AN IMPLEMENTATION OF THE PARALLEL k-CORE DECOMPOSITION ALGORITHM IN GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

PRANAV KONDURI

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Timothy Davis

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Pranav Konduri, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty, Dr. Timothy Davis, prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

CHAPTERS

ABSTRACT

An Implementation of the Parallel k-core Decomposition Algorithm in GraphBLAS

Pranav Konduri
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Timothy Davis
Department of Computer Science and Engineering
Texas A&M University

The k-core of an undirected graph is the largest subgraph in which every vertex has a degree of at least some number k. Computing the k-core, also known as the k-core decomposition algorithm, has significant applications in network analysis, visualization, bioinformatics, and community detection. There exists a sequential procedure, developed by Batagelj and Zaversnik in 2003, that accurately performs k-core decomposition. This implementation has been consistently referenced as the gold standard, due to its $O(n + m)$ runtime. However, due to its large working set and lack of parallelism, its performance suffers on modern big-data graph problems where sheer size tends to overwhelm runtime due to cache misses. A 2014 algorithm designed by Dasari, Desh and Zubair M implements a parallel version of k-core decomposition (ParK) with significant speedup on multithreaded architectures. This paper aims to describe the development and implementation of ParK using the SuiteSparse:GraphBLAS API in C, a robust framework that defines a set of matrix and vector operations based on an algebra of semirings to perform computations on graphs. We show that while the GraphBLAS algorithm underperforms versus the sequential implementation in a full decomposition, a modified version of the algorithm that only computes a partial decomposition given some value k is significantly faster.

ACKNOWLEDGMENTS

Contributors

I would like to whole-heartedly thank my faculty advisor, Dr. Davis, for affording me the opportunity to study in such a fascinating field of computer science. I would also like to thank my fellow undergraduate researchers, Conor Mai, Georgy Thayyil, Abeer Waheed, and Tanner Hoke, for their invaluable guidance and support throughout the course of this research.

I would also like to thank my thesis advisor, Tawfik Hussein, for his motivating words of encouragement and guidance throughout the process of the Undergraduate Research Scholars program, and for his indefatigable patience with my and my cohort's questions and concerns.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my family for their encouragement, patience, and love.

The data analyzed and used for *An Implementation of the Parallel k-core Decomposition Algorithm in GraphBLAS* was provided by the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection).

Some figures in this thesis were provided, with permission, from Dr. Davis.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Undergraduate research was supported by the Computer Science Department at Texas A&M University and additional research funding from Intel and Nvidia.

1. INTRODUCTION

The k-core of an undirected graph is the largest subgraph in which every vertex has a degree of at least some number k. Used as a tool to analyze the clustering and sparsity of a graph, the terminology “k-core” was first designated in 1983 by Seidman et al.[1]. The concept has been integral in many graph applications such as network analysis, visualization, bioinformatics, and community detection [2][3][4], and as a preprocessing step for many other preeminent graph algorithms, such as the computation of the maximum clique [5]. Figure 1.1 outlines the cores of a simple graph of size $n = 13$.

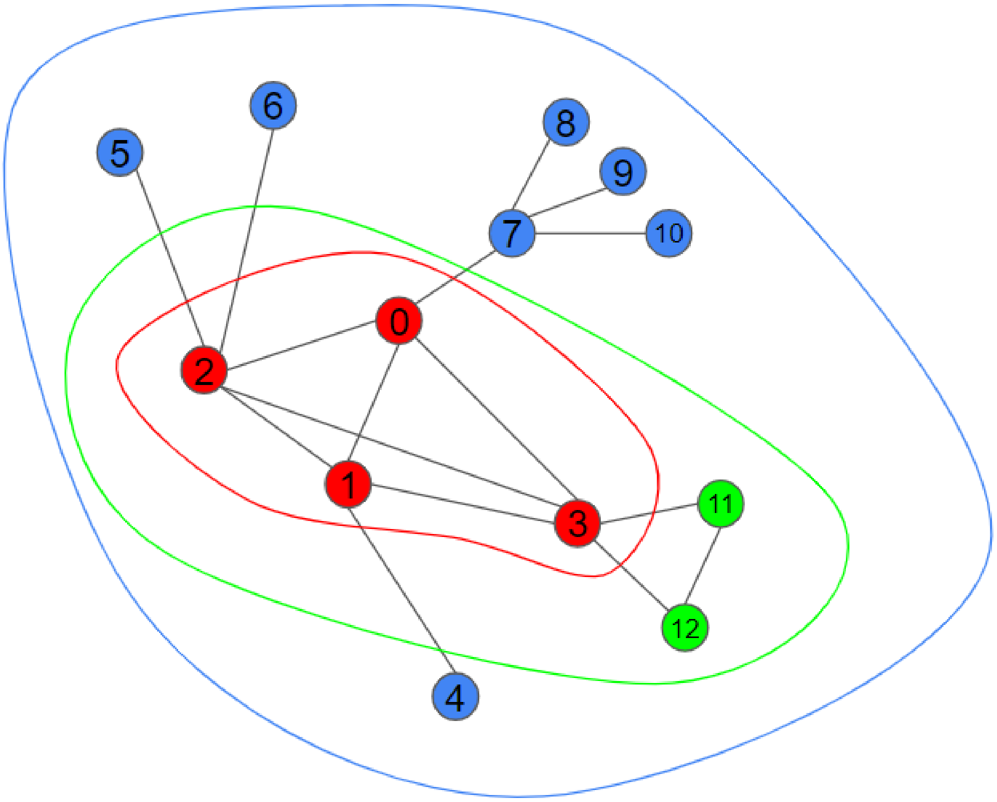


Figure 1.1: The 1 (blue), 2 (green), and 3-cores (red) of a simple graph

An important property of cores is that the $k+1$-core is always a subgraph of the k-core, as to go from k to $k+1$, nodes of k may only be removed, not added. Another is that the largest value of k in which the cardinality of the k-core is nonzero is called the k_{max}. For example, the k_{max} of the graph presented in Figure 1.1 is 3 (the red nodes highlighted and encircled). All of the nodes in the 3-core are valid vertices in the 1 and 2-cores as well, but a node in the 2-core is not guaranteed to be a valid vertex in the 3-core.

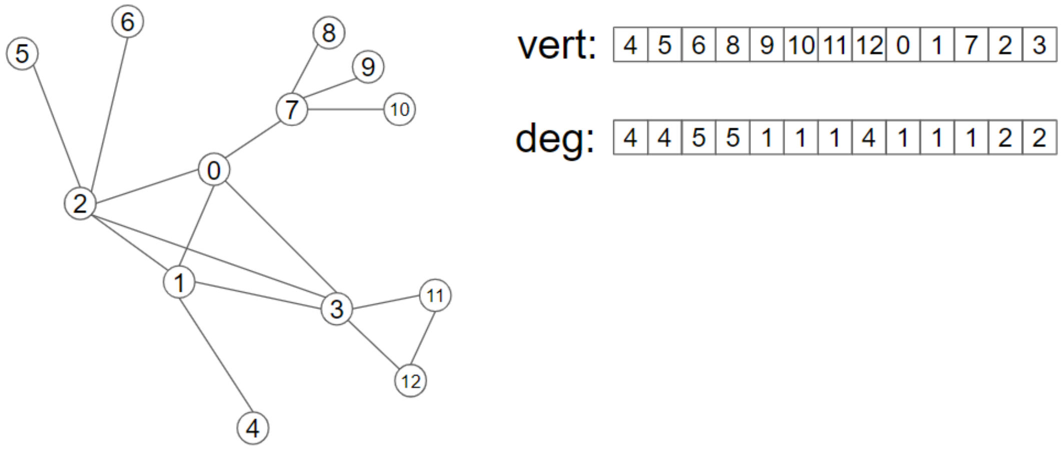
For the sake of this project, we will only be considering computing the k-core of symmetric, undirected graphs; all graphs used in the project that were considered directed will have been made undirected (a 1-way connection will be expanded to a 2-way connection). It is also important to highlight that varying edge weights have no bearing on output, as connectivity is the only relevant statistic.

1.1 The k-core Decomposition Algorithm

The 2003 algorithm composed by Batagelj and Zaversnik (hereby referred to as the BZ algorithm) is currently the state-of-the-art procedure to calculate k-cores of a graph [6]. The entire process has time complexity of $O(m+n)$, where n is the number of vertices and m is the number of edges in the graph. This is due to a bin-sort pre-processing phase, as the vertices of the graph must be ordered in increasing order of degree (Figure 1.2).

Algorithm 1 Batagelj-Zaversnik k-Core Decomposition Algorithm

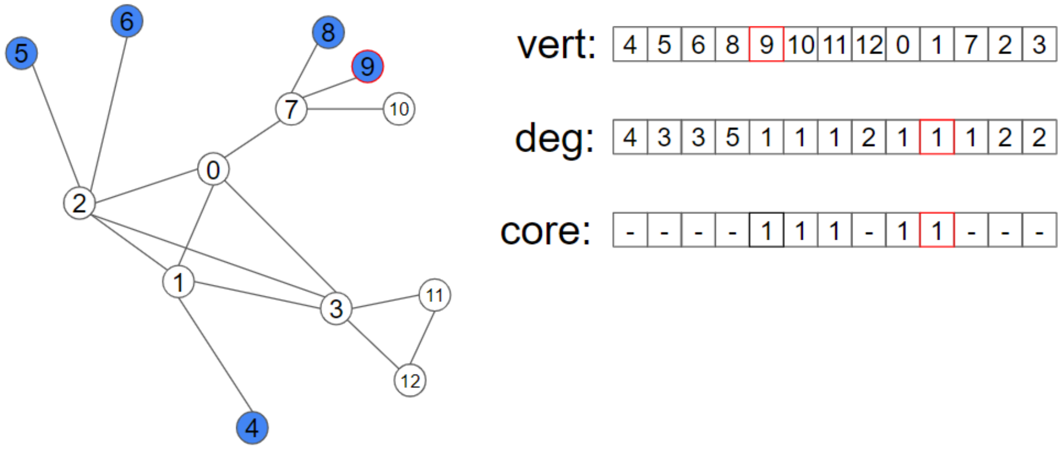
```
Order the set of vertices $vert$ in increasing degree order via bin-sort
$core \leftarrow \emptyset$ (output array)
for Each $v \in vert$ in order do
    $core[v] \leftarrow Degree(v)$
    for Each vertex $u \in Neighbors(v)$ in order do
        if $Degree(u) > Degree(v)$ then
            $Degree(u) = Degree(u) - 1$
            Reorder $vert$
        end if
    end for
end for
return $core$
```



vert:	4	5	6	8	9	10	11	12	0	1	7	2	3
deg:	4	4	5	5	1	1	1	4	1	1	1	2	2

Figure 1.2: BZ algorithm on Figure 1.1 after bin-sort

At each step of Algorithm 1, a vertex in our set is checked. If a vertex's neighbor has a degree over the value of the vertex, then we prune the edge connection between them, effectively decreasing the degree of the neighbor node by one (Figure 1.3).



vert:	4	5	6	8	9	10	11	12	0	1	7	2	3
deg:	4	3	3	5	1	1	1	2	1	1	1	2	2
core:	-	-	-	-	1	1	1	-	1	1	-	-	-

Figure 1.3: BZ algorithm on Figure 1.1 at 5th pass of for loop

Although the algorithm seems rudimentary, the process of obtaining a sorted array of vertices $vert$ is rather complicated, and must include some critical pre-processing steps. Two work arrays, both of size n, must be held: one ($vert$) to contain the vertex indices in sorted order of degree, and another (pos) to hold the position of each vertex in the $vert$ array (that way values do not get mixed up when shuffled). A third work array (bin) of size $max(deg(v))$ must also be instantiated to hold the starting and ending values of each degree in the $vert$ array. Due to this bin-sort, the "reordering of V" (line 7 of the algorithm) then becomes trivial. By swapping the vertex u into the location of the beginning of its bin, and then increasing the bin's start position by 1 effectively moves the vertex "down" a bin without the complication of reordering a size n array. This exchanges a costly $O(n)$ operation for an $O(1)$ operation. This is the key for the low time complexity of the BZ algorithm.

1.2 The Parallel k-Core Decomposition Algorithm (ParK)

Although the BZ solution is computationally quick, it suffers from high space complexity due to the large size of the numerous work vectors. With more and more complex graph problems (with millions of vertices and billions of edges), memory latency becomes a significant problem and tends to drag performance down. In 2014, Dasari, Desh, and Zubair of Old Dominion University in Norfolk, Virginia developed a Parallel k-core Decomposition implementation (hereby listed as ParK) that instead only uses two smaller shared work vectors [7]. The time complexity of the full algorithm converges at $O(k_{max} * n + m)$, and while it is slightly slower when implemented serially, the method is parallelizable to the point where significant speedup can be recorded, especially on the larger graphs where the BZ algorithm tends to struggle.

Algorithm 2 ParK k-core Decomposition Algorithm

```
 1: $deg \leftarrow$ vector representing degree of each node in graph
 2: $curr \leftarrow \emptyset$
 3: $next \leftarrow \emptyset$
 4: $core \leftarrow \emptyset$ (output array)
 5: $todo \leftarrow n$
 6: $level \leftarrow 1$
 7: while $todo > 0$ do
 8:     Add all nodes of $deg[v] = level$ to $curr$
 9:     while $|curr| > 0$ do
10:         $todo \leftarrow todo - |curr|$
11:         for Each vertex $v \in curr$ do
12:             $core[v] \leftarrow deg[v]$
13:             for Each vertex $u \in Neighbors(v)$ in order do
14:                 if $deg[u] > deg[v]$ then
15:                     $deg[u] = deg[u] - 1$
16:                     if $deg[u] = level$ then
17:                         Add $u$ to $next$
18:                     end if
19:                 end if
20:             end for
21:         end for
22:         $curr \leftarrow next$
23:         $next \leftarrow \emptyset$
24:     end while
25:     $level \leftarrow level + 1$
26: end while
27: return $core, level$
```

The ParK algorithm (Algorithm 2) relies on two sections (in the original algorithm paper, abstracted out as subroutines) to optimally compute the k-core of a graph. The *scan* subroutine (shown at line 8), isolates all nodes of a graph whose degree is equal to the given level (at the beginning of the algorithm, that level is 1). This process is slightly different from the BZ algorithm, as the values need not be sorted: an $O(n)$ pass through the degree array is required for each call of the *scan* process. However, this call is easily parallelizable provided the addition of nodes into the $curr$ array is done atomically. The *processSubLevel* (lines 11-21) subroutine is similar to lines 4-10 of the BZ algorithm, but if a node's degree value has been decreased to the given level, it must immediately be added into atomically into the $curr$ array, to be decreased as well. The $next$ array is a temporary holder for that next iteration. Once the *processSubLevel* does not add any new values to remove from the graph ($curr = \emptyset$), the level increments and the process starts over again, until all nodes have been processed ($todo = 0$).

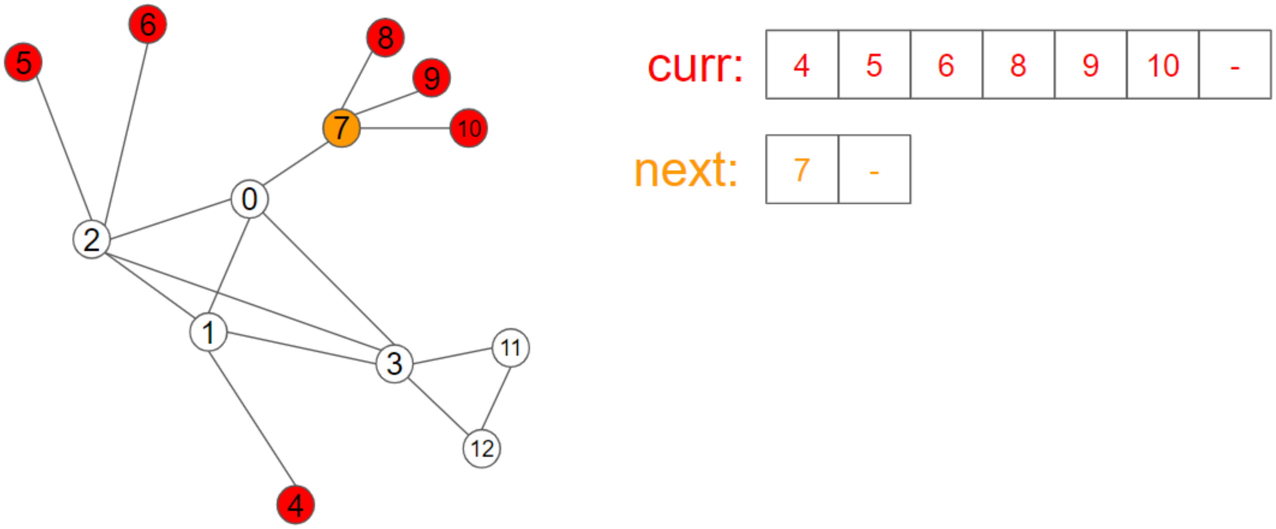


Figure 1.4: The first outer for-loop pass of ParK on Figure 1.1

Note in Figure 1.4, that after the last value in $curr$ has been processed, the degree value at node 7 is decreased to 1 (the same value as $level$), and thus it is added into the $next$ array to be deleted on the next pass.

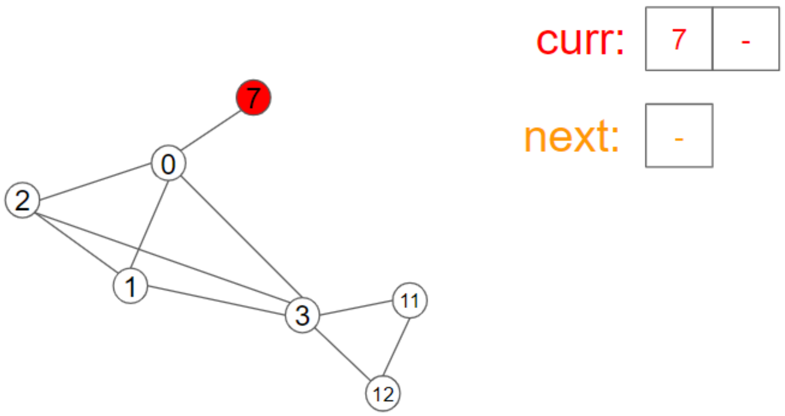


Figure 1.5: The second outer for-loop pass of ParK on Figure 1.1

Figure 1.5 outlines the next pass of the algorithm. Now that the $next$ array is empty, the 2-core is the resulting subgraph of the original input graph once node 7 is removed. The ParK algorithm will continue to iterate until every node is removed from the graph (similarly to BZ), at which point $level$ will be equivalent to k_{max}. On a 32 core machine, the ParK algorithm achieved as much as 6x speedup on the largest matrices as compared to the BZ algorithm.

Although we will not be developing the ParK algorithm as described, we will be basing our GraphBLAS implementation significantly on ParK, and benchmarking the based on an implementation of the sequential BZ algorithm. An implementation of the BZ algorithm (assisted with utility GraphBLAS methods) is available at Appendix A [8].

1.3 The SuiteSparse:GraphBLAS API

SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS standard defining a set of sparse matrix operations on an algebra of semirings. SuiteSparse:GraphBLAS utilizes existing GraphBLAS operations and optimizes them for sparse graphs [9]. Adjacency matrices are represented either as compressed-sparse columns (CSC), compressed-sparse rows (CSR), or hypersparse variations of the two (hyper CSR, hyper CSC). Vectors, which are also defined, are always stored in CSC.

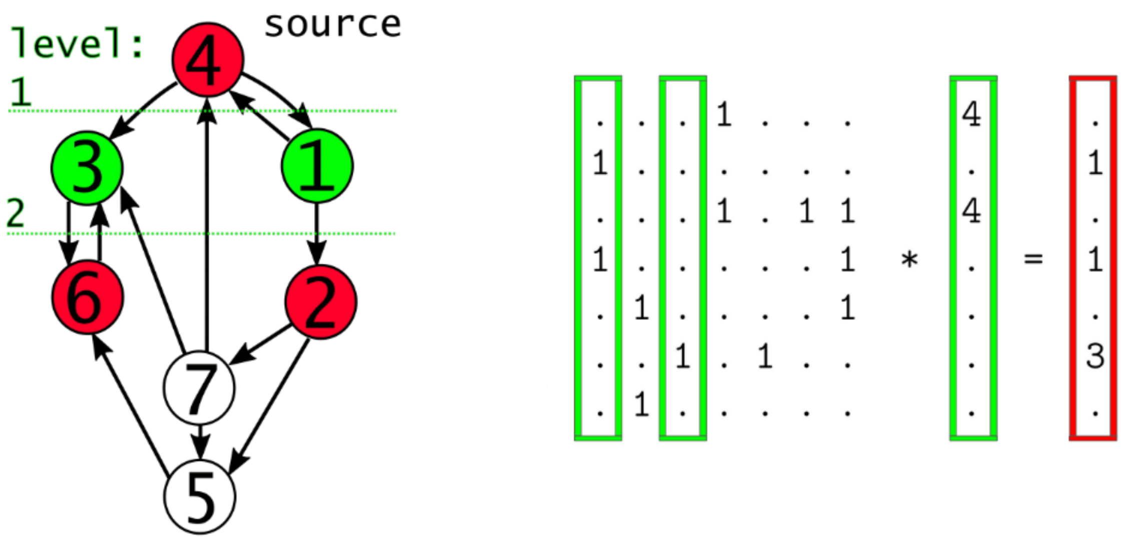
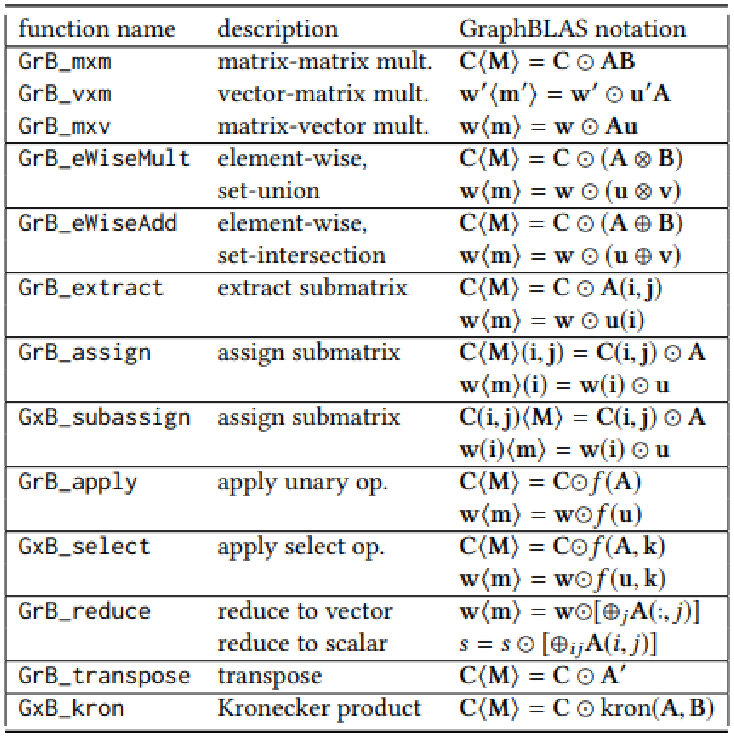


Figure 1.6: Illustration of the Matrix-Vector Multiply in GraphBLAS. Used with permission of Dr. Timothy Davis [10].

As illustrated by Figure 1.6, computations on adjacency matrices are equivalent to computations on graphs. In this image, the transposed adjacency matrix is multiplied with the vector containing only the nodes at positions 1 and 3 (from the second level of a BFS). The values within those nodes refer to the parent node of the values in the current frontier (4 for both cases, as 4 was the original source node). The populated indices in the resulting vector correspond to the neighboring nodes in the graph: 2, 4, and 6. The values at these indices correspond to the parent of that node (1 and 3, the original vector's values). Thus, this matrix-vector multiply is equivalent to multiple passes of a BFS [11]. This concept of highlighting neighbors through a multiply operation is critical to the function of our implementation of the k-core decomposition algorithm.

When the size of the input dominates the overall runtime cost of these algorithms, sparse matrices reduce the time cost. SuiteSparse:GraphBLAS does the hard work in deciding what specific computations would be most efficient for an individual call, so the developer can focus solely on the broader-scale implementation. Matrix-vector multiplies, as well as many other GraphBLAS methods, are executed in parallel and thus achieve significant speedup over sequential versions of the same computations.



function name	description	GraphBLAS notation
`GrB_mxm`	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix mult.	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
`GrB_mxv`	matrix-vector mult.	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise, set-union	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise, set-intersection	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{i},\mathbf{j})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i},\mathbf{j}) = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GxB_subassign`	assign submatrix	$\mathbf{C}(\mathbf{i},\mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C}\odot f(\mathbf{A})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w}\odot f(\mathbf{u})$
`GxB_select`	apply select op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C}\odot f(\mathbf{A},\mathbf{k})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w}\odot f(\mathbf{u},\mathbf{k})$
`GrB_reduce`	reduce to vector reduce to scalar	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w}\odot[\oplus_j \mathbf{A}(:,j)]$ $s = s \odot [\oplus_{ij} \mathbf{A}(i,j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$
`GxB_kron`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A},\mathbf{B})$

Figure 1.7: A selection of SuiteSparse:GraphBLAS methods. Used with permission of Dr. Timothy Davis [12].

1.4 LAGraph

LAGraph is the public library and test harness for translating GraphBLAS code into algorithms, an ongoing project dedicated to utilizing the power of the GraphBLAS libraries [13]. Despite not yet being ready for release as of March 2022, the LAGraph GitHub repository currently has code already fully implemented for strongly-tested implementations of Breadth-First Search, Single Source Shortest Path, Triangle Count, Vertex Centrality, and many more popular graph algorithms [14]. By using clever combinations of the methods shown above in Figure 1.7 (referenced from the SuiteSparse:GraphBLAS user guide) and more, even programmers with minimal knowledge of advanced graph theory can develop complex algorithms with ease. The team currently working on the LAGraph repository include developers from Anaconda, Redis Labs, IBM, Carnegie Mellon University, and undergraduate and graduate students at Texas A&M [10].

The goal of this project is to implement the Parallel k-Core Decomposition algorithm using the GraphBLAS framework and to submit the implementation into the LAGraph repository, benchmarking versus an implementation of the BZ algorithm.

2. METHODS

The algorithm was developed entirely in the C language, as is the case with all of the algorithms developed in the LAGraph test harness. Development, testing, and benchmarking were all conducted on the Texas A&M Computer Science department's *BACKSLASH* system.

2.1 SuiteSparse:GraphBLAS

The SuiteSparse:GraphBLAS standard depicts graphs in data as sparse matrices and defines operations on those matrices as equivalent to computations on graphs. The sparse matrices themselves are stored in memory as sets of sparse vectors. Operations also can be applied for individual vectors on one another as well to help facilitate other operations on graphs.

2.1.1 Methods Used

The key GraphBLAS methods we will be using in this project are:

- ***GrB_vxm:*** Similar to the aforementioned *GrB_mxv*, *GrB_vxm* multiplies a vector with a matrix on a semiring, to produce a vector. The semiring applies onto the intermediate operations of the matrix-vector multiply, thus allowing for specific customization for each algorithm.

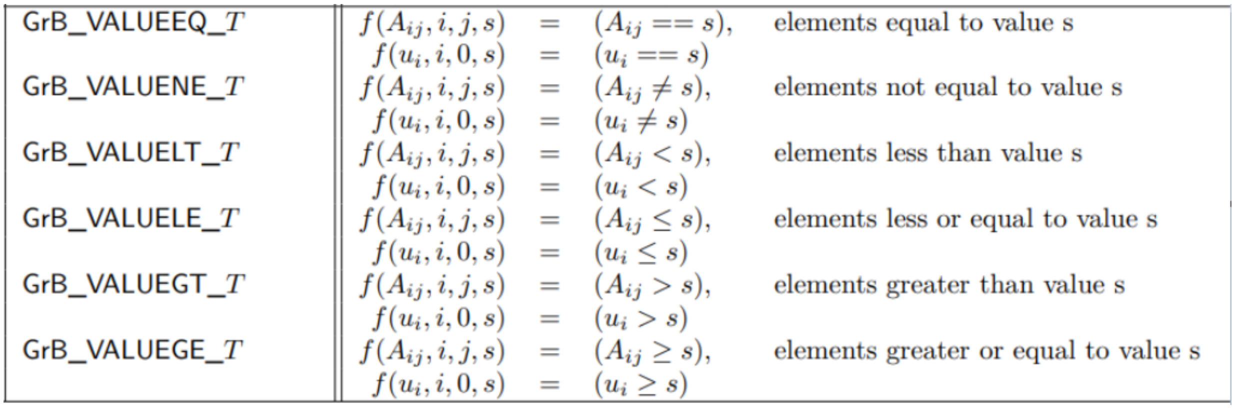
- ***GrB_mxm:*** Multiplies a with another matrix on a semiring, to produce a matrix. Semiring usage and application is similar to the *GrB_vxm* function.

- ***GrB_eWiseAdd (Element-wise addition):*** eWiseAdd performs an element-wise addition on the values of two vectors (vectors only). The calculations persist on the union of the elements of the two vectors (as opposed to the similar eWiseMult, which performs calculations only on the intersection of the elements). Just like the *GrB_vxm* and *GrB_mxm* functions, you must use or define a custom semiring to use to apply for the output vector.

- ***GrB_select:*** Applies an index unary operator onto the elements of a matrix or vector, keeping those that match the selection operator. Used to parse vectors and matrices and only extract entries whose values are above, equal, or below a certain value.

- ***GrB_assign:*** Assigns a subset of values from one source vector into some other destination vector. This does not replace the original destination vector, but instead overwrites the values in the destination vector, thereby preserving previous stored values.
- ***GrB_Matrix_diag:*** Creates a diagonal matrix of size $n * n$ given a vector of size n. A value at index 1 of the vector is then recorded at position $(1, 1)$ in the resulting output matrix.
- ***GrB_Vector_dup:*** Creates a new vector with the same size and elements as an existing vector. This is effectively a deep copy of an existing vector.
- ***GrB_Vector_nrows:*** Returns the number of rows within a matrix. The resulting value is stored as an int64_t data type. This is the same as computing the number of vertices within a graph, or the size of a two-dimensional array.
- ***GrB_Vector_nvals:*** Returns the number of values present within a sparse vector. The resulting value is stored as an int64_t data type. Note that this value does not return the full size of the vector, but the specific number of values that are currently stored within the data. This distinction is important, as this function is used to keep track of the amount of nodes to process.
- ***LAGraph_Property_RowDegree:*** An LAGraph utility function to calculate the row degree of a graph. The degree of each node in the sparse matrix is then returned to a vector of size n.

2.1.2 Index Unary Operators



GrB_VALUEEQ_T	$f(A_{ij}, i, j, s) = (A_{ij} == s)$,	elements equal to value s
	$f(u_i, i, 0, s) = (u_i == s)$	
GrB_VALUENE_T	$f(A_{ij}, i, j, s) = (A_{ij} \neq s)$,	elements not equal to value s
	$f(u_i, i, 0, s) = (u_i \neq s)$	
GrB_VALUELT_T	$f(A_{ij}, i, j, s) = (A_{ij} < s)$,	elements less than value s
	$f(u_i, i, 0, s) = (u_i < s)$	
GrB_VALUELE_T	$f(A_{ij}, i, j, s) = (A_{ij} \leq s)$,	elements less or equal to value s
	$f(u_i, i, 0, s) = (u_i \leq s)$	
GrB_VALUEGT_T	$f(A_{ij}, i, j, s) = (A_{ij} > s)$,	elements greater than value s
	$f(u_i, i, 0, s) = (u_i > s)$	
GrB_VALUEGE_T	$f(A_{ij}, i, j, s) = (A_{ij} \geq s)$,	elements greater or equal to value s
	$f(u_i, i, 0, s) = (u_i \geq s)$	

Figure 2.1: A subset of significant SuiteSparse:GraphBLAS Operators, some of which we use. Used with permission of Dr. Timothy Davis [12].

For each call of the *GrB_select* function, we exercise the option of using index unary operators, flags that check for the presence of some condition in the values of the vectors in the computation. Upon the success of this condition at an index of a vector, the function is then applied to the value at that index. These operators are useful to parse vectors efficiently while only maintaining values that are consistent with certain parameters (greater than a certain constant number, for example). The values in Figure 2.1 above show a snippet of some of the possible operators, some of which are used in the implementation of this algorithm. For the purpose of the k-core algorithm, operators will primarily be used to check the degree of a node; if the degree of a node within the vector is greater than or less than the current k-core level, some operation will then apply to it.

2.1.3 Masks and Descriptors

Another crucial part of the GraphBLAS standard is the ability to use masks, opaque objects that change the output of many functions. Masks can be either vectors or matrices, and must match the dimensions of the output data structure of the method. The purpose of a mask is just that: to mask some elements from appearing in the output structure. If the output of some GraphBLAS calculation intends to place a value at some index i, there must be a value present at i in the mask as well. Normal masks imply that the value must be specifically a boolean TRUE for the mask to apply properly, while structural masks only require the existence of any value in order to apply. Structural masks are important, as they can ignore the values within the vector and just the presence of values at certain locations. Computationally, structural masks are much faster to use.

Masks and input/output vectors and matrices within a computation are then further augmented by descriptors, lightweight flags in a method that change how the other input parameters should be modified before computations are performed. Most common is a complement, which effectively reverses the mask; instead, anything that is NOT caught by the mask is then pushed into the output array. Descriptors can also specify whether a mask should be evaluated as a structural mask or not, if one of the input vectors or matrices should be transposed, or even to call to replace all data in the output structure with the computation's result.

2.1.4 Semirings

Per the SuiteSparse:GraphBLAS user guide, a semiring "defines all the operators required to define the multiplication of two sparse matrices in GraphBLAS" [10]. Semirings can also be applied for matrix-vector multiplications as well, or vector-vector multiplications. In GraphBLAS, semirings contain two designations: a monoid specifying what occurs when we would normally add values together, and a binary operator specifying what occurs when we would normally multiply values with one another. This means that instead of always returning products as outputs, we could instead return a subtraction, or a division (or any number of other operations). Semirings allow for greater user-level abstraction in function calls, and are arguably the most important structures within GraphBLAS due to their flexibility.

2.2 The GraphBLAS KCore Algorithm

Algorithm 3 GraphBLAS k-Core Decomposition Algorithm

```
$A \leftarrow$ the graph to input (user parameter)
$deg \leftarrow$ vector of degree of $A$ using *LAGraph_Property_RowDegree*
$curr \leftarrow \emptyset$
$done \leftarrow \emptyset$
$delta \leftarrow \emptyset$
$core \leftarrow \emptyset$ (output array)
$todo \leftarrow n$
$level \leftarrow 1$
**while** $todo > 0$ **do**
    Add all nodes of $Degree[v] = level$ to $curr$ using *GrB_select*
    Set all nodes remaining in $deg$ to $level$ in $core$ using *GrB_assign*
    **while** $|curr| > 0$ **do**
        $todo \leftarrow todo - |curr|$
        Copy all values in $curr$ into $done$ using *GrB_assign*
        Add all nodes that neighbor values of $curr$ into $delta$ using *GrB_vxm* ($curr * A$)
        Decrement values of $delta$ matching values in $deg$ using *GrB_eWiseAdd*
        $curr \leftarrow$ all nodes of $deg <= level$ using *GrB_select*
    **end while**
    $level \leftarrow level + 1$
**end while**
**return** $core, level$
```

As shown in Algorithm 3, the GraphBLAS k-core decomposition algorithm (hereby referred to as GrB_KC) uses a combination of the methods described to compute the k-core. Given a graph, the algorithm computes the core number of each vertex (the maximum k-core that the node is a part of). While the deg vector (computed by *LAGraph_Property_RowDegree*) has elements in it, it is augmented at each level in order to compute the next k-core. The last set of nodes of the degree vector, before being cleared, is the maximum k-core subgraph.

Within the outer while loop, we must do a procedure to optimally calculate the k-core a specified given level k. The procedure is as follows:

1. Highlight all nodes in the degree vector that are equivalent to the k-core (all nodes where

$Degree(v) = k$). These values will be added to the vector $curr$. All nodes in $curr$ are slotted to be deleted from the k-core.

2. Decrement the total amount of nodes that need to be processed (variable $todo$) by the number of values currently in $curr$.

3. Using *GrB_assign*, add all values of $curr$ into $done$, a vector used to keep track of all nodes that have been removed from the deg vector.

4. Using *GrB_vxm* with the *LAGraph_plus_one_int64* semiring, compile a new vector $delta$ that keeps track of all nodes in the graph that have lost neighbors, as well as how many neighbors they lost. The semiring replaces the multiplication operation with a single value (one) while the addition operation adds all of those ones together. The function multiplies the $curr$ vector with the input matrix, while the semiring effectively increments by one each time a neighbor of a node in the vector is found.

5. Using *GrB_eWiseAdd*, we recalculate the deg vector, decrementing $delta$ from deg. This is done by masking the operations of the *eWiseAdd* function with the $done$ vector, using a complemented mask (only the vertices **NOT** in $done$ will be returned back into deg). This will only return the nodes in the graph that have lost neighbors that we have not already processed.

6. Using *GrB_select*, highlight the new values that need to be removed from the graph (if any). These are any values with a value (a degree) less than or equal to the current value of $level$. These values go into $curr$, which overwrite its previous ones as they have now been fully processed.

7. Loop steps 2-6 until the $curr$ vector is empty (all nodes remaining in the deg vector are part of this level's core. If the deg vector is empty, then the max core is equivalent current value of $level$ and the algorithm finishes).

Step 1 of this procedure produces the same result as the *Scan* procedure in ParK, while steps 4-6 produce a similar result as the *processSubLevel* procedure. What is fascinating is that since the underlying GraphBLAS calls are parallel, the core of the algorithm is realistically only six critical lines of code. A more complete version of the code is available at Appendix B.

Although it may seem as if there are a significant number of vectors being maintained at each pass, it is important to keep in mind that the $curr$ and $delta$ work vectors are sparse, and the number of values in each is often a small percentage of n (rarely more than 5%) for all tested cases. This makes them incredibly efficient as work vectors compared to the BZ algorithm as we minimize cache misses.

2.3 A Speedier, Less Exhaustive Algorithm

What stands out about the GraphBLAS interpretation of the ParK algorithm compared to the other two implementations is that it is not strictly necessary to increment $level$ by one each time we empty out $curr$. BZ and ParK require this stipulation to be met as the degree of neighboring vectors are only decremented by one at a time and then immediately checked (BZ lines 5-10 and ParK lines 13-20). By using the *LAGraph_plus_one_int64* semiring in a vxm call, we can decrement nodes as a batch, allow the degree of neighbor nodes fall well below the value of $level$ (even to 0), and then pick them all up as a batch with a call to *GrB_select*. Thus, it is possible in our implementation to increment $level$ by a much greater number (perhaps a heuristic variable depending on the maximum degree and/or density of the input graph), or start $level$ at some much higher value.

The reason why we do not do this for GrB_KC, is because we want to stay consistent with the implementation of the other algorithms that provide the core number for each and every vertex. Incrementing by a number greater than 1 each loop would give uncertainty to the core numbers for vertices that are eliminated and not in the final k-core. For example, a node could be eliminated at $k = 84$ but realistically be only part of $k = 82$ if the step size is $>= 3$.

However, it is worth mentioning that the primary use case for the k-core decomposition algorithm is to return the k-core for one specific value k- a computation where the corneness values for all nodes are not needed. While the BZ and ParK algorithms are forced to increment from level 1 to k the GraphBLAS implementation can get to k immediately in one set of calls. Thus, the augmented Single k-core Decomposition Algorithm is born:

Algorithm 4 GraphBLAS Single k-Core Decomposition Algorithm

```
$A \leftarrow$ the graph to input (user parameter)
$k \leftarrow$ a pre-specified level (user parameter)
$deg \leftarrow$ vector of degree of $A$ using LAGraph_Property_RowDegree
$curr \leftarrow \emptyset$
$done \leftarrow \emptyset$
$delta \leftarrow \emptyset$
$core \leftarrow \emptyset$ (output array)
$todo \leftarrow n$
Add all nodes of $Degree[v] < k$ to $curr$ using GrB_select
while $|curr| > 0$ and $|deg| > 0$ do
    $todo \leftarrow todo - |curr|$
    Copy all values in $curr$ into $done$ using GrB_assign
    Add all nodes that neighbor values of $curr$ into $delta$ using GrB_vxm ($curr * A$)
    Decrement values of $delta$ matching values in $deg$ using GrB_eWiseAdd
    $curr \leftarrow$ all nodes of $deg < k$ using GrB_select
end while
$core \leftarrow$ all remaining values in deg using GrB_assign
return $core$
```

By utilizing the fact that GrB_KC can quickly obtain the k-core level starting at an arbitrary number trivially, we can use an augmented procedure (Algorithm 4, hereby referred to as GrB_SKC) exploiting that; obtaining only the values for a specific core at a significantly lower computational cost. We no longer care about exhausting every value in deg, and can effectively terminate the program after one iteration of an inner loop of GrB_KC. A more complete version of the code is available at Appendix C. It would be best to use this method when a developer only needs a specific core (i.e. the 4-core), but does not require the entire graph's coreness values.

2.4 Decomposing the Graph

A notable issue with all of the algorithms thus far is that they only provide a vector output labeling the nodes with values, and not an individual graph themselves. This issue can be solved trivially with another procedure in GraphBLAS.

Algorithm 5 GraphBLAS Decompose Utility Algorithm

```
$A \leftarrow$ the graph to input (user parameter)
$k \leftarrow$ a pre-specified level (user parameter)
$core \leftarrow$ vector of coreness values in a graph (user parameter)
$deg \leftarrow$ all nodes of $core >= k$ using GrB_select
$C \leftarrow$ a diagonal matrix constructed from $deg$ using GrB_Matrix_diag
$D \leftarrow C * A * C$ using GrB_mxm (ANY_SECOND_I and MIN_SECOND_I semirings)
return $D$
```

Algorithm 5 is a simple utility function that takes in a graph A, a vector $core$ (the output of either BZ, ParK, GrB_KC, GrB_SKC), and a specified core-level k and outputs a subgraph indicating the k-core. This is done by using two matrix-matrix multiplications in succession on a diagonal matrix containing only the nodes present in the k-core. The choice of semirings guarantee that the vertices and edge values originally stored in A are the same values that are stored for the nodes that eventually are present in the subgraph D. This algorithm will not be benchmarked as it is simply a relevant utility function. A more complete version of the code is available at Appendix D.

3. RESULTS

3.1 Timing

As mentioned, results were benchmarked on Texas A&M's *BACKSLASH* system, a 12-CPU Intel Xeon E5-2695 v2 system with 12 cores per socket, and 2 threads per core. A typical maximum speedup on this platform is around the factor of 12. The test suite consisted of 7 sparse graphs, chosen from the SuiteSparse Matrix Collection (formerly known as the University of Florida SuiteSparse Matrix Collection) [15].

Table 3.1 outlines the timing results of the BZ algorithm (sequential, one thread), the GrB_KC algorithm (parallel, 20 threads), and the GrB_SKC algorithm (parallel, 20 threads, $k = k_{max}$ and $k = 5$). We chose to test different values of k for the GrB_SKC algorithm to signify vastly different use cases. All timing results are displayed in seconds. Note that this project does not include an implementation of the original ParK algorithm.

Table 3.1: Details of Graphs Used and Timing Results Between BZ and GrB algorithms

graph	n ($\times 10^6$)	m ($\times 10^6$)	k_{max}	BZ	GrB_KC	GrB_SKC (k_{max})	GrB_SKC (5)
amazon0601	0.40	2.44	10	0.16	0.65	0.15	0.03
as-Skitter	1.70	11.10	111	0.89	2.98	0.08	0.26
cit-Patents	3.77	16.52	64	4.59	4.77	0.17	0.06
in-2004	1.38	16.92	488	2.20	18.41	0.09	<0.01
soc-Pokec	1.63	30.62	47	0.35	9.18	< 0.01	0.02
hollywood-2009	1.14	113.89	2208	5.13	184.812	0.17	0.04
indochina-2004	7.41	194.11	6869	3.21	6.94	0.39	0.04

Although in some cases the GrB_KC algorithm displays performance closer to the sequential BZ algorithm, there is no speedup witnessed. Although this may seem like a disappointing statistic, it makes sense; as discussed, the GraphBLAS standard is best at producing intermediate results en masse as opposed to sequentially like the BZ algorithm is optimized for. When using the GrB_SKC algorithm with $k = k_{max}$ (the highest possible value of k that can be passed in that would return a degree vector of any kind), the algorithm shows significant speedup (up to 50x, as seen with the soc-Pokec graph). With a more conventional choice of k at 5, even further speedup is witnessed in most test cases (barring the as-Skitter dataset), in which computation takes less than a tenth of a second to complete.

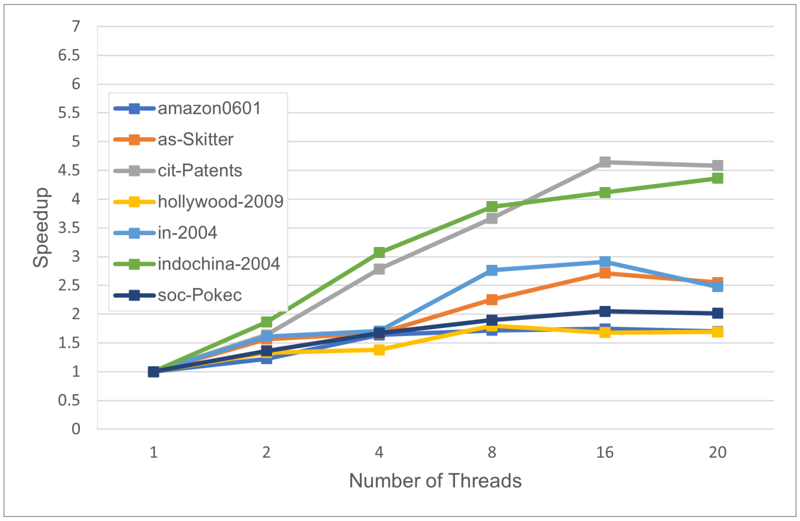


Figure 3.1: Speedup versus 1-thread version of GrB_KC for Multiple Threads

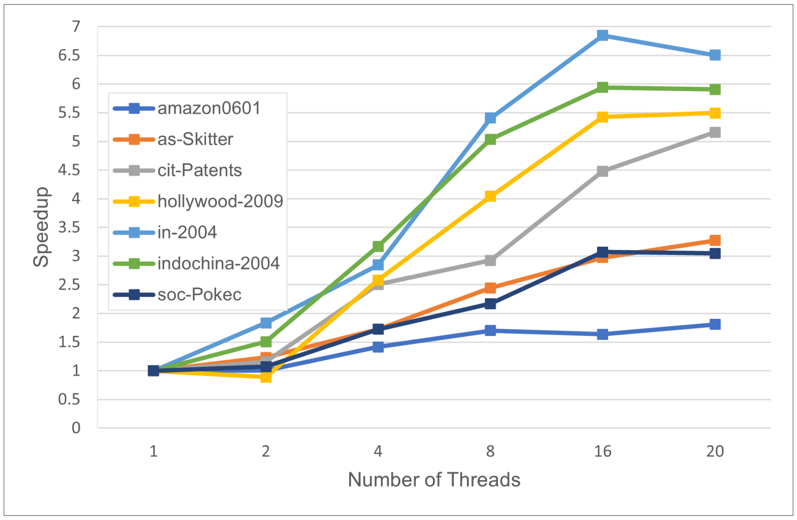


Figure 3.2: Speedup versus 1-thread version of GrB_SKC for Multiple Threads

As seen in Figures 3.1 and 3.2, multithreading does achieve noticeable performance increases when compared to 1-thread timing of the same program. By utilizing an API whose calculations already work in parallel, a developer's code can be written more simply.

4. CONCLUSION

The most important lesson learned through the development of this algorithm is to play to the API's strengths. SuiteSparse:GraphBLAS is best when it has the ability to do lots of computations within a single call, so augmenting the parameters of our question, ever so slightly, to identify that strength resulted in an effective algorithm in GrB_SKC. While the original goal of the project (to create an implementation of the parallel k-core algorithm through GrB_KC) was completed, it is fascinating that by isolating a specific use case, GraphBLAS vastly outperforms the BZ algorithm.

4.1 Future Work

There is still plenty of work to be done with the implementation of this algorithm. Although it has been stated that SuiteSparse:GraphBLAS decides the specific computations for you to optimize performance, speedup could perhaps be achieved further still by analyzing heuristics related to the graph itself and pre-choosing optimal settings. These settings can range from how certain data is stored in internal structures (vectors and matrices), to extra parameters allowing for greater user flexibility.

It is also worth mentioning that this implementation was largely based off of the ParK algorithm presented in 2014. There exists another, more complex parallel k-core algorithm developed by Kabir and Madduri [8] of the Pennsylvania State University in 2017 that found noticeable speedup even over the ParK algorithm (on certain input graphs) called PKC. Due to its comparative complexity to ParK, PKC was not chosen as the guideline for developing GrB_KC. Further developments in improving our GraphBLAS implementation may warrant a PKC-based solution.

REFERENCES

[1] S. B. Seidman, “Network structure and minimum degree,” *Social Networks*, vol. 5, no. 3, p. 269–287, 1983.

[2] J. I. Alvarez-Hamelin, L. Dall’Asta, A. Barrat, and A. Vespignani, “Large scale networks fingerprinting and visualization using the k-core decomposition,” in *Advances in neural information processing systems*, pp. 41–50, 2006.

[3] Y. Cheng, C. Lu, and N. Wang, “Local k-core clustering for gene networks,” in *2013 IEEE International Conference on Bioinformatics and Biomedicine*, pp. 9–15, 2013.

[4] E. Miho, R. Roškar, V. Greiff, and S. T. Reddy, “Large-scale network analysis reveals the sequence space architecture of antibody repertoires,” *Nature Communications*, vol. 10, no. 1, 2019.

[5] R. A. Rossi, D. F. Gleich, and A. H. Gebremedhin, “Parallel maximum clique algorithms with applications to network analysis,” *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. C589–C616, 2015.

[6] V. Batagelj and M. Zaversnik, “An O(m) algorithm for cores decomposition of networks,” *CoRR*, vol. cs.DS/0310049, 2003.

[7] N. S. Dasari, R. Desh, and M. Zubair, “ParK: An efficient algorithm for k-core decomposition on multicore processors,” *2014 IEEE International Conference on Big Data (Big Data)*, 2014.

[8] H. Kabir and K. Madduri, “Parallel k-core decomposition on multicore platforms,” *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.

[9] T. A. Davis, “Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss,” in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–6, 2018.

[10] T. A. Davis, “SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Linear Algebra,” Jan 2022.

[11] J. V. Kepner and J. R. Gilbert, *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[12] T. A. Davis, "User Guide for SuiteSparse:GraphBLAS," Nov 2021.

[13] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "LA-Graph: A community effort to collect graph algorithms built on top of the GraphBLAS," *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019.

[14] GraphBLAS, "LAGraph/src/algorithm at reorg · GraphBLAS/LAGraph." `https://github.com/GraphBLAS/LAGraph/tree/reorg/src/algorithm`.

[15] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, December 2011.

APPENDIX A: CORE C CODE OF AN IMPLEMENTATION OF THE BZ ALGORITHM

```c
/*
*   Given an input graph, returns a vector indicating the coreness
*   number of each node, or the largest k such that the node is a
*   part of the k-core.
*
*   IN/OUT: *core, a pointer to an uint_64_t array of size n
*   IN/OUT: kmax, the largest k such that the k-core is non-empty
*   IN: G, an input graph
*
*/
int GrB_BZ (uint64_t *core, uint64_t *kmax, LAGraph_Graph G)
{
    //export the graph into CSR format using LG_check_export
    GrB_Index *Ap = NULL, *Aj = NULL, *Ai = NULL ;
    void *Ax = NULL ;
    GrB_Index Ap_size, Aj_size, Ax_size, n, ncols, Ap_len, Aj_len, Ax_len ;
    size_t typesize ;
    LG_check_export (G, &Ap, &Aj, &Ax, &Ap_len, &Aj_len, &Ax_len, &typesize, msg) ;

    //create the arrays
    uint64_t *vert = NULL, *pos = NULL, *bin = NULL;
    uint64_t *deg = NULL;
    uint64_t maxDeg = 0;
    deg = malloc(n, sizeof(uint64_t)) ;
    vert = malloc(n, sizeof(uint64_t)) ;
    pos = malloc(n, sizeof(uint64_t)) ;

    for(uint64_t i = 0; i < n; i++){
        deg[i] = Ap[i+1] - Ap[i];
        if (deg[i] > maxDeg)
            maxDeg = deg[i];
    }
    //setup bin array
    bin = calloc(maxDeg + 1, sizeof(uint64_t)) ;

    for(uint64_t i = 0; i < n; i++){
        bin[deg[i]]++;
    }

    uint64_t start = 0;
    for(uint64_t d = 0; d < maxDeg + 1; d++){
        uint64_t num = bin[d];
        bin[d] = start;
        start = start + num;
    }

    //Do bin-sort
    //vert -- contains the vertices in sorted order of degree
    //pos -- contains the positon of a vertex in vert array
    for(uint64_t i = 0; i < n; i++){
        pos[i] = bin[ deg[i] ];
        vert[pos[i]] = i;
        bin[deg[i]] ++;
    }

    for(uint64_t d = maxDeg; d >= 1; d --)
        bin[d] = bin[d-1];
    bin[0] = 0;

    int64_t level = 0;

    //Compute k-core
    for(int64_t i = 0; i < n; i++){
        //get the vertex to check
        uint64_t v = vert[i];
        //populate output matrix
        core[v] = deg[v];
        if(bin[deg[v]] == i){
            level = deg[v];
        }

        uint64_t start = Ap[v];
        int64_t original_deg = Ap[v+1] - Ap[v]; //original deg before decremented
        for(uint64_t j = 0; j < original_deg; j++){
            uint64_t u = Aj[start + j]; //a neighbor node of v

            //if we need to lower the neighbor's deg value, and relocate in bin
            if(deg[u] > deg[v]){
                uint64_t du = deg[u];
                uint64_t pu = pos[u];
                uint64_t pw = bin[du];
                uint64_t w = vert[pw]; //the vertex situated at the beginning of the bin

                //swap around the vertices- w goes to the end, u goes to the beginning
                if(u != w){
                    pos[u] = pw; vert[pu] = w;
                    pos[w] = pu; vert[pw] = u;
                }

                //increase starting index of bin @ du
                bin[du]++;
                //decrease degree of u
                deg[u]--;
            }
        }
    }
    return (GrB_SUCCESS);
}
```

APPENDIX B: CORE C CODE OF THE GrB_KC ALGORITHM

```
/*
*   Given an input graph, returns a vector indicating the coreness
*   number of each node, or the largest k such that the node is a
*   part of the k-core.
*
*   IN/OUT: *core, a pointer to a GrB_Vector
*   IN/OUT: kmax, the largest k such that the k-core is non-empty
*   IN: G, an input graph
*
*/
int GrB_KCore (GrB_Vector *core, uint64_t *kmax, LAGraph_Graph G)
{
    uint64_t level = 0;
    GrB_Vector deg = NULL, curr = NULL, done = NULL, delta = NULL;
    GrB_Index nvals, todo;
    LAGraph_Property_RowDegree(G, msg) ;
    GrB_Vector_dup(&deg, G->rowdegree) ; //original deg vector is
        technically 1-core since 0 is omitted
    GrB_Vector_nvals(&todo, deg) ;

    while(todo > 0){
        level++;
        // Creating curr;
        GrB_select (curr, GrB_NULL, GrB_NULL, GrB_VALUEEcurr_INT64, deg,
            level, GrB_NULL) ; // get all nodes with degree = level
        GrB_Vector_nvals(&nvals, curr);

        //assign remaining values of deg to output
        GrB_assign (*core, deg, NULL, level, GrB_ALL, n, GrB_NULL) ;
        // while curr not empty
        while(nvals > 0){
            // Decrease todo by number of nvals
            todo = todo - nvals ;
            //add anything in curr as true into the done list (using
                structural mask)
            GrB_assign (done, curr, NULL, (bool) true, GrB_ALL, n,
                GrB_DESC_S);

            // Create delta (the nodes who lost friends, and how many they
                lost)
            GrB_vxm (delta, GrB_NULL, GrB_NULL, LAGraph_plus_one_int64,
                curr, A, GrB_NULL);

            // Create new deg vector- keep anything not in done vector (
                using complement and structural mask)
            GrB_eWiseAdd(deg, done, GrB_NULL, GrB_MINUS_INT64, deg, delta,
                GrB_DESC_RSC) ;

            // Update curr, set new nvals
            GrB_select (curr, GrB_NULL, GrB_NULL, GrB_VALUELE_INT64, deg, level, GrB_NULL) ;
            GrB_Vector_nvals(&nvals, curr) ;
        }
    }
    (*kmax) = level;
    return (GrB_SUCCESS);
}
```

APPENDIX C: CORE C CODE OF THE GrB_SKC ALGORITHM

```c
/*
*   Given an input graph and an integer k, returns a vector indicating
*   the nodes belonging to the k-core of the graph.
*
*   IN/OUT: *core, a pointer to a GrB_Vector
*   IN: G, an input graph
*   IN: k, a level at which to apply the k-core algorithm
*/
int GrB_SKCore (GrB_Vector *core, LAGraph_Graph G, int k)
{
    GrB_Vector deg = NULL, curr = NULL, done = NULL, delta = NULL;
    GrB_Index currnvals, degnvals;
    LAGraph_Property_RowDegree(G, msg) ;
    GrB_Vector_dup(&deg, G->rowdegree) ;
    GrB_Vector_nvals(&degnvals, deg) ;

    GrB_select (curr, GrB_NULL, GrB_NULL, GrB_VALUELT_INT64, deg, k,
       GrB_NULL)) ; // get all nodes with degree = level
    GrB_Vector_nvals(&currnvals, curr);

    while(currnvals > 0 && degnvals > 0){
        //add anything in curr as true into the done list (using structural
            mask)
        GrB_assign (done, curr, NULL, (bool) true, GrB_ALL, n, GrB_DESC_S))
            ; //structure to take care of 0-node cases

        // Create delta (the nodes who lost friends, and how many they lost
           ) (push version)
        GrB_vxm (delta, GrB_NULL, GrB_NULL, LAGraph_plus_one_int64, curr, A
           , GrB_NULL);

        // Create new deg vector- keep anything not in done vector (using
           complement and structural mask)
        GrB_eWiseAdd(deg, done, GrB_NULL, GrB_MINUS_INT64, deg, delta,
           GrB_DESC_RSC) ;

        // Update curr, set new nvals
        GrB_select (curr, GrB_NULL, GrB_NULL, GrB_VALUELT_INT64, deg, k,
           GrB_NULL) ;
        GrB_Vector_nvals(&currnvals, curr) ;
        GrB_Vector_nvals(&degnvals, deg) ;
    }
    //Assign values of deg to decomp (all nodes remaining in deg are part
       of the k-core)
    GrB_assign (*core, deg, NULL, k, GrB_ALL, n, GrB_NULL) ;
    return (GrB_SUCCESS);
}
```

APPENDIX D: CORE C CODE OF THE GrB_Decompose ALGORITHM

```c
/*
*   Decomposes a matrix into a subgraph of nodes only
*   included in the k-core
*
*   IN/OUT: *D, a pointer to a GrB_Matrix containing the decomposition
*   IN: G, an input graph
*   IN: *core, a pointer to a GrB_Vector outlining the
*       core of each node in the graph
*   IN: k, a level at which to decompose the graph by.
*       All nodes with a value less than
*       k in decomp will be omitted from the output graph
*/
int GrB_Decompose (GrB_Matrix *D, LAGraph_Graph G, GrB_Vector *core,
   uint64_t k)
{
    GrB_Index nrows, n;
    GrB_Matrix A = NULL, C = NULL;
    GrB_Vector deg = NULL;

    A = G->A;
    GrB_Vector_size(&n, *core) ;

    //Create Vectors and Matrices
    GrB_Vector_new(&deg, GrB_INT64, n) ;
    GrB_Matrix_new(&C, GrB_INT64, n, n) ;
    GrB_Matrix_new(D, GrB_INT64, n, n) ;
    //create deg vector using select
    GrB_select (deg, GrB_NULL, GrB_NULL, GrB_VALUEGE_INT64, *core, k,
       GrB_NULL) ;

    //create decomposition matrix
    GrB_Matrix_diag(C, deg, 0) ;
    GrB_mxm (*D, NULL, NULL, GxB_ANY_SECONDI_INT64, C, A, GrB_NULL) ;
    GrB_mxm (*D, NULL, NULL, GxB_MIN_SECONDI_INT64, *D, C, GrB_NULL) ;
    return (GrB_SUCCESS);
}
```