which show the equivalence of *logspace* uniform circuits to the polynomial-time DTM and to the PRAM (when polylogarithmic-time constraints are also imposed).

$$\mathrm{NC} = \mathrm{PRAM}[\mathrm{proc}(n^{O(1)}), \mathrm{time}(\log^{O(1)} n)]$$

By combining the above equivalences of NC, we deduce an interesting equivalence of the ATM and the PRAM models. Specifically, we deduce that

$$\mathrm{ATM}[\mathrm{space}(O(\log n)), \mathrm{altn}(\log^{O(1)} n)] = \mathrm{PRAM}[\mathrm{proc}(n^{O(1)}), \mathrm{time}(\log^{O(1)} n)]$$

The picture given above shows that the ATM captures parallelism in various ways and it justifies the wide use of the model throughout the literature for studying parallel complexity[5]. Moreover, we can now justify the use of other models of parallel computation for defining the NC class. For instance, it is not uncommon to define NC^k as the class of languages decided by PRAM algorithms running in $O(log^k n)$ time and utilizing a polynomial numder of processors [20], [29].

---

[4]to simulate an ATM, like in the case of a DTM, we have to stack a number of layers of circuits (one on top of the other). Such a layer computes straightforward boolean functions, which require $O(\log n)$ depth circuits when dexsigned with bounded fan-in gates or, $O(1)$ depth circuits when designed with unbounded fan-in gates.

[5]other important results comparing the power of ATM and DTM [39]:

For $f(n) \geq \log n$, $\mathrm{ASPACE}(f(n))=\mathrm{DTIME}(2^{O(f(n))})$.
For $f(n) \geq n$, $\mathrm{ATIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)) \subseteq \mathrm{ATIME}(f^2(n))$.
Consequently, AL=P, AP=PSPACE, APSPACE=EXPTIME.

Note that [28] the introduction of alternation shifts by exactly one level the deterministic hierarchy $\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXPSPACE} \subseteq \ldots$

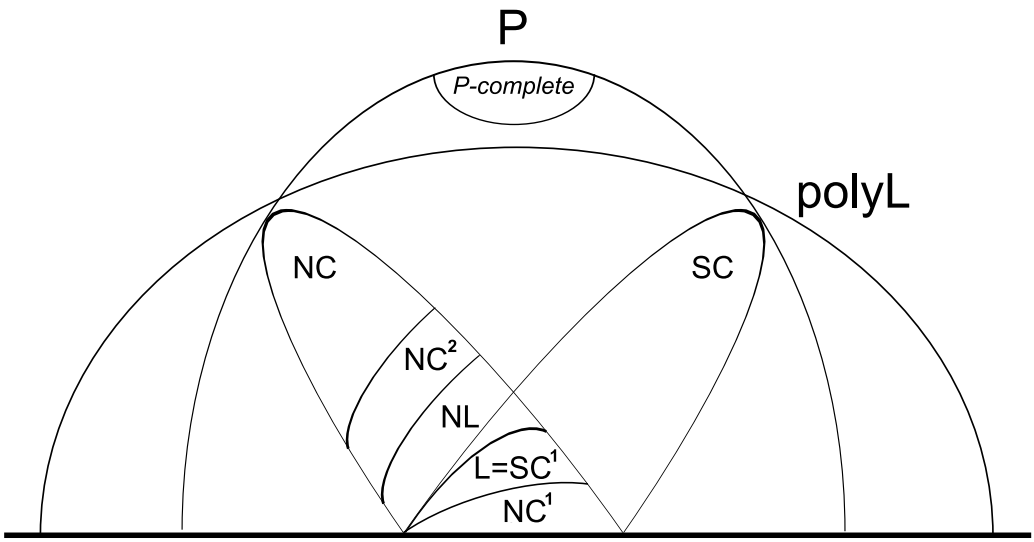


Figure 4.2: Sequential and parallel class relations (known and conjectured)

We conclude this subsection by coming back to the ordinary DTM and, especially, to the SC class (languages decided in polynomial-time and polylogarithmic-space, simultaneously). The SC seems like a perfect candidate for relating sequential and parallel computation. In fact, the definition of the SC is quite similar to that of the NC: it bounds two kind of resources simultaneously, one by a polynomial and one by a polylogarithmic function. Moreover, as we have already seen in section 3.3.1, (i) polynomial-time DTMs are computationally equivalent to polynomial-size circuits, and (ii) polylogarithmic-space DTMs are computationally equivalent to polylogarithmic-depth circuits. At first sight, these facts might lead falsely to the conclusion that NC=SC. The key point missed in this conclusion is the simultaneity of the restrictions imposed by the class definitions. To be precise, the above result (i) was established without limiting the depth of the simulating circuit, or the space of the simulating DTM. Similarly, the result (ii) was not concerned with the circuit size, or the DTM time. In other words, preserving one resource bound during the simulation might substantially alter the amount of another resource. Note that this observation applies to many class comparisons, as for example in the SC versus polyL $\cap$ P case (polyL stands for polylogarithmic-space): a language in polyL $\cap$ P has certainly one polyL and one P algorithm, but does it have one simultaneously? At this point, it

appears that SC and NC are incomparable [40]. One of their most important differences seems to be their relation to the NL class. While NL $\subseteq$ NC (see theorem 4.1.7), it is unclear whether SC $\subseteq$ NL; the NL-complete problems, e.g. the REACHABILITY, are prime candidate members of NC$-$SC. The SC versus NC case is yet another open question encountered in the field. Figure 4.2 illustrates the relations between the parallel and sequential classes as they are thought of today (using proved and conjectured inclusions) [5].

Examples of NC problems

The following classification table exemplifies the NC hierarchy by pointing out some of its characteristic members. Its purpose is to give the complexity of each class in practice, i.e., by associating it to the complexity of specific, well-known, problems.

Classification of well-known feasible highly parallel problems

AC^0	TC^0	NC^1	NC^2	NC^3
addition subtraction matrix add.	multiplication division counting majority parity	matrix mult. FVP summation prefix sum sorting tree isomor.	matrix rank matrix inver. matrix deter. min span tree SWP reachability PBPM $LFMIS_{deg\leq 2}$ MIS, SLE, FFT, JS1, CFL	MBS MPCVP

The acronyms used in the above lists are as follows: *FVP* (boolean Formula Value Problem), *SWP* (Shortest Weighted Paths), *PBPM* (Polynomially Bounded Perfect Matching), *LFMIS* (Lexicographically First Maximal Independent Set), *MIS* (Maximal Independent Set), *SLE* (System of Linear Equations), *FFT* (Fast Fourier Transform), *JS1* (Job Scheduling on one machine), *CFL* (Context Free Language), *MBS* (Maximal Bipartite Set), *MPCVP* (general Monotone Planar Circuit Value Problem).

Typically, the entries of this table refer to decision problems involving integer numbers[6]. Also, since tight lower bound proofs are rare results, the above classification bases mostly on the speed of the algorithms published in the literature until today. In other words, the entries in the above table are amenable to left column transitions (except parity, see proof of theorem 4.1.5). Moreover, since only the AC^0, TC^0, and NC^1 classes have proper inclusion proofs, the remaining columns of this table are amenable to merging (not expected though).

As usually observed in classical complexity theory, imposing constraints on the input set of a problem can significantly reduce its parallel complexity. As a characteristic example, we point out the FVP and the MPCVP. Both problems are special cases of the *CVP* (Circuit Value Problem), in which we are given a boolean circuit with a specific input and we are asked whether it will output '1'. In the general MPCVP, the circuit consists only of AND/OR gates (i.e., monotone) and its edges do not cross when the circuit is depicted in 2-D (i.e., planar graph). In the FVP, instead of a circuit, we are given a boolean formula in fully parenthesized form. Hence, the graph computing its value contains nodes with out-degree ≤ 1. Note that compared to formulas, the circuits are considered more economical in expressing boolean functions because of their ability to reuse subexpressions within their representation. As it turns out, this is a significant difference between the general circuits (unbounded gate fanout) and the boolean formulas (gate fanout ≤ 1): the CVP is a P-complete problem (inherently sequential), while the FVP is NC^1 (very efficiently parallelized). Similarly, the difficulty of evaluating a circuit is reduced to NC^3 by simultaneously restricting to planar and monotone circuits. If we further restrict to *upward stratified* circuits (all edges face to the same direction), then the complexity of MPCVP reduces to NC^2 [46]. To understand the impact of the above restrictions, we mention that the CVP remains P-complete in the following three, individual, cases: i) gates with fanaout ≤ 2, ii) monotone circuits, and iii) planar circuits.

In another direction, the difficulty of a problem can be significantly increased by elaborating on the properties of the requested solution. Consider here the MIS and the LFMIS problems. In the MIS, we seek to extract any maximal independent set from the given graph. MIS is efficiently parallel. However, if we start searching for solutions with specific properties, then the

[6]e.g., the decision version of the "integer addition" problem is stated as follows: ADD(x, y, i)='1' iff the i-th bit of the result $(x + y)$ is '1', where x,$y \in \mathbb{N}$.

problem becomes harder. For instance (see p. 118), requesting the *lexicographically first* Maximal Independent Set renders the problem P-complete (even worse, recall that the *maximum* independent set is NP-complete). The complexity of the problem is reduced –once again– by imposing constraints on its input set. The special case $\text{LFMIS}_{deg\leq 2}$, which is confined to graphs of degree ≤ 2, remains in NC^2.

Although not listed above, members of higher NC^k classes do exist. Such an example is the problem of finding a *Hamiltonian Cycle* in dense graphs with N nodes of degree $\geq N/2$. The decision version of this problem was proved to be in NC^4. Moreover, finding Hamilton Cycles in *robustly expanding digraphs* was recently shown to be in NC^5 (recall that, for general graphs, the HC problem is NP-complete).

4.1.2 Beneath and Beyond NC

At this point, we have formed a clear picture of the most important parallel complexity class, the NC. Hereafter, the study continues beneath and beyond NC, either to shed more light at the lowest level of the hierarchy, or to frame the problems defying feasible highly parallel solutions.

Finer gradation within the NC hierarchy

We start by examining the possibility of further analyzing each NC^{k+1} class to subclasses, similar to the AC^k and TC^k. As shown in the previous subsection (NC examples), such class gradations allow the creation of a more detailed classification of the parallel problems according to their complexity. Recall that the major difference between the AC and NC circuits lies in the fan-in of their gates: the AC gates have unbounded fan-in, while the NC gates feature certain bounds. There is, yet another, category of circuits that use both bounded and unbounded fan-in gates: the "semi-unbounded" fan-in circuits [21]. Let us define the SAC^k class to contain exactly those languages decided by *logspace* uniform families of circuits with $O(\log^k n)$ depth, which use bounded fan-in AND gates, unbounded fan-in OR gates, and NOT gates only at their input level. In other words, the SAC^k class is defined as the AC^k class with a restriction on the AND gates and the NOT gates. Trivially, we have $\text{SAC}^k \subseteq \text{SAC}^{k+1}$, i.e., we get a hierarchy of SAC^k classes (not known to be proper). Again, we define the union of these classes as $\text{SAC} = \bigcup_{k\geq 0} \text{SAC}^k$. Simply by looking at the fan-in of the gates we deduce that a SAC^k circuit

is a special case of an AC^k circuit. Moreover, by using de Morgan laws, we have that $NC^k \subseteq SAC^k$. Therefore, we have SAC=NC=AC=TC. More specifically, we have a finer gradation within the NC hierarchy:

$$NC^k \subseteq SAC^k \subseteq AC^k \subseteq TC^k \subseteq NC^{k+1}$$

Towards the analysis of the efficient parallel classes, significant research effort has focused at the lowest level of the NC hierarchy. This effort has led to the genartion of numerous individual complexity classes. Note however that dividing the NC^1 and NC^2 into multiple subclasses based solely on the running time of the algorithms is a controversial task [40]. This trend is likely to obscure the real bottleneck of parallel computation, which is the number of the processors. For instance, an algorithm with n^2 processors and $\log n$ time (NC^1) might turn, in practice, slower than an algorithm with n processors and $\log^2 n$ time (NC^2). In the real world, we have to take into account factros as the communication overhead, etc. Overall, many believe that the product $processors \times time$ leads to more accurate estimations in practical situations than the numerous subclasses of NC^2 and NC^1.

The class LOGCFL is defined to contain exactly those dicision problems that are *logspace* reducible to a context-free language [40]. Recall that deciding membership in a context free language is in NC^2 (CFL problem, examples' subsection). Therefore, since $L \subseteq NC^2$, we have $LOGCFL \subseteq NC^2$. By using alternative characterizations of the class LOGCFL, based on nondeterministic auxiliary pushdown automata and ATMs, one can deduce that $NL \subseteq LOGCFL \subseteq AC^1$. LOGCFL was proved to be closed under complement (this also holds for NC^k and AC^k trivially, as they are deterministic classes). Overall, it is shown that $LOGCFL=SAC^1$. Within LOGCFL we have the LOGDCFL (D stands for deterministic), which lies in $NC \cap SC$. Besides the above, we can define more classes based on reductions to famous problems. Another common example is the DET class, which is defined to contain exactly those problems that are *logspace* reducible to the "integer matrix determinant" problem (in NC^2, examples' subsection). We know that $NL \subseteq DET \subseteq NC^2$.

To take a closer look inside the class NC^1, we have to use more conservative types of uniformity than the *logspace* (as explained before, this is not necessary for the remaining of the NC hierarchy). Common uniformity choices are the $DLOGTIME$ (see p. 84) and the $ALOGIME$ (with alternating TMs). Note that the AC^0 properly includes the DLOGTIME class of decision problems [40]. The definitions/results of this paragraph assume the

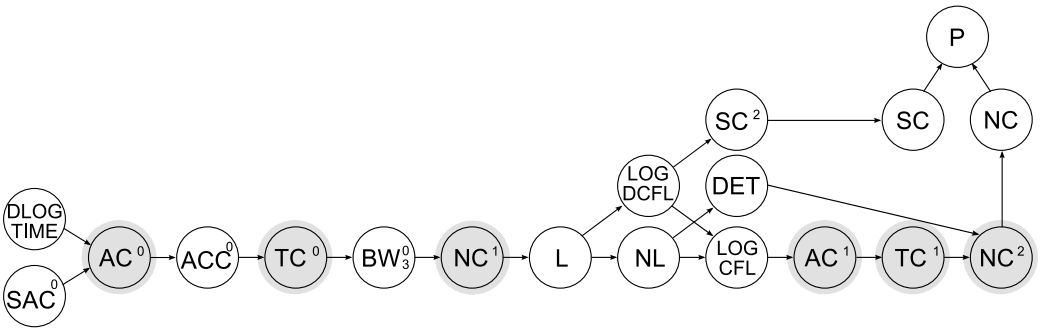


Figure 4.3: Class inclusions at the lower levels of NC

$DLOGTIME$-uniformity. Extending the idea of augmenting our circuits with MAJ gates (see TC circuits), we introduce the MOD gates: an unbounded fan-in MOD(k) gate outputs '1' iff the number of its non-zero inputs is congruent to ($0 \bmod k$) [40]. The MOD gates were introduced to tackle the PARITY problem with AC circuits of constant depth (recall the proof of theorem 4.1.5). This goal was achieved with MOD(2) gates. However, as it turns out, such circuits cannot solve all problems of NC^1. Let us elaborate by defining, first, the class $\mathrm{AC}^0[k]$ to contain exactly those languages decided by polynomial size, constant depth, unbounded fan-in Boolean circuits augmented with MOD(k) gates. Trivially, any $\mathrm{AC}^0[k]$ class contains the AC^0 class. Furthermore, for all primes p, the class $\mathrm{AC}^0[p]$ is strictly contained in NC^1. Specifically, $\mathrm{AC}^0[p]$ can not solve the problem of determining whether ($x \bmod q$) $= 0$, $x \in \mathbb{N}$, for any prime $q \neq p$ (PARITY is in $\mathrm{AC}^0[2]$, but not in $\mathrm{AC}^0[3]$). The classes $\mathrm{AC}^0[p]$ are the largest –known– proper subclasses of NC^1. It is worth noting that, for any composite number k, it is not known yet whether $\mathrm{AC}^0[k]$ is properly included in NC^1: it may as well be the case that $\mathrm{AC}^0[6]$=NP. We define the class ACC^0 to be the union of all $\mathrm{AC}^0[k]$, $k > 1$, and we summarize the above as (ACC stands for "AC with counters")

$$\mathrm{AC}^0 \subset \mathrm{ACC}^0 \subseteq \mathrm{TC}^0$$

Other parallel complexity classes have been defined by adding new or modifying the criteria mentioned so far. For instance, by replacing the unbounded fan-in OR gates of the SAC circuits with unbounded parity gates (XOR) we come up with the "parity SAC" classes $\oplus\mathrm{SAC}^k$. Or, we can divide the AC^0 class into subclasses based on the number of alternations, i, between the AND and OR gates from the input to the output of the circuit (i.e., its

$O(1)$ depth). The classes AC_i^0 form a non collapsing hierarchy within AC^0. In another direction, we can replace the depth criterion of the NC circuits with a width criterion. The class BW^k is defined as the NC^k, except that the $O(\log^k n)$ constraint refers to the width of the circuits [40]. We know that BW = SC. The class BW^0 is also divided into BW_i^0 subclasses, where i counts the absolute width of the circuit. We have that $\mathrm{AC}_i^0 \subseteq \mathrm{BW}_i^0$ for every $i \in \mathbb{N}$ and that $\mathrm{AC}^0 \subset \mathrm{BW}^0$ because PARITY is in BW^0. In fact, it is proved that the BW_i^0 hierarchy collapses and that $\mathrm{BW}_4^0 = \mathrm{BW}^0 = \mathrm{NC}^1$ (holds for both uniform and non-uniform circuits with gate fan-in=2). To sum up, the Hasse diagram in figure 4.3 depicts the class inclusions (proper or not) that we encounter at the lower levels of the NC hierarchy [40] [47].

Beyond NC

As we already mentioned in the previous chapter, any language can be decided by non-uniform circuits of size $O(n2^n)$, e.g., via the disjunctive normal form of its characteristic function. In fact, this bound can be lowered to $O(2^n/n)$ [48]. However, the vast majority of boolean functions require circuits with $\Omega(2^n/n)$ gates. Even worse, if we place a uniformity condition on the circuit families, the uniform circuit complexity of some languages exceeds $O(2^n)$. Such exponential numbers reveal the immense area beyond the NC hierarchy, which uses only polynomial size circuits.

In section 3.3.1 we established an equivalence of the polynomial DTM (i.e., the class P) to the *logspace* uniform circuits. Analogous equivalences between the TM and the circuit model can be established for classes of greater complexity. For instance, we know that the Polynomial Hierarchy (see footnote, p. 106) is also defined by *DC*-uniform boolean circuits of constant depth (in the *Direct Connect* uniformity we can find the size, a gate or an edge of a circuit by using a polynomial-time DTM) [48]. These circuits use $2^{n^{O(1)}}$ gates of unbounded fan-in and the NOT gates appear only at their input level. Moreover, if we drop the constant depth restriction then we obtain exactly the EXP class (i.e., $\mathrm{DTIME}(2^{n^{O(1)}})$). Overall, by carefully controlling the imposed restrictions on our circuits (gate fan-in, input routing, etc) we can get alternative characterizations for various TM classes (NP, PSPACE, etc) [49]. The complexity of these classes increases by allowing more resources to the circuits (e.g., depth). Notice that the above circuits are infeasible (exponential size), but they have a succinct description: any basic information regarding the circuit can be extracted in polynomial time.

One step farther, the following paragraphs examine the consequences of completely dropping the uniformity constraint that is usually imposed to each circuit family. To begin with, we have seen that non-uniform circuits have the extraordinary ability of "deciding" undecidable languages (recall however that, describing such circuits algorithmically is itself an undecidable problem). So, how can we relate these circuits to the well-studied TM classes? Is such a comparison worthy?

Before continuing to answer the above questions, we must define the Turing Machines which "take advice". This notion was introduced to capture non-uniform complexity classes. We can envisage such a TM as a machine with two inputs: the first, x, is the string for which we are interested whether $x \in L$ for some language L. The second, a_n, is a string which is given as an "advice" to the TM, in order to facilitate the computation. Formally [48],

Definition 4.1.2. *Let T,w:* $\mathbb{N} \to \mathbb{N}$ *be functions. The class of languages decidable by time-T(n) TMs with w(n) advice, denoted DTIME(T(n))/w(n), contains every L such that there exists a sequence* $\{a_n\}$, $n \in \mathbb{N}$, *of binary strings with* $|a_n| = w(n)$ *and a TM M satisfying*

$$\forall x \in \{0,1\}^n \quad M(x, a_n) = 1 \Leftrightarrow x \in L$$

On input (x, a_n) *the machine M runs for at most* $O(T(n))$ *steps.*

The languages decided in this way may not as well be recursive, i.e. decidable by regular Turing Machines. Consider the example of an advice-TM deciding a unary language L with a single bit of advice: $a_n = 1$ iff $1^n \in L$. Clearly, L may stand here for any unary language, recursive or not, because the above definition imposes no restrictions on the advice sequence $\{a_n\}$ (e.g. its construction).

Based on the definition of the advice-TM, various classes have been studied in the literature. They are characterized by the time spent to accept/reject a string (polynomial, nondeterministic-polynomial, exponential) and by the width of the advice string (polynomial, logarithmic). We point out here the P/poly class, for which we give the following two definitions and we prove that they are equivalent [48] [23].

Definition 4.1.3. *The class P/poly contains exactly those languages decided by polynomial-time TMs which use polynomial-size advices.*

Definition 4.1.4. *The class P/poly contains exactly those languages decided by families of circuits with polynomial-size.*

Theorem 4.1.10. *The definitions 4.1.3 and 4.1.4 of the class P/poly are equivalent.*

Proof. We will use bidirectional simulations, similar to those of section 3.3.1.

For the first direction, assume that a language L has a TM M running in polynomial time and using polynomial-size advices (in $n = |x|$). That is, there exists a family of advices $\{a_n\}$ and a TM M, such that M on input (x, a_n) will decide whether $x \in L$ in a polynomial number of steps. Since M exists, we know how to construct a circuit C_n to simulate $M(x, a_n)$ (as in the proof of theorem 3.3.1). The resulting circuit will have polynomial size, because the computation of M is itself polynomially bounded. Moreover, since the family $\{a_n\}$ exists, we know that there exists a family of circuits $\{C_n^a\}$ such that C_n^a outputs a_n without requiring any input (constant output): every member C_n^a is simply a hardwiring of gates, according to the string a_n (we are interested only on the existance of C_n^a, not in the difficulty of constructing it). The size of C_n^a is bounded by the length of a_n, i.e. it is also polynomial in n. By appending C_n^a to the aforementioned C_n we get a polynomial size circuit and moreover, we assemble a family of circuits to decide L (notice that this family is rendered nonuniform, because of the argument used in the construction of C_n^a).

For the second direction, assume that L has a polynomial circuit. That is, there exists a family of circuits, $\{C_n\}$, such that $C_n(x) = 1 \Leftrightarrow x \in L$, $n = |x|$. We will use the advice sequance $\{a_n\}$, where $a_n =< C_n >$ is the description of the n-th circuit of $\{C_n\}$ (the difficulty to construct $< C_n >$ is irrelevant here). A DTM can input a_n and x to simulate $C_n(x)$ in polynomial time, because the size of $C_n(x)$ is polynomially bounded in $|x|$. □

It is straightforward to see from definition 4.1.3 that any language within P also resides within P/poly. In fact, the class inclusion is proper, as there are many problems in P/poly which are not members of P: e.g. all the non-recursive languages of P/poly (recall the previous example with the $L \subseteq \{1\}^n$ decider). This conclusion is in accordance with the discussion of section 3.3.1, if we consider definition 4.1.4, thus

$$\mathrm{P} \subset \mathrm{P/poly}$$

The question that arises naturally here concerns the relation of NP to P/poly. Is NP a subset of P/poly? We know that the converse is false (due to the existence of non-recursive problems in P/poly), but what if this inclusion

does not hold either? The answer to this conundrum might even resolve the P versus NP conflict: if NP $\not\subseteq$ P/poly, then P$\neq$NP. For this reason, circuit complexity is often considered as a means of separating important complexity classes. Indeed, there is evidence that the NP class is not a subclass of P/poly: according to the Karp-Lipton theorem [50], such an "odd" inclusion would imply that the entire Polynomial Hierarchy collapses to its second level[7]. This result also strengthens the conjecture mentioned in section 3.3.1, i.e. that no NP-complete problem has a polynomial size circuit. We give bellow a proof sketch of the Karp-Lipton theorem [48].

Theorem 4.1.11. *If $NP \subseteq P/poly$ then PH=Σ_2^p.*

Proof (sketch). To show that $PH = \Sigma_2^p$ it suffices to show that $\Pi_2^p \subseteq \Sigma_2^p$ (and thus, $\Pi_2^p = \Sigma_2^p$). To show such an inclusion, it suffices to prove that a Π_2^p-complete problem belongs in Σ_2^p.

The problems in Π_2^p are described by using the syntax $\forall x \exists y \phi(x, y)$, while those in Σ_2^p by using the syntax $\exists x \forall y \phi(x, y)$. The goal of this proof is to transform a problem of the former syntax to an equivalent of the latter syntax in polynomial time. We point out the Π_2^pSAT language (Π_2^p-complete), which consists of all unquantified Boolean formulas $\phi(x, y)$ satisfying

$$\forall x \exists y \phi(x, y) = 1, \quad x, y \in \{0, 1\}^n \tag{4.1}$$

By the assumption $NP \subseteq P/poly$, there exists a family of polynomial-size circuits $\{C_n\}$ deciding SAT (i.e. Σ_1^pSAT). Specifically, $\{C_n\}$ decides whether for some given string x and formula $\phi(x, y)$ the following holds: $\exists y \phi(x, y) = 1$, for $y \in \{0, 1\}^n$. Moreover, it is known how to modify a decision algorithm/circuit so that it can output the solution of the problem (if any). Such a circuit $\{C'_n\}$ will input a formula ϕ and a string x to output

[7] that is, any problem within PH (and probably within PSPACE, as PH $\subseteq$ PSPACE) could be solved by a polynomial-time NTM with a polynomial number of queries to an oracle solving some NP-complete problem ($\Sigma_2^p = NP^{NP}$). Recall [5] that PH is a sequence of classes, which are defined recursively, based on TM using oracles. Specifically, at the zeroth level we have the classes $\Delta_0^p = \Sigma_0^p = \Pi_0^p = P$. At the first level we have $\Delta_1^p = P$, $\Sigma_1^p = NP$, $\Pi_1^p = coNP$. The classes in each of the upper levels are defined based on a polynomial-time TM using an oracle from the previous level: $\Delta_{i+1}^p = P^{\Sigma_i^p}$, $\Sigma_{i+1}^p = NP^{\Sigma_i^p}$, $\Pi_{i+1}^p = coNP^{\Sigma_i^p}$. Note that each level includes all of the previous levels and that PH=$\bigcup_{i \geq 0} \Sigma_i^p$. An equivalent way to define PH is to let Σ_i^p (respectively Π_i^p) denote those languages accepted by polynomial-time ATMs that make less than i alternations starting from an existential (respectively universal) state [48].

a string y (if any exists). Note that the circuits of the family $\{C'_n\}$ will still have polynomial size.

Let us denote by w a potential description of the circuit C'_n (by assumption, C'_n exists). Consider the language consisting of all Boolean formulas $\phi(x, y)$ satisfying

$$\exists w \forall x \big(\; w = < C'_n > \quad \wedge \quad \phi\,(x, C'_n(\phi, x)) = 1 \; \big) \tag{4.2}$$

If 4.1 is true (false) for a specific formula $\psi(x, y)$ then, by the definition of C'_n, 4.2 is also true (false) for $\psi(x, y)$. Hence, we have reduced the problem described by 4.1 to the problem described by 4.2. Moreover, the reduction is performed with Σ_2^p requirements: the circuit C'_n is constructed by using a SAT oracle (which inputs ψ and x) and the evaluation of C'_n, as well as its construction, requires at most polynomial time. Consequently, if $NP \subseteq P/poly$, then $\Pi_2^p\text{SAT} \in \Sigma_2^p$. □

An even more extreme assumption that NP is contained in P/log (a strict subset of P/poly) implies that PH collapses at its first level, i.e. P=NP [50]. Similar results given in [50] show that it is rather unlikely for deterministic-exponential-time or polynomial-space problems to have polynomial-size circuits. Specifically,

If	EXP $\subseteq$ P/poly	then	EXP $= \Sigma_2^p$
If	PSPACE $\subseteq$ P/poly	then	PSPACE $= \Sigma_2^p \cap \Pi_2^p$
Iff	PSPACE $\subseteq$ P/log	then	PSPACE = P

Note as a corollary of the first that, EXP$\subseteq$P/poly and P=NP will never hold together. These inclusions, when combined, contradict the Time Hierarchy Theorem [48]: if P=NP, then PH collapses and thus, P=Σ_2^p=EXP.

We conclude the current section by referring to a randomized complexity class of parallel computation, namely the RNC class. The RNC class is the union of all RNC^k classes, $k > 0$, where each RNC^k is defined as the NC^k, except that the circuit is *probabilistic* [1] [5]. More specifically, we have a boolean circuit, which inputs a number of extra bits (intuitively, the random bits). The number of extra bits is bounded by a polynomial $p(n)$, $n = |x|$. If $x \in L$, the circuit outputs '1' with probability at least 1/2, i.e., at least half of the $2^{p(n)}$ outputs are correct. Otherwise, if $x \notin L$, the circuit outputs '0' with probability 1 (zero sided error). The introduction of randomization

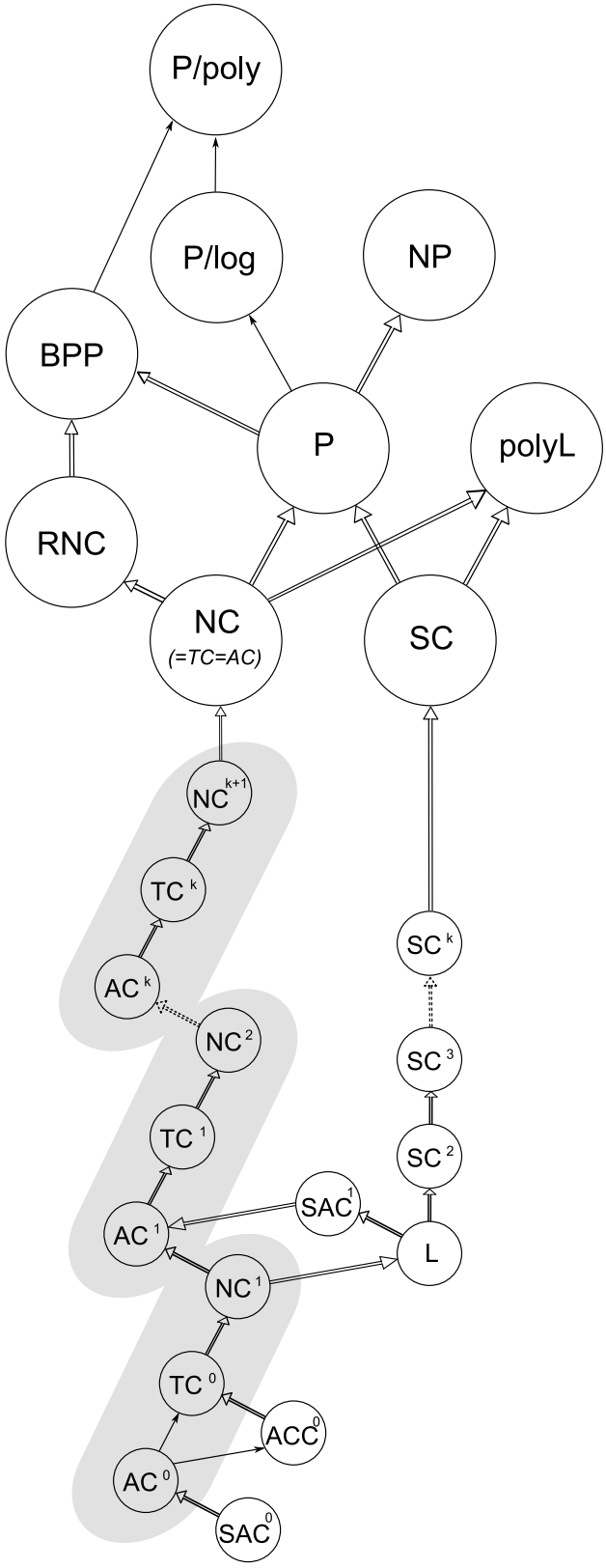


Figure 4.4: Known class relations (single arrows denote *proper* class inclusions)

to the NC class has the same effect with the RP case (randomized P) [5]. As we will see in section 4.2, the first important class beyond NC is, most probably, the class of P-complete problems (still inside P). Randomization is a potential way of attacking these problems, analogous to that of attacking NP-complete problems with randomized polynomial algorithms. A characteristic example is the "perfect matching" problem, which is shown to be in RNC [5]. More specifically, it lies in RNC $\cap$ co-RNC, i.e., there is a feasible parallel algorithm that runs in expected polylogarithmic time for any input, giving always the correct output [40]. Unfortunately, perfect matching is not yet proved to be either NC or P-complete[8]. In fact, as with the P versus NP case, no problem in RNC was ever proved to be P-complete; most researchers believe that randomization is not adequate for solving such difficult problems, neither P-complete (with RNC) nor NP-complete (with RP). Regarding the relations of the RNC to the other classes, we expect that RNC is in P (mere conjecture). Also, it is straightforward to show that NC $\subseteq$ RNC, because the use of the random bits is not mandatory. We know that RNC $\subseteq$ BPP (problems solvable with randomized sequential algorithms with two-sided error in expected polynomial time), which in turn is contained in P/poly according to Adleman's theorem [47]. Figure 4.4 summarizes most of the aforementioned –proved– class relations with a Hasse diagram (double arrows denote class inclusions and single arrows denote *proper* class inclusions).

4.2 P-completeness

Are all problems in the class P equally difficult to solve? A first, negative, answer can be obtained by using their sequential-time as a comparison measure: the Time Hierarchy Theorem [5] establishes that some problems require strictly greater amounts of time than others, i.e. the $DTIME(n^k)$ hierarchy is proper. There are other viewpoints, though, from which we cannot answer the above question with certainty[9]. As it turns out, we cannot even tell whether all P problems are easy enough to be solved in narrow space (polyL), or in moderate parallel time bounds (NC). Separating P from such

[8] recall that the special case PBPM of this problem is in NC^2 (examples' subsection).

[9] e.g., we also know, from the Space Hierarchy Theorem, that the $DSPACE(\log^k n)$ hierarchy is proper. However, we cannot use this result to further divide P with respect to space, because we expect that polyL $\nsubseteq$ P (the distinguishing problems of each polyL level might lie beyond P). In other words, we cannot tell whether SC is a proper hierarchy.

classes is an issue addressed, among others, by the theory of P-completeness.

The most direct method for separating two classes is, arguably, by showing that a specific member of one class does not reside in the other. To find such distinctive members we, naturally, search among the most difficult problems of a class (after all, we are working with difficulty classifications). Hence, the theory of P-completeness focuses on identifying the "hardest" problems of P. Indeed, as practice has shown, the *P-complete* problems are the prime candidate members of P−NC, of P−L, etc. They seem to lack feasible highly parallel, as well as, polylogarithmic-space solutions.

To identify the "hard" problems of a class we use *reductions*. In general, we say that a problem A is reduced to a problem B, if we can solve any instance of A by transforming it to an instance of B, and then, solving B. The complexity of transforming the instances determines the relative difficulty of the two problems. Intuitively, if the transformation is easy enough (compared to the complexity of B), then solving A is at most as difficult as solving B. Note that we can choose among many types of transformations (and reductions), depending on the underlying complexity class and the intended application. By using reductions, we can partially order the problems of a class according to their difficulty. The "hardest" problems are those to which we can reduce any problem of their class. We call such a problem *complete* and we can use it as a representative of its class.

Several problems have been identified as P-complete [1]. Even though it is conjectured that they all lie beyond NC, none of them was ever proved to do so. Such a striking result would imply that all P-complete problems lie beyond NC, and moreover, it would separate NC from P once and for all (the same holds for the 'P versus L' conundrum). Decades of failed attempts reveal the magnitude of the challenge and in no case do they invalidate the merit of the theory. Besides provably separating the classes, P-completeness gives us some insight to the limits of parallelization and it expands our knowledge of difficult –or, seemingly difficult– problems. In practice, a large set of known P-complete problems can, actually, guide the designer in solving efficiently parallel his own problem. For example, identifying a P-complete subproblem suggests that the decomposition of the larger problem should be reformulated. Or, it suggests that some other method should be applied (e.g. approximation). In other examples, there exist problems featuring both highly parallel and inherently sequential solutions; the designer can benefit by consulting the theory.

Historically, the notion of P-completeness appeared at roughly the same

time as that of NP-completeness (1970's) [1]. The motivation was to study the relations between sequential time and space. Cook was the first to raise the question of whether everything computable in polynomial time is also computable in polylogarithmic space ('P versus polyL', still open). Cook also gave the first P-completeness result (the "Path Systems" problem), by using logarithmic-space reducibility. Thereafter, many problems were shown to be P-complete, including the "Circuit Value" by Ladner (1975). The introduction of the class NC (Pippenger, 1978) and the importance of P-completeness in the study of parallel computation (Goldschlager et al., 1982) led to several new results by using NC reducibility. Nonetheless, there still exist numerous problems that are not known to be either P-complete or NC. Such a characteristic example is the Greatest Common Divisor, which, as the rest of the open problems, is not expected to be in NC (compare to the Factoring problem, which is not expected to be in P). Today, the general –but unproven– consensus is that the P-complete problems are inherently sequential (i.e. they are feasible but not feasible highly parallel).

In the following of this section we study P-completeness for the purposes of parallel computation. We describe the notion of reducibility and its uses. We define the *P-complete* problems and we give various examples. Moreover, we explain the NC $\subset$ P conjecture and we consider its consequences.

4.2.1 Reductions and Complete Problems

One of the central ideas used in P-completeness is the *reduction* of one problem to another. That is, the idea of finding the solution of one problem by solving another problem. For example, sometimes, we can tell whether $x \in L_A$ by transforming the instance x to a new instance $f(x)$ and answering whether $f(x) \in L_B$. The existence and the complexity of such functions, f, help us order/classify the problems according to their own complexities. Intuitively, if f exists and can be computed with less or equal complexity to that of deciding L_B, then deciding L_A cannot be more difficult than deciding L_B.

The computational complexity of a function is formalized and studied with the same methods presented in the previous section. Recall that the class NC was defined there only for decision problems. Let us define here the function analog of this class, namely the FNC. The FNC is the set of functions, which can be computed by boolean circuits of polynomial-size and polylogarithmic-depth [1]. As with the NC hierarchy, the FNC is the union of

all FNC^k classes (circuit-depth = $O(\log^k n)$). The gates of the FNC circuits have bounded fan-in and the circuit families are *logspace* uniform. FNC captures the functions that can be computed feasible highly parallel. Note that, by definition, NC $\subseteq$ FNC. Other important function classes are the FP (functions computed by polynomial-time DTMs), the FL (by logarithmic-space DTMs), the nondeterministic FNP and FNL, the nonuniform FNL/poly (by NDTM with polynomial-size advices), etc.

We continue by formalizing the notion of reductions [1].

Definition 4.2.1. *A language L_A is many-one reducible to a language L_B, written $L_A \leq_m L_B$, if there is a function f such that $x \in L_A$ if and only if $f(x) \in L_B$.*

We say that L_A is P many-one reducible to L_B, written $L_A \leq_m^P L_B$ if and only if the function f is in FP.

We say that L_A is logspace many-one reducible to L_B, written $L_A \leq_m^L L_B$ if and only if the function f is in FL.

We say that L_A is NC^k many-one reducible to L_B, for $k \geq 1$, written $L_A \leq_m^{NC^k} L_B$, if and only if the function f is in FNC^k.

We say that L is NC many-one reducible to L_A, written $L_A \leq_m^{NC} L_B$, if and only if the function f is in FNC.

We call the above reductions *many-one* because many distinct instances of the original problem A can be mapped to a single instance of the new problem B. A generalization of the many-one reducibility is the Turing reducibility. The idea is that we are not limited in a single transformation $f(x)$, and thus, to a single question of the form $f(x) \in L_B$ in order to answer whether $x \in L_A$. Instead, we are allowed to use a machine, which can issue multiple questions to an oracle giving answers for the problem B. In practice, multiple queries are more useful when we turn from decision to *search* problems[10]. We will define Turing reducibility here by using PRAMs equipped with an extra instruction-call to query their oracle (the *B-oracle*). Notice that, if we assume a unit cost for the instruction-call, the complexity of the Turing reduction captures the difficulty of reducing A to B independently of the true complexity of B. Formally [1],

Definition 4.2.2. *A search problem A is Turing reducible to a search problem B, written $A \leq_T B$, if and only if there is a B-oracle-PRAM that solves A.*

[10]i.e. find a value (a string, a path, etc) with a specific property. When the answer to a search problem is unique (for any input), then we have a *function* problem.

We say that A is P Turing reducible to B, written $A \leq_T^P B$, if and only if the B-oracle-PRAM on inputs of length n uses $n^{O(1)}$ time and $n^{O(1)}$ processors.

We say that A is NC Turing reducible to B, written $A \leq_T^{NC} B$, if and only if the B-oracle-PRAM on inputs of length n uses $log^{O(1)}n$ time and $n^{O(1)}$ processors.

Similar to the PRAM extension with oracles, we define the *B-oracle*-circuit families. These circuits include oracle gates for solving B. Such a gate is considered as a vertex with k inputs and l outputs. The k-input will be an encoding of an instance of the problem B, and the l-output will be the encoding of the corresponding solution to B. By convention, the oracle gate has depth $\log(k+l)$ and size $(k+l)$. Oracle circuit families serve the same purpose as oracle PRAMs. They are introduced to study a more detailed reduction than the $\leq_T^{NC}$ defined above. Specifically [1],

Definition 4.2.3. *A search problem A is NC^k Turing reducible to a search problem B, for $k \geq 1$, written $A \leq_T^{NC^k} B$ if and only if there is a uniform B-oracle circuit family $\{C_n\}$ that solves A, and each member C_n has depth $O(log^k n)$ and size $n^{O(1)}$.*

As already mentioned, reductions are often used to reckon whether a problem is more difficult than another (either in terms of complexity or computability). Furthermore, we can use reductions to reckon whether two problems are equally difficult [1].

Definition 4.2.4. *Suppose that for some reducibility $\leq$ we have $A \leq B$ and $B \leq A$. Then we say that A and B are equivalent under $\leq$.*

The above reductions have certain properties, which can be used to reach certain conclusions easier and faster. For instance, we know that they are *transitive*, i.e. if $A \leq B$ and $B \leq \Gamma$, then $A \leq \Gamma$. To prove such a claim for a many-one reduction, we must compose the two functions transforming the instances of the two given problems. For this, notice that the classes FP, FL, FNC^k and FNC are closed under composition [1] (it can be shown with straightforward simulations, except from FL, which requires the technique used in the proof of theorem 4.1.6 for passing the "intermediate result" [5]). To prove the claim for a Turing reduction, it is straightforward to construct a machine (circuit) utilizing the second given oracle and simulating the operations of the two given machines (e.g. by interleaving their computations). Overall [1],

Proposition 4.2.1. *All the reductions given in definitions 4.2.1, 4.2.2 and 4.2.3 are transitive.*

Moreover, since Turing reducibility is a generalization of many-one reducibility, it is straightforward to show that [1]

Proposition 4.2.2. *If $A \leq_m B$, then $A \leq_T B$. If $A \leq_m^P B$, then $A \leq_T^P B$. If $A \leq_m^{NC} B$, then $A \leq_T^{NC} B$.*

The aforementioned reductions are useful in parallel computation for specific reasons. In some cases, we want our transformation to be in FNC (or in L) so that we can solve our problem highly parallel (i.e., transform it and then solve another highly parallel problem). In other cases, we might simply want to show the complexity class of a specific problem (by comparing to other problems of that class). In most cases, we are interested in ordering problems within P, and NC, according to their complexity. Let us elaborate on the latter two situations, which involve comparisons. When comparing problems through reductions, one should be careful with the design of the transformation function or the oracle machine: the reduction should be no more complex than the problem itself (e.g., the exponential cost of a falsely chosen transformation could mask the polynomial cost of the problems under examination and thus, mislead the comparison). Therefore, before continuing, we define the *compatibility* of a reduction and we explain why the aforementioned reductions are suitable for the purposes of P-completeness [1].

Definition 4.2.5. *Let $\leq$ be a resource bounded reducibility and let $\mathcal{C}$ be a complexity class. We say that $\leq$ is compatible with $\mathcal{C}$ if and only if for all problems A and B, when $A \leq B$ and $B \in \mathcal{C}$, then $A \in \mathcal{C}$.*

It is easy to see that the resource-bounded many-one reductions given in definition 4.2.1 are compatible with P. That is, if we reduce a problem A (of unknown complexity) to a problem B $\in$ P by using one of the $\leq_m^P$, $\leq_m^L$, $\leq_m^{NC^k}$, $\leq_m^{NC}$ reductions, we actually deduce that A $\in$ P (the transformation of the input x can be computed in polynomial time, as well as the solution of B). Similarly, the many-one reductions $\leq_m^L$ and $\leq_m^{NC}$ are compatible with NC and the many-one reduction $\leq_m^{NC^k}$ is compatible with NC^k. The Turing reductions $\leq_T^P$ and $\leq_T^{NC}$ are also compatible with P and NC respectively (we can replace the oracle-instruction by an actual machine, as we know that the number of the oracle "executions" is bounded by the running time of the original oracle-PRAM). The $\leq_T^{NC^1}$ is compatible with NC^k for each $k \geq 1$ as

it preserves circuit depth. Note finally that, for any reduction $\leq$ compatible with $\mathcal{C}$ and for any problems A, B, if $A \leq B$ and $A \notin \mathcal{C}$, then $B \notin \mathcal{C}$. This is an immediate consequence of definition 4.2.5, which can be used for proving that a problem does not reside in a particular class $\mathcal{C}$.

Having defined the above tools for ordering and classifying problems, we can now give the notion of a *complete* problem. Assume that for some class $\mathcal{C}$ and some problem $B \in \mathcal{C}$, we know that every problem $A \in \mathcal{C}$ can be reduced to B by using a reduction compatible with $\mathcal{C}$ ($\forall A, A \leq B$). That is, we know that B is at least as "difficult" as any other problem in $\mathcal{C}$ and that it also resides in $\mathcal{C}$. Such a problem is called $\mathcal{C}$-complete[11]. A $\mathcal{C}$-complete problem can be viewed as a representative of the class $\mathcal{C}$. In certain cases, proving a complexity result for that problem is as good as proving a result for every problem within $\mathcal{C}$. For instance, consider that in order to show a class inclusion $\mathcal{C}_1 \subseteq \mathcal{C}_2$, it suffices to show that a $\mathcal{C}_1$-complete problem resides in $\mathcal{C}_2$ (because no problem in $\mathcal{C}_1$ is more difficult than the $\mathcal{C}_1$-complete problem, which itself can be solved under $\mathcal{C}_2$ constraints). Further, when $\mathcal{C}_1 \subseteq \mathcal{C}_2$, to show that the two classes are equal, it suffices to show that a $\mathcal{C}_2$-complete problem resides in $\mathcal{C}_1$ (which implies that $\mathcal{C}_2 \subseteq \mathcal{C}_1$). Overall, when a $\mathcal{C}_1$-complete problem resides in $\mathcal{C}_2$, and at the same time a $\mathcal{C}_2$-complete problem resides in $\mathcal{C}_1$, then $\mathcal{C}_1 = \mathcal{C}_2$.

As the title of this section indicates, we will focus on a specific set of complete problems: the P-complete problems. Formally [1],

Definition 4.2.6.
A language L_B is "P-hard under NC reducibility" if $L_A \leq_T^{NC} L_B$ for every $L_A \in P$.
A language L_B is "P-complete under NC reducibility" if $L_B \in P$ and L_B is P-hard.

Definition 4.2.6 can be extended –in the obvious way– to include function problems (FP-complete) and search problems (quasi-P-complete). Note that, one can give P-completeness results using reductions different than the NC

[11] not all classes have complete problems. For instance, no complete problems are known for "semantic" classes like RP and BPP (*Randomized Polynomial-Time* and *Bounded-Error Probabilistic Polynomial-Time*). Moreover, it is believed that some classes do not have complete problems at all. Such an example is the Polynomial Hierarchy (non-semantic class), for which we know that the existence of a PH-complete problem implies the collapse of PH to some finite level [5]. Note that the same holds for the NC hierarchy, which would collapse at the level of the –alleged– complete problem (it also collapses if NC=P).

(weaker forms of reducibility, possibly down to NC^1 many-one). However, we avoid P reducibility due to the NC $\subset$ P conjecture (we avoid allowing more computational power to the transformation than to the "complete" problem itself).

Proofs and examples of P-complete problems

To exemplify the use of the above, we present a number of P-complete problems and some commonly used proof techniques. We begin with the Circuit Value Problem (CVP), which is, probably, the most famous of the class. CVP is a decision problem: given the description of a boolean circuit C_n and a bit vector $v = [v_1, \ldots, v_n]$, is $C_n(v) = 1$? We show that [5]

Theorem 4.2.3. *CVP is P-complete*

Proof. It suffices to show that CVP $\in$ P and that A $\leq_m^L$ CVP for every A $\in$ P (note that $\leq_m^L$ is compatible with P and that it is a weaker form of reducibility than $\leq_T^{NC}$, which was used in definition 4.2.6).

It is straightforward to evaluate the output of the circuit in polynomial time: read its description and evaluate every gate in a bottom-up fashion.

To show that every A $\in$ P can be reduced to CVP, recall Theorem 3.3.1. Since problem A has a polynomial-time DTM M^A, we can construct a circuit C_n^A to simulate $M^A(x), |x| = n$, by using a logarithmic-space DTM. Therefore, any input x of A can be transformed in a string $< x, C_n^A >$ such that $x \in A$ if and only if $< x, C_n^A > \in CVP$. Since the above transformation requires only logarithmic space, we deduce that A $\leq_m^L$ CVP for every A $\in$ P. $\square$

Having proved that CVP is P-complete, we can use it to show other P-completeness results. Specifically, to establish that A $\in$ P is P-complete, it suffices to give a compatible reduction CVP $\leq$ A. Such a result implies that any problem B $\in$ P can be reduced to A by composing the two transformations (B $\leq$ CVP and CVP $\leq$ A), e.g., by using the "communication" technique used in the proof of Theorem 4.1.6. In fact, CVP plays the same role in P-completeness theory that the Satisfiability Problem (SAT) does in

NP-completeness theory[12]. Analogous to SAT, CVP is most often used to show that other problems are P-complete. Moreover, as with SAT, many variants of the CVP are also P-complete. This holds even for restricted CVP variants, which can sometimes simplify a proof. Such a case is examined next: the Monotone CVP. MCVP is the same problem, except that the given circuit consists only of AND and OR gates (no NOT gates). Nonetheless [1],

Theorem 4.2.4. *MCVP is P-complete*

Proof. First, note that as CVP $\in$ P, so is MCVP $\in$ P. To show that MCVP is P-hard, we will reduce CVP to it.

We construct a DTM which inputs an instance of CVP, i.e. $< x, C_n >$, and constructs an instance of MCVP, i.e. $< x', C'_m >$ with $|x'| = m = 2n = 2|x|$. First, the DTM inverts every bit of x and by concatenation constructs $x' =< x; \bar{x} >$. This action will allow the removal of any NOT gate from the first level of C_n (by using a connection directly to the inverted bit of the input). We extend this idea to the remaining levels of C_n by using a classical technique of circuit design, called *double-rail logic* (the idea is that for every signal-edge within C_n, there will also be a new signal of the opposite value –without using NOT gates). The DTM reads the description of C_n and when it encounters an AND gate v_i with $v_i = v_j \wedge v_k$, it outputs two gates: v_i and $\bar{v_i} = \bar{v_j} \vee \bar{v_k}$ (the bar here denotes only a symbol, not an operation). Similarly, for an OR gate $v_i = v_j \vee v_k$ it outputs v_i and $\bar{v_i} = \bar{v_j} \wedge \bar{v_k}$. For a NOT gate $v_i = \neg v_j$ it outputs $v_i = 1 \wedge \bar{v_j}$ and $\bar{v_i} = 0 \vee \bar{v_j}$.

Basically, this is a recursive definition of the circuit construction. It is easy to show by induction that, starting from x and $\bar{x}$, for any signal s_i of C_n, there is a signal $\bar{s_i}$ of C'_m such that $\bar{s_i} = \neg s_i$ (the new circuit C'_m is double the size of C_n). This allows C'_m to have the same functionality with C_n, but without using any NOT gates (when necessary, a node connects directly to the inverted signal). We are interested in the output gate of C'_m, which has the same value with C_n, that is, $< x, C_n > \in$ CVP iff $< x', C'_m > \in$ MCVP. It is clear that the above computations can be performed by the DTM in *logspace* (simple counters and indexing variables are required) and thus, CVP $\leq^L_m$ MCVP. $\square$

[12]SAT is a well known decision problem: given a boolean expression ϕ (usually in conjunctive normal form –CNF), is is satisfiable? SAT is computationally equivalent to the Circuit-SAT problem (i.e., for a given circuit C_n, is there an input assignment so that C_n will output '1'?) [5].

Other variants of the CVP that are P-complete are [1]:

◇ NAND-CVP: The circuit consists only of NAND gates. Reductions are often simplified when only one type of gate needs to be simulated.

◇ Top-CVP: Topological CVP, i.e. the vertices in the circuit are numbered and listed in topological order.

◇ P-CVP: Planar CVP, where the circuit is a *planar* graph, i.e. it can be drawn in the plane with no edges crossing. Note that Monotone-Planar-CVP $\in$ NC.

◇ AM2-CVP: Alternating Monotone Fanin 2, Fanout 2 CVP, where on any path from an input to an output, the gates are required to alternate between OR and AND.

◇ Arith-CVP: Arithmetic CVP, where we are given an arithmetic circuit with dyadic operations $+, -, *$, together with inputs $x_1, \ldots, x_n$ from a ring.

There are numerous P-complete problems, which are related to circuit complexity, graph theory, searching graphs, combinatorial optimization and flow, local optimality, logic, formal languages, algebra, geometry, real analysis, games, etc [1]. For each the above categories, we list one representative P-complete problem bellow:

◇ CVP: Circuit Value Problem (see above).

◇ LFMIS: Lexicographically First Maximal Independent Set. Given a graph G with ordered vertices and a designated vertex v, is v in the lexicographically first maximal independent set of G?

◇ BDS: Breadth-depth Search. Given an undirected graph G with a numbering on the vertices, and two designated vertices u and v, is vertex u visited before vertex v in the breadth-depth first search of G induced by the vertex numbering?

◇ LP: Linear Programming. Given an integer $n \times d$ matrix A, an integer $n \times 1$ vector b, and an integer $1 \times d$ vector c, find a rational $d \times 1$ vector x such that $Ax \leq b$ and cx is maximized. Typically, this is an FP-complete problem.

- ◇ UNAE3SAT: Unweighted, Not-all-equal Clauses, 3SAT, FLIP. Given a 3-CNF boolean formula ϕ, find a *locally optimal* assignment for ϕ. A truth assignment satisfies a clause under the not-all-equals criterion when it has at least one true and one false literal. The cost of the assignment is the number of the satisfied clauses. An assignment s is locally optimal if it has maximum cost among its *neighbors*, where the *neighbors* are assignments that can be obtained from s by flipping the value of one variable. Typically, this is an FP-complete problem.

- ◇ HORN-SAT: The CNF SAT with the special case of *Horn* clauses, i.e. each clause is a disjunction of literals having at most one positive literal.

- ◇ Memb-CFG: Context-free Grammar Membership. Given a context-free grammar G and a string x, is $x \in L(G)$?

- ◇ IM: Iterated Mod. Given the integers a and $b_1, \ldots, b_n$, is $(\ldots((a \bmod b_1) \bmod b_2)\ldots) \bmod b_n = 0$?

- ◇ PHULL: Point Location on A Convex Hull. Given an integer d, a set S of n points in $\mathbb{Q}^d$, and a designated point $p \in \mathbb{Q}^d$, is p on the convex hull of S?

- ◇ IIRF: Inverting An Injective Real Function. Given an *NC real function* f defined on $[0,1]$, compute x_0 with error less than 2^{-n}, such that $f(x_0) = 0$. The function is increasing and has the property that $f(0) < 0 < f(1)$ (it has a unique root). A real function f is in NC if an approximation to $f(x)$ with error less than 2^{-n}, for $x \in [-2^n, +2^n]$, can be computed in NC.

- ◇ LIFE: Game of Life. Given an initial configuration of the "Game of Life", a time bound T expressed in unary, and a designated cell c of the grid, is cell c live at time T?

We conclude this subsection with a P-complete problem, which can give us some insight to the forthcoming question NC $\stackrel{?}{=}$ P. In the Generic Machine Simulation Problem (GMSP) [1], we are given the description of a Turing machine $< M >$, an integer T coded in unary and an input string x. We are asked whether M accepts x within T steps. The solution to this problem is essentially a Universal Turing Machine (UTM) equipped with a counter for keeping track of the time T. The reason that T is given in unary is to

avoid the exponential time cost of the UTM[13]. We give the following proof by using NC^1 Turing reducibility [1]:

Theorem 4.2.5. *GMSP is P-complete*

Proof. First, note that GMSP $\in$ P. A modified UTM$'$ will simulate T steps of the given machine. For each one of them it will simply read the description $<M>$ of the given machine together with its input x. The execution time is polynomially bounded by the length of the UTM$'$ input, because the input includes both T (in unary) and $<M;x>$.

We now show how to reduce any A $\in$ P to the GMSP. Assume that M^A decides A and that $p(n)$ is an easily computable function, denoting an upper bound on the running time of M^A. Both M^A and $p(n)$ are considered known for the purposes of this proof. Hence, when given an input x for A, with $|x| = n$, we can transform it to the string $f^A(x) = x; <M^A>; 1^{p(n)}$. Clearly $x \in$ A iff $f^A(x) \in$ GMSP.

We argue here that $f^A(x)$ can be computed with NC^1 circuits. The first part of $f^A(x)$ is a plain copy of the input x to the output of the circuit (i.e. depth$=O(1)$, size$=O(|x|)$). The second part is a straightforward generation of a description string, which does not depend on x (hardwired circuitry of constant size and depth). The third part involves an actual computation, that of $p(n)$. Recall that this is a uniform circuit, described by a DTM. The DTM must count up to n, compute $p(n)$ and generate the corresponding number of gates, all in *logspace*. Notice that $p(n)$ need not be a tight bound on M^A, just a polynomial easy to compute. Therefore, we can choose $p(n) = 2^{k \log n}$ for some appropriate constant k. This computation results in a '1' followed by $k \log n$ 0's, i.e. it can be stored in *logspace*. It only remains to 'expand' this number to its unary representation at the output of the DTM (in the form of gates). This can be done by successively decreasing the computed number by one and appending a gate description at the output tape (the subtraction is also *logspace*). The resulting circuit has constant depth and polynomial size. Also, the family of the above circuits is highly uniform. □

[13] Algorithms that run polynomially in the length of their input just because their input is represented in unary are called *pseudopolynomial*. This characterization is related to the notion of *strong* NP-completeness; we know that, unless P=NP, strong NP-complete problems do not have even pseudopolynomial algorithms [5].

4.2.2 NC versus P

The 'NC versus P' question is analogous to the 'P versus NP' conundrum, which is encountered in the theory of NP-completeness. There, we ask whether all problems with feasible *verifiers*[14] also have feasible solutions (sequential polynomial time). Here, we ask whether all problems with feasible solutions also have feasible highly parallel solutions (polynomial work in polylogarithmic time). Most researchers believe that the answers to both questions are negative, but up until today both of them remain open.

According to the previous subsection, to show that NC=P, it suffices to show that a P-complete problem resides in NC. From a circuit point of view (see CVP), we must come up with a technique to compute the value of a polynomial circuit in polylogarithmic time by using a polynomial number of processors. From a DTM point of view (see GMSP), we must come up with an interpreter/compiler, which will input a sequential code and it will produce an equivalent feasible highly parallel code. The obstacles encountered in both directions seem rather impossible to overcome. Many theoretical efforts have failed, while at the same time, the approaches followed in the industry lead only to moderate speedups. Everyday experience shows that the P-complete problems are quite difficult to parallelize and most likely, as we say, they are *inherently sequential*. Next, we give some of the evidence that NC $\neq$ P and we discuss some of the implications.

Evidence that NC $\neq$ P

Let us begin by reporting that *monotone*-NC $\neq$ *monotone*-P [42]. That is, provably, we can separate the two classes if we confine ourselves to monotone functions (f is monotone if flipping a bit from '0' to '1' in any argument to f cannot cause the value of f to change from '1' to '0'). However the separation of the general NC and P classes is a more difficult task[15]. As mentioned above, if we could just discover how to simulate efficiently in parallel every polynomial DTM, then every feasible sequential computation could be translated automatically into a highly parallel form. Besides DTM, this observation holds for any programming language. However, such a highly

[14] a feasible verifier is a polynomial-time sequential algorithm, which can verify the correctness of an answer to a problem if it is given a succinct certificate (witness, proof) for that specific answer [48].

[15] the same holds for P vs NP, for which we also know that *monotone*-P$\neq$*monotone*-NP.

parallel interpreter is not likely to exist. This machine should be able to deduce nontrivial properties of programs by just reading their code. In many cases, finding these properties is provably an intractable, or even undecidable problem. Modern compilers exploit only properties, which tend to be very syntactic, local, or relatively simplistic. Their optimization techniques rarely make radical alterations to the set of intermediate values computed by the program (either to the method or to the computation order). Overall, the state of the art parallelizing compilers generate code with modest execution speedup (some constant factor, of great importance in the industry, but rather indifferent in our theoretical endeavors) [1].

The evidence that general simulations cannot be performed fast is strengthened by taking a closer look to known highly parallel algorithms. In many cases all such algorithms are strikingly different from good sequential algorithms for the same problem. Therefore, their automatic generation from their sequential counterparts seems far beyond the capabilities of modern parallelizing compilers [1].

Despite the above –empirical– observations, the literature includes many efforts to speed-up the execution of a general sequential machine. Alas, all of them require parallel machines with exponentially many processors. For instance, we know that any DTM running in time T can be simulated by a CREW-PRAM in $O(\sqrt{T})$ time but with $2^{\omega(\sqrt{T})}$ processors (by precomputing in parallel the huge table of the DTM transition function and then performing rapid table look-ups). Similar results have been given for general RAM simulations. For circuits, we know that any bounded fan-in boolean circuit with size S has an equivalent circuit with depth $O(S/\log S)$ and size $2^{\Omega(S/\log S)}$. We have similar results for circuit simulations of general DTMs. Note that, when we impose a feasibility constraint on the aforementioned techniques we get only a constant speedup [1].

Indeed, efficiently parallel general simulations seem impossible. What if we drop the 'generality' requirement? As expected, such a cutback allows us to design feasible highly parallel simulators for weaker relatives of the polynomial-time DTM. We can simulate machines solving only specific subclasses of P, depending on their allowed resources. Unfortunately, these subclasses seem to be separated from P by an exponential gap. This exponential gap, hard as it is to bridge, is yet another evidence that NC$\neq$P.

We will mention here [1] some of these machines together with their inferiority to the –general– polynomial-time DTM. Usually, to obtain such a convenient machine we have to impose a constraint on two kind of resources

simultaneously: time and space, time and processors, space and tree size, etc. First, consider the examples of a Finite State Machine and of a *logsapce* DTM. Both machines decide only a portion of P (to be precise for the second, we know that L $\subseteq$ NC^2 and we believe that L$\neq$P –open problem). Both can be simulated highly parallel due to the following observation. Intuitively, they carry little 'state' information. Therefore, the first and second halves of their computation are only loosely coupled. We can afford to simulate the two halves (recursively) by trying all midpoint 'states' in parallel, and selecting the correct when the first half is known. Note here that we have to supply these machines with an exponentially greater amount of resources in order to decide any problem in P. As another example, consider a generalization of the *logspace* DTM: a *logspace* and $2^{\log^{O(1)} n}$-time bounded auxiliary Push Down Automaton, or a *logspace* and $2^{\log^{O(1)} n}$-tree size bounded ATM. As before, the limited space and the loosely connected subtrees of the computation allows for highly parallel simulators. Note however that a nearly exponential gap remains before we are certain that this machine can decide any problem in P. More examples exist, which reveal the gap between the generic NC simulation and the generic P simulation. In all cases, if we relax one of the resource constraints (the gap), the known highly parallel simulation degrades to a polynomial time one. Given the above picture, one could say that NC=P when we find out that a nearly exponential increase at some resource adds no extra power to a model [1].

As a final piece of evidence that NC$\neq$P, we mention that natural approaches provably fail [1]. Many approaches in designing a general highly parallel simulation come "naturally" to those who have studied the problem. One can prove that their failure is intrinsic. That is, one can formulate an abstract model embodying all these ideas and prove that it is impossible to achieve a result strong enough to show that NC equals P.

All of the evidence given in this subsection may be quite compelling, but it is ultimately inconclusive. It does not prove that a feasible highly parallel simulator does not exist. For all that we know, it is still possible that NC equals P, or NP, or even PH. However, any such result would be a surprise to the community.

Inherently sequential, strong & strict complete, approximability

We have seen that the P-complete problems are the most difficult problems of P and that they most probably do not reside in NC (if one is not in NC,

then none is). Such a concession necessitates the use of alternative, feasible, methods for speeding up their solutions: randomization or approximation. Are all difficult problems amenable to these methods? The answer to this question requires further study of the P-complete class and it involves tools and notions, which are briefly described in the following paragraphs.

Inherently sequential is a notion introduced to permit classification of algorithms with respect to their potential parallelization [1]. It targets those algorithms for which parallelization does not result in significant time savings. Note that the term is also used to characterize problems, namely those which have feasible sequential solutions but have –or seem to have– no feasible highly parallel solution. To highlight the difference between the two usages of the term we mention that, even though we know of no provably inherently sequential problem (NC $\stackrel{?}{=}$ P), we know of many provably inherently sequential algorithms. Further, we know that some problems have both an inherently sequential algorithm and a feasible highly parallel one. Let us elaborate by giving the exact definition of the *inherently sequential RAM algorithm*. The definition examines the "nature" of the intermediate operations of the algorithm. Intuitively, the dependencies between the algorithm's intermediate values determine the degree at which we can parallelize the operations. Technically, a RAM algorithm Π is inherently sequential if the language $L_\Pi = \{x\#i\#j|$ bit i of $f_\Pi(x)$ is $j\}$ is P-complete; the *flow* function $f_\Pi(x)$ inputs x (the input of Π) and outputs a string containing every intermediate value of the computation $\Pi(x)$ in order. In other words, an algorithm is considered inherently sequential when its flow function is at least as difficult to compute as a P-complete problem. Some of the algorithms that have been proved to be inherently sequential are the standard depth-first search, the greedy algorithm for computing a lexicographically first maximal path, the Gaussian elimination with partial pivoting for solving a system of equations, etc [1]. Notice though that, the maximal path can be computed highly parallel by using randomization and that solving a system of equations is in NC. After all, proving that an algorithm is inherently sequential is only a clue that the problem might not be efficiently parallelized. A general result is that any polynomial time algorithm computing the solution to a P-complete problem is indeed inherently sequential. As with the DTMs of P-complete problems, it is widely believed that the inherently sequential algorithms cannot be automatically parallelized by compilers (even when they solve an NC problem).

Another type of algorithm studied in P-completeness is the *pseudo-NC*

algorithm [1]. It is called pseudo because it features NC performance only when given its input in unary encoding. Specifically, in *number problems*, replacing the binary input with unary results in an exponential increase of the length of the input. Hence, a problem might be solved efficiently in terms of this input length. Another characteristic of such problems is that, even with binary inputs, there exist efficient solutions under the assumption that the numbers involved in the computation are restricted to small integers. For these reasons, problems with *pseudo-NC* algorithms hold a special place within the FP-complete class (they are considered border cases, in a sense, easier than the others). This distinction gives rise to the notion of *strong* P-completeness. By definition, a problem is strongly P-complete if there exists a polynomial p such that the problem remains P-complete even when the numbers contained to its input (its instances) have magnitudes of at most $p(|x|)$. That is, a problem is strongly P-complete when it cannot be solved by a pseudo-NC algorithm, unless of course NC=P. Note that problems which do not involve numbers are all strongly P-complete. Examples of strongly P-complete are the CVP, the Linear Inequalities (a variation of LP), First Fit Decreasing Bin Packing, etc [1].

Strongly P-complete problems cannot be solved by a *fully NC approximation scheme*, unless NC=P [1]. One way of attacking a problem once proven P-complete, is to approximate it. Roughly, assume that we are given a combinatorial optimization problem Π and that we try to maximize/minimize some quantity. Can we obtain a near optimal solution by using an NC algorithm, instead of using an inherently sequential one to obtain the optimal? In some cases, this is plausible. Formally, for an instance $I \in \Pi$, let $OPT(I)$ denote the optimal solution and $A(I)$ the solution provided by some NC-approximation algorithm (running in parallel time $log^{O(1)}(|I|)$ and using $|I|^{O(1)}$ processors). $R_A(I) = A(I)/OPT(I)$ defines the approximation ratio of A for that specific instance I (generally, we are interested in the infimum of R_A over all instances of the problem Π). We say that, an NC algorithm A with inputs $\epsilon > 0$ and $I \in \Pi$ is an *NC approximation scheme for* Π if and only if it delivers a candidate solution on instance I with ratio $R_A(I) \leq 1 + \epsilon$. Moreover, we call this algorithm *fully NC approximation scheme for* Π (FNCAS), if the NC constraints are also met against the quantity $1/\epsilon$ besides the length $|I|$ (i.e. polylogarithmic parallel time and polynomial processors in $1/\epsilon$). The difference between the two approximations (simple and fully) is that when we try to approximate the solution very closely ($\epsilon \to 0$) the former might require exponential resources, while

the latter only NC. Provably, the existence of a FNCAS for a problem Π implies the existence of a pseudo-NC algorithm for Π [1]. This is actually the contraposition of the first statement of the paragraph, which establishes that not all problems are approximable. Such an example is the LFMIS-size problem (LFMIS –section 4.2.1– but asking for the size of the set). Examples of fully NC approximable problems are the First Fit Decreasing Bin Packing, the 0-1 Knapsack, the Makespan, the List Scheduling, etc[16] [1].

In another direction, when it is too much to ask a shift from polynomial to polylogarithmic time, we must raise the question of a limited polynomial parallel speedup (e.g. speedup=$\sqrt{n}$). We already have a positive answer on that for some problems in P, but not for all of them [1]. The theory of *strict P-completeness* studies P under this prism [52]. It tries to identify complete problems that exhibit some parallel speedup, and for which any further performance improvement would imply that all problems in P can have a limited polynomial speedup. The key idea is to use a reduction preserving speedup. That is, find NC reductions requiring 'little' transformation resources; with such reductions, any speedup of the P-complete problem could also have an impact to the reduced problem (the running time is not 'masked' by the reduction itself). Roughly, we define as *strict T(n)-P-complete* a problem with parallel running time $\tilde{O}(T(n))$ when[17], for any problem A∈P, the value $T(f^A(x))$ is bounded by the sequential RAM time of A (f is the many-one NC reduction of A). The first strict $\sqrt{n}$-P-complete problem was the Square Circuit Value Problem (boolean, monotone, alternating, synchronous, fan-in=2 and *square*, i.e. the number of gates at every level equals its depth).

4.3 The Parallel Computation Thesis

The previous sections present results and give proofs relating certain complexity classes. Note that, each of these classes is defined on top of a specific model of computation. In other words, each of the presented results relates only two models (when we overlook potential class/model equivalences). Are

[16] NC-approximation schemes can also be used for NP- and PSPACE-hard problems [51].

[17] *"soft Oh"* notation: $g(n) \in \tilde{O}(T(n))$ if and only if $\exists\ c > 0,\ k > 0,\ n_0 \in \mathbb{N}$ s.t. $\forall n > n_0$, $g(n) \leq T(n) \cdot (\log n)^k$. We use $\tilde{O}$ when we wish to ignore polylogarithmic factors.

there any generic statements to include 'all' models of parallel computation? This section discusses such a statement, which is analogous to the *extended Church-Turing* Thesis.

The Church-Turing Thesis (CTT) is a consensus over what should be considered *computable* by the computability/complexity theorists [53]. It expresses their common belief that the standard Turing machine can compute whatever was once vaguely referred to as "effectively calculable". Specifically, the CTT asserts that *"a function is effectively calculable if and only if it is recursive"*. Provably [25], this statement is equivalent to *"... if it is λ-definable"* and to *"... if it is Turing-computable"*, i.e. computed by a TM. The belief that the CTT is true stems from the fact that every model of computation proposed so far was eventually shown to have power equal to that of the TM. The CTT itself cannot be proved, as it attempts to relate the vague notion of "effectively calculable" to the mathematical –well defined– notions of recursion, TM, etc (some consider the thesis as a definition for the computable functions, while others think of it as a natural law).

Over the years, the CTT has appeared in various forms. One example is the *extended* CTT (ECTT), which makes the stronger assertion that *"the TM is as efficient as any computing device can be"*. That is, if a function is computable by some device in time $T(n)$, then it is computable by a TM in time $(T(n))^{O(1)}$. This polynomial relation also expresses the belief that P truly captures the notion of efficiently solvable problems, i.e. that P contains all those problems solved in polynomial time by any model (*polynomial-time CTT* [41]). Note that, if we replace 'time' by 'work' (=time×processors), then the ECTT also includes the parallel models[18].

The ECTT can be rephrased as *"time on all reasonable sequential models of computation is polynomially related"* (Sequential Computation Thesis) [54]. Before making a similar statement in the context of parallel computation, we elaborate on the characterization "reasonable". The term itself does not have a universally accepted definition, even though its purpose is somewhat clear. It serves as a filter excluding models with extraordinary abilities, e.g. deciding non-recursive languages (see non-uniform circuits), or generating exponentially large words in only polynomially many steps (see RAM with unit-cost multiplication instruction, or P-systems with membrane

[18] in some cases, ECTT (and CTT in general) has been called into question. The criticism bases mainly on models which are quite different from the paper-and-pencil framework of Turing, as for example the Quantum Computer [48] [53].

division [35]). Roughly, we will characterize a model as reasonable if it can be physically implemented with a moderate amount of resources[19]. To continue, the **Parallel Computation Thesis** (PCT) asserts that [28]:

> *On reasonable models, sequential space is polynomially related to parallel time.*

That is, if a function can be computed sequentially in space $f(n)$, then it can be computed in parallel time $O(f^k(n))$, for some constant k, and vice versa, i.e. Seq-SPACE$\left(f^{O(1)}(n)\right)$ = Par-TIME $\left(f^{O(1)}(n)\right)$. The thesis associates sequential and parallel machines. Moreover, it implies that, time on all reasonable parallel models is polynomially related.

We have already seen evidence in the previous sections that, the PCT (and its implication) holds . Recall the inclusions $NC^1 \subseteq L \subseteq NC^2$ proved in section 4.1.1. In the same section, we also showed the relation of general NC circuits to the logarithmic-space ATM. Recall that we had already proved the equivalence of NC circuits and polylogarithmic-time PRAMs in section 3.3.2. These two equivalences also implied (section 4.1.1) that polylogarithmic-time PRAMs are equivalent to logarithmic-space ATMs. The discussion in section 3.3.1 reported that for general uniform boolean circuits we have $NSPACE(S) \subseteq UniformDepth\,(S^2) \subseteq DSPACE\,(S^2)$. Notice that any of the simulations between parallel machines described in the previous chapter suffered –at most– a polynomial time slowdown.

Besides the above, various results have been published in the literature relating parallel time and sequential space. For instance, we know that for uniform aggregates, for conglomerates and for ATMs the following inclusions hold (X represents here the UAG, the CONG, or the ATM) [29]

$$\text{DSPACE}(f(n)) \subseteq \text{X-TIME}(f^2(n)) \subseteq \text{DSPACE}(f^2(n))$$

A similar result is known for the CREW-PRAM, for the Hardware Modification Machines (see p. 46) and for the SIMDAGs (see p. 13) when $f(n) \geq \log n$ (Y represents here the HMM, the SIMDAG, or the CREW-PRAM) [29]

$$\text{DSPACE}(f(n)) \subseteq \text{Y-TIME}(f(n)) \subseteq \text{DSPACE}(f^2(n))$$

[19] in a circular definition, one could denote by reasonable those models which satisfy the sequential and parallel computation theses. In other words, a new model should be considered reasonable if it is able to simulate some existing reasonable model (e.g. the TM) and vice versa [29].

For Vector Machines (see p. 47) we know that when $f(n) \geq \log n$ [32]

$$\text{DSPACE}(f(n)) \subseteq \text{VM-TIME}(f^2(n)) \subseteq \text{DSPACE}(f^4(n))$$

We can summarize the commonalities of the proofs that establish results of the aforementioned kind as follows [29]. First, consider the simulation of the sequential space (inspired by Savitch's theorem [5]). Let M be a $S(n)$ space bounded machine which, as a consequence, has at most $2^{O(S(n))}$ configurations. Construct the transition matrix or the transition graph of these configurations and compute its transitive closure by repeated squaring the matrix (e.g see proof of theorem 4.1.7) or by path doubling within the graph. Note that this step of the simulation should require logarithmic time in the number of the configurations and hence, polynomial time in $S(n)$. The simulation accepts if and only if there is a transition from the initial to the accepting configuration of M. Second, consider the simulation of parallel time. Perform a depth first search on the instructions executed by the parallel machine (e.g. see proof of theorem 4.1.6). Note that the search is executed in a recursive fashion and that the depth of the recursion is bounded by the time consumed by the simulated machine, say $T(n)$. As a result, no more space than polynomial in $T(n)$ should be required. To make a decision about the input, during the simulation we check whether a final state or an accepting instruction is reached.

The PCT can serve as a "rule of thumb" to aid the engineer in setting certain performance targets when designing his algorithm. For instance, if she/he is aware of a polylogarithmic-space sequential algorithm for a given problem, then she/he can expect that a highly parallel solution also exists for that problem, whichever –reasonable– parallel machine she/he has at her/his disposal (a reasonable model should be neither too powerful nor too weak). Notice, however, that the PCT examines only the cases where a single kind of resource is bounded at each domain (sequential space $\rightleftharpoons$ parallel time). That is, to say, the PCT is indifferent to the amount of parallel processors that it involves. Recall that, in order to avoid impractical solutions, we simultaneously bound the parallel time and the hardware cost. Of course, dealing with simultaneity is a more difficult task, especially when formulating general statements to include several distinct models. Nonetheless, parallel computation is, most often, concerned with simultaneity, and thus, we should "sharpen" our thesis accordingly.

In this direction, the PCT features two more formulations: an *extension* and a *generalization* [54]. The extended PCT asserts that, *"on any reason-*

able parallel machine, time and hardware are simultaneously polynomially related to Turing Machine reversals and space", where a reversal is said to occur when any head of the TM changes direction. The term *hardware* refers here to the product *space*×*wordsize* of a parallel machine and is considered as a good measure when memory cost dominates the cost of the processing elements (e.g. elements with minimal instruction sets). Equivalently, the term might refer to the width of a uniform circuit (the circuit-width, compared to the circuit-size, is sometimes considered as a more accurate measure of the circuit cost, because it captures the amount of hardware that is 'activated' at each time instant). The PCT and its extension are not true for massively parallel computers, which utilize an exponential number of processors (such machines can decide any NP language in sub-logarithmic time) [54]. To include such models, the generalized PCT asserts that *"on a parallel machine, time is related within a constant to the alternations of an Alternating Turing Machine, while at the same time, address complexity is related polynomially to the time of that ATM"*. The term *address complexity* refers here to the number of bits required to describe an individual unit of hardware (e.g. it refers to the wordsize of a shared memory machine, to the logarithm of the size of a circuit, etc). Similar to the PCT, cross simulations between shared-memory machines and ATMs, as well as simulations between networks of processors and DTMs give support to the generalized and the extended PCT [54]. As a final note, the aforementioned theses also highlight the theoretical efforts to capture the inherent correspondence of sequential cost to parallel cost: sequential-space to parallel-time, alternations-number to parallel-time, alternating-time to address-complexity, etc.

Chapter 5

Conclusion

This thesis has reviewed selected topics from the theory of parallel computation. It includes two main parts, one concerned with the formal description of the parallel computation and one devoted to the thorough study of its complexity.

Following the diachronic efforts of the research community to capture both computation and communication, and to balance between theory and practice, this thesis has presented the plethora of models proposed in the literature for studying parallel computation. The presentation was divided between the processor-based models and the circuit-based models. A second distinction was made between shared-memory and message-passing models. Starting with the highly abstract and continuing to the technology oriented models, this survey highlighted the differences of the resulting machines and discussed their uses and abilities. Also, from a slightly different standpoint, it reported an architectural taxonomy of the parallel machines, which is widely referenced in the applied computer science.

Subsequently, the thesis examined the computational power of the described models by using common simulations techniques. It showed that all shared memory models feature similar power (logarithmic slowdown factors) and that the distributed memory models can simulate optimally the PRAM when given sufficient parallel slackness. Moreover, it proved that, under certain resource bounds, the uniform families of Boolean circuits compute the same set of functions with the Turing machines and the PRAM.

In the second part, the thesis reviewed the most important achievements in the area of parallel complexity theory. It defined the complexity classes capturing the feasible highly parallel computation and it proved various in-

clusions among them. Furthermore, it related these classes to well-known classes of sequential computation. It reported a number of tractable problems and it classified them according to the efficiency of their parallel solutions. Beyond the realm of feasible computation, the thesis investigated the effects of parallelization and gave alternative, circuit based, characterizations to classes of higher complexity. As a means of studying the exact limits of efficient parallelization, the theory of P-completeness was explained. Reductions and completeness proofs were given, as well as a list of practical P-complete problems. Subsequently, the famous 'NC versus P' conundrum was discussed, along with the implications and the possibility of separating the two classes. Finally, the Parallel Computation Thesis was presented to relate the solutions offered by distinct models of computation.

Bibliography

[1] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, first edition, 1995.

[2] Behrooz Parhami. *Introduction to Parallel Processing Algorithms and Architectures*. Kluwer Academic, first edition, 1999.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Multithreaded Algorithms*, chapter 27 of Introduction to Algorithms. MIT Press, third edition, 2009.

[4] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, 1978.

[5] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, first edition, 1994.

[6] John H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, first edition, 1993.

[7] Leslie M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29:1073–1086, Oct 1982.

[8] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 11–21, 1989.

[9] Phillip B. Gibbons. A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, 1989.

[10] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Informatica (Springer Berlin)*, 21:339–374, Nov 1984.

[11] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica (Springer New York)*, 12:72–109, Sep 1994.

[12] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, Aug 1990.

[13] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. BSP vs LogP. In *Proceedings of the eighth annual ACM symposium on Parallel Algorithms and Architectures*, pages 25–32, 1996.

[14] Alexandros Gerbessiotis. *Topics in Parallel and Distributed Computation*. PhD thesis, Harvard University, Cambridge, Massachusetts, 1993.

[15] Duncan K.G. Campbell. A survey of models of parallel computation. Technical report, University of York, CiteSeerX, 1997.

[16] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming (IOS Press)*, 6(3):249–274, 1997.

[17] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing (Elsevier)*, 24(14):1947–1980, Dec 1998.

[18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, Aug 1993.

[19] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, first edition, 1991.

[20] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, first edition, 1992.

[21] Heribert Vollmer. *Introduction to circuit complexity: a uniform approach*. Springer-Verlag Berlin, first edition, 1999.

[22] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, Dec 1977.

[23] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, first edition, 1997.

[24] Larry Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences (Academic Press)*, 22(3):365–383, Jun 1981.

[25] Stathis Zachos. Computability and complexity, course notes. National Technical University of Athens, 2004.

[26] Patrick W. Dymond and Stephen A. Cook. Complexity theory of parallel time and hardware. *Information and Computation (Elsevier)*, 80(3):205–226, Mar 1989.

[27] Clark D. Thompson. The VLSI complexity of sorting. *IEEE Transactions on Computers,*, (12):1171–1184, Dec 1983.

[28] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, Jan 1981.

[29] Raymond Greenlaw and H. James Hoover. *Parallel Computation: Models and Complexity Issues*, chapter 45 of Algorithms and Theory of Computation Handbook (editor M. Atallah). CRC Press LLC, 1999.

[30] Stephen A. Cook and Patrick W. Dymond. Parallel pointer machines. *Computational Complexity (Birkhauser, Springer)*, 3(1):19–30, Mar 1993.

[31] Russ Miller, V. K. Prasanna-Kumar, Dionisios I. Reisis, and Quentin F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42(6):678–692, Jun 1993.

[32] Vaughan R. Pratt and Larry J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences (Elsevier)*, 12(2):198–221, Apr 1976.

[33] Klaus Sutner. On the computational complexity of finite cellular automata. *Journal of Computer and System Sciences (Academic Press)*, 50(1):87–97, Feb 1995.

[34] Lila Kari and Grzegorz Rozenberg. The many facets of natural computing. *Communications of the ACM*, 51(10):72–83, Oct 2008.

[35] Gheorghe Paun. *Introduction to Membrane Computing*, chapter 1 of Applications of Membrane Computing. Springer-Verlag, 2006.

[36] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, Apr 1990.

[37] Eric E. Johnson. Completing an MIMD multiprocessor taxonomy. *ACM SIGARCH Computer Architecture News*, 16(3):44–47, Jun 1988.

[38] Ravi B. Boppana and Michael Sipser. *The complexity of finite functions*, chapter from the Handbook of Theoretical Computer Science (vol. A). MIT Press, 1991.

[39] Michael Sipser. *Introduction to the Theory of Computation.* Thomson, second edition, 2006.

[40] D. S. Johnson. *A catalog of complexity classes*, chapter 2 of Handbook of Theoretical Computer Science: Algorithms and Complexity (editor J. van Leeuwen). MIT Press/Elsevier, 1990.

[41] Eric Allender, Michael C. Loui, and Kenneth W. Regan. *Complexity Classes*, chapter 27 of Algorithms and Theory of Computation Handbook (editor M. Atallah). CRC Press LLC, 1999.

[42] Ran Raz and Pierre McKenzie. Separation of the monotone NC hierarchy. *Combinatorica (Springer Berling)*, 19(3):403–435, Mar 1999.

[43] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Theory of Computing Systems (Springer New York)*, 17(1):1432–4350, Dec 1984.

[44] Dexter C. Kozen. *Theory of Computation (Texts in Computer Science)*. Springer-Verlag London, 2006.

[45] C. Slot and P. van Emde Boas. On tape versus core: an application of space efficient perfect hash functions to the invariance of space. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 391–400, 1984.

[46] Nutan Limaye, Meena Mahajan, and Jayalal M. N. Sarma. Upper bounds for monotone planar circuit value and variants. *Computational Complexity (Birkhauser Basel, Springer)*, 18(3):377–412, Oct 2009.

[47] Scott Aaronson, Greg Kuperberg, and Christopher Granade. Complexity zoo, http://qwiki.stanford.edu/wiki/Complexity_Zoo, May 2010.

[48] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[49] H. Venkateswaran. Circuit definitions of nondeterministic complexity classes. *SIAM Journal on Computing*, 21(4):655–670, Aug 1992.

[50] Richard M. Karp and Richard J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 302–309, 1980.

[51] Harry B. Hunt, Madhav V. Marathe, Venkatesh Radhakrishnan, S. S. Ravi, Daniel J. Rosenkrantz, and Richard E. Stearns. NC-approximation schemes for NP- and PSPACE-Hard problems for geometric graphs. *Journal of Algorithms (Elsevier)*, 26(2):238–274, Feb 1998.

[52] Anne Condon. A theory of strict P-completeness. *Journal of Computational Complexity (Birkhauser Basel)*, 4(3):220–241, Sep 1994.

[53] Andrew Chi-Chih Yao. Classical physics and the ChurchTuring thesis. *Journal of the ACM*, 50(1):100–105, Jan 2003.

[54] Ian Parberry. *Parallel Complexity Theory*. Pitman, first edition, 1987.