1

Introduction

In this chapter we provide an intuitive introduction to the topic of approximability and parallel computation. The method of approximation is one of the well established ways of coping with computationally hard optimization problems. Many important problems are known to be NP-hard, therefore assuming the plausible hypothesis that P≠NP, it would be impossible to obtain polynomial time algorithms to solve these problems.

In Chapter 2, we will give a formal definition of optimization problem, and a formal introduction to the topics of PRAM computation and approximability. For the purpose of this chapter, in an optimization problem the goal is to find a solution that maximizes or minimizes an objective function subjected to some constrains. Let us recall that in general to study the NP-completeness of an optimization problem, we consider its decision version. The decision version of many optimization problems is NP-complete, while the optimization version is NP-hard (see for example the book by Garey and Johnson [GJ79]). To refresh the above concepts, let us consider the Maximum Cut problem (MAXCUT).

Given a graph G with a set V of n vertices and a set E of edges, the MAXCUT problem asks for a partition of V into two disjoint sets V_1 and V_2 that maximizes the number of edges crossing between V_1 and V_2. From now on through all the manuscript, all graphs have a finite number of vertices n. The foregoing statement of the MAXCUT problem is the optimization version and it is known to be NP-hard [GJ79]. The decision version of MAXCUT has as instance a graph $G = (V, E)$ and a bound $k \in \mathbb{Z}^+$, and the problem is to find the partition V_1 and V_2 such that the number of edges crossing the two sets of vertices is greater than or equal to k. The decision version of MAXCUT is NP-complete [GJS76]. The MAXCUT is a problem of great practical importance in the design of interconnexion networks and in statistical physics (see for example [BGJR88]).

Decision problems are easy to encode as languages over a finite alphabet, where for each problem the language is the set of instances having answer *yes*. Given an instance of a decision problem, to test if it has a solution is equivalent to deciding if the instance belongs to the language. The theory of NP-completeness was developed at the beginning of the 70s, in terms of decision problems ([Coo71], [Lev73], [Kar72]). The idea of approximate difficult problems is previous to the development of the theory of NP-completeness [Gra66]. Johnson in a seminal paper [Joh74] did a systematic treatment to approximate in polynomial time the solution to many of the optimization versions in the original list of NP-complete problems, given by Karp [Kar72]. Since then, the topic has become a standard part in courses on algorithms (see for example [CLR89]). Intuitively, for $0 < \epsilon \leq 1$, an **ϵ-approximation** is an algorithm that outputs a value s, such that the optimal solution lies in the interval $[s\epsilon, s/\epsilon]$.

Concurrently with this development of approximability to hard problems, in order to gain speed and computer power, research in parallel architectures was flourishing. As a consequence, a lot of work was taking place developing the foundations of massive parallel computation and parallel algorithm design. The most popular theoretical model of parallel computation that has been used is the **Parallel Random Access Machine** (PRAM), introduced by Fortune and Wyllie [FW78] and by Goldschlager [Gol78]. A PRAM consists of a number of sequential RAM processors, each with its own memory, working synchronously and communicating among themselves through a common shared memory. In one step, each processor can access one memory location (for reading or writting on it), or execute a single RAM operation. Although performing the same instructions, the processors can act on different data.

The simplicity of the PRAM model has led to its acceptance as the model used to identify the structural characteristics that allow us to exploit parallelism for solving the problem. However, it should be noted that the PRAM model hides levels of algorithmic and programming complexity concerning *reliability*, *synchronization*, *data locality* and *message passing*. Nevertheless, as we shall mention in Section 1.3, several techniques have been developed to simulate PRAM algorithms by more realistic models of parallel computation.

1.1 Sequential and Parallel Computation

The sequential model we use is the RAM machine, in which we measure **time** by number of steps and **space** by number of memory locations. As usual

by an **efficient** sequential algorithm we mean one that takes polynomial time using a RAM machine. The set of problems that can be solved by such algorithms constitute the class **P**. To familiarize the reader with the notation we use to describe algorithms, let us give a program for the Prefix-Sums problem. The input to the Prefix-Sums problem is a sequence $(x_1, \ldots, x_n)$, and the output is a sequence $(s_1, \ldots, s_n)$ where for each $1 \leq i \leq n$, $s_i = \sum_{j=1}^{i} x_j$. In Algorithm 1 we give the sequential code for the Prefix-Sums operation.

```
    PRESUMS (x[1 : n])
1   s[1] := x[1];
2   for i = 2 to n do
3       s[i] := s[i − 1] + x[i]
```

Algorithm 1: Sequential Prefix-Sums

The use of a parallel machine changes the parameters to measure efficiency. In sequential RAM models, the usual measure is time, and to a lesser extent space. In the parallel setting we have to measure the use of two resources, number of parallel steps (parallel time) and the maximum number of processors needed in any parallel step. By an **efficient** parallel algorithm we mean one that takes polylogarithmic time using a polynomial number of processors, and can be implemented on a PRAM machine. Problems that can be solved within these constraints are said to belong to the class **NC**. Thus problems in class NC are regarded as being solved in parallel, using a polylogarithmic number of parallel steps and using a number of processors that at most is polynomial, both measure functions in the size of the input to the problem. Through all this book, given a problem, we shall refer to an NC *algorithm* as a parallel algorithm for the problem that can be implemented with a PRAM using a polylogarithmic number of steps and a polynomial number of processors. The same abuse of nomenclature will be used with other parallel complexity classes, like RNC and ZNC.

We write parallel algorithms in an informal pseudolanguage. The description of an algorithm is given in sequences of macroinstructions. The main difference from sequential algorithms is the use of a new instruction **for all** ⟨condition⟩ **pardo**; all statements following this sentence are executed in parallel for all processors whose identifier satisfies the condition. Algo-

rithm 2 solves the Prefix-Sums problem in a PRAM, for sake of simplicity we have assumed that the input data is indexed from 0 to $n-1$.

```
    PPRESUMS (x[0 : n − 1])
1   for d = 1 to log n do
2       for all i mod 2^{d+1} = 0 and 0 ≤ i ≤ n pardo
3           x[i + 2^{d+1} − 1] := x[i + 2^d − 1] + x[i + 2^{d+1} − 1]
```

Algorithm 2: Parallel Prefix-Sums

In general we will use the product time $\times$ processors to derive bounds that allow us to compare the performance of sequential and parallel algorithms. Notice that a parallel step involving n processors can be performed by a single processor in time n. The above algorithm takes time $O(\log n)$ and uses $O(n)$ processors. Thus if we use a sequential machine to simulate the algorithm the simulation will take time $O(n \log n)$; therefore the parallel algorithm is not optimal.

An optimal parallel algorithm will be one such that the product $\times$ processors is equal to the optimal (best bound) sequential complexity. It is relatively easy to derive an optimal implementation of the above algorithm (see for example [JaJ92], [Rei93]), although we will not give it here. The Prefix-Sums operation has been analyzed for centuries as the recurrence $x_i = a_1 + x_{i-1}$. The first parallel circuit was suggested by Ofman [Ofm63].

1.2 Some Problems to Approximate

Let us begin with an easy example of a problem that can be approximated in parallel. The Maximum Satisfiability problem consists in, given a boolean formula F in conjunctive normal form, finding a truth assignment that satisfies the maximum number of clauses simultaneosuly. It is easy to derive an approximation algorithm for the Maximum Satisfiability problem. Note that the assignment $x_i = 1$ (for $1 \leq i \leq n$) satisfies all clauses with a positive literal and the assignment $x_i = 0$ (for $1 \leq i \leq n$) satisfies all the clauses with a negative literal. Therefore, given a boolean formula F with m clauses and with variables $x_1, \ldots, x_n$, taking from the two previous assignments the one satisfying most clauses, such an assignment satisfies at least $m/2$ clauses, and since there are m clauses, it is at least "half as good" as an optimal

assignment. Also it is easy to see that with a polynomial number of processors it can be checked in logarithmic time whether an assignment satisfies at least one half of the clauses, and therefore this provides a trivial parallel algorithm for approximating Maximum Satisfiability within a constant factor of the optimal solution.

Let us move to a more involved example, the Maximum Cut problem. In the following, we shall prove that there is a PRAM algorithm to approximate MAXCUT within a constant factor of the optimal solution. The proof uses the technique of *derandomization*. Our presentation is based on the paper by Luby [Lub86].

To clarify the proof, let us consider the 0-1 Labeling problem. Given a graph $G = (V, E)$, with n nodes, a **labeling** is an assignment of labels from the set $\{0, 1\}$ to the nodes of G. For a labeling $l : V \to \{0, 1\}$ define the **cost** of the labeling l by $X(l) = \sum_{\{u,v\} \in E} |l(u) - l(v)|$. The 0-1 Labeling problem consists in finding the labeling that maximizes the cost. Thus taking as partition the set of vertices with the same label the labeling problem is equivalent to the MAXCUT problem.

Notice that for a graph with n vertices, there are 2^n possible labelings of it. A naive aproach to solving the problem is a search in the whole space of labelings, as is done in Algorithm 3. Notice that we can represent a labeling by a binary string x taking values from 0^n to 1^n. The variable c computes the maximum cost of all labelings, so at the end the value of c will be the maximal cost, and the variable l will hold a labeling with maximal cost.

```
    MAXCUT (G)
1   n := |V|; c := 0; l := 2^n;
2   for x = 0 to 2^n − 1 do
3       d := 0;
4       for i = 1 to n do
5           for j = i + 1 to n do
6               if (i, j) ∈ E and x_i ≠ x_j
7                   then d := d + 1;
8       if d > c then
9           c := d; l := x
```

Algorithm 3: Solving MAXCUT by exhaustive search

Although Algorithm 3 takes exponential time, we can investigate further properties of the cost function. Given G, let Ω denote the set of all 2^n possible labelings. We can see the set Ω as a probability space, in which each labeling is a sample point, to be taken with probability $1/|\Omega|$, and the function X is interpreted as a random variable on Ω. Therefore one can ask about the average cost of a labeling. The expected value of X, denoted by $\mu[X]$, is defined as

$$\mu[X] = \sum_{l \in \Omega} X(l) \Pr\{l\}.$$

To find a bound for the expectation, let us consider an alternative scheme. For each node $u \in V$ define an indicator random variable l_u that takes values 0 and 1 with probability 1/2. Consider n independent random variables, one for each node $u \in V$. Notice that the *joint distribution* of the n independent random variables is the uniform distribution on the set Ω. Therefore we can express the expectation through labelings or through the outcome of the n random variables. The main difference is that now we have only to analyze the contribution of each edge to the final cost. A given edge contributes 0 when both labels are the same, but when they differ the contribution is 1. Therefore we can express the expectation as

$$\mu[X] = \sum_{e=(u,v) \in E} (1 \Pr\{l_u = l_v\} + 0 \Pr\{l_u \neq l_v\}). \tag{1.1}$$

Recall that l_u is selected from $\{0, 1\}$, with probability 1/2, and as the random variables are independent, we get

$$\begin{aligned} \Pr\{l_u = l_v\} &= \Pr\{l_u = 1 \text{ and } l_v = 1\} + \Pr\{l_u = 0 \text{ and } l_v = 0\} \\ &= \frac{1}{4} + \frac{1}{4} = \frac{1}{2}, \end{aligned}$$

and also we obtain the same value for the probability of being different. Thus, substituting in (1.1) we conclude that

$$\mu[X] = \sum_{e=(u,v) \in E} \frac{1}{2} = \frac{|E|}{2}.$$

This probabilistic result says that if an element of Ω is generated at random, then with high probability (greater than or equal to 1/2) we will have an element for which the cost function is greater than or equal to the expected value. In Algorithm 4 we present a simple schema that with high probability computes a labeling with a cost above the average.

This algorithm can be easily parallelized, just replace the sequential **for**

```
    RCUT (G)
1   n := |V|;
2   for i = 1 to n do
3       toss a fair coin to assign a value 0 or 1 to l[i]
```

Algorithm 4: Computing with high probability a cut at least average

```
    PRCUT (G)
1   n := |V|;
2   for all 1 ≤ i ≤ n pardo
3       toss a fair coin to assign a value 0 or 1 to l[i]
```

Algorithm 5: Computing with high probability a cut at least average in parallel

by a **forall** sentence, as is done in Algorithm 5. Thus we have a randomized parallel algorithm that in constant time, using n processors, with high probability produces a labeling with cost at least the average cost. It is not difficult to do a better analysis. The average cost is $|E|/2$ and the maximum cost is bounded above by the total number $|E|$ of edges in the graph. Thus if we could compute a value c' such that $|E|/2 \leq c' \leq |E|$ then we are sure that the optimum cost is in the interval $[c', 2\,c']$. Therefore, c' is a $\frac{1}{2}$-approximation to the optimal cut. Thus algorithms RCUT and PRCUT are randomized algorithms that with high probability produce a $\frac{1}{2}$-approximation to the optimal cut.

However, we want a deterministic algorithm; to get it, we use a general technique used to derandomize some algorithms. Nice introductions to derandomization techniques are given in Chapter 15 of the book of Alon, Spencer and Erdös [ASE92] and the Ph. Dissertation of Berger [Ber90]. The key observation is that in the expectation analysis we only make use of the fact that the labels of two elements are independent, and we did not need the independence of a larger set of labels. Therefore to have a probability space in which the analysis of the expectation is the same as before, we need a labeling space in which two properties hold: the first one, that the

probability of a node getting label 0 (1) is 1/2; the second one, that the labels of any two elements must be independent. This last property is usually called **pairwise independence**. The interesting point in this approach is that while a probability space with full independence has exponential size, it is possible to construct a probability space of polynomial size, with only pairwise independence. Clearly, if a random variable over the small space takes a value with positive probability, then some element of the space must be bigger than or equal to this value. Therefore by exhaustive search the small space can be used to derive a non-random algorithm.

In our case, the main idea to get the polynomial space of labeling is to start with a set of polynomial size, in which an element can be described with few bits, and then use some kind of *hash function* to determine the corresponding labeling. The notion of hash function or hash table appears when we want to address a small number of elements, stored in a huge array. Here the small set represents the set of keys, assigned to labelings. Furthermore, consider hash funcions as random variables, over the set of keys, K, with uniform distribution.

To be precise, take the set of keys K as the set of pairs (a, b) with $1 \leq a, b \leq n$, so K has n^2 points. Define the labeling associated to key (a, b) in the following way:

$$l_{a,b}(v) = \begin{cases} 0 & \text{if } a + b\,v \text{ is even,} \\ 1 & \text{otherwise.} \end{cases}$$

Fix a vertex $v \in V$; considering all possible elements in K, the corresponding labeling assigns label 1 to v in one half of the cases, and 0 in the other half. But that means that in the space of labelings addressed through K the probability that v gets label 0 (1) is 1/2. Therefore the first requirement if fulfilled.

Now fix two vertices u and v, and consider all elements in $K \times K$. For a given tuple $(a, b, c, d) \in K \times K$ the corresponding labelings assign the same label to both vertices when $x = a + b\,u + c + d\,v$ is even, and different labels when x is odd. A straightforward computation gives that for half of the elements in $K \times K$ the corresponding x value will be even and for the other half it will be odd, therefore the probability that two labelings agree (or disagree) on two nodes is 1/2. Thus the analysis of $\mu[X]$ can be done, in the space of labelings obtained through K, and the new expected value is still bounded by $|E|$.

This property guarantees that if we search through all points in the small space and take the one with maximum cost, we get a labeling that has cost bigger than or equal to the average cost. Therefore, performing an

exahustive search, we can obtain a $\frac{1}{2}$-approximation to the problem. The foregoing argument implies that Algorithm 6 computes an approximation to the MAXCUT on a given graph $G = (V, E)$. The labeling corresponding to the final values of $a1, b1$ is a $\frac{1}{2}$-approximation to the MAXCUT on G. Moreover, the complexity of the algorithm is $O(1)$ parallel steps and it uses $O(n^4)$ processors.

```
    MAXCUT (G)
1   n := |V|; c := 0; a1 := 0; b1 := 0;
2   for all 1 ≤ a, b ≤ n pardo
3       for all 1 ≤ i, j ≤ n pardo
4           if (i, j) ∈ E and a + b i + a + b j is odd
5               then l[⟨a, b⟩, ⟨i, j⟩] := 1;
6       PPRESUMS(l[⟨a, b⟩]);
7   for d = 1 to 2 log n do
8       for all i mod 2^{d+1} and 0 ≤ i ≤ n pardo
9           l[i + 2^{d+1} − 1, 2n] :=
                max(l[i + 2^d − 1, 2n], l[i + 2^{d+1} − 1, 2n]);
10      c := l[2n, 2n]/2
```

Algorithm 6: Approximating MAXCUT in parallel.

Algorithm 6 illustrates one of the canonical techniques to obtain approximations with NC algorithms. In the following chapters, we will describe different techniques to approximate in parallel difficult problems. We will define different parallel approximation complexity classes, and study the consequences of a problem being in one of those classes, as well as the relationship among the classes.

1.3 Realistic Models

An important matter for the reader could be the use of the PRAM machine as our model of parallel computation. At the beginning of the chapter, we already indicated that the PRAM model provides a robust framework for the development and analysis of parallel algorithms, as it is a formal model to measure the amount of inherent parallelism in a given problem. Quite a bit of work has been done in studying whether a PRAM algorithm can be efficiently implemented on a "realistic" parallel machine, by realistic meaning

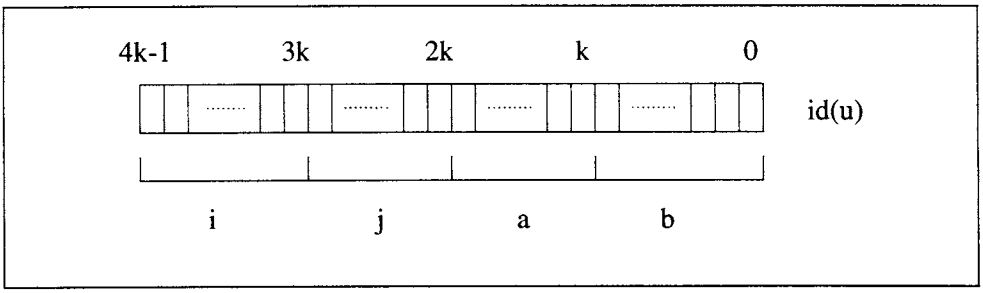

Fig. 1.1: Interpretation of the identifier of a node

a machine that takes into account factors such as the data locality, the synchronization or the cost of message passing. This last issue is one of the most relevant points in the simulation, and basically depends on the **bisection width** or **bandwidth** of the underlying topology connecting the processors in the realistic machine. The bandwidth is the maximum number of messages per instruction that can pass between one half of the processors and the other half. In general, while the PRAM model has unlimited bandwidth, most of the real machines have limited bandwidth, the exception being the hypercube (see for example the book of Leighton for a deep study of synchronous networks of processors [Lei93]). It should be noticed that there are efforts towards building machines with very high bandwidth, like the TERA machine [ACC90] and the PRAM being built at Saarbrücken [ADK+93].

In general a PRAM algorithm could be implemented on a distributed memory machine with not too much loss of efficiency, depending on the number of processors used and the size of the data. Most of the simulations will take a polylogarithmic cost in the number of parallel steps (see for example the survey papers by McColl [McC93] and Ranade [Ran96]).

Let us show how to implement the algorithm described for the Maximum Cut problem, on a real machine, the hypercube. A q-dimensional hypercube is formed by 2^q synchronous processors, each processor with a different identifier of q bits, a processor is connected to all processors at Hamming distance one. Processors may have the computing capacity we wish; from being workstations to processors with a few bits of memory.

Assume that $n = 2^k$ for some k, and consider a $4k$-dimensional hypercube, notice that now each node has an identifier formed by $4k$ bits. This identifier $id(u)$ will be divided into 4 numbers of k bits each (see Figure 1.1). The first $2k$ bits correspond to the pair a, b of numbers used to define the hash function, and the other $2k$ bits to nodes i, j that represent a possible edge in the graph. For a given node u we will represent the bits of its identifier by $u_{4k-1}, \ldots, u_0$ and the corresponding pieces by i_u, j_u, a_u, b_u.