LEBANESE AMERICAN UNIVERSITY

Scalable Parallel Algorithms for Dynamic Programming on Tree
Decomposition

By

Mohamad M. Yassine

A thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

School of Arts and Sciences

August 2017

THESIS APPROVAL FORM

Student Name: Mohamad Yassine I.D. #: 201003973
Thesis Title: Scalable Parallel Algorithms for Dynamic
HS in Computer Science Programming on Tree Decomposition
Program: Computer Science
Department: Computer Science and Mathematics
School: Arts and Sciences

The undersigned certify that they have examined the final electronic copy of this thesis and approved it in Partial Fulfillment of the requirements for the degree of:

Masters in the major of Computer Science

Thesis Advisor's Name	<u>Faisal Abu Khan</u>	Signature		DATE: <u>21</u> / <u>8</u> / <u>2017</u> <small>Day Month Year</small>
Committee Member's Name	<u>Ramzi Haraty</u>	Signature		DATE: <u>21</u> / <u>8</u> / <u>2017</u> <small>Day Month Year</small>
Committee Member's Name	<u>Leila Issa</u>	Signature		DATE: <u>21</u> / <u>8</u> / <u>2017</u> <small>Day Month Year</small>

THESIS COPYRIGHT RELEASE FORM

LEBANESE AMERICAN UNIVERSITY NON-EXCLUSIVE DISTRIBUTION LICENSE

By signing and submitting this license, you (the author(s) or copyright owner) grants the Lebanese American University (LAU) the non-exclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic formats and in any medium, including but not limited to audio or video. You agree that LAU may, without changing the content, translate the submission to any medium or format for the purpose of preservation. You also agree that LAU may keep more than one copy of this submission for purposes of security, backup and preservation. You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant LAU the rights required by this license, and that such third-party owned material is clearly identified and acknowledged within the text or content of the submission. IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN LAU, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT. LAU will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Name: Mohamad Gassine

Signature:

Date: 21-8-2017

PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that:

1. I have read and understood LAU's Plagiarism Policy.
2. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.
3. This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Mohamad Yassine

Signature:

Date: 21-8-2017

ACKNOWLEDGMENT

This thesis would not have been possible without the support of many people.

Many thanks to my advisor, Dr. Faisal Abu-Khzam, for tracking my work and bearing with me. Also, thanks to my committee members, Dr. Ramzi Haraty and Dr. Leila Issa, who offered guidance and support.

And finally, thanks to my parents and my friends who endured this long process with me, always offering support and love especially Karim Jahed for his patience and great support.

Scalable Parallel Algorithms for Dynamic Programming on Tree Decomposition

Mohamad M. Yassine

ABSTRACT

Dynamic programming on tree decompositions is often a key for solving a wide-range of optimization problems that would otherwise be intractable. Operations on tree decompositions are amenable to parallelization. However, a systematic approach towards such parallelization is still lacking. In this work, we present a generic and flexible framework for parallel dynamic programming on tree decompositions that can be easily adapted to solve any graph theoretic optimization problem on graphs of bounded treewidth. We show the effectiveness of our framework using the Dominating Set problem as a case study. Our experiments show notable speedups compared to the serial approach.

Keywords: Parallel Processing, Tree Decomposition, Treewidth, Dynamic Programming, Dominating Set.

TABLE OF CONTENTS

Chapter	Page
I Introduction	1
II Preliminaries	5
III Dynamic Programming on Tree Decomposition	12
IV A Parallel Framework	20
4.1 Common Parallel Computing Techniques	20
4.2 Task Encoding	22
4.3 Initialization	27
4.4 Dynamic Load Balancing	31
4.5 Termination	36
V Experimental Analysis	40
5.1 Implementation	40
5.2 Testbed	41
5.3 Experimental Results	41
VI Concluding Remarks	45

LIST OF FIGURES

Figure	Page
1 An example of tree decomposition.	3
2 An example of a dominating set instance. The optimal solution is selecting vertices c and d	6
3 An example of tree decomposition with different treewidths.	9
4 An example of a nice tree decomposition.	11
5 Tables of dynamic programming on tree decomposition.	19
6 An example Adjacency Matrix and Adjacency List	23
7 A vertex array V in a tree node.	24
8 A coloring set S of V	24
9 An example of an unbalanced tree decomposition with 127 nodes. . .	25
10 A layered a tree decomposition	34
11 An example of an unbalanced tree decomposition	36
12 An example of a nice tree decomposition of figure 11	37
13 Running time for the sequential and parallel framework for unbalanced nice tree decomposition	44
14 Running time for the sequential and parallel framework for near balanced nice tree decomposition	44

LIST OF TABLES

Table	Page
1 Sequential and Parallel Running Times in seconds on unbalanced nice Tree Decomposition	43
2 Average waiting time in seconds to receive results for unbalanced nice Tree Decomposition	43
3 Sequential and Parallel Running Time in seconds of near balanced nice Tree Decomposition	43
4 Average waiting time in seconds to receive results of near balanced nice Tree Decomposition	43

Chapter One

Introduction

Most graph theoretic optimization problems are known to be computationally intractable, or *NP*-Hard. Such problems are unlikely to have polynomial-time algorithms. They are usually solved using exhaustive search, which leads to algorithms that have an exponential (or super-polynomial) running time depending on the size of the instance (graph). Although there are efficient and clever algorithms formulated for some of the problems to reduce the running time, their complexity will always remain super-polynomial as long as $P \neq NP$. For graph problems, this hardness depends also on the special class of graphs in question. A notoriously hard example is the DOMINATING SET problem, which finds application in a large variety of domains such as routing problems Wu, Cardei, Dai, and Yang (2006).

Parameterized complexity theory is a recent branch of computational complexity introduced by Downey & Fellows Downey and Fellows (2012). The main objective is to classify computationally intractable problems with respect to one or more input *parameters* rather than the size of input. For instance, in the dominating set problem, rather than finding the exact solution set, we restrict the solution to be of size k (and it has to be feasible as well). Other parameters can also be used such as graph genus, treewidth, etc... The computational complexity of many problems can be polynomial with respect to the input size while being super-polynomial with respect to the used parameter(s). Thus, some problems become tractable when certain parameters are fixed.

In such cases the problems are classified as fixed-parameter tractable (*FPT*) in the realm of parameterized complexity. Problems that are unlikely to be *FPT*, or hard for the class *FPT*, are classified as being $W[t]$ -hard for some $t \geq 1$ while problems that are not solvable in polynomial time when the parameter is a small fixed constant are para-NP hard.

Some optimization problems can be solved via the well known *Dynamic Programming* technique (DP). DP is used to divide a complex problem into sub-problems and store the optimal solution of each sub-problem to avoid recomputing it. DP has been used in old classical graph theoretic problems like, for example, obtaining the length of the shortest path between any two vertices in a graph. There are two well known DP algorithms for this problem: the Bellman–Ford algorithm (see Ekpanyapong, Waterwai, and Lim (2006)) and the Floyd–Warshall algorithm (see Wimmer and Lammich (2017)). The DP technique is mainly a space–time trade-off where storing the data in a table allows to speed up computations.

Parallel algorithms are adopted to speed up highly demanding computations when multiple machines are available. The most used technique is divide and conquer which consists of partitioning the input of a problem among the processing units, or cores, so that each core solves its assigned subproblem then solutions are combined to obtain a final answer. Another approach that has been used recently to design parallel exact algorithms for *NP*-hard problems consists of partitioning the search space of a problem instead of partitioning the input itself. This is especially useful for graph problems since the input-partitioning approach is often infeasible. In fact, the search space partitioning approach proved to be highly effective on many hard graph theoretic problems (see Abu-Khzam, Daudjee, Mouawad, and Nishimura (2015); Abu-Khzam, Langston, and Shanbhag (2004); Abu-Khzam, Langston, Shanbhag, and Symons (2006); Abu-Khzam and Mouawad (2012) for more information).

Restructuring the graph is another method used to speed up computations, and one of the widely used restructuring is tree decomposition (TD). It basically maps the graph

Figure 1: An example of tree decomposition.

into a tree structure which is later explored. An example of mapping a graph into a TD is illustrated in figure 1. With this technique, we further reduce the graph into subgraphs so that algorithms are efficient on each subgraph. A reason why TD is widely used is due to the effective use of dynamic programming by assigning a table to each tree node, whenever applicable. Dynamic programming on tree decomposition can be used for many combinatorial problems and have also been employed for problems from computational biology, constraint satisfaction, and probabilistic networks Bodlaender and Koster (2010).

Dynamic programming on tree decomposition is obviously amenable to parallelization, mainly because the parent-child computations can be performed in a rather independent manner. The thesis aims at developing a parallel framework for solving domination-like problems using the (generic) Dominating Set problem as a case study. Our framework, which focuses mainly on the dynamic programming on tree decomposition method, promises to be highly scalable in terms of number of computational units or cores.

Only two attempts have been made in the literature on parallelizing dynamic programming on tree decomposition. The first is done by Sullivan, Weerapurage, and Groër (2013), and their work primarily focuses on the maximum weighted independent set problem, and they used a parallel framework called MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation), which is used to manage dependencies and hide latency, global name spaces task distribution, non-process-centric computing, and dynamic load balancing and data redistribution Sullivan et al. (2013). The second is presented by Wang, Chen, Liu, and Hu (2013), and their work utilized map reduce instead of MPI, and only focuses on the maximum weighted independent set problem as well. We note that none of the two mentioned approaches develops a general framework for tackling graph problems.

The thesis is organized as follows. In chapter II, we introduce some graph theoretic terms and give some needed definitions. In chapter III, we overview dynamic programming on tree decomposition. In chapter IV, we present our framework. In chapter V, we present our testbed implementation and experimental results. Finally, we conclude this thesis in chapter VI.

Chapter Two

Preliminaries

Throughout this thesis, we adopt common graph theoretic notations and terminologies described in Diestel (2006). Let $G = (V, E)$ be finite, simple, and undirected graph with vertex set V and edge set E . We denote by $n = |V|$ the number of vertices in G and by $m = |E|$ the number of edges. If $(u, v) \in E$, u and v are said to be adjacent. We denote by $N_G(v)$ the neighborhood of vertex v , i.e. the set of all vertices adjacent to v . The degree of a vertex v , $deg_G(v)$, is defined as the number of edges incident to v and it is equivalent to $|N_G(v)|$ in simple loop-less graphs. We also assume that the graphs will be connected otherwise the technique we are using can be applied for each connected component of the graph. Let $H = (V', E')$ be a subgraph of G such that $V' \subseteq V$ and $E' \subseteq E$. H is said to be an induced subgraph of G if it is obtained (from G) via vertex deletion only. In other words every edge in G that connects two vertices from V' must belong to E' . A complete graph, or clique, is a connected graph where every pair of vertices are adjacent.

A dominating set S in G is a set of vertices satisfying $S \cup \{v \in V : N(v) \cap S \neq \emptyset\} = V$. In other words, every vertex of G is either in S or a neighbor of a vertex in S (i.e., dominated by S). Thus, the minimum Dominating Set problem aims at partitioning vertices into two groups: the dominating and the dominated, so that the dominating set is as small as possible. The need for domination appears in many domains such as social networks and wireless networks. A dominating set in a wireless network can be

considered as selecting a subset of wireless routers such that the routers reach all other wireless routers in the networks in order to minimize the communication overhead. In social networks such as Facebook, a minimum dominating set would be a small set of people with the largest number of friends so that one can use the dominating elements to spread news and positive influence.

Figure 2: An example of a dominating set instance. The optimal solution is selecting vertices c and d .

Dominating set has been extensively studied in the literature due to its substantial importance in social network and computer network applications. There are over 300 papers related to domination in graphs. It is so widely used that there were several variants of the problem, such as connected dominating set and global defensive alliance. Another closely related problem is the Independent Set problem where the required set has to cover all other vertices and the vertices in the set have to be totally disconnected (no edges between any pair of vertices in the independent set).

An obvious exact algorithm for Dominating set is brute force enumeration which takes time $O(2^n)$, but the current-best exact algorithm has running time in $O(1.5048^n)$ Rooij, Bodlaender, and Rossmanith (2009). If the input graph has treewidth k and a corresponding tree decomposition is given, then we can solve Dominating Set in $O^*(3^k)$.

In many real settings, the absolute minimum is of little importance. Rather, a certain

satisfactory dominating set size can be required. This number is often called the natural parameter. In fact, the (natural) parameterized version of dominating set is formally stated as follows: given a simple undirected graph G and a parameter k , is it possible to find a dominating set of size at most k in G ? The minimum k such that the graph G has an k -dominating set is called the domination number of G .

While solving NP -Hard problems in efficient time is highly unlikely, it turns out that the introduction of some extra parameters to the problem might render the problem tractable when the parameters are fixed. A Fixed-Parameter tractable (FPT) problem with a parameter k is a problem that is solvable in time that is exponential only with respect to its fixed parameter k and polynomial with respect to the input size. More formally, a problem is FPT if it has an algorithm with running time in $O(f(k)n^c)$ where c is a constant and f is an arbitrary function of k .

The notion of a tree decomposition was introduced by Robertson and Seymour in 1984. Roughly speaking, this is nothing but mapping of the vertices of a graph G into a tree T such that each vertex of the graph is mapped into a subtree of T . Moreover, every node of the tree contains an induced sub-graph of G . “Graph minors. III. Planar tree-width” (1984). We will label a tree decomposition as (T, Y) of a graph G where T is the tree decomposition and Y are the set of tree nodes: $Y = \{y_1, y_2, \dots, y_m\}$. The tree decomposition has the following characteristics:

1. $\bigcup_i y_i = V$
2. $\forall e = (u, v) \in E \Rightarrow \exists (u, v) \in y_i$
3. Intersection Property: $\forall y_i, y_j, y_k \in T$, if y_j is on a path from y_i to y_k in T , then $y_i \cap y_k \subseteq y_j$

To avoid any confusion, we will denote the term *vertex* to refer to an element in the graph G and the term *node* or *bag* to refer to an element in the tree decomposition T . It is worth noting here that, due to (2) and (3) above, the vertex set of every clique

in the graph G must be totally contained in a node of the corresponding tree in a tree decomposition of G .

The width of a tree decomposition is defined as $\max_{v \in T} |X_v| - 1$, and the *treewidth* of the graph G , or $tw(G)$, is the smallest width achievable over all possible tree decompositions of G . It is sometimes (informally) considered a measure of how tree-like a graph is or how accurately a given graph can be approximated by a tree. The Treewidth problem is simply posed as follows: given an undirected graph G , what is the treewidth of G , or: is it possible to compute a tree decomposition with the minimum width? This problem is *NP*-Hard Korach and Solel (1993) but falls in the class FPT when parameterized by the treewidth itself. Yet the Treewidth problem is known to be too difficult to solve due to the large super-exponential function of the parameter in the running time of the best-so-far algorithm, but some approximation strategy achieve satisfactory outcomes Bodlaender (1997). For several special graph classes, there exist polynomial time algorithms to determine the treewidth of graphs in the class, e.g. for chordal graphs, circular arc graphs, circle graphs, and distance hereditary graphs Bodlaender (1997). This exploit of k helped solve huge instances of many problems without falling prey to their exponential explosions Abscher et al. (2014). An example showing more than one tree decomposition of a graph is provided below in Figure 3.

Tree Decomposition is widely used because the treewidth of a graph could very well be small for some important graph classes such as outerplanar graphs. Small treewidth implies the existence of small separators in the given graph, which is a property that is plausible for divide and conquer algorithms, making some hard computational problems easier to solve exactly instead of resorting to approximation or heuristic techniques. Alas, using dynamic programming on tree decomposition is a memory hungry as we shall see when we present our approach in the next section.

Many approximation algorithms were developed to obtain the treewidth of a graph within a factor of 1.5 from the optimal on planar graphs, but it is difficult to calculate such factor on arbitrary graphs Bodlaender (1997). Nevertheless there are some heuris-

Figure 3: An example of tree decomposition with different treewidths.

tics that provide a “good” tree decomposition in polynomial time. Charwat (2015)

A *nice tree decomposition* can be constructed from any tree decomposition in polynomial time in approximately $O(|tw|^2|Y|)$ with the same tw of T . A nice tree decomposition is characterized by the following properties of its nodes:

1. A node that has no children (Leaf node).
2. A node whose bag is a superset of at least one of the bags of its children (Increase node).
3. A node whose bag is a subset of the union of the bags of its children (Reduce node).
4. A node such that its bag and its children are equal (Stable node).

A nice tree decomposition simplifies the tree and turns it into a rooted binary tree, but increases the height as well as the number of nodes (by approximately $O(|tw||Y|)$). This also simplifies the dynamic programming technique that we intend to use on tree decomposition since now there is a relationship between the parent and its children as shown in figure 4.

In general, using nice tree decompositions does not give additional algorithmic possibilities, but it considerably eases the design of algorithms, and one can also expect in several cases to have better constant factors in the running time of algorithms that use nice instead of normal tree decompositions Bodlaender (1997). Constructing a nice tree decomposition works well in our favor, at a cost which we will elaborate on in the next chapter. There can be multiple types of nice tree decompositions. Each author would add their own properties such as Bodlaender who adds all leaf nodes should be of size of 1. This makes it a very nice tree decomposition (see Bodlaender, Bon-sma, and Lokshtanov (2013)). The most common nice tree decomposition, and the one we are using in this thesis is due to T. Kloks Kloks (1994). He shows the following properties of nice tree decomposition:

1. Every node in T has at most two children making T a binary tree.
2. Y is a collection $\{y_i\}$ of subsets of the vertex set V of G such that:
 - (a) If a node y_i has children y_j and y_k , then $y_i = y_j = y_k$ (Join node).
 - (b) If a node y_i has a single child y_j , then either:
 - i. $|y_i| = |y_j| - 1$ and $y_i \subset y_j$ (*forget node*).
 - ii. $|y_i| = |y_j| + 1$ and $y_j \subset y_i$ (*introduce node*).
 - (c) If a node y_i has no children (*leaf node*).

We shall use the above notion of nice tree decomposition in the sequel to introduce the dynamic programming technique and our parallel DP framework.

Figure 4: An example of a nice tree decomposition.

Chapter Three

Dynamic Programming on Tree Decomposition

In this chapter we introduce a sequential dynamic programming on tree decomposition framework. We shall mainly focus on the following three main aspects of the computation:

Traversal. Where does the computation begin? For example, do we start from the root or from the leaves?

Operations. What are the main operations performed to compute a solution?

Validation. How does the computation framework validate solutions obtained by a tree node?

Dynamic Programming on tree decomposition is widely used to solve *NP*-hard problems in polynomial time on graphs of bounded treewidth. The general method works by constructing all possible solution sets at every tree node. Of course, since the treewidth is assumed to be bounded, the total number of solution sets is assumed to be small. For covering problems, we can generate all possible combinations for each bag in the tree decomposition. Then, each (generated) combination is a possible solution for the subgraph induced by the vertices in this bag. As a result, we will enumerate,

in each bag, $\sum_{k=0}^{|y_i|} \binom{|y_i|}{k} = 2^{|y_i|}$ subsets. We will store such data in a set (also reference as tables) denoted as S in the sequel. In terms of memory it is exponential as we store $\sum_{i=0}^{|T|} 2^{|y_i|}$ integers in memory. This is an issue if we wish to store this amount of memory on a single computer. In general the dynamic programming paradigm is popular for its efficiency that is mainly due to avoiding recomputing solutions to common subproblems Corneil and Keil (1987). Moreover, there have been some methods of decreasing the size of the sets done by Charwat Charwat (2015). However, even after parsing the combinations to filter out invalid solutions, we would still have a huge number of computations. This issue will be tackled in the next chapter.

There are two possible ways of approaching dynamic programming on tree decomposition. The “Top-down” approach works by starting the computation from the root and moving all the way to the leaves in a “post-order” traversal in order to recursively calculate partial solutions for each node in the tree, and again visit each node beginning from the root to the leaves in the same way to construct the final solution. On the other hand, the “Bottom-up” approach works by starting from the leaf nodes and moving upwards where the final solution will also be stored at the root. This approach can be handled in two ways, the first is a layered approach where the computation is done on the nodes of the same layer in a backward breadth first search fashion. This traversal adds a barrier for each level in the tree making the computation leveled. In our case, we used the “Top-Down” approach on nice tree decomposition. Since all nodes in T contain partial information of G , we will be computing partial solutions for each node and these solutions will be joined as we are progressing through the tree. At the root, all feasible solutions will be filtered and stored. Depending on the problem at hand, we will select the optimal solution for the problem given. It can be characterized as “computing tables of characterizations of partial solutions” for each node in a tree decomposition of bounded width Bodlaender (1997).

The complexity of traversing the tree is $O(n)$ since we will only have to visit each tree node once, and usually the complexity comes from the computation due to the

size of the solutions obtained as we progress through the tree. The calculations will become node specific at a cost of having more bags than the tree decomposition. A process might run out of memory if the treewidth is huge, and as the parent computes a solution with its children via the introduce or join node operations then the solution set will become larger in size.

The computation can be even simpler if we were solving the parameterized version of the domination problem because we can drop the number of combinations from any bag that are larger than the given parameter. Equivalently, we can enumerate solutions in the same manner.

Thus, our framework consists of the following three operations.

Generate We used the concept of coloring to denote the possible states of a vertex. We generate all possible colorings.

Filter This is the validation operation that will be fed the generated combinations and filter out the invalid combinations (those that are not feasible local solutions).

Reduce This is the computation that will be performed between the parent and the child, or depending on the tree node type.

We will be using the “Top-Down” approach, in other words a post-order walk utilizing the algorithm for Alber et al. Alber and Niedermeier (2002) which was shown to take $O(3^k|V|)$ time. We also consume $O(3^{|y_i|})$ memory since the generation of vertices is basically the permutation with allowed repetition. Since the nice tree decomposition provides a good structure of the tree, performing Dynamic Programming on it is made easier because it becomes node-specific. In the algorithm, we assume that a vertex will have one of the following three possible colors:

- Black: the vertex is in the solution.
- White: the vertex has a black neighbor (in the solution corresponding to this combination).

- Gray: no decision has been taken for the current vertex.

A valid solution would be a solution where all the white vertices are justified in the sense that every white vertex has at least one black neighbor in the solution. Two colorings are similar if for instance we have colorings A and B , then the coloring of common vertices found in both A and B should exactly have the same color. Similarly, a compatible coloring for a join node that has node A as the parent and nodes B and C as the children is where the colorings of the vertices found in B and C have exactly the same coloring in A . For example, if A colors vertex v black or gray, then B and C must assign the same color to v . If A colors a vertex v white, then either B and C should color v white, but none of them may color it black.

The sequential algorithm is shown in algorithm 1 below. The algorithm proceeds as follows for each tree node type while moving in a post order manner in the tree:

- Leaf: Generate all possible colorings for the current vertices
- Introduce: For each combination in the child node, we will attempt to color and add the extra vertex as shown in algorithm 5 and goes as follows:
 - Gray: We simply color the extra vertex gray and store it.
 - White: If it provides a valid solution, we will store it. Otherwise, we will discard it.
 - Black: We will color it black, update the cost, and store it.
- Forget: Generate all possible colorings for the node. For each combination in the node, we fetch similar colorings between its combinations and all the child's combinations such that the extra vertex is either colored black or white as shown in algorithm 4 and goes as follows:
 - If a black and a white combination is obtained, we will select the one with the minimum cost and store it.
 - Otherwise, if a black or white solution is obtained, we will store it.

- Join: Generate all possible colorings for the node. For each combination in the node, We fetch compatible colorings between its combinations and all the children's combinations and store the ones with the minimum cost as shown in algorithm 3.

The fact that the above algorithm adopts a dynamic programming approach stems from constructing partial solutions and storing them in a set of feasible solutions, stored at the parent. The set size will increase as we come across an introduce node since it is the only operation where we attempt to color the introduced vertex with the given colors.

Many strategies have been proposed for solving problems on graphs of bounded treewidth using a variant of the dynamic programming approach. For any such strategy, an algorithm for a given problem must describe the tables involved and also describe how tables are updated Telle and Proskurowski (1993).

Many enhancements to solving DS via DP on TD were made using a space-time trade-off. For instance, one known method is to keep track of the state, dominated or not, of each vertex in the graph and not just the ones in the node's bag. This helps selecting the best partial solution at each node, and avoids re-computations and gives a substantial speedup from $O(9^k n)$ to $O(4^k n)$ Alber and Niedermeier (2002). The two limiting factors are the amount of memory required to store the dynamic programming tables of partial results from numerous subproblems at multiple tree nodes simultaneously, and the total time required for the generation and subsequent node-by-node analysis of the tree decomposition Sullivan et al. (2013).

There are serial frameworks developed such as the D-FLAT system, which combines DP with answer-set programming (ASP). Here, the user specifies the DP algorithm in the form of an answer-set program which is executed at each node of the decomposition, thereby defining the DP algorithm explicitly, and other programs such as SEQUOIA, dynASP, and dynPATRIX. However, these systems are either designed

as tools for prototypical DP implementations, which are not easily extensible to new application areas, or suffer from high memory demands for storing partial solutions during the computation Charwat (2015). This begs for a parallel framework, which supports extensible applications in different areas and resolves the high memory demands.

Algorithm 1 DP-Reduce

Input: Tree Node *treeNode*

```

for each childNode in treeNode do
    DP-Reduce(childNode)
end for
if treeNode is leaf then
    Generate-All-Combinations(treeNode, 0, colors, coloring)
end if
if treeNode is introduce then
    DP-Introduce(treeNode)
end if
if treeNode is forget then
    DP-Forget(treeNode)
end if
if treeNode is join then
    DP-Join(treeNode)
end if

```

Algorithm 2 Generate-All-Combination

Input: Tree Node *treeNode*, index *index*, colors *colors*, coloring *coloring*

```

if size of coloring = size of treeNode vertices then
    if coloring is valid then
        Add coloring to treeNode solutions
    end if
    return
end if
for each color in colors do
    coloring  $\leftarrow$  color
    Generate-All-Combination(treeNode, index + 1, colors, coloring)
    coloring remove color
end for

```

Algorithm 3 DP-Join

Input: Tree Node *treeNode*

Generate-All-Combinations(*node*, 0, colors, coloring)

for each *combination* in *combinations* **do**

 minCost $\leftarrow +\infty$

for each *LeftChildNodecombination* in *LeftChildNodecombinations* **do**

for each *RightChildNodecombination* in *RightChildNodecombinations* **do**

if Is-Compatible(*combination*, *LeftChildNodecombination*, *RightChildNodecombination*) **then**

 cost $\leftarrow$ *LeftChildNodecombination*.cost + *RightChildNodecombination*.cost

 cost $\leftarrow$ cost - *combination*.cost

if cost < minCost **then**

 minCost = cost

end if

end if

end for

end for

combination->cost = minCost

end for

Algorithm 4 DP-Forget

Input: Tree Node *treeNode*

Generate-All-Combinations(*node*, 0, colors, coloring)

for each *childNodeCombination* in *childNodeCombinations* **do**

 whiteCombination $\leftarrow$ Get-Similar-Coloring(*childNodeCombination*, *node*, *extraVertex*, white)

 blackCombination $\leftarrow$ Get-Similar-Coloring(*childNodeCombination*, *node*, *extraVertex*, black)

if whiteCombination $\neq \emptyset$ and blackCombination $\neq \emptyset$ **then**

node->store(Minimum-Cost(whiteCombination, blackCombination))

else if whiteCombination $\neq \emptyset$ **then**

node->store(whiteCombination)

else if blackCombination $\neq \emptyset$ **then**

node->store(blackCombination)

end if

 Add-Extra-Vertex(*childNodeCombination*, gray)

node->store(*childNodeCombination*)

 Add-Extra-Vertex(*childNodeCombination*, white)

if All-Extra-Vertex-Neighbors-Gray(*childNodeCombination*, *extraVertex*) **then**

 Add-Extra-Vertex(*childNodeCombination*, black)

node->store(*childNodeCombination*)

end if

end for

Algorithm 5 DP-Introduce

Input: Tree Node *treeNode*

```
for each childNodeCombination childNodeCombinations do
  Add-Extra-Vertex(childNodeCombination,white)
  if childNodeCombination is valid then
    node->store(childNodeCombination)
  end if
  Add-Extra-Vertex(childNodeCombination,gray)
  node->store(childNodeCombination)

  Add-Extra-Vertex(childNodeCombination,white)

  Add-Extra-Vertex(childNodeCombination,black)
  node->store(childNodeCombination)
end for
```

Algorithm 6 Compute-Function

Input: tree node *node*

```
for each child in node.children do
  Compute-Function(child)
end for
node.set  $\leftarrow$  Generate-Sets()
node.set  $\leftarrow$  Filter-Sets()
for each child in node.children do
  node.set  $\leftarrow$  compute(node, child)
end for
```

Figure 5: Tables of dynamic programming on tree decomposition.

Chapter Four

A Parallel Framework

In this chapter, we introduce our parallel framework. Our underlying topology is the clique topology. For us, the hardware can follow any physical arrangement as long as there exists a communication path between each pair of computation units in a system. The parallel framework addresses the four essential components of any parallel algorithm:

Task Encoding. What constitute a single task? How do we represent it?

Initialization. How do we begin the computation? What task is initially assigned to which computation unit?

Load Balancing. How are tasks created and exchanged? How do we make sure that all computation units have equal task loads?

Termination Detection. When does the computation terminate?

4.1 Common Parallel Computing Techniques

In parallel algorithm design, even those deemed embarrassingly parallel, we have to decide on a suitable inter-process communication protocol. For instance, the following questions are often posed when designing a parallel algorithm: which computation

unit(s) should read the data? How is the data partitioned and distributed to computation units? Should one computation unit read the data and broadcast it to all other computation units involved in the computation? Will this process be responsible for gathering the data at the end of computation to build the final result? Or should all computation units be allowed to communicate with each other?

The above considerations lead us to the realization that almost all parallel algorithms impose some kind of a communication graph that determines how processes cooperate with each other. This communication graph is virtual and it is imposed solely by the definition of process-to-process communication in the algorithm. See Abu-Khzam, Mouawad, and Jahed (2015) for further details and related methodologies.

The master-worker pattern is one of the most widely used paradigm for designing distributed algorithms. In a master-worker computation, a special computational unit, dubbed *master* is responsible for task distribution, load balancing, and termination detection. Communication between workers in a master-worker computation is typically avoided, making the master a bottleneck for communication. Despite its limited scalability, and mainly due to its simplicity, the master-worker approach has been used to solve many important problems Durillo, Nebro, Luna, and Alba (2008).

At the cost of added complexity, a fully-decentralized approach allows for greater scalability. This is since the communication throughput is not bounded by a single node anymore. In a fully-distributed system every node is, at the same time, a master and a worker. In other words, each node can assign and distribute tasks to other units, and can also receive and process jobs from other nodes. Decentralized algorithms typically dictates specific protocols for initialization, task distribution, progress monitoring, and termination detection. Since our goal is out-most scalability, we adopt a fully-decentralized paradigm when designing our framework.

4.2 Task Encoding

In figure 4 of chapter II, we showed that each tree node is a discrete, smaller instance of the original problem. For each tree node, the algorithm in 1, computes the optimal solution for the sub-graph discovered so far. As a dynamic programming top-down algorithm, computing the optimal solution at a given tree node requires the optimal solutions of each child node. Therefore, we must be careful not to schedule a task (node) for computation prior to its children.

When dealing with network-based computer systems, one of the most important factors is to minimize the amount of data transfer between computation units. This is especially true for heterogeneous compute clusters where the bandwidth and latency can vary greatly between two different pairs of computation units. Since in a distributed environment tasks are exchanged between computation units, the amount of data transfer can quickly become a bottleneck for the computation. Computation units will be spending too much time exchanging tasks rather than working on their own tasks especially since a task could be large being a subproblem consisting of a subgraph and other information about the state of the search.

Since we are dealing with a graph problem and each task corresponds to a subgraph, task encoding depends heavily on how we represent a graph. Two very well known data structures to encode a given graph $G = (V, E)$ are the adjacency matrix and the adjacency list representations. We will be using the adjacency list (AL) because it is a more compact representation that requires $\Theta(|E|)$ of storage. The AL is typically implemented using an n cell array of linked lists where each list contains the neighbors of a single vertex in the graph. Figure 6 shows an example of both data structures. All computation units will contain the graph structure in order to check for neighbors, and the tree decomposition of G will be given as well.

In the tree decomposition example, an instance is basically a tuple containing r tree nodes. Each tree node contains an induced sub-graph H of G , a set S , its children

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	1	0
2	1	1	0	1	0	0	0
3	0	1	1	0	1	0	0
4	0	0	0	1	1	1	0
5	0	1	0	0	1	0	1
6	0	0	0	0	1	1	0

Figure 6: An example Adjacency Matrix and Adjacency List

children, and parent p . The data structure we will use to encode a given tree node is $y = (V, S)$. In the tree node representation, a node with $n = |V|$ vertices is modeled using an n cell array t . An entry in the array $y_{V[i]}$ is simply the vertex label from G as illustrated in figure 7, and S will be the generated set of all possible colorings or solutions obtained from the child as discussed in the previous chapter. $y_S[i]$ will reference the i^{th} coloring in the set S . Of course the sets will be validated, and invalid colorings will be discarded. Obviously, the solution set of the tree node representation has a space of complexity of $\Theta(3^n)$. Each computation unit will store the tree and graph structures, but they do not have to store all the other tree nodes' solutions.

$$V \Rightarrow \boxed{1} \boxed{2}$$

Figure 7: A vertex array V in a tree node.

$$S = \left\{ \begin{array}{l} \{g, g\} \\ \{b, b\} \\ \{w, w\} \\ \{g, b\} \\ \{g, w\} \\ \{b, w\} \\ \{b, g\} \\ \{w, g\} \\ \{w, b\} \end{array} \right.$$

Figure 8: A coloring set S of V .

If each tree node stores its generated set along with the solutions obtained for each tree node on a single computation unit, the computation unit has to have a huge amount of memory. Thus, making it impractical and infeasible due to the huge amount of memory. For instance, the tree shown in figure 9 shows that such tree with their sets is difficult to store on a single computer. In order to solve this problem, we can delegate an ancestry path for the computation unit. In other words, each computation unit will be assigned a path from the leaf to the root or to an assigned parent. Afterwards, we can assign the remaining paths to the least loaded computational units. Unfortunately, this will not work because tree nodes have different bag sizes and will keep other computational unit idle. Transmitting the data with their coloring set has a huge

Figure 9: An example of an unbalanced tree decomposition with 127 nodes.

negative impact on the system congesting the network and negatively impacting the performance.

This problem calls for task-encoding schema that do not require each computation unit to store each tree nodes' set. Each computation unit will store the tree structure to avoid sending the whole tree, and not all computation units will store the whole states of the computation of the tree decomposition. That way, when a computation unit receives a task from a computation node, it only has to queue or compute it. One of other advantage of completing a computation is that the child's set can be dropped freeing the memory of that computation unit memory for more tasks.

We will be using a completely decentralized approach for our framework because of its efficiency and scalability. Each computation unit will be able to easily find its task which is basically the tree node from the tree decomposition, and from that node it begins its computation.

After each completed task, the computation unit that was responsible for the computation will send the new solution set back to the originator. For the "bottom-up" approach, the computation unit that has The parent node will receive and eliminate the solutions obtained from its children. This is because not all those solutions are valid to its other children. Thus, it will be decreasing the set size even further for the next evaluation. After the final task is completed and evaluated, the computation unit that contains the root node will store all feasible solutions of the graph. On the other hand, through using the "top-down" approach, we simply have to send the index of the tree node to an idle computation unit which will then either enqueue or compute it.

One of the main features that enables the framework to tackle other graph problems is the validation function which is present in all three operations of algorithm 6, and the generate function that enables the user to choose the different colorings they wish to use. The addition of a custom validation function in this parallel framework makes it flexible. This function takes the coloring obtained along with a tree node's induced sub-graph, and the user has to determine if the coloring obtained form a solution.

We can finally conclude this section with the encoding of a node in a tree decomposition as follows:

type is an enum label of the type of tree node in the nice tree decomposition.

vertices is a list of vertices found in a tree node.

solutions is a set of all feasible colorings of the tree node.

P_{node} to denote the parent of the current node.

C_{node} to denote the indices of its children nodes.

extra – vertex to denote the vertex difference between the parent and child node.

As Well as the task structures along with states which goes as follows:

- Status Update Task:

status is a boolean status for every computation unit in the cluster (busy or idle).

- Tree Node Index Task:

treeNodeIndex is the tree node index inside the tree decomposition.

originator is the index of the computation unit that created the task.

- Result Task:

treeNodeSet is the tree node results obtained.

originator is the index of the computation unit that will receive the data.

4.3 Initialization

One traditional centralized approach for initializing the computation is for some root computation unit to start, and for other computation units to remain idle and await to receive their tasks from the central computation unit to begin their computation. While the simplicity and automatic nature of this approach is rather appealing, an overhead maybe placed on a single computation unit, but a necessary one. As long as the root does not compute like the other computation units, the system will be congested and restricted to this single computation unit. Clearly, a centralized approach is not an approach that can scale to thousands of computation units or tree nodes.

In the “bottom-up” approach, we would work create an organized task list TL which comes in the form of a queue. The task list TL is organized level-by-level starting from the leaves moving upwards until the root, and the size of TL will be exactly the number of nodes in tree decomposition. One way of distributing the tree nodes is by using the circular algorithm which means that we assign the first tree node in TL to the first computation unit, the second one to the second, etc. We keep on repeating until all the tree nodes have been assigned. That way, we go up by levels in the tree adding only the parent nodes which are not already in the list as shown in Algorithm 7. This approach will cause starvation and might block other computation units, and as such we did something different.

As we stated previously, we will be using the “top-down” approach. We demonstrate how the parallel algorithm in algorithm 9 works. The first computation unit will begin moving through the tree from the root. As it progresses, when the a tree node has more than one child, we have to account for a few cases on when to delegate. We must always make sure that a computation unit does not delegate all the tree node children, and that it has has to have tasks for itself. We have to check if a certain THRESHOLD is met. The threshold can be done as one of either on the node level such that we delegate sub-trees to other computation units. In other words, we will calculate the number of tree nodes for each sub-tree and based on that value, we will either keep the task or

delegate. This is impractical due to the number of vertices might vary drastically from sub-tree to sub-tree, and as such we will use the threshold as the number of vertices found in a sub-tree. The threshold is calculated via the number of vertices for each sub-tree, and delegate accordingly. Afterwards, we will continue traversal of the tree until we reach a point where we have to wait for the data from the delegated nodes.

Part of the initialization step is to initialize the embedded MPI framework. This is simply creating the thread that will handle the communications between each computation unit. It will always listen for each computation unit for any updates. For instance, if the computation task is presented with a new task, it will automatically enqueue it so that the computation thread can read from that queue synchronously and either delegate or compute it. It will also broadcast the status of the computation unit to all other computation units in order to obtain a task. Lastly, it will also transmits the results to the idle process. How it works is that each computation unit will probe an MPI receive if it exists.

We can deduce from the “bottom-up” approach that we guarantee no computation unit will skip a level until all computation units reach the same level. In this model, all the computation units will start the computation and at the end of their computation, they will send their solution sets to their respective parent independent from other computation units. As illustrated in the first part of Algorithm 8, the tree nodes distribution will be somewhat equal among the computation units, and because of advanced cluster system which allowed all computation units to have common storage, broadcasting the shared data can be avoided! Eventually, all computation units will start working from the TL and each computation unit can find out where each node is located by simply indexing the tree. In the “top-down” approach, we neither have to distribute the tasks at the beginning nor have to keep track to whom we delegated. We will encode the originator in the task itself so that the computation unit that did the computation send the solution back to the originator who requires the solution to continue the computation.

Optimally, we would like all computation units to keep working on assigned tasks and

not remain idle. In other words, we do not want to synchronize the computation units at each level so that the level acts as a barrier and only when all the computation units reached this level, they may proceed. Our fully distributed computation initialization scheme does not require any cooperation between computation units when it comes to computing solutions. The only time a computation unit remains idle is when a parent is waiting for the child to complete its computation.

Algorithm 7 Create-Task-List

Input: The tree decomposition T , computation units cu

Output: Distribution of the tree nodes to the computation units

```

sort-tree-by-level( $T$ )
for  $i \leftarrow 0$  to  $|T|$  do
     $task\_list \leftarrow i/|cu|$ 
end for
return  $task\_list$ 

```

Algorithm 8 Worker-List-Traversal

Input: tree node sets c , Graph G , computation unit cu

```

 $task\_list \leftarrow \text{Create-Task-List}(T)$ 

for each  $node$  in  $task\_list$  do
    if  $node$  is parent then
        for  $\forall child \in node.children$  do
             $node.set \leftarrow \text{Get-Valid-Sets}(child)$ 
        end for
    else
         $node.set \leftarrow \text{Generate-Sets}()$ 
    end if
     $node \leftarrow \text{compute}(node, P_{node})$ 
end for

```

Algorithm 9 Parallel-DP

Input: Tree Node *treeNode*

```
for each childNode in treeNode do
  if childNode is last or Threshold then
    Parallel-DP(childNode)
  else
    Delegate(childNode)
  end if
end for
for each childNode in treeNode do
  if childNode is delegated then
    wait for childNode
  end if
end for
if treeNode is leaf then
  Generate-All-Combinations(treeNode, 0, colors, coloring)
end if
if treeNode is introduce then
  DP-Introduce(treeNode)
end if
if treeNode is forget then
  DP-Forget(treeNode)
end if
if treeNode is join then
  DP-Join(treeNode)
end if
```

Algorithm 10 Parallel-Compute-Function

Input: tree node *node*

```
for each child in node.children do
  if child is last or Threshold then
    Parallel-Compute-Function(child)
  else
    Delegate(child)
  end if
end for
for each child in treeNode do
  if child is delegated then
    wait for child
  end if
end for
node.set  $\leftarrow$  Generate-Sets()
node.set  $\leftarrow$  Filter-Sets()
for each child in node.children do
  node.set  $\leftarrow$  compute(node, child)
end for
```

4.4 Dynamic Load Balancing

Our initial task assignment strategy allows a computation unit to independently find the units handling its neighboring tree nodes (those corresponding to parents and children of the tree nodes it is handling). However, the strategy does not guarantee that the weights of its tasks will be close to those of the other computation units. In other words, what if the size of the tree nodes it contains are hefty, or the tree has an imbalanced structure which will lead to a very asynchronous workload that is very difficult to balance Sullivan et al. (2013). This is true since the tree nodes in T may vary in size. This means that some computation units may finish their tasks earlier than other computation units and as such they become idle. When a computation unit becomes IDLE, it is allowed to move on in the task list as long as the parent's children have finished its computation. For this section, we are interested in the tree node (i.e., task) distribution.

In a centralized load balancing scheme, one computation unit is designated as master while other computation units serve as workers. The master's sole purpose is to generate and serve tasks to workers. However, a centralized load balancing approach is not scalable as the master computation unit will quickly become a bottleneck when thousands of computation units are involved in the computation. In a fully distributed load balancing scheme, all computation units serve both the role of a master and the role of a worker. Particularly, every computation unit is allowed to compute and distribute tasks to other computation unit rather than each computation unit to have their set of tasks ahead of time. Decentralized load balancing schema are a perfect match for dynamic programming algorithms. This is true since each computation unit contain tree nodes of the main tree. This property abolishes the communication overhead required to make sure that all given tasks are distributed to all computation units.

One requirement to ensure good load balancing and to reduce the overall communication overhead is to make sure to always distribute the tree nodes such that their total is equal, but unfortunately this problem is *NP*-Hard because it would be similar to the

k -Subset problem where we have to divide a set into k subsets such that all subsets have equal sum, and we also have to take into account the size of each node which gives this problem two dependencies. If a task is big enough to require a lot of time, the computation unit is less likely to finish and move up the tree in a timely manner. Weighting the tasks has always been a challenge for decentralized load balancing scheme as it requires the calculation of every task's complexity.

One added bonus of decentralized computations is the minimal communication they entail. A computation unit can easily obtain the set it requires, and continue working on its task. For that purpose, each computation unit must keep track of two pieces of information:

process_availability is a flag that set a computation unit as idle or busy doing computation so that it may pass on the task or work on it.

task_originator is the computation unit index of who generated the task so that the computation that is working on it sends it back to it rather than having a chain of callbacks.

In the “bottom-up” approach, as we progress through the *task_list*, we will hit points where a parent can have multiple children, and as such will force a computation unit c_i , which has on the children node, to remain idle so that the parent can get the new set. This is impractical because what if c_i has another task which can be executed concurrently with this task (This task would be on the same level), Thus, when c_i detects that a parent is preoccupied, it may skip this task on a condition that it does not move up the tree and remain on the same level.

To see how we may handle load balancing for task tasks assigned to computation units, consider the tree decomposition shown in figure 10. We will discuss four scenarios illustrated in this figure. In these four scenarios, the nodes are scattered in such away that each computation node will contain three tree nodes. We will be given a computation c_0 in each scenario, and we will observe and study c_0 's behavior and how it will

overcome certain obstacles.

In the first scenario (the red scenario), c_0 contains the tree nodes $[y_{14}, y_8, y_3]$. This is a typical illustration of our algorithm and it would act in a layered fashion meaning it will begin with node y_{14} , and wait for all other computation units to complete level 4 before progressing to level 3. Once it finishes and reach y_8 , it cannot progress because of y_8 's non-computed children which are y_{16} and y_{15} before it can progress to y_3 at level 2. One thing c_0 has overcome, and that is the fact that once it reaches the next level, it will have to wait for the solutions to be computed. Because of the fact that it remained idle, distributing the tasks as such is impractical, and it could have worked on the children tasks itself.

In the second scenario (The green scenario), c_0 contains the tree nodes $[y_5, y_6, y_7]$. We can note all these nodes are on the same level 3. c_0 has to remain idle for all its descendant nodes to finish (Shown with the 3 dots) matching before it can start to do any matching with its parent. This will keep the process that contains the parent which is y_2 idle while it could take one of the children to begin computation. Nevertheless, it will be optimal because only c_0 will have to communicate with with computation unit that houses y_2 .

In the third scenario (The yellow scenario), c_0 contains the tree nodes $[y_{10}, y_{11}, y_{12}]$. We can note all these nodes are on the same level 3 and that y_{10} and both y_{11} and y_{12} are cousins. In trees, two nodes are cousins of each other if they are at same level and have different parents. If y_{13} has already began to compute a matching with y_4 , y_{11} and y_{12} will have to remain idle. Thus, if the first node in the queue of execution is either y_{12} or y_{11} , c_0 will remain idle until y_4 becomes free before it continues. Similarly if y_3 was preoccupied and y_{10} was next in the task list, then c_0 has to remain idle. To avoid this issue, we can allow c_0 to skip tasks that will keep it idle to find another node which can start matching. This will halt the computation for either sub-trees while the computation units that contains the roots could have computed them if they were idle. This scenario will cause a bottleneck for each subtree.

In the fourth scenario (The magenta scenario), c_0 contains the tree nodes $[y_2, y_9, y_4]$. We can note that all these nodes are from different subtrees. We can deduce that y_9 should be first in the task list followed by either y_2 or y_4 . In such a scenario, the idea of delegation can come in handy in order to minimize the communication, and c_0 can delegate y_2 to idle computation units if c_0 was busy with other computations like y_4 or y_9 . At level 2 or 3, most of the computation units will be idle, and it would speed up the process to give an idle computation unit a task from a different subtree to compute independently from the current computation, but it will also cause a blockade while moving up the tree for each subtree. If c_0 began computation at y_9 , the subtrees that has the y_2 and y_4 will be blocked and waiting for c_0 to finish. This will lead to a bottleneck. We can conclude from this scenario that distributing tree nodes randomly from different subtrees is inefficient.

Figure 10: A layered a tree decomposition

To speed up computations, we noted before that parent-child matching can be done independently for each subtree, and this provides us the chance to scale up the computation drastically. For the “bottom-up” approach, rather than approaching the computation in a layer by layer fashion which can be easily done (but also cause a bottleneck)

we can traverse the tree node by node because once a parent's children finish their computation, the parent will have the final result for the current subtree, and can directly communicate its results to its own parent. On the other hand, for the “top-down” approach, we will allow one computation node to begin from the root, and as it progresses through the tree (and if a tree node has more than one child) we will delegate its children to other computation units except for one which this process will handle. Likewise, the computation units that receive the delegated tree node will traverse this subtree and perform the same delegation. In this manner, no computation process will remain idle or blocked, and each computation unit will receive a task when it reports that it is idle.

Figure 11: An example of an unbalanced tree decomposition

4.5 Termination

We can distinguish between two types of termination: positive termination and negative termination. Positive termination occurs when a “Yes” answer was found by some computation unit. In our case, we will always find a yes answer because we will be targeting the optimal solution, but in the case of parameterized problems a negative termination may occur if a computation unit does not successfully find a set of vertices that lead to a solution of size k (or less) in the graph. Nevertheless the instance will be solved and all computation unit will terminate their computation.

Termination detection is for a computation unit to be able to autonomously detect that the computation has terminated (be it positively or negatively) in order to terminate its own process. A naive strategy for termination detection is for a computation unit to keep track of the status of each other computation unit. The status is typically either idle (no tasks/tree nodes available) or busy (already processing a task). A computation unit can safely terminate in this case whenever all computation units become idle. This binary strategy, however, highly depends on broadcasting messages and assumes

Figure 12: An example of a nice tree decomposition of figure 11

a fully connected topology. Whenever a computation unit receives/finishes a task, it has to send a “broadcast” message informing all other computation units. In a computation involving thousands of computation units, many computation units can change status very frequently, which results in having the network flooded with broadcast messages that degrade the computational throughput. Although broadcast algorithms are becoming more and more efficient, they are considered the worst collective operations in high performance communication (at least in the OpenMPI implementation Rabenseifner (2000)). In this section, we present a couple of simple strategies to terminate all computation units without using collective broadcast messages.

A simple way to ensure that a computation is complete is to check if all the computation units have completed their current tasks found in the *task_list*. Thus, we can safely terminate. The solution will always be with the computation unit that has the root node of the tree, and will be the final computation unit to terminate. In doing so, we do not take advantage of computation unit to delegate tasks to. This will not speed up the computation because subtrees can be computed independently, but yet the current computation unit will remain busy with a different heavy task and probably with other heavy tasks.

Another strategy that we adopt in order to ensure that all the computation is complete is to allow the computation unit with the root node to broadcast a message when it finishes computing the last task. This determines that the computation units are safe to terminate. Using this strategy, we achieve better speed-up since we will have better load balancing (the idle computation units will be used for task delegation). Yet, we will pay for one computation unit to broadcast to the other computation units that the computation has terminated.

If a “no instance” is allowed, this can be easily detected because, as we progress through levels in the tree, we will have empty sets as solution sets among the computation units, and thus after a few levels we can safely terminate the computation because no solution or solution of size k or less has been found.

The strategy we are adopting is simply detecting that all the processes become idle. The communications are done on a separate thread than the main thread which is actually doing the computation.

Chapter Five

Experimental Analysis

To evaluate the performance of our parallel framework we implemented the dynamic programming on tree decomposition algorithm for Dominating Set presented in chapter IV using our framework. The algorithms were implemented in C++ using the Open-MPI library, a Message Passing Interface (MPI) implementation. We ran a number of experiments to measure the running time, memory usage, communication overhead, and the performance of various parts of our framework. In this chapter, we first begin by presenting a brief overview of the implementation strategy followed by our testbed and experimental results.

5.1 Implementation

We use the adjacency matrix data structure to represent the input graph. This is justified since some of the graph instances we are tackling are dense by nature. Moreover, we use the *tree* encoding illustrated earlier. We do not have to create a copy of each subgraph corresponding to a tree node since the edges can be easily queried from the original graph. Thus, we can reserve more memory for the sets, and we will also use a structure called *set* found in the *tree* encoding which contains all the possible coloring generated or obtained at the start of the feasible solutions.

All computation units begin reading the input graph and constructing the adjacency

matrix and tree decomposition simultaneously. In the beginning, all computation units will be idle awaiting for tasks. The processor with the smallest rank will begin traversing the tree decomposition for the root node in a post-order manner. Whenever a processor reaches a non-unary node, it delegates as much of the nodes' children to IDLE processes as possible. The more a computation unit moves deeper in the tree, the more it will delegate tasks.

5.2 Testbed

We used synthetic data generated by the open source library "htd" so we may generate graphs with up to 500 vertices (see Abseher, Musliu, and Woltran (2016)). "htd" utilizes multiple algorithms including the "quickBB" algorithm to obtain lower and upper bounds on the width of an optimum tree decomposition. For more details regarding the algorithm we refer the reader to Gogate and Dechter (2004). We also acquired data from the well known collaborative online database **treedecompositions.com**. As an open database, anyone can submit graphs along with valid tree decompositions (not necessarily optimal). Users can browse graph instances and submit possibly better tree decompositions. Unfortunately, the largest graph in that database has only 150 vertices, which is why we also resorted to synthetic instances.

5.3 Experimental Results

We ran our parallel framework on a cluster composed of 16 compute nodes each equipped with 2 vCPUs. Therefore, we can run our experiments on a maximum of 32 cores. Each node is equipped with 2GB of RAM and 40GB of SSD storage. All nodes are interconnected via a high-speed 10GB Ethernet network. Furthermore, We ran our serial code on a single node of that cluster with the same specifications.

We tested our framework on graphs obtained together with their nice tree decompositions from the above sources. Most instances obtained from the treedecompo-

sition.com database have skewed tree decompositions in the sense that the trees are path-like. We synthesized other graphs that have more balanced tree decomposition and used the htd library to obtain their corresponding nice tree decompositions. To set a benchmark running time, we fed those instances to the sequential framework (Algorithm 1), and show the running times in table 1 and table 3.

We compare the parallel against the sequential as the treewidth and the number of vertices varies. The results are shown in tables 1 and 3. For the sequential algorithm, it took some time to compute a solution with a large number of vertices. Moreover, it started to take more than 10 minutes as the treewidth reached 8. The results obtained are normal because of the running time of the algorithm which is exponential. When the treewidth value is 10, it took more than 15 hours for the process to complete and terminate. As the treewidth increases to 15 and above, the algorithm becomes even more memory hungry and as such the nodes in our cluster ran out of memory for a processor around 10 minutes into the computation.

As expected, the parallel algorithm performed far better than the sequential algorithm. The sequential algorithm's performance exponentially grew as the treewidth grew. The parallel algorithm's performance was as expected, and performed very well when it came to distributing the load among the processes. Table 1 shows the performances of the parallel framework while varying the number of cores. We tested the scalability of the framework by using 4, 8, 16, and 32 cores. The framework showed good scalability as it scaled up to 32 cores when the nice tree decomposition was nearly balanced as shown in table 3. We can see that using 4 cores, the algorithm took 623 seconds as opposed to 214 seconds when using 32 cores. However, we did observe that our approach does not scale well when the trees are path-like. Thus, it does not contain many join nodes, which affects the amount of task delegation.

Table 1: Sequential and Parallel Running Times in seconds on unbalanced nice Tree Decomposition

TW	 V 	 E 	C1	C4	C8	C16	C32
4	100	158	345	211	166	125	128
4	200	354	478	230	207	168	160
5	300	537	628	351	299	269	247
5	400	733	868	463	400	465	335
4	500	652	1118	662	623	563	565
8	51	140	2034	1651	1616	1564	1237
9	40	183	1661	689	595	574	584

Table 2: Average waiting time in seconds to receive results for unbalanced nice Tree Decomposition

TW	 V 	 E 	C4	C8	C16	C32
4	100	158	10	8	6	4
4	200	354	12	9	6	5
5	300	537	11	8	7	5
5	400	733	13	10	8	7
4	500	652	15	11	9	8
8	51	140	20	16	10	11
9	40	183	37	60	32	12

Table 3: Sequential and Parallel Running Time in seconds of near balanced nice Tree Decomposition

TW	 V 	 E 	C1	C4	C8	C16	C32
5	517	525	1276	561	401	288	197
6	518	531	1539	674	501	338	219
7	519	538	1855	878	591	395	274
8	520	546	2162	920	663	507	325

Table 4: Average waiting time in seconds to receive results of near balanced nice Tree Decomposition

TW	 V 	 E 	C4	C8	C16	C32
5	517	525	20	20	20	17
6	518	531	19	20	19	17
7	519	538	21	20	19	19
8	520	546	20	19	17	16

Figure 13: Running time for the sequential and parallel framework for unbalanced nice tree decomposition

Figure 14: Running time for the sequential and parallel framework for near balanced nice tree decomposition

Chapter Six

Concluding Remarks

We presented a scalable parallel framework that uses dynamic programming on tree decomposition to solve domination-like problems on graphs. Our framework is a combination of an efficient task encoding approach, and a novel methodology to control communications and memory. We explored different approaches for implementing the parallel framework and proposed a decentralized approach as a resulting solution. Testbed implementation and experimental results showed that our framework can provide a solution with near linear speedup.

Future work includes adding multi-threading to the dynamic programming set operations, and the possibility of using a larger environment to accompany memory of size of the generated solutions. Our approach performs better on balanced trees and the performance degrades as the tree decomposition becomes close to a path decomposition. Our future plan includes amending the proposed framework to be able to handle such extreme cases.

We have addressed the generic (classical) Dominating Set problem. The same approach works for similar domination problems since in most (if not all) such problems the status of a vertex during the search for a solution can be encoded via a coloring scheme. A challenging, but not impossible, task would be to address the connectivity constraint. Connected domination is of particular interest due to a number of applications. Currently the fastest exact algorithm for Connected Dominating Set (CDS) runs

in $O(1.8619^n)$ Abu-Khzam, Mouawad, and Liedloff (2011). Our future plan includes a parallel CDS algorithm on graphs of bounded treewidth. For this we intend to consider the approach presented Cygan et al. (2011) along with the framework presented in this thesis.

References

- Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., & Woltran, S. (2014). The d-flat system for dynamic programming on tree decompositions. In *European workshop on logics in artificial intelligence* (pp. 558–572).
- Abseher, M., Musliu, N., & Woltran, S. (2016). *htd—a free, open-source framework for tree decompositions and beyond* (Tech. Rep.). Technical report, DBAI-TR-2016-96, TU Wien.
- Abu-Khzam, F. N., Daudjee, K., Mouawad, A. E., & Nishimura, N. (2015). On scalable parallel recursive backtracking. *J. Parallel Distrib. Comput.*, 84, 65–75.
- Abu-Khzam, F. N., Langston, M. A., & Shanbhag, P. (2004). Scalable parallel algorithms for difficult combinatorial problems: A case study in optimization. In M. H. Hamza (Ed.), *Proceedings of the IASTED international conference on parallel and distributed computing and networks, innsbruck, austria, february 17-19, 2004* (pp. 649–654). IASTED/ACTA Press.
- Abu-Khzam, F. N., Langston, M. A., Shanbhag, P., & Symons, C. T. (2006). Scalable parallel algorithms for fpt problems. *Algorithmica*, 45(3), 269–284.
- Abu-Khzam, F. N., & Mouawad, A. E. (2012). A decentralized load balancing approach for parallel search-tree optimization. In *Parallel and distributed computing, applications and technologies (pdcat), 2012 13th international conference on* (pp. 173–178).
- Abu-Khzam, F. N., Mouawad, A. E., & Jahed, K. A. (2015). Highly scalable parallel search-tree algorithms: The virtual topology approach. In *Cluster computing (cluster), 2015 ieee international conference on* (pp. 518–518).
- Abu-Khzam, F. N., Mouawad, A. E., & Liedloff, M. (2011). An exact algorithm for connected red–blue dominating set. *Journal of Discrete Algorithms*, 9(3), 252–262.
- Alber, J., & Niedermeier, R. (2002). Latin 2002: Theoretical informatics: 5th latin american symposium canun, mexico, april 3–6, 2002 proceedings. In S. Rajsbaum (Ed.), (pp. 613–627). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Bodlaender, H. L. (1997). Mathematical foundations of computer science 1997: 22nd international symposium, mfcs ’97 bratislava, slovakia, august 25–29, 1997 proceedings. In I. Prívvara & P. Ružička (Eds.), (pp. 19–36). Berlin, Heidelberg: Springer Berlin Heidelberg.

- Bodlaender, H. L., Bonsma, P., & Lokshtanov, D. (2013). Parameterized and exact computation: 8th international symposium, ipec 2013, sophia antipolis, france, september 4-6, 2013, revised selected papers. In G. Gutin & S. Szeider (Eds.), (pp. 41–53). Cham: Springer International Publishing.
- Bodlaender, H. L., & Koster, A. M. (2010). Treewidth computations i. upper bounds. *Information and Computation*, 208(3), 259–275.
- Charwat, G. (2015). Dynamic programming on tree decompositions using binary decision diagrams: Research summary. In *Iclp (technical communications)*.
- Corneil, D. G., & Keil, J. M. (1987). A dynamic programming approach to the dominating set problem on k-trees. *SIAM Journal on Algebraic Discrete Methods*, 8(4), 535–543.
- Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J. M., & Woźtaszczyk, J. O. (2011). Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of computer science (focs), 2011 ieee 52nd annual symposium on* (pp. 150–159).
- Diestel, R. (2006). *Graph theory*. Springer.
- Downey, R. G., & Fellows, M. R. (2012). *Parameterized complexity*. Springer Publishing Company, Incorporated.
- Durillo, J. J., Nebro, A. J., Luna, F., & Alba, E. (2008). A study of master-slave approaches to parallelize nsga-ii. In *Parallel and distributed processing, 2008. ipdps 2008. ieee international symposium on* (pp. 1–8).
- Ekpanyapong, M., Waterwai, T., & Lim, S. K. (2006). Statistical bellman-ford algorithm with an application to retiming. In *Proceedings of the 2006 asia and south pacific design automation conference* (pp. 959–964).
- Gogate, V., & Dechter, R. (2004). A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on uncertainty in artificial intelligence* (pp. 201–208).
- Graph minors. iii. planar tree-width. (1984). *Journal of Combinatorial Theory, Series B*, 36(1), 49 - 64.
- Kloks, T. (1994). *Treewidth: Computations and approximations*. U.S. Government Printing Office.
- Korach, E., & Solel, N. (1993). Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics*, 43(1), 97–101.
- Rabenseifner, R. (2000). Automatic mpi counter profiling. In *42nd cug conference* (pp. 396–405).
- Rooij, J. M. M., Bodlaender, H. L., & Rossmanith, P. (2009). Algorithms - esa 2009: 17th annual european symposium, copenhagen, denmark, september 7-9, 2009. proceedings. In A. Fiat & P. Sanders (Eds.), (pp. 566–577). Berlin, Heidelberg:

Springer Berlin Heidelberg.

- Sullivan, B. D., Weerapurage, D., & Groër, C. (2013, May). Parallel algorithms for graph optimization using tree decompositions. In *Parallel and distributed processing symposium workshops phd forum (ipdpsw), 2013 ieee 27th international* (p. 1838-1847).
- Telle, J. A., & Proskurowski, A. (1993). Algorithms and data structures: Third workshop, wads '93 montréal, canada, august 11–13, 1993 proceedings. In F. Dehne, J.-R. Sack, N. Santoro, & S. Whitesides (Eds.), (pp. 610–621). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Wang, Q., Chen, M., Liu, Y., & Hu, Z. (2013). Towards systematic parallel programming of graph problems via tree decomposition and tree parallelism. In *Proceedings of the 2nd acm sigplan workshop on functional high-performance computing* (pp. 25–36).
- Wimmer, S., & Lammich, P. (2017). The floyd-warshall algorithm for shortest paths. *Archive of Formal Proofs*, 2017.
- Wu, J., Cardei, M., Dai, F., & Yang, S. (2006). Extended dominating set and its applications in ad hoc networks using cooperative communication. *IEEE Transactions on Parallel and Distributed Systems*, 17(8), 851–864.