Chapter 1

Introduction

1.1 Introduction

This book is on parallel algorithms. The goal of the book is to present a unified treatment of a wide variety of algorithms. Linear programming, or integer programming, serves as a tool for providing insights into a large class of problems. The shortest path problem, the max-flow problem, the stable marriage problem, and the weighted bipartite matching problem can all be modeled and analyzed using linear programming techniques. Linear programming formulation provides additional insights into the problem such as notions of dual problems, certificates for optimality, and lower and upper bounds on the objective function. In this book, we present another general technique called *lattice-linear predicate detection* that can solve many problems. We use this method to solve the generalization of many fundamental problems in combinatorial optimization, including the stable marriage problem [GS62], the shortest path problem [Dij59], and the assignment problem [Mun57]. Due to the importance and applications of these problems, each one has been the subject of numerous books and thousands of papers. The classical algorithms to solve these problems are the Gale-Shapley algorithm [GS62] for the stable marriage problem, Dijkstra's algorithm [Dij59] for the shortest path problem, and Kuhn's Hungarian method [Mun57] to solve the assignment problem (or equivalently, Demange, Gale, Sotomayor auction-based algorithm [DGS86] for market clearing prices). Could there be a single efficient algorithm that solves all of these problems?

The book presents a technique that solves not only these problems but more *general* versions of each of the above problems. We seek the optimal solution for these problems that satisfy additional constraints modeled using a *lattice-linear* predicate [CG98]. When there are no additional constraints (i.e., when the set of constraints is empty), our approach reverts to addressing the classical versions of these problems.

Our technique requires the underlying search space to be viewed as a distributive lattice [Bir67, DP90]. Common to all these seemingly disparate combinatorial optimization problems is the structure of the *feasible* solution space. The set of all stable marriages, the set of all feasible rooted trees for the shortest path problem, and the set of all market clearing prices are all closed under the meet operation of the lattice. If the order is appropriately defined, then finding the optimal solution (the man-optimal stable marriage, the shortest path cost vector, the minimum market clearing price vector) is equivalent to finding the infimum of all feasible solutions in the lattice.

We note here that it is well-known that the set of stable marriage and the set of market clearing price

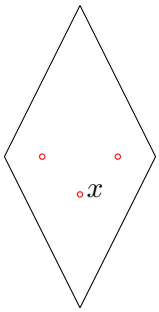

Figure 1.1: Search space with feasible solutions. The feasible solutions are shown in red. The solution x is the least feasible solution.

vectors form distributive lattices. The set of stable marriages forms a distributive lattice is given in [Knu97] where the result is attributed to Conway. The set of market clearing price vectors forms a distributive lattice is given in [SS71]. However, the algorithms to find the man-optimal stable marriage and the minimum market clearing price vectors are not derived from the lattice property. In our method, once the lattice-linearity of the feasible solution space is established, the algorithm to find the optimal solution falls out as a consequence. The reference [Gar20] derives the Gale-Shapley's algorithm, Dijkstra's algorithm and Demange-Gale-Sotomayor's algorithm from a single algorithm by exploiting the lattice property.

The lattice-linear predicate detection method to solve the combinatorial optimization problem is as follows. The first step is to define a lattice of vectors, L, such that each vector is *assigned* a point in the search space. For the stable marriage problem, the vector corresponds to the assignment of men to women (or equivalently, the choice number for each man). For the shortest path problem, the vector assigns a cost to each node. For the market clearing price problem, the vector assigns a price to each item. The comparison operation ($\leq$) is defined on the set of vectors such that the least vector, if feasible, is the extremal solution of interest. For example, in the stable marriage problem if each man orders women according to his preferences and every man is assigned the first woman in the list, then this solution is the man-optimal solution whenever the assignment is a matching and has no blocking pair. Similarly, in the shortest path problem and the minimum market clearing price problem, the zero vector would be optimal if it were feasible.

The second step in our method is to define a boolean predicate B that models feasibility of the vector. For the stable marriage problem, an assignment is feasible iff it is a matching and there is no blocking pair. For the shortest path problem, an assignment is feasible iff there exists a rooted spanning tree at the source vertex such that the cost of each vertex is greater than the cost of traversing the path in the rooted tree. For the minimum market clearing price problem, a price vector is feasible iff it is a market clearing price vector.

The third step is to show that the feasibility predicate is a lattice-linear predicate [CG98]. The lattice-linearity property allows one to search for a feasible solution efficiently. If any point in the search space is not feasible, it allows one to make progress towards the optimal feasible solution without any need for exploring multiple paths in the lattice. Moreover, multiple processes can make progress towards the feasible solution independently. In a finite distributive lattice, it is clear that the maximum number of such advancement steps before one finds the optimal solution or reaches the top element of the lattice is equal to the height of the lattice. Once this step is done, we get the following outcomes.

First, by applying the lattice-linear predicate detection algorithm to unconstrained problems, we get the Gale-Shapley algorithm for the stable marriage problem, Dijkstra's algorithm for the shortest path problem and Demange, Gale, Sotomayor's algorithm for the minimum market clearing price. In fact, the lattice-linear predicate detection method yields a parallel version of these algorithms and by restricting these to their sequential counterparts, we get these classical sequential algorithms.

Second, we get solutions for the constrained version of each of these problems, whenever the constraints are lattice-linear. We solve the *Constrained stable marriage Problem* where in addition to men's preferences and women's preferences, there may be a set of lattice-linear constraints. For example, we may require that Peter's regret [GI89] should be less than that of Paul, where the *regret* of a man in a matching is the choice number he is assigned. We note here that some special cases of the constrained stable marriage problems have been studied. Dias et al [DdFdFS03, CM16] study the stable marriage problem with restricted pairs. A restricted pair is either a *forced* pair which is required to be in the matching, or a *forbidden* pair which must not be in the matching. Both of these constraints are *lattice-linear* and therefore can be modeled in our system. The constrained shortest path problem asks for a rooted tree at the source node with the smallest cost at each vertex that satisfies additional constraints of the form "the cost of reaching node x is at least the cost of reaching node y", "the cost of reaching x must be equal to the cost of reaching y", and "the cost of reaching x must be within δ of the cost of reaching y". For the market clearing price problem, we consider constraints on the clearing prices of the form that item i must be priced at least as much as item j, or the difference in prices for item i and j must not exceed δ.

Third, by applying a constructive version of Birkhoff's theorem on finite distributive lattices [Bir67, DP90], we give an algorithm that outputs a succinct representation of all feasible solutions. In particular, the join-irreducible elements [DP90] of the feasible sublattice can be determined efficiently (in polynomial time). For the constrained stable marriage problem, we get a concise representation of all stable marriages that satisfy given constraints. Thus, our method yields a more general version of rotation posets [GI89] to represent all *constrained* stable marriages. Analogously, we get a concise representation of all constrained integral market clearing price vectors.

1.2 Computing Performance

Parallel algorithms are a crucial aspect of today's technology. At one time, computer processing power grew faster every year. However, this is no longer the case. Increased performance is now typically achieved by adding processors. Even smartphones are multi-core. There are two main styles of using multiple processors to accomplish computing goals: distributed and parallel computing systems.

Distributed computing systems contain multiple processors, with their own memory spaces, that are connected by a communication network. Typically, a single task is divided amongst individual computers which communicate and collaborate with each other by passing messages via the network.

On the other hand, parallel computing systems consist of multiple processors, referred to as processing elements (PE), which communicate using a shared memory space. This book focuses on algorithms that use and optimize parallel computing systems. The actual implementation of parallel computing systems is more detailed and complex than the diagram in Fig. 1.2. For instance, each PE typically has its own cache memory. We use the simplified model for this book.

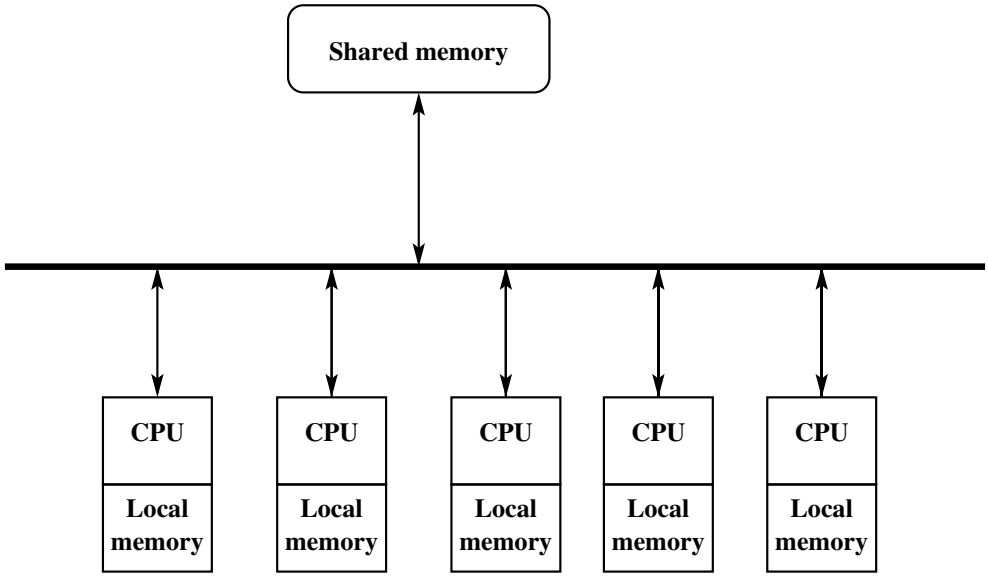

Figure 1.2: A parallel system

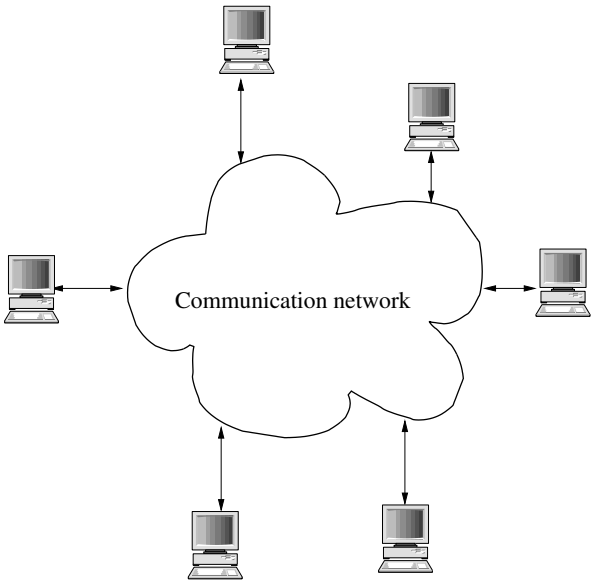

Figure 1.3: A distributed system

1.3 The Parallel Random Access Machine

When multiple processing elements (PE) are accessing the shared memory space, it is straightforward to understand when they are accessing different memory locations at each point in time. However, what happens when multiple PEs try to read and write to the same memory location?

To describe simultaneous memory access between two or more PE, we use the following letters to describe that access:

- **E** = Exclusive
- **C** = Concurrent
- **R** = Read
- **W** = Write

Parallel computing machines will be referred to as **PRAM** (Parallel Random Access Machine). PRAMs can be group into four main categories describing the model's behavior for simultaneous access to the same memory location:

- **EREW** (Exclusive Read Exclusive Write) - This is the most restrictive model in which only one PE can read or write to a memory location at a time.
- **CREW** (Concurrent Read Exclusive Write) - This is the most frequently used model in which multiple PEs can read a memory location at the same time, but only one can write at a time.
- **ERCW** (Exclusive Read Concurrent Write) - In this model, multiple PEs can write on a memory location but the read is exclusive. This model is generally not useful.
- **CRCW** (Concurrent Read Concurrent Write) - Multiple PEs can both read and write to a memory location simultaneously.

It is understandable in modern computing that multiple PEs reading from the same memory location simultaneously should not cause an issue, but concurrent writes could. Therefore concurrent writes are further defined as:

- **Common** - All values being written by the multiple PEs are all the same (common), and therefore allowed.
- **Priority** - A predetermined priority of the PEs is used to determine which value gets written.
- **Arbitrary** - A random value is picked to be written.

We use CREW and CRCW type PRAM models in this book.

1.4 Reduce

Let us now take a look at how parallel algorithms can be used to solve a simple problem. Consider an array A of size $n = 8$, $A = [8, 2, 9, 1, 3, 5, 11, 7]$. To find the maximum entry in the array A using a sequential algorithm, one starts at the beginning of the array and compares each entry keeping track of the maximum

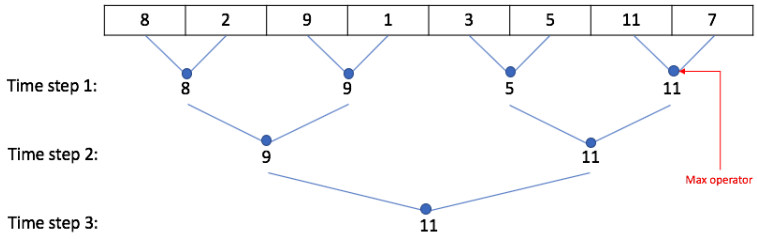

Figure 1.4: Computing the maximum of an array using the *Reduce* operation.

value along the way, until the end of the array is reached. The time complexity for this sequential algorithm is $O(n)$.

This problem can be solved in parallel by allowing each PE to compute a pairwise maximum value as shown in Fig. 1.4. In this example, the first time step entails four PEs computing the pairwise maximum on the values of the original array. The result is that the problem has now been reduced from eight elements in the array to only four elements. During the second time step, two PEs compute the pairwise maximum of the values computed in the previous step. Finally, the overall maximum value is calculated from the two values from the previous step. This algorithm is called the **Reduce**. The time complexity for this parallel algorithm is $O(\log n)$.

Notice that the *max* operator is associative, meaning that the order in which the operators are performed does not matter. It turns out that this same Reduce algorithm can be used with any operator that is associative including the minimum, the sum, the multiplication, the logical AND, and the logical OR operators. There are no concurrent writes in this algorithm and can be easily implemented on a CREW PRAM.

In addition to the time complexity, it is also useful to analyze the amount of work being done by our algorithms. The work complexity can be thought of as the *sequential* time complexity of the algorithm. In our example of computing the maximum of an array, we do $O(n)$ work even though the parallel time complexity is $O(\log n)$. Notice that, in comparison to the sequential algorithm, the *Reduce* algorithm reduces the time complexity (from $O(n)$ to $O(\log_2 n)$), and adds no additional work complexity. We define such a parallel algorithm as **work optimal**; its work complexity matches that of the best known sequential algorithm for the problem.

We now show a trick that can frequently be used for parallel algorithms. This trick cascades the sequential algorithm with the parallel algorithm as follows. We first consider the array of n divided into $n/\log_2 n$ subarrays, each of size $\log_2 n$. For each subarray of size $\log_2 n$, we use a core to find the maximum sequentially. We have used $n/\log_2 n$ cores and $O(\log_2 n)$ time to get the maximum of all subarrays. In the second step, we use the parallel algorithm to compute the maximum of $n/\log_2 n$ numbers. The second step requires at most $O(\log n)$ time. The key advantage of this approach is that we have used only $n/\log_2 n$ cores. The total time complexity of the *cascaded* algorithm is still $O(\log n)$.

1.5 Brent's Scheduling Principle

So far our simplified analysis of parallel algorithms has assumed that there are an unlimited number of PEs available for the parallel algorithm. However, this may not always be realistic, so it is useful to analyze

the time performance considering hardware limitations.

To accomplish this, in addition to n (the problem size), we introduce another variable p as the number of available PEs (or cores). We define $T(n)$ as the parallel time complexity of problem size n with any number of cores, $W(n)$ as the work complexity of n, and $T(n,p)$ as the bounded time complexity due to both problem size n and a certain number of available cores p. Brent's Scheduling Principle states:

$$T(n,p) = O(\frac{W(n)}{p} + T(n)) \tag{1.1}$$

This can be derived by summing the work performed at each parallel time step:

$$\begin{aligned} T(n,p) &= \sum_{i=1}^{T(n)} (\frac{W^i(n)}{p} + 1) \\ &= \frac{1}{p} \sum_{i=1}^{T(n)} W^i(n) + \sum_{i=1}^{T(n)} 1 \\ &= \frac{W(n)}{p} + T(n) \end{aligned} \tag{1.2}$$

Now let us apply this principle to the cascaded reduce algorithm. We observed that the time complexity $T(n) = O(\log_2 n)$, and work $W(n) = O(n)$. We used $p = n/\log_2 n$ cores, so let us apply Brent's Scheduling Principle using this number of cores:

$$\begin{aligned} T(n,p) &= O(\frac{W(n)}{p} + T(n)) \\ &= O(\frac{n}{n/\log_2 n} + \log_2 n) \\ &= O(\log_2 n) \end{aligned} \tag{1.3}$$

We see here that Brent's Scheduling principle gives the same answer as the time complexity previously computed when the number of cores is set to $n/\log_2 n$.

1.6 Parallel Complexity: Class NC

We now turn our attention to the parallel complexity of the problems. The class NC is the set of problems that can be solved *efficiently* on a parallel computer. By *efficiently* on a parallel computer, we mean that the parallel time it takes to solve a problem of size n is $O(\log^k(n))$ for a constant k. Observe that for the sequential complexity, we use *efficiently* to mean that the problem can be solved in time equal to $O(n^k)$ for a constant k. We require a significant speedup on a parallel computer for the problem to be in the class NC.

For the formal definition, we first define a *language* L to be a subset of strings over the alphabet $\Sigma = \{0,1\}$. The language-recognition problem is a binary decision problem that takes an arbitrary string s and decides whether the string belongs to L.

Definition 1.1 *A langauge L is in NC if there exists a PRAM algorithm that can decide for any input string s of size n whether s belongs to L in $O(\log^k n)$ time for a fixed k using $O(n^c)$ processors for some fixed c.*

Since a single processor can simulate behavior of multiple processors, it is clear that $NC \subseteq P$. Is the converse true? This is a major open problem in computation complexity. The current belief is that the converse is false and there are problems for which there do not exist any efficient parallel algorithms. In a manner analogous to P vs NP question, we define a subclass of problems in P, called *P-complete* that we think are the hardest to parallelize.

We first define the notion of *NC-reductions*. A language L_1 is *NC-Reducible* to L_2, if there exists an NC algorithm to transform an input string s_1 of L_1 to a string s_2 such that $s_1 \in L_1$ iff $s_2 \in L_2$. Clearly, if L_1 is *NC-reducible* to L_2, then any NC algorithm for L_2 can be converted to an NC algorithm for L_1. Hence, L_2 is at least as hard as L_1. It is clear that the notion of *NC-reducible* is transitive.

We can now define the notion of *P-completeness*.

Formally,

Definition 1.2 *A language L is P-complete if it is in P and every language in P is NC-reducible to L.*

It follows that if L is *P-complete*, then $L \in NC$ implies $NC = P$.

Hence, it is unlikely that there exists any NC algorithm for any *P-complete* problem (similar to the belief that it is unlikely that there exists any polynomial time algorithm for any NP-complete problem).

We now list some *P-complete* problems.

- *Circuit Value Problem (CVP)*: Given a boolean circuit consisting of $\neg$, $\vee$ and $\wedge$ gates, and the input x_1, x_2, x_n, determine if the value of the circuit is 1.
- *Ordered depth-first-search*: Given a directed graph and three vertices s, u and v, determine if u is visited before v when a depth-first-search is started at s.
- *Maxflow*: Given a directed graph with two distinguished vertices s and t and the maximum capacity of each edge, determine if the maxflow in the graph is odd.
- *Linear Inequalities*: Given an $n \times m$ matrix and a n-dimensional vector b, determine if there exists a vector x such that $Ax \leq b$.

A proof of *P-completeness* of these problems is available in [JáJ92].

1.7 Problems

1. Model the problem of finding a satisfying assignment of a boolean expression as searching for an element in an appropriate distributive lattice.
2. Model the problem of finding the cost of reaching a vertex from a given vertex in a weighted graph as a search for a feasible element in a distributive lattice.

1.8 Bibliographic Remarks

There are many books on sequential and parallel algorithms. For sequential algorithms, the reader is referred to [CLRS01, KT06, HSR01]. For parallel algorithms, the reader is referred to [JáJ92, LB00, SMDD19]