Towards Systematic Parallel Programming of Graph Problems via Tree Decomposition and Tree Parallelism

Qi Wang
STAP Group, School of Software
Shanghai Jiao Tong University, China
aywq@sjtu.edu.cn

Meixian Chen
Dpt of Computer Science & Engineering
Shanghai Jiao Tong University, China
misam.chen@sjtu.edu.cn

Yu Liu
The Graduate University for Advanced Studies, Japan
yuliu@nii.ac.jp

Zhenjiang Hu
National Institute of Informatics, Japan
hu@nii.ac.jp

Abstract

Many graph optimization problems, such as the *Maximum Weighted Independent Set* problem, are NP-hard. For large scale graphs that have billions of edges or vertices, these problems are hard to be computed directly even using popular data-intensive frameworks like MapReduce or Pregel that are deployed on large computer-clusters, because of the extremely high computational complexity. On the other hand, many studies have shown the existence of polynomial time algorithms on graphs with bounded treewidth, which makes it possible to solve these problems on large graphs. However, the algorithms are usually difficult to be understood or parallelized.

In this paper, we propose a novel programming framework which provides a user-friendly programming interface and automatic in-black-box parallelization. The programming interface, which is a simple and straightforward abstraction called Generate-Test-Aggregate (GTA for short), is used to describe a set of graph problems. We propose to derive bottom-up dynamic programming algorithms on tree decompositions from the user-specified GTA algorithms, and further transform the bottom-up algorithms to parallel ones which run in a divide-and-conquer manner on a list of subtrees. Besides, balanced tree partition strategies are discussed for efficient parallel computing. Our preliminary experimental results on the Maximum Weighted Independent Set problem demonstrate the practical viability of our approaches.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; G.2.2 [*Mathematics of Computing*]: Graph algorithms

General Terms Algorithms, Design

Keywords graph problems, parallelization, GTA, tree decomposition, treewidth, transformation

FHPC '13, September 23, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2502323.2502331

1. Introduction

Many practical computing problems concern large graphs such as social networks and linked data [8, 13, 33], which may contain billions of edges or vertices. The computational complexity of such large datasets has exhausted the limits of a single computer, which demands parallelization of graph problems on parallel systems. Thus the approaches of using MapReduce-like frameworks to handle large graphs are actively studied [13, 15, 27], and also some specialized graph-programming frameworks such as Pregel [33], GraphLib [32] and PowerGraph [20] are developed to more efficiently resolve large-graph computation problems. However, many graph optimization problems, such as the *Maximum Weighted Independent Set* problem, are NP-hard. For large input graphs, these problems are hard to be computed even using MapReduce or Pregel that are deployed on large computer clusters, because such graph computations are often exponential in the size of the input graphs.

Arnborg et al. [4] showed that many NP-hard problems posed in monadic second-order logic can be solved in polynomial time using dynamic programming techniques on input graphs with bounded treewidth. Many problems on graph, such as graph optimization problems (e.g., *Minimum Vertex Cover*, *Maximum Weighted Independent Set*), can be solved via tree decomposition using bottom-up dynamic recursive algorithms. Despite more and more researchers have investigated the possibility for applying treewidth in innovative ways to help solve their problems [1, 5, 21, 28, 49], much less work is focused on parallelizing the algorithms for graph problems which are based on tree decomposition. Sullivan et al. [46] was the first one to parallelize algorithms for optimization problems. Their task-oriented bottom-up dynamic programming approach is shared-memory environment centered and is hard to be imported to MapReduce-like frameworks thus hardly handle bigger datasets that beyond the capacity of a single machine. Moreover, since the parallelization of their approach is only on the leaf nodes, the performance would be very inefficient if the shape of the tree decomposition is ill-balanced.

To parallelize the optimization problems on graphs and provide good programmability, we propose a novel programming framework which provides a user-friendly programming interface and automatic in-black-box parallelization. As a tree decomposition of a graph is an instance of the tree data structure, our parallelization approach for graph problems is based on both tree decomposition and tree parallelization. For tree parallelization, it is a known challenge to provide an efficient divide-and-conquer parallel algo-

rithm on trees with good load balance. Morihata et al. [41] developed a method for systematically constructing scalable divide-and-conquer parallel programs on trees based on *the third homomorphism theorem* and the zipper structure. However, the divisions of subtrees in a zipper may be imbalanced, making the parallel program inefficient. In our approach, we extend the third homomorphism theorem to tree decomposition to develop parallel algorithms, and we further extend the zipper for balanced tree partition. Our parallel algorithms, which run in a divide-and-conquer manner, have good load balance even when the shape of a tree decomposition is ill-balanced. Our programming interface is an expressive and general pattern of **Generate-Test-Aggregate** (GTA for short). The GTA programming pattern is simple and straightforward: firstly, the **generate** function generates all possible solutions candidates (e.g., all permutations), secondly, the **test** functions filters the candidates with certain predicates, and finally the **aggregate** function aggregates valid solutions. Users do not need to understand the parallelization details but just need to know this naive GTA abstraction and do sequential programming. Then our framework will transform the user-specified GTA programs to efficient parallel programs (e.g., MapReduce-like programs).

Our technical contributions in this paper are threefold. First, we propose the GTA abstraction for users to easily specify optimization problems on graphs with bounded treewidth. Second, we extend the application of the third homomorphism theorem to tree decompositions, and propose an approach to transforming bottom-up dynamic programming algorithms on tree decomposition to divide-and conquer parallel algorithms on zipper. Third, we propose a parallelization framework to automatically transform user-specified GTA algorithms to parallel ones which are suitable to be implemented on MapReduce-like framework.

The rest of this paper is organized as follows. Section 2 briefly reviews some preliminary knowledge. Section 3 introduces our parallelization framework. Section 4 and Section 6 illustrate our framework with some examples. Section 5 discusses tree parallelization. Section 7 shows the preliminary experimental results. Related work is discussed in Section 8. Finally, Section 9 concludes the paper with the future work.

2. Preliminaries

2.1 Notations

In this paper, the notations are mainly based on the functional language Haskell [6]. The parentheses for function applications may be omitted, i.e., $f\ a$ equals to $f(a)$. Functions are curried and bound to the left, thus $f\ a\ b$ equals to $(f\ a)\ b$. Function application has higher priority than those for operators, thus $f\ a \oplus b = (f\ a) \oplus b$. Operator $\circ$ denotes a function composition, and its definition is $(f \circ g)(x) = f(g(x))$.

Tuples are written like (a, b) or (a, b, c). A list is denoted by brackets split by commas. We use $[\,]$ to denote an empty list, and $+\!\!+$ to denote list concatenation. A list that has only one element is called a *singleton*. Operator $[\cdot]$ takes a value and returns a singleton with it.

Function *id* is the identity function. Function *fst (snd, thd)* extracts the first (the second, the third) element of the input tuple.

2.2 List homomorphism and the third homomorphism theorem

DEFINITION 1 (List homomorphism [7]). *Function $h :: [A] \to B$ is said to be a list homomorphism if there exists function $f :: A \to B$ and an associative operator $\odot :: B \to B \to B$ such that*

$$\begin{array}{lcl} h\ [\,] & = & \imath_\odot \\ h\ [a] & = & f\ a \\ h\ (x +\!\!+ y) & = & h\ x \odot h\ y \end{array}$$

hold, where $\imath_\odot$ is the unit of $\odot$.

List homomorphism is useful for developing parallel programs on list. The associativity of $\odot$ guarantees that a list can be divided at anywhere and the computation result is the same. [41]

THEOREM 1 (The third homomorphism theorem [19]). *Function h is a list homomorphism iff there exist two binary operators $\oplus$ and $\otimes$ such that the following equations hold.*

$$\begin{array}{lcl} h([a] +\!\!+ x) & = & a \oplus h\ x \\ h(x +\!\!+ [a]) & = & h\ x \otimes a. \end{array}$$

The third homomorphism theorem states that if a function can be computed in both leftward and rightward manners, then there exists a divide-and-conquer parallel algorithm to evaluate the function.

Morihata et al. [41] extend the third homomorphism theorem to regular data structures, including trees.

2.3 List homomorphisms on MapReduce

Google's MapReduce [15] is a popular programming model for processing large datasets in a massively parallel manner.

List homomorphisms fit well with MapReduce, because their input can be freely divided into sub-lists which can be distributed among machines. Then on each machine the programs are computed independently, and the final result can be got by a merging procedure. In fact, it has been shown that list homomorphisms can be efficiently implemented using MapReduce [31]. Therefore, if we can derive an efficient list homomorphism to solve a problem, we can solve the problem efficiently with MapReduce, enjoying its advantages such as automatic load-balancing, fault-tolerance, and scalability.

2.4 Graph definitions

Formally, a *graph* $G = (V, E)$ is a set of vertices V and a set of edges E formed by unordered pairs of vertices. All graphs in this paper are assumed to be finite, simple and undirected.

In a weighted graph, each edge or vertex is associated with some value (weight). The *weight* of a vertex is denoted as $w(v)$. We say $H = (W, F)$ is a *subgraph* of $G = (V, E)$, denoted as $H \subseteq G$, if both $W \subseteq V$ and $F \subseteq E$. An *induced subgraph* is one that satisfies $(x, y) \in F$ for every pair $x, y \in W$ such that $(x, y) \in E$. We denote the induced subgraph of G with vertices $X \subseteq V$ as G_X.

2.5 Graph optimization problems

In this paper we are interested in a class of important graph problems that optimally select a set of nodes from a given graph. We assume each vertex in a graph is assigned with an int weight value $w(v)$. If the weight of each vertex is one, these problems are to maximize or minimize the size of the selection set satisfying a certain condition.

Maximum Weighted Independent Set Given a graph $G = (V, E)$, an independent set (S) of the graph is a set of vertex satisfies the following condition:

$$\forall v, u \in S \Longrightarrow \neg(v, u) \in E$$

The *Maximum Weighted Independent Set* problem is to find an independent set with the maximum total weight.

Minimum Weighted Vertex Cover Given a graph $G = (V, E)$, a vertex cover (S) of the graph is a set of vertex satisfies the following condition:

$$\forall (v, u) \in E \Longrightarrow v \in S \vee u \in S$$

The *Minimum Weighted Vertex Cover* problem is to find a vertex cover with the minimum total weight.

Minimum Weighted Dominating Set Given a graph $G = (V, E)$, a dominating set (S) of the graph is a set of vertex satisfies the following condition:

$$\forall v \in V \Longrightarrow v \in S \vee (\exists u, (v, u) \in E \wedge u \in S)$$

The *Minimum Weighted Dominating Set* problem is to find a dominating set with the minimum total weight.

Apart from these optimization problems, we will also discuss about an important NP-hard satisfiable problem on graph.

Vertex Coloring Given a graph $G = (V, E)$, assign a color c_v to each vertex $v \in V$ such that the following holds:

$$\forall (v, u) \in E \Longrightarrow c_v \neq c_u$$

2.6 Tree decomposition and treewidth

Tree decomposition and treewidth were first introduced by Robertson and Seymour [44] in their fundamental work on graph minors.

DEFINITION 2 (Tree decomposition [21]). *A tree decomposition of a graph $G = (V, E)$ is a pair $(\{B_t, t \in I\}, T)$ where $B_t \subseteq V, I = \{1, ..., n\}$, and $T = (I, F)$ is a tree such that the following conditions are satisfied:*

- *the union of the subsets B_t equals the vertex set $V (1 \leq t \leq n)$, i.e. $\bigcup_{t \in I} B_t = V$;*
- *for every edge $(v, u) \in E$, there is an $t \in I$ with $u, v \in B_t$; and*
- *for every $v \in V$, if B_i and B_j contain v for some $i, j \in \{1, 2, ..., n\}$, then B_k also contains v for all k on the (unique) path in T connecting i and j. In other words, the set of nodes whose subsets contain v form a connected subtree of T .*

The subsets B_i are often referred to as *bags* of vertices. The *width* of a tree decomposition $(\{B_t, t \in I\}, T)$ is $max_{t \in I}|B_t| - 1$. The *treewidth* $\tau(G)$ of G is the minimum width over all tree decompositions of G. Figure. 1 shows an example of a tree decomposition of width two.

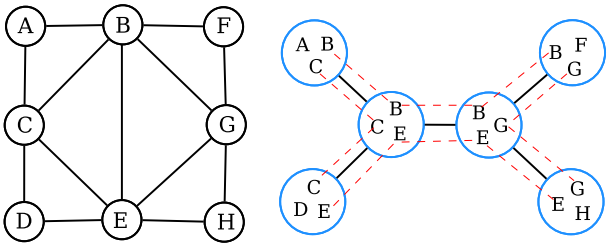

Figure 1: An example of a tree decomposition of width two: blue circles (big circles) denote the bags; red dashed lines connect the same vertices between adjacent bags.

2.7 Zippers on binary trees

A zipper [41] is a list whose elements are contexts that are left after a walk. Based on walking downward from the root of a tree, we construct a zipper as follows: when we go down-right from a node, we add its left child to the zipper; when we go down-left, we add the right child to the zipper. For example, Figure. 2 shows the correspondence between a zipper and a walk from the root to the black leaf.

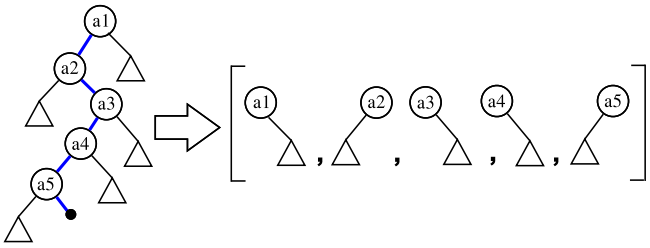

Figure 2: A zipper structure, which expresses a path from the root to the black leaf. The path is shown in the blue (thicker) line.

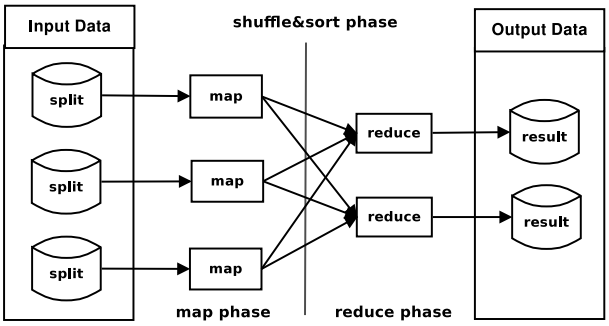

Figure 3: The MapReduce computation model

2.8 Functional description of MapReduce

To formally describe our MapReduce implementation later, we introduce the following functional description of MapReduce (as in [31]). As shown in Figure. 3, the MapReduce model consists of three phases: MAP, SHUFFLE&SORT and REDUCE.

To make the discussion precise, we introduce a specification of the MapReduce programming model in a functional programming manner. The standard programming interface of the MapReduce framework contains the following four functions. [1]

- Function f_{MAP} is invoked during the MAP phase and applied on each input key-value pair. Its type is defined as follows.

  $$f_{\text{MAP}} :: (k1, v1) \to [(k2, v2)].$$

  Function f_{MAP} takes a key-value pair and returns a list of intermediate key-value pairs.
- Function f_{HASH} is a parameter function for the shuffling and grouping process, which takes the key of an intermediate key-value pair, and generates a key with which the key-value pair is grouped. Its type is defined as follows.

  $$f_{\text{HASH}} :: k2 \to k3.$$
- Function f_{COMP} is a parameter function for the sorting process, which compares two keys in sorting the values in a group. Its type is defined as follows.

  $$f_{\text{COMP}} :: k2 \to k2 \to \{-1, 0, 1\}.$$
- Function f_{REDUCE} is invoked during the REDUCE phase, which takes a key and a list of values associated with the key and merges the values. Its type is defined as follows.

  $$f_{\text{REDUCE}} :: (k3, [v2]) \to (k3, v3).$$

[1] In order to distinguish them with the functions in Haskell, we change their names.

Now a functional specification of the MapReduce framework can be given as follows, which accepts four functions f_{MAP}, f_{HASH}, f_{COMP} and f_{REDUCE} and transforms a set of key-value pairs to another set of key-value pairs.

$$
\begin{aligned}
&\mathit{MapReduce}\ f_{\text{MAP}}\ f_{\text{HASH}}\ f_{\text{COMP}}\ f_{\text{REDUCE}}\ \mathit{input}\\
&\quad = \textbf{let}\ \mathit{sub1} = \mathit{map}_S\ f_{\text{MAP}}\ \mathit{input}\\
&\qquad\quad \mathit{sub2} = \mathit{map}_S\ (\lambda(k', \mathit{kvs}).\\
&\qquad\qquad\qquad (k', \mathit{map}\ \mathit{snd}\ (\mathit{sortKey}\ f_{\text{COMP}}\ \mathit{kvs})))\\
&\qquad\quad (\mathit{shuffleKey}\ f_{\text{HASH}}\ \mathit{sub1})\\
&\quad \textbf{in}\ \mathit{map}_S\ f_{\text{REDUCE}}\ \mathit{sub2}
\end{aligned}
$$

Function map_S is a set version of the map function: i.e., it applies the input function to each element in the set. Function *shuffleKey* takes a function f_{HASH} and a set of lists of key-value pairs, flattens the set, and groups the key-value pairs based on the new keys computed by f_{HASH}. The result type after *shuffKey* is $\{(k3, \{k2, v2\})\}$. Function *sortKey* takes a function f_{COMP} and a set of key-value pairs, and sorts the set into a list based on the relation computed by f_{COMP} .

3. Overview

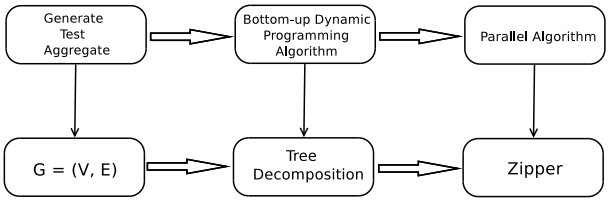

Figure 4: The transformation flow of our parallelization framework.

In this section, we introduce our high-level parallelization framework for solving graph optimization problems. Users only need to write the problem-specific code in the style of Generate-Test-Aggregate (GTA) [17] and our framework will automatically derive a parallel program to solve the corresponding problem. Everything related to the transformation of data structures and workload distribution would also be handled by the framework.

Figure. 4 shows the transformation flow of our parallelization framework, which presents the transformations both in the data structure level and the algorithm level. Given a graph with bounded treewidth and an algorithm on the graph which is defined using the GTA abstraction, we first derive a bottom-up dynamic programming algorithm from the GTA algorithm to reduce the computational complexity to polynomial time. Then we transform the bottom-up algorithm to a parallel algorithm on zipper to further speed up the computation.

3.1 Transformation of data structures

In the data structure level, our framework first transforms the input graph to a tree decomposition. As a zipper is a list of subtrees and a list provides a good characterization for divide-and-conquer parallel programs, our framework then transforms the tree decomposition to zipper structures.

Graphs to tree decompositions A tree decomposition is constructed from the input graph, which is an instance of the tree data structure. The data type for a tree decomposition is defined as:

$$\textbf{data}\ \mathit{Tree}\ b = \mathit{Node}\ b\ [\mathit{Tree}\ b] \mid \mathit{Leaf}$$

For an input graph with bounded treewidth w, the value of w can be recognized, and a corresponding width w tree decomposition be constructed in linear time[10]. The time dependence of this algorithm on w is exponential.

There are many existing algorithms and tools to construct tree decompositions. In our current framework, we use INDDGO [21] to generate tree decompositions for input graphs.

Tree decompositions to zippers Morihata et al. gave a definition of zipper on binary trees in [41]. As a tree decomposition is usually not a binary tree, we extend the definition of zipper on binary trees to that on tree decompositions. For a zipper on a tree decomposition, the elements in the list are trees with one hole.

A good feature of a tree decomposition is that, the order of the children of a node is not significant during the computation. Thus, we can consider the hole is always the rightmost child of a node. Figure. 5 shows a zipper on tree decomposition in this view.

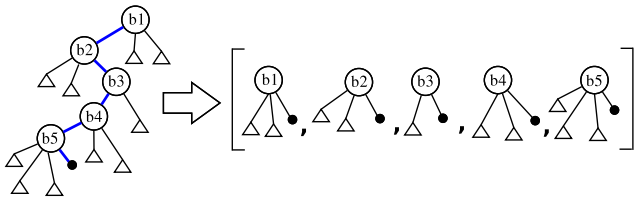

Figure 5: An example of zipper on tree decompositions, which expresses a path from the root to the black leaf. The path is shown in the blue (thicker) line.

The data type for the tree elements in the zipper can be defined as:

$$\textbf{data}\ \mathit{Tree}'\ b = \mathit{Node}'\ b\ [\mathit{Tree}\ b] \mid \mathit{Leaf}'$$

The zipper structures for tree decompositions can be specified in the following type.

$$\textbf{type}\ \mathit{Zipper}\ b = [\mathit{Tree}'\ b]$$

We use function $walk$ to construct a zipper from a tree decomposition.

$$\mathit{walk} :: \mathit{Tree} \to \mathit{Zipper}$$

To restore a zipper to a tree decomposition (not necessarily the original one), we use a leftward combination on the zipper to fill the hole of the previous element as the rightmost child.

$$
\begin{aligned}
&\mathit{z2t} && :: && \mathit{Zipper} \to \mathit{Tree}\\
&\mathit{z2t}\ [] && = && \mathit{Leaf}\\
&\mathit{z2t}\ ([\mathit{Node}'\ b\ ts] +\!\!+ l) && = && \mathit{Node}\ b\ (ts +\!\!+ \mathit{z2t}\ l\ )
\end{aligned}
$$

3.2 Transformation of algorithms

In the algorithm level, our framework derives bottom-up dynamic programming algorithms on tree decompositions from the user-specified Generate-Test-Aggregate (GTA) algorithms, then further transforms the bottom-up algorithms to parallel ones which run in a divide-and-conquer manner on the zipper.

GTA algorithms on graphs First, we use the GTA abstraction as interfaces to describe graph problems. The GTA, which is user-friendly and straightforward, represents three conceptual phases in solving a graph problem.

Generate is to generate all possible solution candidates for a graph problem. For instance, the *generate* function for Maximum-Weighted-Independent-Set (MWIS) is to list all subsets of the vertices of the input graph.

Test is to test if a solution candidate satisfies certain desired properties and filter out unsatisfied ones. For instance, the *test* function for MWIS is to test whether a subset of vertices is an independent set of the input graph.

Aggregate is to select a valid solution or make a summary of valid solutions with an aggregating computation. For instance, the *aggregate* function for MWIS is to find an independent set with the maximum total weight.

Bottom-up dynamic programming algorithm on tree decomposition Then we derive a bottom-up dynamic programming algorithm from the GTA algorithm.

In general, the algorithms to solve graph problems using tree decomposition have the following scheme. First, a tree decomposition of the input graph is constructed. Then, a dynamic programming algorithm is executed on the tree decomposition. For each node of the tree decomposition, a table is computed. For a decision problem, the table for the root of the tree T shows the answer.

B.Courcelle [14] showed a large set of problems that can be solved in polynomial time using tree decomposition when the graph is restricted to bounded treewidth. These problems are usually solved by bottom-up dynamic algorithms. Nevertheless, it is difficult to give a uniform algorithm to solve these problems.

By providing a uniform abstraction to describe graph problems, we are able to derive bottom-up dynamic programming algorithms for a class of interesting graph problems. The GTA functions defined by users are used as part of the bottom-up algorithms. Thus, the bottom-up algorithms for different graph problems have similar structures, the differences lie only in the user defined algorithms using the GTA abstraction.

The transformation from a GTA algorithm to a dynamic programming algorithm will be illustrated with an example in Section 4.

Parallel algorithm on zippers At last, we transform the bottom-up algorithms to parallel algorithms on zipper. Another advantage of zipper is that there are solid theoretical results to guarantee the correctness of using zipper to parallelize computation on tree decomposition.

Morihata et al. [41] proved that, for a problem on regular data structures, if two sequential algorithms, a bottom-up (upward) one and a top-down (downward) one, compute the same value, then there exists a divide-and-conquer parallel program that computes the same value as the sequential programs.

One feature of a tree decomposition of an undirected graph is that, the root of the tree decomposition is not fixed. So we may choose any node as the root, and the bottom-up dynamic programming algorithm will get the same result. Take the tree decomposition in Figure. 6 as an example, if we consider the node A as the root of the tree decomposition, we can write a bottom-up algorithm P; if we select the leaf node B as the root, then P can be considered as a top-down algorithm in the path along B to A. As the data type of a tree decomposition is regular, we can extend the parallelization result in [41] from tree to tree decomposition, using the third homomorphism theorem to guarantee the existence of divide-and-conquer parallel algorithms.

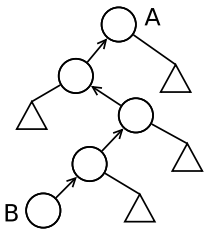

Figure 6: An example to help illustrate bottom-up and top-down in a tree decomposition.

Here, we extend the definition of decomposition on binary trees [41] to that on tree decompositions.

DEFINITION 3 (function decomposition on tree). *A decomposition of function $h :: Tree \to A$ is triple $(\phi, \odot, \varphi)$ that consists of associate operator $\odot :: B \to B \to B$ and two functions $\phi :: Tree' \to B$ and $\varphi :: B \to A$ such that*

$$\begin{array}{lcl} \varphi \circ h' & = & h \circ z2t \\ h'[] & = & \imath_\odot \\ h'[b] & = & \phi\, b \\ h'[x +\!\!+ y] & = & h'x \odot h'y. \end{array}$$

hold, where $\imath_\odot$ is the unit of $\odot$.

It is worth noting that a function decomposition h' can be seen as a list homomorphism (see DEFINITION 1) on zippers. Thus, if we can provide the three associative operators $(\phi, \odot, \varphi)$, we can get a scalable divide-and-conquer parallel program. The parallel program p can be expressed by *map* and *reduce* as:

$$p = \varphi \circ reduce\ (\odot) \circ map\ \phi$$

Our approach of the parallel algorithm on zipper is that, for each subtree in a zipper, we carry out the bottom-up algorithm in parallel to generate partial results for subtrees and then merge the partial results. One difficulty here is how to merge the partial results in a consistent and efficient way. We will present our approach via an example in Section 4.

4. Algorithm Parallelization Example

Maximum-Weighted-Independent-Set (MWIS) is a well-known NP-hard graph optimization problem. In this section, we will use the MWIS problem as an example to illustrate the algorithm transformations of our parallelization framework.

4.1 GTA algorithms on graphs

First, we express the algorithm ($mwis_G$) for the MWIS problem in the form of GTA. We use $g@(vs, es)$ to represent an instance of graph with vertex set as vs and edge set as es.

$$G = (V, E)$$

$$\begin{array}{l} generate :: [V] \to [[(V, Bool)]] \\ generate\ [] = [[]] \\ generate\ ([v] +\!\!+ vs) \quad = \quad [[(v, e)] +\!\!+ ls | \\ \qquad\qquad\qquad\qquad\quad e \leftarrow [True, False], \\ \qquad\qquad\qquad\qquad\quad ls \leftarrow generate\ vs] \end{array}$$

$$\begin{array}{l} test :: G \to [(V, Bool)] \to Bool \\ test\ g@(vs, es)\ xs \quad = \quad \bigwedge /[\neg((v, True) \in xs \\ \qquad\qquad\qquad\qquad\quad \wedge (u, True) \in xs) | \\ \qquad\qquad\qquad\qquad\quad (v, u) \leftarrow es] \end{array}$$

$$\begin{array}{l} weight :: [(V, Bool)] \to Int \\ weight\ xs = +/[w(v) | (v, b) \leftarrow xs, b == True] \end{array}$$

$$\begin{array}{l} mwis_G :: G \to Int \\ mwis_G\ g@(vs, es) \quad = \quad max[weight(xs) | \\ \qquad\qquad\qquad\qquad\quad xs \leftarrow generate\ vs, \\ \qquad\qquad\qquad\qquad\quad test\ g\ xs] \end{array}$$

Here, $\oplus/$ is defined as:

$$\oplus/[a_1, a_2, ..., a_n] = a_1 \oplus a_2 \oplus ... \oplus a_n$$

For each vertex v in V, we use $True$ (or $False$) to annotate the selecting state of v. A selection set of a set of vertices V, is a list of all the v in V and its corresponding selecting state, i.e. $[(V, Bool)]$. Function $generate$ lists all the possible selection sets. Function $test$ accepts two parameters, a graph G and one of its

selection set xs, and decides whether xs satisfies the property that it is an independent set of the graph. Function $weight$ computes the total weight of all the selected vertices in a selection set. The $aggregate$ function in MWIS is to find the one with the maximum weight of all the selection sets.

If we naively compute a graph problem in the form of GTA, it takes exponential time of the input size, as GTA is actually a brute-force approach. In this example, it takes $O(2^{|V|})$ time to solve the MWIS problem on graph $G = (V, E)$ using the GTA algorithm.

4.2 Bottom-up algorithm on tree decomposition

Using bottom-up dynamic programming algorithms on tree decomposition to solve MWIS has been discussed in [11, 21]. We first follow the idea and derive a bottom-up function $mwis$ on tree decomposition using the GTA functions as part of the implementation. We define g_{b_t} as the induced subgraph of g on vertices in b_t.

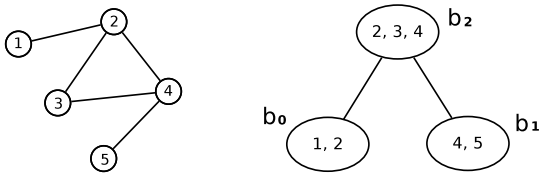

Figure 7: An example to help illustrate the bottom-up algorithm to solve MWIS. The number in a circle in the left graph represents both the id and the weight of the node. Each node in the right graph is a bag of vertices in the left graph.

$$
\begin{aligned}
mwis\ (Node\ b_t\ [])\ &=\ [(xs, weight\ xs) \mid \\
&\quad xs \leftarrow generate\ b_t, \\
&\quad test\ g_{b_t}\ xs] \\
\\
mwis\ (Node\ b_t\ children)\ &=\ [(xs, weight\ xs + \\
&\quad (+/[inherit\ xs\ t' \mid \\
&\quad t' \leftarrow children])) \mid \\
&\quad xs \leftarrow generate\ b_t, \\
&\quad test\ g_{b_t}\ xs] \\
\\
\textbf{where}\ inherit\ xs\ t'\ &=\ max[(value' - \\
&\quad weight(xs \cap xs')) \mid \\
&\quad (xs', value') \leftarrow mwis(t'), \\
&\quad consistent(xs, xs')] \\
consistent(xs, xs')\ &=\ \bigwedge /[(b == b') \mid \\
&\quad (v, b) \leftarrow x, \\
&\quad (v', b') \leftarrow x', \\
&\quad v == v']
\end{aligned}
$$

$$mwis_{value}\ tree = max\ (map\ snd(mwis\ tree))$$

For example, in Figure. 7, if we run function $mwis$ on leaf b_0, it first generates all the possible selection sets of vertices in node b_0, then tests if they are independent sets in the induced graph g_{b_0}. Function $mwis$ returns a list of tuples of selection set xs and its corresponding weight sum. For example, on leaf b_0:

$$mwis\ (Node\ b_0\ []) = [([], 0), ([1], 1), ([2], 2)]$$

Similarly, on leaf b_1 :

$$mwis\ (Node\ b_1\ []) = [([], 0), ([4], 4), ([5], 5)]$$

Carrying out function $mwis$ on node b_2, similar to the procedure on a leaf node, first generates and tests all the possible independent sets in the induce graph g_{b_2}, then function $inherit$ is used to pass up the weight contributions of the vertices in its child nodes. For each independent set xs, we choose the $consistent$ selection set which maximizes the contributed weight in each of its children. Here, $consistent$ means for the same vertices appearing in two different nodes, the selecting states should be the same.

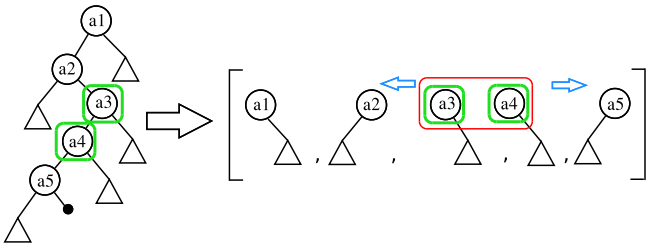

Figure 8: An example to show computation on zipper.

$$
\begin{aligned}
&mwis\ (Node\ b_2\ [b_0, b_1]) = \\
&\quad [([], 0 + 1 + 5), ([2], 2 + 5), ([3], 3 + 1 + 5), ([4], 1 + 4)]
\end{aligned}
$$

If the treewidth is w, there are at most $2^{(w+1)}$ many generating marking ways on each node, and there are $O(|V|)$ many nodes in a tree decomposition. The MWIS problem can be solved in $O(|V| \cdot 2^{(w+1)})$ time using the bottom-up algorithm.

4.3 The parallel algorithm on zippers

In the bottom-up algorithm, we need to remember the selecting state xs in the root of current subtree, which is used for the further computation of ancestors (testing $consistent$ condition). While on zipper, as shown in figure 8, we should remember the marking way of the leftmost and the rightmost subtree roots, for the leftward and rightward merging of partial results in zipper.

We first duplicate the selecting state at the root of each subtree:

$$mwis'\ tree = [(xs, xs, value) | (xs, value) \leftarrow mwis\ tree]$$

We modify the bottom-up function $mwis$ on tree decomposition t to get a leftward sequential function $mwis_{up}$ on t's corresponding zipper.

$$
\begin{aligned}
&mwis_{up} = mwis' \circ z2t \\
&mwis_{up}[(Node'\ b_t\ children)] = mwis'(Node\ b_t\ children) \\
&mwis_{up}([a] +\!\!+ ls)\ =\ [(xs_a, xs'_{ls}, value_a + value_{ls} \\
&\quad - weight(xs_a \cap xs'_{ls})) \mid \\
&\quad (xs_a, xs'_a, value_a) \leftarrow mwis_{up}\ a, \\
&\quad (xs_{ls}, xs'_{ls}, value_b) \leftarrow mwis_{up}\ ls, \\
&\quad consistent\ xs'_a\ xs_{ls}]
\end{aligned}
$$

If the root of the rightmost subtree of zipper is considered as the root of its original tree decomposition, similar to $mwis_{up}$, we can compute the Maximal-Weighted-Independent-Set in rightward manner (similarly to $mwis_{up}([a] +\!\!+ ls)$, we can get $mwis_{down}(ls +\!\!+ [a])$ from $mwis_{down} ls$ and $mwis_{down} a$). When a function can be evaluated in both leftward and rightward manners, the third homomorphism theorem guarantees the existence of a parallel algorithm.

We, therefore, construct a parallel algorithm in the following way, with the definition of associative operator $\odot$:

$$mwis_{par} = mwis' \circ z2t$$

$$mwis_{par}[(Node'\ b_t\ children)] = mwis'(Node\ b_t\ children)$$

$$\odot :: [[(V, Bool)]] \to [[(V, Bool)]] \to [[(V, Bool)]]$$

$$
\begin{aligned}
mwis_{par}(a +\!\!+ b) &= mwis_{par}\ a \odot mwis_{par}\ b \\
&= [(xs_a, xs'_b, value_a + value_b \\
&\quad - weight(xs'_a \cap xs_b))\,| \\
&\quad (xs_a, xs'_a, vlaue_a) \leftarrow mwis_{par}\ a, \\
&\quad (xs_b, xs'_b, value_b) \leftarrow mwis_{par}\ b, \\
&\quad consistent\ xs'_a\ xs_b]
\end{aligned}
$$

$$mwis_{value}\ tree = max\ (map\ thd\ mwis_{par}(walk\ tree))$$

For each subtree, we can use function $mwis'$ to compute the partial results in parallel. Independent sets of two successive lists can be merged, if the selecting states of the rightmost root of the left list and the leftmost root of the right list are *consistent*.

If there are p processors, and the size of zipper is n, it takes $O(|V| \cdot 2^{(w+1)}/p)$ time to compute the result of sub-list in parallel. A merging of two sub-list result takes $O(2^{2(w+1)})$ many computations. It takes $O((n \log n)/p \cdot 2^{2(w+1)})$ in the merging procedure. From the practical view, the merging procedure is much faster, as the size of the pairs of selecting state can be largely reduced with the *consistent* condition.

5. Tree Parallelization

In this section, we present how to parallelize the computations on tree decompositions. We discuss in detail how to partition a tree decomposition to a zipper-based structure and how to apply parallel algorithms on this zipper-based structure to the MapReduce model.

As a tree decomposition is an instance of the tree structure, we will not distinguish between *tree decomposition* and *tree* in this section.

5.1 Overview

There are three basic approaches to parallelizing the computations on trees: the leaf-level bottom-up approach, the tree contraction/reduction approach and the divide-and-conquer approach. The leaf-level bottom-up approach, in which parallelization is only on the leaf nodes, performs bad on ill-balanced trees (such as the monadic tree). Tree contraction, which requires a set of operations to satisfy a certain condition, is hard to use [41]. And tree contraction is mainly designed for the shared memory environment. The divide-and-conquer approach, which partitions a tree into subtrees and computes independent subtrees in parallel, is suitable for modern parallel environments such as distributed memory environment and cloud.

A zipper is a path from the root node to a leaf node. A tree can be partitioned into subtrees along the path. However, the nodes of subtrees in a zipper may be imbalanced, making the parallel program inefficient. To this end, we propose a concept of recursive partition on zipper to achieve good partition on trees.

5.2 Tree partition

Tree partition is an important part of the divide-and-conquer approach. There are two goals in partitioning a tree: one is to partition a tree evenly, so that the tree can be computed in parallel with good load balance; the other is to minimize the dependencies between partitioned trees so that communication between processors can be decreased.

From Figure. 2 we can see that, all the subtrees in a zipper have a uniform structure: each subtree has a hole and the hole is either the left child or the right child of the root node. This feature not only provides a uniform way to design algorithms on subtrees but also limits dependencies only to two adjacent subtrees in a zipper.

To achieve the two goals in partitioning a general tree, we extend the zipper for binary trees to a *hierarchical zipper* for general trees. Our idea is: to keep a uniform structure, we only choose the leftmost child or the rightmost child when selecting a path from the root to a leaf node; if the size of a subtree in a zipper is larger than a *threshold*, we partition the subtree again to a new zipper. Such recursive partition forms a *hierarchical zipper* (see Figure. 9) which is a tree. Each node in the tree is a zipper.

Path selection strategy Here, we describe two strategies in walking downward from the root node to a leaf node.

Random strategy. We randomly pick the leftmost child or the rightmost child as the next node in the path. In this strategy, we don't need preprocessing on the tree.

Maximum descendants strategy. Each time, we choose the child node with the maximum number of descendants. This strategy can decrease the height of the resultant hierarchical zipper tree in most cases. However, this strategy needs preprocessing on the tree: for each node, we need to record the size of the tree rooted at the node, i.e. the number of descendants.

Deciding threshold In our partition, we limit the size of each subtree to a threshold. The threshold is decided using the following equation:

$$T = N/(P * 2)$$

where N is the number of tree nodes, and P is the number of processors and T is the threshold value.

Underlying implementation We describe our underlying implementations of the hierarchical zipper. Figure. 9 gives an example of a hierarchical zipper.

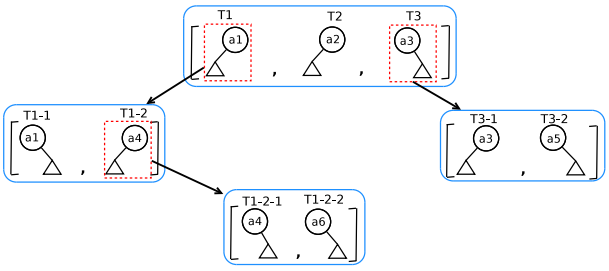

Figure 9: An example of a hierarchical zipper: the subtrees in red dotted rectangles are partitioned to new zippers pointed by the arrows; the id of a subtree is shown on its top.

Subtree id. Each subtree in the hierarchical zipper has an id in the form of X-Y. X is the id of the zipper the subtree belongs to and Y is its index in the zipper. The id of a zipper is the same with the subtree the zipper is partitioned from.

We add a T to the head of an id for easy description. For example, in Figure. 9, the id T1-2-1 means the subtree is the first element in the zipper for subtree T1-2.

Subtree flag. Each subtree has a flag which records whether a subtree is the last element in a zipper. This flag is used in the combination of the results of subtrees in a zipper. The combination is finished if the result of the last subtree in the zipper has been combined, For example, in Figure. 9, the flag for subtree T3-1 is false because T3-1 is the first element in the zipper.

Storage. All the subtrees are stored in a list in a post-order traversal of the hierarchical zipper. The partitioned subtrees (in red dotted rectangles) are not collected. For example, the subtrees in Figure. 9 are stored as [T1-2, T1-2-1 ,T1-2-2 ,T2, T3-1, T3-2].

5.3 Parallel algorithm on hierarchical zipper

To provide a parallel algorithm on the hierarchical zipper, we prepare the following four functions. In the following, $Tree'$ is the type of a subtree, B is the type of the intermediate result of a subtree and A is the type of final result for the problem.

- $compute :: Tree' \to B$

  Compute and return the intermediate result of a subtree. This function corresponds to the ϕ operation in DEFINITION 3.

- $combine :: B \to B \to B$

  Merge the results of two subtrees. This function corresponds to the $\odot$ operation in DEFINITION 3.

- $recover :: B \to B \to B$

  Recover from the combined result of a zipper to the result of its original subtree.

- $extract :: B \to A$

  Compute the final result of the complete tree from the result of a hierarchical zipper. This function corresponds to the φ operation in DEFINITION 3.

The recover function When we design algorithms on zipper, we usually compute auxiliary information to help to combine subtrees. For example, the height example in [41] computes the height of a subtree as its first result and the depth of the hole as its second result. For a subtree t with height h, the result of the subtree $compute\ t$ is $(h, 1)$. However, if we partition the subtree to another zipper z and combine the results of all the subtrees, i.e. $reduce\ combine\ z$, the result tuple will be (h, x). Here, x is the final depth of the hole, which equals the size of subtrees in the zipper. Thus, we need a $recover$ function to guarantee that the result on zipper can be recovered to the result on the subtree.

On the other hand, when a subtree is partitioned to a zipper, the root node of the subtree becomes the root node of the first subtree in the zipper (see node $a1$ in Figure. 9). As the hole is always in the root node and the first subtree contains the root node, we can recover the result from the result on zipper and the result of the first subtree in the zipper.

Algorithm description The parallel algorithm on hierarchical zipper consists of a map process and a reduce process.

Map process. In the map process, we perform the $compute$ function on each subtree and passes the intermediate results to the reduce process.

Reduce process. In the reduce process, we group the received intermediate results by zipper id and sort the elements in each group by index. In each group, we apply the $combine$ function on intermediate results with consecutive indices. If all the results in a group have been combined, we use the $recover$ function to recover the result and sent the result to the reduce process. The reduce process is repeated until the result of the top-level zipper has been computed. Then the $extract$ function is performed to compute the final result.

Apply to the MapReduce model We show how to apply the parallel algorithm to the MapReduce model in an iterative manner. We divide the MapReduce passes (rounds) into a working pass and iterative passes. The iterative pass repeats until the top-level zipper has been computed. In the following, we summarize the two kinds of MapReduce passes.

Here, K is the type of the subtree id. The *split* function splits an id into a zipper id and an index, and return them in a pair with the zipper id as the first result. The *comp* function will return 1 if the first argument is greater than the second, 0 if the two arguments are equal and -1 otherwise.

The working pass of MapReduce. The first pass of MapReduce is the working pass, which computes the results of all the subtrees and combines parts of the results. The input to the MAP phase is a list of key-value pairs of ids and subtrees, while the f_{MAP1} function takes one pair and performs $compute$ on the subtree. In the SUFFLE&SORT phase, the f_{HASH} function is used to group results by zipper id and the f_{COMP} function is used to sort the elements in each group by index. Finally, the REDUCE phase combines the results in each group. The working pass of MapReduce can be represented as follows:

$$
\begin{aligned}
&MapReduce\ f_{\text{MAP1}}\ f_{\text{HASH}}\ f_{\text{COMP}}\ f_{\text{REDUCE}}\\
&\textbf{where}\\
&\quad f_{\text{MAP1}} :: (K, Tree') \to [(K, B)]\\
&\quad f_{\text{MAP1}}\ (k, t) = [(k, compute\ t)]\\
&\quad f_{\text{HASH}} :: K \to K\\
&\quad f_{\text{HASH}}\ k = fst\ (split\ k)\\
&\quad f_{\text{COMP}} :: K \to K \to \{-1, 0, 1\}\\
&\quad f_{\text{COMP}}\ k1\ k2 = comp\ (snd\ (split\ k1))\ (snd\ (split\ k2))\\
&\quad f_{\text{REDUCE}} :: (K, [B]) \to (K, B)\\
&\quad f_{\text{REDUCE}}\ (k, as) = (k, recover\ (reduce\ combine\ as))
\end{aligned}
$$

The iterative passes of MapReduce. Other passes of MapReduce except the first one are iterative passes. The iterative passes combine remaining parts of the results. In an iterative pass, the MAP phase does no computation and the other two phases are the same as in the working pass. The iterative pass of MapReduce can be represented as:

$$MapReduce\ ([\cdot])\ f_{\text{HASH}}\ f_{\text{COMP}}\ f_{\text{REDUCE}}$$

Result extraction. When all the MapReduce passes end, we get a result key-value pair (k, b). Then the $extract$ function is applied to compute the final result, which is represented as:

$$extract \circ snd$$

6. More Examples

6.1 More optimization problems

Using the GTA abstraction, we can parallelize algorithms solving more optimization problems such as vertex cover and independent set with constraints.

Similar to the MWIS problem, we first use the $generate$ function to list all the possible selecting set of vertices, then use the $test$ function, which indicates the properties of the selected set, to filter the legal marking way. As for the minimal vertex cover set problem, the $test$ function can be expressed as:

$$
\begin{aligned}
&test :: G \to [(V, Bool)] \to Bool\\
&test\ g@(vs, es)\ xs \quad = \quad \bigwedge / [(v, True) \in xs\\
&\qquad\qquad\qquad\qquad \vee (u, True) \in xs |\\
&\qquad\qquad\qquad\qquad (v, u) \leftarrow es]
\end{aligned}
$$

Its $aggregate$ function is min.

The $test$ function could be a combination of several properties. For example, if we want to compute the maximal weighted independent set of even weighted vertices, the $test$ function could be defined as:

$$
\begin{aligned}
&test :: G \to [(V, Bool)] \to Bool\\
&test_1\ g@(vs, es)\ xs \quad = \quad \bigwedge / [\neg((v, True) \in xs\\
&\qquad\qquad\qquad\qquad \wedge (u, True) \in xs) |\\
&\qquad\qquad\qquad\qquad (v, u) \leftarrow es]\\
&test_2\ g@(vs, es)\ xs \quad = \quad \bigwedge / [w(v)\%2 == 0 |\\
&\qquad\qquad\qquad\qquad (v, b) \leftarrow xs,\\
&\qquad\qquad\qquad\qquad b == True]\\
&test\ g@(vs, es)\ xs \quad = \quad (test_1\ g@(vs, es)\ xs)\\
&\qquad\qquad\qquad\qquad \wedge (test_2\ g@(vs, es)\ xs)
\end{aligned}
$$

Then the corresponding $aggregate$ function is max.

Further, we can rewrite function $mwis$ and $mwis_{par}$ with the new defined $test$ function to derive parallel algorithms for the corresponding problems.

6.2 The vertex coloring problem

Vertex coloring is an important NP-hard satisfiable problem on graph. For graph of bounded treewidth, H. Bodlaender [9] proposed how to compute vertex coloring in polynomial time using tree decomposition. We briefly present here how to express k vertex coloring in form of GTA, and how to use our parallel framework to solve it.

$$
\begin{aligned}
&G = (V, E)\\
&\textbf{data } Color = C_1|C_2|...|C_k\\
\\
&generate :: [V] \to [[(V, Color)]]\\
&generate [] \quad = \quad [[]]\\
&generate\ ([v] +\!\!+ vs) \quad = \quad [[(v, e)] +\!\!+ ls|\\
&\qquad\qquad\qquad\qquad e \leftarrow [C_1, C_2, ..., C_k],\\
&\qquad\qquad\qquad\qquad ls \leftarrow generate\ vs]\\
\\
&test :: G \to [(V, Bool)] \to Bool\\
&test\ g@(vs, es)\ xs \quad = \quad \bigwedge /[c_v \neq c_u|\\
&\qquad\qquad\qquad\qquad (v, u) \leftarrow es,\\
&\qquad\qquad\qquad\qquad (v, c_v) \leftarrow xs,\\
&\qquad\qquad\qquad\qquad (u, c_u) \leftarrow xs]\\
\\
&weight :: [(V, Bool)] \to Int\\
&weight\ xs = 0\\
\\
&exist\ t \quad = \quad if\ t \neq \varnothing\ then\ 0\\
&\qquad\qquad\quad else\ -\infty\\
\\
&mwis_G :: G \to Int\\
&mwis_G\ g@(vs, es) \quad = \quad exist\ [weight(xs)|\\
&\qquad\qquad\qquad\qquad xs \leftarrow generate\ vs,\\
&\qquad\qquad\qquad\qquad test\ g@(vs, es)\ xs]
\end{aligned}
$$

For a graph G, the $generate$ function lists all the possible coloring ways of the vertices, and the $test$ function filters legal coloring ways. If there is a legal coloring way, we use the aggregate function $exist$ to set its value as 0, otherwise as $-\infty$. A parallel algorithm can be derived by rewriting the one of MWIS with a new definition of $generate$, $test$ and $aggregate$ functions. More specifically, we use $exist$ to replace max as the $aggregate$ function, which can guarantee the if an induced subgraph cannot be k-colored (the aggregated weight value is $-\infty$), then the whole graph can not be k-colored (summation of $-\infty$ with anything is $-\infty$). However, the parallel algorithm for vertex coloring generated by our general framework is not optimal. For more optimization for vertex coloring algorithm, one can refer to [9].

Many practical problems such as job scheduling [34] and register allocation in program analysis [12] can be reduced to vertex coloring. A parallel algorithm of vertex coloring is promising to efficiently solve these problems.

6.3 Further discussion on the domination-type problems

Our framework can solve graph combinatorial problems whose selected vertices xs satisfy $test$ on graph G and a subset of xs also satisfies $test$ on the corresponding induced subgraph.

For the minimum dominating set problem, it can be expressed in the form of GTA, but the generated parallel algorithm would be incorrect if we naively use the approach we do for MWIS. We have observed the following fact: for a selection set xs failing certification of $test$, which contains vertices that are neither of selecting state nor dominated by vertices of selecting state, might become legal in the further computation as the non-dominated vertices might be dominated by vertices appearing in the ancestor nodes. That means, we will miss some selection set candidates for the minimal dominating set problem if we do parallelization like that for MWIS.

The minimum dominating set problem can be efficiently solved via tree decomposition [2, 47]. Telle [47] showed how to describe domination-type problems with the state of each vertex and its neighbors. In order to parallelize a class of domination-type problems, we will introduce function $test\_alive$, which is to remember some selection sets that fail certification of $test$ for the current state but might satisfy $test$ later.

Due to the limitation of scope, here we mainly present our original idea of parallelizing graph program via tree decomposition and introduce a high-level parallelization framework. One important part of our future work is to extend the domain of graph problems of our parallelization approach, which includes solving dominating-like problems with parallel algorithms and deriving $test\_alive$ function from $test$ automatically.

7. Evaluation

As a proof of concept and feasibility, we experimentally validate our approach through a very preliminary experiment.

7.1 Experiment environment

All experiments in this section are performed on a Linux (Ubuntu 12.04 64-bit) compute node equipped with 8GB of RAM and two processors each with 4 cores (Intel(R) Xeon(R) CPU E5620 @ 2.40GHz). All our experiments are conducted on an implementation of the MapReduce model using Java multi-thread. Results are averaged over 5 iterations.

7.2 Experiment on graph problems

We conducted experiments on the Maximum Weighted Independent Set problem.

Graph Data Our graph data is a partial k-tree generated by keeping 100% of the edges from a random 10-tree on 100,000 nodes [21]. The graph has 100,000 nodes and 999,945 edges, and the treewidth is 10.

We use an open-source tool, INDDGO [21], to construct tree decomposition of the graph. After construction, the tree decomposition of the graph has 72,523 nodes. The height of the tree decomposition is 22 and the maximum degree is 149.

Results We compare our approach (MWIS-par) with a Java-implemented sequential program (MWIS-seq) using the dynamic programing algorithm described in [21].

The running time result of this experiment is shown in Figure. 10.

The speedup over the sequential program (MWIS-seq) is shown in Figure. 11. From the figure we can see that, the parallel algorithm for the MWIS problem achieves a nearly linear speedup over the sequential version.

The memory overhead of this experiment is shown in Figure. 12. As the number of subtrees increases with cores and we need to duplicate the selecting state at the root of each subtree for merging partial results in zipper both leftward and rightward, our approach requires more memory as the cores increases.

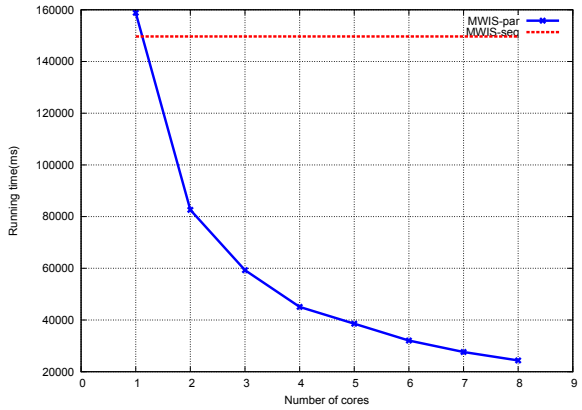

Figure 10: Running time of the MWIS problem with cores.

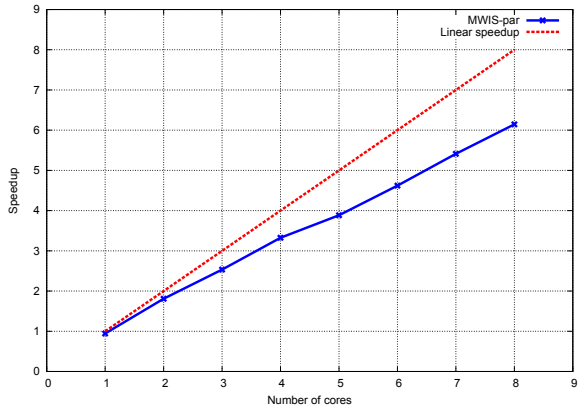

Figure 11: Speedup of the MWIS problem with cores.

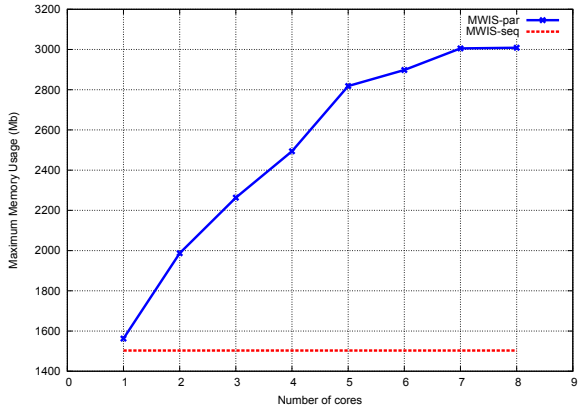

Figure 12: Memory usage of the MWIS problem with cores.

8. Related Work

In this section, we discuss some related work in the areas of graph and tree parallelization, and tree decomposition and treewidth.

8.1 Graph parallelization

Graph parallelization, especially on large scale graphs, has been studied intensively in recent years. There are two directions in this research that are most related to our work.

Graph-parallel abstractions Several works proposed graph-parallel abstractions. Pregel [33] is a bulk synchronous message passing abstraction in which all vertex-programs run simultaneously in a sequence of super-steps. GraphLab [32] is an asynchronous distributed shared-memory abstraction in which vertex-programs have shared access to a distributed graph with data stored on every vertex and edge. Gonzalez et al. [20] showed that the natural graphs commonly found in the real-world have power-law degree distributions, which challenge the assumptions made by these abstractions. So they proposed the PowerGraph [20] abstraction which exploits the Gather-Apply-Scatter model of computation to factor vertex-programs over edges, splitting high-degree vertices and exposing greater parallelism in natural graphs. Our works differ with theirs in that our Generate-Test-Aggregate abstraction is designed to describe graph problems, such as graph optimization problems, while their abstractions are designed to specify general computations on graphs.

Solving graph problems in parallel Silvio Lattanzi et al. [29] presented a design technique called filtering, which uses the parallelization of MapReduce to reduce the size of the input so that the resulting problem instance can be solved on a single machine. They gave algorithms for several graph problems such as minimum spanning trees, maximal matchings, approximate weighted matchings, approximate vertex and edge covers and minimum cuts. However, their algorithms for graph problems are still on graphs while ours are transformed to algorithms on tree decompositions. Sullivan et al. [46] proposed a task-oriented parallel bottom-up dynamic programming algorithm on tree decomposition to solve the maximum weighted independent set problem. Their approach is shared-memory environment centered and would be inefficient if the shape of the tree decomposition is imbalanced. Our approach, however, can derive a parallel divide-and-conquer algorithm with good load balance.

8.2 Tree decomposition

Recently, tree decomposition and treewidth get growing attention from researchers. Hicks et al. [23] gave a general overview of tree decomposition technique for discrete optimization. Though tree decomposition is the basis of our work, how to construct tree decompositions from graphs is not our focus in this paper.

Tree decomposition and treewidth Arnborg et al. [3] proved that determining whether a graph G has a treewidth of at most k is NP-complete. But for an input graph with bounded treewidth w, the value of w can be recognized, and a corresponding width w tree decomposition constructed in linear time[10]. Existing algorithms for determining or approximating treewidth can be categorized into exact algorithms, upper bound algorithms and lower bound algorithms. Overviews of these algorithms are given in [11, 23]. Fu [18] gave a detailed description of these algorithms, implemented some of the algorithms and made a comparison of the implemented algorithms by category through experiments. Groer et al. [21] also made a comparison of the performance of elimination ordering heuristics on a set of test graphs. Sullivan et al. [46] presented their implementation for parallelizing the construction of tree decomposition.

Applications of tree decomposition or treewidth The probably first tree decomposition based algorithm that has been shown of practical interest is given by Lauritzen and Spiegelhalter [30]. They solve the inference problem for probabilistic (or Bayesian belief) networks by using tree decompositions. Ogawa et al. [42] proposed

an approach for program analysis, such as dead code detection and register allocation, through recursive graph traversal instead of iterative procedures based on the fact that most programs have well-structured control flow graphs, i.e. graphs with bounded tree width. Wei [49] proposed a method to answer shortest-path queries based on tree decompositions. Akiba et al. [1] designed algorithms and data structures for efficient shortest-path query processing for two specific classes of graphs: graphs with small treewidth and complex networks. However, they didn't consider about parallelization of their algorithms.

8.3 Tree parallelization

Using tree decomposition, we are able to reduce algorithms for some graph problems to algorithms on tree decompositions (trees). Thus, existing tree parallelization techniques can be used to parallelize computations on the tree decomposition, so as to parallelize the graph problems. Here, we give a brief overview of these techniques.

Parallel tree contraction Tree contraction, which was first proposed by Miller and Reif [40], is a useful framework for developing parallel programs on trees, and many computations have been implemented on it. However, parallel tree contraction is hard to use, because it requires a set of operations that satisfy a certain condition [41]. To this end, Matsuzaki et al. [37] proposed a systematic method of deriving efficient tree contraction algorithms from recursive functions on trees.

Parallel tree reduction Tree reductions are often implemented with a tree contraction algorithm. Matsuzaki et al. [39] developed a code generation system based on tupled-ring property to automatically transform user's recursive reduction programs with annotations into parallel programs. Kakehi et al. [25, 26] developed a framework for parallel reductions on trees over distributed memory environment by exploiting serialized trees as the data representation and a property called extended distributivity. Emoto and Imachi [16] proposed a MapReduce algorithm for tree reductions and implemented it on Hadoop.

Parallel tree skeleton Parallel skeletons provide parallelizable computational patterns in a concise way and conceal the complicated parallel implementations from users. Skillicorn [45] first formalized a set of binary-tree skeletons. Matsuzaki et al. [35] proposed an implementation of these parallel tree skeletons on binary trees on distributed systems. Matsuzaki et al. [36] also proposed two parallelization transformations to help programmers to systematically derive efficient parallel programs using tree skeletons. Later, they presented rose trees in the form of binary trees and proposed a set of rose-tree skeletons [38] which are implemented on their binary-tree skeleton library.

Homomorphism-based parallelization Skillicorn [45] modeled operations on structured text such as XML using parameterized tree homomorphism functions on binary trees. Morihata et al. [41] generalized the third homomorphism theorem [19] to trees and developed a method for systematically constructing scalable divide-and-conquer parallel programs on trees from two sequential programs. Our approach is based on their idea.

Tree partition Tree partition plays an important role in the divide-and-conquer approach. A zipper is a list of trees, which idea was first described by Huet [24] in 1997. Morihata et al. [41] considered a zipper as a one-hole context and proposed recursive division on one-hole contexts to divide a tree. Our tree partition approach is also based on zipper, but in a different division strategy. M-bridge is another approach to partition a tree. Miller et al. [43] gave definitions of M-bridge and proved some properties of M-bridge. M-bridge finds a set of vertices that subdivide a tree into independent subtrees of approximately equal size. It will be interesting to see if we can make use of this approach in our tree partition process.

9. Conclusions and Future Work

In this paper, we present an approach to transforming bottom-up dynamic programming algorithms on tree decomposition to parallel algorithms on zipper. As far as we know, our approach is the first one to parallelize computations on tree decompositions in a divide-and-conquer manner with good load balance, which is suitable for the MapReduce model. We also introduce the GTA abstraction for easy programming of graph problems. Our proposed parallelization framework can transform the user-specified GTA programs to efficient parallel programs automatically. Our preliminary results show that the algorithms we proposed are not only interesting from a theoretical viewpoint, but also are viable and useful in practice.

We aim to solve practical problems via tree decomposition and tree parallelism. Well structured programs are proved to have a small treewidth [22, 48], and the notions of tree decomposition and treewidth provide a new and efficient approach for program analysis [42]. Tree decomposition is also potential to promote computing in social networks. The target set selection problem [5] in social networks can be reduced to independent-like or domination-like problem on graph. We believe our parallelization framework is towards solving such practical problems.

For future work, as discussed in Section 6.3, we are actively extending the domain of graph problems of our parallelization approach. We also plan to implement the parallelization framework as a library, for example on Hadoop [50], so that large scale graph problems can be tackled in distributed memory environments. As the graphs we considered in this paper are undirected graphs, our another future work is to extend our framework to directed graphs.

References

[1] T. Akiba, C. Sommer, and K.-i. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 144–155, 2012.

[2] J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *LATIN 2002: Theoretical Informatics*, pages 613–627. Springer, 2002.

[3] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[4] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, Apr. 1991.

[5] O. Ben-Zwi, D. Hermelin, D. Lokshtanov, and I. Newman. Treewidth governs the complexity of target set selection. *Discrete Optimization*, 8(1):87–96, 2011.

[6] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2 edition, May 1998. ISBN 0134843460.

[7] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, 1987. ISBN 0-387-18003-6.

[8] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, Mar 2009. ISSN 1552-6283.

[9] H. L. Boblaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *J. Algorithms*, 11(4):631–643, Dec. 1990.

[10] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25 (6):1305–1317, 1996.

[11] H. L. Bodlaender and A. M. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[12] G. J. Chaitin. Register allocation & spilling via graph coloring. In *ACM Sigplan Notices*, volume 17, pages 98–105. ACM, 1982.

[13] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engg.*, 11(4):29–41, July 2009. ISSN 1521-9615.

[14] B. Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

[15] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[16] K. Emoto and H. Imachi. Parallel tree reduction on MapReduce. *Procedia Computer Science*, 9:1827–1836, 2012.

[17] K. Emoto, S. Fischer, and Z. Hu. Generate, test, and aggregate: a calculation-based framework for systematic parallel programming with mapreduce. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pages 254–273, 2012.

[18] Y. Fu. Computing the treewidth of graphs, 2011.

[19] J. Gibbons. The third homomorphism theorem. *J. Funct. Program.*, 6 (4):657–665, 1996.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. OSDI'12, pages 17–30, 2012.

[21] C. S. Groer, B. D. Sullivan, and D. P. Weerapurage. INDDGO: Integrated network decomposition & dynamic programming for graph optimization. Technical Report ORNL/TM-2012/176, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2012.

[22] J. Gustedt, O. A. Mæhle, and J. A. Telle. The treewidth of Java programs. In *Algorithm Engineering and Experiments*, pages 86–97. Springer, 2002.

[23] I. V. Hicks, A. M. C. A. Koster, and et al. Branch and tree decomposition techniques for discrete optimization, 2005.

[24] G. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, Sept. 1997. ISSN 0956-7968.

[25] K. Kakehi, K. Matsuzaki, K. Emoto, and Z. Hu. A practicable framework for tree reductions under distributed memory environments. Technical report, 2006.

[26] K. Kakehi, K. Matsuzaki, and K. Emoto. Efficient parallel tree reductions on distributed memory environments. In *Proceedings of the 7th International Conference on Computational Science, Part II*, ICCS '07, pages 601–608, 2007. ISBN 978-3-540-72585-5.

[27] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, 2009.

[28] A. Koster. Frequency assignment - models and algorithms, 1999.

[29] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. SPAA '11, pages 85–94, 2011.

[30] S. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems (with discussion). *Journal of the Royal Statistical Society series B*, 50: 157–224, 1988.

[31] Y. Liu, Z. Hu, and K. Matsuzaki. Towards systematic parallel programming over mapreduce. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 39–50, 2011.

[32] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012. ISSN 2150-8097.

[33] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2.

[34] D. Marx. Graph coloring problems and their applications in scheduling. In *Proceedings of John von Neumann PhD Students Conference*. Citeseer, 2004.

[35] K. Matsuzaki, Z. Hu, and M. Takeichi. Implementation of parallel tree skeletons on distributed systems. In *proceeedings of the third Asina workshop on Programming Languages and Systems*, pages 258–271, 2002.

[36] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Proceedings of Annual European Conference on Parallel Processing (Euro-Par 2003)*, pages 789–798, 2003.

[37] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 15(3): 321–336, 2005.

[38] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallel skeletons for manipulating general trees. *Parallel Comput.*, 32(7):590–603, Sept. 2006. ISSN 0167-8191.

[39] K. Matsuzaki, Z. Hu, and M. Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In *Proceedings of the eighteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 39–48, 2006.

[40] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–489, Portland, Oregon, October 1985. IEEE.

[41] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '09, pages 177–185, 2009.

[42] M. Ogawa, Z. Hu, and I. Sasano. Iterative-free program analysis. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 111–123, 2003.

[43] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann, 1993.

[44] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[45] D. B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3:42–68, 1995.

[46] B. D. Sullivan, D. P. Weerapurage, and C. S. Groer. Parallel algorithms for graph optimization using tree decompositions. Technical Report ORNL/TM-2012/194, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2012.

[47] J. A. Telle. Complexity of domination-type problems in graphs. *Nordic Journal of Computing*, 1(1):157–171, 1994.

[48] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[49] F. Wei. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 99–110, 2010.

[50] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.