Chapter 2

Parallel Algorithms: Central Concepts

The first half of this chapter presents some of the more important concepts on the theory side, mainly from parallel complexity theory. The second half is devoted to issues that are central when evaluating parallel algorithms in practice.

Most of the material in this chapter is introductory in nature and provides the necessary background for reading the subsequent chapters. Parts of the text, such as Section 2.2.3 on Amdahl's law and Section 2.2.2.2 on superlinear speedup do also attempt to clarify topics about which there have been some confusion in recent years.

2.1 Parallel Complexity Theory

"The idea met with much resistance. People argued that faster computers would remove the need for asymptotic efficiency. Just the opposite is true, however, since as computers become faster, the size of the attempted problems becomes larger, thereby making asymptotic efficiency even more important."

John E. Hopcroft in his Turing Award Lecture Computer Science: The Emergence of a Discipline [Hop87].

This section presents those aspects of parallel complexity theory that are most relevant for researchers interested in parallel algorithms. It does not claim to be a complete coverage of this large, difficult and rapidly expanding
field of theoretical computer science. Most of the concepts are presented in a relatively informal manner to make them easier to understand. The reader should consult the references for more formal definitions.

Not all of the material is used in the following chapters, it is mainly provided to give the necessary background, but also to introduce the reader to very central results and theory that to a large extent are unknown among the practitioners. Parallel complexity theory is a relatively new and very fascinating field—it has certainly been the main inspiration for my work.

2.1.1 Introduction

The study of the amount of computational resources that are needed to solve various problems is the study of computational complexity. Another kind of complexity, which has been given less attention within computer science, is descriptive, or descriptional complexity. An algorithm which is very complex to describe has a high descriptional complexity, but may still have a low computational complexity [Fre86a, WW86]. In this thesis, when I use the term complexity I mean computational complexity.

The resources most often considered in computational complexity are running time or storage space, but many other units may be relevant in various contexts.\footnote{As an example, the total number of messages (or bits) exchanged between the processors is often measured for evaluating algorithms in distributed systems [Tiw87].}

The history of computational complexity goes back to the work of Alan Turing in the 1930's. He found that there exists so called undecidable problems that are so difficult that they can not be solved by any algorithm. The first example was the Halting Problem [GJ79]. However, as described by two of the leading researchers in the field, Stephen A. Cook and Richard Karp [Coo83, Kar86], the field of computational complexity did not really start before in the 1960's, and the terms complexity and complexity class were first defined by Juris Hartmanis and Richard Stearns in the paper On the computational complexity of algorithms [HS65].

Complexity theory deals with computational complexity. The most central concept here is the notion of complexity classes. These are used to classify problems according to how difficult or hard they are to solve. The most difficult problems are the undecidable problems first found by Turing. The most well-known complexity class is the class of\footnote{As an example, the total number of messages (or bits) exchanged between the processors is often measured for evaluating algorithms in distributed systems [Tiw87].} NP-complete problems. Other central concepts are upper and lower bounds on the amount of computing resources needed to solve a problem.
Parallel complexity theory deals with the same issues as "traditional" complexity theory for sequential computations, and there is a high degree of similarity between the two [And86]. The main difference is that complexity theory for parallel computations allows algorithms to use several processors in parallel to solve a problem. Consequently, other computing resources such as the degree of parallelism and the degree of communication among the processors are also considered. The field emerged in the late 1970's, and it has not yet reached the same level of maturity as complexity theory for sequential computations.

A highly referenced book about complexity theory is Computers and Intractability, A Guide to the Theory of NP-Completeness by Garey and Johnson [GJ79]. It gives a very readable introduction to the topic, a detailed coverage of the most important issues with many fascinating examples, and provides a list of more than 300 NP-complete problems. A more compact "survey" is the Turing Award Lecture An Overview of Computational Complexity given by Stephen A. Cook [Coo83]. The Turing Award Lectures given by Hopcroft and Karp [Hop87, Kar86] are also very inspiring.


2.1.2 Basic Concepts, Definitions and Terminology

To be able to discuss complexity theory at a minimum level of precision we must define and explain some of the most central concepts. This section may be skipped by readers familiar to algorithm analysis and complexity.

Problems, algorithms and complexity

Garey and Johnson [GJ79] define a problem to consist of a parameter description and a specification of the properties required of the solution. The description of the parameters is general. It explains their "nature" (structure), not their values. If we supply one set of values for all the parameters we have a problem instance. As an example, consider sorting. A formal description of the sorting problem might be

**SORTING**

**PARAMETER DESCRIPTION:** A sequence of items \( S = \{a_1, a_2, \ldots, a_n\} \) and a total ordering \( \leq \) which states whether \( a_j \leq a_k \) is true
for any pair of items \((a_j, a_k)\) from the set \(S_1\).

**Solution Specification:** A sequence of items \(S_2 = \langle a_{k_1}, a_{k_2}, \ldots, a_{k_n} \rangle\) which contains the same items as \(S_1\) and satisfies the given total ordering, i.e. \(a_{k_1} \leq a_{k_2} \leq \ldots \leq a_{k_n}\).

If we restrict ourselves to the set of integers, which are totally ordered by our standard interpretation of \(\leq\), any finite specific set of finite integers makes one instance of the sorting problem.

For most problems there exist some instances that are easy to solve, and others that are more difficult to solve. The following definition is therefore appropriate.

**Definition 2.1 (Algorithm)** ([GJ79] p. 4)
An algorithm is said to **solve a problem** \(I\) if that algorithm can be applied to any instance \(I\) of \(I\) and is guaranteed always to produce a solution for that instance \(I\). \(\Box\)

We will see that some algorithms solve all problem instances equally well, while other algorithms have a performance that is highly dependent on the characteristics of the actual instance.\(^2\)

In general we are interested in finding the best or “most efficient” algorithm for solving a specific problem. By **most efficient** one means the minimum consumption of *computing resources*. Execution time is clearly the resource that has been given most attention. This is reasonable for sequential computations—especially today when most computers have large memories. For parallel computations other resources will often be crucial. Most important in general are the number of processors used and the amount of communication.

The time used to run an algorithm for solving a problem usually depends on the **size of the problem instance**. A tool for comparing the efficiency of various algorithms for the same problem should ideally be a yardstick which is independent of the problem size.\(^3\) However, the relative performance of two algorithms will most often be significantly different for various sizes of the problem instance. This fact makes it natural to use a “judging tool” which has the size of the problem instance as a parameter. A **time complexity function** is such a tool and perhaps the most central concept in complexity theory.

\(^2\)As an example, consider the concept of *presortedness* in sorting algorithms [Man85].

\(^3\)As we shall see in Section 2.1.4, complexity classes provide this kind of “problem size independent comparison”.

16
Definition 2.2 (Time complexity function)

A time complexity function for an algorithm is a function of the size of the problem instances that, for each possible size, expresses the largest number of basic computational steps needed by the algorithm to solve any possible problem instance of that size.

It is common to represent the size of a problem instance as a measure of the "input length" needed to encode the instance of the problem. In this thesis, I will use the more informal term problem size. (See the discussion of input length and encoding scheme at pages 5–6 and 9–11 of [GJ79].) For most sorting algorithms, it is satisfactory to represent the size of the problem instance as the number of items to be sorted, denoted $n$. For simplicity, we will use $n$ to represent the problem size of any problem for the rest of this section.

Worst case, average case and best case

Definition 2.2 leads to another central issue—the difference between worst case, average case and best case performance. Even if we restrict to problem instances of a given fixed size, there may exist easy and difficult instances. Again, consider sorting. Many algorithms are much better at sorting a sequence of $n$ numbers which to a large degree are in sorted order than a sequence in total unorder [McG89]. The definition describes that a time complexity function represents an upper bound for all possible cases, and a more explicit term would therefore be worst case time complexity function. The worst case function provides a guarantee and is certainly the most frequently used efficiency measure. If not otherwise stated, this is what we mean by a time complexity function in this thesis.

Best case time complexity functions might be defined in a similar way. Average case complexity functions are of obvious practical importance. Unfortunately, they are often very difficult to derive.

Definition 2.2 states that time is measured in "basic computational steps". It is therefore necessary to do assumptions about the underlying computational model that executes the algorithm. This important topic is treated in Section 2.1.3. For the following text, it is appropriate to think of a basic computational step as one machine instruction.

Order notation and asymptotic complexity

There is an infinite number of different complexity functions. The concept

\footnote{At least in the theory community.}
Table 2.1: Examples on the use of big-O (order notation)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>complexity function</th>
<th>order expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$10n + 2n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>B</td>
<td>$40n + 10n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>C</td>
<td>$3n + 120n \log n$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>D</td>
<td>$100$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

is therefore not an ideal tool to make broad distinctions between algorithms. We need an abstraction of the (exact) complexity functions which makes them more convenient to describe and discuss at a higher level and easier to compare, without losing their most important information. Order notation has shown to be a successful abstraction of this kind. When we say that $f(x) = O(g(x))$ we mean, informally, that $f$ grows at the same rate or more slowly than $g$ when $x$ is very large.$^5$

**Definition 2.3 (Big-O) ([Wil86] p. 9)**

$f(x) = O(g(x))$ if there exist positive constants $c$ and $x_0$ such that $|f(x)| < cg(x)$ for all $x > x_0$. □

The order notation provides a compact way of representing the growth rate of complexity functions. This is illustrated in Table 2.1 where we have shown the exact complexity functions and their corresponding order expressions for four artificial algorithms.$^6$ Note that it is common practice to include only the fastest growing term. (Algorithm D is a so called constant time algorithm whose execution time does not grow with the problem size.)

It is important to realise that order expressions are most precise when they represent asymptotic complexity, i.e. the problem size $n$ becomes infinitely large. (The “error” introduced by omitting the low order terms$^7$ approaches zero as $n \to \infty$.) Note also that we omit the multiplicative constants—algorithms A and B are regarded as equally efficient. This is

---

$^5$f(x) = O(g(x)) is read as “f(x) is of order (at most) g(x),” or “f(x) is big-oh of g(x)”.

$^6$Consider algorithm C. It would be correct to say that its complexity function is $O(n^2)$ since $n^2$ grows faster than $n \log n$. However, that would be a weaker statement and there is in general no reason for “saying less than we know” in this context.

$^7$As done when describing A as $O(n^3)$ instead of $O(n + n^2)$.
essential for making order notation robust to changes of technology and implementation details.

Upper and lower bounds
A substantial part of computer science research consists of designing and analysing new algorithms that in some sense\textsuperscript{8} are more efficient than those previously known. Such a result is said to establish an upper bound. Most upper bounds focus on the execution time, they describe asymptotic behaviour, and they are commonly expressed by using the Big-O notation.

**Definition 2.4 (Upper bound)** ([Akl85] p. 3)
An upper bound (on time) $U(n)$ is established by the algorithm that, among all known algorithms for the problem, can solve it using the least number of steps in the worst case.

Several sequential algorithms exist that sort $n$ items in $O(n \log n)$ time. One example is the Heapsort algorithm invented by J. Williams [Wil64]. Consequently, we know that sorting can be done at least that fast on sequential computers. Each single of these algorithms implies the existence of an upper bound of $O(n \log n)$, but note that the order notation is too rough to assess which single algorithm corresponds to the upper bound.

**Lower bounds** are much more difficult to derive, as expressed by Cook in [Coo83]: "The real challenge in complexity theory, and the problem that sets the theory apart from the analysis of algorithms, is proving lower bounds on the complexity of specific problems."

**Definition 2.5 (Lower bound)** ([Akl85] p. 3)
A lower bound (on time) $L(n)$ tells us that no algorithm can solve the problem in fewer than $L(n)$ steps in the worst case.

While an upper bound is a result from analysing a single algorithm—a lower bound gives information about all algorithms that have been or may be designed to solve a specific problem. If a lower bound $L(n)$ has been proved for a problem $\Pi$, we say that the inherent complexity of $\Pi$ is $L(n)$.

A lower bound is of great practical importance when designing algorithms since it clearly states that there is no reason for trying to design algorithms that would be more efficient.

---

\textsuperscript{8}The possible variations in factors such as computational model, assumptions about the input (size and characteristics), and charging of the different computational resources, imply the existence of several algorithms that all in some way are most efficient for solving a (general) problem.
The influence of assumptions is illustrated by the following lower bound given by Akl in [Akl85]. This bound is trivial but highly relevant in many practical situations. See also A "Zero-Time" VLSI Sorter by Miranker et al. [MLK83].

If input and/or output are done sequentially, then every parallel sorting algorithm requires $\Omega(n)$ time units.

It is common to use big-omega, $\Omega$, to express such lower bounds. $\Omega(n)$ in this context is read as "omega of $n$" or "of order at least $n$". It can informally\(^9\) be perceived as the opposite of 'O'. As done by several authors (Harel [Har87], Garey and Johnson [GJ79]) we will stick to big-O notation for expressing orders of magnitude.\(^10\)

An algorithm with running time that matches the lower bound is said to be (time) optimal.

### 2.1.3 Models of Parallel Computation

"Thus, even in the realm of uniprocessors, one lives happily with high-level models that often lie. This should cool down our ambition for "true to life", realistic multiprocessor models."

Marc Snir in Parallel Computation Models—Some Useful Questions [Sni89].

While the RAM (random access machine) computational model introduced by Aho, Hopcroft and Ullmann in [AHU74] have been dominating for sequential computations we are still missing such a unifying model for parallel processing [Val90, San89b]. A very large number of models of parallel computation have been proposed. The state is further cluttered up by the fact that various authors in the field often use different names on the same model. They also often have different opinions on how a specific model should operate [San89b].

This section does not aim at giving a complete survey of all models. It starts by shortly mentioning some classification schemes and surveys, and then introduces the P-RAM model and its numerous variants.

\(^9\) In fact the 'Ω' is the "negation" of 'o' which is a more precise variant of 'O'. ('Θ', often read as "of order exactly" is still more precise.) Exact definitions of the five symbols 'O', 'Ω', 'Θ', 'o' and '~' that are used to compare growth rates may be found in [Wil86].

\(^10\) This is to avoid introducing excessive notation. $\Omega(f(n))$ may be expressed as of order at least $O(f(n))$. 

20
The P-RAM model is the most central concept in this thesis, and is therefore thoroughly described in Chapter 3. The section ends with a discussion of important general properties of models, and some comments on a very interesting model recently proposed by Leslie Valiant [Val90].

2.1.3.1 Computing Models—Taxonomies and Surveys

There has been proposed a very large number of models for parallel computing ranging from realistic, message passing, asynchronous multiprocessors—to idealised, synchronous, shared memory models. These models and all the various kinds of parallel computers that have been built form a diverse, and complicated dynamic picture. This “mess” has motivated the development of numerous taxonomies and classification schemes for models of parallel computing and real parallel computers.

Flynn’s taxonomy

Without doubt, the most frequently used classification of (parallel) computers is Flynn’s taxonomy presented by Michael Flynn in 1966 [Fly66]. The taxonomy classifies computers into four broad categories:

SISD  
*Single Instruction stream – Single Data stream*

The von Neumann model, the RAM model, and most serial (i.e. uniprocessor) computers fall into this category. A single processor executes a single instruction stream on a single stream of data. However, SISD computers may use parallelism in the form of pipelining to speed up the execution in the single processor. The CRAY-1, one of the most successful supercomputers is classified as a SISD machine by Hwang and Brigg [HB84].

SIMD  
*Single Instruction stream – Multiple Data stream*

Models and computers following this principle have several processing units each operating on its own stream of data. However, all the processing units are executing the same instruction from one instruction stream, and they may therefore be realised with a single control unit. Array processors are in this category. Classical examples are ILLIAC IV, ICL DAP and MPP [HB84]. The Connection Machine [Hil87, HS86] is also SIMD. Most SIMD computers operates synchronously using a single global clock.

Many authors classify the PRAM model as SIMD, but this is not consistent with the original papers defining that model (see Section
MISD  *Multiple Instruction stream – Single Data stream*

This class is commonly described to be without examples. It can be perceived as several computers operating on a single data stream as a “macro-pipeline” [HB84]. Pipelined vector processors, which commonly are classified as SISD, might also be put into the MISD class [AG89].

MIMD  *Multiple Instruction stream – Multiple Data stream*

These machines have several processors each being controlled by its own stream of instructions and each operating on its own stream of data. Most multiprocessors fall into this category. The processors in MIMD machines may communicate through a shared global memory (tightly coupled), or by message passing (loosely coupled). Examples are Alliant, C.mmp, CRAY-2, CRAY X-MP, iPSC, Ncube [AG89, HJ88].

Some authors use the term MIMD almost as synonymous with asynchronous operation [Akl89, Ste90, Dun90]. It is true that most MIMD computers which have been built operate asynchronously, but there is nothing wrong with a synchronous MIMD computer (see also Section 3.2.1).

In practice Flynn’s taxonomy gives only two categories of parallel computers, SIMD and MIMD. Consequently, it has long been described as too coarse [Sny88, AG89].

**Duncan’s taxonomy and other classification schemes**

Ralph Duncan [Dun90] has very recently given a new taxonomy of parallel computer architectures. It is illustrated in Figure 2.1. Duncan’s taxonomy is a more detailed classification scheme than Flynn’s taxonomy. It is a high-level practically (or machine) oriented scheme—well suited for classifying most contemporary parallel computers into a small set of classes. The reader is referred to Duncan’s paper for more details. Many other taxonomies have been proposed. Basu [Bas87] has described a taxonomy which is tree structured in the same way as Duncan’s taxonomy, but more detailed and systematic. Other classification schemes have been given by Hindler (see for instance [HB84], page 37), Feng (see [HB84], page 35), Kuck (see [AG89], page 112), Snyder [Sny88], and Treleaven (see [AG89], page 113).
Surveys
A good place to start reading about existing computers and models for parallel processing is Chapter 2 of *Designing Efficient Algorithms for Parallel Computers* written by Quinn [Qui87]. Surveys are also given by Kuck [Kuc77], Mikloško and Kotov [MK84], Akl [Akl89], DeCegama [DeC89], and Almasi and Gottlieb [AG89]. H. T. Kung, who is well known for his work on systolic arrays, has written a paper *Computational models for parallel computers* [Kun89] which advocates the importance of computational models. The scope of the paper is restricted to 1D (i.e. linear) processor arrays, but Kung describes 9 different computational models for this rather restricted kind of parallel processing. This illustrates the richness of parallel processing.

Theoretical models
The taxonomies and surveys mentioned so far are mainly representing parallel computers that have been built. Some of the more realistic models for parallel computation may be classified with these taxonomies. However, many of the more theoretic models do not fit into the schemes. Again, we

---

Figure 2.1: Duncan’s taxonomy of parallel computer architectures [Dun90].
notice the gap between theory and practice.

The paper *Towards a complexity theory of synchronous parallel computation* written by Stephen Cook [Coo81] describes many of the most central models for parallel computation that are used in the theory community. In Cook’s terminology these are uniform circuit families, alternating Turing machines, conglomerates, vector machines, parallel random access machines, aggregates and hardware modification schemes. The paper is highly mathematical, and the reader is referred to the paper for more details. A *Taxonomy of Problems with Fast Parallel Algorithms*, also written by Cook [Coo85] is mainly focusing at algorithms but does also give an updated view of the most important models. The paper gives an extensive list of references to related work.

The first part of *Routing, Merging and Sorting on Parallel Models of Computation* by Borodin and Hopcroft [BH85], and *Parallel Machines and their Communication Theoretical Limits* by Reischuk [Rei86] give overviews with a bias which are closer to practical computers. More specifically, they are giving more emphasis on the shared memory models which are easier to program but less mathematical convenient.

Perhaps the most “friendly” introduction to theoretical models for parallel computation for computer scientists is Chapter 2 of James Wyllie’s PhD Thesis *The Complexity of Parallel Computations* [Wyl79]. Wyllie motivates and describes the P-RAM model which probably is the most used theoretical model for expressing parallel algorithms. It is described below.

### 2.1.3.2 The P-RAM Model and its Variants

“Most parallel algorithms in the literature are designed to run on a PRAM.”

Alt, Hagerup, Mehlhorn and Preparata in *Deterministic Simulation of Idealised Parallel Computers on More Realistic Ones*, [AHMP86].

“The standard model of synchronous parallel computation is the P-RAM.”

Richard Anderson in *The Complexity of Parallel Algorithms*, [And86].

---

12I have not studied all the details of this and some of the other mathematical papers which are referenced. The purpose of the short description and the references is to introduce the material and give exact pointers for further reading.
The parallel random-access machine (PRAM) is by far the most popular model of parallel computation."

Bruce Maggs in *Beyond Parallel Random-Access Machines*, [Mag89].

The most popular model for expressing parallel algorithms within theoretical computer science is the P-RAM model [Agg89, Col89, Kar89, RS89].

**The Original P-RAM of Fortune and Wyllie**
The parallel random access machine (P-RAM) was first presented by Steven Fortune and James Wyllie in [FW78]. It was further elaborated in Wyllie's well-written Ph.D. thesis *The Complexity of Parallel Computations* [Wyl79].
The P-RAM is based on random access machines (RAMs) operating in parallel and sharing a common memory. Thus it is in a sense the model in the world of parallel computations that corresponds to the RAM (Random Access Machine) model that certainly is the prevailing model for sequential computations. The processors operate synchronously. The connection to the RAM model should not be a surprise since John Hopcroft was the thesis advisor of James Wyllie.

Today, a large number of variants of the P-RAM model are being used.
The original P-RAM model of Fortune and Wyllie, which is the main underlying concept of this work, is described in Chapter 3. Below we describe its main variants, and mention some of the other names that have been used on these.

**The EREW, CREW and CRCW Models**
Today, the mostly used name on the original P-RAM model is probably CREW PRAM [SV84]. CREW is an abbreviation for the very central concurrent read exclusive write property. The main reason for this name is the possibility to explicitly distinguish the model from the two other variants—EREW PRAM and CRCW PRAM.

The EREW PRAM does not allow concurrent read from the global memory. It may be regarded as more realistic than the CREW PRAM, and some authors present algorithms in two variants—one for the CREW PRAM and another for the EREW PRAM model. The EREW PRAM algorithms are in general more complex, see for instance [Col88].

---

13This means that several processors may at the same time step read the same variable (location) in global memory, but they may not write to the same global variable simultaneously.
The CRCW PRAM allows concurrent writing in the global memory and is less realistic than the CREW PRAM. It exists in several variants, mainly differing in how they treat simultaneous “writes” to the same memory location (see below). Reischuk [Rei86] discusses the power of concurrent read and concurrent write. The ERCW variant, which is the fourth possibility, is in general not considered because it seems more difficult to realise simultaneous writes than simultaneous reads [Rei86].

The CREW PRAM model has probably become the most popular of these variants because it is the most powerful model which also is “well-defined”—the CRCW variant exists in many subvariants.

**The various CRCW models**

One of the problems with the CRCW PRAM model is the use of several definitions for the semantics of concurrent writing to the global memory. The main subvariants for the handling of two or more simultaneous write operations\(^{1,4}\) to the same global memory cell are:

1. **Common value only.** An arbitrary number of processors may write to the same global memory location provided that they all write the same value. Writing different values is illegal. [Fei88, Akl89, BH85, Rei86]

2. **Arbitrary winner.** One arbitrary of the processors which are trying to write to the same location succeeds. The value attempted written by the other processors are lost. (This gives nondeterministic operation). [Fei88, Rei86, BH85]

3. **Ordered.** The processors are ordered by giving them different priorities, and the processor with highest priority wins. [Fei88, Rei86, Akl89, BH85]

4. **Sum.** The sum of the quantities written is stored. [Akl89]

5. **Maximum.** The maximum of the values written is stored. [Rei86]

6. **Garbage.** The resulting value is undefined. [Fei88]

**PRAC, PRAM, WRAM etc.—Terminology**

The various variants of the PRAM model have been used under several

---

\(^{1,4}\) This is often called a write conflict.
different names. The following list is an attempt to reduce possible confusion and to mention some other variants.

\( PRAC \), corresponds to EREW PRAM [BH85].

\( PRAM \), is the original P-RAM model of Fortune and Wyllie, and has often been given rather different names. In [QD84] the term MIMD-TC-R is used, and in [Qui87] SIMD-SM-R is used.

\( WRAM \), corresponds to CRCW PRAM [BH85, GR88].

\( CRAM \), \( ARAM \), \( ORAM \), and \( MRAM \), have been used by Reischuk [Rei86] to denote the, respectively, Common value only, Arbitrary winner, Ordered, and Maximum variants of the CRCW PRAM model (see above).

\( DRAM \), is short for distributed random access machine and has been proposed by Leiserson and Maggs as a more realistic alternative to the PRAM model [Mag89].

2.1.3.3 General Properties of Parallel Computational Models

Motivation

A computing model is an idealised, often mathematical description of an existing or hypothetical computer. There are many advantages of using computing models.

- **Abstractions simplify.** When developing software for any piece of machinery a computing model should make it possible to concentrate on the most important aspects of the hardware, and hiding low-level details.

- **Common platform.** A computing model should be an agreed upon standard among programmers in a project team. This makes it easier to describe and discuss measured or experienced performance of various parts of a software system. A good model will increase software portability, and also provide a common language for research and teaching [Sni89]. (See also the paragraph below about Valiant’s bridging BSP model.)

- **Analysis and performance models.** In theoretical computer science computational models have played a crucial role by providing a common base for algorithm analysis and comparison. The RAM model has

\[ \text{Note that the literature use a wide variety of terms for this concept. Examples are computing model, computer model, computation model, computational model, and model of computation.}\]
been used as a common base for (asymptotic) analysis and comparison of sequential algorithms. For parallel computations the P-RAM model of Fortune and Wyllie [FW78, Wyl79] has made it possible, for a typical analysis of parallel algorithms, to concentrate on a small set of well defined quantities; number of parallel steps (time), number of processors (parallelism) and sometimes also global memory consumption (space). More realistic models typically use a larger number of parameters to describe the machine, and on the other side of the specter we have special purpose performance models which are highly machine dependent and often also specific to a particular algorithm. (See for instance [AC84, MB90].)

What is the “right” model?
The selection of the appropriate model for parallel computation is certainly one of the most widely discussed issues among researchers in parallel processing. This is clearly reflected in the proceedings of the NSF-ARC Workshop on Opportunities and Constraints of Parallel Computing [San89a]. There are many difficult tradeoffs in this context. Some examples are easy (high-level) programming vs. efficient execution (e.g. shared memory vs. message passing) [Bil89], general purpose model vs. special purpose model (e.g. P-RAM vs. machine or algorithm specific models), easy to use vs. mathematical convenience (e.g. P-RAM vs. “boolean circuit families”[Coo85]), and easy to analyse vs. easy to build (e.g. synchronous vs. asynchronous).

Instead of arguing for some specific (type of) model to be the best, we will describe important properties of a good model. Some of these properties are overlapping, and unfortunately, several of the desired properties are conflicting.

1. Easy to understand. Models that are difficult to understand, or complex in some sense (for instance by containing a lot of parameters or allowing several variants) will be less suitable as a common platform. Different ways of understanding the model will lead to different use and reduced possibilities of sharing knowledge and experience. The importance of this issue is exemplified by the PRAM model. In spite of being one of the simplest models for parallel computations it has been understood as a SIMD model by many researchers and as a MIMD model by others (see Section 3.2.1).

2. Easy to use. Designing parallel algorithms is in general a difficult
task. A good model should help the programmer to forget unnecessary low-level details and peculiarities of the underlying machine. The model should help the programmer to concentrate on the problem and the possible solution strategies—it should not add to the difficulties of the program design process. Simple, high level models are in general most easy to use. They are well suited for teaching and documentation of parallel algorithms. However, when designing software for contemporary parallel computers one is often forced to use more complicated and detailed models to avoid “loosing contact with the realities”. Synchronous models are in general easier to program than asynchronous models. This is reflected by the fact that some authors use the term chaotic models to denote asynchronous models [Gib89]. Shared memory models seem to be generally more convenient for constructing algorithms [BH85, Par87].

3. Well defined. A good model should be described in a complete and unambiguous way. This is essential for acting as a common platform. The so-called CRCW PRAM model exists in many variants (see page 26) and is therefore less popular than the CREW PRAM model.

4. General. A model is general if it reflects many existing machines and more detailed models. The use of general models yields more portable results. The RAM model has been a very successful general model for uni-processor machines.

5. Performance representative. Marc Snir has written an interesting paper [Sni89] discussing at a general level to what extent high level models lead to the creation of programs that are efficient on the real machine. Informally, good models should give a performance rating of (theoretical) algorithms (i.e. run on the model) that is closely related to the rating obtained by running the algorithms on a real computer [Sni89] (See also [Sny88]). In other words, the practically good algorithms should be obtained by refining the theoretically good ones.

6. Realistic. Many researchers stress the importance of a computing model to be feasible [Agg89, Col89, Sny88]. Models that can be realised with current technology without violating too many of the model assumptions have many advantages. Above all, they give a model performance which is representative of real performance as discussed above. With respect to feasibility, there is a great difference
between models that are based on a fixed connection network of processors and models based on the existence of a global or shared memory. The fixed connection models are much easier to realise, and in fact the best way of implementing shared memory models with current technology [BH85, RBJ88]. Unfortunately, the fixed connection models are in general regarded as more difficult to program. Similarly, asynchronous operation of the processors is realistic but generally accepted as leading to more difficult programming [Ste90]. On the other hand there are researchers arguing for more idealised and powerful models than the PRAM (which is the standard shared memory model) [RBJ88]. Note also that there are reasonable arguments for a model to be “far from” current technology [Vis89, Val90]. This is explained in the following property and in the next paragraph.

7. **Durable.** Uzi Vishkin [Vis89] argues that computing models should be robust in time in contrast to technological feasibility which rapidly keeps advancing. Again, the RAM model is an example. Changing models too often will greatly reduce the possibilities of sharing information and building on other work. However, machines will and should change—new technological opportunities continue to appear. In practice, this is an argument against realistic models such as asynchronous fixed connection models (sometimes also called message passing models). A similar view has been expressed by Leslie Valiant [Val90].

8. **Mathematical convenience.** Stephen Cook describes shared memory models as unappealing for an enduring mathematical theory due to the arbitrariness in its detailed definition. He advocates uniform Boolean circuit families as more attractive for such a theory [Coo85]. In my view, models based on circuit families are not suited for expressing large, practical parallel systems. Most algorithm designers use the PRAM shared memory model, and it should be noted that it contains two drastic assumptions which are introduced for mathematical convenience [Wy79]. Assuming *synchronous operation* of all the processors makes the notion of “running time” well defined and is crucial for analysing time complexity of algorithms. Assuming *unbounded parallelism* (i.e. an unbounded number of processors is available) makes it possible to handle asymptotical complexity.
2.1.3.4 Valiant’s Bridging BSP Model

Leslie Valiant has recently written a very interesting paper [Val90] where he advocates the need for a bridging model for parallel computation, and proposes the *bulk-synchronous parallel (BSP)* model as a candidate for this role.

He attributes the success of the von Neumann model of sequential computation to the fact that it has been a bridge between software and hardware. On the one side the software designers have been producing a diverse world of increasingly useful and resource demanding software assuming this model. On the other side the hardware designers have been able to exploit new technology in realising more and more powerful computers providing this model.

Valiant claims that a similar standardised *bridging model* for parallel computation is required before general purpose parallel computation can succeed. It must imply convenient programming so that the software people can accept the model over a long time. Simultaneously, it must be sufficiently powerful for the hardware people to continuously provide better implementations of the model. A realistic model will not act as a bridging model because technological improvements are likely to make it old-fashioned too early. For the same reason a bridging model should also be simple and not defined at a too detailed level. There should be open design choices allowing better implementations without violating the model assumptions [Val90].

Valiant’s BSP model of parallel computation is defined as the combination of three attributes: a number of processing components, a message router and a synchronisation facility. The *processing components* perform processing and/or memory functions. The *message router* delivers messages between processing components. The *synchronisation facility* is able to synchronise all or a subset of the processing components at regular intervals. The length of this interval is called the *periodicity*, and it may be controlled by the program. In this sense, the BSP model may be seen as a compromise between asynchronous and synchronous operation. Phillip B. Gibbons has expressed sim-

---

16 An example from another field of computer science is the success of the relational model in the database world. This may be explained by the fact that the relational model has acted as a bridge between users of database systems and implementors. When proposed, the model was simple and general, high level and “advanced”. Database system designers have worked hard for a long time to be able to provide efficient implementations of the relational model. (This example was pointed out to me by P. Thanisch [Tha90].)
ilar thoughts and use the term semi-synchronous for this kind of operation [Gib89].

A computer based on the BSP model may be programmed in many styles, but Valiant claims that a PRAM language would be ideal. The programs should have so-called parallel slackness, which means that they are written for \( v \) virtual processors run on \( p \) physical processors, \( v > p \). This is necessary for the compiler to be able to “massage” and assemble bulks of instructions executed as supersteps giving an overall efficient execution. Valiant’s BSP model is a totally new computing concept requiring drastically new designs of compilers and runtime systems.

The BSP model can be realised in many ways and Valiant outlines implementations using packet switching networks or optical crossbars as two alternatives [Val90].

2.1.4 Important Complexity Classes

There are three levels of problems. The simplest problem is what we meet in everyday life—solving a specific instance of a problem. The next level up is met by algorithm designers—making a general receipt for solving a (subset of) all possible instances of a problem. Above that we have a metatheoretic level where the whole structure of a class of problems are studied [Fre86a]. At this highly mathematical level one is interested in various kinds of relationships between a large number of complexity classes (see for instance Chapter 7 in Garey and Johnson [GJ79]).

A complexity class can be seen as a practical interface between these two topmost levels. The complexity theorist classifies various problems and also extends the general knowledge of the different complexity classes, while the algorithm designer may use this knowledge once parts of his practical problem have been classified.

2.1.4.1 \( P, NP \) and NP-completeness

The most important complexity classes for sequential computations are \( P, NP \) and the class of \( NP \)-complete problems (often abbreviated \( NPC \) [GJ79],[Har87]). It is common to define complexity classes in terms of Turing

\[17\]Reading this at the end of the work reported in this thesis was highly encouraging—most of the practical work has been the design and use of a prototype high level PRAM language, see Chapter 3.

\[18\]It seems to me that Cole’s parallel merge sort algorithm, described in Chapter 4, may be a good example on a program with parallel slackness.
Machines and language recognition ([GJ79, RS86]), but more informal definitions will suffice. In the discussion of “reasonable” parallelism at page 36 we will see that these complexity classes also are highly relevant for parallel computations.

**Definition 2.6 (The class P)**
The class \( P \) is the class of problems that can be solved on a sequential computer by a deterministic polynomial time algorithm. □

**Definition 2.7 (Polynomial time algorithm) ([GJ79] p. 6)**
A polynomial time algorithm is an algorithm whose time complexity function is \( O(f(n)) \) for some polynomial function \( f \), where \( n \) is the problem size. □

**Definition 2.8 (Exponential time algorithm) ([GJ79] p. 6)**
An exponential time algorithm is an algorithm whose time complexity function can not be bounded by a polynomial function. □

The word “deterministic” might have been omitted from Definition 2.6 since it corresponds to our standard interpretation of what an algorithm is. The motivation for including it becomes clear when we have defined the larger class \( NP \) which also includes problems that can not be solved by polynomial time algorithms.

**Definition 2.9 (The class NP)**
“Standard formulation”: The class \( NP \) is the class of problems that can be solved by a nondeterministic polynomial time algorithm.

“Practical formulation”: The class \( NP \) is the class of problems which has solutions that can be verified (to be a solution or not) by a deterministic polynomial time algorithm. □

Note that we avoided the term reasonable computer in the standard formulation above. This is because the concept *nondeterministic algorithm* intentionally contains a so called guessing stage [GJ79] which (informally) is assumed to guess the correct solution. This feature is magical [Har87] and makes the properties of the underlying machine model irrelevant.

The most celebrated complexity class is the class of \( NP\)-complete problems. Informally the class can be said to contain the hardest problems in \( NP \). Though it is not proved, it is generally believed that none of these problems can be solved by polynomial time algorithms. Today, close to 1000 problems are known to be \( NP\)-complete [Har87]. If you find an algorithm that
solves one of these problems in polynomial time—you have really done a
giant breakthrough in computer science. This should come clear from the
definition of the class $NP$.

**Definition 2.10 (NP-complete problems, (NPC))**

$NPC$ is the class of $NP$-complete problems. A problem $\Pi$ is $NP$-complete
if (i): $\Pi \in NP$, and (ii): For any other problem $\Pi'$ in $NP$ there exists a
polynomial transformation from $\Pi'$ to $\Pi$. □

A *polynomial transformation* from $A$ to $B$ is a (deterministic) polynomial
time algorithm for transforming (or reducing) any instance $I_A$ of $A$ to a

corresponding instance $I_B$ of $B$ so that the solution of $I_B$ gives the required
answer for $I_A$. As a consequence (see Lemma 2.1 at page 34 of [GJ79]) it can
be proved that a polynomial time algorithm for solving problem $B$ combined
with this polynomial transformation yields a polynomial time algorithm for

solving $A$. Thus Definition 2.10 states that a polynomial time algorithm for

a single $NP$-complete problem implies that all problems in $NP$ can be solved
in polynomial time. The relationship between the classes, $NP$, $P$ and the
class $NPC$ containing the $NP$-complete problems, is shown in Figure 2.2.

Reading part (ii) of Definition 2.10 one might think that much work is
required to prove that a problem is $NP$-complete. Fortunately, it is not
necessary to derive polynomial transformations from all other problems in

$NP$. This and other aspects of $NP$-completeness will become clear in Section

2.1.4.3 at page 37 where we describe $P$-completeness and its strong
similarity with $NP$-completeness.
2.1.4.2 Fast Parallel Algorithms and Nick's Class

With respect to parallelisation, the problems inside $P$ can be divided into two main groups—the class $NC$ and the class of $P$-complete problems. The class $NC$ contains problems which can be solved by "fast" parallel algorithms that use a "reasonable" number of processors.

**Definition 2.11 (Nick’s class, NC)**

Nick’s Class $NC$ is the class of problems that can be solved in polylogarithmic time (i.e., time complexity $O(\log^k n)$ where $k$ is a constant) with a polynomial number of processors (i.e., bounded by $O(f(n))$ for some polynomial function $f$, where $n$ is the problem size). □

$NC$ is an abbreviation for Nick’s Class, and is now the commonly used name for this complexity class, which was first identified and characterised by Nicholas Pippenger in 1979 [Pip79].

It is relatively easy to prove that a problem is in $NC$. Once one algorithm with $O(\log^{k_1} n)$ time consumption using no more than $O(n^{k_2})$ processors that solves the problem has been found, the membership in $NC$ is proved. Such an algorithm is often called a $NC$-algorithm. Thus Batcher’s $O(\log^2 n)$ time sorting network using $n/2$ processors (see Section 4.1.3) proves that sorting is in $NC$. Many other important problems are known to be in $NC$. Some examples are matrix multiplication, matrix inversion, finding the minimum spanning forest of a graph [Coo83], and the shortest path problem ([CM88] p. 107). The book *Efficient Parallel Algorithms* written by Gibbons and Rytter [GR88] provides detailed descriptions of a large number of $NC$-algorithms.

$NC$ is robust

The robustness of Nick’s Class is probably the main reason for its popularity. It is robust because it to a large extent is insensitive to differences between many models of parallel computation. $NC$ will contain the same problems whether we assume the EREW, CREW or CRCW PRAM model, or some other models such as uniform circuits [Coo85, And86, Har87, GR88]. The robustness of $NC$ allows us to ignore the polylogarithmic factors that separate the various models.

More refined complexity classes such as $NC^1$ and $NC^2$ may be defined within $NC$. $NC^1$ is used to denote the class of problems in $NC$ that can be

---

19 This is shown by various theoretical results describing how theoretical models may be simulated by more realistic models, see for instance [AHMP86, MV84, SV84, Up84].
solved in $O(\log n)$ time, whereas $NC^2$ is the problems solvable in $O(\log^2 n)$ time. These classes are not robust to changes in the underlying model and should therefore only be used together with a statement of the assumed model. Many other subclasses within $NC$ are described in Stephen Cook's article "A Taxonomy of Problems with Fast Parallel Algorithms" [Coo85].

"Reasonable" parallelism
Classifying a polynomial processor requirement as "reasonable" in general may need some explanation. Practitioners would for most problems say that a processor requirement that grows faster than the problem size might not be used in practice. However, in the context of complexity theory it is common practice to say that problems in $P$ can be solved in "reasonable" time as opposed to problems in $NPC$ that (currently) only can be solved in unreasonable (i.e. exponential) time. Similarly, we distinguish between reasonable (polynomially) and unreasonable (exponentially) processor requirements.

$NP$-complete problems can (at least in the theory)\textsuperscript{20} be solved in polynomial time if we allow an exponential number of processors "to remove the magical nondeterminism" (see [Har87] p. 271). However, if we restrict to a "reasonable" number of processors, the $NP$-complete problems remain outside $P$.\textsuperscript{21} In this sense the traditional complexity class $P$ is very robust. $P$ contains the same problems whether we assume one of the standard sequential computational models or parallel processing on a reasonable number of processors. A polynomial number of processors may be simulated with a polynomial slowdown on a single processor. This implies that all problems in $NC$ must be included in the class $P$.

$NC$ may be misleading
In theoretical computer science there has in the past years been published a large number of papers proving various (sub)problems to be in $NC$. These results are of course valuable contributions to parallel complexity theory. However, if you are searching for good and practical parallel algorithms, the titles of these papers may often be misleading if you do not know the terminology or do not understand the limitations of complexity theory. For

\textsuperscript{20}It is far from clear that it can be done in practice due to inherent limitations of three-dimensional space [Coo85, Har87, Fis88].

\textsuperscript{21}To see this, assume the opposite—a parallel algorithm using $p_1(n)$ processors that solves an $NP$-complete problem in $p_2(n)$ time, where $p_1(n)$ and $p_2(n)$ are both polynomials. Simulating this algorithm on a single processor would then give a polynomial time algorithm ($O(p_1(n) \times p_2(n))$, the problem must be in $P$ and we must have $NPC = P$. 

36
instance “Fast Parallel Algorithm for Solving . . .” may denote an algorithm with $O(\log^4 n)$ time consumption with very large complexity constants which makes it slower in practice than well known and simpler algorithms for the same problem. Similarly, “Efficient Parallel Algorithm for Solving . . .” may entitle a polylogarithmic time algorithm with $O(n^5)$ processor requirement resulting in a very high cost and a very low efficiency—if we use the definition of efficiency which is common in other parts of the parallel processing community (see page 47). It is therefore possibly more appropriate to describe problems in $NC$ as “highly parallelisable” [GR88]. Further, the term $NC$-algorithm is more precise and may often be less misleading than to say that an algorithm is fast or efficient.

It has been argued that too much focus has been put on developing $NC$-algorithms [Kar89]. The search for algorithms with polylogarithmic time complexity has produced a lot of algorithms with a large (but polynomially bounded) number of processors. These algorithms typically have a very high cost compared with sequential algorithms for the same problems—and consequently low efficiency. Richard Karp has therefore proposed to define an algorithm to be efficient if its cost is within a polylogarithmic factor of the cost of the best sequential algorithm for the same problem [Kar89]. See also the recent work by Snir and Kruskal [Kru89].

2.1.4.3 Inherently Serial Problems and P-Completeness

P-completeness

Consider the problems in $P$. The subset of these which are in $NC$ may be regarded as easy to parallelise, and those outside $NC$ as difficult or hard to parallelise. Proving that a problem is outside $NC$ (i.e. proving a lower bound) is however very difficult.

An easier approach is to show problems to be as least as difficult to parallelise as other problems. The notion of $P$-completeness helps us in doing this. Just as the $NP$-complete problems are those inside $NP$ which are hardest to solve in polynomial time, the class of $P$-complete problems are those inside $P$ which are hardest to solve in polylogarithmic time using only a polynomial number of processors.\textsuperscript{22}

\textbf{Definition 2.12 (P-complete problems, PC)} ([GR88] p. 235)

$PC$ is the class of $P$-complete problems. A problem $\Pi$ is $P$-complete if:

\textsuperscript{22}Note that some early papers on $NP$-completeness used the term $P$-complete for those problems that are common today to denote as $NP$-complete, see for instance [SG76].
$\Pi \in P$, and ii): For any other problem $\Pi'$ in $P$ there exists a $NC$-reduction from $\Pi'$ to $\Pi$. $\square$

**Definition 2.13 (NC-reduction)** ([GR88] p. 235)

A $NC$-reduction from $A$ to $B$ (written as $A \preceq_{NC} B$) is a (deterministic) polylogarithmic time bounded algorithm with polynomially bounded processor requirement for transforming (or reducing) any instance $I_A$ of $A$ to a corresponding instance $I_B$ of $B$ so that the solution of $I_B$ gives the required answer for $I_A$. $\square$

Consequently, if one single $P$-complete problem can be solved in polylogarithmic time on a polynomial number of processors—then all problems in $P$ can be solved within the same complexity bounds using the $NC$-reduction required by the definition.

Just as there is no known proof that $P \neq NP$, nobody has been able to prove that $NC \neq P$. However, most researchers believe that the hardest parallelisable problems in $P$ (the $P$-complete problems) are outside $NC$. A $P$-completeness result has therefore the same practical effect as a lower bound—discouraging efforts for finding $NC$-algorithms for solving the problem. The relationship between the classes, $P$, $NC$ and the class $PC$ containing the $P$-complete problems, is shown in Figure 2.3. The figure also shows the three main complexity classes for sequential computations.

**NC-reductions**

A $NC$-reduction is technically different from the transformation used in classical definitions of $P$-completeness [GR88]. As done by Gibbons and Rytter
[GR88] we will use the notion of NC-reductions to avoid introducing additional details about space complexity. Classical definitions use so-called logarithmic space reductions, and P-complete problems are often termed as "log-space complete for P". However, by using the so-called parallel computation thesis it can be shown that a log-space reduction corresponds to a NC-reduction, and they will be used as synonyms in the rest of the text.

A NC-reduction may informally be perceived as a "fast" problem reduction executable on a "reasonable" parallel model. It is interesting to note that Gibbons and Rytter [GR88] have observed that NC-reductions typically work very locally. For example an instance $G$ of a graph problem may often be transformed (reduced) to a corresponding instance $G'$ of another graph problem by transforming the local neighbourhood of every node in $G$. This is intuitively not surprising. There exists many examples where the ability to do computations based on local information (instead of global, central resources) significantly improves the possibilities of achieving highly parallel implementations.

A very important property of the "NC-reducability" relation, denoted $\preceq_{NC}$, is that it is transitive [Par87, And86]. As described below, this property is central in reducing the efforts needed for proving a problem to be P-complete.

**P-completeness proofs**

Part ii) of Definition 2.12 of P-complete problems may make us believe that the work needed to prove a new problem to be P-complete is substantial since we must provide a NC-reduction from every problem known to be in P. Fortunately, this is not true due to the following observation.

**Observation 2.1** (Corollary 5.2.3 in [Par87])

If $A$ is P-complete, $B \in P$, and $A \preceq_{NC} B$ then $B$ is P-complete.

**Proof:** We want to show that $B$ is P-complete. Since $B \in P$, what remains to show is that for every problem $\Pi \in P$ there exists a NC-reduction from $\Pi$ to $B$, i.e. $\Pi \preceq_{NC} B$. Since $A$ is known to be P-complete Definition 2.12 implies $\Pi \preceq_{NC} A$. The assumption $A \preceq_{NC} B$ and the transitivity of $\preceq_{NC}$ then imply that $\Pi \preceq_{NC} B$. (From the proof of Lemma 2.3 in Garey and

---

23 A *log-space reduction* is a transformation in the same sense as the polynomial transformation described at page 34 but with a space consumption which is bounded by $O(\log n)$ [GJ79, And86].

24 This result informally states that sequential memory space is equivalent to parallel time up to polynomial differences [Har87, And86, Par87].
Observation 2.1 tells us that we may use the following more practical approach to prove that a problem $\Pi$ is $P$-complete.

1. Show that $\Pi \in P$.

2. Show that there exists a $NC$-reduction from some known $P$-complete problem to $\Pi$.

However, we still have the "chicken before egg problem"—we simply cannot use this approach to prove some first problem to be $P$-complete!

The first problem shown to be $P$-complete was the PATH problem\textsuperscript{25} presented by Stephen Cook in 1973 [Coo74]. Shortly after, Jones and Laaser [JL77] showed six other problems to be $P$-complete. Cook proved the $P$-completeness of the PATH problem by describing a generic log-space reduction of an arbitrary problem in $P$ to the PATH problem. Similarly, Jones and Laaser provided a generic reduction for the UNIT RESOLUTION problem [JL77].

Informally, such a generic reduction (to e.g. the PATH problem) is a description of how an arbitrary deterministic polynomial time computation on a Turing machine may be simulated by solving a corresponding instance of the PATH problem. The generic reduction to the PATH problem invented by Cook is an extremely important result in the history of (parallel) complexity theory. This, and other similar generic reductions are complicated mathematical descriptions— which we do not need to know in detail to use the $P$-completeness theory in practice.

In addition to the original papers ([Coo74, JL77]) interested readers are referred to Chapter 7 in [GR88] which describes a generic reduction to the GENERABILITY problem. See also the very good description of Stephen Cook's seminal theorem (Cook's Theorem) found in Section 2.6 of [GJ79]. This theorem provided the first $NP$-complete problem (SATISFIABILITY) by a generic reduction from an arbitrary problem in $NP$.

Part 1 of the proof procedure, to show that the problem is in $P$, is often a straightforward task. All we need is to show the existence of some deterministic polynomial time uni-processor algorithm solving the problem.

\textsuperscript{25} Called PATH SYSTEM ACCESSIBILITY by Garey and Johnson [GJ79] p. 179. The problem is also described in [JL77].
One exception is "linear programming" which was shown to be in P by Khachian as late as in 1979 [Kha79].

To find a NC-reduction from some known P-complete problem is in general not so easy. A good way to start is to study documented P-completeness proofs of related problems. Chapter 3 in the book by Garey and Johnson [GJ79] contains a lot of fascinating problem transformations, and it is highly recommended—in spite of the fact that the problem transformations reported there are polynomial reductions and not necessarily NC-reductions. However, Anderson reports that most of the transformations used in NP-completeness proofs can also be performed as NC-reductions [And86].

What is difficult is to select a well suited known P-complete problem and to find a systematic way of mapping instances from that problem into instances of the problem we want to prove P-complete. Once the reduction has been found, it is often relatively easy to see that it can be performed by a NC-algorithm. That part is therefore omitted from most P-completeness proofs.

The Circuit Value Problem
Since the pioneering work by Cook in 1971 we have got an increasing number of known P-complete problems. The most frequently used problem in P-completeness proofs is the circuit value problem (CVP) and its variants [And86]. The circuit value problem was shown to be P-complete by Ladner in 1975 [Lad75].

Informally, an instance of the circuit value problem is a combinational circuit (i.e., a circuit without feedback loops) built from Boolean two-input gates and an assignment to its inputs. To solve the problem means to compute its output [Par87]. The kind of gates allowed in CVP plays an important role. CVP is P-complete provided that the gates form a so-called complete basis [Par87, And86]. The most common set of gates is {AND, OR, NOT}. Two important variants of CVP are the monotone circuit value problem (MCVP) where we are restricted to use only AND or OR gates and the planar circuit value problem (PCVP) where the circuit can be constructed (on the plane) without wire crossings. MCVP and PCVP were both proved P-complete by Goldschlager [Gol77]. Other variants of CVP are described in [GSS82, And86, Par87]. Note that small changes in a problem definition can make dramatic effects on the possibility of a fast parallel solution. As an example, both MCVP and CVP are P-complete, but the monotone and planar CVP can be solved by a NC-algorithm.
**P-complete problems—Examples**

The set of problems proved and documented to be $P$-complete contains a growing set of interesting problems. Below is a short list containing some of the more important of these. The reader should consult the references for more detailed descriptions of the problems and their proofs (see also [GR88]).

*Maximum network flow* proved $P$-complete by Goldschlager et. al. in 1982 [GSS82].

*Linear programming* proved as member of $P$ by Khachian [Kha79] and proved as $P$-complete or harder\(^{20}\) by Dobkin et. al. in 1979 [DLR79].

*General deadlock detection* proved $P$-complete by Spirakis in 1986 [Spi86].

*Depth first search* proved $P$-complete by John Reif in 1985 [Rei85].\(^{27}\)

*Unification* proved $P$-complete by Dwork, Kanellakis and Mitchell 1984 [DKM84]. Kanellakis describes in [Kan87] how this result depends on the representation of the input.

*Unit resolution for propositional formulas* proved $P$-complete (among several other problems) by Jones and Laaser in 1977 [JL77].

*Two player game* proved $P$-complete by Jones and Laaser in 1977 [JL77].

A $P$-complete problem is often termed as *inherently serial*—it often contains a subproblem or a constraint which requires that any solution method must follow some sort of serial strategy. For some problems such as “two player game” (and perhaps also “general deadlock detection”) it is relatively easy to intuitively see the inherent sequentiality. For others such as “maximum network flow” it is much harder.

In general it is difficult to claim that a problem is inherently serial because it in our context means that all possible algorithms for solving it are inherently serial. On the contrary, it is often easier to identify inherently serial algorithms. This is captured in the notion of $P$-complete algorithms discussed in the following paragraph.

\(^{20}\)This is often termed $P$-hard. Informally it means that the problem may be outside $P$, but if it can be proved as member of $P$ then $P$-hardness implies $P$-completeness.

\(^{27}\)It is more correct to term depth first search as a $P$-complete algorithm, see page 43.

42
**P-complete algorithms**

A vast amount of research has been done on sequential algorithms. When designing a parallel algorithm for a specific problem it is often fruitful to look at the best known sequential algorithms. Many sequential algorithms have fairly direct parallel counterparts.

Unfortunately, many of the most successful sequential algorithms are very difficult to translate (transform) into very fast parallel algorithms. Such algorithms may be termed inherently serial. They are often using a strategy which is basing decisions on accumulated information. A good example is J. B. Kruskal's minimum spanning tree algorithm [Kru56] which is a typical greedy algorithm (see for instance [AHU82] pages 321–324). Greedy algorithms are in general sequential in nature—they typically build up a solution set item by item, and the choice of which item to add depends on the previous choices.

In his thesis *The Complexity of Parallel Algorithms* [And86] Richard Anderson shows that greedy algorithms for several problems are inherently serial—he proves them to be P-complete. Anderson gives the following definition of a P-complete algorithm [And86]:

An algorithm \( A \) for a search problem is \( P \)-complete if the problem of computing the solution found by \( A \) is \( P \)-complete.

Another example of an inherently serial algorithm is depth-first search (DFS) ([Tar72]) which is used as a subalgorithm in many efficient sequential algorithms for graph problems. John Reif has shown that the DFS algorithm is \( P \)-complete by proving that the DFS-ORDER problem\(^{28}\) is \( P \)-complete [Rei85].

Fortunately, the fundamental difference between problems and algorithms makes the detection of an algorithm to be \( P \)-complete to a less discouraging result. A \( P \)-complete algorithm does not say that the corresponding problem is inherently serial—it merely states that a different approach than parallelising the sequential algorithm must be attempted to possibly obtain a fast way to solve the problem with parallelism. For example, the natural greedy algorithm for solving the maximal independent set problem is \( P \)-complete, but a different approach can be used to solve the problem with a NC algorithm [And86, GS89].

---

\(^{28}\) Informally described, this problem is to find the depth-first search visiting order of the nodes in a directed graph.
Table 2.2: Summary of complexity theory for sequential and parallel computations.

<table>
<thead>
<tr>
<th></th>
<th>sequential computations</th>
<th>parallel computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>The main class of problems studied:</td>
<td>$NP$</td>
<td>$P$</td>
</tr>
<tr>
<td>The “easiest” problems in the main class, i.e. the class of problems that may be solved “efficiently”:</td>
<td>The class $P$</td>
<td>The class $NC$</td>
</tr>
<tr>
<td>The fundamental question:</td>
<td>Is $P = NP$?</td>
<td>Is $NC = P$?</td>
</tr>
<tr>
<td>Definition of a problem that may be solved “efficiently”:</td>
<td>The existence of a sequential algorithm solving the problem in polynomial time</td>
<td>The existence of a parallel algorithm solving the problem in polylogarithmic time using a polynomial number of processors</td>
</tr>
<tr>
<td>The “hardest” problems in the main class, i.e. problems that probably can not be solved “efficiently”:</td>
<td>The $NP$-complete problems</td>
<td>The $P$-complete problems</td>
</tr>
<tr>
<td>A “first” problem in the hardest class:</td>
<td><strong>SATISFIABILITY (SAT)</strong></td>
<td><strong>PATH</strong></td>
</tr>
<tr>
<td>Technique for proving membership in the “hardest” class:</td>
<td>$P$-reducibility</td>
<td>$NC$-reducibility</td>
</tr>
</tbody>
</table>
NP-completeness vs. P-completeness
Table 2.2 shows the similarities of P-completeness and NP-completeness. It does also give a quick summary of the theory.

2.2 Evaluating Parallel Algorithms

2.2.1 Basic Performance Metrics

In the rest of this thesis it is assumed that algorithms are executed on the CREW PRAM model. This assumption makes it possible to define the basic performance metrics for evaluating parallel algorithms.

Time, Processors and Space

Definition 2.14 (Time)
The time, running time, or execution time for an algorithm executed on a CREW PRAM is the number of CREW PRAM clock periods elapsed from the first to the last CREW PRAM instruction used to solve the given instance of a problem. The time is expressed in CREW PRAM time units or simply time units.

This definition corresponds to what many authors call parallel (running) time [Akl85]. A large number of processors may perform the same or different CREW PRAM instructions in one CREW PRAM clock period. Definition 2.14 makes it possible to use the same concept of time for parallel algorithms and serial algorithms executed as uni-processor CREW PRAM programs. Note that the definition implies that time used by an algorithm strongly depends on the size and possibly also the characteristics of the actual problem instance.

On models which reflect parallel architectures based on message passing it is common to differentiate between computational steps and routing steps (see for instance [Akl85, Akl89]). As described in Chapter 3, access to the global memory from the processors in a CREW PRAM is done by one single

---

29 More precisely, a CREW PRAM implementation of an algorithm.
30 Each processor is assumed to have a rather simple and small instruction set. (Nearly all instructions use one time unit, some few (such as divide) use more. The instruction set time requirement is defined as parameters in the simulator—and therefore easy to change. See Appendix A.)